Unit - III

Chapter - 5 Dynamic Programming (5 - 1) to (5 - 72)

5.1 General Strategy ..... 5 - 2
5.1.1 Comparison with other Algorithmic Strategies ..... 5 - 2
5.1.2 Steps of Dynamic Programming ..... 5 - 3
5.2 Principle of Optimality ..... 5 - 4
5.3 Elements of Dynamic Programming ..... 5 - 4
5.4 Applications of Dynamic Programming ..... 5 - 5
5.5 The 0/1 Knapsack Problem ..... 5 - 5
5.5.1 Algorithm ..... 5 - 30
5.5.2 Analysis ..... 5 - 31
5.6 Coin Change-making Problem ..... 5 - 31
5.7 Bellman-Ford Algorithm ..... 5 - 53
5.8 Multistage Graph Problem ..... 5 - 59
5.9 Travelling Salesman Problem ..... 5 - 65
5.10 Multiple Choice Questions with Answers ..... 5 - 70

Unit - IV

Chapter - 6 Backtracking (6 - 1) to (6 - 42)

6.1 General Method ..... 6 - 2
6.2 Recursive Backtracking and Iterative Backtracking Algorithm ..... 6 - 6
6.3 Applications of Backtracking ..... 6 - 8
6.4 The n-Queen Problem ..... 6 - 8
6.4.1 How to Solve n-Queen's Problem ? ..... 6 - 9
6.4.2 Algorithm ..... 6 - 13
6.5 Sum of Subsets ..... 6 - 19
6.6 Graph Coloring ..... 6 - 25
6.6.1 Algorithm ..... 6 - 27

6.6.2 Analysis ..... 6 - 28
6.7 The 0/1 Knapsack Problem ..... 6 - 31
6.8 Multiple Choice Questions with Answers ..... 6 - 41

Unit - V

Chapter - 7 Branch and Bound (7 - 1) to (7 - 72)

7.1 Basic Concept ..... 7 - 2
7.1.1 Comparison between Backtracking and Branch and Bound ..... 7 - 2
7.2 The Method ..... 7 - 2
7.2.1 General Algorithm for Branch and Bound ..... 7 - 3
7.3 Control Abstractions for Least Cost Search ..... 7 - 5
7.3.1 Control Abstraction for LC Search ..... 7 - 6
7.4 Bounding ..... 7 - 7
7.4.1 Job Sequencing with Deadlines ..... 7 - 8
7.5 FIFO Branch and Bound ..... 7 - 11
7.6 LC Branch and Bound ..... 7 - 14
7.7 0/1 Knapsack Problem ..... 7 - 16
7.7.1 LC Branch and Bound Solution ..... 7 - 18
7.7.2 FIFO Branch and Bound Solution ..... 7 - 20
7.8 Traveling Salesperson Problem ..... 7 - 36
7.8.1 Row Minimization ..... 7 - 38
7.8.2 Column Minimization ..... 7 - 38
7.8.3 Full Reduction ..... 7 - 38
7.8.4 Dynamic Reduction ..... 7 - 40
7.9 Multiple Choice Questions with Answers ..... 7 - 70

Unit - VI

Chapter - 8 Computational Complexity (8 - 1) to (8 - 22)

8.1 The Classes : P, NP, NP, Complete ..... 8 - 2

8.1.1 The Classes P and NP ..... 8 - 2
8.1.1.1 Example of P Class Problem ..... 8 - 3
8.1.1.2 Example of NP Class Problem ..... 8 - 4
8.1.1.3 Difference between P and NP Class Problems ..... 8 - 5
8.2 Non-Deterministic Algorithms ..... 8 - 7
8.2.1 Nondeterministic Algorithm for 0/1 Knapsack Problem ..... 8 - 9
8.3 The NP Hard Problem ..... 8 - 10
8.4 Concept of Reduction ..... 8 - 12
8.5 Satisfiability Problem ..... 8 - 13
8.6 Proofs for NP Complete Problems ..... 8 - 14
8.6.1 Clique ..... 8 - 14
8.6.2 Vertex Cover ..... 8 - 16
8.7 Multiple Choice Questions with Answers ..... 8 – 19
Laboratory ..... (L - 1) to (L - 22)
Solved Model Question Papers ..... (M - 1) to (M - 4)

UNIT - I

1 Introduction

Syllabus

Proof Techniques : *Contradiction, Mathematical Induction, Direct proofs, Proof by counter example, Proof by contraposition.*

Analysis of Algorithm : *Efficiency - Analysis framework, asymptotic notations - big O, theta and omega.*

Analysis of Non-recursive and recursive algorithms : *Solving Recurrence Equations using Masters theorem and Substitution method.*

Contents

1.1 *Introduction to Proof Techniques* ..... **May-07,08,09,10,11,13,17, Dec.-06,07,09,11,13,16,17,** ..... Marks 10
1.2 *Notion of Algorithm* ..... **Dec.-09, May-14** ..... Marks 6
1.3 *Efficiency - Analysis Framework* ..... **Dec.-08,10,11,12,13, May-09,14,15, April-15,** ..... Marks 10
1.4 *Asymptotic Notations* ..... **Dec.-06, 11,13, May-08,09,11,13, April-16,Dec.-13,** ..... Marks 10
1.5 *Solving Recurrence Equations* ..... **Dec.-06,07,11,13,15, May-07,09,10,15,16,** ..... Marks 10
1.6 *Analysis of Non-recursive Algorithms* ..... **May-12, May-15,** ..... Marks 12
1.7 *Analysis of Recursive Algorithms* ..... **May-08,15, Dec.-10,** ..... Marks 8
1.8 *Multiple Choice Questions* **GATE-05, 07, 08**

UNIT - III

5 Dynamic Programming

Syllabus

General strategy, Principle of optimality, 0/1 knapsack Problem, Coin change-making problem, Bellman - Ford Algorithm, Multistage Graph problem(using Forward computation), Travelling Salesman Problem

Contents

5.1	General Strategy	..... **Dec.-06, 07,08, 11, 18, May-19,**	Marks 6
5.2	Principle of Optimality	..... **Dec.-06, 07,13, May-15,** **April-18, March-20**	Marks 17
5.3	Elements of Dynamic Programming	..... **Dec.-06,07, May-07,08,10,**	Marks 6
5.4	Applications of Dynamic Programming		
5.5	The 0/1 Knapsack Problem	..... **May-07,08,15,17,18,** **Dec.-07,11,12, April-18,**	Marks 10
5.6	Coin Change-making Problem	..... **April-18,**	Marks 5
5.7	Bellman - Ford Algorithm	..... **May-19, April-18,**	Marks 10
5.8	Multistage Graph Problem	..... **May-18, April-10,**	Marks 10
5.9	Travelling Salesman Problem	..... **May-08,12,13,15, 18, Dec.-11, 18,**	Marks 10
5.10	Multiple Choice Questions		

5.1 General Strategy

SPPU : Dec.-06, 07,08, 11,18, End Sem. May-19, End Sem., Marks 6

Dynamic programming is typically applied to optimization problems.

For each given problem, we may get any number of solutions from which we seek for optimum solution (i.e. minimum value or maximum value solution). And such an optimal solution becomes the solution to the given problem.

5.1.1 Comparison with other Algorithmic Strategies

Divide and Conquer and Dynamic Programming

Sr. No.	Divide and Conquer	Dynamic Programming
1.	The problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of sub-problems are collected together to get the solution to the given problem.	In dynamic programming many decision sequences are generated and all the overlapping sub-instances are considered.
2.	In this method duplications in sub-solutions are neglected. i.e., duplicate subsolutions may be obtained.	In dynamic computing duplications in solutions is avoided totally.
3.	Divide and conquer is less efficient because of rework on solutions.	Dynamic programming is efficient than divide and conquer strategy.
4.	The divide and conquer uses top down approach of problem solving (recursive methods).	Dynamic programming uses bottom up approach of problem solving (iterative method).
5.	Divide and conquer splits its input at specific deterministic points usually in the middle.	Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points it determines which split point is optimal.

Greedy Algorithm Vs. Dynamic Programming

In this section we will discuss *"What are the differences and similarities between Greedy algorithm and dynamic programming ?"*

Sr. No.	Greedy method	Dynamic programming
1.	Greedy method is used for obtaining **optimum solution.**	Dynamic programming is also for obtaining **optimum solution.**
2.	In Greedy method a set of **feasible solutions** is obtained and the picks up the optimum solution.	There is **no** special set of **feasible solutions** in this method.
3.	In Greedy method the optimum selection is **without revising previously** generated solutions.	Dynamic programming considers **all possible sequences** in order to obtain the optimum solution.
4.	In Greedy method there is **no** as such **guarantee** of getting **optimum solution.**	It is guaranteed that the dynamic programming will generate optimal solution using **principle of optimality.**

5.1.2 Steps of Dynamic Programming

Dynamic programming design involves 4 major steps :

1. Characterize the structure of optimal solution. That means develop a mathematical notation that can express any solution and subsolution for the given problem.
2. Recursively define the value of an optimal solution.
3. By using bottom up technique compute value of optimal solution. For that you have to develop a recurrence relation that relates a solution to its subsolutions, using the mathematical notation of step 1.
4. Compute an optimal solution from computed information.

Ex. 5.1.1 *Comment on the statement : "In dynamic programming, many decision sequences may be generated".* SPPU : Dec.-08, Marks 2

Sol. : Dynamic programming is typically applied to optimization problems for a given problem we may get any number of solutions. From all those solutions we seek for optimum solution and such an optimum solution becomes the solution to the given problem. In this method subproblem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem. Hence in dynamic programming many decision sequences are generated and all the overlapping sub instances are considered.

Review Questions

Q.1 What is the major difference between greedy method and dynamic programming ?
SPPU : Dec.-06, 07, Marks 2, Dec.-18, End Sem, May-19, End Sem., Marks 5

Q.2 What are the common step in the dynamic programming to solve any problem ? Compare dynamic programming with greedy approach. SPPU : Dec.-11, Marks 6

5.2 Principle of Optimality

SPPU : Dec.-06, 07,13, May-15, Marks 17, April-18, March-20

The dynamic programming algorithm obtains the solution using principle of optimality.

The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal."

When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.

For example : Finding of shortest path in a given graph uses the principle of optimality.

Review Questions

Q.1 What is the "Principle of optimality" ?
SPPU : Dec.-06, 07, April 18, March-20, In Sem, Marks 2

Q.2 What is dynamic programming ? Is this the optimization technique ? Give reasons. What are its drawbacks ? Explain memory functions. SPPU : Dec.-13, Marks 17

Q.3 What is principle of optimality ? Differentiate between greedy and dynamic method.
SPPU : May-15 [End sem] Marks 5

5.3 Elements of Dynamic Programming

SPPU : Dec.-06,07, May-07,08,10, Marks 6

1) **Optimal substructure** - The dynamic programming technique makes use of principle of optimality to find the optimal solution from subproblems.

2) **Overlapping subproblems** - The dynamic programming is a technique in which the problem is divided into subproblems. The solutions of subproblems are shared to get the final solution to the problem. It avoids repetition of work and we can get the solution more efficiently.

Review Question

Q.1 Name the elements of dynamic programming. How does the dynamic programming solve the problem ? SPPU : Dec.-06, 07, May-07, 08, Marks 6, May-10, Marks 4

5.4 Applications of Dynamic Programming

Various applications that make use of dynamic programming strategy are -

1. The 0/1 Knapsack problems
2. Bellman Ford Algorithm
3. Multistage Graph Problem
4. Optimal Binary Search Trees
5. Traveling Salesman Problem

5.5 The 0/1 Knapsack Problem

SPPU : May - 07,08,15,17, 18, Dec.-07,11,12, April-18, Marks 10

As we know that dynamic programming is a technique for solving problems with overlapping subproblems. These subproblems are typically based on recurrence relation. In this section we will discuss the method of solving **knapsack problem** using dynamic programming approach. The knapsack problem can be defined as follows : If there are n items with the weights w_1, w_2, ... w_n and values (profit associated with each item) v_1, v_2, ... v_n and capacity of knapsack to be W, then find the most valuable subset of the items that fit into the knapsack.

To solve this problem using dynamic programming we will write the recurrence relation as :

table [i, j] = maximum {table [i – 1, j], v_i + table [i – 1, j – w_i]} if j ≥ w_i

table [i – 1, j] if j < w_i

That means, a table is constructed using above given formula. Initially,
table [0, j] = 0 as well as table [i, 0] = 0 when j ≥ 0 and i ≥ 0.
The initial stage of the table can be

	0	1		$j - w_i$		j		W
0	0	0	...	0	...	0	...	0
	:			table [i – 1, j – w_i]				
i – 1	0					table [i – 1, j]		
i	0					table [i, j]		
	:							
n	0							table [n, W]

Goal i.e., maximum value of items

The table [n, W] is a goal i.e., its gives the total items sum of all the selected items for the knapsack.

From this goal value the selected items can be traced out. Let us solve the knapsack problem using the above mentioned formula -

Ex. 5.5.1 *For the given instance of problem obtain the optimal solution for the knapsack problem.*

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

The capacity of knapsack is W = 5.

Sol. : Initially, table [0, j] = 0 and table [i, 0] = 0. There are 0 to n rows and 0 to W columns in the table.

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

Now we will fill up the table either row by row or column by column. Let us start filling the table row by row using following formula :

$$\text{table}[i, j] = \begin{cases} \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \text{ when } j \geq w_i \\ \text{or} \\ \text{table}[i-1, j] \text{ if } j < w_i \end{cases}$$

table [1, 1] With i = 1, j = 1, $w_i = 2$ and $v_i = 3$.

As $j < w_i$ we will obtain table [1, 1] as

$$\text{table}[1, 1] = \text{table}[i-1, j]$$
$$= \text{table}[0, 1]$$

$\therefore$ table [1, 1] = 0

table [1, 2] With i = 1, j = 2, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 2] as

$$\text{table}[1, 2] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \text{maximum } \{(\text{table}[0, 2]), (3 + \text{table}[0, 0]\}$$
$$= \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 2] = 3

table [1, 3] With i = 1, j = 3, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 3] as

$$\text{table}[1, 3] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \text{maximum } \{\text{table}[0, 3], 3 + \text{table}[0, 1]\}$$
$$= \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 3] = 3

table [1, 4] With i = 1, j = 4, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$, we will obtain table [1, 4] as

$$\text{table}[1, 4] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \text{maximum } \{\text{table}[0, 4], 3 + \text{table}[0, 2]\}$$
$$= \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 4] = 3

table [1, 5] With i = 1, j = 5, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 5] as

$$\text{table}[1, 5] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \text{maximum } \{\text{table}[0, 5], 3 + \text{table}[0, 3]\}$$
$$= \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 5] = 3

The table with these values can be

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

Now let us fill up next row of the table.

table [2, 1] With i = 2, j = 1, $w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 1] as

$$\text{table}[2, 1] = \text{table}[i-1, j]$$
$$= \text{table}[1, 1]$$

$\therefore$ table [2, 1] = 0

table [2, 2] With $i = 2, j = 2, w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 2] as

$$\text{table } [2, 2] = \text{table } [i - 1, j]$$
$$= \text{table } [1, 2]$$

$\therefore$ table [2, 2] = 3

table [2, 3] With $i = 2, j = 3, w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 3] as

$$\text{table } [2, 3] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [1, 3], 4 + \text{table } [1, 0]\}$$
$$= \text{maximum } \{3, 4 + 0\}$$

$\therefore$ table [2, 3] = 4

table [2, 4] With $i = 2, j = 4, w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 4] as

$$\text{table } [2, 4] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [1, 4], 4 + \text{table } [1, 1]\}$$
$$= \text{maximum } \{3, 4 + 0\}$$

$\therefore$ table [2, 4] = 4

table [2, 5] With $i = 2, j = 5, w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 5] as

$$\text{table } [2, 5] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [1, 5], 4 + \text{table } [1, 2]\}$$
$$= \text{maximum } \{3, 4 + 3\}$$

$\therefore$ table [2, 5] = 7

The table with these computed values will be

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

table [3, 1] With $i = 3, j = 1, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 1] as

$$\text{table } [3, 1] = \text{table } [i - 1, j]$$
$$= \text{table } [2, 1]$$

$\therefore$ table [3, 1] = 0

table [3, 2] With $i = 3, j = 2, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 2] as

$$\text{table } [3, 2] = \text{table } [i - 1, j]$$
$$= \text{table } [2, 2]$$

$\therefore$ table [3, 2] = 3

table [3, 3] with $i = 3, j = 3, w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 3] as

$$\text{table } [3, 3] = \text{table } [i - 1, j]$$
$$= \text{table } [2, 3]$$

$\therefore$ table [3, 3] = 4

table [3, 4] With $i = 3, j = 4, w_i = 4$ and $v_i = 5$

As $j \leq w_i$, we will obtain table [3, 4] as

$$\text{table } [3, 4] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [2, 4], 5 + \text{table } [2, 0]\}$$
$$= \text{maximum } \{4, 5 + 0\}$$

$\therefore$ table [3, 4] = 5

table [3, 5] With $i = 3, j = 5, w_i = 4$ and $v_i = 5$

As $j \geq w_i$, we will obtain table [3, 5] as

$$\text{table } [3, 5] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [2, 5], 5 + \text{table } [2, 1]\}$$
$$= \text{maximum } \{7, 5 + 0\}$$

$\therefore$ table [3, 5] = 7

The table with these values can be

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

table [4, 1] With $i = 4, j = 1, w_i = 5, v_i = 6$

As $j < w_i$, we will obtain table [4, 1] as

$$\text{table } [4, 1] = \text{table } [i - 1, j]$$
$$= \text{table } [3, 1]$$

$\therefore$ table [4, 1] = 0

table [4, 2] With $i = 4, j = 2, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 2] as

$$\text{table } [4, 2] = \text{table } [i - 1, j]$$
$$= \text{table } [3, 2]$$

$\therefore$ table [4, 2] = 3

table [4, 3] With $i = 4, j = 3, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 3] as

$$\text{table } [4, 3] = \text{table } [i - 1, j]$$
$$= \text{table } [3, 3]$$

$\therefore$ table [4, 3] = 4

table [4, 4] with $i = 4, j = 4, w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 4] as

$$\text{table } [4, 4] = \text{table } [i - 1, j]$$
$$= \text{table } [3, 4]$$

$\therefore$ table [4, 4] = 5

table [4, 5] With $i = 4, j = 5, w_i = 5$ and $v_i = 6$

As $j \geq w_i$, we will obtain table [4, 5] as

$$\text{table } [4, 5] = \text{maximum } \{\text{table } [i - 1, j], v_i + \text{table } [i - 1, j - w_i]\}$$
$$= \text{maximum } \{\text{table } [3, 5], 6 + \text{table } [3, 0]\}$$
$$= \text{maximum } \{7, 6 + 0\}$$

$\therefore$ table [4, 5] = 7

Thus the table can be finally as given below

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7)

This is the total value of selected items

How to find actual knapsack items ?

Now, as we know that table [n, W] is the total value of selected items, that can be placed in the knapsack. Following steps are used repeatedly to select actual knapsack item

```
Let, i = n and k = W then
while (i>0 and k>0)
{
    if(table [i,k] ≠ table[i-1,k])then
    mark i^th item as in knapsack
    i = i-1 and k=k- w_i          //selection of i^th item
  else
    i = i-1  //do not select i^th item
}
```

Let us apply these steps to the problem given in example 5.5.1. As we have obtained the final table –

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

→ Start from here

i = 4 and k = 5

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

i.e., table [4, 5] = table [3, 5]

∴ do not select i^{th} i.e., 4^{th} item.

Now set $i = i - 1$

$i = 3$

Capacity →

Items ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] = table [i – 1, k]

i.e., table [3, 5] = table [2, 5]

do not select i^{th} item i.e., 3^{rd} item.

Now set $i = i - 1 = 2$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
✓2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [2, 5] ≠ table [1, 5]

select i^{th} item.

That is, select 2^{nd} item.

Set $i = i - 1$ and $k = k - w_i$

i.e. $i = 1$ and $k = 5 - 3 = 2$

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓1	0	0	3	3	3	3
✓2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [1, 2] ≠ table [0, 2]

select i^{th} item.

That is select 1^{st} item.

Set $i = i - 1$ and $k = k - w_i$

i.e. $i = 0$ and $k = 2 - 2 = 0$

Thus we have selected **item 1** and **item 2** for the knapsack. This solution can also be represented by solution vector **(1, 1, 0, 0).**

Let us now discuss the algorithm for the knapsack problem using dynamic programming.

Ex. 5.5.2 *Using dynamic programming, solve the following knapsack instance :*
$n = 3$, $[\omega_1, \omega_2, \omega_3] = [1, 2, 2]$ *and* $[P_1, P_2, P_3] = [18, 16, 6]$ *and* $M = 4$.

Sol. :

Item	Weight	Value
1	1	18
2	2	16
3	2	6

The capacity of knapsak is M = 4. Initially table [0, j] and table [i, 0] = 0

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

$$\text{table } [i, j] = \begin{cases} \text{maximum } \{\text{table } [i-1, j], V_i + \text{table } [i-1, j-W_i]\} \\ \text{or} \\ \text{table } [i-1, j] \text{ if } j < W_i \end{cases}$$

table [1, 1] With i = 1, j = 1, W_i = 2 and V_i = 3.

As $j < W_i$ we will obtain table [1, 1] as

table [1, 1] = max {table [0, 1], 18 + table [0, 0]}

= max {0, 18}

∴ **table [1, 1] = 18**

table [1, 2] With i = 1, j = 2, W_i = 1 and V_i = 18.

table [1, 2] = max {table [0, 2], 18 + table [0, 1]

= max {0, 18 + 0}

∴ **table [1, 2] = 18**

table [1, 3] With i = 1, j = 3, W_i = 1 and V_i = 18.

table [1, 3] = max {table [0, 3], 18 + table [0, 2]

= max {0, 18 + 0}

∴ **table [1, 3] = 18**

table [1, 4] With i = 1, j = 4, W_i = 1 and V_i = 18

table [1, 4] = max {table [0, 4], 18 + table [0, 3]

= max {0, 18 + 0}

∴ **table [1, 4] = 18**

	0	1	2	3	4
0	0	0	0	0	0
1	0	18	18	18	18
2	0				
3	0				

Consider table [2, 1] with i = 2, j = 1, W_i = 2, V_i = 16

As $j < W_i$

table [i, j] = table [i – 1, j]

table [2, 1] = table [1, 1]

table [2, 1] = 18

table [2, 2] with i = 2, j = 2, W_i = 2, V_i = 16

As $j > W_i$

table [2, 2] = max {table [i – 1, j], V_i + table [i – 1, j – W_i]}

= max {table [1, 2], 16 + table [1, 0]}

= max [18, 16 + 0]

table [2, 2] = 18

table [2, 3] with i = 2, j = 3, W_i = 2, V_i = 16

table [2, 2] = 18 + 16 = 34

table [2, 4] with i = 2, j = 4, W_i = 2, V_i = 16

table [2, 4] = 34

	0	1	2	3	4
0	0	0	0	0	0
1	0	18	18	18	18
2	0	18	18	34	34
3	0				

table [3, 1] with i = 3, j = 1, W_i = 2, V_i = 6

As $j < W_i$

table [i, j] = table [i – 1, j]

table [3, 1] = table [2, 1]

table [3, 1] = 18

Thus we will fill up the table as

	0	1	2	3	4
0	0	0	0	0	0
1	0	18	18	18	18
2	0	18	18	34	34
3	0	18	18	24	34

table [3, 2] with $i = 3, j = 2, W_i = 2, V_i = 6$

table [3, 2] = max {table [i – 1, j], V_i + table [i – 1, j – W_i]}

= max {18, 6 + 0}

table [3, 2] = 18

table [3, 3] with $i = 3, j = 3, W_i = 2, V_i = 6$

table [3, 3] = max {18, 6 + 0} = 24

table [3, 4] with $i = 3, j = 4, W_i = 2, V_i = 6$

table [3, 4] = max {34, 6 + 18} = 34

Thus the maximal value is $ 34

As table [3, 4] = table [2, 4]

Do not select 3rd item.

Now set i = i = 3 – 1 = 2

Select item 2 and then item 1. The solution vector will be **(1, 1, 0)**

Ex. 5.5.3 *Apply the dynamic programming following instance of the knapsack problem and solve.* **SPPU : May-15, 18 [In Sem], Marks 5**

Item	Weight	Value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Capacity W = 5

Sol. : Initially, table [0, j] and table [i, 0] = 0. There are 0 to n rows and 0 to W columns. in the table.

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

Now we will fill up the table row by row. Let us start filling the table row by row using following formula :

$$\text{table } [i, j] = \begin{cases} \text{maximum } \{\text{table } [i-1, j], v_i + \text{table } [i-1, j-w_i] \\ \text{or} \qquad \text{when } j \geq w_i \\ \text{table } [i-1, j] \text{ if } j < w_i \end{cases}$$

table [1, 1] with $i = 1, j = 1$ and $w_i = 2, v_i = 12$

As $j < w_i$

table [1, 1] = table [i – 1, j]

= table [0, 1]

table [1, 1] = 0

table [1, 2] with $i = 1, j = 2, w_i = 2, v_i = 12$

As $j \geq w_i$

table [1, 2] = max {table [i – 1, j], v_i + table [i – 1, j – w_i]}

= max {0, 12 + 0}

∴ **table [1, 2] = 12**

table [1, 3] with $i = 1, j = 3, w_i = 2, v_i = 12$

As $j \geq w_i$

table [1, 3] = max {table [i – 1, j], v_i + table [i – 1, j – w_i]}

= max {0, 12 + 0}

∴ **table [1, 3] = 12**

table [1, 4] with $i = 1, j = 4, w_i = 2$ and $v_i = 12$

As $j \geq w_i$

table [1, 4] = max {table [i – 1, j], v_i + table [i – 1, j – w_i]}

= max {0, 12 + 0}

= 12

table [1, 5] with $i = 1, j = 5, w_i = 2$ and $v_i = 12$

As $j \geq w_i$

$$table\ [1, 5] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{0, 12 + 0\}$$

$\therefore$ **table [1, 5] = 12**

The table with these values will be.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					

table [2, 1] with $i = 2, j = 1, w_i = 1, v_i = 10$

As $j \geq w_i$

$$table\ [2, 1] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{0, 10 + 0\}$$

$\therefore$ **table [2, 1] = 10**

table [2, 2] with $i = 2, j = 2, w_i = 1, v_i = 10$

As $j \geq w_i$

$$table\ [2, 2] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [1, 2], 10 + table\ [1, 1]\}$$
$$= \max \{12, 10 + 0\}$$

table [2, 2] = 12

table [2, 3] with $i = 2, j = 3, w_i = 1, v_i = 10$

As $j \geq w_i$

$$table\ [2, 3] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [1, 3], 10 + table\ [1, 2]\}$$
$$= \max \{12, 10 + 12\}$$

$\therefore$ **table [2, 3] = 22**

table [2, 4] with $i = 2, j = 4, w_i = 1, v_i = 10$

As $j \geq w_i$

$$table\ [2, 4] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [1, 4], 10 + table\ [1, 3]\}$$

table [2, 4] = 22

table [2, 5] with $i = 2, j = 5, w_i = 1, v_i = 10$

As $j \geq w_i$

$$table\ [2, 5] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [1, 5], 10 + table\ [1, 4]\}$$

table [2, 5] = 22

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					

table [3, 1] with $i = 3, j = 1, w_i = 3, v_i = 20$

As $j \geq w_i$

$$table\ [3, 1] = table\ [i-1, j]$$
$$= table\ [2, 1]$$

$\therefore$ **table [3, 1] = 10**

table [3, 2] with $i = 3, j = 2, w_i = 3, v_i = 20$

As $j < w_i$

$$table\ [3, 2] = table\ [i-1, j]$$
$$= table\ [2, 2]$$

$\therefore$ **table [3, 2] = 12**

table [3, 3] with $i = 3, j = 3, w_i = 3, v_i = 20$

As $j \geq w_i$

$$table\ [3, 3] = \max \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [2, 3], 20 + table\ [2, 0]\}$$
$$= \max \{22, 20 + 0\}$$

$\therefore$ **table [3, 3] = 22**

table [3, 4] with $i = 3, j = 4, w_i = 3, v_i = 20$

As $j \geq w_i$

$$table\ [3, 4] = \{table\ [i-1, j], v_i + table\ [i-1, j-w_i]\}$$
$$= \max \{table\ [2, 4], 20 + table\ [2, 1]\}$$
$$= \max \{22, 20 + 10\}$$

$\therefore$ **table [3, 4] = 30**

table [3, 5] with $i = 3, j = 5, w_i = 3, v_i = 20$

As $j \geq w_i$

$$\text{table}[3, 5] = \max \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \max \{\text{table}[2, 5], 20 + \text{table}[2, 2]\}$$
$$= \max \{22, 20 + 12\}$$

$\therefore$ **table [3, 5] = 32**

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					

table [4, 1] with $i = 4, j = 1, w_i = 2, v_i = 15$

As $j < w_i$

$$\text{table}[4, 1] = \text{table}[i-1, j]$$
$$= \text{table}[3, 1]$$

$\therefore$ **table [4, 1] = 10**

table [4, 2] with $i = 4, j = 2, w_i = 2, v_i = 15$

As $j \geq w_i$

$$\text{table}[4, 2] = \max \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \max \{\text{table}[3, 2], 15 + \text{table}[3, 0]\}$$
$$= \max \{12, 15 + 0\}$$

$\therefore$ **table [4, 2] = 15**

table [4, 3] with $i = 4, j = 3, w_i = 2, v_i = 15$

As $j \geq w_i$

$$\text{table}[4, 3] = \max \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \max \{\text{table}[3, 3], 15 + \text{table}[3, 1]\}$$
$$= \max \{22, 15 + 10\}$$

$\therefore$ **table [4, 3] = 25**

table [4, 4] with $i = 4, j = 4, w_i = 2, v_i = 15$

As $j \geq w_i$

$$\text{table}[4, 4] = \max \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \max \{\text{table}[3, 4], 15 + \text{table}[3, 2]\}$$
$$= \max \{30, 15 + 12\}$$

$\therefore$ **table [4, 4] = 30**

table [4, 5] with $i = 4, j = 5, w_i = 2, v_i = 15$

As $j \geq w_i$

$$\text{table}[4, 5] = \max \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \max \{\text{table}[3, 5], 15 + \text{table}[3, 3]\}$$
$$= \max \{32, 15 + 22\}$$

$\therefore$ **table [4, 5] = 37**

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$i = 4, k = 4$

As $\text{table}[i, j] \neq \text{table}[i-1, k]$

i.e. $\text{table}[4, 5] \neq \text{table}[3, 5]$

Select i^{th} i.e. **4^{th} item.**

Now $i = i - 1$ and $k = k - w_i$

$\therefore$ $i = 4 - 1 = 3$ and $k = 5 - 2 = 3$

As $\text{table}[i, k] = \text{table}[i-1, k]$

i.e. $\text{table}[3, 3] = \text{table}[2, 3]$

$\therefore$ Do not select i^{th} i.e. 3^{rd} item.

Now set $i = i - 1$

$\therefore$ $i = 2$ and $k = 3$.

As $\text{table}[2, 3] \neq \text{table}[1, 3]$

$\therefore$ Select i^{th} i.e **2^{nd} item.**

Set $i = i - 1$ and $k = k - w_i$

$\therefore$ $i = 1$ and $k = 3 - 1 = 2$.

As $\text{table}[1, 2] \neq \text{table}[0, 2]$

Select i^{th} i.e. **1^{st} item.**

Now set $i = i - 1$ and $k = k - w_i$

$\therefore$ $i = 0$ and $k = 0$

Thus solution to this Knapsack problem is **(item 1, item 2, item 4) with total profit = 37.**

Ex. 5.5.4 *Consider 0/1 knapsack problem N = 3, w = (4, 6, 8), p = (10, 12, 15) using dynamic programming devise the recurrence relations for the problem and solve the same. Determine the optimal profit for the knapsack of capacity 10.* **SPPU : Dec.-11, 12, Marks 10**

Sol. : The recurrence relation would be

$$\text{table}[i, j] = \begin{cases} \max \{\text{table}[i-1, j], p_i + \text{table}[i-1, j-w_i]\} \text{ when } j \geq w_i, \\ \text{or} \\ \text{table}[i-1, j], \text{ if } j < w_i) \end{cases}$$

initially table [0, j] = 0 and table [i, 0] = 0. There will be 0 to n rows and 0 to W columns in table.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										

Now we will fill up table row by row

table [1,1] with i = 1, j = 1, $w_i = 4$, $p_i = 10$

As $j < w_i$

table [1, 1] = table [i – 1, j] = table [0, 1]

∴ **table [1, 1] = 0**

table with i = 1, j = 2, $w_i = 4$, $p_i = 10$

As $j < w_i$

table [1, 2] = table [i – 1, j] = table [0, 2]

table [1, 2] = 0

Thus table [1, 3] = 0

Now table [1, 4] with i = 1, j = 4, $w_i = 4$, $p_i = 10$ is

As $j = w_i$.

table [i, j] = max {table [i – 1, j], p_i + table [i – 1, j – w_i]}

table [1, 4] = max {table [0, 4], 10 + table [0, 0]}

table [1, 4] = 10

table [1, 5] with i = 1, j = 5, $w_i = 4$, $p_i = 10$.

Table [i, j] = max {table [i – 1, j], pi + table [i – 1, – w_i]

= max (table [0, 5], 10 table [0, 1])

table [1, 5] = 10

Similarly table [1, 6] = table [1, 7] = table [1, 8] = table [1, 9] = table [1, 10] = 10

The table will then be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10	10
2											
3											

table [2, 1] with i = 2, j = 1, $w_2 = 6$, $p_2 = 12$

As $j < w_2$

table [2, 1] = table [1, 1]

table [2, 1] = 0

Similarly table [2, 2] = table [2, 3] = table [2, 4] = table [2, 5] = 0

Now table [2, 6] with i = 2, j = 6, $w_2 = 6$, $p_2 = 12$.

As $j = w_2 = 6$

table [2, 6] = max {table [i – 1, j], p_i + table [i – 1, j – w_i]}

= max {table [1, 6], 12 + table [1, 0]}

table [2, 6] = 12

Similarly **table [2, 7] = table [2, 8] = table [2, 9] = 12**

Now with table [2, 10] , i = 2, j = 10, $w_2 = 6$, $p_2 = 12$.

table [2, 10] = max {table [i – 1, j], p_i + table [i – 1, j – w_i]}

= max {table [1, 10], 12 + table [1, 4]}

table [2, 10] = 22

The table will then be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0										

table [3, 1] with i = 3, j = 1, $w_3 = 8$, $p_3 = 15$.

$j < w_i$ is true for $j = 1$ to 7.

Hence

table [3, 1] = table [2, 1] = 0
table [3, 2] = table [2, 2] = 0
table [3, 3] = table [2, 3] = 0
table [3, 4] = table [3, 5] = 0
table [3, 6] = table [2, 6] = 12
table [3, 7] = table [2, 7] = 12

Now with table [3, 8] $i = 3$, $j = 8$, $w_3 = 8$, $p_3 = 15$.

As $j = w_i$

$$\text{table } [3, 8] = \max \{\text{table } [i-1, j], p_i + \text{table } [i-1, j-w_i]\}$$
$$= \max \{\text{table } [2, 8], 15 + \text{table } [2, 0]\}$$
$$\text{table } [3, 8] = 15$$

Similarly table [3, 9] = 15

But with table [3, 10] $i = 3$, $j = 10$, $w_3 = 8$, $p_3 = 15$.

$$\text{table } [3, 10] = \max \{\text{table } [2, 10], 15 + \text{table } [2, 2]\}$$
$$\textbf{table [3, 10]} = \textbf{22}$$

The table will be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	10	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0	0	0	0	0	0	12	12	15	15	22

Now let us apply steps to identify selected items.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	10	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0	0	0	0	0	0	12	12	15	15	22

$i = 3$, $k = 10$

check table $[i, k] \stackrel{?}{=}$ table $[i-1, k]$

check if table [3, 10] = table [2, 10]

i.e. $22 = 22$

$\therefore$ Do not select i^{th} i.e. 3^{rd} item.

Now set $i = i - 1$

$\therefore$ $i = 3 - 1 = 2$

Now $i = 2$, $k = 10$

$\therefore$ table $[2, 10] \stackrel{?}{=}$ table [1, 10]

As table [2, 10] $\neq$ table [1, 10]

i.e. $10 \neq 0$

$\therefore$ **Select i^{th} i.e. 2^{nd} item**

Set $i = i - 1$ and $k = k - w_i$

$\therefore$ $i = 2 - 1 = 1$ and $k = 10 - 6 = 4$

Now $i = 1$, $k = 4$

$\therefore$ table $[1, 4] \stackrel{?}{=}$ table [0, 4]

As $10 \neq 0$

Select i^{th} i.e. 1^{st} item

Set $i = i - 1$ $k = k - w_i$

$\therefore$ $i = i - 1$ $k = 4 - 4$

$i = 0$ $k = 0$.

Terminate the process by selecting item 1 and item 2

$\therefore$ Solution is **(1, 1, 0).**

Ex. 5.5.5 *Solve the knapsack problem using Dynamic programming for number of objects $n = 4$, given capacity $M = 8$.* **SPPU : April 18, In Sem., Marks 5**

Items	1	2	3	4
Value	15	10	9	5
Weight	1	5	3	4

Sol. : Initially table[0, j] = table[i,0] = 0 in the table[n, M]

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

Now we will fill up the table row by row using following formula -

$$table[i, j] = \max\{table[i-1], v_i + table[i-1, j-w_i]\} \text{ when } j \le w_i$$

or

$$table[i-1, j] \text{ if } j < w_i$$

Step 1 :

table[1, 1] with i = 1, j = 1, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 1], v_i + table [0, 0] = max{0, 15 + 0} ∴ **table[1, 1] = 15**	table[1, 2] with i = 1, j = 2, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 2], v_i + table [0, 1] = max{0, 15 + 0} ∴ **table[1, 2] = 15**
table[1, 3] with i = 1, j = 3, w_i = 1 V_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 3], v_i + table [0, 2] = max{0, 15 + 0} ∴ **table[1, 3] = 15**	table[1, 4] with i = 1, j = 4, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 4], v_i + table [0, 3] = max{0, 15 + 0} ∴ **table[1, 4] = 15**
table[1, 5] with i = 1, j = 5, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table [i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 5], v_i + table [0, 4] = max{0, 15 + 0} ∴ **table[1, 5] = 15**	table [1, 6] with i = 1, j = 6, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 6], v_i + table [0, 5] = max{0, 15 + 0} ∴ **table[1, 6] = 15**
table[1, 7] with i = 1, j = 7, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 7], v_i + table[0, 6] = max{0, 15 + 0} ∴ **table[1, 7] = 15**	table[1, 8] with i = 1, j = 8, w_i = 1 v_i = 15 As j ≥ w_i **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[0, 8], V_i + table [0, 7] = max{0, 15 + 0} ∴ **table[1, 8] = 15**

The partially filled table is

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0								
3	0								
4	0								

Step 2 :

table[2, 1] with i = 2, j = 1, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** table [i – 1, j] = table [1, 1] = 15 ∴ **table[2, 1] = 15**	table[2, 2] with i = 1, j = 2, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** table [i – 1, j] = table [1, 2] = 15 ∴ **table[2, 2] = 15**
table[2, 3] with i = 2, j = 3, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** table [i – 1, j] = table [1, 3] = 15 ∴ **table[2, 3] = 15**	table[2, 4] with i = 2, j = 4, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** table [i – 1, j] = table [1, 4] = 15 ∴ **table[2, 4] = 15**
table[2, 5] with i = 2, j = 5, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** max {table [i – 1, j] v_i + table [i – 1, j – w_i]} = max {table [1, 5], v_i + table [1, 0] = max {15, 10 + 0} ∴ **table[2, 5] = 15**	table[2, 6] with i = 2, j = 6, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** max {table [i – 1, j] v_i + table [i – 1, j – w_i]} = max {table [1, 6], v_i + table [1, 1] = max {15, 10 + 0} ∴ **table[2, 6] = 25**
table[2, 7] with i = 2, j = 7, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** max {table [i – 1, j] v_i + table [i – 1, j – w_i]} = max {table [1, 7], v_i + table [1, 2] = max {15, 15 + 15} ∴ **table[2, 7] = 25**	table[2, 8] with i = 2, j = 8, w_i = 5 v_i = 10 As j ≥ w_i **Formula Used :** max {table [i – 1, j] v_i + table [i – 1, j – w_i]} = max {table [1, 8], v_i + table [1, 3] = max {15, 10 + 15} ∴ **table[2, 8] = 25**

The partially filled table is

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0	15	15	15	15	15	15	15	15
3	0								
4	0								

Step 3 :

table[3, 1] with i = 3, j = 1, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** table[i – 1, j] = table[2, 1] = 15 **∴ table[3, 1] = 15**	table[3, 2] with i = 3, j = 2, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** table[i – 1, j] = table[2, 2] = 15 **∴ table[3, 2] = 15**
table[3, 3] with i = 3, j = 3, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 3], v_i + table [2, 0] = max{15, 9 + 0} **∴ table[3, 3] = 15**	table[3, 4] with i = 3, j = 4, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 4], v_i + table [2, 1] = max{15, 9 + 15} **∴ table[3, 4] = 24**
table[3, 5] with i = 3, j = 5, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 5], v_i + table [2, 2] = max{15, 9 + 15} **∴ table[3, 5] = 24**	table[3, 6] with i = 3, j = 6, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 6], V_i + table [2, 3] = max{25, 9 + 15} **∴ table[3, 6] = 25**
table[3, 7] with i = 2, j = 7, $w_i = 3$ $v_i = 9$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 7], V_i + table [2, 4] = max{25, 9 + 15} **∴ table[3, 7] = 25**	table[3, 8] with i = 3, j = 8, $w_i = 3$ $v_i = 9$ As $j \geq W_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[2, 8], v_i + table [2, 5] = max{25, 9 + 15} **∴ table[3, 8] = 25**

The partially filled table is

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0	15	15	15	15	15	15	15	15
3	0	15	15	15	15	15	15	15	15
4	0								

Step 4 :

table[4, 1] with i = 4, j = 1, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** table[i – 1, j] = table[3, 1] = 15 **∴ table[4, 1] = 15**	table[4, 2] with i = 4, j = 2, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** table[i – 1, j] = table [3, 2] = 15 **∴ table[4, 2] = 15**
table[4, 3] with i = 4, j = 3, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** table[i – 1, j] = table[3, 3] = 15 **∴ table[4, 3] = 15**	table[4, 4] with i = 4, j = 4, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[3, 4], v_i + table [3, 0] = max{24, 5 + 0} **∴ table[4, 4] = 24**
table[4, 5] with i = 4, j = 5, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[3, 5], v_i + table [3, 1] = max{24, 5 + 15} **∴ table[4, 5] = 24**	table[4, 6] with i = 4, j = 6, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[3, 6], v_i + table [3, 2] = max{25, 5 + 15} **∴ table[4, 6] = 25**
table[4, 7] with i = 4, j = 7, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[3, 7], v_i + table [3, 3] = max{25, 5 + 15} **∴ table[4, 7] = 25**	table[4, 8] with i = 4, j = 8, $w_i = 4$ $v_i = 5$ As $j \geq w_i$ **Formula Used :** max{table[i – 1, j], v_i + table [i – 1, j – w_i]} = max{table[3, 8], v_i + table [3, 4] = max{25, 5 + 24} **∴ table[4, 8] = 29**

The partially filled table is

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0	15	15	15	15	15	15	15	15
3	0	15	15	15	15	15	15	15	15
4	0	15	15	15	15	15	15	15	15

Selection of items

Step 1 : i = 4, k = 8 As table[i, k] ≠ table[i – 1, k] i.e. table[4, 8] ≠ table[3, 8] **Select i^{th} i.e. 4^{th} item**	**Step 2 :** Now i = i – 1 and k = w_i ∴ i = 3 and k = 8 – 4 = 4 As table[i, k] ≠ table[i – 1, k] i.e. table[3, 4] ≠ table[2, 4] **Select i^{th} i.e. 3^{rd} item**
Step 3 : Now i = i – 1 and k = k – w_i ∴ i = 2 and k = 4 – 3 = 1 As table[i, k] ≠ table[i – 1, k] i.e. table[2, 1] = table[1, 1] **Do not Select i^{th} i.e. 2^{nd} item**	**Step 4 :** Now i = i – 1 ∴ i = 1 and k = 1 As table[i, k] ≠ table[i – 1, k] i.e. table[1, 1] ≠ table[0, 1] **Select i^{th} i.e. 1^{st} item**

Thus the solution to Knapsack item is **(1, 0, 1, 1)**

5.5.1 Algorithm

```
Algorithm Dynamic_Knapsack(n,W,w[ ],v[ ])
//Problem Description:This algorithm is for obtaining knapsack
//solution using dynamic programming
//Input: n is total number of items, W is the capacity of
//knapsack, w[ ] stores weights of each item and v[ ] stores
//the values of each item.
//Output: Returns the total value of selected items for the
//knapsack.
for (i ←0 to n) do
{
for (j←0 to W) do
{
```



```
  table[i,0]=0  // table initialization
  table[0,j]=0
}
}
for (i←0 to n) do
{
for (j←0 to W) do
{
 if(j<w[i]) then
  table[i,j]← table[i-1,j]
 else if(j>=w[i]) then
  table[i,j]←max (table[i-1,j],(v[i]+table[i-1,j-w[i]]) )
}
}
return table[n,W]
```

5.5.2 Analysis

In this algorithm the basic operation is **if ... else if** statement within two nested for loops.

Hence

$$c(n) = \sum_{i=0}^{n} \sum_{j=0}^{W} 1 = \sum_{i=0}^{n} W - 0 + 1$$

$$= \sum_{i=0}^{n} (W+1) = \sum_{i=0}^{n} W + \sum_{i=0}^{n} 1$$

$$= W \cdot \sum_{i=0}^{n} 1 + \sum_{i=0}^{n} = W(n-0+1) + (n-0+1)$$

$$= W_n + W + n + 1$$

Thus $c(n) \approx W_n$ **The time complexity of this algorithm is Θ (nW).**

Review Question

Q.1 Write an algorithm for 0/1 knapsack problem using dynamic programming approach.

SPPU : May-07, 08, Dec.-07, Marks 8

5.6 Coin Change-making Problem

SPPU : April-18, Marks 5

Suppose there are some coins available. The values of these coins are as follows.

1 dollar = 100 cents

1 quarter = 25 cents
1 dimes = 10 cents
1 nickels = 5 cents
1 pennies = 1 cent

Now the making change problem is to pay the desired amount of money by using as few coins as possible.

For example : If a customer wants to pay 3.37 $ then he should pay 3 dollars (= 300 cents) + 1 quarter (= 25 cents) + 1 dime (= 10 cents) + 2 pennies (= 2 cents) = 3.37 $.

This method uses **Greedy** approach because at every step it chooses the largest coin it can. Furthermore, once the coin is selected then that is the final. This approach does not allow to change the decision. Unfortunately although **Greedy** approach is very efficient, it works only for limited number of instances. For example if there exists coins of 1, 4 and 6 units and we have to make change for 9 units. Then there are two ways :

1. Greedy Method : Select one coin of 6 unit and 3 coins of 1 unit = **4 coins.**

2. Better Method : Select two coins of 4 units and 1 coin of 1 unit = **3 coins.**

This better method is devised by dynamic programming approach.

For solving this problem using dynamic programming approach we need to build a table, in which rows correspond to denomination and column corresponds to 0 to N units.

Ex. 5.6.1 *Solve making change problem for $d_1 = 1$, $d_2 = 4$, $d_3 = 6$, $n = 3$ and $N = 8$ units.*

Sol. : Given that :

$d_1 = 1, d_2 = 4, d_3 = 6, n = 3, N = 8$ units.

Hence we will create a table with rows ranging from 1 to 3 and columns ranging from 0 to 8.

Initially c [i, 0] = 0 ∴ c [1, 0] = c [2, 0] = c [3, 0] = 0

j →

		0	1	2	3	4	5	6	7	8
i = 1	$d_1 = 1$	0								
i = 2	$d_2 = 4$	0								
i = 3	$d_3 = 6$	0								

We can perform computations row-by-row from left to right, or column-by-column from top to bottom or diagonally.

Here we will perform computation **column-by-column** and fill up the table accordingly.

We will use following formula for computations.

1. If i = 1 then c [i, j] = 1 + c [1, j – d_1]
2. If j < d_i then c [i, j] = c [i – 1, j]
3. Otherwise c [i] [j] = min (c [i – 1, j], 1 + c [i, j – d_i])

c [1, 1] with i = 1, j = 1 d_1 = 1.

As i = 1

Formula used : 1 + c [1, j – d_1]
= 1 + c [1, 1 – 1]]
= 1 + c [1, 0]
= 1 + 0
= 1

∴ **c [1, 1] = 1**

c [2, 1] with i = 2, j = 1, d_2 = 4

As j < d_2 i.e. 1 < 4

Formula used : c [i – 1, j]
= c [1, 1]
= 1

∴ **c [2, 1] = 1**

c [3, 1] with i = 3, j = 1, d_3 = 6

As j < d_3 i.e. 1 < 6

Formula used : c [i – 1, j]
= c [3 – 1, 1]
= c [2, 1]
= 1

∴ **c [3, 1] = 1**

c [1, 2] with i = 1, j = 2, d_1 = 1

As i = 1

Formula used : 1 + c [1, j – d_1]
= 1 + c [1, 2 – 1]
= 1 + c [1, 1]
= 1 + 1

$= 2$

$\therefore$ **c [1, 2] = 2**

c [2, 2] with i = 2, j = 2, $d_2 = 4$

As $j < d_2$ i.e. $2 < 4$

Formula used : $c[i-1, j]$

$= c[1, 2]$

$= 2$

$\therefore$ **c [2, 2] = 2**

c [3, 2] with i = 3, j = 2, $d_3 = 6$

As $j < d_3$ i.e. $2 < 6$

Formula used : $c[i-1, j]$

$= c[2, 2]$

$= 2$

$\therefore$ **c [2, 2] = 2**

c [1, 3] with i = 1, j = 3, $d_1 = 1$

As i = 1

Formula used : $1 + c[1, j - d_1]$

$= 1 + c[1, 3-1]$

$= 1 + c[1, 2]$

$= 1 + 2$

$= 3$

$\therefore$ $c[1, 3] = 3$

c [2,3] with i = 2, j = 3, $d_2 = 4$

As $j < d_3$ i.e. $3 < 4$

Formula used : $c[i-1, j]$

$= c[1,3]$

$= 3$

$\therefore$ **c [2, 3] = 3**

c [3, 3] with i = 3, j = 3, $d_3 = 6$

As $j < d_3$ i.e. $3 < 6$

Formula used : $c[i-1, j]$

$= c[2, 3]$

$= 3$

$\therefore$ **c [3, 3] = 3**

c [1, 4] with i = 1, j = 4, $d_1 = 1$

As i = 1

Formula used : $1 + c[1, j - d_1]$

$= 1 + c[1, 4-1]$

$= 1 + c[1, 3]$

$= 1 + 3$

$= 4$

$\therefore$ **c [1, 4] = 4**

c [2, 4] with i = 2, j = 4, $d_2 = 4$

As $i \neq 1$ and $j > d_2$

Formula used : $\min(c[i-1, j], 1 + c[i, j - d[i]])$

$= \min(c[2-1, 4], 1 + c[2, 4-4])$

$= \min(c[1, 4]\ 1 + c[2, 0])$

$= \min(4, 1 + 0)$

$= 1$

$\therefore$ **c [2, 4] = 1**

c [3, 4] with i = 3, j = 4, $d_3 = 6$

As $j < d_3$ i.e. $4 < 6$

Formula used : $c[i-1, j]$

$= c[2, 4]$

$= 1$

$\therefore$ **c [3, 4] = 1**

c [1, 5] with i = 1, j = 5, $d_1 = 1$

As i = 1

Formula used : $1 + c[1, j - d_1]$

$= 1 + c[1, 5-1]$

$= 1 + c[1, 4]$

$= 1 + 4$

$= 5$

$\therefore$ **c [1, 5] = 5**

c[2, 5] with i = 2, j = 5, $d_2 = 4$

As $i \neq 1$ and $j > d_2$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_2])$

$= \min (c [1, 5], 1 + c [2, 5 - 4]$

$= \min (c [1, 5], 1 + c \{2, 1])$

$= \min (5, 1 + 1)$

$= 2$

$\therefore$ **c [2, 5] = 2**

c [3, 5] with i = 3, j = 5, $d_3 = 6$

As $j < d_3$ i.e. $5 < 6$

Formula used : $c [i - 1, j]$

$= c [3 - 1, 5]$

$= c [2, 5]$

$\therefore$ **c [3, 5] = 2**

c [1, 6] with i = 1, j = 6, $d_1 = 1$

As i = 1

Formula used : $1 + c [1, j - d_1]$

$= 1 + c [1, 6 - 1]$

$= 1 + c [1, 5]$

$= 6$

$\therefore$ **c [1, 6] = 6**

c [2, 6] with i = 2, j = 6, $d_2 = 4$

As $i \neq 2$ and $j > d_2$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_2]$

$= \min (c [1, 6], 1 + c [2, 2])$

$= \min (6, 1 + 2)$

$= 3$

$\therefore$ **c [2, 6] = 3**

c [3, 6] with i = 3, j = 6, $d_3 = 6$

As $i \neq 3$ and $j > d_3$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_3])$

$= \min (c [2, 6], 1 + c [3, 0]$

$= \min (3, 1 + 0)$

$= 1$

$\therefore$ **c [3, 6] = 1**

c [1, 7] with i = 1, j = 7 $d_1 = 1$

As i = 1

Formula used : $1 + c [1, j - d_1]$

$= 1 + c [1, 6]$

$= 1 + 6$

$\therefore$ **c [1, 7] = 7**

c [2, 7] with i = 2, j = 7, $d_2 = 4$

As $i \neq 1$ and $j > d_2$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_2])$

$= \min (c [1, 7], 1 + c [2, 3])$

$= \min (7, 1 + 3)$

$= 4$

$\therefore$ **c [2, 7] = 4**

c [3, 7] with i = 3, j = 7, $d_3 = 6$

As $i \neq 1$ and $j > d_3$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_3]$

$= \min (c [2, 7], 1 + c [3, 1])$

$= \min (4, 1 + 1)$

$= 2$

$\therefore$ **c [3, 7] = 2**

c [1, 8] with i = 1, j = 8, $d_1 = 1$

As i = 1

Formula used : $1 + c [1, j - d_1]$

$= 1 + c [1, 7]$

$= 1 + 7$

$= 8$

$\therefore$ **c [1, 8] = 8**

c [2,8] with i = 2, j = 8, $d_2 = 4$

As $i \neq 1$ and $j > d_2$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_2])$

$= \min (c [1, 8], 1 + c[2, 4])$

$= \min (8, 1 + 1)$

$= 2$

$\therefore$ **c [2, 8] = 2**

c [3, 8] with i = 3, j = 8, $d_3 = 6$

As $i \neq 1$ and $j > d_3$

Formula used : $\min (c [i - 1, j], 1 + c [i, j - d_3]$

$= \min (c [2, 8], 1 + c [3, 2])$

$= \min (2, 1 + 2)$

$= 2$

$\therefore$ **c [3, 8] = 2**

Thus we can represent the table filled up by above computations will be

		0	1	2	3	4	5	6	7	8
i = 1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
i = 2	$d_2 = 4$	0	1	2	3	1	2	3	4	2
i = 3	$d_3 = 6$	0	1	2	3	1	2	1	2	2

↑

Total number of coins

The value c [n, N] i.e. c [3, 8] = 2 represents the minimum number of coins required to get the sum for 8 units. Hence we require 2 coins for getting 8 units. The coins are :

4 units = 1 coin

\+ 4 units = 1 coin

8 units = 2 coins

Ex. 5.6.2 *Given coins of denominations 2, 4 and 5 with amount to be pay is 7. Find optimal number of coins and sequence of coins used to pay given amount using dynamic method.*

Sol. : $d_1 = 2$, $d_2 = 4$, $d_3 = 5$. N = 7

We will create a table with rows ranging from 1 to 3 and column ranging from 0 to 7.

Initially C [i, 0] = 0

$\therefore$ C[1, 0] = C [2, 0] = C [3, 0] = 0

		0	1	2	3	4	5	6	7	8
i = 1	$d_1 = 2$	0								
i = 2	$d_2 = 4$	0								
i = 3	$d_3 = 5$	0								

Computations can be performed column by column; and fill up the table accordingly.

C [1, 1] with i = 1, j = 1, $d_1 = 2$

As i = 1

C [1,1] = 1+ C[1, j – d1]

C [1,1] = 1+ C[1, – 1]

C [1, 1] = ∞

C [2, 1] with i = 2, j = 1, $d_2 = 4$

C [2, 1] = C [i – 1, j]

C [2, 1] = C [1, 1]

C [2, 1] = ∞

C [3, 1] with i = 3, j = 1, $d_3 = 5$

As $j < d_3$

C [i, j] = C [i – 1, j]

C [3, 1] = C [2, 1]

$\therefore$ **C [3, 1] = ∞**

C [1, 2] with i = 1, j = 2, $d_1 = 2$

As i = 1

C [i, j] = 1+ C [1, j – d_1]

C [1, 2] = 1 + C [1, 0] = 1 + 0

$\therefore$ **C [1, 2] = 1**

C [2, 2] with i = 2, j = 2, $d_2 = 4$

As $j < d_2$

$C[i, j] = C[i-1, j]$

$C[2, 2] = C[1, 2]$

$C[2, 2] = 1$

C[3, 2] with $i = 3, j = 2, d_3 = 5$

As $j < d_3$

$C[i, j] = C[i-1, j]$

$\therefore \quad C[3, 2] = C[2, 2]$

$\therefore \quad$ **$C[3, 2] = 1$**

C [1, 3] with $i = 1, j = 3\ d_1 = 2$

As i = 1

$C[i, j] = 1 + C[1, j - d_1]$

$= 1 + C[1, 3-2] = 1 + C[1, 1]$

$C[1, 3] = \infty$

C [2, 3] with $i = 2, j = 3, d_2 = 4$

As $j < d_2$

$C[i, j] = C[i-1, j]$

$C[2, 3] = C[1, 3]$

$\therefore \quad$ **$C[2, 3] = \infty$**

C [3, 3] with $i = 3, j = 3, d_3 = 5$

As $j < d_3$

$C[i, j] = C[i-1, j]$

$\therefore \quad C[3, 3] = C[2, 3]$

$\therefore \quad$ **$C[3, 3] = \infty$**

C [1, 4] with $i = 1, j = 4, d_1 = 2$

As i = 1

$C[i, j] = 1 + C[1, j - d_1]$

$C[1, 4] = 1 + C[1, 2] = 1 + 1$

$\therefore \quad$ **$C[1, 4] = 2$**

C [2, 4] with $i = 2, j = 4, d_2 = 4$

As $d_2 = j$ and $i \neq 1$

$C[i, j] = \min(C[i-1, j], 1 + C[i, j - d[i]])$

$= \min(C[1, 4], 1 + C[2, 4-4])$

$= \min(C[1, 4], 1 + 0)$

$C[2, 4] = \min(2, 1)$

C [3, 4] with $i = 3, j = 4$ and $d_3 = 5$

As $j < d_3$

$C[i, j] = C[i-1, j]$

$C[3, 4] = C[2, 4]$

$\therefore \quad$ **$C[3, 4] = 1$**

C [1, 5] with $i = 1, j = 5$ and $d_1 = 2$

As i = 1

$C[i, j] = 1 + C[i, j - d_1]$

$= 1 + C[1, 3]$

$C[1, 5] = \infty$

C [2, 5] with $i = 2, j = 5$ and $d_2 = 4$

$C[i, j] = \min(C[i-1, j]\ 1 + C[i, j - d[i]])$

$= \min(C[1, 5], 1 + C[2, 1])$

$C[2, 5] = \min(\infty, 1 + \infty)$

$C[2, 5] = \infty$

C[3, 5] with $i = 3, j = 5$ and $d_3 = 5$

$C[i, j] = \min(C[i-1, j], 1 + C[i, j - d[i]])$

$= \min(C[2, 5], 1 + C[3, 0])$

$C[3, 5] = \min(\infty, 1 + 0)$

$C[3, 5] = 1$

C[1, 6] with $i = 1, j = 6, d_1 = 2$

As i = 1

$C[i, j] = 1 + C[i, j - d_1]$

$C[1, 6] = 1 + C[1, 4] = 1 + 2$

$C[1, 6] = 3$

C [2, 6] with $i = 2, j = 6, d_2 = 4$

$C[i, j] = \min(C[i-1, j], 1 + C[i, j - d[i]])$

$= \min(C[1, 6], 1 + C[2, 2])$

$C[2, 6] = \min(3, 1+1)$

C [2, 6] = 2

C [3, 6] with i = 3, j = 6, $d_3 = 5$

C[i, j] = min (C [i – 1, j], 1 + C [i, j – d [i]])
= min (C [2, 6], 1 + C [3, 1])
= min (2, 1 + ∞)

C[3, 6] = 2

C [1, 7] with i = 1, j = 7 and $d_1 = 2$

As i = 1

C [i, j] = 1+ C [i, j – d_1]
= 1 + C [1, 5] = 1 + ∞

C [1, 7] = ∞

C [2,7] with i = 2, j = 7 and $d_2 = 4$

C [i, j] = min (C [i – 1, j], 1 + C [i, j – d [i]])
= min (C [1, 7], 1 + C [2, 3])
= min (∞, 1 + ∞)

C [2, 7] = ∞

C [3, 7] with i = 3, j = 7 and $d_3 = 5$

C [i, j] = min (C [i – 1, j], 1 + C [i, j – d [i]])
= min (C [2, 7], 1 + C [3, 2])
= min (∞, 1 + 1)

C [3, 7] = 2

The table can be

		0	1	2	3	4	5	6	7
i = 1	$d_1 = 2$	0	∞	1	∞	2	∞	3	∞
i = 2	$d_2 = 4$	0	∞	1	∞	1	∞	2	∞
i = 3	$d_3 = 5$	0	∞	1	∞	1	1	2	2

↑
Total number of coins

That means 2 coins are required for sum as 7. The coins are,

$d_1 = 2 \rightarrow 1$ coin
$d_3 = 5 \rightarrow 1$ coin

Total 7 with 2 coins

Ex. 5.6.3 *Solve making change problem using Dynamic Programming. (Denominations : d1 = 1, d2 = 4, d3 = 6). Give your answer for making change of ₹ 9.*

Sol. : $d_1 = 1$, $d_2 = 4$, $d_3 = 6$, N = 9 ₹, n = 3. Hence we will create a table with rows ranging from 1 to 3 and columns ranging from 0 to 9.

Initially C[i, 0] = 0 C[1, 0] = C [2, 0] = C [3, 0] = 0

j →

	0	1	2	3	4	5	6	7	8	9
$d_1 = 1$	0									
$d_2 = 4$	0									
$d_3 = 6$	0									

We will complete the table **column-by-column :** Refer example 5.6.1.

	0	1	2	3	4	5	6	7	8	9
$d_1 = 1$	0	1	2	3	4	5	6	7	8	
$d_2 = 4$	0	1	2	3	1	2	3	4	2	
$d_3 = 6$	0	1	2	3	1	2	1	2	2	

c[1, 9] with i = 1, j = 9, $d_1 = 1$

As i = 1

Formula used : 1 + c [1, j – d_1]
= 1 + c [1, 8]
= 1 + 8
= 9

∴ **c [1, 9] = 9**

c [2, 9] with i = 2, j = 9, $d_2 = 4$

As i ≠ 1 and j > d_2

Formula used : min (c [i – 1, j], 1 + c [i, j – d_2])
= min (c [1, 9], 1 + c [2, 5])
= min (9, 1 + 2)
= 3

∴ **c [2, 9] = 3**

c [3, 9] with i = 3, j = 9, $d_3 = 6$

As i ≠ 1 and j > d_3

Formula used : min (c [i – 1, j], 1 + c [i, j – d_3])

= min (c [2, 9], 1 + c [3, 3])

= min (3, 1 + 3)

= 3

∴ **c [3, 9] = 3**

Thus the complete table filled up by all the computations will be

	0	1	2	3	4	5	6	7	8	9
$d_1 = 1$	0	1	2	3	4	5	6	7	8	9
$d_2 = 4$	0	1	2	3	1	2	3	4	2	3
$d_3 = 6$	0	1	2	3	1	2	1	2	2	3

Total number of coins

The value c [n, N] i.e. c [3, 9] = 3 represents the minimum number of coins required to get the sum for 9 units. Hence coins will be

$d_2 = 4$	1 coin
$d_2 = +4$	1 coin
$d_1 = +1$	1 coin
₹ 9	= 3 coins

Ex. 5.6.4 *Solve making change problem using dynamic technique. d1 = 1, d2 = 3, d3 = 5, d4 = 6. Calculate for making change of ₹ 8.*

Sol. : Let,

d1 = 1, d2 = 3, d3 = 5, d4 = 6

We will create a table with rows ranging from 1 to 4 and columns ranging from 0 to 8.

Initially c [i, 0] = 0

		0	1	2	3	4	5	6	7	8
i = 1	d1 = 1	0								
i = 2	d2 = 3	0								
i = 3	d3 = 5	0								
i = 4	d4 = 6	0								

Computations can be performed column by column and fill up the table.

c [1, 1] with i = 1, j = 1, d1 = 1.

As i = 1

c [1, 1] = 1 + c [1, j – d1]

= 1 + c [1, 0]

= 1 + 0

c [1, 1] = 1.

c [2, 1] with i = 2, j = 1, d2 = 3.

Here j < di i.e. 1 < 3

∴ c [i, j] = c [i – 1, j]

c [2, 1] = c [2 – 1, 1]

= c [1, 1]

c [2, 1] = 1

c [3, 1] with i = 3, j = 1, d3 = 5

j < di i.e. 1 < 5

∴ c [i, j] = c [i – 1, j]

c [3, 1] = c [3 – 1, 1] = c [2, 1]

c [3, 1] = 1

c [4, 1] with i = 4, j = 1, d4 = 6

j < d4 i.e. 1 < 6

∴ c [i, j] = c [i – 1, j]

c [4, 1] = c [4 – 1, 1]

= c [3, 1]

c [4, 1] = 1

c [1, 2] with i = 1, j = 2, d1 = 1

Here As i = 1

∴ c [i, j] = 1 + c [1, j – d1]

= 1 + c [1, 1] = 1 + 1

c [1, 2] = 2

c [2, 2] with i = 2, j = 2, d2 = 3

As j < di i.e. 2 < 3

∴ c [i, j] = c [i – 1, j]

= c [2 – 1, 2]

$= c [1, 2]$

$c [2, 2] = 2$

c [3, 2] with i = 3, j = 2, d3 = 5

As $j < di$ i.e. $2 < 5$

$\therefore \quad c [i, j] = c [i - 1, j]$

$= c [3 - 1, 2]$

$= c [2, 2]$

c [3, 2] = 2

c [4, 2] with i = 4, j = 2, d4 = 6

As $j < di$ i.e. $2 < 6$

$\therefore \quad c [i, j] = c [i - 1, j]$

$= c [4 - 1, 2]$

$= c [3, 2]$

c [4, 2] = 2

c [1, 3] with i = 1, j = 3, d1 = 1

As i = 1

$\therefore \quad c [i, j] = 1 + c [1, j - d1]$

$= 1 + c [1, 3 - 1]$

$= 1 + c [1, 2]$

c [1, 3] = 3

c [2, 3] with i = 2, j = 3, d2 = 3

$i \neq 1$ and $j = d_i$

$\therefore$ Formula used $c [i, j] = \min (c [i - 1, j], 1 + c [i, j - di])$

$\therefore \quad c [2, 3] = \min (c [2 - 1, 3], 1 + c [2, 3 - 3])$

$= \min (c [1, 3], 1 + c [2, 0])$

$= \min (3, 1 + 0)$

c [2, 3] = 1

c [3, 3] with i = 3, j = 3, d3 = 5

$j < di$ i.e. $3 < 5$

$\therefore \quad c [i, j] = c [i - 1, j]$

$= c [3 - 1, 3]$

$= c [2, 3]$

c [3, 3] = 1

c [4, 3] with i = 4, j = 3, d4 = 6

As $j < d4$

$\therefore \quad c [i, j] = c [i - 1, j]$

$= c [4 - 1, 3]$

$= c [3, 3]$

c [4, 3] = 1

Partially the table will be -

	0	1	2	3	4	5	6	7	8
d1 = 1	0	1	2	3					
d2 = 3	0	1	2	1					
d3 = 5	0	1	2	1					
d4 = 6	0	1	2	1					

c [1, 4] with i = 1, j = 4, d1 = 1

As i = 1

$\therefore \quad c [i, j] = 1 + c [1, j - d1]$

$= 1 + c [1, 4 - 1]$

$= 1 + c [1, 3]$

c [1, 4] = 4

c [2, 4] with i = 2, j = 4, d2 = 3

As $i \neq 1$ and $j > d2$

$\therefore \quad c [i, j] = \min (c [i - 1, j], 1 + c [i, j - d_i])$

$= \min (c [2 - 1, 4], 1 + c [2, 4 - 3])$

$= \min (c [1, 4], 1 + c [2, 1])$

$= \min (4, 1 + 1)$

c [2, 4] = 2

c [3, 4] with i = 3, j = 4, d3 = 5

As $j < d_i$ i.e. $4 < 5$

$\therefore \quad c [i, j] = c [i - 1, j]$

$= c [3 - 1, 4]$

$= c [2, 4]$

c [3, 4] = 2

c [4, 4] with i = 4, j = 4, d4 = 6

As $j < d_i$

∴

$$c[i, j] = c[i-1, j] = c[4-1, 4]$$
$$= c[3, 4]$$

c [4, 4] = 2

c [1, 5] with i = 1, j = 5, d1 = 1.

As i = 1

$$c[i, j] = 1 + c[i, j-d1]$$
$$= 1 + c[1, 5-1]$$
$$= 1 + c[1, 4]$$

c [1, 5] = 5.

c [2, 5] with i = 2, j = 5, d2 = 3

Here $i \neq 1$ and $j > di$

∴ Formula used

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j-di])$$
$$c[2, 5] = \min(c[2-1, 5], 1 + c[2, 5-3])$$
$$= \min(c[1, 5], 1 + c[2, 2])$$
$$= \min(5, 1 + 2)$$

c [2, 5] = 3

c [3, 5] with i = 3, j = 5, d3 = 5

$$c[3, 5] = \min(c[i-1, j], 1 + c[i, j-di])$$
$$= \min(c[3-1, 5], 1 + c[3, 5-5])$$
$$= \min(c[2, 5], 1 + c[3, 0])$$
$$= \min(3, 1 + 0)$$

c [3, 5] = 1

c [4, 5] with i = 4, j = 5, d4 = 6.

As j < d4

∴

$$c[i, j] = c[i-1, j] = c[4-1, 5]$$
$$= c[3, 5]$$

c [4, 5] = 1

c [1, 6] with i = 1, j = 6, d1 = 1

As i = 1

$$c[i, j] = 1 + c[i, j-d1]$$
$$= 1 + c[1, 6-1]$$
$$= 1 + c[1, 5]$$

c [1, 6] = 6

c [2, 6] with i = 2, j = 6, d2 = 3

Here $i \neq 1$ and $j > d2$

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j-di])$$
$$= \min(c[2-1, 6], 1 + c[2, 6-3])$$
$$= \min(c[1, 6], 1 + c[2, 3])$$
$$= \min(6, 1 + 1)$$

c [2, 6] = 2

c [3, 6] with i = 3, j = 6, d3 = 5

∴

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j-di])$$
$$= \min(c[3-1, 6], 1 + c[3, 6-5])$$
$$= \min(c[2, 6], 1 + c[3, 1])$$
$$= \min(2, 1 + 1)$$

c [3, 6] = 2

c [4, 6] with i = 4, j = 6, d4 = 6

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j-di])$$
$$= \min(c[3, 6], 1 + c[4, 0])$$
$$= \min(2, 1 + 0)$$

c [4, 6] = 1

Continuing in this fashion, we get -

	0	1	2	3	4	5	6	7	8
d1 = 1	0	1	2	3	4	5	6	7	8
d2 = 3	0	1	2	1	2	3	2	3	4
d3 = 5	0	1	2	1	2	1	2	3	2
d4 = 6	0	1	2	1	2	1	1	2	(2)

↑

Total number of coins

The value c [4, 8] = 2 represents total number of coins to get sum of ₹. 8. The coins are,

d2 i.e. 3 units = 1 coin
+ d3 i.e. 5 units = 1 coin

8 units (₹) = 2 coins

Ex. 5.6.5 *Solve the following making change problem using dynamic programming method : Amount ₹ 7 and Denominations : (₹ 1, ₹ 2 and ₹ 4).*

Sol. : d1 = 1, d2 = 2 and d3 = 4 and N = 7

We will create a table with rows ranging from 1 to 3 and column ranging from 0 to 7.

Initially C[i,0]=0

C[1,0] = C[2,0] = C[3,0] = 0

		0	1	2	3	4	5	6	7
i = 1	d1 = 1	0							
i = 2	d2 = 2	0							
i = 3	d3 = 4	0							

Computations can be performed column by column and fill up the table accordingly.

We will use following formula

1. If i = 1 then C[i, j] = 1 + C[1, j – d1]
2. If j < di then C[i, j] = 1 + C[i – 1, j]
3. Otherwise C[i, j] = min(C[i –1, j], 1 + C[i, j – d1])

Step 1	C[1,1] with i = 1, j = 1,d1 = 1 As i = 1, **Formula Used :** 1 + C[1,j – d1] = 1 + C[1,1 – 1] = 1 + C[1,0] = 1 + 0 = 1 **C[1,1] = 1**	C[2,1] with i = 2, j = 1, d2 = 2 As j < d2 **Formula Used :** C[i – 1,j] = C[1,1] = 1 **C[2,1] = 1**	C[3,1] with i = 3, j = 1, d3 = 4 As j < d3 **Formula Used :** C[i – 1,j] = C[3 – 1,1] = C[2,1] **C[3,1] = 1**
Step 2	C[1,2] with i = 1, j = 2, d1 = 1 As i = 1 **Formula Used:** 1+C[1,j – d1] = 1+C[1,1] = 1 + 1 **C[1,2] = 2**	C[2,2] with i = 2,j = 2, d2 = 2 As j = d2 and i 1 **Formula Used:** min(C[i 1,j],1 + C[i,j di]) = min(C[1,2] , 1 + C[2,0] = min (2, 1 + 0) **C[2,2] = 1**	C[3,2] with i = 3, j = 2, d 3 = 4 As j < d3 **Formula Used :** C[i – 1,j] = C[3 – 1,2] = C[2,2] **C[3,2] = 1**

TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

Step 3	C[1,3] with i = 1, j = 3,d1 = 1 As i = 1 **Formula Used :** 1+C[1,j – d1] = 1+C[1,2] = 1 + 2 **C[1,3] = 3**	C[2,3] with i = 2, j = 3, d2 = 2 As j > d3 **Formula Used :** = min(C[i – 1,j], 1+C[i,j di]) = min(C[1,3], 1+C[2,1]) = min (3, 2) **C[2,3] = 2**	C[3,3] with i = 3, j = 3, d 3 = 4 As j<d3 **Formula Used :** C[i – 1,j] = C[3 – 1,3] = C[2,3] **C[3,3] = 2**
Step 4	C[1,4] with i = 1, j = 4, d1 = 1 As i = 1 **Formula Used:** 1+C[1,j – d1] = 1+C[1,4 – 1] = 1 + C[1,3] **C[1,4] = 4**	C[2,4] with i = 2, j = 4, d2 = 2 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[1,4], 1+C[2,2]) = min (4, 1+1) **C[2,4] = 2**	C[3,4] with i = 3, j = 4,d 3 = 4 As j = d3 and i **Formula Used :** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[2,4], 1+C[3,0] = min (2,1+0) **C[3,4] = 1**
Step 5	C[1,5] with i = 1, j = 5, d1 = 1 As i = 1 Formula Used : 1+C[1,j – d1] = 1+C[1,5 – 1] = 1+C[1,4] **C[1,5] = 5**	C[2,5] with i = 2, j = 5, d2 = 2 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i,j-di]) = min(C[1,5], 1+C[2,3]) = min (5,1+2) **C[2,5] = 3**	C[3,5] with i = 3, j = 5, d3 = 4 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i, j – di]) = min(C[2,5], 1+C[3,1]) = min (3, 1+1) **C[3,5] = 2**
Step 6	C[1,6] with i = 1, j = 6,d1 = 1 As i = 1 **Formula Used :** 1+C[1,j – d1] = 1+C[1,6 – 1] = 1+C[1,5] **C[1,6] = 6**	C[2,6] with i = 2, j = 6,d2 = 2 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[1,6], 1+C[2,4]) = min (6, 1+2) **C[2,6] = 3**	C[3,6] with i = 3, j = 6, d3 = 4 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[2,6], 1+C[3,2]) = min (3, 1+2) **C[3,6] = 3**
Step 7	C[1,7] with i = 1, j = 7, d1 = 1 As i = 1 **Formula Used :** 1+C[1,j – d1] = 1+C[1,7 – 1] = 1+C[1,6] **C[1,7] = 7**	C[2,7] with i = 2, j = 7,d2 = 2 As j > d2 **Formula Used:** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[1,7], 1+C[2,5]) = min (7, 1+3) **C[2,7] = 4**	C[3,7] with i = 3, j = 7,d3 = 4 As j > d2 **Formula Used :** = min(C[i – 1,j], 1+C[i,j – di]) = min(C[2,7], 1+C[3,3]) = min (4, 1+2) **C[3,7] = 3**

Thus we can represent the table filled up with above computations will be -

		0	1	2	3	4	5	6	7
i = 1	d1 = 1	0	1	2	3	4	5	6	7
i = 2	d2 = 2	0	1	1	2	2	3	3	4
i = 3	d3 = 4	0	1	2	2	1	2	3	3

↑

Total number of coins

The value C[n,N] i.e C[3,7] = 3 which represents total number of coins required to get the sum 7 are 3.

Thus we get

₹ 1 -> 1 coin

₹ 2-> 1 coin

₹ 3 -> 1 coin

Sum=7

```
Algorithm Number_of_Coins (N)
{
// Problem Description : This algorithm computes
//minimum number of coins required to make
// change for N units.
    for ( i ← 1 to n) do
    initialize array d[i] with the values of coins.
for ( i ← 1 to n ) do
        c [i, 0] ← 0          Initialising first column by 0
For ( i ← 1 to n ) do
{
    For ( i 1 to N ) do       Initialising first column by 0
    {
        if (i = 1 & & j < d [ i] ) then
            c [ i ] [ j ] = infinity ;
        else if ( i = 1 ) then
            c [ i ] [ j ] = 1+ c [ 1, j – d [1] ];
        else if ( j <d [ i ] ) then
```



```
            c [ i ] [ j ] = c [j – 1, j];
        else
            c [ i ] [ j ] = min ( c [i – 1, j], 1+ c [i, j – d [i]);
        }
}
return c [ n, N]          Total number of coins in the change
```

Analysis - As the basic operation in above algorithms is to fill up the table, which is performed within nested for loops. Hence the time complexity of this algorithm is **Θ (nN).**

Review Questions

Q.1 Explain coin change problem with suitable example.

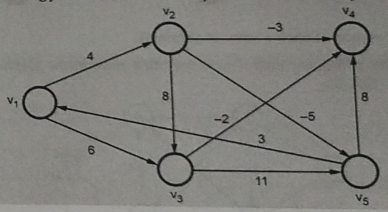
Q.2 Give an algorithm for coin change problem. Analyze it.

5.7 Bellman-Ford Algorithm

SPPU : May-19, April-18, Marks 10

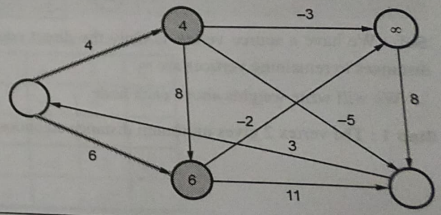
- The Bellman-Ford algorithm works for finding single source shortest path. It works for **negative** edge weights.
- This is a single source shortest path algorithm.
- Let us understand this algorithm with the help of some example.

Ex. 5.7.1 *Consider following for which the shortest path can be obtained from source vertex v_1.*

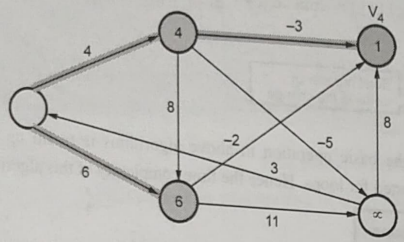


Sol. :

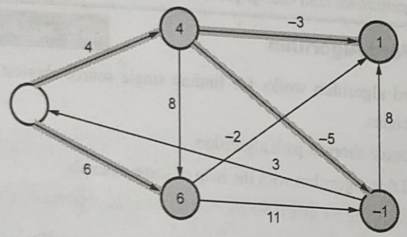
Step 1 : From vertex v_1 we find the distances to v_2 and v_3. The direct edges corresponds to the weights of destination vertices.



Step 2 : Modify vertex v_4 by its minimum weight i.e. 1. The path is shown by shaded area.

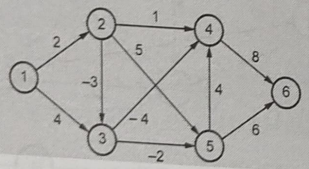


Step 3 : Process vertex v_5 by its minimum weight –1. The path is shown by shaded area.



Thus minimum distance of each vertex is obtained.

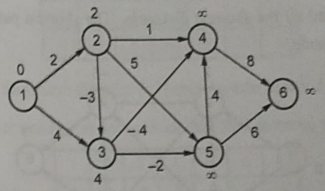
Ex. 5.7.2 *Find shortest path from node 1 to every other node in the graph as given below using Bellman and Ford Algorithm.*



Sol. : We have a source vertex 1. Only the direct edges to other v ertices are consider. The distances to remaining vertices are ∞.

We will write weights above each node.

Step 1 : The vertex 2 gives minimum distance so choose **vertex 2**



It can be represented as

1	2	3	4	5	6
0	2	4	∞	∞	∞

Step 2 : Now we will consider 2 as parent node and find distance to all other vertices

1	2	3	4	5	6
0	2	4	∞	∞	∞
0	2	–1	3	7	∞

For vertex 3
Dist (1, 2)
+ dist (2, 3)
= 2 + (– 3) = – 1

For vertex 4
dist (1, 2) + dist (3 4)
= 2 + 1 = 3

For vertex 5
dist (1, 2) + dist (2, 5)
= 2 + 5 = 7

Step 3 : Now consider 3 as parent node

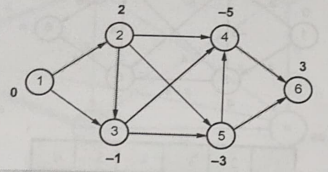
1	2	3	4	5	6
0	2	4	∞	∞	∞
0	2	–1	3	7	∞
0	2	–1	**– 5**	**–3**	∞

The distance to vertex 4 and vertex 5 get updated, by considering node 3 as parent node

Step 4 : Consider node 4 as parent node shortest distance are then

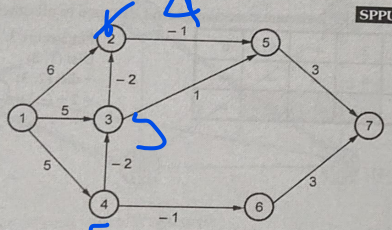
1	2	3	4	5	6
0	2	4	∞	∞	∞
0	2	–1	3	7	∞
0	2	–1	–5	–3	∞
0	2	–1	–5	–3	3

Thus we have obtained all the shortest distances. The shortest path distances are shown in boldface above each node.



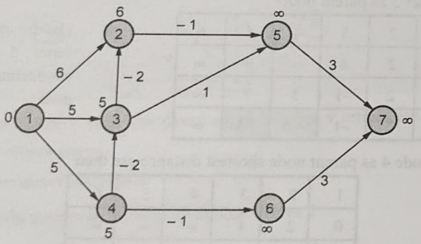
Ex. 5.7.3 Use Bellman Ford algorithm to find shortest path for the following graph.

SPPU : May-19, Marks 10



Sol. : We have source vertex 1. Only direct edges of vertices are considered as distances. The distance to remaining vertices are ∞. We will write weights above each node.

Step 1 :



It can be represented as

1	2	3	4	5	6	7
0	6	5	5	∞	∞	∞

Now vertex 3 gives minimum distance 5. Hence choose vertex 3.

Step 2 : We will consider vertex 3 as parent node and find distance to all other vertices.

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

Step 3 : Here vertex 2 gives minimum distance

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

Step 4 : Now vertex 4 gives minimum distance. Hence choose vertex 4.

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

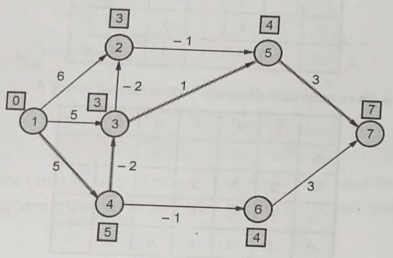
Step 5 : Here vertex 5 gives minimum distance. Hence choose vertex 5.

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

Step 6 : Here vertex 6 gives minimum distance. Hence choose vertex 6.

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

Thus we have obtained all the shortest distances. The shortest path distances are shown in boldface above each node.



Algorithm

```
Algorithm Bellman Ford (vertices, edges, source)
{
// Problem Description : This algorithm finds
// the shortest path using Bellman Ford method
for (each vertex v)
{
    if (v is source) then
    v distance ← 0
    else
    v. distance ← infinity
    v. prede ← Null
```

Graphical initialization


```
}
for (i ← 1 to toal_ vertices – 1)
{
    for (each edge uv)
    {
    U ← uv. source
    V ← uv. desination
    if (v. distance > u. distance + uv. weight ) then
    {
        v. distance ← u. distance + uv. weight
        v. prede ← u
    }
}
for (each edge uv)
{
    u ← uv. source
    v ← uv. destination
    if (v. distance > u.distance + uv. weight) then
    {
        Write ("Graph has negative edges")
        return False
    }
}
} // end of for return True
} // end of algorithm
```

Newly obtained minimum distance

Relaxing edges

In above algorithm, we have used a term "**relaxing edges**". The process of relaxing edge (u, v) means testing whether we can get more shorter distance to vertex v from u. The **relaxation** step **decrease** the value of the shortest path estimated earlier and modify the predecessors of vertex v. The running time of Bellman-Ford algorithm is **O(nm)**.

Review Question

Q.1 Write Bellman ford algorithm to find shortest path and analyze it.

SPPU : April-18, In Sem, Marks 5

5.8 Multistage Graph Problem

SPPU : May-18, April-19, Marks 10

A multistage graph G = (V, E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k ≥ 2. In multistage graph problem we have to find the

TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

shortest path from source to sink. The cost of each path is calculated by using the weight given along that edge. The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path. In multistage graph problem we have to find the path from S to T. There is set of vertices in each stage. The multistage graph can be solved using forward and backward approach.

Let us solve multistage problem for both the approaches with the help of some example.

Consider the graph G as shown in the Fig. 5.8.1.

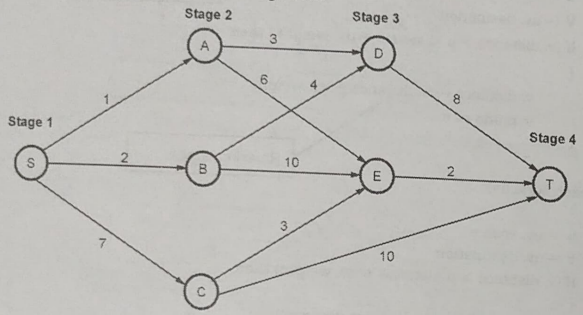


Fig. 5.8.1 Multistage graph

There is single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

$$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\} \quad \ldots (1)$$

We will now compute d(A, T), d(B, T) and d(C,T).

$$d(A, T) = \min \{3 + d(D, T), 6 + d(E, T)\} \quad \ldots (2)$$

$$d(B, T) = \min \{4 + d(D, T), 10 + d(E, T)\} \quad \ldots (3)$$

$$d(C, T) = \min \{3 + d(E, T), d(C, T)\} \quad \ldots (4)$$

Now let us compute d(D, T) and d(E, T).

$$\mathbf{d(D, T) = 8}$$

$$\mathbf{d(E, T) = 2} \qquad \textbf{backward vertex = E}$$

Let us put these values in equations (2), (3) and (4).

$$d(A, T) = \min \{3 + 8, 6 + 2\}$$

TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

$$\mathbf{d(A, T) = 8} \qquad \textbf{A - E - T}$$

$$d(B, T) = \min \{4 + 8, 10 + 2\}$$

$$\mathbf{d(B, T) = 12} \qquad \textbf{A - D - T}$$

$$d(C, T) = \min \{3 + 2, 10\}$$

$$\mathbf{d(C, T) = 5} \qquad \textbf{C - E - T}$$

$$\therefore \quad d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\}$$

$$= \min \{1 + 8, 2 + 12, 7 + 5\}$$

$$= \min \{9, 14, 12\}$$

$$d(S, T) = 9 \qquad \textbf{S - A - E - T}$$

Forward approach

$$d(S, A) = 1$$

$$d(S, B) = 2$$

$$d(S, C) = 7$$

$$d(S, D) = \min \{1 + d(A, D), 2 + d(B, D)\}$$

$$= \min \{1 + 3, 2 + 4\}$$

$$\mathbf{d(S, D) = 4}$$

$$d(S, E) = \min \{1 + d(A, E), 2 + d(B, E), 7 + d(C, E)\}$$

$$= \min \{1 + 6, 2 + 10, 7 + 3\}$$

$$= \min \{7, 12, 10\}$$

$$\mathbf{d(S, E) = 7} \qquad \textbf{i.e. Path S - A - E is chosen.}$$

$$d(S, T) = \min \{d(S, D) + d(D, T), d(S, E) + d(E, T), d(S, C) + d(C, T)\}$$

$$= \min \{4 + 8, 7 + 2, 7 + 10\}$$

$$d(S, T) = 9 \qquad \textbf{i.e. Path S - E, E - T is chosen.}$$

$\therefore$ The minimum cost = 9 with the path **S - A - E - T.**

This method is called **forward reasoning**.

The algorithm for forward approach is –

```
Algorithm Forward_Gr (G, stages, n, p)
//  Problem description : This algorithm is for
//  forward  approach of multistage graph
//  Input : The multistage graph G = (V, E),
//  'Stages' is the variable representing number of stages
//  n is total number of vertices of G
//  p is an array for restoring path
```

```
//      Output : The path with minimum cost
                cost[i] ← 0
        for (i ← n – 2 downto 0)
        {
                r ← Get_min (i,n) //      r is an edge with min cost
                cost[i] ← c[i][r] + cost[r]
                        d[i] ← r
        }
        //      finding minimum cost path
                p[0] ← 0
                p[stages – 1] ← n – 1
        for (i ← 1 to stages–1) do
                p[i] ← d[p[i+1]] ;
```

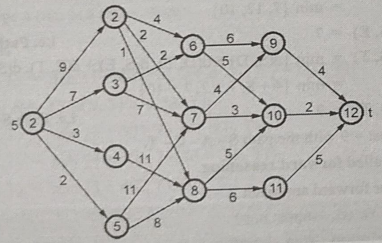
Analysis : The algorithm has Θ (|V|+|E|) time complexity.

Using dynamic programming strategy, the multistage graph problem is solved. This is because in multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage. Thus the sequence of decisions are taken by considering overlapped solution.

In dynamic programming, we may get any number of solutions for given problem. From all these solutions we seek for optimal solution, finally optimal solution becomes the solution to given problem. Multistage graph problem is solved using this same approach.

Ex. 5.8.1 Find the minimum cost path from source (s) to sink (t) of the following multistage graph.

SPPU : May-18, End Sem Marks 10



Sol. : Step 1 :

$d(1,2) = 9$

$d(1,3) = 7$

$d(1,4) = 3$

$d(1,5) = 2$

Step 2 :

$d(1,6) = \min\{(9 + d(2,6)), (7 + d(3, 6))\}$

$= \min\{(9 + 4), (7+2)\}$

$\boxed{d(1,6) = 9}$ Here we choose **vertex 3** as minimum distance source.

$d(1,7) = \min\{(9 + d(2,7)), (7 + d(3,7)), (2 + d(5,7))\}$

$= \min\{(9 + 2), (7 + 7), (2 + 11)\}$

$\boxed{d(1,7) = 11}$ Here we choose **vertex 2** as minimum distance source.

$d(1,8) = \min\{(9 + d(2,8)), (3 + d(4,8)), (2 + d(5,8))\}$

$= \min\{(9 + 1), (3 + 11), (2 + 8)\}$

$\boxed{d(1,8) = 10}$

Here we choose either **vertex 2** or **vertex 5** as minimum distance source.

Step 3 :

$d(1,9) = \min\{((1,6) + d(6,9)), ((1,7) + d(7,9))\}$

$= \min\{(9 + 6), (11 + 4)\}$

$\boxed{d(1,9) = 15}$

Here we choose **vertex 6** or **vertex 7** as minimum distance source.

$d(1,10) = \min\{((d(1,6) + d(6, 10)), (d(1,7) + d(7,10)), (d(1,8) + d(8,10))\}$

$= \min\{(9 + 5), (11 + 3), (10 + 5)\}$

$\boxed{d(1,10) = 14}$

Here we choose **vertex 6** or **vertex 7** as minimum distance source.

$d(1,11) = (d(1,8) + d(8,11))$

$= (10 + 6)$

$\boxed{d(1,11) = 16}$

Step 4 :

$d(1,12) = \min\{(d(1,9) + d(9,12)), (d(1,10) + d(10,12)),$

$(d(1,11) + d(11,12))\}$

$= \min\{(15 + 4), (14 + 2), (16 + 5)\}$

$\boxed{d(1,12) = 16}$

Here we choose **vertex 10** as minimum distance vertex.

Step 5 : From step 1 to step 4 we can obtain minimum distance as -

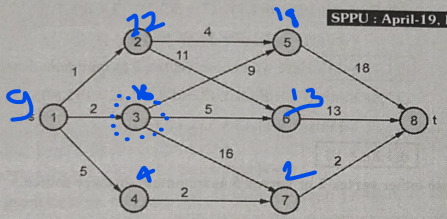
$1 - 3 - 6 - 10 - 12$

$1 - 2 - 7 - 10 - 12$

The **minimum cost = 16**

Ex. 5.8.2 Find minimum cost path from source (s) to sink (t) of the following multistage graph.

SPPU : April-19, In Sem, Marks 5



Solution :

At stage 1, we will consider vertex 1 as source 's'

At stage 2, we will consider vertex 2, 3 and 4 as source.

At stage 3, we will consider vertex 5, 6 and 7 as source.

At stage 4, we will consider vertex 8 as target 't'

Now, using forward approach we will find **d(s,t)**

Step 1 : We will first obtain the direct distances from source 's'.

$\therefore$ $d(1,2) = 1$

$d(1,3) = 2$

$d(1,4) = 5$

Step 2 :

$d(1,5) = \min\{(1 + d(2,5)), (2 + d(3,5))\}$

$= \min\{(1 + 4), (2 + 9)\}$

$\boxed{d(1,5) = 5}$ Here **vertex 2** gives us minimum diatance.

$d(1,6) = \min\{(1 + d(2,6)), (2 + d(3,6))\}$

$= \min\{(1 + 11), (2 + 5)\}$

$= \min\{12,7\}$

TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

$\boxed{d(1,6) = 7}$ Here **vertex 3** gives us minimum diatance.

$d(1,7) = \min\{(2 + d(3,7)), (5 + d(4,7))\}$

$= \min\{(2 + 16), (5 + 2)\}$

$\boxed{d(1,7) = 7}$ Here **vertex 4** gives us minimum diatance.

Step 3 :

$d(1,8) = \min\{(5 + d(5,8)), (7 + d(6,8)), (7 + d(7,8))\}$

$= \min\{(5 + 18), (7 + 13), (7 + 2)\}$

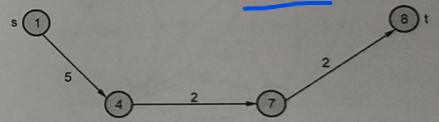
$\boxed{d(1,8) = 9}$ Here **vertex 7** gives us minimum diatance.

Step 4 :

To reach to target node 8 (i.e. t) we choose **vertex 7** which gives us minimum distance.

From step 2, we observe the d(1,7). For minimum distance of this path, we have chosen vertex 4.

Hence we can conclude, the minimum path as $1 - 4 - 7 - 8$ i.e. $s - 4 - 7 - t$



Minimum cost = 9

5.9 Travelling Salesman Problem

SPPU : May-08,12,13,15, 18, Dec.-11, 18, Marks 10

- **Problem Description**

  Let G be directed graph denoted by (V, E) where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij}. The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.

- A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour.
- The traveling salesperson problem is to find the tour of minimum cost.
- Dynamic programming is used to solve this problem.

Step 1 : Let the function C(1, V–{1}) is the total length of the tour terminating at 1. The objective of TSP problem is that the cost of this tour should be minimum.
Let d[i, j] be the shortest path between two vertices i and j.

Step 2 : Let V_1, $V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour. Then $(V_1, V_2, \ldots V_n)$ must be a shortest path from V_1 to V_n which passes through each vertex exactly once.

Here the principle of optimality is used. The path V_i, $V_{i+1}, \ldots V_j$ must be optimal for all paths beginning at V(i), ending at V(j), and passing through all the intermediate vertices $\{V_{(i+1)} \ldots V_{(j-1)}\}$ once.

Step 3 : Following formula can be used to obtain the optimum cost tour.

$Cost(i, S) = \min \{d[i, j] + Cost(j, S-\{j\})\}$ where $j \in S$ and $i \notin S$.

Consider one example to understand solving of TSP using dynamic programming approach.

Ex. 5.9.1 *For the given diagraph, obtain optimum cost tour.*

SPPU : May-08, Marks 10, May-13, May-18 End. Sem., Dec.-18, End Sem., Marks 8

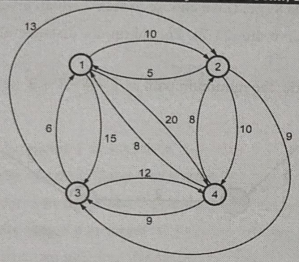


Fig. 5.9.1

Sol. : The distance matrix can be given by,

to → / from ↓	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

First we will select any arbitrary vertex say select 1.

Now process for intermediate sets with increasing size.

Step 1 : Let $S = \phi$ then,

$$Cost(2, \phi, 1) = d(2, 1) = 5$$
$$Cost(3, \phi, 1) = d(3, 1) = 6$$

$$Cost(4, \phi, 1) = d(4, 1) = 8$$

That means we have obtained dist (2, 1), dist (3, 1) and dist (4, 1).

Step 2 :

$$Candidate (S) = 1$$

Apply the formula,

$$Cost(i, S) = \min \{d[i, j] + Cost(j, S - \{j\})\}$$

Hence from vertex 2 to 1, vertex 3 to 1 and vertex 4 to 1 by considering intermediate path lengths we will calculate total optimum cost.

$$Cost(2, \{3\}, 1) = d(2, 3) + Cost(3, \phi, 1)$$
$$= 9 + 6$$
$$Cost(2, \{3\}, 1) = 15$$
$$Cost(2, \{4\}, 1) = d(2, 4) + Cost(4, \phi, 1)$$
$$= 10 + 8$$
$$Cost(2, \{4\}, 1) = 18$$
$$Cost(3, \{2\}, 1) = d(3, 2) + Cost(2, \phi, 1)$$
$$= 13 + 5$$
$$Cost(3, \{2\}, 1) = 18$$
$$Cost(3, \{4\}, 1) = d(3, 4) + Cost(4, \phi, 1)$$
$$= 12 + 8$$
$$Cost(3, \{4\}, 1) = 20$$
$$Cost(4, \{2\}, 1) = d(4, 2) + Cost(2, \phi, 1)$$
$$= 8 + 5$$
$$Cost(4, \{2\}, 1) = 13$$
$$Cost(4, \{3\}, 1) = d(4, 3) + Cost(3, \phi, 1)$$
$$= 9 + 6$$
$$Cost(4, \{3\}, 1) = 15$$

Step 3 : Consider candidate (S) = 2

$$Cost(2, \{3, 4\}, 1) = \min \{[d(2, 3) + Cost(3, \{4\}, 1)], [d(2, 4) + Cost(4, \{3\}, 1)]\}$$
$$= \min \{[9 + 20], [10+15]\}$$
$$Cost(2, \{3, 4\}, 1) = 25$$
$$Cost(3, \{2, 4\}, 1) = \min \{[d(3, 2) + Cost(2, \{4\}, 1)], [d(3, 4) + Cost(4, \{2\}, 1)]\}$$
$$= \min \{[13+18], [12+13]\}$$

$$\text{Cost } (3, \{2, 4\}, 1) = 25$$

$$\text{Cost } (4, \{2, 3\}, 1) = \min \{[d(4, 2) + \text{Cost } (2, \{3\}, 1)], [d(4, 3) + \text{Cost } (3, \{2\}, 1)]\}$$
$$= \min \{[8 + 15], [9 + 18]\}$$
$$\text{Cost } (4, \{2, 3\}, 1) = 23$$

Step 4 : Consider candidate (S) = 3. i.e. Cost (1, {2, 3, 4}) but as we have chosen vertex 1 initially the cycle should be completed i.e. starting and ending vertex should be 1.

∴ We will compute,

$$\text{Cost } (1, \{2, 3, 4\}, 1) = \left\{ \begin{array}{l} [d (1, 2) + \text{Cost } (2, \{3, 4\}, 1)], [d (1, 3) + \text{Cost } (3, \{2, 4\}, 1] \\ [d(1, 4) + \text{Cost } (4, \{2, 3\}, 1)] \end{array} \right\}$$
$$= \min ([10 + 25], [15 + 25], [20 + 23])$$
$$= 35$$

Thus the optimal tour is of path length 35.

Consider step 4 now, from vertex 1 we obtain the optimum path as d(1, 2). Hence select vertex 2. Now consider step 3, in which from vertex 2 we obtain optimum cost from d(2, 4). Hence select vertex 4. Now in step 2 we get remaining vertex 3 as d(4, 3) is optimum. Hence optimal tour is 1, 2, 4, 3, 1.

Ex. 5.9.2 *For a directed graph the edge length matrix is given below. Solve the travelling salesperson problem, for this 4 city graph using dynamic programming method. What will be the time complexity for n city TSP problem solved using this method.*

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

SPPU : May-12, Marks 7

Sol. : We will start with vertex 1.

We will now process for intermediate sets with increasing size.

Step 1 :

Let $S = \phi$ then

$$\text{cost } (2, \phi, 1) = d (2, 1) = 12$$

$$\text{cost } (3, \phi, 1) = d (3, 1) = 10$$

$$\text{cost } (4, \phi, 1) = d (4, 1) = 20$$

That means we have obtained distance (2, 1), (3, 1) and distance (4, 1).

Step 2 :

Candidate (S) = 1

Now we will apply formula,

$$\text{cost} = \min$$

Hence from vertex 2 to 1, vertex 3 to 1 and vertex 4 to 1 by considering intermediate path lengths we will calculate total optimal cost.

$$\text{cost } (2, \{3\}, 1) = d(2, 3) + \text{cost } (3, \phi, 1)$$
$$= 13 + 10$$

cost (2, {3}, 1) = 23

$$\text{cost} - (2, \{4\}, 1) = d (2, 4) + \text{cost } (4, \phi, 1)$$
$$= 6 + 20$$

cost (2, {4}, 1) = 26

$$\text{cost } (3, \{2\}, 1) = d (3, 2) + \text{cost } (2, \phi, 1)$$
$$= 9 + 12$$

cost (3, {2}, 1) = 21

$$\text{cost } (3\{4\}, 1) = d(3, 4) + \text{cost } (4, \phi, 1)$$
$$= 5 + 20$$

Cost (3,{4}, 1) = 25

$$\text{cost } (4, \{2\}, 1) = d (4, 2) + \text{cost } (2, \phi, 1)$$
$$= 15 + 12$$

cost (4, {2}, 1) = 27

$$\text{cost } (4, \{3\}, 1) = d (4, 3) + \text{cost } (3, \phi, 1)$$
$$= 10 + 10$$

Cost (4, {3}, 1) = 20

Step 3 : Consider candidate (S) = 2

$$\text{cost } (2, \{3, 4\}, 1) = \min \{d [2, 3] + \text{cost } (3, \{4\}, 1), d[2,4] + \text{cost } (4, \{3\}, 1)\}$$
$$= \min \{13 + 25, 6 + 20\}$$
$$= 26$$

$$\text{Cost } (3, \{2, 4\}, 1) = \min \{d [3,2] + \text{cost } (2, \{4\}, 1), d[3,4] + \text{cost } (4, \{2\}, 1)\}$$
$$= \min \{9 + 26, 5 + 27\}$$

$$\text{Cost } (3, \{2, 4\}, 1) = 32$$

$$\text{Cost } (4, \{2, 3\}, 1) = \min \{d [4, 2] + \text{cost } (2, \{3\}, 1), d [4, 3] + \text{cost } (3, \{2\}, 1)\}$$
$$= \min \{15 + 23, 10 + 21\}$$

$$\text{Cost } (4, \{2, 3\}, 1) = 31$$

Step 4 : Consider candidate (S) = 3. Hence the cost will be cost (1, {2, 3, 4}). But we have chosen vertex 1 initially the cycle should be completed.

∴ we will compute

$$\text{cost}(1, \{2, 3, 4\}, 1) = \min \begin{Bmatrix} d[1, 2] + \text{cost}(2, \{3, 4\}, 1), \\ d[1, 3] + \text{cost}(3, \{2, 4\}, 1), \\ d[1, 4] + \text{cost}(4, \{2, 3\}, 1) \end{Bmatrix} = \min(9 + 26,\ 8 + 32,\ 8 + 31)$$

$$= \min(9 + 26,\ 8 + 32,\ 8 + 31)$$

$$= \min(35, 40, 39) = 35$$

Thus the optimal tour is of cost 35.

Consider step 4 now, from vertex 1 we obtain the optimum path as d (1, 2). Hence select vertex **2.**

Now consider step 3, in which from vertex 2 we obtain optimum cost from d [2, 4] hence select vertex **4.**

In step 2, we get remaining vertex as 3 as d(4, 3) is optimum. Hence optimal tour sequence is **1-2-4-3-1.**

The total time required by the algorithm to execute is O ($nm - m^2$) which turns out to be O (n^3)

Review Questions

Q.1 Explain how dynamic programming is used to obtain optimal solution for travelling salesperson problem. Also explain why this technique is not used to solve TSP for large number of cities. **SPPU : Dec.-11, Marks 6**

Q.2 What is travelling salesperson problem ? How it can be solved by dynamic programming strategy ? **SPPU : May-15 In Sem, Marks 5**

5.10 Multiple Choice Questions

Q. 1 Which method can be used when the solution to a problem can not be viewed as the result of a sequence of decisions?

a Greedy method. b Divide and conquer.

c Dynamic Programming. d Backtracking.

Q. 2 The _____ states that whenever the initial state and decision are given, the remaining decisions must constitute an optimal decision sequence.

a principle of locality b principle of optimality

c principle of backtracking d none of the above

Q. 3 The principle of optimality is used by _____.

a brute force b divide and conquer

c dynamic programming d backtracking

Q. 4 The goal is obtained at _____ in 0/1 Knapsack using dynamic programming

a table[0,0] b table[0,n]

c table[n,w] d table[n,n]

Q. 5 The solution for the given of Knapsack is _____.

M=6 n=3

(p1, p2, p3)=(1,2,5)

(w1, w2, w3)=(2,3,4)

a 011 b 101 c 001 d 010

Answer Keys for Multiple Choice Questions

Q.1	c	Q.2	b	Q.3	c	Q.4	c	Q.5	b

Notes

UNIT - IV

6 Backtracking

Syllabus

General method, *Recursive backtracking algorithm, Iterative backtracking method. n-Queen problem, Sum of subsets, Graph coloring, 0/1 Knapsack Problem.*

Contents

6.1	General Method	..... **Dec.-06,10,11,12,13, May-08,11,15, 18,** ..... Marks 18
6.2	Recursive Backtracking and Iterative Backtracking Algorithm	..... **Dec.-06,07, 18, May-07,15, 18,**..... Marks 8
6.3	Applications of Backtracking	
6.4	The n-Queen Problem	..... **May-11,12,14, 19,** ..... Marks 10
6.5	Sum of Subsets	..... **May-10,11,15,17, 18, 19, Dec.-08,09,11,12,15, 18,**..... Marks 12
6.6	Graph Coloring	..... **May-08,10,11,17, Dec.-12,13,** .... Marks 12
6.7	The 0/1 Knapsack Problem	..... **Dec.-11, May-12,14,15, 18,** ..... Marks 16
6.8	Multiple Choice Questions	

implicit → rule in which how each ele in tuple is related
exp → rule that restricts each ele to be chosen from given set

6.1 General Method

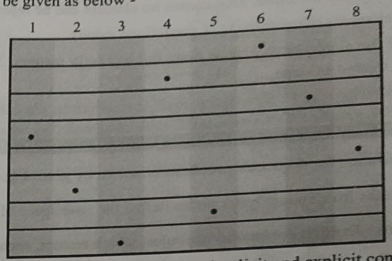
SPPU : Dec.-06,10,11,12,13, May-08,11,15, 18, Marks 18

- In the backtracking method
  1. The desired solution is expressible as an n tuple $(x_1, x_2, \ldots x_n)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function $C(x_1, x_2, \ldots x_n)$.
- The problem can be categorized into three categories. For instance - For a problem P let C be the set of constraints for P. Let D be the set containing all solutions satisfying C then Finding whether there is any feasible solution ?- Is the decision problem.
  What is the best solution ? - Is the optimization problem.
  Listing of all the feasible solution - Is the enumeration problem.
- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major **advantage** of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.
- Backtracking is a **depth first search** with some **bounding function.**
- Backtracking algorithm solves the problem using **two types of constraints -**
  Explicit constraint and **Implicit constraints**
- **Definition : Explicit constraints** are the rules that restrict each element x_i has to be chosen from given set only. Explicit constraints depends on particular instance I of the problem. All the tuples from solution set must satisfy the explicit constraints.
- **Definition : Implicit constraints** are the rules that decide which tuples in the solution space of I satisfy the criterion function. Thus the implicit constraints represent by which x_i in the solution set must be related with each other.

For example

Example 1 : 8 - Queen's problem - The 8-queen's problem can be stated as follows. Consider a chessboard of order 8×8. The problem is to place 8 queens on this board such that no two queens can attack each other. That means no two queens can be placed on the same row, column or diagonal. The solution to 8-queens problem can be obtained using backtracking method.

- The solution can be given as below -



This 8 Queen's problem is solved by applying implicit and explicit constraints.

The **explicit constraints** show that the solution space S_i must be = {1, 2, 3, 4, 5, 6, 7, 8} with $1 \le i \le 8$. Hence solution space consists of 8^8 8-tuples.

The **implicit constraint** will be -

1) No two x_i will be same. That means all the queens must lie in different columns.
2) No two queens can be on the same row, column or diagonal.

Hence the above solution can be represented as an 8-tuple {4, 6, 8, 2, 7, 1, 3, 5}.

Example 2 : Sum of subsets -

There are n positive numbers given in a set. The desire is to find all possible subsets of this set, the contents of which add onto a predefined value M.

In other words,

Let there be n elements given by the set $w = (w_1, w_2, w_3, \ldots, w_n)$ then find out all the subsets from whose sum is M.

For example

Consider n = 6 and $(w_1, w_2, w_3, w_4, w_5, w_6) = (25, 8, 16, 32, 26, 52)$ and M = 59 then we will get desired sum of subset as (25, 8, 26).

We can also represent the sum of subset as (1, 1, 0, 0, 1, 0). If solution subset is represented by an n-tuple $(x_1, x_2, \ldots x_n)$ such that x_i could be either 0 or 1. The $x_i = 1$ means the weight w_i is to be chosen and $x_i = 0$ means that weight w_i is not to be chosen.

The **explicit constraint** on sum of subset problem will be that any element $x_i \in \{i | i$ is an integer and $1 \le i \le x\}$. That means we must select the integer from given set of elements.

The **implicit constraints** on sum of subset problem will be -

1) No two elements will be the same. That means no element can be repeatedly selected.
2) The sum of the elements of subset = M (a predefined value).
3) The selection of the elements should be done in an orderly manner. That means (1, 3, 7) and (1, 7, 3) are treated as same.

Ex. 6.1.1 *Define following terms : State space, explicit constraints, implicit constraints, problem state, solution states, answer states, live node, E-node, dead node, bounding functions.*
SPPU : May-11, Marks 8, Dec.-11, Marks 16, Dec-12, Marks 18, May 18, End Sem., Marks 8

Sol. : The tree organization for 4-queen's problems is shown by given Fig. 6.1.1(on next page).

State space : All paths from root to other nodes define the state space of the problem.

Explicit constraints : Explicit constraints are rules, which restrict, each vector element to be chosen from given set.

Implicit constraints : Implicit constraints are rules which determine which of the tuples in the solution space satisfy the criterion function.

Problem states : Each node in the state space tree is called problem state.

Solution states : The solution states are those problem states s for which the path from root to s defines a tuple in the solution space. In some trees the leaves define solution states.

Answer states : These are the leaf nodes which correspond to an element in the set of solutions. These are the states which satisfy the implicit constraints.

Live node : A node which is generated and whose children have not yet been generated is called live node.

E-node : The live node whose children are currently being expanded is called E-node.

Dead node : A dead node is a generated node which is not to be expanded further or all of whose children have been generated.

Bounding functions : Bounding functions will be used to kill live nodes without generating all their children. A care should be taken while doing so, because atleast one answer node should be produced or even all the answer nodes be generated if problem needs to find all possible solutions.

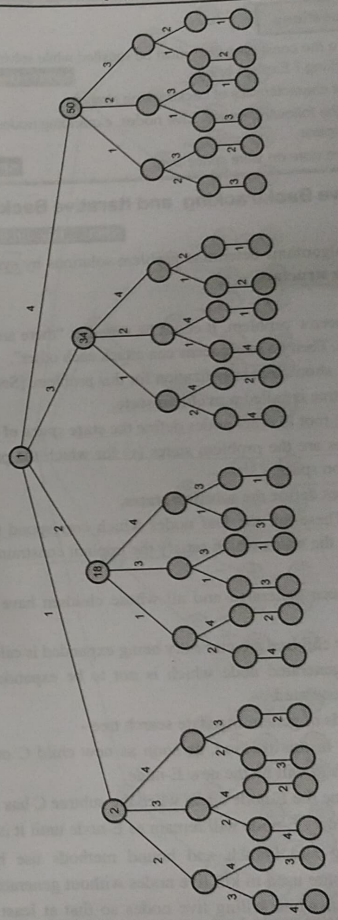


Fig. 6.1.1 State space tree for 4-queen's problems

Review Questions

Q.1 What are the constraints that must be satisfied while solving any problem using backtracking ? Explain briefly. **SPPU : Dec.-06, Marks 4, Dec.-10, Marks 6**

Q.2 Enlist the characteristics of backtracking strategy. **SPPU : May-08, Marks 2**

Q.3 Explain the following terms : Live nodes, expanding nodes, bounding function and solution space. **SPPU : Dec.-13, Marks 8**

Q.4 Write short note on state space tree. **SPPU : May-15, End Sem, Marks 2**

6.2 Recursive Backtracking and Iterative Backtracking Algorithm

SPPU : Dec.-06,07, 18, May-07,15, 18, Marks 8

Backtracking algorithms determine problem solutions by systematically searching for the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can be placed on 4×4 chessboard. Then no two queens can attack each other".

Following Fig. 6.2.1 shows tree organization for this problem. [See Fig. 6.2.1 on next page]

- Each node in the tree is called **a problem state.**
- All paths from the root to other nodes define the **state space of** the problem.
- The solution states are the problem states (s) for which the path from root to s defines a **tuple** in the solution space.

In some trees the leaves define the **solution states.**

- **Answer states :** These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy the implicit constraints.

For example

- A node which is been generated and all whose children have not yet been generated is called **live node.**
- The live node whose children are currently being expanded is called **E-node.**
- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.
- There are two methods of generating state search tree -

i) Backtracking - In this method, as soon as new child C of the current E-node N is generated, this child will be the new E-node.

The N will become the E-node again when the subtree C has been fully explored.

ii) Branch and Bound - E-node will remain as E-node until it is dead.

- Both bracktracking and branch and bound methods use bounding functions. The bounding functions are used to kill live nodes without generating all their children. The care has to be taken while killing live nodes so that at least one answer node or all answer nodes are obtained.

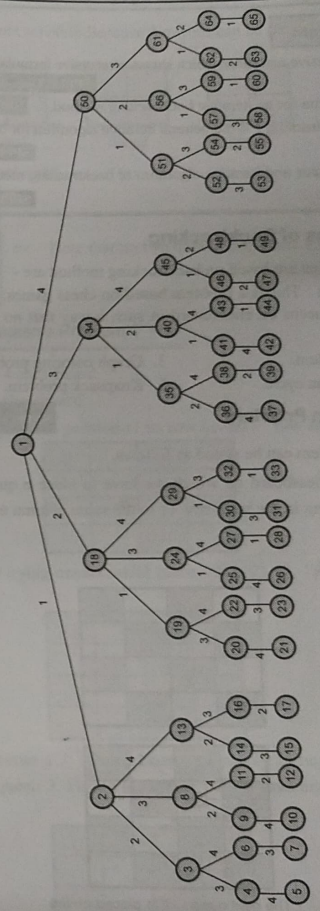


Fig. 6.2.1 State-space tree for 4-queen's problem

Review Questions

Q.1 Write a recursive algorithm which shows a recursive formulation of the backtracking technique. **SPPU : Dec.-06, Dec.-18, End Sem, Marks 8**

Q.2 Write a schema for an iterative backtracking method. **SPPU : May-07, Dec.-07, Marks 8**

Q.3 What is backtracking ? Write general iterative algorithm for backtracking. **SPPU : May-15, End Sem, Marks 8**

Q.4 Write a recursive and Iterative algorithm of backtracking method. **SPPU : May-18, End Sem, Marks 8**

6.3 Applications of Backtracking

Various applications that are based on backtracking method are -

1. 8 Queen's problem : This is a problem based on chess games. By this problem, it is stated that arrange the queens on chessboard in such a way that no two queens can attack each other.
2. Sub of subset problem.
3. Graph coloring problem.
4. Finding Hamiltonian cycle.
5. Knapsack problem.

6.4 The n-Queen Problem

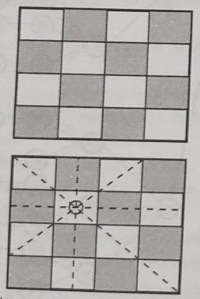
SPPU : May-11,12,14, 19, Marks 10

The n-queen's problem can be stated as follows.

Consider a n × n chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

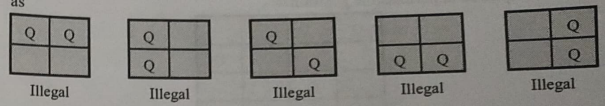
For example

Consider 4 × 4 board

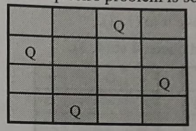


The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 2-Queen's problem is not solvable-Because 2-queens can be placed on 2 × 2 chessboard as



- But 4-queen's problem is solvable.

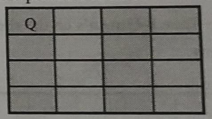


⇐ Note that no two queens can attack each other.

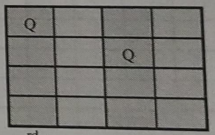
6.4.1 How to Solve n-Queen's Problem ?

Let us take 4-queens and 4 × 4 chessboard.

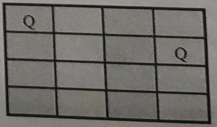
- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row i.e. on 1st row and 1st column.



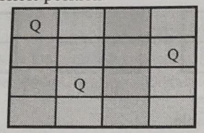
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2nd row and 3rd column.



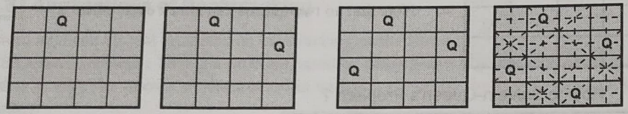
- This is the dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2nd queen at (2, 4) position.



- The place 3rd queen at (3, 2) but it is again another dead end as next queen (4th queen) cannot be placed at permissible position.



- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).

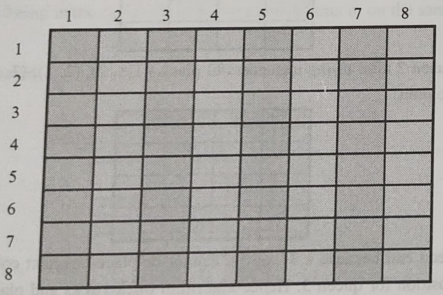


Thus solution is obtained.
(2, 4, 1, 3) in rowwise manner.

The state space tree of 4-queen's problem is shown in Fig. 6.4.1 [See Fig. 6.4.1 on next page].

Now we will consider how to place **8-Queen's** on the chessboard.

- Initially the chessboard is empty.



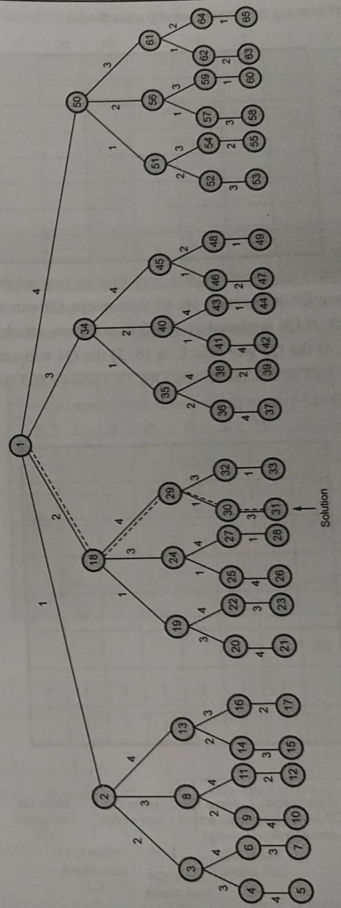
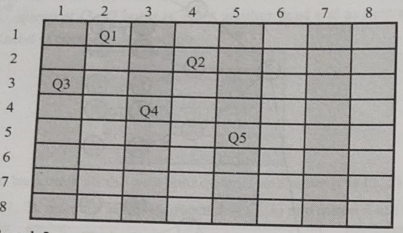
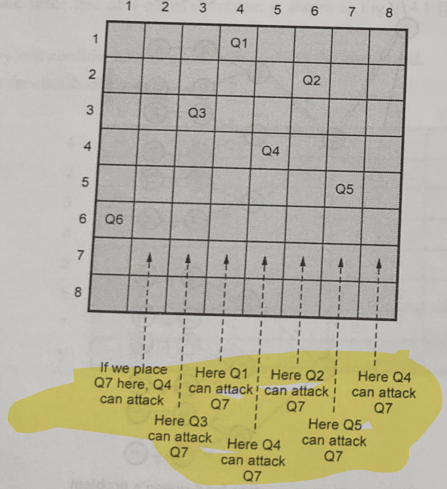


Fig. 6.4.1 State space tree for 4-queen's problem

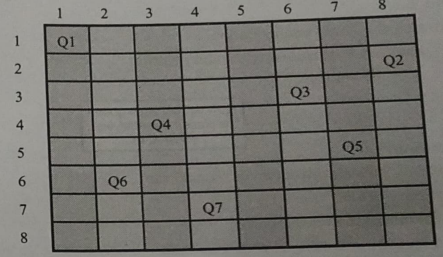
Now we will start placing the queens on the chessboard.



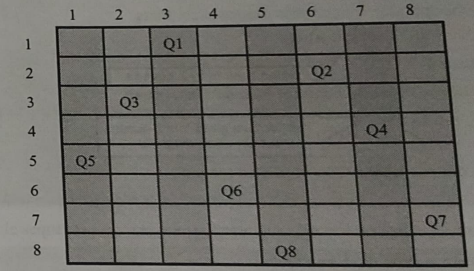
Thus we have placed 5 queens in such a way that no two queens can attack each other. Now if we want to place Q6 at location (6, 6) then queen Q5 can attack, if Q6 is placed at (6, 7) then Q1 can attack, if Q6 is placed at (6, 8) then Q2 can attack it. Similarly at (6, 5) the Q5 will attack it. At (6, 4) the Q2 will attack, at (6, 3) the Q4 will attack, at (6, 2) the Q1 will attack and at (6, 1) Q3 will attack the queen Q6. This shows that we need to backtrack and change the previously placed queens positions. It could then be -



Hence we have to backtrack to adjust already placed queens.



But again Q8 cannot be placed at any empty location safely. Hence we need to backtrack. Finally the successful placement of all the eight queens can be shown by following figure.



6.4.2 Algorithm

```
Algorithm Queen(n)
//Problem description: This algorithm is for implementing n
//queen's problem
//Input : total number of queen's n.
        for column ← 1 to n do
        {
            if(place(row,column))then
            {
             board[row]column//no conflict so place queen
```

This function checks if two queens are on the same diagonal or not.

```
            if(row=n)then//dead end
          print_board(n)
          //printing the board configuration
              else//try next queen with next position
                  Queen(row+1,n)
              }
        }
```

Row by row each queen is placed by satisfying constraints.

```
Algorithm place(row,column)
//Problem Description : This algorithm is for placing the
//queen at appropriate position
//Input : row and column of the chessboard
//output : returns 0 for the conflicting row and column
//position and 1 for no conflict.
        for i ← 1 to row-1 do
        { //checking for column and diagonal conflicts
          if(board[i] = column)then
          return 0
        else if(abs(board[i]- column) = abs(i - row))then
            return 0
        }
        //no conflicts hence Queen can be placed
    return 1
```

Same column by 2 queen's

This formula gives that 2 queens are on same diagonal

C Functions

```
/* By this function we try the next free slot
and check for proper positioning of queen
*/
void Queen(int row,int n)
{
        int column;
        for(column=1;column<=n;column++)
        {
            if(place(row,column))
            {
             board[row] = column; //no conflict so place queen
             if(row==n)   //dead end
```



```
                print_board(n);
              //printing the board configuration
              else  //try next queen with next position
                  Queen(row+1,n);
              }
        }
}
int place(int row,int column)
{
        int i;
        for(i=1;i<=row – 1;i++)
        { //checking for column and diagonal conflicts
            if(board[i] == column)
                return 0;
            else
                if(abs(board[i] – column) == abs(i – row))
                    return 0;
        }
        //no conflicts hence Queen can be placed
        return 1;
}
```

Ex. 6.4.1 *Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).*

Sol. : As the feasible sequence is given, we will place the queens accordingly and then try out the other remaining places.

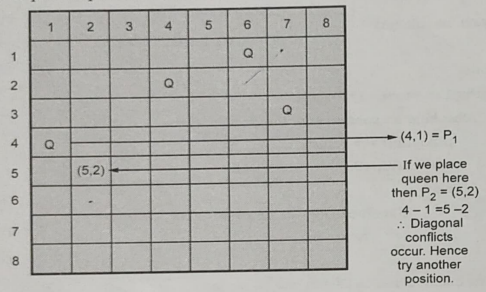
	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5								
6								
7								
8								

The diagonal conflicts can be checked by following formula -

Let, $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions. Then P_1 and P_2 are the positions that are on the same diagonal, if

$$i + j = k + l \quad \text{or}$$
$$i - j = k - l$$

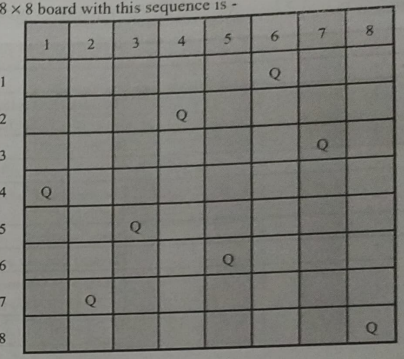
Now if next queen is placed on (5, 2) then



It can be summarized below.

Queen Positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				As 4 – 1 = 5 – 2 conflict occurs.
6	4	7	1	3				5 – 3 ≠ 4 –1 or 5 + 3 ≠ 4 + 1 ∴ Feasible
6	4	7	1	3	2			As 5 + 3 = 6 + 2. It is not feasible.
6	4	7	1	3	5			Feasible
6	4	7	1	3	5	2		Feasible
6	4	7	1	3	5	2	8	List ends and we have got feasible sequence.

The 8-queens on 8 × 8 board with this sequence is -



8-queens with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

Ex. 6.4.2 *Draw the tree organization of the 4-queen's solution space. Number the nodes using depth first search.*

Sol. : The state space tree for 4-queen's problem consists of 4! leaf nodes. That means there are 24 leaf nodes. The solution space is defined by a path from root to leaf node.

The solution can be obtained for 4 queen's problem. For instance from 1 to leaf 31 represents one possible solution.

Ex. 6.4.3 *For a feasible sequence (7, 5, 3, 1) solve 8 queen's problem using backtracking.*

SPPU : May-14, Marks 10

Sol. : While placing the queen at next position we have to check whether the current position chosen is on the same diagonal of previous queen's position.

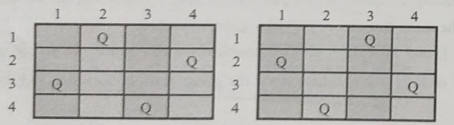
If $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions then P_1 and P_2 lie on the same diagonal if $i + j = k + l$ or $i - j = k - l$. Let us put the given feasible sequence and try out remaining positions.

(1), 2, (3), 4, (5), 6, (7), 8 → Already occupied position.

i values i.e. row values	j → 1	2	3	4	5	6	7	8	
	7	5	3	1					
	7	5	3	1	2				The remaining vacant position is 2. But 4 – 1 = 5 – 2. Not feasible.
	7	5	3	1	4				No conflict.
	7	5	3	1	4	6			(3, 3) = (6, 6) Conflict occurs.
	7	5	3	1	4	8			List ends and we can not place further queen's.
	7	5	3	1	4				Hence backtrack.
	7	5	3	1	6				The next free slot 6 is tried.
	7	5	3	1	6	2			Not feasible because (7, 1) = (2, 6) 1 + 7 = 6 + 2. That is on same diagonal.
	7	5	3	1	6	4			(1) 2 (3) (4) (5) (6) (7) 8 Occupied are circled.
	7	5	3	1	6	4	2		
	7	5	3	1	6	4	2	8	Conflict because (3, 3) = (8, 8)
	7	5	3	1	6	4	2		Hence backtrack.
	7	5	3	1	6	4	8		(6, 5) = (8, 7). Not feasible.
	7	5	3	1	6	4			Backtrack.
	7	5	3	1	6	8			
	7	5	3	1	6	8	2		
	7	5	3	1	6	8	2	4	List ends with solution of 8 queen's.

Ex. 6.4.4 *Obtain any two solutions to 4-queen's problem. Establish the relationship between them.*

Sol. : The solutions to 4-queen's problem are as given below.



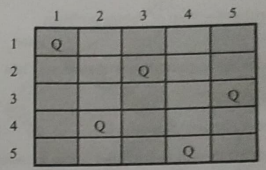
Solution 1 (2, 4, 1, 3)

Solution 2 (3, 1, 4, 2)

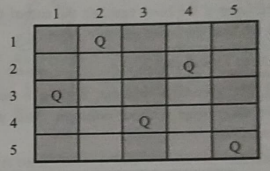
If these two solutions are observed then we can say that second solution can be simply obtained by reversing the first solution.

Ex. 6.4.5 *Generate at least 3 solutions for 5-queen's problem.* **SPPU : May-12, Marks 8**

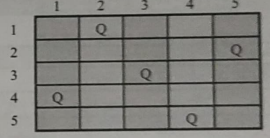
Sol. : The solutions are as given below.



Solution 1 (1, 3, 5, 2, 4)



Solution 2 (2, 4, 1, 3, 5)



Solution 3 (2, 5, 3, 1, 4)

Review Questions

Q.1 Analyze the 8-queen problem using backtracking strategy of problem solving. **SPPU : May-11, Marks 8**

Q.2 What is backtracking technique ? Find one solution for 4 - Queen's problem. Show all the steps and explain why you need to backtrack.

Q.3 State the principle of backtracking and write backtracking algorithm for N-Queen problem. **SPPU : May-19, End Sem, Marks 8**

6.5 Sum of Subsets

SPPU : May-10,11,15,17, 18, 19, Dec.-08,09,11,12,15, 18, Marks 12

Problem Statement

Let, $S = \{S_1, \ldots, S_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

It is always convenient to sort the set's elements in ascending order. That is,

$$S_1 \leq S_2 \leq \ldots \leq S_n$$

Let us first write a general algorithm for sum of subset problem.

Algorithm

Let, S be a set of elements and d is the expected sum of subsets. Then -

Step 1 : Start with an empty set.

Step 2 : Add to the subset, the next element from the list.

Step 3 : If the subset is having sum d then stop with that subset as solution.

Step 4 : If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5 : If the subset is feasible then repeat step 2.

Step 6 : If we have visited all the elements without finding a suitable subset and if backtracking is possible then stop without solution. This problem can be well understood with some example.

Ex. 6.5.1 *Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Solve it for obtaining sum of subset.* **SPPU : May-17, 18, End Sem, Dec. 18, End Sem, Marks 8**

Sol. :

Initially subset = { }	Sum = 0	–
5	5	Then add next element.
5, 10	15 ∵ 15 < 30	Add next element.
5, 10, 12	27 ∵ 27 < 30	Add next element.
5, 10, 12, 13	40	Sum exceeds d = 30 hence backtrack.
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack
5, 10, 12, 18	45	Sum exceeds d ∴ Not feasible. Hence backtrack.
5, 10		
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible ∴ Backtrack.
5, 10		
5, 10, 15	30	Solution obtained as sum = 30 = d

∴ The state space tree can be drawn as follows. {5, 10, 12, 13, 15, 18}

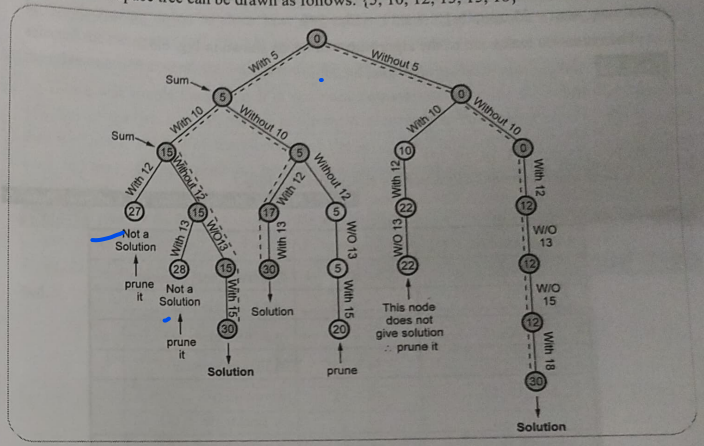


Fig. 6.5.1 State space tree for sum of subset

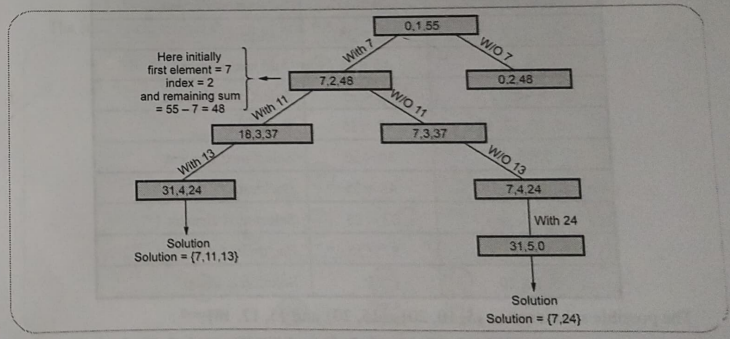


Fig. 6.5.2

Ex. 6.5.2 *Let m = 31 and W = {7, 11, 13, 24}. Draw a portion of state space tree for solving sum-of-subset problem for the above given algorithm.*

Sol. : Initially we pass (0, 1, 55) to sum-of-subset function. The sum = 0, index = 1 and remaining_sum = 55 initially [$\because$ 7 + 11 + 13 + 24].

The recursive execution of the algorithm will be, as shown in Fig. 6.5.2.

Ex. 6.5.3 *Analyze sum of subset algorithm on data :*
M = 35 and
i) W = {5, 7, 10, 12, 15, 18, 20}
ii) W = {20, 18, 15, 12, 10, 7, 5}
iii) W = {15, 7, 20, 5, 18, 10, 12}
Are there any discernible differences in the computing time ?

SPPU : May-11, Marks 10, May 19, End Sem, Marks 5

Sol. :

Initially Subset = { }	Sum = 0	
5	5	Then add next element
5, 7	12 < 35	Add next element
5, 7, 10	22 < 35	Add next element
5, 7, 10, 12	34 < 35	Add next element
5, 7, 10, 12, 15	49 > 35	Sum exceeds $\therefore$ Backtrack
5, 7, 10, 15	37 > 35	Backtrack
5, 7, 12	24 < 35	Select next element
5, 7, 12, 15	39 > 35	Backtrack
5, 10	15 < 35	Add next element
5, 10, 12	27< 35	Select next element
5, 10, 12, 15	42 > 35	Backtrack
5, 10, 15	30 < 35	Select next element
5, 10, 15, 18	48 > 35	Backtrack
5, 10, 18	33 < 35	Select next element
5, 10, 18, 20	53 > 35	Backtrack
5, 10, 20	35	Solution is found

The possible solutions are {5, 10, 20}, {15, 20} and {5, 12, 18}.

The sequence (ii) W = {20, 18, 15, 12, 10, 7, 5} and (iii) W = {15, 7, 20, 5, 18, 10, 12} are not in non-decreasing order. The (ii) sequence is in decreasing order and (iii) is totally unsorted.

For sum of subset problem, For efficient findings of solution the weights must be arranged in non-decreasing order. to show how this order is beneficial consider a scenario we have selected the elements {5, 7, 10, 12} which sums up 34. Now if we select the next element 15, then this sum will exceed the limit because 49 > 35.

So we will simply backtrack. It is not even necessary to examine the remaining elements {18, 20} of the list. If at all, the list is arranged according to decreasing order of the weights then we have to analyse all the elements of the given list. Similarly if the list is unsorted then each and every element needs to be analysed for required sum. Thus the sequences (ii) and (iii) makes the computing time worst.

Ex. 6.5.4 *What is backtracking method for algorithmic design ? Solve the sum of subset problem using backtracking algorithmic strategy for the following data N = 4 (w_1, w_2, w_3, w_4) = (11, 13, 24, 7) and M = 31.*

SPPU : Dec.-11,12, Marks 12

Sol. :

Initially subset = { }	sum = 0	
11	13 < 31	Add next element
11, 13	24 < 31	Add next element
11, 13, 24	48 > 31	Backtrack
11, 13	24 < 31	Add 7
11, 13, 7	31	Solution is found i.e. { 11, 13, 7 }

The state space tree as shown in Fig. 6.5.3 -

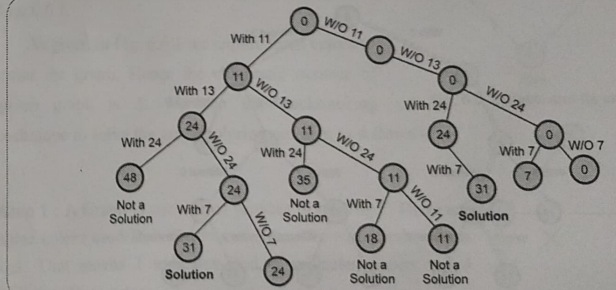


Fig. 6.5.3

Thus subsets {11, 13, 7} and {24, 7} are the solutions.

Ex. 6.5.5 *Draw a pruned state space tree for a given sum of subset problem. S = {3, 4, 5, 6} and d = 13*

Sol. : Fig. 6.5.4 state space tree the total sum is given inside each node. Pruned nodes has cross at its bottom.

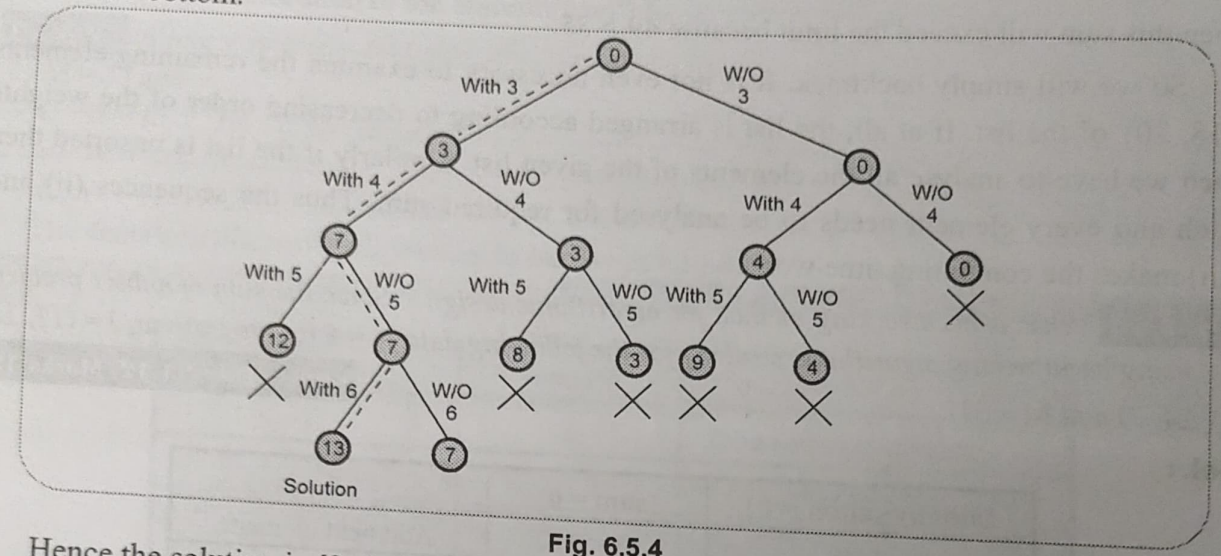


Fig. 6.5.4

Hence the solution is **{3, 4, 6}.**

Ex. 6.5.6 *Differentiate between backtracking and branch and bound. Draw state space tree for given sum of subset problem : Set of elements = { 3, 5, 6, 7 } and d = 15*

SPPU : Dec.-15, End Sem, Marks 8

Sol : Difference : (Refer section 7.1.1)

Following is a state space tree in which total sum is represented inside each node. Pruned node has a cross at its bottom. (Fig. 6.5.5)

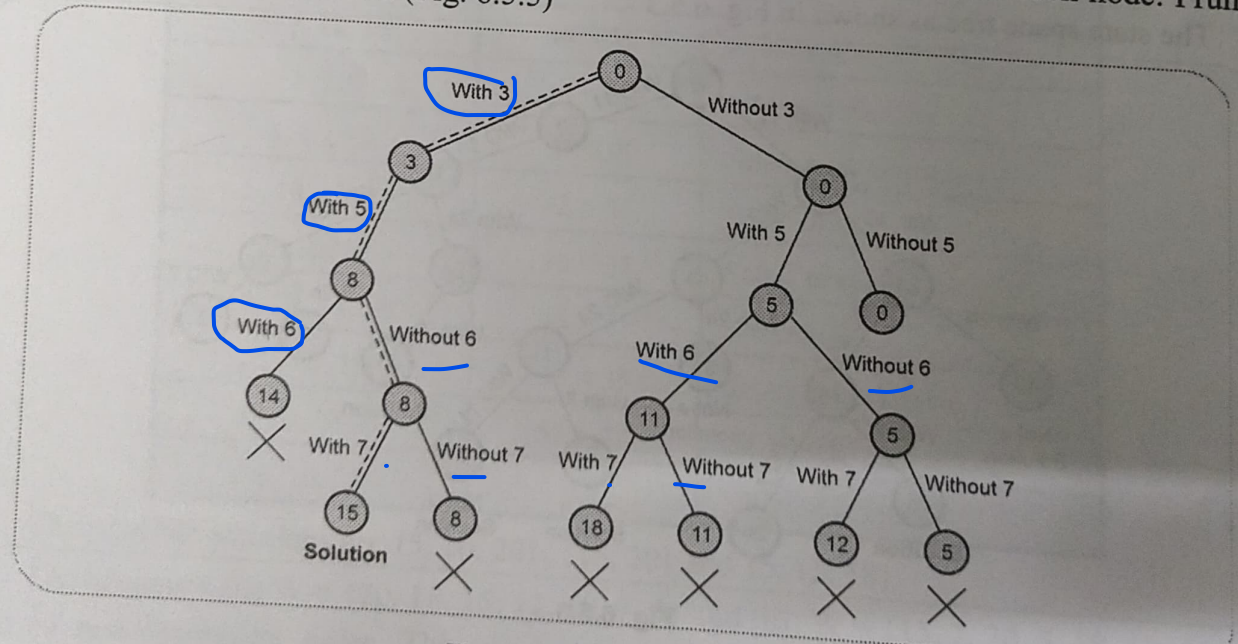


Fig. 6.5.5 State space tree

The solution is = **{3, 5, 7}**

Strategy to Solve Sum of Subsets Problem

1. Choose each number and sum it up. Compare it with the given sum d.
2. If the sum of chosen numbers is less than d then add the next subsequent element.
3. If the sum of chosen numbers is greater than d then remove one element and try some another number in the set.
4. Thus all the permutations of the numbers for summing up should be attempted.
5. When we get the sum=d then declare the chosen set of numbers as the solution.

Review Question

Q.1 Write a recursive backtracking algorithm for sum of subsets of problem.

SPPU : Dec.-08, 09, May-10, May-15, End Sem, Marks 8

6.6 Graph Coloring

SPPU : May-08,10,11,17, Dec.-12,13, Marks 12

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used. This problem is also called m-coloring problem. If the degree of given graph is d then we can color it with d + 1 colors. The least number of colors needed to color the graph is called its chromatic number. For example : Consider a graph given in Fig. 6.6.1.

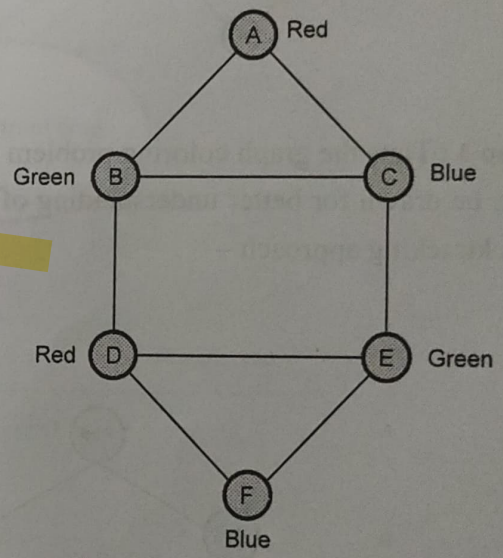
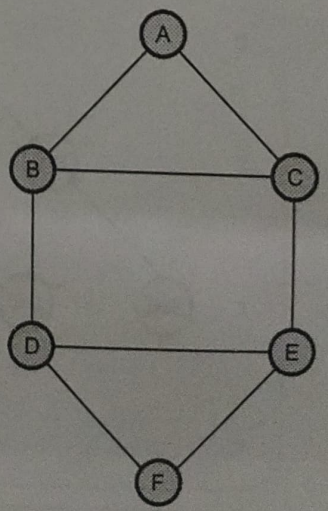


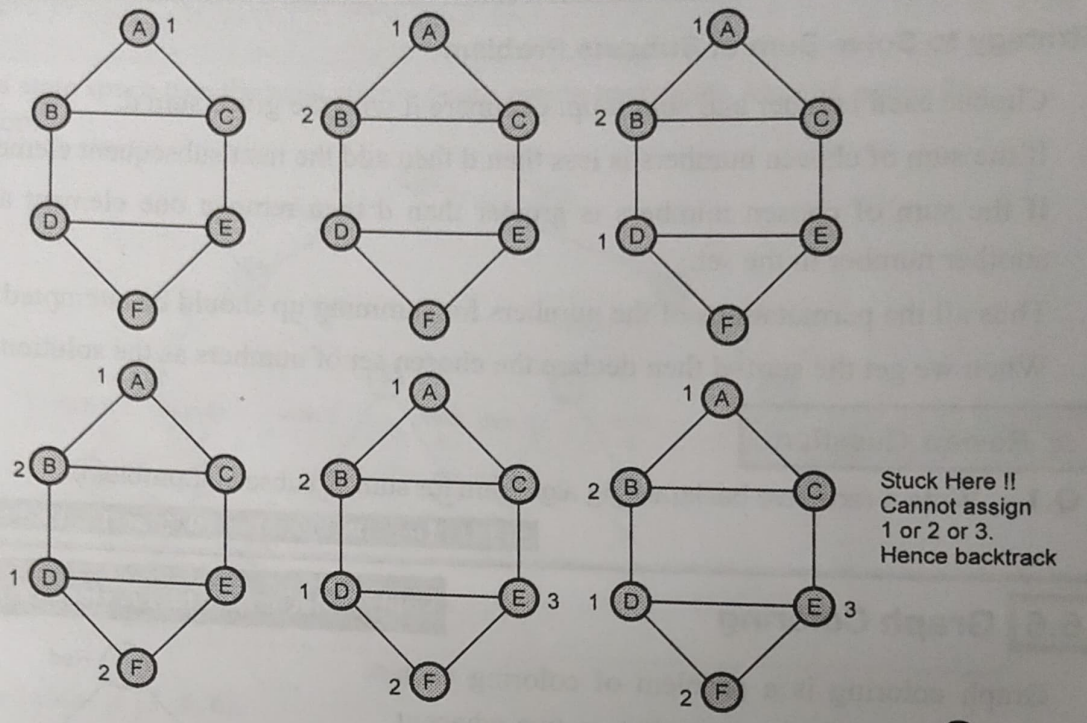
Fig. 6.6.1 Graph and its coloring

As given in Fig. 6.6.1 we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows -

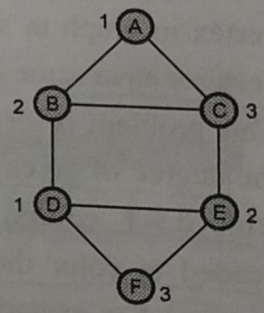


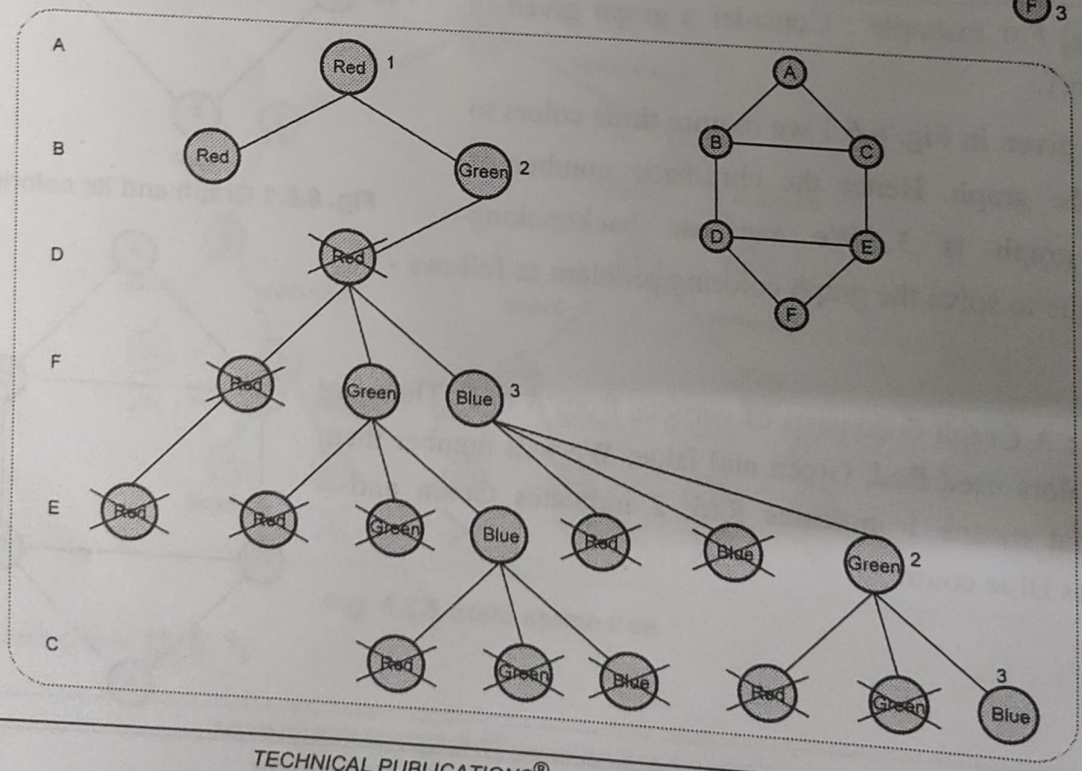
Step 1 : A Graph G consists of vertices from A to F. There are three colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.

Step 2 :



Step 3 : Thus the graph coloring problem is solved. The state-space tree can be drawn for better understanding of graph coloring technique using backtracking approach –





Here we have assumed, color index Red = 1, Green = 2 and Blue = 3.

6.6.1 Algorithm

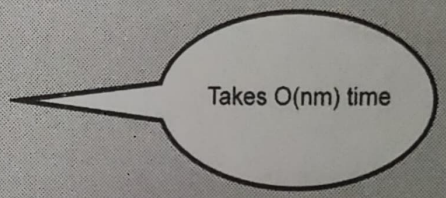
The algorithm for graph coloring uses an important function Gen_Col_Value() for generating color index. The algorithm is -

```
Algorithm Gr_color(k)

//The graph G[1 : n,1 : n] is given by adjacency matrix.
//Each vertex is picked up one by one for coloring
//x[k] indicates the legal color assigned to the vertex
{
 repeat
 {   // produces possible colors assigned

      Gen_Col_Value(k);

 if(x[k] = 0) then
    return; //Assignment of new color is not possible.
 if(k=n) then // all vertices of the graph are colored
   write(x[1:n]); //print color index
 else
  Gr_color(k+1) // choose next vertex
 }until(false);
}
```



The algorithm used assigning the color is given as below

```
Algorithm Gen_Col_Value(k)
 //x[k] indicates the legal color assigned to the vertex
 // If no color exists then x[k] = 0
 {
// repeatedly find different colors
```

```
repeat
 {
   x[k]  (x[k]+1) mod (m+1) ;  //next color index when
                               //it is highest index
   if(x[k] = 0) then     // no new color is remaining
            return;
   for (j  1 to n) do
   {
      // Taking care of having different colors for adjacent
      // vertices by using two conditions i) edge should be
      // present between two vertices
      // ii) adjacent vertices should not have same color
    if(G[k,j]!=0) AND (x[k] x[j])) then
     break;
   }
      //if there is no new color remaining
   if ( j= n+1) then
      return;
  }until(false);
}
```

6.6.2 Analysis

The Gr_color takes computing time of $\sum_{i=0}^{n-1} m^i$. Hence total computing time for this algorithm is $O(nm^n)$.

If we trace the above algorithm then we can assign distinct colors to adjacent vertices. For example : Consider, a graph given below can be solved using three colors.

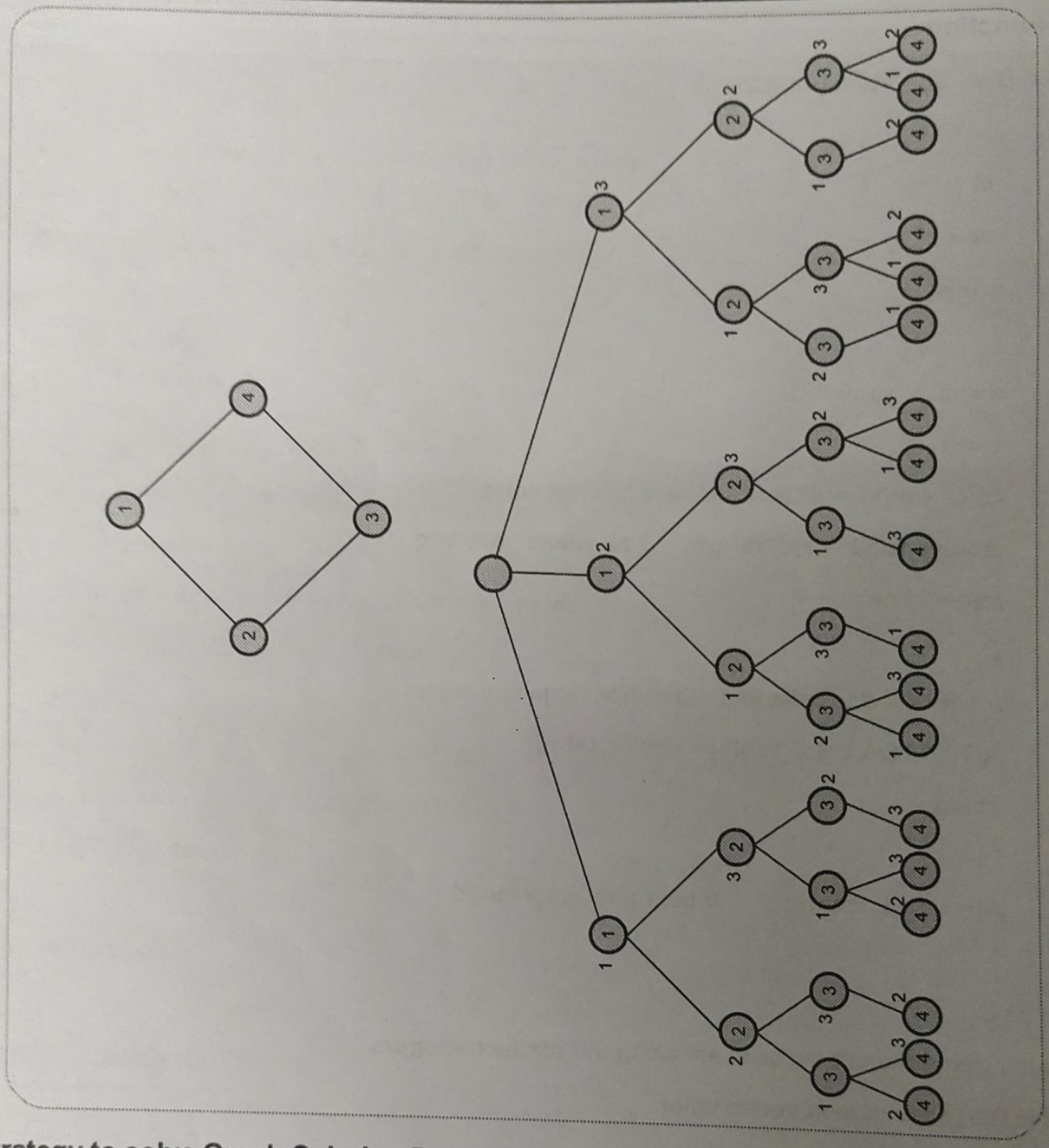
Consider,

Red = 1, Green = 2 and Blue = 3.

The number inside the node indicates vertex number and the number outside the node indicates color index.

C Function

```
void Gen_Col_Value(int k,int n)
{
    int j;
    int a,b;
    while(1)
    {
    a=color_tab[k]+1;
    b=m+1;
    color_tab[k] = a%b;      // next highest color
    if(color_tab[k]==0) return;   // all colors have been used
    for(j=1;j<=n;j++)
    {
    // check if this color is distinct from adjacent colors
    if(G[k][j] && color_tab[k]==color_tab[j])
     break;
    }
    if(j==n+1) return;       // next new color found
  }
}
// such that adjacent vertices are assigned distinct integers
// k is the index of next vertex color.
void Gr_coloring(int k,int n)
{
     Gen_Col_Value(k,n);
    if(color_tab[k]==0) return;   // No new color available
if(k==n)  return;    // at most m colors have been
    else Gr_coloring(k+1,n);  // used to color the n vertices
}
```



Strategy to solve Graph Coloring Problem

1. Choose some color say c and apply to some vertex say v.
2. Go on choosing different colors that are available from the list and apply to each successive vertex. After applying the color just check whether it is same with the adjacent color or not.
3. If the chosen color is not same to the adjacent colors then go to the next adjacent vertex otherwise try out some another color from the list or change the already assigned colors.
4. Repeat this procedure for coloring all the vertices.

Ex. 6.6.1 *Construct plannar graph for following map. Explain how to find m-coloring of this planar graph by using m-coloring backtacking algorithm.*

SPPU : May-11, Marks 10, May-17, End Sem, Marks 8

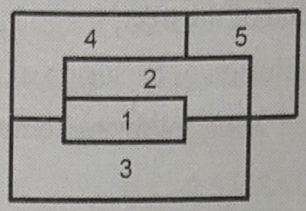


Fig. 6.6.2

Sol. : Refer section 6.6.1 for m-coloring of plannar.

Steps for graph coloring -

1. Number out each vertex (V0, V1, V2,...Vn-1)
2. Number out the available colors (C0,C1,C2,...Cn-1)
3. Start assigning Ci to each Vi. While assigning the colors note that two adjacent vertices should not be colored by the same color. And least number of colors should be used.
4. While assigning the appropriate color, just backtrack and change the color if two adjacent colors are getting the same.

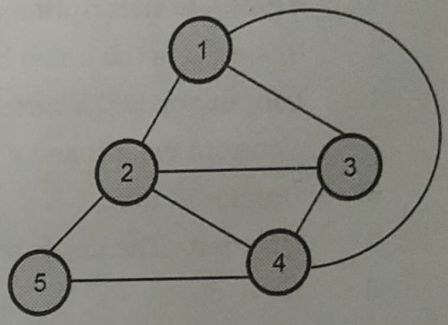


Fig. 6.6.3

Review Questions

Q.1 Write recursive backtracking schema for m-coloring of the graph. Determine the time complexity of the same. **SPPU : May-08, 10, Marks 12**

Q.2 Discuss and analyze the problem of graph coloring using backtracking. **SPPU : Dec.-12, Marks 6**

Q.3 Explain graph coloring problem. **SPPU : Dec.-13, Marks 8**

6.7 The 0/1 Knapsack Problem

SPPU : Dec.-11, May-12,14,15, 18, Marks 16

In this section we will discuss "How Knapsack problem be solved using backtracking". The Knapsack problem can be stated as follows.

Problem statement : "If we are given n objects and a Knapsack or a bag in which the object i with weight is to be placed. The Knapsack has a capacity **m**. Then the profit value that can be earned is p_i. Then **objective** is to obtain filling up of Knapsack with maximum profit earned, but it should not exceed the capacity m of Knapsack.

To fill the given Knapsack we select the object with some weight and having some profit. This selected object is put in the Knapsack. Thus Knapsack is filled up with selected objects. Note that the Knapsack's capacity m should not be exceeded. Hence the first item gives best pay off per weight unit and last one gives the worst pay off per weight unit.

Hence we must arrange the given data in non-increasing order i.e.

$$p_i / w_i \geq p_{i+1} / w_{i+1}$$

- First of all we compute upper bound of the tree.
- We design a state space tree by inclusion or exclusion of some items.

The upper bound can be computed using following formula.

$$\longrightarrow \quad ub + (1 - (c - m) / w_i) * p_i$$

The algorithm for computing an upper bound is as given below -

```
1   Algorithm Bound_Calculation (cp,cw,k)
2   //Problem description : This algorithm
3   //calculates the upper bound, for each item
4   //Input : cp is the current profit, cw is the
5   //current weight and k is the index of just
6   //removed item
7   //Output : The upper bound value can be returned
8   ub ← cp
9   c ← cw
10    for (i ← k+1 to n) do
11  {
12      c ← c+w[i]
13      if (c < m) then              //m is capacity of knapsack
14        ub ← ub + p[i]
15      else
16        {
17            ub ← ub + (1–(c–m)/w[i])*p[i]
18            return ub
19        }
20      return ub
21  }
```

This function is invoked by a Knapsack function **Bk (k, cp, cw)**. The algorithm for the same is as given below.

Algorithm Bk (k,cp,cw)

```
1   //Problem description : This algorithm is to obtain
2   //solution for knapsack problem. Using this algorithm
3   //a state space tree can be generated.
```

```
4     //Input : Initially k=1, cp=0 and cw=0. The
5     //k represents the index of currently referred
6     //item. The cp and cw represent the profit and
7     //weights of items, so far selected.
8     //Output : The set of selected items that are
9     //satisfying the objective of knapsack problem.

10    if (cw+w[i]<= m) then
11    {
12      temp[k] ← 1              //temp array stores currently selected object
13      if (k<n) then
14  Bk(k+1,cp+p[k],cw+w[k])              //recursive call
15  if ((cp+p[k] > final_profit) AND (k==n)) then
16  { //final_profit is initially –1, it represents final profit
17  final_profit ← cp + p[k]
18      final_wt ← cw + w[k]
19      for (j ← 1 to k) then
20          x[j] ← temp[j]
21    }
22 }
23    if (Bound_calculation (cp,cw,k)  final_profit) then
24    {
25        temp[k] ← 0
26        if (k < n) then
27            Bk(k+1, cp, cw)
28        if((cp > final_profit) AND (k=n)) then
29        {
30            final_profit ← cp
31            final_wt ← cw
32            for(j ← 1 to k)
33                x[j] ← temp [j]
34        }
35 }
```

Generates left child

Finally get the solution

Generates right child

Invoke this function in order to obtain upper bound

Finally get the solution

Ex. 6.7.1 *Obtain the optimal solution to the Knapsack problem n = 3, m = 20 (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).* **SPPU : May-14, Marks 10**

Sol. : Given that n = 3 and m = Capacity of Knapsack = 20.

We have

i	p [i]	w [i]	p [i] / w [i]
1	25	18	1.3
2	24	15	1.6
3	15	10	1.5

As we want p[i] / w[i] ≥ p[i+1] / w [i+1], let us rearrange the items as

i	p [i]	w [i]	p [i] / w [i]
1	24	15	1.6
2	15	10	1.5
3	25	18	1.3

Step 1 : Here we will trace knapsack backtracking algorithm using above values.

Initially set **final_profit = – 1**

Bk (1, 0, 0)

∴ k = 1

cp = 0

cw = 0

Check if (cw + w [k] <= m)

i.e. if (0 + w [1] = 0 + 15 <= 20) → yes

∴ set temp [1] = 1

if (k < n) i.e. if (1 < 3) ? → yes

∴ Bk (k + 1, cp + p [k], cw + w [k]) is called

Hence Bk (2, 0 + 24, 0 + 15) will be called. Refer line 14 of **Algorithm Bk ( ).**

Step 2 :

k = 2

cp = 24

cw = 15

Check if (cw + w [k] <= m)

i.e. if (15 + w [2] i.e. 15 + 10 <= 20) → no

Hence we will calculate ub i.e. upper bound.

Step 3 : For calculating upper bound, initially set

$$ub = cp = 24$$

$$c = cw = 15$$ Refer line 8 and 9 of Algorithm Bound_calculation

for (i = k+1 to n) we will update upper bound value.

Consider i = k+1 = 3

$$c = c + w[i]$$

$$c = 15 + w[3]$$

$$\therefore \quad c = 15 + 18 = 33$$

As 33 > 20 i.e. exceeding the Knapsack capacity, we will compute ub as

$$ub = ub + (1 - (c = m) / w[i]) * p[i]$$ ← See line 17 of Bound_calculation

$$\therefore \quad ub = 24\ (1 - (15 - 20) / w[3] * p[3]$$

$$= 24 + (1 - 5 / 18) * 25$$

$$\mathbf{ub = 30.94 \approx 31}$$

As (ub > final_profit) i.e. 30.94 > – 1

Set temp [2] = 0

Now refer line 27 of **Bk** algorithm, a recursive call with k = 3 is made.

Step 4 :

$$k = 3$$

$$cp = 24 + p[2] = 24 + 15 = 39$$

$$cw = 15 + w[2] = 15 + 10 = 25$$

Check if (cw + w [k] <= m)

i.e. if (25 + w [3] = 25 + 18 < = 20) → no

Hence we will calculate upper bound.

Step 5 : For calculating upper bound, initially set

$$ub = cp = 39$$
$$c = cw = 25$$ } Refer line 8 and 9 of algorithm **Bound_calculation**

Set i = k + 1 i.e. 3 + 1 = 4 But i = 4 > n

Hence we will keep ub as it is

$$\therefore \quad ub = 39$$

As (39 > final_profit) i.e. 39 > – 1

Set temp [k] = temp [3] = 0

Now k = n = 3 and cp = 39 > final_profit (i.e. – 1)

final_profit = cp = 39
final_wt = cw = 25
} Refer line 30 and 31 of algorithm **Bk**

Now copy all the contents of temp array to an array x [ ]. Refer line 32, 33 of **Algorithm Bk.**

Hence **x = {1, 0, 0}** i.e. item 1 with **weight = 15** and **profit = 24** is selected.

The state space tree can be drawn as

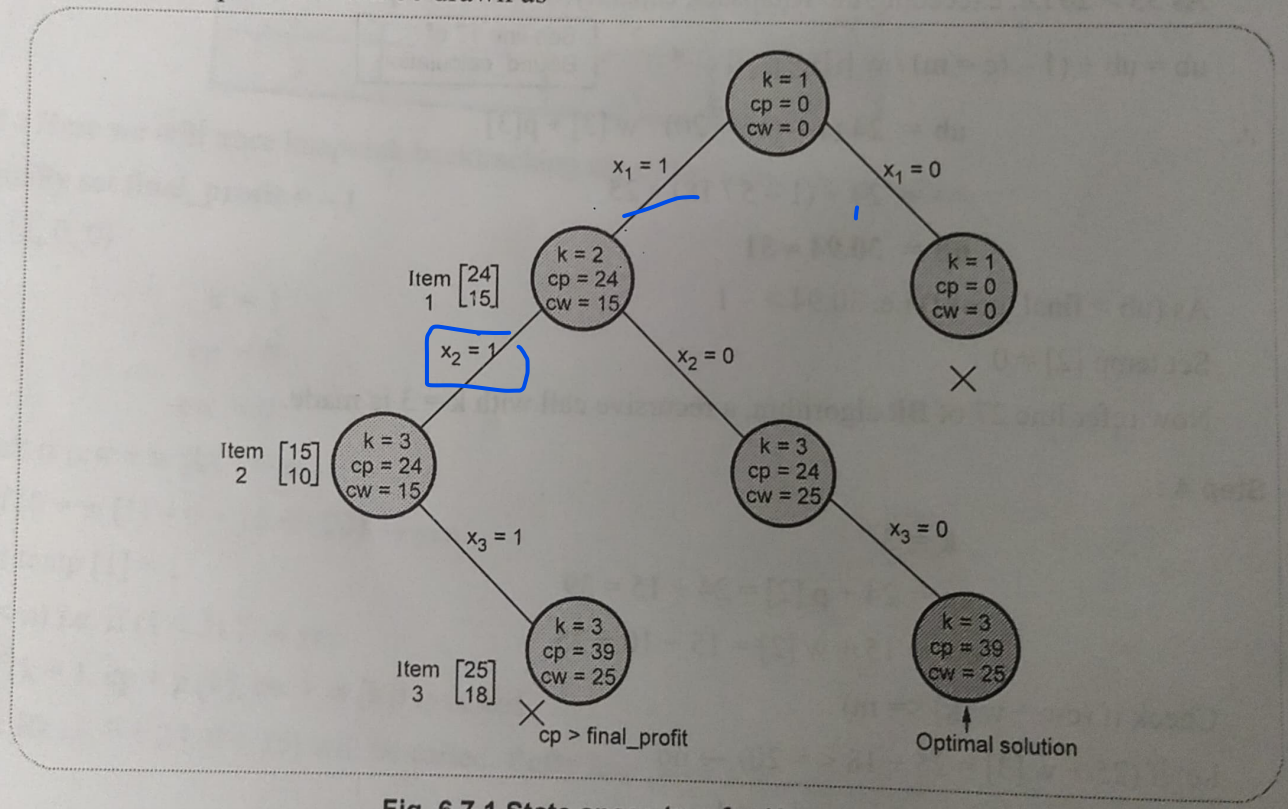


Fig. 6.7.1 State space tree for Knapsack

Here the left branch indicates inclusion of item and right branch indicates exclusion of items. Hence $x_1 = 1$ means 1^{st} item selected and $x_2 = 0$, $x_3 = 0$ means 2^{nd} and 3^{rd} items are not selected. The state space tree can be created in DFS manner.

C Function

```
float Bound_Calculation(int cp,int cw,int k)
{
    int ub,c,i;
    ub=cp;c=cw;
    for(i=k+1;i<=n;i++)
    {
        c=c+w[i];
        if(c<m)
            ub=ub+p[i];

        else
            return(ub+(1-(c-m)/w[i])*p[i]);
    }
    return ub;
}
void BK(int k,int cp,int cw)
{
    int new_k,new_cp,new_cw,j;
    //Generate left child
    if(cw+w[k]<=m)
    {
        temp[k]=1;
        if(k<n)
        {
            new_k=k+1;
            new_cp=cp+p[k];
            new_cw=cw+w[k];
            BK(new_k,new_cp,new_cw);
        }
        if((new_cp>final_profit)&&(k==n))
        {
        final_profit=new_cp;
        final_wt=new_cw;
        for(j=1;j<=k;j++)
            x[j]=temp[j];
```

```
        }
    }
    //Generate right child
    if(Bound_Calculation(cp,cw,k)>=final_profit)
    {
        temp[k]=0;
        if(k<n)
            BK(k+1,cp,cw);
        if((cp>final_profit)&&(k==n))
        {
            final_profit=cp;
            final_wt=cw;
            for(j=1;j<=n;j++)
            x[j]=temp[j];
        }
    }
}
```

Ex. 6.7.2 *Consider the following instance for knapsack problem using backtracking n = 8, P = (11, 21, 31, 33, 43, 53, 55, 65) W = (1, 11, 21, 23, 33, 43, 45, 55) M = 110.*

SPPU : Dec.-11, Marks 6, May-12, Marks 16

Sol. : We will arrange all the items in $\frac{P_i}{W_i} > \frac{P_{i+1}}{W_{i+3}} > \frac{P_{i+2}}{W_{i+3}} \ldots$

The instance will be

Item	P_i	W_i	P_i / W_i
1	11	1	11
2	21	11	1.90
3	31	21	1.47
4	33	23	1.43
5	43	33	1.30
6	53	43	1.23
7	55	45	1.22
8	65	55	1.18

Let M = 110

Initially total-profit = – 1

cp = 0, cw = 0 We use formula as

cp = cp + p[i], cw = cw + w[i] where i represents item number.

cp = cp + 11 = 0 + 11 = 11

cw = cw + 1 = 0 + 1 = 1 < m ∴ select item1

k = 1, p [i] = 21

w [i] = 11

∴ cp = 11 + 21 = 32

cw = 1 + 11 = 12 < 110 ∴ select item 2

k = 2 p [3] = 31

w [3] = 21

∴ cp = 32 + 31 = 63

cw = 12 + 21 = 33 < 110 ∴ select item 3

k = 3 p [4] = 33

w [4] = 23

cp = 63 + 33 = 96

cw = 33 + 23 = 56 < 110 ∴ select item 4

k = 4 p [5] = 43

w [5] = 33

∴ cp = 96 + 43 = 139

cw = 33 + 56 = 89 < 110 ∴ select item 5

k = 5 p [6] = 53

w [6] = 43

∴ cp = 139 + 53 = 192

cw = 99 + 43 = 142 > 110

∴ not to select item 6

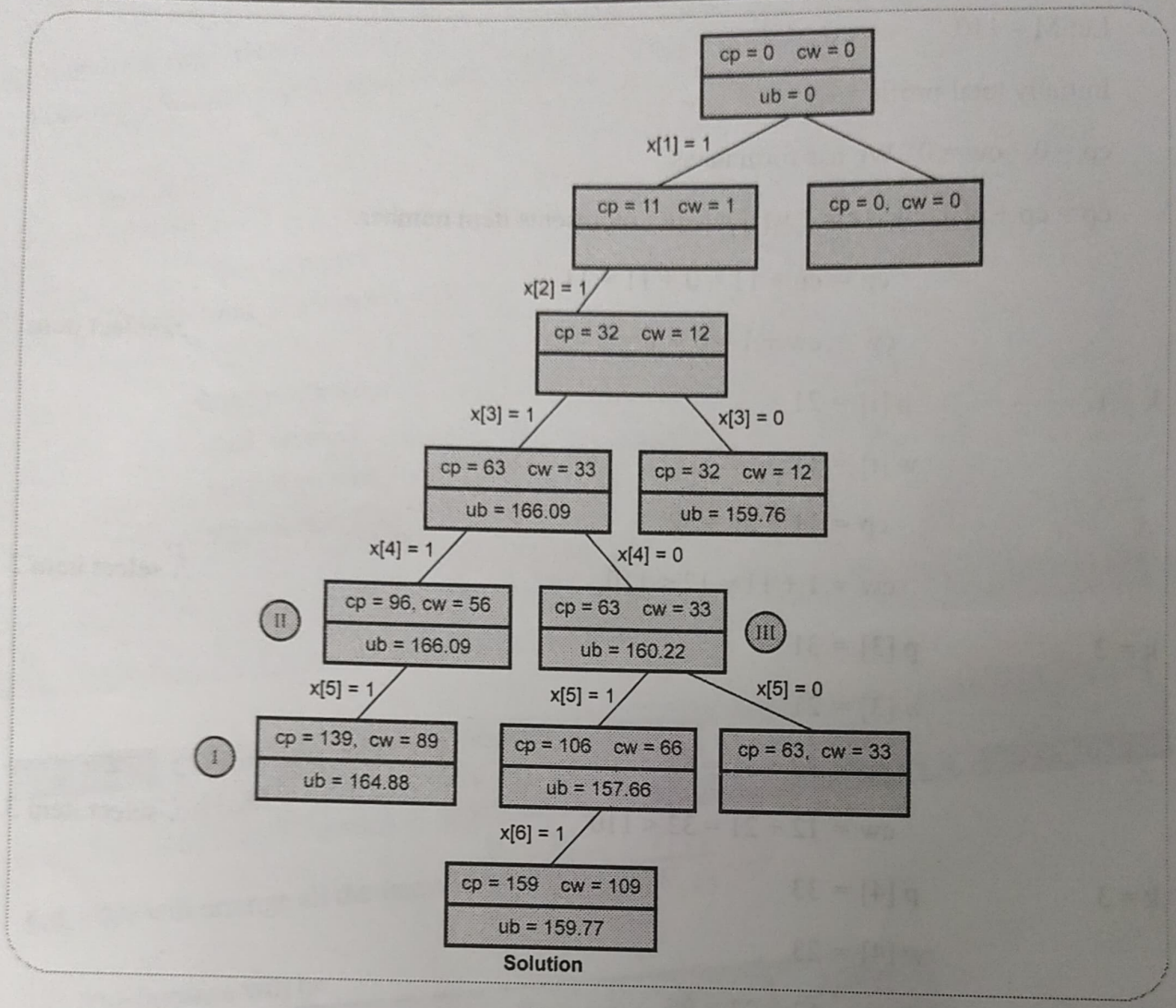


Fig. 6.7.2

Computation at node I

$$ub = cp + \left(\frac{m - cw}{W_{i+1}}\right) * P_{i+1}$$

$$ub = 11 + 21 + 31 + 33 + 43 + \left(\frac{110 - 89}{43}\right) * 53$$

$$ub = 164.88$$

Computation at node II

$$ub = 11 + 21 + 33 + \left(\frac{110 - 56}{33}\right) * 43$$

$$= 166.09$$

Computation at node III

$$ub = 11 + 21 + 31 + 43 + 53 + \left(\frac{110 - 109}{45}\right) * 55$$

$$ub = 160.22$$

Thus upper bound will be,

$11 + 21 + 31 + 33 + 43 + \left(\frac{110-89}{43}\right) * 53 = 164.88$ on selecting item 6, 7, 8 the total weight will exceed the capacity hence we will back track at item 4.

That means item 1, item 2, item 3, item 4 are selected. Thus upper bound will be calculated and a space tree is constructed, as shown in Fig. 6.8.2.

The solution giving maximum profit will be by selection of **item 1, item 2, item 3, item 5** and **item 6.**

Review Question

Q.1 Write an algorithm for 0/1 knapsack problem using backtracking method.

SPPU : May-15, 18, End Sem, Marks 8

6.8 Multiple Choice Questions

Q. 1 A backtracking algorithm one problem instance might generate only _____ .

a O(n) nodes b (n) nodes

c n! nodes d 2^n nodes

Q. 2 In n-queen's problem when n=4 then how many leaf nodes exist ?

a 16 b 64 c 24 d 32

Q. 3 _____ are those solutions state s for which the path from root to s defines the tuple that is a member of the set of solutions of the problem

a Problem state b Answer state

c Live nodes d None of the above

Q. 4 Following is a solution for 5 queen's problem

a (1,2,5,3,4) b (1,2,3,4,5)

c (5,4,3,2,1) d (1,3,5,2,4)

Q. 5 Following data structure is used for generating the solution using backtracking

a Binary Tree b Heap c State Space Tree d B-Tree

UNIT - V

7 Branch and Bound

Syllabus

The method, Control abstractions for Least Cost Search, Bounding, FIFO branch and bound, LC branch and bound, 0/1 Knapsack problem - LC branch and bound and FIFO branch and bound solution, Traveling salesperson problem - LC branch and bound

Contents

7.1	Basic Concept	..... **May-15,**	Marks 5
7.2	The Method	..... **May-07,09, Dec.-07,**.....	Marks 8
7.3	Control Abstractions for Least Cost Search	**Dec.-08, 10, May-12, 15, 17, 18,**	Marks 8
7.4	Bounding	..... **Dec.-12, May-14, 18,**	Marks 18
7.5	FIFO Branch and Bound	..... **May-18,**	Marks 8
7.6	LC Branch and Bound	..... **Dec.-13, 18, May-14, 19,**	Marks 12
7.7	0/1 Knapsack Problem	..... **Dec.-10,12,16, 18,** **May-08,09,10,15,18, 19, Oct-16,**	Marks 16
7.8	Traveling Salesperson Problem.....	**May-08,12,14,15, 17, 18, 19,** **Dec.-08,11,12,13,15,16, 18,**	Marks 18
7.9	Multiple Choice Questions		

7.1 Basic Concept

SPPU : May-15, Marks 5

- Branch and bound is a general algorithmic method for finding optimal solutions of various optimization problems.
- Branch and bounding method is a general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower.
- It often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

7.1.1 Comparison between Backtracking and Branch and Bound

Sr. No.	Backtracking	Branch and bound
1.	Solution for backtracking is traced using depth first search.	In this method, it is not necessary to use depth first search for obtaining the solution, even the Breadth first search, best first search can be applied.
2.	Typically decision problems can be solved using backtracking.	Typically optimization problems can be solved using branch and bound.
3.	While finding the solution to the problem bad choices can be made.	It proceeds on better solutions. So there cannot be a bad solution.
4.	The state space tree is searched until the solution is obtained.	The state space tree needs to be searched completely as there may be chances of being an optimum solution any where in state space tree.
5.	Applications of backtracking are - M coloring, Knapsack.	Applications of branch and bound are - Job sequencing, TSP.

Review Question

Q.1 Explain the term : Compare backtracking and branch and bound method.

SPPU : May-15, End Sem, Marks 5

7.2 The Method

SPPU : May-07,09, Dec.-07, Marks 8

- In branch and bound method a state space tree is built and all the children of E nodes (a live node whose children are currently being generated) are generated before any other node can become a live node.

- For exploring new nodes either a **BFS** or **D-search** technique can be used.
- In Branch and bound technique, **BFS-like** state space search will be called **FIFO** (First In First Out) search. This is because the list of live node is First In First Out list (**queue**). On the other hand the **D-search** like state space search will be called **LIFO** search because the list of live node is Last in First out list (**stack**).
- In this method a space tree of possible solutions is generated. Then partitioning (called as branching) is done at each node of the tree. We compute lower bound and upper bound at each node. This computation leads to selection of answer node.
- **Bounding functions** are used to **avoid the generation of subtrees** that do not contain an answer node.

7.2.1 General Algorithm for Branch and Bound

The algorithm for branch and bound method is as given below.

```
Algorithm Branch_Bound()
{
//E is a node pointer;
  E ← new(node);        // This is the root node which is the
                        // dummy start node
  //H is  heap for all the live nodes.
  // H is a min-heap for minimization problems,
  //H is a max-heap for maximization problems.
  while (true)
  {
   if (E is a final leaf) then
      {
            //E is an optimal solution
            write( path from E to the root);
            return;
      }
    Expand(E);
    if (H is empty) then //if no element is present in heap
              {
                    write(" there is no solution");
                    return;
      }
    E ← delete_top(H);
  }
}
```

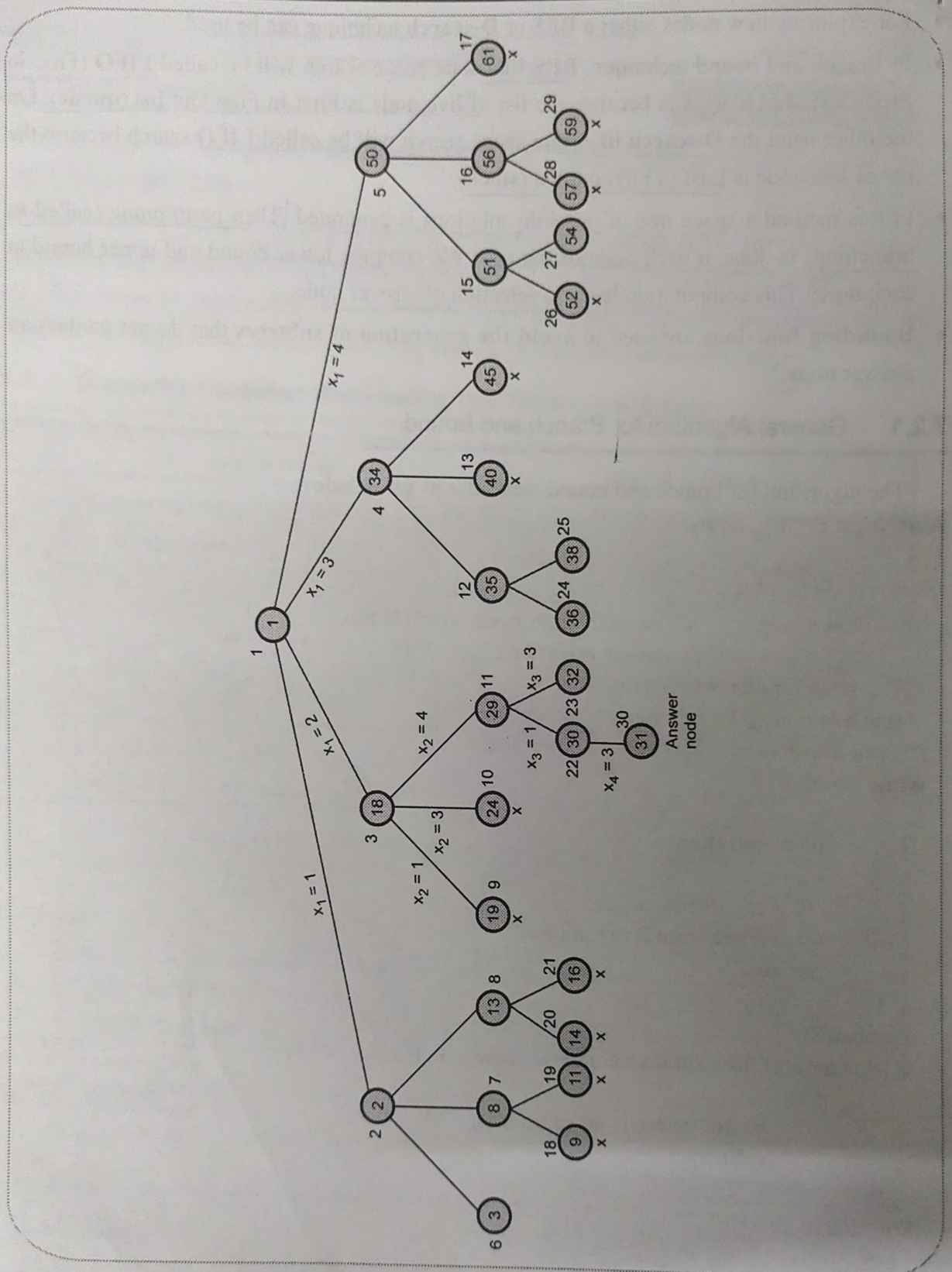


Fig. 7.2.1 Portion of state space tree using 4-queens

TECHNICAL PUBLICATIONS - an up-thrust for knowledge

Following is an algorithm named Expand is for generating state space tree -

```
Algorithm Expand(E)
{
  Generate all the children of E;
  Compute the approximate cost value of each child;
  Insert each child into the heap H;
}
```

For Example

- Consider the 4-queens problem using branch and bound method.
- In branch and bound method a bounding function is associated with each node.
- The node with-**optimum bounding function** becomes an **E-node.** And children of such node get expanded. The state space tree for the same is as given Fig. 7.2.1.
- In Fig. 7.2.1 based on bounding function the nodes will be selected and answer node can be obtained.
- The numbers that are written outside the node indicates the order in which evaluation of tree takes place. (Refer Fig. 7.2.1 on previous page)

Review Questions

Q.1 Write a short note on branch and bound method. **SPPU : May-07, 09, Marks 8**

Q.2 Describe in brief the general strategy used in branch and bound method. **SPPU : Dec.-07, Marks 4**

7.3 Control Abstractions for Least Cost Search

SPPU : Dec.-08, 10, May-12, 15, 17, 18, Marks 8

- In branch and bound method the basic idea is selection of E-node. The selection of E-node should so perfect that we will reach to answer node quickly.
- Using FIFO and LIFO branch and bound method the selection of E-node is very complicated and somewhat blind.
- For speeding up the search process we need to intelligent ranking function for live nodes. Each time, the next E-node is selected on the basis of this ranking function. For this ranking function additional computation (normally called as cost) is needed to reach to answer node from the live node.
- The Least Cost (LC) search is a kind of search in which least cost is involved for reaching to answer node. At each E-node the probability of being an answer node is checked.
- BFS and D-search are special cases of LC search.

- Each time the next E-Node is selected on the basis of the ranking function (smallest c^(x)). Let g^(x) be an estimate of the additional effort needed to reach an answer node from x. Let h(x) to be the cost of reaching x from the root and f(x) to be any non-decreasing function such that

$$c\hat{}(x) = f(h(x)) + g\hat{}(x)$$

- If we set g^(x) = 0 and f(h(x)) to be level of node x then we have BFS.
- If we set f(h(x)) = 0 and g^(x) g^(y) whenever y is a child of x then the search is a D-search.
- An LC search with bounding functions is known as LC Branch and Bound search.
- In LC search, the cost function c(.) can be defined as
  - i) If x is an answer node then c(x) is the cost computed by the path from x to root in the state space tree.
  - ii) If x is not an answer node such that subtree of x node is also not containing the answer node then c(x) = ∞.
  - iii) Otherwise c(x) is equal to the cost of minimum cost answer node in subtree x.
- c^(.) with f(h(x)) = h(x) can be an approximation of c(.).

7.3.1 Control Abstraction for LC Search

The control abstraction for Least cost search is given as follows.

```
Algorithm LC_search()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{
        if( tr is answer node) then
        {
            write(tr);   //output the answer node
            return;
        }
        E←tr             //set the E-node
        repeat
        {
            for(each child x of E) do
            {
                if(x is answer node) then
                {
                    output the path from x to root;
```


TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

```
                    return;
                }
//the new live node x will be added in the list of live nodes
        Add_Node(x);
        x ← parent ← E;  //pointing the path from x to root;
        }
        if(no more live nodes) then
        {
            write("Can not have answer node!!");
            return;
        }
                        // E points to current E-node
E ← Least_cost();       //finds the least cost node to set
                        //next E-node
}until(false);
}
```

In above algorithm if x is in tr then c(x) be the minimum cost answer node in tr. The algorithm uses two functions Least_cost and Add_Node() function to delete or add the live node from the list of live nodes. Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node. The function Least_cost() looks for the next possible E-node.

Review Questions

Q.1 Explain in detail control abstraction for LC search.

SPPU : Dec.-08, Marks 8; Dec.-10, Marks 6

Q.2 What is LC search ? How does it help in finding a solution for branch and bound algorithm?

SPPU : May-12, Marks 8

Q.3 Explain the term : Least cost branch and bound.

SPPU : May-15, End Sem, Marks 5

Q.4 What is LC search ? Explain in detail control abstraction for LC search.

SPPU : May-17, End Sem, Marks 8

Q.5 Explain the term LC search.

SPPU : May-18, End Sem, Marks 2

7.4 Bounding

SPPU : Dec.-12, May-14, 18, Marks 18

- As we know that the bounding functions are used **to avoid the generation of sub trees that do not contain the answer nodes.** In bounding lower bounds and upper bounds are generated at each node.
- A **cost function c^(x)** is such that c^(x) ≤ c(x) is used to provide the lower bounds on solution obtained from any node x.

- Let *upper* is an upper bound on cost of minimum-cost solution. In that case, all the live nodes with c^(x)> upper can be killed.
- At the start the upper is usually set to ∞. After generating the children of current E-node, *upper* can be updated by minimum cost answer node. Each time a new answer node can be obtained.
- Heuristic function : The heuristic is the one, which is used to decide which node is the best in the search tree to expand next. The heuristic function is specific to the problem under consideration. The heuristic function or the evaluation function is denoted by f'. The heuristic function or the evaluation function decides the next move to be made in the search space. The convention used could be a smaller value of the evaluation function leads earlier to the goal state. The next node to be expanded in the search tree will be the one having lowest f ' value.

7.4.1 Job Sequencing with Deadlines

We will consider an example of job sequencing with deadlines to understand the computation of c^(x) and upper. Let us see the problem statement first.

Problem statement :

Let there be n jobs with different processing times. With only one processor the jobs are executed on or before given deadlines. Each job i is given by a tuple (P_i, d_i, t_i) where t_i is the processing time required by job i. If processing of job i is not completed by deadline d_i then penalty P_i will occur. The objective of this problem is to select a subset J such that penalty will be minimum among all possible subsets. Such a J should be optimal subset.

Ex. 7.4.1 *Let n = 4*

Job index	P_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

Then select an optimal subset J with optimal penalty. What will be the penalty corresponding to optimal solution ? **SPPU : May-14, Marks 18**

Sol. : The state space tree can be drawn for proper selection of job i for creating J. There are two ways by which the state space tree can be drawn : Fixed tuple size formulation and variable tuple size formulation. The variable tuple size formulation can be as shown below.

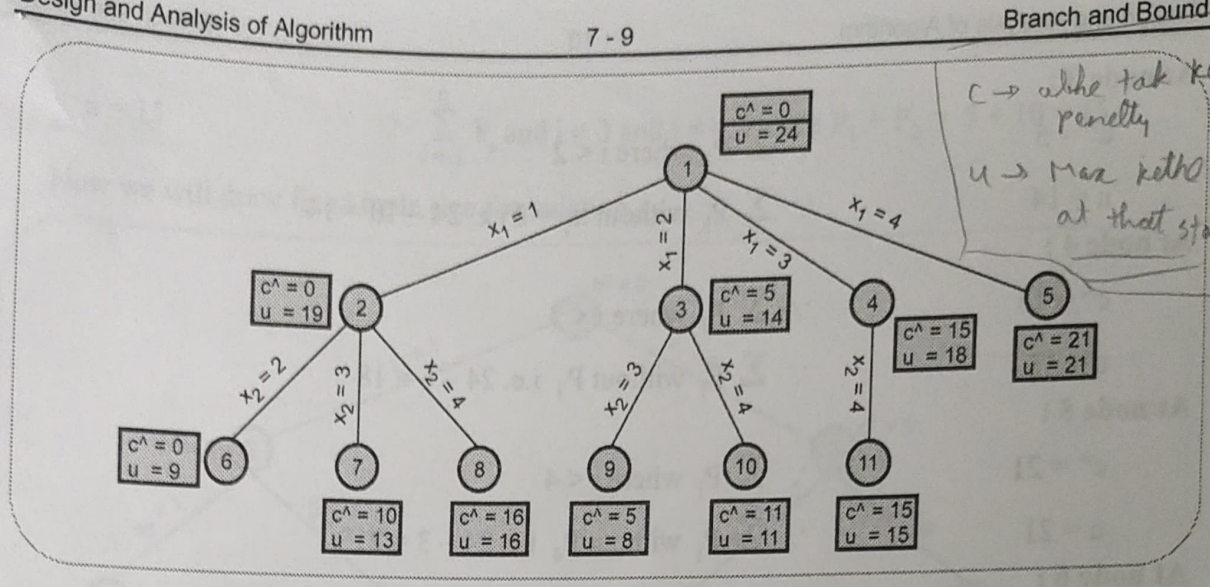


Fig. 7.4.1 State space tree with vaiable tuple size formulation

The function c^(x) can be computed for each node x as

$$c\hat{}(x) = \sum_{i<m \; i \notin S_x} P_i$$

where $m = \max \{i \mid i \in S_x\}$

and S_x be subset of jobs selected for J at node x.

The upper bound can be computed using function u(x). Then,

$$u(x) = \sum_{i \notin S_x} P_i$$

Here u(x) corresponds to the cost of the solution S_x for node x.

For node 1 (root) there are 4 childrens. These children are for selection of either 1 or 2 or 3 or for job 4. Hence $x_1 = 2$ or $x_1 = 3$ or $x_1 = 4$.

For node 2 we have with $x_1 = 1$ and therefore we can select either 2, 3 or 4. Hence $x_2 = 2$ or $x_2 = 3$ or $x_2 = 4$ corresponds to node 6, 7 or 8. Continuing in this fashion, we have drawn the state space tree.

At node 1 :

$c\hat{} = 0$

$u = 24$ $\because \sum_{i=1}^{n} P_i = 5 + 10 + 6 + 3$

At node 2 :

$c\hat{} = 0$ $\because \sum P_i$ where $i < 1 = 0$

$u = 19$ $\because$ the sum of P_i excluding $x_1 = 1$, i.e. $P_1 = 5$.

At node 3 :

$c\hat{} = 5$ $\because \sum P_i$ where $i < 2$

$u = 14$ $\because \sum P_i$ without $x_1 = 2$ i.e. $24 - 10 = 14$

At node 4 :

$c\hat{} = 15$ $\because \sum P_i$ where $i < 3$

$u = 18$ $\because \sum P_i$ without P_3 i.e. $24 - 6 = 18$

At node 5 :

$c\hat{} = 21$ $\because \sum P_i$ where $i < 4$

$u = 21$ $\because \sum P_i$ without P_4 i.e. $24 - 3 = 21$

At node 6 :

$c\hat{} = 0$ $\because \sum_{i<2} P_i$ and not containing $x_1 = 1$ i.e. P_1

$u = 9$ $\because \sum_{i=1}^{n} P_i$ such that $i \neq 2$ and $i \neq 1$. Hence $6 + 3 = 9$

At node 7 :

$c\hat{} = 10$ $\because \sum_{i<3} P_i$ where $i \neq 1$ and $i \neq 3$. Hence $P_2 = 10$.

$u = 13$ $\because \sum_{i=1}^{n} P_i$ such that $i \neq 1$ and $i \neq 3$. Hence $P_2+P_4 = 13$

At node 8 :

$c\hat{} = 16$ $\because \sum_{i<4}^{n} P_i$ and $i \neq 1$ and $i \neq 4$. Hence $P_2 + P_3 = 16$.

$u = 16$ $\because \sum_{i=1}^{n} P_i$ and $i \neq 1$ and $i \neq 4$. Hence $P_2 + P_4 = 16$

At node 9 :

$c\hat{} = 5$ $\because \sum_{i<3}^{n} P_i$ but $i \neq 2$. Hence $P_1 = 5$.

$u = 8$ $\because \sum_{i=1}^{n} P_i$ but $i \neq 2$ and $i \neq 3$. That is $P_1 + P_4 = 8$

At node 10 :

$c\hat{} = 11$ $\because \sum_{i<4}^{n} P_i$ but $i \neq 2$. Hence $P_1 + P_3 = 5 + 6 = 11$.

$u = 11$

At node 11 :

$c\hat{} = 15$ $\because \sum_{i<4}^{n} P_i$ and $i \neq 3$. Hence $P_1 + P_2 = 5 + 10 = 15$

$u = 15$ $\because \sum_{i=1}^{n} P_i$ and $i \neq 3$ and $i \neq 4$. Hence $P_1 + P_2 = 5 + 10 = 15$

Now we will draw fixed tuple size formulation.

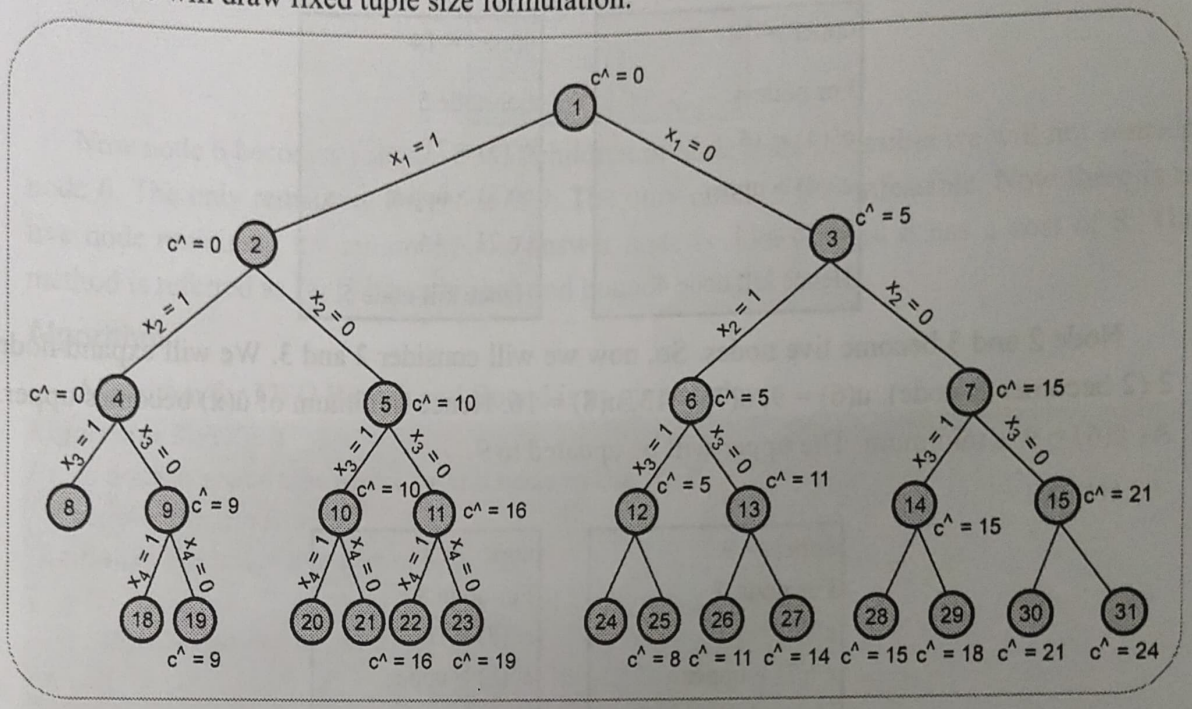


Fig. 7.4.2 State space tree for fixed tuple size formulation

In fixed tuple size formulation. $x_i = 1$ means job i is selected $x_i = 0$ means job i is not selected for formation of J (the optimal subset). Initially, c^ = 0 for node 1 as no job is selected. For node 3, it indicates omission of job 1. Hence, penalty is 5. For node 5 there is omission of job 2. Hence c^ = penalty = 10. For node 7 there is omision of job 1 and job 2. Hence c^=15. For node 13 there is omission of job 1 and job 3. Hence penalty is 11. Therefore c^ = 11. Continuing in this fashion c^ are computed.

Review Questions

Q.1 Explain the term :

a) Branch and bound b) Bounding function

c) Various searching techniques in branch and bound.

d) Heuristic function

SPPU : Dec.-12, Marks 8

Q. 2 Explain the terms : Bounding function.

SPPU : May-18, End Sem. Marks 2

7.5 FIFO Branch and Bound

SPPU : May18, Marks 8

To understand FIFO branch and bound, we will consider the variable tuple size formulation. The state space tree can be drawn as below. For job sequencing problem as -

For node u = 24, the upper is now 24. Then nodes 2, 3, 4 and 5 are generated u(2) = 19, u(3) = 14, u(4) = 18, u(5) = 21. The minimum value of u(x) will be upper. Hence upper will be updated to 14.

upper = 14	upper = 14
For node 4	For node 5
c^(4) = 15	c^(5) = 21
c^(4) > upper	c^(5) > upper
i.e 15 > 14	i.e 21 > 14
Hence kill node 4.	Hence kill node 5.

Node 2 and 3 become live nodes. So, now we will consider 2 and 3. We will expand node 2 (2 becomes E-node). u(6) = 9, u(7) = 13, u(8) = 16. Hence minimum of u(x) becomes upper. As u(6) = 9 is minimum. The upper will be updated to 9.

Hence,

upper = 9	upper = 9
For node 7	For node 8
c^(7) = 10	c^(8) = 16
c^(7) > upper	c^(8) > upper
Hence kill node 7.	Hence node 8 is not considered

The live node 3 being live node becomes E-node now. Now children 9 and 10 are generated. u(9) = 8 and u(10) = 11. Hence upper is updated to 8.

Fig. 7.5.1 Variable tuple sized formulation

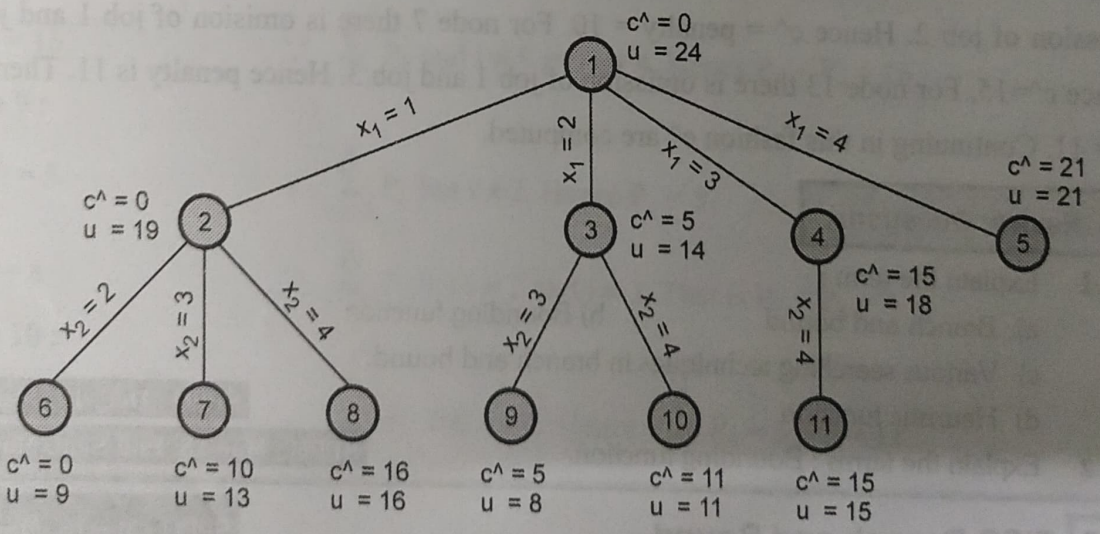


Fig. 7.5.1 Variable tuple sized formulation

Hence,

upper = 8
For node 10
c^(10) = 11
c^(10) > upper
Hence kill node 10.

Now node 6 becomes E-node. But as children of node 6 are infeasible we will not consider node 6. The only remaining live node is 9. The only child of 9 is infeasible. Now there is no live node remaining the minimum cost answer node is node 9. And it has a cost of 8. This method is referred as FIFO based branch and bound.

Algorithm

Algorithm for FIFO Branch and Bound is as given below

```
Algorithm FIFOBB()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{
    if( tr is answer node) then
    {
        u:=min(cost[tr],(upper(tr)+E))
        write(tr);    //output the answer node
        return;
    }
    E←tr          //set the E-node
    repeat
    {
    for(each child x of E) do
    {
        if(x is answer node) then
        {
            output the path from x to root;
            return;
        } //the new live node x will be added in the list of          //live nodes
            Insert_Q(x);
            x.parent ←E;       //pointing the path from x to root;
        if((x is answer node) AND cost(x)<u) then
        {
            u←min(cost[tr],(upper(tr)+E))
        }
    }
```

```
    if(no more live nodes) then
    {
        write("No more Live nodes now");
        write("least Cost is =",u);
        return;
    }
    E← Del_Q();        // E points to current E-node
                       // deletes the least cost node from Q
                       // to set next E-node
    } until(false);
}
```

In above algorithm if x is in tr then c(x) be the minimum cost answer node in *tr*. The computation of u value is done. We always choose the minimum values of cost of the children generated from the current E node.

The algorithm uses two functions Del_Q and () and Insert_Q() function to delete or add the live node from the list of live nodes stored in queue. Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node.

Review Questions

Q.1 Present a program schema for a FIFO branch and bound search for a least cost answer node.

Q.2 Write an algorithm for FIFO branch and bound. **SPPU : May-18, End Sem. Marks 8**

7.6 LC Branch and Bound

SPPU : Dec.-13, 18, May-14, 19, Marks 12

In LC branch and bound method. For node 1 upper = 24. Now, node 1 becomes an E-node. Then node 2, 3, 4 and 5 are generated. The upper is now 14. As c^(4) > upper and c^5 > upper. Hence node 4 and 5 are killed. For node 2 and 3 the cost c^(2) is 0, node 2 becomes an E-node. Hence children node 6, 7 and 8 are generated. The upper is now 9 (because u(6) = 9). Node 7 and 8 are killed. Node 6 is selected as it has minimum cost. But both the children of node 6 are infeasible. So kill node 6. Now, node 3 becomes E-node. Then node 9 and 10 are generated. The upper = 8 now, since c^(10) > upper we will kill node 10. Now only remaining node is 9. Next E-node becomes node 9. Its only child is infeasible. As there are no live nodes remaining, we will terminate search with node 9 as an answer node. The principle idea in LC search is choosing of minimum cost node each time.

This method of LC based branch and bound with appropriate c^(.) and u(.) is called as LCBB (LIFOBB).

Algorithm

Algorithm for LIFO Branch and Bound is as given below

```
Algorithm LIFOBB()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{
        if( tr is answer node) then
        {
            u←min(cost[tr],(upper(tr)+E))
            write(tr);    //output the answer node
            return;
        }
        E←tr          //set the E-node
        repeat
        {
            for(each child x of E) do
            {
                if(x is answer node) then
                {
                    output the path from x to root;
                    return;
                } //the new live node x will be added in the list of live nodes
                PUSH(x);
                x.parent← E;       //pointing the path from x to root;
                if(  (x is answer node) AND cost(x)<u) then
                {
                    u←min(cost[tr],(upper(tr)+E))
                }
            }
            if(no more live nodes) then
            {
                write("No more Live nodes now");
                write("least Cost is =",u);
                return;
            }      // E points to current E-node
            E ←POP();  // deletes the least cost node from stack to set next E-node
        } until(false);
}
```

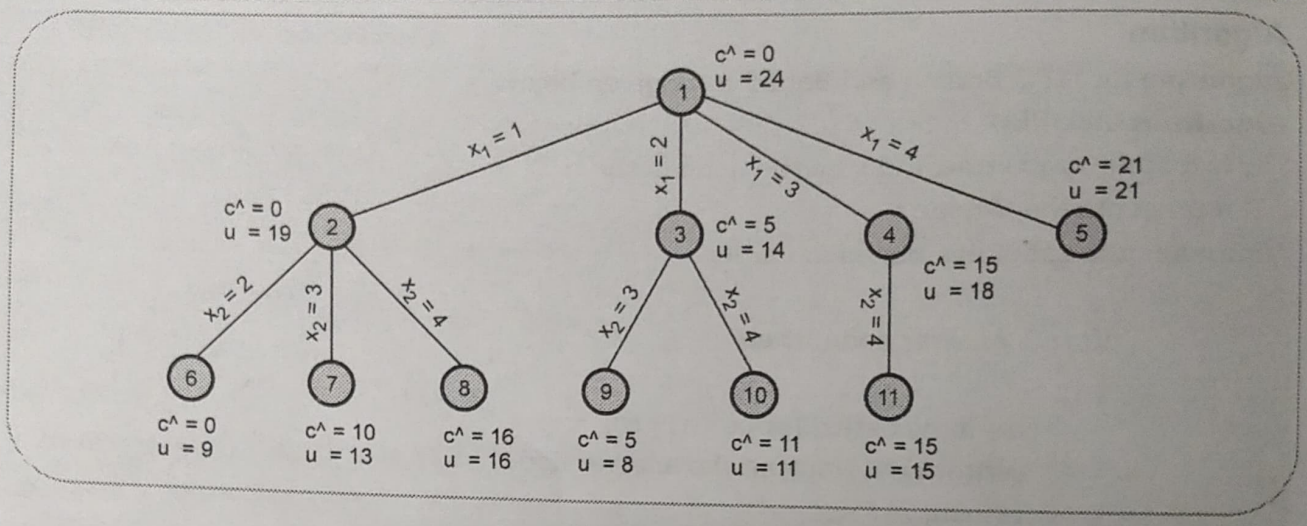


Fig. 7.6.1

In above algorithm if x is in tr then c(x) be the minimum cost answer node in tr. The computation of u value is done. We always choose the minimum values of cost of the children generated from the current E node.

The algorithm uses two functions POP () and PUSH() function to delete or add the live node from the list of live nodes stored in stack. Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node.

Review Questions

Q.1 Write a program schema for a LIFO branch and bound search for Least-cost answer node.

Q.2 Explain the terms :

Branch and Bound, LC, LIFO and Bounding function. How are LIFO and LC techniques different ?

SPPU : Dec.-13, Marks 9

Q.3 Explain for Branch and Bound -

i) LIFO search ii) FIFO search iii) LC search

SPPU : May-14, Marks 12

Q.4 What is least cost search ? Explain in detail control abstraction for LC search.

SPPU : Dec.-18, End Sem. Marks 8

Q.5 Write an algorithm for least cost (LC) branch and bound.

SPPU : May-19, End Sem. Marck 5

7.7 0/1 Knapsack Problem

SPPU : Dec.-10,12,16, 18, May-08,09,10,15,18, 19, Oct-16, Marks 8

Problem statement :

The 0/1 Knapsack problem states that - There are 'n' objects given and capacity of Knapsack is 'm'. Then select some objects to fill The knapsack in such a way that it should not

exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is a maximization problem. That means we will always seek for maximum $P_i x_i$ (where P_i represents profit of object x_i). We can also get $\sum P_i x_i$ maximum iff $-\sum P_i x_i$ is minimum.

$$\text{Minimize profit} - \sum_{i=1}^{n} P_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} W_i x_i$$

Such that $\sum P_i x_i \leq m$ and

$x_i = 0$ or 1 where $1 \leq i \leq n$

We will discuss the branch and bound strategy for 0/1 Knapsack problem using fixed tuple size formulation. We will design the state space tree and compute c^(.) and u(.) where c^(x) represents the approximate cost used for computing the least cost c(x). Clearly u(x) denotes the upper bound. As we know upper bound is used to kill those nodes in the state space tree which can not lead to the answer node.

Let, x be the node at level j. Then we will draw the state space tree for fixed tuple formulation having levels $1 \leq j \leq n + 1$.

Then we need to compute c^(x) and u(x). Such that $c^(x) \leq c(x) \leq u(x)$ for every node.

Algorithm

The algorithm for computing c^(x) is as given below.

```
Algorithm C_Bound(total_profit,total_wt,k)
//total_profit denotes the current total profit
//total_wt denotes the current total weight
// k is the index of last removed object
//w[i] represents the weight of object i
//p[i] represents the profit of object i
//m is the weight capacity of knapsack
{
    pt←total_profit;
    wt←total_wt;
    for(i←k+1 to n) do
    {
        wt←wt+w[i];
        if(wt<m) then pt←pt+p[i];
        else return (pt+(1 – (wt-m)/w[i])*p[i]);
    }
    return pt;
}
```

The algorithm for computing u(x) is as given below

```
Algorithm U_Bound(total_profit,total_wt,k,m)
//total_profit denotes the current total profit
//total_wt denotes the current total weight
// k is the index of last removed object
//w[i] represents the weight of object i
//p[i] represents the profit of object i
//m is the weight capacity of knapsack
{
      pt←total_profit;
      wt←total_wt;
      for(i←k+1 to n) do
      {
            if(wt+w[i]<=m) then
            {
                  pt←pt – p[i];
                  wt←wt+w[i];
            }
      }
      return pt;
}
```

7.7.1 LC Branch and Bound Solution

The LC branch and bound solution can be obtained using fixed tuple size formulation.

The steps to be followed for LCBB solution are

1. Draw state space tree.
2. Compute c^(.) and u(.) for each node.
3. If c^(x) > upper kill node x.
4. Otherwise the minimum cost c^(x) becomes E-node. Generate children for E-node.
5. Repeat step 3 and 4 until all the nodes get covered.
6. The minimum cost c^(x) becomes the answer node. Trace the path in backward direction from x to root for solution subset.

Ex. 7.7.1 *Consider Knapsack instance n = 4 with capacity m = 15 such that,*

Object i	P_i	W_i
1	10	2
2	10	4
3	12	6
4	18	9

SPPU : Dec.-16, 18 End Sem, Marks 18; May-17, 18, 19 End Sem, Marks 10

Sol. : Let us design state space tree using fixed tuple size formulation. The computation of c^(x) and u(x) for each node x is done.

In Fig. 7.7.1 at each node a structure is drawn in which computation of c^ and u is given. The tag field is useful for tracing the path.

Using algorithm U_Bound u(x) is computed and using algorithm C_Bound c^(x) is computed.

u(1) can be computed as -

for i = 1, 2 and 3

$$\therefore \quad \sum P_i = -(10 + 10 + 12)$$

$$\therefore \quad u(1) = -32$$

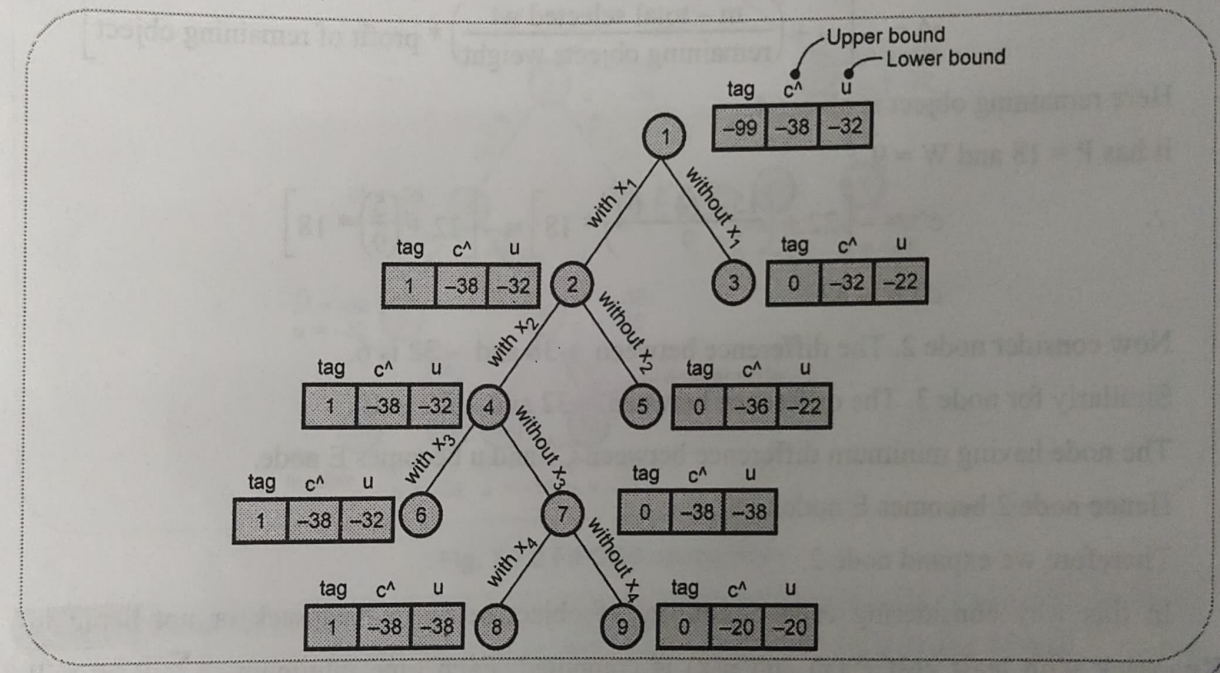


Fig. 7.7.1 LCBB solution space tree

If we select i = 4, then it will exceed capacity of Knapsack.

The computation of c^(1) can be done as follows :

$$c\hat{}1) = -\left[u(1) + \left[\frac{m - \text{Current total weight}}{\text{Actual weight of remaining object}} \right] * \left[\begin{matrix} \text{Actual profit} \\ \text{of remaining object} \end{matrix} \right] \right]$$

$$c\hat{}1) = -\left[32 + \left[\frac{15 - (2 + 4 + 6)}{9} \right] * 18 \right]$$

$$= -\left(32 + \frac{3}{9} * 18 \right)$$

$$c\hat{}1) = -38$$

Now at node 2,

Since the item 1 is selected

The values of c^ and u will be the same i.e. $c^{\wedge} = -38$, $u = -32$

Now at node 3

Do not select item 1.

$\therefore$ u(3) can be computed as

$$-\sum pi = -(10+12) \text{ i.e. with item 2 and 3.}$$

$$= 22$$

$$\therefore \quad u = 22$$

For computation of c^

$$c^{\wedge} = -\left[u + \left(\frac{m - \text{total selected wt}}{\text{remaining objects weight}} \right) * \text{profit of remaining object} \right]$$

Here remaining object is object 4.

It has P = 18 and W = 9

$$\therefore \quad c^{\wedge} = -\left[22 + \left(\frac{15-(4+6)}{9}\right) * 18\right] = -\left[22 + \left(\frac{5}{9}\right) * 18\right]$$

$$c^{\wedge} = -32$$

Now consider node 2. The difference between -38 and -32 is 6.

Similarly for node 3. The difference between -32 and -22 is 10.

The node having minimum difference between $C^{\wedge}$ and u becomes E node.

Hence node 2 becomes E node.

Therefore we expand node 2.

In this way considering each possibility of object being in Knapsack or not being in Knapsack with least cost c^(x) and u(x) is computed. Each time minimum $-\sum P_I x_i$ will become E-node an we will get the answer node as node 8. (Refer Fig. 7.7.1). If we trace the tag field we will get tag(2) – tag (4) – tag (7) – tag (8). i.e. 1101. Hence $x_4 = 1$, $x_3 = 0$, $x_2 = 1$ and $x_1 = 1$. We will select object x_4, x_2 and x_1 to fill up the Knapsack and gain maximum profit.

7.7.2 FIFO Branch and Bound Solution

The space tree with variable tuple size formulation can be drawn and c^(.) and u(.) is computed (We have considered the same Knapsack problem which is discussed in section 7.7.1).

Initially, upper = u(1) = – 32. Then children of node 1 are generated. Node 2 becomes E-node and hence children 4 and 5 are generated. Node 4 and 5 are added in the list of live

nodes. Next, node 3 becomes E-node and children 6 and 7 are generated. As c^(7) > upper we will kill node 7. Hence node 6 will be added in the list of live nodes. Node 4 is E-node and children 8 and 9 are generated. The upper is updated and it is now upper = u(9) = – 38. Nodes 8 and 9 are added in the list of live nodes. Node 5 and 6 becomes the next E-node but as c^(5) > upper and c^(6) > upper, kill nodes 5 and 6. Node 8 becomes next E-node and children 10 and 11 are generated. As node 10 is infeasible do not consider it. c^(11) > upper. Hence kill node 11. Node 9 becomes next E-node and upper = –38. Children 12 and 13 are generated. But c^(13) > upper. So kill node 13. Finally, node 12 becomes an answer node. Therefore, solution is $x_1 = 1, x_2 = 1, x_3 = 0$ and $x_4 = 1$.

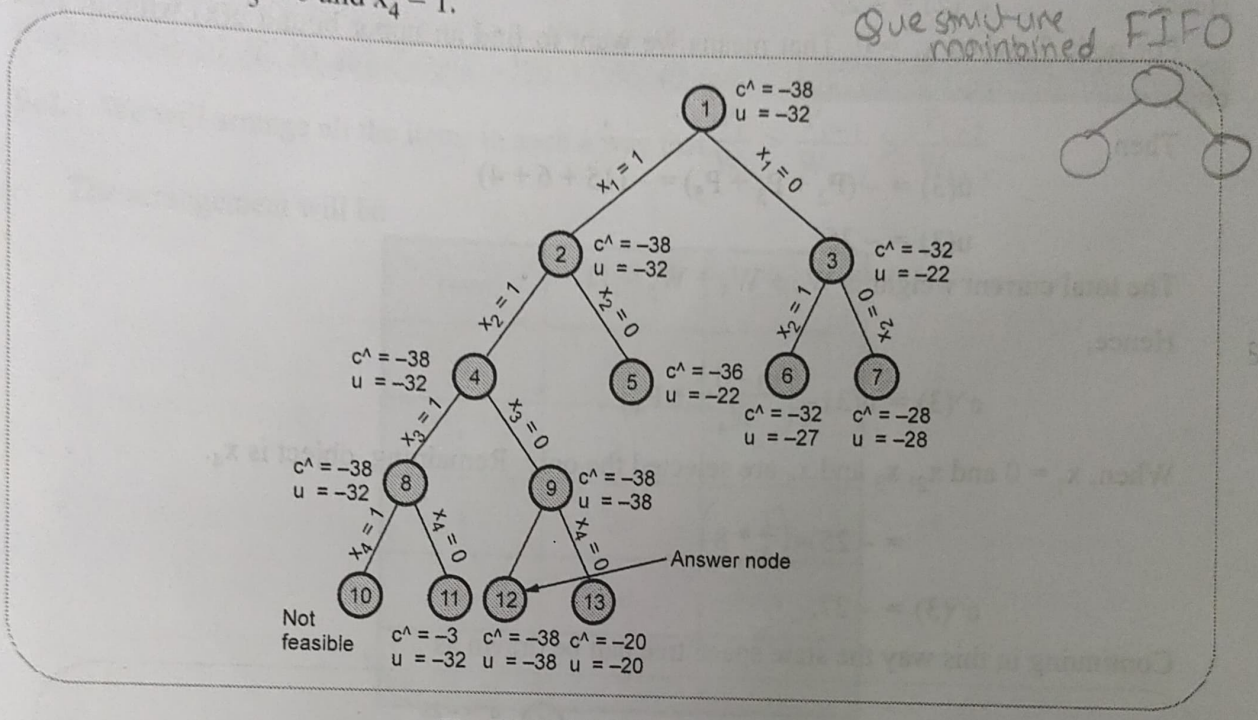


Fig. 7.7.2 FIFOBB space tree

Ex. 7.7.2 *Draw the portion of state space tree generated by LCKNAP for the Knapsack instances : n = 5, $(P_1, P_2, \ldots\ldots P_5) = (10, 15, 6, 8, 4)$, $(W_1, W_2, \ldots\ldots W_5) = (4,6,3,4,2)$ and M = 12.*

Sol. : We will compute u(x) and c^(x) for each node in a state space tree. The fixed tuple size formulation is considered to construct LCBB space tree. Following formulae will be used to compute u(.) and c^(.).

$$u(x) = -\sum P_i x_i$$

$$c^{\wedge}(x) = u(x) - \left[\frac{m - \text{Current total weight}}{\text{Actual weight of remaining object}}\right] * \left[\text{Actual profit of remaining object}\right]$$

For x = 1 i.e for root node.

$$u(x) = u(1) = -\sum P_i \text{ where } i = 1, 2 \text{ and } 5.$$
$$= -(P_1 + P_2 + P_5) = -(10 + 15 + 4) = -29$$

As by selecting P_1, P_2 and P_5 we get the total weight $W_1 + W_2 + W_5$ = Actual size of Knapsack. Hence,

$$c\hat{}(x) = c\hat{}(1) = u(1) - 0$$

$$c\hat{}(1) = -29$$

Again, for node 2 we select $x_1 = 1$. Hence computation of u(2) is,

$$u(2) = -\sum P_i = -(P_1 + P_2 + P_5)$$

$$u(2) = -29$$

Hence, $c\hat{}(2) = -29.$

For node 3, when $x_1 = 0$. That means we want to find an upper bound u(x) without first object.

Then,

$$u(3) = -(P_2 + P_3 + P_5) = -(15 + 6 + 4)$$

$$u(3) = -25$$

The total current weight is $W_2 + W_3 + W_5 = 11$.

Hence,

$$c\hat{}(3) = u(3) - \left(\frac{m - 11}{W_4} * P_4\right)$$

When, $x_1 = 0$ and x_2, x_3 and x_5 are selected the only. Remaining object is x_4.

$$= -25 - \left(\frac{1}{4} * 8\right)$$

$$c\hat{}(3) = -27.$$

Continuing in this way the state space tree can be drawn as :

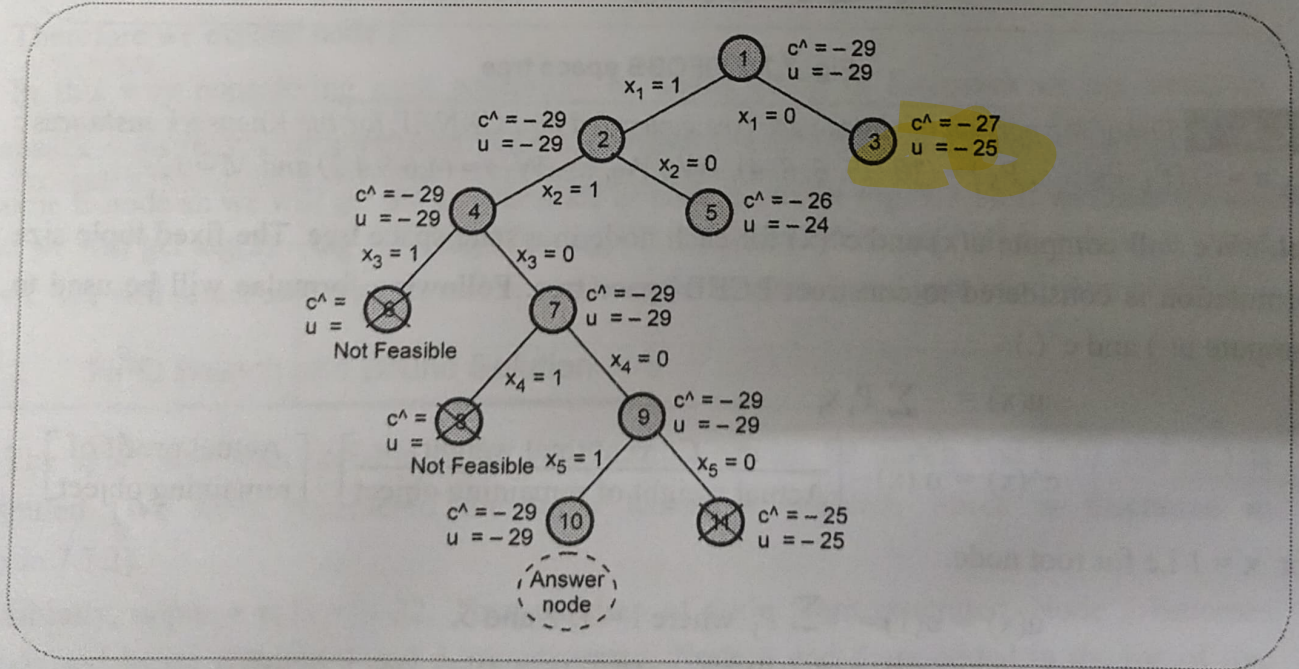


Fig. 7.7.3 Portion of state space tree

In Fig. 7.7.3, the upper is – 29. As node 2 has c^(2) = – 29. Hence node 2 becomes an E node. Therefore generate children of node 2. Node 5 has c^(5) > upper, hence we kill node 5 immediately. Node 4 becomes an E node. The children 6 and 7 of node 4 are generated. Node 6 does not form a feasible solution. Hence node 7 becomes an E-node. Continuing in this fashion the state space tree is generated. We get the solution to this Knapsack instance at node 10. Hence solution is $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$.

Ex. 7.7.3 *Find an optimal solution for the following 0/1 Knapsack instance using branch and bound method :*

Number of objects n = 5, capacity of knapsack m = 100

Profits = (10, 20, 30, 40, 50), weights = (20, 30, 66, 40, 60) **SPPU : Oct-16, (In Sem), Marks 10**

Sol. : We will arrange all the items in such a way that $\frac{P_i}{W_i} > \frac{P_{i+1}}{W_{i+1}} > \frac{P_{i+2}}{W_{i+2}} \cdots$

The arrangement will be

Item	P_i	W_i
4	40	40
5	50	60
2	20	30
1	10	20
3	30	66

- Here we will select item 4, item 5. Now if we select item 2 then total weight > m. Hence by selecting item 4 and item 5 the upper bound for profit is obtained i.e. 40 + 50 = 90 ∴ we will write ub = – 90 for root node, of BB tree. As by choosing item 4 and item 5 the total capacity of knapsack is fulfilled.

Hence $c^{\wedge} = ub = -90$

- Now if we don't select item 4. Then consider selection of item 5 and item 2. Then upper bound for profit is ub = – 70. By this selection the total weight = 60 + 30 = 90. Now we need only weight 10 to reach to maximum capacity of knapsack.

  ∴ Just consider next item for selection i.e. item 1 but we need its fractional weight

  ∴ $c^{\wedge}$ = ub + fractional profit of next item = $(70) + \left(\frac{10}{20} * 10\right) = 75$

  ∴ $c^{\wedge} = -75$

- The partial state space tree will be -

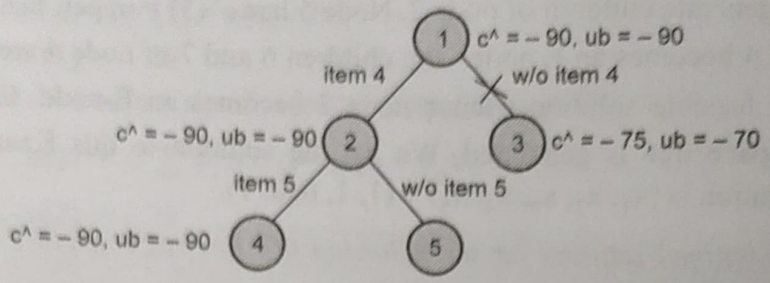


Fig. 7.7.4

We will choose node 2 because it is having less upper bound than node 3. The node 4 and node 5 has the same and ub as that of node 2 because it considers selection of item 4 and item 5.

Now consider node 5 i.e. select item 4 and item 2 ub = – (40 + 20) = – 60. The next remaining weight of item 1 is 20 which can be added as a whole.

$\therefore$ $c^{\wedge} = -$ (ub + weight of item 1) = – (60 + 10)

$\therefore$ $c^{\wedge} = -70$

$\therefore$ At node 5 $c^{\wedge} = -70$, ub = – 60

- But as node 4 is having lesser upper bound than node 5. Select node 4.

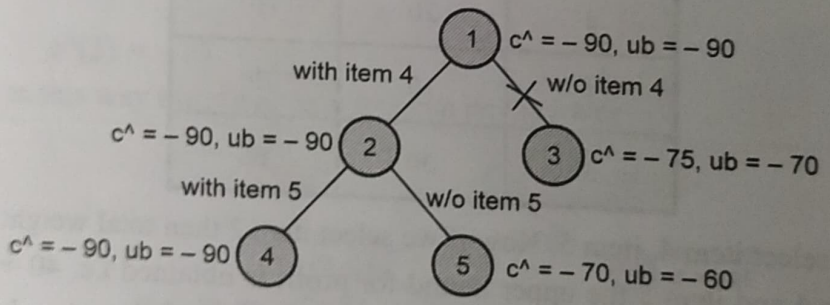


Fig. 7.7.4 (a)

- Now can not further select item 2, item 1 or item 3 because by selecting item 4 and item 5 the knapsack has reached to its capacity.
- Hence solution = {item 4, item 5} with maximum **profit = 90.**

Solution using Alternate Method : Let us solve the Knapsack problem using alternate method of branch and bound.

Ex. 7.7.4 *Consider 0/1 Knapsack instance n = 4 with capacity 10 kg such that*

Item	*Profit (in ₹)*	*Weight (in kg)*
1	40	4
2	42	7
3	20	5
4	12	3

Find maximum profit using first in first out branch and bound (FIFOBB) method. Use fixed size formation for state space tree. **SPPU : May-15, End Sem, Marks 8**

Sol. : To fill the given Knapsack we select the object with some weight and having some profit. This selected object is put in the Knapsack. Thus the Knapsack is filled up with selected objects. Note that the Knapsack's capacity W should not be exceeded. Hence the first item gives best pay off per weight unit. And last one gives the worst pay off per weight unit.

$$v_1/w_1 \geq v_2/w_2 \geq v_3/w_3 \ldots\ldots v_n/w_n$$

- First of all we compute upperbound of the tree.
- We design a state space tree by inclusion or exclusion of some items.

The upper bound can be computed using following formula.

$$ub = v + (W - w)\ (v_{i+1}/w_{i+1})$$

Consider 4 items as -

Item	Weight	Value	Value/Weight
1	4	$ 40	10
2	7	$ 42	6
3	5	$ 20	4
4	3	$ 12	4

W = Capacity of Knapsack
= W = 10

We will first compute the upper bond by using above given formula -

$$ub = v + (W - w)\ (v_{i+1} / w_{i+1})$$

Initially v = 0, w = 0 and $v_{i+1} = v_1 = 40$ and $w_{i+1} = w_1 = 4$. The capacity W = 10.

$\therefore$ $ub = 0 + (10 - 0)\ (40/4) = (10)\ (10)$

$ub = 100\ \$$

Now we will construct a state space tree by selecting different items.

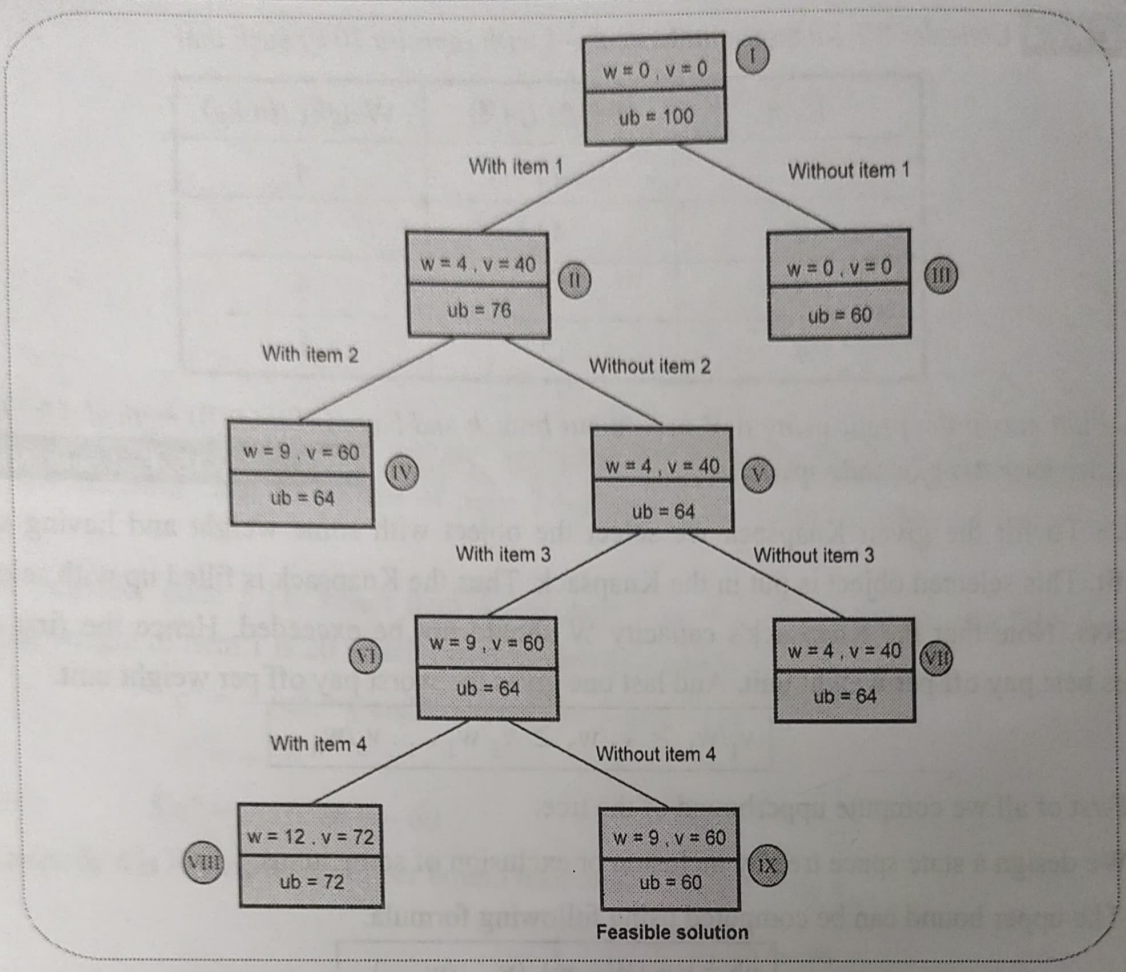


Fig. 7.7.5 State space tree for knapsack problem

Computation at node I

i.e. root of state space tree.

Initially, $w = 0$, $v = 0$ and $v_{i+1}/w_{i+1} = v_1/w_1 = 40/4 = 10$.

The capacity $W = 10$.

$\therefore$ $ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$= 0 + (10 - 0)\,(10)$

$\therefore$ $ub = 100$

Computation at node II

At node II in state space tree we assume the selection of item 1.

$\therefore$ $v = 40$

$w = 4$

The capacity $W = 10$

Now $v_{i+1} / w_{i+1} \rightarrow$ means next item to item 1

i.e. $v_2 / w_2 = 6$

$\therefore$ $ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$= 40 + (10 - 4) * 6$

$= 40 + 6 * 6$

$ub = 76$

Computation at node III

At node III in state space tree we assume that item 1 is not selected.

$\therefore$ $v = 0$

$w = 0$

The capacity $W = 10$

Next to item 1 is item 2

$\therefore$ $v_{i+1}/w_{i+1} \rightarrow$ means 2

i.e. $v_2/w_2 = 6$

$\therefore$ $ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$= 0 + (10 - 0)\,(6)$

$ub = 60$

Computation at node IV

This is a node at which we have selected item 1 and item 2 already

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 = 20/5 = 4$

$\therefore$ $v = 40 + 20 = 60$

$w = 4 + 5 = 9$

The capacity $W = 10$

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 \rightarrow 4$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 60 + (10 - 9) * 4$

$= 60 + 4$

$= 64$

Computation at node V

At this node item 2 is not selected and only item 1 is selected.

$\therefore$ $v = 40$

$w = 4$

$W = 10$

Next item will be item 3

$\therefore \quad v_{i+1} / w_{i+1} = v_3 / w_3 = 4$

$\therefore \quad ub = v + (W - w) * v_{i+1} / w_{i+1}$

$= 40 + (10 - 4) * 4$

$= 40 + 6 * 4$

$ub = 64$

Computation at node VI

Node VI is an instance at which only item 1 and item 3 are selected. And item 2 is not selected.

$v = 40 + 20 = 60$

$w = 4 + 5 = 9$

The capacity W = 10.

The next item would be $v_{i+1} / w_{i+1} \rightarrow$ item 4

$\therefore \quad v_4 / w_4 = 4$

$\therefore \quad ub = v + (W - w)\, v_4/w_4$

$= 60 + (10 - 9) * 4$

$= 60 + 4$

$ub = 64$

Computation at node VII

Node VII is an instant at which item 1 is selected, item 2 and item 3 are not selected.

$\therefore \quad v = 40$

$w = 4$

$W = 10$

The next item being selected is item 4.

$v_{i+1}/w_{i+1} = v_4/w_4 = 4$

$\therefore \quad ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 40 + (10 - 4) * 4$

$= 40 + 24$

$ub = 64$

Computation at node VIII

At node VIII, we consider selection of item 1, item 3, item 4. There is no next item given problem statement.

$\therefore \quad v_{i+1}/w_{i+1} = 0$

$\therefore \quad w = 4 + 5 + 3 = 12 \quad \rightarrow$ But this is exceeding capacity W = 10.

$v = 40 + 20 + 12 = 72$

$W = 10$

$\therefore \quad ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$ub = 72 + (10 - 12) * 0$

But as weight of selected items exceed the capacity W this is **not a feasible solution.**

Computation at node IX

At node IX, we consider selection of item 1 and item 3. There is no next item given.

$\therefore \quad v_{i+1} / w_{i+1} = 0$

$\therefore \quad w = 4 + 5 = 9$

$v = 40 + 20 = 60$

$W = 10$

$\therefore \quad ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 60 + (10 - 9) * 0$

$ub = 60$

The node IX is a node indicating maximum profit of selected items with maximum weight of item = 9 i.e. < capacity of Knapsack (W = 10).

Thus for the given instance of Knapsack's problem we get **{item 1, item 4}** with maximum weight 9 and maximum profit gained = 60 $ as a solution.

Ex. 7.7.5 *Solve the following instance of the Knapsack problem by branch and bound algorithm.*

Item	*Weight*	*Values*
1	*10*	*$ 100*
2	*7*	*$ 63*
3	*8*	*$ 56*
4	*4*	*$ 12*
W = 16		

Sol. : We can solve this problem using the branch and bound method by calculating bounding function by using the following formula -

$$ub = v + (W - w) * v_{i+1}/w_{i+1}$$

We will draw the state space tree, for each node we will compute ub i.e. upper bound value and then expand the tree accordingly.

We will first arrange the items $v_1/w_1 > v_2/w_2 \ldots v_i / w_i$

Item	Weight	Values	v_i / w_i
1	10	$ 100	10
2	7	$ 63	9
3	8	$ 56	7
4	4	$ 12	3

Computation at Node I

No item is selected initially.

Hence $w = 0$, $v = 0$, $W = 16$, $v_{i+1}/w_{i+1} = v_1/w_1 = 10$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$ub = 0 + *16 - 0) * 10$

$ub = 160$

Computation at Node II

Select item 1.

Hence $w = 10$, $v = \$ 100$, $v_{i+1}/w_{i+1} = v_2/w_2 = 9$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 100 + (16 - 10) * 9$

$= 100 + 54$

$ub = 154$

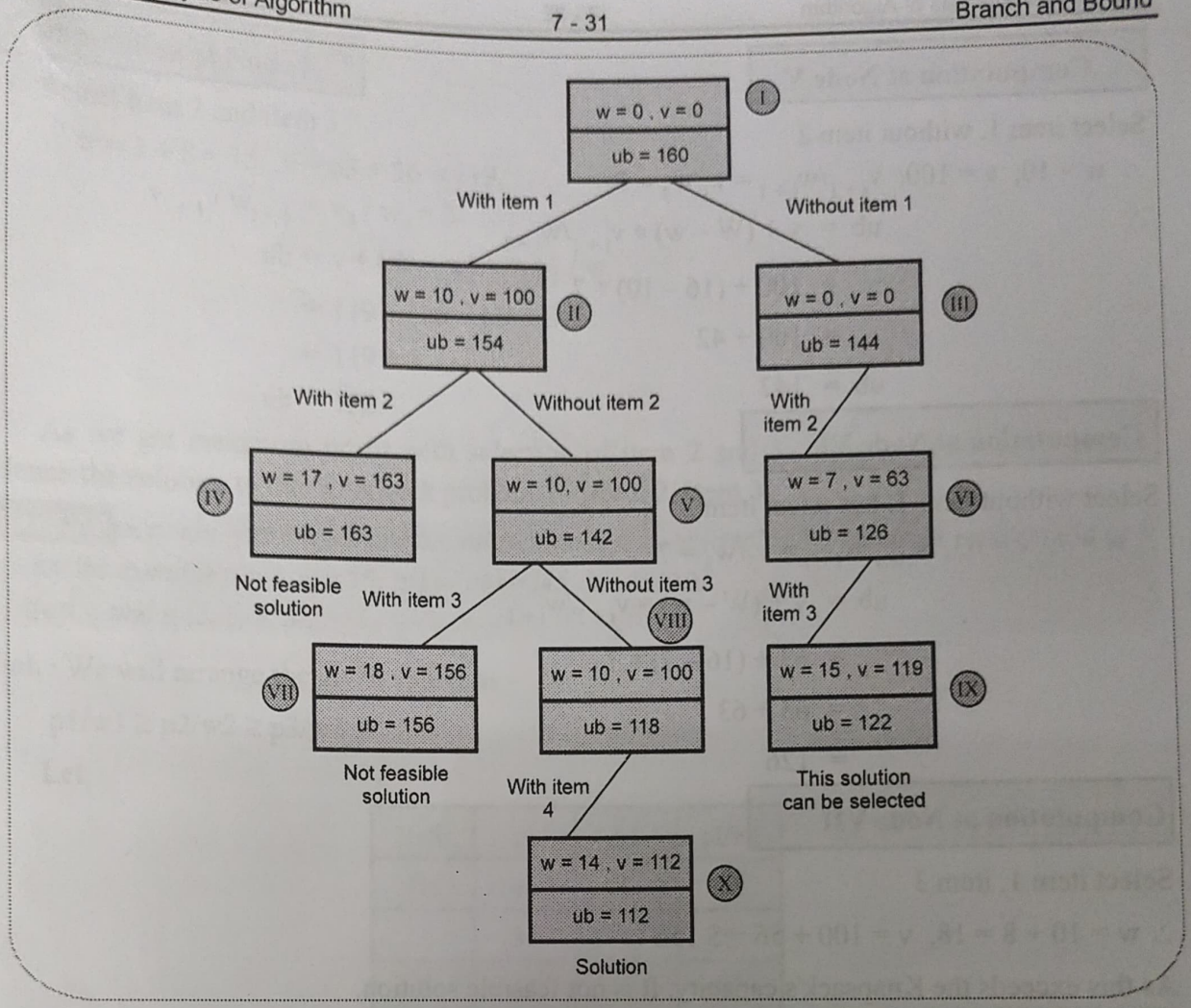


Fig. 7.7.6 State space tree for Knapsack problem

Computation at Node III

Select without item 1.

Hence $w = 0,\ v = 0,\ v_{i+1}/w_{i+1} = v_2/w_2 = 9$

$$\therefore \quad ub = v + (W - w) * v_{i+1}/w_{i+1}$$

$$= 0 + (16 - 0) * 9$$

$$ub = 144$$

Computation at Node IV

Select item 1 and item 2

$\therefore\ w = 10 + 7 = 17,\ v = 100 + 63 = \$\ 163.$

But this exceeds the capacity W.

Computation at Node V

Select item 1, without item 2

$\therefore$ $w = 10$, $v = 100$, $v_{i+1}/w_{i+1} = v_3/w_3 = 7$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 100 + (16 - 10) * 7$

$= 100 + 42$

$ub = 142$

Computation at Node VI

Select without item 1, but select item 2

$\therefore$ $w = 7$, $v = 63$, $v_{i+1}/w_{i+1} = v_3/w_3 = 7$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 63 + (16 - 7) * 7$

$= 63 + 63$

$= 126$

Computation at Node VII

Select item 1, item 3

$\therefore$ $w = 10 + 8 = 18$, $v = 100 + 56 = \$ 156$

As this exceeds the Knapsack's capacity. It is not feasible solution.

Computation at Node VIII

Select item 1, without item 2 and without item 3.

$w = 10$, $v = 100$, $v_{i+1}/w_{i+1} = 3$

$\therefore$ $ub = v + (W' - w) * v_{i+1}/w_{i+1}$

$= 100 + (16 - 10) * 3$

$= 100 + 18$

$= 118$

Computation at Node IX

Select item 1 and item 4

$\therefore$ $w = 10 + 4 = 14$, $v = 100 + 12 = 112$, $v_{i+1}/w_{i+1} = 0$ as there is no next item for selection.

$ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 112 + (16 - 14) * 0$

$= 112$

Computation at Node X

Select item 2 and item 3

$\therefore$ $w = 7 + 8 = 15$, $v = 63 + 56 = 119$,

$v_{i+1} / w_{i+1} = v_4 / w_4 = 3$

$\therefore$ $ub = v + (W - w) * v_{i+1} / w_{i+1}$

$= 119 + (16 - 15) * 3$

$= 119 + 3$

$ub = 122$

As we get maximum profit with selection of item 2 and 3. Also the ub is also greater. Hence the solution to this Knapsack problem is **{item 2, item 3}**.

Ex. 7.7.6 *Explain how branch and bound technique is used to solve 0/1 knapsack problem. Solve it for the example n = 4, m = 15, {p1 ... p4) = {45, 30, 45, 10). {w1 ...w4} = {3, 5, 9, 5}.*

Sol. : We will arrange the given values as -

p1/w1 ≥ p2/w2 ≥ p3/w3...

Let,

Item	pi	wi	pi/wi
1	45	3	15
2	30	5	6
3	45	9	5
4	10	5	2

We will compute the upper bound of the tree using the following formula -

$ub = p + (n - w) (pi + 1 / w i + 1)$

Computation at node I

$p = 0, w = 0$

$ub = 0 + (15 - 0) (45/3) = (15) (15)$

$= 225$

We will construct a state space tree by selecting different items. Refer Fig. 7.7.7

Computation at node II

Select item 1

$\therefore$ $p = 45$ and $w = 3$, the capacity $m = 15$

$\therefore$ $ub = p + (m - w) (pi + 1 / wi + 1)$

$\therefore$ ub = 45 + (15 – 3) (30/5) = 45 + (12) (6) = 45 + 72

ub = 117

Computation at node III

At node III in state space tree we assume that item 1 is not selected.

Item 1 is not selected.

p = 0, w = 0, m = 15.

Item 2 is next to item 1 $\therefore$ pi + 1/wi + 1 = p2/w2

ub = p + (m – w) (pi + 1 /w i + 1) = 0 + (15 – 0) (30 / 5)

ub = 90

Computation at node IV

At node IV we assume that we have selected item 1 and item 2 already.

$\therefore$ p i + 1/w i + 1 = p3/w3 = 5

p = 45 + 30 = 70

w = 3 + 5 = 8 and m = 15

$\therefore$ ub = p + (m – w) (p i + 1/w i + 1) = 70 + (15 – 8) (5) = 70 + (7) (5)

ub = 105

Computation at node V

At node V, item 2 is not selected. Only item 1 is selected. Next item that can be selected is item 3. Hence i + 1 = 3.

$\therefore$ p = 45, w = 3, m = 15

p i + 1/w i + 1 = p3/w3 = 5

$\therefore$ ub = 45 + (15 – 3) = 45 + 12

ub = 57

Computation at node VI

Node VI is an instance at which only item 1 and item 3 are selected and item 2 is not selected.

$\therefore$ p = 45 + 45 = 90

w = 3 + 9 = 12

The capacity m = 15

The next item p i + 1/w i + 1 = p4/w4 = 2

ub = p + (m – w) (pi + 1 / w i + 1)

= 90 + (15 – 12) (2)

The state space tree can be –

$= 90 + 6$

ub = 96

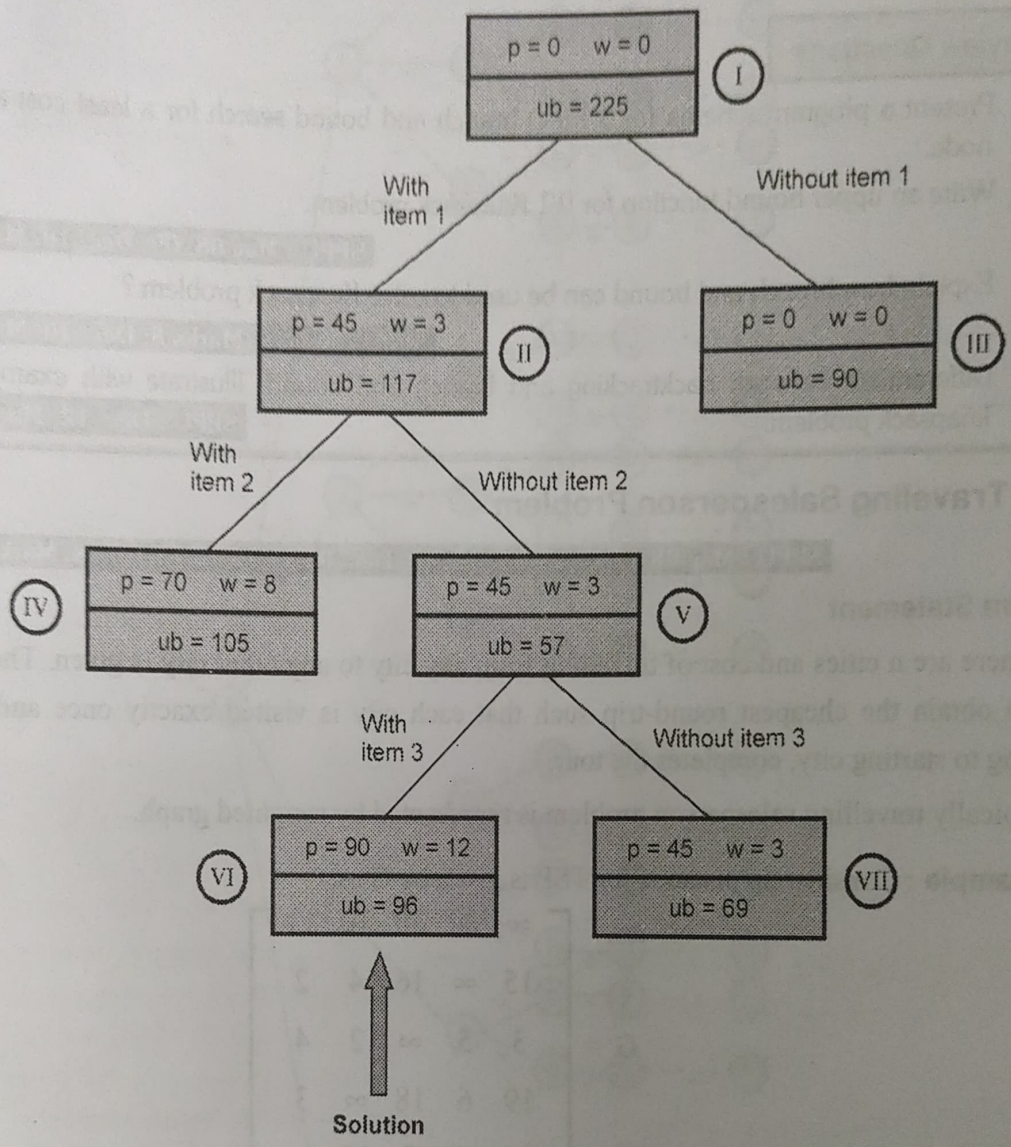


Fig. 7.7.7

Computation at node VII

Node VII is an instance at which item 1 is selected. Item 2 and 3 are not selected.

$p = 45, w = 3$, capacity $m = 15$.

The next item being selected is item 4.

$pi + 1/wi + 1 = p4/w4 = 2$

$ub = p + (m - w)(pi + 1 / w i + 1) = 45 + (15 - 3)(2) = 45 + 24$

ub = 69

Item 1 and **Item 3** are selected because total weight w < m i.e. 12 < 15 and maximum profit i.e. 90 is obtained with this selection.

Review Questions

Q.1 Present a program schema for a FIFO branch and bound search for a least cost answer node.

Q.2 Write an upper bound function for 0/1 Knapsack problem.

SPPU : May-08, 09, Dec.-10, Marks 6

Q.3 Explain how branch and bound can be used to solve Knapsack problem ?

SPPU : May-10, Marks 8; Dec.-10, Marks 6

Q.4 Differentiate between backtracking and branch and bound. Illustrate with example of knapsack problem.

SPPU : Dec.-12, Marks 16

7.8 Traveling Salesperson Problem

SPPU : May-08,12,14,15, 17, 18, 19, Dec.-08,11,12,13,15,16, 18, Marks 18

Problem Statement

If there are n cities and cost of travelling from any city to any other city is given. Then we have to obtain the cheapest round-trip such that each city is visited exactly once and then returning to starting city, completes the tour.

Typically travelling salesperson problem is represented by weighted graph.

For example : Consider an instance for TSP is given by G as,

$$G = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

There n = 5 nodes. Hence we can draw a state space tree with 5 nodes as shown in Fig. 7.8.1. (Refer Fig. 7.8.1 on next page)

Tour(x) is the path that begins at root and reaching to node x in a state space tree and returns to root. In branch and bound strategy cost of each node x is computed. The travelling salesperson problem is solved by choosing the node with optimum cost. Hence,

c(x) = cost of tour (x) where x is a leaf node.

The c^(x) is the approximation cost along the path from the root to x.

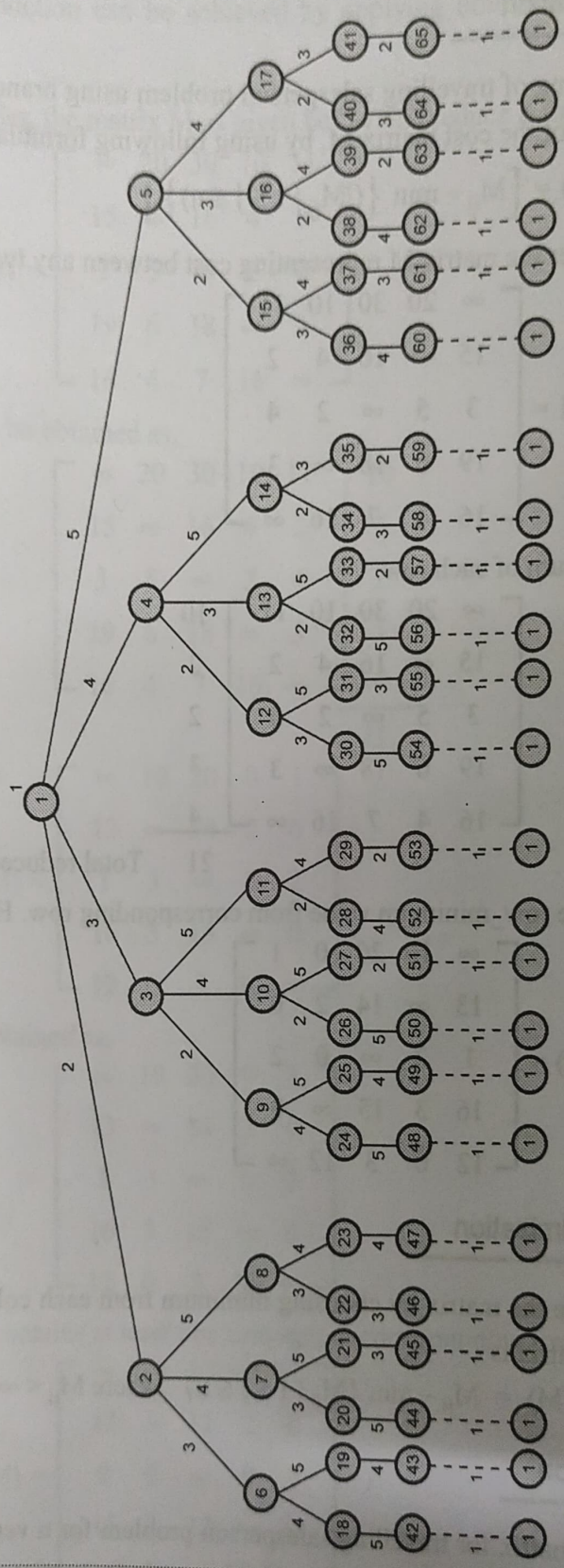


Fig. 7.8.1 State space tree for TSP

7.8.1 Row Minimization

To understand solving of travelling salesperson problem using branch and bound approach we will reduce the cost of the cost matrix M, by using following formula.

$$\text{Red\_Row}(M) = \left[M_{ij} - \min\left\{(M_{ij} \mid 1 \le j \le n)\right\}\right] \quad \text{where } M_{ij} < \infty$$

For example : Consider the matrix M representing cost between any two cities.

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

We will find minimum of each row.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

21 Total reduced cost.

We will subtract the row_minimum value from corresponding row. Hence,

$$\text{Red\_Row}(M) = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

7.8.2 Column Minimization

Now we will reduce the matrix by choosing minimum from each column. The formula for column reduction of matrix is

$$\text{Red\_Col}(M) = M_{ji} - \min\{M_{ji} \mid 1 \le j \le n\} \quad \text{where } M_{ji} < \infty$$

7.8.3 Full Reduction

Let M be the cost matrix for travelling salesperson problem for n vertices then M is called reduced if each row and each column consists of either entirely entries or else contain at least

one zero. The full reduction can be achieved by applying both row_reduction and column reduction.

For example : Consider, the matrix M as given below and reduce it fully.

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Red_Row (M) can be obtained as,

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

21

$$\text{Red\_Row (M)} = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Red_Col(M) be obtained as,

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

If row or column contains at least one zero ignore corresponding row or column.

$$\text{Red\_Col(M)} = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Thus total reduced cost will be

$$= \text{cost (Red\_Row (M))} + \text{cost (Red\_Col(M))}$$

$$= 21 + 4$$

$$= 25$$

Thus
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 1 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \xrightarrow[\text{reduced matrix}]{\text{fully}} \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

7.8.4 Dynamic Reduction

We obtained the total reduced cost as 25. That means all tours in the original graph have a length at least 25.

Using dynamic reduction we can make the choice of edge i → j with optimum cost.

Steps in dynamic reduction technique

1. Draw a state space tree with optimum cost at root node.
2. Obtain the cost of matrix for path i → j by making i^{th} row and j^{th} column entries as ∞. Also set M[i] [j] = ∞.
3. Cost of corresponding node x with path i, j is optimum cost + reduced cost + M[i][j].
4. Set node with minimum cost as E-node and generate its children. Repeat step 1 to 4 for completing tour with optimum cost.

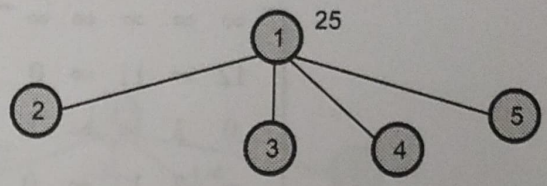
For example :

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

SPPU : May-15,17, 19, End Sem, Dec.-16, 18, End Sem, Marks 18

Fully reduced matrix is,

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$



Optimum cost = 21 + 4 = 25.

Fig. 7.8.2 Portion of state space tree

Consider path 1, 2. Make 1st row and 2nd column ∞. And set M [2] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

ignore
ignore
ignore
ignore
ignore

↓ ↓ ↓ ↓ ↓

ignore ignore ignore ignore ignore

Cost of node 2 is

25 + 0 + 10

↑ ↑ ↑

optimum cost — reduced cost — old value of M[1] [2]

= 35

Consider path 1, 3. Make 1st row = ∞. 3rd column = ∞ and M[3] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

↓

11 ← This is only min value obtained from column

Cost of node 3 is

= 25 + 11 + 17

↑ ↑ ↑

optimum cost — reduced cost — M[1] [3]

= 53

Consider path 1, 4.

The 1st row = 4th column = M[1] [4] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

There is no min value from row or column.

Cost of node 4 is = 25 + 0 + 0 = 25

Consider path 1, 5.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} \\ \rightarrow 2 \\ \\ \rightarrow 3 \\ \\ \end{matrix}$$

Cost of node 5 is = 25 + 5 + 1 = 31

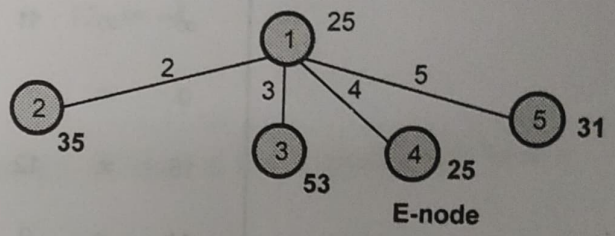


Fig. 7.8.3 Portion of state space tree

Now, as cost of node 4 is optimum, we will set node 4 as E-node and generate its children nodes 6, 7, 8.

Consider path 1, 4, 2 for node 6. Set 1st row, 4th row to ∞. Set 2nd column to ∞. Also M[4] [1] = ∞, M[2] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Cost of node 6 = 25 + M [4, 2]

= 25 + 3 = 28

Consider path 1, 4, 3 for node 7. Set 1st row, 4th row to . Set 4th column, 3rd column to ∞. Set M [4] [1] = M[3] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \rightarrow 2$$

↑

11

Cost of node 7 = 25 + 13 + M [4] [3]

= 25 + 13 + 12

= 50

Consider path 1, 4, 5 for node 8.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \rightarrow 11$$

Cost of node 8 = 25 + 11

= 36

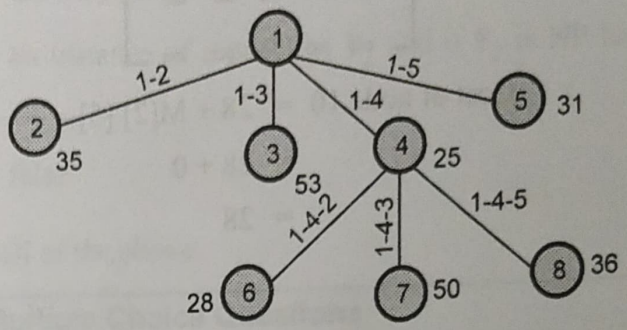


Fig. 7.8.4 Portion of state space tree

Now as cost of **node 6 is minimum**, node 6 becomes an **E-node**. Hence generate children for node 6. Node 9 and 10 are children nodes of node 6.

Consider **path 1, 4, 2, 3 for node 9**. Set 1st row, 4th row, 2nd row to ∞. Set 4th column, 2nd column and 3rd column to ∞.

Set M[1, 4] = M[1, 2] = M[1, 3] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \rightarrow 2$$

↑

11

Cost of node 9

= 28 + 13 + 11

↑ optimum cost ↑ reduced cost ↑ M[1] [3]

= 52

Consider path **1, 4, 2, 5 for node 10.** Set 1st row, 4th row , 2nd row to . Set 4th column, 2nd column and 5th column to ∞. Set M [1, 4] = M[1, 2] = M [1, 5] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

There is no reduced cost

Cost of node 10 = 28 + M[2] [5]
= 28 + 0
= 28

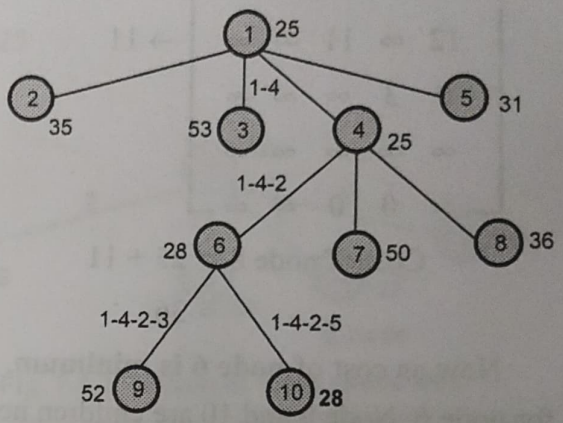


Fig. 7.8.5 Portion of state space tree

Node 10 becomes E-node now.

As for node 10 only child being generated is node 11. We set path as **1, 4, 2, 5, 3**. To complete the tour we return to 1. Hence the state space tree is,

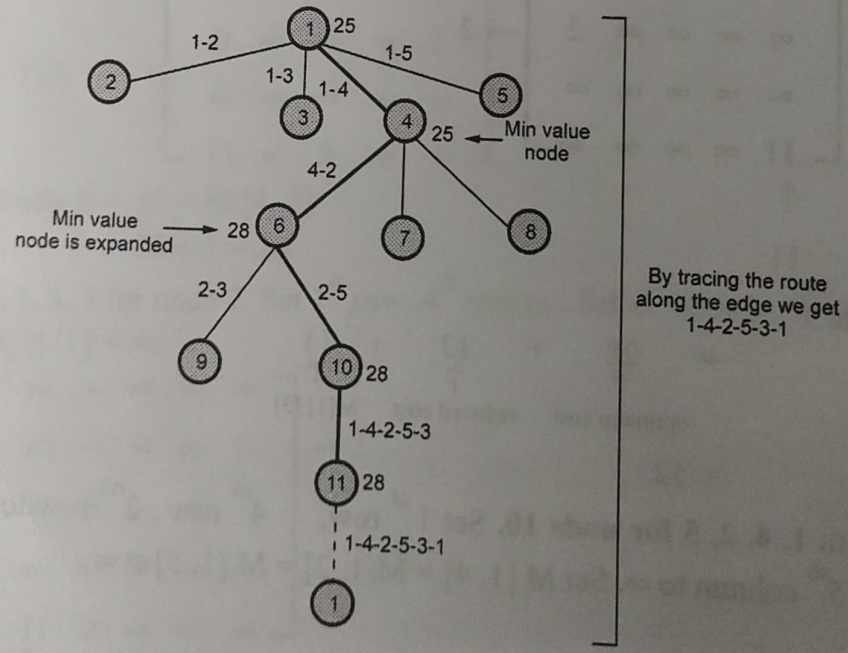


Fig. 7.8.6

Hence the optimum cost of the tour is 28.

Ex. 7.8.1 *Apply the branch and bound algorithm to solve the TSP for the following cost matrix.*

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

Sol. :

Step 1 : We will find the minimum value from each row and subtract the value from corresponding row.

∞	11	10	9	6	→ 6
8	∞	7	3	4	→ 3
8	4	∞	4	8	→ 4
11	10	5	∞	5	→ 5
6	9	5	5	∞	→ 5
					23

⇒

∞	5	4	3	0
5	∞	4	0	1
4	0	∞	0	4
6	5	0	∞	0
1	4	0	0	∞

reduced Matrix

Fig. 7.8.7

Now we will obtain minimuim value from each column. If any column contains 0 then ignore that column and a fully reduced matrix can be obtained.

∞	5	4	3	0
5	∞	4	0	1
4	0	∞	0	4
6	5	0	∞	0
1	4	0	0	∞
↑	↑	↑	↑	↑
1	Ignore	Ignore	Ignore	Ignore

Subtracting ⇒ 1 from 1st column

∞	5	4	3	0
4	∞	4	0	1
3	0	∞	0	4
5	5	0	∞	0
0	4	0	0	∞

Total reduced cost = Total reduced row cost + Total reduced column cost

= 23 + 1

= 24

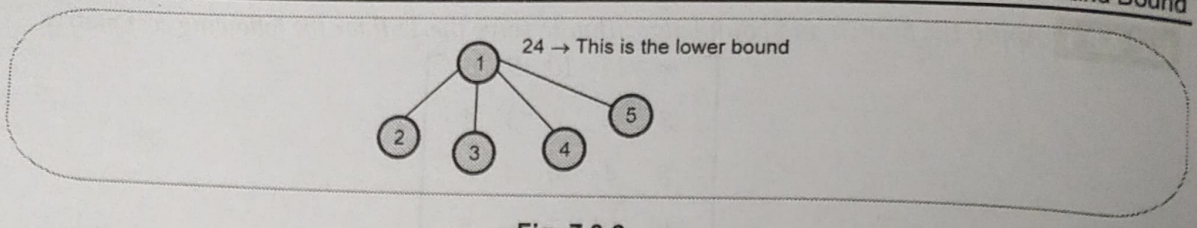


Fig. 7.8.8

Now we will set 24 as the optimum cost.

Step 2 : Now we will consider the paths [1, 2], [1, 3], [1, 4] and [1, 5] of state space tree as given above consider path [1, 2] make 1st row, 2nd column to ∞. Set M [2] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
3	∞	∞	0	4	→ ignore
5	∞	0	∞	0	→ ignore
0	∞	0	0		→ ignore

Now we will find min value from each corresponding column.

∞	∞	∞	∞	∞	
∞	∞	4	0	1	
3	∞	∞	0	4	There is no minimum value
5	∞	0	∞	0	from any column
0	∞	0	0	∞	
↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	

Hence total reduced cost for node 2 is -

= Optimum cost + old value of M [1] [2]

= 24 + 5

= **29**

Consider path (1, 3). Make 1st row, 3rd column to be ∞.

Set M [3] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	∞	0	1	→ ignore
∞	0	∞	0	4	→ ignore
5	5	∞	∞	0	→ ignore
0	4	∞	0		→ ignore
↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	

There is no minimum value from any row and column

TECHNICAL PUBLICATIONS® - an up-thrust for knowledge

Hence total cost for node 3 is

= optimum cost + M [1] [3]

= 24 + 4

= **28**

Consider path (1, 4). Make 1st row, 4th column to be ∞. Set M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ 1
3	0	∞	∞	4	→ ignore
∞	5	0	∞	0	→ ignore
0	4	0	∞	∞	→ ignore

Subtracting ⇒ 1 from 2nd row Subtracting

∞	∞	∞	∞	∞
4	∞	3	∞	0
3	0	∞	∞	4
∞	5	0	∞	0
0	4	0	∞	∞
↑	↑	↑	↑	↑
Ignore	Ignore	Ignore	Ignore	Ignore

The total cost for node 4 is

= Optimum cost + M [1] [4] + Minimum row cost

= 24 + 3 + 1

= **28**

Consider the path (1, 5). Make 1st row, 5th column to be ∞. Set M [5] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	0	∞	→ ignore
3	0	∞	0	∞	→ ignore
5	5	0	∞	∞	→ ignore
∞	4	0	0	∞	→ ignore
↑	↑	↑	↑	↑	
3	Ignore	Ignore	Ignore	Ignore	

Subtracting 3 from 1st column.

∞	∞	∞	∞	∞
1	∞	4	0	∞
0	0	∞	0	∞
2	5	0	∞	∞
∞	4	0	0	∞

The total cost at node 5 is

= Optimum cost + Reduced column cost + Old value M [1] [5]

= 24 + 3 + 0

= 27

The partial state space tree will be -

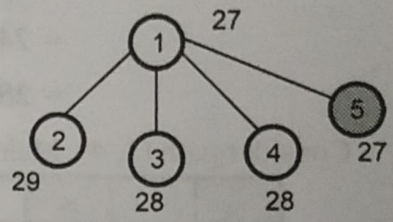


Fig. 7.8.9

The node 5 shows minimum cost. Hence node 5 will be an E node. That means we will select node 5 for expansion.

Step 3 : Now we will consider the paths [1, 5, 2], [1, 5, 3] and [1, 5, 4], of above state space tree do the further computations. Consider path 1, 5, 2. Make 1st row, 5th row and second column as . Set M [5] [1] and M [2] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
3	∞	∞	0	4	→ ignore
5	∞	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore
↑ 3	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	

Now subtracting 3 from 1st column, we get -

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

Hence total cost for node 6 will be -

= Optimum cost at node 5 + Column-reduced cost + M [5] [2]

= 27 + 3 + 4

= 34

Consider path [1, 5, 3]. Make 1st row, 5th row and 3rd column as ∞.

Set M [5] [1] = M [3] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	∞	0	∞	→ ignore
∞	0	∞	0	∞	→ ignore
5	5	∞	∞	∞	→ 5
∞	4	∞	0	∞	→ ignore

Subtracting $\Rightarrow$ 5 from 4th row

∞	∞	∞	∞	∞
4	∞	∞	0	∞
∞	0	∞	0	∞
1	1	∞	∞	∞
∞	4	∞	0	∞
↑	↑	↑	↑	↑
1	Ignore	Ignore	Ignore	Ignore

Subtract 1 from 1st column

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

The total cost for node 7 will be -

= Optimum cost at node 5 + Column reduced cost + M [5] [3]

= 27 + 1 + 0

= 28.

Consider the path [1, 5, 4]. Make first row, fifth row and forth column to be ∞. Set M [5] [1] = M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ 1
3	0	∞	∞	4	→ ignore
∞	5	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore

Subtracting 1 from $\Rightarrow$ 2nd row

∞	∞	∞	∞	∞
3	∞	3	∞	0
3	0	∞	∞	4
∞	5	0	∞	0
∞	∞	∞	∞	∞
↑ 1	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore

Subtracting 3 from 1^{st} column $\Rightarrow$

∞	∞	∞	∞	∞
0	∞	3	∞	0
0	0	∞	∞	4
∞	5	0	∞	0
∞	∞	∞	∞	∞

The total cost for node 8 will be

= Optimum cost at node 5 + Column-reduced + Row-reduced cost + M [5] [4]

= 27 + (1 + 3) + 0

= 31

The partial state space tree will be as shown in Fig. 7.8.10

The node 7 has optimum cost. Hence 7 becomes an E node and it will be expanded further.

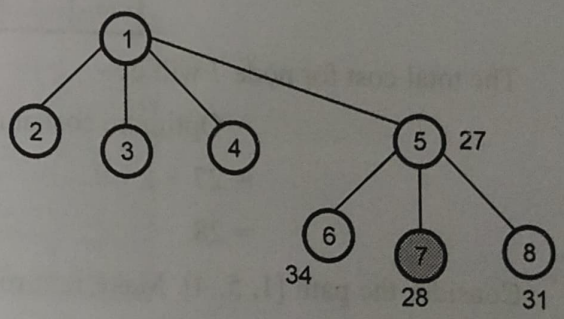


Fig. 7.8.10

Step 4 : Now we will consider paths [1, 5, 3, 2] and [1, 5, 3, 4] for further computations.

Consider path 1, 5, 3, 2 and make 1^{st}, 5^{th}, 3^{rd} row as ∞ and 2^{nd} column as ∞.

Set M [2] [1] = M [3] [1] = M[5][1] = ∞.

The matrix becomes -

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
∞	∞	∞	∞	∞	→ ignore
5	∞	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore
↑ 5	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore	

Subtract 5 from 1st column.

∞	∞	∞	∞	∞
∞	∞	4	0	1
∞	∞	∞	∞	∞
0	∞	0	∞	0
∞	∞	∞	∞	∞

The reduced cost at node 9 [i.e. path [1, 5, 3, 2]] will be

= Optimum cost at node 7 + Column-reduced cost + M [3] [2]

= 28 + 5 + 0

= 33

Consider the path [1, 5, 3, 4]. Make 1st row, 5th row, 3rd row and 4th column as ∞.

Set M [5] [1] = M [3] [1] = M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ ignore
∞	∞	∞	∞	∞	→ ignore
∞	5	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore

Subtracting 1 from second row,

∞	∞	∞	∞	∞
3	∞	3	∞	0
∞	∞	∞	∞	∞
∞	5	0	∞	0
∞	∞	∞	∞	∞
↑ 3	↑ 5			

Subtracting 3 from 1st column and 5 from 2nd column

∞	∞	∞	∞	∞
0	∞	3	∞	0
∞	∞	∞	∞	∞
∞	0	0	∞	0
∞	∞	∞	∞	∞

The total reduced cost at node 10 [i.e. path 1, 5, 3, 4] is

= Optimum cost at node 7 + Row-reduced cost + M [3] [4]

= 28 + (3 + 5) + 0

= 36

The partial state space tree will be –

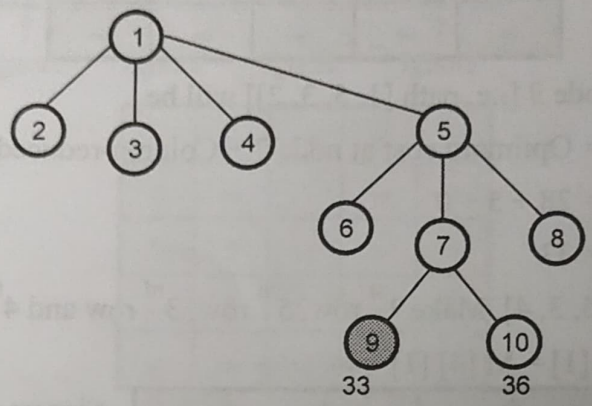


Fig. 7.8.11

Step 5 : Now consider path [1, 5, 3, 2, 4]

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	5	0	∞	0
∞	∞	∞	∞	∞

↓

5 mminvalue

Now subtract 5 from 2nd column

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	0
∞	∞	∞	∞	∞

The reduced cost at node 11 will be -

= Optimum cost at node 9 + column reduced cost + M [2] [4]

= 33 + 5 + 0

= 38

The final state space tree will be –

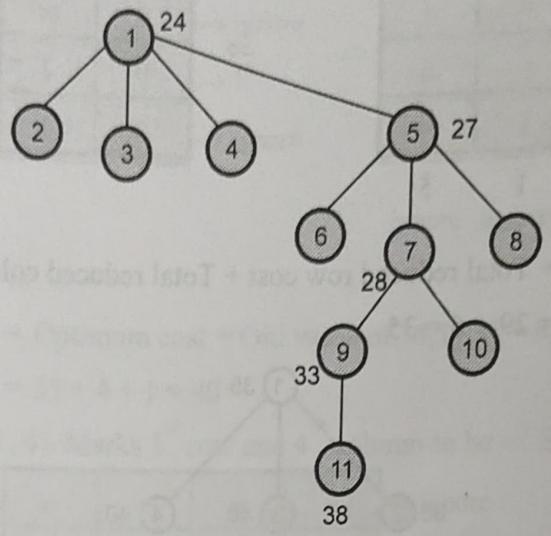
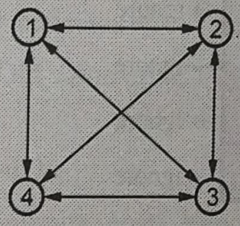


Fig. 7.8.12

The tour with minimum cost is 29. The path will be **1, 5, 3, 2, 4, 1.**

Ex. 7.8.2 *What is travelling salesperson problem ? Find the solution of the following travelling salesperson problem using dynamic approach and branch and bound approach.*

SPPU : May-12, Marks 16



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 0 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Fig. 7.8.13

Sol. :

Step 1 : We will find the minimum value from each row and subtract it from corresponding row.

∞	10	15	20	10
5	∞	9	10	5
6	13	∞	12	6
8	8	9	∞	8
				29

$\Rightarrow$

∞	0	5	10
0	∞	4	5
0	7	∞	6
0	0	1	∞

Reduced Matrix

Now we will obtain min value from each column. If some column contains 0 value then ignore that column. The fully reduced matrix can be obtained as –

∞	0	5	10
0	∞	4	5
0	7	∞	6
0	0	1	∞
↑ ignore	↑ ignore	1	5

⇒

∞	0	4	5
0	∞	3	0
0	7	∞	1
0	0	0	∞

Total reduced cost = Total reduced row cost + Total reduced column cost

= 29 + 6 = 35

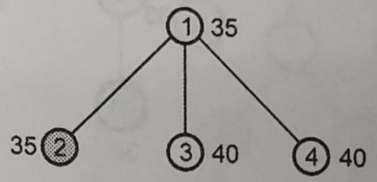


Fig. 7.8.14

The 35 is now set as optimum cost

Step 2 : Now consider path [1, 2]. Mark 1st row and 2nd column as ∞. Also set M[2][1] = ∞. Then we will find min from each row.

∞	∞	∞	∞	→ ignore
∞	∞	3	0	→ ignore
0	∞	∞	1	→ ignore
0	∞	0	∞	→ ignore

Now we will find min from each column

Hence total reduced cost for node 2 is

= Optimum cost + Old value of M[1][2]

= 35 + 0 = 35

∞	∞	∞	∞	
∞	∞	3	0	There is no min value
0	∞	∞	1	from any column
0	∞	0	∞	
↑ ignore	↑ ignore	↑ ignore	↑ ignore	

Now consider path (1, 3). Make 1st row and 3rd column to be ∞. set M[3][1] = ∞.

∞	∞	∞	∞	→ ignore
0	∞	∞	0	→ ignore
∞	7	∞	1	→ 1
0	0	∞	∞	→ ignore

⇒

∞	∞	∞	∞
0	∞	∞	0
∞	6	∞	0
0	0	∞	∞

↑ ignore ↑ ignore

Total cost for node 3 is

= Optimum cost + Old value of M[1][3] + Reduced cost

= 35 + 4 + 1 = 40

Now consider path (1, 4). Marks 1st row and 4th column to be ∞. Set M[4][1] = ∞.

∞	∞	∞	∞	→ ignore
0	∞	3	∞	→ ignore
0	7	∞	∞	→ ignore
∞	0	0	∞	→ ignore

↑ ignore ↑ ignore ↑ ignore ↑ ignore

Optimum cost for node 3 is =

Optimum cost + Old value of M[1][4]

= 35 + 5 = 40

The partial tree will be

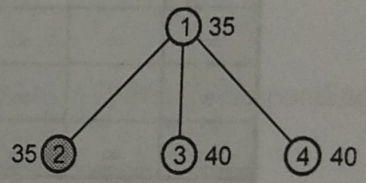


Fig. 7.8.15

Node 2 with minimum cost. Hence it will be expanded further.

Step 3 : Consider paths [1, 2, 3] , [1, 2, 4].

Consider path 1, 2, 3. Make 1st row, 2nd row and 3rd column as ∞. Mark M[2][1] and M[3][1] = ∞.

∞	∞	∞	∞	→ X
∞	∞	∞	∞	→ X
∞	7	∞	1	→ 1
0	0	∞	∞	→ X

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	6	∞	0
0	0	∞	∞

Hence total cost for node 5 will be

= Optimum cost at node 2 + Reduced cost + M[2][3]

= 35 + 1 + 3 = 39

Consider path [1, 2, 4]. Make 1^{st} row, 2^{nd} row and 4^{th} column as ∞.

M[2][1] = M[4][1] = ∞

∞	∞	∞	∞	→ X
∞	∞	∞	∞	→ X
∞	7	∞	0	→ X
0	0	∞	∞	
↑ X	↑ X	↑ X	↑ X	

No minimum value

∴ Optimum cost at node 6 will be

= Optimum cost at node 2 + Reduced cost + M[2][4]

= 35 + 0 + 0 = 35

The partial tree will be

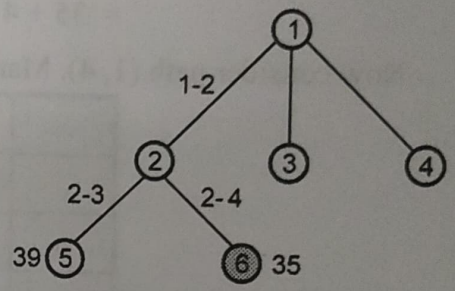


Fig. 7.8.16

Step 4 : Consider path [1, 2, 4, 3].

Mark 1^{st} row = 2^{nd} row = 4^{th} row = 3^{rd} column = ∞

M[3][1] = M[4][1] = M[2][1] = ∞

∞	∞	∞	∞	→ X
∞	∞	∞	∞	→ X
∞	7	∞	1	→ 1
∞	∞	∞	∞	→ X

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	6	∞	0
0	0	∞	∞

Subtract 6 from 2^{nd} column.

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	0	∞	0
∞	∞	∞	∞

Optimum cost = Optimum cost at node 6 + M[4][3]

= 35 + 0 = 35

Final state space tree will be

The path will be 1 - 2 - 4 - 3 - 1.

The tour with minimum cost is 35.

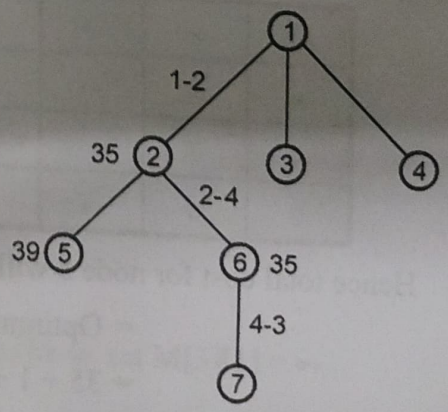


Fig. 7.8.17

Ex. 7.8.3 *Explain branch and bound strategy. Take an example of travelling salesman problem using branch and bound.* **SPPU : Dec.-13, Mark 1**

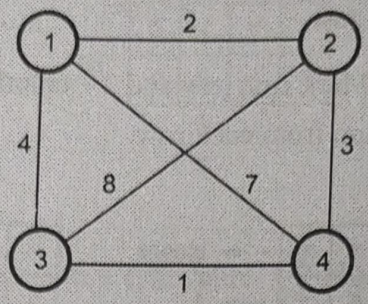


Fig. 7.8.18

Sol. : Branch and bound strategy

Refer section 7.2.1.

We will create adjacency matrix for given graph.

	1	2	3	4
1	∞	2	4	7
2	2	∞	8	3
3	4	8	∞	1
4	7	3	1	∞

Step 1 : We will find the minimum value from each row and subtract it from corresponding row.

∞	2	4	7	2
2	∞	8	3	2
4	8	∞	1	1
7	3	1	∞	1

Total reduced row cost 6

⇒

∞	0	2	5
0	∞	6	1
3	7	∞	0
6	2	0	∞

Now we will obtain minimum value from each column. If some column contains 0 value the ignore that column. The fully reduced matrix can be -

Matrix M [Used in further steps]

∞	0	2	5
0	∞	6	1
3	7	∞	0
6	2	0	∞

← This is reduced matrix

ignore ignore ignore ignore

$\therefore$ Total reduced cost = Total reduced row cost + Total reduced column cost
= 6 + 0 = 6

The 6 is now set as optimum cost.

Step 2 : Consider path [1, 2]. Mark first row and 2nd column as ∞. Also set M[2, 1] = ∞. We will then find minimum from each row.

The reduced matrix will be

∞	∞	∞	∞	→ ignore
∞	∞	6	1	→ 1
3	∞	∞	0	→ ignore
6	∞	0	∞	→ ignore

∞	∞	∞	∞
∞	∞	5	0
3	∞	∞	0
6	∞	0	∞

3 ignore

The reduced matrix will be

∞	∞	∞	∞
∞	∞	5	0
0	∞	∞	0
3	∞	0	∞

$\therefore$ Reduced cost for node 2 = Optimum cost + Old value M [1, 2] + Reduced cost
= 6 + 0 + (1 + 3)
= 10

Step 3 : Consider path (1, 3). Mark first row and 3rd column as ∞. Also set M [3, 1] = ∞.

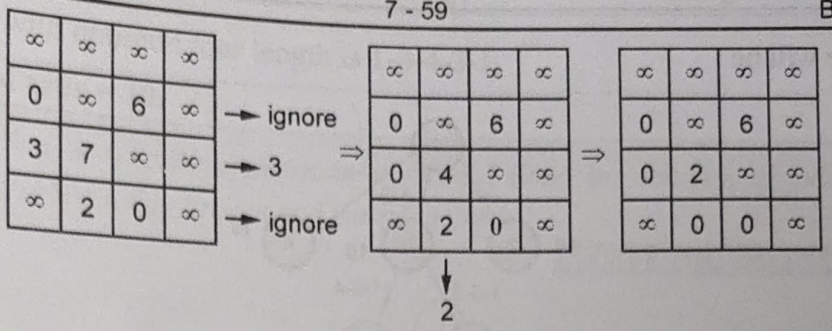
∞	∞	∞	∞	
0	∞	∞	1	ignore ⇒
∞	7	∞	0	
6	2	∞	∞	→ 2

∞	∞	∞	∞
0	∞	∞	1
∞	7	∞	0
4	0	∞	∞

ignore

Cost for node 3 = Optimum cost + Old value M [1, 3] + Reduced cost
= 6 + 2 + (2 + 0)
= 10

Step 4 : Consider path . Mark first row and 4th column as ∞. Set M [4, 1] = ∞



Reduced cost at node 4

= Optimum cost + Old value + Reduced cost = 6 + 5 + (3 + 2) = 16

The partial tree will be

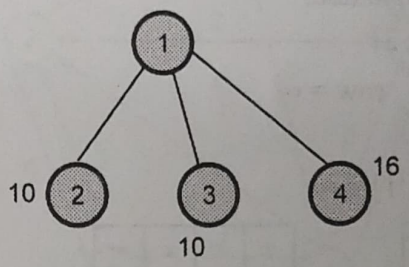
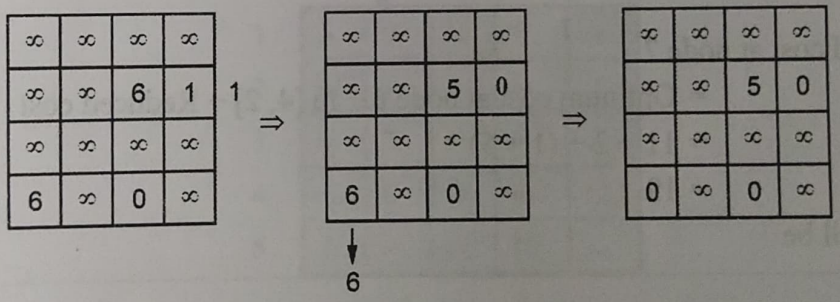


Fig. 7.8.19

We can choose either node 2 or node 3. Let us select node 3. We will expand it further.

Step 5 : Consider path [1, 3, 4] and [1, 3, 2] First consider path [1, 3, 2]. Mark 1st, 3rd row as ∞ and 2nd column as ∞. Also M [2, 1] = M [3, 1] = ∞.



Reduced cost at node 5 will be optimum cost at node 3 + M [3, 2] + Reduced cost

= 10 + 7 + (1 + 6)

= 24

Consider path [1, 3, 4]. Mark row 1 = row 3 = ∞ and column 4 = ∞. M [4, 1] = M [3, 1] = ∞.

From Both row and column. we cannot obtain any minimum value. Hence reduced cost = 0.

∴ Reduced cost at node 6 will be optimum cost at node 3 + m [3, 4] + Reduced cost = 10 + 1+ 0 = 11

∞	∞	∞	∞
0	∞	6	∞
∞	∞	∞	∞
∞	2	0	∞

Partial tree will be

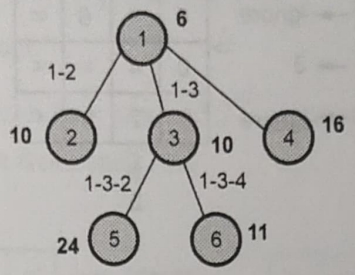


Fig. 7.8.20

Naturally node 6 will be expanded.

Step 6 : Consider path [1, 3, 4, 2]

Mark 1st row = 3rd row = 4th row = ∞

Mark 2nd column = ∞

M [2, 1] = M [4, 1] = M [3, 1] = ∞

∞	∞	∞	∞
∞	∞	6	1
∞	∞	∞	∞
∞	∞	∞	∞

1 ⇒

∞	∞	∞	∞
∞	∞	5	0
∞	∞	∞	∞
∞	∞	∞	∞

5 ⇒

∞	∞	∞	∞
∞	∞	0	0
∞	∞	∞	∞
∞	∞	∞	∞

∴ Reduced cost at node 7

= Optinum cost at node 6 + m [4, 2] + Reduced cost

= 11 + 2 + (1 + 5)

= 19

The tree will be

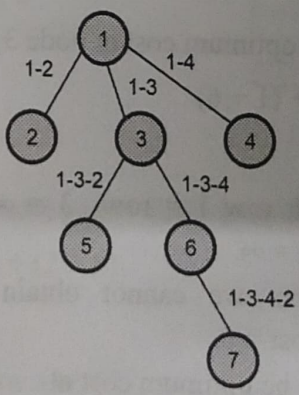


Fig. 7.8.21

Thus path with optinum tour length is **1-3-4-2-1.**

The cost of tour = 10.

Ex. 7.8.4 *What is travelling salesman problem ? Find the solution of following travelling salesman problem using branch and bound method.*

SPPU : Dec.-15, (End Sem), Marks 18

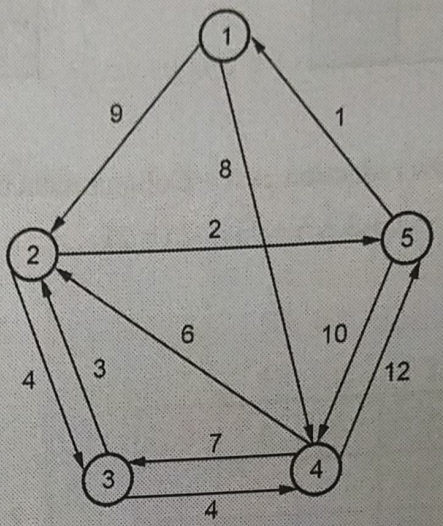


Fig. 7.8.22

Sol. : We will create adjacency matrix for given graph.

	1	2	3	4	5
1	∞	9	∞	8	∞
2	∞	∞	4	∞	2
3	∞	3	∞	4	∞
4	∞	6	7	∞	12
5	1	∞	∞	10	∞

Step 1 : We will find the minimum value from each row and this value is subtracted from corresponding row to get ROW reduced matrix.

	1	2	3	4	5	
1	∞	9	∞	8	∞	8
2	∞	∞	4	∞	2	2
3	∞	3	∞	4	∞	3
4	∞	6	7	∞	12	6
5	1	∞	∞	10	∞	1

Reduce Row →

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

Now we will reduce column by each minimum value from corresponding column.

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

↓ 1

Reduce → Column

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

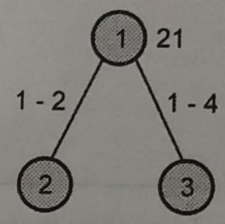
Total reduced cost = Row reduction cost + Column reduction cost

$= (8 + 2 + 3 + 6 + 1) + (1) = 21$

Cost (1) = 21

The reduced matrix is

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



Step 2 : Now we will select path 1–2 : **Node 2.**

Cost of edge 1–2 is M[1, 2] = 1. Set row # 1 as ∞, set column # 2 as set M[2, 1] = ∞. The resultant cost matrix will be

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	∞	∞	1	∞	0	
3	∞	∞	∞	1	∞	→ 1
4	∞	∞	0	∞	6	
5	0	∞	∞	9	∞	

Reduce matrix for row reduction for row # 3

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

There is no column reduction. Hence

$$\text{Cost (2)} = \text{Cost (1)} + M[1, 2] + \text{Reduced cost}$$
$$= 21 + 1 + 1$$
$$\text{Cost (2)} = 23$$

Step 3 : Select path 1–4 : **Node 3**

Cost of edge 1–4 is M[1, 4] = 0 set row # 1 as , set column # 4 as ∞.

Set M [4 ,1] = ∞.

The resultant cost matrix will be

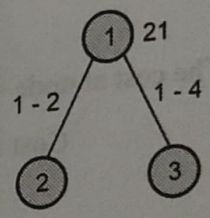
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	1	∞	0
3	∞	0	∞	∞	∞
4	∞	0	0	∞	6
5	0	∞	∞	∞	∞

There no value for row reduction or column reduction. Hence

Cost (3) = Cost (1) + M [1, 4]

$= 21 + 0$

Cost = 21



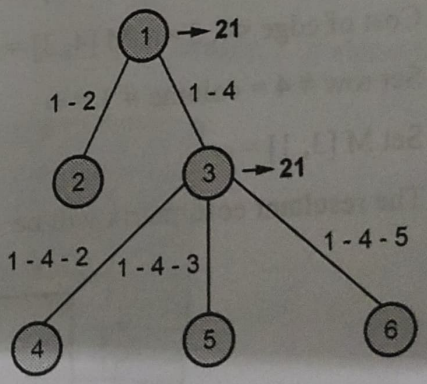
Here

We will choose node 3 for expansion as it has optimum cost.

Step 4 : The vertices possible from vertex 4 are 2, 3 and 5. Now we will start from cost matrix at node 3.

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

Cost (3)



Now we will choose vertex 2 : **Node 4 (1–4–2)**

- Cost of edge <4, 2> = 0
- Set row #4 = ∞, Column # 2 = ∞
- Set M [2, 1] = ∞

The resultant matrix will be

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

↓
1

Reduced cost = 1 (column 3)

The reduced cost matrix is

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

The cost at node 4 is

$$\text{Cost (4)} = \text{Cost (3)} + \text{Reduced cost} + M[4, 2]$$
$$= 21 + 1 + 0$$
$$\text{Cost (4)} = 22$$

Step 5 : Choose vertex 3 : **Node 5** (path 1 – 4 – 3)

- Cost of edge <4, 3> is M [4, 3] = 0
- Set row # 4 = column # 3 = ∞
- Set M [3, 1] = ∞
- The resultant cost matrix will be

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

- There is no reduced column of row value.

The cost at node 5 is

- Cost (5) = Cost (3) + Reduced cost + M [4 , 3]

  = 21 + 0 + 0

- Cost (5) = 21

Step 6 : Choose vertex 5 : **Node 6** (Path 1–4–5)

- Cost of edge <4, 5> is M [4, 5] = 6
- Set row # 4 = column # 5 = ∞
- Set M [5, 1] = ∞
- The resultant matrix is

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

↓
1

- Reduced cost = Column cost = 1

Hence the reduced matrix is obtained by subtracting 1 from column 3 of above matrix

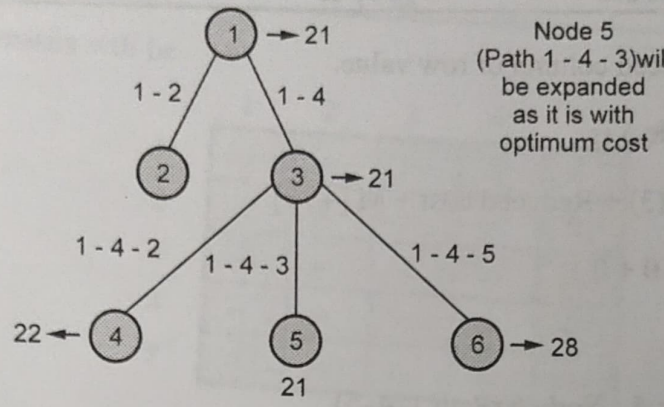
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	0	∞	∞
3	∞	0	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞

The cost at node 6 is

Cost (6) = Cost (3) + Reduced cost + M [4 , 5]

= 21 + 1 + 6

Cost (6) = 28



Step 7 : The vertices possible from vertex 3 is 2. Hence we will start from cost matrix at node 5.

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

Now choose vertex 2 : Node 7 (path : 1–4–3–2)

- Cost of edge <3, 2>, M [3, 2] = 0
- Set row # 3 = column # 2 = ∞
- Set M [2 ,1] = ∞
- The resultant matrix will be

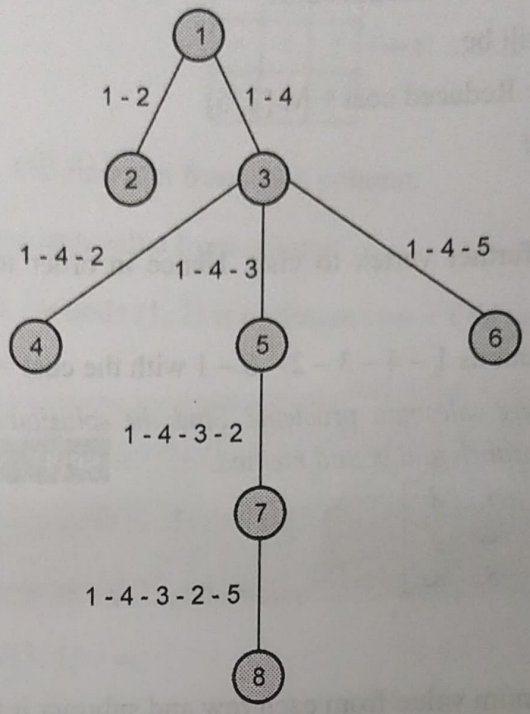
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	0
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	0	∞	∞	∞	∞

- There is no reduced row or column value.
- The cost at node 7 will be
- **Cost (7)** = Cost (5) + Reduced cost + M [3, 2]

  = 21 + 0 + 0

  Cost (7)= 21

Step 8 : The vertex possible from 2 is 5



We will start from cost matrix at node 7

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

Cost (7)

The vertex 5 is chosen for node 8 (path 1 – 4 – 3 – 2 – 5)

- Cost of edge <2, 5>, M [2, 5] = 0
- Set row # 2 = column # 5 = ∞
- Set M [5, 1] = ∞
- The resultant matrix will be

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞

- There is no reduced row or column value
- The cost at node 8 will be
- Cost (8) = cost (7) + Reduced cost + M [2, 5]
  = 21 + 0 +0

Cost (8) = 21

Thus now there is no further vertex to visit. Hence in order to complete the tour, Visit vertex 1 from vertex 5.

Thus we get optimum tour as 1 – 4 – 3 – 2 – 5 – 1 with the cost = (8 + 7 + 3 + 2 + 1) = **21**

Ex. 7.8.5 *What is travelling salesman problem? Find the solution of the following travelling salesman problem using branch and bound method.* **SPPU : May-18, End Sem, Marks 10**

$$\text{Cost Matrix} = \begin{bmatrix} \infty & 4 & 2 \\ 3 & \infty & 4 \\ 1 & 8 & \infty \end{bmatrix}$$

Sol. :

Step 1 : We will find minimum value from each row and subtract it from corresponding row

∞	4	2	2
3	∞	4	3
1	8	∞	1

→

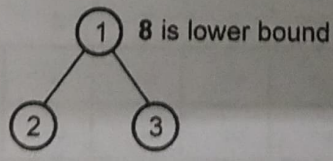
∞	2	0
0	∞	1
0	7	∞

Now we will obtain min value from each column. If some column contains 0, then ignore that column.

The fully reduced matrix is

∞	0	0
0	∞	1
0	5	∞

Total reduced cost = Reduced row cost + Reduced column cost
= (2 + 3 + 1) + (2)
= 8

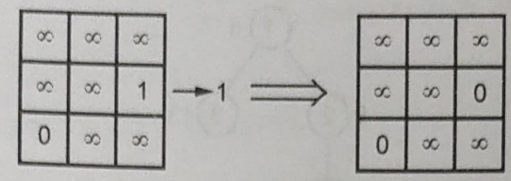


The 8 is now set as optimum cost.

Step 2 : i) Now consider path (1, 2)

Mark 1st row and 2nd column as also set M[2, 1] = ∞

Then we will find min from each row



Now we will find min from each column.

There is no min value from column.

Total cost for node (1, 2) is optimum cost + Old value M[1, 2] + Reduced cost

$$= 8 + 0 + 1$$

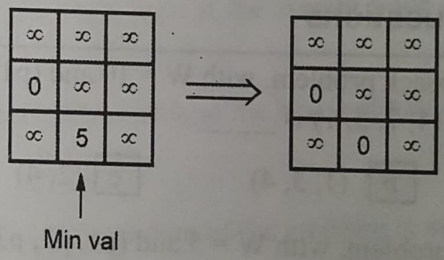
$$= 9$$

ii) Now consider path (1, 3)

Mark 1st row ∞ and 3rd column.

Also set M[3, 1] = ∞.

Then we will find min from each row

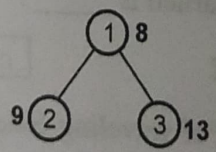


∴ Total cost for node (1, 3) is optimum cost + Old value M[1, 3] + Reduced cost

$$= 8 + 0 + 5$$

$$= 13$$

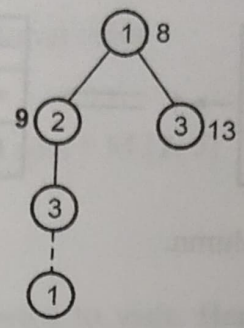
The partial space tree is



As node 2 is with minimum cost node 2 becomes **E-node.** Hence only one child is generated for this node. So the optimum tour length path is **1 – 2 – 3**

TECHNICAL PUBLICATIONS®- an up-thrust for knowledge

The state space tree is



The cost of tour is 9

Review Questions

Q.1 Explain how branch and bound method can be used to solve travelling salesperson problem.

SPPU : May-08, Dec.-08, Marks 6

Q.2 Explain the branch and bound algorithmic strategy for solving the problem take an example of traveling salesman problem using branch and bound.

SPPU : Dec.-11, 12, Marks 10

Q.3 Explain dynamic reduction with all steps with respect to Travelling Salesperson problem.

SPPU : May-14, Marks 6

7.9 Multiple Choice Questions

Q. 1 The solution to the Knapsack problem, with W = 10 and (p1, p2, p3, p4) = (42, 49, 30,4) and (w1, w2, w3, w4) = (4, 7, 5, 1) is _____.

a (1, 3) b (1, 3, 4) c (2, 4) d (2, 3)

Q. 2 Consider the Knapsack problem, with W = 5 and (p1, p2, p3, p4) = (12, 10, 20, 15) and (w1, w2, w3, w4) = (2, 1, 3, 2) The Upper Bound at the root node of a state space tree is _____.

a 30 b 42 c 50 d 37

Q. 3 Consider the Knapsack problem, with W = 5 and (p1, p2, p3, p4)=(12, 10, 20, 15) and (w1, w2, w3, w4)=(2, 1, 3, 2)
The maximum profit that can be earned is _____.

a 30 b 42 c 50 d 37

Q. 4 The worst case time complexity of traveling Salesperson problem using branch and bound is _____.

a O(n^2) b O(2^n)

c O() d O(nlogn)

UNIT - VI

8 Computational Complexity

Syllabus

Non Deterministic algorithms, *The classes : P, NP, NP Complete, NP Hard, Satisfiability problem, Proofs for NP Complete Problems : Clique, Vertex Cover*

Contents

8.1 *The Classes : P, NP, NP, Complete* ..... **Dec. 12, May 08,** ..... *Marks 10*

8.2 *Non-Deterministic Algorithms* **May-07,10,11,14, 18, 19,**

..... **Dec.-06,10,** ..... *Marks 8*

8.3 *The NP Hard Problem* ..... **Dec.-06,07,13, 18, May-14,15,** ... *Marks 16*

8.4 *Concept of Reduction*

8.5 *Satisfiability Problem*

8.6 *Proofs for NP Complete Problems*..... **May-14,15, 18, 19.**

..... **Dec.-09,16, 18,**..... *Marks 8*

8.7 *Multiple Choice Questions*

8.1 The Classes : P, NP, NP, Complete

SPPU – Dec. 12, May 08, Marks 10

8.1.1 The Classes P and NP

There are two groups in which a problem can be classified. The first group consists of the problems that can be solved in polynomial time. For example : Searching of an element from the list O(logn), sorting of elements O(logn).

The second group consists of problems that can be solved in non-deterministic polynomial time. For example : Knapsack problem $O(2^{n/2})$ and Travelling Salesperson problem($O(n^2 2^n)$).

- Any problem for which answer is either yes or no is called decision problem. The algorithm for decision problem is called **decision algorithm.**
- Any problem that involves the identification of optimal cost (minimum or maximum) is called optimization problem. The algorithm for optimization problem is called **optimization algorithm.**
- **Definition of P** - Problems that can be solved in polynomial time. ("P" stands for polynomial).

  Examples - Searching of key element, Sorting of elements, All pair shortest path.
- **Definition of NP -** It stands for "non-deterministic polynomial time". Note that NP does not stand for "non-polynomial".

  Examples - Travelling salesperson problem, Graph coloring problem, Knapsack problem, Hamiltonian circuit problems
- The NP class problems can be further categorized into NP-complete and NP hard problems.
- A problem D is called NP-complete if -
  i) It belongs to class NP
  ii) Every problem in NP can also be solved in polynomial time.
- If an NP-hard problem can be solved in polynomial time then all NP-complete problems can also be solved in polynomial time.
- All NP-complete problems are NP-hard but all NP-hard problems can not be NP-complete.

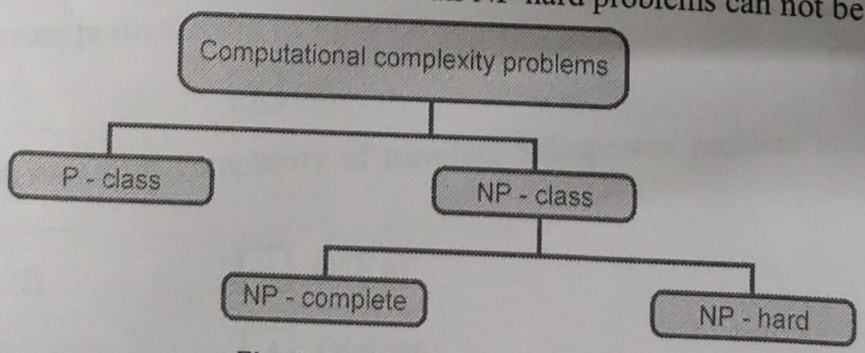


Fig. 8.1.1 Complexity classes

- The NP class problems are the decision problems that can be solved by non deterministic polynomial algorithms.
- **Computational Complexity** - The computational problems is an infinite collection of instances with a solution for every instance.

  In computational complexity the solution to the problem can be "yes" or "no" type. Such type of problems are called **decision problem.** The computational problems can be **function problem**.

  The function problem is a computational problem where single output is expected for every input. The output of such type of problems is more complex than the decision problem.

  The computational problems also consists of a class of problems, whose output can be obtained in polynomial time.
- **Complexity classes -** The complexity classes is a set of problems of related complexity. It includes function problems, P classes, NP classes, optimization problem.
- **Intractability -** Problems that can be solved by taking a long time for their solutions is known as intractable problems.

  If NP is not same as P then NP complete problems are called intractable problems.

8.1.1.1 Example of P Class Problem

Kruskal's algorithm : In Kruskal's algorithm the minimum weight is obtained. In this algorithm also the circuit should not be formed. Each time the edge of minimum weight has to be selected, from the graph. It is not necessary in this algorithm to have edges of minimum weights to be adjacent. Let us solve one example by Kruskal's algorithm.

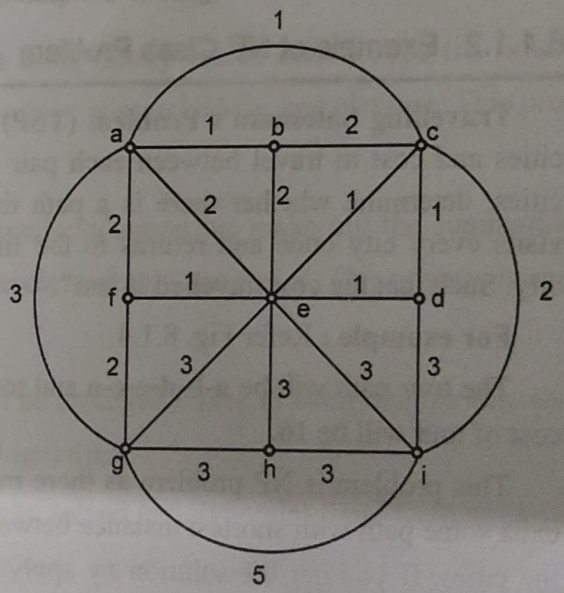


Fig. 8.1.2 Graph G

Example :

Find the minimum spanning tree for the following figure using Kruskal's algorithm.

In Kruskal's algorithm, we will start with some vertex and will cover all the vertices with minimum weight. The vertices need not be adjacent. (Refer Fig. 8.1.3).

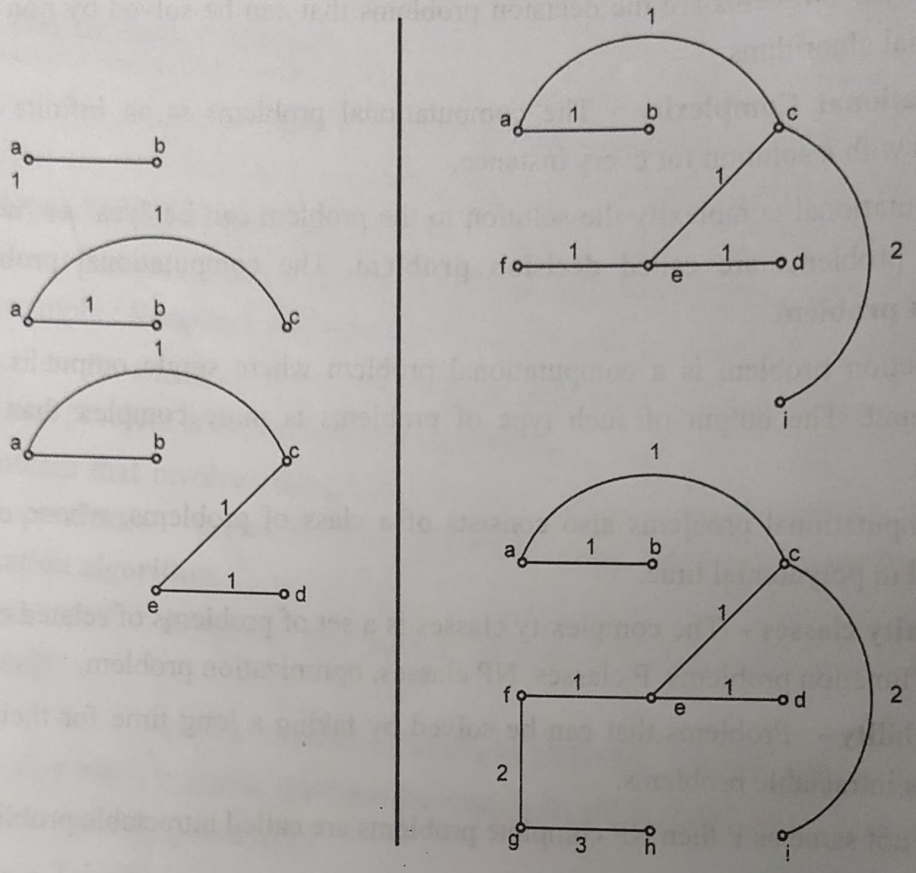


Fig. 8.1.3 Computing minimum spanning tree

8.1.1.2 Example of NP Class Problem

Travelling Salesman's Problem (TSP) : This problem can be stated as " Given a set of cities and cost to travel between each pair of cities, determine whether there is a path that visits every city once and returns to the first city. Such that the cost travelled is less".

For example : Refer Fig. 8.1.4.

The tour path will be **a-b-d-e-c-a** and total cost of tour will be **16.**

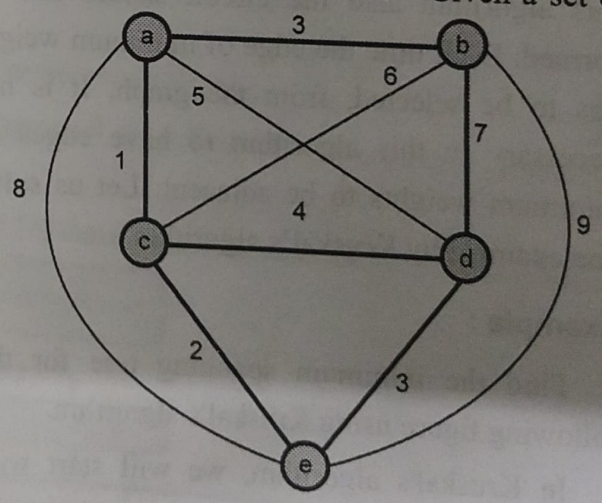


Fig. 8.1.4

This problem is NP problem as there may exist some path with shortest distance between the cities. If you get the solution by applying certain algorithm then Travelling Salesman problem is **NP Complete Problem**. If we get no solution at all by applying an algorithm then the travelling salesman problem belongs to **NP hard class**.

8.1.1.3 Difference between P and NP Class Problems

P Class Problems	NP Class Problems
An algorithm in which for given input the definite output gets generated is called Polynomial time algorithm(P class)	An algorithm is called non deterministically polynomial time algorithm (NP class) when for given input there are more than one paths that the algorithm can follow. Due to which one can not determine which path is to be followed after particular stage.
All the P class problems are basically deterministic.	All the NP class problems are basically non deterministic.
Every problem which is a P class is also in NP class.	Every problem which is in NP is not the P class problem.
P class problems can be solved efficiently.	NP class problems can not be solved efficiently as efficiently as P class problems.
Examples : Binary Search, Bubble Sort	**Examples :** Knapsack problem, Traveling Salesperson problem.

Ex. 8.1.1 *List three problems that have polynomial time algorithms. Justify your answer.*

Sol. : The problems that can be solved in polynomial time are called P - class problems. For example -

1. **Binary search -** In searching an element using binary search method, the list is simply divided at the mid and either left or right sublist is searched for key element. This process is carried out in O(logn).
2. **Evaluation of polynomial -** In a polynomial evaluation we make out the summation of each term of polynomial. The evaluated result is simply an integer. This process is carried out in O(n) time.
3. **Sorting a list -** The elements in a list can be arranged either in ascending or descending order. This procedure is carried out in O(nlogn) time.

This shows that all the above problems can be solved in polynomial time.

Ex. 8.1.2 *Show that both P and NP are closed under the operation union, intersection, concatenation and kleen closure (*)* **SPPU : Dec.-12, Marks 10**

Sol. : Assume $L_1, L_2 \in NP$. That means there exists machines M_1 and M_2 that M_1 decides L_1 in nondeterministic time $O(n^p)$ and M_2 decides L_2 in nondeterministic time $O(n^p)$.

1. Union

- Let, M be a machine that gets w as input.
- Run M_1 for input w. If M_1 accepted then accept.
- Else run M_2 for input w. If M_2 accepted then accept.
- Otherwise reject.

Thus the longest branch in any computation tree on input w of length n is $O(n^{\max\{p, q\}})$.
Thus M is polynomial time nondeterministic decider for $L_1 \cup L_2$.

2. Intersection

- Let, M be a machine that gets w as input.
- Run M_1 on w. If M_1 rejected then reject.
- Else run M_2 for input w. If M_2 rejected then reject.
- Otherwise accept.

Thus the longest branch in any computation tree on input w of length n is $O(n^{\max\{p, q\}})$.
Thus M is polynomial time nondeterministic decider for $L_1 \cap L_2$.

3. Concatenation

- Let, M be a machine that gets input w.
- Split input w nondeterministically as w_1, w_2 such that $w = w_1 w_2$.
- Run M_1 for w_1. If M_1 rejected then reject.
- Else run M_2 for w_2. If M_2 rejected then reject.
- Else accept.

Thus the longest branch in any computation tree for input w of length n is $O(n^{\max\{p, q\}})$. M is polynomial time nondeterministic decider for $L_1 \bullet L_2$.

4. Kleen closure

- If $w = \varepsilon$ then accept.
- Nondeterministically select a number m such that $1 \leq m \, |w|$.
- Now split w into m pieces such that $w = w_1 w_2 \ldots w_m$.
- For all i such that $1 \leq i \; m$, run M_1 for w_i. If M_1 rejected then reject.

The first two steps take O(n) time.

The fourth step takes $O(n^q)$. Thus we get M for w which is polynomial time nondeterministic decider for L_1^*.

Review Questions

Q.1 When do you claim that an algorithm is of a polynomial complexity ?

SPPU : May-08, Marks 2

Q.2 Explain the following :

i) Computational complexity.
ii) Decision problems.
iii) Deterministic and non-deterministic algorithms.
iv) Complexity classes.

SPPU : Dec.-12, Marks 8

8.2 Non-Deterministic Algorithms

SPPU : May-07,10,11,14, 18, 19, Dec.-06,10, Marks 8

- The algorithm in which every operation is uniquely defined is called **deterministic algorithm**.
- The algorithm in which every operation may not have unique result, rather there can be specified set of possibilities for every operation. Such an algorithm is called non-deterministic algorithm. Non-deterministic means that no particular rule is followed to make the guess.
- The non-deterministic algorithm is a two stage algorithm-
  - i) **Non-deterministic (Guessing) stage** - Generate an arbitrary string that can be thought of as a candidate solution
  - ii) **Deterministic ("Verification") stage** - In this stage it takes as input the candidate solution and the instance to the problem and returns *yes* if the candidate solution represents actual solution.

```
Algorithm Non_Determin()
// A[1:n] is a set of elements
// we have to determine the index i of A at which element x is
//located.
{
 // The following for-loop is the guessing stage
       for i=1 to n do
        A[i] := choose(i);

 // Next is the verification(deterministic) stage
     if (A[i] = x) then
     {
          write(i);
          success();
```

```
    }
    write(0);
    fail();
}
```

In the above given non-deterministic algorithm there are three functions used -

1) **Choose -** Arbitrarily chooses one of the element from given input set.
2) **Fail -** Indicates the unsuccessful completion.
3) **Success -** Indicates successful completion.

The algorithm is of non-deterministic complexity O(1), when A is not ordered then the deterministic search algorithm has a complexity Ω (n).

Ex. 8.2.1 *Give the non-deterministic algorithm for sorting elements in non decreasing order.*

SPPU : May-11, Marks 8

Sol. :

```
Algorithm Non_DSort(A,n)
// A[1:n] is an array that stores n elements which are positive integers B[1:n] be an auxiliary
// array in which elements are put at appropriate positions
{
// guessing stage
    for i ← 1 to n do
        B[i] ← 0;//initialize B to 0
    for i ← 1 to n do
    {
        j ← choose(1,n)//select any element from the input set
        if(B[j]!=0) then
            fail();
        B[j] ← A[i];
    }
    //verification stage
    for i ← 1 to n–1 do
        if(B[i]>B[i+1]) then
            fail();// not sorted elements
        write(B[1:n]);// print the sorted list of elements
        success();
}
```

The time required by a non-deterministic algorithm with some input set is minimum number of steps needed to reach to a successful completion if there are multiple choices from input set leading to successful completion. In short, a non-deterministic algorithm is of complexity O(f(n)) where n is the total size of input.

8.2.1 Nondeterministic Algorithm for 0/1 Knapsack Problem

The *0/1 Knapsack Decision Problem* is to determine, for a given profit P, whether it is possible to load the knapsack so as to keep the total weight no greater than m, while making the total profit at least equal to Pt. This problem has the same parameters as the 0-1 Knapsack Optimization Problem(as discussed in earlier chapters) plus the additional parameter Pt.

The non deterministic algorithm for 0/1 knapsack problem is as given below -

```
Algorithm Non_DKnaps()
//p[1:n] is a profit each object i where 1 ≤ i ≤ n
//w[1:n] is the weight of each object i where 1 ≤ i ≤ n
//Profit and Wt represent the current total profit and weight
// m is the capacity of knapsack
{
//initially no item is selected
 Wt:=0;
Profit:=0;

// guessing stage
      for i:=1 to n do
      {
            x[i]:=choose(0,1);
            Wt:=Wt+x[i]*w[i];
            Profit:=Profit+x[i]*p[i];
      }
//verification stage
// selected items exceed the capacity or less profit is earned

      if((Wt>m) or (Profit<pt)) then
            fail();
      else
            success();//maximum profit earned
}
```

Review Questions

Q.1 Consider the following search algorithm :

```
j = any value between 1 to n
If (a[j] = x) then print "Success";
else
print "Fails"
```

Is this algorithm non-deterministic ? Justify your answer. **SPPU : Dec.-06, 10, Marks 8**

Q.2 Write a non-deterministic Knapsack algorithm. **SPPU : May-07, 10, Marks 8**

Q.3 What is a deterministic and non-deterministic algorithm ? Write a non-deterministic algorithm for searching element. **SPPU : May-14, May-19, End Sem, Marks 8**

Q.4 What is nondeterministic algorithm? Write the nondeterministic algorithm for sorting the element of an array. **SPPU : May-18, End Sem, Marks 8**

8.3 The NP Hard Problem

SPPU : Dec.-06,07,13, 18, May-14,15, Marks 16

- As we know, P denotes the class of all deterministic polynomial language problems and NP denotes the class of all non-deterministic polynomial language problems. Hence

$$P \subseteq NP.$$

- The question of whether or not

$$P = NP$$

holds, is the most famous outstanding problem in the computer science.

- Problems which are known to lie in P are often called as *tractable*. Problems which lie outside of P are often termed as *intractable*. Thus, the question of whether P = NP or P ≠ NP is the same as that of asking whether there exist problems in NP which are intractable or not.
- The relationship between P and NP is depicted by Fig. 8.3.1.
- We don't know if P = NP. However in 1971 S. A. Cook proved that a particular NP problem known as SAT(Satisfiability of sets of Boolean clauses) has the property that, if it is solvable in polynomial time, so are all NP problems. This is called a "NP-complete" problem.

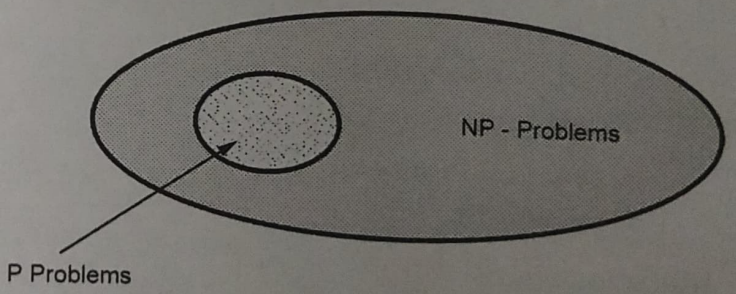


Fig. 8.3.1 P and NP problems assuming P ≠ NP

- Let L_1 and L_2 are two problems then problem L_1 reduces to L_2 if and only if there is a way to solve L_1 by deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

L_1 reduces to L_2 can be denoted as $L_1 \propto L_2$. In other words we can say that if there exists any polynomial time algorithm which solves L_2 then we can solve L_1 in polynomial time. We can also state that if $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$.

- A NP problem such that, if it is in P, then NP = P. If a (not necessarily NP) problem has this same property then it is called **"NP-hard"**. Thus the class of NP-complete problem is the intersection of the NP and NP-hard classes.

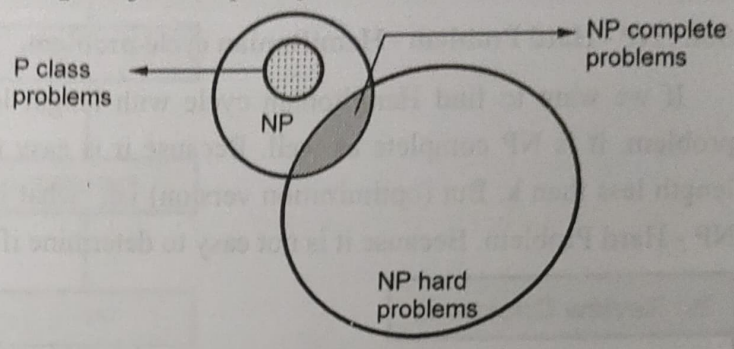


Fig. 8.3.2 Relationship between P, NP, NP-complete and NP-hard problems

- Normally the decision problems are NP-complete but optimization problems are NP-hard. However if problem L_1 is a decision problem and L_2 is optimization problem then it is possible that $L_1 \propto L_2$. For instance the Knapsack decision problem can be Knapsack optimization problem.
- There are some NP-hard problems that are not NP-complete. For example *halting problem.* The halting problem states that : "Is it possible to determine whether an algorithm will ever halt or enter in a loop on certain input ?"

  Two problems P and Q are said to be polynomially equivalent if and only if $P \propto Q$ and $Q \propto P$.

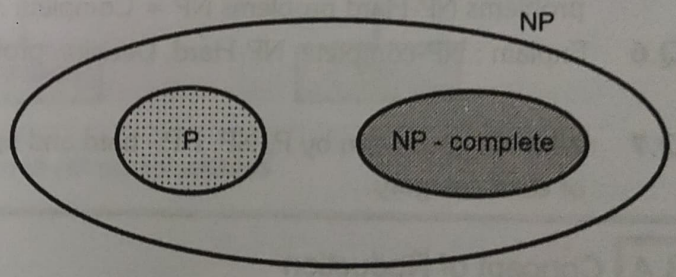


Fig. 8.3.3

Difference between NP Hard and NP Complete Problems

The problems that can be solved in polynomial time are called P - class problems. For example -

1. **Binary search -** In searching an element using binary search method, the list is simply divided at the mid and either left or right sublist is searched for key element. This process is carried out in O(logn).
2. **Evaluation of polynomial -** In a polynomial evaluation we make out the summation of each term of polynomial. The evaluated result is simply an integer. This process is carried out in O(n) time.
3. **Sorting a list -** The elements in a list can be arranged either in ascending or descending order. This procedure is carried out in O(nlogn) time.

This shows that all the above problems can be solved in polynomial time.

Ex. 8.3.1 *Specify any one example of NP-hard problem. Also mention that why it is NP hard ?*

SPPU : May-15, End Sem, Marks 8

Sol. : NP - Hard Problem - Hamiltonian cycle problem.

If we want to find Hamiltonian cycle with length less than k then this is a determinist problem. It is NP complete as well. Because it is easy to determine Hamiltonian cycle with length less than k. But (optimization version) i.e. "what is the shortest Hamiltonian cycle ?" is NP - Hard Problem. Because it is not easy to determine if cycle obtained is shortest or not.

Review Questions

Q.1 What do you mean by NP hard problem. **SPPU : Dec.-06, Marks 2**

Q.2 Comment on the statement : "Two problems L1 and L2 are polynomially equivalent if and only if L1 $\propto$ L2 and L2 $\propto$ L1". **SPPU : Dec.-06, 07, Marks 4**

Q.3 Give a relationship between P, NP, NP complete and NP hard. **SPPU : Dec.-07, Marks 6**

Q.4 Explain the terms decision problem, NP, NP-Hard and NP-Complete. State examples of each. State if P a proper subset of NP or NP is a proper subset of P. Explain with the help of diagram. **SPPU : Dec.-13, Marks 16**

Q.5 What are the conditions to prove that a problem P is NP-Complete ? Are all NP-Complete problems NP-Hard problems NP = Complete ? **SPPU : Dec.-13, Marks 16**

Q.6 Explain : NP-complete, NP-Hard, Decision problem and polynomial time algorithm. **SPPU : May-14, Marks 8**

Q.7 What do you mean by P, NP, NP - hard and np - complete problems ? Give an example of each category. **SPPU : Dec.-18, End sem, Marks 8**

8.4 Concept of Reduction

To prove whether particular problem is NP complete or not we use polynomial time reducibility. That means if

$$A \xrightarrow{Poly} B \xrightarrow{Poly} C \text{ then } A \xrightarrow{Poly} C.$$

The **reduction** is an important task in NP completeness proofs. This can be illustrated by Fig. 8.4.1.

Various types of reductions are

Local replacement - In this reduction A $\rightarrow$ B by dividinginput to A in the form of components and then these components can be converted to components of B.

Component design - In this reduction A $\rightarrow$ B by building special component for input B that enforce properties required by A.

The reduction can be denoted as $A \leq T^P B$.

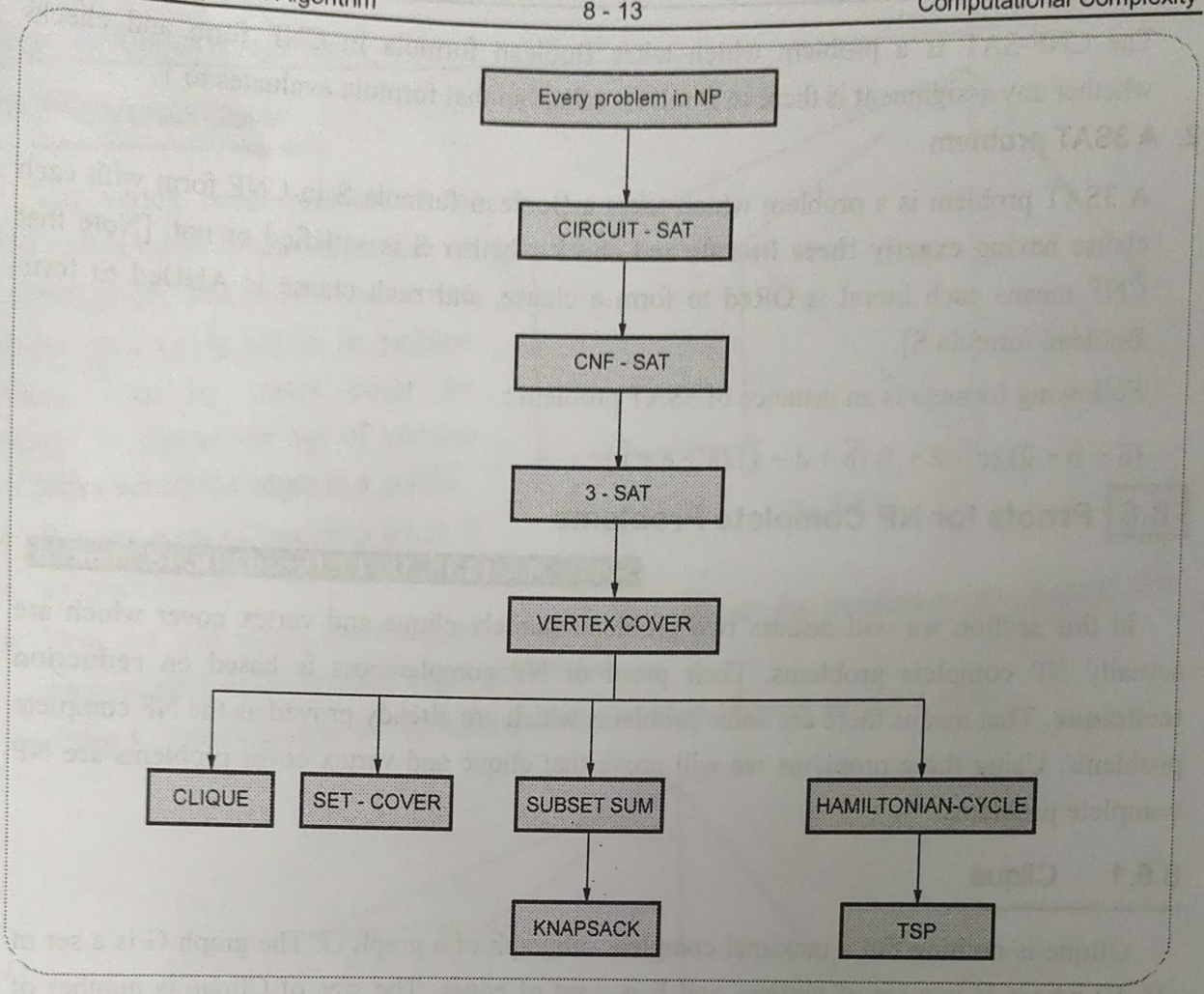


Fig. 8.4.1 Reduction in NP completeness

8.5 Satisfiability Problem

Before understanding the 3 SAT problem we will get introduced with the satisfiability problem.

1. CNF - SAT problem

This problem is based on Boolean formula. The Boolean formula has various Boolean operations such as OR(+), AND (·) and NOT. There are some notations such as → (means implies) and ↔ (means if and only if).

A Boolean formula is in **Conjuctive Normal Form (CNF)** if it is formed as collection of subexpressions. These subexpressions are called **clauses**.

For example

$(\overline{a} + b + d + \overline{g})\ (c + \overline{e})\ (\overline{b} + d + \overline{f} + h)\ (a + c + e + \overline{h})$

This formula evaluates to 1 if b, c, d are 1.

The CNF-SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

2. A 3SAT problem

A 3SAT problem is a problem which takes a Boolean formula S in CNF form with each clause having **exactly three literals** and check whether S is satisfied or not. [Note that CNF means each literal is ORed to form a clause, and each clause is ANDed to form Boolean formula S].

Following formula is an instance of 3SAT problem :

$(\overline{a} + b + \overline{g})\ (c + \overline{e} + f)\ (\overline{b} + d + \overline{f})\ (a + e + \overline{h})$

8.6 Proofs for NP Complete Problems

SPPU : May-14,15, 18, 19. Dec.-09,16, 18, Marks 8

In this section we will discuss two problems namely clique and vertex cover which are actually NP complete problems. Their proof of NP completeness is based on **reduction technique**. That means there are some problems which are already proved as the NP complete problems. Using these problems we will prove that clique and vertex cover problems are NP complete problems.

8.6.1 Clique

Clique is nothing but a maximal complete subgraph of a graph G. The graph G is a set of (V, E) where V is a set of vertices and E is a set of edges. The size of Clique is number of vertices present in it.

For example : (See Fig. 8.6.1)

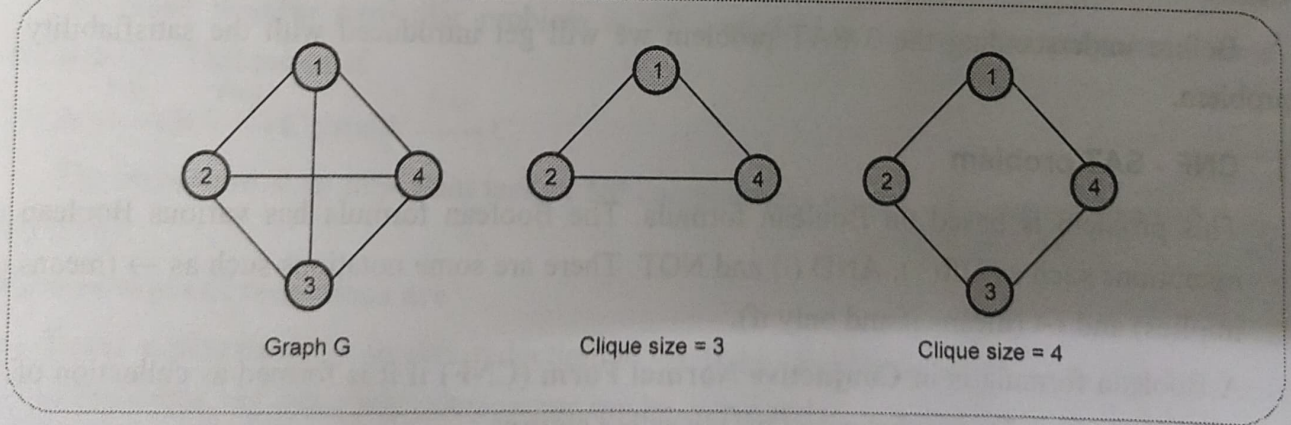


Fig. 8.6.1

Note that Clique of size = 3 and 4 are complete subgraph of graph G.

Max Clique Problem - It is an optimization problem in which the largest size Clique is to be obtained.

8.6.2 Vertex Cover

The vertex cover problem is to find vertex cover of minimum size in a given graph. The word vertex cover means each vertex covers its incident edges. Thus by vertex cover we expect to choose the set of vertices which cover all the edges in a graph.

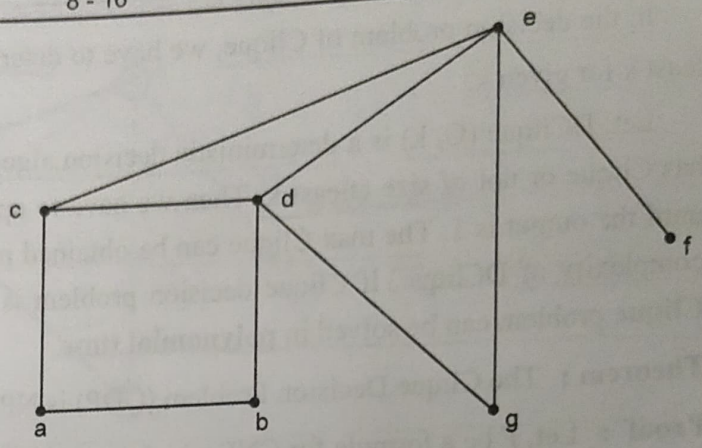


Fig. 8.6.4

- **For example :** Consider a graph G as given below.
- Now we will select some arbitrary edge and delete all the incident edges. Repeat this process until all the incident edges get deleted.

Step 1 : Select an edge a-b and delete all the incident edges to vertex a or b.

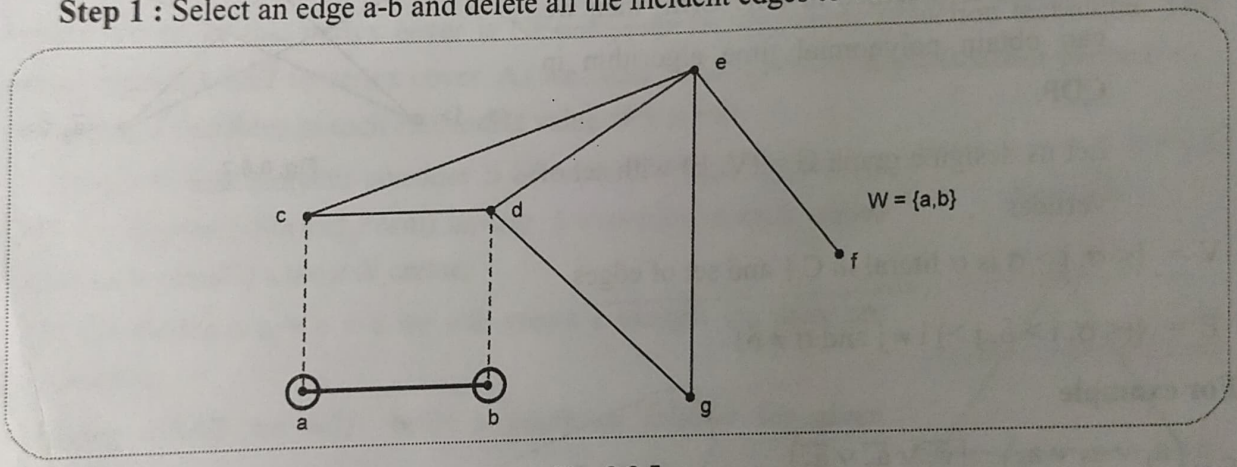


Fig. 8.6.5

Step 2 : Select an edge c-e and delete all the incident edges to vertex c or e.

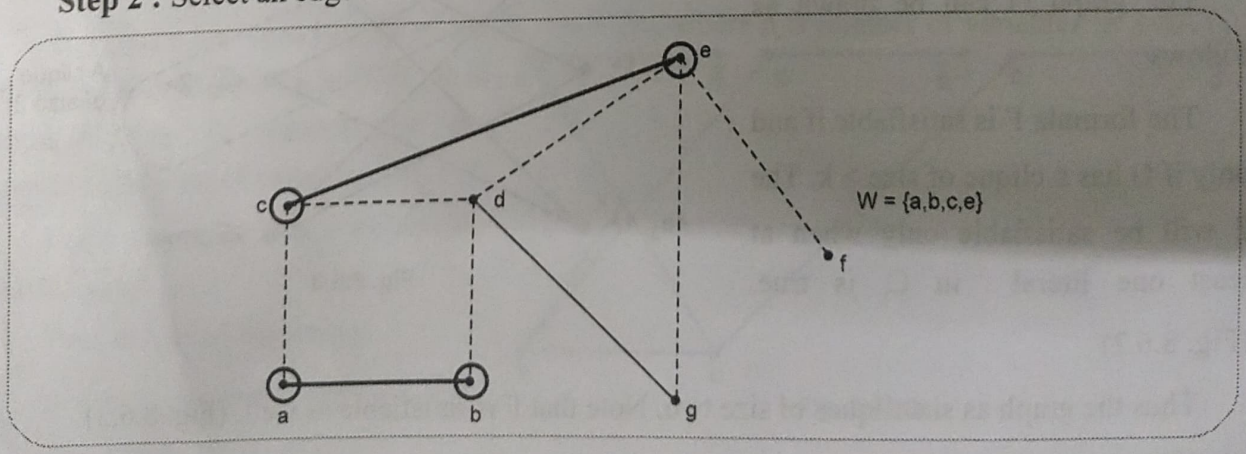


Fig. 8.6.6

In the decision problem of Clique, we have to determine whether G has a Clique of size at least k for given k.

Let, DClique (G, k) is a deterministic decision algorithm for determining whether graph G has Clique or not of size atleast k. Then we have to apply DClique for k = n, n – 1, n – 2,... until the output is 1. The max Clique can be obtained in time ≤ n · F(n) where F(n) is the time complexity of DClique. If Clique decision problem is solved in **polynomial time** then **max Clique** problem can be solved in **polynomial time.**

Theorem : The Clique Decision Problem (CDP) is NP-complete.

Proof : Let, F be a formula for CNF which is satisfiable.

$$F = C_1 \wedge C_2 \wedge \ldots C_k$$

where C is a clause. Every clause in CNF is denoted by a_i where $1 \le i \le n$. If length of F is F and is obtained in time O(m) then we can obtain polynomial time algorithm in CDP.

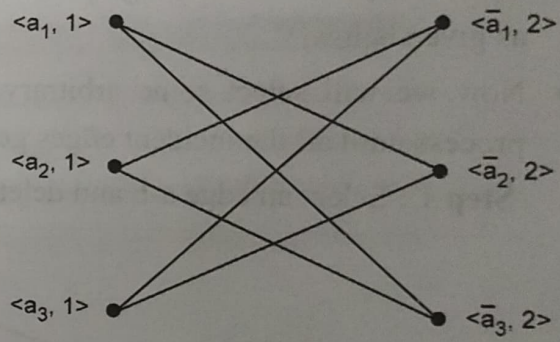


Fig. 8.6.2

Let us design a graph G = (V, E) with set of vertices

V = {< σ, i > σ is α literal in C_i} and set of edges

E = {(< σ, i > δ, j >) i ≠ j and σ ≠ $\bar{\delta}$}

For example

$$F = \begin{pmatrix} a_1 \vee a_2 \vee a_3 \\ c_1 \end{pmatrix} \wedge \begin{pmatrix} \overline{a_1} \vee \overline{a_2} \vee \overline{a_3} \\ c_1 \end{pmatrix}$$

The graph G can be drawn as follows

The formula F is satisfiable if and only if G has a clique of size > k. The F will be satisfiable only when at least one literal in C_i is true. (Fig. 8.6.2)

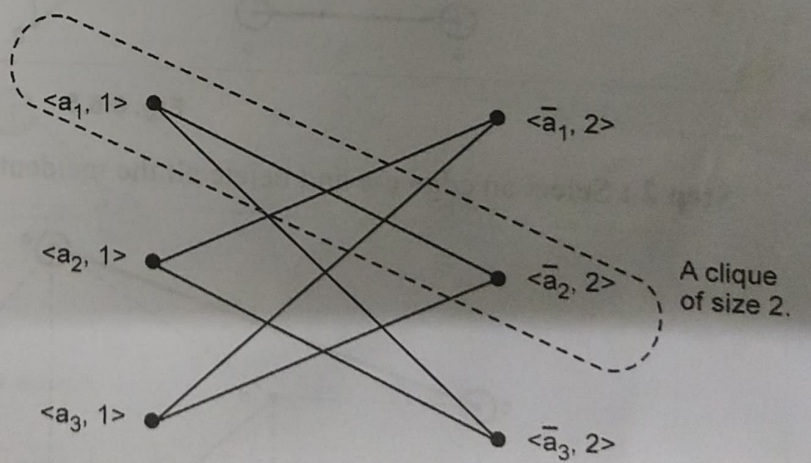


Fig. 8.6.3

Thus the graph as six cliques of size two. Note that F is satisfiable as well. (Fig. 8.6.3)

As CNF satisfiability is NP complete CDP is also NP complete.

Step 3 : Select an edge d-g. All the incident edges are already deleted.

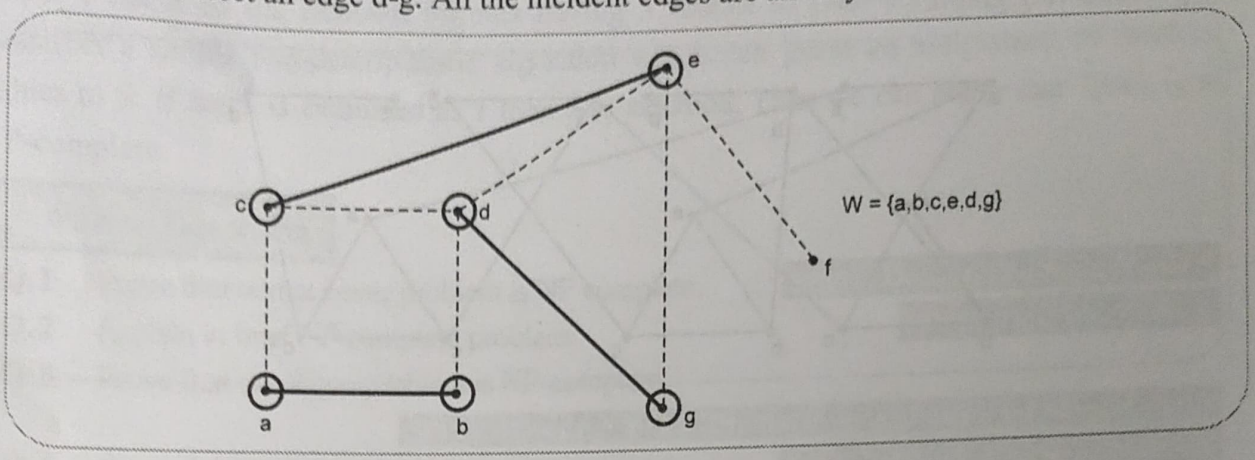


Fig. 8.6.7

Thus we obtain vertex cover as {a, b, c, d, e, g}.

Theorem : The vertex cover is NP-complete.

Proof : To prove that vertex cover is NP-complete we will apply reduction technique. That means reduce 3-SAT to vertex cover. As we know 3-SAT to basically a Boolean expression s containing 3 variables in each clause and value of S is true.

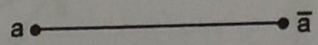


Fig. 8.6.8

To prove this theorem consider S be some Boolean formula in CNF (conjunctive Normal Form) having 3 variables in each clause no for each variable a we will create,

If the clause is a + b + c we will create a triangle (as there are 3 - variables)

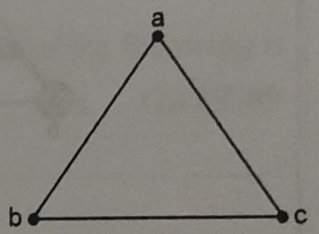


Fig. 8.6.9

Using 3-SAT we will build a graph as follows for given expression.

$(a + b + c)\ (a + b + \bar{c})\ (\bar{a} + c + \bar{d})$

The graph has vertex cover of size K = n + 2m, where n is number of variables in 3-SAT, m is number of clauses in CNF, K is total number of vertices that are present in the set of vertex cover.

For a clause (a + b + c) we can build a graph as :

Thus for given expression

$(a + b + c)\ (a + b + \bar{c})\ (\bar{a} + c + \bar{d})$

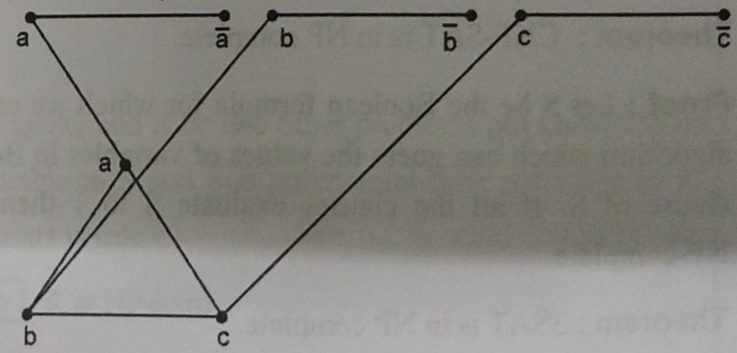


Fig. 8.6.10

we get following graph.

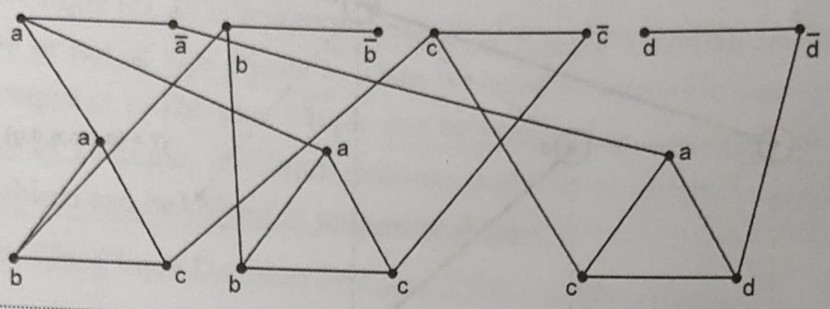


Fig. 8.6.11

Here n = Number of variable = 4

m = Number of clauses = 3

$\therefore$ K = n + 2m = 4 + 2(3) = 10

In vertex cover we get K = 10 vertices which cover all the vertices. The vertex cover is shown by following graph in which the covering vertices are rounded.

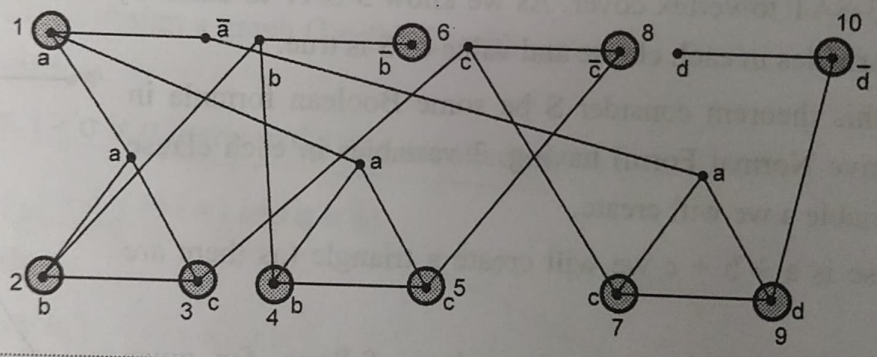


Fig. 8.6.12

Thus K = n + 2m is proved as K = 10. This is how 3-SAT can be reduced to vertex cover. The 3-SAT is NP-complete. Hence vertex is NP-complete is proved.

Theorem : CNF-SAT is in NP complete.

Proof : Let S be the Boolean formula for which we can construct a simple non-deterministic algorithm which can guess the values of variables in Boolean formula and then evaluates each clause of S. If all the clauses evaluate S to 1 then S is satisfied. Thus CNF-SAT is in NP-complete.

Theorem : 3SAT is in NP complete.

Proof : Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S. If the S is evaluated as 1 then S is satisfied. Thus we can prove that 3SAT is in NP-complete.

Review Questions

Q.1 Prove that vertex cover problem is NP complete. **SPPU : Dec.-16 End Sem, Marks 8**

Q.2 Explain in brief NP complete problem. **SPPU : Dec.-09, Marks 4**

Q.3 Prove that : A clique problem is NP-complete. **SPPU : May-14, May-15, 18, End Sem, Dec.-18, End Sem, Marks 8**

Q.4 Prove that satisfiability problem in NP complete. **SPPU : May-19, End Sem, Marks 8**

8.7 Multiple Choice Questions

Q. 1 Any problem for which the answer is either zero or one is called _____ .

[a] decision problem [b] optimization problem

[c] clique problem [d] all of the above

Q. 2 All deterministic search algorithm have time complexity of _____ .

[a] O(n) [b] O(1) [c] Ω (n) [d] θ (n)

Q. 3 Let X be a problem that belongs to the class NP. Then which one of the following is TRUE? **GATE-09**

[a] There is no polynomial time algorithm for X.

[b] If X can be solved deterministically in polynomial time, then P = NP.

[c] If X is NP-hard, then it is NP-complete.

[d] X may be undecidable.

Q. 4 Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true ? **GATE-06**

[a] R is NP-complete [b] R is NP-hard

[c] Q is NP-complete [d] Q is NP-hard