Design and Analysis of Algorithm
Unit-4

Dynamic Programming

Dynamic programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- In contrast to the divide-and-conquer method, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table.

Optimization problems

- We typically apply dynamic programming to ***optimization problems***.

- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution *an* optimal solution to the problem.

Dynamic programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1) Characterize the structure of an optimal solution.
2) Recursively define the value of an optimal solution.
3) Compute the value of an optimal solution, typically in a bottom-up fashion.
4) Construct an optimal solution from computed information.

Dynamic programming

Using dynamic programming, we shall solve the following problems:-

- Matrix-chain multiplication
- Longest common subsequence problem
- 0-1 Knapsack problem
- All pairs shortest path problem

Matrix-chain multiplication problem

This problem is stated as following:-

Given a sequence (chain) $< A_1, A_2, \ldots\ldots\ldots, A_n >$ of n matrices where for $i = 1, 2, 3, \ldots\ldots\ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots\ldots\ldots A_n$ in a way that minimizes the number of scalar multiplications.

Example: Find all parenthesization of matrix for n=4.

Solution:

1) ((A1A2)(A3A4))
2) (((A1A2)A3)A4)
3) (A1(A2(A3A4)))
4) ((A1(A2A3))A4)
5) (A1((A2A3)A4))

Matrix-chain multiplication problem

Example: Consider the following chain of matrices < A1, A2, A3 > . The dimension of matrices are the following:-

A1 = 10x100, A2 = 100x5, A3= 5x50

Find the optimal parenthesization of these matrices.

Solution: All the parenthesization of these matrices are:-

1) ((A1A2)A3)
2) (A1(A2A3))

Number of scalar multiplications in (1)

$$= 10*100*5 + 10*5*50$$

$$= 5000 + 2500 = 7500$$

Number of scalar multiplications in (2)

$$= 100*5*50 + 10*100*50$$

$$= 25000 + 50000 = 75000$$

Therefore, the solution ((A1A2)A3) is optimal solution.

Number of parenthesizations

Let P(n) denote the number of parenthesizations for n matrices. It is computed as following:-

P(n) = 1 if n=1

= $\sum_{k=1}^{n-1} P(k)P(n-k)$ if n >=2

Dynamic programming approach for Matrix-chain multiplication problem

Step 1: In this step, we find the optimal substructure and then use it to construct an optimal solution to the problem.

Let $A_{i..j}$ → Matrix that results from evaluating the product $A_i A_{i+1} \ldots\ldots\ldots A_j$. Where $i \le j$.

- If $i < j$, then to parenthesize the product $A_i A_{i+1} \ldots\ldots\ldots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$.
- The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

Matrix-chain multiplication problem

Step 2:

- Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.
- m[i,j] is recursively defined as following:-

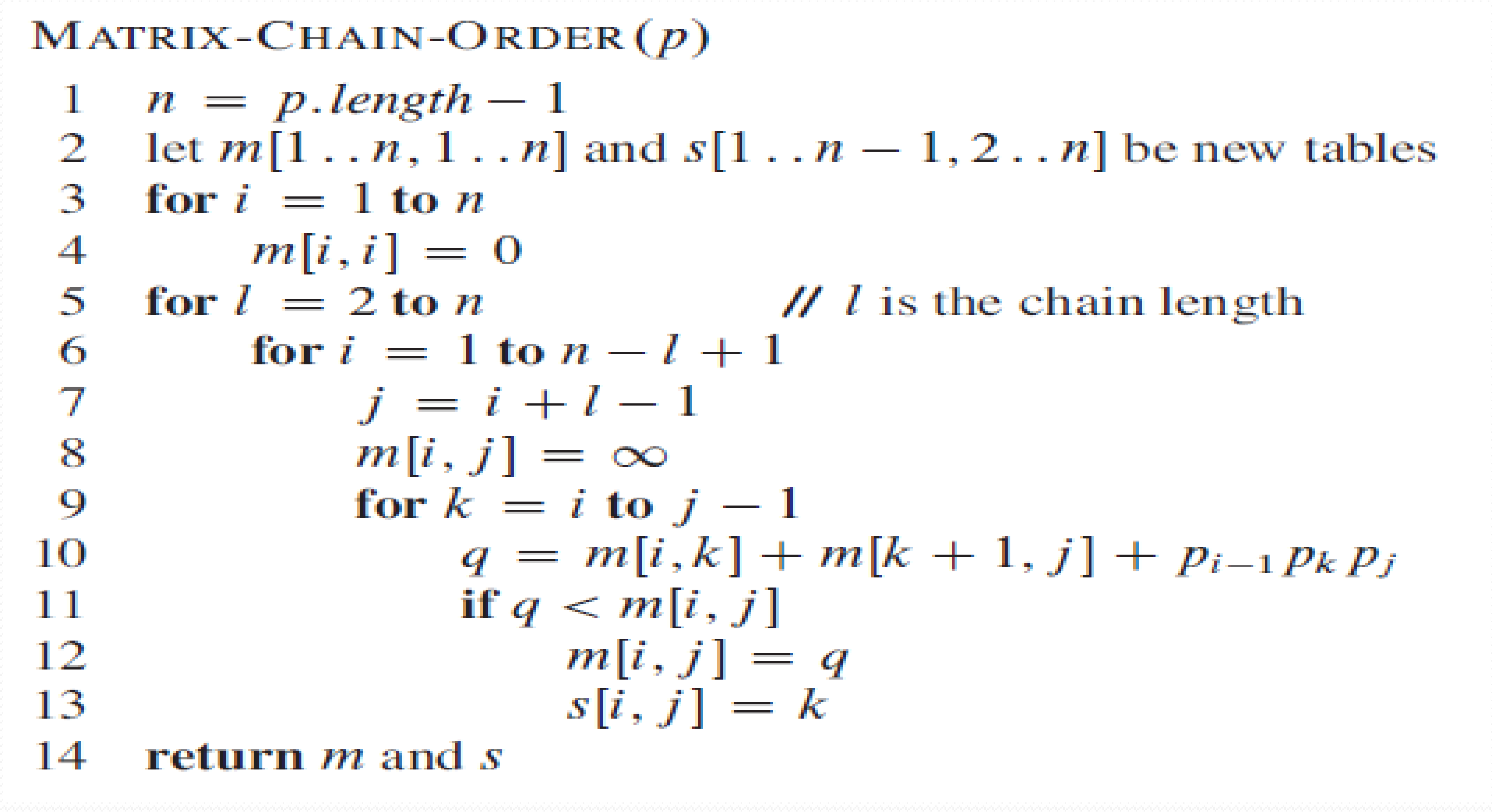
$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\,. \end{cases}$$

- We define s[i, j] to be the value of k at which m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$ is minimum.
- We compute the optimal solution using matrix s.
- And matrix m is used to compute the value of optimal solution.

Matrix-chain multiplication problem

Step 3: In this step, we compute the matrices m and s.

Following algorithm is used to compute the matrices m and s. This algorithm assumes that matrix A_i has dimensions $p_{i-1}xp_i$ for i = 1, 2, ………, n. Its input is a sequence p = < p_o, p_1, ….., p_n >, where p.*length* = n+1.



```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                  // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + $p_{i-1} p_k p_j$
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Matrix-chain multiplication problem

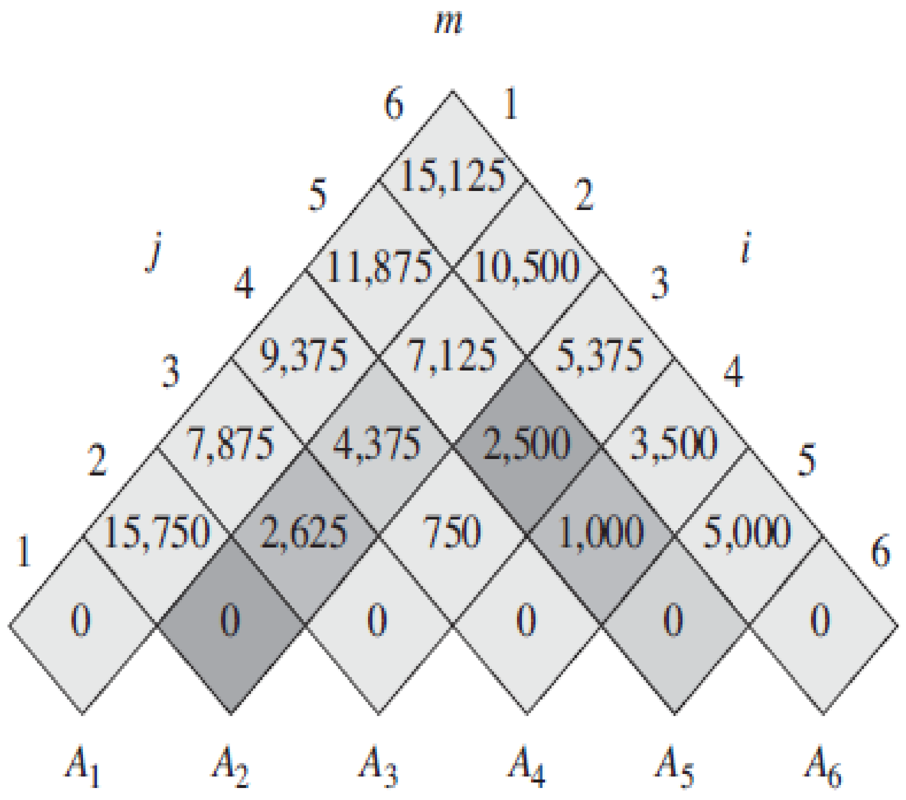
Example: Consider the following matrix chain multiplication problem.

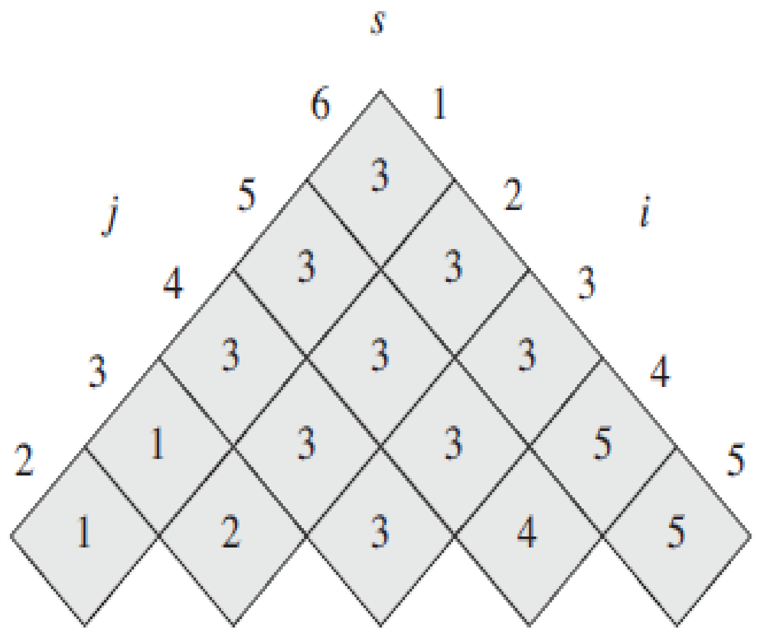
$A_1 \rightarrow$ 30x35, $A_2 \rightarrow$ 35x15, $A_3 \rightarrow$ 15x5,

$A_4 \rightarrow$ 5x10, $A_5 \rightarrow$ 10x20, $A_6 \rightarrow$ 20x25.

Find the optimal solution and its value.

Solution:

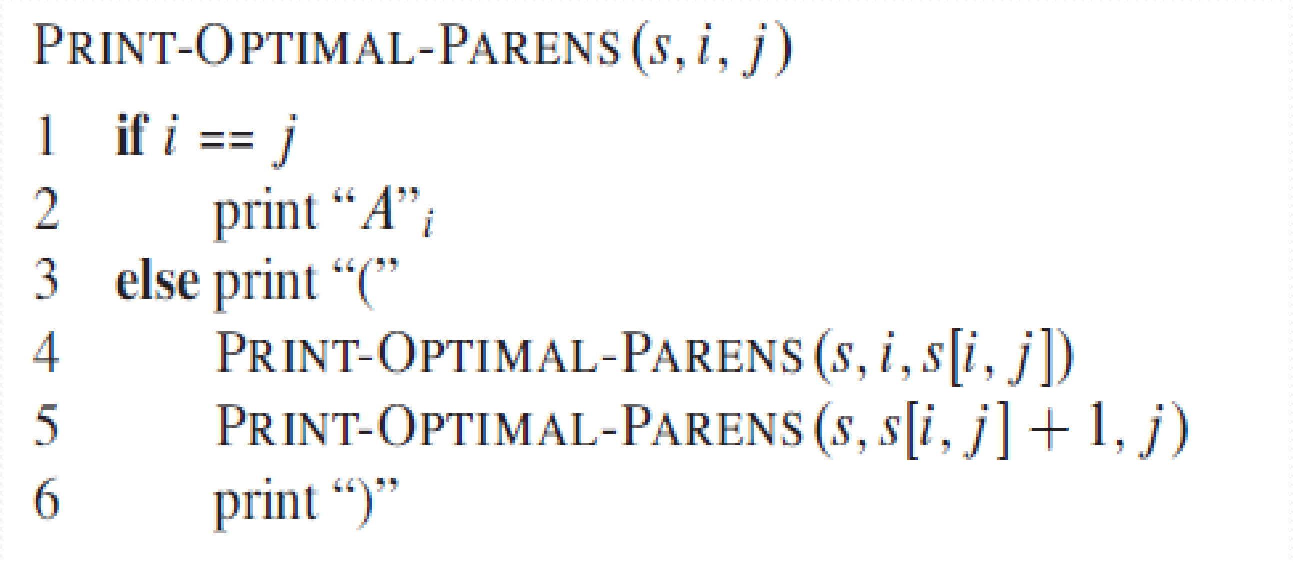




Matrix-chain multiplication problem

Step 4: Constructing an optimal solution

Following algorithm is used to create optimal solution i.e. it finds optimal parenthesization.



```
PRINT-OPTIMAL-PARENS(s, i, j)
if $i == j$
    print "$A$"$_i$
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

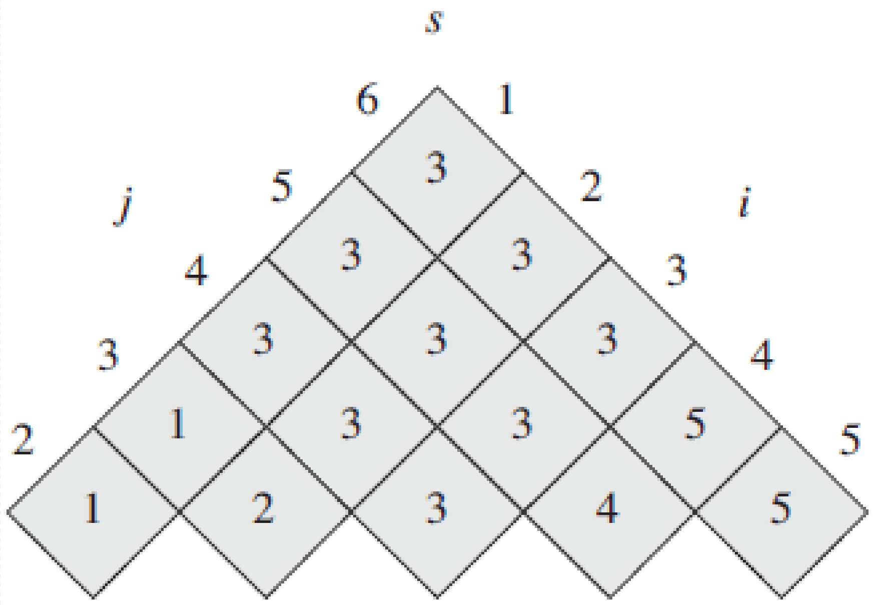
- The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of < A_1,A_2 ,……. , A_n >.

Time complexity of this algorithm is O(n).

Matrix-chain multiplication problem

Step 4: Constructing an optimal solution for previous example

The initial call PRINT-OPTIMAL-PARENS(s, 1, 6) prints an optimal parenthesization .



Optimal solution will be
((A1(A2A3))((A4A5)A6))

Longest Common Subsequence Problem (LCS problem)

In the ***longest-common-subsequence problem***, we are given two sequences $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots\ldots, y_n >$ and wish to find a maximum length common subsequence of X and Y.

Longest Common Subsequence Problem (LCS problem)

Example:

- If X = < A, B, C, B, D, A, B > and Y= < B, D, C, A, B, A >, the sequence < B, C, A > is a common subsequence of both X and Y .
- The sequence < B, C, A > is not a *longest* common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence < B, C, B, A >, which is also common to both X and Y , has length 4.
- The sequence < B, C, B, A > is an LCS of X and Y , as is the sequence < B, D, A, B >, since X and Y have no common subsequence of length 5 or greater.

Solving the LCS problem using dynamic programming

Step-1: Characterizing a longest common subsequence

Prefix of **a sequence:**

Given a sequence $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$, we define the ith ***prefix*** of X, for i = 0, 1, 2, ………., m, as $X_i = < x_1, x_2, \ldots\ldots\ldots, x_i >$.

For example, if X = < A, B, C, B, D, A, B >, then X4= <A, B, C, B > and X_0 is the empty sequence.

Solving the LCS problem using dynamic programming

Optimal substructure of an LCS

Let $X = < x_1, x_2, \ldots\ldots\ldots, x_m >$ and $Y = < y_1, y_2, \ldots\ldots\ldots, y_n >$ be sequences, and let $Z = < z_1, z_2, \ldots\ldots\ldots, z_k >$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Solving the LCS problem using dynamic programming

Step 2: A recursive solution

Let c[i, j] be the length of an LCS of the sequences X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula:-

c[i,j] = 0 if i = 0 or j = 0

= c[i-1,j-1]+1, if i, j > 0 and $x_i = y_j$

= max{ c[i-1, j], c[i,j-1] }, if i, j > 0 and $x_i \neq y_j$

Let b is the matrix which stores one of the following three arrows, ←, ↑ and ↖.

b[i,j] = ← , if c[i,j] = c[i,j-1]

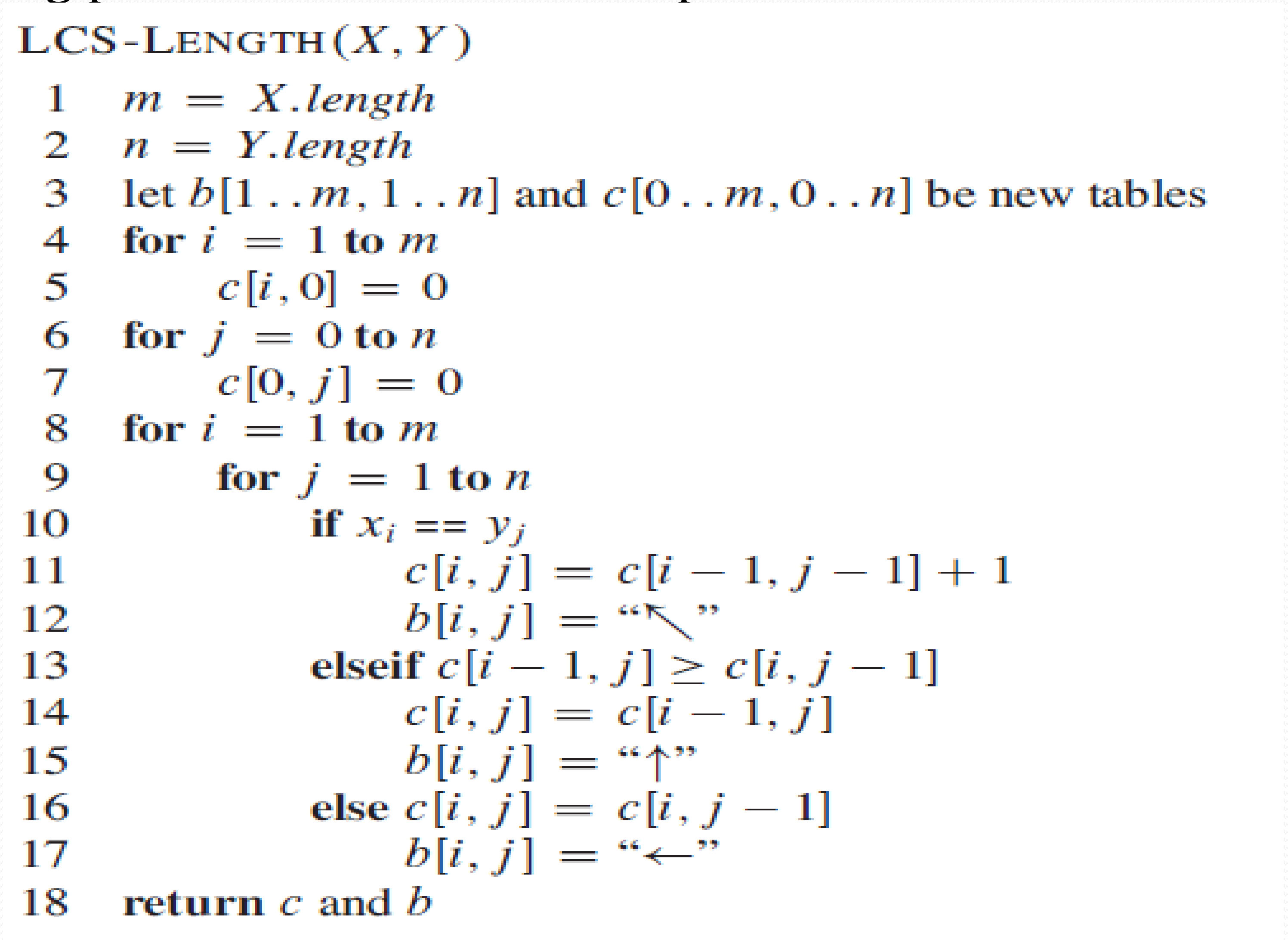
b[i,j] = ↑ , if c[i,j] = c[i-1,j]

b[i,j] = ← , if c[i,j] = c[i-1,j-1]

LCS problem using dynamic programming

Step-3: Computing the length of an LCS

Following procedure is used to compute the table b and c.



```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if $x_i == y_j$
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

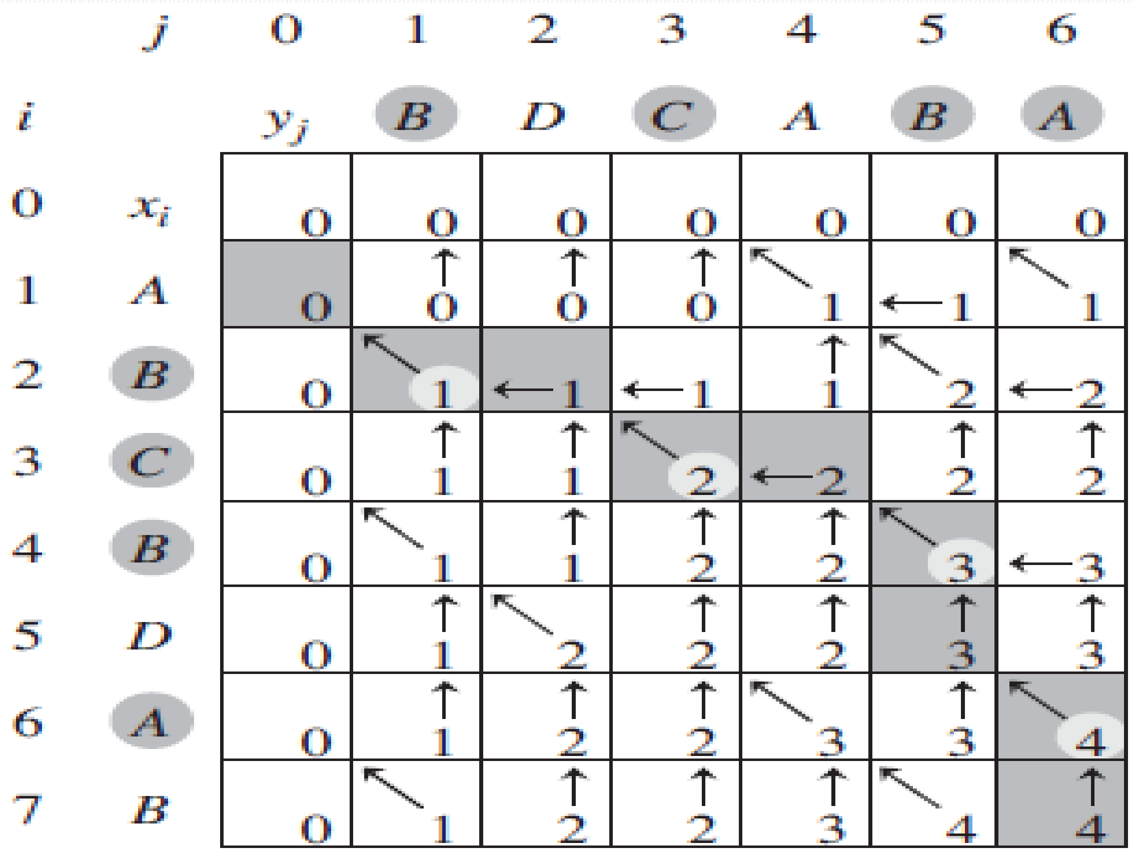
LCS problem using dynamic programming

Step-3: Computing the length of an LCS.

Example: Find LCS of the following sequences:-

X = < A, B, C, B, D, A, B > and Y = < B, D, C, A, B, A >.

Solution: We compute the table corresponding to b and c as following:-

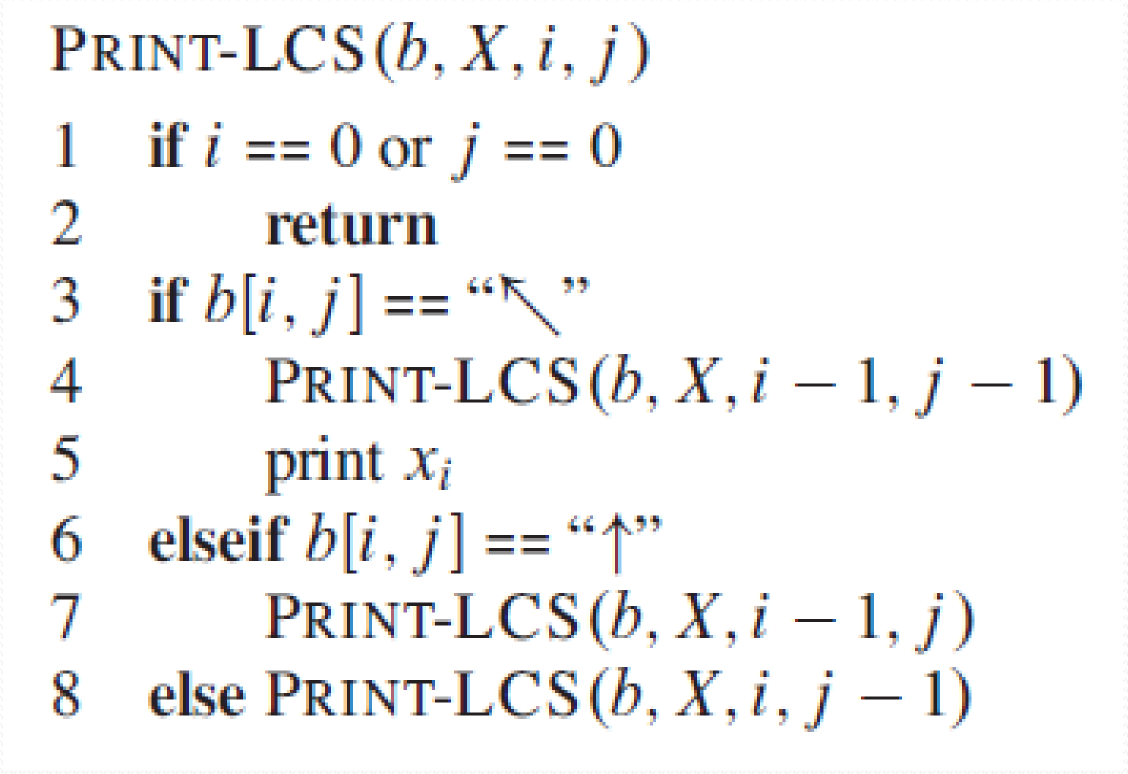


	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

LCS problem using dynamic programming

Step-4: Constructing an LCS

- The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS(b, X, m, n).



```
PRINT-LCS($b, X, i, j$)
if $i == 0$ or $j == 0$
        return
if $b[i, j] ==$ "↖"
        PRINT-LCS($b, X, i-1, j-1$)
        print $x_i$
elseif $b[i, j] ==$ "↑"
        PRINT-LCS($b, X, i-1, j$)
else PRINT-LCS($b, X, i, j-1$)
```

LCS problem using dynamic programming

Step-4:

Time complexity of this algorithm = O(m+n).

The LCS of sequences taken in previous example = BCBA.

0-1 Knapsack using dynamic programming

- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

- Based on the nature of the items, Knapsack problems are categorized as
    1. Fractional Knapsack
    2. 0-1 Knapsack

0-1 Knapsack using dynamic programming

Fractional Knapsack

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x_i}$ of **i**th item.

0-1 Knapsack

In this version of Knapsack problem, we cannot break an item, either pick the complete item or don't pick it (0-1 property).

0-1 Knapsack using dynamic programming

Dynamic programming approach for solving 0-1 Knapsack problem is explained as following:-

Let c[i,w] denotes the value of the solution for items 1,2,3,….,i and maximum weight w.

It is defined as

$$c[i,w] = \begin{cases} 0 & \text{,if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{, if } i > 0 \text{ and } w_i > w \\ \max\{ v_i + c[i-1,w-w_i], c[i-1,w] \} & \text{, if } i > 0 \text{ and } w_i \leq w \end{cases}$$

0-1 Knapsack using dynamic programming

Dynamic programming algorithm for solving 0-1 Knapsack problem is the following:-

```
Dynamic-0-1-Knapsack(v, w, n, W)
for l ← 0 to W
        c[0,l] ← 0
for i ← 1 to n
        c[i,0] ← 0
        for l ← 1 to W
                if $w_i \le l$ then
                        if $v_i + c[i-1,l-w_i] > c[i-1,l]$ then
                                $c[i,l] \leftarrow v_i + c[i-1,l-w_i]$
                        else
                                c[i,l] ← c[i-1,l]
                else
                        c[i,l] ← c[i-1,l]
return c
```

Time complexity of this algorithm = $O(nW)$

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 4, \; W = 5$

$(v_1, v_2, v_3, v_4) = (3, 4, 5, 6)$

$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$

Solution:

Here, we have to calculate c[4, 5].

$c[4,5] = \max\{ v_4 + c[3,0], c[3,5] \} = \max\{ 6 + 0, c[3,5] \}$

$= \max\{ 6, c[3,5] \} = \max\{ 6, 7\} = \mathbf{7}$

$c[3,5] = \max\{ v_3 + c[2, 1], c[2, 5] \} = \max\{ 5 + 0, c[2, 5] \}$

$= \max\{ 5, c[2, 5] \} = \max\{ 5, 7\} = 7$

$c[2,5] = \max\{ v_2 + c[1, 2], c[1, 5] \} = \max\{ 4 + 3, 3\} = 7$

Therefore, the value of the optimal solution is 7. And the optimal solution is (1, 2).

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	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 table is used to find the value of the optimal solution.

Therefore, value of optimal solution = 7

0-1 Knapsack using dynamic programming

Solution by Tabular Method

	1	2	3	4	5
1	↑	←	←	←	←
2	↑	↑	←	←	←
3	↑	↑	↑	←	↑
4	↑	↑	↑	↑	↑

This table is used to find the optimal solution.

Therefore, optimal solution = (1, 2)

0-1 Knapsack using dynamic programming

Example: Solve the following 0-1 knapsack.

$n = 6, \ W = 100$

$(v_1, v_2, v_3, v_4, v_5, v_6) = (40, 35, 20, 4, 10, 6)$

$(w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 40, 20, 10, 10)$

Solution:

	0	10	20	30	40	50	60	70	80	90	100
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	40
2	0	0	0	0	0	35	35	35	35	35	40
3	0	0	0	0	20	35	35	35	35	55	55
4	0	0	4	4	20	35	35	39	39	55	55
5	0	10	10	14	20	35	45	45	49	55	65
6	0	10	16	16	20	35	45	51	51	55	65

0-1 Knapsack using dynamic programming

From table in the previous slide, the value of optimal solution will be 65.

Following is the table to compute the optimal solution.

The optimal solution will be (2, 3, 5).

	10	20	30	40	50	60	70	80	90	100
1	↑	↑	↑	↑	↑	↑	↑	↑	↑	←
2	↑	↑	↑	↑	←	←	←	←	←	↑
3	↑	↑	↑	←	↑	↑	↑	↑	←	←
4	↑	←	←	↑	↑	↑	←	←	↑	↑
5	←	←	←	↑	↑	←	←	←	↑	←
6	↑	←	←	↑	↑	↑	←	←	↑	↑

Floyd’s Warshall Algorithm for shortest path

- This algorithm is used to solve the all-pairs shortest-paths problem on a directed graph G =(V,E).
- This algorithm is based on the dynamic programming approach.
- Let $\mathbf{d^{(k)}_{ij}}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ………, k}.
- When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $\mathbf{d^{(0)}_{ij} = w_{ij}}$.

Floyd's Warshall Algorithm for shortest path

Recursive formula to compute $d^{(k)}_{ij}$

We define $d^{(k)}_{ij}$ as the following:-

$$d^{(k)}_{ij} = w_{ij} \quad \text{if } k = 0$$

$$= \min\{ d^{(k-1)}_{ij} , d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \} \quad \text{if } k \geq 1.$$

- Because for any path, all intermediate vertices are in the set {1, 2, ..., n}, therefore the matrix $D^{(n)} = (d^{(n)}_{ij})$ gives the final answer.

Floyd’s Warshall Algorithm for shortest path

Constructing a shortest path

- We define $\pi^{(k)}_{ij}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1, 2, ……, k}.
- When **k = 0**, a shortest path from i to j has no intermediate vertices at all. Therefore,

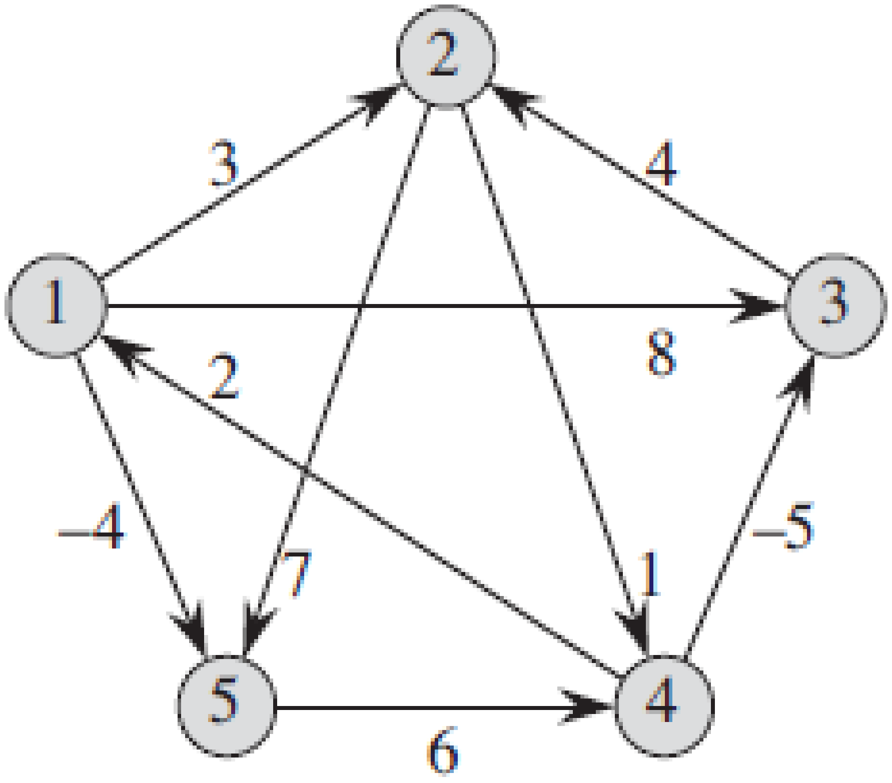
$$\pi^{(0)}_{ij} = \text{NIL} \qquad \text{if } i=j \text{ or } w_{ij} = \infty$$
$$= i \qquad \text{if } i \neq j \text{ and } w_{ij} < \infty.$$

- **For k ≥ 1.**

$$\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}\ , \text{ if } d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$
$$= \pi^{(k-1)}_{kj}\ , \text{ if } d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$$

Floyd's Warshall Algorithm for shortest path

Example: Apply the Floyd's Warshall algorithm in the following graph :-



Floyd's Warshall Algorithm for shortest path

Solution: Weighted matrix of this graph is the following:-

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Therefore,

$$D^{(o)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(o)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

Here, we have to calculate $D^{(5)}$ and $\Pi^{(5)}$. It is calculated as following:-

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

•

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Floyd's Warshall Algorithm for shortest path

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

- Matrix $D^{(5)}$ stores the shortest distance between each pair of vertices.
- Matrix $\Pi^{(5)}$ is used to find shortest path between each pair of vertices.

Floyd's Warshall Algorithm for shortest path

```
Floyd-Warshall(W, n)
        $D^{(0)} = W$
        for i ← 1 to n
            for j ← 1 to n
                if ($i=j$ or $w_{ij}=\infty$)
                    $\pi^{(0)}_{ij}$ = Nil
                if ($i\neq j$ and $w_{ij}< \infty$)
                    $\pi^{(0)}_{ij} = i$
        for k ← 1 to n
            for i ← 1 to n
                for j ← 1 to n
                    if ($d^{(k-1)}_{ij} \leq d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$)
                        $d^{(k)}_{ij} = d^{(k-1)}_{ij}$
                        $\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}$
                    else
                        $d^{(k)}_{ij} = d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$
                        $\pi^{(k)}_{ij} = \pi^{(k-1)}_{kj}$
```

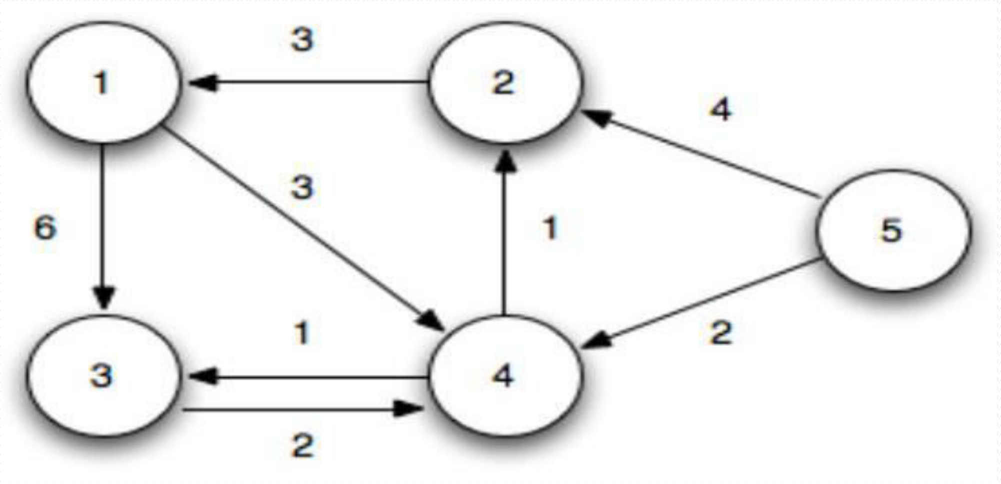
Time complexity of this algorithm is $\theta(n^3)$.

AKTU Examination Questions

1. Define principal of optimality. When and how dynamic programming is applicable.
2. Give Floyd Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example.
3. Difference between Greedy Technique and Dynamic programming.
4. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.
5. What is dynamic programming? How is this approach different from recursion? Explain with example.
6. Solve the following 0/1 knapsack problem using dynamic programming. P=[11,21,31,33] w=[2,11,22,15] c=40, n=4.

AKTU Examination Questions

7. Define Floyd Warshall Algorithm for all pair shortest path and apply the same on following graph:



7. Define the terms—LCS, Matrix Chain multiplication & Bellman-Ford algorithm.
8. Explain the Floyd Warshall algorithm with an example.
9. Find an optimal parenthesization of a matrix chain product whose sequence of dimensions is {10, 5, 3, 12, 6}.

Backtracking

Backtracking

- **Backtracking** is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

- Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

Backtracking

- In the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2, \dots, x_n)$, where the x_i are chosen from some finite set S_i.
- Using backtracking, we find one vector that maximizes(or minimizes or satisfies) a criterion function $P(x_1, x_2, \dots, x_n)$.
- Sometimes, this method finds all the vectors that satisfies P.
- If m_i is the size of set S_i, then $m = m_1 m_2 \dots\dots m_n$ tuples are possible that satisfy the criterion function P.
- The brute force approach would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum.
- The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials.

Backtracking

- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions P $(x_1, x_2, \dots, x_i)$. (sometimes called bounding functions) to test whether the vector being formed has any chance of success.

- If the partial vector $(x_1, x_2, \dots, x_i)$ can in no way lead to an optimal solution, then m_{i+1} ……. m_n possible test vectors can be ignored entirely.

Backtracking

Explicit constraints

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Common example of explicit constraints are

(1) $x_i \geq 0$ or S_i = The set of all positive real numbers

(2) $x_i = 0$ or 1 or $S_i = \{0, 1\}$

All tuples that satisfy the explicit constraints define a possible solution space.

Implicit constraints

The implicit constraints are rules that determine which of the tuples in the solution space satisfy the criterion function.

Backtracking

State space tree

When we search the solutions of the problem using backtracking, a tree is formed by the result of search. This tree is said to be sate space tree. Each node in the tree represents a state.

Problem state

Each node in the tree is called problem state.

Solution state

Solution states are those problem states s for which the path from root to s defines a tuple in the solution space.

Answer state

Answer states are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem.

Backtracking

State space

All the paths from the root to other nodes define the state space of the problem.

Live node

A node which has been generated and all of whose children have not yet been generated is called a live node.

E-node

The live node whose children are currently being generated is called the E-node (node being expanded).

Dead node

A dead node is a generated node which is not to be expanded further or all of whose children have been generated.

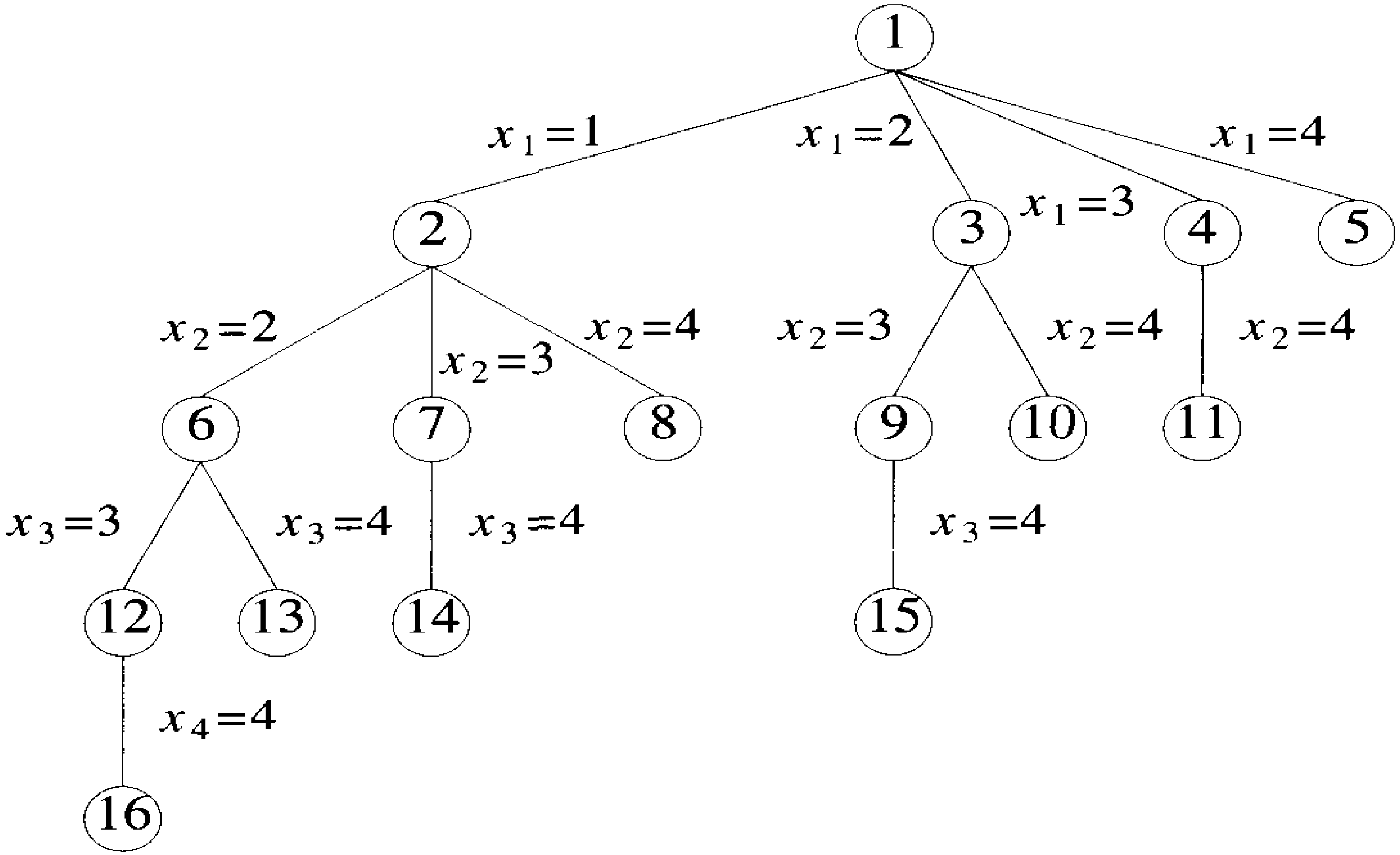
Backtracking

Note: Bounding functions are used to kill live nodes without generating all their children.

Note: Depth first node generation with bounding functions is called backtracking.

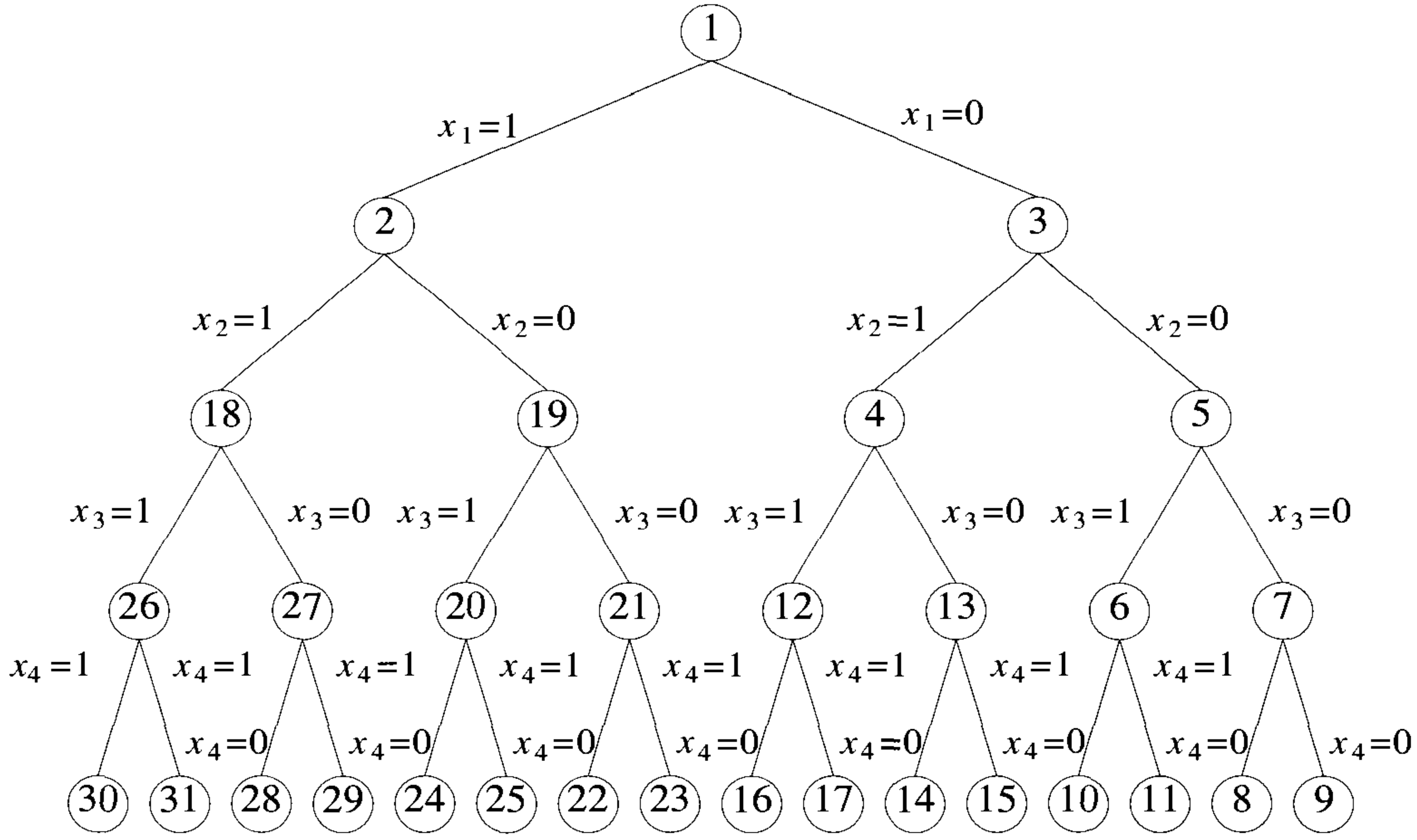
Backtracking

Example:



Backtracking

Example:



N-Queen Problem

Statement:

In this problem, we have given N queens and a NxN chessboard. The objective of the problem is to place all the N queens on this chessboard in such way no two queens lie on the same row, column or diagonal.

N-Queen Problem

Example: Solve the 4-Queen Problem.

Solution:

$\Rightarrow$

Q			

Initial position of chessboard

N-Queen Problem

Q			
		Q	

⇒

Q			
			Q
	Q		

⇒

	Q		
			Q
Q			
		Q	

Final solution

Backtracking algorithm for N-Queen

- According to this approach, we put 1st queen in 1st row, 2nd queen in 2nd row and so on. We have to only find column number of the queen.
- Therefore, the solution vector will be $(x_1, x_2, \ldots\ldots\ldots, x_n)$. Here x_i is the column number of ith queen.
- The explicit constrains for this problem are that no two x_i's can be same and no two queens can be on the same diagonal.
- Using these two constraints, the size of solution space are n!.
- Suppose two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if

$$i-j = k-l \quad \text{or} \quad i+j = k+l$$

$$\Rightarrow \quad i-k = j-l \quad \text{or} \quad i-k = -(j-l)$$

Therefore, two queens lie on the same diagonal iff

$$|i-k| = |j-l|$$

Backtracking algorithm for N-Queen

An algorithm f or determining the solution of n-queens problem is:-

```
        N-Queen(k, n)
1.      for i ← 1 to n
2.          do if  Place(k,i) = True
3.              then  x[k] ← I
4.                  if k=n
5.                      then for  j←1 to n
6.                          do print x[j]
7.                  else
8.                      N-Queen(k+1, n)
```

Backtracking algorithm for N-Queen

An algorithm f or determining the solution of n-queens problem is:-

```
        Place(k, i)
1.      for j ← 1 to k-1
2.              do if  (x[j]=i) or (abs(x[j]-i) = abs(j-k))
3.                      then  return (False)
4.      return(True)
```

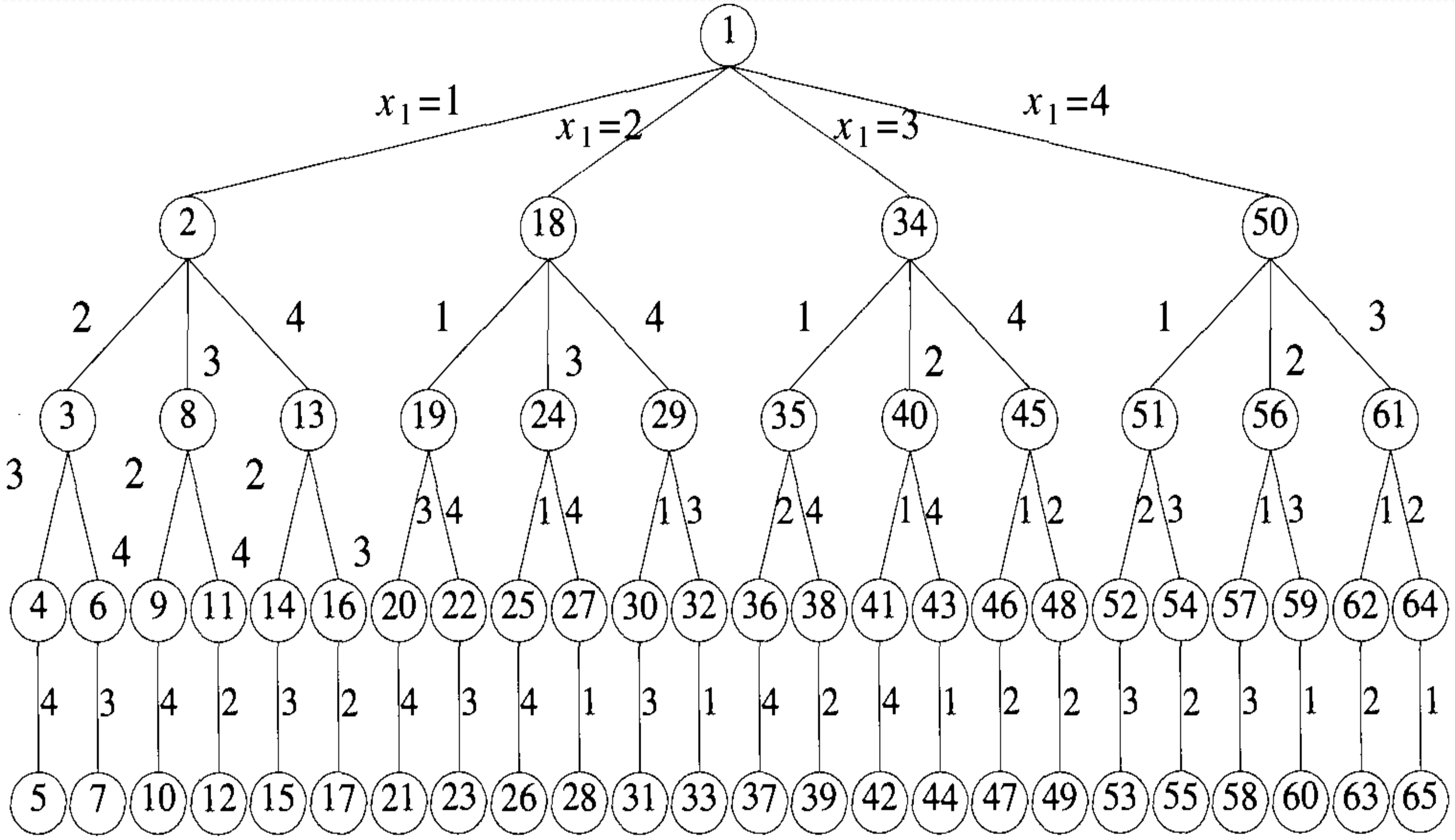
The initial call is N-Queen(1, n).

Running time of this algorithm is O(n!).

Backtracking algorithm for N-Queen

Example: Draw the state space tree for 4-queen problem.

Solution: State space tree for 4-queen problem is the following:-



Backtracking algorithm for N-Queen

Example: Solve the 8-queens problem.

Solution:

		Q					
					Q		
	Q						
						Q	
Q							
			Q				
							Q
				Q			

Sum of Subsets problem

Statement: In this problem, we are given n distinct positive numbers (called weights) and one another number m. We have to find all combinations of these numbers whose sums are m.

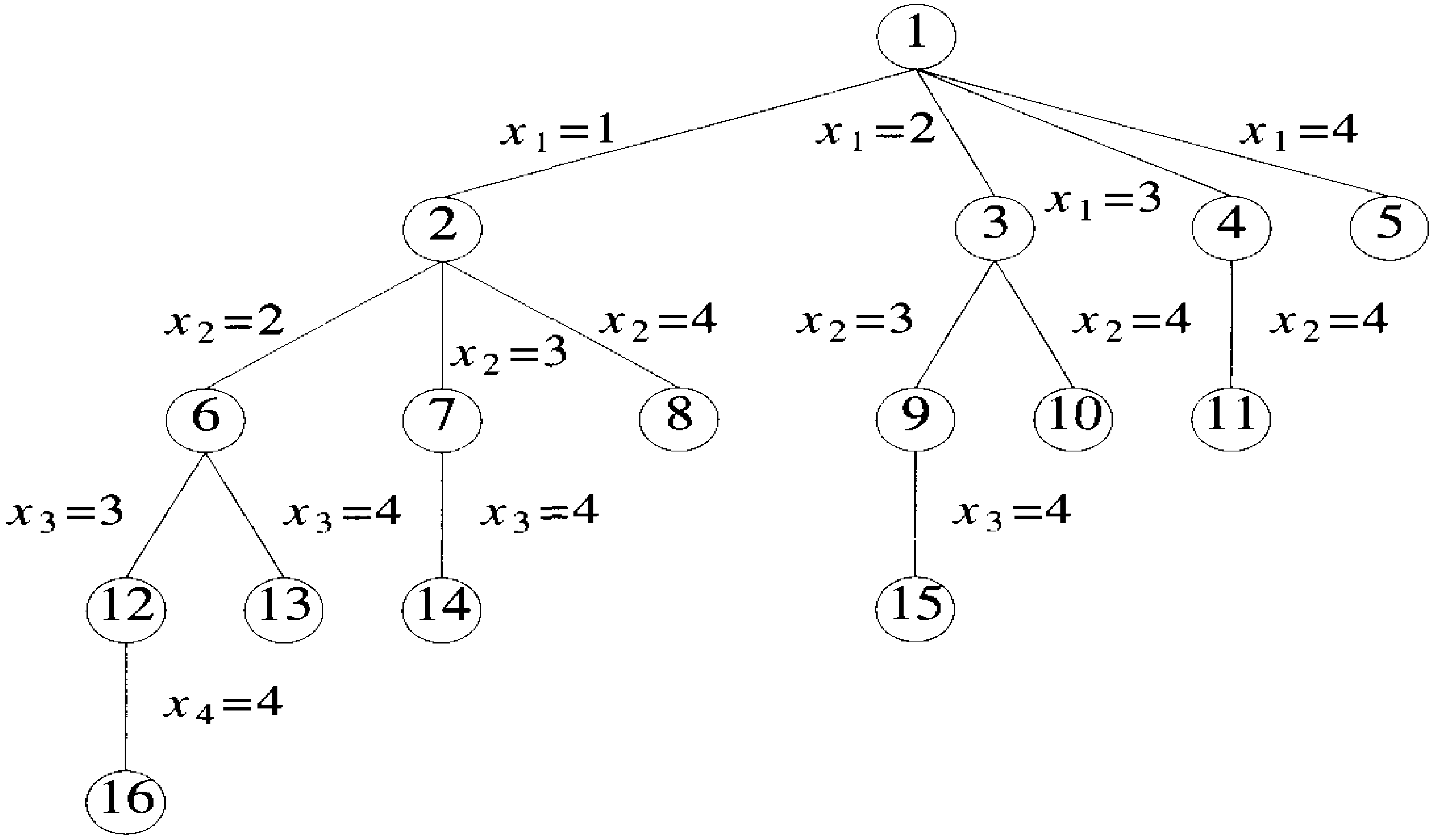
This problem can be formulated using either fixed or variable sized tuples.

Example: Consider n=4 , (w1, w2, w3, w4) = (11, 13, 24, 7) and m=31.

- The desired subsets are (11, 13, 7) and (24, 7).
- The solution in variable-size tuple will be (1, 2, 4) and (3,4).
- The solution in fixed-size tuple will be (1, 1, 0, 1) and (0, 0, 1, 1).

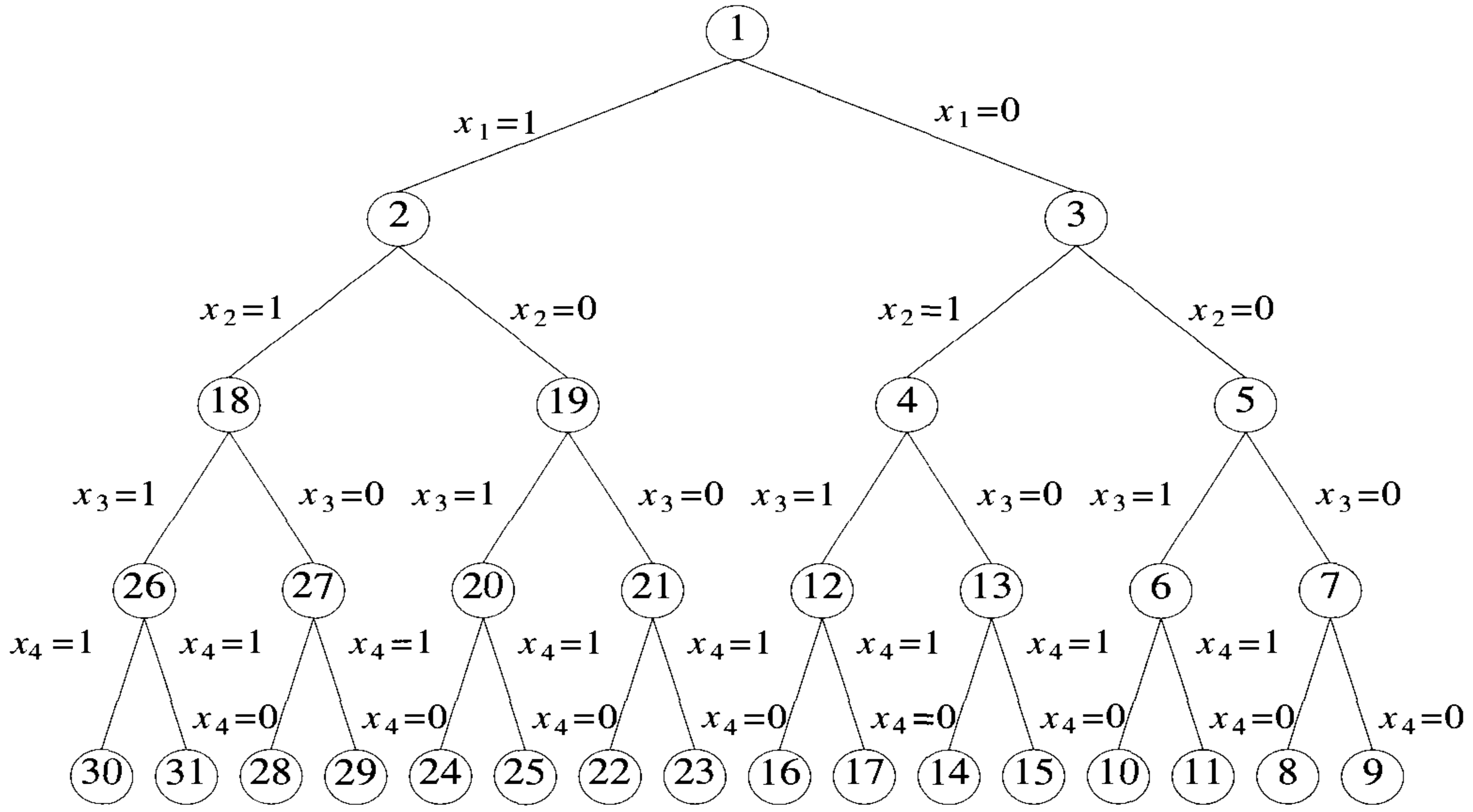
Sum of Subsets problem

The state space tree for this question in case of variable-size tuple is the following:-



Sum of Subsets problem

The state space tree for this question in case of fixed-size tuple is the following:-



Backtracking approach for Sum of Subsets problem

Backtracking solution for this problem uses fixed-sized tuple strategy.

In this case, the value of x_i in the solution vector will be either 1 or 0 depending on whether weight w_i is included or not.

The bounding function is $B_k(x_1, x_2, \ldots\ldots, x_k)$ = true iff

$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$

Clearly $(x_1, x_2, \ldots\ldots, x_k)$ can not lead to an answer node if this condition is not satisfied.

If all the weights are initially in ascending order then $(x_1, x_2, \ldots\ldots, x_k)$ can not lead to an answer node if

$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$

Backtracking approach for Sum of Subsets problem

Therefore, we use the following bounding function

The bounding function is $B_k(x_1, x_2, ..., x_k)$ = true iff

$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$

and

$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

Note: Here, we have assume all the weights are in ascending order.

Backtracking approach for Sum of Subsets problem

```
SumOfSub(s,k,r)
1.x[k] ← 1
2.if (s+w[k] = m)
3.      for j←1 to k
4.              print x[j]
5.else if (s+w[k]+w[k+1] ≤ m)
6.              SumOfSub(s+w[k], k+1, r-w[k])
7.if (( s+r-w[k] ≥ m) and (s+w[k+1] ≤ m))
8.x[k] ← 0
9.              SumOfSub(s, k+1, r-w[k])
```

The initial call is SumOfSub(0, 1, $\sum_{i=1}^{n} w_i$)

Backtracking approach for Sum of Subsets problem

Time complexity of this algorithm is $O(2^n)$.

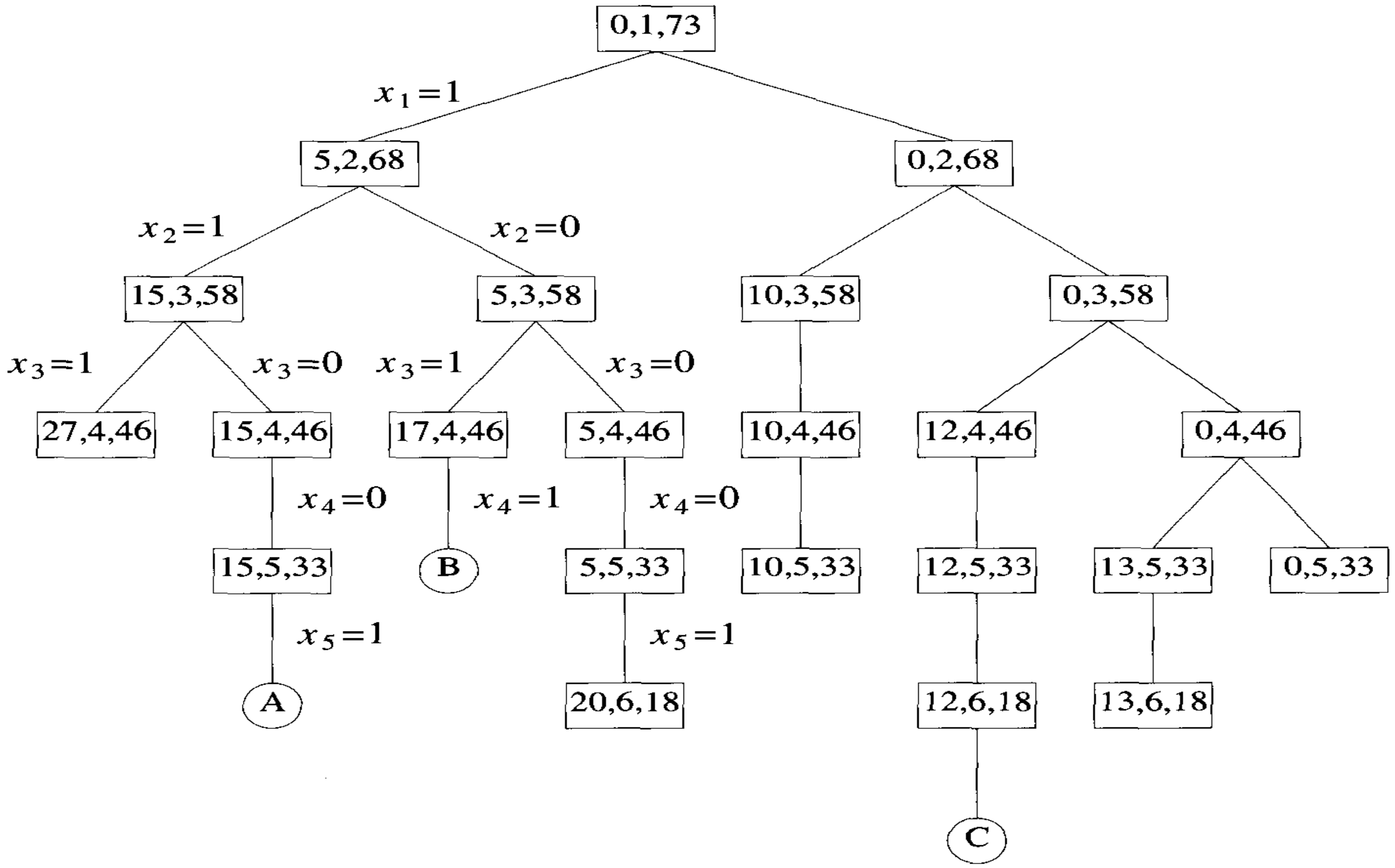
Because

Total number of nodes $= 1+2+2^2+2^3+\ldots\ldots.+2^n$

$= 2^{n+1}-1$

Example: Let w = {5, 10, 12, 13, 15, 18} and m = 30. Find all possible subsets of w that sum to m. Do this using SumOfSub. Draw the portion of the state space tree that is generated.

Solution:



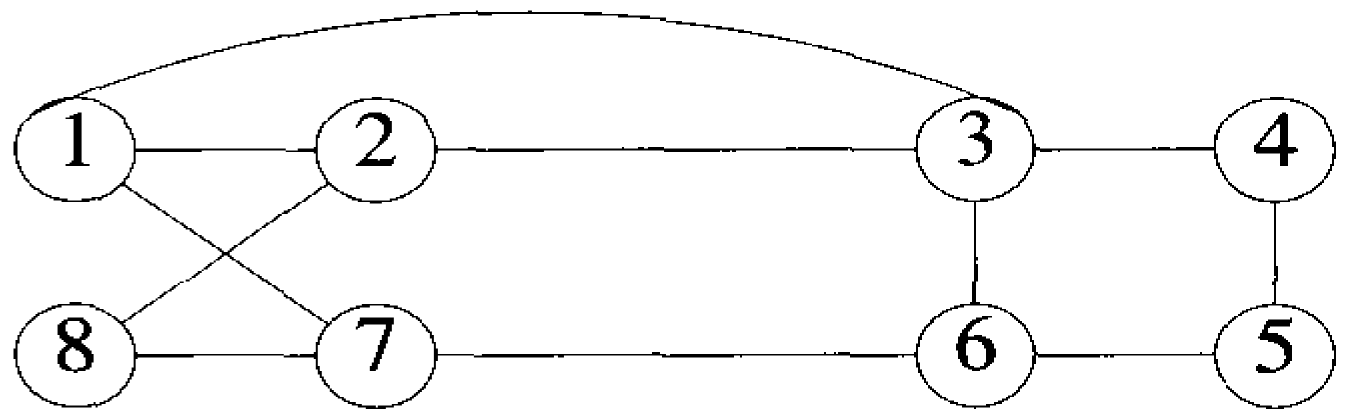
Hamiltonian Cycle Problem

Hamiltonian cycle: A cycle in a graph is said to be Hamiltonian cycle if it contains all the vertices of the graph and no vertex is repeated.

Statement: In Hamiltonian cycle problem, we have to find a Hamiltonian cycle present in the given graph.

Hamiltonian Cycle Problem

Example: Consider the following graph.



Find a Hamiltonian cycle in this graph.

Solution: Hamiltonian cycles are

(1, 3, 4, 5, 6, 7, 8, 2, 1)

(1, 2, 8, 7, 6, 5, 4, 3, 1)

Backtracking algorithm for Hamiltonian Cycle Problem

- The backtracking solution vector $(x_1, x_2, \ldots\ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle.

- First we choose, x_1, can be any vertex of the n vertices.

- We will take $x_1 = 1$. To avoid printing same cycle n-times, we require that $x_1 = 1$.

Backtracking algorithm for Hamiltonian Cycle Problem

- If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots\ldots, x_{k-1}$ and v is connected by an edge to x_{k+1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1.

- This algorithm is started by first initializing the adjacency matrix G[1:n][1:n], then x[2:n] to 0 and x[1] to 1 and then executing algorithm Hamiltonian(2).

- Time complexity of this algorithm = **$O(2^n n^2)$.**

Backtracking algorithm for Hamiltonian Cycle Problem

```
Algorithm Hamiltonian($k$)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix $G[1:n, 1:n]$. All cycles begin at node 1.
{
    repeat
    { // Generate values for $x[k]$.
        NextValue($k$); // Assign a legal next value to $x[k]$.
        if ($x[k] = 0$) then return;
        if ($k = n$) then write ($x[1:n]$);
        else Hamiltonian($k + 1$);
    } until (false);
}
```

Backtracking algorithm for Hamiltonian Cycle Problem

```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k]=0$. If $k=n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(n+1)$; // Next vertex.
        if $(x[k]=0)$ then return;
        if $(G[x[k-1], x[k]] \neq 0)$ then
        { // Is there an edge?
            for $j := 1$ to $k-1$ do if $(x[j]=x[k])$ then break;
                                  // Check for distinctness.
            if $(j=k)$ then // If true, then the vertex is distinct.
                if $((k<n)$ or $((k=n)$ and $G[x[n], x[1]] \neq 0))$
                    then  return;
        }
    } until (false);
}
```

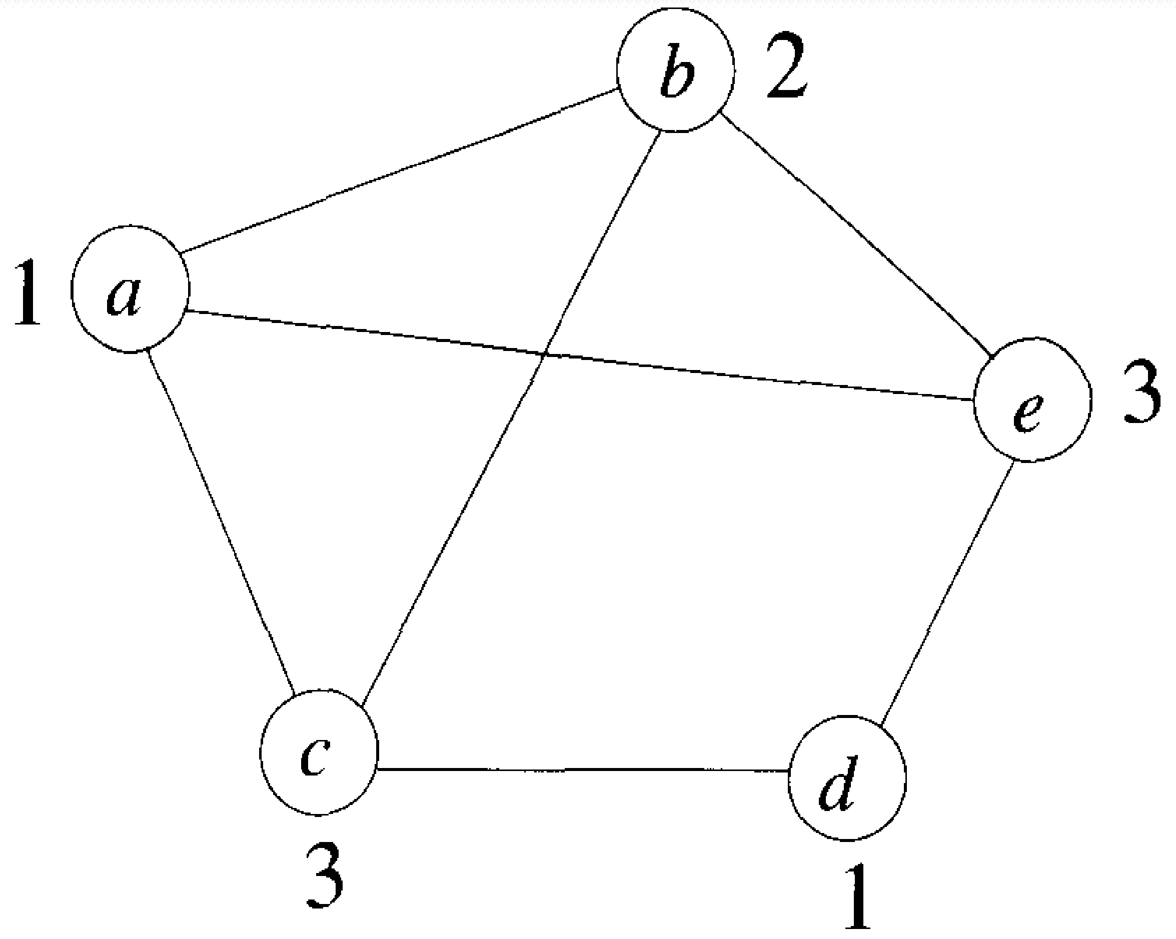
Graph Coloring Problem

- Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the **m-colorability decision problem.**

- The **m-colorability optimization problem** asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the **chromatic number** of the graph.

Graph Coloring Problem

Example: Consider the following graph



Find the chromatic number of this graph.

Backtracking algorithm for Graph Coloring Problem

- Following algorithm determines all the different ways in which a given graph can be colored using at most m colors.

- In this algorithm, a graph is represented by its adjacency matrix G[1:n, 1:n]. The colors are represented by the integers 1, 2, ……., m and the solutions are represented by the n-tuple (x_1, x_2, ……., x_n), where x_i is the color of node i.

- Function **mColoring** is begun by first assigning the graph to its adjacency matrix, setting the array x[] to 0, and then invoking the statement **mColoring(1)**.

Backtracking algorithm for Graph Coloring Problem

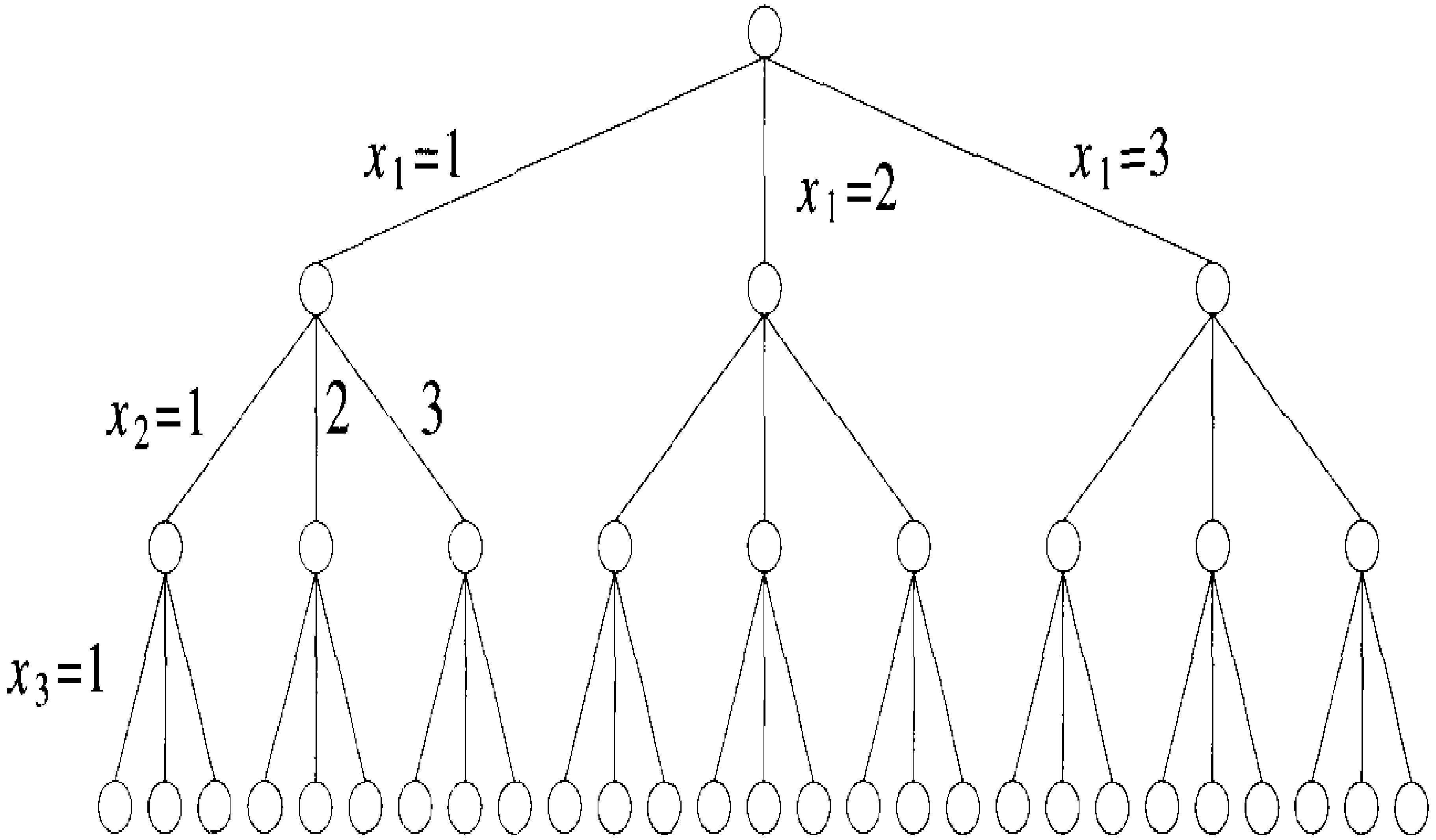
```
mColoring(k)
{
        while(1)
        {
                NextValue(k)
                if (x[k] = 0)
                        return
                if(k=n)
                        print x[1:n]
                else
                        mColoring(k+1)
        }
}
```

Backtracking algorithm for Graph Coloring Problem

```
NextValue(k)
{
    while(1)
    {
        x[k] ← (x[k] + 1) mod (m+1)
        if (x[k] = 0)
            return
        for ( j← 1  to  n )
            if(G[k,j] ≠ 0 and (x[k] = x[j]))
                break
        if (j=n+1)
            return
    }
}
```

Backtracking algorithm for Graph Coloring Problem

State space tree when n = 3 and m = 3 is



Backtracking algorithm for Graph Coloring Problem

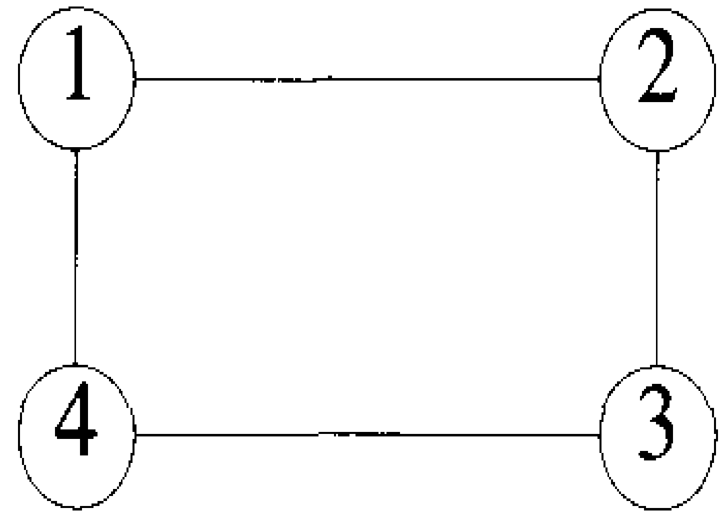
- An upper bound on the computing time of mColoring can be arrived at by the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$.
- At each internal node, O(mn) time is spent by NextValue to determine thechildren corresponding to legal colorings. Hence the total time is bounded by

$$\sum_{i=0}^{n-1} m^{i+1} n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - m)/(m-1)$$

$$= O(nm^n)$$

Backtracking algorithm for Graph Coloring Problem

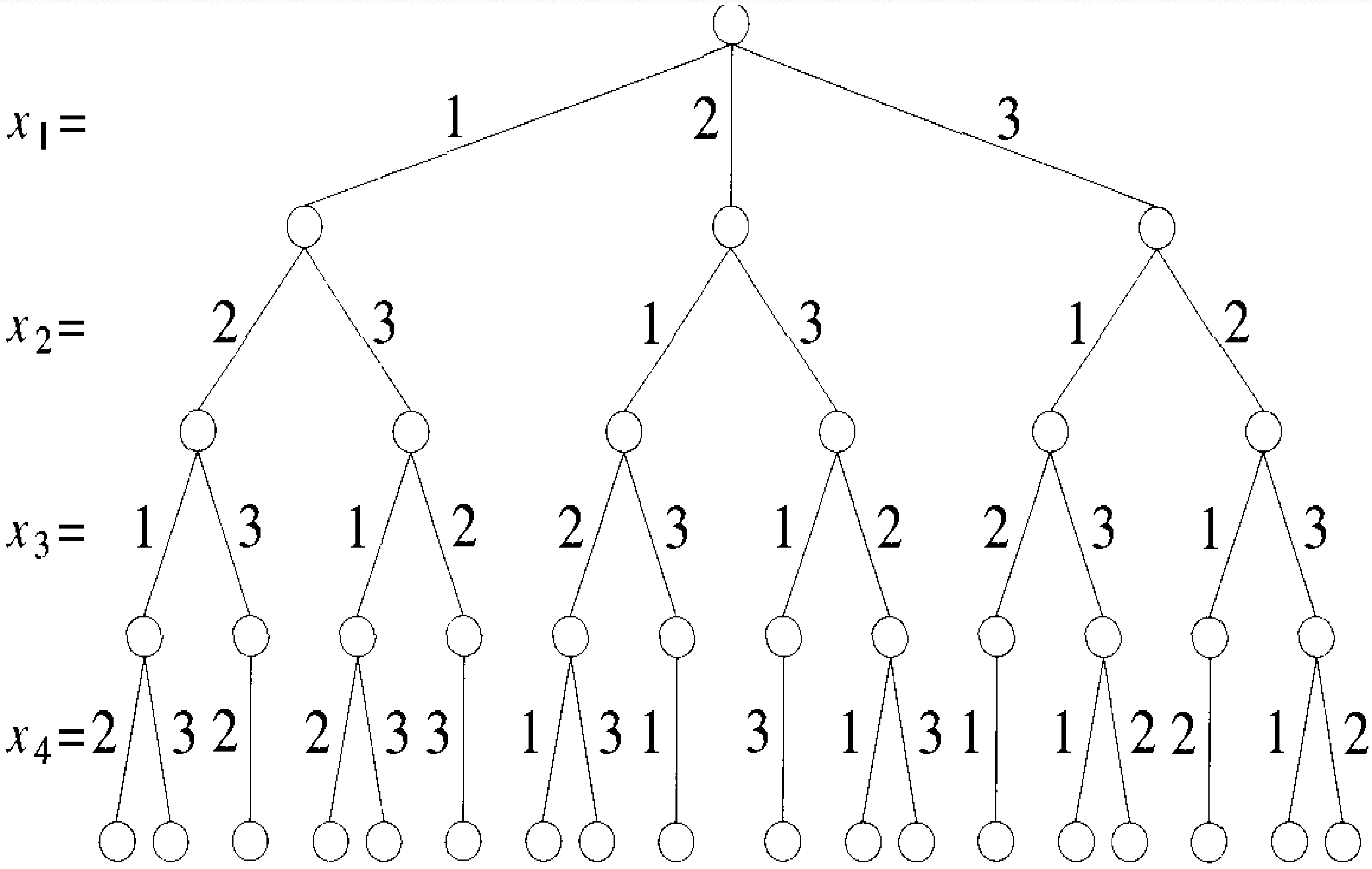
Example: Consider the following graph:-



- Here m = 3.
- Draw state space tree for this graph using mColoring.

Backtracking algorithm for Graph Coloring Problem

Solution: State space tree for the graph will be



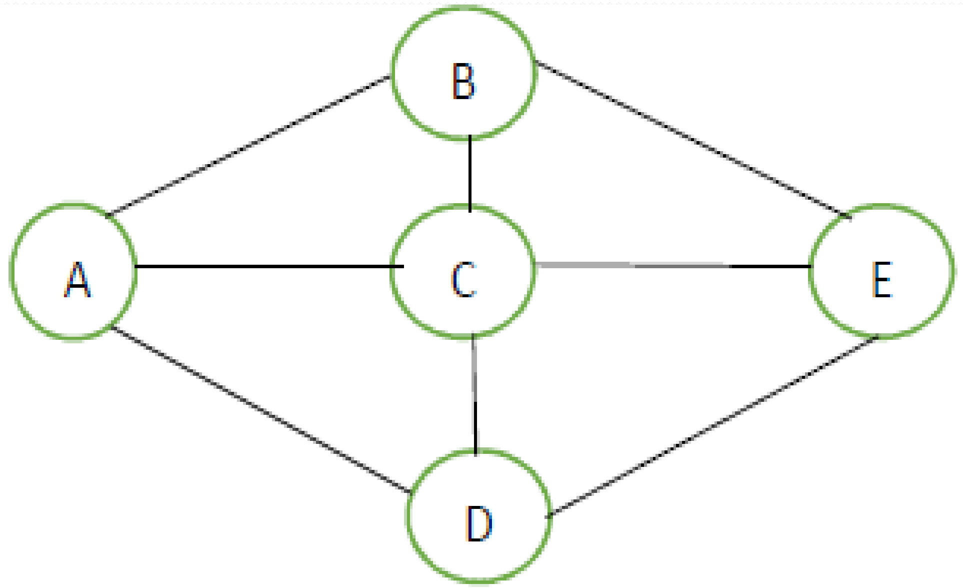
AKTU Examination Questions

1. Explain application of graph coloring problem.
2. Solve the Subset sum problem using Backtracking, where

   n=4, m=18, w[4] = {5, 10, 8, 13}
3. Define Graph Coloring.
4. What is backtracking? Discuss sum of subset problem with the help of an example.
5. Explain Implicit and Explicit constraints of N-queen Problem.

AKTU Examination Questions

6. What is the difference between Backtracking and Branch & Bound? Write Pseudo code for Subset Sum Problem using Backtracking. Give example for the same.
7. Consider a graph G=(V,E). We have to find a Hamiltonian cycle using backtracking method.



Branch and Bound

Branch and Bound

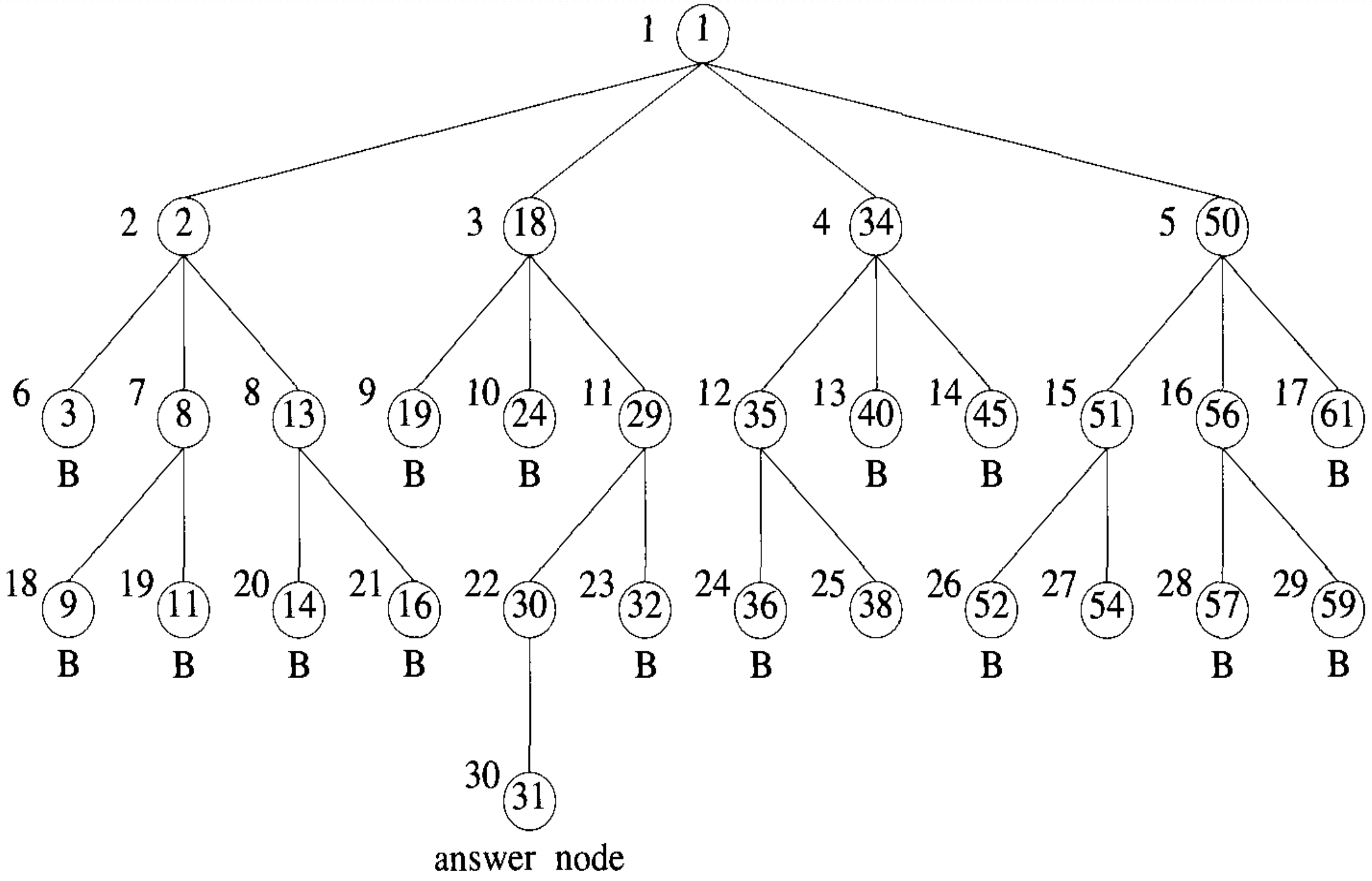
- A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution.
- Branch and bound is a systematic method for solving optimization problems. B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case.

Branch and Bound

- Branch and bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.
- There are two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- In branch and bound terminology, a BFS-like state space search will be called FIFO(First In First Out) search as the list of live nodes is a first-in-first-out list (or queue).
- A D-search-like state space search will be called LIFO(Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack).

Example: Consider the 4-queen problem.

FIFO branch-and-bound algorithm to search the state space tree for this problem.



Least Cost (LC) Search Branch and Bound(LCBB)

- In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind.
- The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- The search for an answer node can often be speeded by using an "intelligent" ranking function c’ for live nodes.
- The next E-node is selected on the basis of this ranking function.
- A search strategy that uses a cost function c’(x) to select the next E-node would always choose for its next E-node a live node with least c’. Hence, such a search strategy is called an LC-search(Least Cost search).

Travelling Salesperson Problem

In this problem, we have given a directed graph with cost matrix.

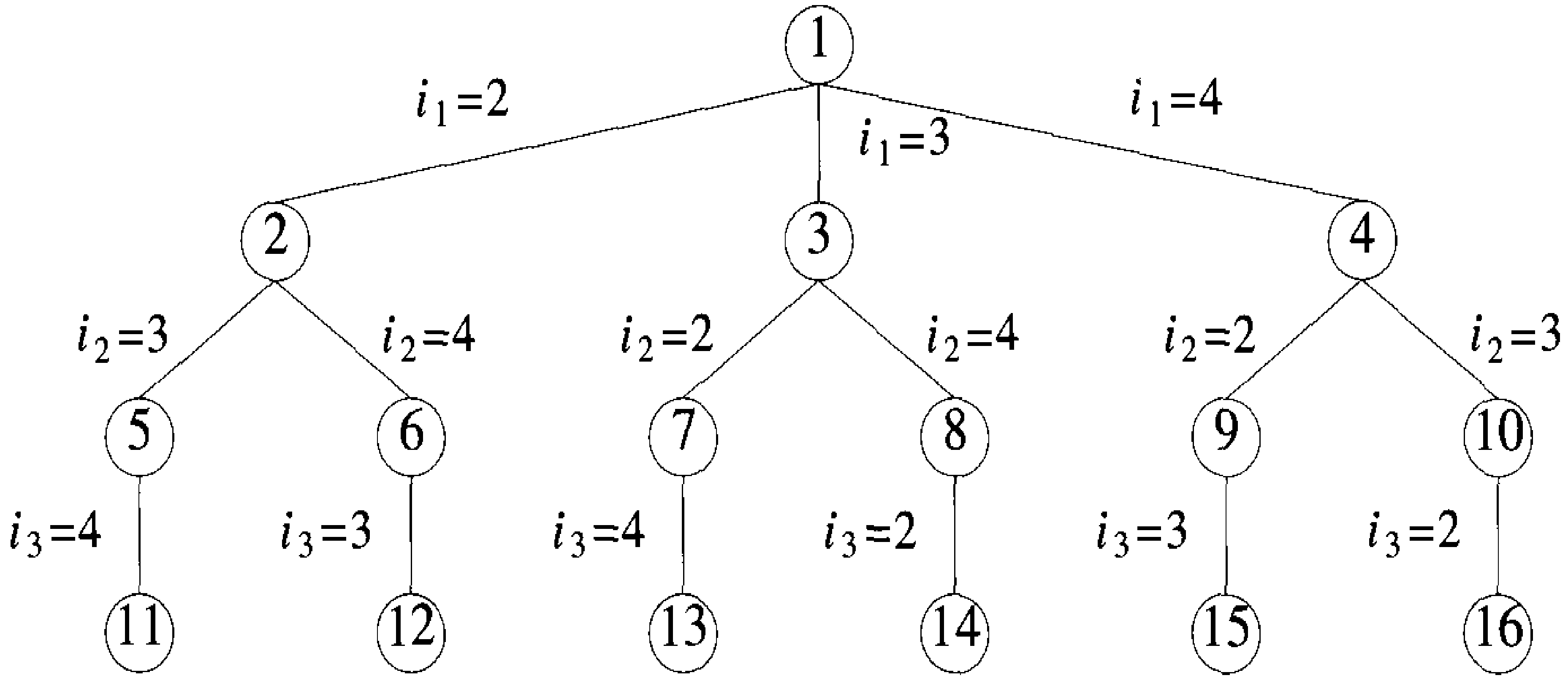
We have to find a shortest tour of the graph which contains all the vertices of the graph with no vertex repeated except first and last vertex.

Travelling Salesperson Problem

- Let G = (V, E) be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} denote the cost of edge(i, j), $c_{ij} = \infty$ if $(i,j) \notin E$, and let $|V| = n$.
- Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by S= {1, **π**, 1 ! **π** is a permutation of (2,3,... , n)}.Then $|S| = (n-1)!$.
- The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $(i_j, i_{j-1}) \in E$, $0 \leq j \leq n-1$ and $i_0 = i_n = 1$.
- S can be organized into a state space tree.

Travelling Salesperson Problem

Following figure shows the tree organization for the case of a complete graph with |V| = 4.



State space tree for the traveling salesperson problem with n = 4 and $i_o = i_4 = 1$

Travelling Salesperson Problem

- Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, $i_3 = 2$, and $i_4 = 1$.

Reduced row or column:

A row (column) is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.

Reduced cost matrix:

A matrix is said to be reduced iff every row and column is reduced.

Travelling Salesperson Problem

- To solve the travelling salesperson problem using LCBB method, we construct state space tree.
- We compute reduced cost matrix at each node of the state space tree. And also we compute a cost function c' at each node.
- Reduced cost matrix of root node is computed directly from given cost matrix of the graph.
- If C is the cost matrix of the graph, then reduced cost matrix of C is the reduced cost matrix corresponding to root node.
- If c' represent the cost function at each node to determine the optimal tour, then

c' for root node = Total amount subtracted in determining reduced cost matrix at root node

Travelling Salesperson Problem

Reduced cost matrix of the remaining node is computed in the following way:-

Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge (i,j) in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows:

(1) Change all entries in row i and column j of A to ∞.

(2) Set A(j,1) to ∞.

(3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

If r is the total amount subtracted in the step (3), then

$$c'(S) = c'(R) + A(i,j) + r$$

Travelling Salesperson Problem

Example: Solve the following travelling salesperson problem whose cost matrix is the following:-

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Solution: The reduced cost matrix corresponding to root node will be the following:-

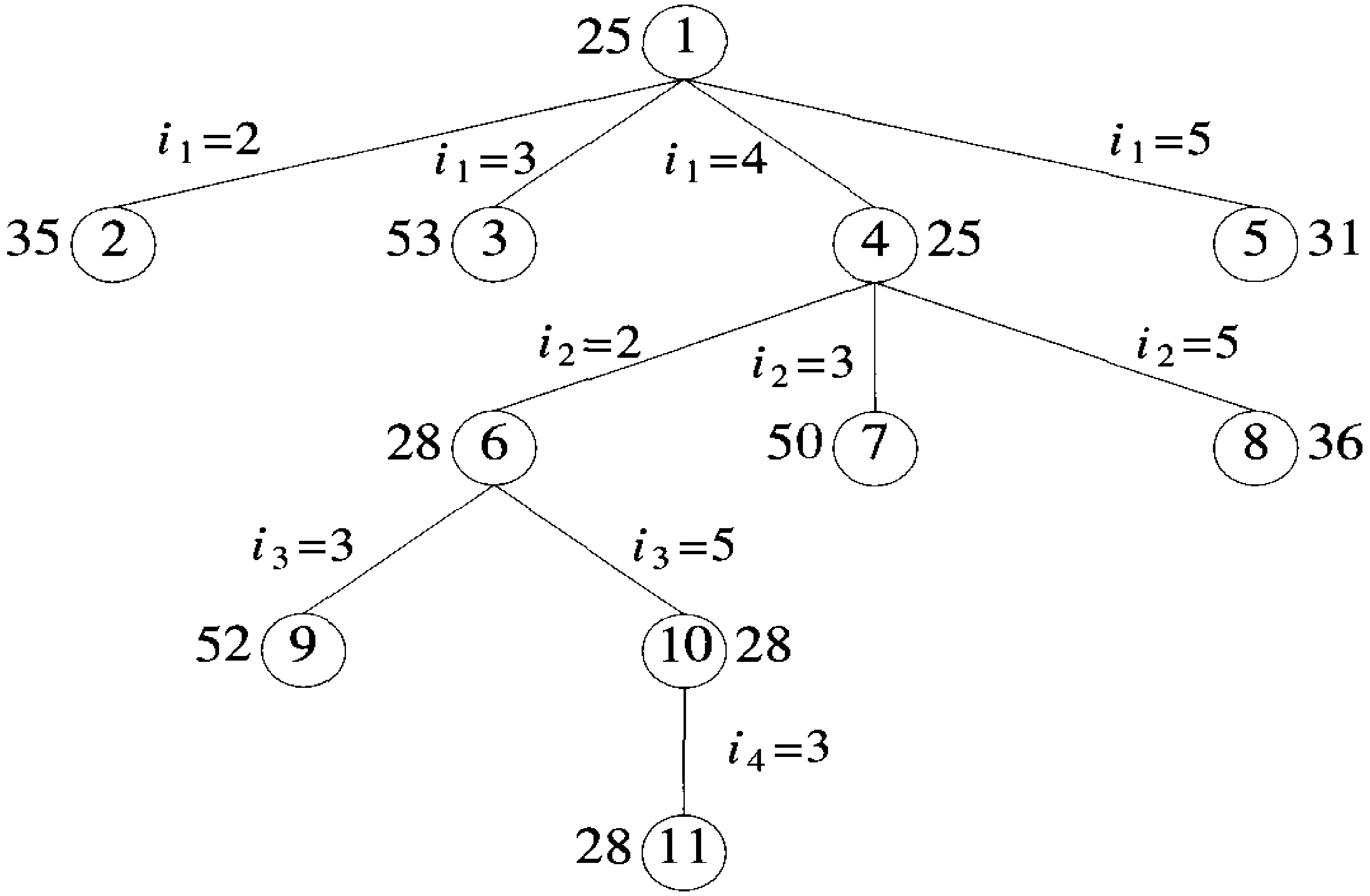
$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

And value of cost function at root node is

c'(root) = 25

Travelling Salesperson Problem

The portion of the state space tree that gets generated is



$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

AKTU Examination Questions

1. Differentiate between Backtracking and Branch and Bound Techniques.
2. How BFS is differ from DFS.
3. Consider I=<I1,I2,I3>; W=<5,4,3>; V=<6,5,4> and W=7, we have to pack this knapsack using the branch and bound technique.