SPPU

unit 4

Backtracking & branch n bound

BACKTRACKING

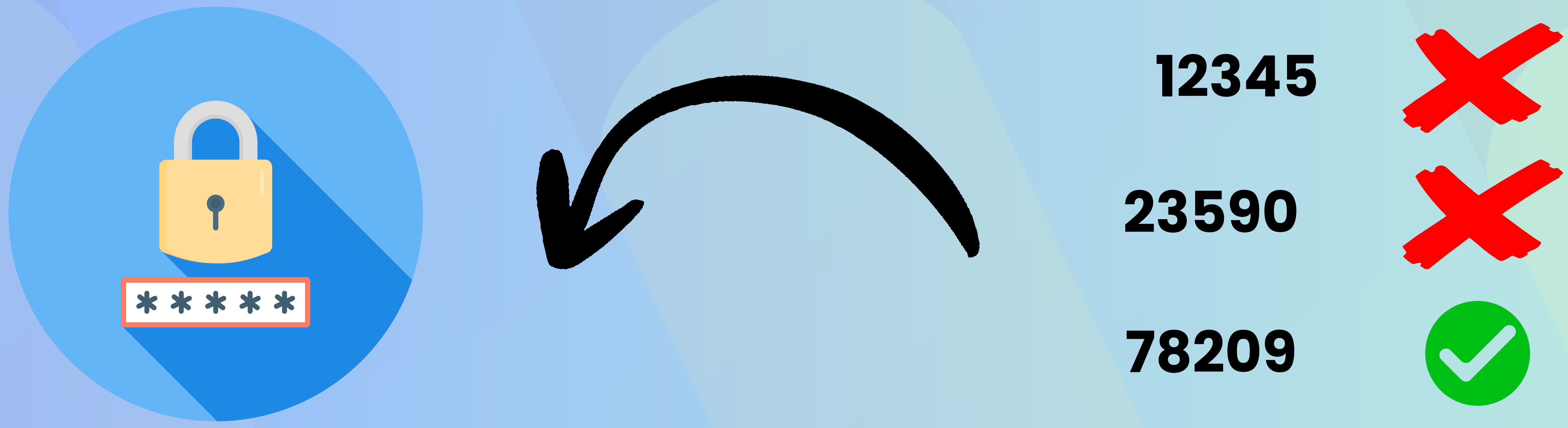
DEFINITION

- Backtracking is a problem-solving technique that builds solutions step-by-step, and undoes steps (backtracks) as soon as it becomes clear that the current path cannot lead to a valid solution.

KEY IDEA

- ✓ Try a choice
- ✓ If it works → continue
- ❌ If it fails → undo the choice & try another option

EXAMPLE



WHEN IS BACKTRACKING USED?

- 🧩 Puzzles (Sudoku, Maze, Crossword)
- 🔐 Constraint satisfaction problems
- 📍 Searching in a large solution space
- 📑 Optimization problems

WHY BACKTRACKING IS USEFUL?

- ⭐ Avoids checking every possible solution blindly
- ⭐ Saves time by eliminating wrong paths early
- ⭐ Systematic and guaranteed to find solution if it exists

CONTROL ABSTRACTION

GENERAL CONTROL ABSTRACTION (LOGIC OF BACKTRACKING)

```
  Algorithm Backtrack(PartialSolution)
if PartialSolution is a complete and valid solution:
    record solution
    return

for each possible choice:
    if choice is feasible:
        add choice to PartialSolution
        Backtrack(PartialSolution)      // explore further
        remove choice from PartialSolution  // BACKTRACK
```

EXPLANATION OF CONTROL ABSTRACTION

✓ A partial solution is gradually extended
✓ Every step must satisfy feasibility (constraints)
✓ If a choice leads to an invalid state → undo the step (backtrack)
✓ Continue exploring until all valid solutions are generated

TIME ANALYSIS

Backtracking is a brute-force search technique;
therefore, its time performance is generally poor in the worst case.

FACTORS AFFECTING TIME COMPLEXITY

The running time depends mainly on: Number of choices (branching factor m) at each step
Depth / number of decision levels (n) in the search tree

DEFINITIONS :-

1. STATE SPACE

The complete set of all possible states (configurations) through which a problem may move to reach the final solution.

Example:

In a maze game, every possible position inside the maze is part of the state space.

2. EXPLICIT CONSTRAINTS

Constraints that are clearly stated in the problem and must be satisfied.

Example:

In the N-Queens problem, no two queens can be in the same row or column — this rule is explicitly stated.

3. IMPLICIT CONSTRAINTS

Constraints that are not directly mentioned but must be satisfied to make the solution meaningful.
Example:
In N-Queens, no two queens should attack each other diagonally — this condition is not always explicitly written, but it must be satisfied.

4. PROBLEM STATE

Any intermediate step or configuration during the solution process.
Example:
Placing 3 queens out of 8 on the chessboard is a problem state (partial solution).

5. SOLUTION STATE

A state that satisfies all constraints of the problem.
Example:
A chessboard arrangement with all 8 queens placed legally (no conflicts) is a solution state.

6. ANSWER STATES

Solution states that are stored or returned as final valid results.
Example:
If 92 valid arrangements exist for 8-Queens, each of those 92 arrangements is an answer state.

7. LIVE NODE

A node (state) that can still be expanded because it might lead to a solution.
Example:
In Sudoku, a partially filled board that still has valid moves left is a live node.

8. E-NODE (EXPANSION NODE)

The live node currently selected for expansion — generating its next states/children.
Example:
If solving Sudoku and the solver is currently filling numbers into row 3, that partially filled board is the E-node at that moment.

9. DEAD NODE

A node that cannot lead to a valid solution, so it is not expanded further.
Example:
In Sudoku, if a placement causes a repetition in a row/column/subgrid, that board becomes a dead node.

10. DEAD NODE

A function used to identify and discard non-promising states early to reduce search time.
Example:
In the Traveling Salesman Problem (TSP), if the current travel cost already exceeds the best known path cost, the bounding function rejects that node — no further exploration needed.

GRAPH COLORING

WHAT IS GRAPH COLORING?

Graph coloring is the process of assigning colors to the vertices of a graph such that:

- No two adjacent vertices have the same color (implicit constraint)
- The number of colors is minimized or within a given limit

WHY BACKTRACKING FOR GRAPH COLORING?

Backtracking is used because: ✔ We explore color assignments step by step
✔ If any assignment violates a constraint → undo the assignment (backtrack)
✔ Finally, we reach a valid coloring if it exists

STEPS TO SOLVE GRAPH COLORING USING BACKTRACKING

- Start with the first vertex
- Assign the first valid color (which does not conflict with already-colored adjacent vertices)
- Move to the next vertex and try valid colors recursively
- Backtrack if needed

  If no valid color exists for the current vertex → change the color of the previous vertex
- Stop when all vertices are colored successfully

b) Explain the backtracking with graph coloring problem. Find solution for following graph **[8]**

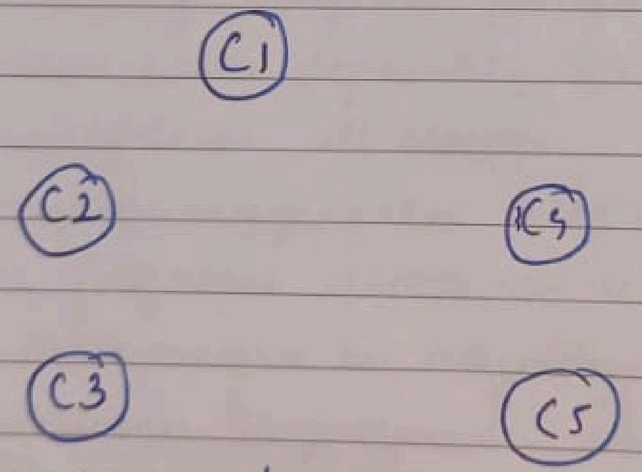
	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

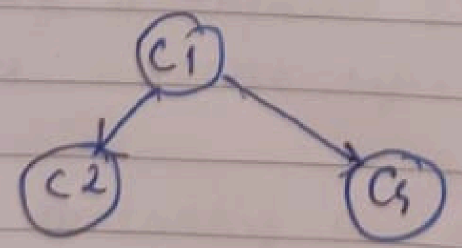
	C1	C2	C3	C4	C5
C1	0	1	0	1	0
C2	1	0	1	0	0
C3	0	1	0	1	1
C4	1	0	1	0	1
C5	0	0	1	0	0

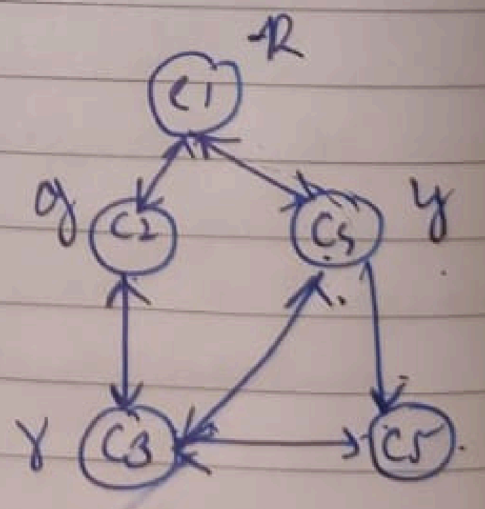
→ Here 1 represent there is an edge present between the ~~that~~ two vertices.
0 represent no edge is present.

Constructing the graph as follows :-



drawing the edges from nodes.





Now, choosing the colors as :-
red, green, yellow.

red → 1
green → 2
yellow → 3

Starting with first vertex

C1 → colors = [1, 0, 0, 0, 0]
C1

C2 → colors = [1, 2, 0, 0, 0]
as C2 is adjacent to C1, we cant assign same color.

C3 → colors = [1, 2, 1, 0, 0]
as C3 is adjacent to C2 & C5

C4 → colors = [1, 2, 1, 3, 0]

$\boxed{\text{C5} \to \text{colors} = [1, 2, 1, 3, 2]}$

SUM OF SUBSETS PROBLEM

DEFINITION

The Sum of Subsets problem is to find all subsets of a given set whose elements add up to a specific target sum.
It is a combinatorial search problem that can be efficiently solved using backtracking.

PROBLEM STATEMENT

Given:
- A set of integers
- A target sum (T)

Goal:
✓ Find all possible subsets whose sum = T

WHY BACKTRACKING ?

Backtracking allows us to: ✔ Explore including/excluding elements one by one
❌ Undo choices that exceed the target (pruning)
✔ Continue until all valid subsets are found

STEPS TO FOLLOW

- Start with an empty subset and the target sum
- At each step → either include or exclude the current element
- If the subset sum equals target → record/print subset
- If the sum exceeds target or elements are finished → backtrack
- Continue until all subsets are explored

Example 4.6.1 *Consider a set* $S = \{5, 10, 12, 13, 15, 18\}$ *and* $d = 30$. *Solve it for obtaining sum of subset.*

$S = \{5, 10, 12, 13, 15, 18\}$ $d = 30$ (sum = 30)

1: Empty subset : [ ]
sum = 0   index = 0

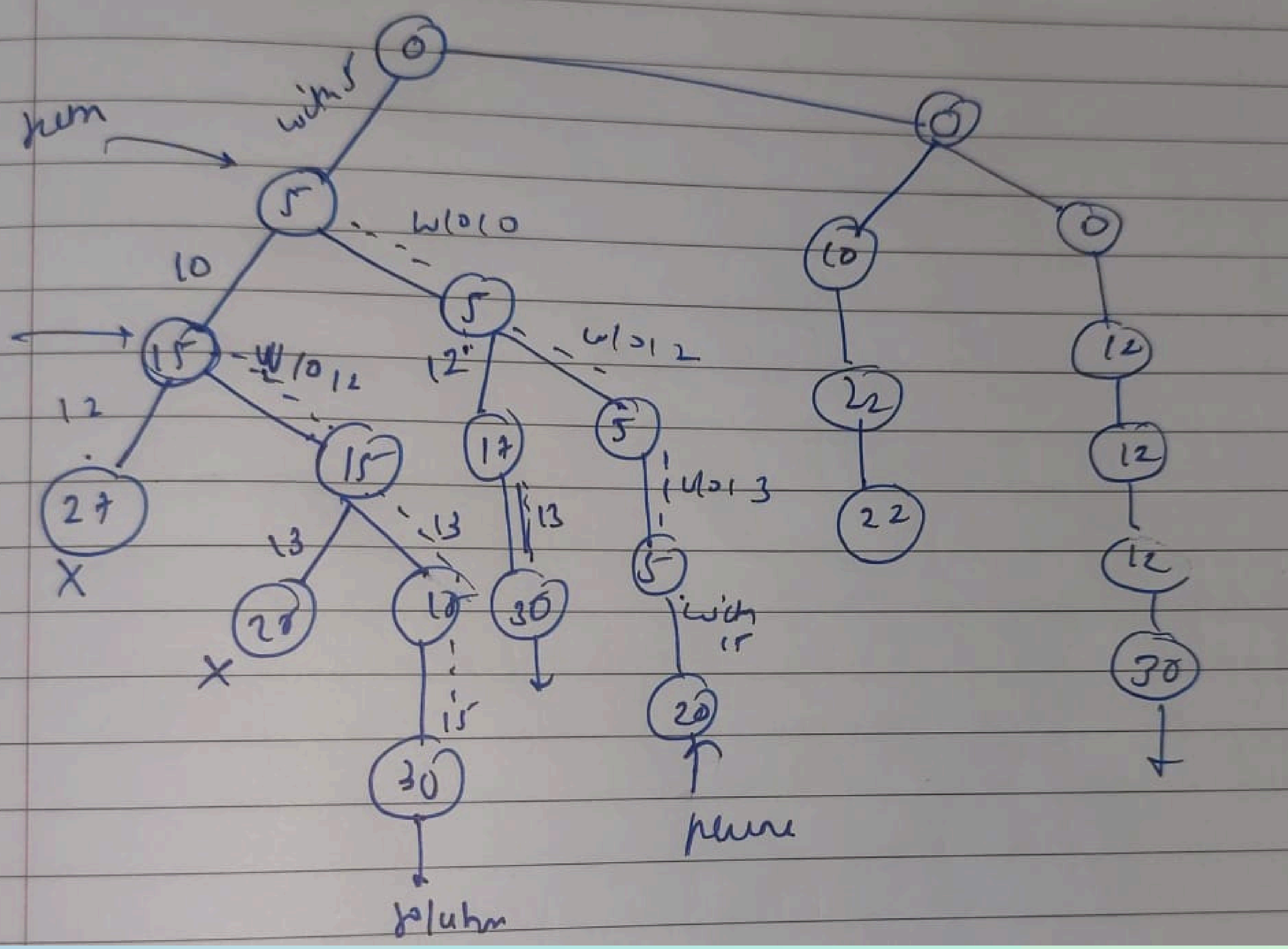
2] Creating a table :-

Subset = [ ]	sum	decision
[ ]	0	start
[5]	5	include
[5,10]	15	include
[5,10,12]	27	include
[5,10,12,13]	40	backtrack
[5,10,12,15]	42	backtrack
[5,10,12,18]	45	backtrack
[5,10]	[illegible]	
[5,10,13]	28	
[5,10,13,15]	33	backtrack
[5,10,~~13~~]		
[5,10,15]	30	solution obtained

Valid subsets are
[5, 10, 15]

State space tree

$\{5, 10, 12, 13, 15, 18\}$



BRANCH AND BOUND

- General algorithmic method for finding optimal solutions in optimization problems
- Used when Greedy method and Dynamic Programming fail
- Often slower and can lead to exponential time complexity in worst cases
- Can perform reasonably fast on average if applied carefully

KEY CONCEPT

- Builds a state space tree to search for the optimal solution
- Uses bounding functions to evaluate nodes
- Prunes / eliminates branches that cannot lead to an optimal solution
- Focuses computation only on promising nodes

TERMINOLOGY

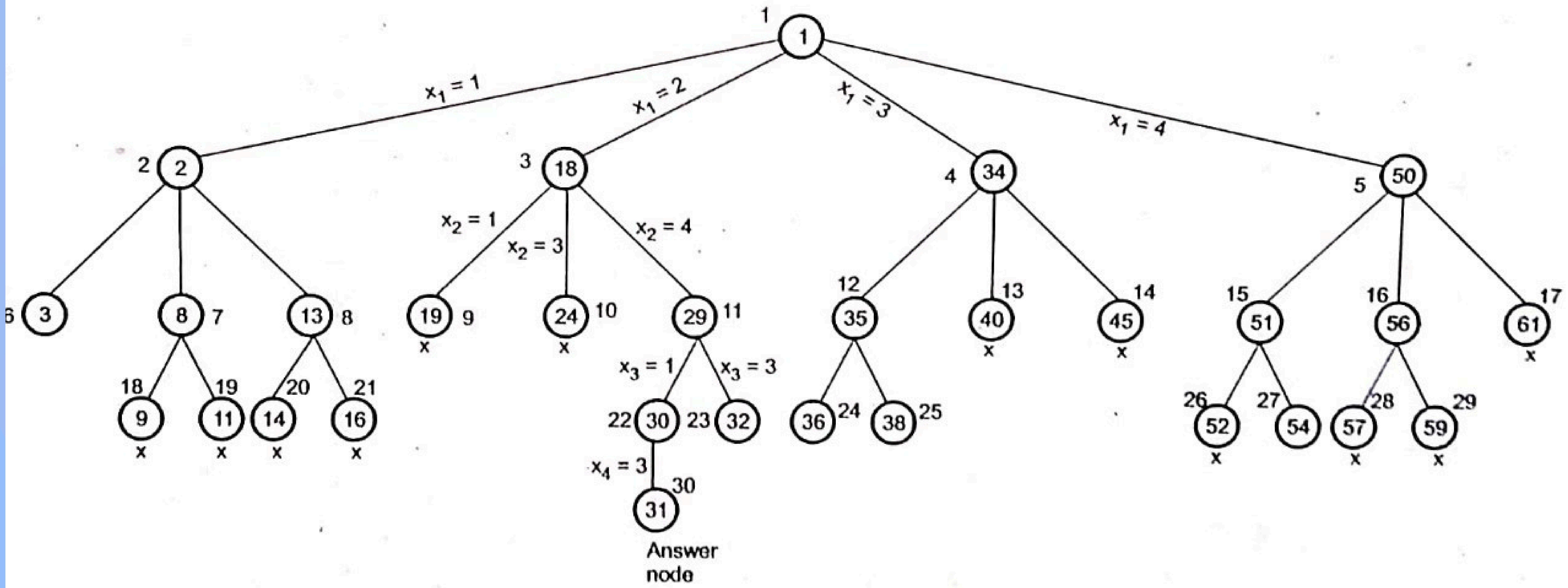
- Bounding Function → Determines whether a node is worth exploring
- E-Node (Expanding Node) → Node with best / optimum bound selected for expansion
- Pruning → Rejecting nodes that violate bound or are not promising

EXAMPLE: 4-QUEENS PROBLEM

- Branch and Bound can be applied to place 4 queens on a chessboard such that no two queens attack each other

1. Bounding function eliminates:
2. Row conflicts
3. Column conflicts
4. Diagonal conflicts

- Nodes expanded in order of best bound until a solution node is reached


x1 = 1
x1 = 2
x1 = 3
x1 = 4
x2 = 1
x2 = 3
x2 = 4
x3 = 1
x3 = 3
x4 = 3
Answer node

BOUNDING

CONCEPT OF BOUNDING

- Bounding helps avoid exploring sub-trees that cannot lead to an optimal solution
- A bounding function is used at every node to estimate the best possible solution that can come from that node
- Lower bound ($\hat{c}(x)$) and upper bound values are calculated to decide whether a node should be expanded or pruned

HOW BOUNDING WORKS

A cost function $\hat{c}(x)$ is used to compute the lower bound of the solution from node x

If the lower bound > current upper bound, the node is pruned (killed) because it cannot lead to a better solution

Upper bound represents the cost of the best solution found so far

UPDATING THE UPPER BOUND

Initially, the upper bound is set to ∞ (or a very large value)

When a new solution / answer node is found, the upper bound is updated with its cost

This keeps reducing the search space by eliminating nodes with cost higher than the updated upper bound

EXAMPLE:-

Knapsack Capacity = 10 kg

Items:

Item	Weight	Profit
A	6	30
B	3	14
C	4	16

Bounding at Root Node

We consider all items → Maximum possible profit = 30 + 14 + 16 = 60

So Upper Bound (UB) = 60

Branch from Root

- Include item A

Weight = 6 kg → Remaining = 4 kg

We try to add best remaining item B or C

Max possible profit = 30 + 16 = 46 → UB = 46

- Exclude item A

Items left: B & C

Max possible profit = 14 + 16 = 30 → UB = 30

Bounding / Pruning Decision

We always expand the node with higher bound

So expand the node including A (UB = 46)

If any child node's bound < current best (upper) → ❌ Prune

TRAVELLING SALESMAN PROBLEM

Definition

Travelling Salesman Problem (TSP) is a problem in which a salesman must visit every city exactly once and return to the starting city, and the objective is to find the shortest possible route that completes this tour.

Example 4.11.1 *Apply the branch and bound algorithm to solve the TSP for the following cost matrix.*

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix} \begin{matrix} \rightarrow 6 \\ \rightarrow 3 \\ \rightarrow 4 \\ \rightarrow 5 \\ \rightarrow 5 \end{matrix}$$

23

a) row reduction :-
find minimum ele from each row & subtract to get final matrix after subtraction

$$\begin{bmatrix} \infty & 5 & 4 & 3 & 0 \\ 5 & \infty & 4 & 0 & 1 \\ 4 & 0 & \infty & 0 & 4 \\ 6 & 5 & 0 & \infty & 0 \\ 1 & 4 & 0 & 0 & \infty \end{bmatrix}$$

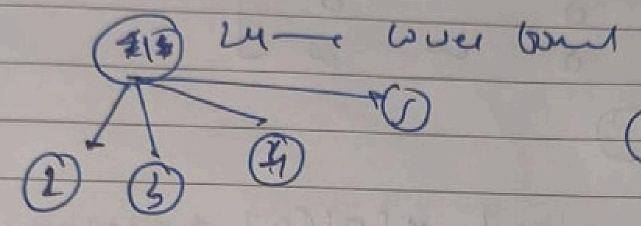
$$\begin{matrix} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & \times & \times & \times & \times \end{matrix} \longrightarrow 1$$

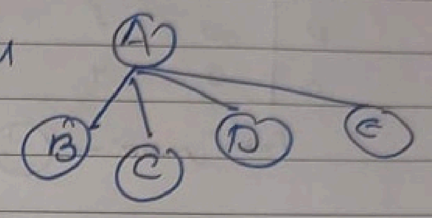
b) column reduction
Same for coln. If any coln contain 0 then ignore that coln & a fully reduced matrix can be obtained

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \begin{bmatrix} \infty & 5 & 4 & 3 & 0 \\ 4 & \infty & 4 & 0 & 1 \\ 3 & 0 & \infty & 0 & 4 \\ 5 & 5 & 0 & \infty & 0 \\ 0 & 4 & 0 & 0 & \infty \end{bmatrix} \longrightarrow M1$$

Total reduced cost = total reduced row cost + total reduced coln cost
$23 + 1 = 24$

Set 24 as optimum cost





now $A \to B$ — coln ∞ and $B \to A \Rightarrow \infty$

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 4 & 0 & 1 \\ 3 & \infty & \infty & 0 & 4 \\ 5 & \infty & 0 & \infty & 0 \\ 0 & \infty & 0 & 0 & \infty \end{bmatrix} \begin{matrix} \text{ign} \\ \text{i} \\ \text{i} \\ \text{i} \\ \text{i} \end{matrix}$$

min ignore all.

total reduced cost for node B is ⇒
optimum cost + old value of M[A][B]
$24 + 5$
29

Now $A \to C$

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 4 & \infty & \infty & 0 & 1 \\ \infty & 0 & \infty & 0 & 4 \\ 5 & 5 & \infty & \infty & 0 \\ 0 & 4 & \infty & 0 & \infty \end{bmatrix}$$

no minimum value
total cost → $24 + 4 = 28$.

Now A → D

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array}\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 4 & \infty & 4 & \infty & 1 \\ 3 & 0 & \infty & \infty & 4 \\ \infty & 5 & 0 & \infty & 0 \\ 0 & 4 & 0 & \infty & \infty \end{bmatrix}$$

minimum → 1

total ⇒ optimum + old M[A][D] + minimum new cost

$24 + 3 + 1$

$\rightarrow 28$.

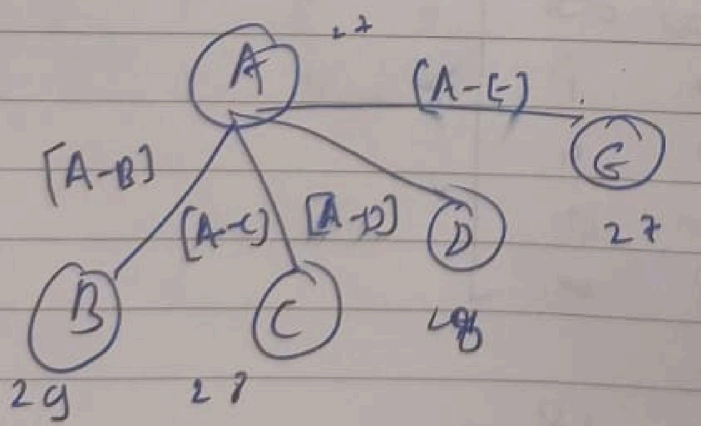
A → E

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array}\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 4 & \infty & 4 & 0 & \infty \\ 3 & 0 & \infty & 0 & \infty \\ 5 & 5 & 0 & \infty & 4 \\ \infty & 4 & 0 & 0 & \infty \end{bmatrix}$$
$$\begin{array}{c} \uparrow \\ 3 \end{array}$$

col^n minimum = 3

$24 + 3 + 0$

$= 27$



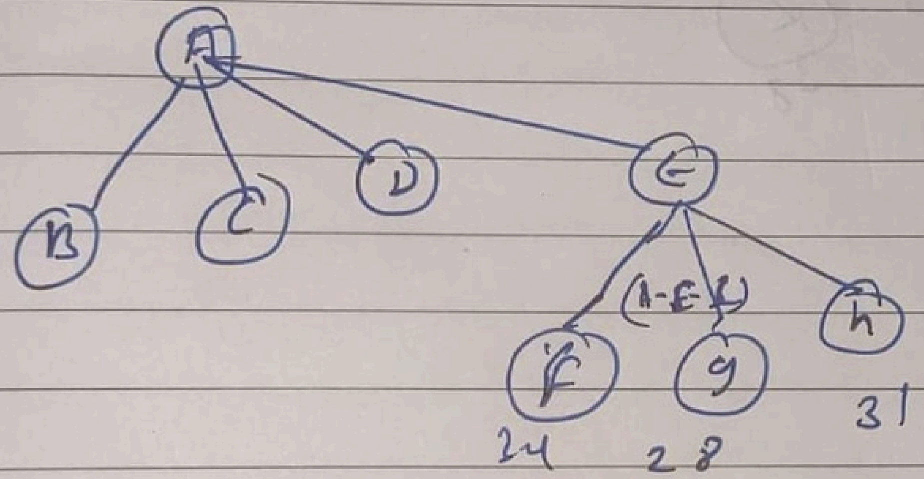
← select this for expansion as it is minimum

now consider (A, E, B)

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array}\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 4 & 0 & 1 \\ 3 & \infty & \infty & 0 & 4 \\ 5 & \infty & 0 & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$
$$\begin{array}{c} \uparrow \\ 3 \end{array}$$

$27 + 3 + 4 = 34$

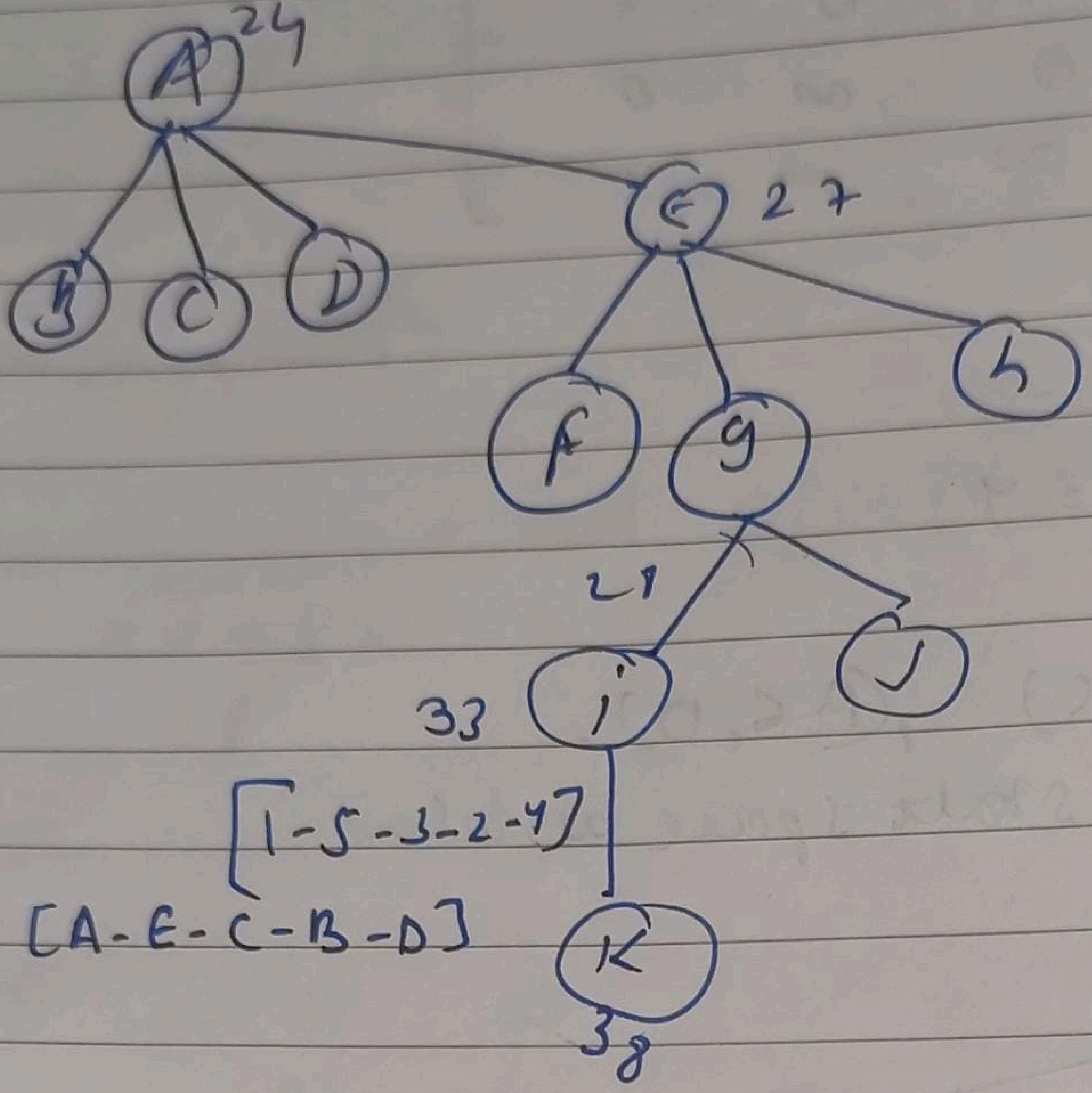
Same for (A, E, C), (A, E, D).
you will get state space as below



↑ select g for expansion.

Now, (A, E, C, B)
and so on

final state space tree will be



A
24
B
C
D
E
27
f
g
h
21
i
33
J
[1-5-3-2-4]
[A-E-C-B-D]
K
38