Unit-7. Backtracking and Branch & Bound

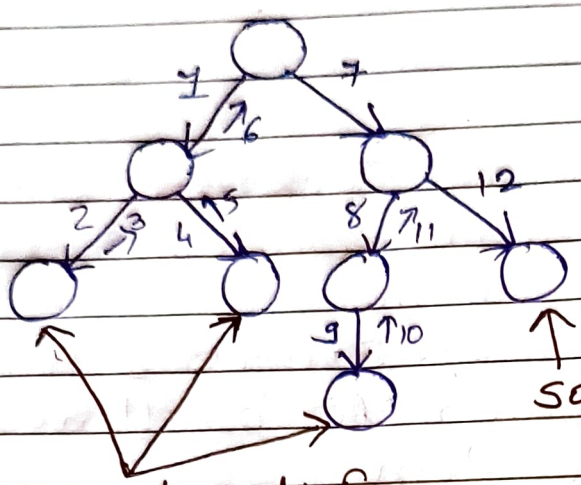
DOMS | Page No. 1
Date / /

* Topics :-

- Introduction of Back tracking
- The Eight queens problem
- Knapsack problem
- Introduction of Branch & Bound
- Travelling salesmen problem
- Minimax problem
- Assignment Problem
- Hamiltonial Problem
- Subset-sum Problem
- Greeph-colooring Problem

* Introduction of Backtracking :-

- Backtracking is recursive approach.
- Backtracking algorithms are generally used when we have set of choices, and we don't know which choice will lead to correct solution. Backtracking algorithm generates all partical candidates that could generate complete solution.
- Each choice leads to new set of partial solutions. Partial solutions are explored in DFS order.
- If partial solution does not satisfy the constraint, it will not be explored further. Algorithm backtracts from that point and explores the next possible candidate.



Does not satisfy constraints so don't explore further and backtrack

⇒ Applications of Backtracking :-

1. Sudoku
2. Crosswords
3. Knapsack problem
4. N-Queen problem
5. Logic programming language like Prolog.

⇒ How Backtracking works :- (Algorithm)

→ Step 1 : Return success if the current point is a viable solution.

→ Step 2 : Otherwise, if all paths have been exhausted, return failure because there is no feasible solution.

→ Step 3 : If the current point is not an endpoint, backtrack and explore other

points, then repeat the preceding steps

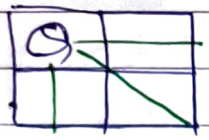
* The N-Queen Problem :-

- N-Queen problem is defined as; "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal."

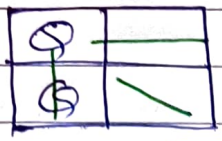
⇒ K-promising solution :- A solution is called k-promising if it arranges the k-queens such a way that, they can not threat each other.

⇒ for N = 1 :- This is trivial case.

⇒ for N = 2 :-



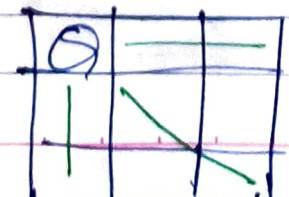
1- promising solution



No solution

So, we can not put 2 Queens on 2 x 2 chessboard.

⇒ for N = 3 :-



1- promising

2- promising

We can't put 3 queens on 3 x 3 chess board.

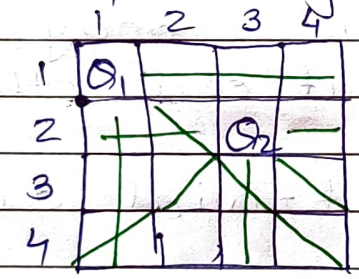
⇒ For N=4 :-

⇒ 4-Queen Problem :-

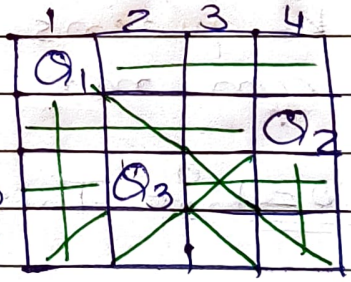
→ Given 4x4 chess board, arrange four queens in a way, such that no two queens attack each other.

→ That is, no two queens are placed in same row, column or diagonal.

- Now putting queens Q_1, Q_2, Q_3, Q_4.



No solution



No solution

- Let start with position (1,1). Q_1 is the only queen so there is no issue. partial solution is <1>.

- We cannot place Q_2 on position (2,1) or (2,2). Position (2,3) is accepteable. Partial solution is <1,3>.

- Q_3 cannot be placed on position (3,1) as Q_1 attacks her. And it cannot be placed at (3,2), (3,3) or (3,4) as Q_2 attacks her. There

is no way to put Q_3 in third row.

- Hence algorithm backtracks and goes back to previous solution and readjusts the position of queen Q_2. Q_2 is moved from position (2,3) to (2,4). Partial solution is <1,4>.

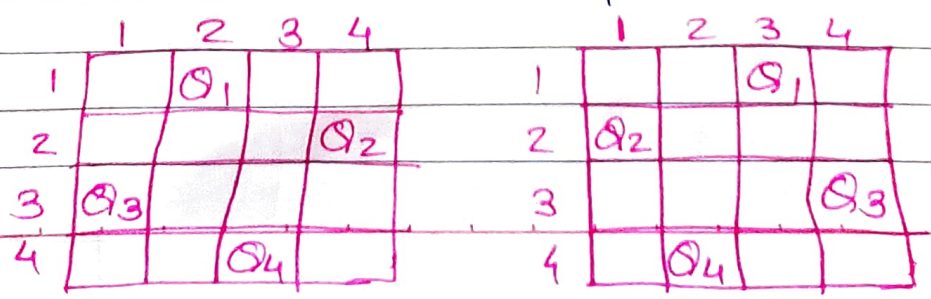
- Now, Q_3 can be placed at position (3,2). Partial solution is <1,4,3>.

- Queen Q_4 cannot be placed anywhere in row four. So again backtrack to previous solution and readjust the partial solution.

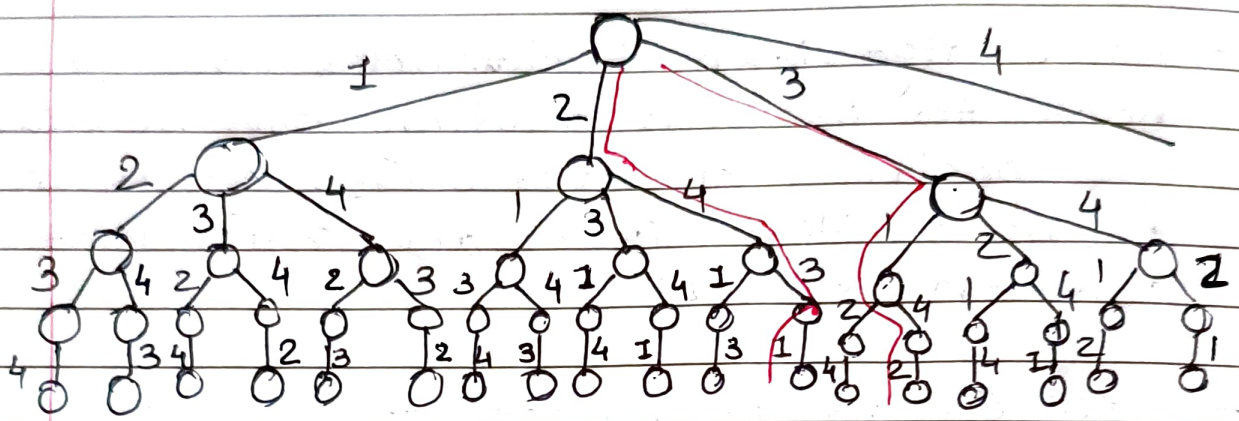
- Q_3 cannot be placed on (3,3) & (3,4). So algorithm backtracks even further.

- Now, all possible choices for Q_2 are already explored, hence algorithm goes back to partial solution <1> and move the queen Q_1 from (1,1) to (1,2). And this process continues until solution is found.

- All possible solutions for 4-queen are shown below:



⇒ State space tree for 4-Queen problem:-



→ Possible solutions are :- <2, 4, 3, 1>
<3, 1, 4, 2>

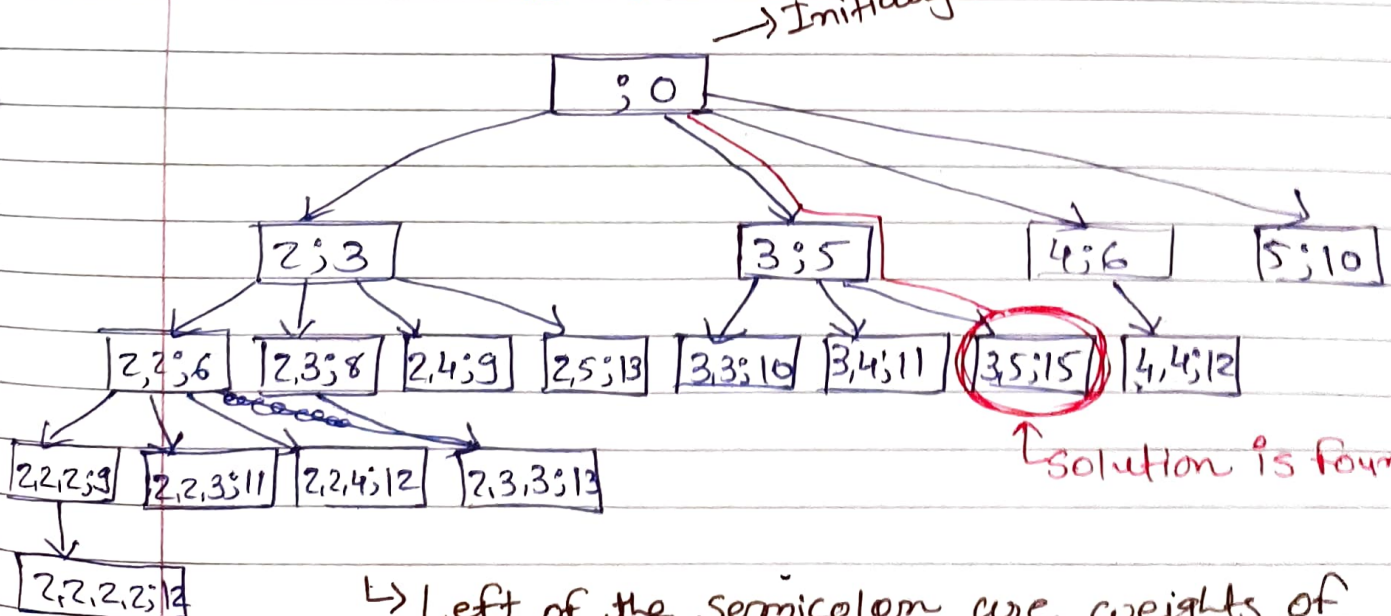
Number of Queens	Possible Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724

* Knapsack Problem :-

⇒ **Problem :-** Given set of items $I_1, I_2, \ldots$ In having weight $w_1, w_2, \ldots, w_n$ and value $v_1, v_2, \ldots, v_n$ fill the knapsack of capacity W such that total weight of items in knapsack should not exceed the knapsack capacity and same time maximize the profit.

- We may take an object or leave behind, but we may not take fraction of an object.

⇒ **Example :-** $w = (2, 3, 4, 5)$
$V = (3, 5, 6, 10)$
$W = 8$

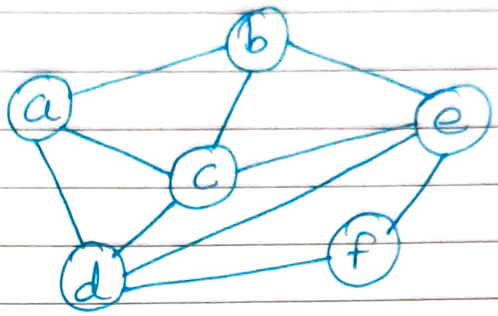


↳ Left of the semicolon are weights of selected objects.
↳ Right of the semicolon is the current total value of load. (profit)

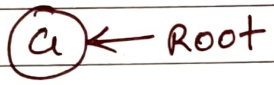
* Hamiltonial Circuit Problem:-

- For given a graph $G=(V, E)$, we start our search from any arbitrary vertex say 'a'.
- This vertex 'a' becomes the root of our implicit tree.
- The first element of our partial sol^n is the first intermediate vertex of the Hamiltonian cycle that is to be constructed.
- The next adjacent vertex is selected by alphabetical order.
- If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached.
- In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed.
- The search using backtracking is successful if a Hamiltonian cycle is obtained.

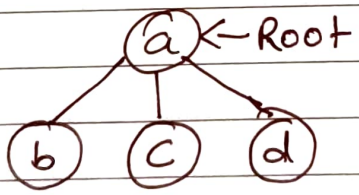
⇒ Example :-



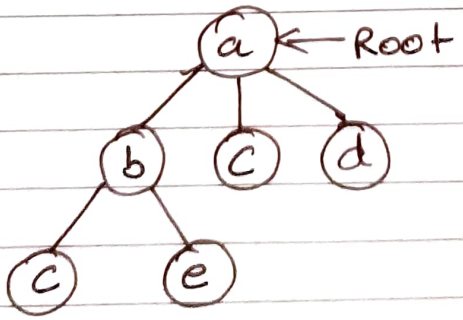
- We start our search with vertex 'a'. This vertex 'a' becomes the root of our implicit tree.



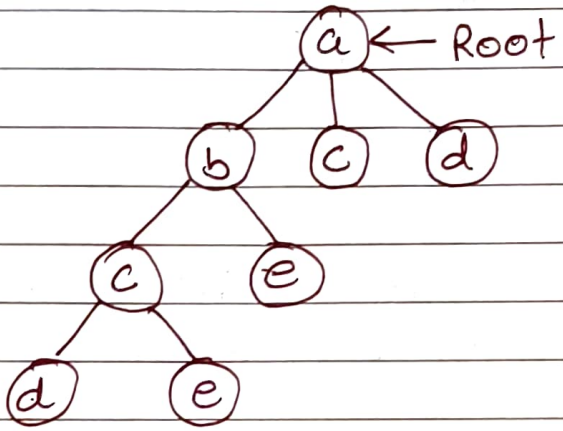
- Now, we choose vertex 'b' adjacent to 'a' as it comes first in alphabetical order (b, c, d).



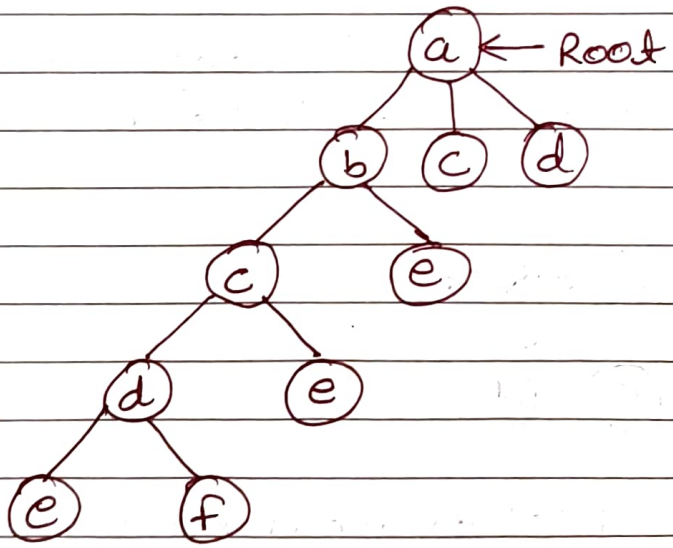
- Next, we select 'c' adjacent to 'b'.



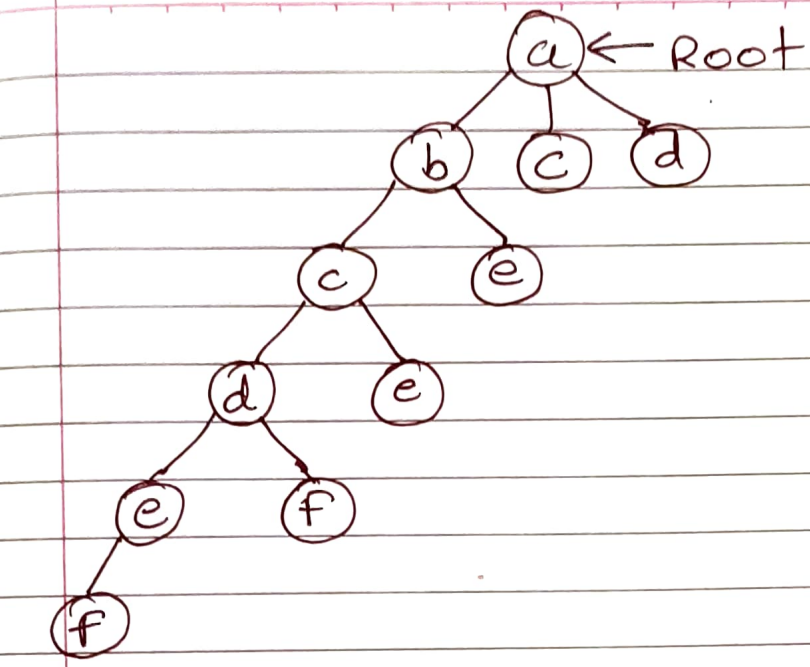
- Next, we select 'd' adjacent to 'c'.



- Next, we select 'e' adjacent to 'd'.

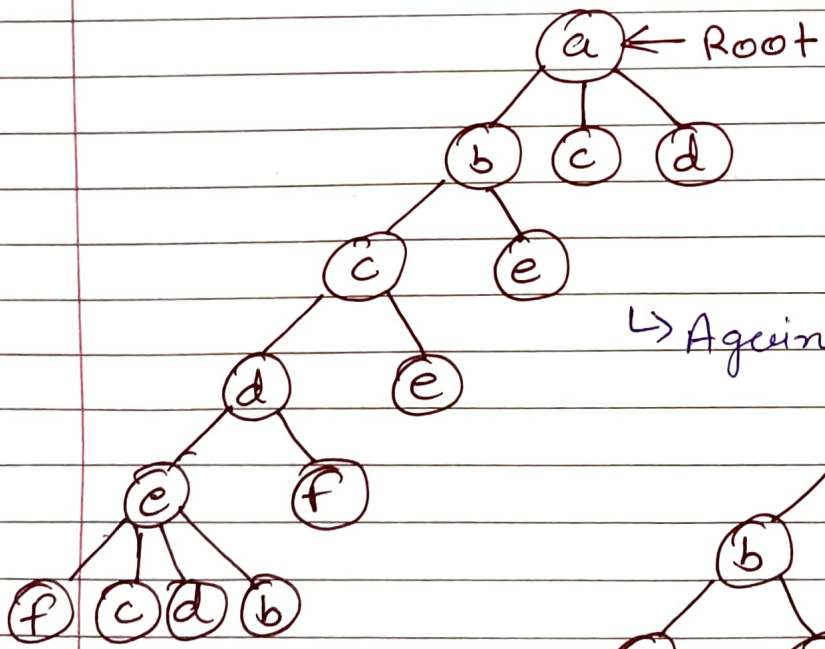


- Next, we select vertex 'f' adjacent to 'e'. The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



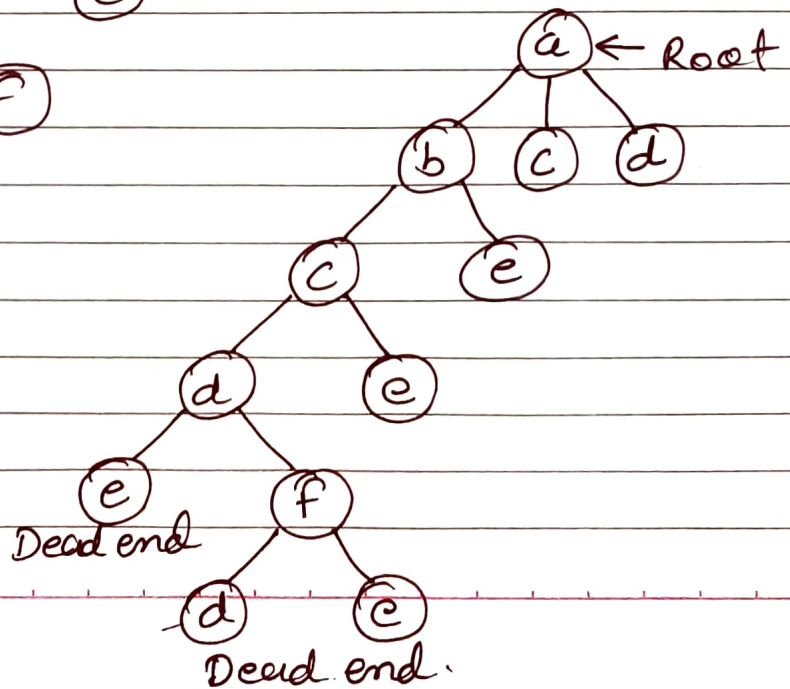
dead end.

- Now, Backtrack.



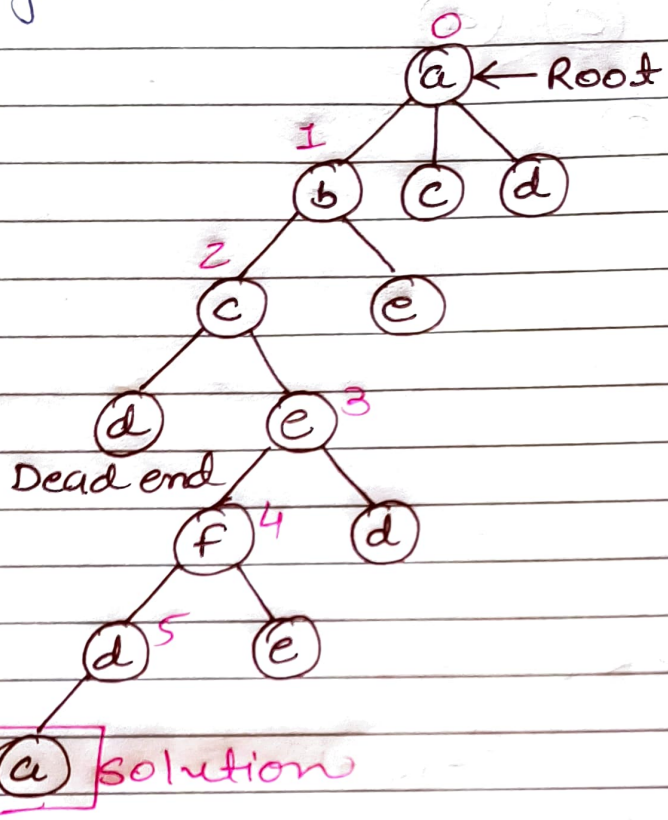
Dead end

↳ Again Backtrack.



Dead end.

- Again Backtrack.



- Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

* Subset - Sum Problem :-

- Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

⇒ Example :- set [] = {3, 34, 4, 12, 5, 2},
sum = 9
→ Answer = True
There is a subset (4, 5) with sum 9.

⇒ Example :- set [] = {3, 34, 4, 12, 5, 2},
Sum = 30
→ Answer = False
There is no subset that add up to 30.

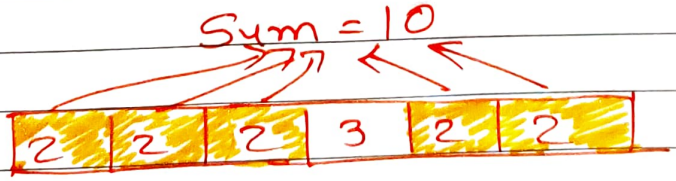
⇒ Understand this approach using Dynamic Programming :-
(Recursion)

→ Two cases :-

(1) Consider the last element and now the required sum = target sum - value of 'last' element and number of elements = total elements - 1

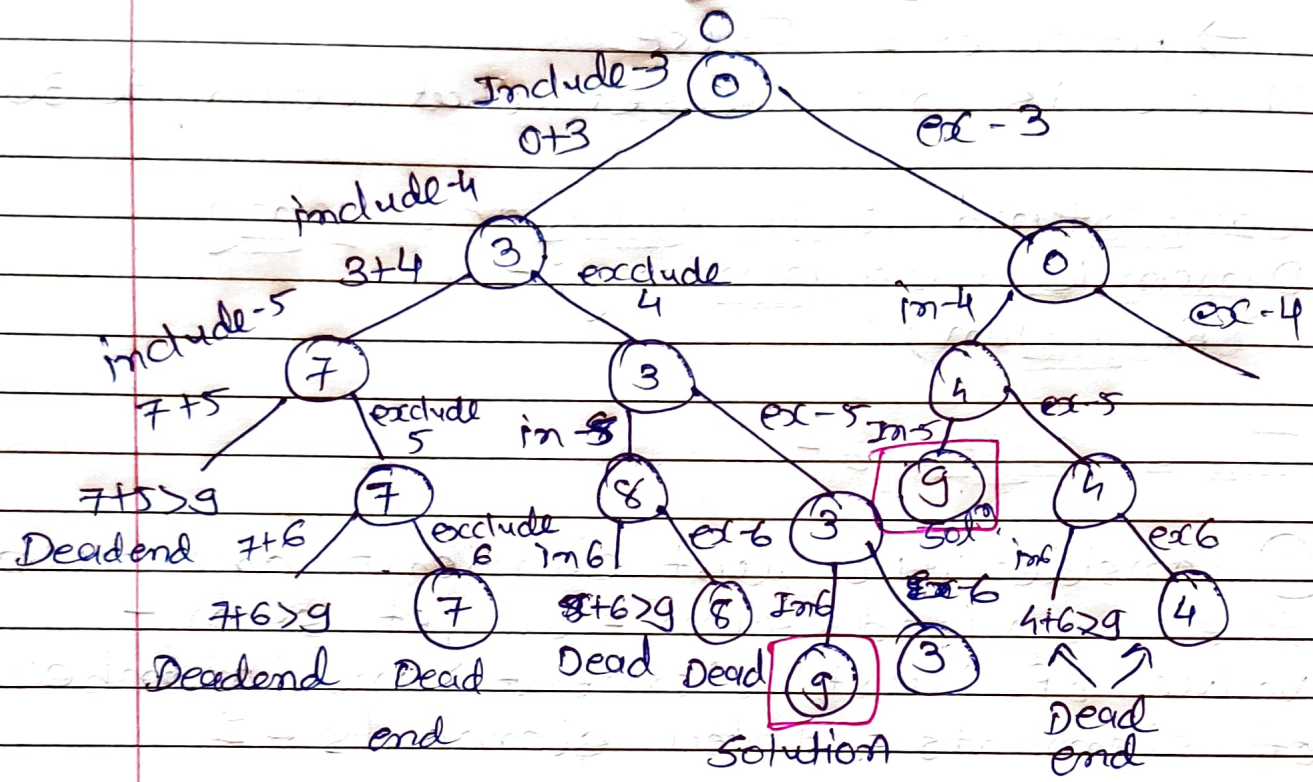
(2) Leave the 'last' element and now the required sum = target sum and number of elements = total elements - 1

⇒ Example :- arr[] = {2, 2, 2, 3, 2, 2}, Sum = 10
→ Answer = True



⇒ Example :- Given a set S = <3, 4, 5, 6> and Sum = 9. obtain the sum using Back tracking approach.

- Initially $S = \langle 3, 4, 5, 6 \rangle$ & sum = 9
  $S' = \langle \phi \rangle$



↳ The left child of the root node indicates that we have to include 'Si' from the set 'S'.

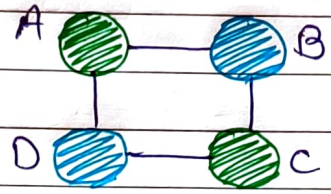
↳ The Right child of the root node indicates that we have to exclude 'Si'.

↳ Each node stores the sum of the partial solution elements.

↳ If at any stage the number equals to 'sum' then the search is successful and terminates.

* Graph Coloring Problem :-

↳ We have been given a graph and we are asked to color all vertices with the 'm' numbers of given colors, in such a way that no two adjacent vertices should have the same color.



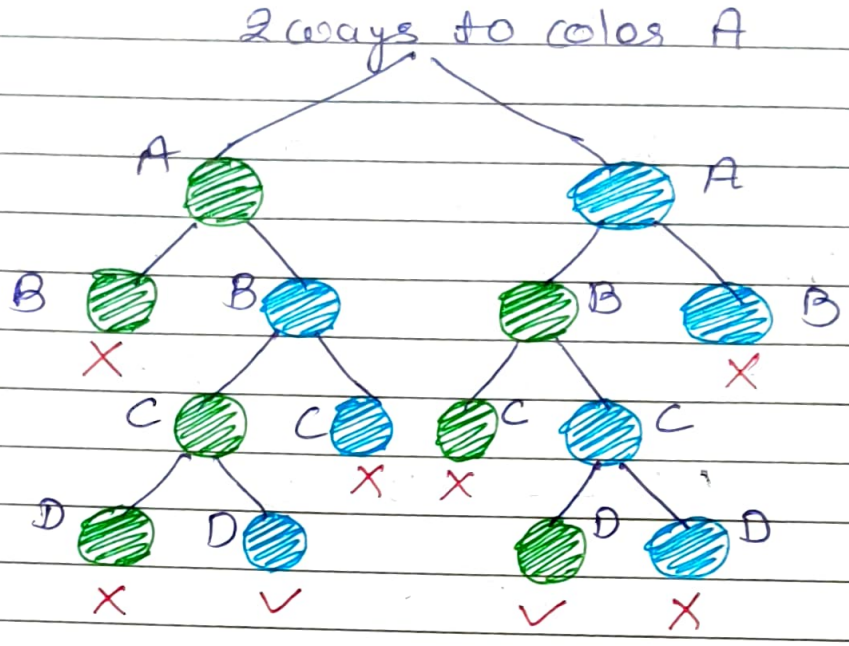
↳ colors Required :- ■■

- It is possible to color all the vertices with the given colors then we have to output the colored result, otherwise output 'no solution possible'.

⇒ Steps :-

1. Different colors :-
   A. Confirm whether it is valid to color the current vertex with the current color.
   B. If yes then color it and otherwise try a different color.
   C. Check if all vertices are colored or not.
   D. If not then move to the next adjacent uncolored vertex.

2. If no other color is available then backtrack.



* Introduction to Branch and Bound :-

- Branch and Bound is used to solve optimization problems.
- The optimized solution is obtained by state space tree.
- The branch and bound approach is based on the principle that the total set of feasible solutions can be partitioned into smaller subsets of solutions.
- These smaller subsets can then be evaluated systematically until the best solution is found.

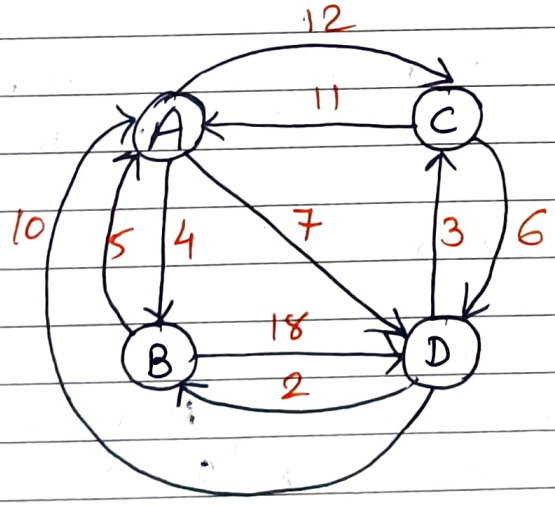
* Travelling Salesman Problem :-

⇒ Problem :- Given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with constraint of minimizing the travelling distance.

⇒ Applications :-

1. Network design
2. Transpartation route design.

⇒ Example :- Solve travelling Salesman Problem using Brench and Bound algorithm in the following graph.



Step 1 :- Initial cost matrix :-

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

Reduce matrix :-

↳ To reduce a matrix, perfor row reduction and column reduction of the matrix separately.

↳ A row or a column is said to be reduced if it contains at least one entry '0' in it.

⇒ Row Reduction :-

↳ If the row already contains an entry '0', then there is no need to reduce that row.

↳ If the row does not contains an entry '0', then,

1. Reduce that particular row.
2. select smallest value element from that row, Subtract that element from each element of that row.

	A	B	C	D	
A	∞	4	12	7	4
B	5	∞	∞	18	5
C	11	∞	∞	6	6
D	10	2	3	∞	2

⇒

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞

⇒ Column Reduction :-

↳ If the column already contains an entry '0', then there is no need to reduce that column.

↳ If the column does not contains an entry '0', then,

(1) Reduce that particular column.

(2) Select the smaller value element from that column, subtract that element from each element of that column.

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞
	0	0	1	0

⇒

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Total Reduction cost = Row Reduction cost + Column Reduction Cost

= 17 + 1

= 18

Step 2 :- Choosing to go to vertex-B (Path A → B)

- Choose column reduction matrix of step-1, M[A, B] = 0
- set row-A and column-B to ∞
- set M[B, A] = ∞

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

⇒

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

⇒ Row Reduction & column Reduction:-

	A	B	C	D	
A	∞	∞	∞	∞	-
B	∞	∞	∞	13	13
C	5	∞	∞	0	0
D	8	∞	0	∞	0
	5	-	0	0	

⇓

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	0	∞	∞	0
D	3	∞	0	∞

Cost(2) = Cost(1) + Sum of reduction elements + m[A,B]
= 18 + (13+5) + 0
= 36

Step 3:- Choosing to go to vertex-C (Path A→C)

- Choose column Reduction matrix of step-01
  m[A,C] = 7
- Set row-A and column-C to ∞
- Set m[C,A] = ∞

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

⇒

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

⇒ Row Reduction & column Reduction :-

	A	B	C	D	
A	∞	∞	∞	∞	-
B	0	∞	∞	13	0
C	∞	∞	∞	0	0
D	8	0	∞	∞	0
	0	0	-	0	

cost(3) = cost(1) + Sum of reduction elements + m[A, C]
= 16 + 0 + 7
= 25

Step 4:- choosing to go to vertex - D (Path A → D)

- Choose column Reduction matrix of step-01
  m[A, D] = 3
- set row - A and column - D to ∞
- set m[D, A] = ∞

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

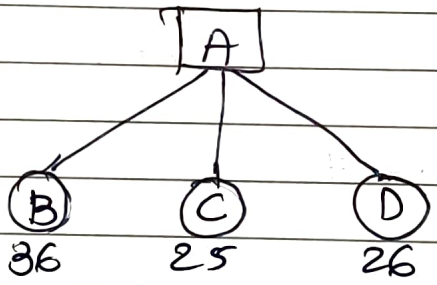
⇒

	A	B	C	D	
A	∞	∞	∞	∞	-
B	0	∞	∞	∞	0
C	5	∞	∞	∞	5
D	∞	0	0	∞	0
	0	0	0	-	

⇒ Row reduction & column reduction :-

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

cost(4) = cost(1) + Sum of reduction elements + m[A,D]
= 18 + 5 + 3
= 26



→ Cost for node-C is lowest, so we prefer to visit node-C (path A→C)

Step 5 :- choosing to go to vertex B (path A→C→B)

- choose column reduction matrix of A→C, m[C,B] = ∞
- Set row-C and column-B to ∞
- Set m[B,A] = ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

⇒

	A	B	C	D	
A	∞	∞	∞	∞	–
B	∞	∞	∞	13	13
C	∞	∞	∞	∞	–
D	8	∞	∞	∞	8
	0	–	–	0	

⇒ Row reduction & column reduction :-

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	0	∞	∞	∞

cost(5) = cost(3) + Sum of reduction elements + m[C, B]
= 25 + (13 + 8) + ∞
= ∞

Step :- Choosing to go to vertex D (A → C → D)

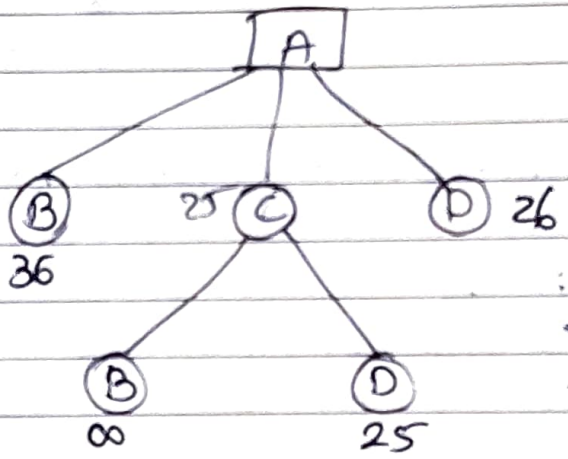
- Choose column reduction matrix of A → C, m[C, D] = 0
- Set row - C and column - D to ∞:
- Set m[D, A] = ∞.

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

⇒

	A	B	C	D	
A	∞	∞	∞	∞	–
B	0	∞	∞	∞	0
C	∞	∞	∞	∞	–
D	∞	0	∞	∞	0
	0	0	–	–	

cost(6) = cost(3) + sum of reduction elements + m[C,D]
= 25 + 0 + 0
= 25



↳ Here, cost of C→D is lowest so, we choose path C→D.

Step 7: Choosing to go to vertex-B (A→C→D→B)

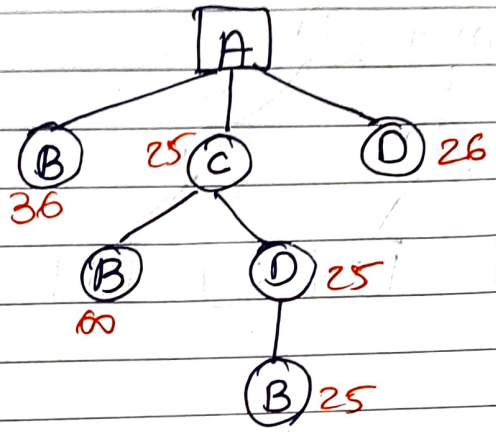
- Choose column reduction matrix of A→C→D
m[D,B] = ∞
- Set row-D and column-B to ∞
- Set m[B,A] = ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

→

	A	B	C	D	
A	∞	∞	∞	∞	–
B	∞	∞	∞	∞	–
C	∞	∞	∞	∞	–
D	∞	∞	∞	∞	–
	–	–	–	–	

cost(7) = cost(6) + Sum of reduction elements + m[D, B]
= 25 + 0 + 0
= 25

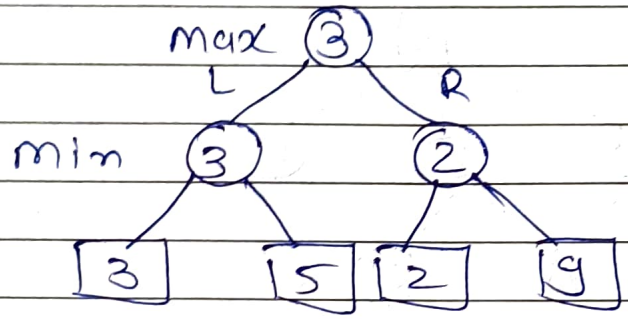


↳ Optimal path is = A → C → D → B → A
↳ Cost of optimal path = 25 units

* Minimax Principle :-

- Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.
- It is widely used in two player turn-based games such as Tic-tac-toe, chess, mancala, etc.
- In minimax the two players are called maximizer and minimizer.

- The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.



* Assignment Problem :-

→ Problem :- Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment.

→ It is required to perform all jobs by assigning exactly one job to each agent in such a way that the total cost of the assignment is minimized.

⇒ Two approaches :-

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).

2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

	Job 1	Job 2	Job 3	Job 4
A	9	(2)	7	8
B	6	4	(3)	7
C	(5)	8	1	8
D	7	6	9	(4)

- Job 2 is assigned to worker A. (lowest value).
- Since Job 2 is assigned to worker A, cost becomes 2 and job 2 and worker A becomes unavailable.
- Now we assign Job 3 to worker B as it has minimum cost from list of unassigned jobs.
- Cost becomes $2+3=5$ and Job 3 and worker B also becomes unavailable.
- Finally, Job 1 gets assigned to worker C. as it has minimum cost among unassigned jobs and Job 4 gets assigned to worker D as it is only left job.
- Total cost = $2+3+5+4=14$