UNIT IV STATE SPACE SEARCH ALGORITHMS

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem – Graph colouring problem **Branch and Bound:** Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

4. Backtracking

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

4.1 Backtracking Algorithm

A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. If the current solution does not satisfy the constraints, then backtrack and try other solutions. Thus, recursion is used in this approach.

> Definition: Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, and removing those solutions that fail to satisfy the constraints.
> When it starts exploring the solutions, a bounding function is applied so that the algorithm can check if the so-far built solution satisfies the constraints. If it does, it continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before.

4.1.1 Pseudo-code of Backtracking Algorithm

```
Backtrack(x)
  if x is not a solution
     return false
  if is a new solution
     add to the list of solution
  backtrack(expand x)
```

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.

It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution. A backtracking algorithm uses the depth-first search method.

4.1.2 Types of Backtracking Algorithm

There are three types of problems in backtracking

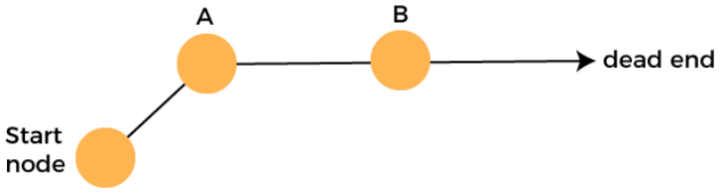
- Decision Problem – Here, we search for a feasible solution.
- Optimization Problem – Here, we search for the best solution.
- Enumeration Problem – Here, we find all feasible solutions.

4.1.3Working of Backtracking Problem

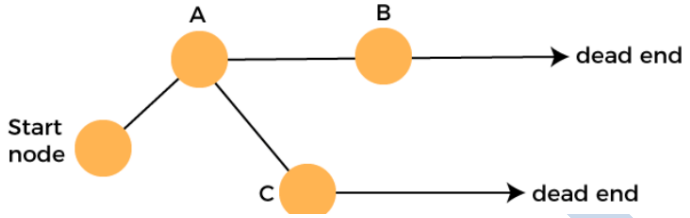
Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

Example 1:

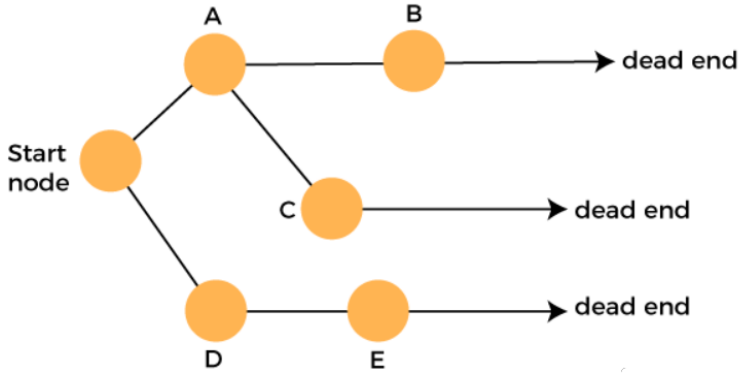
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



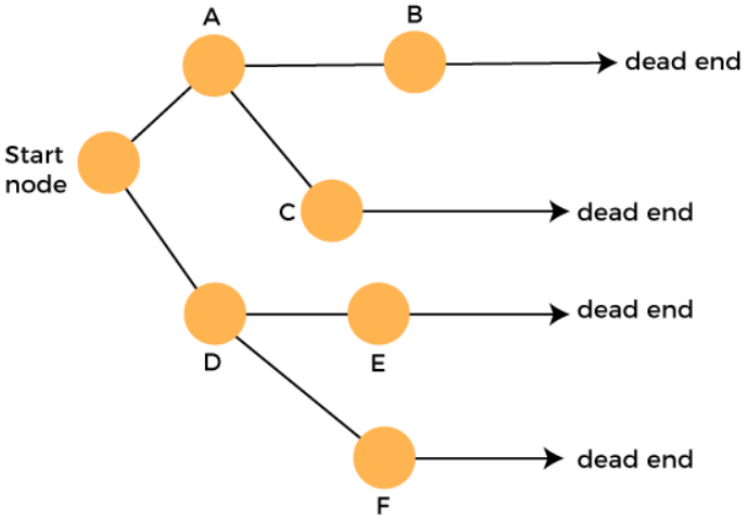
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



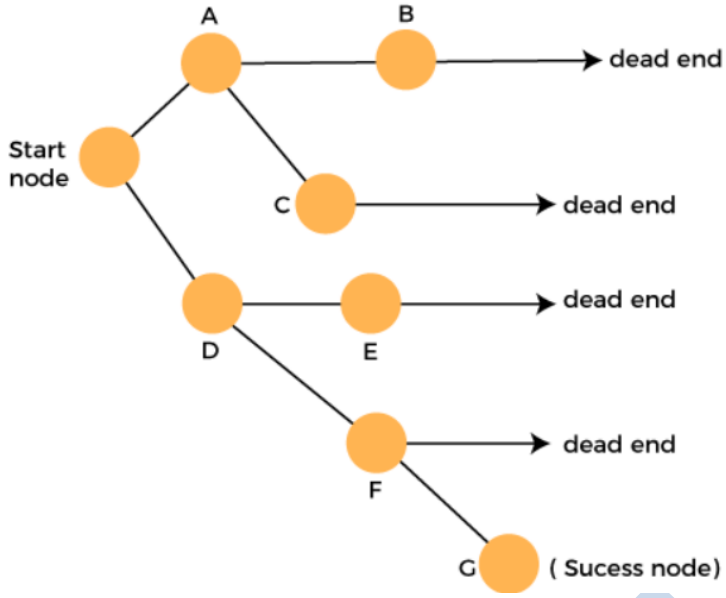
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution, we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.

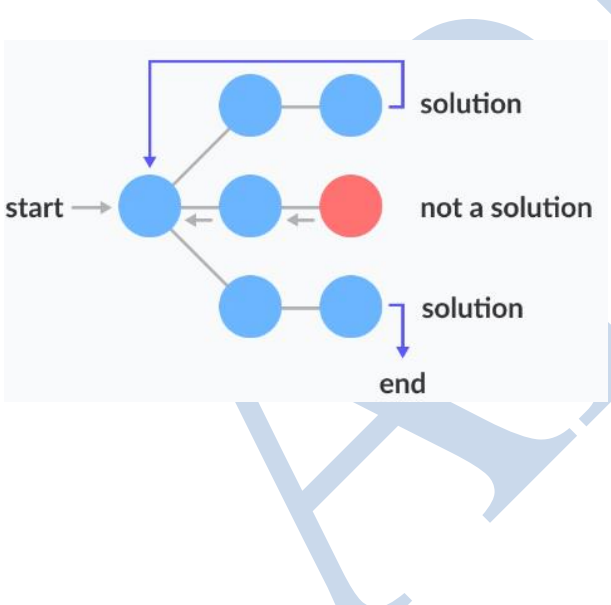


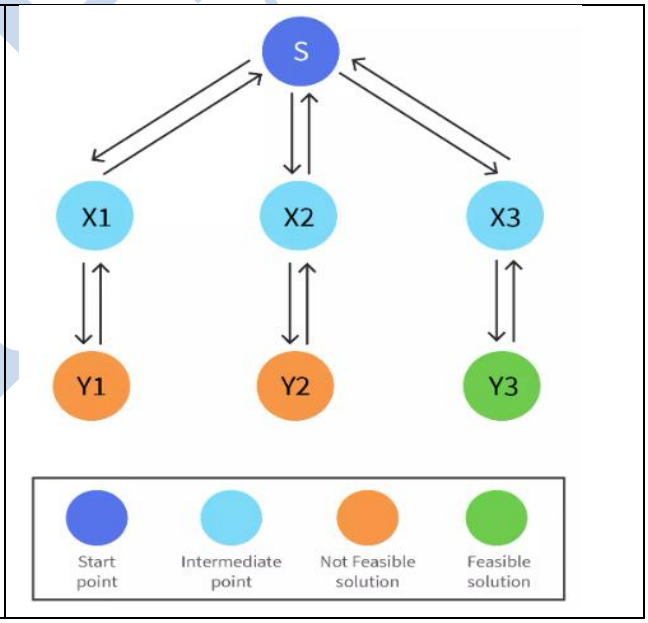
Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



4.1.4 Define State Space Tree

A space state tree is a tree representing all the possible states (solution or non-solution) of the problem from the root as an initial state to the leaf as a terminal state.





Example 2

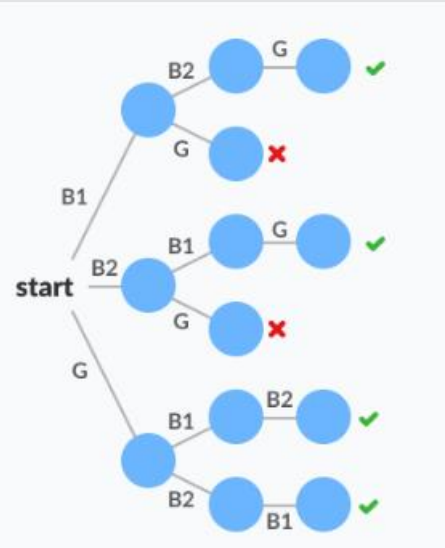
Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.
Constraint: Girl should not be on the middle bench.

Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

All the possibilities are:



The following state space tree shows the possible solutions.



4.1.6 Differences between Decision problem and Optimization problem

Decision problem	Optimization problem
Answer is either zero or one/ true or false. Example: Checking Prime number.	Identifying an optimal value of a given cost function is known as an optimization problem. Example: Single source shortest path.
Decision problem asks us to check if something is true, i.e., 'Yes' or 'No.'	An optimization problem asks us to find, among all feasible solutions ,that maximizes or minimizes a given objective.

4.1.7 Difference between the Backtracking and Recursion

Backtracking	Recursion
Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions which satisfies conditional constraints.	Recursion is a technique that calls the same function again and again until you reach the base case.

Terms in Backtracking

The terms related to the backtracking are:

Live Node: The node on which you are currently and which can be generated further.

E Node: Those nodes which are further generated and could become a successful node.

Dead Node: The node which cannot be generated any further and is essentially a dead end.

Success Node: That node that provides a feasible solution.

Algorithm 1: Backtracking Algorithm for solving the N-Queens Problem

```
Data: Q[n]: an array contains the positions of n queens; k: index
      of the first empty row
Result: All the possible placement of n non-attacking queens on a
        chessboard
Procedure NQueen(Q[n], k)
    if k == n + 1 then
        return Q;
    end
    for j = 1 to n do
        valid = True;
        for i = 1 to k-1 do
            if (Q[i]=j) or (Q[i]=j+k-i) or (Q[i]=j-k+i) then
                valid = False;
            end
            if valid = True then
                Q[k] = j;
                NQueen(Q[n], k+1);
            end
        end
    end
end
```

In case of greedy and dynamic programming techniques, we will use Brute force approach. It means, we will evaluate all possible solutions, among which, we select one solution as optimal solution. In backtracking technique, we will get same optimal solution with less number of steps. So we use backtracking technique. We can solve problems in an efficient way when compared to other methods like greedy method and dynamic programming. In this we will use bounding functions (criterion functions), implicit and explicit conditions. While explaining the general method of backtracking technique, there we will see implicit and explicit constraints.

The major advantage of backtracking method is, if a partial solution (x_1 ,x_2 ,x_3.....,x_i) can't lead to optimal solution then (x_{i+1}...x_n) solution may be ignored entirely.

Explicit constraints:

- In n queens problem, the n queens can be placed in nxn chess board in n^n ways.

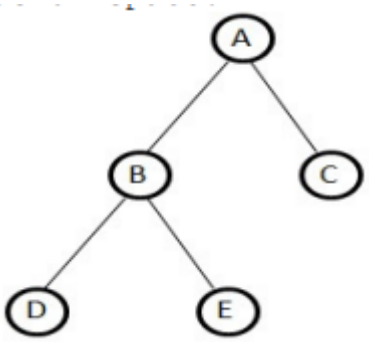
Implicit constraints:

- No 2 queens can be on the same row, same column and same diagonal.

Let us see some terminology which is being used in this method.

1) *Criterion Function*: it is a function p(x1,x2,x3,...xn) which needs to be maximized or minimized for a given problem.

2) *Solution Space :* All tuples that satisfy the explicit constraints define a possible solution space for a particular instance 'i' of the problem. For example consider the following tree. ABD, ABE, AC are the tuples in solution space.



3) *Problem state*: each node in the tree organization defines a problem state. So, A,B ,C are problem states.

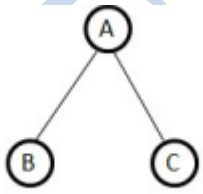


Fig) Tree (Problem State)

4) *Solution states*: These are those problem states S for which the path from the root to S define a tuple in the solution space.

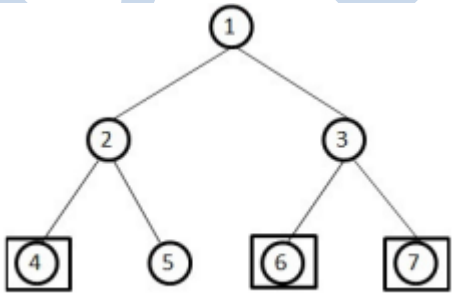


Fig) Tree (Solution state)

Here square nodes indicate solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e. (1,2,4), (1,3,6) and (1,3,7) are the solution states.

5) *State space tree:* if we represent solution space in the form of a tree then the tree is referred as the state space tree. For example given is the state space tree of 4-queen problem. Initially x1=1 or 2 or 3 or 4. It means we can place first queen in either of 1/2/3/4 column. If x1=1 then x2 can be paced in either 2nd, 3rd , or 4th column. If x2=2 then x3 can be placed either in 3rd or 4th column. If x3=3 then x4=4. So nodes 1-2-3-4-5 is one solution in solution space. It may or may not be feasible solution. Similarly we can observe the remaining solutions in the figure.

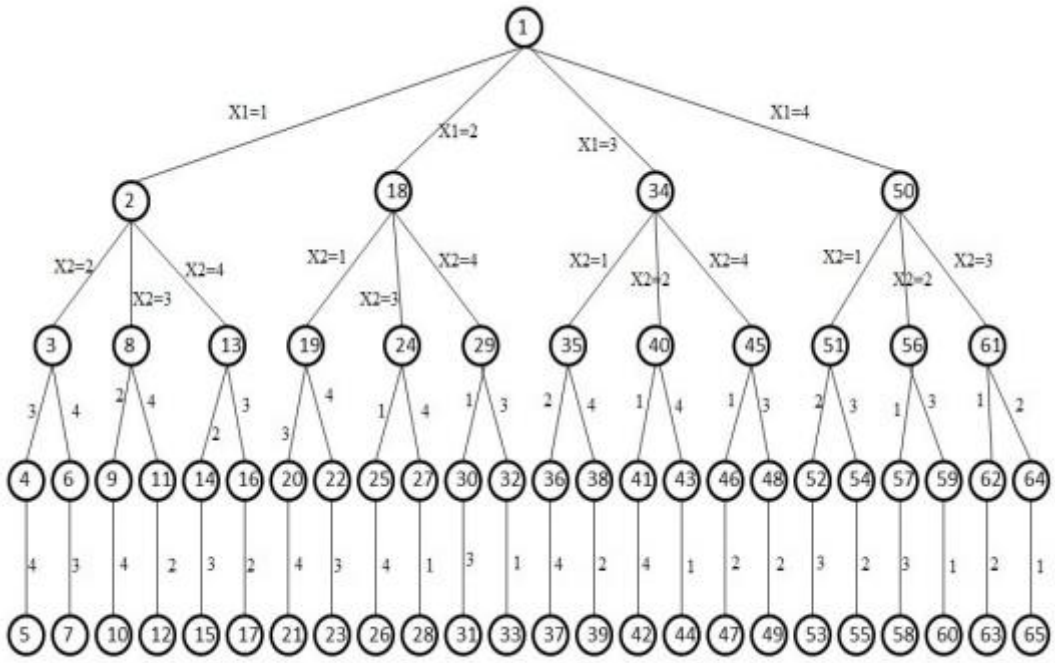


Fig) Tree organization of the 4 queen solution space

6)*Answer states :* These solution states s for which the path from the root to S defines a tuple which is a member of the set of solutions.(i.e. it satisfies the implicit constraints) of the problem. Here 3,4, are answer states. (1,3) and (1,2,4) are solution states.

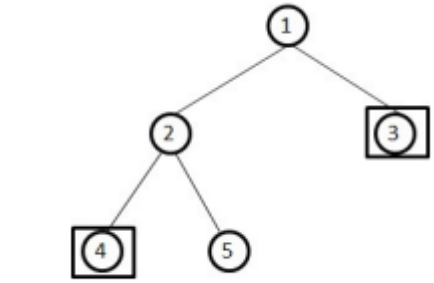


Fig) Tree (answer states)

7) *Live node*: A node which has been generated and all of whose children have not yet been generated is live node. In the fig (a) node A is called live node since the children of node A have not yet been generated.

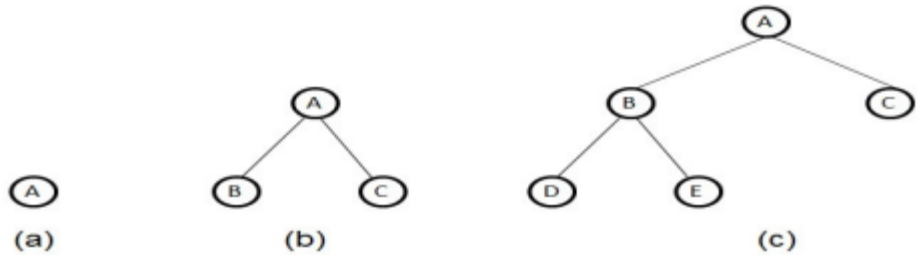


Figure) Live nodes

In fig (b) node A is not a live node but B,C are live nodes. In fig(c) nodes A,B are not live and D,E C are live nodes.

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

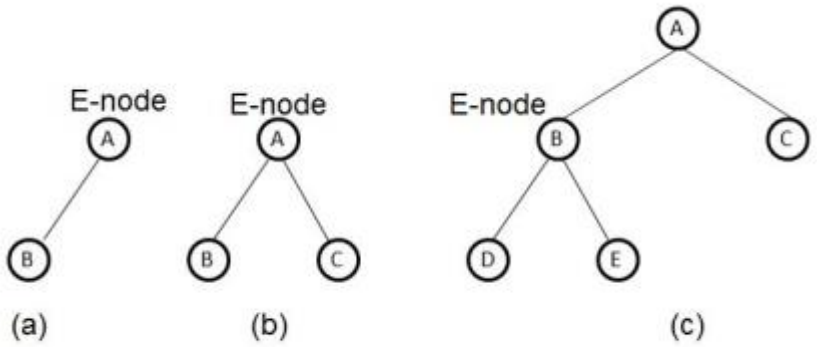


Figure) E-nodes

9) *Dead node*: it is a generated node that is either not to be expanded further or one for which all of its children have been generated. In figure(a) nodes A,B,C are dead nodes since node A's children already generated and Nodes B,C are not expanded.

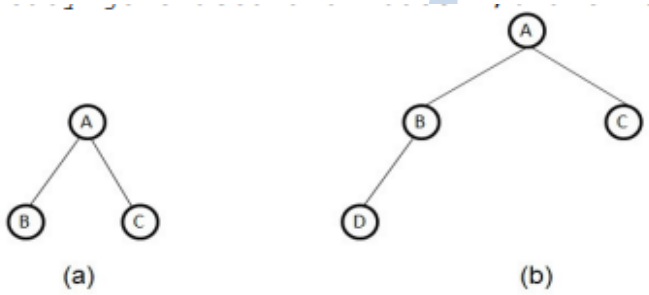


Figure) Dead nodes

In figure (b) assumed that node B can generate one more node so nodes A,D,C are dead nodes.

N- Queens problem

Consider an nxn chess board. Let there are n Queens. These n Queens are to be placed on the nxn chess board so that no two queens are on the same column, same row or same diagonal. n-queens Problem: The n-queens problem is a generalization of the 8-queens problem. Now n-queens are to be placed on an nxn cross board so that no two attack; that is no two queens are on the same row, column, or diagonal. The solution space consists of all $n!$ permutations of $n-tuple$ $(1,2,3,..n)$.

The following figure shows a possible tree organization for the case n = 4. A tree such as this is called a permutation tree. The edges are labelled by possible values of xi.

Finding solutions through State Space Tree:

State space tree can be drawn using backtracking and by this, we will be able to find the **a**ll possible solution for the 4 queen problem.

- The simple logic of generating the state space tree is to keep exploring all possible solutions using backtracking and stop exploring that solution wherever two queens are attacking each other.

For all the positions(columns) in the current row:

- Check for all the previous rows is there a queen or not?
- Check for all the previous diagonal columns is there a queen or not?

- If any of these conditions are true, then backtrack to the previous row and move the previous queen 1 step forward.
- Otherwise, put the current queen in the position and move to the next row.

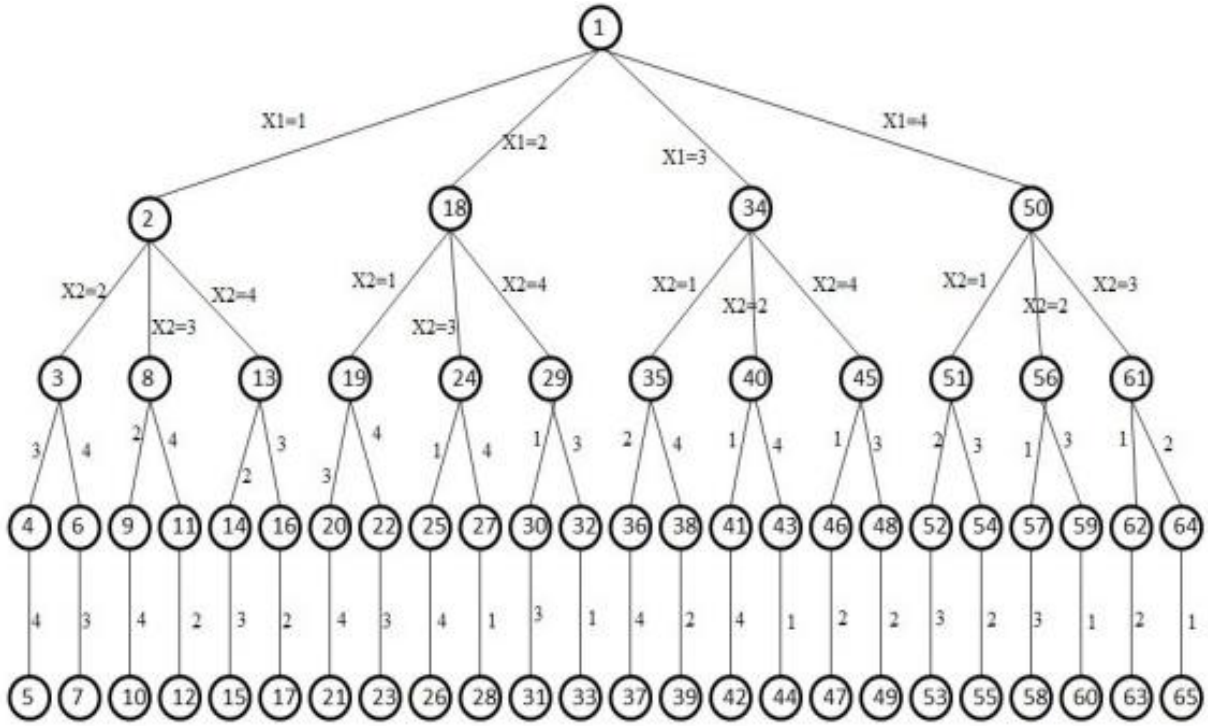


Figure) The organization of 4-queens solution space

nqueens(k,n)	**place(k,i)**	
nqueens(1,4)	place(1,1) returns True	so x[1]=1
nqueens(2,4)	place(2,1) returns False	
	place(2,2) returns False	
	place(2,3) returns True	so X[2]=3
nqueens(3,4)	place(3,1) returns False	
	place(3,1) returns False	
	place(3,1) returns False	
	place(3,1) returns False	
		Backtracking
nqueens(2,4)	place(2,4) returns True	so X[2]=4
nqueens(3,4)	place(3,1) returns False	
	place(3,2) returns True	so X[3]=2
nqueens(4,4)	place(4,1) returns False	
	place(4,2) returns False	
	place(4,3) returns False	
	place(4,4) returns False	
		Backtracking
nqueens(1,4)	place(1,2) returns True	so x[1]=2

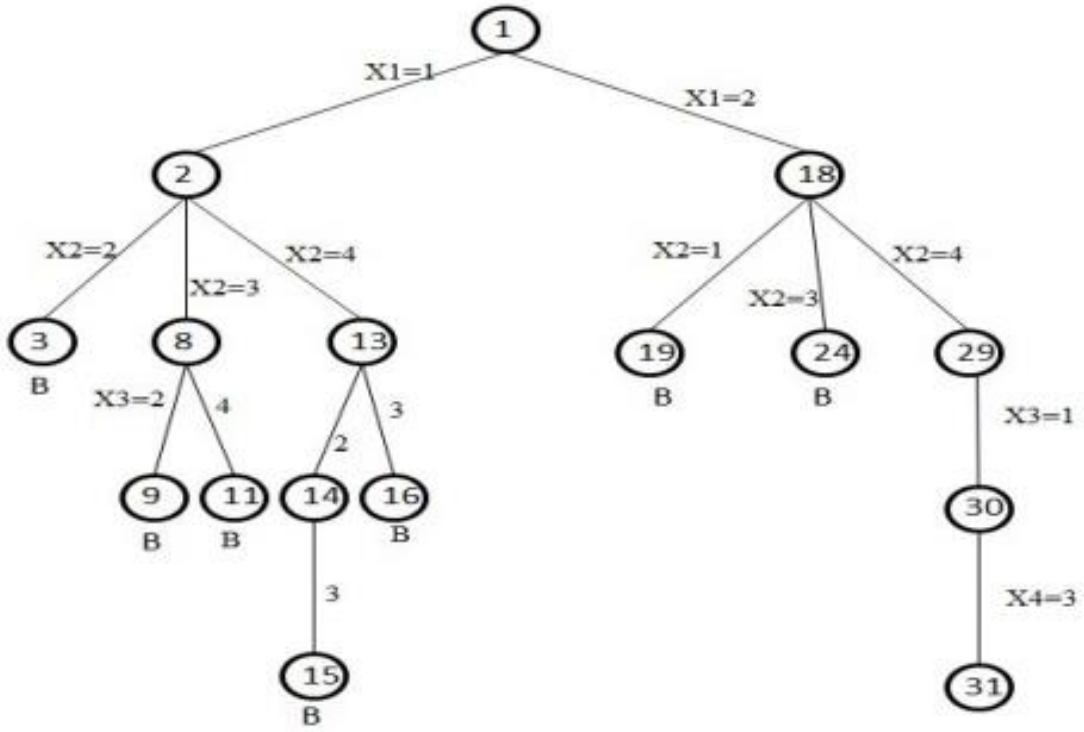
nqueens(2,4) place(2,1) returns False
place(2,2) returns False
place(2,3) returns False
place(2,4) returns True so X[2]=4

nqueens(3,4) place(3,1) returns True so X[3]=1

nqueens(4,4) place(4,1) returns False
place(2,2) returns False
place(2,3) returns True so X[4]=3

The solution vector for a 4x4 cross board to place 4 non attacking queens is

x[1]=2
x[2]=4
x[3]=1
x[4]=3



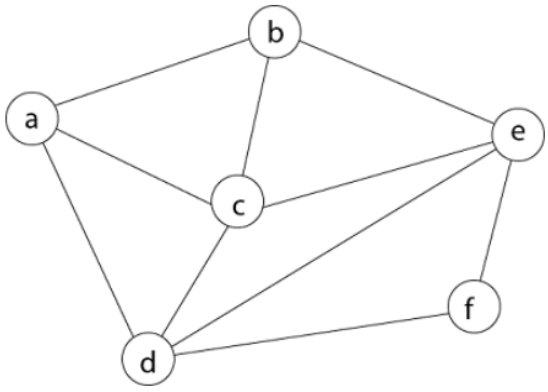
Portion of the tree that is generated during backtracking

Hence we got our solution as **(2, 4, 1, 3)**, this is the one possible solution for the 4-Queen Problem. For another solution, we will have to backtrack to all possible partial solutions he other possible solution for the 4 Queen problem is **(3, 4, 1, 2)**

Hamiltonian Circuit Problems

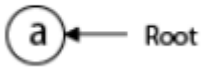
Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle 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: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.

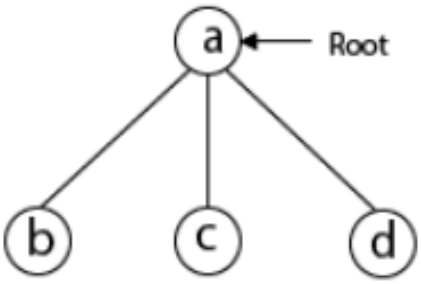


Solution:

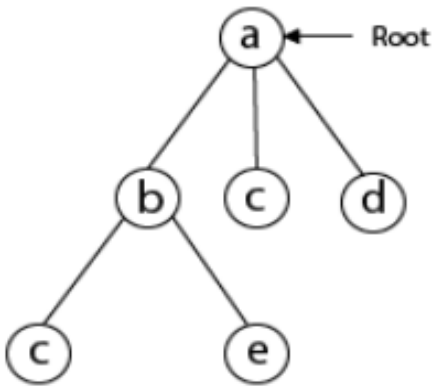
Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



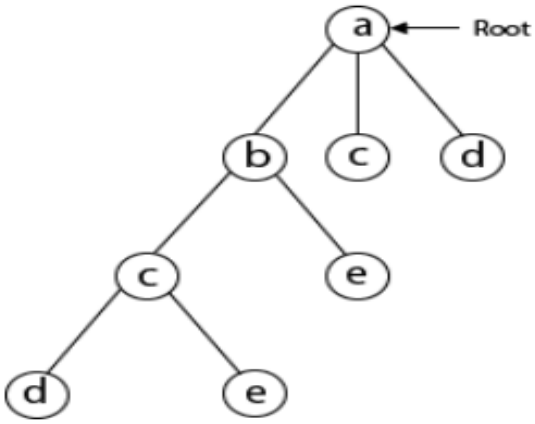
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



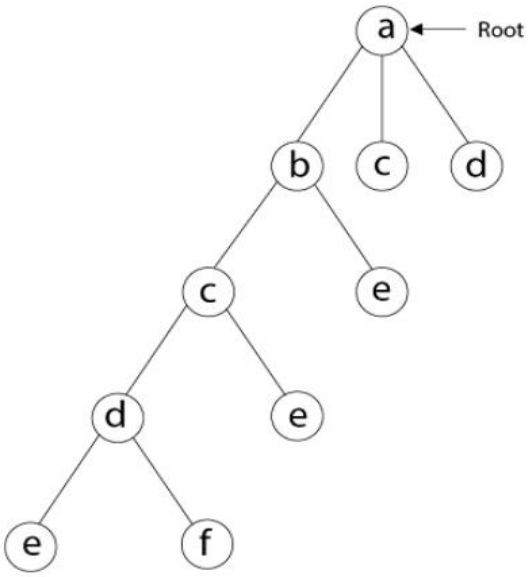
Next, we select 'c' adjacent to 'b.'



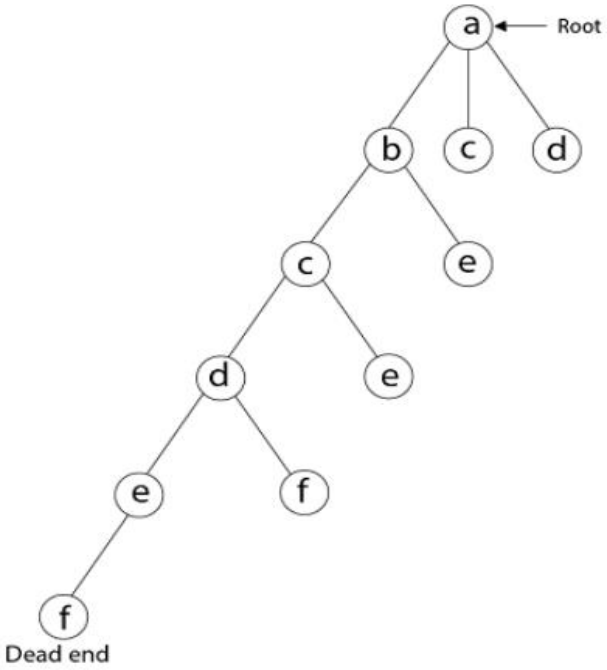
Next, we select 'd' adjacent to 'c.'



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

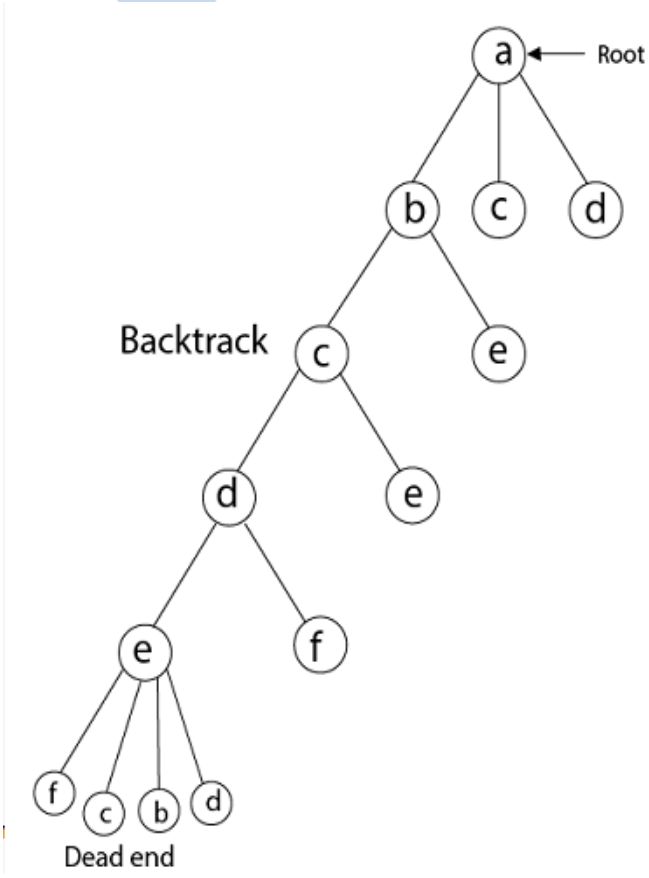


Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.

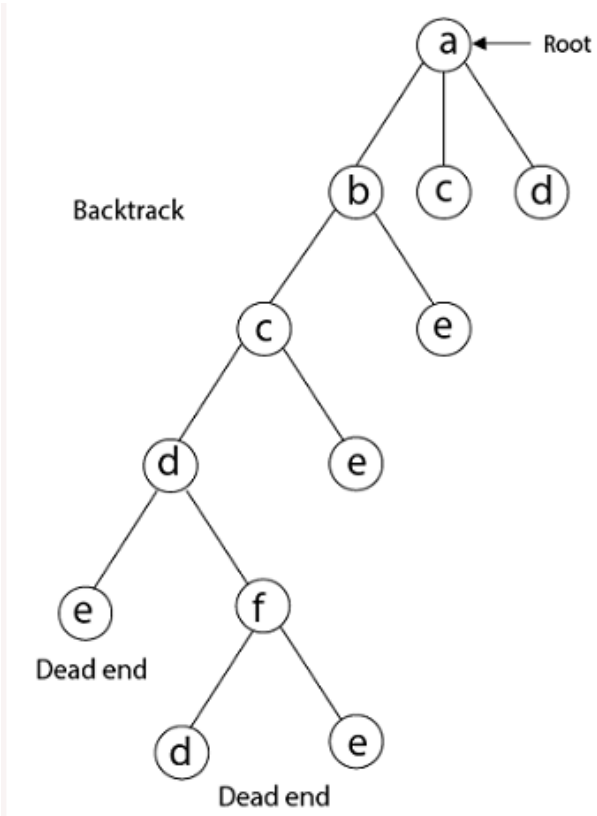


From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

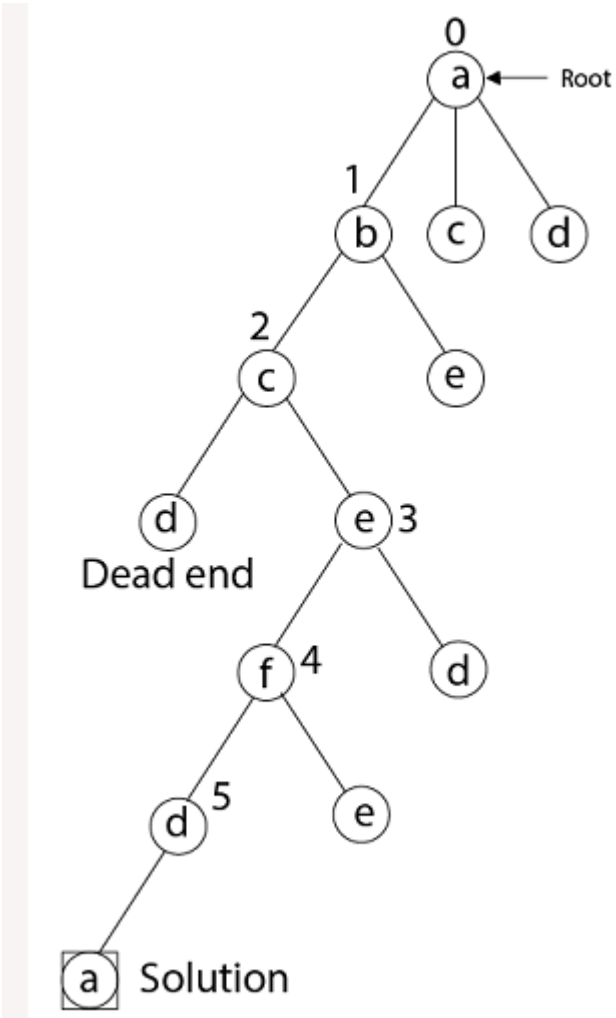
Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).



Again Backtrack



Again Backtrack



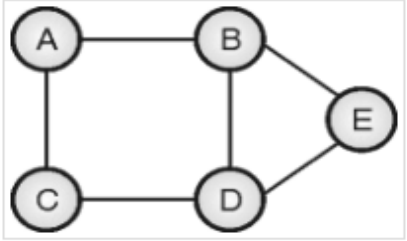
Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Hamiltonian cycle

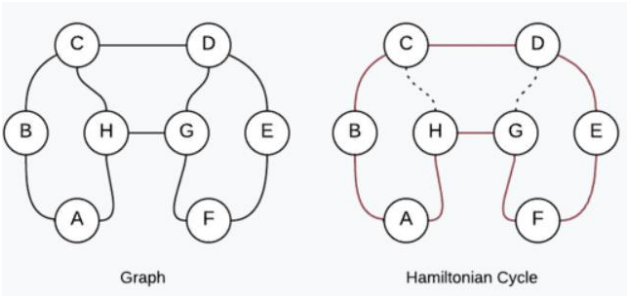
The Hamiltonian cycle is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges

The input to the problem is an undirected, connected graph. For the graph shown, the path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.

Example 1



Example 2



Subset Sum problem

Problem statement

We are given 'n' distinct positive integers and a target_sum. We have to find the combinations of these numbers which exactly sum up to the target_sum value.

Example

Let n=5 and given positive integers are my_list={1,2,3,4,5}. Let the given target_sum=6.The subsets that produce the total of 6 are {1,5},{2,4} which is the output of our program. This is because 1+5=2+4=6.

Approach

We use backtracking approach. It is a recursive algorithm that uses brute force concept. The term backtracking implies that if the current solution is not suitable, then move a step back and try other solutions.

Algorithm.

- Start with an empty set.
- Include the next element from list to set.
- If the numbers in the set sum up to given target_sum, It is a solution set.
- If the set doesnot sum upto the target_sum or if we have reached the end of my_list, then backtrack the set until we find a solution set.
- If we get a feasible solution, go to step 2.
- If we have traversed all the elements and if no backtracking is possible, then stop without solution.

Variables & Declarations :

1. n stores the number of positive integers.
2. my_list stores the given numbers.
3. target_sum is the variable that stores the required sum given by user.
4. We use a solution vector x[] where for every element in my_list,
   x[i]=1, If the element my_list[i] is included in the set and x[i]=0 otherwise.
5. total variable stores the sum of all elements in the given list.
6. s value stores the sum of all elements included in the subset.
7. rem stores the sum that has not been included.

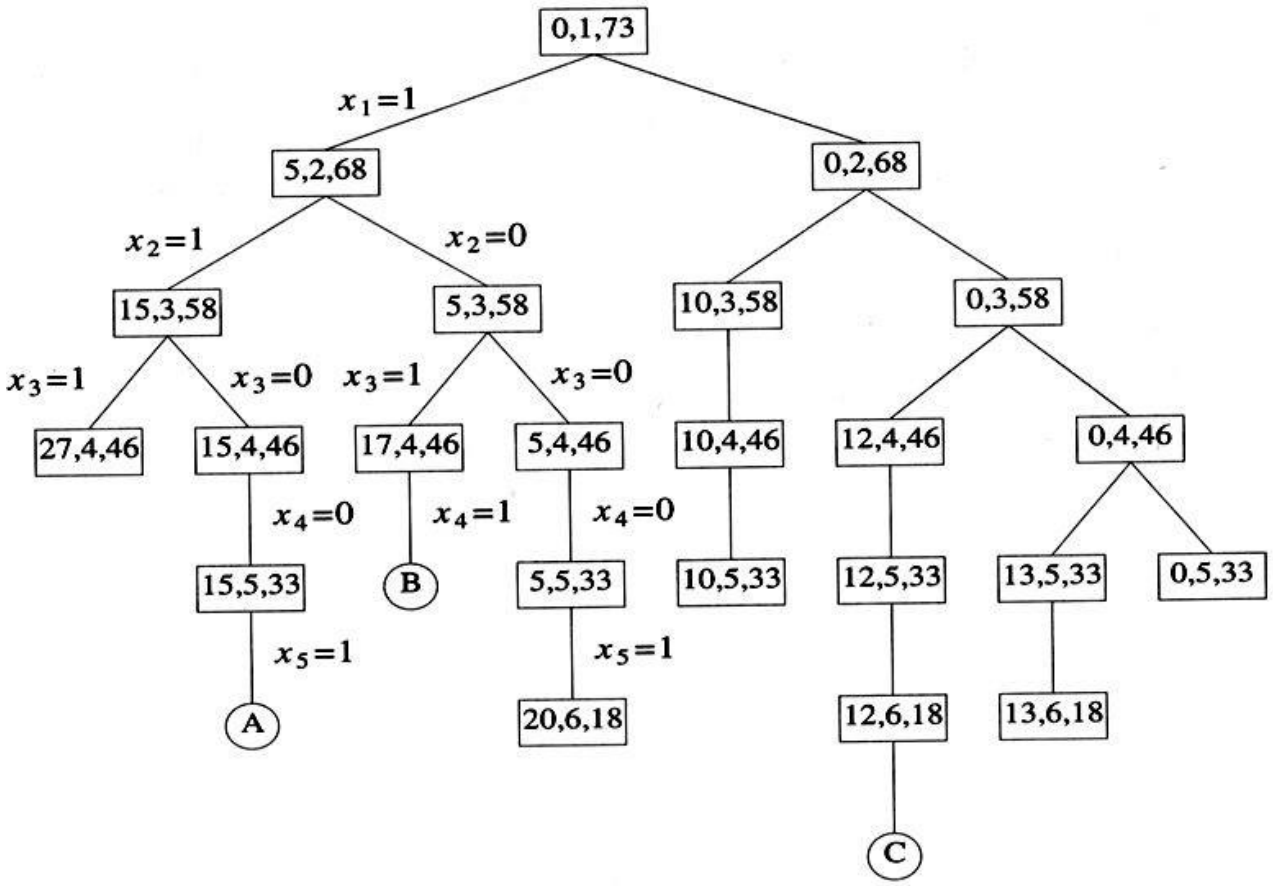
State space tree :

Example:

n=6, my_list={5,10,12,13,15,18}, target_sum=30

The data in rectangular boxes denote values of s,k,rem.

A,B,C are the solution sets which include {5,10,15} , {5,12,13} , {12,18}



Complexity Analysis

It is intuitive to derive the complexity of sum of subset problem. In state space tree, at level i, the tree has 2′ nodes. So given n items, total number of nodes in tree would be 1 + 2 + 2 ^ 2 + 2 ^ 3 +..2^ n

T(n)=1+2+2^ 2 +2^ 3 +. 0.2 ^ n = 2 ^ (n + 1) – 1 = O(2 ^ n)

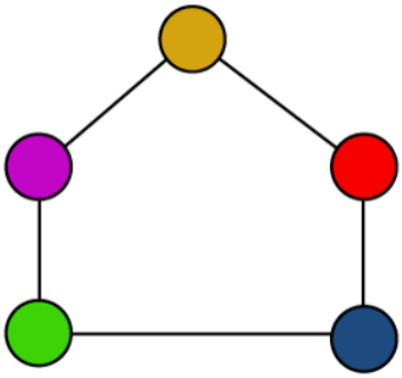
Thus, sum of sub set problem runs in exponential order.

Graph coloring

Introduction

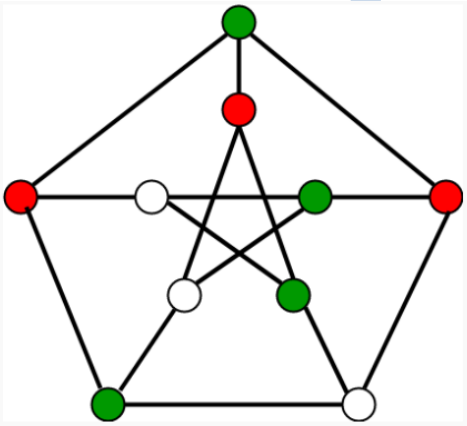
Graph coloring can be described as a process of assigning colors to the vertices of a graph. In this, the same color should not be used to fill the two adjacent vertices. We can also call graph coloring as Vertex Coloring. In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color. This type of graph is known as the Properly colored graph.

In this graph, we are showing the properly colored graph, which is described as follows:



m Coloring Problem

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color Following is an example of a graph that can be colored with 3 different colors:

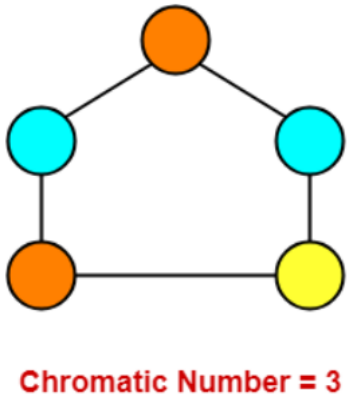


The least possible value of 'm' required to color the graph successfully is known as the chromatic number of the given graph.

Chromatic Number

Definition

Chromatic Number is the minimum number of colors required to properly color any graph (or)Chromatic Number is the minimum number of colors required to color any graph, such that no two adjacent vertices of it are assigned the same color.



- No two adjacent vertices are colored with the same color.
- Minimum number of colors required to properly color the vertices = 3.
- Therefore, Chromatic number of this graph = 3.
- We cannot properly color this graph with less than 3 colors

Chromatic Number Of Graphs;

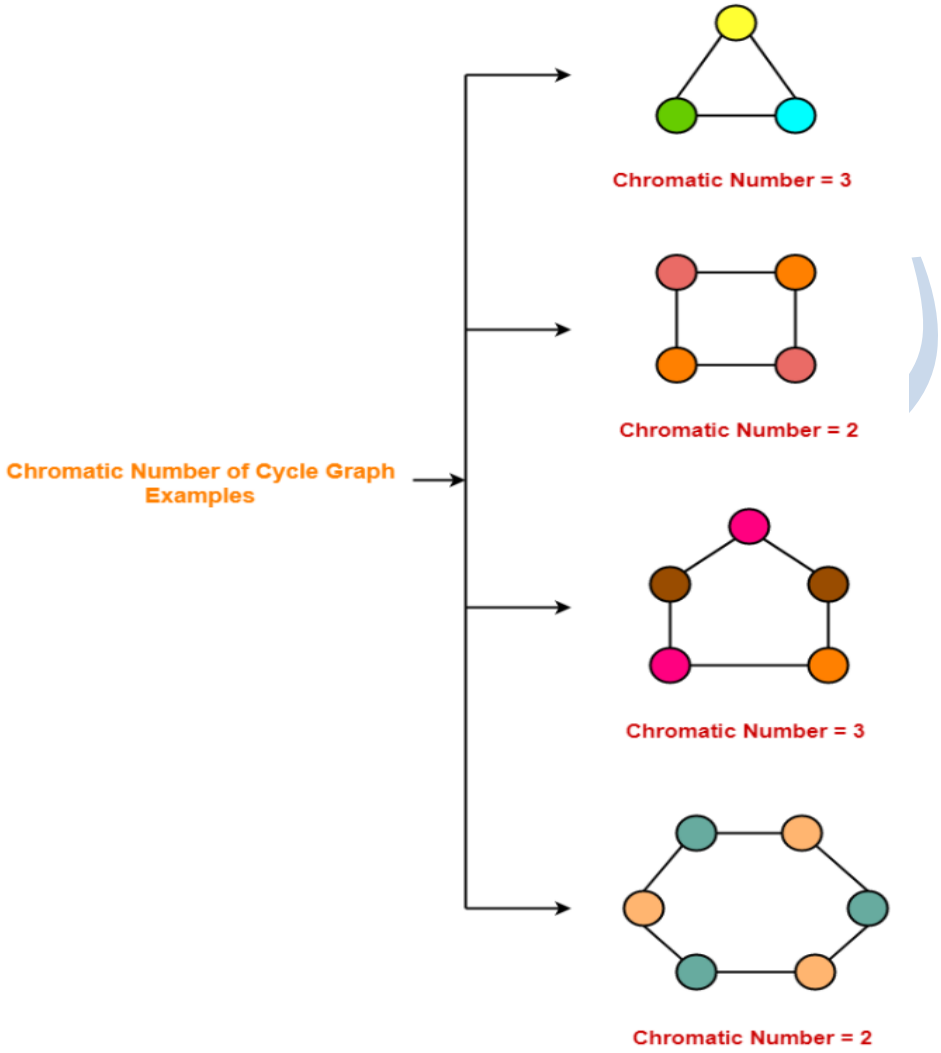
Chromatic Number of some common types of graphs are as follows:-

1. Cycle Graph

A simple graph of 'n' vertices (n>=3) and 'n' edges forming a cycle of length 'n' is called as a cycle graph.

In a cycle graph, all the vertices are of degree 2.

- If number of vertices in cycle graph is even, then its chromatic number = 2.
- If number of vertices in cycle graph is odd, then its chromatic number = 3.



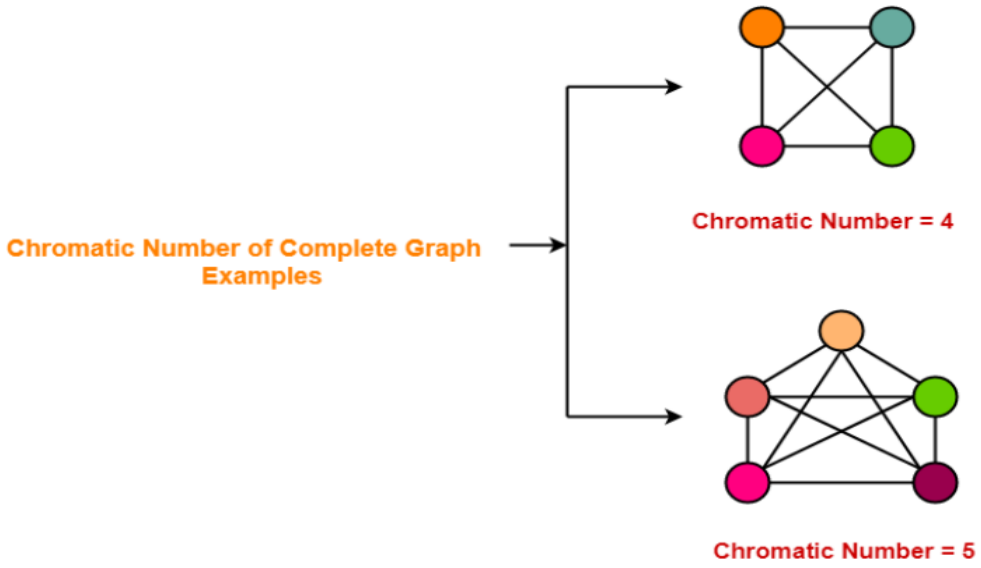
Note:

- All the above cycle graphs are also planar graphs.

2. Complete Graph

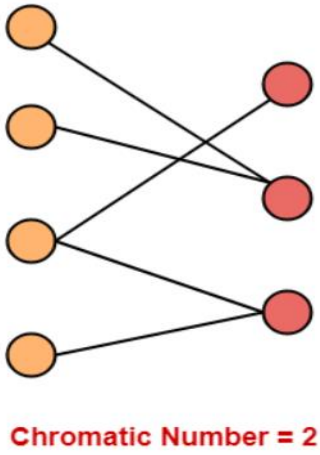
In a complete graph, each vertex is connected with every other vertex.

Chromatic Number of any Complete Graph= Number of vertices in that Complete Graph



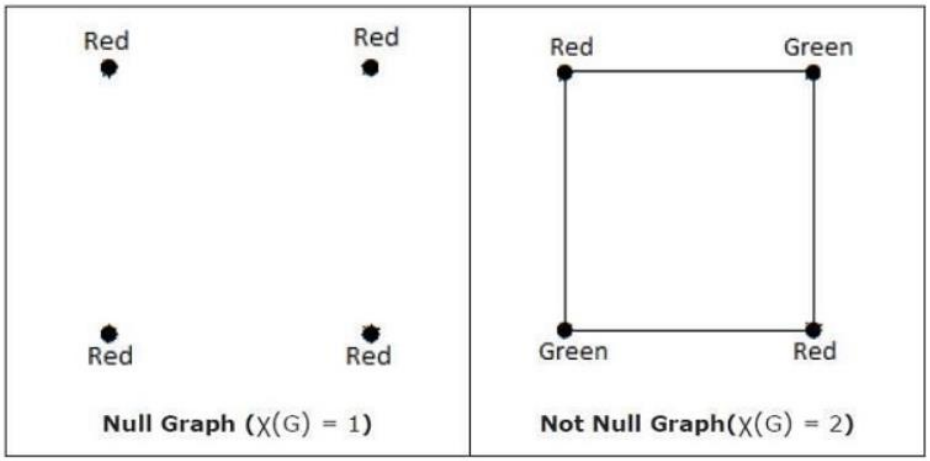
3. Bipartite Graphs

A Bipartite Graph consists of two sets of vertices X and Y.
The edges only join vertices in X to vertices in Y, not vertices within a set.



The minimum number of colors required for vertex coloring of graph 'G' is called as the chromatic number of G, denoted by X(G).

$\chi(G) = 1$ if and only if 'G' is a null graph. If 'G' is not a null graph, then $\chi(G) \geq 2$.

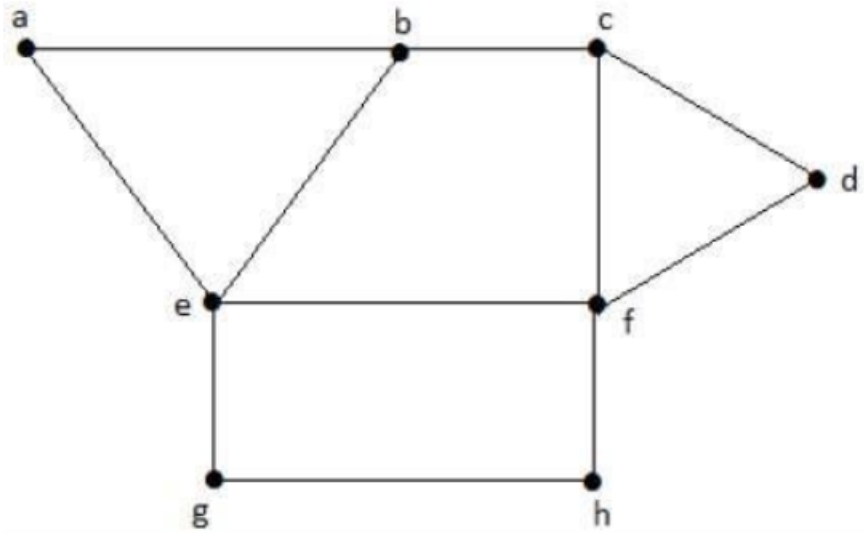


Region Coloring

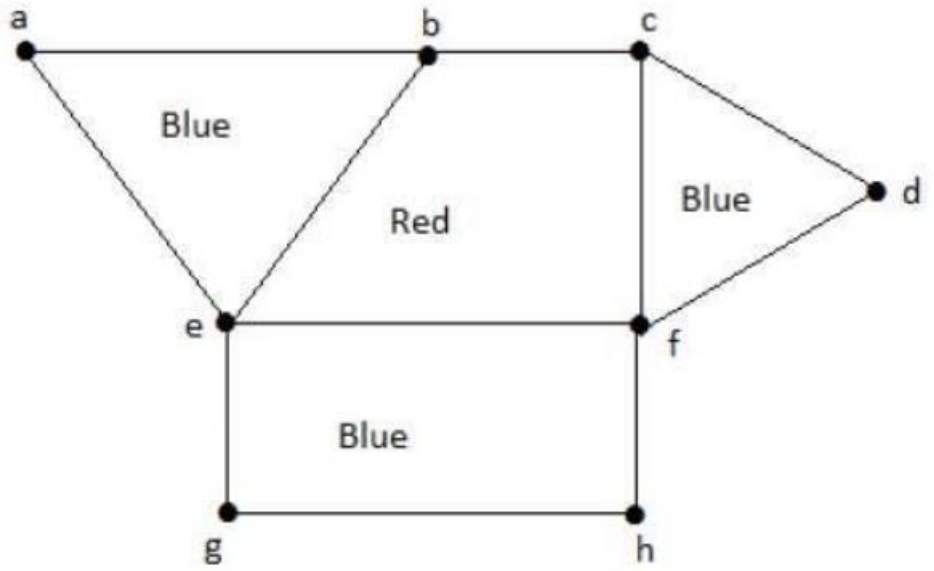
Region coloring is an assignment of colors to the regions of a planar graph such that no two adjacent regions have the same color. Two regions are said to be adjacent if they have a common edge.

Example

Take a look at the following graph. The regions 'aeb' and 'befc' are adjacent, as there is a common edge 'be' between those two regions.



Similarly, the other regions are also coloured based on the adjacency. This graph is coloured as follows –

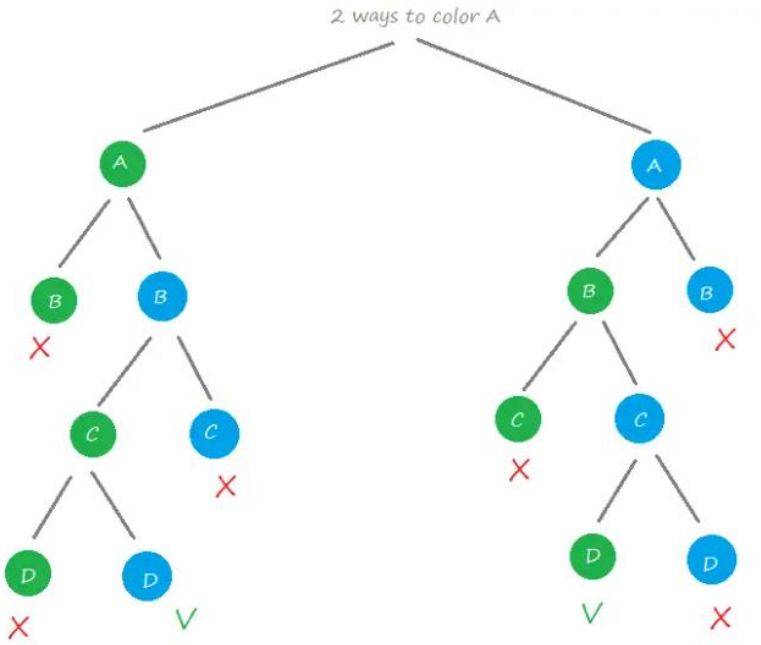


Steps To color graph using the Backtracking Algorithm:

1. Different colors:
    - Confirm whether it is valid to color the current vertex with the current color (by checking whether any of its adjacent vertices are colored with the same color).
    - If yes then color it and otherwise try a different color.
    - Check if all vertices are colored or not.
    - If not then move to the next adjacent uncolored vertex.
2. If no other color is available then backtrack (i.e. un-color last colored vertex).

Here backtracking means to stop further recursive calls on adjacent vertices by returning false. In this algorithm Step-1.2 (Continue) and Step-2 (backtracking) is causing the program to try different color option.

- Continue – try a different color for current vertex.
- Backtrack – try a different color for last colored vertex.



Graph Coloring Applications-

Applications of graph coloring

- Map Coloring
- Scheduling the tasks
- Preparing Time Table
- Assignment
- Conflict Resolution
- Sudoku

Branch and bound - Definition

- Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.
- A branch and bound algorithm provide an optimal solution to an NP-Hard problem by exploring the entire search space. Through the exploration of the entire search space, a branch and bound algorithm identify possible candidates for solutions step-by-step.

Different search techniques in branch and bound:
The Branch algorithms incorporate different search techniques to traverse a state space tree. They are:

- Least Cost Search - node with the least cost selected as a next E-node
- Breadth First Search - known as FIFO search & live nodes are searched in FIFO order
- Depth First Search - known as LIFO search & live nodes are searched in LIFO order

APEC

15 – Puzzle Problem

This problem was invented by Sam Loyd in 1878. There are 15 numbered tiles placed on a square frame with capacity of 16 tiles. Given an initial arrangement and the objective is to transform any initial arrangement to a goal arrangement through a series of legal moves.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal arrangement

- A legal move is the one that move a tile adjacent to an Empty Space(ES) to ES.
- Each move creates a new arrangement of tile called states of the puzzle.
- Initial and final arrangements are called initial state and goal states respectively.
- There are 16! (20.9 * 1012 = 20,922,789,888,000) different arrangements of the tiles on the frame. Only half of them are reachable from any given initial state.
- If we number positions 1-16, position(i) is the frame position containing the tile number i in the goal arrangement

Check whether the goal state is achieved from the initial state given below

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Initial State

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal State

Initial State

less(1) = 0 less(2) = 0 less(3) = 0 less(4) = 0 less(5) = 0 less(6) = 0 less(7) = 0 less(8) = 1 less(9) = 1 less(10) = 1 less(11) = 0 less(12) = 0 less(13) = 1 less(14) = 1 less(15) = 1

r = 2

$\sum$ = 0+0+0+0+0+0+0+1+1+1+0+0+1+1+1+2 = 8

Goal State

less(1) = 0 less(2) = 0 less(3) = 0 less(4) = 0 less(5) = 0 less(6) = 0 less(7) = 0 less(8) = 0 less(9) = 0 less(10) = 0 less(11) = 0 less(12) = 0 less(13) = 0 less(14) = 0 less(15) = 0

r = 4

$\sum$ = 15*0 + 4 = 4

Since the initial and the goal state are even parity the goal state is reachable from the initial state.

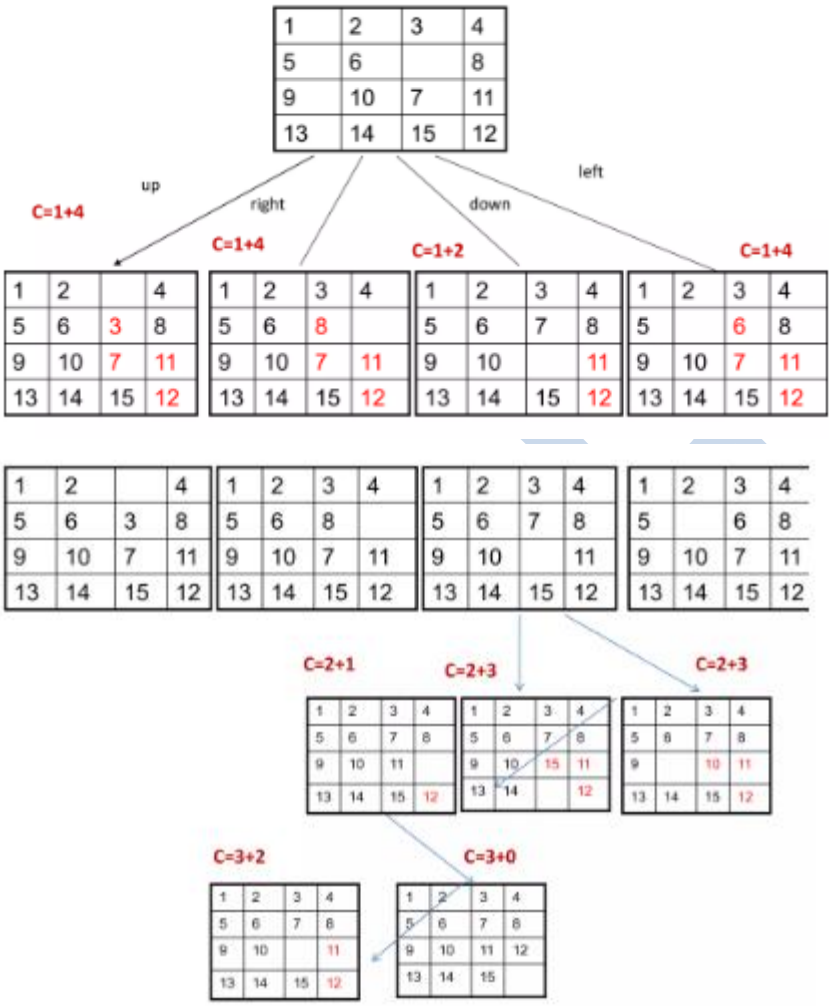
Instead of calculating c(x), calculate ĉ(x) as

ĉ(x) = f(x) + ĝ(x)

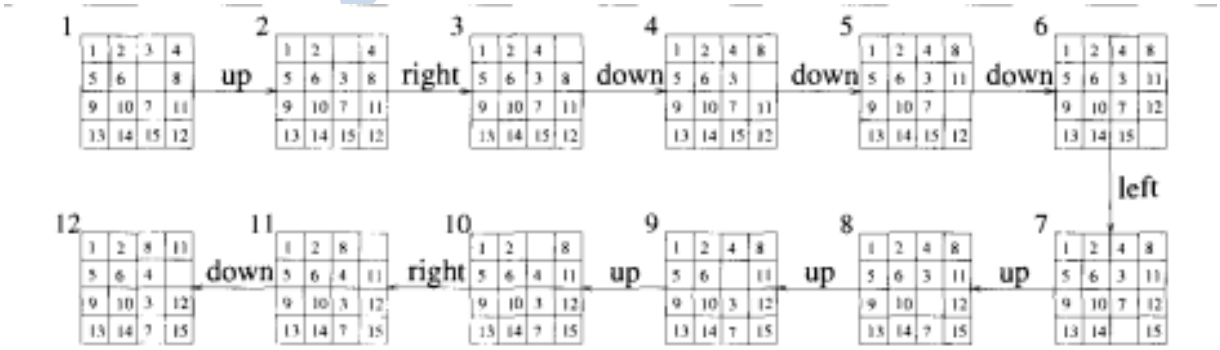
• f(x) : length of the path from root to x.

• ĝ(x) : number of non blank tiles not in their goal position.

In the state space tree, node1 dies after leaving behind nodes 2, 3, 4 and 5.
Calculating ĉ (x) for these nodes, we have
ĉ(2) = 1+4 = 5
ĉ(3) = 1+4 = 5
ĉ(4) = 1+2 = 3
ĉ(5) = 1+4 = 5
Since ĉ (4) is the lowest value, next E-node is node 4

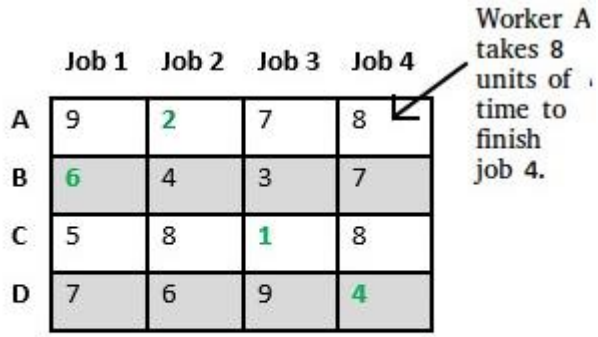


First ten steps in a depth first search



Assignment problem using Branch and Bound

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 worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



An example job assignment problem. Green values show optimal job assignment that is A-Job2, B-Job1 C-Job3 and D-Job4

The selection rule for the next node in BFS and DFS is "blind". i.e. the selection rule 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 optimal solution can often be speeded by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an optimal solution. It is similar to BFS-like search but with one major optimization. Instead of following FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

Algorithm 1: Job Assignment Problem Using Branch And Bound

```
Data: Input cost matrix M[][]
Result: Assignment of jobs to each worker according to optimal
        cost
Function MinCost(M[][])
while True do
    E = LeastCost();
    if E is a leaf node then
        print();
        return;
    end
    for each child S of E do
        Add(S);
        S → parent = E;
    end
end
```

There are two approaches to calculate the cost function:

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

For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).
In this article, the first approach is followed.

Let's take below example and try to calculate promising cost when Job 2 is assigned to worker A.

	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

Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

	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

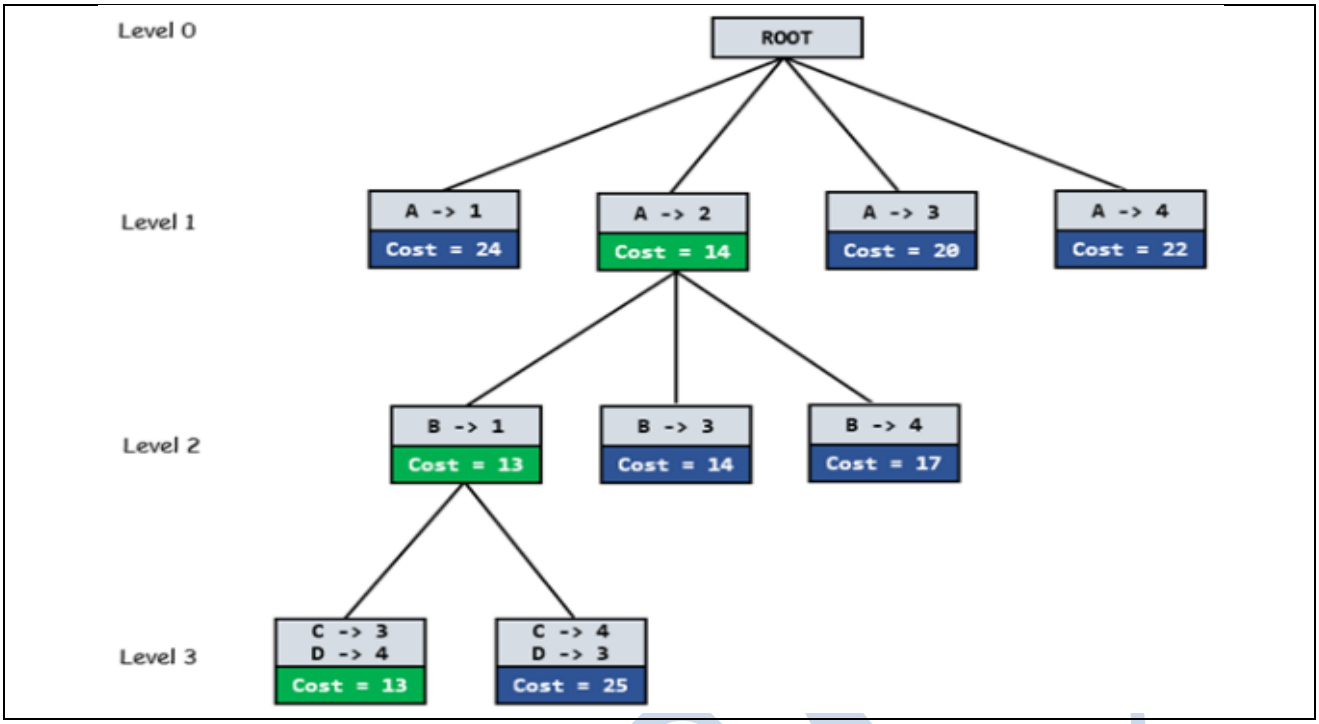
Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.

	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

Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker C as it is only Job left. Total cost becomes 2 + 3 + 5 + 4 = 14.

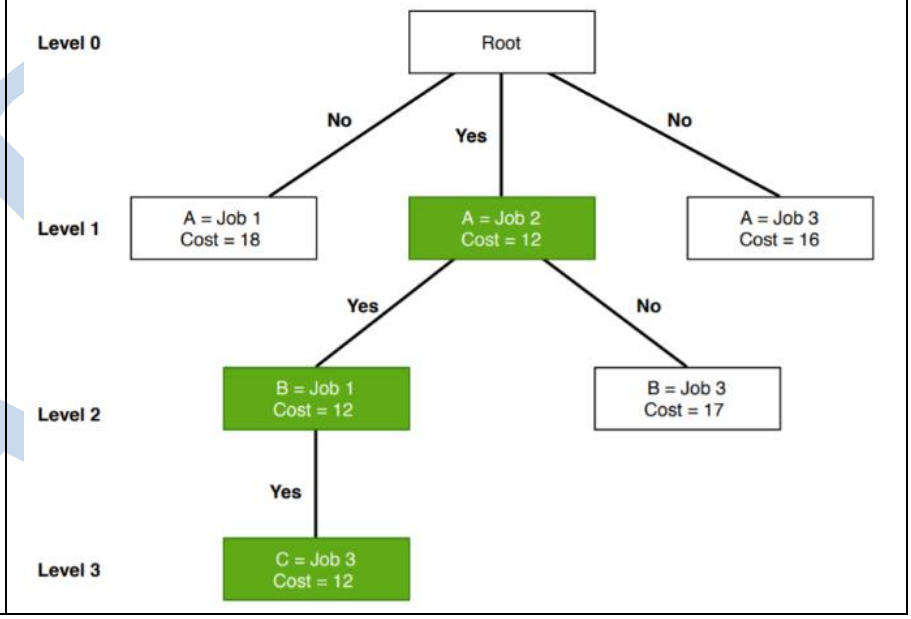
	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Below diagram shows complete search space diagram showing optimal solution path in green.

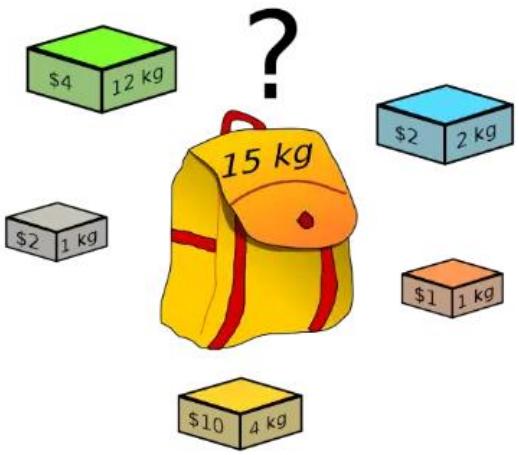


Problem 2

	Job 1	Job 2	Job 3
A	9	3	4
B	7	8	4
C	10	5	2



0-1 Knapsack problem using Branch and Bound



Definition

- Suppose a person has to fill up his knapsack by selecting various objects which will give him maximum profit.

Mathematical representation of knapsack problem

Number the objects from 1 to n and introduce a vector of binary variables X_j (j = 1, ... ,n)

$$x_j = \begin{cases} 1 & \text{if object } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

P_j is a measure of the profit given for object j, c the size of the knapsack,

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c,$$

$$x_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n,$$

Steps

- Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
- If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than U, then replace the value of U with this value. Then, kill all unexplored nodes which have cost function greater than this value.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
- While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

Time and Space Complexity

- Worst case time complexity - $O(2^n)$
- Best case time complexity - O(n).
- Space complexity -Exponential.

Algorithm

```
1   Algorithm UBound(cp, cw, k, m)
2   // cp, cw, k, and m have the same meanings as in
3   // Algorithm 7.11. w[i] and p[i] are respectively
4   // the weight and profit of the ith object.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           if (c + w[i] ≤ m) then
10          {
11              c := c + w[i]; b := b − p[i];
12          }
13      }
14      return b;
15  }
```

Example

Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initial upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

Solution

In this problem we will calculate lower bound and upper bound for each node. Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.

Profit = P1 + P2 + P3 = 10 + 10 +12

So, *Upper bound = 32*

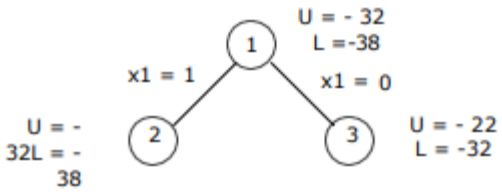
To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

$$\text{Lower bound} = 10 + 10 + 12 + \left(\frac{3}{9} X\ 18\right) = 32 + 6 = 38$$

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, *Upper bound (U) = -32 & Lower bound (L) = -38*

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



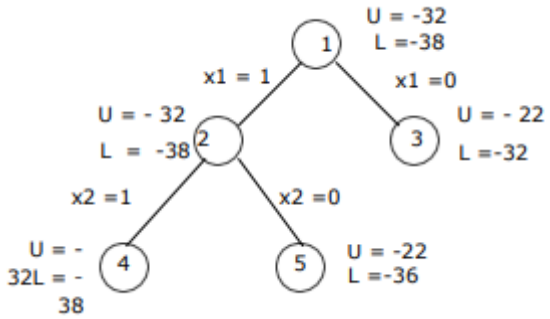
Now we will calculate upper bound and lower bound for nodes 2, 3.
For node 2, x1= 1, means we should place first item in the knapsack.
U = 10 + 10 + 12 = 32, make it as -32

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, \text{ make it as -38}$$

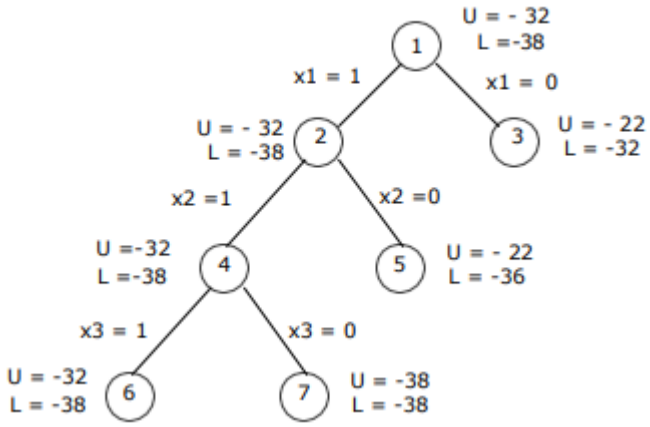
For node 3, x1 = 0, means we should not place first item in the knapsack.
U = 10 + 12 = 22, make it as -22

$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, \text{ make it as -32}$$

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3
For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10
Choose node 2, since it has minimum difference value of 6.

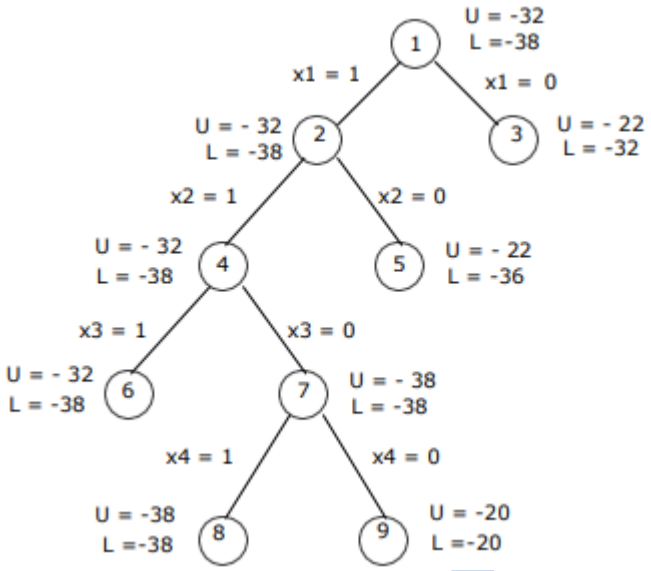


Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.
For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14
Choose node 4, since it has minimum difference value of 6



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.
For node 6, U – L = -32 + 38 =6
For node 7, U – L = -38 + 38 =0

Choose node 7, since it is minimum difference value of 0



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 =0

For node 9, U – L = -20 + 20 =0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

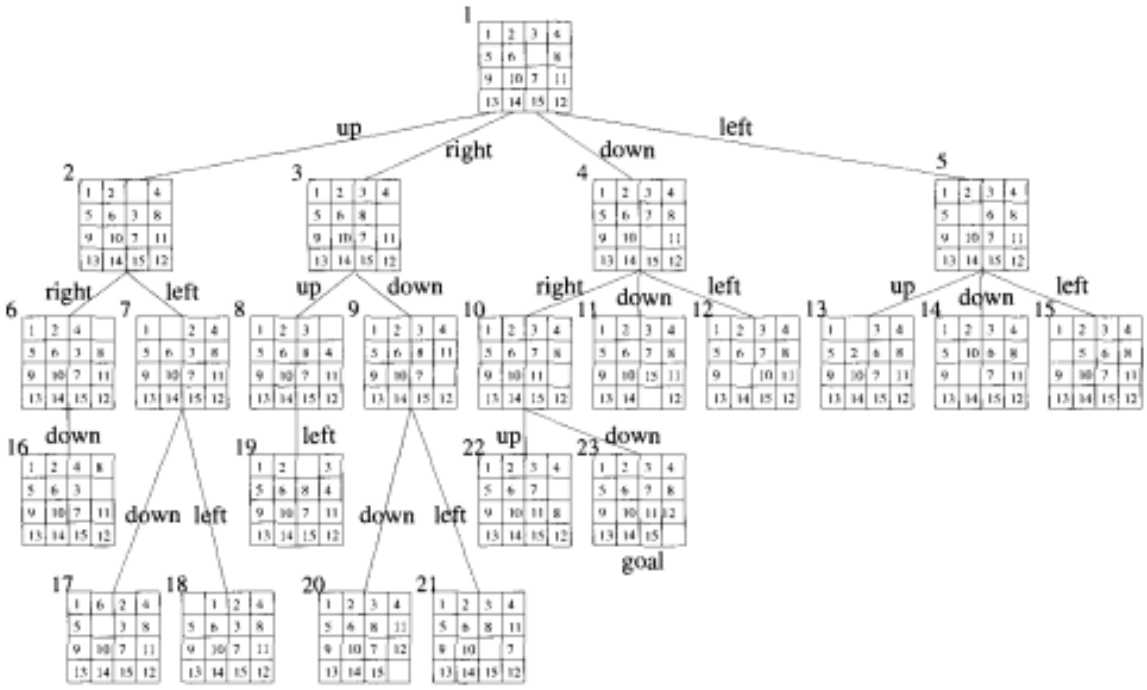
Consider the path from 1 →2 →4 →7 →8

$X_1 = 1$; $X_2 = 1$; $X_3 = 0$; $X_4 = 1$

The solution for 0/1 Knapsack problem is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

$$Maximum\ Profit = \sum P_i X_i = 10 * 1 + 10 * 1 + 12 * 0 + 18 * 1 = 38$$

Final State space tree



Edges are labeled according to the direction in which the empty space moves

Travelling Salesman Problem – Solved using Branch and Bound

Problem statement

"Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point"

Applications

- Used in network design, and transportation route design.

Main Objective

- Minimize the travelling distance.
- We can start tour from any random city and visit other cities in any order. With n cities, n! different permutations are possible.

Least Cost Branch & Bound using Static State Space Tree for Travelling Salesman Problem

Branch and bound is an effective way to find better solution in quick time by pruning some of the unnecessary branches of search tree.

Working

Consider directed weighted graph G = (V, E, W), where node represents cities and weighted directed edges represents direction and distance between two cities.

i. Initially, graph is represented by cost matrix C, where

C_{ij} = cost of edge, if there is a direct path from city i to city j

C_{ij} = ∞, if there is no direct path from city i to city j.

ii. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.

iii. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.

iv. Prepare state space tree for the reduce matrix.

v. Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.

vi. If $< i, j >$ edge is to be included, then do following :
- Set all values in row i and all values in column j of A to ∞
- Set A[j, 1] = ∞
- Reduce A again, except rows and columns having all ∞ entries

vii. Compute the cost of newly created reduced matrix as,

$$\boldsymbol{Cost} \quad = \quad \boldsymbol{L} + \boldsymbol{Cost}(\boldsymbol{i},\boldsymbol{j}) + \boldsymbol{r}$$

Where, L is cost of original reduced cost matrix and r is A[i, j].

viii. If all nodes are not visited then go to step 4.

Reduction procedure is described below:

Raw Reduction:

Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities.
M can be reduced as follow:

$$MRowRed \quad = \quad \{Mij - min\{Mij \mid 1 \leq j \leq n, and\ Mij < \infty\}\}$$

Problem:

Find the solution of following travelling salesman problem using branch and bound method.
Consider the following distance matrix:

M =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Solution:

Row reduction	
Find the minimum element from each row and subtract it from each cell of matrix.	Reduced matrix would be:

$M =$

∞	20	30	10	11	→ 10
15	∞	16	4	2	→ 2
3	5	∞	2	4	→ 2
19	6	18	∞	3	→ 3
16	4	7	16	∞	→ 4

$M_{RowRed} =$

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Row reduction cost is the summation of all the values subtracted from each rows:

Row reduction cost (M) = 10 + 2 + 2 + 3 + 4 = 21

Column reduction

Matrix MRowRed is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$$MColRed = \{Mji - min\{Mji \mid 1 \leq j \leq n, and\ Mji < \infty\}\}$$

From the above reduced matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

$M_{RowRed} =$

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞
↓	↓	↓	↓	↓
1	0	3	0	0

Column reduced matrix MColRed would be:

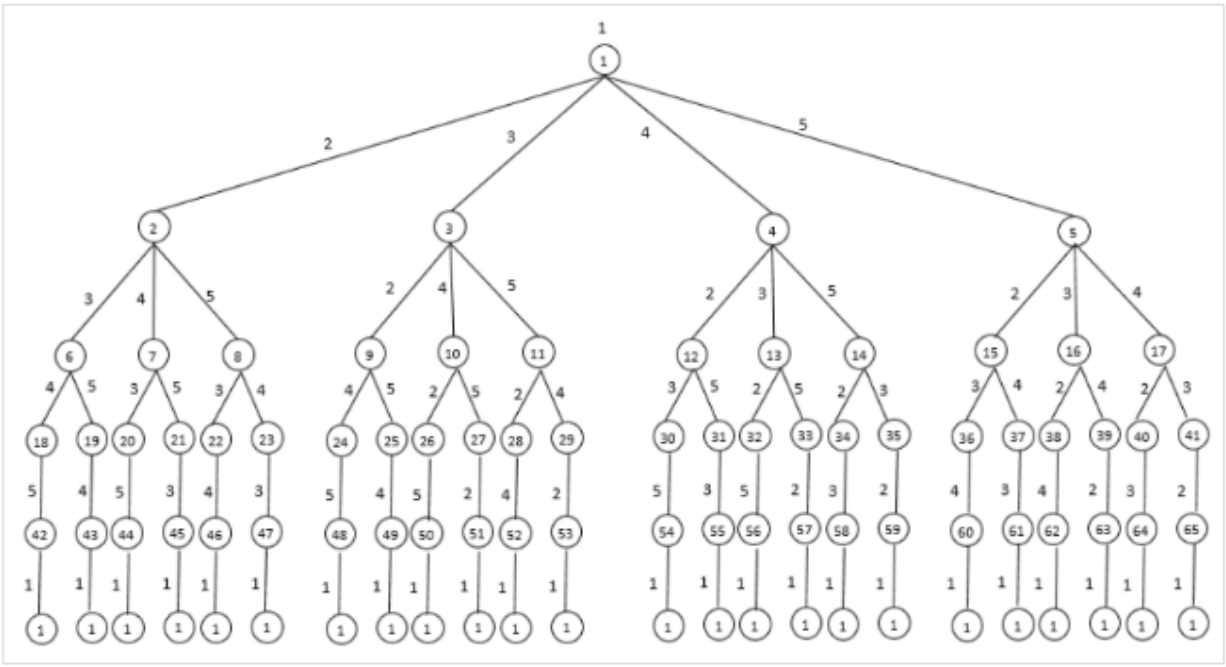
$M_{ColRed} =$

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Each row and column of MColRed has at least one zero entry, so this matrix is reduced matrix.

Column reduction cost (M) = 1 + 0 + 3 + 0 + 0 = 4

State space tree for 5 city problem is depicted as follows. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.



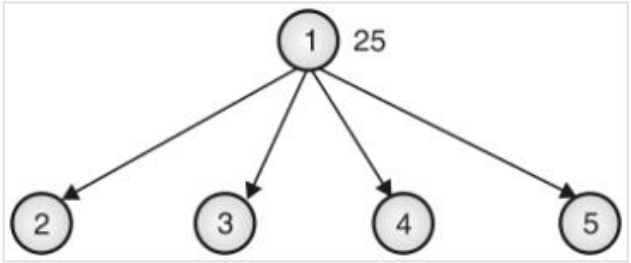
Cost of M1 = C(1)

= Row reduction cost + Column reduction cost

= (10 + 2 + 2 + 3 + 4) + (1 + 3) = 25

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree:



Let us find cost of edge from node 1 to 2, 3, 4, 5.

<u>Select edge 1-2:</u>

Set M1 [1] [ ] = M1 [ ] [2] = ∞

Set M1 [2] [1] = ∞

Reduce the resultant matrix if required.

<u>Select edge 1-3</u>

Set M1 [1][ ] = M1 [ ] [3] = ∞

Set M1 [3][1] = ∞

Reduce the resultant matrix if required.

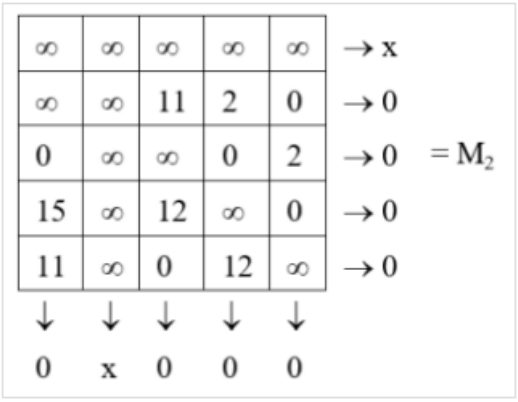
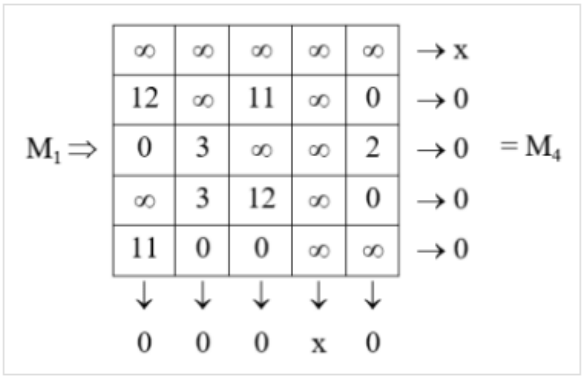
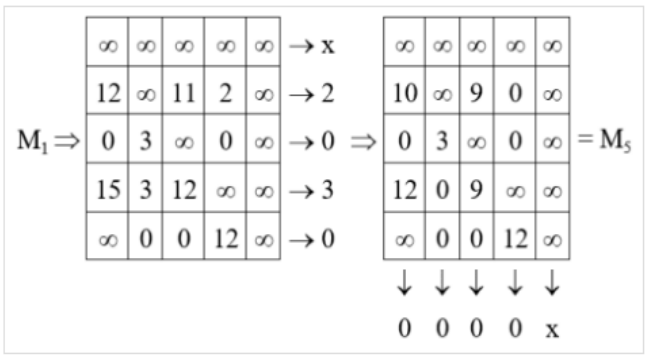
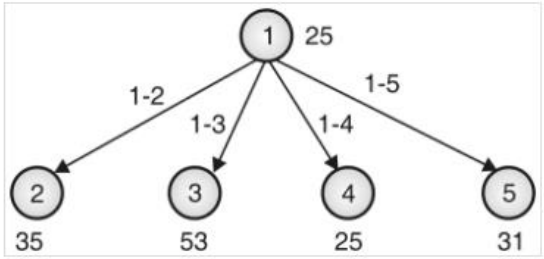
$M_1 \Rightarrow$

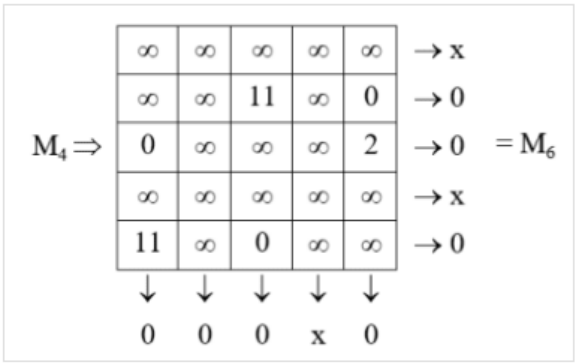
∞	∞	∞	∞	∞	→ x
12	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
11	0	x	0	0	

$\Rightarrow$

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$= M_3$

 M2 is already reduced. Cost of node 2 : C(2) = C(1) + Reduction cost + M1 [1] [2] = 25 + 0 + 10 = 35	Cost of node 3: C(3) = C(1) + Reduction cost + M1[1] [3] = 25 + 11 + 17 = 53
Select edge 1-4: Set M1 [1][ ] = M1[ ][4] = ∞ Set M1 [4][1] = ∞ Reduce resultant matrix if required.  Matrix M4 is already reduced. Cost of node 4: C(4) = C(1) + Reduction cost + M1 [1] [4] = 25 + 0 + 0 = 25	Select edge 1-5: Set M1 [1] [ ] = M1 [ ] [5] = ∞ Set M1 [5] [1] = ∞ Reduce the resultant matrix if required.  Cost of node 5: C(5) = C(1) + reduction cost + M1 [1] [5] = 25 + 5 + 1 = 31
State space diagram:  Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5. Let's explore all three nodes.	
Select path 1-4-2 : (Add edge 4-2) Set M4 [1] [] = M4 [4] [] = M4 [] [2] = ∞ Set M4 [2] [1] = ∞ Reduce resultant matrix if required.	Select edge 4-3 (Path 1-4-3): Set M4 [1] [ ] = M4 [4] [ ] = M4 [ ] [3] = ∞ Set M [3][1] = ∞ Reduce the resultant matrix if required.

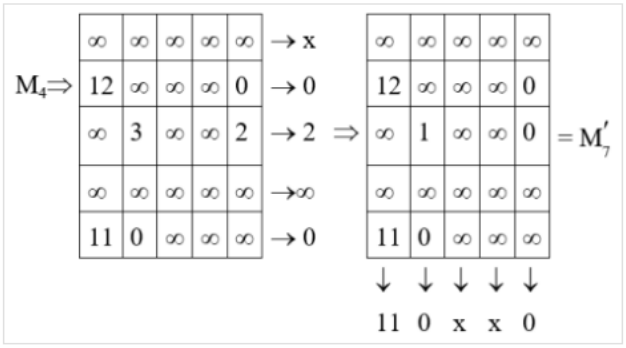


Matrix M6 is already reduced.

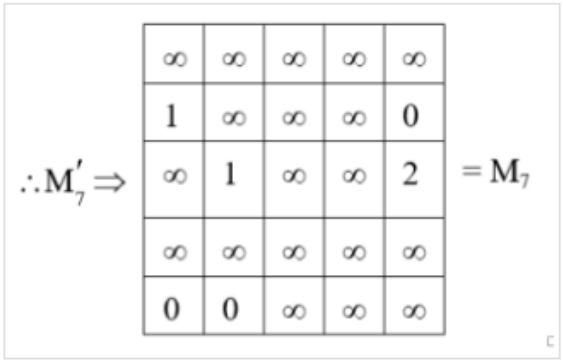
<u>Cost of node 6:</u>

C(6) = C(4) + Reduction cost + M4 [4] [2]

= 25 + 0 + 3 = 28



M'7 is not reduced. Reduce it by subtracting 11 from column 1.

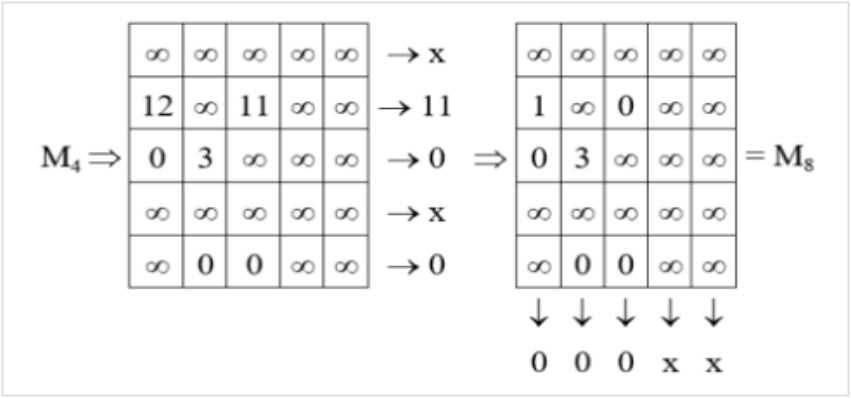


<u>Cost of node 7:</u>

C(7) = C(4) + Reduction cost + M4 [4] [3]

= 25 + 2 + 11 + 12 = 50

<u>Select edge 4-5 (Path 1-4-5):</u>



Matrix M8 is reduced.

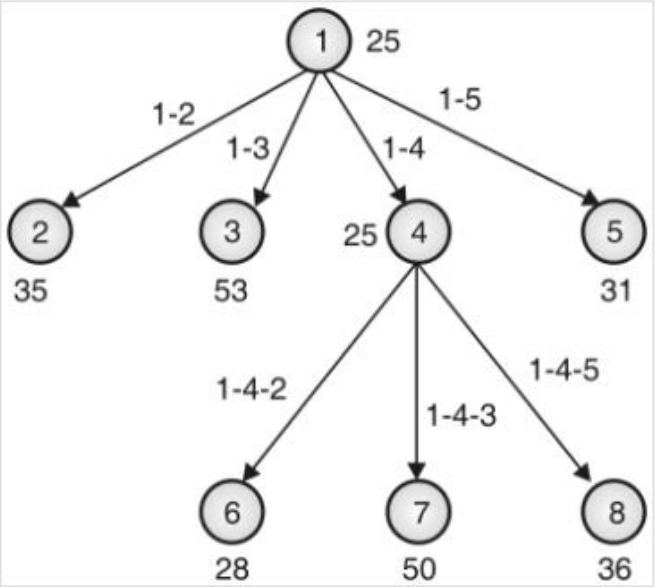
<u>Cost of node 8:</u>

C(8) = C(4) + Reduction cost + M4 [4][5]

= 25 + 11 + 0 = 36

State space tree

Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.



Add edge 2-3 (Path 1-4-2-3):

Set M6 [1][ ] = M6 [4][ ] = M6 [2][ ] = M6 [][3] = ∞

Set M6 [3][1] = ∞

Reduce resultant matrix if required.

$M_6 \Rightarrow$

∞	∞	∞	∞	∞	→ x
∞	∞	∞	∞	∞	→ x
0	∞	∞	∞	2	→ 0
∞	∞	∞	∞	∞	→ x
11	∞	∞	∞	∞	→ 11

$\Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	2
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

$= M_9'$

↓ ↓ ↓ ↓ ↓

0 x x x 2

$\therefore M_9' \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

$= M_9$

Cost of node 9:

C(9) = C(6) + Reduction cost + M6 [2][3]

= 28 + 11 + 2 + 11 = 52

Add edge 2-5 (Path 1-4-2-5):

Set M6 [1][ ] = M6 [4][ ] = M6 [2][ ] = M6 [ ][5] = ∞

Set M6 [5][1] = ∞

Reduce resultant matrix if required.

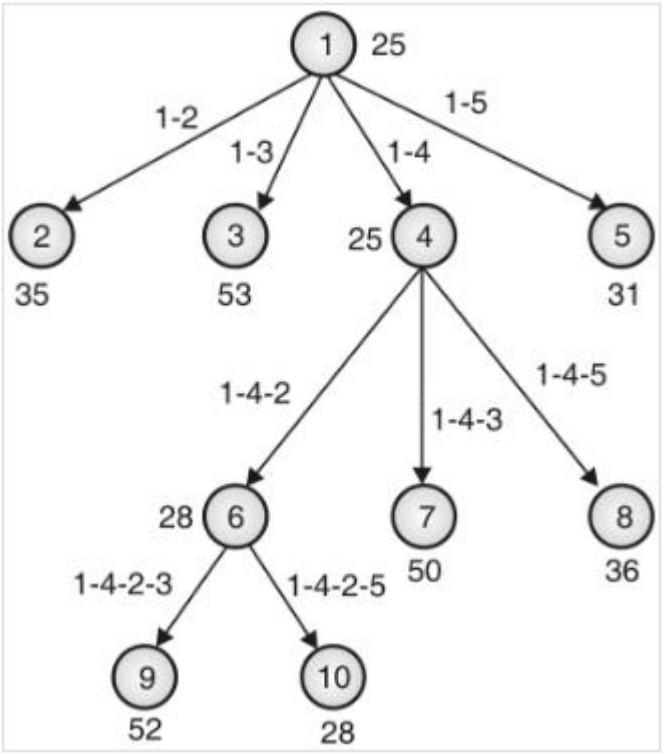
$\therefore M_6 \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

$= M_{10}$

Cost of node 10:
C(10) = C(6) + Reduction cost + M6 [2][5]
= 28 + 0 + 0 = 28

State space tree



Add edge 5-3 (Path 1-4-2-5-3):

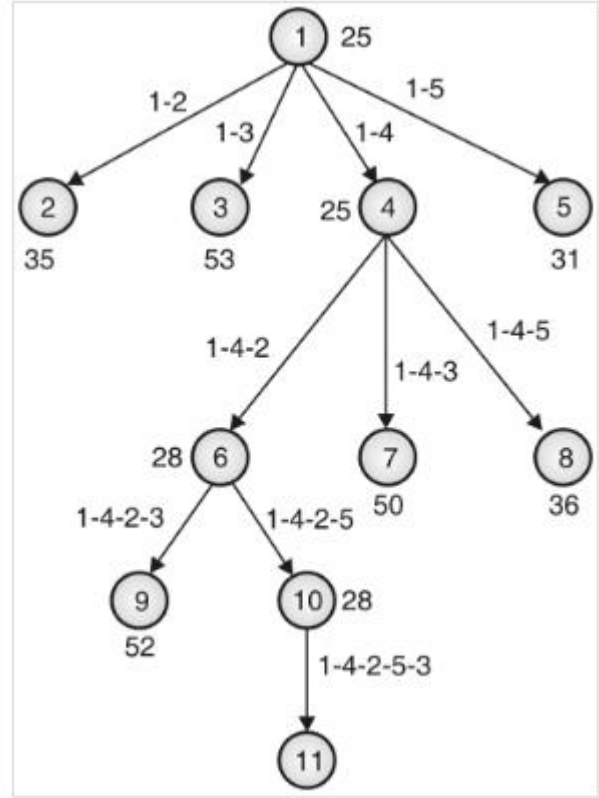
$\therefore M_{10} \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

$= M_{11}$

Cost of node 11:
C(11) = C(10) + Reduction cost + M10 [5][3]
= 28 + 0 + 0 = 28

State space tree



Shortest Path is 1-4-2-5-3