Design and Analysis of Algorithm

UNIT – 5: Backtracking and Branch and Bound

Introduction to Backtracking Algorithm:

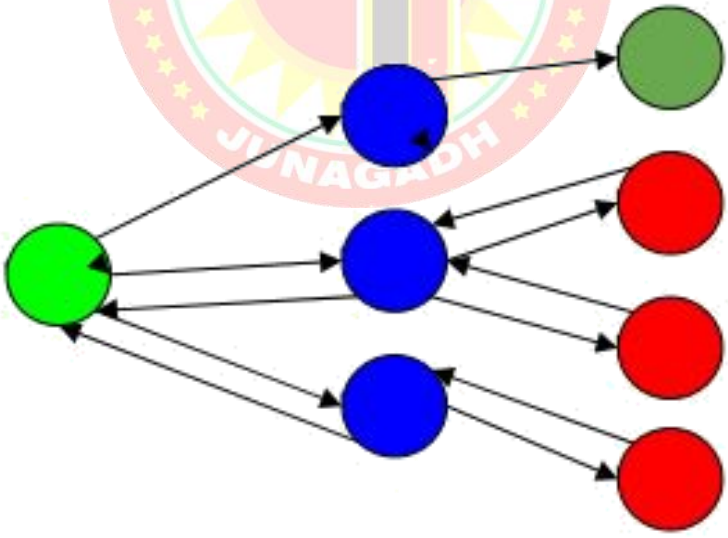
Backtracking is a technique based on **algorithm** to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example,



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution.

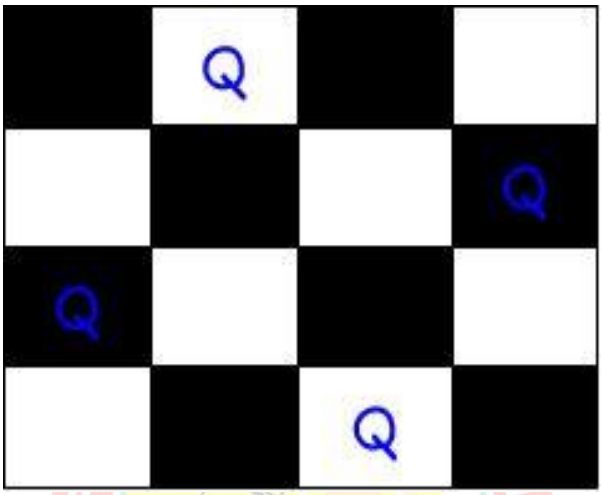
Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find track to the next point to find solution.

The N - queen's problem:

In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other.

A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way.

Here, the solution is −



Here, we will do 8-Queen problem.

Input:

The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a normal chess board.)

Output:

The matrix that represents in which row and column the N Queens can be placed. If the solution does not exist, it will return false.

1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

In this output, the value 1 indicates the correct place for the queens.

The 0 denotes the blank spaces on the chess board.

Algorithm:

IsValid (board, row, col)

Input: The chess board, row and the column of the board.

Output − True when placing a queen in row and place position is a valid or not.

```
Begin
   if there is a queen at the left of current col, then
      return false
   if there is a queen at the left upper diagonal, then
      return false
   if there is a queen at the left lower diagonal, then
      return false;
   return true //otherwise it is valid place
End
```

solveNQueen (board, col)

Input − The chess board, the col where the queen is trying to be placed.

Output − The position matrix where queens are placed.

```
Begin
   if all columns are filled, then
      return true
   for each row of the board, do
      if isValid(board, i, col), then
         set queen at place (i, col) in the board
         if solveNQueen(board, col+1) = true, then
            return true
         otherwise remove queen from place (i, col) from board.
   done
   return false
End
```

Example:

```
#include<iostream>
using namespace std;
#define N 8

void printBoard(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         cout << board[i][j] << " ";
      cout << endl;
   }
}

bool isValid(int board[N][N], int row, int col) {
   for (int i = 0; i < col; i++)    //check whether there is queen in the left or not
      if (board[row][i])
         return false;
   for (int i=row, j=col; i>=0 && j>=0; i--, j--)
      if (board[i][j])       //check whether there is queen in the left upper diagonal or not
         return false;
   for (int i=row, j=col; j>=0 && i<N; i++, j--)
      if (board[i][j])      //check whether there is queen in the left lower diagonal or not
         return false;
   return true;
}

bool solveNQueen(int board[N][N], int col) {
   if (col >= N)           //when N queens are placed successfully
      return true;
   for (int i = 0; i < N; i++) {     //for each row, check placing of queen is possible or not
      if (isValid(board, i, col) ) {
         board[i][col] = 1;      //if validate, place the queen at place (i, col)
         if ( solveNQueen(board, col + 1))    //Go for the other columns recursively
            return true;
         board[i][col] = 0;        //When no place is vacant remove that queen
      }
   }
   return false;       //when no possible order is found
}

bool checkSolution() {
   int board[N][N];
   for(int i = 0; i<N; i++)
      for(int j = 0; j<N; j++)
         board[i][j] = 0;      //set all elements to 0
```

```
   if ( solveNQueen(board, 0) == false ) {     //starting from 0th column
      cout << "Solution does not exist";
      return false;
   }
   printBoard(board);
   return true;
}

int main() {
   checkSolution();
}
```

Output:

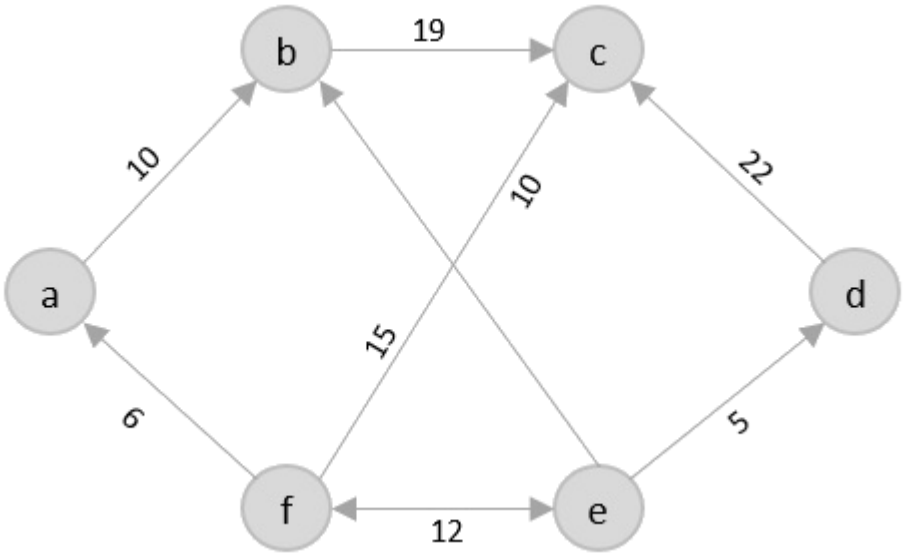
```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

Travelling Salesman Problem:

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known.

The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the vertex ‘a’, they need to travel through all the remaining vertices b, c, d, e, f and get back to ‘a’ while making sure that the cost taken is minimum.



There are various approaches to find the solution to the travelling salesman problem: naïve approach, greedy approach, dynamic programming approach, etc.

In this tutorial we will be learning about solving travelling salesman problem using backtracking approach.

Travelling Salesperson Algorithm:

As the definition for backtracking approach states, we need to find the best optimal solution locally to figure out the global optimal solution.

The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

Algorithm:

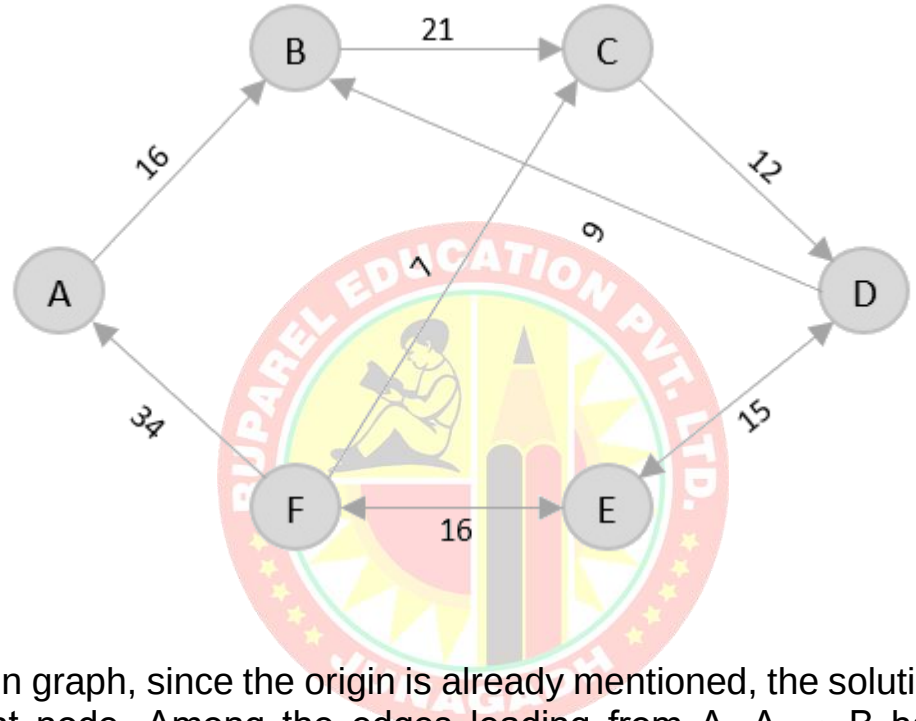
- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.

- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.

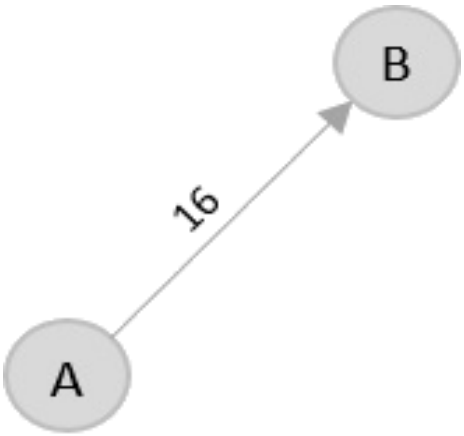
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples:

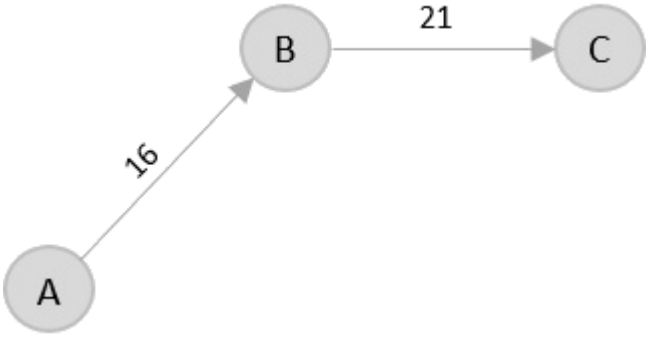
Consider the following graph with six cities and the distances between them –



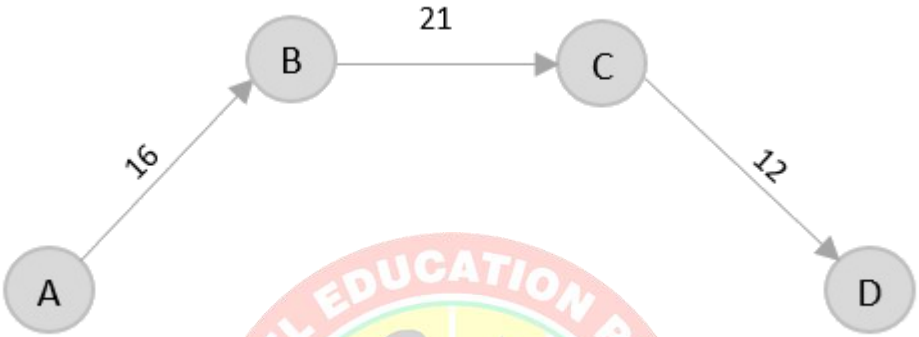
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



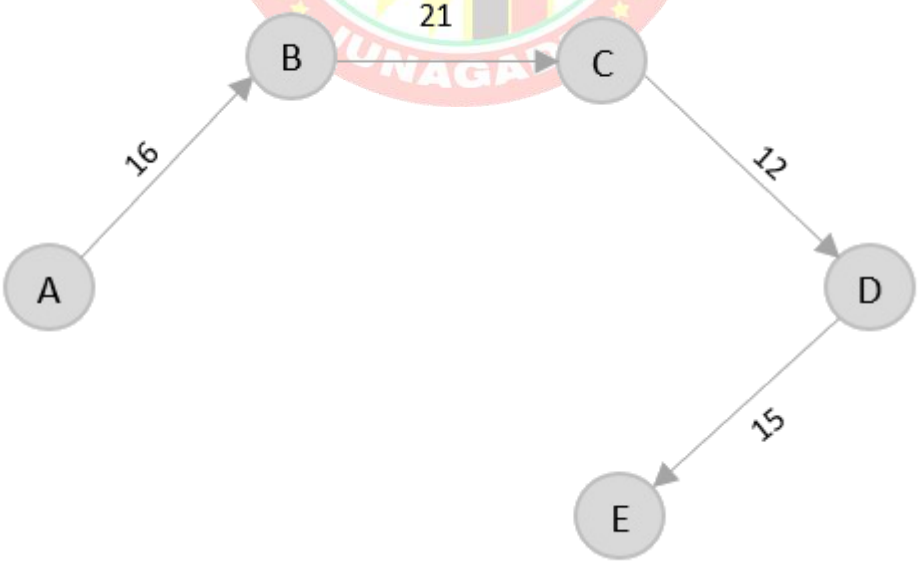
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



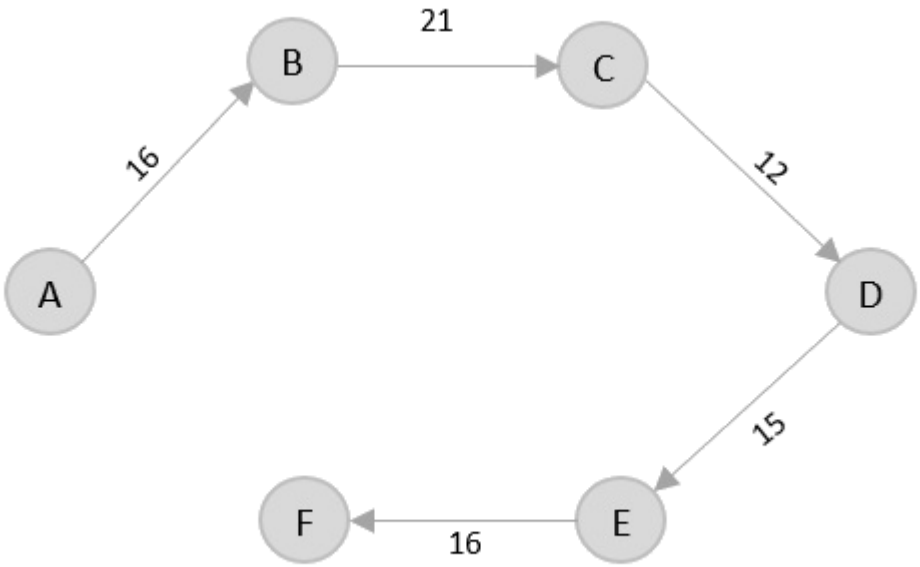
There's only one edge between C → D, therefore it is added to the output graph.



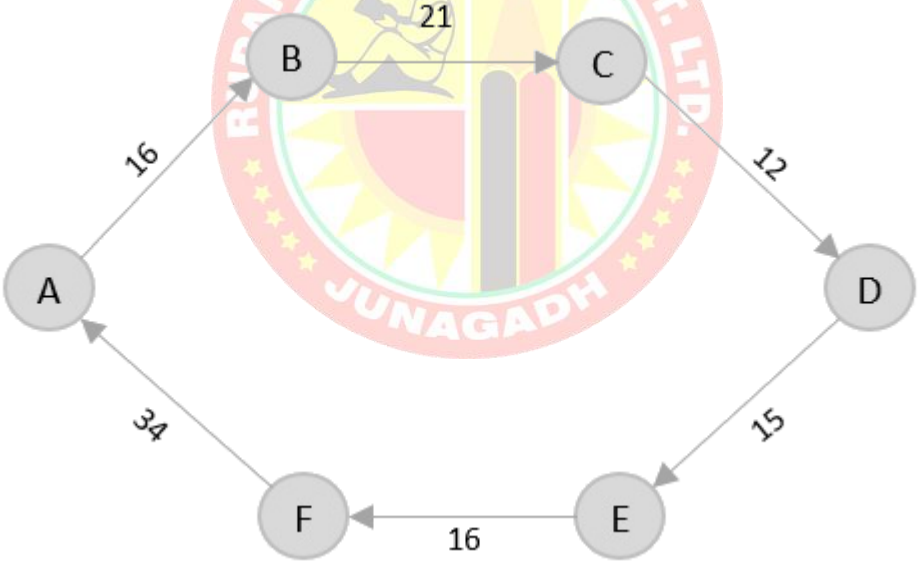
There's two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.



There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = **114**.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.

Mini-Max Algorithm in Games:

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

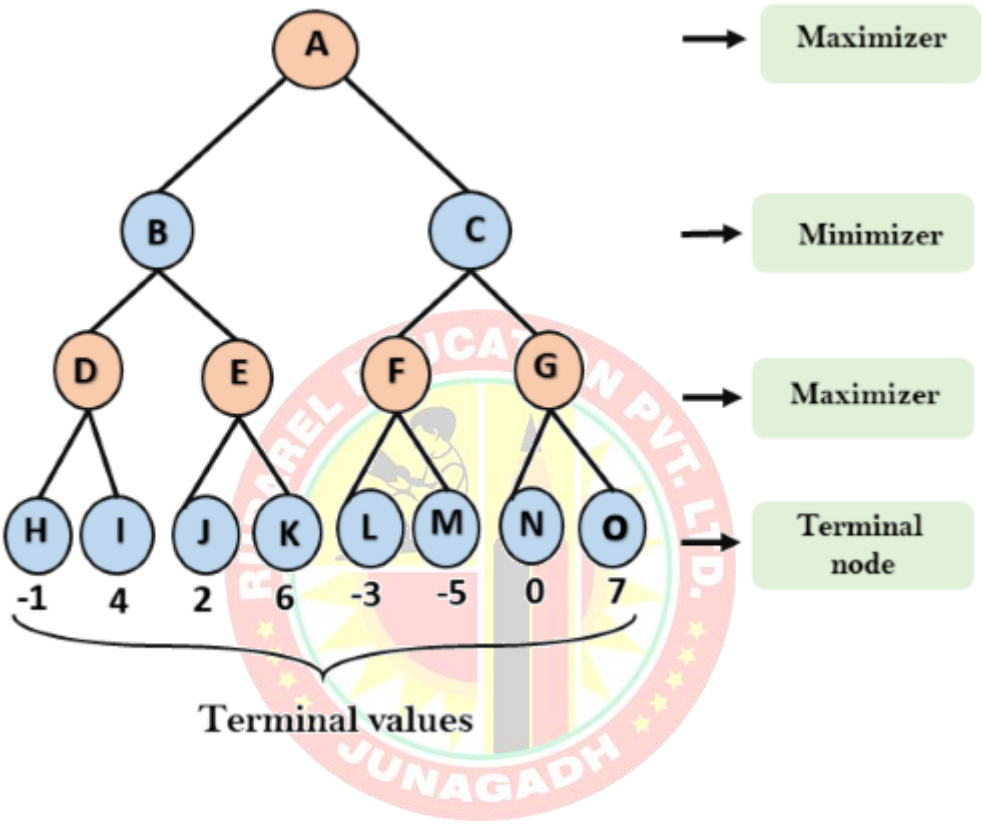
Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1:

In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states.

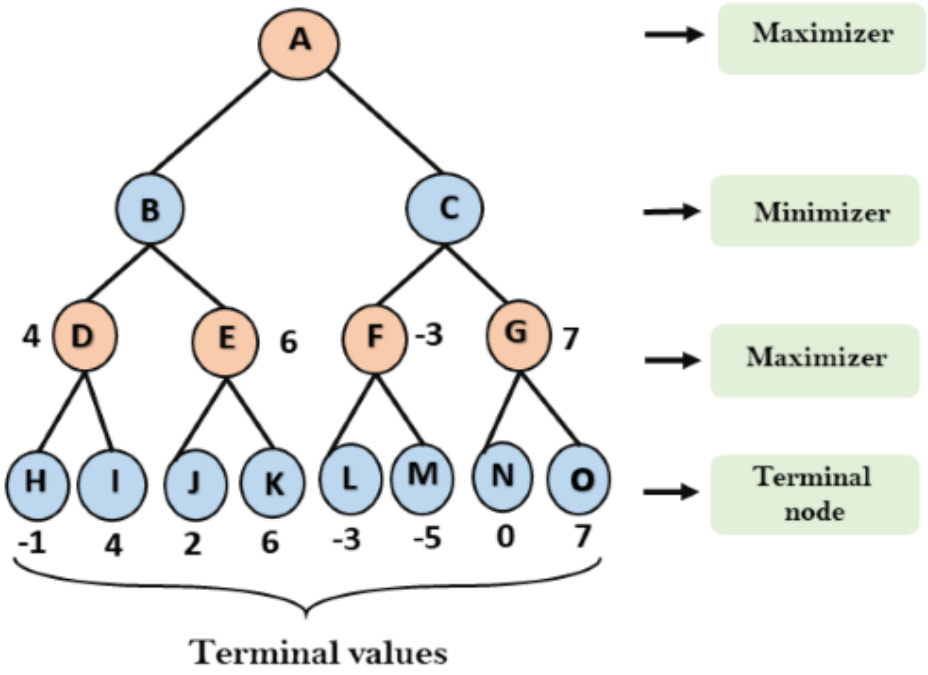
In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



Step 2:

Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

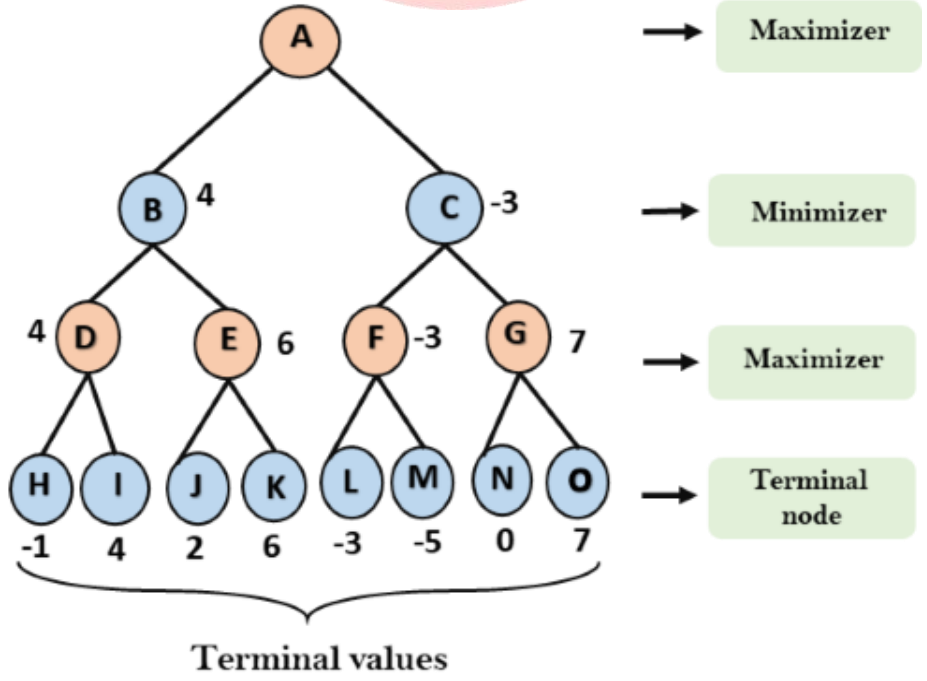
- For node D max (-1, 4) = 4
- For Node E max (2, 6) = 6
- For Node F max (-3, -5) = -3
- For node G max (0, 7) = 7



Step 3:

In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

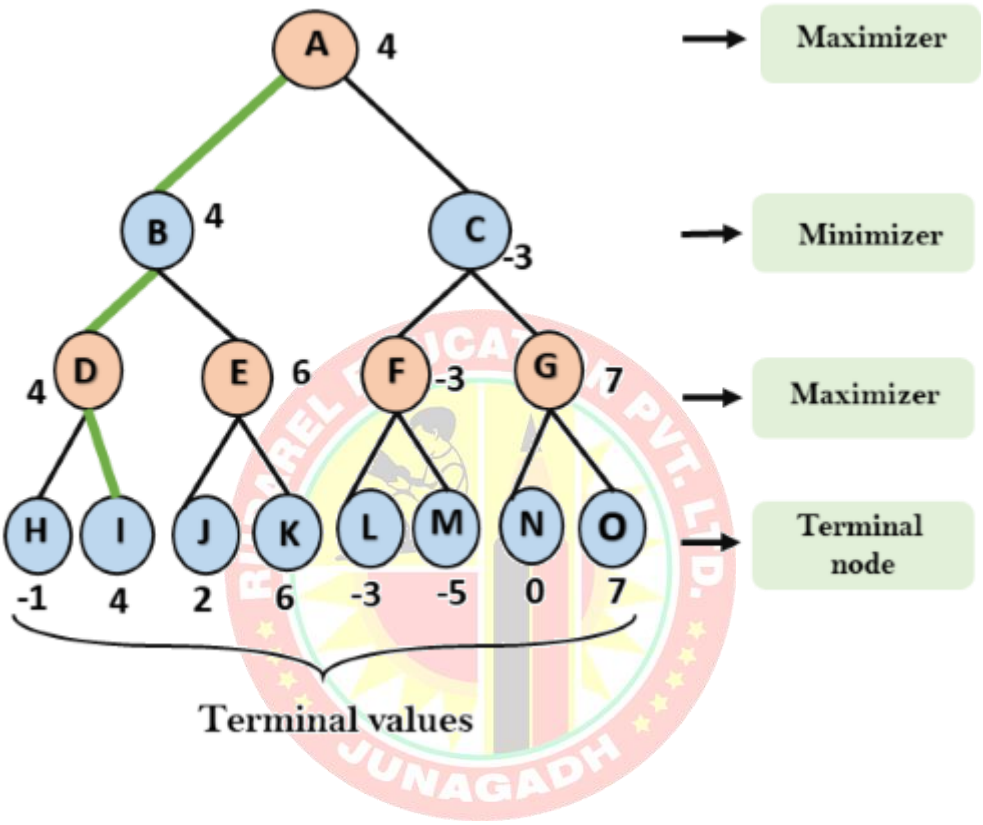
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4:

Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max (4, -3)= 4



That was the complete workflow of the minimax two player game.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide.