BASUDEV GODABARI DEGREE COLLEGE, KESAIBAHAL

Department of Computer Science

"Self Study Module"

Module Details:

- **Class – 6th Semester**
- **Subject Name: COMPUTER SCIENCE**
- **Paper Name : "Artificial Intelligence"**

UNIT-2: STRUCTURE

Problem Solving and Searching Techniques: Problem Characteristics, Production Systems, Control Strategies, Breadth First Search, Depth First Search, Hill climbing and its Variations, 27 Heuristics Search Techniques: Best First Search, A* algorithm, Constraint Satisfaction Problem, Introduction to Game Playing, Min-Max and Alpha-Beta pruning algorithms.

Learning Objective

After Learning this unit you should be able to
Know Automation, Smart Decision Making
Enhanced Customer Experience
Medical Advances, Solving Complex Problems, Managing Repetitive Tasks

You Can use the Following Learning Video link related to above topic:

https://youtu.be/uB3i-qV6VdM?list=PLxCzCOWd7aiHGhOHV-nwb0HR5US5GFKFl
https://youtu.be/s-s9ilkMVj8?list=PLxCzCOWd7aiHGhOHV-nwb0HR5US5GFKFl
https://youtu.be/qul0f79gxGs?list=PLxCzCOWd7aiHGhOHV-nwb0HR5US5GFKFl
https://youtu.be/2Tp3aSz8jFE

You Can also use the following Books

Text books

Artificial Intelligence a Modern Approach, Stuart Russell and Peter Norvig, Pearson 3/ed.

Reference books

1. Artificial Intelligence, Rich &Knight , TMG , 3 e/d.
2. DAN.W. Patterson, Introduction to A.I and Expert Systems – PHI, 2007 3. W.F. Clocksin

https://www.pdfdrive.com/

Problem Solving Techniques in AI

The process of problem-solving is frequently used to achieve objectives or resolve particular situations. In computer science, the term "problem-solving" refers to artificial intelligence methods, which may include formulating ensuring appropriate, using algorithms, and conducting root-cause analyses that identify reasonable solutions. Artificial intelligence (AI) problem-solving often involves investigating potential solutions to problems through reasoning techniques, making use of polynomial and differential equations, and carrying them out and use modelling frameworks. A same issue has a number of solutions, that are all accomplished using an unique algorithm. Additionally, certain issues have original remedies. Everything depends on how the particular situation is framed.

Cases involving Artificial Intelligence Issues

Artificial intelligence is being used by programmers all around the world to automate systems for effective both resource and time management. Games and puzzles can pose some of the most frequent issues in daily life. The use of ai algorithms may effectively tackle this. Various problem-solving methods are implemented to create solutions for a variety complex puzzles, includes mathematics challenges such crypto-arithmetic and magic squares, logical puzzles including Boolean formulae as well as N-Queens, and quite well games like Sudoku and Chess. Therefore, these below represent some of the most common issues that artificial intelligence has remedied:

- Chess
- N-Queen problem
- Tower of Hanoi Problem
- Travelling Salesman Problem
- Water-Jug Problem

A Reflex Agent: But What's It?

Depending on their ability for recognising intelligence, these five main artificial intelligence agents were deployed today. The below would these be agencies:

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents
- Learning Agents

This mapping of states and actions is made easier through these agencies. These agents frequently make mistakes when moving onto the subsequent phase of a complicated issue; hence, problem-solving standardized criteria such cases. Those agents employ artificial intelligence can tackle issues utilising methods like B-tree and heuristic algorithms.

Approaches for Resolving Problems

The effective approaches of artificial intelligence make it useful for resolving complicated issues. All fundamental problem-solving methods used throughout AI were listed below. In accordance with the criteria set, students may learn information regarding different problem-solving methods.

Heuristics

The heuristic approach focuses solely upon experimentation as well as test procedures to comprehend a problem and create a solution. These heuristics don't always offer better ideal answer to something like a particular issue, though. Such, however, unquestionably provide effective means of achieving short-term objectives. Consequently, if conventional techniques are unable to solve the issue effectively, developers turn to them. Heuristics are employed in conjunction with optimization algorithms to increase the efficiency because they merely offer moment alternatives while compromising precision.

Searching Algorithms

Several of the fundamental ways that AI solves every challenge is through searching. These searching algorithms are used by rational agents or problem-solving agents for select the most appropriate answers. Intelligent entities use molecular representations and seem to be frequently main objective when finding solutions. Depending upon that calibre of the solutions they produce, most searching algorithms also have attributes of completeness, optimality, time complexity, and high computational.

Computing Evolutionary

This approach to issue makes use of the well-established evolutionary idea. The idea of "survival of the fittest underlies the evolutionary theory. According to this, when a creature successfully reproduces in a tough or changing environment, these coping mechanisms are eventually passed down to the later generations, leading to something like a variety of new young species. By combining several traits that go along with that severe environment, these mutated animals aren't just clones of something like the old ones. The much more notable example as to how development is changed and expanded is humanity, which have done so as a consequence of the accumulation of advantageous mutations over countless generations.

Genetic Algorithms

Genetic algorithms have been proposed upon that evolutionary theory. These programs employ a technique called direct random search. In order to combine the two healthiest possibilities and produce a desirable offspring, the developers calculate the fit factor. Overall health of each individual is determined by first gathering demographic information and afterwards assessing each individual. According on how well each member matches that intended need, a calculation is made. Next, its creators employ a variety of methodologies to retain their finest participants.

1. Rank Selection
2. Tournament Selection
3. Steady Selection
4. Roulette Wheel Selection (Fitness Proportionate Selection)
5. Elitism

BFS Algorithm in Java

What is BFS?

Breadth-First Search (BFS) is based on traversing nodes by adding the neighbors of each node to the traversal queue starting from the root node. The BFS for a graph is similar to that of a tree, with the exception that graphs may have cycles. In contrast to depth-first search, all neighbor nodes at a given depth are investigated before proceeding to the next level.

BFS Algorithm

The following are the steps involved in employing breadth-first search to explore a graph:

1. Take the data for the graph's adjacency matrix or adjacency list.
2. Create a queue and fill it with items.
3. Activate the root node (meaning that get the root node at the beginning of the queue).
4. Dequeue the queue's head (or initial element), then enqueue all of the queue's nearby nodes from left to right. Simply dequeue the head and resume the operation if a node has no nearby nodes that need to be investigated. (Note: If a neighbor emerges that has previously been investigated or is in the queue, don't enqueue it; instead, skip it.)
5. Continue in this manner until the queue is empty.

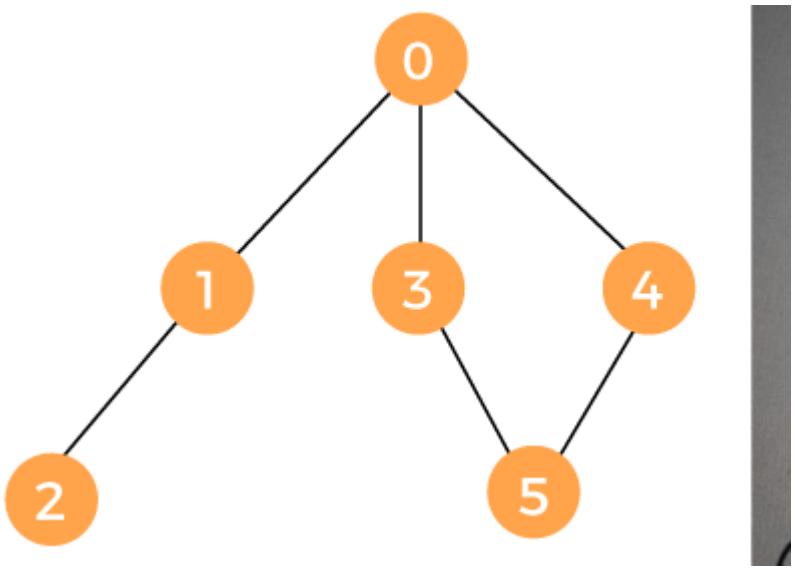
BFS Applications

Because of the algorithm's flexibility, Breadth-First Search is quite useful in the real world. These are some of them:

1. In a peer-to-peer network, peer nodes are discovered. Most torrent clients, such as BitTorrent, uTorrent, and qBittorent, employ this process to find "seeds" and "peers" in the network.
2. The index is built using graph traversal techniques in web crawling. The procedure starts with the source page as the root node and works its way down to all secondary pages that are linked to the source page (and this process continues). Because of the reduced depth of the recursion tree, Breadth-First Search has an inherent advantage here.
3. The use of GPS navigation systems using the GPS, conduct a breadth-first search to locate nearby sites.
4. Cheney's technique, which employs the concept of breadth-first search, is used to collect garbage.

Example BFS Traversal

To get started, let's look at a simple example. We'll start with 0 as the root node and work our way down the graph.



Step 1: Enqueue(0)

Queue

```
0
```

Step 2: Dequeue(0), Enqueue(1), Enqueue(3), Enqueue(4)

Queue

1	3	4

Step 3: Dequeue(1), Enqueue(2)

Queue

3	4	2

Step 4: Dequeue(3), Enqueue(5). We won't add 1 to the queue again because 0 has already been explored.

Queue

4	2	5

Step 5: Dequeue(4)

Queue

2	5

Step 6: Dequeue(2)

Queue

5

Step 7: Dequeue(5)

Queue

The queue is empty now so we'll stop the process.

Breadth-First Search Java Program

There are several approaches of dealing with the code. We'll mostly discuss the steps involved in implementing a breadth first search in Java. An adjacency list or an adjacency matrix can be used to store graphs; either method is acceptable. The adjacency list will be used to represent our graph in our code. When implementing the Breadth-First Search algorithm in Java, it is much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head (or start) of the queue.

The graph used to demonstrate the code will be identical to the one used in the previous example.

BFSTraversal.java

```
import java.io.*;
import java.util.*;
```

```
public class BFSTraversal
{
   private int node;       /* total number number of nodes in the graph */
   private LinkedList<Integer> adj[];     /* adjacency list */
   private Queue<Integer> que;          /* maintaining a queue */
   BFSTraversal(int v)
   {
      node = v;
      adj = new LinkedList[node];
      for (int i=0; i<v; i++)
      {
         adj[i] = new LinkedList<>();
      }
      que = new LinkedList<Integer>();
   }
   void insertEdge(int v,int w)
   {
      adj[v].add(w);      /* adding an edge to the adjacency list (edges are bidirectional in this example) */
   }
   void BFS(int n)
   {
      boolean nodes[] = new boolean[node];       /* initialize boolean array for holding the data */
      int a = 0;
      nodes[n]=true;
      que.add(n);       /* root node is added to the top of the queue */
      while (que.size() != 0)
      {
         n = que.poll();       /* remove the top element of the queue */
         System.out.print(n+" ");    /* print the top element of the queue */
         for (int i = 0; i < adj[n].size(); i++)  /* iterate through the linked list and push all neighbors into queue *
/
         {
            a = adj[n].get(i);
            if (!nodes[a])      /* only insert nodes into queue if they have not been explored already */
            {
               nodes[a] = true;
               que.add(a);
            }
         }
      }
   }
   public static void main(String args[])
   {
      BFSTraversal graph = new BFSTraversal(6);
      graph.insertEdge(0, 1);
      graph.insertEdge(0, 3);
      graph.insertEdge(0, 4);
      graph.insertEdge(4, 5);
      graph.insertEdge(3, 5);
      graph.insertEdge(1, 2);
      graph.insertEdge(1, 0);
      graph.insertEdge(2, 1);
      graph.insertEdge(4, 1);
      graph.insertEdge(3, 1);
      graph.insertEdge(5, 4);
      graph.insertEdge(5, 3);
      System.out.println("Breadth First Traversal for the graph is:");
      graph.BFS(0);
```

```
        }
    }
```

Output:

```
Breadth First Traversal for the graph is:
0 1 3 4 2 5
```

DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

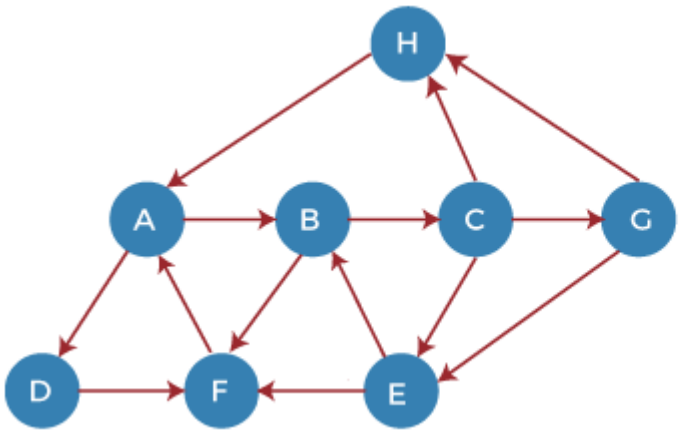
[END OF LOOP **Step 6:** EXIT

Pseudocode

```
1.  DFS(G,v)   ( v is the vertex where the search starts )
2.        Stack S := {};   ( start with an empty stack )
3.        for each vertex u, set visited[u] := false;
4.        push S, v;
5.        while (S is not empty) do
6.          u := pop S;
7.          if (not visited[u]) then
8.            visited[u] := true;
9.            for each unvisited neighbour w of uu
10.             push S, w;
11.         end if
12.       end while
13.    END DFS()
```

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1.] Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

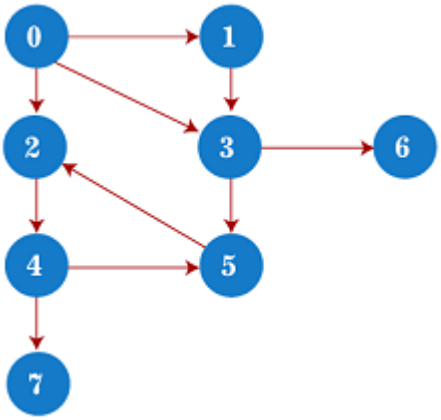
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Implementation of DFS algorithm

Now, let's see the implementation of DFS algorithm in Java.

In this example, the graph that we are using to demonstrate the code is given as follows -




```
1.  /*A sample java program to implement the DFS algorithm*/
2.
3.  import java.util.*;
4.
5.  class DFSTraversal {
6.    private LinkedList<Integer> adj[]; /*adjacency list representation*/
7.    private boolean visited[];
8.
9.    /* Creation of the graph */
10.   DFSTraversal(int V) /*'V' is the number of vertices in the graph*/
11.   {
12.     adj = new LinkedList[V];
13.     visited = new boolean[V];
14.
15.     for (int i = 0; i < V; i++)
16.       adj[i] = new LinkedList<Integer>();
17.   }
18.
19.   /* Adding an edge to the graph */
20.   void insertEdge(int src, int dest) {
21.     adj[src].add(dest);
22.   }
23.
24.   void DFS(int vertex) {
25.     visited[vertex] = true; /*Mark the current node as visited*/
26.     System.out.print(vertex + " ");
27.
28.     Iterator<Integer> it = adj[vertex].listIterator();
```

```
    while (it.hasNext()) {
      int n = it.next();
      if (!visited[n])
        DFS(n);
    }
  }

  public static void main(String args[]) {
    DFSTraversal graph = new DFSTraversal(8);

        graph.insertEdge(0, 1);
        graph.insertEdge(0, 2);
        graph.insertEdge(0, 3);
        graph.insertEdge(1, 3);
        graph.insertEdge(2, 4);
        graph.insertEdge(3, 5);
        graph.insertEdge(3, 6);
        graph.insertEdge(4, 7);
        graph.insertEdge(4, 5);
        graph.insertEdge(5, 2);

        System.out.println("Depth First Traversal for the graph is:");
        graph.DFS(0);
  }
}
```

Output

```
Depth First Traversal for the graph is:
0 1 3 5 2 4 7 6
```

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

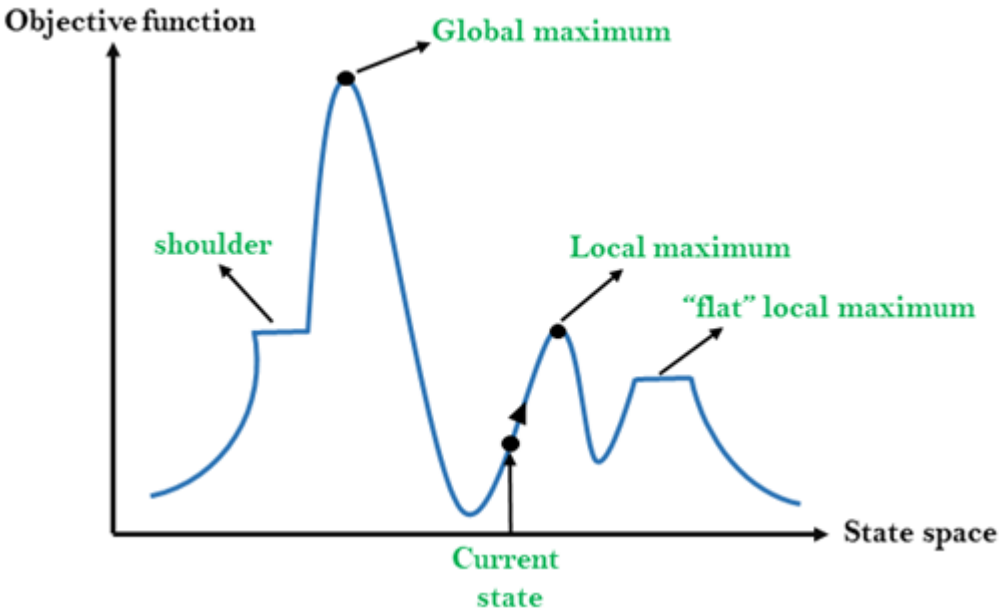
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:

- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    a. If it is goal state, then return success and quit.
    b. Else if it is better than the current state then assign new state as a current state.
    c. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
    a. Let SUCC be a state such that any successor of the current state will be better than it.
    b. For each operator that applies to the current state:
        a. Apply the new operator and generate a new state.
        b. Evaluate the new state.
        c. If it is goal state, then return it and quit, else compare it to the SUCC.
        d. If it is better than SUCC, then set new state as SUCC.
        e. If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Mini-Max Algorithm in Artificial Intelligence

- 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.

Pseudo-code for MinMax Algorithm:

```
1. function minimax(node, depth, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
5. if MaximizingPlayer then      // for Maximizer Player
6. maxEva= -infinity
7.  for each child of node do
8.  eva= minimax(child, depth-1, false)
9. maxEva= max(maxEva,eva)       //gives Maximum of the values
10. return maxEva
11.
12. else                 // for Minimizer player
13.  minEva= +infinity
14.  for each child of node do
15.  eva= minimax(child, depth-1, true)
16.  minEva= min(minEva, eva)       //gives minimum of the values
17.  return minEva
```

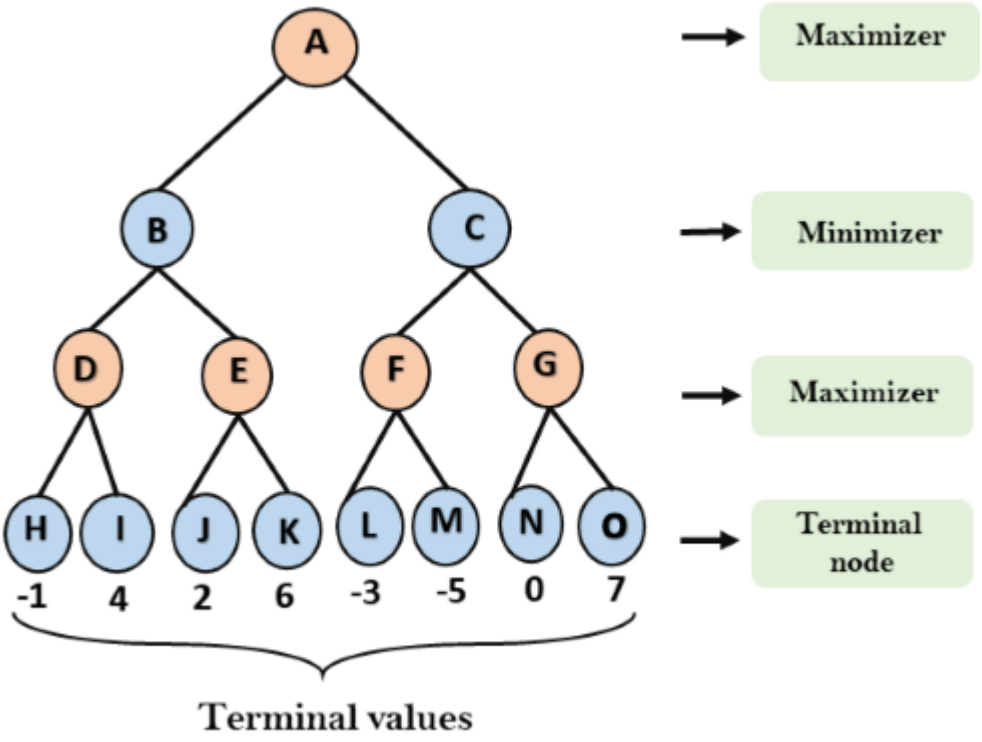
Initial call:

Minimax(node, 3, true)

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:

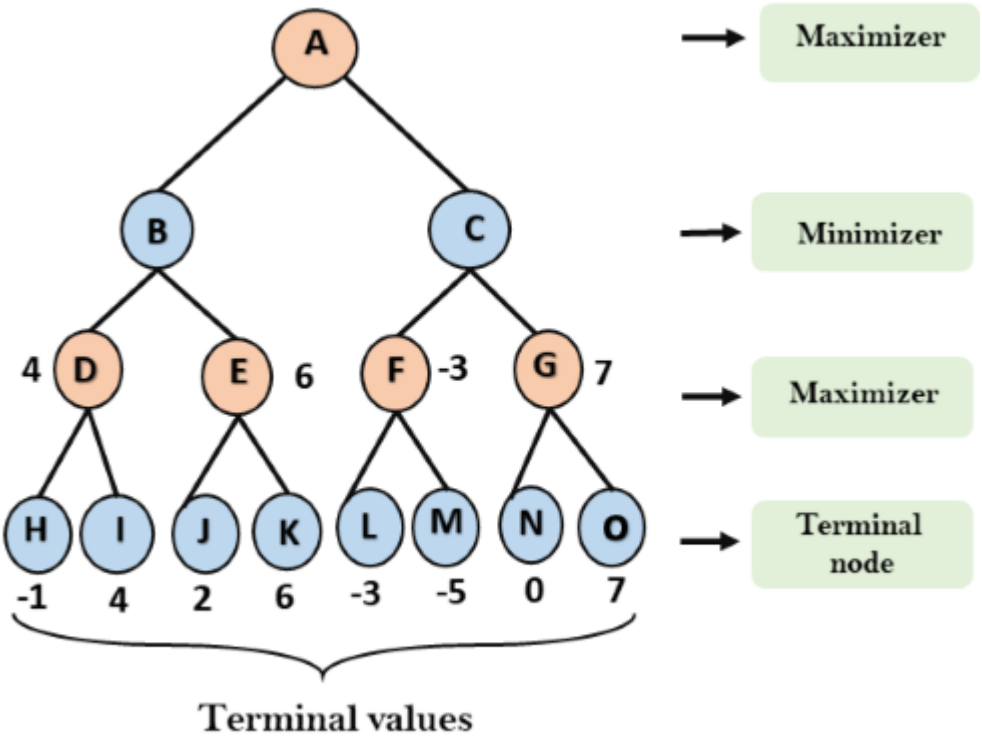
Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- 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.

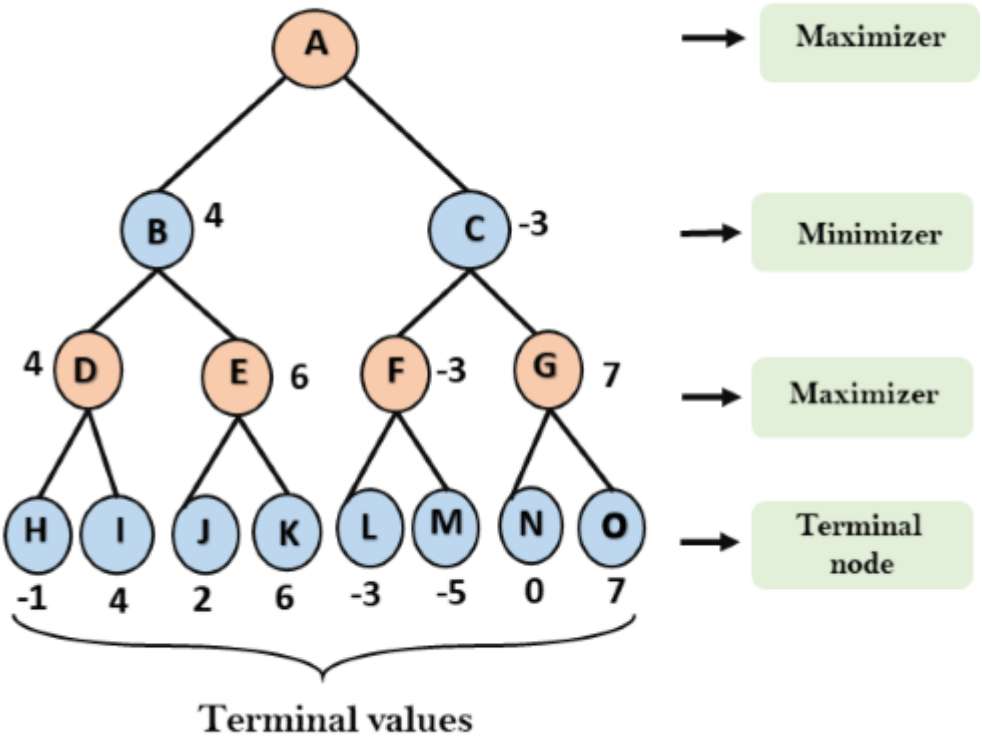
- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6

- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



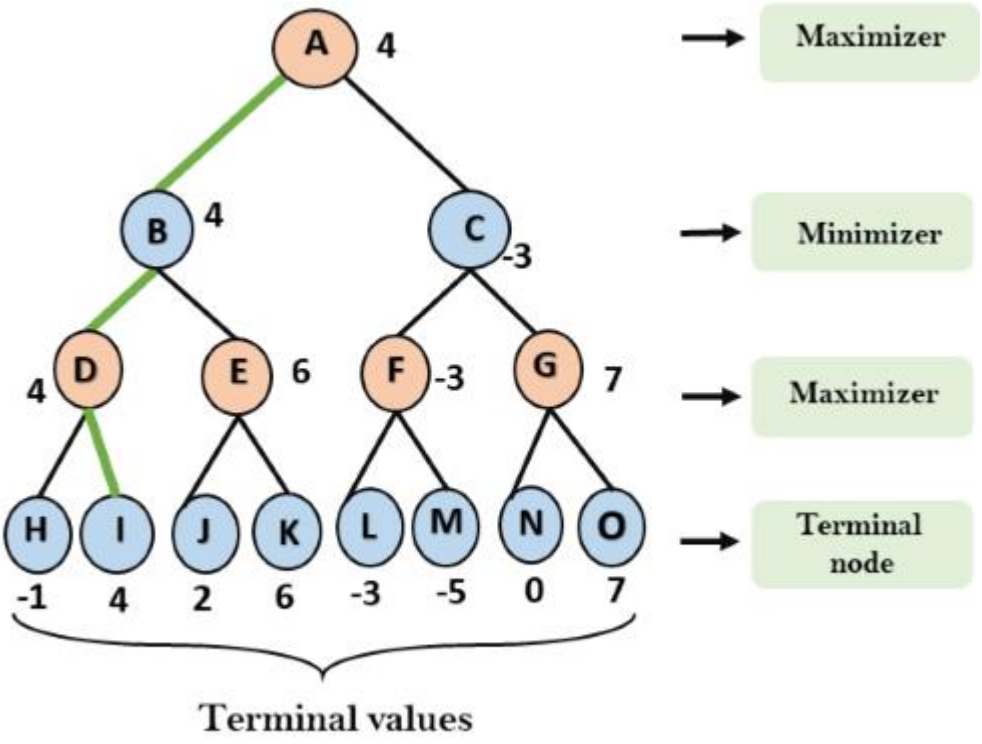
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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.

Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

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. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
    a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
    b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Pseudo-code for Alpha-beta Pruning:

```
1. function minimax(node, depth, alpha, beta, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
3. return static evaluation of node
4.
5. if MaximizingPlayer then      // for Maximizer Player
6.    maxEva= -infinity
```

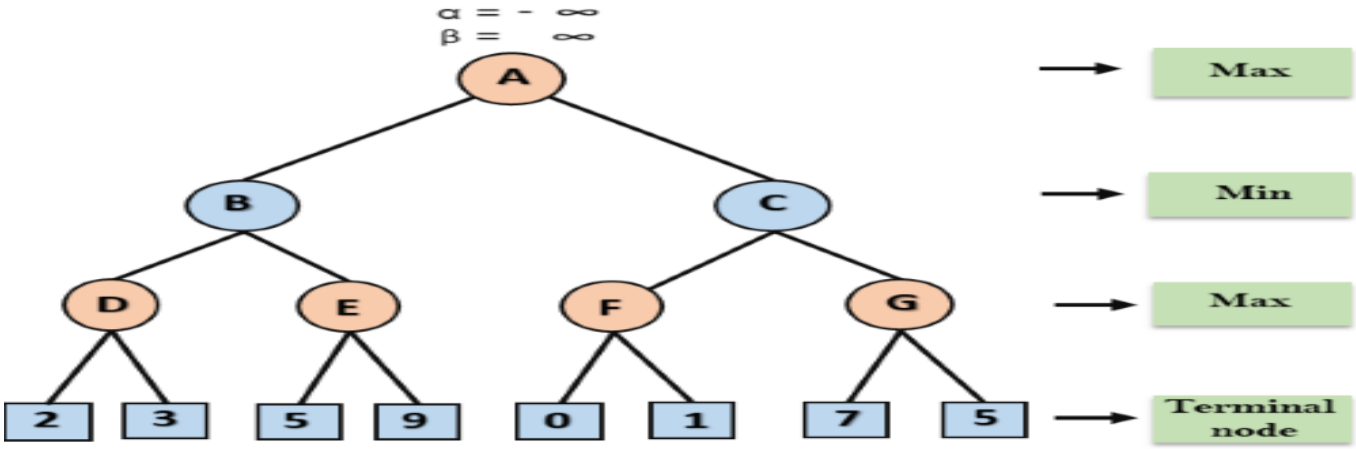
```
    for each child of node do
    eva= minimax(child, depth-1, alpha, beta, False)
   maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break
 return maxEva

else                    // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break
 return minEva
```

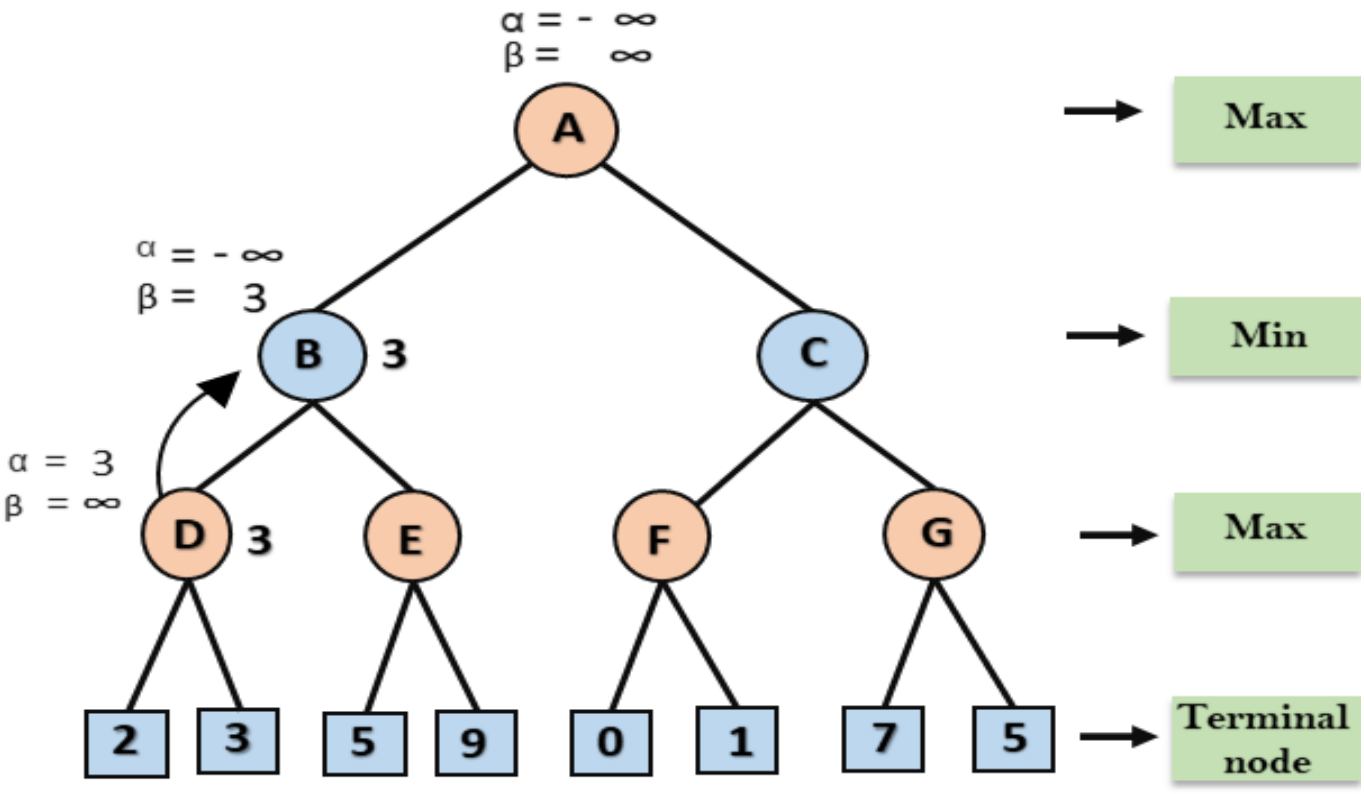
Working of Alpha-Beta Pruning:

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



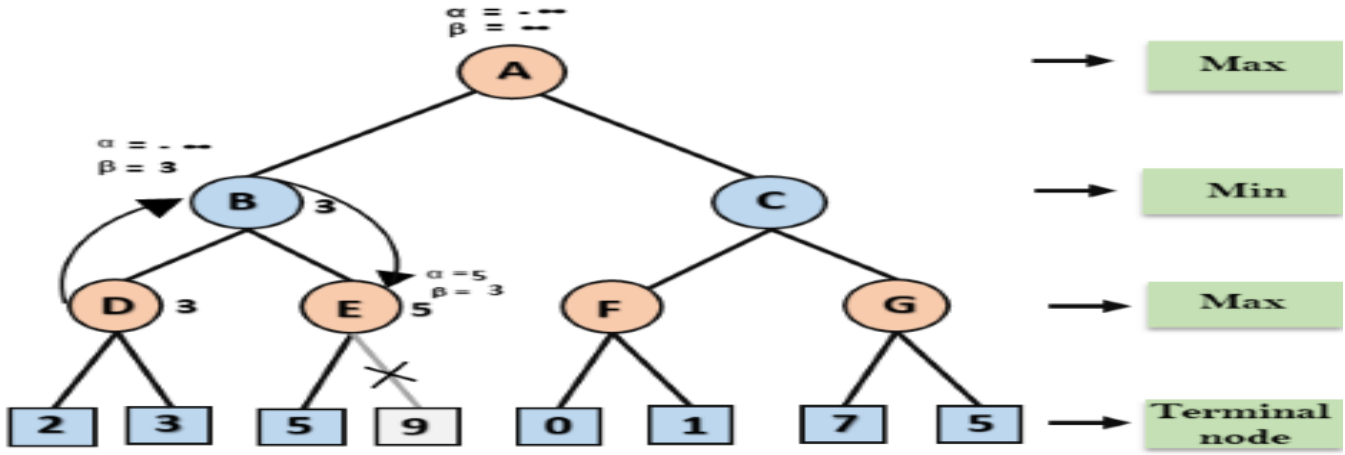
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.



Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

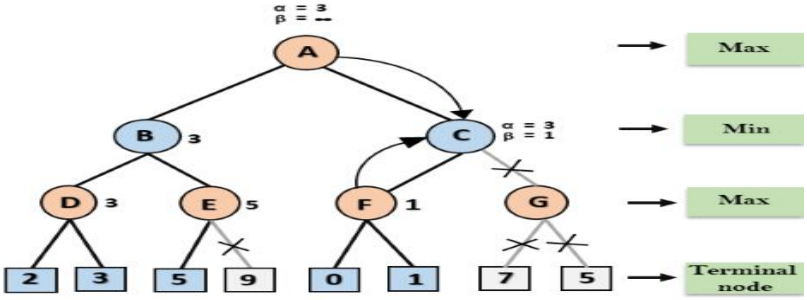
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.