Jaipur Engineering College & Research Centre, Jaipur
Department of Computer Science and Engineering

Lecture Notes
Artificial Intelligence [6CS4-05]
Unit 2

Vision of the Department:

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

Mission of the Department:

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.
M2: To provide opportunities for interaction between academia and industry.
M3: To provide platform for lifelong learning by accepting the change in technologies.
M4: To develop aptitude of fulfilling social responsibilities.

Program Outcomes (PO):

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and Computer Science & Engineering specialization to the solution of complex Computer Science & Engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex Computer Science and Engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex Computer Science and Engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety,and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of Computer Science and Engineering experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex Computer Science Engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional Computer Science and Engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional Computer Science and Engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the Computer Science and Engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in Computer Science and Engineering.
10. **Communication**: Communicate effectively on complex Computer Science and Engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the Computer Science and Engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest contextof technological change in Computer Science and Engineering.

Program Educational Objectives (PEO):

PEO1: To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenges.
PEO2: To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems in Computer Science and Engineering
PEO3: To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues for Computer Science & Engineering.
PEO4: To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career in Computer Science & Engineering.
PEO5: To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge in Computer Science & Engineering.

Course Outcomes (COs):

CO1: Understand the concept of Artificial Intelligence and apply various Searching techniques.
CO2: Illustrate various Game Playing in Artificial Intelligence system.
CO3: Analyze different Knowledge Representation Techniques, Neural Network, Planning, Uncertain Knowledge and Reasoning.
CO4: Apply basic concepts of Learning, Natural Language Processing, Robotics and Expert Systems in AI.

Syllabus:

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year-VI Semester: B.Tech. Computer Science and Engineering

6CS4-05: Artificial Intelligence

Credit: 2 **Max. Marks: 100(IA:20, ETE:80)**

2L+0T+0P **End Term Exam: 2 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	01
2	**Introduction to AI and Intelligent agent:** Different Approach of AI, Problem Solving : Solving Problems by Searching, Uninformed search, BFS, DFS, Iterative deepening, Bi directional search, Hill climbing, Informed search techniques: heuristic, Greedy search, A* search, AO* search, constraint satisfaction problems.	03
3	**Game Playing:** Minimax, alpha-beta pruning, jug problem, chess problem, tiles problem	06
4	**Knowledge and Reasoning:** Building a Knowledge Base: Propositional logic, first order logic, situation calculus. Theorem Proving in First Order Logic. Planning, partial order planning. Uncertain Knowledge and Reasoning, Probabilities, Bayesian Networks.	06
5	**Learning:** Overview of different forms of learning, Supervised base learning: Learning Decision Trees, SVM, Unsupervised based learning, Market Basket Analysis, Neural Networks.	07
6	**Introduction to Natural Language Processing:** Different issue involved in NLP, Expert System, Robotics.	05
	Total	28

Game Playing

Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called **adversarial searches**, often known as **Games**.

Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Mini-Max Algorithm

- 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 Min-Max 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
```

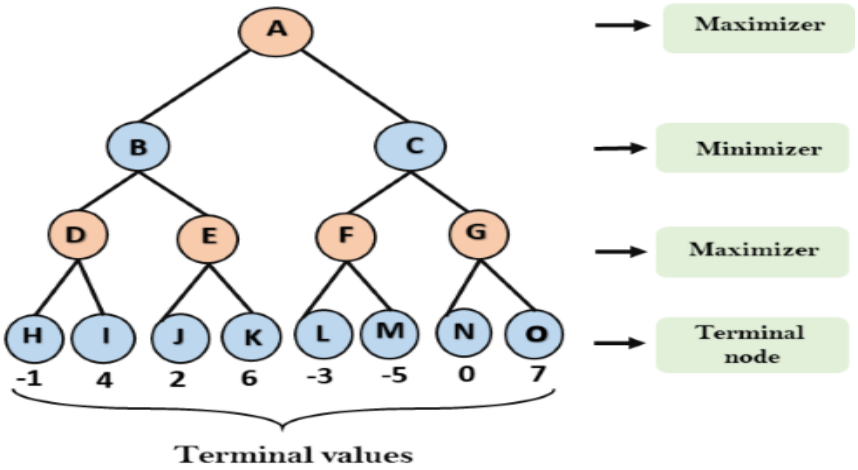
Initial call:

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:

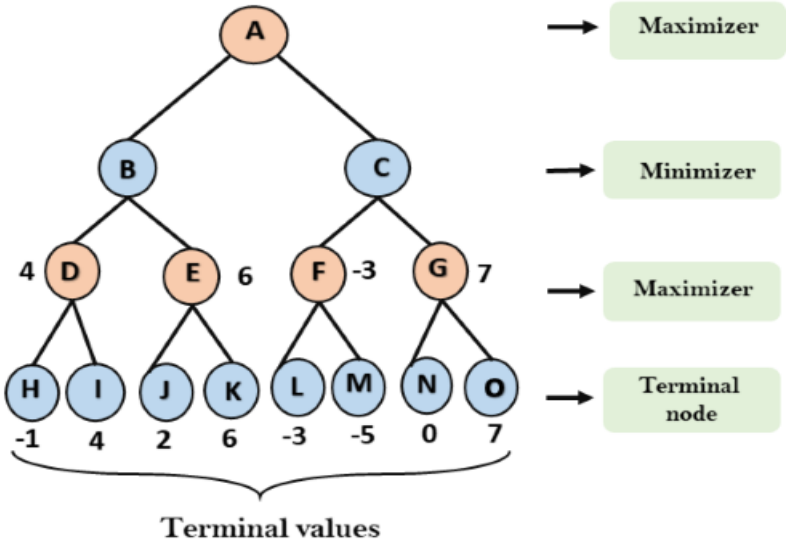
Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =-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

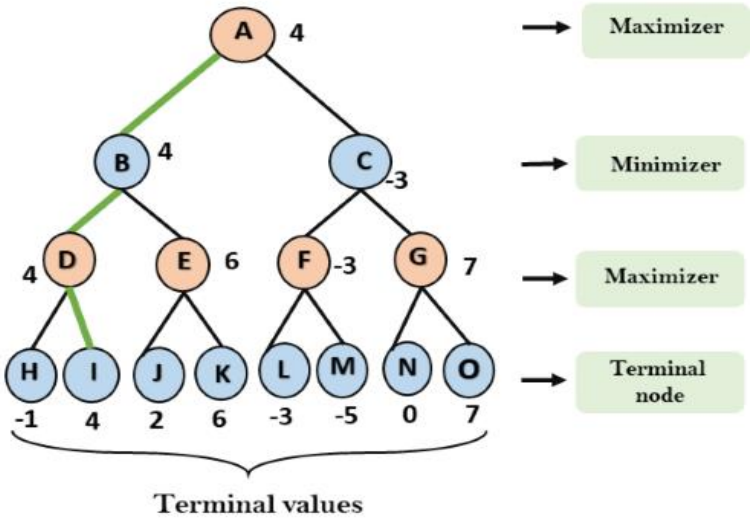
- For node D max(-1,- -∞) => max(-1,4)= 4

- 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



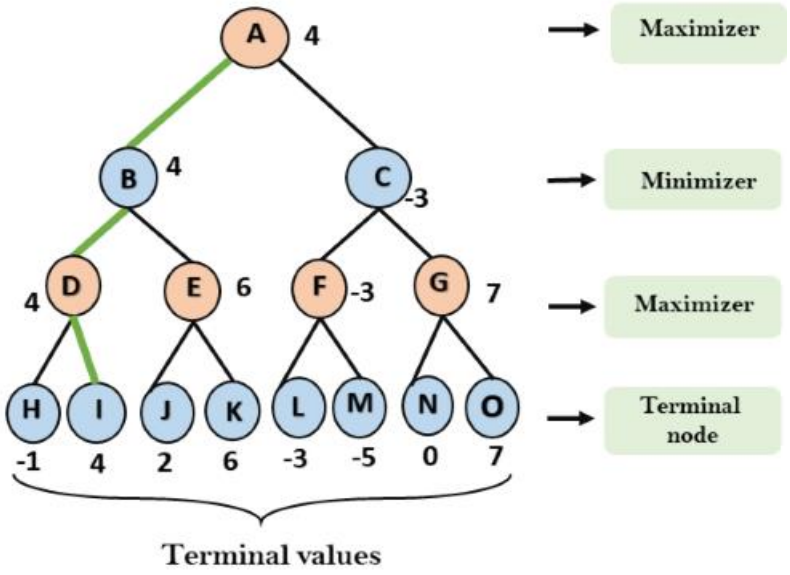
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Pseudo-code for Alpha-beta Pruning:

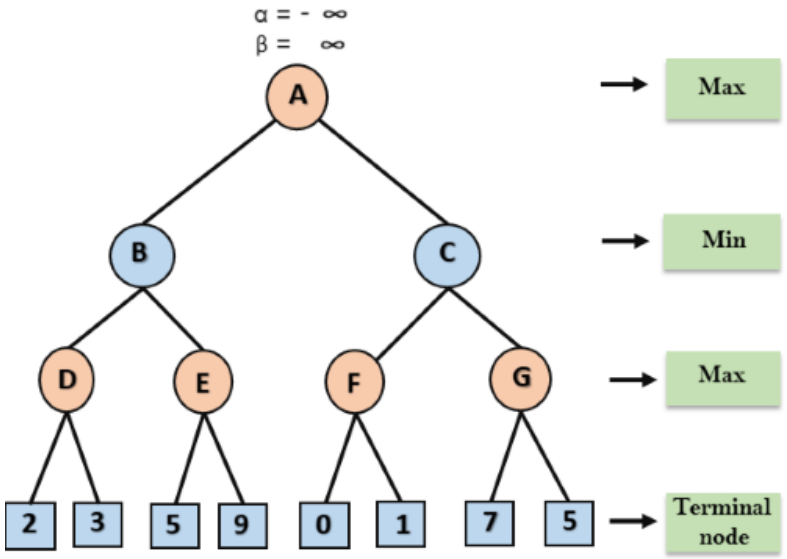
```
1. function minimax(node, depth, alpha, beta, maximizingPlayer) is
2. if depth ==0 or node is a terminal node then
```

```
3.  return static evaluation of node
4.
5.  if MaximizingPlayer then      // for Maximizer Player
6.     maxEva= -infinity
7.     for each child of node do
8.     eva= minimax(child, depth-1, alpha, beta, False)
9.    maxEva= max(maxEva, eva)
10.  alpha= max(alpha, maxEva)
11.   if beta<=alpha
12. break
13. return maxEva
14.
15.else                    // for Minimizer player
16.   minEva= +infinity
17.   for each child of node do
18.   eva= minimax(child, depth-1, alpha, beta, true)
19.   minEva= min(minEva, eva)
20.   beta= min(beta, eva)
21.    if beta<=alpha
22.  break
23. return minEva
```

Working of Alpha-Beta Pruning:

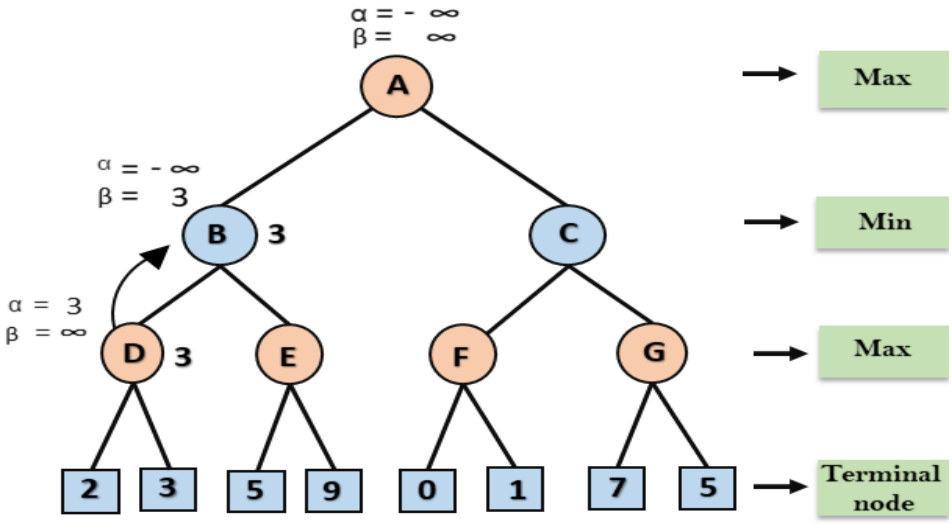
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



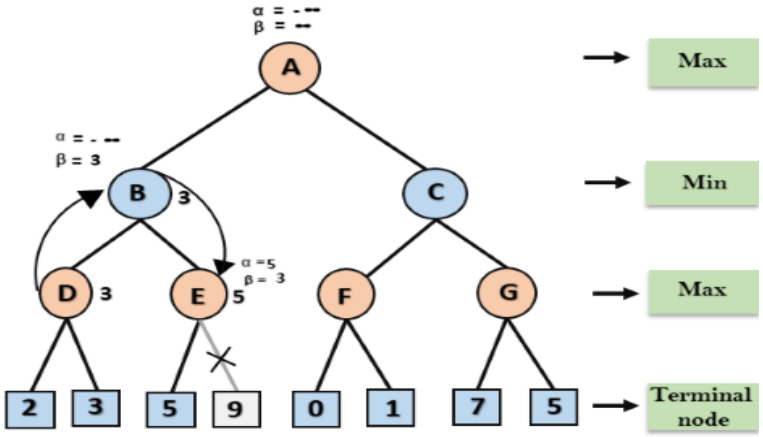
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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.



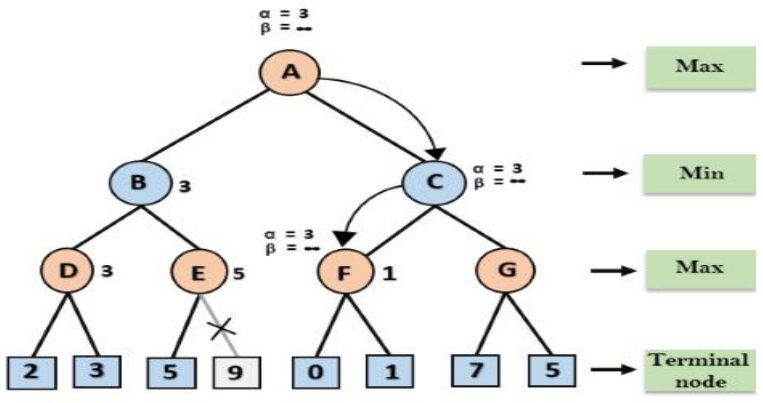
In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

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.

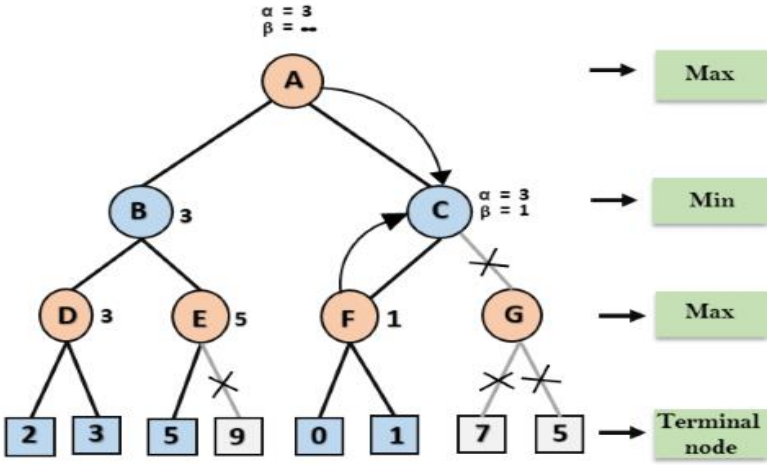


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C. At node C, α=3 and β= +∞, and the same values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

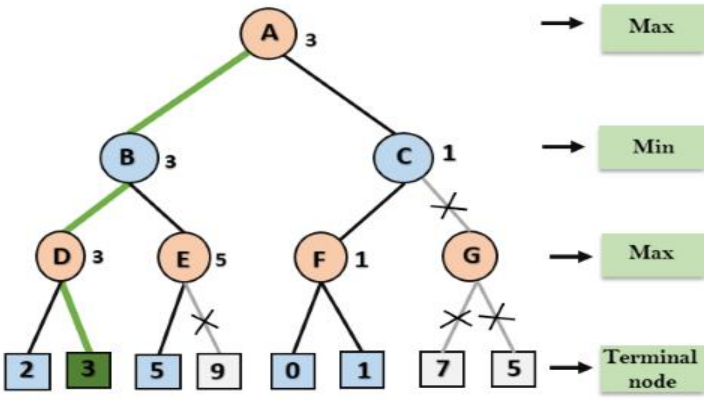
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Water Jug Problem

In the **water jug problem**, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

So, to solve this problem, following set of rules were proposed:

Production rules for solving the water jug problem

Here, let ***x*** denote the 4-gallon jug and ***y*** denote the 3-gallon jug.

S.No.	Initial State	Condition	Final state	Description of action taken
1.	(x,y)	If x<4	(4,y)	Fill the 4 gallon jug completely
2.	(x,y)	if y<3	(x,3)	Fill the 3 gallon jug completely

3.	(x,y)	If x>0	(x-d,y)	Pour some part from the 4 gallon jug
4.	(x,y)	If y>0	(x,y-d)	Pour some part from the 3 gallon jug
5.	(x,y)	If x>0	(0,y)	Empty the 4 gallon jug
6.	(x,y)	If y>0	(x,0)	Empty the 3 gallon jug
7.	(x,y)	If (x+y)<7	(4, y-[4-x])	Pour some water from the 3 gallon jug to fill the four gallon jug
8.	(x,y)	If (x+y)<7	(x-[3-y],y)	Pour some water from the 4 gallon jug to fill the 3 gallon jug.
9.	(x,y)	If (x+y)<4	(x+y,0)	Pour all water from 3 gallon jug to the 4 gallon jug
10.	(x,y)	if (x+y)<3	(0, x+y)	Pour all water from the 4 gallon jug to the 3 gallon jug

The listed production rules contain all the actions that could be performed by the agent in transferring the contents of jugs. But, to solve the water jug problem in a minimum number of moves, following set of rules in the given sequence should be performed:

Solution of water jug problem according to the production rules:

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

On reaching the 7^{th} attempt, we reach a state which is our goal state. Therefore, at this state, our problem is solved.

Tiles Problem 1

Black and White Sliding Tiles Problem:

There are 7 tiles, 3 are white and 3 are black and there is a white space in the middle. All the black tiles are placed left and all the white tiles are placed right. Each of the white or black tiles can move to the blank space and they can hope over one or two tiles to reach the blank space.

1. Analyze the state-space of the problem.
2. Propose a heuristic for this problem to move all the white tiles left to all the while tiles, the position of the blank space is not important.

Initial State:

TILES: **BBB WWW**

Goal State:

TILES: **WWW BBB**

Assumptions

- Heuristics for Black Tile = tile-distance to the first white tile to its right
- Heuristics for White Tile = tile-distance to the first black tile to it's left * -1
- We will run the script as long as we will have a non zero value for a tile.
- If any black tile does not contain any white tile to its left then that tile will have Zero Heuristics
- If any white tile does not contain any black tile to its right then that tile will have Zero Heuristics
- We will have a selector value initially set as White.
- For each iteration, we will change the value to black and white and so no.
- Depending on the selector we will choose that colored tile with the highest tile says, Tile X.
- X will be moved to the blank space.
- If the tile jumps to its next tile the cost:= 1 if hop one tile then cost:= 2 if jumps 2 tiles then cost:= 3.

Algorithm:

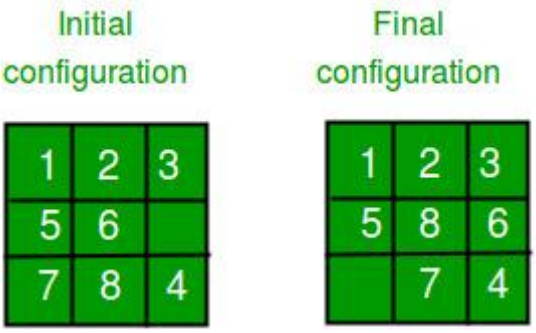
1. Selector:= white
2. Foreach black tile b
3. do h(b):= tile-distance to the first white tile to its right

4. Foreach white tile w
5. do h(w):= tile-distance to the first black tile to its left * -1
6. While any tile has a non-zero h value
7. Do
8. If selector:= white then,
9. Select the while-tile with the highest h value, say X
10. If the h(X):= 0 and it does not have the blank space to its left then,
11. Select another while-tile with the next-highest h value, say X
12. Selector := black
13. Else
14. Select the black-tile with the highest h value, say X
15. If the h(X):= 0 and it does not have the blank space to its left then,
16. Select another black-tile with the next-highest h value, say X
17. Selector:= white
18. If X and the blank space is in 2-tiles distance then,
19. Move X to the blank space and record the cost
20. Foreach black tile b
21. do h(b):= tile-distance to the first white tile to its right
22. Foreach white tile w
23. do h(w):= tile-distance to the first black tile to it's left * -1
24. End while

Tiles Problem 2

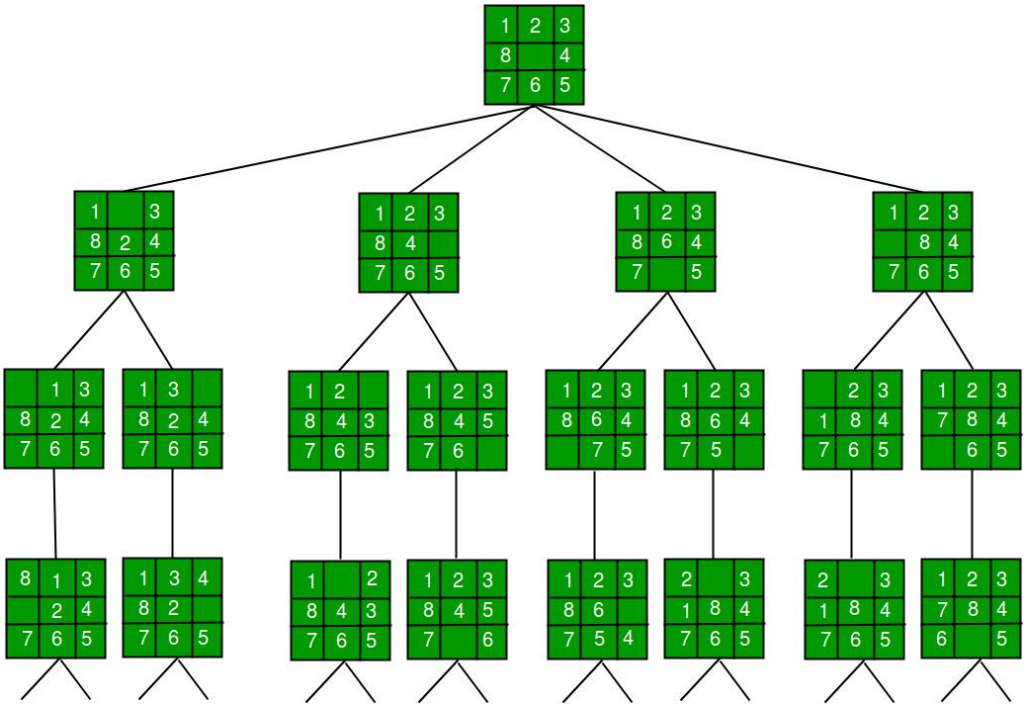
- 8 puzzle problem

- Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space. For example,



1. DFS (Brute-Force)

We can perform a depth-first search on state space (Set of all configurations of a given problem i.e. all states that can be reached from the initial state) tree.



State Space Tree for 8 Puzzle

In this solution, successive moves can take us away from the goal rather than bringing closer. The search of state space tree follows the leftmost path from the root regardless of the initial state. An answer node may never be found in this approach.

2. BFS (Brute-Force)

We can perform a Breadth-first search on the state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

3. Branch and Bound

The search for an answer node 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 answer node. It is similar to the backtracking technique but uses BFS-like search.
There are basically three types of nodes involved in Branch and Bound
1. Live node is a node that has been generated but whose children have not yet been generated.
2. E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.
Cost function:
Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where

g(X) = cost of reaching the current node from the root

h(X) = cost of reaching an answer node from X.

Ideal Cost function for 8-puzzle Algorithm:
We assume that moving one tile in any direction will have 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where

f(x) is the length of the path from root to x (the number of moves so far) and

h(x) is the number of non-blank tiles not in their goal position (the number of mis -placed tiles).

There are at least h(x) moves to transform state x to a goal state

Complete Algorithm:

```
/* Algorithm LCSearch uses c(x) to find an answer node
 * LCSearch uses Least() and Add() to maintain the list
   of live nodes
 * Least() finds a live node with least c(x), deletes
   it from the list and returns it
 * Add(x) adds x to the list of live nodes
 * Implement list of live nodes as a min-heap */
```

```
struct list_node
{
  list_node *next;

  // Helps in tracing path when answer is found
  list_node *parent;
  float cost;
}

algorithm LCSearch(list_node *t)
{
  // Search t for an answer node
  // Input: Root node of tree t
  // Output: Path from answer node to root
  if (*t is an answer node)
  {
     print(*t);
     return;
  }

  E = t; // E-node

  Initialize the list of live nodes to be empty;
  while (true)
  {
    for each child x of E
    {
       if x is an answer node
       {
         print the path from x to t;
         return;
       }
       Add (x); // Add x to list of live nodes;
       x->parent = E; // Pointer for path to root
    }

    if there are no more live nodes
    {
      print ("No answer node");
      return;
    }

    // Find a live node with least estimated cost
```

```
            E = Least();

            // The found node is deleted from the list of
            // live nodes
          }
        }
```

.