**Batch: C3 Roll No.: 121**

**Experiment / assignment / tutorial No. 6**

|  |
| --- |
| **Title:** Implementation of Alpha Beta Pruning. |

**Objective:** Implementation of Alpha-Beta Pruning algorithm

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **Course Outcome** | **After successful completion of the course students should be able to** |
| **CO2** | Analyse and solve problems for goal based agent architecture (searching and planning algorithms). |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **“Artificial Intelligence: a Modern Approach” by Russel and Norving, Pearson education Publications**
2. **“Artificial Intelligence” By Rich and knight, Tata Mcgraw Hill Publications**
3. [**www.cs.sfu.ca/CourseCentral/310/oschulte/mychapter5.pdf**](http://www.cs.sfu.ca/CourseCentral/310/oschulte/mychapter5.pdf)
4. [**http://cs.lmu.edu/~ray/notes/asearch/**](http://cs.lmu.edu/~ray/notes/asearch/)
5. **www.cs.cornell.edu/courses/cs4700/2011fa/.../06\_adversarialsearch.pdf**
6. **https://algoscale.com/blog/why-you-need-to-know-about-adversarial-search-in-ai/**
7. **https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/**
8. **https://en.wikipedia.org/wiki/Alpha%E2%80%93beta\_pruning**

**Pre Lab/ Prior Concepts:** Two/Multi player Games and rules, state-space tree, searching algorithms and their analysis properties

**Historical Profile: -** The game playing has been integral part of human life. The multiplayer games are competitive environment in which everyone tries to gain more points for himself and wishes the opponent to gain minimum.

The game can be represented in form of a state space tree and one can follow the path from root to some goal node, for either of the player.

**New Concepts to be learned:** Adversarial search, minmax algorithm, minmax pruning,

**Adversarial Search:**

Adversarial search is a type of problem-solving technique used in artificial intelligence, specifically in the domain of game theory and decision-making. It involves searching through the possible moves of two or more opposing players in a game or competitive environment to determine the best possible strategy for a given player.

The key components and concepts associated with adversarial search are as follows:

1. Game Tree: Adversarial search typically involves representing the game as a tree structure, where each node represents a possible game state, and edges represent possible moves that can be made by the players. The root of the tree represents the initial game state, and the branches represent subsequent moves.
2. Minimax Algorithm: The minimax algorithm is a fundamental technique used in adversarial search. It is based on the principle of minimizing the maximum possible loss (hence the name minimax). In a two-player zero-sum game (where one player's gain is the other player's loss), the goal is to maximize one's own gain while minimizing the opponent's gain. The minimax algorithm recursively evaluates possible moves by considering the opponent's best response to each move.
3. Evaluation Function: In many cases, it's not feasible to search the entire game tree to determine the best move. Therefore, an evaluation function is used to estimate the desirability of a particular game state. This function assigns a numerical value to each game state, representing how favorable it is for the player.
4. Alpha-Beta Pruning: Alpha-beta pruning is an optimization technique used to reduce the number of nodes evaluated in the minimax algorithm. It works by eliminating branches of the game tree that are guaranteed to be worse than previously examined branches, thus speeding up the search process.
5. Depth-Limited Search: In practical scenarios, it's often not feasible to search the entire game tree to its terminal nodes due to the exponential growth of the tree. Depth-limited search limits the depth of the search tree, allowing the algorithm to focus on a subset of possible moves.
6. Heuristic Functions: Heuristic functions are used to guide the search process by providing an estimate of the desirability of a particular game state. These functions are typically based on domain-specific knowledge and help to prioritize certain branches of the search tree over others.

Overall, adversarial search algorithms aim to find the optimal strategy for a player in a competitive environment by exploring the possible moves of both players and selecting the best course of action based on the anticipated moves of the opponent. These techniques are widely used in games such as chess, checkers, and Go, as well as in decision-making scenarios with multiple competing agents.

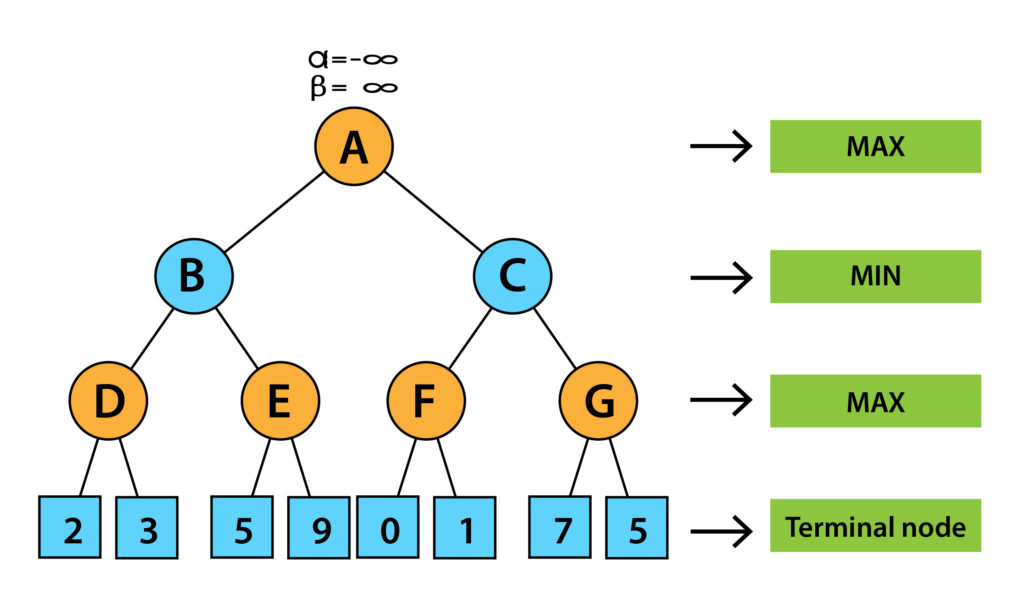
**Alpha-beta pruning algorithm:**

**Key points in Alpha-beta Pruning**

* Alpha:    Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is – ∞.
* Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.
* The condition for Alpha-beta Pruning is that α >= β.
* Each node has to keep track of its alpha and beta values. Alpha can be updated only when it’s MAX’s turn and, similarly, beta can be updated only when it’s MIN’s chance.
* MAX will update only alpha values and MIN player will update only beta values.
* The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
* Alpha and Beta values only be passed to child nodes.

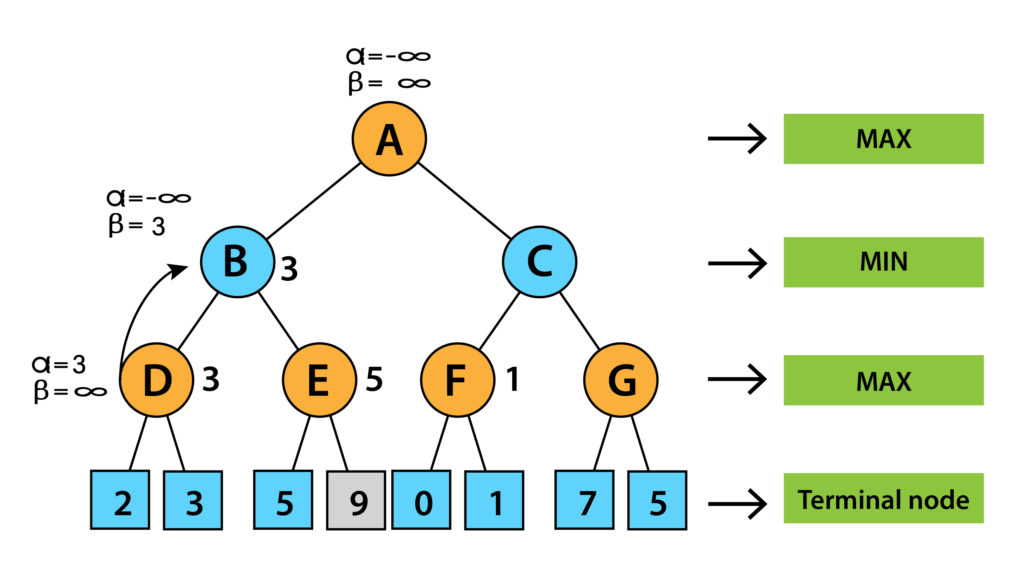
**Working of Alpha-beta Pruning**

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. α = -∞ and β= +∞. We will prune the node only when alpha becomes greater than or equal to beta.



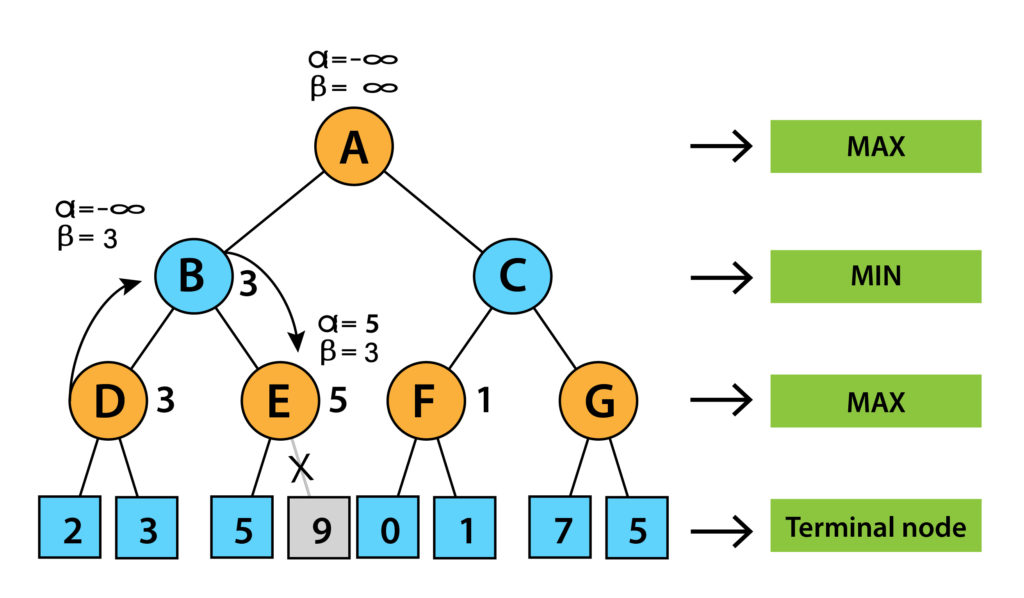
2. Since the initial value of alpha is less than beta so we didn’t prune it. Now it’s turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.



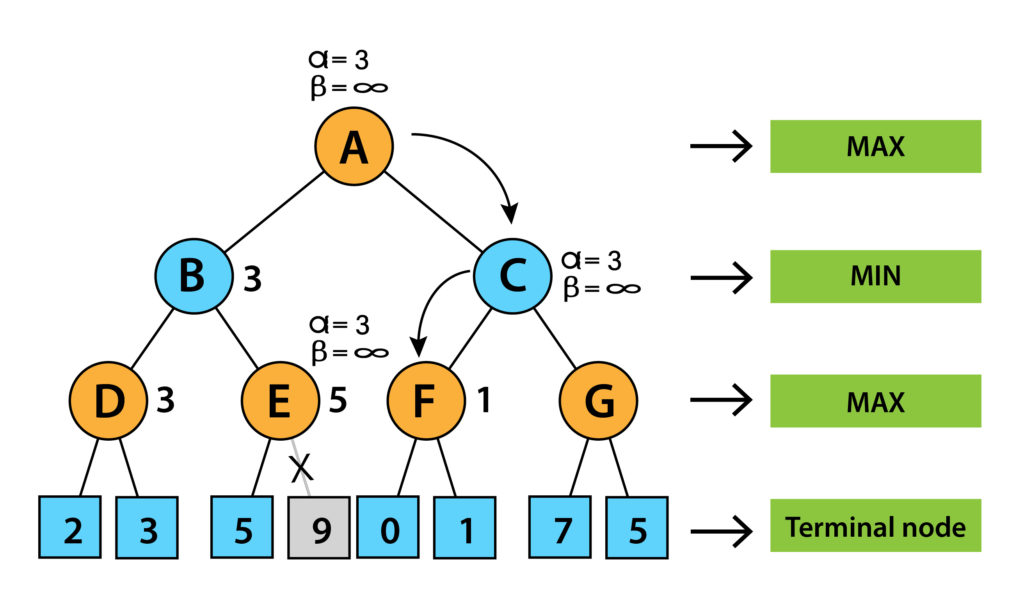
In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

4. Now it’s turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

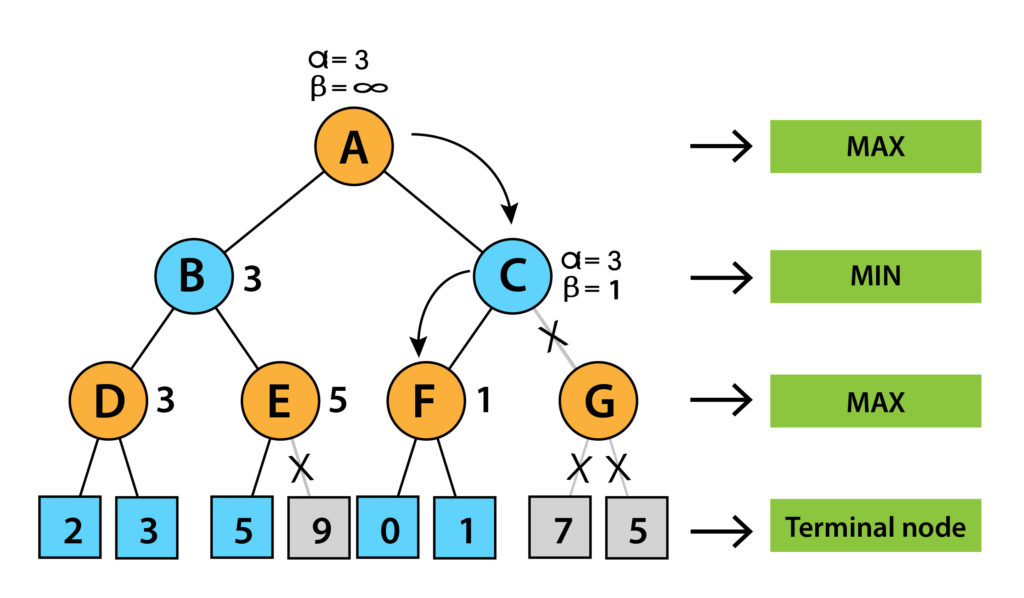


6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

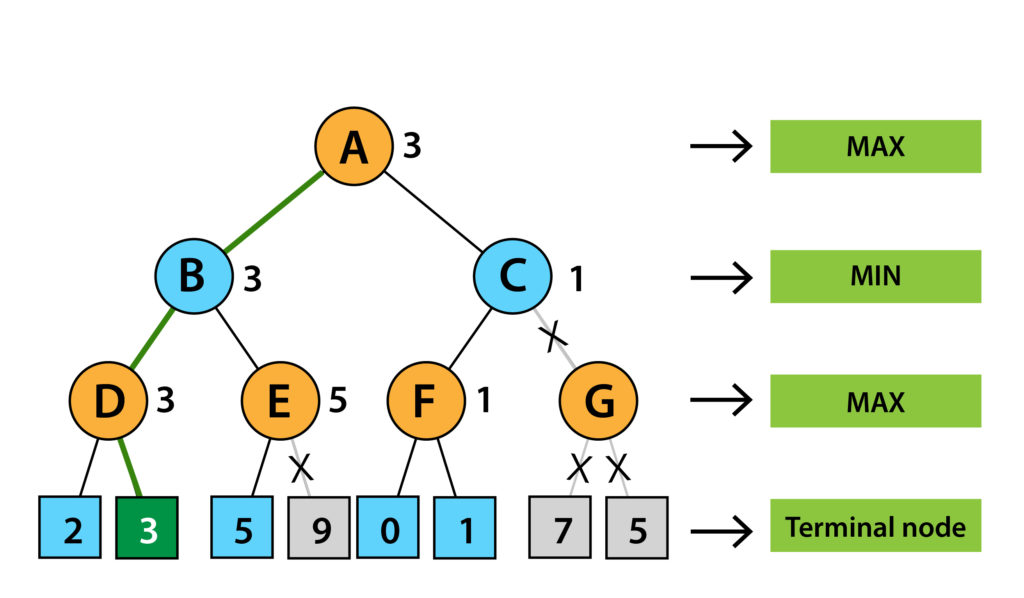
7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.



8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn’t compute the entire subtree G.



Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.



The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.

**Chosen Problem:**

Tic Tac Toe

**Solution tree for chosen Problem:**

**Code:**

board = [[0,0,0],[0,0,0],[0,0,0]]

# board = [[0,0],[0,0]]

# board = [[1,0,0],[0,1,0],[0,0,1]]

#aim can be 1 for X and -1 for O

#return value 1 for X and -1 for O

mylist = [[],[]]

def traverse(alpha,beta,aim,board,steps):

    for i in range(len(board)):

        for j in range(len(board[0])):

            if(board[i][j]==0):

                board[i][j] = aim

                if winner(board,aim): # is a node

                    value = aim

                    # print(board)

                    # print(f'{aim} wins')

                    # print(board,value,alpha,beta,True)

                    board[i][j]=0

                    steps.append((aim,(i,j)))

                    mylist[(1+aim)//2].append(steps)

                    return board,value,alpha,beta,True

                elif winner(board,aim\*-1): # is a node

                    value = aim \*-1

                    # print(board)

                    # print(f'{aim\*-1} wins')

                    # print(board,value,alpha,beta,True)

                    board[i][j]=0

                    steps.append((aim,(i,j)))

                    mylist[1+aim].append(steps)

                    return board,value,alpha,beta,True

                elif no\_winners(board):

                    #  print(board,0,alpha,beta,True)

                    #  print("no winners")

                     board[i][j]=0

                     return board,0,alpha,beta,True

                \_,value\_traversed,alpha\_traversed,beta\_traversed,is\_node = traverse(alpha,beta,aim\*-1,board,steps+[(aim,(i,j))])

                if(is\_node):

                        if(aim>0):

                              alpha\_traversed = value\_traversed

                        else:

                              beta\_traversed = value\_traversed

                if(aim>0):

                        if alpha\_traversed>alpha:

                            alpha = alpha\_traversed

                else:

                        if beta\_traversed<beta:

                            beta = beta\_traversed

                if(alpha>=beta):

                    #prune

                    value = aim

                    # print(board,value,alpha,beta,True)

                    board[i][j]=0

                    # print("prune")

                    return board,value,alpha,beta,True

                board[i][j]=0 #backtrack

                # mylist.append((i,j,value\_traversed))

    # print(board)

    # print("Backtrack without prune")

    value = aim

    return board,value,alpha,beta,True

def winner(board,aim):

    win = 0

    col = -2

    row = -2

    n = len(board)

    for i in range(n):

        if all(board[i][j] == board[i][0] and board[i][0] != 0 for j in range(n)):

            win = board[i][0]

            row = i

    for j in range(n):

        if all(board[i][j] == board[0][j] and board[0][j] != 0 for i in range(n)):

            win = board[0][j]

            col = j

    if all(board[i][i] == board[0][0] and board[i][i] != 0 for i in range(n)):

        win = board[0][0]

    if all(board[i][n - i - 1] == board[0][n - 1] and board[i][n - i - 1] != 0 for i in range(n)):

        win = board[0][n - 1]

    return win == aim

def no\_winners(board):

    prod = 1

    for i in board:

        for j in i:

            prod=prod\*j

    if(prod!=0):

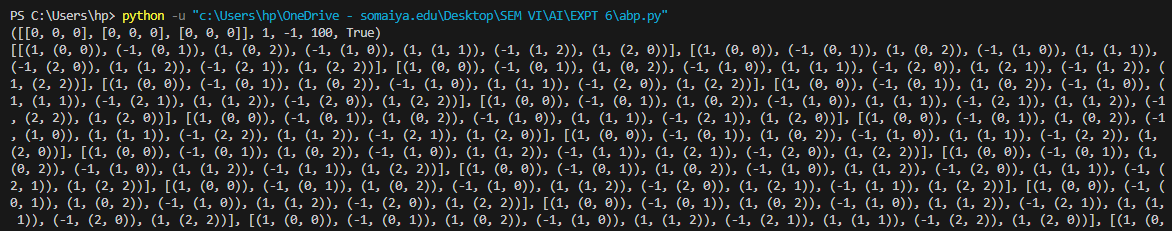
        return True

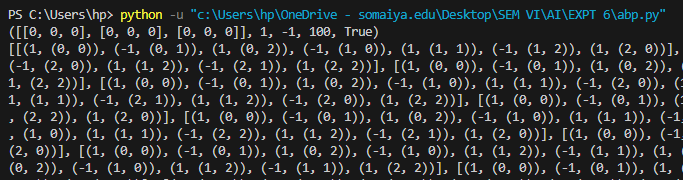
    return False

print(traverse(-100,100,1,board,[]))

print(mylist[1])

**Output:**

****

****

**Explanation of Output:**

This code essentially generates a list of sequences which lead to a win for player 'X'. The code has been run for a 3-by-3 tic-tac-toe game. The output is one single superlist, which is a list of various scenarios. Let us look at the first one. The first scenario is the first list. This is a list of tuples, where each tuple consists of the current player ('1' for 'X' and '-1' for 'O') and a tuple of x-position and y-position. So initially 'X' selects position (0,0).

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | 2 |
| 0 | X |  |  |
| 1 |  |  |  |
| 2 |  |  |  |

Next, 'O' selects position (0,1).

|  |  |  |
| --- | --- | --- |
| X | O |  |
|  |  |  |
|  |  |  |

Next, 'X' selects position (0,2).

|  |  |  |
| --- | --- | --- |
| X | O | X |
|  |  |  |
|  |  |  |

Next, 'O' selects position (1,0).

|  |  |  |
| --- | --- | --- |
| X | O | X |
| O |  |  |
|  |  |  |

Next, 'X' selects position (1,1).

|  |  |  |
| --- | --- | --- |
| X | O | X |
| O | X |  |
|  |  |  |

Next, 'O' selects position (1,2).

|  |  |  |
| --- | --- | --- |
| X | O | X |
| O | X | O |
|  |  |  |

Next, 'X' selects position (0,2) and wins the game.

|  |  |  |
| --- | --- | --- |
| X | O | X |
| O | X | O |
| X |  |  |

So this program works in a way that prints out the steps so that 'X' wins.

**Post Lab objective Questions:**

1. **Which search is equal to minmax search but eliminates the branches that can’t influence the final decision?**
   1. Breadth-first search
   2. Depth first search
   3. Alpha-beta pruning
   4. None of the above

**Answer: c. Alpha-beta pruning**

1. **Which values are independent in minmax search alogirthm?**
   1. Pruned leaves x and y
   2. Every states are dependant
   3. Root is independent
   4. None of the above

**Answer: c. Root is independent**

**Post Lab Subjective Questions:**

* + - 1. **Explain the concept of adversarial search**

**Ans.** In artificial intelligence, deep learning, machine learning, and computer vision, adversarial search is basically a kind of search in which one can trace the movement of an enemy or opponent. The step which arises the problem for the user or the user or the agent doesn’t want that specific step task to be carried out. Such searches are important in chess, business strategy tools, trading kind of websites, and war-based games where there are AI agents. In an adversarial search, the user can change the current state but has no control over the next stage. The next state control is in the hands of the opponent and is unpredictable.

**Adversarial Search in AI**

While talking about such searches in AI, adversarial search is one of the most important kinds of search. It is very prominent in gaming techniques. The use of the adversarial technique can be found in different games as in games the AI agent has been surrounded by a kind of competitive environment. The goal has been defined initially by the user and the agents compete or fight with one another in order to achieve that goal so that the win can be achieved. The adversarial search is important, and each agent must have known the strategy of the next agent this will create a competitive environment in a game.

**Important Features of Adversarial Search**

Adversarial searches have some features which make them unique as compared to the conventional techniques used in searches. There are certain features that can be observed in adversarial searches and with the help of this one can determine how such features make things more interesting and important in our case.

* The game in which adversarial searches have been used must have been a two-player game.
* The two-player game must have been in such a way that the game should have been played in the form of turn-taking. As we have observed in chess, ludo, Poker, etc.
* The information provided should have been perfect otherwise it becomes impossible for the model or the system to determine a strategy and to give a decision based on the results.
* The rules must have been precise. Formal should have been used in order to get better results.
* The actions must have been in a smaller number. This can increase the accuracy with better optimization.

While keeping the above factors in mind can help in developing the game more interesting. In this way, there are certain different advantages as well while adding adversarial searches in a two-player kind of game. These benefits are as follows.

* The game becomes more competitive and becomes hard to solve.
* Some games come under the luck of chance like in the dice games these games become very interesting as every time rolling some dice gives a different number and each number corresponds to a different and exciting move.
* Using this search technique can make the games so fast that with the help of which games become more interesting and competitive.
  + - 1. **Explain how alpha-beta pruning improves memory efficiency of algorithm**

**Ans.** The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node).

With an (average or constant) branching factor of *b*, and a search depth of *d* plies, the maximum number of leaf node positions evaluated (when the move ordering is pessimal) is *O*(*b*×*b*×...×*b*) = *O*(*bd*) – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about *O*(*b*×1×*b*×1×...×*b*) for odd depth and *O*(*b*×1×*b*×1×...×1) for even depth, or . In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation. The explanation of *b*×1×*b*×1×... is that all the first player's moves must be studied to find the best one, but for each, only the second player's best move is needed to refute all but the first (and best) first player move—alpha–beta ensures no other second player moves need be considered. When nodes are considered in a random order (i.e., the algorithm randomizes), asymptotically, the expected number of nodes evaluated in uniform trees with binary leaf-values is ΘFor the same trees, when the values are assigned to the leaf values independently of each other and say zero and one are both equally probable, the expected number of nodes evaluated is Θ((b/2)d), which is much smaller than the work done by the randomized algorithm, mentioned above, and is again optimal for such random trees. When the leaf values are chosen independently of each other but from the [0,1] interval uniformly at random, the expected number of nodes evaluated increases to Θ(bd/log(d)) in the d→∞ limit, which is again optimal for these kind random trees. Note that the actual work for "small" values of d is better approximated using 0.925d0.747.

A chess program that searches four plies with an average of 36 branches per node evaluates more than one million terminal nodes. An optimal alpha-beta prune would eliminate all but about 2,000 terminal nodes, a reduction of 99.8%.

Normally during alpha–beta, the subtrees are temporarily dominated by either a first player advantage (when many first player moves are good, and at each search depth the first move checked by the first player is adequate, but all second player responses are required to try to find a refutation), or vice versa. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them. In practice, the move ordering is often determined by the results of earlier, smaller searches, such as through iterative deepening.

Additionally, this algorithm can be trivially modified to return an entire principal variation in addition to the score. Some more aggressive algorithms such as MTD(f) do not easily permit such a modification.

* + - 1. **Explain how a game of chess may benefit from min-max and alpha-beta pruning algorithms.**

**Ans.** In the context of chess, the Minimax algorithm and its enhancement, the Alpha-Beta Pruning algorithm, are fundamental techniques used in designing intelligent agents or engines that play the game.

1. Minimax Algorithm:
   * The Minimax algorithm is a decision rule used in decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst-case scenario.
   * In chess, each player aims to maximize their chances of winning while minimizing the opponent's chances. The Minimax algorithm simulates this process by considering all possible moves and their consequences up to a certain depth in the game tree.
   * It works by recursively evaluating positions in the game tree, where each node represents a possible move. The algorithm alternates between maximizing the score for the current player and minimizing the score for the opponent, assuming that the opponent will also play optimally.
   * At the end of the search, the algorithm selects the move that leads to the best outcome, according to the evaluation function.
2. Alpha-Beta Pruning:
   * Alpha-Beta Pruning is an optimization technique used to reduce the number of nodes evaluated by the Minimax algorithm.
   * It works by keeping track of two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.
   * During the search, if it is found that a move leads to a position worse than what is already known, it can be pruned or discarded, as it will not be chosen anyway. This reduces the number of nodes that need to be evaluated.
   * By pruning unnecessary branches of the game tree, Alpha-Beta Pruning significantly improves the efficiency of the Minimax algorithm, allowing it to search deeper into the game tree within the same amount of time.

**Benefits in Chess:**

Applying these algorithms to chess has several benefits:

* Efficiency: By using the minimax algorithm with alpha-beta pruning, the AI can decide the best move more efficiently as it avoids exploring irrelevant nodes in the game tree.
* Depth: The algorithms allow the AI to calculate more moves ahead within the same time limit. This can lead to a significant improvement in play, especially in complex positions where long-term planning is required.
* Optimality: If the game tree is explored to a sufficient depth (which might be the case in endgames with few pieces left), the AI can play optimally and will not lose unless there are forced mate sequences.