K. J. Somaiya College of Engineering, Mumbai-77

Batch: C - 1	**Roll No.: 16010122323**
Experiment 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**
4. **http://cs.lmu.edu/~ray/notes/asearch/**
5. **www.cs.cornell.edu/courses/cs4700/2011fa/.../06_adversarialsearch.pdf**

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 fundamental technique in artificial intelligence used for determining the best move or strategy in a two-player game where both players have opposing goals. It involves exploring the game tree to ascertain the optimal move for one player while considering the potential counter-moves of the opponent. In adversarial search, each possible move is evaluated based on a heuristic evaluation function, which estimates the utility of the move for the player. The algorithm selects the move that leads to the most favorable outcome for the player, assuming the opponent will also make their best moves. The minimax algorithm is one of the most commonly used techniques in adversarial search. It operates by recursively traversing the game tree, evaluating each node by alternating between maximizing the player's score and minimizing the opponent's score. This process continues until a terminal state, where the game ends, is reached. Another crucial algorithm in adversarial search is alpha-beta pruning. It is an enhancement of the minimax algorithm designed to reduce the number of nodes explored in the game tree. Alpha-beta pruning eliminates irrelevant nodes by maintaining bounds on the possible scores that can be achieved. This pruning process significantly reduces the computational effort required to find the optimal move.

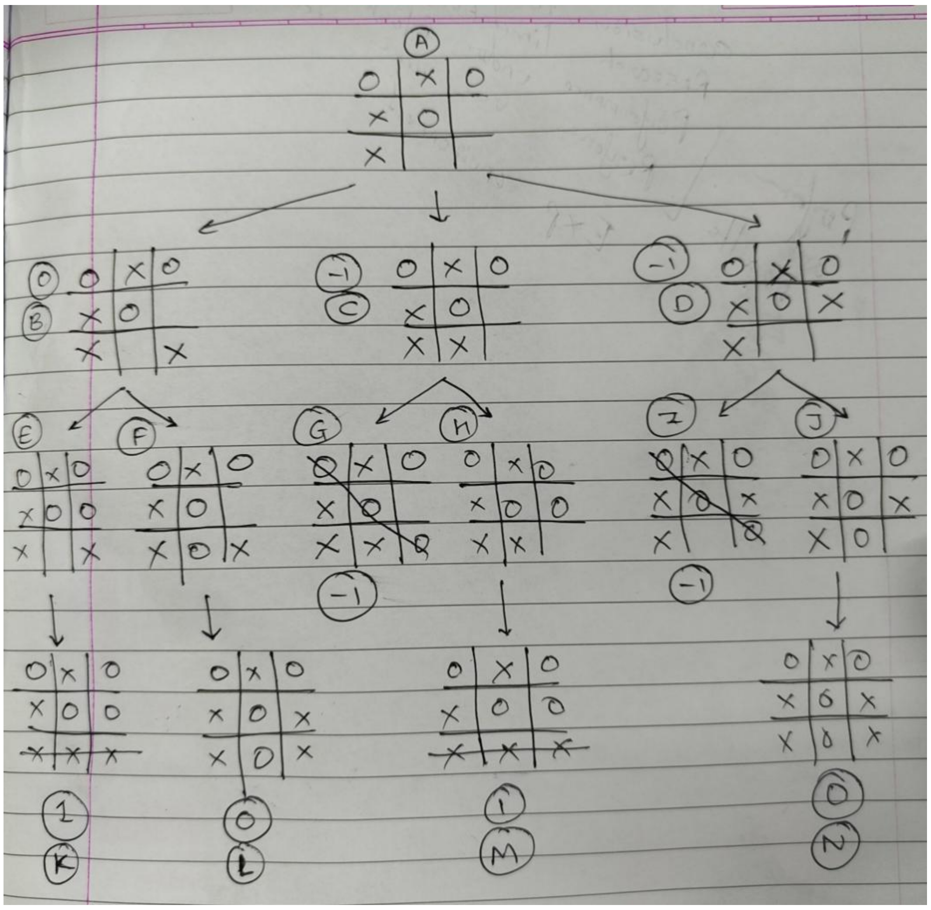
Alpha-beta pruning algorithm:

Alpha-beta pruning is an optimization technique used in adversarial search algorithms, such as minimax, to reduce the number of nodes that need to be evaluated in the game tree. The basic idea of alpha-beta pruning is to eliminate nodes in the game tree that will never be reached because there is no need to evaluate them. The alpha-beta pruning algorithm maintains two values, alpha and beta, which represent the best values found so far for the maximizing player and the minimizing player, respectively. Initially, alpha is set to negative infinity, and beta is set to positive infinity. As the search progresses, these values are updated based on the best possible scores found so far. When the algorithm explores a node in the game tree, it evaluates the node and updates the alpha and beta values accordingly. If the alpha value becomes greater than or equal to the beta value, then the algorithm can prune the subtree rooted at that node, as it will never be reached in the game because the opponent would never make such a move. By eliminating these irrelevant nodes, the alpha-beta pruning algorithm reduces the number of nodes that need to be evaluated, resulting in faster search times and more efficient game- playing strategies.

Chosen Problem:

Alpha-beta pruning is a technique used in game tree search algorithms to improve the efficiency of the search. It is particularly useful in games like Tic-Tac-Toe where the game tree is relatively small and can be fully searched. In Tic-Tac-Toe, the game tree can be represented as a tree of all possible moves from the initial state. Each node in the tree represents a game state, and the edges represent possible moves. The goal of the algorithm is to find the best move from the current state, which is the move that leads to the highest score for the current player. To perform alpha-beta pruning in Tic-Tac-Toe, the algorithm starts at the root node and evaluates all possible moves. It then recursively evaluates the resulting states, keeping track of alpha and beta values along the way. If the current player is trying to maximize their score, the algorithm updates alpha with the maximum score seen so far, and if the opponent is trying to minimize the score, it updates beta with the minimum score seen so far. If the algorithm encounters a node where the beta value is less than or equal to the alpha value, it knows that the opponent can force a score that is worse than the current alpha value, so it can prune the rest of the subtree rooted at that node. This reduces the number of nodes that need to be evaluated, which improves the efficiency of the algorithm.

Solution tree for chosen Problem:

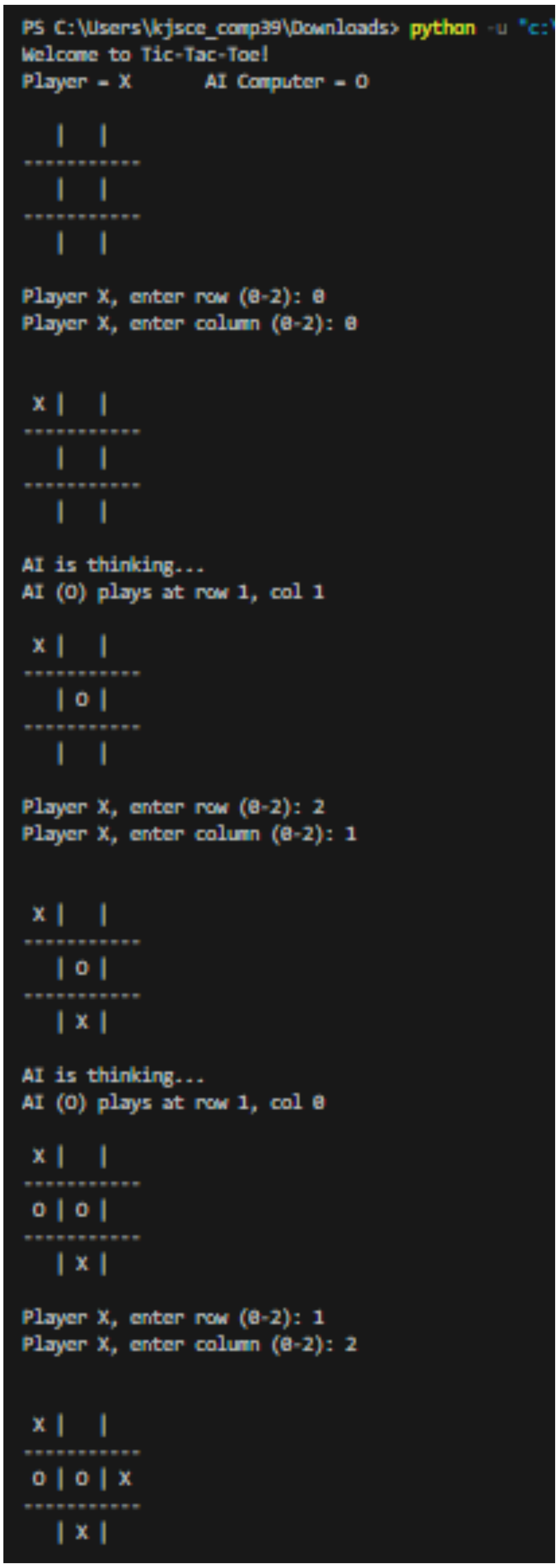


Implementation:

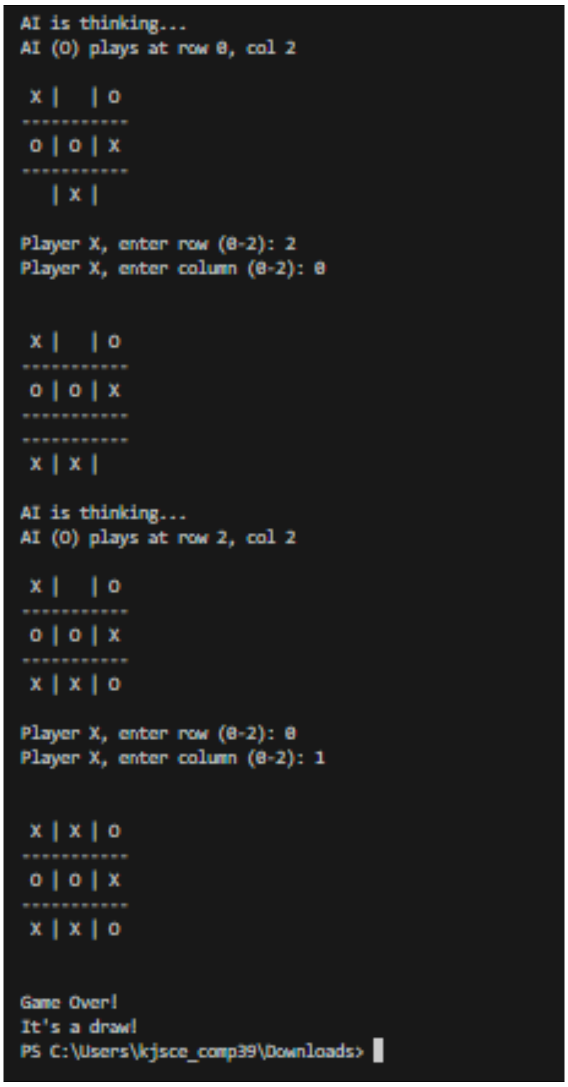
```
import math
from typing import List, Tuple, Optional, Set

EMPTY_SPACE = ' '
AI_MARKER = 'O'
PLAYER_MARKER = 'X'
WIN = 1
DRAW = 0
LOSS = -1
START_DEPTH = 0

Board = List[List[str]]
Move = Tuple[int, int]


WINNING_STATES: List[List[Tuple[int, int]]] = [
    [(0, 0), (0, 1), (0, 2)],
    [(1, 0), (1, 1), (1, 2)],
    [(2, 0), (2, 1), (2, 2)],
```

```python
    [(0, 0), (1, 0), (2, 0)],
    [(0, 1), (1, 1), (2, 1)],
    [(0, 2), (1, 2), (2, 2)],
    [(0, 0), (1, 1), (2, 2)],
    [(2, 0), (1, 1), (0, 2)],
]


def print_game_state(state: int):
    if state == WIN:
        print("AI wins!")
    elif state == DRAW:
        print("It's a draw!")
    elif state == LOSS:
        print("Player wins!")

def print_board(board: Board):
    print()
    for i, row in enumerate(board):
        print(f" {row[0]} | {row[1]} | {row[2]} ")
        if i < 2:
            print("-----------")
    print()

def get_legal_moves(board: Board) -> List[Move]:
    legal_moves = []
    for r in range(3):
        for c in range(3):
            if board[r][c] == EMPTY_SPACE:
                legal_moves.append((r, c))
    return legal_moves

def position_occupied(board: Board, pos: Move) -> bool:
    return board[pos[0]][pos[1]] != EMPTY_SPACE

def get_occupied_positions(board: Board, marker: str) ->
List[Move]:
    occupied = []
    for r in range(3):
        for c in range(3):
            if board[r][c] == marker:
                occupied.append((r, c))
    return occupied

def board_is_full(board: Board) -> bool:
    for row in board:
        if EMPTY_SPACE in row:
            return False
    return True

def game_is_won(occupied_positions: List[Move]) -> bool:
    occupied_set: Set[Move] = set(occupied_positions)
    for win_state in WINNING_STATES:
        if all(pos in occupied_set for pos in win_state):
            return True
    return False

def get_opponent_marker(marker: str) -> str:
    return PLAYER_MARKER if marker == AI_MARKER else AI_MARKER

def get_board_state(board: Board, ai_marker: str = AI_MARKER) ->
int:
    if game_is_won(get_occupied_positions(board, ai_marker)):
        return WIN

    player_marker = get_opponent_marker(ai_marker)
    if game_is_won(get_occupied_positions(board, player_marker)):
        return LOSS

    if board_is_full(board):
        return DRAW


    return DRAW


def game_is_done(board: Board) -> bool:
    return get_board_state(board) != DRAW or board_is_full(board)


def minimax_optimization(board: Board, marker: str, depth: int,
alpha: float, beta: float) -> Tuple[float, Optional[Move]]:
    best_move: Optional[Move] = None
    current_state = get_board_state(board, AI_MARKER)

    if current_state != DRAW or board_is_full(board):
        return float(current_state), best_move

    legal_moves = get_legal_moves(board)

    if marker == AI_MARKER:
        best_score = -math.inf
        for curr_move in legal_moves:
            board[curr_move[0]][curr_move[1]] = marker
            score, _ = minimax_optimization(
                board, get_opponent_marker(marker), depth + 1,
alpha, beta
            )
            board[curr_move[0]][curr_move[1]] = EMPTY_SPACE # Undo
move

            adjusted_score = score - (depth * 0.01)

            if adjusted_score > best_score:
                best_score = adjusted_score
                best_move = curr_move
            alpha = max(alpha, best_score)
            if beta <= alpha:
                break
        return best_score, best_move

    else:
        best_score = math.inf
        for curr_move in legal_moves:
            board[curr_move[0]][curr_move[1]] = marker
            score, _ = minimax_optimization(
                board, get_opponent_marker(marker), depth + 1,
alpha, beta
            )
            board[curr_move[0]][curr_move[1]] = EMPTY_SPACE # Undo
move

            adjusted_score = score + (depth * 0.01)

            if adjusted_score < best_score:
                best_score = adjusted_score
                best_move = curr_move
            beta = min(beta, best_score)
            if beta <= alpha:
                break # Prune
        return best_score, best_move


def main():
    board: Board = [[EMPTY_SPACE for _ in range(3)] for _ in
range(3)]

    print("Welcome to Tic-Tac-Toe!")
    print(f"Player = {PLAYER_MARKER}\t AI Computer = {AI_MARKER}")
    print_board(board)

    current_marker = PLAYER_MARKER

    while not game_is_done(board):
        if current_marker == PLAYER_MARKER:
            valid_input = False
            while not valid_input:
                try:
                    row_str = input(f"Player {PLAYER_MARKER}, enter row (0-2): ")
                    row = int(row_str)
                    col_str = input(f"Player {PLAYER_MARKER}, enter column (0-2): ")
                    col = int(col_str)
                    print()

                    move = (row, col)

                    if not (0 <= row <= 2 and 0 <= col <= 2):
                        print("Invalid input. Row and column must be between 0 and 2.")
                    elif position_occupied(board, move):
                        print(f"The position ({row}, {col}) is occupied. Try another one...")
                    else:
                        board[row][col] = PLAYER_MARKER
                        valid_input = True

                except ValueError:
                    print("Invalid input. Please enter numbers only.")
                except Exception as e:
                    print(f"An unexpected error occurred: {e}")
        else:
            print("AI is thinking...")
            _, best_move = minimax_optimization(
                board, AI_MARKER, START_DEPTH, -math.inf, math.inf
            )

            if best_move:
                board[best_move[0]][best_move[1]] = AI_MARKER
                print(f"AI ({AI_MARKER}) plays at row {best_move[0]}, col {best_move[1]}")
            else:
                print("AI could not find a move.")

                legal = get_legal_moves(board)
                if legal:
                    fallback_move = legal[0]
                    board[fallback_move[0]][fallback_move[1]] =
AI_MARKER
                    print(f"AI ({AI_MARKER}) plays fallback at row
{fallback_move[0]}, col {fallback_move[1]}")


        print_board(board)
        if not game_is_done(board):
            current_marker = get_opponent_marker(current_marker)


    print("\nGame Over!")
    final_state = get_board_state(board, AI_MARKER)
    print_game_state(final_state)


if __name__ == "__main__":
    main()
```

Output:



```
PS C:\Users\kjsce_comp39\Downloads> python -u "c:
Welcome to Tic-Tac-Toe!
Player - X       AI Computer - O

   |   |
-----------
   |   |
-----------
   |   |

Player X, enter row (0-2): 0
Player X, enter column (0-2): 0


 X |   |
-----------
   |   |
-----------
   |   |

AI is thinking...
AI (O) plays at row 1, col 1

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

Player X, enter row (0-2): 2
Player X, enter column (0-2): 1


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

AI is thinking...
AI (O) plays at row 1, col 0

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

Player X, enter row (0-2): 1
Player X, enter column (0-2): 2


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



```
AI is thinking...
AI (O) plays at row 0, col 2

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

Player X, enter row (0-2): 2
Player X, enter column (0-2): 0


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

AI is thinking...
AI (O) plays at row 2, col 2

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

Player X, enter row (0-2): 0
Player X, enter column (0-2): 1


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


Game Over!
It's a draw!
PS C:\Users\kjsce_comp39\Downloads>
```

Post Lab objective Questions:

1. **Which search is equal to minmax search but eliminates the branches that can't influence the final decision?**
    a. Breadth-first search
    b. Depth first search
    c. Alpha-beta pruning
    d. None of the above

Answer: Alpha-beta pruning

2. **Which values are independent in minmax search alogirthm?**
    a. Pruned leaves x and y
    b. Every states are dependant
    c. Root is independent
    d. None of the above

Answer: Pruned leaves x and y

Post Lab Subjective Questions:

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

Adversarial search is a type of search algorithm used in artificial intelligence that is specifically designed to find the optimal decision or strategy for a player in a competitive game or scenario. In adversarial search, the algorithm attempts to evaluate the possible outcomes of the game by taking into account the moves of both players and predicting their future moves based on the current state of the game.

The basic idea behind adversarial search is to model the game as a tree, where each node represents a possible game state and each edge represents a possible move by a player. The algorithm then uses various search techniques, such as the minmax algorithm, to traverse the tree and evaluate the potential outcomes of the game.

One important aspect of adversarial search is that it assumes that the opposing player is trying to win the game as well, and is making moves that are intended to block or interfere with the other player's strategy. Therefore, the algorithm must be able to anticipate and account for these moves in its evaluation of the game state.

Adversarial search is commonly used in a variety of games, such as chess, poker, and Go, as well as in other competitive scenarios, such as military strategy and cybersecurity. The goal of adversarial search is to find the best possible strategy or decision that a player can make in order to maximize their chances of winning the game, while also taking into account the actions of their opponent.

Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game.

Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where human intelligence and logic factor is used, excluding other factors such as luck factor. Tic-tac-toe, chess, checkers, etc., are such type of games where no luck factor works, only mind works.
Mathematically, this search is based on the concept of 'Game Theory.' According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.

2. Explain how alpha-beta pruning improves memory efficiency of algorithm

Alpha-beta pruning is a search algorithm used in adversarial search to improve the memory efficiency of the search algorithm. The main idea behind alpha-beta pruning is to reduce the number of nodes that are evaluated during the search process by eliminating those nodes that are guaranteed to lead to a worse outcome.
During the search process, the alpha-beta algorithm maintains two values, alpha and beta, which represent the bounds of the best score found so far for the maximizing player and the minimizing player, respectively. As the search progresses, the algorithm compares the current score of each node with the alpha and beta values, and if the score falls outside the bounds, the algorithm eliminates the node and all its descendants. This pruning process reduces the number of nodes that need to be evaluated, which improves the memory efficiency of the algorithm. By eliminating nodes that are guaranteed to lead to a worse outcome, the algorithm is able to focus on the more promising nodes and avoid wasting resources on nodes that will not impact the final decision.
Overall, alpha-beta pruning is an effective technique for improving the memory efficiency of search algorithms in adversarial search problems, and is widely used in game playing applications such as chess and checkers.

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

The game of chess is a classic example of a competitive, two-player game that can be solved using adversarial search algorithms such as the min-max and alpha-beta pruning algorithms. Here are a few ways in which these algorithms can benefit the game of chess:

- Efficient Search: Chess has a very large search space, with a typical game consisting of around 35 possible moves at each turn. The min-max and alphabeta pruning algorithms are designed to efficiently search through large decision trees, and can quickly eliminate branches that are unlikely to lead to a good outcome. This allows the algorithm to focus on the most promising moves, and can greatly reduce the number of nodes that need to be evaluated.
- Evaluation Function: In order to search through the decision tree, the min-max and alpha- beta pruning algorithms require an evaluation function that assigns a score to each possible game state. In chess, an evaluation function might take into account factors such as the number of pieces on the board, the relative strength of each player's position, and the potential for future moves. By using an

evaluation function, the algorithm can quickly eliminate moves that are unlikely to lead to a good outcome, and focus on moves that are more promising.

- Pruning: Alpha-beta pruning can be particularly effective in reducing the search space for chess. Because chess is a zero-sum game, where one player's gain is the other player's loss, it is possible to prune large sections of the decision tree by eliminating moves that are guaranteed to be worse than other moves that have already been evaluated. This can greatly reduce the number of nodes that need to be evaluated, and can lead to significant improvements in the efficiency of the algorithm.

Overall, the min-max and alpha-beta pruning algorithms can be very effective in solving the game of chess, and have been used to develop some of the strongest chessplaying computer programs in the world. These algorithms can help players to quickly evaluate potential moves and develop winning strategies, and can help to push the boundaries of what is possible in this classic game.

Conclusion: Thus, we have understood and performed the Implementation of Alpha Beta Pruning