**Batch: A-3 Roll No.: 16010122104**

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

|  |
| --- |
| **Title:** Implementation of Adversarial Search Algorithm |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Objective:** Implementation of Alpha-Beta Pruning algorithm

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **Course Outcome** | **After successful completion of the course students should be able to** |
| **CO 2** | 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**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_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, min-max algorithm, Alpha-Beta pruning,

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

**Adversarial Search Concept:-**

Adversarial search applies to scenarios where two or more players with conflicting objectives compete against each other, typically in turn-based games. Unlike standard search algorithms where a single agent seeks an optimal path, adversarial search must account for an opponent's optimal counterplays.

In the context of game playing:

* Players take turns making moves
* Each player aims to maximize their advantage while minimizing their opponent's
* The game state after each move can be represented as nodes in a game tree
* The evaluation of positions depends on which player's turn it is

**Alpha-beta pruning algorithm:**

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm that significantly reduces the number of nodes explored in the game tree without affecting the final decision.

Key concepts:

* Alpha (α) represents the minimum score that the maximizing player is guaranteed
* Beta (β) represents the maximum score that the minimizing player is guaranteed
* Pruning occurs when a move is found that proves a path is worse than a previously examined path

The main condition for pruning is when α ≥ β, meaning we can safely eliminate (prune) branches that won't affect the final decision.

**Chosen Problem:**

**Ultimate Tic-Tac-Toe** is a two-player game played on a 3×3 grid of tic-tac-toe boards. Each move dictates where the next player must play, and the goal is to win three small boards in a row on the large board. The game’s complexity makes it an excellent candidate for adversarial search algorithms like Minimax with Alpha-Beta pruning.

**Solution tree for chosen Problem:**

Root: All boards empty (X to play)

|

|-- X plays (4,4) [center of center board]

|

|-- O plays (3,3) in center board

| |

| |-- X must play in top-left small board (various options)

|

|-- O plays (3,4) in center board

| |

| |-- X must play in top-center small board (various options)

|

|-- O plays (3,5) in center board

| |

| |-- X must play in top-right small board (various options)

|

... (and so on for all empty cells in center board)

**Source Code:**

import copy

# Constants

EMPTY = '.'

PLAYER\_X = 'X'

PLAYER\_O = 'O'

DRAW = 'D'

def opponent(player):

    return PLAYER\_O if player == PLAYER\_X else PLAYER\_X

class SmallBoard:

    def \_\_init\_\_(self):

        self.cells = [[EMPTY for \_ in range(3)] for \_ in range(3)]

        self.winner = None

    def make\_move(self, row, col, player):

        if self.cells[row][col] != EMPTY or self.winner is not None:

            return False

        self.cells[row][col] = player

        self.check\_winner()

        return True

    def check\_winner(self):

        lines = []

        # Rows, columns, diagonals

        lines.extend(self.cells)

        lines.extend([[self.cells[r][c] for r in range(3)] for c in range(3)])

        lines.append([self.cells[i][i] for i in range(3)])

        lines.append([self.cells[i][2 - i] for i in range(3)])

        for line in lines:

            if line[0] != EMPTY and all(cell == line[0] for cell in line):

                self.winner = line[0]

                return

        if all(self.cells[r][c] != EMPTY for r in range(3) for c in range(3)):

            self.winner = DRAW

    def is\_full(self):

        return all(self.cells[r][c] != EMPTY for r in range(3) for c in range(3))

    def get\_legal\_moves(self):

        if self.winner is not None:

            return []

        return [(r, c) for r in range(3) for c in range(3) if self.cells[r][c] == EMPTY]

class UltimateTicTacToe:

    def \_\_init\_\_(self):

        self.boards = [[SmallBoard() for \_ in range(3)] for \_ in range(3)]

        self.board\_winners = [[None for \_ in range(3)] for \_ in range(3)]

        self.global\_winner = None

    def copy(self):

        new\_game = UltimateTicTacToe()

        new\_game.boards = [[copy.deepcopy(self.boards[r][c]) for c in range(3)] for r in range(3)]

        new\_game.board\_winners = [row[:] for row in self.board\_winners]

        new\_game.global\_winner = self.global\_winner

        return new\_game

    def make\_move(self, board\_row, board\_col, cell\_row, cell\_col, player):

        board = self.boards[board\_row][board\_col]

        if not board.make\_move(cell\_row, cell\_col, player):

            return False

        self.board\_winners[board\_row][board\_col] = board.winner

        self.check\_global\_winner()

        return True

    def check\_global\_winner(self):

        lines = []

        lines.extend(self.board\_winners)

        lines.extend([[self.board\_winners[r][c] for r in range(3)] for c in range(3)])

        lines.append([self.board\_winners[i][i] for i in range(3)])

        lines.append([self.board\_winners[i][2 - i] for i in range(3)])

        for line in lines:

            if line[0] is not None and line[0] != DRAW and all(cell == line[0] for cell in line):

                self.global\_winner = line[0]

                return

        if all(self.board\_winners[r][c] is not None for r in range(3) for c in range(3)):

            self.global\_winner = DRAW

    def get\_legal\_moves(self, next\_board):

        # next\_board: (row, col) or None

        moves = []

        if self.global\_winner is not None:

            return moves

        if next\_board is not None:

            r, c = next\_board

            if self.boards[r][c].winner is None:

                for cell in self.boards[r][c].get\_legal\_moves():

                    moves.append((r, c, cell[0], cell[1]))

                if moves:

                    return moves

        # If forced board is won/full, can play anywhere legal

        for br in range(3):

            for bc in range(3):

                if self.boards[br][bc].winner is None:

                    for cell in self.boards[br][bc].get\_legal\_moves():

                        moves.append((br, bc, cell[0], cell[1]))

        return moves

def evaluate(game, player):

    # Simple evaluation: +1000 for win, -1000 for loss, +10 for each small board won, -10 for each lost

    if game.global\_winner == player:

        return 1000

    if game.global\_winner == opponent(player):

        return -1000

    score = 0

    for r in range(3):

        for c in range(3):

            winner = game.board\_winners[r][c]

            if winner == player:

                score += 10

            elif winner == opponent(player):

                score -= 10

    return score

def minimax(game, depth, alpha, beta, maximizing\_player, player, next\_board):

    if depth == 0 or game.global\_winner is not None:

        return evaluate(game, player), None

    moves = game.get\_legal\_moves(next\_board)

    if not moves:

        return evaluate(game, player), None

    best\_move = None

    if maximizing\_player:

        max\_eval = float('-inf')

        for move in moves:

            new\_game = game.copy()

            new\_game.make\_move(\*move, player)

            next\_br, next\_bc = move[2], move[3]

            if new\_game.boards[next\_br][next\_bc].winner is not None:

                next\_forced = None

            else:

                next\_forced = (next\_br, next\_bc)

            eval, \_ = minimax(new\_game, depth-1, alpha, beta, False, player, next\_forced)

            if eval > max\_eval:

                max\_eval = eval

                best\_move = move

            alpha = max(alpha, eval)

            if beta <= alpha:

                break

        return max\_eval, best\_move

    else:

        min\_eval = float('inf')

        for move in moves:

            new\_game = game.copy()

            new\_game.make\_move(\*move, opponent(player))

            next\_br, next\_bc = move[2], move[3]

            if new\_game.boards[next\_br][next\_bc].winner is not None:

                next\_forced = None

            else:

                next\_forced = (next\_br, next\_bc)

            eval, \_ = minimax(new\_game, depth-1, alpha, beta, True, player, next\_forced)

            if eval < min\_eval:

                min\_eval = eval

                best\_move = move

            beta = min(beta, eval)

            if beta <= alpha:

                break

        return min\_eval, best\_move

import copy

import random

# --- (All your previous code here: SmallBoard, UltimateTicTacToe, etc.) ---

# ... (Paste all your code above up to the minimax function) ...

def random\_opponent\_move(game, next\_board):

    moves = game.get\_legal\_moves(next\_board)

    if not moves:

        return None

    return random.choice(moves)

def print\_board(game):

    """Pretty print the current state of the ultimate board."""

    def cell\_str(cell):

        return cell if cell != EMPTY else ' '

    for br in range(3):

        for r in range(3):

            row = []

            for bc in range(3):

                row.extend([cell\_str(game.boards[br][bc].cells[r][c]) for c in range(3)])

                if bc < 2:

                    row.append('|')

            print(' '.join(row))

        if br < 2:

            print('-' \* 17)

if \_\_name\_\_ == "\_\_main\_\_":

    game = UltimateTicTacToe()

    player = PLAYER\_X

    next\_board = None  # No constraint for first move

    depth = 3  # AI search depth

    turn = 0

    while game.global\_winner is None:

        print(f"\nTurn {turn+1}:")

        print\_board(game)

        if player == PLAYER\_X:

            # AI's turn

            score, move = minimax(game, depth, float('-inf'), float('inf'), True, PLAYER\_X, next\_board)

            print(f"AI ({PLAYER\_X}) chooses: {move} (score: {score})")

        else:

            # Random opponent's turn

            move = random\_opponent\_move(game, next\_board)

            print(f"Opponent ({PLAYER\_O}) chooses: {move}")

        if move is None:

            print("No moves left!")

            break

        game.make\_move(\*move, player)

        # Determine next forced board

        next\_br, next\_bc = move[2], move[3]

        if game.boards[next\_br][next\_bc].winner is not None:

            next\_board = None

        else:

            next\_board = (next\_br, next\_bc)

        # Switch player

        player = opponent(player)

        turn += 1

    print("\nFinal board:")

    print\_board(game)

    print("\nGame Over!")

    if game.global\_winner == DRAW:

        print("It's a draw!")

    else:

        print(f"Winner: {game.global\_winner}")

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

    game = UltimateTicTacToe()

    player = PLAYER\_X

    next\_board = None  # No constraint for first move

    depth = 3  # Increase for stronger AI, decrease for faster response

    score, move = minimax(game, depth, float('-inf'), float('inf'), True, player, next\_board)

    print("Best move for", player, "is:", move, "with score", score)

**Output Screenshots:**

**A screenshot of a computer program

Description automatically generated**

**Conclusion:**

**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: a. Pruned leaves x and y**

**Post Lab Subjective Questions:**

1. **What is the main purpose of the Alpha-Beta pruning algorithm in game trees?**

**Ans:**

The main purpose of the Alpha-Beta pruning algorithm is to optimize the Minimax search by eliminating branches in the game tree that don’t affect the final decision. This allows the algorithm to search deeper in the same amount of time, improving performance without sacrificing accuracy.

1. **How does Alpha-Beta pruning improve the efficiency of the Minimax algorithm?**

**Ans:**

Alpha-Beta pruning improves efficiency by cutting off unnecessary branches during the search. If it’s already known that a move is worse than a previously examined one, the algorithm prunes (ignores) that branch. This reduces the number of nodes evaluated from O(b^d) to O(b^(d/2)) in the best case, where b is the branching factor and d is the depth.

1. **Explain the terms alpha and beta in the context of the Alpha-Beta pruning algorithm.**

**Ans:**

Alpha (α): The best (highest) value that the maximizing player can guarantee so far.

Beta (β): The best (lowest) value that the minimizing player can guarantee so far.

If the minimizer sees that a move leads to a value less than or equal to alpha, it will prune that branch because the maximizer won’t let it happen.

1. **What condition must be met for a node to be pruned during Alpha-Beta pruning?**

**Ans:**

A node is pruned when:

In a maximizing node, if α ≥ β, we stop evaluating further children.

In a minimizing node, if β ≤ α, we also stop evaluating further children. This condition ensures that further exploration cannot improve the outcome.

1. **Compare and contrast Alpha-Beta pruning with the standard Minimax algorithm.**

**Ans:**

| **Feature** | **Minimax Algorithm** | **Alpha-Beta Pruning** |
| --- | --- | --- |
| Goal | Find the optimal move | Same, but more efficient |
| Efficiency | Evaluates all nodes | Prunes unnecessary branches |
| Time Complexity | O(b^d) | O(b^(d/2)) in best case |
| Space Complexity | Same | Same |
| Decision Quality | Optimal | Optimal |
| Practical Use | Less efficient in deep trees | Preferred in real-time games |