CSE 356 – Artificial Intelligence

ASSIGMENT 2

Submitted by:

Ahmed Hani Ahmed 7387

Mohamed Hazem Hafez 7729

Mahmoud Haytham 7560

Due on:

March 21st, 2024

Connect 4 AI agent using minimax algorithm

Intro:

Connect 4, a timeless classic among board games, offers players a deceptively simple yet strategically rich experience. With the objective of aligning four colored discs in a row on a grid, players must navigate a landscape of potential moves, balancing offense and defense to outmaneuver their opponent. This seemingly straightforward premise belies the depth of strategic thinking required to excel in the game, making Connect 4 an ideal playground for the exploration of artificial intelligence (AI) techniques.

In recent years, the fusion of Connect 4 with AI has seen remarkable progress, particularly with the adoption of the Minimax algorithm. Rooted in game theory, Minimax serves as a guiding principle for decision-making in two-player zero-sum games like Connect 4. By systematically analyzing potential moves and their consequences, Minimax empowers AI agents to simulate future game states, enabling them to make informed decisions aimed at minimizing potential losses and maximizing potential gains. Through this synergy of game theory and AI, connect 4 has become not just a test of human intellect, but also a proving ground for the capabilities of modern AI systems in strategic decision-making.

Using Minimax Algorithm

Pseudocode

```
function minimax(position, depth, maximizingPlayer):
    if maximizingPlayer:
        player = 2
    else:
        player = 1

    if depth == 0 or depth >= depth_limit or position.isOver():
        playerScore = score_position(position, 1)
        botScore = score_position(position, 2)
        return botScore - playerScore

    if maximizingPlayer:
        maxEval = -infinity
        for each child in position:
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = infinity
        for each child in position:
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

Explanation

The minimax algorithm is a decision-making algorithm commonly used in two-player games with perfect information, such as chess, checkers, or tic-tac-toe. Its primary objective is to determine the optimal move for a player by recursively exploring the game tree.

In essence, the algorithm simulates gameplay by considering all possible moves that can be made from the current game state and evaluating their outcomes. It alternates between two types of players: maximizing and minimizing.

Maximizing player: A player who aims to maximize their own score or utility while minimizing the opponent's score or utility.

Minimizing player: A player who aims to minimize their opponent's score or utility while maximizing their own score or utility.

The minimax algorithm proceeds by recursively exploring the game tree to a certain depth or until reaching a terminal state. At each step, it evaluates the

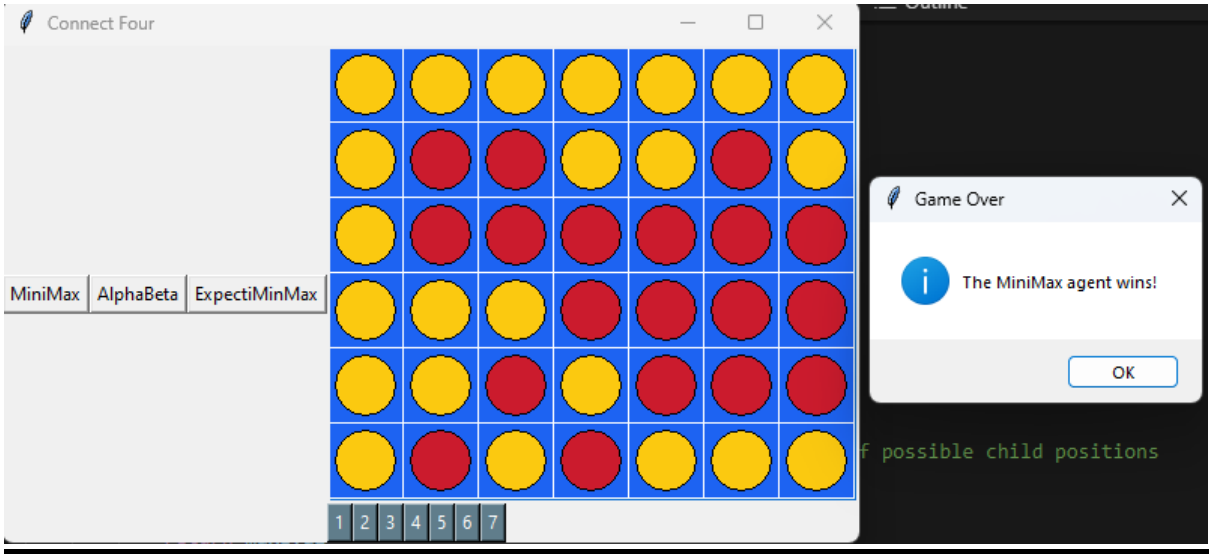
current game position and computes a score or utility value indicating the desirability of that position for the current player.

When it's the turn of a maximizing player, the algorithm selects the move that leads to the highest score, assuming the opponent will make moves to minimize this score. Conversely, when it's the turn of a minimizing player, the algorithm selects the move that leads to the lowest score, assuming the opponent will make moves to maximize this score.

Through this recursive exploration and evaluation of potential moves, the minimax algorithm determines the best move for the current player, assuming both players play optimally. It provides a systematic approach to decision-making in adversarial settings by considering all possible future outcomes and selecting the move that maximizes the player's chances of winning while minimizing the opponent's chances.

Overall, the minimax algorithm is a fundamental concept in artificial intelligence and game theory, providing a framework for strategic decision-making in games with multiple players and complex decision trees.

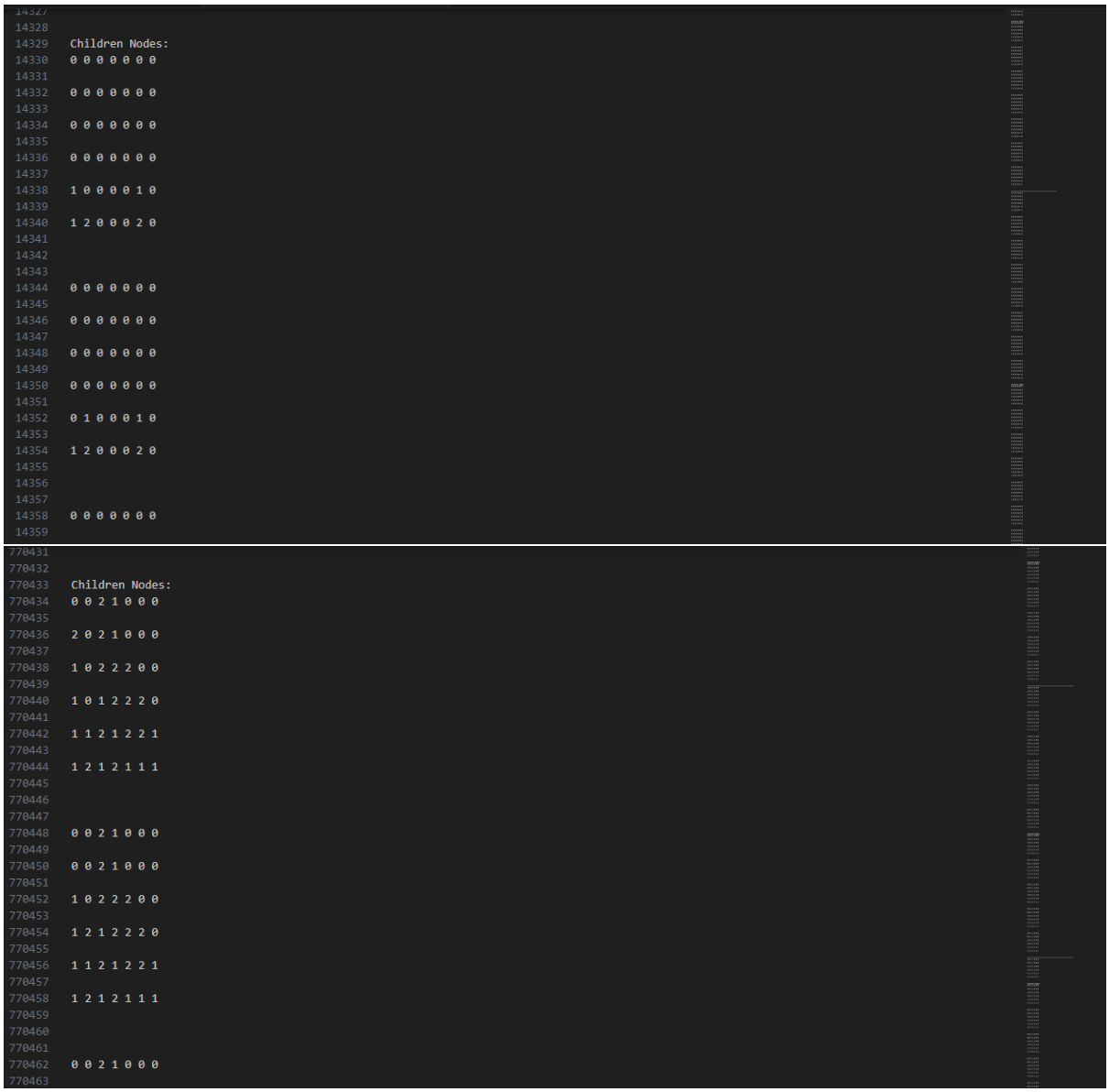
Sample run




```
Player's score = 7
AI score = 10
```

```
Expaneded nodes number: 2801
Time taken for this move: 3.493612051010132
```

Expanded Tree



Using Minimax Algorithm using Alpha Beta pruning

Pseudocode

```
function minimaxAB(position, depth, maximizingPlayer, alpha, beta):
    if maximizingPlayer:
        player = 2
    else:
        player = 1

    if depth == 0 or depth >= depth_limit or position.isOver():
        playerScore = score_position(position, 1)
        botScore = score_position(position, 2)
        return botScore - playerScore

    if maximizingPlayer:
        maxEval = -infinity
        for each child in position:
            eval = minimaxAB(child, depth - 1, false, alpha, beta)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = infinity
        for each child in position:
            eval = minimaxAB(child, depth - 1, true, alpha, beta)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return minEval
```

Explanation

Alpha-beta pruning is an optimization technique applied to the minimax algorithm to further reduce the number of nodes evaluated in the game tree. It works by eliminating branches of the tree that are guaranteed to be suboptimal, thus reducing the search space and improving the algorithm's efficiency.

Game Tree Exploration: Similar to the basic minimax algorithm, the alpha-beta pruning version begins by exploring the game tree recursively, considering all possible moves and their outcomes. The goal is to determine the optimal move for the current player while minimizing the number of nodes evaluated.

Maximization and Minimization: The algorithm alternates between maximizing and minimizing players, similar to the basic minimax algorithm. A maximizing player aims to choose moves that lead to positions with higher utility values, while a minimizing player aims to choose moves that lead to positions with lower utility values.

Alpha and Beta Values: In addition to maintaining utility values, the algorithm also keeps track of two values: alpha and beta. Alpha represents the maximum score the maximizing player is assured of, while beta represents the minimum score the minimizing player is assured of.

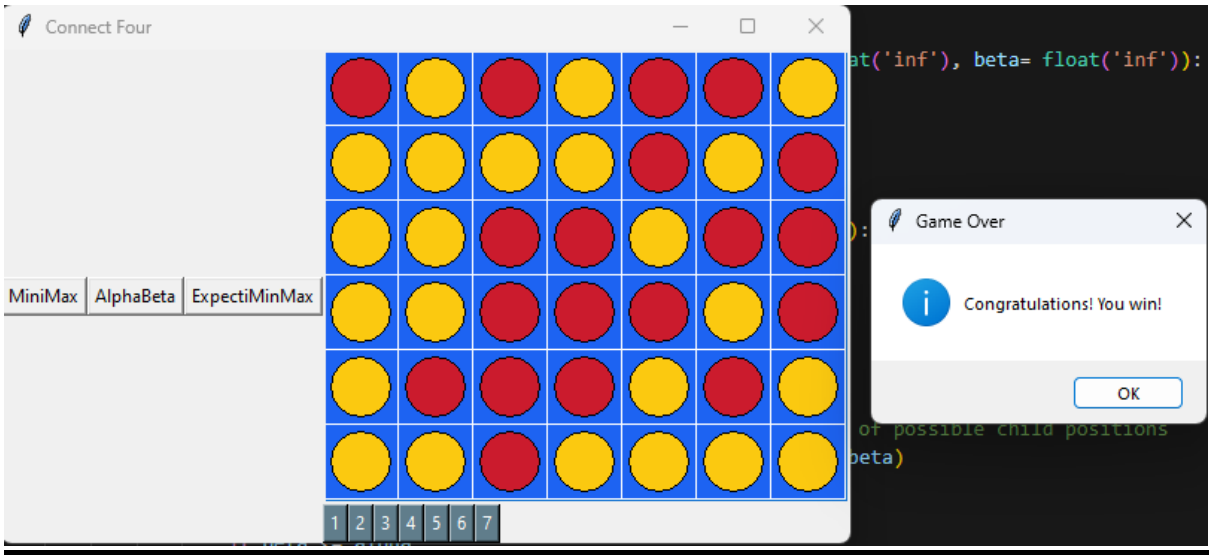
Pruning: As the algorithm progresses, it updates the alpha and beta values based on the evaluations of different moves. When exploring a maximizing player's node, the algorithm updates alpha with the maximum utility value found so far. Conversely, when exploring a minimizing player's node, it updates beta with the minimum utility value found so far.

Pruning Condition: At each level of the tree, the algorithm checks if the beta value becomes less than or equal to the alpha value. If this condition is met, it indicates that the current branch of the tree cannot affect the final decision, as it won't lead to a better outcome than what's already been found. In such cases, the algorithm prunes the remaining subtree, skipping further exploration of that branch.

Efficiency Improvement: By pruning branches of the tree that are guaranteed to be suboptimal, alpha-beta pruning significantly reduces the number of nodes evaluated compared to the basic minimax algorithm. This leads to faster computation times, making it particularly useful for games with large branching factors and deep search trees.

Overall, the alpha-beta pruning version of the minimax algorithm provides an efficient approach to decision-making in adversarial settings, enabling players to make strategic moves while minimizing computational overhead. It's a fundamental technique used in AI agents for board games and other competitive environments.

Sample run




```
Expaneded nodes number: 2
Player's score = 7
AI score = 5
```

```
Expaneded nodes number: 2801
Time taken for this move: 3.376499891281128
```

Comment

It is possible to beat the AI as the used depth is not so high, but as we increase the depth the difficulty becomes harder.

Expanded Tree

Using ExpectiMinimax

Pseudocode

```
function expectiminimax(position, depth, maximizingPlayer):
    if maximizingPlayer:
        player = 2
    else:
        player = 1

    if depth == 0 or depth >= depth_limit or position.isOver():
        player_score = score_position(position, 1)
        bot_score = score_position(position, 2)
        return bot_score - player_score

    if maximizingPlayer:  # Bot's turn
        max_value = -infinity
        for each child, prob in get_child_nodes_with_probabilities(position, player):
            child_value = expectiminimax(child, depth - 1, false)
            max_value += prob * child_value
        return max_value
    else:  # Player's turn
        min_value = infinity
        for each child, prob in get_child_nodes_with_probabilities(position, player):
            child_value = expectiminimax(child, depth - 1, true)
            min_value += prob * child_value
        return min_value
```

Explanation

The expectiminimax function is an extension of the minimax algorithm, specifically designed to handle uncertainty in games where chance plays a role, such as dice-based games or card games. It evaluates possible moves while considering the probabilities associated with each outcome.

At its core, the algorithm explores the game tree recursively, similar to the basic minimax algorithm. It evaluates potential moves and outcomes, alternating between maximizing and minimizing players to select the best move.

The function begins by determining the current player based on the maximizingPlayer parameter. If the current player is maximizing, it signifies that the algorithm is trying to maximize the score for that player; otherwise, it aims to minimize the score.

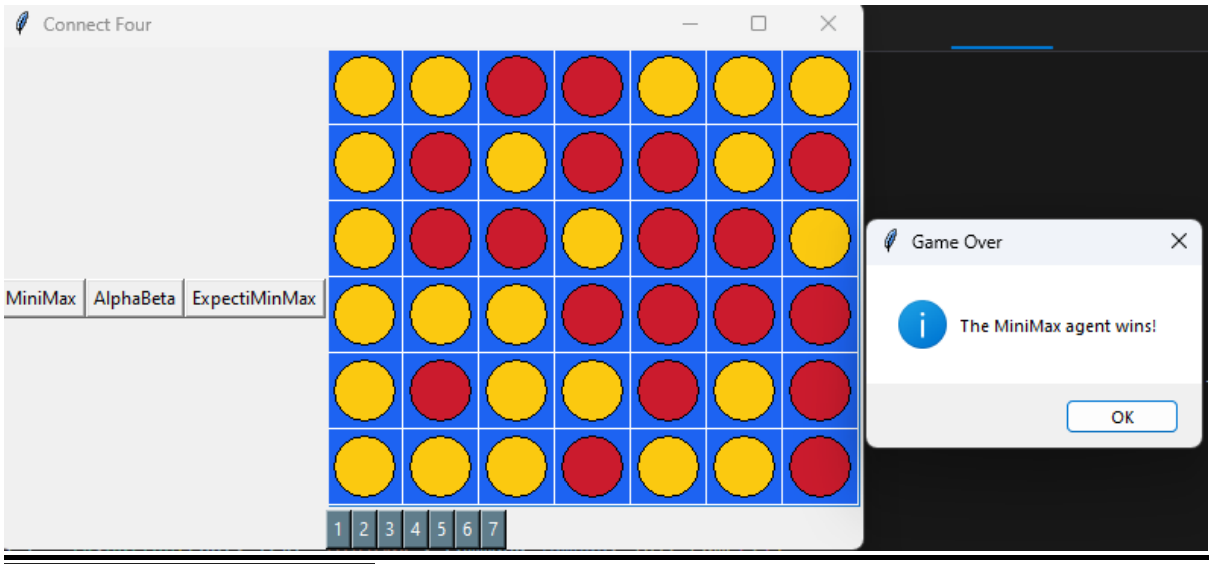
The algorithm evaluates whether the current game state is a terminal state or if the maximum depth limit has been reached. If either condition is met, the function calculates and returns the utility value of the position for the current player.

If the game is not over, the function generates all possible legal moves from the current position and recursively evaluates each resulting position. However, unlike the basic minimax algorithm, the expectiminimax function also considers the probabilities associated with each child node.

During the evaluation of child nodes, the algorithm weights the utility values of each node by their probabilities. This allows the algorithm to make informed decisions in uncertain environments, where the outcome of a move may not be deterministic.

Through this process of recursive exploration and probabilistic evaluation, the expectiminimax algorithm effectively searches the game tree, taking into account both the player's decisions and the uncertain nature of the game. It provides a systematic approach to decision-making in games with chance elements, enabling players to make strategic moves while considering the probabilities of different outcomes.

Sample run

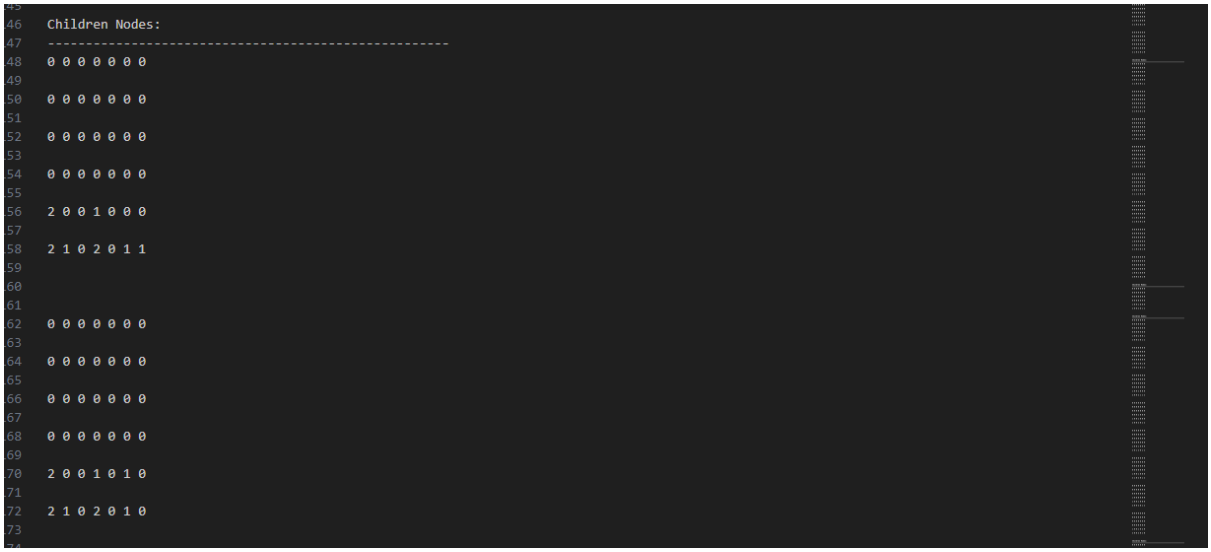



```
Player's score = 3
AI score = 6
```

```
Time taken for this move: 2.5907044410705566
```

Expanded tree

```
Children Nodes:
--------------------------------------------------
0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 1 2 0 0

0 0 0 2 2 2 0

0 2 2 1 2 1 1

0 1 1 2 1 1 1


0 0 0 0 0 0 0

0 0 0 0 0 0 0

0 0 0 1 2 1 0

0 0 0 2 2 2 0

0 2 2 1 2 1 0

0 1 1 2 1 1 1


0 0 0 0 0 0 0
```



Comparison

According to time taken by each algorithm we find that the best algorithm which takes the least time is the expectiminimax algorithm and the second best is the alpha beta pruning which is just a little bit better than the minimax.