Artificial Intelligence
Adversarial Search

Fall 2024
Parsa Darban 810100141
This file contains the report and results of the simulations conducted.

Game

In this section, we will examine the game Connect4, which is a two-player strategic game. Each player takes turns dropping a piece into a column, and the piece falls to the lowest available space in that column. If four pieces are placed consecutively in a row, column, or diagonal, the player wins.

In this game, you will play with red pieces, and the yellow pieces represent your opponent, the computer. The starting player is chosen randomly. Assume that the computer plays optimally, just like you.

At the beginning of the code, we will create the necessary libraries and variables.

Next, we will create a graphical section that will enhance our understanding of the game, although it is not necessary for viewing the results.

In the next section, we will initialize the game board and its parameters. Since the project asks us to use Alpha-Beta pruning, we will set a flag for it. Then, we will focus on placing a piece on the game board and finding the next available row in a specific column. Additionally, we need to check whether a winner is determined at this point.

In the `evaluate_window` function, we assess a 4-cell window for scoring purposes in the Minimax algorithm. For example, if three cells are occupied by the player and one is empty, it receives a specific positive score. Conversely, if three cells are occupied by the opponent and one is empty, it gets a negative score to block the opponent.

The `score_position` function calculates the total score for the board from the perspective of a specific player and returns the result for the Minimax function.

The `is_terminal_node` and `get_valid_locations` functions are related to the game state.

Additionally, there is a heuristic for the Minimax function, which returns a high score for a board that results in a win, a low score for a board that results in a loss, and otherwise a score based on `score_position`.

The `best_cpu_score` and `get_cpu_move` functions are both related to the opponent's (CPU's) move. Specifically, `get_cpu_move` makes a move based on the CPU's score and controls randomness using a coefficient.

The `get_human_move` function calls the Minimax function with the given depth and returns the column and row selected for the player's move.

The remaining sections of the code are related to gameplay execution and result evaluation.

Before analyzing the results, let's explore the Minimax algorithm. This algorithm, which is an adversarial search algorithm, is used to find the value of each state.

We know that each of our terminal nodes can have a specific score, but we don't know when we will reach them. Additionally, our game is zero-sum, meaning that the score of the two players in each state are complementary. Therefore, if we are in a maximizing state, we need to choose the maximum value from each of our child nodes, while these child nodes, in turn, must find the minimum value from the next set of child nodes, and so on.

As you can understand, this algorithm requires a significant amount of memory and time to execute. Essentially, it operates like a depth-first search (DFS). In fact, if we could evaluate all possible game states, we would be able to determine the winner of the game. However, due to the time and memory limitations, this is impractical. Therefore, we need to apply certain constraints to reduce the search space.

One of the techniques to optimize the Minimax algorithm is Alpha-Beta pruning. This pruning ensures that if we find a value for a state that we are confident will not be affected by exploring other branches, we can skip evaluating those branches.

The pseudocode for this is as follows, and we can use it as a basis for implementing Alpha-Beta pruning:

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

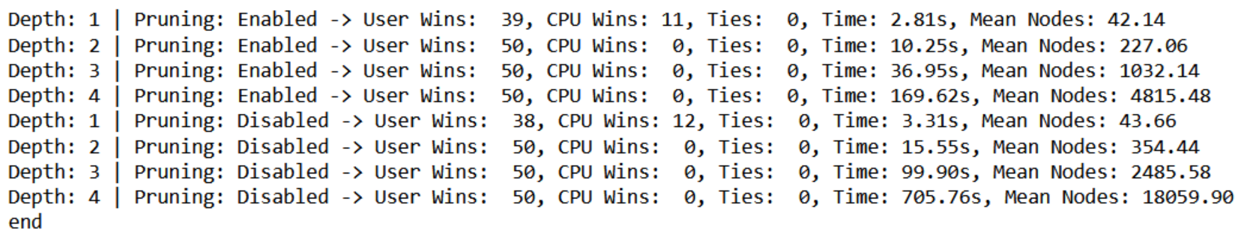
If we assume that alpha represents the best score for a maximizing state, and we are currently in that state, the child nodes will be in a minimizing state. We will then evaluate each child node, taking the maximum score from the child nodes. If at any point, this value exceeds beta (the minimum score that the minimizer is guaranteed), we know that the remaining branches in this subtree cannot affect the final result, and thus we can prune (stop evaluating) that branch.

However, this method still faces problems in terms of time and memory. Additionally, the speed of this approach depends on the order of the child states. This means that alpha or beta may be found in the last possible state, which can affect the Minimax algorithm.

Given the problem mentioned, we turn to a trick that we previously used in search algorithms. This involves setting a depth limit for the algorithm. We start by defining a fixed depth for the algorithm and then, by increasing the depth, we observe how the result changes.

Conclusion

We have applied the depth-limiting method both for pruning and without pruning. The results for depths one through four are as follows:



```
Depth: 1 | Pruning: Enabled -> User Wins:  39, CPU Wins: 11, Ties:  0, Time: 2.81s, Mean Nodes: 42.14
Depth: 2 | Pruning: Enabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 10.25s, Mean Nodes: 227.06
Depth: 3 | Pruning: Enabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 36.95s, Mean Nodes: 1032.14
Depth: 4 | Pruning: Enabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 169.62s, Mean Nodes: 4815.48
Depth: 1 | Pruning: Disabled -> User Wins:  38, CPU Wins: 12, Ties:  0, Time: 3.31s, Mean Nodes: 43.66
Depth: 2 | Pruning: Disabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 15.55s, Mean Nodes: 354.44
Depth: 3 | Pruning: Disabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 99.90s, Mean Nodes: 2485.58
Depth: 4 | Pruning: Disabled -> User Wins:  50, CPU Wins:  0, Ties:  0, Time: 705.76s, Mean Nodes: 18059.90
end
```

1. As we can see, with an increase in depth, the time spent increases as well. This is because the number of states that need to be evaluated grows larger. This fact also explains the increase in the average number of states checked. As we observe, the player's win rate also improves with the increased depth. This behavior is expected, as increasing the depth allows the algorithm to explore more potential moves and outcomes, making better decisions based on a deeper understanding of the game state. However, this comes at the cost of increased computational time and memory usage, as the search tree expands exponentially.

2. If we create a function, like a heuristic, for each state that assigns a score to the leaves of a path and then sorts them in order, naturally, our pruning will become faster as well. Note that the heuristic in the code is used to evaluate the overall score of the game board, not for prioritizing the leaves.

3. The branching factor indicates the number of different moves available to the player at each stage. The branching factor has a significant impact on the performance of search algorithms such as minimax. At the beginning of the game, the branching factor is higher (close to 7), which increases the number of nodes that the algorithm must evaluate. In the mid and late stages of the game, as the branching factor decreases, the number of nodes to be evaluated reduces, and the algorithm can reach a conclusion more quickly.

4. Alpha-beta pruning eliminates many branches that do not need to be evaluated, which significantly reduces the number of states that need to be checked. With fewer states to evaluate, the Minimax algorithm runs faster because it no longer has to assess all possible branches. However, there is a trade-off where some precision is lost during this pruning process. Specifically, when a value is chosen, and we stop evaluating other leaves in that branch, it does not affect the values of alpha or beta. This pruning is performed because we have already found the min or max value in that subtree, meaning further exploration would not change the outcome.

5. Minimax considers all possible moves of the opponent and aims to find the optimal strategy to counter the strongest moves. However, when the opponent plays randomly, much of this analysis becomes pointless because the opponent may choose random moves instead of logical ones.

   The ExpectMax algorithm can also be used, which calculates a weighted sum of probabilistic types and selects the maximum of them. Using this algorithm is not necessarily optimal, but it can be used when better probabilities can be guessed.

   The MCTS (Monte Carlo Tree Search) algorithm is a suitable alternative for dealing with random opponents. This algorithm simulates various moves across multiple games and analyzes the results statistically. MCTS makes decisions based on multiple simulations and operates probabilistically rather than through deep search, making it more suitable for opponents who play randomly. This algorithm can quickly identify which moves yield better results on average, even if the opponent behaves randomly.