AI PRINCIPLES & TECHNIQUES

Programming Report

Four-In-A-Row

Giuliano Rojas s1097694

Sandro Kvrgić s1091371

Radboud University 6/10/2023

# 1 Introduction

For this project, our goal is to architect an intelligent opponent that can compete against a human player. Firstly, we must design a working data structure, in this case, a Tree Search algorithm that will allow us to store and track the states of the game. We then are aiming to achieve this intelligent opponent by implementing a Minimax algorithm with Alpha-beta pruning integrated. Based on certain heuristics given, the intelligent opponent will then be able to calculate and evaluate the possible set of outcomes and playable scenarios, until it results in the best possible move to be played. By using both a Minimax algorithm with pruning and one without, this gives us the possibility to experiment on the time-complexities of the runtime through both approaches, to better understand search algorithms.

2 Specification

The project revolves around the commonly known game of four-in-a-row. The game is played on a 6 by 7 board, where the objective of the game is to get 4 tokens of the player to form a row, this can be either vertically, horizontally, or diagonally, all whilst keeping the other player from winning. Whoever gets four of their tokens first in a row wins. The given program has the basic logic of the game already implemented. It uses a 2-dimensional array to store the plays made by the players, in this way, it can be checked for any case in which a player achieves 4 in a row. Our first task is to initially create a tree data structure that will expand the possible game states depending on the possible moves, to a specified level of depth or until a win or lose scenario is reached. This will allow us to then apply to the different stored game states an evaluation through the heuristics. Which will then be used to calculate the most favorable game state. After managing an appropriate form of storing our game states, we need to develop a system to find the optimal next play for the intelligent opponent.

For this, we are using the Minimax algorithm, which consists of two parts Minimizer and Maximizer which take turns. The algorithm starts in the parent node, and it uses a depth-first search expansion until it reaches the leaf node. The first step depends on how deep in the tree the leaf node is. It can either start with the Maximizer or the Minimizer, which are parents of that leaf node, but regardless of which one it starts with, it always compares the heuristics of all of the child nodes connected to it. Maximizer will choose the node with the biggest heuristic, meaning it chooses the best possible move in that state. On the other hand, Minimizer will choose the node with the smallest heuristic, which represents the best possible move the opponent can make. When the choice is made, the heuristic of the node is updated to either a minimized or maximized heuristic, and the process is repeated. Interchanging the Minimizer and the Maximizer while doing backpropagation, ensures that the moves which are made are always optimal.

However, this algorithm can be improved if Alpha-beta pruning is integrated into the Minimax algorithm. Using it enables a more optimal search for the next best move, greatly reducing the number of calculations. Alpha is the minimum heuristic which is assured to the Maximizer in the current state, whilst beta is the maximum heuristic assured to the Minimizer in the current state. Alpha is initialized with a negative infinity and beta with a positive infinity for its value, and those values are constantly updated while going through the tree. If the alpha value is bigger or equal to the beta value, then the path gets pruned, meaning discarded. Using this method, the algorithm does not check all the possible nodes because it prunes the pathways in the tree which cannot give a more optimal solution than it has already explored.

3 Design

Pseudocode for MinMax:

|  |
| --- |
| function MinMax( board, current player, alpha, beta, depth)  if depth is 0 board terminates {  returns heuristic evaluation of board  }  if current player is playerId { # maximizing player turn  for each move i in board {  if move is valid {  newBoard = board.getNewBoard(i, current player)  value1 = MinMax(newBoard, 3 – current player, depth –1) #recursion  maxValue = max(maxValue, value1) #update maxValue }  } returns maxValue  } else {  for each move i in board {  if move is valid {  newBoard = board.getNewBoard(i, current player)  value2 = MinMax(newBoard, 3 – current player, depth –1) #recursion  minValue = min(minValue, value2) #update minValue }  } returns minValue |

Pseudocode for Alpha-Beta Pruning:

|  |
| --- |
| function MinMax with Alpha Beta Pruning( board, current player, alpha, beta, depth)  if depth is 0 board terminates {  returns heuristic evaluation of board  }  if current player is playerId { # maximizing player turn  maxValue is negative infinity  for each move i in board {  if move is valid {  newBoard = board.getNewBoard(i, current player)  value1 = MinMaxABP(newBoard, 3 – current player, alpha, beta, depth –1)  maxValue = max(maxValue, value1)  alpha = max(alpha, value1)  if alpha is bigger or equal than beta { # alpha-beta pruning  break }} #beta pruned  } returns maxValue  } else {  minValue is positive infinity # minimizing player turn  for each move i in board {  if move is valid {  newBoard = board.getNewBoard(i, current player)  value2 = MinMaxABP(newBoard, 3 – current player, alpha, beta, depth –1)  minValue = min(minValue, value2)  beta = min(beta, value2)  if alpha is bigger or equal than beta { #alpha-beta pruning  break }} # alpha pruned  } returns minValue |

4 Implementation

For the Tree Structure implementation, a TreeNode class was created in order to keep track of the nodes within the tree structure. This data structure will allow us to visualize the gameboard configurations. The TreeNode class contains multiple getters that allow us to analyze each node. When printing the tree, it will recursively do so and reveal data such as at what level of depth that node is found, the playerId which reveals who played that move, and the specific column that was played. However, the actual construction of the tree structure occurs within the Game class. Inside this class we find a buildGameTree method that serves in recursively adding children to a TreeNode object which saves its children within an ArrayList of type TreeNode. With this TreeNode object completely built it is then possible to store the game configurations. Making the tree structure complete.

To implement the minMax algorithm we initially created a private function within the MinMaxPlayer class. Inside this class, we have access to the functions and methods required in order to begin the implementation. However, one of the changes to the code was that we decided to change is move the *winning* and *isOver* function from the Game class into that of the MinMaxPlayer class. This facilitated the ability to check the board for a winner. In order to get the leaf node’s heuristics. Next, we implemented the initial maximizing node already in the makeMove function of the MinMaxPlayer class, so that each turn the max value for it is taken. From that method we then call the MinMax algorithm. Which is inside a for-loop to initialize the first 7 nodes of the possible moves available. The logic for the minMax is then applied to find the best next move as noted within the pseudocode. To switch between the minMax with pruning and without, one can add the alpha and beta parameters, (Integer.MIN\_VALUE) and (Integer.MAX\_VALUE) respectively, in the makeMove's minMax call to switch between the two algorithms.

Lastly implementing the Alpha-beta pruning was a swift procedure. Initially, we copied the minMax algorithm previously coded and added the alpha and beta parameter to the new MinMax algorithm. To begin the pruning factor to the code, we then used the alpha and beta parameters within each for loop as in the pseudocode. We then update the alpha value when maximizing and similarly we update the beta value when minimizing. Then comparing the values, at the end of the process to break the loop and therefore, eliminating the branches which lead to less optimal node states.

5 Complexity and Testing

The runtime complexity of the Minimax algorithm is O(b^d) where b is the branching factor, which essentially means the average of possible moves at each state. Variable d represents the depth of the tree, meaning that the runtime complexity is exponential in the depth of the tree.

When alpha-beta pruning is implemented into the Minimax algorithm the runtime complexity can drastically improve. The best possible scenario is O(b^(d/2)), but pruning sometimes does not work, causing the worst scenario leading to the time complexity of O(b^d). In the case of our game, the average time complexity is a lot closer to the O(b^(d/2)) than to O(b^d).

To prove the time complexities which are provided above, we are going to test the algorithms on the game. All of the tests will be conducted in similar scenarios, with the same moves for the human player and mostly the same moves for the algorithm. In the table below you can see the number of evaluations of board states based on the testing criteria. The higher the number of evaluations the higher the runtime complexity is. By changing the board size, depth, and number of connected elements X or O required to win the game, denoted by N, we hope to get results that will align with the time complexities of our algorithms.

6 Results

|  |  |  |
| --- | --- | --- |
| Game Specifications | Minimax algorithm | Minimax algorithm with alpha-beta pruning |
| Board: 7\*6  Depth:4  N:4 | 11,093 | 2388 |
| Board: 7\*6  Depth:6  N:4 | 608,970 | 47,993 |
| Board: 7\*6  Depth:8  N:4 | 22,952,098 | 314,139 |
| Board: 7\*6  Depth:6  N:5 | 630,424 | 42,803 |
| Board: 7\*6  Depth:6  N:6 | 733,611 | 43,637 |
| Board: 9\*8  Depth:6  N:4 | 2,817, 158 | 89,574 |
| Board: 11\*10  Depth:6  N:4 | 9,623,646 | 152, 340 |

*\*The sequence of moves which were used to get these results are {4,2,3,5,4, if it has not ended at this point, play any move which does not block the win}.*

7 Discussion

Based on the results of both algorithms we can conclude that the Minimax algorithm with alpha-beta pruning is much more efficient. We can also observe that the larger the number of evaluations in the Minimax algorithm, leads to the highest level of efficiency in the Minimax algorithm with alpha-beta pruning. Depth has a major influence on the time complexity, where the larger the depth, the larger the number of evaluations and runtime. Board size also greatly impacts the time complexity in the same way but to a slightly lesser degree than depth. The smallest influence on the time complexity is N, it has a small effect on the time complexity, but nowhere near the depth and board size. We conclude that the Minimax algorithm with alpha-beta pruning is a better option than Minimax algorithm and it should always be used to reduce the complexity of the task.

8 Conclusion & Reflection

From the results we can conclude that both versions of the MinMax algorithm and the Tree data structure have been successfully implemented. It is also apparent that the addition of Alpha-beta pruning greatly reduces the runtime complexity of the program, guaranteeing a quicker, more efficient, intelligent opponent. There were a couple obstacles that made the implementation of the program difficult, specifically the heuristics, these can cause instances in which the code for the calculation of the makeMove breaks and gives a play on a column that is already full. We attempted to counter this in various ways by checking if the play was valid prior to taking it into consideration, however we were unsuccessful. This error happens rarely and for the most part, the game is still playable and the MinMax player with a high level of depth still makes for a worthy adversary against a human candidate. One of the things we could improve is essentially how the heuristics are calculated so that no errors occur.

9 References

<https://youtu.be/l-hh51ncgDI?si=jyTCyH3c4BXA2UfF>

<https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/>