AI PRINCIPLES & TECHNIQUES

Four in a Row

Quince Kersten *S1073935*
Gregors Lasenbergs *S1075747*
Radboud University *October 6, 2023*

---

1 Introduction

Game decision trees can get very large when the complexity of the game increases, so it is important to always make sure that the algorithms we use are optimal both in time and space. We developed two game search agents that reasoned in the game tree searching for the fastest way to find subset of the tree with the winning path. Adversarial search algorithms differ from the regular search algorithms by having an enemy that is changing the game tree in every direction you don't want to. Games are "zero sum" and both players play to win.

To better understand both Mini-max algorithm and Mini-max algorithm with Alpha-Beta pruning, we implemented them both in Java programming language and compared the complexity of both of them by measuring the amount of nodes explored in the tree by each algorithm. For comparing the algorithms it is not that important for us to check who wins, so for the result part we made both algorithms play against each other and checked the complexity for them when exploring the same game tree.

Contents

2 Method

2.1 Node Class and Building the Game Tree

In the context of implementing the minimax and alpha-beta pruning algorithms, constructing the game tree plays a crucial role. The game tree represents all possible moves and game states, enabling the algorithms to evaluate and choose the best move for the current player. This section introduces the `Node` class and discusses the process of building the game tree.

2.1.1 Node Class

The `Node` class is a fundamental component of our implementation. It encapsulates essential information about a game state within the tree. It includes the following attributes:

- `board`: This attribute stores the current board state, representing the arrangement of pieces on the game board.
- `player`: The `player` attribute indicates whose turn it is, with values 1 or 2 representing Player 1 and Player 2, respectively.
- `move`: The `move` attribute records the move that led to the current state, typically denoted by the column index where a piece is placed.
- `children`: A list of child nodes, represented by the `children` attribute, contains references to possible future game states.

Child nodes are added to a node when exploring possible moves, allowing for the construction of the game tree. For instance, when a player makes a move, a child node is created to represent the resulting game state, and it is added to the parent node's list of children.

```
public class Node {
    private Board board; // The current board state
    private int player;  // The player whose turn it is (1 or 2)
    private int move;    // The move that led to this state (column index)
    private List<Node> children; // Children nodes

    // Constructor
    public Node(Board board, int player, int move) {
        this.board = board;
        this.player = player;
        this.move = move;
        this.children = new ArrayList<>();
    }

    // Getters for board, player, and move
    public Board getBoard() {
        return board;
    }

    public int getPlayer() {
        return player;
    }

    public int getMove() {
        return move;
    }

    // Add a child node to the current node
```

```
    public void addChild(Node child) {
        children.add(child);
    }

    // Get the list of child nodes
    public List<Node> getChildren() {
        return children;
    }
}
```

2.1.2 Building the Game Tree

The process of building the game tree is facilitated by the `buildGameTree` method. This method is responsible for recursively exploring possible moves and game states, creating child nodes for each valid move. The `depth` parameter limits the depth of the tree, preventing it from growing indefinitely and ensuring that the search space remains manageable.

The `buildGameTree` method starts from the root node and explores all possible moves until it reaches a terminal state, which could be a win, loss, or when the specified depth limit is reached. For each valid move, it creates a child node representing the resulting game state and recursively calls itself to continue building the tree. The `isMax` parameter is crucial for the subsequent evaluation process in both the minimax and alpha-beta pruning algorithms, as it determines whether the current player is maximizing or minimizing.

```
    private void buildGameTree(Node node, int depth, boolean isMax) {
        int winner = Game.winning(node.getBoard().getBoardState(), gameN);
        if (depth == 0 || winner != 0) {
            // Terminal node reached or game over
            return;
        }

        for (int i = 0; i < node.getBoard().width; i++) {
            if (node.getBoard().isValid(i)) {
                Board newBoard = node.getBoard().getNewBoard(i, node.
                    getPlayer());
                Node childNode = new Node(newBoard, (node.getPlayer() ==
                    1) ? 2 : 1, i);
                node.addChild(childNode);

                buildGameTree(childNode, depth - 1, !isMax);
            }
        }
    }
```

2.1.3 Why Building a Game Tree is Useful

Building a game tree is a fundamental step in implementing algorithms like minimax and alpha-beta pruning because of exploration of possible moves. The game tree allows the algorithm to systematically explore all possible moves and their consequences. This exhaustive exploration ensures that no promising moves are overlooked, leading to a more informed decision-making process.

In summary, building a game tree provides a structured and systematic approach to decision-making in games, enabling algorithms like minimax and alpha-beta pruning to make informed and efficient choices, ultimately enhancing their performance in game-playing scenarios.

2.2 Minimax algorithm

Mini-max algorithm is one of the algorithms designed to reason with game decision trees using adversarial search. Adversarial search starts evaluating the tree bottom-up (depth-first) inspecting from the points of view of the different players. The agent it self is MAX that maximizes the move, while the opponent is MIN that minimizes the score. Each move from the player point of view is always the best move they can make in any situation.

2.2.1 Design

The leaf nodes get evaluated first with the simple heuristic while the parent node takes up the value of it's children. The root node is always the current state of the board that is from the MAX perspective. Node gets the node utility by propagating backwards the value from one of the leaf nodes. The same applies for any Internal nodes. So it makes it easy to implement this in code by using recursion.

2.2.2 Implementation

The output of Mini-max function is the best move they can make in the current situation by inspecting nodes with a given depth.

Input:

1. Node: Node Object
2. Depth: Integer
3. isMax: Boolean

The input is the root node holding useful information like board state, the move made in the state, value and children nodes. Depth is a number that indicates the depth of non-terminal leaf nodes. isMax is a true or false statement which is necessary to recursively call the function again later, but from the perspective of the opponent player.

```
private int minimax(Node node, int depth, boolean isMax) {
    int winner = Game.winning(node.getBoard().getBoardState(), gameN);
    if (node.getChildren().isEmpty()) {
        return heuristic.evaluateBoard(playerId, node.getBoard());
    }
    if (isMax) {
        int maxEval = Integer.MIN_VALUE;
        for (Node child : node.getChildren()) {
            int eval = minimax(child, depth - 1, false);
            maxEval = Math.max(maxEval, eval);
        }
        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;
        for (Node child : node.getChildren()) {
            int eval = minimax(child, depth - 1, true);
            minEval = Math.min(minEval, eval);
        }
        return minEval;
    }
}
```

2.2.3 Complexity

The time complexity of Mini-max is based on the size of the game tree it needs to explore. Without any pruning happening all the possible moves and game states need to be examined, the time complexity is exponential. It is typically expressed as $O(b^d)$, where: b is the branching factor, representing the number of possible moves or choices at each decision point in the game. d is the depth of the game tree, representing the number of levels of moves into the future that Mini-max considers.

2.3 Alpha-beta pruning algorithm

Almost all of the properties for Mini-max also stand for Mini-max with Alpha-Beta pruning except for few key difference. Same as for Mini-max, Mini-max with Alpha-Beta pruning generates the tree depth-first, left-to-right, and evaluates the tree bottom-up, propagating final values of nodes as initial estimates for their parent node. But with Alpha-Beta pruning the algorithm also prunes the nodes of the tree that it "knows" are worse then one of the already explored nodes in the queue.

2.3.1 Design

You can achieve Alpha-Beta pruning by saving the current best child node in the queue for each of the players. And then only explore nodes that have better value then the current best move value and by not exploring further any node that has a value smaller then the current best value in the queue.

2.3.2 Implementation

With the new implementation we have two new input values, namely, alpha and beta. Alpha will hold the current best move value in the tree for MAX player in the search tree, while Beta holds the best move value for the MIN player. The technique may exclude game tree nodes that will have no effect on the outcome by continually updating and comparing these values during the search. This reduces the number of nodes that must be investigated and makes the search more effective. We decide if we explore a node further in the tree by checking if beta value is larger then alpha value. If the beta value is smaller or equal then alpha, we prune this branch.

```
private int minimax(Node node, int depth, boolean isMax, int alpha, int
    beta) {
        ...
    if (isMax) {
        int maxEval = Integer.MIN_VALUE;
        for (Node child : node.getChildren()) {
            int eval = minimax(child, depth - 1, false, alpha, beta);
            maxEval = Math.max(maxEval, eval);
        // New lines for Alpha
            alpha = Math.max(alpha, eval);
            if (beta <= alpha) {
                break;
            }
        }
        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;
        for (Node child : node.getChildren()) {
            int eval = minimax(child, depth - 1, true, alpha, beta);
            minEval = Math.min(minEval, eval);
        // New lines for Beta
            beta = Math.min(beta, eval);
```

```
            if (beta <= alpha){
                break;
            }
        }
        return minEval;
    }
}
```

2.3.3 Complexity

The complexity for Alpha-Beta pruning algorithm in the worst case scenario can be the same as the one for the regular Mini-max algorithm. While in the best case scenario it can get much more effective resulting in a significantly reduced number of nodes that need to be explored. In such cases, the best case time complexity can approach linear time, denoted as $O(bd)$. Where again b is the branching factor and d is the depth of the game tree.

3 Results

N in a row	Board Size (Width x Height)	Depth	# Evaluations MM	# Evaluations AB	# Moves made
4	7x6	5	103,214	9,175	40
4	7x6	6	598,256	24,764	42*
4	7x6	7	4,396,112	89,044	28
4	8x8	5	277,532	19,260	64*
4	8x8	6	1,844,580	59,995	35
4	8x10	6	2,253,944	77,331	43
5	7x6	5	111,429	10,473	42*
5	7x6	6	708,912	33,881	42*
5	7x6	7	4,457,143	192,250	42*
5	8x8	5	296,163	21,562	64*
5	8x8	6	2,153,530	77,060	64*
5	8x10	6	2,593,293	94,133	80*

Table 1: Comparing different numbers of evaluations of the Mini-max algorithm and alpha-beta, based on different N in a row, board sizes (width and height), and depth, when Mini-max algorithm and alpha-beta are playing against each other. Also showing the number of moved made on the board

* symbol indicates that the board was filled and the game ended in draw

3.1 Interpretation of results

As we explained before in the complexity of Mini-max chapters, we can see that the evaluation amount increases exponentially as we increase the depth. When width is increased, in our case, the branching factor, we can see that the evaluation amount increases with it, but not as much as with the depth. Increasing N in a row also results in the increase of evaluations taken as sometimes might change the tree depth, because the leaf nodes are further away and you get less short tries when one of the players is close to winning. But all of this is true only for Mini-max without pruning.

If we look at the Mini-max with Alpha-Beta pruning we see that the increase of evaluations is not that steep. But we also could not exactly measure how steep is the increase,

because the amount of evaluations for Alpha-Beta has an element of luck. If the best move is found at the start when evaluation all the possible moves the pruning happens faster, while if the best move is the last move from all the checked moves, then no pruning happens in this case.

4 Discussion

It is very likely that there would be even more evaluations for Alpha-Beta algorithm, if we would have started checking the best move from the opposite side of the Mini-max without pruning algorithm preferred side. Because the best move at the start of the game is more likely to be next to the opponents move.

Besides the Alpha-Beta algorithm does not work optimally because of the way we implemented it. The pruning only happens when calculating the values not when making the tree. This means although it is still much more efficient then the Mini-max algorithm since it does much less calculation it still makes nodes in the tree that are not necessary.

5 Conclusion

In this project, we explored adversarial search algorithms, focusing on Mini-max and Alpha-Beta pruning in the context of "N in a Row." We emphasized the importance of efficient decision-making in games and introduced the `Node` class to construct a game tree to store all possible moves and evaluate these.
From our results the effect of the Alpha-Beta pruning on the run-time efficiency became clear. As expected the pruning has a big positive impact on the run-time efficiency in practice, although both algorithms share the same worst-case complexity.
In conclusion, our results confirm our before thought ideas about the effect of Alpha-Beta pruning. Because of the ability of Alpha-Beta pruning to not compute for non optimal game branches it is able to make decisions more quickly.

6 References

- Kwisthout, J. (2023). Game search [Slide show; Radboud Universiteit Nijmegen]. https://brightspace.ru.nl/d2l/le/content/431290/viewContent/2337508/View