COMP 424 Final Project Report
Colosseum Surivival

Taha Rhaouti (260974004), Reda Toumi (260912484)

December, 2023

1 Introduction

Game-playing AI algorithms serve as a great way to test the limits of AI. We've explored a multitude of techniques throughout this class, delving into concepts such as minimax, alpha-beta pruning, Monte-Carlo search tree, A*, heuristic evaluations, and iterative deepening, each offering unique insights into the complex interplay between computational efficiency and strategic decision-making within games and other applications, such as optimization. Here, we will be working on developing an AI agent that can play a game called Colosseum Survival.

Colosseum Survival is a 2-player turn-based strategy game in which two players move in an $M \times M$ chessboard and put barriers around them until they are separated in two closed zones. Here, M will be considered to be between 6 to 12 inclusive. Each player will try to maximize the number of blocks in its zone to win the game.

2 Motivation

Our goal is for an AI to be able to play this game and perform predominantly against a random agent, and consistently better than an average human player. Steps to achieve this would be to "explain" the rules to the agent, by making him know what moves are possible, and what moves are not. In addition, we will try to make the agent choose moves that have high probability leading to a winning game. We defined these probabilities using heuristic evaluations, and we will explain our choice of heuristics in the next section. Alongisde heuristics, we used a minimax search tree to improve our move choices, coming with a slight cost in computational efficiency. Because

we are restricted to 2 seconds runtime per turn, optimizing our code was crucial.

Achieving a balance between depth of search and computational constraints proved to be a central challenge. We aimed to optimize our agent's depth of search, employing sophisticated algorithms such as Linear Programming pre-processing, BFS in incorporating domain-specific heuristics to guide its decision-making process, all of this wrapped around by a 2-layer Mini-Max search tree with $\alpha - \beta$ pruning to improve its computational efficiency.

I believe the play quality of our agent is satisfactory, as the student agent can choose relatively moves bugless and seamlessly, in a competitive time, well below the 2 seconds limit. Though this agent winrate can be challenged, and our heuristics being not perfect, it solves our game-playing problem quite brilliantly, and can be translated to other chessboard games.

3 Agent Design

Here, we will be discussing the design of our agent, and the different techniques we used to implement it. We will start by covering the theoretical aspects, and move on the implementation details, then end with a quantitative analysis of our agent's performance.

3.1 Theory

Algorithm

The deterministic nature of the game and it being a 2-player instance allows us to use a minimax search tree with $\alpha - \beta$ to choose the best move.

Definition 3.1. ***Minimax*** *is a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case scenario. When dealing with gains, it is referred to as "maximin"—to maximize the minimum gain. Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty.*[1]

In this game, we will be using the version adapted to alternating moves games. In addition, we will be using the $\alpha - \beta$ variant of minimax, which

[1] Maschler, Solan, Zamir, Game Theory

should output the same answer, as they both use the same initial seach tree, but pruning some branches of the search tree throughout the execution of the algorithm, thus improving the computational efficiency of our agent.

The pseudo code for the algorithm we relied on in our implementation is:

Algorithm 1 $\alpha - \beta$ Minimax

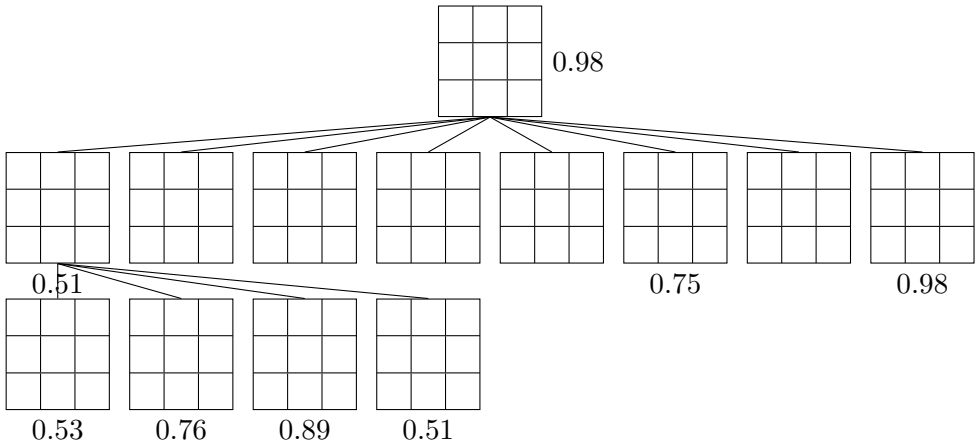
```
function AlphaBeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node then
        return heuristic(node)
    end if
    if maximizingPlayer then
        value ← −∞
        for child in node.children do
            value ← max(value, AlphaBeta(child, depth − 1, α, β, false))
            if value > β then break (* β cutoff *)
            end if
            α ← max(α, value)
        end for
        return value
    else
        value ← ∞
        for child in node.children do
            value ← min(value, AlphaBeta(child, depth − 1, α, β, true))
            if value < α then break (* α cutoff *)
            end if
            β ← min(β, value)
        end for
        return value
    end if
end function
```

Logic and Heuristics

We applied this algorithm on a search tree that contains at each level, all the possible moves for the player whose turn it is to play we computed using the `all_valid_moves` method. So each node in the tree is a valid move for a player, and each child of that node is a possible move by the other player after the node move is played.

Each leaf node is associated with a heuristic value, which is computed using the `combined_heuristic` method for a certain move/node. This yields a search tree where our $\alpha - \beta$ minimax algorithm is applicable.

An example of such a tree would looke like this:



Levels here are 0 (Max), 1 (Min), 2 (Max). The number of nodes at each level has the possible number of moves would generally decrease as barriers would be placed, thus restricting movement, here the head node would be the actual state of the chessboard, and its children the possible moves of our student agent, the following children in the second level would be the movements of the opponent, and so on... Here, we're assuming the opponent will pick at level 1 the move that will minimize the heuristic value from the leaves in level 2, and the student agent will pick in level 0 the move that is the maximum of the minimum picks in level 1. And the final algorithm will return the move corresponding to the maximum value in level 0.

The values in the leaves are computed using the `combined_heuristic` method, which is a weighted linear combination of the following heuristics:

1. **Chasing Heuristic**: This heuristic is used to minimize the distance between the student agent and the opponent. It represents an aggressive strategy, where the student agent tries to corner the opponent.

2. **Blocking Heuristic**: This heuristic tries to minimize the number of moves the opponent can make. This follows our ideology of the aggressive play-style, as limiting the opponent's moves will increase the chances of having more blocks in the final score. This heuristic is closely related to the Chasing Heuristic.

3. **Center Heuristic**: This heuristic tries to check the position after the

move relative to the center of the chessboard. This is based on the idea that if controlling the center of the chessboard, will usually lead to the agent controlling more blocks and in a defensive manner, the agent will be less likely to be cornered and have more valid moves.

4. **Endgame Heuristic**: This heuristic is used to check if the game is in an endgame state, where the agent will try to finish the game if it's in a winning position, and avoid the move otherwise.

3.2 Implementation

When it's the student agent turn, the student agent is given a state through calling the `step` method, which is comprised of the following attributes:

- **chessboard**: A $M \times M \times 4$ matrix representing the chessboard, where M is the size of the chessboard. For each position in $M \times M$, the last dimension is a vector of size 4 that shows whether a barrier is placed on that cell for the 4 directions.
- **my_pos**: our student agent position onn the chessboard.
- **adv_pos**: The position of the opponent on the chessboard.
- **max_step**: The maximum number of steps allowed in the game, which is $\lfloor \frac{M+1}{2} \rfloor$ here.

We start by finding all the possible moves our student agent can make given our state. This is done by calling the `all_valid_moves` method, which works as follows:

`all_valid_moves`: Set an empty list for the valid moves found. Then initalize a state queue that will have the initial position alongside the current # of steps taken so far. So state queue wil have $(my\_pos, 0)$ initially. Then, as far as the state queue is not empty, you take out the first element and check if the number of steps taken so far is less than the maximum number of steps allowed. Then check for all four possible moves that are derived from the four directions, whether the move is valid or not using the move validity conditions that were given. If it is valid, add it to the visited dictionary. Then, add the move to the state queue and increment by 1 the number of steps taken to get there. For all moves, we got in visisted, we check whether barriers were placed, and only consider the moves where there is no barrier

to be valid. This BFS queue algorithm will efficiently generate all possible moves given our initial state.

Now that we have a list of all possible moves, we can create a mirror list for heuristic value for each move. This is done by looping through the `valid_moves` list and calling

After

4 Quantitative analysis and results

5 Discussion

5.1 Advantages and Disadvantages

Our algorithm has many advantages:

- Minimax is a deterministic algorithm, so it will always give the same output for the same input. It is also complete, so it will always find a solution if it exists in finite time.
- Algorithm will yield an optimal solution if the heuristic is near-perfect, and assuming opponent plays optimally.
- Uses $O(k^2)$ space, where k is the maximum number of steps allowed in each move.
- Aggressive play-style by trying to corner the opponent and look for an endgame, can finish the game quickly if given the opportunity, which is in accordance with using depth= 2.

However, it also has some disadvantages, such as:

- Our heuristic being far from near-perfect, can lead to sub-optimal moves.
- The algorithm doesn't perform well against a random agent, due to the randomness of the agent's moves, and the fact that minimax's pessimistic nature assumes the opponent will play optimally.
- Contrary to other game-playing algorithms, such as MCTS, minimax doesn't utilitize a random component and/or backpropagation, which

can be useful in some cases. If you go down a wrong path, you will keep going down that path, and not explore other paths.

- Against opponents employing different tactics or defensive play styles, our agent might struggle to adapt and perform suboptimally.
- The runtime is still the same same as simple Minimax, which is $O(k^4)$, that is bad. Though with pruning, the amortized runtime is $O(\sqrt{k^4}) = O(k^2)$. This is still quite a large runtime, especially for the cases where the chessboard is 12 by 12.

5.2 Alternative approaches

An alternative approach to implementing our agent would be to tailor our heuristic functions first. This can be done by spending more time on choosing the appropriate weights for each heuristic subfunction in the `combined_heuristic` method. In addition, adding an attribute that checks how far are we into the game, and changing the weights accordingly will make our agent adapt better depending on the game state. Because our heuristic function is a linear combination of the subfunctions, we can use Linear Programming to find the optimal weights. Now that we have a heuristic function that is quite reliable, and can be qualified as near-perfect, we can use it in a simpler algorithm, either by directly returning the best move without any depth. Or implement an A* search using a heap, which will give us the optimal move given sufficient time. That was our initial approach, but we decided to focus less on the heuristics and more on the algorithm itself, which we think is more balanced and interesting. Also the A* algorithm would be more computationally expensive, as it would require a heap, and the runtime would be $O(k^4)$, which is not ideal given our runtime limit.

Another approach would be keeping our heuristic implementation, but changing the algorithm. We can use a Monte-Carlo Tree Search, which is a randomized algorithm, and can be used to explore more paths. MCTS navigates the search tree by iteratively choosing promising nodes at each depth until it reaches the tree's frontier (leaf nodes). This selection process relies on a tree policy utilizing Upper Confidence Trees (UCT). MCTS doesn't require an explicit evaluation function or domain-specific knowledge, making it applicable to a wide range of problems without intricate heuristic design. Unlike minimax, MCTS doesn't depend on fixed search depths, enabling it to adjust dynamically and allocate computational resources where they are most effective in the search tree. We thought about implementing this algorithm, but we decided otherwise after realizing that our heuristics were

already strong enough (92% winrate of heuristic only against random agent) and that implementing a MCTS would come with negligible improvements in performance compared to a simpler minimax that is easier to implement and doesn't come with the delicate tuning of parameters that MCTS requires.

5.3 Improvements

One thing we realized about our algorithm, is that for each move, we generate the search tree of depth 2 given the game state. This is not optimal, as depending on the move we make, we would have to re-generate the same search tree up to depth 1. To optimize the algorithm's performance, we could consider implementing a caching mechanism to store and reuse the subtree information from previous moves. By doing this, you can avoid regenerating the same subtree multiple times. This could even potentially let us go to depth 3 and stay within the 2 seconds runtime limit, which would cover a larger search space and yield substantial improvements in performance.

In terms of our heuristics, we could improve them by adding more sub-functions, such as a defensive heuristic, which would be the opposite of the chasing heuristic, and would try to maximize the distance between the student agent and the opponent. The aggressive and defensive heuristics would then have adaptive weights, depending on the game state. This would make our agent more versatile and able to adapt to different play styles. In addition, we could consider a heuristic that tries to know what the opponent heuristic logic is, and try to counter. Such a heuristic would use a pattern recognition dataset to predict the opponent's playstyle and next moves would be a good idea, giving us a headstart and a high chance of winning, though this kind of dynamic heuristic would be quite complex.

6 Conclusion

We were able to build an AI agent through Minimax with Alpha-Beta pruning and a blend of heuristic evaluations. Our agent showcased a penchant for aggressive gameplay, aiming to corner opponents while strategically limiting their moves. Yet, this exploration revealed areas ripe for improvement. Balancing computational constraints with depth of search, refining heuristics for more nuanced evaluations all could still be enhanced. In essence, this project emphasizes the iterative nature of AI development. While our agent demonstrated promise, the pursuit of sophistication in heuristics and algorithmic enhancements remains an ongoing endeavor, fueling future advancements in game-playing AI.

Bibliography

Maschler, Michael; Solan, Eilon; Zamir, Shmuel (2013). Game Theory. Cambridge University Press. pp. 176–180. ISBN 9781107005488.

Charles Hermite; letter to C.W. Borchardt, "Men of Mathematics", E. T. Bell, New York 1937, p. 464.

Simmons, George F.. Calculus Gems: Brief Lives and Memorable Mathematics. United States, American Mathematical Society, 2020.