JUNGLE GAME ADVERSARIAL SEARCH METHODS FOR TWO PLAYER BOARD GAMES

Jakub Głatki[1], Claudio Savelli[2]

[1]up202101824@up.pt

[2]up202111375@up.pt

I. INTRODUCTION

The implemented program revolves around the jungle board game. The game is about two players, having eight animals each, trying to get into the enemy's base, called "dojo". Each animal has different power, being able to eliminate opponent's animals if their power is equal or smaller than their own. The game is also varied by other elements, traps, and water, which has a different interaction with different animals. The complete rules of the game are attached in the folder.

In our implementation, the game is displayed in the console, in text form. There are going to be four game modes: player vs player, player vs computer, computer vs computer and the research mode. The computer moves are implemented using a MiniMax algorithm, with alpha-beta cuts. The user has an option of choosing different difficulties for the computer. The difficulty is chosen using different evaluation functions. Besides that, it is also possible for the user to choose the depth level of MiniMax algorithm.

II. SEARCH PROBLEM

The search problem contains four different components, which will be described below:

State representation: It describes how the state of the game is represented during any given moment of duration of the game:

- **Board [7x9]**: with cells of different kinds - grass, water, trap or dojo. Grass and water are neutral, while traps and dojo belong to specific players. Each cell might have an animal, which belongs to one of the two players.
- **Eight animals for each player**: with different power and set of moves (animal's specific move sets are described in attached document).
- **Player**: who makes current move.
- Chosen depth level for player (only relevant if the player is a bot).
- Chosen difficulty level for player (only relevant if the player is a bot).
- **Information**: if player is human or bot.

Initial state of the game: It is always the same, as the game does not have any random elements:

- Board with animals on their starting positions.
- Player 1 to move.
- Chosen game mode (player vs player, player vs computer, computer vs computer and research mode).
- Information about the player's difficulty and depth level in case of them being bots.

Objective state: It represents information about the end state of the game, showing the winner:

- *def testFinalGame(self, p1: Player, p2: Player, board: Board,)*: returns True or False in case of win of one of the players and notifies who has won the game.
- *def noPossibleMoveForPlayer(self, player: Player, board: Board)*: returns True in case the player has no valid moves, which makes him loose the game.

Operators: They are being used for the game to work as intended:

- *def calculateMove(self, animal: Animal, board: Board, direction: str)*: after choosing the animal to move and the direction of the move, this function returns the movement that the animal will perform on the chessboard. The method uses other two operators to check if move in chosen direction is valid.

- *def killAnimal(self, board: Board, endingx: int, endingy: int)*: if there is opponent's animal in the landing square of a moving animal, and the moving animal is able to capture it, this method removes the opponent's animal from the board and from the game state.
- *def moveAnimal(self, animal: Animal, board: Board, endingx: int, endingy: int)*: Change the position of the moving animal on the board.
- *def isValidStartingPoint(self, player: Player, board: Board, x: int, y: int)*: Checks if the chosen starting point has an animal belonging to a player who makes move.
- *def isValidEndingPoint(self, animal: Animal, board: Board, endingX: int, endingY: int, startingX: int, startingY: int)*: Checks if the move is valid, by checking if the chosen field is next to the starting point (or across the water in case of tiger and lion), does not go into water (expect in the case of mouse), does not go into own dojo, and does not have an animal with a power stronger of the moving animal (except it stays in the trap).

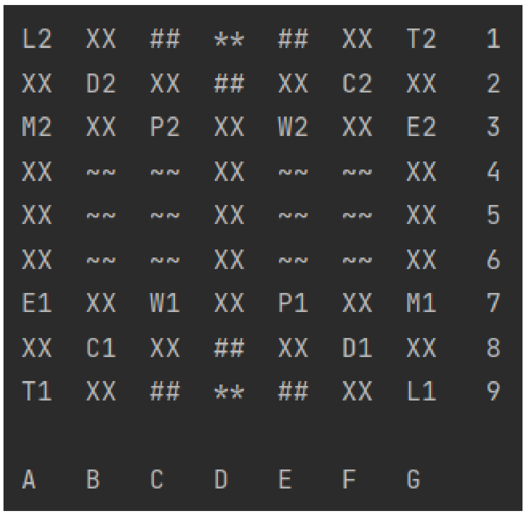
III. IMPLEMENTATION DETAILS

The approach of the program implementation was to divide it into 5 steps. At the beginning the search problem was formulated, to establish what components will be necessary for the project to work correctly. Then the implementation of the game for human players was made. Afterwards a MiniMax algorithm was implemented, with a very basic evaluation function. The last step was about developing more complex evaluation functions, testing them with different depth and values, to establish the best one and optimizing the code.

The program was implemented in the Python programming language, with PyCharm as an IDE, and git version control. The pragmatisms of object-oriented programming were used, with the usage of model-view-controller project pattern. The 'view' part is responsible for communication with the player, showing him the board, all the menus, and letting him control the game. In the 'model' all classes responsible for the representation of game state are stored, starting from *State()*, which stores *Players()*, and *Board()*. In the *Player()* class there is an information of *Animals()* which belong to them, and the list of their *LastMoves()*, to not repeat them. *Board()* contains a matrix of *Cells()*, which make the board, and puts the *Animals()* on their starting positions. The main class of the 'controller' part is *GameController()*. It takes information which game mode has been chosen, the other parameters necessary to run the game, and starts a game loop, which ends when one of the win conditions is reached (which is checked by the *EndingGameController()*).

The game loop is different depending on the chosen game mode. When the game mode which involves computer playing is chosen the program goes into *MinimaxController()* method. A MiniMax algorithm is being made there, and a chosen move is returned. MiniMax algorithm uses *EvaluationFunctionController()*, which uses one of the four evaluation functions, that will be described later.

The program was implemented with a text interface. *Figure 1* represents the board it its initial state:



```
L2  XX  ##  **  ##  XX  T2   1
XX  D2  XX  ##  XX  C2  XX   2
M2  XX  P2  XX  W2  XX  E2   3
XX  ~~  ~~  XX  ~~  ~~  XX   4
XX  ~~  ~~  XX  ~~  ~~  XX   5
XX  ~~  ~~  XX  ~~  ~~  XX   6
E1  XX  W1  XX  P1  XX  M1   7
XX  C1  XX  ##  XX  D1  XX   8
T1  XX  ##  **  ##  XX  L1   9

A   B   C   D   E   F   G
```

Fig 1. Initial state of the board

The animals are represented by the first letter of their name and number of the player they belong to, and cells are represented by different symbols: ** for dojo, ## for traps, XX for grass and ~~ for water.

IV. COMPLEXITY OF THE GAME

The complexity in the Jungle game (just like the complexity of a traditional chess game) depends on two dimensions:

Decision complexity: The difficulty to choose the correct movement. Each player can move a piece in at most 4 directions and each player has at his disposal 8 pieces at most. For this reason, the number of moves possible to select is, at most, 8 * 4 = 32.

Wanting to make a quick comparison with the game of chess, there the moves that a player can make are, on average (there has been instead considered the worst possible case), ~35. Therefore, it is possible to realize how, from the point of view of the complexity of the decision, there is a simpler problem, an element that will help during the progress in the study.

Space complexity: Number of possible positions in the search space.

On our chessboard, it is possible to have, at most, a number of positions equal to:

$$N_{moves} = D_{(51,16)} + D_{(51,15)} * D_{(12,1)} + D_{(51,14)} * D_{(12,2)} =$$
$$= \frac{51!}{(51-16)!} + \left(\frac{51!}{(51-15)!} * \left(\frac{12!}{(12-1)!}\right)\right) + \left(\frac{51!}{(51-14)!} * \left(\frac{12!}{(12-2)!}\right)\right) \simeq 10^{26}$$

This number is not totally exact, since it must be considered that some positions cannot be achieved, but it makes the idea with the order of magnitude. The equation for the calculation of the possible arrangements is made up of three elements because it also takes into consideration the possibility that none, one or two mouses are in the water. Again, to make a comparison with the classic game of chess, in the classic game the number of possible positions is between 10^{43} and 10^{50}.

V. BRIEF DESCRIPTION OF THE MINIMAX APPROACH

The MiniMax research is based on the minimax theorem, proved by John von Neumann in 1928.

It is based on the principle of minimizing the probability of defeat by considering the worst possible scenario. When dealing with gains, it is referred to as "maximin" — to maximize the minimum gain. Calculating the maximin value of a player is done in a worst-case approach: for each possible action of the player, there is a check of all possible actions of the other players and determine the worst possible combination of actions — the one that gives us the smallest value. Then, it is determined which action can be taken to make sure that this smallest value is the highest possible.

Therefore, with this approach, the scenario is considered, in which our opponent will always make the best possible move and not a random one, and starting from this principle it is analysed, through a research tree, the combination of moves that leads us to reach the maximum value in our game.

The theorem holds for any zero-sum game. A zero-sum game is defined as a game in which a player's gain or loss is balanced by the gain or loss of another player, in fact, by adding together the gains of the two players there will be zero-sum. Jungle Checkers is a zero-sum game for two players: only one player can win and with each move, a player can increase or decrease his distance from victory, going to gain or lose value. This change in proximity is inversely proportional to the possibility of the opponent winning the game.

Obviously forming a tree that calculates all the possible continuations of a game even going to end in depth that tends to infinity is computationally impossible, which is why there must be some compromises.

Having a limited depth of our tree available, it must be ensured that the algorithm, through an evaluation function, can calculate the current value of that position on the board. Therefore, once a fixed depth of the search tree is reached, instead of continuing to go deeper the evaluation function is called, which gives us the current estimate of the value of our current game situation. Obviously, it is important to write the evaluation function correctly so that our algorithm can get close to winning.

In the drawing, the black moves are those made by our opponent, while the white ones are ours. The goal of black is to always choose the continuation that leads us to the worst possible solution (considering that the player will play perfectly), while the goal of white is to minimize, as already said, our chance of defeat. For this reason, MiniMax algorithm got its name from the fact that one player wants to Mini-mize (black circles) the outcome while the other tries to Max-imize it (white circles).

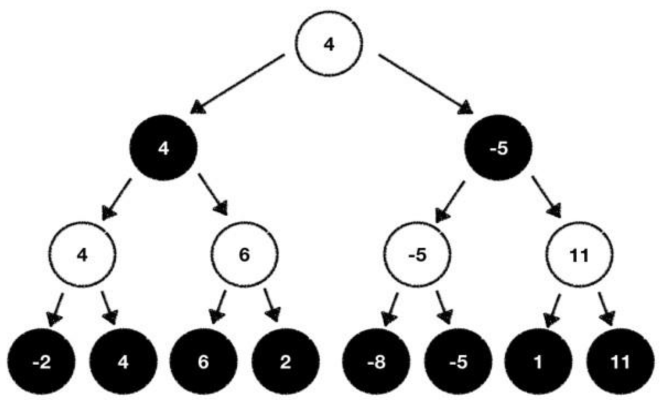


Fig 2. A min- max tree

Below there is the pseudocode of the Minimax algorithm:

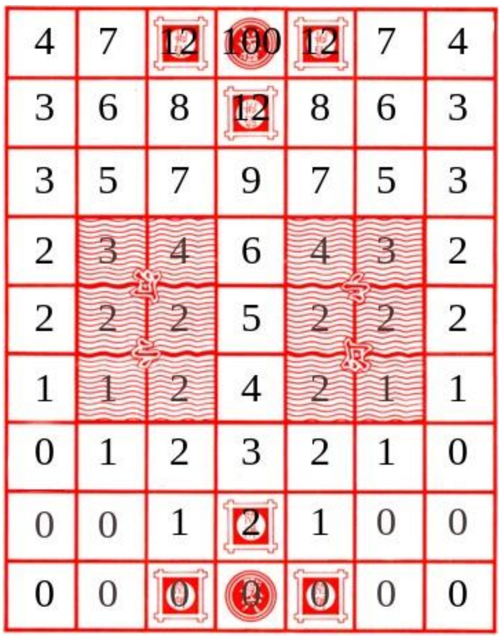
```
minimax function (knot, depth)
    IF node is a terminal node OR depth = 0
        returns the heuristic value of the node
    IF the opponent has to play
        $\alpha := +\infty$
        FOREACH child of node
            $\alpha := \min(\alpha, \text{minimax}(\text{child}, \text{depth}-1))$
    OTHERWISE we have to play
        $\alpha := -\infty$
        FOREACH child of node
            $\alpha := \max(\alpha, \text{minimax}(\text{child}, \text{depth}-1))$
    return $\alpha$
```

VI. EVALUATION FUNCTION

Defining a good evaluation function is crucial for achieving good result. To do it manually it is necessary to know the game very well and to be able to give a precise evaluation of all the pieces that player has in the field in order to define a function that brings good results.

During the development process the evaluation function was constantly evolving, following the results of various tests. At the beginning, the function only takes into consideration the value of the pieces on the field and the arrival at the opposing dojo. The animals were given a value according to their power, with some adjustments: pieces with special movement got additional value.

The second step was to give a value to each field on the board. When an animal belonging to the player is on a cell, the evaluation function reward him with the value of the cell. The valuation of cells was promoting aggressive moves, giving bigger value to the fields closer to the opponent's dojo, with additional value given to central ones.



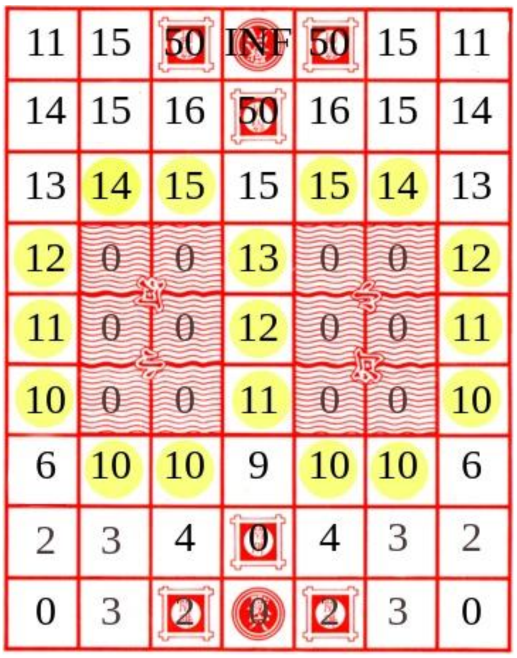
4	7	12	100	12	7	4
3	6	8	12	8	6	3
3	5	7	9	7	5	3
2	3	4	6	4	3	2
2	2	2	5	2	2	2
1	1	2	4	2	1	1
0	1	2	3	2	1	0
0	0	1	2	1	0	0
0	0	0	0	0	0	0

Fig 3. First evaluation matrix for player 1

Afterwards each animal got its own evaluation matrix. Matrixes were promoting more the position of stronger animals, as well as taking into consideration their special traits like availability of jumping over the water of tigers and lions. Also, the value of animals in comparison to the value of fields was increased, to put more emphasis on owning the animals. The newly made matrices were also taking into stronger consideration the defensive aspect of the game, giving additional value to the fields next to player's own dojo. For the last evaluation function, some ideas from the matrices proposed by J. Burnett [5] were taken; he also gave each animal its own value matrix, with valuations of fields varying from 0 to 50. There were used some interesting approaches, like making the panther focused on defending the central aisle, as only three opponent animals can capture it, or sending wolf, cat, and dog to defend the players traps.

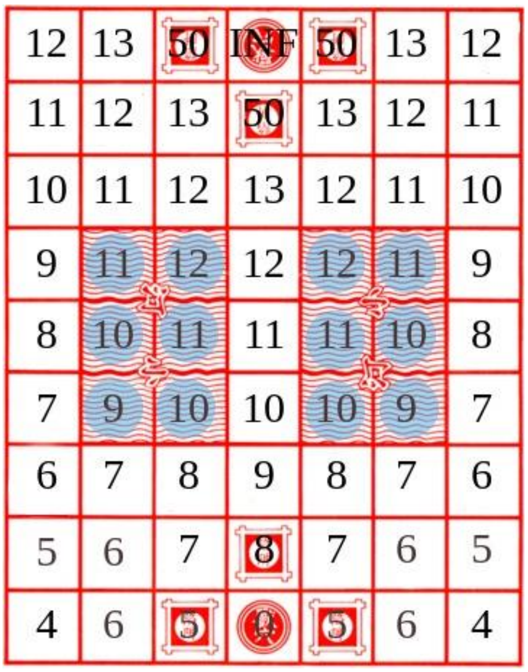
Evaluation matrices done for two animals are attached: the lion (*Figure 4*) and the mouse (*Figure 5*).

As showed, with a unique evaluation matrix for each animal it is possible to push a bot to use in the best way the individual special abilities of each animal. The lion will have greater value in a position adjacent to the water since it will be able to jump while the mouse will have greater value if in the water because it is able both to block the opponent's jumps and to defend itself naturally as no other animal can enter that cell.



11	15	50	INF	50	15	11
14	15	16	50	16	15	14
13	14	15	15	15	14	13
12	0	0	13	0	0	12
11	0	0	12	0	0	11
10	0	0	11	0	0	10
6	10	10	9	10	10	6
2	3	4	0	4	3	2
0	3	2	0	2	3	0

Fig 4. Lion evaluation matrix for player 1



12	13	50	INF	50	13	12
11	12	13	50	13	12	11
10	11	12	13	12	11	10
9	11	12	12	12	11	9
8	10	11	11	11	10	8
7	9	10	10	10	9	7
6	7	8	9	8	7	6
5	6	7	8	7	6	5
4	6	5	0	5	6	4

Fig 5. Mouse evaluation matrix for player 1

Furthermore, it has been decided to add the following function in our last evaluation function: *def isMenaced(self, animal: Animal):* it allows us to see if one of our animals, at the end of that movement, is threatened by an opposing animal. In this case, since a player is giving the possibility to our opponent to make a "free" capture (which can sometimes be advantageous but not always), there is given a penalty to the animal considered proportional to its current value on the board.

VII. ALPHA-BETA PRUNING

Analysing the whole tree can be expensive computationally, given that, in any case, analysing every possible move with a tree of a certain depth exponentially increases the number of moves to be analysed. For this reason pruning is used, which is, stopping the search in-depth before the necessary if it is being realized that the path, which is being followed is certainly not useful for finding the best move. The pruning technique used in this project is called alpha-beta.

The alpha-beta pruning is explained through this example, where the previously generated tree is being reused.

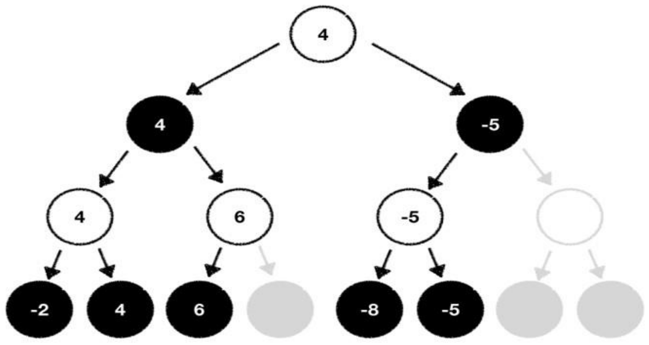


Fig 6. Min- max tree with alpha-beta pruning

Let's start by analysing the tree from the left. There will be analysed each leaf individually to obtain the desired result, so in order, the algorithm will go to "discover" the following leaves: -2, 4, the white takes the maximum (4). After that w continue going to the right and find the leaf with the value 6. At this moment pruning comes into action since the first black branch will select the lowest number knowing all the possible continuations, and seeing that on the right branch the algorithm already discovered have a 6 (so his opponent will go certainly in that direction), then it is useless to continue to analyse this branch since it will surely have a value ≥ 6 and will therefore select the move that brings it to the left.

Now let's move on to the sub-tree on the right, find the leaf -8 and -5 in order, so the algorithm knows that white will take the maximum (-5). Here alpha-beta pruning comes into action again. In fact, given that white will always select the path that takes him to the maximum possible considering the opponent's best move, it is already possible to see how black will lead us to perform a series of moves that leads us to have a final value that will be at least ≤ - 5, therefore certainly less than 4 that the algorithm would instead get by taking the other branch. Therefore, at this moment, our research is finished and, thanks to pruning, the algorithm had to analyse only 11 of the 15 nodes (just a bit over 2/3 of the full tree) to choose the best path following the MiniMax algorithm (The path that reaches the final node 4).

Alpha-Beta pruning got its name from the parameters alpha and beta which are used to keep track of the best score either player can achieve while walking the tree. There can therefore be seen how, using this approach, the work of our computer has been reduced by cutting the useless branches and thus obtaining a result in a shorter time.

Below there is the pseudocode of the MiniMax algorithm with alpha-beta pruning:

```
FUNCTION alpha_beta
(node, depth, α, β, maximize)
    IF depth = 0 O node is terminal
        RETURN heuristic value of the node
    IF maximizes
        v: = -∞
        FOREACH child of the node
            v: = max (v, alpha_beta (child, depth - 1, α, β, FALSE))
            α: = max (α, v)
            IF β ≤ α
                STOP THE CYCLE (* cut according to β *)
        RETURN v
    OTHERWISE
        v: = + ∞
        FOREACH child of the node
            v: = min (v, alpha_beta (child, depth - 1, α, β, TRUE))
            β: = min (β, v)
            IF β ≤ α
                STOP THE CYCLE (* cut according to α *)
        RETURN v
```

VIII. APPLICATION OF ALPHA-BETA PRUNING ON OUR APPLICATION AND CONSIDERATIONS

This general description of the MiniMax method with alpha-beta cuts is necessary for us to fully understand the following chapter, aimed at underlining how important it is to use these tools correctly. As already said, the cut is not always effective, but only when being in certain favourable situations in which its known that all the other branches to be analysed are useless for choosing the best path.

It is essential to order the tree in such a way that as much pruning as possible can be done. For this to happen, it is necessary to order the possible moves in such a way that those that are probably most advantageous are analysed first, so that when the algorithm is on the second part of the tree, it can make a cut that eliminates numerous nodes, saving in this way a lot of time.

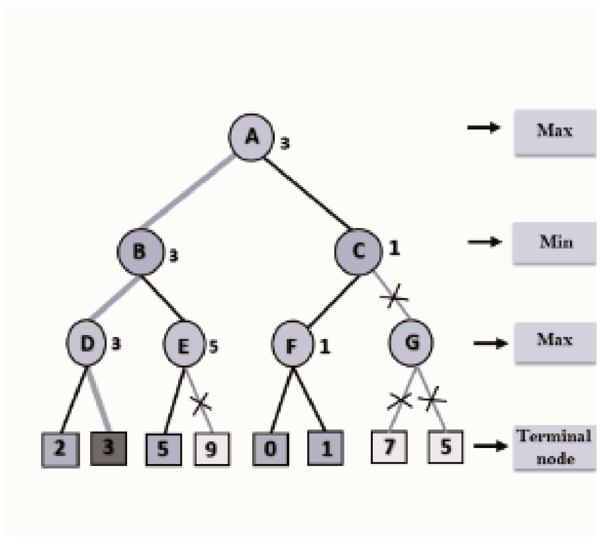


Fig 7. Importance of ordering in alpha-beta pruning

A simplified example can be seen in this case, in which it has been (although not optimally) ordered set of moves in such a way as to obtain the ones that are probably most advantageous first. In the following case, even if the result is not so satisfying since the tree is small, let's see how having more advantageous moves at the beginning allows us to eliminate the last branch on the right already knowing that taking into consideration the right subtree the algorithm certainly have a possible continuation that leads it to a more disadvantageous position.

After a few tests, very interesting results has been reached since with just one small modification it is possible to obtain a noticeable improvement in terms of times and cuts. Our goal was to try to have stronger moves at the beginning, and surely the strongest moves are obtained by eating an opponent's piece or when an important piece moves in a more favourable position. During the first tests it had been simply inserted (not thinking about pruning) the animals in increasing order of strength (Mouse, Cat etc.) but immediately it has been realised that the algorithm had some problems since it made few cuts and the average time for a move with depth 3 was approximately 6.7* seconds. After that, in an attempt to solve this problem, there appeared a thought that simply reversing the order of the animals (Elephant, Lion etc.) could at least be advantageous for the alpha-beta cut principles, since the strongest animals are also those who have more chances of eating opposing pieces or making stronger movements (e.g. jumping over the river) and the results were immediately evident as the algorithm went with this very simple change to an average of 2.2* seconds per move compared to the previous 6.7* seconds.

A game between two CPUs with difficulty 3 and depth 3 has an average number of 96* turns, which means that a game lasts, on average, 7.2* minutes less (10.7 minutes to only 3.5).

Another interesting consideration made was to put at the beginning of the queue all those moves that allowed the capture of an opponent's piece, but this did not lead to such profound results, so it has been decided to eliminate this additional function.

**Test carried out considering about 100 games.*

In the algorithm there is a second pruning (*lastMoves.isARecentMove(action)* *and* *MiniMaxLastMoves.isARecentMove(action)*) taking inspiration from some similar functions implemented in the game of chess. With this second pruning, the algorithm saves the last moves that the current player has made and the moves that have been analyzed within the call of the MiniMax function, in order to avoid generating branches that simply mirror variations of branches already generated. Therefore, within the game, there are 3 stacks that save the last moves, a personal one for each player and a third one that is created and deleted within the MiniMax algorithm for avoiding internal repetitions.

IX. IMPORTANCE OF DEPTH

Having a good evaluation function which is able to make the best possible move to give a value suitable for the current state is very important, in fact with a bad evaluation function our bot will always play the game badly, also going to analyse many possible continuations. On the other hand, the depth of the algorithm has at least as much of an importance.

Potentially it is possible to have an infinite depth of research, but in reality, it is necessary to make compromises in order to have a good move that analyses as many continuations as possible but with "short" times. Let's imagine playing a game of chess against a computer that needs tens of minutes to take a move, surely even if the move will have a high value, but ruining the sense of the game (without considering that in games such as chess every player has at his disposal a timer that once exhausted means game lost).

Therefore, as already said, there must be compromises, because on the one hand the machine has to thoroughly analyse the current state of the board (a chess grandmaster is said to be able to see up to 15/20 moves in front), on the other a maximum average time to maintain to make each move. For this reason, it is equally important to optimize our code as much as possible to allow the algorithm to go deeper while remaining within the established times.

To give a chess example, the level of the players is based on a value called ELO, numerous studies and researches have been carried out on the correspondence between the depth of a search algorithm and the corresponding strength of the bot. Below there are some interesting studies found in the research phase and some graphs taken from them.

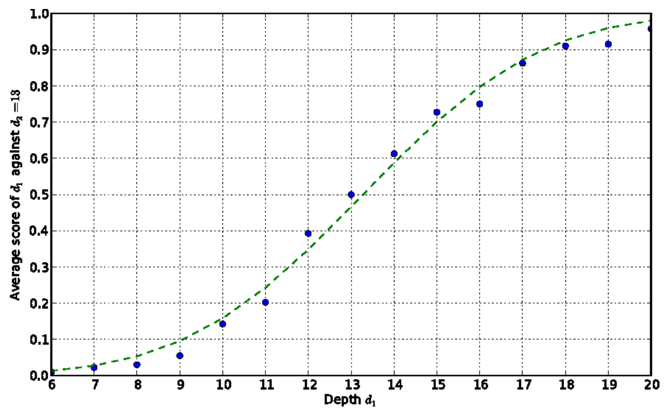


Fig 8. Average score over 200 games for each depth d1 against depth d2 =13

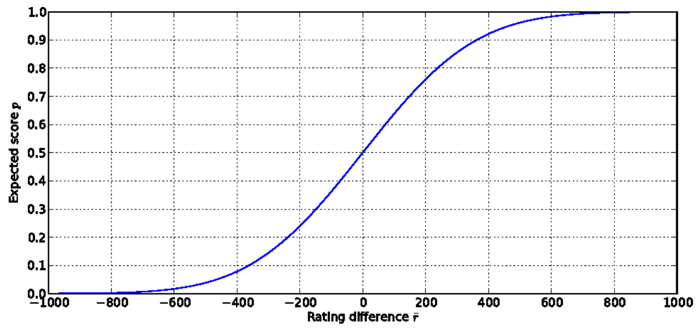


Fig 9. The ELO Curve

Depth d_1	$d_1 - 20$	$(d_1 - 20) \times \eta_{(20)}$	Strength (Elo)	95% conf. int.
20	0	0	2894	[2859, 2929]
19	-1	-66	2828	[2786, 2868]
18	-2	-133	2761	[2714, 2807]
17	-3	-199	2695	[2642, 2745]
16	-4	-265	2629	[2570, 2684]
15	-5	-331	2563	[2498, 2623]
14	-6	-398	2496	[2426, 2562]
13	-7	-464	2430	[2354, 2500]
12	-8	-530	2364	[2282, 2439]
11	-9	-596	2298	[2209, 2378]
10	-10	-663	2231	[2137, 2317]
9	-11	-729	2165	[2065, 2255]
8	-12	-795	2099	[1993, 2194]
7	-13	-861	2033	[1921, 2133]
6	-14	-928	1966	[1849, 2071]

Fig 10. Estimated strength of the engine at different search depths

During the project, a particular focus has been set on testing how both different levels are playing against each other, but especially on the fact how the depth affected computer’s performance. The main problem with tests of the whole game was the complexity of it. With the depth of 4 the algorithm was checking over 70000 possible board states with each move, taking on average over 70 seconds to calculate them. With an average of over 50 moves per game the depth 4 was barely possible to test, with any higher completely exceeding our computational power to test them. Due to that on the tests on the whole working game has been checked the results of games up to depth 4, on the same difficulty, on a probe of 100 games. Surely such a low depth and small number of games are unable to give completely adequate results, but there are still some conclusions which can be drawn from them.

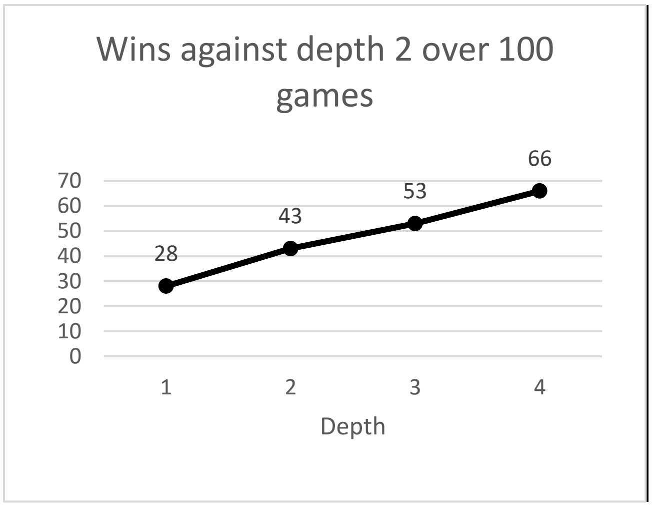


Fig 11. Wins against depth 4 over 100 games on the hard difficulty

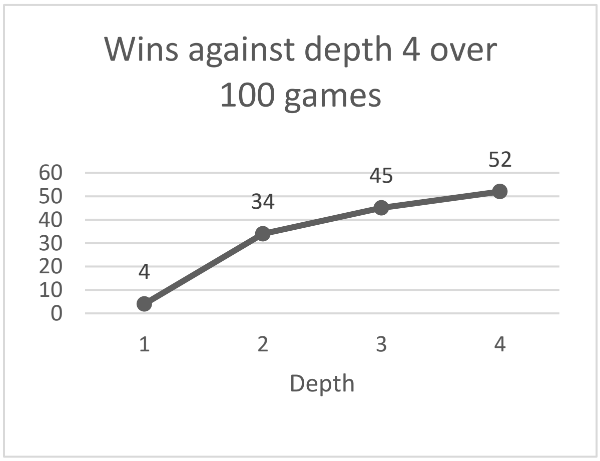


Fig 12. Wins against depth 4 over 100 games on the hard difficulty

Player 1				Player 2				Avg.nr.
Level	Depth	Score	Time	Level	Depth	Score	Time	of rounds
1	1	44	0,03s	1	1	56	0,04s	69
2	1	63	0,05s	1	3	37	2,02s	128
2	3	41	1,78s	3	1	59	0,07s	89
3	1	44	0,17s	3	1	56	0,21s	47
3	1	28	0,07s	3	2	72	0,89s	74
3	1	12	0,16s	3	3	88	2,65s	34
3	1	4	0,12s	3	4	96	25,14s	35
3	2	43	0,88s	3	2	57	0,91s	42
3	2	47	1,15s	3	3	53	1,98s	73
3	2	34	1,10s	3	4	66	22,87s	37
3	3	52	2,24s	3	3	48	2,15s	60
3	3	45	2,45s	3	4	55	26,29s	43
3	4	52	19,45s	3	4	48	21,17s	75

Fig 13. Comparison of different scores against different levels.

The charts presented in *Figure 11*, *Figure 12* and table in *Figure 13* show that with the bigger difference of the depth the bigger difference in the scores is. The games played on the similar depth have similar score, there is not much difference between player on depth 2 and depth 3, each of them could beat the other. With the games against depth 1 the importance of it is more visible, when it was able to win just 4 games out of 100 against depth 4, and 12 against depth 3. It was expected that, at least against depth 4, the number of wins would be 0, but apparently the difference between them is not big enough to guarantee a win for depth 4. The hypothesis for that is that there are some scenarios, where the interest in just the next move can provide the win, also due to suboptimal moves, while the MiniMax algorithm assumes that the opponent will be playing the game perfectly. Also, the fact that the implemented evaluation functions are not perfect does not allow computer to play perfect games. In individual game situations (as showed in the presentation or the attached video), our computer can evaluate the state and choose the best continuation. The depth of chess showed that to achieve really good results it is not enough to test things on depth of 4, much bigger values are necessary for that.

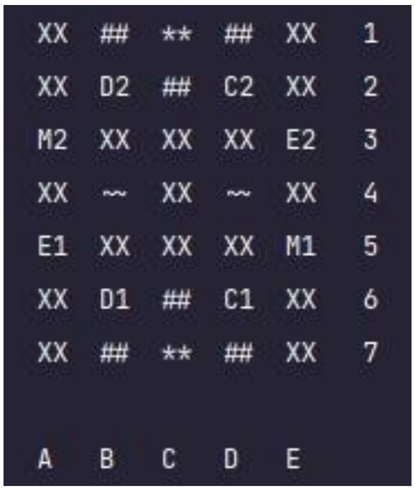
In fact, playing chess at depth 4 is far from having a good chess player. Below it is presented a player's depth-level correspondence [9]:

- Depth 4: Newbie Player
- Depth 8: PC, Good Human Player
- Depth 12: Deep Blue, Kasparov

In addition, there is a fundamental component that those who have played chess will have already understood by reading these lines, which unfortunately is not easy to interpret through an evaluation function, especially because it is not possible to afford a depth of 10/15 nodes that could somehow bring us to solve, at least in part, this problem: the gameplan.

Our computer does not have an overview that allows it to set the style of play according to the situation, as already said the aim is to create a rating function that favours an offensive game, the problem is that this offensive game is always the favourite, both when it has value, but also when our opponent begins to build a game state in which he begins to threaten our dojo from multiple points. Unfortunately, the large size of our board combined with the insufficient depth of search for the limits of the MiniMax algorithm means that our computer suffers when it has to change its style of play to defend itself from long-term attacks or is in any case unable to create gameplans.

To try to solve this problem a smaller board with fewer pieces has been created. It was supposed to make the game less complex by decreasing both decision and game complexity, with less possible board states to calculate, and fasten the testing process. The new board was made of 5x7 matrix, instead of 7x9, and had 4 animals for each player, instead of 8.



```
XX  ##  **  ##  XX   1
XX  D2  ##  C2  XX   2
M2  XX  XX  XX  E2   3
XX  ~~  XX  ~~  XX   4
E1  XX  XX  XX  M1   5
XX  D1  ##  C1  XX   6
XX  ##  **  ##  XX   7


A   B   C   D   E
```

Fig 11. Little board

The smaller board helped with resolving the problem but was still not able to go as deep as described before chess engines. For the first move of depth 7 computer was checking over 1,800,000 board states, which took it over 11 minutes. *Figure 15* and *figure 16* show what were the scores with bigger depth difference than the ones in *Figure 11* and *Figure 12*. From them it is clear that depth increase equals scores increase, especially considering that depth 1 was not able to win a single game against the opponent with depth 6 onwards.

With the new board is now possible to carry out a greater number of tests at greater depth in less time, also allowing us to more easily analyse the different game situations created.

The algorithm went from an average of around 4000* nodes analysed per round in 2.2* seconds to just 1600* nodes analysed per round in 0.37* seconds.

In addition, the games are also shorter, since it goes from 96* to 30* rounds per game. This means that, on average, a game between two opponents of the same level with a large board need 3.5 minutes with 384,000 nodes analysed, the small instead 11 seconds with only 48,000 nodes analysed.

**Test carried out considering about 100 games and a bot with depth 3 and difficulty 3.*

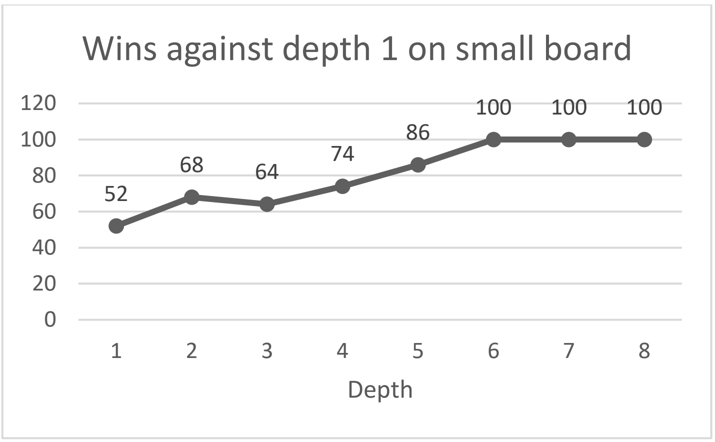


Fig 15. Wins against depth 1 over 100 games on the hard difficulty, small board

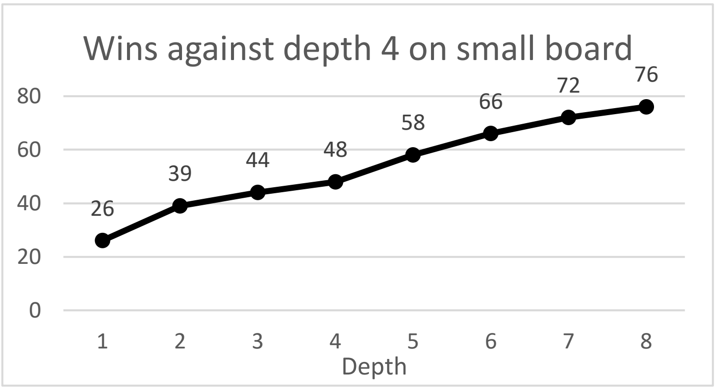


Fig 16. Wins against depth 1 over 100 games on the hard difficulty

Player 1				Player 2				Avg nr.
Level	Depth	Score	Time	Level	Depth	Score	Time	of rounds
3	1	48	0,01s	3	1	52	0,01s	22
3	1	32	0,01s	3	2	68	0,17s	17
3	1	36	0,01s	3	3	64	0,43s	20
3	1	26	0,01s	3	4	74	2,32s	25
3	1	14	0,01s	3	5	86	7,41s	17
3	1	0	0,01s	3	6	100	44,25s	20
3	1	0	0,01s	3	7	100	273,50	18
3	1	0	0,01s	3	8	100	352,40	15
3	2	42	0,17s	3	2	58	0,17s	20
3	2	40	0,17s	3	3	60	0,45s	53
3	4	61	2,19s	3	2	39	0,17s	24
3	4	56	2,15s	3	3	44	0,46s	25
3	4	52	2,49s	3	4	48	2,43s	22
3	4	42	2,81s	3	5	58	7,88s	43
3	4	34	2,91s	3	6	66	45,01s	24
3	4	28	2,24s	3	7	72	289,03	19
3	4	24	2,29s	3	8	76	376,88	28

Fig 17. Comparison of different scores against different level on small board

Now that is possible to do more in-depth tests, it is possible to better appreciate the graphs and see how important the in-depth analysis is in this type of problem. The player of depth 4 was achieving similar results against the one with depth from 3 to 5, but for both the lower, and higher values the variance of wins was much higher.

In fact, now it is possible to appreciate from the graph a curve that recalls (even vaguely) the same one previously proposed by the chess ELO.

X. WHAT'S NEXT

We have made a first general view of the problem, analysing a possible solution. During our study, we discovered several interesting changes/improvements to be made to our code and our approach to the problem.

Therefore, even if we have not had the opportunity because of available time to go to deepen and thoroughly test these roads, we have considered interesting to insert what are the next steps we intend to take to deepen the problem and improve our software.

There will be briefly described the five possible improvements that most impressed us, and to which we will certainly give further analysis over time.

a. NEGAMAX

Negamax is a small variation of the minimax algorithm which is based on the properties of zero-sum games with two players.

By definition, the position value of player A in a certain game is the negation of the position value of player B. Thus, the moving player will look for a move that maximizes the negation of the position value resulting from the move: this position of the successor must have been evaluated by the opponent. The veracity of this statement is maintained regardless of whether A or B has to move. This means that a single calculation can be used to value all positions. This is a simplification of the minimax, which requires A to choose the move with the highest sequence value and B with the minimum.

Therefore, to summarize, Negamax is a simplification of the MiniMax code following the following mathematical property:

$$\max(a, b) = -\min(-a, -b)$$

The latter makes the code much cleaner, faster to compile and easier to read. Below there is the pseudocode of the Negamax algorithm:

```
function negamax (node, depth, α, β)
    if the node is a terminal node or depth = 0
        return heuristic value of the node
    else
        FOREACH child
            α: = max (α, -negamax (child, depth-1, -β, -α))
            {what follows, if verified,constitutes alpha-beta pruning}
            if α ≥ β
                return β
        return α
```

b. PARALLEL COMPUTING

Parallel computing is the simultaneous execution of the source code of one or more programs (divided and corrected adapted) on more microprocessors or more cores of the same processor to increase the computing performance of the processed system.

As the MiniMax algorithm is naturally set, it would be possible to implement the parallel calculation to this problem in such a way as to be able to simultaneously analyse more subtrees taking less time to arrive at the end of the analysis.

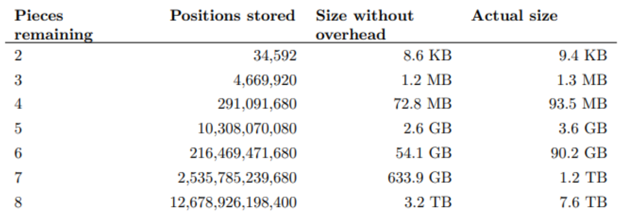
c. ENDGAME DATABASES

Just like in chess, it is possible to build and save a database within the program that saves all possible moves when there are few pieces on the board, to avoid going too deep into the search tree.

It is possible to save, occupying only a little memory, all the possible best continuations for both players considering the current state. It allows not only to have a stronger endgame phase on the part of the machine but allows it, once it has ensured that on the board there are a number of pieces ≤ N (where N is the maximum number of pieces in the states analysed on the database) to call the database and take the final result directly from it. Obviously, due to the discourse made previously on complexity, it is not possible to build a database considering the initial situation of the game at the moment, but it is possible to do it for the later stages.

By decreasing the number of pieces on the field (and therefore the possible combinations of states and the relative complexity) it is possible to reach a situation in which analysing and saving all the continuations becomes possible both at the computational level and at the level of memory space to be reserved for the database.

In one of the papers [4], the calculation was carried out that connects the number of missing pieces with the weight of the relative database in terms of space (the board he used is smaller and the total number of pieces on the field is only 8 and not 16, however, the table still gives the idea at least at a qualitative level, unfortunately for a matter of time it was not possible for us to carry out the following calculation considering the state of a real Jungle chessboard).



Pieces remaining	Positions stored	Size without overhead	Actual size
2	34,592	8.6 KB	9.4 KB
3	4,669,920	1.2 MB	1.3 MB
4	291,091,680	72.8 MB	93.5 MB
5	10,308,070,080	2.6 GB	3.6 GB
6	216,469,471,680	54.1 GB	90.2 GB
7	2,535,785,239,680	633.9 GB	1.2 TB
8	12,678,926,198,400	3.2 TB	7.6 TB

Fig. 18: Storage requirements of the endgame database

RESOLUTION OF THE “HUMAN FACTOR”

As is has been able to ascertain, and as has been already remarked several times, the importance of a suitable evaluation function is crucial for the correct functioning of our algorithm: otherwise, our machine would not be able to correctly evaluate the position and it would make all previous work effectively useless.

The evaluation function is, in fact, a good starting point, but there are many considerations to make to write a good evaluation function, many of which are not even easily representable by us humans.

Furthermore, the considered specific case of Jungle, in which, although it is not easy to write an optimal evaluation function since there are numerous considerations that need to be made (importance of the pieces, special movements, particular pieces can capture only particular pieces, positioning, victory by reaching the dojo, etc.), there are board games where the considerations to be made are far superior and deeper, such as in Go, where, to write a suitable evaluation function it would take immense interdisciplinary knowledge in multiple fields without considering an absolute mastery of the game itself.

However, various solutions allow us to solve the following problem, there will be descried only a couple of the possible ones, which are the ones that has been considered more interesting and that follow two diametrically opposite paths.

d. GENETIC ALGORITHM

A possible solution to the problem is to have the computer creating the evaluation function. One of the methods is using the genetic algorithm.

A genetic algorithm is a heuristic algorithm used to solve optimization problems for which no other efficient algorithms of linear or polynomial complexity are known. The adjective "genetic", inspired by the principle of natural selection and biological evolution theorized in 1859 by Charles Darwin, derives from the fact that, like the Darwinian evolutionary model that finds explanations in the branch of biology called genetics, genetic algorithms implement mechanisms conceptually similar to those of the biochemical processes.

In summary, genetic algorithms consist of algorithms that allow evaluating different starting solutions (as if they were different biological individuals) and that by recombining them (similarly to sexual biological reproduction) and introducing elements of disorder (similarly to random genetic mutations) to produce new solutions (new individuals) who are evaluated by choosing the best ones (environmental selection) to converge towards "best" solutions. Each of these phases of recombination and selection can be called generation like those of living beings.

Without going into a detailed description of how this algorithm works, in our case, the goal is to create different evaluation functions (simply going to modify the numerical values within the matrices that characterize the value of each animal) and exploit the algorithm for evaluating which are the "strongest genes" and joining them together. To evaluate the strongest evaluation functions, it is sufficient to carry out tournaments in which each function of a generation challenges itself with all the other functions of its generation, in such a way to be able to obtain, by analysing the number of wins, which are the strongest and keep going deep into the method until getting the "best" function.

e. MONTE CARLO TREE SEARCH (MCTS)

Monte Carlo tree search (MCTS) is a heuristic search algorithm developed for searching in decision trees, which finds applications in the solution of board games.

With this algorithm it is possible not only to find solutions to deterministic games (such as chess) but, due to its nature and method of implementation, it is also possible to use it for non-deterministic games (such as poker).

MCTS works by expanding the search tree of the most promising moves by sampling the random search space using the Monte Carlo method. The research is based on the execution of numerous playouts, where each playout consists of the execution of an entire game starting from the current position, selecting the moves at random. The match result is then used to weigh the run (and each component of it) and the weight determines the likelihood of making the same move in subsequent playouts.

So, instead of using an evaluation function Monte Carlo Method generates a set of possible continuations (states) of the game and analyses them and the results coming from them (victory or defeat) for evaluating which is the best approach to the problem.

There will be no discussion of the analytical discussion of the Monte Carlo method as it is an advanced topic in the world of statistics and stochastic models, it is enough to know that the "law of large numbers" are being exploited to find the "true" game result for each potential move that can be played.

The Monte Carlo approach is a greedy approach, which will generally tend to extend the depth of the tree more than the width of the tree.

A fundamental tool for this method is the Upper Confidence Bound (UCB) which, applied to the tree, allows us to balance the binomial exploration/exploitation by periodically exploring nodes seen less frequently, potentially discovering more optimal paths than those that they are currently being exploited. Each node (possible state) is always represented by two numbers, one to determine the "value" of the node, the other to count the number of visits that have been made to that node. When the new node is created both parameters are set to 0.

It is possible to summarize the concept of the Monte Carlo Tree Search in four phases:

Selection: Through the UCT formula the next node to explore and the potential best move to start generating our series of random games are going to be chosen.

$$UCT = x_i + C\sqrt{\frac{\ln{(N)}}{n_i}}$$

In the formula x_i represents the average value of the state of the game, C is the "temperature" that is defined manually and is the central point of our research, N represents the total number of simulations and n_i represents the current nodes visits. As said earlier, it is possible to balance the exploration/exploitation binomial precisely through temperature. As with other research methods (like in the simulated annealing algorithm), a high temperature allows us to increase the exploration level in order to visit less-visited nodes, a low temperature instead allows us to go deeper into the search for nodes already explored to collect more information about them.

Expansion: After selecting the new node to explore through the selection process, algorithm is going to generate the child.

Simulation: A simulation choice is made between the different moves until a result is reached.

Backpropagation: After determining the value of the newly added node, the remaining tree needs to be updated. This last step is then performed from the new node to the root node. During this process, the number of simulations stored and the results obtained in each node is increased each time.

This, as already specified, is not intended to be a complete explanation of the following method, but it is a small summary that allows us to broadly understand its operation.

Below there is the pseudocode for the Monte Carlo algorithm:

```
class Node:
  def __init__(self, m, p): # move is from parent to node
    self.move, self.parent, self.children = m, p, []
    self.wins, self.visits  = 0, 0

  def expand_node(self, state):
    if not terminal(state):
      for each non-isomorphic legal move m of state:
        nc = Node(m, self) # new child node
        self.children.append(nc)
def update(self, r):
  Class Node:
    def __init__(self, m, p): # move is from parent to
node
      self.move, self.parent, self.children = m, p, []
      self.wins, self.visits  = 0, 0

    def expand_node(self, state):
      if not terminal(state):
        for each non-isomorphic legal move m of state:
          nc = Node(m, self) # new child node
          self.children.append(nc)
     def update(self, r):
   self.visits += 1
       if r==win:
         self.wins += 1

     def is_leaf(self):
       return len(self.children)==0

     def has_parent(self):
       return self.parent is not None

   def mcts(state):
     root_node  = Node(None, None)
     while time remains:
       n, s = root_node, copy.deepcopy(state)
       while not n.is_leaf():    # select leaf
         n = tree_policy_child(n)
         s.addmove(n.move)
      n.expand_node(s)         # expand
      n = tree_policy_child(n)
      while not terminal(s):    # simulate
        s = simulation_policy_child(s)
      result = evaluate(s)
      while n.has_parent():     # propagate
        n.update(result)
        n = n.parent

  return best_move(tree)
```

XI. PYGAME

For the development of the UI was used Pygame [10], a Python library that allows you to simplify the writing of video games, containing within it elements of computer graphics and sound libraries designed to be exploited within a Python program. Being a library and not being python native for this type of use, the result is not comparable to what could have been obtained using other Engines such as Unity or Unreal Engine.

However, considering that the main goal of this project was to develop a simple artificial intelligence and test various algorithms and to see its efficiency, this problem can take a back seat. Given this consideration, the final graphic interface is in any case particularly pleasant as regards the gaming experience, and undoubtedly more captivating as regards the user experience.

It is possible to choose, through the terminal, the game mode and the difficulty of our opponent, after having done this, it is possible to move completely on the graphic interface to be able to play or observe a game.

Regarding the user experience, it was decided to insert a highlighting system, able to show the player where he can move with the selected piece and improve the game experience. Furthermore, using the 'h' key, it is possible to request the help of our AI who will make the move in the place of our player.

As for the images taken, they were all collected from sites that share and allow the free use of the latter, the same was done with the background music. As for the menu interface, it was done via Canva.

All images and media, to improve the performance, are loaded and saved in memory when the game is started, in this way it is possible to avoid reloading these images at each update (which are on average 28 per second), thus improving the efficiency of the program

Jungle Game
AI Course project - FEUP 2022
Follow the details in the terminal to enter the necessary information. Then come back here to play!
During the game, you can press 'h' during your turn to ask our AI for help!

Fig. 19: Main Menu Wallpaper

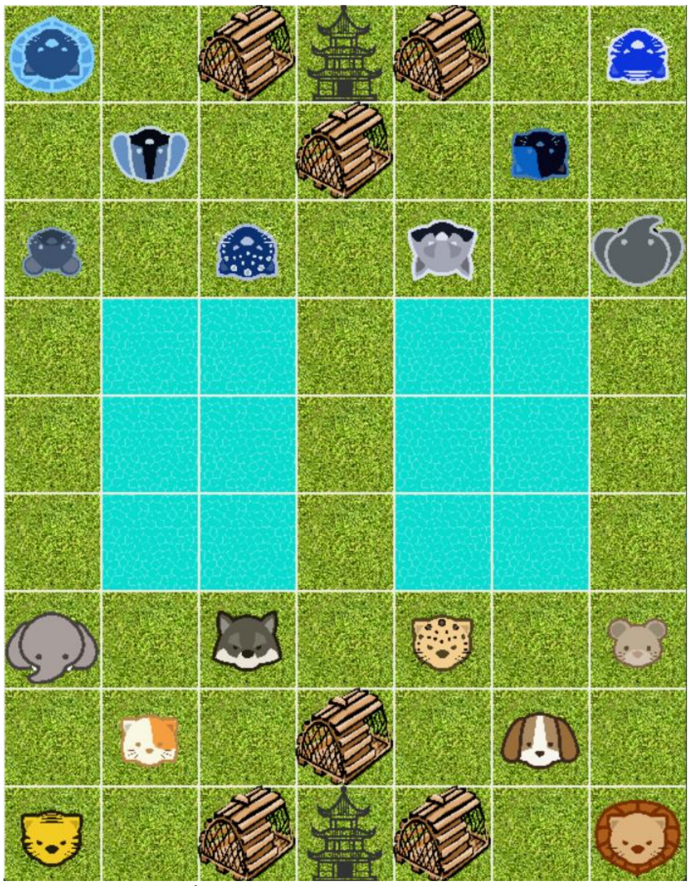


Fig. 20: Game User Interface

XII. CONCLUSIONS

The whole development process, and especially the testing, showed how important the depth and the evaluation function are in the MiniMax algorithm. Even though the developed evaluation methods were making expected moves in pre prepared scenarios, they were sometimes making suboptimal moves during real-game scenarios.

With the increasing depth, this problem was decreasing. Recursive nature of MiniMax algorithm makes it expensive to compute, considering that each potential board state generates up to 32 more in the player’s next move.

If not for the computational power, the MiniMax algorithm is great for those “chess-like” games: calculating the best moves for the player and the opponent, to choose the optimal way of playing sounds in its nature like a perfect way of playing. With the infinite depth it could be possible to solve the game, making one of the players always winning with optimal moves.

Due to the facts stated above it appeared extremely important to optimize the MiniMax algorithm: the usage of alpha-beta cuts and remembering the last played moves allowed the program to run at a reasonable depth, but still under our expectations, considering that a game of really good level has not been achieved, even on the smaller board with fewer animals, but at least we got results allowing us to make some conclusions of our implementation of the program.

PERSONAL CONSIDERATIONS

Considering all the steps that were followed during the development of the software and the drafting of the report, we were very satisfied not only with the work done but above all with everything he left us, since this was the first job in this area for both of us.

Being able to work in a practical way on this software, and not doing it only from a theoretical point of view, has allowed us to touch first-hand what are the great differences between theory and practice. We have observed how important certain elements are in the internal programming of an AI algorithm, in which not considering even a small thing can lead to collapses in terms of efficiency (suffice it to see that to have one of the most profound improvements we have had in this field "was enough" to invert two arrays).

Undoubtedly the central point of all our work, also considering how the MiniMax algorithm works, which is potentially perfect for this type of problem and has as its only limit the time and the amount of computing power, was on the one hand, as already mentioned, the particular attention to efficiency, on the other hand, to allow for a valid and fun gaming experience considering our limits.

The construction of the evaluation tables, in fact, was also another crucial point, to make the most of the algorithm and induce to play offensive games. In fact, having developed a game, we have also had the user experience particularly at heart, which does not have to face endless games against a computer that plays defensively, but frenetic and fun games.

Furthermore, given that this type of data structure and simplifications are not enough to create an AI capable of going very deep in the search for the best move (as in the case of chess engines), it was necessary to push it to prefer an offensive game.

We do not want to underestimate the attention paid, therefore, in the drafting of our evaluation tables, which have given us very satisfying results, considering that, comparing our final table with that of Bas van Boven [4] at the same depth, our AI worked slightly better.

Putting aside the purely practical and programming aspect now, we want to pay attention to what this project has given us also from a theoretical point of view, which, as I said, is perhaps less evident but still fundamental.

On the one hand, it was necessary to go and study not only the code but the principles underlying all these algorithms, doing in-depth personal research, for understanding what the strengths and weaknesses of our program could be, act on the latter and obtain increasingly satisfactory results. In fact, creating such a project, we realized that it is not enough to use the code behind MiniMax for example, but it is essential to implement it in the right way and use the right data structure that we have, and the only way to do so in the right way is knowing in an almost perfect way every single step of this algorithm, to manage in the best way its strengths and weakness.

We realized that these are essential to obtain at least satisfactory results. In fact, our entire first part of the project, immediately after developing the game for two humans, was based on study and research, a moment that we found extremely formative and fun, thanks to the excellent information pages found online and the huge number of papers available.

From the point of view of data structures, it is necessary to spend a few more words. The use of normal python paradigms has certainly allowed us to obtain satisfactory results, but, at the end of our project, we realized that using external libraries more suited to this type of work (e.g., Panda or NumPy) would perhaps have further optimized our results.

Returning instead to talking about the theoretical study, not only the study on books or on informative web pages has proved to be very important, but it was also very important to read and interface with other reports and articles or university thesis carried out by other students and teachers on this same or similar draft (e.g., chess or checkers). This, being the first time we have had a similar approach, has allowed us to get to know this world.

Having to write a paper in IEEE format was also extremely interesting and challenging on our part, having been our first time.

So, to conclude, what this project has left us is not just the creation of software, but much more: a series of tools, knowledge, awareness, and first experiences about this matter and what it means to face it.

We found this experience extremely formative because it allowed us to have a more general but not for this reason not complete vision of the many algorithms and approaches that can be had in artificial intelligence field, also going to define more precisely the intrinsic meaning of this word.

We also had the opportunity to have an idea, even if only a general one, of what it means to work on a project with these ends.

One thing we also appreciated is that this knowledge took place only in part directly, but the individual action was also on a high level to deepen certain aspects, intensifying our spirit of research and curiosity towards the matter.

Furthermore, the opportunity to work in a group, divide the necessary work and go to discuss and collaborate to achieve a result was also extremely fruitful and challenging from our point of view.

References:

[1] Artificial Intelligence, a Modern Approach – Stuart Russel

[2] Python implementation of algorithms from «Artificial Intelligence, a Modern Approach» (https://github.com/aimacode/aima-python)

[3] Minimax and Monte Carlo Tree Search – Philipp Muens (https://philippmuens.com/minimax-and-mcts)

[4] Solving Jungle Checkers – Bas van Boven (https://theses.liacs.nl/pdf/2013-2014BasvanBoven.pdf)

[5] Discovering and Searching Loosely Coupled Subproblems in Dou Shou Qi – Joseph Burnett

[6] The Impact of Search Depth on Chess playing strength – Diogo R. Ferreira

[7] Monte Carlo Tree Search (MCTS) (https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/) (https://webdocs.cs.ualberta.ca/~hayward/396/jem/mcts.html)

[8] Analysis of the possible application of genetic algorithms for creating a heuristic function evaluating boards in the Jungle game (https://github.com/bax533/JungleChess_Evolution/blob/master/sprawozdanie.pdf)

[9] Artificial Intelligence Lectures – Luis Paulo Reis

[10] Pygame (https://www.pygame.org)