Fanorona: Unbeatable AI player

Catia Teixeira, *Student, FEUP,* Rojan Aslani, *Student, FEUP*

***Abstract*—Fanorona is a two-player board game that has been played for hundreds of years by Malagasy. This article provides an overview of the game rules and complexity analysis, as well as a detailed description of the methodology and implementation details of three different artificial intelligence (AI) algorithms that play Fanorona. The authors aimed to evaluate the algorithms' performance in different game scenarios, including games played between human players, between human and computer players, and between two computer players. Moreover, three board sizes and three difficulty levels are available for configuration. The implemented AI algorithms are Minimax, Minimax with Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS). The results showed that MCTS on average is slower in making a move in comparison to the Minimax algorithms. However, it has a higher chance of winning in smaller board sizes, with fewer moves. All three AI players were able to win against a random player. Moreover, it was interesting to observe that even though it tends to be slower, MCTS seems to be able to adopt a more offensive attitude in the game. It would be interesting to further explore the hyper-parameter tuning of the algorithms in future works.**

***Index Terms*—Fanorona, Board game, Adversarial Search, Artificial Intelligence, Minimax, Monte Carlo Tree Search.**

I. Introduction

FANORONA is a two-player strategic board game that originated in Madagascar. Its rules are quite similar to those of the game of checkers, but it has its own unique features [1].

Fanorona is played with black and white stones placed on a rectangular board. The size of the board varies depending on the variant of the game being played. The three standard variants are Fanoron-Telo (3 by 3), Fanoron-Dimy (5 by 5), and Fanoron-Tsivy (5 by 9). An example of the Fanaron-Tsivy board is presented in Fig. 1 [2].

Players alternate turns, starting with White. Each player has 22 stones of their color on the board. The stones are placed where the grid lines cross, leaving the central point empty. A line represents the path along which a stone can move during the game. There are crossings that are strong and weak. In weak intersections, only horizontal and vertical moves are possible, while in a strong intersection, it is also possible to move a stone diagonally [2], [3].

There are two kinds of moves: non-capturing (*paika* move) and capturing. A *paika* move consists of moving one stone along a line to an adjacent intersection without removing any of the opponent's stones. Capturing moves, which imply the removal of one or more stones of the opponent, are necessary for a win and have to be played in preference to *paika* moves. There are two ways to do a capturing move:

1) **Approach:** moving the capturing stone to a point adjacent to an opponent's stone, which must be on the continuation of the capturing stone's movement line.

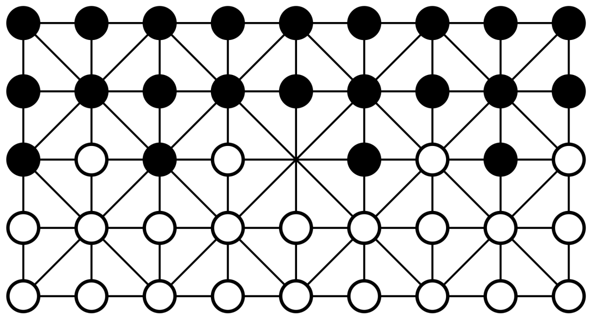

Fig. 1: Fanoron-Tsivy board (5x9) with the initial position of the stones [3].

2) **Withdrawal:** the capturing stone moves from a point adjacent to the opponent's stone, away from the stone along the continuation of the line between them.

When an opponent stone is captured, all opponent stones in line beyond that stone (as long as there is no interruption by an empty point or an own stone) are captured as well. Approach and withdrawal captures cannot be made at the same time – the player must choose one or the other.

The capturing stone is allowed to continue making successive captures, with the following restrictions:

- The stone is not allowed to arrive at the same position twice.
- It is not permitted to move twice consecutively in the same direction (first to make a withdrawal capture, and then to make an approach capture) as part of a capturing sequence.

The winner is chosen when they take all the opponent's stones or leave the opponent in a position where they can't move. If neither player manages to win, then the game is a draw [2]. According to several authors, when reaching the perfect play level, the game tends to reach a draw condition [3], [4].

A. Complexity analysis

It is important to quantify the game's complexity as it allows analysis of the difficulty of the game and allows choosing the most adequate techniques. According to [3], after performing 10,000 games by Alpha-Beta players, Fanoron-Tsivy was estimated to have a game-tree complexity of 10^{46}. Moreover, the space complexity was calculated to be 10^{21}. These values are comparable to those of the game of checkers, which has a game-tree complexity and state-space complexity of 10^{31} and 10^{20}, respectively [5]. However, the game-tree and state-space complexity of Fanorona is somewhat higher than those of checkers.

B. Related work

It is interesting to mention that up till now, Fanorona has never been strongly solved. A different number of strategies and techniques have been developed in the past towards solving the game. According to [4]:

1) The Minimax algorithm: The basic and classic approach technique is used for a variety of two-player games. The most popular variant is the Alpha-Beta pruning.

2) The WINIMAX algorithm: This algorithm tries to maximize the win probability based on Minimax and the computing of p_{win}.

3) The Memory Test (MT) framework: Based on null-window searches and transposition tables, this is a high-level game-tree search resulting in a best-first behavior of depth-first, classic Alpha-Beta search algorithm.

4) The Monte Carlo Tree Search (MCTS) algorithm: This algorithm is based on simulations of full-game and applying the sequence of actions that lead to a higher number of wins.

5) The opening book: This is a general, well-known technique for solving board games. Some positions are stored at the start of the game with the best moves for each of them. These openings may be proposed by human experts on the specific game.

6) The proof number search algorithm: A general algorithm that is able to determine if a given goal can be achieved. A best-first algorithm that requires all nodes to be stored in memory.

7) The rote learning: Memorizes encountered results, saving time for later occurrences by retrieving the stored results and increasing perfect-play performance. Gaining experience over time, its performance improves.

8) The endgame databases: This technique consists of computed tables that contain the optimal solutions for a specific endgame or a set of endgames in a game. They provide a complete and optimal solution to a specific sub-problem of the game. One big part in its implementation is the optimization of retrieval and storage.

II. METHODOLOGY

As this project focuses on two players with conflicting goals who are trying to explore the same search space for the solution, this project is formulated as an adversarial search problem. Fanorona's environment is accessible, deterministic, non-episodic, static, and discrete, hence, it is a reasonable game to solve by computer. In this section, we will discuss the details of the formulation and implementation of the project.

A. Formulation of the problem

The source code for the Fanorona board and the Minimax agent was obtained from *marcogx*'s GitHub repository[1], and the source code for MCTS player was obtained from *ahenrij*'s Github repository [2]. Withal vast alterations were applied to the code to adapt to several configurations, adding new AI players, organizing the code in different classes, as well as adding and adjusting several base functionalities.

[1]https://github.com/marcogx/Fanorona-Game-AI

[2]https://github.com/ahenrij/fanorona

The initial state of the game is demonstrated in Fig. 1. The operators for the movements of the stones were defined according to the position of the stone, more specifically, in each of the intersections, the possible directions of the movement were denoted. The variety of board sizes made it challenging to automatize this function at this point, however, in future works a more efficient way of doing these operations could be explored. The evaluation function is presented in Eq. 1 and Eq. 2 and will be explained in detail in the upcoming subsections. The objective test for this game is quite simple, aiming to remove all stones of the opponent on the board, and leaving none behind. Hence, the objective test is whether the number of opponent stones is equal to zero.

B. Implementation details

To implement the Fanorona board game, Python programming language was used in Visual Studio Code and PyCharm Community integrated development environments. In total six modules were created:

1) **main.py:** main control log
2) **player.py:** generates and controls the different types of players
3) **game.py:** controls general functionalities, including making moves and user interface (UI) control
4) **board.py:** represents the game board
5) **stats.py:** generates records about the configuration of the game, the time duration, the number of moves, and the winner, and exports them into a Comma Separated Values (CSV) file in a tabular format
6) **config.py:** holds global variables that are used and have to be uniform within all files

A Graphical UI (GUI) was developed using pygame package. The GUI allows the user to configure the game by defining the following parameters:

- Game mode: Computer-Computer, Human-Computer, Human-Human
- 1st and 2nd players: Human, Minimax, Minimax with Alpha-Beta pruning, MCTS
- Board size: 3x3, 5x5, 9x5
- Difficulty: Easy, Medium, Hard

C. Approach

1) Evaluation function (Heuristic): The proposed heuristic function was used both as the evaluation function for Minimax agents and as a biasing function for MCTS agent. The function basically compares the chance of winning of each of the players at a given game state, according to the number of stones present on the board, as well as the position of the stones (extra points for strong intersections), as indicated in Eq. 1. The overall score is calculated by subtracting the opponent's points from the player's point, and dividing it by the total point, as demonstrated in Eq. 2.

$$points = Number\ of\ player's\ stones + 0.5 \times stones\ on\ strong\ intersections \quad (1)$$

$$\frac{Player\ points - Opponent\ points}{Player\ points + Opponent\ points} \quad (2)$$

2) Draw condition: In this program, the game is considered a draw when the players repeat three consecutive moves.

3) Hint button: For the human-computer game mode, a hint button was implemented. The button is only available in this mode and the optimal move is shown only after clicking on the hint button. This functionality uses a Minimax with Alpha-Beta pruning Agent with a maximum search depth of 2.

D. Implemented algorithms

Several computer agents were implemented, as mentioned earlier. The simplest one is the Random agent, followed by AI agents such as Minimax, Minimax with Alpha-Beta pruning, and MCTS. The AI agents are all utility-based agents and act based not only on what the goal is but also on the best way to reach that goal.

*1) **Random Agent**:* The random agent was used as the computer player in easy mode. As the name implies, this agent makes random moves by getting the list of all possible moves and choosing the first one in the list.

*2) **Minimax Agent**:* The Minimax algorithm is a basic depth-first search algorithm. It is an approach to create an AI engine for any Zero-Sum, perfect Information, two players games. The two players are identified as MIN and MAX. It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up through the tree as the recursion unwinds [6]. The most popular variant of the Minimax algorithm is the Alpha-Beta pruning [7].

The optimal strategy of this algorithm is determined by working out the Minimax value of each state of the tree. This value is the utility of being in that state, assuming both players play optimally until the end of the game. MAX prefers to move to a state of maximum value when it is its turn to move, while MIN prefers a state of minimum value (see Fig. 2) [6].

Once the algorithm reaches the maximum depth, it uses the evaluation function (Eq. 2) to choose the higher-scoring move. The pseudo-code for the implementation of Minimax algorithm in this Fanorona game can be found in Appendix A-A.

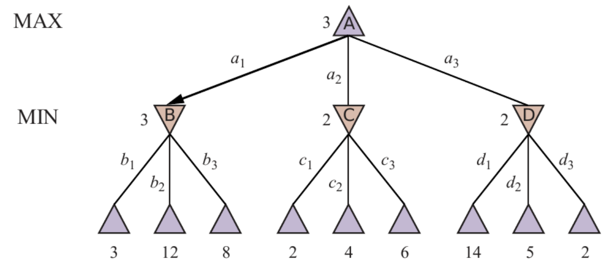

Fig. 2: Outline of Minimax [6].

In this work, the difficulty of the game was defined by the depth of the Minimax algorithm, as demonstrated in Tab. I. The depth of the search tree is decreased for Fanoron-Tsivy boards. This adjustment was done to avoid prolonged decision-making time of the agents, due to the big size of the search space. With the decrease of the depth by just 1 value, the agent became much faster, and the game was playable in reasonable time intervals.

*3) **Minimax with Alpha-Beta pruning**:* When using Minimax algorithm, the number of game states grows exponentially as the depth of the tree increases. This can become problematic in terms of computational capacity.

The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated. Given a position, if the player has a better choice either at the same level of the tree or at any point higher up in the tree, it will choose not to explore that branch further and not to move to that position [6]. To implement Minimax with Alpha-Beta pruning, two additional parameters are added to the Minimax algorithm, describing the bounds on the backed-up values that appear in the path:

- Alpha (α): value of highest valued choice found so far
- Beta (β): value of lowest valued choice found so far

These values are updated as the algorithm goes through the tree and prunes the remaining branches at a node as soon as the value is known to be worse than the current value for α or β for MAX or MIN, respectively (see Fig. 3) [6]. The pseudo-code of this AI algorithm can be found in Appendix A-B. The depths for each difficulty level of the game are demonstrated in Tab. I.

A problem worth mentioning once limiting the search depth on both Minimax and Minimax with Alpha-Beta pruning is the horizon effect. When searching with depth limits, the algorithm can possibly make a move that results in a loss, without being able to detect the consequences in the current depth of the tree. Similarly, the optimal moves may be beyond the reach of the maximum tree depth. This can lead these algorithms to be at a disadvantage and can lead to the agents reaching loops in the game.

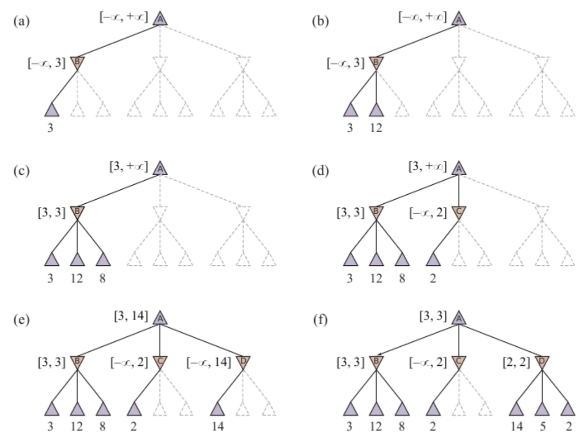

Fig. 3: Outline of Minimax with Alpha-Beta pruning (stages in alphabetical order.) [6]

TABLE I: Parameter setting of different difficulty levels and board sizes for Minimax and Minimax with Alpha-Beta pruning algorithms (d: Maximum depth).

Difficulty	Board size	Minimax d	Minimax-$\alpha\beta$ d
Medium	3x3	3	3
Medium	5x5	3	3
Medium	9x5	2	2
Hard	3x3	4	5
Hard	5x5	4	5
Hard	9x5	3	4

*4) **Monte Carlo Tree Search (MCTS)**:* The basic MCTS strategy does not use a heuristic evaluation function, using instead an estimation of the state by averaging the utility over several simulations of complete games from the given state. As shown in Fig. 4, the algorithm goes through 4 steps repeatedly:

1) **Selection**: choose a move - starting from the root of the search tree, guided by the selection policy
2) **Expansion**: grow the search tree by generating a new child for the selected node
3) **Simulation**: perform a rollout from the newly generated child node - The moves are chosen for both players according to the rollout policy
4) **Back-propagation**: use the result of the simulation to update all the search tree nodes going up until the root

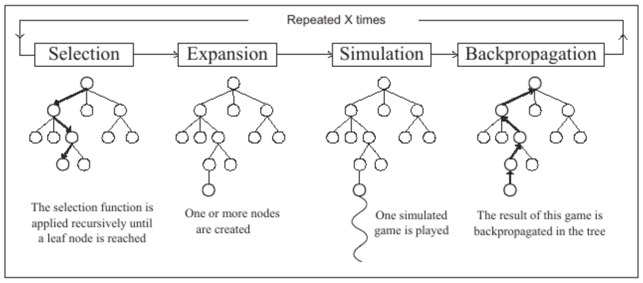
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 4: Outline of Monte Carlo Tree Search [8].

This way, the algorithm uses a selection policy that balances exploration and exploitation to determine which nodes to expand next [6]. The first formula for balancing the two is called the Upper Confidence Bound 1 (UCB1). Upper Confidence Trees (UCT) is based on UCB1 formula and the Adaptive Multi-Stage Sampling algorithm [9], [10]. The UCT formula used for this project is represented in Eq. 3.

$$UCT = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{2 \times ln(PARENT(n))}{N(n)}} \quad (3)$$

Where

- $U(n)$ is the total utility of node n
- $N(n)$ is the number of times node n has been visited
- $PARENT(n)$ is the number of times the parent node of n has been visited
- C is a constant that determines the balance between exploration and exploitation

The utility $U(n)$ was applied as the difference between the number of wins and the number of loses of node n.

The constant 2 inside the square root is used to scale the exploration term, such that it is proportional to the square root of the number of times the parent node has been visited. By scaling the exploration term with a constant inside the square root, the algorithm can adjust the degree of exploration based on the prior knowledge about the parent node [10].

The constant C in the UCT formula (Eq. 3) defines the balance between exploration and exploitation. A higher value of C will result in more exploration, while a lower value of C will result in more exploitation. By choosing an appropriate value for C, the algorithm can balance the need for exploration with the need for exploitation, depending on the characteristics of the search space and the problem domain [6].

In this work, a value of 2 was chosen for constant C, determined by practical experimentation. It is worth noting that this value of C results in a relatively high degree of exploration. Lower values of C have been shown to result in more exploitation and less exploration, which can be beneficial in some cases. However, in the practical experimentation conducted for this work, lower values of C resulted in poorer results, indicating that a higher degree of exploration was needed to effectively search the search space.

Although MCTS can be applied in the absence of domain-specific knowledge, another significant aspect of MCTS is the use of this knowledge of the game rules, which can be used to bias the search towards positions that are more likely to lead to a win, achieving improvements in performance [11].

In the specific case of this work, Eq. 2 was used to bias the search of the algorithm in the simulation phase. In each iteration of the simulation, the algorithm calculates all possible moves and applies Eq. 2 for scoring. After, the best scoring move is chosen as the next move of the rollout. The simplified pseudo-code can be found in Appendix A-C.

It is worth mentioning, one possible way to improve the algorithm for future work would be to include the verification of the draw condition in this phase and use it to exclude that move or end the rollout early if that is the only possible move available. This can exceptionally decrease the decision-making process of the agent.

Regarding parameter setting for different board size and difficulties of the game, the maximum rollout depth (depth of simulation tree) and maximum number of iterations was adjusted. The values for these parameters can be found in Tab. II.

TABLE II: Parameter setting of different difficulty levels and board sizes for Monte Carlo Search Tree.

Difficulty	Board size	Rollout depth	Nº of iterations
Medium	3x3	500	100
Medium	5x5	∞	100
Medium	9x5	500	10
Hard	3x3	500	200
Hard	5x5	∞	500
Hard	9x5	500	20

III. Results and Discussion

As a result of this work, a python based program was developed that allows users to play Fanorona games in a variety of different configurations, with a user-friendly interface.

The computer performance was measured in different difficulties and board sizes, for all players (Random, Minimax, Minimax with Alpha-Beta pruning, and MCTS). Subsequently, the game results of 243 configurations were generated. The effectiveness of the algorithms was measured by their winning rate. It is interesting to mention that the human players failed to win any games against the AI agents.

The game results of AI vs random players were evaluated, as well as the time duration, and the number of movements per game. The game time duration can be an important indication of the performance of the agents. For example, comparing the game time of MCTS vs. Minimax and MCTS vs. MCTS can allow an indirect indication of the speed of decision-making of the agents. The same argument applies to the total number of movements in a game.

The results showed that Minimax and Minimax with Alpha-Beta pruning algorithms win over Random player 100% of the games, regardless of the difficulty and board size. Regarding the MCTS algorithm, it wins over random player 67% of the games, as shown in Appendix D. The match between MCTS and the random player leads to a loss of MCTS or a draw in the bigger board sizes. This is due to the configurations of MCTS in bigger board sizes, which doesn't allow a deep enough search in the search tree and lacks a sufficient number of iterations to ensure a win, due to the big state space. This choice of configuration was taken in order to ensure the playability of the game within an acceptable time frame, as mentioned earlier.

Furthermore, each AI algorithm played against itself, in different difficulties (depths). This was done to evaluate the effectiveness of the AI in hard vs. medium mode. In all cases, the player with hard difficulty won over the player with medium difficulty. This confirms that the deeper the search tree, the more robust the agent becomes in the Fanorona game.

According to Tab. V in Appendix D, comparing the time per move for Minimax and Minimax with Alpha-Beta pruning in medium difficulty, we can confirm that Alpha-Beta pruning does indeed decrease the decision-making time for the agent. Regarding the hard difficulty, the comparison would not be valid, due to the different depths of the tree between the two agents (for details about the depths revisit Tab. I).

Regarding the speed of decision-making of the agents, according to Tab. V the average time per move of MCTS is higher in comparison to the other two agents, as the search space is much larger. This applies to all cases except for the medium difficulty on board size 9x5, which is understandable given that the number of iterations in this configuration is the smallest (Tab. II).

When MCTS agent played against all other AI agents, it was verified that even though in the smallest board size all agents are fast and have taken about the same amount of time to play against MCTS, in bigger board sizes MCTS is the slowest agent (Fig. 5). Moreover, we can conclude that MCTS playing against Minimax, Minimax with Alpha-Beta pruning, or random player, the game time durations are quite short and similar, making the three mentioned agents much faster than MCTS. This is understandable because the bigger the board gets, the bigger the search space, and the more time it takes for MCTS to explore the nodes. On the other hand, Minimax and Minimax with Alpha-Beta pruning players have a limited depth in the search tree and the time duration is not majorly impacted.

Next, all agents competed against the random agent to evaluate the number of movements per game (Fig. 6). It was interesting to observe that in the smaller board size (3x3 and 5x5) MCTS is the agent that needs the least number of movements to finish the game. On the other hand, in the bigger board size, MCTS is the one that takes the most number of moves to finish the game. Again, this occurrence happens because of the configuration of the hyperparameters of MCTS agent in 9x5 board size, as mentioned before.

It is interesting to mention that in games of MCTS vs. Minimax or Minimax with Alpha-Beta pruning, the winners are consistent, as shown in Appendix C and Appendix E. This is expectable as the approaches of Minimax and Minimax with Alpha-Beta pruning are identical.

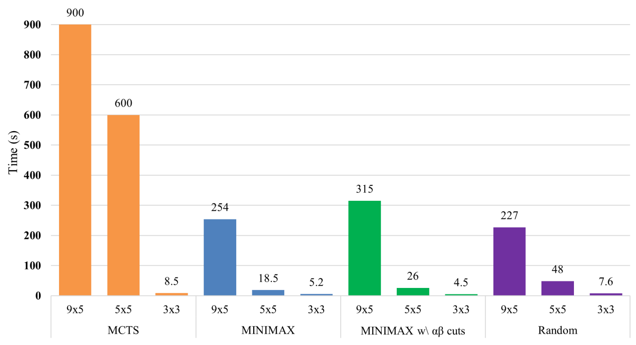

Fig. 5: MCTS agent games time duration.

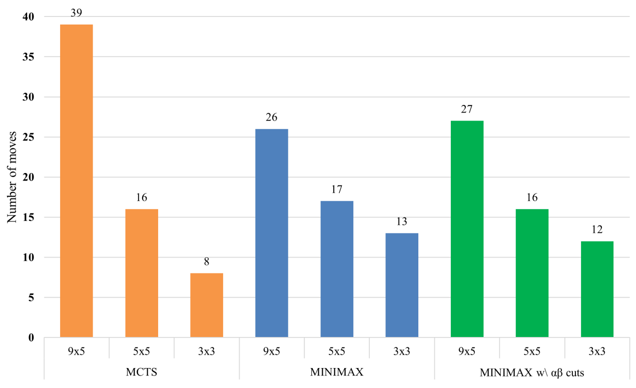

Fig. 6: Number of moves per game - AI Agents against Random agent.

IV. Conclusion

In this work, a Python-based Fanorona game was developed with various players, including a random player and 3 AI players - Minimax, Minimax with Alpha-Beta pruning, MCTS.

Overall, it was confirmed that all AI players win against a random player. Minimax with Alpha-Beta pruning proves to be slightly faster than Minimax player. Both Minimax algorithms guarantee to find optimal moves regardless of the difficulty level and board size, but are computationally expensive and are prone to get stuck in loops.

MCTS agent seems to be able to adopt a more offensive attitude, win the game in fewer moves, and play quite efficiently in smaller board sizes, but overall takes much longer to make a decision. In contrast with Minimax-based agents that in certain situations continuously avoid losing by entering a "loop" (Horizon problem, i.e., repeatedly deciding to move between two places neither winning nor losing), MCTS more actively pursues strategies that in the long run lead to a win. For example, it was observed, MCTS was able to choose to lose one stone and in the next move was able to capture a number of opponents' stones that compensate for the initial loss. Altogether, both Minimax agents show promising results.

For future works, it would be interesting to allow the players to make consecutive moves in one turn. This option was not implemented in the current algorithm. It would also be interesting to explore other draw situations, especially for the AI players.

Regarding the algorithm, in future works, it can definitely be optimized to decrease space complexity.

Another interesting topic to explore further would be successor generation ordering in the Minimax algorithms. This can speed up the algorithm and reduce cases of draw, against other AI agents.

It would be important to future investigate the hyperparameter-tuning of the most relevant variables in UCB for MCTS. For MCTS the incorporation of draw condition in the rollout phase can possibly drastically decrease the computation time and improve its overall performance.

Acknowledgment

The authors would like to thank *marcogx* and *ahenrij* for sharing their codes for the Fanorona game on GitHub.

Appendix A
Pseudo-Codes

A. Minimax

Algorithm 1 Minimax

```
function PLAY($game, state$)
    $player \leftarrow game$.CURRENT_PLAYER
    $move \leftarrow$ MINIMAX-SEARCH($game, state$)
    $new\_state \leftarrow game$.MAKE-MOVE($player, state, move$)
    return $new\_state$
end function

function MINIMAX-SEARCH($game, state$)
    $depth \leftarrow 0$
    $best\_v, best\_m \leftarrow$ MAX-VALUE($game, state, depth$)
    return $best\_m$
end function

function MAX-VALUE($game, state, depth$)
    if $game$.IS-TERMINAL($state$) then
        return $game$.UTILITY($state, player$), $null$
    end if
    $value, move \leftarrow -\infty$
    for all $m \in game$.GET-MOVES($state, player$) do
        $n\_state \leftarrow game$.MAKE-MOVE($player, state, m$)
        $v, m2 \leftarrow$ MIN-VALUE($game, n\_state, depth + 1$)
        if $v > value$ then
            $value, move \leftarrow v, m$
        end if
    end for
    return $value, move$
end function

function MIN-VALUE($game, state, depth$)
    if $depth \geq game$.difficulty then
        return $game$.UTILITY($state, player$), $null$
    end if
    if $game$.IS-TERMINAL($state$) then
        return $game$.UTILITY($state, player$), $null$
    end if
    $value, move \leftarrow +\infty$
    for all $m \in game$.GET-MOVES($state, player$) do
        $n\_state \leftarrow game$.MAKE-MOVE($player, state, m$)
        $v, m2 \leftarrow$ MAX-VALUE($game, n\_state, depth + 1$)
        if $v < value$ then
            $value, move \leftarrow v, m$
        end if
    end for
    return $value, move$
end function
```

B. Minimax with Alpha-Beta cuts

Algorithm 2 Minimax with Alpha-Beta cuts

```
  function PLAY($game, state$)
      $player \leftarrow game$.CURRENT_PLAYER
      $move \leftarrow$ MINIMAX-ALPHABETA-SEARCH($game, state$)
      $new\_state \leftarrow game$.MAKE-MOVE($player, state, move$)
      return $new\_state$
  end function

  function MINIMAX-ALPHABETA-SEARCH($game, state$)
      $depth \leftarrow 0$
      $best\_v, best\_m \leftarrow$ MAX-VALUE($game, state,$
                                                  depth, -$\infty, +\infty$)
    return $best\_m$
end function

function MAX-VALUE($game, state, depth, \alpha, \beta$)
    if $game$.IS-TERMINAL($state$) then
        return $game$.UTILITY($state, player$), $null$
    end if
    $value, move \leftarrow -\infty$
    for all $m \in game$.GET-MOVES($state, player$) do
        $n\_state \leftarrow game$.MAKE-MOVE($player, state, m$)
        $v, m2 \leftarrow$ MIN-VALUE($game, n\_state, depth + 1$)
        $\alpha \leftarrow$ MAX($\alpha, v$)
        if $v \geq \beta$ then
            return $value, move \leftarrow v, m$
        end if
    end for
    return $value, move$
end function

function MIN-VALUE($game, state, depth, \alpha, \beta$)
    if $depth \geq game$.difficulty then
        return $game$.UTILITY($state, player$), $null$
    end if
    if $game$.IS-TERMINAL($state$) then
        return $game$.UTILITY($state, player$), $null$
    end if
    $value, move \leftarrow +\infty$
    for all $m \in game$.GET-MOVES($state, player$) do
        $n\_state \leftarrow game$.MAKE-MOVE($player, state, m$)
        $v, m2 \leftarrow$ MAX-VALUE($game, n\_state, depth + 1$)
        $\beta \leftarrow$ MIN($\beta, v$)
        if $v \leq \alpha$ then
            return $value, move \leftarrow v, m$
        end if
    end for
    return $value, move$
end function
```

C. Monte Carlo Tree Search

Algorithm 3 Monte Carlo Tree Search (simplified)

```
  function PLAY($game, state$)
      $player \leftarrow game$.CURRENT_PLAYER
      $move \leftarrow$ MONTECARLO-SEARCH($game, state,$
                          $max\_rollout\_depth, n\_iterations, 2$)
      $new\_state \leftarrow game$.MAKE-MOVE($player,$
                                              $state, move$)
      return $new\_state$
  end function

  function MONTECARLO-SEARCH($game, state,$
                      $max\_rollout\_depth, n\_iterations, C$)
      $root \leftarrow$ NODE($game, state$)
      for $i \in [0, n\_iterations[$ do
          RUN-ITERATION($game, state,$
                                  $max\_rollout\_depth, C$)
      end for
      $best\_child \leftarrow root$.BEST_CHILD($C$)
      $move, node \leftarrow best\_child$.PARENT
      return $move$
  end function

  function RUN-ITERATION($game, state,$
                                  $max\_rollout\_depth, C$)
      $current\_n \leftarrow root$
      while not $game$.IS-TERMINAL($current\_n.state$) do
          $current\_n$.EXPAND
          $current\_n$.SIMULATE
          $current\_n \leftarrow current\_n$.BEST_CHILD($C$)
      $current\_n$.BACK-PROPAGATE
```

APPENDIX B
MINIMAX VS. MINIMAX WITH ALPHA-BETA PRUNING

TABLE III: Game results of Minimax agent vs. Minimax with Alpha-Beta pruning agent in different difficulty levels and board sizes. Difficulty level indicates the difficulty level applied for both agents, according to the details mentioned in Section II. Results were generated running on an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 16.0GB RAM computer.

Difficulty	Size	Winner	Time (s)	Moves
Medium	3x3	Minimax	0.8	13
Medium	5x5	Minimax-$\alpha\beta$	1.7	26
Medium	9x5	Minimax	1.8	43
Hard	3x3	Minimax	0.6	10
Hard	5x5	Draw	4.1	33
Hard	9x5	Minimax-$\alpha\beta$	91.9	30

APPENDIX C
MINIMAX VS. MCTS

TABLE IV: Game results of Minimax agent vs. MCTS agent in different difficulty levels and board sizes. Difficulty level indicates the difficulty level applied for both agents, according to the details mentioned in Section II. Results were generated running on an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 16.0GB RAM computer.

Difficulty	Size	Winner	Time (s)	Moves
Medium	3x3	MCTS	10.1	13
Medium	5x5	Minimax	8.7	17
Medium	9x5	MCTS	140.1	25
Hard	3x3	MCTS	6.6	10
Hard	5x5	MCTS	46.5	22
Hard	9x5	Minimax	429.6	35

APPENDIX D
AI VS RANDOM

TABLE V: Game results of Random agent vs. different AI agents in different difficulty levels and board sizes. Size refers to the board size. "t" represents the time of the whole game, and t/M represents the average time per move for the players in the game. Results were generated running on a Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 16.0GB RAM computer.

AI	Difficulty	Size	Winner	t (s)	Moves	$\overline{t/M(s)}$
Minimax	Medium	3x3	AI	0.9	13	0.07
	Medium	5x5	AI	1.0	16	0.06
	Medium	9x5	AI	2.5	26	0.10
	Hard	3x3	AI	0.7	12	0.06
	Hard	5x5	AI	2.8	18	0.16
	Hard	9x5	AI	2.5	26	0.10
Minimax-$\alpha\beta$	Medium	3x3	AI	0.9	14	0.06
	Medium	5x5	AI	1.0	16	0.06
	Medium	9x5	AI	1.6	24	0.07
	Hard	3x3	AI	0.7	12	0.06
	Hard	5x5	AI	1.5	18	0.08
	Hard	9x5	AI	6.8	30	0.23
MCTS	Medium	3x3	AI	4.2	10	0.42
	Medium	5x5	Draw	7.3	16	2.19
	Medium	9x5	AI	1.2	36	0.03
	Hard	3x3	AI	6.6	10	0.66
	Hard	5x5	AI	37.8	23	1.64
	Hard	9x5	Random	48.0	39	1.23

APPENDIX E
MCTS VS. MINIMAX WITH ALPHA-BETA PRUNING

TABLE VI: Game results of MCTS agent vs. Minimax with Alpha-Beta pruning agent in different difficulty levels and board sizes. Difficulty level indicates the difficulty level applied for both agents, according to the details mentioned in Section II. Results were generated running on an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 16.0GB RAM computer.

Difficulty	Size	Winner	Time (s)	Moves
Medium	3x3	MCTS	0.8	13
Medium	5x5	Minimax-$\alpha\beta$	14.8	24
Medium	9x5	MCTS	414.1	57
Hard	3x3	MCTS	9.2	10
Hard	5x5	MCTS	68.2	22
Hard	9x5	Minimax-$\alpha\beta$	1279.7	37

REFERENCES

[1] H. A. Raboanary, J. A. Raboanary, and T. H. Raboanary, "Using a genetic algorithm for mining patterns from endgame databases," in *2012 African Conference for Sofware Engineering and Applied Computing*, pp. 94–100, 2012.
[2] D. Walker, *A Book of Historic Board Games*. Lulu.com, 2014.
[3] M. Schadd, M. Winands, J. Uiterwijk, H. Herik, and M. BERGSMA, "Best play in fanorona leads to draw," *New Mathematics and Natural Computation*, vol. 04, pp. 369–387, 11 2008.
[4] H. A. Raboanary, T. H. Raboanary, and J. A. Raboanary, "Towards optimal play fanorona," in *AFRICON 2015*, pp. 1–5, 2015.
[5] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.
[6] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*. Pearson, 4 ed., 2020.
[7] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.
[8] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," *New Mathematics and Natural Computation*, vol. 04, pp. 343–357, 11 2008.
[9] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, "An adaptive sampling algorithm for solving markov decision processes," *Operations Research*, vol. 53, no. 1, pp. 126–139, 2005.
[10] T. Joppen and J. Fürnkranz, "Ordinal monte carlo tree search," in *Monte Carlo Search* (T. Cazenave, O. Teytaud, and M. H. M. Winands, eds.), (Cham), pp. 39–55, Springer International Publishing, 2021.
[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.