
# AI Game Programing Documentation

Alexander Erzmann & Yassin Es Saim

*Winter semester 2024/2025*

**Awalé - Implementing a minimax algorithm with alpha-beta-pruning**

Status:
January 7, 2025

# Contents

# 1. Introduction

As part of the AI Game Programming course within the M1 Informatique degree program at Université Côte d'Azur, students were tasked with designing and implementing an AI algorithm for a competitive final project. The goal was to develop a strategic AI capable of excelling in a customized variation of the traditional Awalé game, a centuries-old board game with deep cultural roots in various regions of Africa.

To address this challenge, the Minimax algorithm enhanced with alpha-beta pruning has been implemented, a method renowned for optimizing decision making in adversarial games. This algorithm evaluates potential moves based on a predefined search depth and heuristic function, aiming to identify the most advantageous moves in any given game state. The quality of AI decisions directly reflects the effectiveness of this implementation.

This document begins with a detailed explanation of the rules governing our variation of Awalé. Next, the game implementation is presented, focusing on how it enforces the rules and sets the stage for the AI's operation. Finally, the heart of the project, the design and refinement of the Minimax algorithm with alpha-beta pruning, is described, along with the strategies used to improve its performance and the ability of AI to compete successfully. A final conclusion rounds off the documentation.

# 2. Game Rules: Awalé Variation

This chapter outlines the rules for the customized Awalé variation used in the AI implementation. These rules define the structure of the game, its objectives, and the mechanics of play, sowing, capturing, and winning.

## 2.1 Game Structure

The game board consists of 16 holes, with 8 holes per player. Holes are sequentially numbered from 1 to 16, with sowing progressing clockwise: Hole 16 follows clockwise hole 16. The first player controls the odd-numbered holes (1, 3, 5, etc.), while the second player controls the even-numbered holes (2, 4, 6, etc.). Note that the numbers used for the holes in the code will be transposed, as arrays are 0-based. There are two seed colors: red and blue. At the beginning of the game, each hole contains 2 red seeds and 2 blue seeds. Note: This differs significantly from traditional Awalé variants.
Figure 2.1 shows the initial setup of a board. In the upper left corner is the hole number 1, in the upper right corner is the hole number 8, in the lower right corner is the hole number 9 and in the lower left corner is the hole number 16.

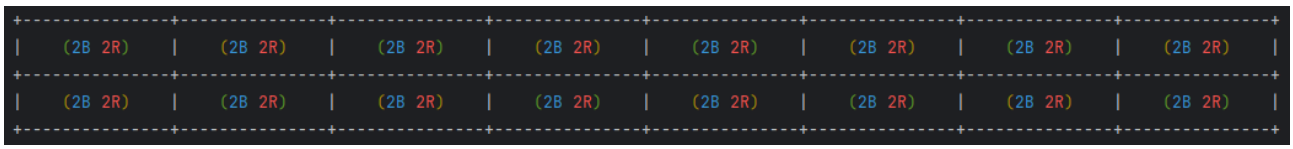


**Figure 2.1:** Initial Board Layout

## 2.2 Objective

The primary goal is to capture more seeds than your opponent. With a fixed total of 64 seeds on the board, a player can win because he has more seeds captured than the other one but there are also more exceptional end reasons, covered in section 2.5.

## 2.3 Sowing Mechanics

The players take turns selecting a hole under their control and distributing its seeds in a process called sowing. The process depends on the color of the seeds chosen for the move:

- Selecting a Color: The player selects a hole and chooses one color (either red or blue) from the seeds in that hole. The selected seeds are removed from the hole, leaving the seeds of the opposite colour, if existing.
- Sowing Blue Seeds: Blue seeds are distributed one by one in all holes, progressing clockwise. The starting hole (where the seeds were removed) is skipped. For example, if a hole contains 16 seeds, the 16th seed is placed in the hole following the starting hole.
- Sowing Red Seeds: Red seeds are distributed only into the opponent's holes, progressing clockwise. The starting hole is also skipped.

## 2.4 Capturing Mechanics

Capturing seeds occurs when a player's move results in a hole containing exactly two or three seeds. This triggers a chain reaction:

- Seeds in the target hole are captured and set aside. If the preceding hole (counter-clockwise) also contains exactly two or three seeds, those seeds are captured as well. The process continues until a hole is reached that does not contain exactly two or three seeds.

Capturing is independent of seed color, and players may capture seeds from their own or their opponent's holes.

## 2.5 Game End Conditions

The game concludes under any of the following conditions:

- One player captures 33 or more seeds, securing victory.
- Both players capture exactly 32 seeds, resulting in a draw.
- Fewer than 8 seeds remain on the board. The remaining seeds are disregarded, and the player with the most captured seeds wins.
- The current player has no valid move left (starving). In this case the opponent captures all remaining seeds on the board and the player with more seeds has won.

# 3. Implementation

This section presents briefly the implemented architecture and classes for the Awale Game and the move deciding alogrithm. To simplify, the application does not use a graphical user interface but uses the console to print the game state. [1].
The classes can be roughly differentiated between game classes, the actual awale board class and player classes. The game classes manage creating a game, holding game relevant objects and managing the game state and turns. The board class represents a awale board, including all the seeds and e.g. rule to sow or capture seeds. The player classes define classes to differentiate between human and AI player but also manage to determine the players turn. In the following a more detailed look in the most important aspects will be given.

## 3.1 Module: Game

An Awale game is saved in the Game class. This contains the board and the two players as well as the Boolean status 'isRunning'. After starting the application, the GameMode is selected. Depending on whether PLAYER_VS_AI_LOCAL, AI_VS_PLAYER_LOCAL or AI_VS_AI_LOCAL was selected, different player instances (see section 3.3) are assigned to the new game instance. This is done using the GameFactory class. The Game class itself is responsible for keeping the current game state (Board, Player1 and Player2 as well as isRunning) and for the correct running of the game. The method performNextMove() is central here, which ensures that Player 1 and Player 2 alternately perform a move. In addition to some auxiliary methods that ensure that only the current player performs his move and that this is done correctly, the system also reacts to the possible end of a game by checking the board for the end of the game and outputting the corresponding data to the console. Ensuring compliance with the correct rules of the game is not carried out in the class itself. It is merely a superordinate interface, which primarily utilises the methods of the board, in which the rules of the game are then also ensured. In Figure 3.1 shows an overview of the relevant classes and their methods.

## 3.2 Module: Board

As already indicated in the previous section, the board primarily executes the players moves regarding the rules mentioned in chapter 2. As shown in Fig. 3.2, the board consists of a

[1] Please note that ChatGPT was used to assist the implementation process and fix bugs faster.

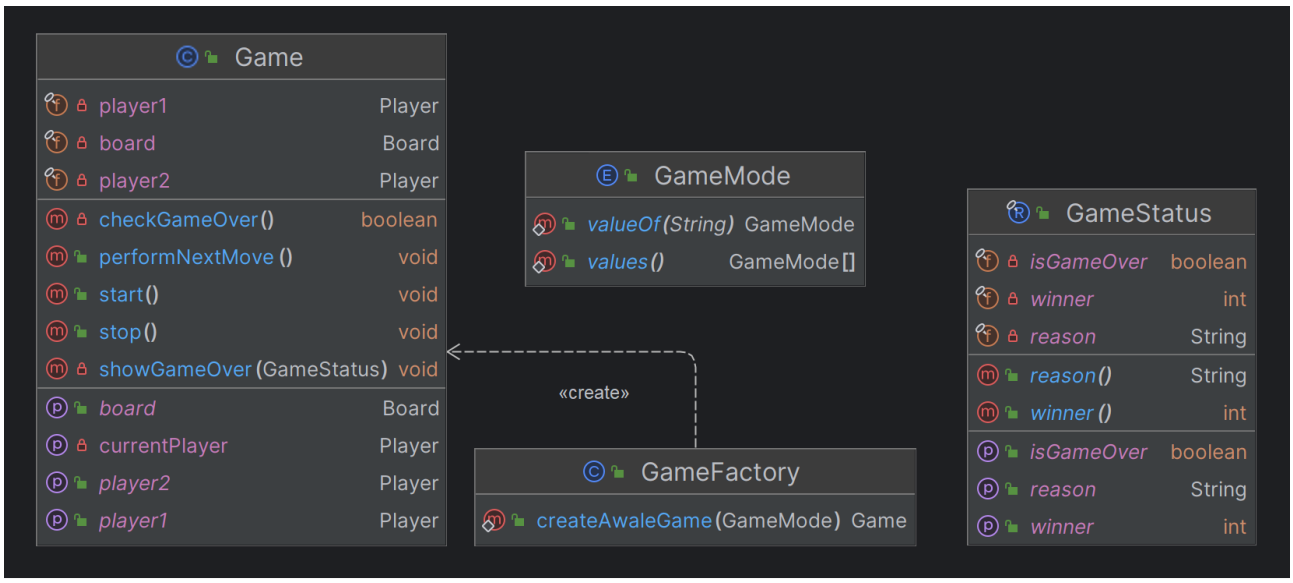


**Figure 3.1:** Game Classes Diagramm

two-dimensional array for storing the holes and the two coloured seed counts, the current player (1 or 2) and the seeds already collected for player 1 and player 2. Also there is the number of moves already made stored here, which was mainly of interest for the AI algorithm in chapter 4.

The board now has all the methods required to make a move. This includes the central sowSeeds(int hole, SeedColor seedColor) for sowing seeds. This method uses a number of hidden helper functions to differentiate between sowing red and blue seeds and to collect the seeds. Before the seeds are sown, however, appropriate queries are made to check whether sowing is permitted according to the rules of the game. For example, the hole must contain seeds and also belong to the current player. There are methods for obtaining the number of seeds (possibly by colour) in a hole. In addition, the change of player is also triggered via the board (see switchPlayer()) and the board has a method for checking all possible end conditions (see checkGameStatus()).

To ensure that the user can follow the course of the game, the current game status, i.e. the board layout, is printed to the console after each move. This is done using methods within the board class that output the board in a clear and readable format.
Last but not least, the board also contains some methods that are interesting for the AI algorithm. There is a method for copying the current board and for sowing seeds for simulation purposes (see sowSeedsForSImulation()). The latter method has analogue auxiliary methods, but with the difference that no actual manipulations are performed on the current board, only the potential number of collected seeds is output. More on this in chapter 4.


**Figure 3.2:** Board Class Diagramm

## 3.3 Module: Player

The last important area concerns the player classes. Basically, it is differentiated between AIPlayer and HumanPlayer. A player has a makeMove(Board board) method, which ensures that a console input is requested from the HumanPlayer and processed accordingly. Checks are made here to ensure that the input is correct and if it matches the required format, the sowSeeds are triggered in the board. In AI, the AIManager is used to determine a move, which is then forwarded to the board for sowing the seeds. The AIManager contains the findMove() method, which determines a move according to the selected strategy. The current options here are findRandomMove and findBestMove, whereby the latter implements the required Minimax algorithm. More on this in the next chapter. Figure 3.3 shows the classes discussed as a UML diagram.

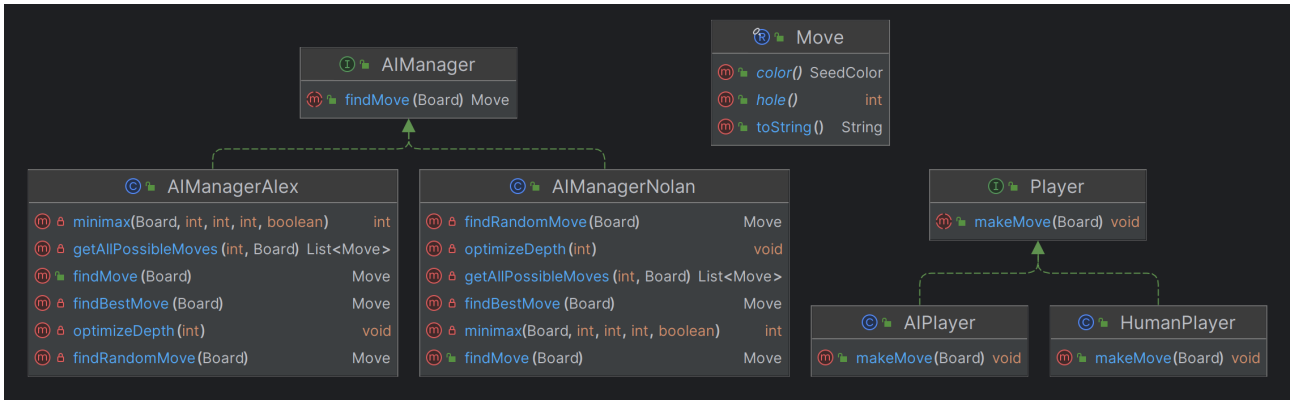


**Figure 3.3:** Player Classes Diagramm

# 4. Algorithm

As described in the previous section, the most important part of the contest, the AI algorithm, is executed in the AIManager class.

As already described, the findMove() method in the AIManager is called to determine a move. Actually there are two different strategies to find a move. findRandomMove() to find a random move and findBestMove() to find the best possible move using the implemented minimax algorithm. The findRandomMove() method simply chooses a random player hole and a random colour repeatedly until a valid combination is found. The findBestMove() method starts with a random move, sorts all possible moves in a certain way, uses the minimax algorithm with alpha-beta pruning to determine a move and returns that move.

This chapter will give more insight into the concrete implementation of the findBestMove() method, including the sorting, the minimax algorithm and the decision made to improve its quality and performance.

## 4.1 Sorting Moves

Sorting the moves is a very important preparation to speed up the alpha-beta pruning, which results in higher possible depths. In this case several ways to sort the moves were tested. Finally the moves were sorted first by considering the seeds captured after this move in descending order (not considering any further moves), and in case of a tie, by considering the seeds in a hole in descending order.

## 4.2 Minimax Algorithm using Alpha-Beta-Pruning

After the moves were sorted, the minimax algorithm will be executed using the moves in the given order. Each move will be executed using the boards methods "...ForSimulation" described in section 3.2 and each state will simply be evaluated using the difference between the current points of player one and two. In case a alpha or beta pruning is possible, this path will break. In case a winning state is reached in the simulation, the maximum or minimum (depending on the player) is returned to highlight this move.

The following example was used as a foundation for the algorithm [Jav24]. This is the pseudocode showing the algortihm:

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
    maxEval = -infinity
    for each child of node do
    if (winning move)
        return +infinity
    eval = minimax(child, depth-1, alpha, beta, False)
    maxEval = max(maxEval, eva)
    alpha= max(alpha, maxEva)
    if beta<=alpha
        break
    return maxEval

else                          // for Minimizer player
    minEval = +infinity
    for each child of node do
    if (winning move)
        return -infinity
    eval = minimax(child, depth-1, alpha, beta, true)
    minEval = min(minEval, eva)
    beta= min(beta, eva)
    if beta<=alpha
        break
    return minEval
```

## 4.3 Optimizing Depth

A final step to increase the speed of the algorithm is to dynamically optimise the depth. In this implementation the depth is defined in terms of the number of possible moves. The less moves a player has, the smaller is the number of nodes in depth one, which means that the algorithm will be faster to process all moves.

A first idea to increase the depth dynamically taking into account the number of moves made, tried to do the same. But this method does not really ensure that the depth is only increased when the tree is less complex. Most likely the number of possible moves will decrease as more moves are made, but this is not necessarily the case.

With the current approach, a depth between 5 and 12 is achievable.

# 5. Conclusion

The decision making AI using a minimax algorithm with alpha-beta-pruning presented in this documentation is by far not perfect but it presents an AI that sorts the moves using a specific methodology, applies the Minimax algorithm taking into account alpha-beta pruning and dynamically adjusts the depth of the algorithm. However there are multiple ways to improve the AI:

One way to improve the algortihm is to improve the sorting of the possible moves. However, it is important that sorting is not too complicated. This is just preliminary work to find possible prunings earlier. If sorting takes too long, this will also have a negative effect on the result.

A second way to improve the algorithm is to optimise the heuristic that evaluates the current state of the board. In the current implementation we only consider the difference between the two player scores. This is more like a greedy algorithm and only considers the current state, but does not identify states with high future potential to get more points for the current player.

All these ways may already increase the achievable depth. In addition the way we dynamically update the current depth could possibly adjusted in a way which ensures the depth is always as high as possible considering the maximum executing time of 2 seconds.

By addressing these areas—move sorting, heuristic optimization, and dynamic depth adjustment—the algorithm can become more efficient and effective, paving the way for a more robust and competitive decision-making AI within the given time constraints.

# List of Figures

# Bibliography

[Jav24] Javatpoint. Alpha-beta pruning, 2024.