AI Game 2025 Report

Kacem Abdelbaki, Thomas Lang

January 12, 2026

1 Introduction

The objective of this project was to develop an autonomous Artificial Intelligence agent capable of competing in a custom Mancala variant. The game is played on a circular board comprising 16 holes, with holes 1 through 8 assigned to the first player and holes 9 through 16 to the second. At the start of the game, the board is initialized symmetrically, with each hole containing exactly two red seeds, two blue seeds, and two transparent seeds. The primary goal is to capture more seeds than the opponent, with the possibility of a draw if both players capture 48.

The game mechanics distinguish themselves from standard Mancala through specific color-based sowing rules. Red seeds are distributed into every hole encountered during the sowing process , whereas blue seeds are distributed exclusively into the opponent's holes. When selecting transparent seeds, the player must strategically declare whether they will function as red or blue seeds. Crucially, transparent seeds retain their identity and are always distributed before any other seeds of the designated color. A capture event is triggered when a sowing action brings the seed count of a hole to exactly two or three, regardless of color. This mechanic is recursive; if the preceding hole also contains two or three seeds, those are captured as well, continuing until the condition is no longer met.

Our architectural choices were heavily influenced by the strict operational constraints of the competition. The bot is required to calculate and execute a move within 2 seconds. Furthermore, the game is finite, bounded by a maximum of 400 moves total. If this limit is reached, the game stops immediately, and the winner is determined by the current seed count. Additionally, the game concludes if the total number of seeds remaining on the board falls strictly below ten.

These constraints necessitated an approach that prioritizes efficient time management and robust defensive play to maintain competitiveness throughout the potential duration of the match.

2 Game Engine Implementation

To facilitate the rapid simulation required by our search algorithm, we developed a lightweight C++ game engine designed to enforce the rules strictly while maintaining high performance. The core architecture relies on an object-oriented approach where the board state is encapsulated within `Field` and `Board` classes, managed by a static `GameRules` logic controller.

2.1 Board Representation

The board comprises sixteen `Field` objects that individually track red, blue, and transparent seeds to support the game's specific color mechanics. These objects utilize helper functions to manage seed updates.

2.2 Move Execution Logic

The centralized playMove() function dynamically adjusts sowing to distribute red seeds sequentially or blue seeds exclusively into opponent holes. This logic also strictly enforces the priority of transparent seeds, ensuring they are distributed before standard colors.

2.3 Capturing and End-Game State

After sowing, the engine captures seeds backwards from the last hole as long as they contain exactly two or three items. It also immediately awards all remaining seeds to the current player if the opponent is left with no valid moves.

2.4 Testing Infrastructure: The Referee System

To ensure our agent could operate reliably under tournament conditions, we utilized a Java-based referee ("Arbitre"). This tool was critical for verifying the stability of our inter-process communication and time management logic.

2.4.1 Process Orchestration

The referee operates as a supervisor, launching two separate instances of the game engine (e.g., `JoueurA` and `JoueurB`) as independent system processes. It manages the game loop by capturing the Standard Output (stdout) of the active player—containing moves like "14TB"—and piping it directly to the Standard Input (stdin) of the opponent. This automated setup allowed us to run the competetion even between different base codes (example: Python based code VS C++ based code).

2.4.2 Time Constraint Enforcement

Crucially, the referee enforces a strict time limit on every move of 3 seconds. The referee utilizes Java's `Future` and `ExecutorService` to interrupt and disqualify any agent that fails to respond within the allocated window, ensuring our IDDFS implementation correctly returns a fallback move when time runs short.

2.5 Communication Protocol

The bot operates as a console application, communicating with the game referee via Standard Input and Output (stdin/stdout). To ensure robustness, we implemented a command parser capable of interpreting:

- **Move Commands:** Strings such as "14TB" or "3R" are parsed to update the internal `Board` state.
- **Control Signals:** "START" triggers the first move search, while "RESULT" signals the end of the match.

3 Search Strategy and Optimization

3.1 Introduction

Our initial approach focused on a standard Minimax implementation to model the adversarial nature of the game. In an effort to enhance strategic depth, we subsequently explored the viability of a Monte Carlo Tree Search (MCTS) architecture. However, experimental analysis revealed that MCTS was inefficient given the strict time constraint of two seconds per move. Furthermore, the competition environment which requires the bot to execute on external hardware precluded the deployment of large, pre-computed opening libraries that are often necessary to bootstrap MCTS performance. Consequently, we adopted an Iterative Deepening Depth-First Search (IDDFS) framework. This approach combines the exhaustive tactical precision of Minimax with the flexibility of iterative search, ensuring the agent can explore deep variations while guaranteeing a valid move is always available when the time limit expires.

3.2 Alpha-Beta Pruning

The core is a standard Minimax algorithm enhanced with Alpha-Beta pruning. This reduces the search space by eliminating branches that are mathematically worse than the principal variation found so far, allowing the engine to search deeper within the same time frame.

3.3 Iterative Deepening Depth-First Search (IDDFS)

To solve the problem of the strict two-second time limit, we chose an algorithm called Iterative Deepening Depth-First Search (IDDFS). Instead of deciding in advance how deep to search (for example, always looking 6 moves ahead), this algorithm works in steps. It first looks 1 move ahead, then 2 moves, then 3 moves, and keeps going deeper until the time runs out.

3.3.1 Time Management Safety

The main reason we used IDDFS is safety and time efficiency. In a normal search, if we try to look too deep and the computer takes 2.1 seconds, we lose the turn. If we only look one level less we maybe only exploit half of the time we could use. With IDDFS, we always have a plan so we can look for the best path until we have no more time and then return the current best plan. If the timer runs out while we are calculating depth 8, we can simply stop and play the best move we found at depth 7 [2]. This makes that the bot can give a valid answer no matter when we ask it to stop.

3.3.2 Improving Speed with Alpha-Beta

Restarting the search from the beginning each time makes the bot faster. This is because of we order the moves. The best move we found at depth 3 is very likely to also be the best move at depth 4. By checking this move first in the new search, the Alpha-Beta pruning logic can immediately see that other moves are worse and skip them (pruning). This indication from the previous level allows the bot to search much deeper than if it started from scratch.

3.4 Transposition Table and Zobrist Hashing

To stop the bot from doing the same work twice, we implemented a Transposition Table. In this game it is possible to reach the exact same board position by playing moves in a different order. So we implement a memory so our bot doesn't treat a previously visited game state as new state.

To save these positions efficiently, we used a technique called Zobrist Hashing. This method assigns a unique random number to every possible number of seeds in every hole. By mixing these numbers together using a math operation called exclusive OR, we create a hash (a unique address) for the current board state. This allows the bot to instantly recognize if it has seen a position before. [4].

We store these fingerprints in a large table along with the score and the best move found for that position. Because our table has limited space, we only replace an old entry if the new result comes from a deeper calculation This ensures that our memory is always filled with the highest quality information.

3.5 Move Ordering and Killer Heuristics

For Alpha-Beta pruning to work efficiently, it is critical to examine the best moves first. If the bot finds a strong move early, it can immediately prune large sections of the game tree that are clearly worse. To achieve this, we implemented a sorting system for every turn. We first examine the move suggested by the Transposition Table (if State is already known), as it was already identified as the best move during the previous search depth.

Next, we prioritize so called «Killer Moves». These are moves that were successful in causing a "cutoff" (refutation) in sibling nodes at the same depth. The logic is simple: a move that is powerful in a very similar board position is likely to be effective here as well. By trying these high-probability moves before the rest, we significantly increase the rate of pruning, often doubling or tripling the effective search speed compared to a random order.

4 Evaluation Function Logic

The leaf nodes are evaluated by `Evaluate.cpp`. The function returns a single integer representing the favorability of the state for the maximizing player.

4.1 Score Scaling and Winning Conditions

- **Victory:** If the game is over, we return a big score (100,000). But we subtract the number of moves played: $Score = 100,000 - \text{moves}$. This makes that the bot to choose the fastest path to victory and the slowest path to defeat.
- **Material Weight:** The ruling argument for if we are in a good or bad position is the current score difference. This is why we multiply the score difference by 200.0.

  $$Eval = (P1_{score} - P2_{score}) \times 200$$

  Reasoning: The big factor allows us later to apply bonuses and malusses for less important things without outperforming the importance of the score difference.

4.2 Positional Heuristics

Beyond raw score, we analyze the board layout to guide the bot toward advantageous states.

4.2.1 Vulnerability Management

The capture mechanics of the game are triggered when a hole reaches exactly two or three seeds. Consequently, holes currently containing one or two seeds represent critical "danger zones" on the board.

- **Defensive Penalty:** If the agent possesses a hole with 1 or 2 seeds, it is immediately vulnerable to an opponent's sowing move that could bring the count to 2 or 3, triggering a capture. We apply a penalty (-40 points) to such states, obligating the bot to either empty these holes or add seeds to them to reach a safe count.
- **Offensive Bonus:** On the other hand, if the opponent leaves holes with 1 or 2 seeds, we interpret this as a tactical opportunity and award a bonus ($+30$ points). This encourages the bot to steer the game toward states where the opponent is exposed to future captures.

4.2.2 Seed Accumulation

Having a large number of seeds in a single hole provides strategic flexibility. It allows to sow seeds across the entire length of the board, potentially reaching the opponent's side to execute deep captures or to alter the parity of distant holes. To encourage this long-term planning, our evaluation function adds a small bonus ($+10$ points) for any hole containing more than 8 seeds.

4.2.3 Mobility and Starvation

Starving the opponent leads to capturing all remaining seeds [1]. This is really powerful in both ways we have to make sure not to fall in this trap and at the same time we try to make our opponent fall into it.

- **Aggression:** If the opponent has 0 valid moves, we apply a massive bonus ($+50,000$), treating it essentially as a win. Restricting opponent mobility (1 move left) is also heavily rewarded ($+600$).
- **Survival:** Conversely, if our own mobility drops to 0, we apply a massive penalty ($-50,000$).
- **General Mobility:** We add a small bonus for having more active holes than the opponent ($+20 \times \text{my\_holes} - 30 \times \text{opp\_holes}$), encouraging flexible board states.

5 Conclusion

Building this AI was a challenge because we had to balance complicated game rules with a strict time limit. We started with a simple MinMax and then tried to improve it by implementing a MCTS. Many missed trys and days later we threw this idea away ant wen't back to a MinMax approach but to use the maximum of our timer we switched to Iterative Deepening (IDDFS) approach. This permitted us to stay always in the time limit but not to loose to much time.

On the technical side, the Transposition Table brings us an edge if we are in repetitive cycles.On the evaluation function we focused heavily on the Starvation rule and by punishing low mobility and rewarding fields with many seeds, our bot plays a safe game. If we are in advantage it tries to wear the opponent down over 400 moves or trap them into having no moves left.

In short, our bot combines deep calculation with (in our opinion) good strategy. It plays a safe, careful game but remains aggressive enough to instantly attack whenever the opponent makes a mistake.

References

[1] Game Rules 2025.txt

[2] GeeksforGeeks. (2025). *Iterative Deepening Search (IDS) in AI.* [Online]. Available: `https://www.geeksforgeeks.org/artificial-intelligence/iterative-deepening-search-ids-in-ai/`

[3] Selman, B. (2014). *CS 4700: Foundations of Artificial Intelligence - Adversarial Search.* Cornell University. [Online]. Available: `https://www.cs.cornell.edu/courses/cs4700/2014fa/slides/CS4700-Games1_v5.pdf`

[4] GeeksforGeeks. (2024). *Zobrist Hashing.* [Online]. Available: `https://www.geeksforgeeks.org/zobrist-hashing/`

[5] Chess Programming Wiki. (2024). *Killer Heuristic.* [Online]. Available: `https://www.chessprogramming.org/Killer_Heuristic`