SYSTÈMES NUMÉRIQUES POUR L'HUMAIN
ÉCOLE UNIVERSITAIRE DE RECHERCHE

EUR DS4H - Master 1 Informatique parcours Intelligence Artificielle

General game playing algorithms implementation for awale

Students :
Antoine De Chabanne Curton La Palice,
Hugo Viana

Professor :
Jean-Charles Régin

Abstract

This project aimed at developing the best possible AI to play the game of awale. We implemented our AI in C because it is a low-level language and efficiency was our priority. We tested 4 different well known general game playing algorithms: Minimax, Alpha-beta pruning with and without iterative deepening and variable depth and Monte Carlo tree search. We tested our algorithms against each others and came to the conclusion that contrary to our original idea, Monte Carlo tree search performed poorly for awale. The best algorithm being the Alpha-beta pruning with iterative deepening thanks to a bigger depth than the other algorithms and variable depth. We tested two evaluation functions: one that just computes the difference in scores and one that adds a coefficient that encourages reducing the number of adversary moves possibilities. The second one performed better than the first.

2026

Contents

1 Game implementation

1.1 Game State Representation

The game state is encoded by the following structure:

```
typedef struct Jeu {
    int rouge[16];
    int bleu[16];
    int transparent[16];
    int score[2];
    bool joueurMachine;
    bool joueurActuel;
    Temps t;
    int nbCoups;
} Jeu;
```

The board is represented using three parallel arrays of fixed size, each corresponding to one seed color. This representation avoids nested data structures and provides constant-time access to the number of seeds of each color in any hole. Given that sowing and capture operations frequently require color-specific manipulation, separating colors into distinct arrays simplifies the implementation and reduces conditional branching in the core game logic.

Players are encoded using a boolean value, allowing player switching through logical negation and direct indexing into the score array. This representation is lightweight and particularly well-suited to adversarial search algorithms, which frequently alternate the active player. A move counter is included explicitly to enforce the hard upper bound on game length and to guarantee termination during AI simulations.

1.2 Move Encoding

Moves are represented by the following structure:

```
typedef struct Coup {
    int trou;
    int couleur;
} Coup;
```

A move is fully characterized by the selected hole and an integer encoding of the chosen color. This encoding unifies standard and transparent moves within a single representation, allowing the AI to treat all legal actions uniformly. As a result, move generation and evaluation remain agnostic to the semantic interpretation of colors at higher levels of the algorithm.

1.3 Initialization and State Duplication

Game initialization is performed by the function:

```
Jeu* initJeu(bool joueurMachine)
```

The initial configuration is deterministic and symmetric, which is essential for reproducibility and controlled experimentation. No randomness is introduced at the game level, ensuring that stochastic behavior originates exclusively from the AI algorithms.

To support adversarial search, game states can be duplicated using:

```
Jeu* copierJeu(const Jeu* src)
```

This function performs a deep copy of the entire state, including board configuration, scores, and metadata. This design prevents unintended side effects when exploring hypothetical futures in Minimax or Monte Carlo Tree Search, where multiple branches of the search tree must evolve independently.

1.4 Sowing Mechanism

Seed distribution is handled by:

```
int distribuerGraines(Jeu* jeu, int trou, int couleur)
```

Instead of scattering rule-specific conditionals throughout the code, the implementation abstracts movement behavior into a single step value derived from the selected color. This allows the sowing loop to remain compact and uniform.

The origin hole is explicitly skipped during sowing, enforcing the rule at the lowest level of the transition function. The function returns the index of the last hole reached, which determines whether a capture phase is triggered.

1.5 Capture Processing and Terminal Enforcement

Captures are resolved by:

```
void capturerGraines(Jeu* jeu, int trou)
```

The capture procedure traverses the board backwards from the final sowing position. At each step, the total number of seeds in the current hole is evaluated independently of color. The capture chain terminates immediately once the condition is violated, resulting in an efficient and faithful implementation of chained captures.

After capture resolution, the implementation checks whether the opposing player has any remaining seeds. If not, all remaining seeds are assigned to the current player, guaranteeing that the game reaches a terminal configuration. Integrating this starvation check directly into the capture logic simplifies correctness reasoning and avoids redundant board traversal.

1.6 Move Parsing and Serialization

Moves can be constructed from textual input using:

```
Coup* coupDepuisString(const char* buffer)
```

This parser is primarily intended for reading the move played by an adversary in the competition.

Moves can be serialized back into a textual form using:

```
char* sortirCoup(Coup* coup)
```

1.7 Terminal Conditions

Game termination is determined by:

```
bool estFinPartie(Jeu* jeu)
```

All stopping conditions are centralized in a single predicate, rather than being distributed across the codebase. In addition to score-based and board-based termination, a hard upper bound on the number of moves is enforced. This constraint is essential for AI-driven play, where repetitive or low-impact actions could otherwise lead to non-terminating simulations.

1.8 Move Generation for Search Algorithms

Legal actions are generated using:

```
int genererCoupsEnfants(Jeu* jeu, Coup** coupsEnfants)
```

Move generation is restricted to holes owned by the current player by iterating over the board with a stride of two. For each hole, all legal color choices are enumerated explicitly, including both interpretations of transparent seeds. This ensures that the full action space is explored without deferring implicit decisions to later stages of the search. The order of the moves is then randomized. Such randomization is particularly useful in Monte Carlo simulations, where deterministic move ordering could introduce systematic bias.

1.9 Random Move Selection

Random playout moves are generated by:

```
Coup* creerCoupAleatoire(Jeu* jeu)
```

This function samples uniformly from the set of legal actions and is primarily used during Monte Carlo rollouts. By relying on the same move generation logic as deterministic search, the implementation maintains consistency across AI paradigms while avoiding duplicated code.

2 Search algorithms

The adversarial search component was developed incrementally, starting from a basic Minimax algorithm and progressively extending it with pruning and time management mechanisms. Each stage preserves the interface of the previous one, allowing direct comparison between variants and facilitating debugging and validation.

2.1 Minimax Search and Root-Level Move Selection

The initial implementation is based on a standard depth-limited Minimax search, implemented by the function:

```
double minimax(Jeu* jeu, int profondeur, double alpha, double beta,
               bool maximisant, double (*evaluation)(Jeu*))
```

This function performs a recursive evaluation of game states using deep copies of the board. At each node, all legal moves are generated, applied to independent copies of the current state, and evaluated recursively. Terminal positions are detected explicitly and evaluated using a dedicated endgame evaluation function, while non-terminal leaves at depth zero are evaluated heuristically.

Minimax alone is not sufficient to select an action at the root of the search tree. For this reason, a separate function is introduced to handle move selection:

```
Coup* choisirMeilleurCoup(Jeu* jeu, int profondeur,
                          double (*minimax)(Jeu*, int, double, double,
                          bool, double (*)(Jeu*)),
                          double (*evaluation)(Jeu*))
```

This function enumerates all legal moves available in the current position, applies each move to a copy of the game state, and invokes the Minimax procedure on the resulting position. The best move is selected according to the returned evaluation values.

A sign correction is applied at the root to normalize scores independently of the active player. This design allows the same evaluation function to be reused uniformly throughout the search tree without embedding player-specific logic into the heuristic.

2.2 Alpha–Beta Pruning

Once the correctness of the Minimax implementation was validated, Alpha–Beta pruning was introduced as a direct optimization. The corresponding function is:

```
double alphaBeta(Jeu* jeu, int profondeur, double alpha, double beta,
                 bool maximisant, double (*evaluation)(Jeu*))
```

This function replaces the recursive Minimax calls while preserving the same interface and evaluation semantics. Pruning is performed whenever the current bounds indicate that further exploration of a branch cannot influence the final decision.

Because the root-level move selection logic is decoupled from the recursive search, the same `choisirMeilleurCoup` function can be reused unchanged with Alpha–Beta pruning simply by passing a different search function pointer. This architectural decision minimizes code duplication and emphasizes the conceptual continuity between Minimax and its optimized variant.

2.3 Iterative Deepening with Fixed Depth

To further improve move ordering and progressively refine evaluations, an iterative deepening strategy was introduced:

```
Coup* choisirMeilleurCoupIteratif(Jeu* jeu, int profondeurMax,
                                  double (*minimax)(Jeu*, int, double,
                                  double, bool, double (*)(Jeu*)),
                                  double (*evaluation)(Jeu*))
```

This function repeatedly invokes the root-level move selection procedure with increasing depth limits, starting from depth one up to a fixed maximum. After each iteration, the move that achieved the best score is promoted to the front of the move list.

This simple reordering strategy improves the effectiveness of Alpha–Beta pruning in subsequent iterations without introducing additional data structures or heuristic tables. Scores are reset between iterations to ensure that deeper searches fully supersede shallower evaluations.

2.4 Iterative Deepening with Time-Bounded Depth

The final extension introduces time management and variable-depth search:

```
Coup* choisirMeilleurCoupIteratifVariable(Jeu* jeu, int limiteTempsInt,
                                          double (*minimax)(Jeu*, int,
                                          double, double, bool,
                                          double (*)(Jeu*)),
                                          double (*evaluation)(Jeu*))
```

In this version, the search depth is no longer fixed a priori. Instead, the algorithm increases depth incrementally until a predefined time budget is exhausted. A timing structure embedded in the game state tracks elapsed time and interrupts the search when the limit is reached.

At each depth, all legal root moves are evaluated unless the time constraint is exceeded earlier. The best move from the deepest fully or partially explored iteration is retained and returned. As in the fixed-depth variant, move ordering is updated between iterations to improve pruning efficiency.

This approach guarantees that a valid move is always produced within the allotted time, while exploiting available computation time to explore the search tree as deeply as possible.

2.5 Monte Carlo Tree Search

After implementing deterministic adversarial search algorithms, a Monte Carlo Tree Search (MCTS) approach was introduced as a complementary decision-making strategy. Unlike Minimax-based methods, MCTS does not rely on a fixed-depth evaluation of the game tree but instead estimates action quality through stochastic simulations.

The search tree is composed of nodes defined by the following structure:

```
typedef struct NoeudMCTS {
    Jeu* jeu;
    Coup* coup;
    int scoreTotal;
    int n;
    struct NoeudMCTS* parent;
    struct NoeudMCTS** enfants;
    int nbEnfants;
} NoeudMCTS;
```

Each node stores a full copy of the game state, the move that led to the node, the cumulative simulation score, and the number of times the node has been visited. This explicit state storage simplifies node expansion and simulation at the cost of increased memory usage, which remains acceptable given the moderate branching factor of the game.

2.5.1 Tree Expansion and Selection

Children of a node are generated explicitly using the same move generation logic as the Minimax algorithms, ensuring consistency between deterministic and stochastic approaches. The list of children is randomized at creation time in order to avoid systematic bias during exploration.

Node selection follows the standard UCB1 criterion, implemented as:

```
double ucb1(NoeudMCTS* noeud, int N)
```

Unvisited nodes are given maximal priority, ensuring that all actions are explored at least once. The exploration constant is deliberately scaled to favor exploration in the early stages of the search, reflecting the high uncertainty typical of this game.

2.5.2 Simulation (Playout)

Once a leaf node is reached, a simulation phase is performed by repeatedly selecting random legal moves until a terminal position is reached or a depth limit is exceeded. This process is implemented by:

```
int plongeon(Jeu* jeu, bool joueurRacine)
```

Simulations operate on deep copies of the game state and rely exclusively on the existing game engine, ensuring that all simulated trajectories are valid according to the rules. The outcome of a simulation is reduced to a win, loss, or draw signal from the perspective of the root player.

In addition to purely random playouts, a guided simulation variant is provided, in which moves are selected using a lightweight heuristic evaluation. This hybrid approach reduces simulation variance while preserving the stochastic nature of MCTS.

2.5.3 Backpropagation

After each simulation, the resulting score is propagated back to the root node using:

```
void retroPropagation(NoeudMCTS* noeud, double score)
```

Each node along the path has its visit count incremented and its cumulative score updated. No discounting is applied, as all outcomes are evaluated from the perspective of the root player.

2.5.4 Time-Bounded Search and Move Selection

The full MCTS procedure is orchestrated by:

```
Coup* choisirMeilleurCoupMCTS(Jeu* jeu, int limiteTempsInt,
                              double (*minimax)(Jeu*, int, double, double,
                              bool, double (*)(Jeu*)),
                              double (*evaluation)(Jeu*))
```

The algorithm iterates selection, expansion, simulation, and backpropagation cycles until the allocated time budget is exhausted. Time management is handled using the same timing mechanism as the iterative deepening Alpha–Beta implementation, ensuring uniform behavior across search paradigms.

At the end of the search, the move corresponding to the most visited child of the root is selected. This criterion prioritizes robustness over raw average score and is consistent with standard MCTS practice.

3 Evaluation functions

Heuristic evaluation functions are used to estimate the utility of non-terminal game states when the search depth limit is reached. As with the search algorithms, the evaluation component was developed incrementally, starting from a minimal baseline and progressively incorporating additional positional features.

All evaluation functions operate on the same game state representation and are expressed from the perspective of a fixed reference player. This design ensures compatibility with the root-level sign correction applied in the search procedures and avoids duplicating player-dependent logic inside the heuristics.

3.1 Terminal State Evaluation

Terminal positions are evaluated separately using the function:

```
double evalFinPartie(Jeu* jeu)
```

This function assigns large positive or negative values to winning and losing positions, respectively, and returns a neutral value in the case of a draw. End-of-game conditions are detected explicitly, including both score-based victories and termination due to insufficient remaining seeds.

By separating terminal evaluation from heuristic evaluation, the search algorithms can handle finished games consistently regardless of the remaining search depth. This also guarantees that decisive outcomes dominate heuristic approximations during backpropagation in both Minimax and MCTS.

3.2 Baseline Material Evaluation

The simplest heuristic evaluation is implemented by:

```
double maxScore(Jeu* jeu)
```

This function evaluates a position solely based on the difference between the two players' captured seeds. It provides a minimal baseline that reflects material advantage without considering positional or dynamic aspects of the game.

This evaluation was primarily used during early development to validate the correctness of the search algorithms, as it is easy to interpret and directly correlated with the game objective.

3.3 Mobility-Augmented Evaluation

To better capture the dynamic nature of the game, a more expressive heuristic was introduced:

```
double evalMinChoix(Jeu* jeu)
```

In addition to the score difference, this evaluation incorporates a mobility term measuring the number of legal actions available to the opponent. Mobility is computed by counting all playable color options in the opponent's holes, with transparent seeds contributing twice due to their dual interpretation.

The mobility term is weighted and combined linearly with the score difference. A sign correction depending on the active player ensures that reduced opponent mobility is evaluated positively from the perspective of the current player. This heuristic therefore favors positions that both increase material advantage and restrict the opponent's future choices.

3.4 Alternative Weighting Variant

A second mobility-based heuristic is provided:

```
double evalMinChoix2(Jeu* jeu)
```

This variant differs only in the relative weighting of the mobility component, which is reduced compared to the previous function. The introduction of this alternative reflects an experimental approach to heuristic tuning, allowing the impact of mobility on decision quality to be evaluated empirically.

By keeping the structure of the evaluation identical and modifying only the coefficients, the implementation facilitates controlled comparisons between heuristics without introducing confounding factors.

4 Best Performing Agent

Among all implemented agents, the best performance was obtained using the time-bounded iterative deepening Alpha–Beta search combined with the `evalMinChoix2` evaluation function. This configuration consistently outperformed plain Minimax, fixed-depth Alpha–Beta, and Monte Carlo Tree Search across all tested scenarios.

The effectiveness of this agent can be attributed primarily to the depth reached by the search under time constraints. Iterative deepening allows the algorithm to exploit the available computation time efficiently while benefiting from improved move ordering between iterations. In practice, this leads to deeper and more reliable exploration of tactically decisive sequences, which are frequent in this game.

The `evalMinChoix2` heuristic further contributes to this performance by prioritizing material advantage while only moderately accounting for mobility. This balance appears to guide the search toward stable and strategically sound positions without overemphasizing short-term positional constraints.

In contrast, the Monte Carlo Tree Search implementation performed significantly worse in this setting. Despite being given comparable time budgets, it failed to consistently identify strong moves and often produced erratic decisions. No definitive explanation for this behavior was identified, and further investigation would be required to determine whether this is due to the structure of the game, the playout strategy, or the specific MCTS parameterization.

Overall, the iterative deepening Alpha–Beta agent with `evalMinChoix2` proved to be the most reliable and effective decision-making strategy developed in this project.