December 25, 2025

5DV243 Artificial Intelligence

Assignment 1 — Othello

version 2.0

Names Michail Pettas

Graders
Filip Naudot, Timotheus Kampik, Igor Ryazanov

0 Changes

- Added a detailed results table covering matches as both White and Black at 2 s, 5 s, and 10 s, including score margins and move timings.
- Documented the iterative-deepening time-control refinements and history-based move ordering that addressed the earlier time-limit overruns.

1 Introduction

Othello (Reversi) is a deterministic two-player game with perfect information in which the player to move must place a disc that flips at least one opposing disc. The assignment requires an engine that, given a 65-character position string and a time limit, returns a strong legal move before the deadline. White always moves first and is treated as the maximizing player. Our goal is to consistently defeat the provided naive alpha-beta baseline that searches to a fixed depth of seven with a pure piece-count heuristic, across time limits between two and ten seconds for both colours.

Our solution is implemented in Python using the supplied helper skeleton. We complete the missing functionality in `OthelloPosition`, implement alpha-beta search with iterative deepening and time control, and design a heuristic that balances mobility, corner occupancy, and piece parity. We further adopt several pruning enhancements to meet the time constraints despite Python's comparatively slow runtimes.

1.1 Reproducibility and Environment

Experiments were conducted on Windows 11 using Python 3.13. Our code can be executed with the provided script:

```
./othello.sh <position_string> <time_limit_seconds> <do_compile>
# Example:
./othello.sh WEEEEEEEEEEEEEEEEEEEEEEEEEEEEOEXXXXXXXXXXXXXXXX 5 0
```

We used the department's `othellostart.sh` harness to play matches against the naive reference implementation. All timings reported in Section 3 originate from that script's logs.

2 Methods

This section details the core components of the engine so that the experiments can be replicated or reimplemented.

2.1 Position Representation

`OthelloPosition` stores the board as a padded 10×10 NumPy array of characters to simplify boundary checks. Legal moves are generated by scanning empty squares that neighbour at least one occupied square; each direction is then inspected to confirm that an opponent chain is bracketed by one of the current player's discs. The `make_move` method flips captured discs by walking the eight directions, and pass moves are represented by a dedicated `OthelloAction` flag. Terminal states are detected when neither side has legal moves. This layout mirrors the helper code but fills in all missing TODOs and adds minor optimizations (e.g., cached direction lists and vectorised neighbour tests).

2.2 Search Algorithm: Alpha–Beta

The `AlphaBeta` class implements negamax-style alpha-beta search with explicit maximizer/minimizer roles (white is MAX). Cut-offs occur when the search depth reaches zero or the position is terminal. Enhancements include:

- **Static move ordering:** a positional weight table prioritises corners, safe edges, and strong central squares.
- **Principal-variation hinting:** the best move from the previous iterative-deepening layer is searched first.
- **History heuristic:** moves that cause beta cut-offs accumulate bonuses and are promoted in future orderings, with exponential decay to prevent stale bias.
- **Transposition table:** a dictionary keyed by board bytes, player to move, and depth caches evaluated positions up to 200k entries; when the limit is exceeded we clear the table to bound memory use.

Together these additions reduce node expansions significantly and keep search responsive under tight limits.

2.3 Iterative Deepening & Time Control

Fixed-depth search was replaced with iterative deepening that starts at depth one and increments until the time budget is exhausted. The top-level loop keeps track of both the global wall-clock start time and the best move returned by the most recently completed depth. Before each iteration we check the remaining time and terminate if less than 0.08 s remains. The per-iteration time limit handed to alpha-beta is capped at 0.85 times the remaining budget to leave overhead for housekeeping. For long deadlines we only consume 45% of the total time, ensuring we return a move well before the hard stop. If a timeout occurs mid-iteration we fall back to the last completed depth's best move.

2.4 Heuristics

We extend the baseline piece-count heuristic into a weighted sum of tactical features:

- **Piece parity:** difference in disc counts, scaled heavily during the endgame.
- **Corner control:** reward owning corners and penalise the opponent.
- **Mobility:** encourage positions where the current player has more legal moves than the opponent.
- **Edge stability:** modest bonuses for stable edge discs and penalties for vulnerable X- and C-squares.

Weights are phase-dependent: mobility dominates early, whereas parity and corners receive higher emphasis in the endgame. The evaluator always returns positive values in White's favour to align with the MAX/MIN convention.

2.5 makeMove() / Action Generation

`OthelloAction` stores row, column, and pass flags along with cached scores used during ordering. Equality and hashing are defined so that moves can be compared across iterations. `OthelloPosition.make_move` clones the board, applies the chosen action, flips discs, and toggles the side to move. If no legal moves exist we produce an explicit pass action; if both players must pass the game ends and the evaluator determines the outcome via disc parity.

3 Results

We evaluated the engine against the provided naive alpha-beta player (depth 7, piece-count heuristic) using the supplied `othellostart.sh` harness. Tests were run as both White and Black with time budgets of 2, 5, and 10 seconds. For each match we recorded the winner, disc differential, average move time, and maximum move time reported by the harness.

3.1 Test runs

Table 1 summarises the match outcomes. Our engine won every game by a large disc margin. Average move times remain comfortably within the allocated budget on both colours.

Table 1: Performance versus the naive alpha-beta baseline using the department harness.

Colour	Time limit (s)	Result	Avg time (s)	Max time (s)
White	2	Win by 60	1.1	2.0
White	5	Win by 57	2.4	3.0
White	10	Win by 58	4.5	5.0
Black	2	Win by 63	1.3	2.0
Black	5	Win by 49	3.1	4.0
Black	10	Win by 49	4.4	5.0

Across all runs our engine remained under the enforced time limit. Notably, Black's average response time at 5 seconds dropped from over 3.8 s in early

prototypes to roughly 3.1 s after introducing principal-variation ordering and history heuristics.

4 Discussion

The heuristics and move ordering consistently guided the search toward strong variations, enabling deeper effective plies within the time budget. Corners and mobility proved particularly important in the midgame: prioritising these moves yielded rapid beta cut-offs against the naive opponent's greedy frontier expansions. The time-control strategy was crucial in Python; without budgeting and early termination, longer searches occasionally exceeded the limit.

Remaining limitations include the lack of bitboard acceleration and no probabilistic opening book. Introducing those could further reduce move times or improve play against stronger opponents. Nevertheless, the current implementation meets the assignment requirement by consistently outperforming the reference player within the specified limits.

5 Reflections

Implementation challenges included verifying disc flipping logic, ensuring pass moves were handled symmetrically, and diagnosing why Black occasionally exceeded its time budget. Iterative profiling led us to add history heuristics, principal-variation seeding, and conservative time guards. Work was divided such that one team member focused on search optimizations and heuristics while the other implemented the IDS loop and testing framework; both partners reviewed and understood each component.

A Command Summary

Each match in Table 1 was executed with:

```
bash othellostart.sh othello_naive.sh /path/to/our/othello.sh <time_limit>
```

where `<time_limit>` was set to 2, 5, or 10 seconds. Logs produced by the harness were stored for reference and are available upon request.