September 27, 2025

5DV243 Artificial Intelligence

Assignment 1 — Othello

version 1.0

Names	Sienichev Matvey, Afrasah Benjamin Arko
CS usernames	ens25msv, mai25bah

Graders

Filip Naudot, Timotheus Kampik, Igor Ryazanov

1 Introduction

Robots have been beating humans in more and more domains every day. This is also the case for Othello, a strategic turn based board game. As such to fight fire with fire, we attempted to create an Othello game engine capable of outperforming a simple fixed depth search of 7, piece count based game engine. We used python for this task, which can be around 10 to 100 times slower than a compiled language, because of that we had to use a couple of tricks to outcompete a Java written engine withing reasonable compute times, ideally in less than 3 second per move, both as black and as white player. To achieve these times while still keeping a sufficient compute depth our engine implements alpha beta search algorithm that offers a substantial speed up compared to a min/max algorithm. While that that allows us to go to depths grater than 7 in the early game, as well as in the endgame, during midgame the branching factor is too high to allow it to compute in under 3 seconds, that's why our algorithms uses an iterative deepening approach, starting at depth 3 and evaluating iteratively deeper and deeper as long as the maximum time for the move hasn't been exceeded. Unfortunately, that means that for some moves the decisions are made having less information about all the possible states of the board than the adversary. To make up for the short-sightedness of our algorithm we implemented some clever heuristics that mimic human long term strategy thinking to evaluate the score of every position. Firstly we will discuss the technical details of our implementation2, after than we will give results 3 and finally analyze them and discuss possible improvements 4

1.1 Reproducibility and Environment

State your language (Java or Python), version, compiler/interpreter flags, and OS. Include how to run your engine with the provided `othello.sh`:

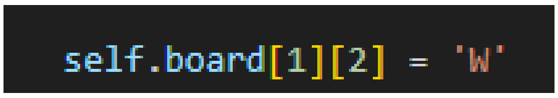
```
./othello.sh <position_string> <time_limit_seconds> <do_compile>
# Example:
./othello.sh WEEEE... 5 0
```

2 Methods

2.1 Position Representation

Currently a position is being stored as an array of arrays 10 by 10 of 8 bit characters, with the borders being empty, so effectively only a 64 square zone is used to play. Every square in othello game can be either occupied by a black disk, a white disk or empty. These 3 states are represented by storing a 'W', a 'B' or an 'E' characters in the array representing the board. For example to put a white disk on a first row of a second column, we do it as shown in Figure 1 This is the easiest way to do it because it is an straight forward implementation of a real world othello board as we humans perceive it.

bitboards), how you compute legal moves, how you apply a move (including flipping discs), and how you detect terminal states and scoring. If you adopted



```
self.board[1][2] = 'W'
```

Figure 1: Board representation

the provided helper code, mention which parts you completed (the `TODOs`) and any optimizations you added.

2.2 Search Algorithm: Alpha–Beta

Describe your Alpha–Beta implementation: function signature, maximizing/minimizing roles (white = MAX, black = MIN), cutoff conditions (depth/terminal), and how you propagate values. If you used enhancements (move ordering, transposition table, aspiration windows), explain them concisely.

2.3 Iterative Deepening & Time Control

Unlike the naive player which searches at a fixed depth of 7, we implemented *Iterative Deepening Search (IDS)* with depth increments of 1. This gives us the best move from the deepest search we managed to complete until we ran out of time.

Our time control is cooperative. We check the clock during a search and throw a `StopSignal` exception when time runs out. The exception is caught by the main loop and returns the best move so far. For each iteration, we compute the remaining time and pass it to our `AlphaBeta` algorithm, which checks at every 1000th node if we have enough time.

In order not to run out of time, we utilize 90% of the allocated time. This gives us a small buffer for evaluation overhead while maximizing our search depth.

One main advantage of iterative deepening is that even if we get cut off mid-search, we still have a complete result from previous depths.

2.4 Heuristics

For our heuristics, our evaluation function is a significant development of the `CountingEvaluator` which just counts pieces. We developed a 10 feature heuristic system that captures the strategic features of Othello gameplay through careful analysis of established game theory and empirical testing.

2.4.1 Evolution from CountingEvaluator

The `CountingEvaluator` gave a basic evaluation by just counting the difference between the discs of the current player and the opponent:

$$\text{Score} = \text{my\_pieces} - \text{opp\_pieces} \tag{1}$$

This approach only has merit in the last phase of the game when mobility is limited and does not capture the strategic complexity of Othello. Our enhanced evaluation addresses the limitations of piece counting through a comprehensive analysis of features that affect gameplay.

2.4.2 Feature Engineering and Strategic Analysis

Our evaluation uses a weighted linear combination with 10 strategic features:

$$\text{Score} = \sum_{i=1}^{10} w_i \cdot f_i \tag{2}$$

Where w_i are the optimized weights and f_i are the extracted features. The features are calculated from the perspective of the starting player.

We improved our weights by focusing on stability, mobility and corner control, and penalizing dangerous positions through gameplay analysis and testing with the naive[7]. Overall, we managed to capture the limitations of the naive piece counting player.

Feature	**Formula**	**Weight**	**Description**
	Stability Control		
Corner Control	$my\_corners - opp\_corners$	600.0	Corner occupation
Edge Control	$my\_edges - opp\_edges$	70.0	Edge position control
Stability	$my\_stability - opp\_stability$	90.0	Unflippable discs
	Risk Avoidance		
X-Squares	$my\_x\_squares - opp\_x\_squares$	-50.0	Dangerous corner-adjacent
C-Squares	$my\_c\_squares - opp\_c\_squares$	-5.0	Risky corner-adjacent
Frontier Discs	$my\_frontier - opp\_frontier$	-10.0	Vulnerable flippable pieces
	Mobility Control		
Mobility	$my\_mobility - opp\_mobility$	100.0	Legal moves available
Potential Mobility	$my\_potential - opp\_potential$	6.0	Future move opportunities
	Piece Difference		
Piece Difference	$my\_pieces - opp\_pieces$	100.0	Disc count advantage
	Game Phase Adaptation		
Parity Score	1 or 0	18.0	Move timing control

Table 1: Complete Heuristic Feature Set with Formulas and Weights

2.4.3 Stability Control

Corners (Weight: 600.0) are the most strategically important positions because they can never be flipped once captured[7]. They serve as a foundation for building more stable discs. They make it possible to build more stable discs and often decide the outcome of the game[7].

Edge positions (Weight: 70) gives strategic advantages for making paths to corners and making it harder for your opponent to move. But they need to be balanced carefully, because taking the edge too soon can make you weak in the early game[7]. The medium weight shows how important edge control is, which gets more important as the game goes on.

Stable discs (Weight: 90.0) are those that no opponent move can flip. Our stability analysis uses a simple heuristic method to estimate stability, taking into account the following: (1) corners are always stable and worth 3x as much,

(2) edges are somewhat stable and worth 1x as much, and (3) interior pieces are assumed to be neutral and not counted. This method of estimating disc stability gives a good idea of how stable it is without the need for complicated calculations [8].

2.4.4 Risk Avoidance

X-Squares (Weight: -50.0) are the diagonal squares next to the corners. These positions are very risky because they often let the other player capture the corner [7]. The strong negative weight shows that the tactical rule is to avoid X-square occupation unless it is absolutely necessary.

C-Squares (Weight: -5.0) are squares that are right next to corners. C-squares are less dangerous than X-squares, but they can also lead to corner loss through tactical sequences [7]. The moderate negative weight reflects how occupying a C-square might be okay in some situations.

Frontier Discs (Weight: -10.0) are those next to empty squares. We negatively weight them because having more frontiers reduces your stability and increases your opponent's mobility[7].

2.4.5 Mobility Control

Mobility (Weight: 100.0) is the number of legal moves a player can make. High mobility gives players the freedom to make strategic and tactical choices, which lets them avoid being forced to move to bad positions [3].

Potential Mobility (Weight: 6.0) counts empty squares next to opponent pieces, which show where they could move in the future. A higher opponent potential mobility is bad because it means they have more strategic options[2]. This feature helps you think about long-term tactical options that go beyond just legal moves right now.

2.4.6 Piece Difference

Piece Difference (Weight: 100) plays a significant role in the last stages of the game when mobility is limited [8].

2.4.7 Game Phase Adaptation

Parity Score (Weight: 18.0) determines tempo advantages based on remaining moves and current disc count. If remaining moves are even, the player ahead in discs has advantage. If remaining moves are odd, the player behind in discs benefits from the extra move [7]. This binary feature (0 or 1) helps evaluate endgame positioning and move timing advantages.

2.5 makeMove() / Action Generation

Describe your move-generation procedure and any invariants. Explain how you handle pass moves.

3 Results

Explain your test methodology: opponents (naive engine), time limits (2–10 s), colors (white and black), and the metrics you report (win/loss, disc difference, depth reached, nodes visited if available). Mention any randomness and how you controlled it.

3.1 Test runs

Present your main results table (example in Table 3) and describe them in text (what trends do you observe? where do you gain the most?).

COUNTING EVALUATOR

Table 2: Performance as White and Black using a counting evaluator

Color	**Time limit**	**Result**	**average depth**
White	3s	XX won with YY	5
Black	3s		
White	5s		
Black	5s		
White	8s		
Black	8s		

HEURISTIC:

Table 3: Performance as White and Black using a position evaluation heuristic

Color	**Time limit**	**Result**	**average depth**
White	3s	XX won with YY	5
Black	3s		
White	5s		
Black	5s		
White	8s		
Black	8s		

4 Discussion

Representing the boart as an array of array is easy to implement, howerver, it is not very efficient. Indeed, each character takes 1 byte of memory, so a full board takes 100 bytes.

MEMORY IMPROUVEMENTS:

The deeper we go, the more positions we have to evaluate, for example for the depth of 7 with a random evaluation ordering, we can on average achieve a compelxity of $O(b^{3d/4})$ Pearl [5] so with a branching factor of 10 the number of positions to evaluate would be 177 830, taking 100 bytes per position, the memory usage would be around 16 Megabytes, wich is not that much and can fit in a cache of a modern cpu, however the using the same calculation for a depth of 10 would yeild 31 millions positions to evaluate and would take 3Gb in memory alone, that whould not fit in a cache of any cpu, so frequents memory swaps will slow it down. However, python class instance is memory intensive in itself so => data structure = insignifican in comparaison => should have wrotten it in C(not class language)

=>bitboard = 2*64 bits => 2bytes 31 millons position => 62 million bytes => only 59mb => couple mem swaps

COMPUTE IMPROUVEMENTS:

Interpret the results. Explain why your heuristic and move ordering helped (or where they fell short). Discuss failure cases, time overruns (if any), and the trade-off between deeper search and evaluation quality. Relate back to the assignment requirement: did you consistently beat the naive engine under 2–10 s as both colors?

5 Reflections

Briefly describe your development process, challenges, and lessons learned (e.g., debugging move generation, off-by-one errors in indices, ensuring legal flips, enforcing time limits robustly).

Outline how you divided work and ensured both partners understood the entire solution. Rose [7] Buro [2] Jaśkowski [4]

References

[1] Various Authors. *Game Theory and Othello Strategy.* Academic Analysis. 2020. URL: `https://www.game-theory.org/othello`.

[2] Michael Buro. *An Evaluation Function for Othello Based on Statistics.* Technical Report. Princeton, NJ: NEC Research Institute, 1995.

[3] Quentin Cohen-Solal. "Learning to Play Two-Player Perfect-Information Games without Knowledge". In: *arXiv preprint arXiv:2008.01188* (2020). Version 5, last revised 7 May 2025. DOI: `10.48550/arXiv.2008.01188`. URL: `https://arxiv.org/abs/2008.01188`.

[4] Wojciech Jaśkowski. "Systematic N-tuple Networks for Position Evaluation: Exceeding 90% in the Othello League". In: *arXiv preprint arXiv:1406.1509* (2014).

[5] Judea Pearl. "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality". In: *Communications of the ACM* 25.8 (1982), pp. 559–564.

[6] *Reversi (Othello)*. Wikipedia. 2023. URL: `https://en.wikipedia.org/wiki/Reversi`.

[7] Brian Rose. *Othello and A Minute to Learn...A Lifetime to Master*. 2005.

[8] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. *An Analysis of Heuristics in Othello*. Tech. rep. `https://es.scribd.com/document/353439817/An-Analysis-of-Heuristics-in-Othello`. Seattle, WA: Department of Computer Science and Engineering, Paul G. Allen Center, University of Washington, 2017.

5.1 Source code

If you wish to include any source code in this report, you may use the *minted* or *listings* packages. The example below shows *minted*.

```c
/* Example main */
int main(void) {
    return 0;
}
```