COMP 424 Final Project Game: *Ataxx*

Authors: Cyprien Armand, Emily Zhang, Austin Wang
Course: COMP 424 - Artificial Intelligence

This document serves as the final project report for COMP 424 at McGill University. It details the design, implementation, and analysis of our AI agent for the Ataxx game.

1. Description of strongest algorithm found

The strongest version of our Ataxx agent we found uses iterative deepening together with alpha–beta search, which is supported by several pruning and ordering techniques that allows the agent through the game tree and gradually increases the search depth until the 2-second limit is reached[1, 2].

We added a transposition table to prevent repeated board evaluations, and we placed strong moves earlier in the search order to strengthen alpha–beta pruning. On top of this, the killer-move heuristic gives priority to moves that produced cutoffs at the same depth in past searches, and the history heuristic highlights moves that have shown good performance across many positions. Together, these techniques created a more efficient search.

In addition, the weights of the evaluation function were tuned using a genetic-algorithm procedure that iteratively tested weight combinations through self-play and selected the highest-performing sets. When the search reaches its depth limit, the agent evaluates the position using features such as material (piece count) and positional strength (how valuable the occupied squares are) [1, 5, 7]. These components allow the agent to choose strong moves reliably within the time constraints.

2. Agent Design and Implementation

The core decision-making of our agent is based on the minimax algorithm with Alpha-Beta pruning seen in class [1, 2]. This approach guarantees the best choice-making assuming we have infinite time and that the opponent also plays optimally.

To handle the strict 2-second time limit we have been constrained by, we implemented an Iterative Deepening Depth-First Search (IDDFS) like seen in class [1, 2]. Instead of searching at a fixed depth (which risked a timeout and then a random move from the game engine), our agent searches depth 1, then 2, etc. If the timer runs out (set to 1.86 in our final version because of the delay of Mimi + code execution), the search is halted, and the best move found in the last fully completed depth is returned. This ensures that we will always have a valid move ready to be submitted, maximizing computation time.

2.1 Transposition Tables

A major inefficiency in game trees is the "transposition" problem, where different sequences of moves lead to identical board states [1, 7]. This creates many states where we would need to check the same board hundreds of times per move, which is very expensive. To address this issue, we implemented a transposition table (TT) [7].

Using a hash map, we store the results of previously analyzed board states (including the depth searched, the score, and the best move). Before we analyze a new move and spend a lot of resources determining whether it is good, we check if that move leads to a state already in our TT. If the position has been evaluated to a sufficient depth, the stored result is returned immediately. This effectively turns the search tree into a search graph, significantly reducing the number of nodes the agent needs to explore [1, 7].

2.2 Heuristic Move Ordering

The efficiency of Alpha-Beta relies on which nodes we visit first. If the best move is examined first, the algorithm can prune a wide range of the search tree [1, 2, 3]. We implemented two advanced heuristics to optimize this ordering in a way that makes the most sense with our approach [3, 4, 7]:

- **Killer move heuristic:** This heuristic assumes that a move causing pruning in the Alpha-Beta algorithm at a specific depth in a previous search is likely to still be strong in future searches. We store up to two of these "killer moves" per depth and prioritize them strongly in subsequent searches [4, 7].
- **History heuristic:** Unlike killer moves, which are specific to a depth, this heuristic maintains a global table of move quality. Every time a move causes pruning, we increment the counter for that target square. Moves with higher history scores are checked first, allowing an evolving heuristic throughout the game to help prioritize which moves to visit first [4, 7].

2.3 The Evaluation Function

While the search algorithm (the muscles) determines how to look ahead and make fast decisions, the evaluation function serves as the strategic core of the algorithm, determining the qualitative value of a state. While other groups may have focused on better search algorithms or maximizing Python's performance, we moved away from simple piece counting and designed a weighted linear combination based on four strategic components [1, 5, 7]. The specific weights of each pillar were determined by our machine learning model (see Section 2.5):

- **Material:** the raw difference in piece count between the two agents [1, 5].
- **Structure (Quads):** We identified that 2x2 blocks of pieces are structurally stable and hard to capture. This discovery led us to use NumPy vectorization (performing operations on entire arrays at once), which instantly detects "quads" across the board [5, 7].
- **Mobility:** the number of valid moves available. A higher score means we have more freedom to play and can either trick the opponent into taking a bait or force them into a corner [1, 5, 7].
- **Positional control:** a map of the board assigning values to each square: corners (high value), edges (medium value), and the center (variable value) [1, 5, 7].

2.4 Other Optimizations

Since Python is "slow" for what we need, we removed any bottlenecks that were avoidable:

- **Vectorization:** We replaced Python loops with NumPy arrays (see Section 2.4) for the evaluation function, allowing us to scan the entire board for specific features.
- **Tuple arithmetic:** We removed object-oriented overhead from the critical search path and instead used simple tuples for coordinates. This increased thinking speed by roughly 300%. We only place the chosen move in the class object once the decision is made, rather than every time we check if a move exists.

2.5 Weights Optimization

The defining feature of our agent is its ability to improve. Instead of guessing values for the evaluation function (e.g., "Is a corner worth 2 points or 3?"), we treated the weights as a "genome" and used a Genetic Algorithm (GA) to evolve the optimal strategy [1, 2].

2.5.1 Genetic Algorithm Pipeline

We created a separate training environment (`training.py`) where agents could play against each other. Each agent possessed a unique set of weights (its DNA) [1, 2].

- **Initialization (Run 0):** We began with a population of randomized weights within reasonable and logical bounds.
- **Selection (Runs 1-9):** In each generation, agents played among themselves. Agents that won more were rewarded with a better fitness score. This concept, seen in class, was the foundation of our entire training.
- **Crossover:** The top two most performant agents were selected to breed. We combined their weights to create a child agent, effectively mixing the strategic strengths of both parents.
- **Mutation:** To prevent the population from getting stuck in a local maximum (as seen with beam search), we introduced random mutations. Occasionally, a weight (e.g., the value for mobility) would be slightly adjusted by a random factor. This mimicked biological evolution and allowed the agent to discover strategies we had not anticipated.

To further refine learning, we implemented what we called "boss fighting" (similar to Curriculum Learning) bounds in our training configuration. Initially, the agent explored wide value ranges to identify strong strategies, and as training progressed, we gradually reduced the bounds to achieve more precise weights (similar to reducing the learning rate in neural network training) [2].

2.5.2 Training Results

We conducted a three-phase training process that evolved alongside our code optimizations. The methodology progressed through stages of increasing complexity:

- **Phase 1 (Runs 1-3):** We began with our baseline "Slow" algorithm searching at Depth 1. This phase established the fundamental behaviors of the agent.
- **Phase 2 (Run 4):** After implementing performance optimizations, we switched to the "Fast" algorithm and increased the search horizon to Depth 2.
- **Phase 3 (Runs 5-9):** For the final optimization, we pushed the agent further by training against Depth 3 opponents to refine its strategic understanding.

The evolutionary process revealed a weighting scheme that deviates from intuitive human strategies. Most notably, the algorithm consistently assigned strongly negative values to center control (-8 to -30), indicating it learned that central positions are vulnerable to multi-directional attacks in Ataxx. Conversely, corner control emerged as critically important with high positive weights (30-55), reflecting their defensive security and strategic value. The evaluation also heavily prioritized 2x2 structural formations (140-160), suggesting that the ML discovered the power of creating stable, interconnected piece clusters that are difficult for opponents to disrupt.

Material advantage maintained a stable, moderate weighting (170-180), serving as a reliable foundation without becoming overly dominant. Edge control and mobility received more variable

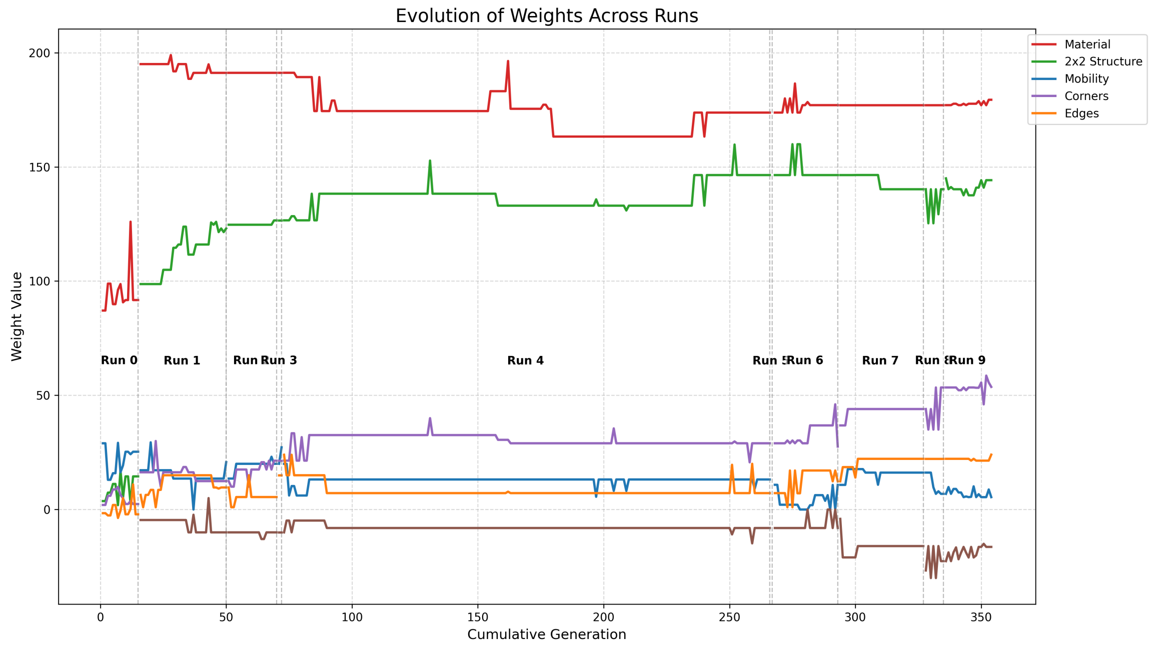

Figure 1: Evolution weights.

but generally positive valuations (7-22 and 5-20 respectively), indicating their situational importance. This evolved scheme represents a nuanced strategic understanding where positional safety, formation stability, and corner dominance outweigh raw material counting.

2.6 Observed results and subsequent modifications

While running our algorithm with all 6 weights, we noticed that some weights, while giving us a very precise evaluation function, were slow to run. For example, 2x2 structures were being checked with NumPy, but as mentioned previously, this library is just not fast enough for our very small time frame. This realization pushed us to remove some parts of the evaluation function. Our final algorithm kept 4 of the 6 weights that we had drawn up; being the positional weights (3) and then material count. As said in class numerous times, depth beats heuristic and that was proved to us while testing our very precise depth 2-3 agent against our newer less precise agent with depth 4-5. The deeper searching agent crushed the depth 2-3 agent at a near 85% win rate proving that the theory was reality [1, 2].

3. Quantitative Performance Analysis

3.1 Search Depth

Across all turns, our agent uses iterative deepening and consistently completes between depth 3 and depth 4 before reaching the time limit. Based on the printed logs, the completed depths were:

$$\{4, 4, 4, 5, 3, 4, 3\},$$

with an average completed depth of approximately 3.454 ply.

The search depth is *not* uniform across branches. Alpha-beta pruning and move ordering cause large parts of the tree to terminate early, while promising lines are explored more deeply [1, 3]. Depth also varies according to the branching factor: positions with 16-41 legal moves reliably reach depth 4, while positions with 74-117 legal moves typically only complete depth 3 due to time constraints.

The main factors affecting search depth were iterative deepening, alpha-beta pruning, move ordering (including TT best moves, killer moves, and the history heuristic), and the use of transposition tables to avoid revisiting states [1, 2, 3, 7]. Depth was also constrained by the fixed time limit, especially in high-branching positions.

3.2 Search Breadth

At each root, our agent logs the number of legal moves. Across the sampled game, we observe the following root breadths:

$$16,\ 29,\ 36,\ 41,\ 74,\ 85,\ 117,$$

with an average of approximately 57 legal moves per position.

Both MAX and MIN have the same raw branching factor, since move generation is symmetric. However, the effective branching factor for our agent is significantly lower due to alpha-beta pruning and move ordering [1, 3]. For example, although a full depth-4 tree with a branching factor above 50 would require millions of nodes, our agent typically explores only 5,000-17,000 nodes per move, which corresponds to an effective branching factor of roughly 5-7.

3.3 Impact of Board Layout

We observed that different board regions naturally lead to different branching factors, which in turn affect how deep the search can go within the time limit.

Corner positions have very low mobility (only three neighbors). This leads to small branching factors (often 10-20 moves) and very effective alpha-beta pruning. As a result, our agent almost always reaches depth 5 in corner-heavy positions.

Edges have moderate mobility (five neighbors), producing about 30-50 legal moves. Pruning is still effective, and the agent usually completes depth 4, though with less margin than in corner positions.

Center positions allow the most clone and jump actions, creating the highest branching factors (up to 117 moves in our logs). These wide trees quickly reach the time limit, and the agent often completes only depth 3 or a shallow depth 4.

We did not add any layout-specific rules to our agent. The differences we observed came entirely from how the branching factor naturally varies across the board. As a possible improvement, we could add layout-aware heuristics or small search extensions that treat high-mobility and low-mobility regions differently. Another possible improvement would be to train the agent with a genetic algorithm on each map separately, allowing it to load different weight sets depending on the layout it encounters during the competition.

3.4 Heuristics, Pruning, and Move Ordering Used

The heuristic provided stable evaluations that guided leaf-node scoring, with material offering the strongest impact. Alpha–beta pruning was the single largest improvement, cutting the search tree size by more than half. The transposition table removed repeated work and significantly improved consistent depth. Killer moves and history ordering improved ordering effectiveness, allowing earlier cutoffs. Iterative deepening improved move quality by allowing deep stable results while staying within the time limit. Together, these methods produced the most considerable improvement in both search efficiency and decision strength.

4. Predicted Win Rates

We tested our agent in 40 game matches against several types of opponents to estimate how it will perform during evaluation. These tests included the random agent, the greedy and MCTS agents, two classmates' agents, and an online Ataxx engine on hard mode.

(i) Random agent. We won all 40 games (100%). Since the random agent often reduces its own mobility, our search algorithm handles it easily. Based on this, we predict a near-100% win rate against the random baseline in the evaluation.

(ii) Greedy and MCTS agents. We achieved 40-0 (100%) against the MCTS agent and 39-1 (97.5%) against the greedy agent. These results show that our pruning and evaluation methods work well against structured play. We therefore predict a 95-100% win rate against the course-provided agents overall.

(iii) Classmates' agents. We tested our agent against five classmates' agents (10 games each) and achieved a combined record of 50–0 (100%). Based on our scripts, our agent consistently searched at depth 3-5, while similarly implemented opponents were limited to depth 1-2. These results suggest that prioritizing a faster, simpler evaluation function to achieve greater search depth was an effective design choice. Based on this, we predict a 90–100% win rate against the pool of classmates' agents.

(iv) Online ataxx engine play. To test ourselves against a standard base, we played 40 games against the online Ataxx engine at `onlinesologames.com` and won all 40 games (100%) [8]. Since hard coded algorithms agents typically do not look as far ahead or manage mobility as consistently, we predict a 95-100% win rate against this opponent.

Across all of our experimental results, the agent performed strongly against random, heuristic, search-based, and human-level opponents, which supports our predictions for the evaluation.

5. Advantages and Disadvantages

5.1 Advantages of Our Approach

- **Self-Correction:** One of the strongest features of our training was its ability to reject extreme values. Early in training, the agent attempted to minimize mobility to 0. However, through natural selection, the population "realized" this was a losing strategy and evolved back toward a balanced value. We did not have to manually tune this; the fitness function naturally selected for robustness.

- **Depth Adaptation:** By shifting our training from Depth 1 to Depth 2 to Depth 3, we forced the weights to adapt to a deeper search horizon. We observed that the agent learned to devalue raw heuristics (such as mobility) in favor of calculation, effectively transitioning from "guessing" to "verifying."

- **Map Generalization:** To prevent the agent from becoming over-specialized (strong only on the standard board), we implemented random map sampling. By forcing the agent to play on maps with walls (e.g., `big_x`, `plus1`), we evolved a weight set that values corners and edges highly, ensuring stability regardless of the tournament map layout.

5.2 Disadvantages and Weaknesses

- **The Trap (Overfitting to Self-Play):** Since our agent only trained against genetic variations of itself, it became hyper-specialized at defeating its own play style. There is a risk that it may struggle against a radically different strategy (e.g., a very aggressive greedy search) simply because it never encountered that style during its evolutionary history.

- **Static Evaluation Limitations:** Our agent uses static weights (e.g., "A corner is worth 53 points"). This lacks context. On a map where a wall blocks a corner, the agent might still value that square highly because its "DNA" tells it to, ignoring the tactical reality that the square may be useless.

- **Evaluation Noise:** In the final stages of training, an opponent might defeat the King (the best version of itself) simply because of a favorable spawn on a map rather than superior intelligence. While we tried to limit this by swapping sides, such "lucky wins" can occasionally pollute the gene pool.

5.3 Expected Failure Modes

- **The Horizon Effect:** Due to time constraints, we optimized our weights for Depth 3. In a tournament setting where the agent searches to Depth 5, these aggressive weights might lead it into a trap that is six moves deep—just beyond its training horizon [1].

- **Time Management Pressure:** Our agent relies on Iterative Deepening. In complex board states with high branching factors, the agent may fail to complete a deeper search within the 2-second limit. If this happens, it is forced to fall back on a shallower, less accurate move, which is the most likely cause of a blunder in the endgame.

6. Future Improvements and Theoretical Optimizations

While our current agent performs well within the constraints of the project, we identified several algorithmic ceilings imposed by our initial design choices. Had we had more time to refactor the core of our solution, we would have moved away from standard Alpha-Beta pruning and Python lists toward more cache-friendly and bitwise operations.

6.1 Internal Representation: The Shift to Bitboards

6.1.1 The issue

The single biggest bottleneck in our current implementation is the representation of the board as a 2D NumPy array. While NumPy is excellent for matrix operations, the overhead of array indexing in Python becomes significant when called millions of times per search. If we were to restart, we would implement bitboards. Ataxx is played on a 7x7 grid (49 squares), which fits perfectly into a single 64-bit integer [7].

6.1.2 Why Bitboards Help

Instead of looping through neighbors to check for captures, valid moves could be calculated using atomic bitwise operations (AND, OR, SHIFT). This would likely increase our node throughput by a factor of 10x to 50x, allowing us to search several layers deeper within the same time limit.

6.2 Search Efficiency: PVS and Aspiration Windows

Our current agent uses standard minimax with Alpha-Beta pruning. While very effective for this class, it assumes a wide search window $(-\infty, +\infty)$. To further optimize pruning, we would implement Principal Variation Search (PVS), also known as NegaScout [7].

6.2.1 The Concept

PVS assumes that our move ordering is good and that the first move examined is likely the best (the Principal Variation). It searches this first move with a full window and then searches all subsequent moves with a "null window" (assuming they are worse). If a subsequent move turns out to be better, we re-search it.

6.2.2 Aspiration Windows

Rather than searching from $-\infty$ to $+\infty$, we could assume that the score at the current depth will be similar to the previous depth. By searching a small window around the previous score (e.g., score ± 50), we could trigger cutoffs much faster [7].

6.3 Transposition Tables with Zobrist Hashing

While we implemented basic caching, a robust Transposition Table (TT) using Zobrist hashing is standard for high-level engines. In Ataxx, many different move orders can result in the exact same board configuration (transposition). By assigning a random 64-bit integer to every piece-square combination and XORing them to create a unique board hash, we could detect these repetitions instantly [6, 7].

This would allow us to store exact scores, lower bounds, and upper bounds, preventing the agent from wasting cycles solving a position it has already solved a few seconds earlier faster than we already do [6, 7]. .

7. Use of LLMs for Training Design (Conditional Section)

We used a Large Language Model only to clarify high-level concepts while designing our solution and our evaluation-weight training pipeline.

7.1 Scope of LLM Usage

LLM queries were limited to conceptual questions about evolutionary algorithms, such as:

> "How does margin-of-victory scoring affect training stability?"
>
> "What mutation schedules are common for small genomes?"
>
> "Is it reasonable to evaluate all individuals on a fixed set of boards?"

7.2 Improving with LLM

We also used LLM to help us figure out where we could make our code faster. This was especially useful in finding out that our mobility calculation, in our evaluation function, or that the helper functions provided to us in the starter code were really slow. The core algorithmic structure on the other hand was purely based off the lecture slides and Bouchit's approach to Ataxx.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, 4th edition. Pearson, 2020.

[2] Jackie CK Cheung and Prakhar Ganesh COMP 424: Artificial Intelligence - Game Playing, Alpha–Beta Pruning, and Search Optimizations. Lecture slides, McGill University, 2025.

[3] Donald Knuth and Ronald Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[4] John Slaney and Sylvie Thiebaux. History Heuristic and the Killer Move: Enhancing Move Ordering in Game Search. Technical Report, National ICT Australia, 2001.

[5] Othello/Reversi AI Heuristics. Standard board-evaluation features including stability, mobility, corner control, edge patterns. Available online: `https://github.com/arminkz/Reversi`.

[6] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, University of Wisconsin, 1970.

[7] T. A. Marsland and J. Schaeffer. Search Techniques. In T. A. Marsland and J. Schaeffer (eds.), *Computers, Chess, and Cognition*, pages 19–54. Springer-Verlag, 1990.

[8] Online Ataxx Engine. `https://onlinesologames.com/ataxx/`. Accessed 2025.