Authors: Megan D'Angelo, Emily Smelyansky, Daria Vorsina
Course Instructors: Jackie CK Cheung and Prakhar Ganesh
Report Due Date: Wednesday, Dec 3, 2025, 8:59PM EST

1 Summary

1.1 Overview of the method

The strongest algorithm created is an algorithm that uses minimax with alpha-beta pruning, iterative deepening, and a transposition table using Zobrist hashing. The alpha-beta pruning helps reduce the search space by not exploring branches that will not change the outcome. This allows the agent to see more moves ahead in the same given time. Iterative deepening helps find the best move with its gradual increase of search depth while making sure it does not surpass the time limit. This guarantees the agent will not return any random move, and finds the best one given the time constraint. The transposition table with Zobrist hashing is another form of optimization: it helps the agent avoid repeating board positions that were already explored previously, helping to increase search depth. All of these components produce an optimized agent with reasonable search depth.

1.2 Play quality

The results in the quantitative section show a 100% win rate against the random agent, the greedy corners agent, and all other previous attempts of building the student agent. The results demonstrate that the agent is able to identify moves better than others in all of these scenarios suggesting it has a robust play quality.

1.3 Reasoning

The design choices for this algorithm are backed up by article readings and conducted evaluations (see quantitative section). According to articles regarding algorithms, alpha-beta pruning and transposition tables are known techniques for improving search efficiency, and the performance report done on them confirms their practical impact on reducing the turn out time. Respecting the time constraint was a big issue when designing the past algorithms despite optimizing the agent with alpha-beta pruning and Zobrist hashing. By introducing iterative deepening and forcing the agent to return a move based on the deepest completed search so far, this guaranteed a move returned in under 2 seconds.

2 Agent Design Explanation

This section explains the theoretical principles underlying the design of our Ataxx agent. It restates the theories and algorithms relevant to our agent and how they map onto our implementations and game.

Ataxx is a deterministic, perfect information, zero-sum game which allows the game to be modeled as a two-player adversarial search problem, where one player aims to maximize their final score while the opponent attempts to minimize it. This makes the minimax algorithm a perfect choice. Minimax searches the game tree by alternating between maximizing and minimizing moves at each depth (Russell & Norvig, 2021, section 6.2). The agent implements minimax using the negamax minimax variant, which takes advantage of the symmetry between players to simplify implementation. It relies on the fact that $\min(\alpha, \beta) = -\max(-\beta, -\alpha)$. So, instead of separate MAX and MIN routines, the moving player looks for a move that maximizes the negation of the value resulting from the move (Negamax, 2025). This variant is still mathematically equivalent and preserves the theoretical guarantees of minimax.

Since unaltered minimax becomes computationally expensive at larging branching factors, and Ataxx has a relatively high branching factor due to each piece having multiple duplicate or move possibilities, we managed this by using alpha-beta pruning to eliminate branches that cannot possibly influence final outcomes. Alpha (α) represents the value of the best (highest value) choice that has been found along the search path for the MAX player, while beta (β) represents the value of the best (lowest value) choice that has been found along the search path for the MIN player. Pruning occurs when $\alpha \geq \beta$ (Russell & Norvig, 2021, section 6.2). The core of alpha-beta pruning is implemented directly in our agent's alpha-beta() function. The function tracks alpha and beta values as usual, and prunes when inconsistencies arise. Due to the negamax implementation, the recursive call receives the negation of the alpha and beta values as arguments in order to swap player perspectives without using separate MAX and MIN routines.

Due to the real-time decision limit of two seconds, the agent must be ready to return the best move found so far. This is achieved by using iterative deepening search, where we start at a depth of 0 and increment the depth limit once a full search has been conducted at the current depth or a solution is found (Russell & Norvig, 2021, section 3.4). If the agent runs out of time before the current search is over, the agent returns the best move found so far. Iterative deepening is implemented in the step() function. The depth variable is originally set at 1 and increases incrementally as the agent completes searches within the time limit. A maximum depth of 8 was set, though it is not functionally necessary and was only added for safety.

In games like Ataxx, the same board state can be reached through different sequences of turns and searching them repeatedly wastes time and computational power. To avoid this, the agent uses a transposition table. In our case, the table is a dictionary keyed with hash values that stores information about previously evaluated positions (Transposition Table -

Chessprogramming Wiki, n.d.). Each entry holds the search depth at which the position was evaluated, the evaluated score, a flag indicating whether the stored value is the result of a full search of the node (EXACT), the score is larger or equal (LOWERBOUND) or the score is smaller or equal (UPPERBOUND), and the best move found from that position. In the agent, the transposition table improves efficiency by reusing previously calculated results when a state is revisited at an equal or shallower depth, and by using stored best moves to improve move ordering.

For the transposition table lookup to be fast and resistant to collisions, we used Zobrist hashing, which assigns (cell, piece) pairs a random 64-bit number. The hash of the board is computed by using the bitwise XOR operation on the relevant random numbers, producing a nearly unique hash in constant time. This method of hashing is fast and its exceptional distribution properties help avoid collisions (Zobrist, 1970). The zobrist_init() function holds the logic for initializing a transposition table with Zobrist hashing. It initializes a three-dimensional Zobrist table using the board size and assigns random 64-bit integers to each cell according to the three possible occupancies (empty, agent 1, agent 2). The hash_board() function applies the XOR operation to these values to produce a stable key for the current board state. This key directly indexes into the transposition table.

Our agent uses move ordering heuristics, as alpha-beta pruning is more effective when the best moves are searched first. We used two move ordering strategies: searching the transposition table for previously successful moves for the same board and ordering that move first with top priority, and scoring moves by simulating them and measuring their immediate gains, placing priority on moves that produce more pieces. Both methods improve pruning, reduce branching factor and allow deeper searches within the move time limit (Russell & Norvig, 2021, section 6.2).

Since it is infeasible to search all the way to terminal states given the time constraints, the agent uses a heuristic evaluation function to estimate the value of non-terminal boards, implemented in the evaluate_board() function. The heuristics we chose for Ataxx are piece difference, corner bonus, opponent mobility penalty, and mobility bonus. Piece difference evaluates the difference between the number of friendly pieces and opponent pieces, corner bonus adds a bonus to moves that control corners as corners have limited adjacency, opponent mobility penalty aims to restrict the opponent's mobility, and mobility bonus aims to increase our mobility. These heuristics are combined into a weighted linear function (Russell & Norvig, 2021, section 6.3). We chose to give the piece difference heuristic a heavier weight as more pieces on the board generally translates to more board control, more future mobility, and more opportunities to constrain the opponent.

All the agent's design choices reflect our goal of creating a smart and powerful agent that can still thrive within the limitations of the assignment. The combination of negamax alpha-beta pruning, iterative deepening, transposition tables with Zobrist hashing, move ordering, and a weighted heuristic function ensures that our agent produces sensible and intelligent moves despite any constraints.

3 Agent's Quantitative Performance

All agent versions developed and presented in this paper beat both random agent and greedy agent. For all versions we relied on heuristics similar and improved compared to those used by the greedy agent. The main concern during our development was turn out time of our agent.
In order to ensure that our agent complies with the requirements of Ataxx combat we made sure our final product's turn out time is consistently under 2 seconds.

Algorithms attempted and the results			
Algorithm	Number of wins for 100 runs against random	Number of wins for 100 runs against greedy	Turn out time
Minimax with $\alpha - \beta$ pruning	100	100	0.0902
Hybrid MCTS and minimax with pruning	100	100	1.6258
Early game Minimax and late game MCTS with pruning	100	100	1.8971
Minimax with $\alpha - \beta$ pruning and hashing	100	100	1.9003

3.1 Minimax with $\alpha - \beta$ Pruning

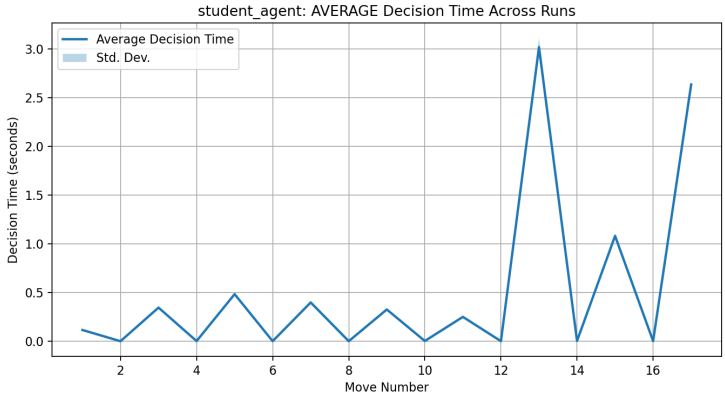

Figure 1: Minimax vs Greedy Corners

Method:

- Standard Minimax search, depth=3
- Optimized with α-β pruning, which reduces the explored game tree size
- Agent uses the same heuristics as greedy corners opponent, and additionally a heuristic which calculates the number of surrounded friendly pieces

Observations:

- Early on in the game, there are few pieces on the board, which makes the game tree relatively small and evaluation fast
- At mid-game, there are more pieces acting and still enough space on the board for many possible moves to be executed. Thus, at this stage, the tree explodes in size
- Late in the game, the number of bad moves that are pruned increases, so the evaluation time can improve. However, if not enough moves are pruned to make it obvious to the agent which next move is best, the large game tree slows evaluation significantly in late-game as well.

Result:

- Even with pruning, minimax struggles due to a high branching factor as the game progresses.
- This agent was our initial default version, to which we later applied optimizations to produce successor versions.

3.2 Hybrid Minimax MCTS Step Agent

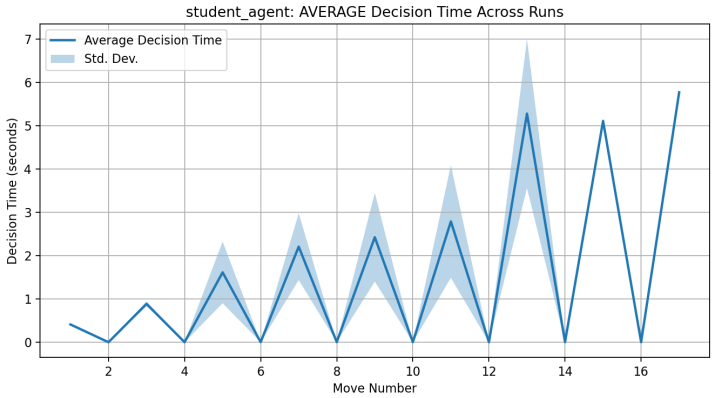

Figure 2: Hybrid Minimax MCTS Step vs Greedy Corners

Method:

At this stage, MCTS was incorporated into step computation. This design choice was motivated by the idea that minimax is an algorithm that predicts the opponent's step to make calculations for its moves. Minimax assumes that the opponent makes equivalently logical choices. This makes a pure minimax overestimate the opponent and miss out on otherwise good moves against a less strategically complex opponent like the greedy-corners agent.

- This implementation uses Minimax of depth=2 and MCTS.

- Optimized MCTS with UCT based scoring for selection, simulated rollout policy, gap and flip heuristics inside rollouts, progressive pruning of low-value children, and move-caching to avoid recomputing valid actions
- Designed to take into consideration the turn out time, aiming to limit turn out time to 2 seconds
- Designed as an MCTS-Minimax hybrid step, where MCTS is used to calculate new game tree states, with fallback to minimax if the evaluation is taking an excessive amount of time.
- To improve speed, several code was refactored to:
  - Reuse the root between turns;
  - Copy minization (board.copy() instead deepcopy());
  - Reduced rollout depth (from 20 to 10)

Observations:

- Time of turn out increases throughout the game starting at under a second in early game to over 4 seconds in late-game.
- Without strategic optimizations described earlier, this implementation can get very computation-heavy.
- This implementation shows the natural behavior of increasing turn out time because the game tree expands as the game progresses.

Result:

- Inconsistent timing, which did not fit in our efficiency constraints suggested that we should reconsider if there was a better way of combining minimax and MCTS strategies in our implementation.

3.3 Early Game Minimax Late Game MCTS Agent

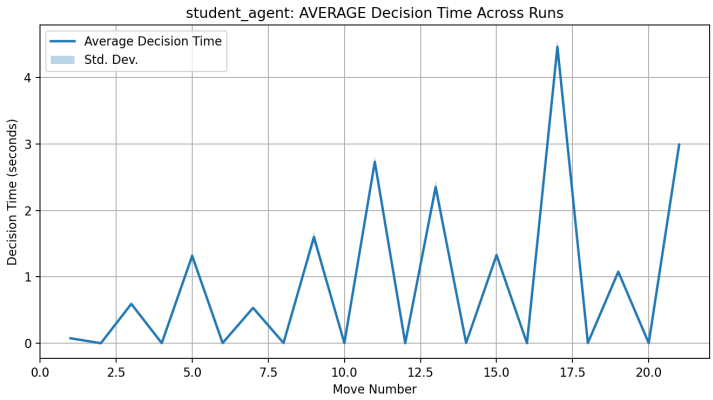

Figure 3: Early-Minimax Late-MCTS vs Greedy Corners

Method:
To try and reduce time costs, we developed an alternative version to test turn out time efficiency using a different strategy to combine minimax and MCTS. Instead of having a hybrid step function which involved both minimax and MCTS evaluation, the use of the strategies became regulated relative to the progress of the game.

- This implementation uses minimax of depth=2 and MCTS.
- Early game, the only strategy employed is minimax
- This implementation introduced the early game threshold constant:
  This value sets the number of pieces that must be present on the board before the game is considered to enter mid-late stages.
- In mid-late game, MCTS is incorporated as the main strategy with minimax fallback
- This implementation (as in the previous versions) uses a time limit to make sure the agent doesn't exceed a maximum turn out time.
- The agent falls back to minimax in mid-late game if there is too little time remaining - if its turn out time for that move exceeds the time limit.
- For efficiency, this implementation keeps the rolling MCTS root, in order to avoid recomputation.
- Additonal optimizations in this version are:
  - Caching moves as in previous versions
  - Use of explicit do and undo move helper functions that enable mutation free rollouts - avoiding frequent board copying
  - Progressive pruning at MCTS selection. This eliminates bad moves and prevents unnecessary game tree traversal

Observations:

- The agent had significant speedup due to the non-algorithmic optimizations used (avoiding board copying for instance)

Results:

- This agent showed significant improvements in efficiency. However, regardless of when we adjusted the early game threshold to be (increasing or decreasing), there was no substantial improvements in turn out times. At its best, the agent would end up reaching 4 seconds in late game because of the tree size.
- When we used pure minimax (with $\alpha - \beta$ pruning), the agent would easily beat the greedy-corners agent at depth=3, but the turn out times were quite high.
- Swapping out pure minimax for a mix of MCTS gave significant efficiency improvements

- Since MCTS (faster than minimax) would take on some computations, the turn out time for the mid and late game was reduced successfully.
- Nevertheless, despite experimentation with different Minimax-MCTS strategy combinations, the acceptable turn out time (2 seconds) could not be achieved.

3.4 Minimax Transposition Table and Zobrist

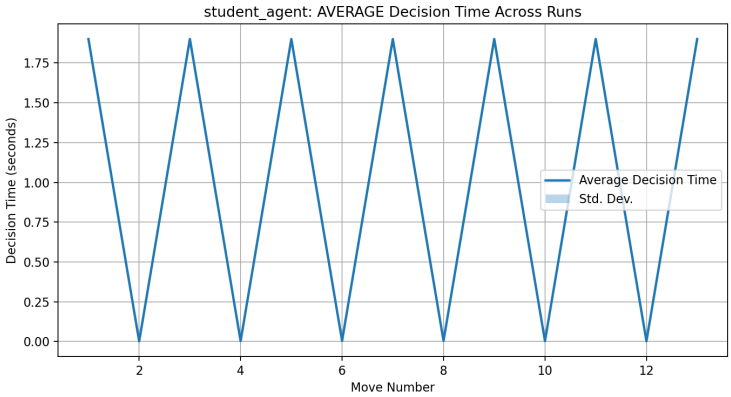

Figure 4: Minimax Transposition Table and Zobrist vs Greedy Corners

Method:

Given that using a hybrid approach of MCTS with minimax was not progressing us towards our goal turn out time, we reverted back to minimax at depth=3.
This agent needed a lot of optimization, so we focused on iterative deepening and using a transposition table with Zobrist hashing.

- Using depth-by-depth search guarantees a valid move even under time constraints
- The agent constructs a Zobrist table, indexed by board coordinates and piece state, effectively producing a unique constant hash for all board states
- Zobrist hashing is important for strong and recalling visited states during $\alpha-\beta$ search. It also reduces redundant computations
- This agent sorts moves using the transposition table's (TT) best move first, and then by heuristic score of the move. This increases likelihood of early β cutoffs - less of the tree is searched

Observations:

- The agent has very stable performance over the span of many rounds of play against the greedy corners agent

Results:

- The agent is always consistent in taking a little over 1.90 seconds per turn out

3.5 Win-rates predictions

3.5.1 (i) random agent

The results show that the random agent is beaten 100% of the time. The prediction is that the student agent always beats the random agent.

3.5.2 (ii) average human player

This part is subjective to whether the human player has knowledge about strategies in Ataxx. If a human player knows the importance of mobility, taking control of the corners, and having layers of pieces surrounding their own pieces, then they are at a somewhat equal standing as the student agent and can at least win 50% of the time. This prediction was also based on having each teammate play against the agent about 10 times, where half the rounds were won by the agent despite the teammates being aware of the heuristic used.

3.5.3 (iii) classmates' agents

Given that the primary focus was on optimizing the agent rather than building the strongest heuristic possible, and by looking at the win percentage grid between students for the practice tournament, we predict a 50% win rate against other students.

4 Trade-offs

4.1 Advantages

- Optimality: alpha-beta pruning drastically reduces turn out time by skipping the exploration of useless moves
- Efficiency: The transposition table stores the moves explored and the scores associated with them, avoiding redundancy, and allowing deeper searches
- Safety: Iterative deepening ensures a move is always available within the time constraint by returning its deepest completed search. This guarantees the agent won't surpass the time given and still returns a calculated move
- Strong heuristic: By combining multiple heuristics such as piece difference, mobility, corner bonuses, the agent is able to evaluate accurately which move is better

4.2 Disadvantages

- Memory usage: If the board becomes significantly big, the transposition tables will have to hold more data regarding the moves and scores, which could drastically increase the memory used for the agent
- Lack of stochasticity: Unlike MCTS, this algorithm can only rely on a heuristic without simulating the plays and receiving the rollout statistics. This could affect the agent's performance if this heuristic is not the most ideal

5 Further improvements

One way the agent could have been improved is to design a better heuristic that takes more strategic factors into account. An example of this would be to evaluate the number of agent's pieces that surround other player pieces. This would have been a great heuristic to add on top of the score of a move because a piece that is protected by other adjacent pieces is harder or even impossible to capture. It's even harder when the piece is surrounded by a second layer of other player pieces (LinuxOnly, n.d.).

Another way the agent could have been improved involves refining the weighting of the heuristics. Some heuristics play a more important role in what would be considered an optimal move, so it would only be logical to increase its score when evaluating the move. The weights would have also been adjusted throughout the game depending on the progress. For example, mobility is crucial in the first half of the game because that's when there are multiple ways of conquering opponent pieces, and taking opportunity to conquer key regions such as corners. Dynamically adjusting the weights depending on the state of the board or the number of moves done could produce a more optimal version of the agent.

Another suggestion would to be to implement Monte Carlo Tree Search into the existing algorithm. This would be appealing because there are many moves possible, which means the branching factor is high. Deterministic rollouts would help approximate whether the player or the opponent will dominate the board. MCTS would provide estimates of many possible moves rather than solely relying on a heuristic. A previous attempt of incorporating MCTS has been done, but it has been removed due to the agent constantly exceeding the time limit in each move. However, this could potentially be solved by cutting early the exploration of undiscovered or least played moves and skip to the exploitation phase, while still fine tuning MCTS in a way to still give it the opportunity to explore. Balancing the exploration-exploitation trade-off would allow the agent to explore promising moves more thoroughly while collecting enough alternative moves to avoid potentially missing out on optimal moves.

An idea that has not been considered, but that the agent could benefit from in terms of optimization, would be to store strong moves or opening sequences before the game even starts. This would have been done by analyzing and compiling what common winning moves are done by the agent in a certain scenario of the gameplay between all the other agent opponents. This would help with optimization because the agent won't have to take time to evaluate the board and calculate the heuristic, just directly apply the stored move.

Overall, the agent uses a strong search algorithm using iterative deepening that respects the time limit, and optimizes using a transposition table and pruning. Nevertheless, there are many more directions that could have been explored to develop more accurate heuristics, more adaptive board evaluations using MCTS, and storing general best move sequences.

REFERENCES

Bouzy, B. (2011). *Move Pruning Techniques for Monte-Carlo Go.* Université René Descartes, Paris. Retrieved from https://helios2.mi.parisdescartes.fr/ bouzy/publications/bouzy-acg11.pdf

GeeksforGeeks. (2023, January 16). *Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning).* Retrieved from https://www.geeksforgeeks.org/dsa/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

LinuxOnly. (n.d.). *Heuristics.* Retrieved from http://www.linuxonly.nl/docs/4/92_Heuristics.html

Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach, Global Edition.* Pearson Higher Ed.

Stack Overflow. (2018, November 19). *Minimax with A-B pruning and transposition table.* Retrieved from https://stackoverflow.com/questions/53377659/minimax-with-a-b-prunning-and-transposition-table

Transposition Table − Chessprogramming wiki. (n.d.). https://www.chessprogramming.org/Transposition_Table

Wang, P., Eisenberg, L., & Vadera, K. (n.d.). *Perversi.* CS 221 Othello Project. Retrieved from https://xenon.stanford.edu/∼lswartz/cs221/perversi.pdf

Wikipedia contributors. (2025, August 21). *Negamax.* Wikipedia. https://en.wikipedia.org/wiki/Negamax

Zobrist, A. L. (1970) *A New Hashing Method with Application for Game Playing.* Technical Report, Univ. Wisconsin. https://research.cs.wisc.edu/techreports/1970/TR88.pdf