Explanation of implementation

MCTS

The agent utilizes the Monte Carlo Tree Search (MCTS) algorithm for the opening and mid-game phases, adhering to the standard "Selection, Expansion, Simulation, Backpropagation" cycle.

Selection

The `find_pv()` function determines the principal variation (PV) path. Starting from the root, it iteratively selects the child node with the highest Upper Confidence Bound (UCB) score using `find_best_ucb()`. To optimize memory usage, nodes only store the `ply` (move) required to transition from their parent. Consequently, `do_move()` is called during traversal to reconstruct the full board state of the selected leaf node.

Expand

Upon reaching a leaf node, the `expand()` function generates all legal moves from the current position. It initializes child nodes for these moves and links them to the tree. If the game has already ended at the current node, the function returns `false`, preventing further expansion.

Simulation

A two-phase simulation strategy is employed to balance breadth and depth. First, a small fixed number of simulations (`INITIAL_SIMULATIONS`) are run for *every* newly expanded child to initialize their statistics. Subsequently, the algorithm performs a larger batch of simulations (determined by `SIMULATION_PER_CHILD`) by consistently selecting the child with the highest UCB score for the rollout.
To support the All-Moves-As-First (AMAF) heuristic efficiently, I implemented the optimization described in the course slides (Lecture 10, p.22). A `played_moves` table tracks moves made during the simulation. Instead of resetting this table after every rollout (which is costly), I use an incrementing `iter` variable. A move is considered "played" in the current simulation only if its stored value matches the current `iter`.

Back propagation

Backpropagation occurs immediately after each simulation. The standard MCTS statistics (`Ntotal`, score sums) are updated for the visited nodes. For the AMAF heuristic, the algorithm iterates through the siblings of the visited nodes; if a sibling's move corresponds to an entry in the `played_moves` table for the current iteration, its AMAF statistics (`N_AMAF`, `sum1_AMAF`) are also updated.

Enhancement

Heuristic Move Selection (Weighted Random)

To improve the quality of random rollouts, I defined a `yummy_table` in `simulation.h` that assigns heuristic weights to different move types (e.g., captures are weighted higher). The `strategy_weighted_random()` function calculates the cumulative weight of all legal moves and uses the PCG random number generator to perform a weighted selection, guiding the simulation toward more realistic play patterns while maintaining necessary randomness.

Granular Win/Loss Scoring

To differentiate between a "decisive victory" and a "narrow win," the simulation return score is calculated as a base win score plus the difference in piece count between the agent and the opponent. I initially attempted to use piece values for this evaluation, but it increased computation time significantly, reducing the simulation throughput from ~80,000 to ~35,000. The piece count method provides a faster, albeit slightly less accurate, metric that allows for a greater total number of simulations.

Depth-i enhancement: Hybrid Search (Alpha-Beta for Endgame)

A common issue with MCTS in this domain is "wandering," where pieces move aimlessly despite a guaranteed win being available. To resolve this, I implemented an `early_termination` check that detects endgame states mentioned on P.33 in slide 10 and switches the agent to an Alpha-Beta search.

The Alpha-Beta evaluation function (`pos_score`) calculates the material advantage (`Piece_Value`) minus the minimum Manhattan distance to enemy pieces. This distance penalty acts as a "compass," incentivizing the agent to close the distance and capture opponents rather than idling. Furthermore, a Transposition Table using Zobrist hashing is utilized to cache board states, improving search efficiency and preventing the agent from entering cyclic move loops.

Experiment results

Configuration Notation The experiments evaluate different MCTS configurations denoted by X+Y, where:
X = INITIAL_SIMULATIONS (Simulations run immediately upon expanding a node)
Y = SIMULATION_PER_CHILD (Simulations run per child node during the simulation phase)

Matchup (Agent A vs Agent B)	Games Played	Agent A Wins	Agent B Wins	Draws	Win Rate (A vs B)	Score (A - B)
1+0 vs 5+25	90	14	29	47	15.5% - 32.2%	37.5 - 52.5
5+100 vs 5+25	90	20	24	46	22.2% - 26.7%	43.0 - 47.0
5+100 vs 5+150	60	15	9	36	25.0% - 15.0%	33.0 - 27.0
5+100 vs 5+200	90	19	17	54	21.1% - 18.9%	46.0 - 44.0

Analysis

The experimental data suggests the existence of a "sweet spot" for the SIMULATION_PER_CHILD parameter. While the 5+25 configuration significantly outperforms the baseline 1+0, further increasing the simulation count per child to 100, 150, or 200 does not yield consistent performance gains. In fact, the 5+150 configuration showed a marked degradation in performance compared to 5+100. This is likely attributable to the strict 5-second time limit per move; an excessive number of simulations per node consumes valuable computation time, thereby restricting the overall depth of the MCTS tree and limiting the agent's strategic horizon.

Implementation Challenges and Debugging

During the development of mcts.cpp, a critical stability issue was encountered involving the All-Moves-As-First (AMAF) update logic. The backpropagate function failed to verify if the played_moves array was null before access. This array is passed as nullptr when backpropagate is called from terminal_update (where no simulation moves occur), causing intermittent segmentation faults and agent crashes.

Impact on Preliminary Data

Interestingly, during the period when this bug was active, preliminary test results for 1+0 vs 5+25 were the inverse of the final results (i.e., the baseline appeared to beat the enhanced agent). This anomaly was probably caused by the enhanced agent frequently crashing or timing out due to the segmentation fault, resulting in automatic forfeits.

Resolution

The issue was isolated by analyzing a replay file from a game that experienced an unexpected timeout. Using GDB, the crash was traced to the specific line accessing the null played_moves pointer:

```
1c1k1e2/3rr1a1/2pc1a1E/3C4 r
Program received signal SIGSEGV, Segmentation fault. 0x000055555559c1c5 in is_move_in_simulation (node=..., played_moves=0x0, iter=3) at mcts/cpp/mcts.cpp:145 145 returnplayed_moves[type_index][node.ply.from()][node.ply.to()] == iter; (gdb) print played_moves $5 = (const long long (*)[32][32]) 0x0
```

The bug was resolved by adding a conditional check to ensure played_moves is not null before attempting AMAF updates.