Adam Adri

CS457

Fall 2024

Project 3: Checkers

Introduction

The objective of this project was to develop an intelligent Checkers player capable of competing against a variety of AI opponents. Using Java and C, the goal was to implement advanced artificial intelligence techniques, including the Minimax algorithm with Alpha-Beta pruning, iterative deepening, and heuristic evaluations. This write-up explains the development process, the challenges encountered, the testing methodologies employed, and the final strategies that led to a competitive checkers player.

Development Process

I began by exploring the provided framework, which consisted of a C-based (myprog.c) and a Java-based (MyProg.java) skeleton for building a smarter AI that defaulted as a random move generator. Ensuring compatibility on my M1 MacBook required troubleshooting dependencies, particularly updating the Makefile to address issues with OpenMotif libraries. Once the environment was set up, I focused on understanding the existing codebase and planning the enhancements needed for an AI player.

To create a competitive AI, I chose to implement the Minimax algorithm with Alpha-Beta pruning. This algorithm allows the program to explore optimal moves efficiently by pruning branches that won't affect the final decision. Additionally, I incorporated iterative deepening to manage computation time dynamically, ensuring the AI could adapt to time constraints per move. For the evaluation function, I started with a simple material advantage heuristic, assigning values to pieces and kings and penalizing opponent pieces accordingly.

Using a 2D array to represent the board state, I utilized bitwise operations to determine piece properties such as color, type (regular piece or king), and position. This efficient representation allowed quick access and updates during move simulations. There existed functions to generate all legal moves for both regular pieces and kings, including recursive functions to identify multi-jump scenarios, which are crucial for capturing multiple opponent pieces in a single turn.

The evaluation function (Eval) was designed to quantify board states based on material advantage and piece positioning. By assigning scores to different board configurations, the AI could rank potential moves during the search process. To enhance the efficiency of the Minimax algorithm, I integrated Alpha-Beta pruning, which reduces the number of nodes explored by eliminating branches that cannot influence the final decision, thus saving computation time. Incorporating iterative deepening allowed the AI to balance search depth with time constraints, starting with a shallow search and progressively deepening it while keeping track of the best move found so far within the allotted time per move.

Enhancements and Optimization

Time management was critical to prevent the AI from exceeding the allowed time per move, which could lead to disqualification. I used system clocks to monitor the time spent on each move and implemented checks to halt the search when the time limit was approaching. Through testing, I realized that the initial heuristic was too simplistic. I enhanced the evaluation function by emphasizing king promotion, encouraging moves that would lead to a piece becoming a king, and by rewarding control of the center and forward movement. I also introduced penalties for moves that led to repeated board positions to avoid endless loops.

When the AI was getting stuck in loops, I implemented a mechanism to track board positions using a hash map. If a board state occurred more than twice, the evaluation function would apply a penalty, discouraging the AI from repeating the same moves. This introduced issues with shared mutable states in recursive functions. To resolve this, I adjusted the code to pass a copy of the board history to each recursive call, ensuring that move evaluations were accurate and that repetitions were effectively penalized.

I also implemented move ordering by prioritizing capture moves and assigning heuristic scores to moves based on their potential impact. This allowed the AI to prune more branches earlier in the search, enabling deeper searches within the same time constraints.

Testing and Validation

To evaluate the AI's performance, I tested it against various opponents provided in the other_players directory, including random bots and depth-limited bots. Matches were conducted using the command “./checkers "java MyProg" ./other_players/opponent 3”, where opponent refers to specific bots like d5 and others. I logged the game results, focusing on move legality, execution times, and outcomes. I played around with the time limit at the end of the command consistently seeing how my bot performed with a .05, .1, .5, 1, and 3 second time limit.

Challenges Encountered

Early in development, the AI occasionally attempted illegal moves, especially with multi-jump sequences. This required revisiting the move generation logic to ensure all generated moves complied with the game's rules. The AI sometimes exceeded the time limit per move, leading to automatic losses. Adjusting the iterative deepening strategy and optimizing the evaluation function helped mitigate this issue. The AI also got stuck in loops, repeating the same board positions. Implementing board position tracking and penalizing repetitions in the evaluation function helped break these loops.

Final Heuristic Adjustments

After extensive testing, I fine-tuned the evaluation function to improve performance. I increased penalties for repetitions to avoid loops, enhanced positional evaluation by adding considerations for piece advancement, control of the center, and king safety, and included a mobility assessment to evaluate the number of legal moves available to both players, rewarding positions with higher mobility.

Results and Analysis

I conducted a series of matches against different opponents to evaluate the AI's performance. The AI performed exceptionally well against simpler opponents, with a 100% win rate against the random bot, even with the time limit being very low (I tested down to .05 seconds). Against stronger opponents, the win rate decreased but remained competitive, achieving a solid win rate against the depth-limited bot at depth 5. At first, I was expecting a higher win rate at a lower time limit and was confused why my bot wasn’t performing too well, but at 3 seconds it seems to consistently win.

The average move time was approximately 2.8 seconds, and moves exceeding the time limit were reduced significantly after optimization, falling to less than 5%. Enhanced heuristics improved decision-making, especially in mid to endgame scenarios, and time management optimizations allowed deeper searches without exceeding time limits.

Discussion

The AI effectively utilized the Minimax algorithm with Alpha-Beta pruning, efficiently exploring optimal moves by pruning irrelevant branches. Dynamic time management through iterative deepening ensured that the AI used the available time effectively, adjusting the search depth as needed. The improved evaluation function, incorporating positional factors and mobility assessments, led to better move selection.

However, in highly complex board states, the AI sometimes couldn't search deeply enough within the time constraints to find the best moves. While improved, the evaluation function could benefit from further refinement to include more advanced heuristics. The project highlighted the importance of move ordering in enhancing the efficiency of Alpha-Beta pruning and the necessity of managing shared state in recursive functions to prevent bugs and ensure accurate evaluations. Finding the right balance between search depth and the breadth of exploration proved to be very important for performance.

Conclusion

Developing this AI Checkers player was a valuable learning experience in applying AI algorithms to game playing. By implementing Minimax with Alpha-Beta pruning and iterative deepening, and by enhancing the evaluation function, the AI became a competitive opponent capable of challenging various bots. The project highlighted the significance of efficient algorithms, effective heuristics, and debugging. While there is always room for improvement, especially in refining heuristics and exploring advanced techniques, the AI demonstrated strong performance against a range of opponents.