Jaden Dawdy

Professor Andersen

CS 457, Intro to AI

November 15, 2024

AI Checkers Project

Goal

The goal of this project was to develop an AI program that is efficient and does well at playing the game checkers. It should be able to always beat randomized bots, and also beat a depth 5 search bot majority of the time. It should also be able to beat a non optimized alpha beta pruning bot most of the time as well.

Initial Assessment

The original codebase provided only basic game mechanics with random move selection. Key limitations included:

No position evaluation

No strategic planning

No search capabilities

No endgame understanding

Pretty much random placement of pieces.

Development Process

Phase 1: Basic Evaluation Function

Initially implemented fundamental piece valuation:

Basic piece counting with weighted values for regular pieces and kings

Simple material advantage calculation

Results: While better than random moves, the AI made obvious tactical mistakes and lacked strategic understanding. It was just very slightly better than random with a win rate of about 65%

Phase 2: Strategic Position Evaluation

Added positional values to improve its defense:

Back row protection bonus to promote defensive positioning

Middle board control bonus to promote advantageous positioning

Protected piece bonus to reward when pieces are in safe positions

Edge piece recognition for additional safety

Results: Discovered that a more defensive playstyle where keeping pieces out of danger is more optimal. Win rate improved to ~80%

Phase 3: Search Implementation

Implemented alpha-beta pruning with several enhancements:

Iterative deepening to maximize search depth within time constraints

Added move ordering to improve pruning efficiency

This point I realized I was not handling the time correctly and the player was taking its turn before the time ended. Adding this allowed it more time to make decisions.

Results: It originally would immediately take its turn without utilizing the time given, so the SecPerTurn was fixed to work properly. Then the searching function was too slow and was solved quickly by implementing move ordering to improve pruning efficiency. Win rate was ~95%.

Phase 4: Advanced Evaluation

Added sophisticated evaluation features based on game state:

Endgame detection with specialized evaluation

Mobility scoring to encourage flexible positions

King mobility bonus for better king utilization

Promotion path evaluation to encourage advancement

Aggressive positioning scoring for winning positions

Key Innovation: The endgame detection significantly improved late-game play by shifting focus from material advantage to piece coordination and king mobility. After these changes it was 100% successful against random and when I tested against d5, it always either won or ended with a draw.

Final Heuristic Explanation

Basic Piece Values:

King value is higher to prioritize using kings to move.

PIECE_VALUE (100)

KING_VALUE (200)

Position-Based Bonuses:

BACK_ROW_BONUS (30): Rewards pieces for reaching the back row.

MIDDLE_BOX_BONUS (35): Favors controlling the center.

PROTECTED_PIECE_BONUS (20): Rewards pieces that are protected by others.

PROMOTION_PATH_BONUS (20): Encourages pieces moving towards the back to promote themselves.

Kings get extra points (15) for controlling central squares.

Strategic Bonuses:

MOBILITY_BONUS (15): Rewards positions with more available legal moves.

AGGRESSIVE_BONUS (25): Rewards pieces that are able to capture opponents.

ENDGAME_KING_BONUS (50): Extra value for kings during endgame.

Endgame Specific Strategy:

Piece count difference becomes more important

Distance to opponent pieces is considered later on because in the beginning it could lead to time cost being higher than its value. So saving it until later when there are less pieces makes it more efficient to wait.

Edge control is valued higher.

Kings get additional bonuses.

Aggressive positioning is weighted more heavily.

Protection Evaluation:

```
private boolean IsProtected(char[][] board, int x, int y, int player)
```

Checks if pieces are:

Protected by friendly pieces

Against the edge of the board

Supporting each other in defensive formations

The program uses these heuristics in combination with a minimax algorithm with alpha-beta pruning to evaluate positions and choose moves. The weights of these factors are carefully balanced to create a strong playing strategy that considers both tactical and strategic elements of the game.

Testing and Results

Created a ./run_test.sh file that ran 100 games between the player and a selected test player.

After each round it printed who won, and at the end it printed the win rates of each player.

When testing with: ./checkers ./computer "java MyProg" 1

```
Results after 100 games:

--------------------------------

Player 1 wins: 0 (0%)

Player 2 wins: 100 (100.00%)

Draws: 0 (0%)
```

When testing: ./checkers ./d5 "java MyProg" 1

```
Results after 100 games:

--------------------------------

Player 1 wins: 0 (0%)

Player 2 wins: 100 (100.00%)

Draws: 0 (0%)
```

Challenges

I encountered many challenges throughout this project. Most were caused due to making small adjustments leading to bigger issues and not being able to revert the changes. One example is move validation. I believed that when trying to find valid moves, I could just use the provided methods but when I did so, I would have some games that would end in saying my player lost due to illegal moves. This led me to creating an isValid function to double check everything and make sure that it is making a valid move before making one. I am still not sure if this is because of a change I made or something that just went unnoticed throughout development.

Future Improvements

Currently there is a big issue with looping. I have tried many ways to prevent it from making the same turns over and over again, but I just couldn’t find a way for it to work the way it should. That is the main issue that would need to be improved upon. There are several bots that when it goes against it, although it has the upper hand with more pieces or just more kings, it still gets stuck to prevent itself from harm. The attempts made ended up decreasing its effectiveness in other areas from lowering its defense priority, or just looping even more.