COSC 3P71: Artificial Intelligence - Assignment 4

Kindeep Singh Kargil
Email: kk17xg@brocku.ca — Student No. 6329817

I. Introduction

The game of chess has been played for thousands of years and is one of the first games Computer Scientists tried to program. It is still an ongoing field of study in computer science as people try to come up with more and more intelligent programs to play the game.

This report explores the use of minimax with alpha beta pruning to create an intelligent agent to play chess. Along with just the AI, approaches to designing a chess engine as well as the strengths and weaknesses of the design used for testing in this case will be discussed.

II. Background

A. Mini-Max

Mini max is a search strategy to play decision based games where we try to generate the entire decision tree from the current game state and try to determine our best move with the use of an evaluation function. It's a perfect application for zero-sum game theory.

In mini-max, we take our current game state and generate all possible states from this state. From all these state we're trying to maximize our profit. We will call ourselves, the agent who made the decision at this layer, max.

After generating the first layer of states, in the game of chess, the current player's turn ends and now it's the other player who has to make a move. We assume the other player will make a move that is the best case scenario for his move, let's call the entity making a decision at this layer mini. Mini will pick the move that minimizes our chances of a win/ maximizes their chances to win.

We keep repeating this to a desired depth. The more the depth we check upto the better.

After we get to the desired depth, we calculate the heuristics for that depth and recursively pick the best solution for the current player, i.e. best for mini or max at the layer.

B. Mini-Max Algorithm

[1]

```
minimax(state , ply , isMax)
    if depth = 0 or state meets termination condition
        return the heuristic value of node
    if isMax
        recursively calculate minimax for each state that
            can follow the current
        return the maximum value found
    else
        recursively calculate minimax for each state that
            can follow the current
        return the minimum value found
```

C. Alpha-Beta pruning

Alpha beta pruning extends the mini max algorithm so that it needs to go over less states, we eliminate the states that can be proven to be worse than the best we have found so far. We maintain two values alpha and beta, alpha is the minimum value found so far for max and beta is the maximum value found so far for min.

When we get to a node, in alpha beta, while calculating heuristic for all possible child states, if (alpha ¿= beta), we know that there is a minimum best value for max which is greater than the maximum score mini, and hence we don't need to further evaluate nodes at the current depth. [2]

D. Alpha-Beta pruning Algorithm

```
alpha = − Infinity
beta = Infinity
alphabeta(state , ply , isMax , alpha , beta)
    if depth = 0 or state meets termination condition
        return the heuristic value of node
    if isMax
        recursively calculate minimax for each state that
            can follow the current
        return the maximum value found
    else
        recursively calculate minimax for each state that
            can follow the current
        return the minimum value found
```

III. Experimental Setup

The chess engine is written in Kotlin.

A. Design

The design follows OOP principles.

This is how the class structure looks: -

- *Board*: Stores the current board state. The board is stored as a 2d list of Tiles. It also stores information on whose turn it is and information to determine if moves like en-passant and castling are allowed.
- *BoardPosition*: An interface that represents a position on the board.
- *Tile*: Represents a tile on the board. It maintains a reference to the current piece on the tile, which is null if the tile is empty. It inherits from BoardPosition and also includes position information.
- *Piece*: Abstract class representing general information for a single piece. It maintains information on detecting if a Move can be executed starting from this piece. Classes Pawn, Bishop, Queen, Rook, Knight, King extend this class and provide move check information as well

as override move conditions as required. Also stores a boolean indicating if the piece is black or white.

- *Move*: Represents a singular move for the board. Holds a start position, an end position and what piece to select in case there was promotion.
- *Game*: Responsible for actually running a particular game. It stores enums to indicate the current game state, and maintains a reference to two Player objects. It's responsible for interacting with the two player objects to execute moves.
- *Player*: An abstract class representing a player that the game can handle. It contains methods that are called when the game asks for the player to determine it's next move as well as methods for the game to send messages to the player. This works particularly well for a console based implementation such as this one and allows to easily specify for modes like AI vs AI.

B. Evaluation function

The initial evaluation function I experimented with assigning different weights to players, the heuristic was calculated as
$\sum$ (owner piece weights) - $\sum$ (opponent piece weights). This generated very unfavourable initial positions.

The evaluation function implemented though, assigns different weights to different pieces just like the previous function, but the new weights also take into account the position on the board. The weights I used are a modified version recommended at chessprogramming.org. [3]

- *Pawn:* Incentives to move ahead and stay out of the edges
- *Knight:* Incentives to stay out of edges and go to the center.
- *Pawn:* Incentives to move ahead and stay out of the edges
- *Bishop:* Incentives to move towards the middle
- *Rook:* Incentives to move toward the opponent king, or stay at start. Deincentivised from going in the middle.
- *Queen:* Incentives to move out of start and stay out of corners
- *King:* Incentives to stay at the start side of the board, and stay out of the horizontal center.

Additionally, the player is penalized heavily for being in checkmate as well as for check, and rewarded similarly if the opponent is in checkmate or check.

Position checks:

C. Serializing board state

The program is able to start from any given board state. To allow this, I've defined a local standard to enter in a board state in plaintext. The program does not check if the board state given is valid, but follows the rules of chess after. The serializeable state is also output on the console after every move along with the current board state as a more visual plaintext.

Here's a sample board state: -

```
 x   x   x   x   x   x   KIB x
PNB PNB x   x   x   RKW PNB PNB
x   x   x   x   x   x   x   x
x   x   x   x   BIW x   x   x
x   x   x   x   x   x   x   x
x   x   x   x   x   BIB x   x
PNW PNW PNW x   x   PNW x   PNW
x   x   KIW RKB x   BIW x   RKW
 false  false  false  false  false  −1 −1
```

The last line in this representation is to account for special moves: -

- Has the white king moved from it's original starting position?
- Has the black king moved from it's original starting position?
- Has the black king moved from it's original starting position?
- Has the black king moved from it's original starting position?
- x coordinate of pawn if pawn has skipped a position in the last move (first 2 step position)
- y coordinate of pawn if pawn has skipped a position in the last move (first 2 step position)

D. Special Moves: -

- *En-passant:* Take a pawn that has skipped your pawns current attack position.
- *Castling:* Move king to one from board end and rook to the other direction of king if they're both at starting positions.
- *Promotion:* Allow changing a pawn to a Queen, Rook, or a Knight if it gets to the vertically opposite edge of the board.

Checkmate, stalemate and Check have been implemented as check if all possible moves to the positions in question can lead to this state, essentialy like brute force.

Checks like 50 moves have not been implemented, they're not particularly relevant to the AI implementation.

IV. Results

A. Evaluation

With the positional heuristics, the ai does make good starting moves as well. Testing against level 3 ai from chess.com demonstrated the competence of this ai.

B. Design

The chess engine design proved mostly successful, several tests to check if the program correctly handles special moves like en-passant, castling as well as checks for states such as stalemate and checkmate correctly worked.

Overlooking checks for a check condition before making the move in the design planning led to having to go for suboptiomal design decisions.

V. Discussions and Conclusions

There's a lot of inefficiencies in this implementation of the move determining alpha beta, which can be significantly improved even with just caching.

The check for checkmate, getting all possible moves etc. Can be optimized a lot, right now they're more or less brute force.

References

[1] "Minimax, wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Minimax

[2] "Alpha-beta pruning, wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[3] "Simplified evaluation function." [Online]. Available: https://www.chessprogramming.org/Simplified_Evaluation_Function