Intelligent Agent Playing Chinese Checkers Using Mini-Max Algorithm

Orcun Demir

***Abstract*— In this report, an intelligent bot designed for playing Chinese Checkers by using Mini-max algorithm will be explained. The goal, methodology and results of the project will be described elaborately and enriched with visuals.**

I. INTRODUCTION

A. Mini-Max Algorithm

Mini-Max algorithm is a recursive algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.[1] When the game board has assigned with an evaluation score, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score. Mini-Max algorithm uses recursion to search through the game-tree and find this optimal states.

B. Alpha-Beta Pruning

This concept maintains two values, alpha and beta, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, alpha is negative infinity and beta is positive infinity, both players start with their worst possible score. Whenever the maximum score that the minimizing player is assured of becomes less than the minimum score that the maximizing player is assured of (beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.[2]

C. Chinese Checker Game

Chinese Checkers is a strategy board game for two players. Players starts on the 3x3 squares on opposite diagonal squares. The goal state for both players is to move their pieces to opposite 3x3 square which their opponent starts. Each piece can move horizontal and vertical in the direction to their goal corner. Pieces can jump over any type of other pieces divided by one empty cells in one move.

II. METHODOLOGY

A. Implementing Game GUI

1) Language and Technology: I used Java for main language. A 2-dimensional 8x8 sized array of Tile Object is used to represent the game board. For rendering game and manage user inputs JavaFX platform has been used.

2) Classes and Enums:

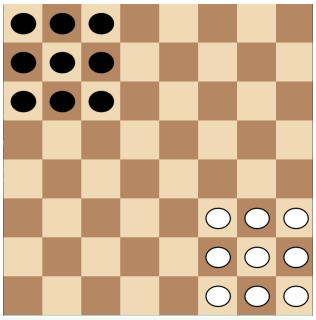

Fig. 1. Initial State

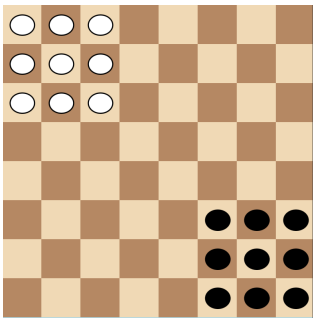

Fig. 2. Goal State for Both Players

a) Tile

Tile is a class for representing square tiles in game. Tile class has this attributes:

- TileType type: Type of this tile. Used for coloring, handling click events.
- Piece piece: If there is a piece on this tile, reference of that piece, otherwise null.
- int coo_x: X coordinate for this tile.
- int coo_y: Y coordinate for this tile.

b) Piece

Piece is a class for representing pieces in game. Piece class has this attributes:

- PieceType type: Type of this piece. Used for coloring, validating moves, evaluating score.
- int x: X coordinate for this piece.
- int y: Y coordinate for this piece.

Methods in Piece class are:

- move(int x, int y): Moves this piece to board[x][y] and renders it.

c) TileType

TileType is an enum for tiles. Possible values are:

- LIGHT
- DARK
- MOVABLE
- PREV_MOVE

LIGHT and DARK tiles are light and dark colored squares for representing board. MOVABLE tiles are purple colored and clickable. When a piece has been clicked, all the tiles that piece can move changes to this type. PREV_MOVE tiles are purple but not clickable. They stands for bots last move.

d) PieceType

PieceType is an enum for pieces. Possible values are:

- LIGHT
- DARK

PieceType has a final moveDir attribute which represents moving direction. -1 for LIGHT and 1 for DARK pieces. It has been used for moving pieces in the right direction on board.

e) Coordinate

Coordinate is a class for transferring data between board states. It has been used in Mini-Max algorithm. Coordinate class has this attributes:

- int x: X coordinate for this data transfer object.
- int y: Y coordinate for this data transfer object.

B. Mini-Max Algorithm

1) Evaluation Function: Every call of Mini-Max algorithm, score starts with 0 and checks this conditions:

$$Score+ = \begin{cases} 5000 & if\ bot(DARK)\ wins \\ -5000 & if\ player(LIGHT)\ wins \end{cases}$$

*Winning is the best case for both maximizing and minimizing players.
For piece in all DARK pieces:

$$Score+ = piece.x + piece.y$$

$$Score+ = 2 \quad if\ piece\ is\ on\ bottom\ right\ 3x3\ corner$$

*Maximizing pieces x and y (ie. going towards bottom right corner) is good for maximizing player. Also putting a piece on bottom right corner is slightly better.
For piece in all LIGHT pieces:

$$Score- = ((7 - piece.x) + (7 - piece.y))$$

$$Score- = 1 \quad if\ piece\ is\ on\ top\ left\ 3x3\ corner$$

*Minimizing pieces x and y (ie. going towards top left corner) is good for minimizing player. Also putting a piece on top left corner is slightly better.

DC_i= Number of DARK pieces on column i
DR_i= Number of DARK pieces on row i
LC_i= Number of LIGHT pieces on column i
LR_i= Number of LIGHT pieces on row i

if DR_7 > 3 or DC_7 > 3 or (DR_7 == 3 and DR_6 > 3) or (DC_7 == 3 and DC_6 > 3)

$$Score- = 5000$$

if LR_0 > 3 or LC_0 > 3 or (LR_0 == 3 and LR_1 > 3) or (LC_0 == 3 and LC_1 > 3)

$$Score+ = 5000$$

*This conditions means losing game because pieces cannot move backwards.

```
1: if depth == 0 or gameHasWinner then
2:     score ← evaluateBoard()
3:     returnscore, lastMovedPiece
4: end if
5: if isMax then
6:     maxScore ← −∞
7:     bestPiece ← null
8:     bestMove ← null
9:     for piece in all DARK pieces do
10:        for move in allValidMoves(piece) do
11:            piece.move(move)
12:            score ← Mini−Max(max_depth, depth−
    1, alpha, beta, false, piece)
13:            maxScore ← max(maxScore, score)
14:            alpha ← max(alpha, score)
15:            if alpha ≤ beta then
16:                break
17:            end if
18:            if maxScore == score then
19:                bestPice ← piece
20:                bestMove ← move
21:            end if
22:        end for
23:    end for
24:    return bestPiece, bestMove, maxScore
25: end if
26: if not isMax then
27:    minScore ←∞
28:    bestPiece ← null
29:    bestMove ← null
30:    for piece in all LIGHT pieces do
31:        for move in allValidMoves(piece) do
32:            piece.move(move)
33:            score ← Mini−Max(max_depth, depth−
    1, alpha, beta, true, piece)
34:            minScore ← min(minScore, score)
35:            beta ← min(alpha, score)
36:            if alpha ≤ beta then
37:                break
38:            end if
39:            if minScore == score then
40:                bestPice ← piece
41:                bestMove ← move
42:            end if
43:        end for
44:    end for
45:    return bestPiece, bestMove, minScore
46: end if
```

2) Pseudo-Code for Mini-Max:

III. RESULTS

After implementing alpha-bet pruning, running times reduced quitely. I tested bot myself at the end and It won 7 of 10 games with 5 depth. Performance increases with more depth. For now 5 or 6 depth is highly playable, but more depth causes slow running times. It's performance can vary depending on local machine.

TABLE I

DIFFERENCE WITH ALPHA-BETA PRUNING

alpha-beta pruning	5 depth	6 depth
No	6.10	1:04.00
Yes	1.64	15.81

REFERENCES

[1] javatpoint.com, 'Mini-Max Algorithm in Artificial Intelligence', [Online]. Available: https://www.javatpoint.com/mini-max-algorithm-in-ai.

[2] wikipedia.org, 'Alpha–beta pruning',[Online]. Available: https://en.wikipedia.org/wiki/Alpha

Source: https://github.com/OrchiDorchi/ChineseChackerAI