Universidade da Beira Interior
Departamento de Informática

Practical Project 1 - Chess Intelligent Agent

Made by:

Guilherme Teixeira

14 november 2022

Contents

Acronyms

UBI Universidade da Beira Interior

AI Artificial Intelligence

Chapter

1

Introduction

1.1 Background

This is a document written in LaTeX and is useful for the better understanding of the first pratical project "Chess Intelligent Agent" of the curricular unit Artificial Intelligence (AI) in Universidade da Beira Interior (UBI). This project has the main goal of familiarize the student with the topic "Search and Optimization".

1.2 Problem and Goals

The first practical project regards the implementation of an “Intelligent Agent”, in Python language, able to play Chess against an opponent, in a server-client framework.

The main goal is to implement in the agent the most complete/smart heuristics as possible, so he can decide its own move, to maximize the chances of winning the game.

1.3 Document organization

In order to reflect the work that has been done, this document is structured as follows:

1. The first chapter – **Introduction** – presents the project, describes the problem, the main goals and the document organization.

2. The second chapter – **Implementation and Testing** – describes the different steps for the successful project implementation and the description of the functions that were developed.

3. The third chapter – **Conclusions** – reports the results obtained with the realization of this project.

Chapter

2

Implementation and Testing

2.1 Introduction

This chapther describes the different steps for the successful project implementation, the tests that were made and the description/explanation of the functions that were developed.

2.2 Reused functions

I saw that I could reuse some functions present the file that has a client that make random plays (RandomPlays.py). The functions that have been reused were:

- pos1_to_pos2(x)

  Transform "1D" position in "2D" coordinates.

- `get_positions_directions(state, piece, p2, directions)`

  get_available_positions(state, p2, piece)

  Both combined to discover the directions that a piece is able to go.

- `sucessor_states(state, player)`

  To know who are the sucessors (board states) of the current state of the board.

- `check_win(cur_state)`

  To check if the current state of the board is a win.

2.3 The functions that I build

2.3.1 Useful functions for the next stages

I created the `Reverse()` function so I can reverse a list further on.

```
def reverse(lst):
    new_lst = lst[::-1]
    return new_lst
}
```

Code snippet 2.1: Function to reverse a list.

I created the `positions_of_pieces(pieces, board)` function so given some `pieces` and a `board`, it returns a list with the 1D positions of the given `pieces` in the `board`.

Example: In the inicial state of the board, if `pieces` are "ai" (the black towers), function will return [0, 7].

```
def positions_of_pieces(pieces, board):
    result = []
    lst = list(pieces)
    for piece in lst:
        for pos in re.finditer(piece, board):
            result.append(pos.start())
    result.sort()
    return result
```

Code snippet 2.2: Function to discover the positions of pieces in the board.

I created the `is_capture(board, move, player)` function so given a `board` and a `player` `move` on that `board`, we check if the `player` captured any opponent pieces with that `move`.

```
def is_capture(board, move, player):
    if player == 0:
        oppontent_pieces_before = positions_of_pieces("abcdefghijklmnop"
            , board)
        oppontent_pieces_after = positions_of_pieces("abcdefghijklmnop",
             move)
    else:
        oppontent_pieces_before = positions_of_pieces("IJKLMNOPABCDEFGH"
            , board)
        oppontent_pieces_after = positions_of_pieces("IJKLMNOPABCDEFGH",
             move)

    if oppontent_pieces_after < oppontent_pieces_before:
        return True
```

```
    else:
        return False
```

Code snippet 2.3: Function to check if a capture happen.

2.3.2 Board Evaluation

2.3.2.1 How to evaluate

The purpose of this function is to evaluate the current state of the board.

For evaluating the board initially, we must think about how a Grandmaster would think in his respective match.

Some of the points which should come in our mind are:

- Avoid exchanging one minor piece for three pawns.
- Always have the bishop in pairs.
- Avoid exchanging two minor pieces for a rook and a pawn.

So, the equations from the above insights will be:

- Bishop > 3 Pawns AND Knight > 3 Pawns
- Bishop > Knight
- Bishop + Knight > Rook + Pawn

By simplifying the above equations, we get Bishop > Knight > 3 Pawns. Also, we must convert the third equation as Bishop + Knight = Rook + 1.5 Pawn as two minor pieces are worth a rook and two pawns.

We will use piece square tables to evaluate our board pieces and the values will be set in an 8x8 matrix such as in chess such that it must have a higher value at favorable positions and a lower value at a non-favorable place.

For example, the probability of the white's king crossing the centerline will be less than 20% and therefore we will place negative values in that matrix.

In the end game (where each player as less than 13 points in material), we will use a different piece square table for the king.

```
kingtablewhite = [
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -20, -30, -30, -40, -40, -30, -30, -20,
    -10, -20, -20, -20, -20, -20, -20, -10,
    20, 20, 0, 0, 0, 0, 20, 20,
    20, 30, 10, 0, 0, 10, 30, 20]

kingtableblack = reverse(kingtablewhite)
```

Code snippet 2.4: Example of king's piece square table.

2.3.2.2 How the function works

Firstly, we must calculate the total number of pieces so that we can pass it into our material function.

```
    counter = Counter(board)
    wp = counter["I"] + counter["J"] + counter["K"] + counter["L"] +
        counter["M"] + counter["N"] + counter["O"] + \
          counter["P"]
    bp = counter["i"] + counter["j"] + counter["k"] + counter["l"] +
        counter["m"] + counter["n"] + counter["o"] + \
          counter["p"]
    wk = counter["B"] + counter["G"]
    bk = counter["b"] + counter["g"]
    wb = counter["C"] + counter["F"]
    bb = counter["c"] + counter["f"]
    wr = counter["A"] + counter["H"]
    br = counter["a"] + counter["h"]
    wq = counter["D"]
    bq = counter["d"]
```

Code snippet 2.5: Calculation of the number of pieces.

Secondly, let's calculate the scores.

The material score is calculated by the summation of all respective piece's weights multiplied by the difference between the number of that respective piece between white and black.

The individual pieces score is the sum of piece-square values of positions where the respective piece is present at that instance of the game.

```
    material = 100 * (wp - bp) + 320 * (wk - bk) + 330 * (wb - bb) + 500
        * (wr - br) + 900 * (wq - bq)

    pawnsq = sum([pawntablewhite[pos] for pos in positions_of_pieces("
        IJKLMNOP", board)])
    pawnsq = pawnsq + sum([-pawntableblack[pos] for pos in
        positions_of_pieces("ijklmnop", board)])

    knightsq = sum([knighttablewhite[pos] for pos in positions_of_pieces
        ("BG", board)])
    knightsq = knightsq + sum([-knighttableblack[pos] for pos in
        positions_of_pieces("bg", board)])

    bishopsq = sum([bishopstablewhite[pos] for pos in
        positions_of_pieces("CF", board)])
    bishopsq = bishopsq + sum([-bishopstableblack[pos] for pos in
        positions_of_pieces("cf", board)])

    rooksq = sum([rookstablewhite[pos] for pos in positions_of_pieces("
        AH", board)])
    rooksq = rooksq + sum([-rookstableblack[pos] for pos in
        positions_of_pieces("aH", board)])

    queensq = sum([queentablewhite[pos] for pos in positions_of_pieces("
        D", board)])
    queensq = queensq + sum([-queentableblack[pos] for pos in
        positions_of_pieces("d", board)])

    if wp + (wk * 3) + (wb * 3) + (wr * 5) + (wq * 9) <= 13 and bp + (bk
        * 3) + (bb * 3) + (br * 5) + (bq * 9) <= 13:
        kingsq = sum([kingtablewhiteEND[pos] for pos in
            positions_of_pieces("E", board)])
        kingsq = kingsq + sum([-kingtableblackEND[pos] for pos in
            positions_of_pieces("e", board)])
    else:
        kingsq = sum([kingtablewhite[pos] for pos in positions_of_pieces
            ("E", board)])
        kingsq = kingsq + sum([-kingtableblack[pos] for pos in
            positions_of_pieces("e", board)])
```

Code snippet 2.6: Material scroe and individual pieces score .

At last, let's calculate the evaluation function which will return the summation of the material scores and the individual scores for white and when it comes for black, let's negate it.

```
    eval = material + pawnsq + knightsq + bishopsq + rooksq + queensq +
        kingsq

    if player == 0:
        return eval
    else:
        return -eval
```

Code snippet 2.7: Material evaluation.

2.3.3 Move Selection

2.3.3.1 What is the Minimax algorithm?

It's a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst-case scenario. In simple words, at each step, it assumes that player A is trying to maximize its chances of winning, and in the next turn player B is trying to minimize the chances of winning.

2.3.3.2 What is the Alpha–beta pruning?

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Connect 4, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

```
def alphabeta(alpha, beta, depthleft):
    global board, player

    bestscore = -9999
    if depthleft == 0:
        return quiesce(alpha, beta)
    for move in sucessor_states(board, player):
        board = move
        score = -alphabeta(-beta, -alpha, depthleft - 1)
        board = ""
        if score >= beta:
            return score
```

```
        if score > bestscore:
            bestscore = score
        if score > alpha:
            alpha = score

    return bestscore
```

Code snippet 2.8: Alpha-beta pruning for the optimization of our execution speed.

```
def quiesce(alpha, beta):
    global board, player

    stand_pat = evaluate_board()
    if stand_pat >= beta:
        return beta
    if alpha < stand_pat:
        alpha = stand_pat

    for move in sucessor_states(board, player):
        if is_capture(board, move, player):
            board = move
            score = -quiesce(-beta, -alpha)
            board = ""

            if score >= beta:
                return beta
            if score > alpha:
                alpha = score
    return alpha
```

Code snippet 2.9: Quiescence search to avoid the horizon effect.

2.3.3.3 Selecting the best move

By applying the previous algorithms, we can now do the selection of the best move of all the possible ones:

```
def selectmove(depth):
    global board, player

    bestMove = ""
    bestValue = -99999
    alpha = -100000
    beta = 100000
    for move in sucessor_states(board, player):
```

```
        board = move
        boardValue = -alphabeta(-beta, -alpha, depth - 1)
        if boardValue > bestValue:
            bestValue = boardValue
            bestMove = move
        if boardValue > alpha:
            alpha = boardValue
        board = ""
    return bestMove
```

Code snippet 2.10: Searches for the best move in a certain depth.

The last function (`decide_move(state, player)`) will decide one of two things:

- If one of the possible next moves end up in a win for the client, then it should do that move.
- Else, it will search (in a chosen depth) for the best move of all the possible ones.

```
def decide_move(state, player):
    global board
    board = state
    win_in_next_play = False
    depth = 1

    moves = sucessor_states(board, player)
    for m in moves:
        if check_win(m) == player:
            win_in_next_play = True
            decided_move = m
            break

    if not win_in_next_play:
        decided_move = selectmove(depth)

    return decided_move
```

Code snippet 2.11: Final function that decides the move.

Chapter

3

Conclusions

3.1 Main conclusions

Having finished this work, we can conclude that it essentially enabled a better consolidation of the contents taught in the classes and a clear understanding of the topic "Search and optimization" in AI.

Bibliography

[1] Berent, A. (2019). Piece Square Table. [Online] https://medium.com/dscvitpune/lets-create-a-chess-ai-8542a12afef.

[2] Gaikwad, A. (2020). Let's create a Chess AI. [Online] https://medium.com/dscvitpune/lets-create-a-chess-ai-8542a12afef.

[3] Proença, T. H. (2022). Problem Statement. [Online] https://adamberent.com/2019/03/02/piece-square-table/.

[Wikipedia] Wikipedia. Alpha–beta pruning. [Online] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.

(3) (2) (1) (Wikipedia)