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.

titlesec

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.

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.

2.3.2 Board Evaluation - Objective Function

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 = [
    20, 30, 10, 0, 0, 10, 30, 20,
    20, 20, 0, 0, 0, 0, 20, 20,
    -10, -20, -20, -20, -20, -20, -20, -10,
    -20, -30, -30, -40, -40, -30, -30, -20,
    -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]

kingtableblack = reverse(kingtablewhite)
```

Code snippet 2.3: Example of king's piece square table.

We will also evaluate mobility, that is a measure of the number of choices (legal moves) a player has in a given position. It is based on the idea that the more choices you have at your disposal, the stronger your position.

2.3.2.2 How the function works

Firstly, we need to check if the current state of the board if a defeat or a win for the player that is currently making a move.

```
    if player == 0:
        if board.find("e") < 0:
            return -math.inf
        if board.find("E") < 0:
            return math.inf
    if player == 1:
        if board.find("E") < 0:
            return -math.inf
        if board.find("e") < 0:
            return math.inf
```

Code snippet 2.4: Calculation of the number of pieces.

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

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

Code snippet 2.5: Calculation of the number of pieces.

After that, 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 = 10 * (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 .

The next step is to calculate mobity, by counting the legal moves of each player and give a little bonus to bishops, knights and rocks.

```
count_white_mob = 0
count_black_mob = 0
for piece in ["a", "b", "d", "e", "f", "g", "h", "i", "j", "k", "l",
     "m", "n", "o", "p"]:
    positions_white = positions_of_pieces(piece, board)
    positions_black = positions_of_pieces(piece.upper(), board)
    for pos in positions_white:
        available_pos = get_available_positions(board, pos1_to_pos2(
            pos), piece)
```

```
            if piece == "b" or piece == "d" or piece == "f" or piece ==
                "g":
                count_white_mob += len(available_pos) + 3
            elif piece == "a" or piece == "h":
                count_white_mob += len(available_pos) + 1
            else:
                count_white_mob += len(available_pos)
        for pos in positions_black:
            available_pos = get_available_positions(board, pos1_to_pos2(
                pos), piece.upper())
            if piece == "B" or piece == "D" or piece == "F" or piece ==
                "G":
                count_black_mob += len(available_pos) + 3
            elif piece == "A" or piece == "H":
                count_black_mob += len(available_pos) + 1
            else:
                count_black_mob += len(available_pos)

    mob = count_white_mob - count_black_mob
```

Code snippet 2.7: Calculating the mobility of each player. .

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 + mob

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

Code snippet 2.8: 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 minimax_alpha_beta(tr, d, play, max_player, alpha, beta):
    if d == 0 or len(tr[-1]) == 0:
        return tr, f_obj(tr[0], play)

    ret = math.inf * pow(-1, max_player)
    ret_nd = tr
    for s in tr[-1]:
        aux, val = minimax_alpha_beta(s, d - 1, play, not max_player,
            alpha, beta)
        if max_player:
            if val > ret:
                ret = val
                ret_nd = aux
            alpha = max(alpha, ret)
        else:
            if val < ret:
                ret = val
                ret_nd = aux
            beta = min(beta, ret)
        if beta <= alpha:
            break

    return ret_nd, ret
```

Code snippet 2.9: Alpha-beta pruning for the optimization of our execution speed.

2.3.3.3 Deciding what move to make

The last function (`decide_move(state, player)`) will search (in a chosen depth) for the best move of all the possible ones. The minimax_alpha_beta(tr,

d, play, max_player, alpha, beta) will choose the best possible board in a certain depth and the get_next_move will recursively search for the father of that board, so he can make the decision on top of the tree.

```
def decide_move(board, play):
    states = expand_tree([board, 0, f_obj(board, play), []],
        depth_analysis, play)
    print('Total nodes in the tree: %d' % count_nodes(states))
    choice, value = minimax_alpha_beta(states, depth_analysis, play,
        True, -math.inf, math.inf)
    next_move = get_next_move(states, choice)

    return next_move[0]
```

Code snippet 2.10: 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)