Universidade da Beira Interior
Departamento de Informática

Nº 1 - 2022: *Chess Intelligent Agent*

Elaborado por:

Tiago Lopes Ribeiro a46346

Orientador:

Professor Doutor HUGO PEDRO PROENÇA

20 de novembro de 2022

Conteúdo

Lista de Excertos de Código

Capítulo

1

Introdução

1.1 Enquadramento

Este relatório foi feito no contexto da unidade curricular de Inteligencia Artificial

1.2 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, o enquadramento para o mesmo e a respetiva organização do documento.
2. O segundo capítulo – **Implementação** – descreve a implementação do agente, explicando funções utilizadas e decisões tomadas nas mesmas.
3. O terceiro capítulo – **Conclusões** – apresenta uma conclusão do projeto.

Capítulo

2

Implementação

2.1 Introdução

O trabalho tinha como objetivo criar uma inteligência artificial para jogar xadrez, utilizando diverssas heuristicas que iriam formar a função objetiva e posteriormente a mesma ser maximizada/minimizada na função de procura Minimax – Alfa-Beta Pruning.

2.2 Secções Intermédias

O trecho de código seguinte mostra os movimentos favoráveis de cada peça:

```
pawnBlack = ([
    [0,  0,  0,  0,  0,  0,  0,  0,],
    [50, 50, 50, 50, 50, 50, 50, 50,],
    [10, 10, 20, 30, 30, 20, 10, 10,],
    [5,  5, 10, 25, 25, 10,  5,  5,],
    [0,  0,  0, 20, 20,  0,  0,  0,],
    [5, -5,-10,  0,  0,-10, -5,  5,],
    [5, 10, 10,-20,-20, 10, 10,  5,],
    [0,  0,  0,  0,  0,  0,  0,  0]
])

pawnWhite = pawnBlack[::-1]

knight = ([
    [-50,-40,-30,-30,-30,-30,-40,-50,],
    [-40,-20,  0,  0,  0,  0,-20,-40,],
    [-30,  0, 10, 15, 15, 10,  0,-30,],
    [-30,  5, 15, 20, 20, 15,  5,-30,],
    [-30,  0, 15, 20, 20, 15,  0,-30,],
```

```
    [-30,  5, 10, 15, 15, 10,  5,-30,],
    [-40,-20,  0,  5,  5,  0,-20,-40,],
    [-50,-40,-30,-30,-30,-30,-40,-50,]

])

bishopBlack = ([
    [-20,-10,-10,-10,-10,-10,-10,-20,],
    [-10,  0,  0,  0,  0,  0,  0,-10,],
    [-10,  0,  5, 10, 10,  5,  0,-10,],
    [-10,  5,  5, 10, 10,  5,  5,-10,],
    [-10,  0, 10, 10, 10, 10,  0,-10,],
    [-10, 10, 10, 10, 10, 10, 10,-10,],
    [-10,  5,  0,  0,  0,  0,  5,-10,],
    [-20,-10,-10,-10,-10,-10,-10,-20,]
])

bishopWhite = bishopBlack[::-1]

rookBlack = ([
    [0,  0,  0,  0,  0,  0,  0,  0,],
    [5, 10, 10, 10, 10, 10, 10,  5,],
    [-5,  0,  0,  0,  0,  0,  0, -5,],
    [-5,  0,  0,  0,  0,  0,  0, -5,],
    [-5,  0,  0,  0,  0,  0,  0, -5,],
    [-5,  0,  0,  0,  0,  0,  0, -5,],
    [-5,  0,  0,  0,  0,  0,  0, -5,],
    [0,  0,  0,  5,  5,  0,  0,  0]
])

rookWhite = rookBlack[::-1]

queen = ([
    [-20,-10,-10, -5, -5,-10,-10,-20,],
    [-10,  0,  0,  0,  0,  0,  0,-10,],
    [-10,  0,  5,  5,  5,  5,  0,-10,],
    [-5,  0,  5,  5,  5,  5,  0, -5,],
    [0,  0,  5,  5,  5,  5,  0, -5,],
    [-10,  5,  5,  5,  5,  5,  0,-10,],
    [-10,  0,  5,  0,  0,  0,  0,-10,],
    [-20,-10,-10, -5, -5,-10,-10,-20]
])

kingBlack = ([
    [-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]
])

kingWhite = kingBlack[::-1]
```

Excerto de Código 2.1: posições favoraveis das peças.

As posições são guardadas num array e para as peças pretas o array é invertido, uma vez que as peças estão no lado oposto do tabuleiro.

Os valores de cada posição foram retirados de *chessprogramingwiki*[1].

A partir da função seguinte o programa encontra a posição da peça no tabuleiro e compara-a com o array mostrado anteriormente e atribui-lhe o valor da posição do mesmo nessa posição.

```
def get_positions_score(board, p, player, table):
    white = 0
    black = 0

    for i in p:
        pos = board.find(i)
        pos2 = pos1_to_pos2(pos)

        for x in range(8):
            for y in range(8):
                if player == 0:
                    if (table[pos2[0]][pos2[1]] == table[x][y]):
                        white += table[x][y]
                if player == 1:
                    if (table[pos2[0]][pos2[1]]== table[x][y]):
                        black += table[x][y]
    return white - black
```

Excerto de Código 2.2: função de atribuição de pontos a cada posição

A próxima função soma os valores de cada peça que foram atribuidos na função anterior através da posição, tendo em conta se as peças são brancas ou pretas. Quanto mais alto for o valor, melhor será a jogada.

```
    def eval_positions(board, player):

    if player == 0:
        wpawn1 = get_positions_score(board, 'i', player, pawnWhite)
```

```
        wpawn2 = get_positions_score(board, 'j', player, pawnWhite)
        wpawn3 = get_positions_score(board, 'k', player, pawnWhite)
        wpawn4 = get_positions_score(board, 'l', player, pawnWhite)
        wpawn5 = get_positions_score(board, 'm', player, pawnWhite)
        wpawn6 = get_positions_score(board, 'n', player, pawnWhite)
        wpawn7 = get_positions_score(board, 'o', player, pawnWhite)
        wpawn8 = get_positions_score(board, 'p', player, pawnWhite)

        wpawns = wpawn1+ wpawn2+ wpawn3 + wpawn4 + wpawn5 + wpawn6 +
            wpawn7 + wpawn8

        wrook1 = get_positions_score(board, 'a', player, rookWhite)
        wrook2 = get_positions_score(board, 'h', player, rookWhite)

        wknight1 = get_positions_score(board, 'b', player, knight)
        wknight2 = get_positions_score(board, 'g', player, knight)

        wbishop1 = get_positions_score(board, 'c', player, bishopWhite)
        wbishop2 = get_positions_score(board, 'f', player, bishopWhite)

        wqueen = get_positions_score(board, 'd', player, queen)

        wking = get_positions_score(board, 'e', player, kingWhite)

        return wrook1 + wrook2 + wknight1 + wknight2 + wbishop1 +
            wbishop2 + wqueen + wking + wpawns
    if player == 1:
        bpawn3 = get_positions_score(board, 'K', player, pawnBlack)
        bpawn4 = get_positions_score(board, 'L', player, pawnBlack)
        bpawn5 = get_positions_score(board, 'M', player, pawnBlack)
        bpawn6 = get_positions_score(board, 'N', player, pawnBlack)

        bpawns = bpawn3 + bpawn4 + bpawn5 + bpawn6

        brook1 = get_positions_score(board, 'A', player, rookBlack)
        brook2 = get_positions_score(board, 'H', player, rookBlack)

        bknight1 = get_positions_score(board, 'B', player, knight)
        bknight2 = get_positions_score(board, 'G', player, knight)

        bbishop1 = get_positions_score(board, 'C', player, bishopBlack)
        bbishop2 = get_positions_score(board, 'F', player, bishopBlack)

        bqueen = get_positions_score(board, 'D', player, queen)

        bking = get_positions_score(board, 'E', player, kingBlack)

        return brook1 + brook2 + bknight1 + bknight2 + bbishop1 +
            bbishop2 + bqueen + bking + bpawns
```

Excerto de Código 2.3: função de soma dos pontos

A próxima função analisa quantas peças inimigas estão a ameaçar as principais peças (torre, cavalo, bispo, rei e rainha) e guarda num array que posteriormente será usado o seu tamanho, ou seja, a quantidade de peças inimigas que estão a ameaçar naquele estado, na função objetivo.

```
def ameaca_ativa(board, p, player):
res = []
pos = board.find(p)
pos2 = pos1_to_pos2(pos)
if player == 0:
    #torre
    if p == 'a':
        for i in range(1, pos2[0] + 7):
            if pos2[0] + i > 7:
                break

            o = board[pos2_to_pos1([pos2[0] + i, pos2[1]])]
            o_ascii = ord(o)
            if o_ascii >= 97 and o_ascii <= 112:
                res.append(o)
                break
            if o_ascii >= 65 and o_ascii <= 80:
                break
    #rei
    if p == 'e':
        for i in range(1, pos2[0] + 7):
            if pos2[0] + i > 7:
                break

            o = board[pos2_to_pos1([pos2[0] + i, pos2[1]])]
            o_ascii = ord(o)
            if o_ascii >= 101:
                res.append(o)
                break
            if o_ascii >= 69:
                break
    #cavalo
    if p == 'b':
        for i in range(1, pos2[0] + 7):
            if pos2[0] + i > 7:
                break

            o = board[pos2_to_pos1([pos2[0] + i, pos2[1]])]
            o_ascii = ord(o)
```

```
                if o_ascii >= 98 and o_ascii <= 103:
                    res.append(o)
                    break
                if o_ascii >= 66 and o_ascii <= 71:
                    break
        #bispo
        if p == 'c':
            for i in range(1, pos2[0] + 7):
                if pos2[0] + i > 7:
                    break

                o = board[pos2_to_pos1([pos2[0] + i, pos2[1]])]
                o_ascii = ord(o)
                if o_ascii >= 99 and o_ascii <= 102:
                    res.append(o)
                    break
                if o_ascii >= 67 and o_ascii <= 70:
                    break
        #rainha
        if p == 'd':
            for i in range(1, pos2[0] + 7):
                if pos2[0] + i > 7:
                    break

                o = board[pos2_to_pos1([pos2[0] + i, pos2[1]])]
                o_ascii = ord(o)
                if o_ascii >= 100:
                    res.append(o)
                    break
                if o_ascii >= 68:
                    break
    if player == 1:
        #... igual mas com os valores das pecas pretas

    return res
```

Excerto de Código 2.4: função de ameaça

A função objetivo é onde são usadas as funções anteriores e onde é atribuido valores a cada peça, mais alto é o valor quanto mais "importante"for, por exemplo, uma vez que o rei é a peça mais importante recebe um valor muito alto, já um peão que não é tão importante recebe um valor mais baixo de 10.

Retorna o *score* das peças mais o seu peso menos o valor da ameaça, uma vez que esta quanto maior pior é para o agente e somamos o valor das posições das peças que quanto maior for melhor é para o agente e mutiplicamos no final pela potencia para destinguir entre pretas e brancas.

```
def f_obj(board, play):
weight_positions = 1e-1
w = 'abcdedghijklmnop'
b = 'ABCDEFGHIJKLMNOP'
pts = [50, 30, 30, 900, 999999999, 30, 30, 50, 10, 10, 10, 10, 10,
    10, 10, 10]
score_w = 0
score_w_positions = 0
for i, p in enumerate(w):
    ex = board.find(p)
    if ex >= 0:
        score_w += pts[i]
        p2 = pos1_to_pos2(ex)
        score_w_positions += weight_positions * p2[0]
score_b = 0
score_b_positions = 0
for i, p in enumerate(b):
    ex = board.find(p)
    if ex >= 0:
        score_b += pts[i]
        p2 = pos1_to_pos2(ex)
        score_b_positions += weight_positions * (7 - p2[0])

val_ameaca = len(ameaca_ativa(board, 'a', play)) + (len(ameaca_ativa
    (board, 'e', play)) * 999999) + len(ameaca_ativa(board, 'b',
    play)) + len(ameaca_ativa(board, 'c', play)) + (len(ameaca_ativa
    (board, 'd', play) *100))
return ((score_w + score_w_positions - score_b - score_b_positions)
    - val_ameaca + eval_positions(board, play)) * pow(-1, play)
```

Excerto de Código 2.5: função objetivo

É esta função que vai ser responsável por ser maximizada/minimizada na função Minimax – Alfa-Beta Pruning, e a partir dai escolher o melhor caminho, ou seja o caminho, ou seja o estado mais favorável ao agente.

A profundidade utilizada é 2, pois após diversos testes, na máquina utilizada para os mesmos, não dava qualquer tipo de problema, apesar de 3 de profundidade ser o ideal, as funções não estão suficientemente optimizadas para que se optenha um bom resultado no tempo limite estipulado do servidor, dando time-out em alguns testes.

Capítulo

3

Conclusões

3.1 Conclusões Principais

Concluimos assim que a partir da função objetivo e do quanto optimizado estiverem todas as funções heuristicas melhor e mais rápido vai ser o agente.

Bibliografia

[1] [Online] https://www.chessprogramming.org/Simplified_Evaluation_Function. Último acesso a 18 de Novembro de 2022.