Inteligência Artificial

Projeto 1

Wana

Bruna Brasil Marques up202007191
Maria Sofia Gonçalves up202006927
Pedro Miguel Correia up202006199
3LEIC08

Wana

- Jogo de tabuleiro para dois jogadores em que cada um tem 8 peças. Em cada jogada, o jogador deve mexer apenas uma das suas peças. Uma peça pode-se mover tanto quanto o jogador quiser ao longo de uma linha. Uma peça não pode ter qualquer interação com qualquer outra peça (não pode ocupar o seu espaço nem passar por cima dela). Uma peça que saia do tabuleiro aparecerá e continuará o seu movimento do lado oposto. Perde o jogador que, no início da sua jogada, tenha uma peça que não tenha nenhuma possibilidade de movimento.

Formulação como Search Problem

- **Representação do estado:** o tabuleiro é guardado internamente uma matriz com os números 0, 1, 2 e 3. O 0 representa Squares nos quais não se pode jogar (os cantos do tabuleiro). O 1 e o 2 representam as peças dos jogadores respetivos. O 3 representa os lugares para os quais uma determinada peça pode ser movida. A vez do jogador é representada por: turn='red' ou turn='blue'
- **Estado inicial:** Existem 3 tabuleiros de tamanhos diferentes (9x9, 12x12 e 15x15, sendo que os jogadores podem escolher a início qual querem). O jogo começa com as peças sempre na posição da figura 1 e com turn='red'.
- **Teste objetivo:** Uma peça de um jogador não ter nenhuma possibilidade de movimento. Quando tal acontece, esse jogador perde – é atingido o estado de Gameover.

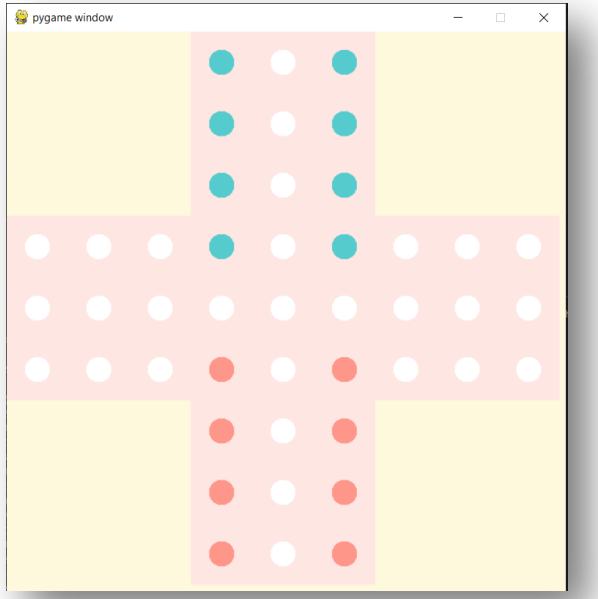


Figura 1 - Tabuleiro

Formulação como Search Problem

- **Operadores:**
  move, moves_up_straight, moves_down_straight, moves_right_straight, moves_left_straight, moves_clockwise
- **Pré-condições:**

move – a peça tem de ter pelo menos uma possibilidade de movimentação

Todas as outras funções devolvem a lista de movimentos possíveis na direção indicada pelo nome da mesma. Cada uma destas funções há uma pré-condição relativamente à linha e coluna em que a peça se encontra para poder realizar um movimento numa determinada direção.

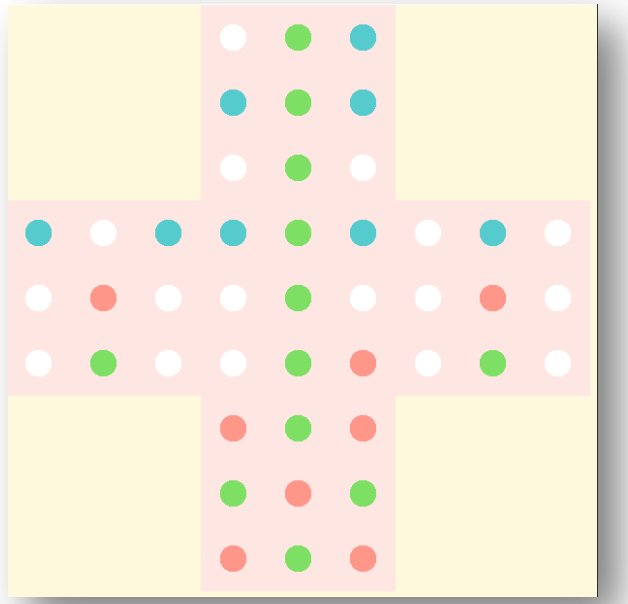
- **Avaliação heurística:**

É escolhida a jogada que fará com que haja a maior diminuição possível do número de movimentações possíveis do oponente.

- **Custo de cada jogada:** 1

Implementação final

- A linguagem escolhida para o desenvolvimento do jogo foi Python, sendo utilizadas as bibliotecas NumPy, PyGame e PyGame_menu. Todo o projeto foi desenvolvido no VS Code.
- O jogo tem 3 modos: Player vs Player, Player vs Computer e Computer vs Computer. Nos modos Player vs Computer e Computer vs Computer, há 3 dificuldades: Easy (são feitos random moves), Medium (é utilizado o algoritmo de Monte Carlo) e Difficult (Minimax). Há dois tipos de Minimax implementados: com e sem Alpha-Beta Cuts.
- É possível ver as possibilidades de jogada (peças temporárias verdes) de um jogador ao clicar numa peça do seu turno.



Algoritmos Implementados

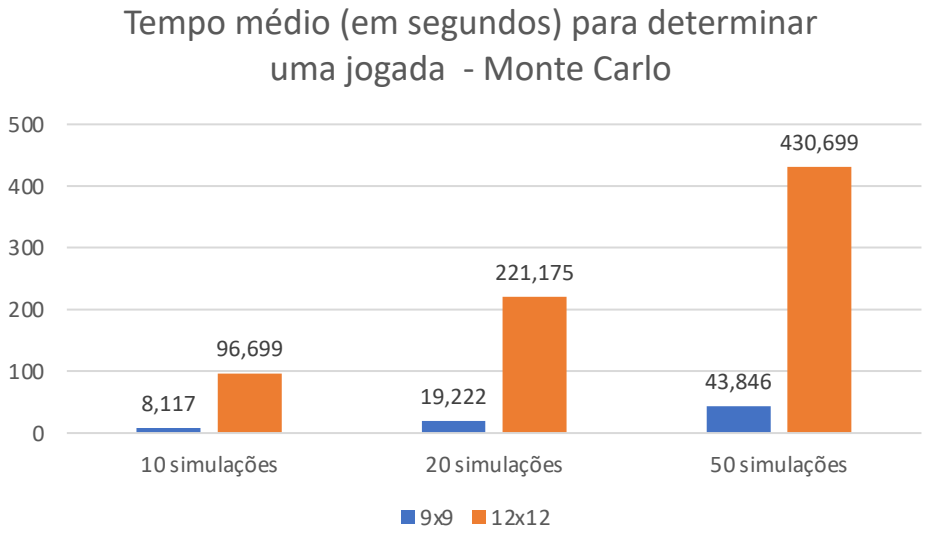
- Minimax – algoritmo que seleciona moves ótimos através de uma enumeração depth-first da game tree.
- Minimax com Alpha-Beta Cuts - calcula o mesmo move ótimo como o Minimax, mas atinge uma eficiência muito maior eliminando as subtrees que são consideradas irrelevantes. Para testar esta diferente implementação do algoritmo de Minimax, é necessário ir a função *computer_move* no ficheiro **board.py**, e para a condição "computerDifficulty=='hard'" descomentar a linha que invoca a função com o algoritmo pretendido e comentar a sem interesse.
- Monte Carlo Tree Search – consiste na construção de uma tree de moves possíveis e respetivos resultados, sendo que depois simula repetidamente a continuações aleatórias dos jogos para cada nó da tree explorado (estimando a probabilidade de esse nó levar a uma vitória do jogador em causa). Dando prioridade aos moves que levam a situações com maior probabilidade de sucesso, há uma diminuição de “caminhos” possíveis a escolher.

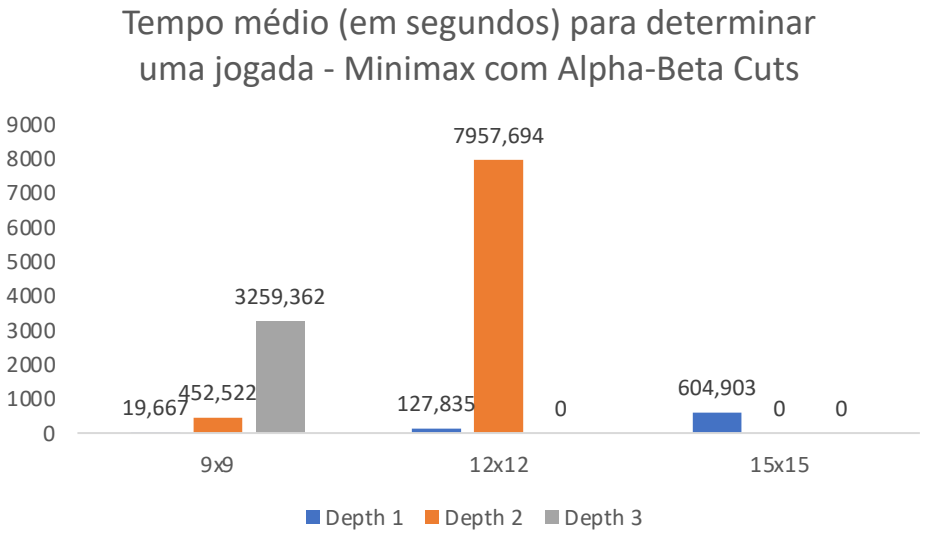
Abordagem

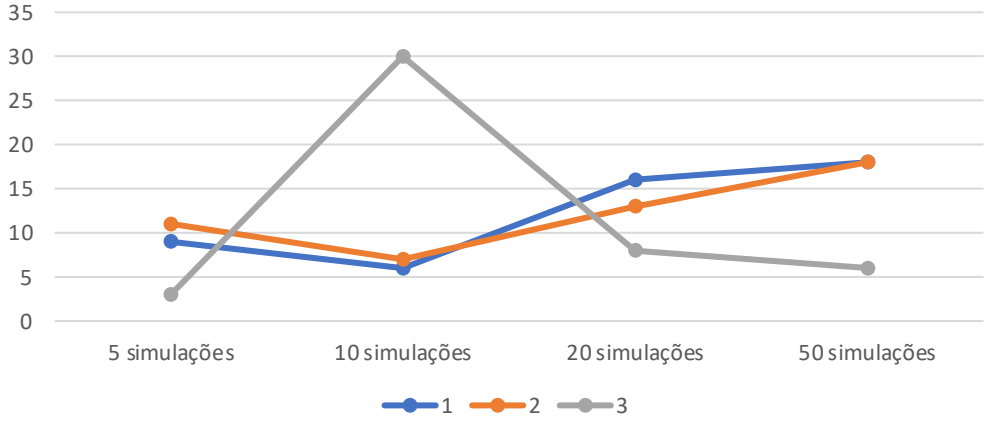
- **Heurística:** É escolhida a jogada que fará com que haja o menor número possível de movimentações do oponente.
- Esta heurística é implementada pela função *evaluate* que retorna o número total de jogadas do oponente.
- Esta função é chamada após ser feito um movimento experimental do jogador atual.
- Nos casos das implementações do Minimax, quando há uma jogada que dá uma vitória ao jogador, esta recebe um score de 99999999, garantindo assim que o algoritmo escolhe precisamente essa.

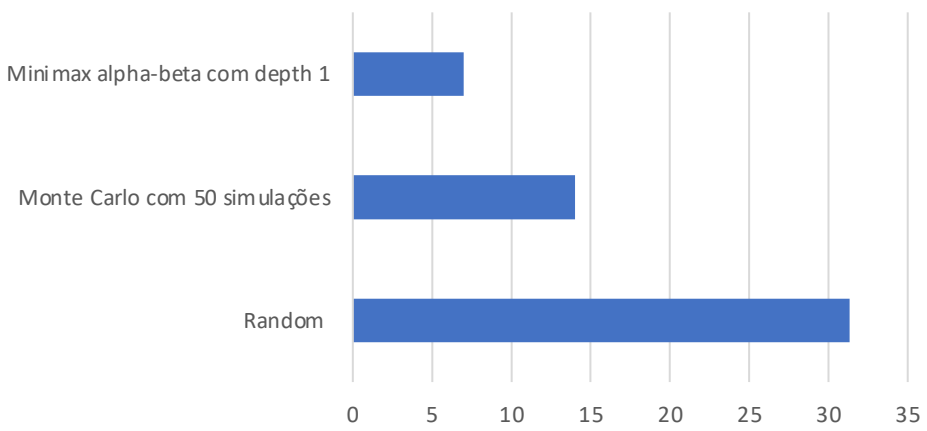
```
def evaluate(board):
    red_score=0
    blue_score=0

    for square in board.squares:
        if square.occupying_piece is not None:
            if square.occupying_piece.color=='red':
                red_score+=len(square.occupying_piece.get_moves(board))
            elif square.occupying_piece.color=='blue':
                blue_score+=len(square.occupying_piece.get_moves(board))

    if board.turn=='red':
        return -blue_score

    # Blue turn
    return -red_score
```

Resultados


Tempo médio (em segundos) para determinar uma jogada - Monte Carlo
500
400
300
200
100
0
8,117
96,699
19,222
221,175
43,846
430,699
10 simulações
20 simulações
50 simulações
9x9
12x12


Tempo médio (em segundos) para determinar uma jogada - Minimax com Alpha-Beta Cuts
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
19,667
452,522
3259,362
127,835
7957,694
0
604,903
0
0
9x9
12x12
15x15
Depth 1
Depth 2
Depth 3


Número de jogadas de um jogador para chegar a vitória - Monte Carlo
35
30
25
20
15
10
5
0
5 simulações
10 simulações
20 simulações
50 simulações
1
2
3


Número de jogadas de um jogador para chegar a vitória
Minimax alpha-beta com depth 1
Monte Carlo com 50 simulações
Random
0
5
10
15
20
25
30
35

Conclusões

- É importante salientar que os valores elevados de tempo de jogada não estão relacionados a implementação dos algoritmos em si mas porque é feito uma cópia do objeto **Board** após cada jogada, ação que temos consciência que não é a ideal para a implementação em Python. Todavia, como só muito próximo da data limite de entrega é que nos apercebemos da consequência de termos implementado assim o jogo, considerámos que seria melhor não alterar isso, uma vez que o jogo e os algoritmos estão corretamente implementados. Assim sendo, a demora deve ser atribuída a esta particularidade e não a uma falta de eficiência dos algoritmos em si.

Conclusões/Bibliografia

- Concluindo, como o Minimax com Alpha-Beta Cuts elimina as subtrees que não vão levar à solução ótima, não chegando sequer a serem analisadas, este algoritmo apresenta resultados muito mais rápidos que o Minimax. É possível observar que, para 50 simulações de Monte Carlo Tree Search, existe uma diferença de apenas 20 segundos por jogada comparando com o Minimax Alpha-Beta com depth 1, no entanto o Minimax Alpha-Beta necessita de metade das jogadas do Monte Carlo para ganhar um jogo. Assim, é também possível ter a perceção de que para o Monte Carlo conseguir ter uma eficiência semelhante à do Minimax com Alpha-Beta Cuts, seria necessário ter um número bastante elevado de simulações.

- https://www.thepythoncode.com/article/make-a-chess-game-using-pygame-in-python
- https://boardgamegeek.com/boardgame/364012/wana