U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

ABOYNE

Adversarial Search Methods for Two-Player Board Games

Inteligência Artificial 2023-2024

Assignment 1 – Grupo A1_49

Diogo Santos | Gonçalo Matos | Luís Contreiras

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

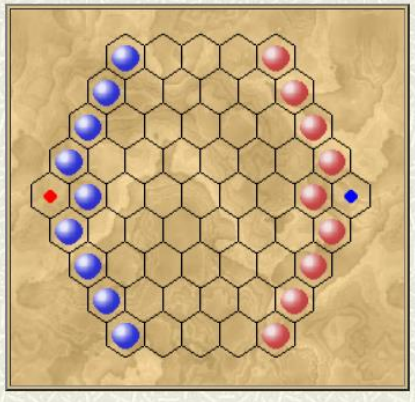
Principais Referências

Para realizar este trabalho, inspiramo-nos na página online referenciada pelos professores acerca do jogo, que nos possibilitou um bom entendimento das regras e objetivos do jogo, tal como o seu desenvolvimento visual e de código.

https://www.di.fc.ul.pt/~jpn/gv/aboyne.htm

ABOYNE

Aboyne is played on a 5x5 hexagonal board with the following setup.



- **GOAL CELL** - The marked cell on the opposite side of the board.
- **BLOCKED STONE** - A stone adjacent to an enemy stone.
- **TURN** - At each turn, each player must move one of his non-blocked stones:
  - A stone may move to an adjacent empty cell or jump over a line of friendly stones landing on the immediate next cell. It that cell is occupied by an enemy stone, that stone is captured.
  - A stone cannot move into the opponent's goal cell.
- **GOAL** - Wins the player that moves a stone into his own goal cell or stalemates the opponent.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Definição do Jogo

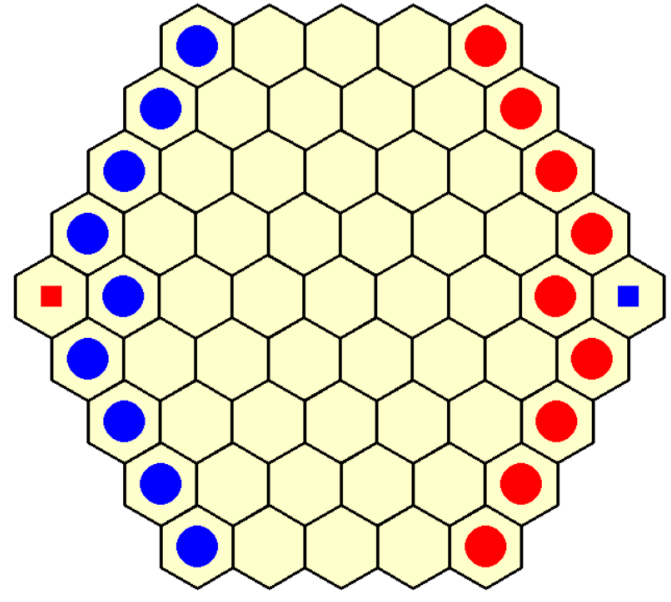
Neste projeto, o objetivo é implementar um jogo para 2 jogadores e realizar diferentes versões deste (human-human, human-computer, computer-computer).

O jogo atribuído ao nosso grupo foi o Aboyne, que é um jogo que se joga num tabuleiro hexagonal 5x5x5x5x5x5, cujo podemos ver pela imagem.

O objetivo do jogo é chegar à célula final, que é o quadrado (simbolizado com a cor da nossa equipa) que está no lado oposto do tabuleiro, atrás das peças do adversário.

Caso as peças fiquem todas bloqueadas, nós decidimos atribuir a vitória ao jogador com maior número de peças no tabuleiro, caso o número de peças seja igual para ambos, atribuímos um empate.

ABOYNE

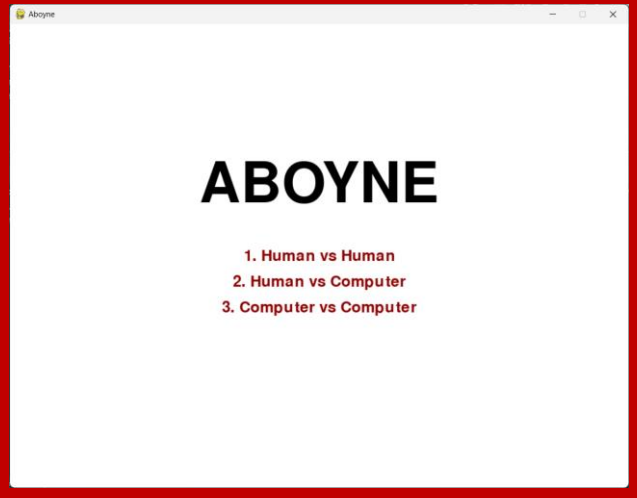


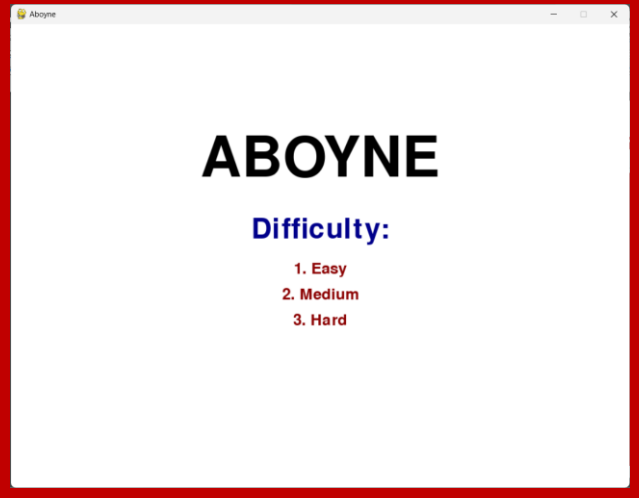
Cada jogador começa com 9 stones (peças redondas) e tem de ir avançando no tabuleiro, de forma a tentar derrotar o adversário. Cada stone consegue mover-se apenas uma célula de cada vez e esta tem de estar vazia, ou também consegue saltar sobre uma stone da mesma equipa.

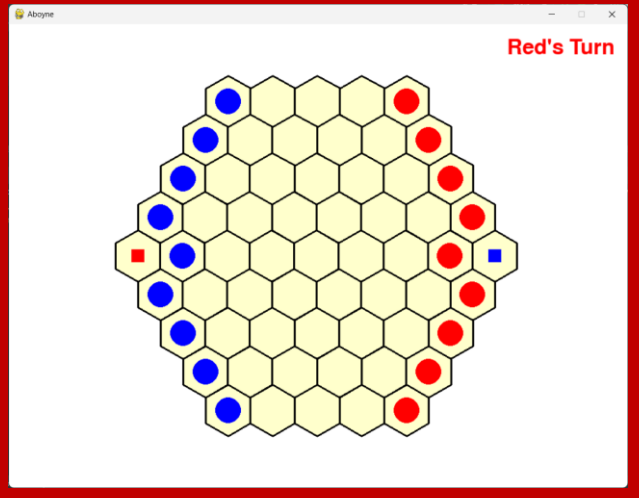
Quando duas stones adversárias colidem, estas ficam imobilizadas, tendo que se jogar com outra stone e tentar capturar a stone adversária, pois a captura de stones adversárias só acontece deste modo.

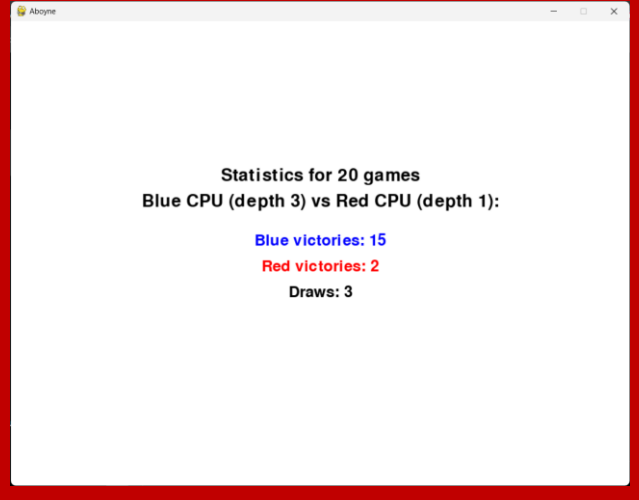
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Interface





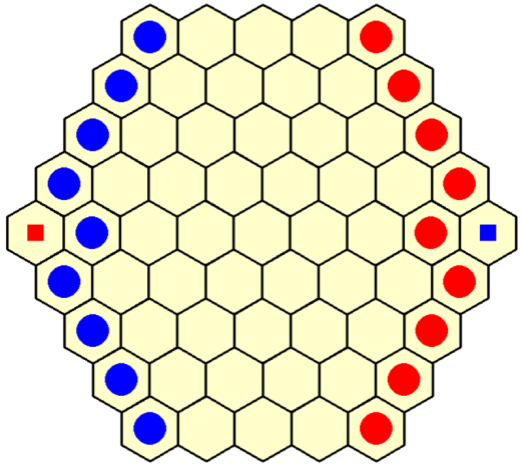




U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Representação de Estados

Estado inicial:



```
self.board = [1, 0, 0, 0, -1,
              1, 0, 0, 0, 0, -1,
              1, 0, 0, 0, 0, 0, -1,
              1, 0, 0, 0, 0, 0, 0, -1,
              0, 1, 0, 0, 0, 0, 0, -1, 0,
              1, 0, 0, 0, 0, 0, 0, -1,
              1, 0, 0, 0, 0, 0, -1,
              1, 0, 0, 0, 0, -1,
              1, 0, 0, 0, -1]
```

```
def __init__(self):
    self.current_player = random.choice([1, -1])
```

```
self.board_indexes = [
    [0, 1, 2, 3, 4],
    [5, 6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15, 16, 17],
    [18, 19, 20, 21, 22, 23, 24, 25],
    [26, 27, 28, 29, 30, 31, 32, 33, 34],
    [35, 36, 37, 38, 39, 40, 41, 42],
    [43, 44, 45, 46, 47, 48, 49],
    [50, 51, 52, 53, 54, 55],
    [56, 57, 58, 59, 60]
]
```

- Quatro módulos: **__main__.py**, **game.py**, **draw.py** e **logic.py**.
- Três classes:
  - **AboyneGame:** classe principal que usa instâncias das outras duas para correr os diferentes modos de jogo.
  - **GameDraw:** classe responsável pela interface gráfica do *pygame.*
  - **GameLogic:** classe responsável por toda a lógica/regras do jogo, algoritmo minimax e heurísticas utilizadas para avaliar movimentos.
- Tabuleiro representado por uma lista de **61** elementos: **1 (peça azul)**, **-1 (peça vermelha)** e **0 (espaço vazio)**.
- O jogador que executa o primeiro movimento é decidido **aleatoriamente**.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Algoritmo

Minimax com alpha-beta pruning

```
def minimax(self, depth, alpha, beta, player, maximizing):
    if ((depth == 0) or (len(self.get_possible_moves(player)) == 0 and maximizing)
            or (len(self.get_possible_moves(-player)) == 0 and not maximizing)):
        return self.evaluate_f8(player), None

    if maximizing:
        available_moves = self.get_possible_moves(player)
        random.shuffle(available_moves)
        max_eval = float('-inf')
        best_move = None
        for (piece_index, new_index) in available_moves:
            copy_board = tuple(self.game_draw.board)
            self.move_piece(piece_index, new_index)
            eval, _ = self.minimax(depth - 1, alpha, beta, player, False)
            self.game_draw.board = list(copy_board)
            if eval > max_eval:
                max_eval = eval
                best_move = piece_index, new_index
            alpha = max(alpha, eval)
            if alpha >= beta:
                break
        return max_eval, best_move
    else:
        available_moves = self.get_possible_moves(-player)
        random.shuffle(available_moves)
        min_eval = float('inf')
        best_move = None
        for (piece_index, new_index) in self.get_possible_moves(-player):
            copy_board = tuple(self.game_draw.board)
            self.move_piece(piece_index, new_index)
            eval, _ = self.minimax(depth - 1, alpha, beta, player, True)
            self.game_draw.board = list(copy_board)
            if eval < min_eval:
                min_eval = eval
                best_move = piece_index, new_index
            beta = min(beta, eval)
            if alpha >= beta:
                break
        return min_eval, best_move
```

Condições finais:

- Profundidade 0 atingida
- Na fase maximizante, jogador maximizante sem jogadas possíveis (todas as peças bloqueadas)
- Na fase minimizante, jogador minimizante sem jogadas possíveis (todas as peças bloqueadas)

Argumento adicional:

- **Player**: de maneira a conseguirmos analisar as jogadas de cada jogador na fase correspondente, acrescentámos o argumento **player (1 ou -1)** que funciona em conjunto com o argumento **maximizing (True ou False)** para ir buscar os movimentos disponíveis do jogador correto:
  if maximizing: available_moves = self.get_possible_moves(player)
  else: available_moves = self.get_possible_moves(-player)

Diferente ordem de geração de filhos:

- Visto a natureza determinística do minimax, interpretámos este pedido como uma maneira de variar o desenrolar dos jogos com o computador. Para obter este efeito, em cada chamada à função baralhamos a lista com os movimentos possíveis do jogador antes da sua análise: **random.shuffle(available_moves)**.

Guardar o estado:

- Devido ao alocamento dinâmico de memória do Python, guardamos o estado do tabuleiro num tuplo: **copy_board = tuple(self.game_draw.board)** antes da análise recursiva de cada jogada. Como o tuplo é imutável, permite-nos assim reestabelecer o estado anterior do tabuleiro após a análise da jogada em questão: **self.game_draw.board = list(copy_board)**.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Heurísticas

Heurística 1: Atribui pontuações diferentes à posição de cada peça de acordo com a sua proximidade à fila do meio

```
def evaluate_f1(self, player):
    counter = 0
    for i in range(61):
        if self.game_draw.board[i] == player:
            if i in [0, 1, 2, 3, 4] or i in [56, 57, 58, 59, 60]:
                counter += 1
            elif i in [5, 6, 7, 8, 9, 10] or i in [50, 51, 52, 53, 54, 55]:
                counter += 2
            elif i in [11, 12, 13, 14, 15, 16, 17] or i in [43, 44, 45, 46, 47, 48, 49]:
                counter += 3
            elif i in [18, 19, 20, 21, 22, 23, 24, 25] or i in [35, 36, 37, 38, 39, 40, 41, 42]:
                counter += 4
            else:
                counter += 5
    return counter
```

Heurística 2: Atribui pontuações diferentes à posição de cada peça de acordo com a sua proximidade à *goal cell*

```
def evaluate_f2(self, player):
    counter = 0
    for i in range(61):
        if self.game_draw.board[i] == player:
            counter += self.piece_distance_to_goal(i, player)
    return self.evaluate_f1(player) - counter*30
```

Heurística 3: Calcula a diferença entre o número de peças azuis e vermelhas, dando assim prioridade a movimentos de captura

```
def evaluate_f3(self, player):
    if player == 1:
        return self.evaluate_f2(player) + sum(self.game_draw.board)*200
    else:
        return self.evaluate_f2(player) - sum(self.game_draw.board)*200
```

Heurística 4: Calcula a diferença entre o número de movimentos de captura disponíveis, dando assim prioridade a um estado do tabuleiro em que haja mais movimentos de captura possíveis pelo jogador

```
def evaluate_f4(self, player):
    ally_possible_captures = 0
    enemy_possible_captures = 0
    for i in range(61):
        if not self.check_blocked_piece(i):
            if self.game_draw.board[i] == player:
                ally_possible_captures += len(list(
                    filter(lambda x: self.game_draw.board[x] == -player, self.highlight_possible_moves(i, player))))
            elif self.game_draw.board[i] == -player:
                enemy_possible_captures += len(list(
                    filter(lambda x: self.game_draw.board[x] == player, self.highlight_possible_moves(i, -player))))
    return self.evaluate_f3(player) + (ally_possible_captures - enemy_possible_captures) * 15
```

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Heurísticas

Heurística 5: Calcula a diferença entre o número de peças ainda disponíveis para jogar por cada lado, dando assim prioridade a movimentos que bloqueassem a propria peça em conjunto com duas ou mais peças adversárias

```
def evaluate_f5(self, player):
    blocked_blue, blocked_red = self.count_blocked_pieces()
    total_blue, total_red = self.game_draw.board.count(1),
self.game_draw.board.count(-1)
    available_blue, available_red = total_blue - blocked_blue, total_red - blocked_red
    if player == 1:
        return self.evaluate_f4(player) + (available_blue - available_red) * 100
    else:
        return self.evaluate_f4(player) + (available_red - available_blue) * 100
```

Heurística 6: Itera pelo tabuleiro averiguando se é possível um movimento final para a *goal cell*, caso seja, executa-o

```
def evaluate_f6(self, player):
    if player == 1:
        if self.game_draw.board[34] == 1:
            return 1000000 + self.evaluate_f5(1)
    else:
        if self.game_draw.board[26] == -1:
            return 1000000 + self.evaluate_f5(-1)
    return self.evaluate_f5(player)
```

Heurística 7: Caso haja uma peça adversária a um movimento de chegar à *goal cell*, se for possível, mover uma peça aliada para o lado dela de maneira a bloqueá-la

```
# Heuristic 7: Block the other player's winning move
def evaluate_f7(self, player):
    if player == 1:
        if self.game_draw.board[18] == -1 and self.check_blocked_piece(18):
            return 250 + self.evaluate_f6(1)
        if self.game_draw.board[27] == -1 and self.check_blocked_piece(27):
            return 250 + self.evaluate_f6(1)
        if self.game_draw.board[35] == -1 and self.check_blocked_piece(35):
            return 250 + self.evaluate_f6(1)
    else:
        if self.game_draw.board[25] == 1 and self.check_blocked_piece(25):
            return 250 + self.evaluate_f6(-1)
        if self.game_draw.board[33] == 1 and self.check_blocked_piece(33):
            return 250 + self.evaluate_f6(-1)
        if self.game_draw.board[42] == 1 and self.check_blocked_piece(42):
            return 250 + self.evaluate_f6(-1)
    return self.evaluate_f6(player)
```

Heurística 8 (final): Caso o jogador adversário não tiver mais movimentos disponíveis, apenas se preocupar em descobrir o caminho para a *goal cell*

```
def evaluate_f8(self, player):
    if len(self.get_possible_moves(-player)) == 0:
        return self.evaluate_f2(player)
    return self.evaluate_f7(player)
```

Resultados experimentais

```
def run_N_games_computer_vs_computer(self, n, depth_blue, depth_red):
    blue_wins = 0
    red_wins = 0
    draws = 0
    total_blue_moves, total_red_moves = 0, 0
    total_blue_time, total_red_time = 0, 0
    start_time = time.time()
    for i in range(n):
        b, r, d, b_m, r_m, b_t, r_t = self.play_computer_vs_computer(depth_blue,
depth_red)
        blue_wins += b
        red_wins += r
        draws += d
        total_blue_moves += b_m
        total_red_moves += r_m
        total_blue_time += b_t
        total_red_time += r_t
    end_time = time.time()
    execution_time = end_time - start_time

    print(f"\nStatistics for {n} games of Blue CPU (depth {depth_blue}) vs Red CPU (depth
{depth_red}):")
    print(f"Blue victories: {blue_wins} ({(blue_wins / n) * 100:.2f}%)")
    print(f"Red victories: {red_wins} ({(red_wins / n) * 100:.2f}%)")
    print(f"Draws: {draws} ({(draws / n) * 100:.2f}%)")
    print(f"Total execution time: {execution_time:.5f}s / Average per game:
{execution_time / n:.5f}s")
    print(f"Average execution time per Blue move: {total_blue_time /
total_blue_moves:.5f}s / per Red move: {total_red_time / total_red_moves:.5f}s\n")

    self.game_draw.display_results(blue_wins, red_wins, draws, n, depth_blue,
depth_red)
    # time.sleep(10)
```

Para testar a performance e eficácia do nosso minimax e das nossas heurísticas, criámos a função **run_N_games_computer_vs_computer**, que recebe como argumentos:

- n: número de partidas a realizar
- depth_blue: profundidade de pesquisa quando o minimax procura a melhor jogada para o jogador azul
- depth_red: profundidade de pesquisa quando o minimax procura a melhor jogada para o jogador vermelho

A função dá return ao número de vitórias de cada jogador, empates, tempos de execução totais e tempos de execução médios por movimento executado.

Esta função não está incluída no menu inicial do jogo, tem de ser chamada com os parâmetros pretendidos no módulo **__main__.py** após a inicialização de um objeto **AboyneGame()**.

Resultados experimentais

Games Executed	Blue Depth	Red Depth	Blue Wins	Red Wins	Draws	Total Execution Time (s)	Blue Average Time Per Move (s)	Red Average Time Per Move (s)
100	1	1	**42**	**47**	**11**	32,715	0,0036	0,0036
	2	1	**56**	**36**	**8**	184,295	0,0198	0,0036
	2	2	**38**	**41**	**21**	297,172	0,0235	0,0259
	3	1	**78**	**12**	**10**	708,038	0,1972	0,0035
	3	2	**57**	**32**	**11**	855,108	0,1466	0,0185
	3	3	**39**	**33**	**28**	2626,057	0,0650	0,0738
	4	1	**55**	**30**	**15**	3527,899	1,1276	0,0036
	4	2	**67**	**20**	**13**	5102,109	1,3071	0,0298
	4	3	**43**	**36**	**21**	6085,971	0,1552	0,0396

Quando **Blue Depth == Red Depth:**

- No geral ficámos satisfeitos com os resultados obtidos nestas circunstâncias, em comparação, o número de empates é consideravelmente maior e a variação entre as vitórias azuis e vermelhas é pequena, se o tamanho da amostra fosse maior ainda menos se notaria esta diferença.

Quando **Blue Depth > Red Depth:**

- Apesar de em todas as situações, o jogador azul ter ganho mais jogos, como era suposto, o aumento da profundidade, que deveria aumentar a probabilidade de a equipa ganhar ainda mais jogos não foi o que aconteceu de todo. Por alguma razão que não conseguimos descobrir exatamente, a margem de vitórias para o jogador azul só foi realmente notória quando a condição **Blue Depth == Red Depth + 2** se verificou.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Conclusão

No geral, estamos relativamente satisfeitos com o trabalho que desenvolvemos, achamos que a implementação do jogo foi muito bem conseguida e também ficámos contentes com as heurísticas que criámos para avaliar as jogadas do computador pois acreditamos que elas eram todas relevantes e bem coerentes. No entanto, tivemos várias dificuldades ao distribuir o peso delas na avaliação das jogadas e com isso acabámos por obter alguns resultados um pouco duvidosos que gostaríamos que fossem mais claros de explicar. Mas acabou por ser uma boa experiência trabalhar com o algoritmo *minimax*, entender como jogos de 2 jogadores com IA são programados sempre foi uma curiosidade para nós e é um conhecimento que pretendemos aprofundar no futuro.

Inteligência Artificial

Diogo Santos (up202108747@fe.up.pt)

Gonçalo Matos (up202108761@fe.up.pt)

Luís Contreiras (up202108742@fe.up.pt)