Lines of Action

Trabalho Final

David Pereira
Miguel Ribeiro
Tiago Sousa

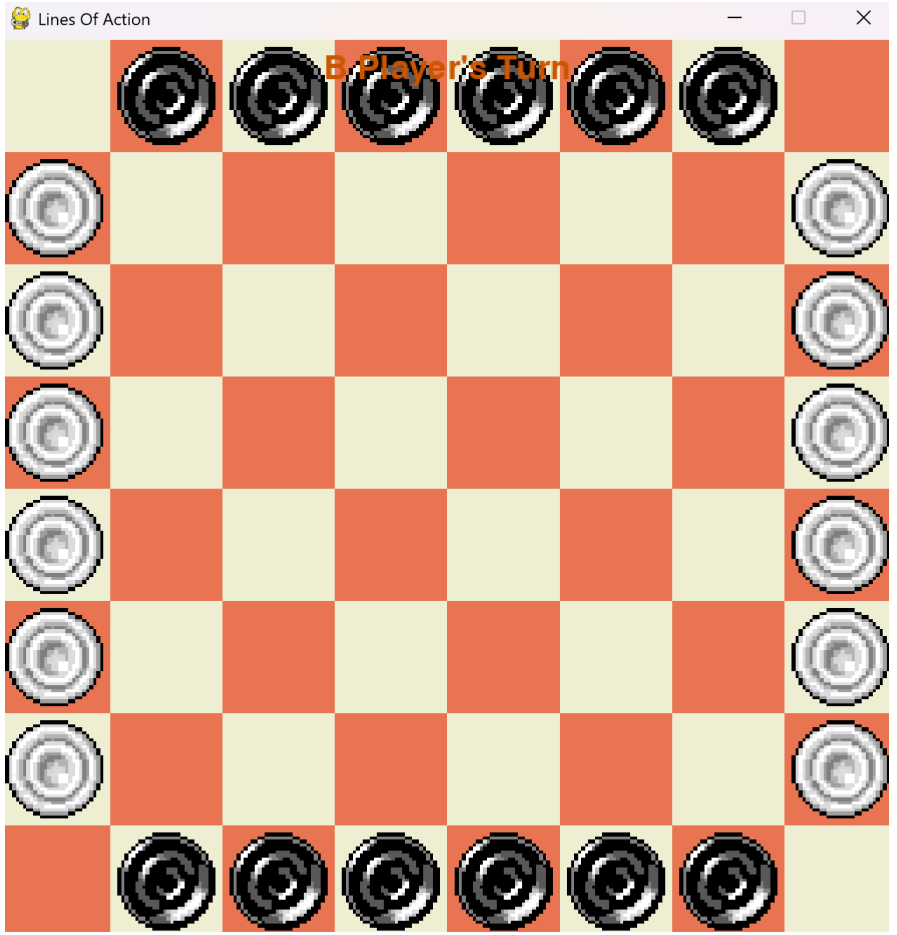
Lines of Action

Jogo de tabuleiro de dimensões 8x8 com 24 peças: 12 brancas e 12 pretas.

O objetivo é conectar todas as peças de uma mesma cor.

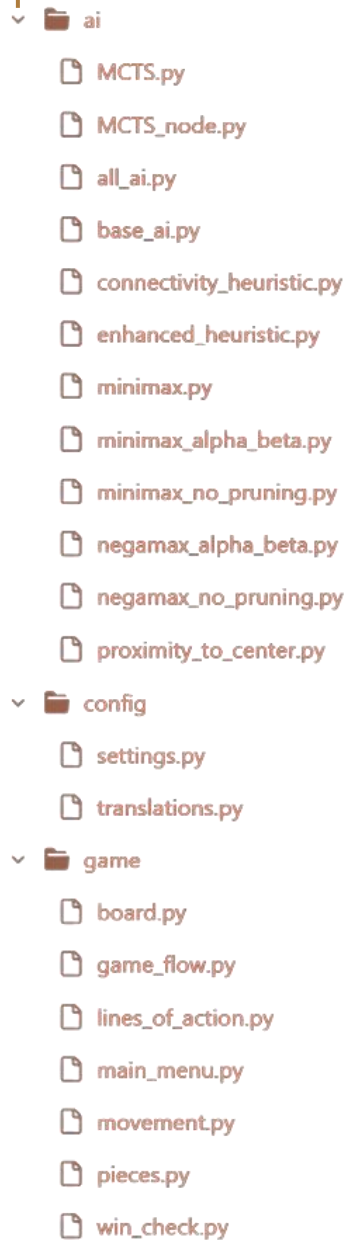
As peças pretas abrem sempre o jogo, podendo selecionar qualquer peça.

Contundo, existem algumas restrições ao movimento.



Problema de Pesquisa

– **Representação dos Estados:** dicionário,
{posição no tabuleiro (se ocupada): Cor da peça}
– **Estado inicial:** pré-definido, sendo o jogo aberto pelas peças pretas
– **Objetivo:** aglomeração das peças da mesma cor
– **Ações/Operadores:** mover na horizontal/vertical/diagonal
– **Resultados:** novo estado do dicionário, bloques e eliminação
– **Custos:** número de movimentos
– **Funções de Utilidade:** *cluster cohesion, board control*...



Trabalho de Implementação

Estrutura num modelo de diretórios e ficheiros de classes, de acordo com pertinência e utilidade específica para a parte do trabalho que contribuem.

A totalidade do jogo pode ser executada no módulo “main.py”, que executa ambos o jogo e o loop do pygame.

Algoritmos

– Minimax e Negamax, com cortes α - β, que se regem por duas funções de utilidade

– Monte Carlo Tree Search, guiado por uma função que procura ocupar um centro móvel com base na posição de todas as peças da mesma cor

```python
class MonteCarloAI(BaseAI):
    def __init__(self, game, color, rollouts=1000):
        super().__init__(game, color)
        self.rollouts = rollouts
        self.win_checker = game.win_checker
        self.heuristic = None
        self.moves = game.movement

    def get_move(self, board_state):
        if not board_state:
            return None

        root = MCTSNode(board_state.copy())
        root.untried_moves = 

        # If no valid moves, 
        if not root.untried_m
            return None

        for _ in range(self.r
            node = self._tree_
            if node is None:
                continue
```

```python
class MinimaxAI(BaseAI):
    def __init__(self, game, color):
        super().__init__(game, color)
        self.settings = game.settings
        self.board = game.board.board_dict
        self.win_checker = game.win_checker
        self.moves = game.movement

    def get_all_valid_moves(self, board, player):
        valid_moves = {}
        for pos in board:
            if board[pos] == player:
                valid_moves[pos] = self.moves.get_valid_moves(pos[0], pos[1])
        return valid_moves

    def random_evaluate(self, board, player):
        return random.randint(-100000, 100000)

    def evaluate(self, board, player):
        opponent = "W" if player == "B" else "B"

        player_positions = [pos for pos, piece in board.items() if piece == player]
        opponent_positions = [pos for pos, piece in board.items() if piece == opponent]

        if self.win_checker.check_win(player, board):
            return 100000  # Player wins
        if self.win_checker.check_win(opponent, board):
            return -100000  # Opponent wins

        def cluster_distance(positions):
            if len(positions) < 2:
                return 0  # No distance to measure
```

Funções de Utilidade

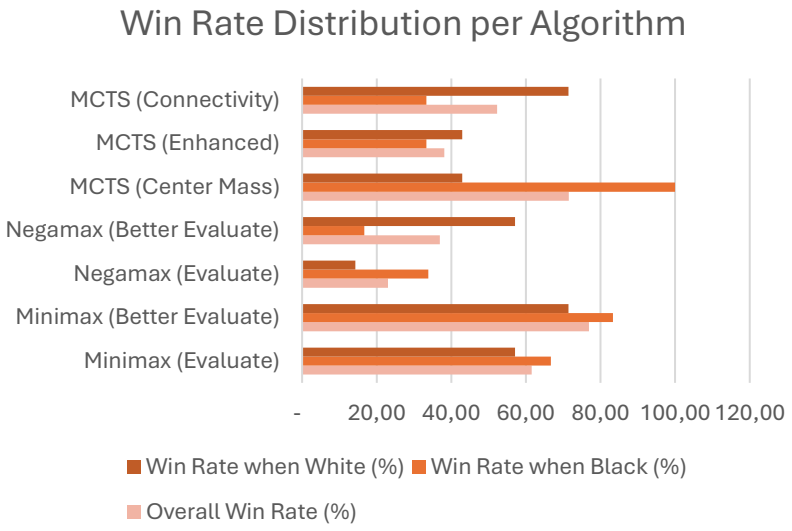
– **Evaluate:** prioriza movimentos que preservem as peças do oponete, conduzam a um maior controlo do centro e à maior proximidade entre peças, num todo

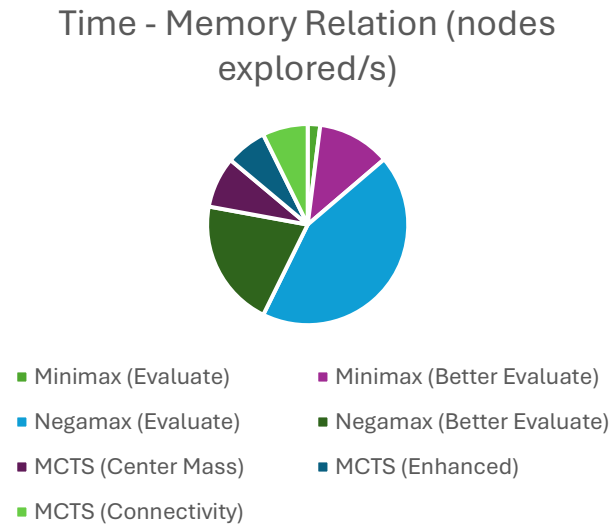
– **Better Evaluate:** implementa *evaluate*, priorizando ainda jogadas com mais movimentos subsequentes possíveis

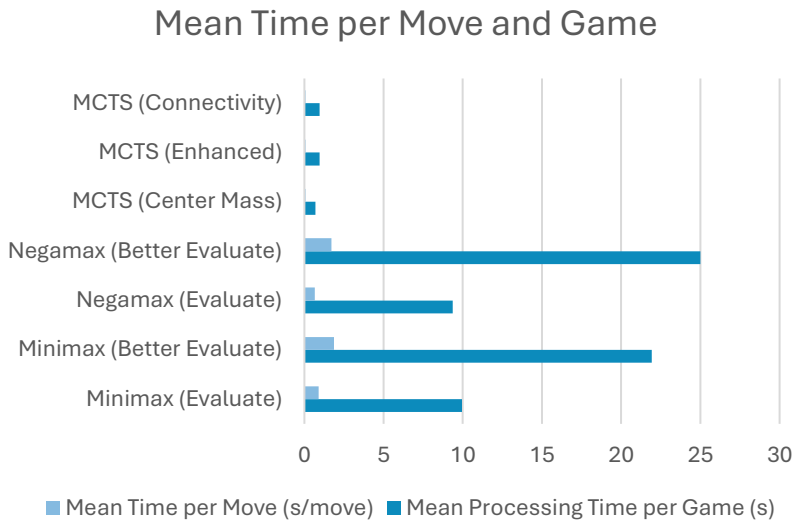
```python
def better_evaluate(self, board, player):
    opponent = "W" if player == "B" else "B"

    # Use dictionaries for fast lookups
    player_positions = {pos: True for pos, piece in board.items() if piece == player}
    opponent_positions = {pos: True for pos, piece in board.items() if piece == opponent}

    def analyze_clusters(board, player_positions):
        visited = set()
        clusters = []

        #cluster_id = 0  # Unique ID for each cluster

        for position in player_positions:
            if position not in visited:
                stack = [position]
                cluster_size = 0

                while stack:
                    row, col = stack.pop()
                    if (row, col) in visited:
                        continue

                    visited.add((row, col))
                    cluster_size += 1

                    # Check all 8 directions for connected pieces
                    for dr, dc in self.settings.directions:
                        nr, nc = row + dr, col + dc
                        if (nr, nc) in player_positions and (nr, nc) not in visited:
                            stack.append((nr, nc))

                clusters.append(cluster_size)
                #cluster_id += 1

        return len(clusters), clusters
```

```python
def evaluate(self, board, player):
    opponent = "W" if player == "B" else "B"

    player_positions = [pos for pos, piece in
    opponent_positions = [pos for pos, piece

    if self.win_checker.check_win(player, boa
        return 100000  # Player wins
    if self.win_checker.check_win(opponent, board):
        return -100000  # Opponent wins

    def cluster_distance(positions):
        if len(positions) < 2:
            return 0  # No distance to measure
        return max(abs(r1 - r2) + abs(c1 - c2) for (r1, c1) in positions for (r2, c2) in positions)

    player_cluster = cluster_distance(player_positions)  # Distance between the furthest pieces
    opponent_cluster = cluster_distance(opponent_positions)
    central_control = sum(abs(r - self.settings.rows // 2) + abs(c - self.settings.cols // 2) for (r, c) in player_positions)

    return (self.settings.rows / max(1, len(opponent_positions))) * -15 + (opponent_cluster - player_cluster) * 20 - central_control * 5
```

Resultados Experimentais







Comparação entre as diferentes IAs permite concluir que, nesta "piscina" de algoritmos, o Minimax com Better Evaluate é o mais bem sucedido

Abertura de jogo mais vantajosa para 5/7 das IAs implementadas.

Interface Gráfica

Conclusões

– Aprendizagem sobre os algoritmos de pesquisa adversarial lecionados em aula, e respetiva implementação

– Abordagem dinâmica e estratégica do problema, assim como do funcionamento, colaboração e interdependência decorrente de trabalhar numa equipa

– Introdução ao pygame, origem da maioria das dificuldades sentidas, assim como de alguns elementos da implementação

– Maior facilidade na gestão, organização e leitura de código

Biblio/Webgrafia

– Matthes, E. (2023) *Python Crash Course*. (3rd Edition). No Starch Press.

– https://docs.python.org

– https://www.pygame.org/docs/

Link do trabalho, GitHub:
Difl4/LoA_game: A video game implementing some adversarial AI models.