2A) Aboyne
Tópico 2: Jogos Adversários com Dois Jogadores

Entrega Final Projeto 1 – Inteligência Artificial

Grupo 54

André dos Santos Faria Relva - up202108695

Pedro Guilherme Pinto Magalhães - up202108756

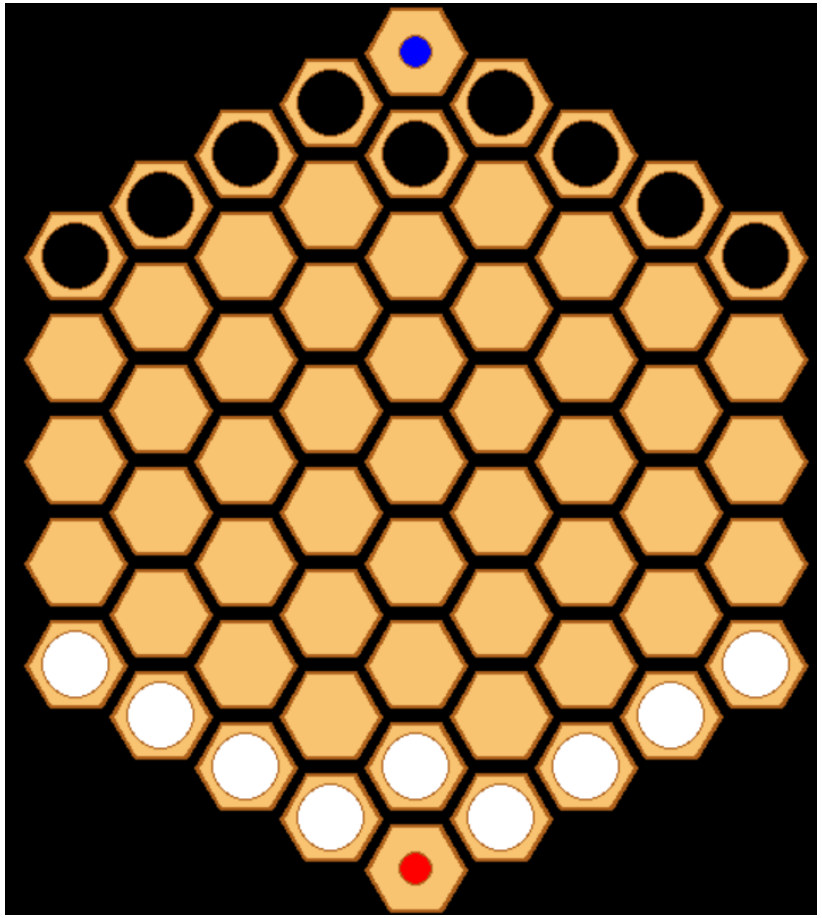
Rafael Azevedo Alves - up202004476

Faculdade de Engenharia da Universidade do Porto

01 de abril de 2024

Definição do Jogo

- **Tabuleiro:** O jogo é feito num tabuleiro hexagonal 5x5
- **Tipo de Peças:** Peças Normais (Peças Pretas e Brancas), Peças Blocked (Peças Pretas e Brancas com um "X" no meio) e Peças Objetivo (Peças Azuis e Vermelhas)
- **Movimento das Peças**: Movimentos Verticais e Diagonais para células adjacentes e vazias ou sobre uma linha de peças do próprio jogador ficando alocado na célula vazia seguinte ou célula com peça do adversário
- **Objetivo:** Um jogador ganha o jogo quando uma das suas peças alcança a sua Peça Objetivo ou quando o adversário não tem mais movimentos disponíveis (todas as suas peças estão bloqueadas ou foram capturadas por peças do adversário)



Formulação do Problema como um Problema de Pesquisa

- **Representação do Estado:** Um tabuleiro hexagonal 5x5 em que cada célula pode estar vazia, conter uma peça do Jogador 1, conter uma peça do Jogador 2, estar bloqueada (adjacente a uma peça inimiga) ou conter a peça objetivo do Jogador 1 ou do Jogador 2.
- **Estado Inicial:** Todas as células estão vazias, exceto as células objetivo de cada jogador e as 9 células de cada lado do tabuleiro (células onde estão as peças de cada jogador) e a vez de jogar pertence ao Jogador 1.
- **Teste Objetivo:** Verifica se um jogador moveu uma peça para a sua célula objetivo ou se o adversário está bloqueado (incapaz de fazer uma jogada legal, ou por ter todas as peças bloqueadas, ou por não ter mais peças).
- **Operadores:**

Nome	Condições prévias	Efeitos	Custo
Mover Peça	A peça a ser movida deve pertencer ao jogador atual, e não deve estar bloqueada.	Move a peça para uma célula vazia adjacente ou salta sobre uma linha de peças amigas, aterrando na célula imediatamente a seguir. Se essa célula estiver ocupada por uma peça inimiga, essa peça é capturada.	1

Formulação do Problema como um Problema de Pesquisa

- **Função de Avaliação/Heurística:** uma função heurística adequada para este jogo pode considerar:
  - A proximidade das peças do jogador à sua célula objetivo.
  - O número de peças que ainda tem disponível.
  - A presença de peças bloqueadas.
  - A possibilidade de criar linhas de peças amigas para facilitar os saltos.
  - O número de peças inimigas que podem ser capturadas.
  - O número de peças inimigas que podem ser bloqueadas.

Implementação

- **Linguagem de Programação:** Python, utilizando pygame para interface gráfica
- **Ambiente de Desenvolvimento:** VSCode e GitHub
- **Estruturas de Dados:**
    1. Tuplos para representar coordenadas no tabuleiro de jogo.
    2. Listas de tuplos para representar as posições das peças de cada jogador.
    3. Classes para organizar e gerir o estado e o comportamento do jogo.

Algoritmos Implementados

Os algoritmos implementados neste projeto foram, os seguintes:

Minimax Ofensivo (com α β cuts)	Minimax Defensivo (com α β cuts)	Monte Carlo Search Tree
Função de avaliação tenta diminuir a distância média das peças à peça objetivo.	Função de avaliação tenta ganhar através de o oponente ficar sem jogadas válidas. Tenta capturar e bloquear peças inimigas.	Simulação de vários jogos completos a partir de jogadas aleatórias de maneira a encontrar a melhor jogada.

```python
# Evaluates the value of a play by the current player, tries to block the opponent's moves
def evaluate_offensive(self):
    player1_score = self.calculate_score_offensive(self.player1, self.p1goal, self.p2blocked)
    player2_score = self.calculate_score_offensive(self.player2, self.p2goal, self.p1blocked)
    return player1_score - player2_score

def calculate_score_offensive(self, pieces, goal, blocked):
    score = 0
    for piece in pieces:
        if piece == goal:  # If piece can go to the goal cell, it's the best possible move
            score += 100
        else:
            distance_to_goal = self.manhattan_distance(piece, goal)
            score += 10 / (distance_to_goal + 1)  # Closer pieces have higher values
            if piece in blocked:  # If piece can capture an enemy piece (blocked)
                score += 20
    return score
```

```python
def evaluate_defensive(self):
    if(self.playing == 1):
        my_pieces = len(self.player1) - len(self.p1blocked)
        opponent_pieces = len(self.player2) - len(self.p2blocked)
    else:
        my_pieces = len(self.player2) - len(self.p2blocked)
        opponent_pieces = len(self.player1) - len(self.p1blocked)
    return my_pieces - opponent_pieces
```

```python
def monte_carlo_tree_search(root, n_iter):
    for _ in range(n_iter):
        print("Iteration: ", _)
        leaf = traverse(root)
        simulation_result = rollout(leaf)
        backpropagate(leaf, simulation_result)
    return best_child(root, 0)
```

Resultados Experimentais

Algoritmo	Vitórias	Derrotas	Número Médio de Jogadas para Ganhar/Perder	Tempo Médio para Obter Jogada (Segundos)
MiniMax Atacante com profundidade 2	6	8	27	0,02
MiniMax Atacante com profundidade 3	9	5	24	0,28
MiniMax Atacante com profundidade 4	8	6	22	2,92
MiniMax Defensivo com profundidade 2	5	5	26	0,01
MiniMax Defensivo com profundidade 3	2	8	27	0,06
MiniMax Defensivo com profundidade 4	9	1	24	0,40
Monte Carlo Tree Search	11	3	58	2,9
Random	0	14	53	0,0

Nota: Alguns jogos MiniMax Defensivo vs MiniMax Defensivo não foi possível obter um vencedor.

Avaliação dos Resultados Experimentais

Após a análise dos resultados obtidos podemos concluir que:

- Quando o mesmo algoritmo se enfrenta (ex: MonteCarlo vs MonteCarlo), não se notou grande diferença entre o número de vezes que cada Jogador (1 ou 2) ganhou revelando que quando o algoritmo é o mesmo, o Jogador que começa não tem grande influência no resultado final
- Apesar disso, em certos casos, começar Player 1 ou Player 2 foi decisivo para ganhar a partida (ex: MiniMax com profundidade 2 ganhou contra MiniMax com profundidade 3 quando era Jogador 1 mas perdeu quando era Jogador 2)
- Jogos entre Jogadores Defensivos resultaram num ciclo infinito de jogadas pois nenhum dos jogadores queria fazer uma jogada mais arriscada para aumentar o risco de perder
- Apesar do algoritmo de Monte Carlo Search Tree ter perdido 2 vezes contra o MiniMax Defensivo com profundidade 4, é o algoritmo mais forte pois o MiniMax Defensivo com profundidade 4 não conseguiu ganhar aos outros MiniMax Defensivos

Conclusão

Em suma, o jogo proposto foi desenvolvido com sucesso, foi concretizada a implementação de todos os diferentes modos de jogo propostos e a implementação de todos os algoritmos propostos para o projeto.

Com a elaboração deste projeto foi possível aprodundar o nosso conhecimento com a linguagem de programação Python, e a implementação dos algoritmos necessários para a realização do mesmo permitiu-nos aprofundar o nosso conhecimento e experiência em problemas e algoritmos de pesquisa adversária.

Referências

- **Pygame**
- **Websites utilizados para pesquisa:**
  - https://www.di.fc.ul.pt/~jpn/gv/aboyne.htm
  - https://sites.google.com/site/boardandpieces/list-of-games/aboyne
  - https://boardgamegeek.com/boardgame/32350/aboyne