Instituto Politécnico de Viana do Castelo

LICENCIATURA EM ENGENHARIA INFORMÁTICA

INTELIGÊNCIA ARTIFICIAL

~ Trabalho Prático 1 – Majorities~

28234 – Rafael André

28239 – Diogo Reis

Jorge Ribeiro e Luis Teófilo

- jribeiro@estg.ipvc.pt
- luisteofilo@estg.ipvc.pt

Majorities

Objetivos

No âmbito da unidade curricular de Inteligência Artificial, foi nos atribuído um jogo, com o objetivo de implementa-lo em Python.

Para isto, iremos ver como funciona o jogo e as suas regras, a metodologia utilizada pelo grupo, e o desenvolvimento .

História

Lançado em 2007 por Bill Taylor, um matemático e professor universitário na University of Canterbury.

Morreu, infelizmente, em 2021, fez também jogos como: Projex, Quaxx, and Slimetrail.

Regras do Jogo

Majorities

Majorities is a deceptively simple game based on majority rule.

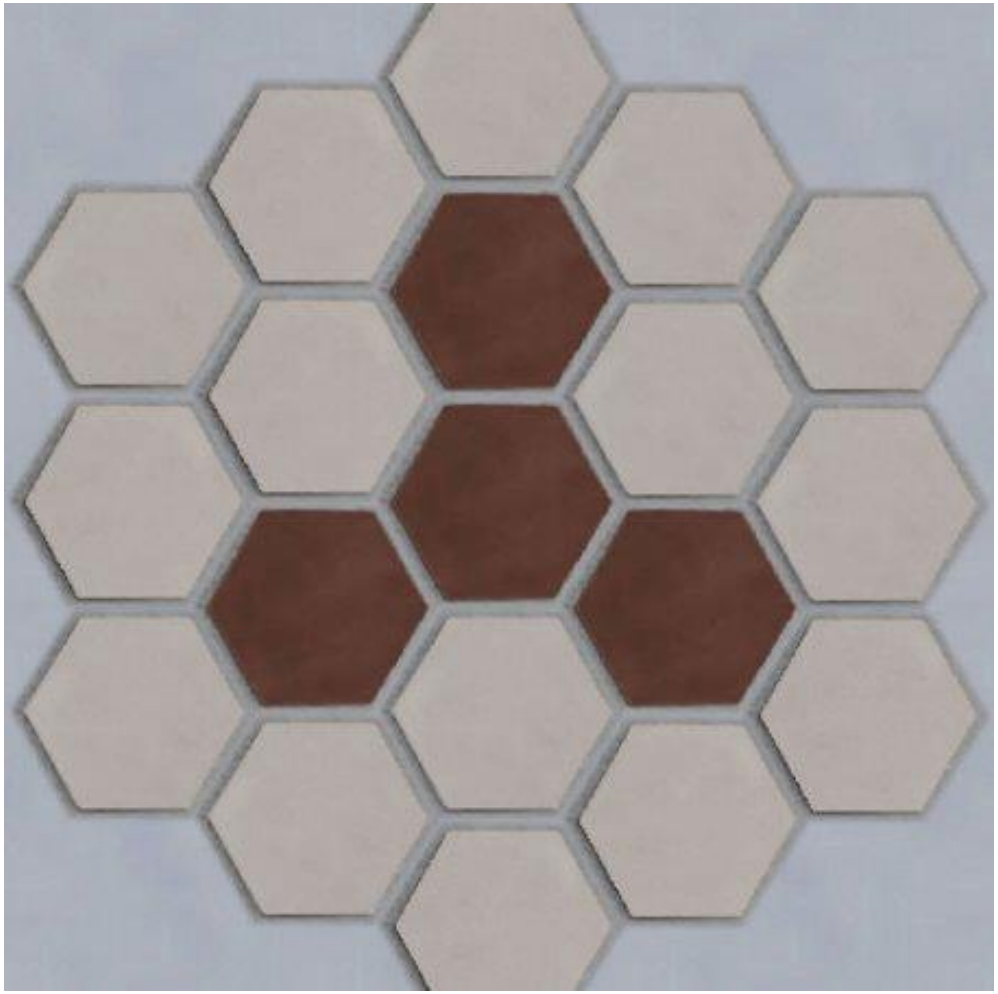
- control the majority of spaces on a line to win the line
- control the majority of parallel lines to win a direction
- control a majority of the three directions to win the game.

Starting with an empty board, white plays one stone on any empty space. After the first play, black and white alternate placing 2 stones each. When the board is full (or probably quite a while before) one player will have won.

Robots: Looks unbeatable, at least until some human devises a strategy.

Tabuleiro do Jogo

- Este é o tabuleiro que o grupo achou por bem desenvolver (3X3), uma vez que os outros tabueleiros eram 5x5 e 7x7, o que levaria a que a duração de uma partida fosse bastante elevada;1
- Para jogar existem dois tipos de peças geralmente as pretas e brancas.



Implementação

- Para implementar este código foi utilizada uma base, fornecida pelo Professor Luís Teófilo, que serviu para ter algumas funções, sendo que as funções das grids e do tabuleiro foram anuladas, uma vez que o grupo optou por outra metodologia para o posicionamento das peças e criação do tabuleiro de jogo.

Tutorial de jogo

- Para auxiliar na execução do programa, e para jogar o jogo, foi feito um vídeo que pode ser acedido a baixo:

Explicação de código

- Aqui está a função do tabuleiro que é demasiado extensa para colocar na totalidade, mas basicamente existem 15 posições e o jogador escolhe uma na primeira jogada para colocar a peça.

```
def board(self, dimensao):
    if(dimensao==3):
        print(f"                  _____                    |", end="     \n")
        print(f"                 /     \\                   |", end="     \n")
        print(f"           _____/   {MajoritiesState.print_cell(0,tabuleiro)}
        print(f"          /     \       /     \\           |", end="     \n")
        print(f"     _____/   {MajoritiesState.print_cell(1,tabuleiro)}   \____
        print(f"    /     \       /     \       /     \\    |", end="     \n")
        print(f"   /   {MajoritiesState.print_cell(3,tabuleiro)}   \_____/
        print(f"   \       /     \       /     \       /   |", end="     \n")
        print(f"    \_____/   {MajoritiesState.print_cell(5,tabuleiro)}   \
        print(f"    /     \       /     \       /     \\    |", end=f"
        print(f"   /   {MajoritiesState.print_cell(7,tabuleiro)}   \_____/
        print(f"   \       /                   \       /   |", end=f"
        print(f"    \_____/           _____     \_____/    |", end=f"     ___/
        print(f"    /     \       /     \       /     \\    |", end=f"    / {Ma
        print(f"   /  {MajoritiesState.print_cell(9,tabuleiro)}   \_____/  {Maj
        print(f"   \       /     \       /     \       /   |", end=f"    / {Maj
        print(f"    \_____/  {MajoritiesState.print_cell(12,tabuleiro)}   \____
        print(f"          \       /     \       /          |", end=f"    / {Maj
        print(f"           \_____/  {MajoritiesState.print_cell(14,tabuleiro)}
        print(f"                 \       /                |", end=f"        \_
        print(f"                  \_____/                 |", end=f"
```

```
def print_cell(cell, tabuleiro):
    players = {
        'A': '\033[96m⦿\033[0m',
        'B': '\033[91m⦿\033[0m',
    }

    for p,v in players.items():
        if tabuleiro[cell] == p:
            return v if cell < 9 else v + ' '

    return tabuleiro[cell]
```

Explicação de código

- A heurística do jogo é simples, ganhando duas das três direções possíveis, vence-se o jogo.
- Ao formular a função do check_winner, o grupo teve de criar um array para cada uma das direções, para motivos de controlo.
- Este pedaço da função verifica todas as direções e vai incrementando o valor das váriaveis Vencedorp, que cada uma corresponde a um jogador.

```
def check_winner():
    Vencedorp1 = 0
    Vencedorp2 = 0

    if(PontDirecao1[0] == 3):
        Vencedorp1 += 1
    if(PontDirecao1[1] == 3):
        Vencedorp2 += 1

    if(PontDirecao2[0] == 3):
        Vencedorp1 += 1
    if(PontDirecao2[1] == 3):
        Vencedorp2 += 1

    if(PontDirecao3[0] == 3):
        Vencedorp1 += 1
    if(PontDirecao3[1] == 3):
        Vencedorp2 += 1

    if Vencedorp1 >= 2:
        return 1
    if Vencedorp2 >= 3:
        return 2

    pecastabuleiro = 0
    for i in tabuleiro:
        if(i == f'A' or i == f'B'):
            pecastabuleiro += 1
```

Explicação de código

- Esta parte do código verifica quem ganhou mais direções quando o tabuleiro está preenchido.

```
if pecastabuleiro == 15:
    Vencedorp1 = 0
    Vencedorp2 = 0
    if(PontDirecao1[0] > PontDirecao1[1]):
        Vencedorp1 += 1
    else:
        Vencedorp2 += 1

    if(PontDirecao2[0] > PontDirecao2[1]):
        Vencedorp1 += 1
    else:
        Vencedorp2 += 1

    if(PontDirecao3[0] > PontDirecao3[1]):
        Vencedorp1 += 1
    else:
        Vencedorp2 += 1

    if(Vencedorp1 > Vencedorp2):
        return 1
    else:
        return 2

return 0
```

Explicação de código

- Esta função, verifica se a jogada é valida ou não, caso não esteja entre 1 e 15, irá retornar com erro.
- E, se a posição já estiver ocupada, também retornará um erro.

```
def validate_action(self, action: MajoritiesAction, x, dimensao) -> bool:
    if(dimensao == 3):
        if(x< 1 or x > 15):
            return True
        if(tabuleiro[x-1] == f'A' or tabuleiro[x-1] == f'B'):
            return True

        return False
```

Explicação de código

- Para a implementação do jogo o grupo conseguiu aplicar os quatro níveis de dificuldade, ou seja, é possível jogarem dois humanos, humano contra o random, humano contra greedy, e por fim, humano contra montecarlo.
- Estes foram os 3 algoritmos que o grupo conseguiu implementar.

Explicação de código - Humano vs Monte Carlo

- Aqui está a classe MonteCarloMajoritiesPlayer, que é responsável pela jogada do jogador MonteCarlo.
- Na função montecarlo, serão feitas as simulações para aumentar a precisão do montecarlo, que necessita de simular vários jogos para obter as probabilidades para cada posição do tabuleiro, aqui irá atribuir a pontuação consoante a jogada, e no fim irá obter-se a probabilidade de vitória se jogar numa certa posição.

```python
class MonteCarloMajoritiesPlayer(MajoritiesPlayer):
        super().__init__(name)

    def montecarlo(state: MajoritiesState,new_state,Direcao1,Direcao2,Direcao3,playerV):
        win = 0
        lost = 0
        d1 = []
        d2 = []
        d3 = []

        for play in range(500):
            playerV = True
            end = 0
            state_clone,d1,d2,d3 = state.clone(new_state,Direcao1,Direcao2,Direcao3)
            while end == 0:
                playerV=False
                play = random.choice(state.get_possible_actions_cloned(state_clone))
                state.update_cloned( state, play, playerV, d1,d2,d3, state_clone)
                play = random.choice(state.get_possible_actions_cloned(state_clone))
                state.update_cloned( state, play, playerV, d1,d2,d3, state_clone)

                playerV = True
                if(state.check_winner_cloned(d1,d2,d3,state_clone)!=0):
                    break

                play = random.choice(state.get_possible_actions_cloned(state_clone))
                state.update_cloned( state, play, playerV, d1,d2,d3, state_clone)
                play = random.choice(state.get_possible_actions_cloned(state_clone))
                state.update_cloned( state, play, playerV, d1,d2,d3, state_clone)

                if(state.check_winner_cloned(d1,d2,d3,state_clone)!=0):
                    break

            if(state.check_winner_cloned(d1,d2,d3,state_clone) == 2):
                    win += 1
            if(state.check_winner_cloned(d1,d2,d3,state_clone) == 1):
                    lost += 1


        return (win * 100)/(win + lost)
```

(variable) state_clone: Any

Explicação de código - Humano vs Humano

- Aqui está a classe HumanoMajoritiesPlayer, que é responsável pela jogada de um jogador.
- Na função get_action, inicialmente iguala-se a variável z a False para obter controlo sobre a jogada do jogador, ou seja, se jogou um peça válida ou não, aí vai ao validate_action no ficheiro state.py, depois dá update ao board, e por fim verifica se existe vencedor, se houver irá retornar o valor 1 ou 2, caso seja 1 o jogador 1 ganhou a partida, caso seja 2 o jogador 2 ganhou a partida, isto para todo o tipo de jogos.

```
class HumanoMajoritiesPlayer(MajoritiesPlayer):

    def __init__(self, name):
        super().__init__(name)

    def get_action(self, state: MajoritiesState, playerV,dimensao):
        z = False

        state.board(self, dimensao)
        jogada = int(input("\nEscolha onde quer jogar:"))
        z=state.validate_action(self,MajoritiesAction,jogada,dimensao)
        if z == True:
            while  z == True:
                jogada = int(input("\nLugar Ocupado/Nao Valido\n\nEscolha outro lugar:"))
                z=state.validate_action(self,MajoritiesAction,jogada,dimensao)

        state.update(state,jogada,playerV,dimensao)
        state.clear()
        if(int(state.check_winner() != 0)):
            state.board(self, dimensao)

        return int(state.check_winner())
```

Explicação de código - Humano vs Random

- Aqui está a classe RandomMajoritiesPlayer, que é responsável pela jogada do jogador Random.
- Na função get_action, a variável jogada será um número entre 1 e 15, que são todas as posições possíveis do tabuleiro, aí irá fazer-se o validate_action, se já estiver ocupada a poisção selecionada, irá gerar outra até não estar ocupada, e de seguida dá os updates ao board e verifica se existe, ou não vencedor.

```
class RandomMajoritiesPlayer(MajoritiesPlayer):

    def __init__(self, name):
        super().__init__(name)

    def get_action(self, state: MajoritiesState, playerV):
        dimensao=3
        jogada = randint(1, 15)
        z=state.validate_action(self,MajoritiesAction,jogada,dimensao)
        if z == True:
            while  z == True:
                jogada = randint(1, 15)
                z=state.validate_action(self,MajoritiesAction,jogada,dimensao)

        state.update(state,jogada,playerV,dimensao)
        if(int(state.check_winner() != 0)):
            state.board()
        return int(state.check_winner())
```

Explicação de código - Humano vs Greedy

- Aqui está a classe GreedyMajoritiesPlayer, que é responsável pela jogada do jogador Greedy.
- Na função get_action, será visto as ações possíveis, e serão vistas também as melhores ações, o que o Greedy irá fazer será ir atrás da melhor jogada para ele, consoante as jogadas disponíveis.

```python
class GreedyMajoritiesPlayer(MajoritiesPlayer):

    def __init__(self, name):
        super().__init__(name)

    def get_action(self, state: MajoritiesState, playerV):
        available_actions = state.get_possible_actions(self)
        max_score = -1
        best_action = []

        for action in available_actions:
            for action2 in available_actions:
                if action != action2:
                    score = state.bestplay(self, state, action, action2, playerV)
                    if score > max_score:
                        max_score = score
                        best_action.clear()
                        best_action.append([action,action2])
                    if score == max_score:
                        best_action.append([action,action2])

        dimensao = 3
        jogada = random.choice(best_action)
        state.update(state, jogada[0], playerV,dimensao)
        state.update(state, jogada[1],playerV,dimensao)

        state.clear()
        if(int(state.check_winner() != 0)):
            state.board(self,dimensao)

        return int(state.check_winner())
```

Explicação de código - Humano vs Monte Carlo

- Na função get_action, serão aplicadas as percentagens de vitória às diversas combinações, e irão ser aplicadas as jogadas no tabuleiro original, poruqe anteriormente era feita uma cópia do tabuleiro atual para a simulação.

```python
def get_action(self,state: MajoritiesState, playerV):
    max_score = -1
    best_action = []

    actions = state.get_possible_actions(self)
    if len(actions) == 2:
        dimensao=3
        state.update(state, actions[0], playerV,dimensao)
        state.update(state, actions[1],playerV,dimensao)

        state.board(self,dimensao)

        return int(state.check_winner())

    for action in actions:
        for action2 in actions:
            if action != action2:
                new_state,d1,d2,d3 = state.sim_play(action,action2,playerV,state)
                playerValor=playerV
                score = MonteCarloMajoritiesPlayer.montecarlo(state,new_state,d1,d2,d3,playerValor)
                print(f'Pecas {action} e {action2} = {score}')
                if score > max_score:
                    max_score = score
                    best_action.clear()
                    best_action.append([action,action2])
                if score == max_score:
                    best_action.append([action,action2])

    dimensao=3
    jogada = random.choice(best_action)
    state.update(state, jogada[0], playerV,dimensao)
    state.update(state, jogada[1],playerV,dimensao)

    if(int(state.check_winner() != 0)):
        state.board(self,dimensao)

    return int(state.check_winner())
```

o teu • de partida

www.ipvc.pt