1. Notebook do projeto

Este documento tem como objetivo estruturar e documentar o projeto desenvolvido no âmbito da unidade curricular de **Inteligência Artificial**.

Ao longo deste notebook, serão abordadas as decisões tomadas durante o desenvolvimento do trabalho, detalhadas as implementações dos algoritmos utilizados e apresentadas as conclusões obtidas a partir dos resultados alcançados.

Índice

1.1 Main do Graphical User Interface (GUI)

Apesar da recomendação inicial, decidiu-se implementar uma **interface gráfica com Pygame**, pois tornará o jogo visulamente mais apelativo.
A célula de código abaixo é responsável por executar as implementações do jogo, das quais se destacam:

- **Player vs Player -** Jogar um jogo de 4 em linha contra um oponente humano;
- **Player vs AI -** Jogar um jogo de 4 em linha contra um algoritmo de Monte Carlo Tree Search, podendo escolher a dificuldade do algoritmo;
- **AI vs AI -** Observar um jogo de Monte Carlo vs Monte Carlo, podendo escolher o número de iterações para cada modelo;
- **Board Editor -** Editar um tabuleiro de 4 em Linha e, se a posição for válida, escolher umas das implementações de Decision Tree para prever o que o Monte Carlo jogaria na posição imputada.

> É importante correr o jogo com as bibliotecas adequadas e, para tal, aconselhamos a criar um ambiente virtual com base no ficheiro requiremenst.txt. Este processo pode ser automatizado correndo estes blocos de código. É imperativo ser por pip pois existem packages não disponiveis através do conda.

```
conda create -n nome_do_env

conda activate nome_do_env

pip install -r requirements.txt
```

In [ ]:
```
from GameMain import ConnectFourGUI
```

```
gui = ConnectFourGUI()
gui.mainMenu()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 4
      1 from GameMain import ConnectFourGUI
      3 gui = ConnectFourGUI()
----> 4 gui.mainMenu()

File c:\Users\diogo\OneDrive\Documents\GitHub\MCTS_and_DecisionTree_for_ConnectFo
urGame\GameMain.py:355, in ConnectFourGUI.mainMenu(self)
    353 if event.type == pygame.QUIT:
    354     pygame.quit()
--> 355     exit()
    356 if event.type == pygame.MOUSEBUTTONDOWN:
    357     x, y = event.pos

NameError: name 'exit' is not defined
```

> **Nota:** é necessário reiniciar o kernel para voltar a correr a GUI. Quando fechar o jogo, irá retornar um erro por causa comunicação entre dependências dos ficheiros

2. Monte Carlo Tree Search (MCTS)

Este é o modelo que constitui o cerne deste projeto. O algoritmo pode ser dividido em 4 fases:

1. **Seleção:** A partir da raiz, percorre-se a árvore selecionando sucessivamente os nós filhos de acordo com uma política (por exemplo, UCB) até chegar a um nó folha.
2. **Expansão:** Se o nó folha não representa um estado terminal, um ou mais filhos são criados e adicionados à árvore.
3. **Simulação:** A partir do novo nó, realiza-se uma simulação (jogadas aleatórias) até que seja alcançado um estado terminal.
4. **Retropropagação:** O resultado da simulação é propagado de volta pela árvore, atualizando os nós visitados de acordo com a estatística selecionada.

Esses passos são repetidos um número predefinidos de vezes.

2.1 Aplicação Prática

No contexto deste projeto, cada nó representa uma posição do tabuleiro, e cada aresta uma jogada válida, tendo portanto cada nó como seus filhos os estados que resultam de jogadas válidas. Assim sendo, o modelo é chamado uma vez por jogada e, dado o tabuleiro atual,simula uma quantidade consideravél de jogos possíveis e retorna a coluna à qual estiver associada o maior número de visitas (decisão discutida adiante).

Segue-se o código da classe MCTS como descrito no ficheiro MCTS.py, e uma breve descrição do seu funcionamento:

```
import math
import random
from typing import Tuple, Any, List, Union

import utils.config as config

from Game.ConnectFour import ConnectFour
from MCTS.node import Node


class MonteCarlo_Single(object):
    """
    Monte Carlo Tree Search algorithm.

    Methods
    -------
    search(root: Node) -> int
        Search the best move from the root node.
     selection(node: Node, turn: int) -> (Node, int)
        Select the best node to expand.
    expansion(node: Node) -> Node
        Expand the node by adding a new child.
    simulation(state_init: ConnectFour, turn: int) -> float
        Simulate a random game from the initial state.
    backpropagation(node: Node, reward: float, turn: int) -> None
        Backpropagate the reward of the simulation to the root node.
    best_child(node: Node) -> Node
        Return the best child of the node.
    """
    def __init__(self, iteration: int = config.ITERATION, exploration:
float = config.EXPLORATION, debug: bool = False) -> None:
        """
        Initialize the Monte Carlo Tree Search algorithm.
        """
        self.iteration = iteration
        self.exploration = exploration
        if debug:
            print(f"Monte Carlo Tree Search: iteration={iteration},
exploration={exploration}")

    def search(self, root: Node) -> tuple[Any, list[Any]]:
        """
        Search the best move from the root node.

        Parameters
        ----------
        root: the root node of the search tree

        Returns
        -------
        int: the best move
        """
```

```
            for _ in range(self.iteration):
                node, turn = self.selection(root, -1)
                reward = self.simulation(node.state, turn)
                self.backpropagation(node, reward, turn)

            prob = []
            for child in root.children:
                prob.append(child.visits / root.visits)

            ans = max(root.children, key=lambda c: c.visits)
            return ans.state.last_move[1], prob

        def selection(self, node: Node, turn: int) -> tuple[Node, int]:
            """
            Select the best node to expand.

            Parameters
            ----------
            node: the node to start the selection from
            turn: the turn of the player who played the move leading to
    this node

            Returns
            -------
            node: the node to expand
            turn: the turn of the player who played the move leading to
    this node
            """
            while not node.is_terminal():
                if not node.fully_explored():
                    return self.expansion(node), -1 * turn
                else:
                    node = self.best_child(node)
                    turn *= -1

            return node, turn

        @staticmethod
        def expansion(node: Node) -> Node:
            """
            Expand the node by adding a new child.

            Parameters
            ----------
            node: the node to expand

            Returns
            -------
            node: the new child
            """
            free_cols = node.state.legal_moves()

            for col in free_cols:
                if col not in node.children_move:
                    new_state = node.state.copy()
                    new_state.play(col)
```

```
                    node.add_child(new_state, col)
                    break

        return node.children[-1]

    @staticmethod
    def simulation(state_init: ConnectFour, turn: int) -> float:
        """
        Simulate a random game from the initial state.

        Parameters
        ----------
        state_init: the initial state of the game
        turn: the turn of the player who played the move leading to
this node

        Returns
        -------
        reward: the reward of the simulated game
        """
        state = state_init.copy()

        while not state.is_over():
            state.play(random.choice(state.legal_moves()))
            turn *= -1

        reward_bool = state.is_over()

        if reward_bool and turn == -1:
            reward = 1.0
        elif reward_bool and turn == 1:
            reward = -1.0
        else:
            reward = 0.0
        return reward

    @staticmethod
    def backpropagation(node: Node, reward: float, turn: int) -> None:
        """
        Backpropagate the reward of the simulation to the root node.

        Parameters
        ----------
        node: the node to start the backpropagation from
        reward: the reward of the simulation
        turn: the turn of the player who played the move leading to
this node

        Returns
        -------
        none
        """
        while node is not None:
            node.visits += 1
            node.reward -= turn * reward
            node = node.parent
```

```
            turn *= -1

    def best_child(self, node: Node) -> Node:
        """
        Return the best child of the node.

        Parameters
        ----------
        node: the node to select the best child from

        Returns
        -------
        node: the best child
        """
        best_score = -float("inf")
        best_children = None

        for child in node.children:
            exploitation = child.reward / child.visits
            exploration = math.sqrt(math.log2(node.visits) /
child.visits)
            score = exploitation + self.exploration * exploration

            if score == best_score:
                if child.visits > best_children.visits:
                    best_children = child
            elif score > best_score:
                best_score = score
                best_children = child

        return best_children
```

2.1.1 Explicação das Funções da Classe

- `__init__`
  Inicializa o algoritmo MCTS, definindo o número de iterações e o parâmetro de exploração.
- `search`
  Executa o ciclo principal do MCTS: seleção, expansão, simulação e retropropagação, retornando a melhor jogada e os scores associados a cada filho da raiz.
- `selection`
  Percorre a árvore a partir de um nó, seguindo a política de seleção (ex: UCB), até encontrar um nó folha ou não totalmente explorado.
- `expansion`
  Expande um nó folha, criando e adicionando um novo filho correspondente a uma jogada ainda não explorada.
- `simulation`
  Realiza uma simulação (jogo aleatório) a partir do estado atual até um estado terminal, retornando a recompensa do resultado.

- `backpropagation`
  Propaga o resultado da simulação de volta pela árvore, atualizando as estatísticas dos nós visitados.

- `best_child`
  Seleciona o melhor filho de um nó com base no critério de exploration/exploitation (UCB), para guiar a busca nas próximas iterações.

Ou seja, a escolha de uma jogada (executada pela função `search`) executa as 4 fases mencionadas acima, iteradas tantas vezes quantas as definidas no ficheiro config.py, para cada dificuldade selecionada.

2.1.2 UCB

Utilizamos como métrica para avaliação dos nodes o UCB, a estatística mais comum em implementações de MCTS, e que pode ser descrita pela seguinte fórmula:

$$UCB = \frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

onde:

- w_i: número de vitórias do nó i
- s_i: número de simulações do nó i
- s_p: número de simulações do nó pai
- c: parâmetro de *exploitation*

Em que c constitui uma constante que equilibra o quanto a *exploration* é favorecida sobre a *exploitation*. Esta constante, neste projeto está também definida no ficheiro de configuração e toma o valor de 1.414, isto é, aproximadamente $\sqrt{2}$, também um valor muito usado por defeito nestas áreas como por exemplo neste artigo de Auer, Cesa-Bianchi & Fischer (2002), Holanda.

Em específico, o UCB é calculado nestas linhas do ficheiro MCTS.py, na função `best_child`:

```
exploitation = child.reward / child.visits
exploration = math.sqrt(math.log2(node.visits) / child.visits)
score = exploitation + self.exploration * exploration
```

2.1.3 Escolha da próxima jogada

Acabadas as iterações do MCTS, cada jogada válida tem a si associada dois valores:

- **Visited (V):** Representa a quantidade de simulações feitas sobre esse nó nas iterações de de MCTS. Vai ser, portanto, $n \mid n \in \mathbb{N},\ 0 \leq n \leq \text{iter}$ em que $iter$ é o número de iterações do MCTS.
- **Reward (R):** Representa o somatório dos rewards das iterações de MTCS. Um reward em cada iteração é definido por:

- **1** se o próprio ganha;
- **0** se há um empate;
- **-1** se o adversário ganha.

Resta apenas definir o critério de escolha, isto é, se consideramos o melhor lance aquele que foi o mais visitado, o mais recompensado ou um equilíbrio das duas características.

Testando as várias hipóteses, no entanto, verificou-se que a melhor performance é obtida ao selecionar o filho com o maior número de visitas. Isto provavelmente deve-se ao facto de um nó ter um elevado número de visitas indicar que, repetidamente, obteve os maiores valores de score UCB, revelando-se consistentemente como o mais promissor.

2.1.4 Tempo de execução

Ao constatar, naturalmente, que um número superior de iterações significa uma melhor performance, tentou-se executar modelos com mais e mais iterações. No entanto, neste processo o modelo começou também a demorar cada vez mais tempo a calcular a sua próxima jogada, o que piora a experiência de jogo.

No sentido de corrigir isto, implementou-se uma outra classe de MCTS que utiliza *paralel processing*, executando várias iterações ao mesmo tempo, e sendo por isso significativamente mais rápido que a sua contraparte, que utiliza apenas um *core* para todas as tarefas.

Eis essa outra implementação, denominada `MonteCarlo` no ficheiro MCTS_optimized.py:

```
import math
import random
import os
from concurrent.futures import ProcessPoolExecutor
from typing import Tuple, Any, Dict

import utils.config as config
from Game.ConnectFour import ConnectFour
from MCTS.node import Node


def worker_mcts(state: ConnectFour, iterations: int, exploration:
float) -> Dict[int, Tuple[float, int]]:
    """
    Each worker runs its own mini-MCTS rooted at the same state.
    Returns: {move: (total_reward, total_visits)}
    """
    root = Node(state.copy())

    def selection(node: Node, turn: int) -> Tuple[Node, int]:
        while not node.is_terminal():
            if not node.fully_explored():
                return expansion(node), -1 * turn
            node = best_child(node)
            turn *= -1
```

```
            return node, turn

        def expansion(node: Node) -> Node:
            for col in node.state.legal_moves():
                if col not in node.children_move:
                    new_state = node.state.copy()
                    new_state.play(col)
                    node.add_child(new_state, col)
                    break
            return node.children[-1]

        def simulation(state: ConnectFour, turn: int, max_depth: int = 20)
    -> float:
            state = state.copy()
            moves = 0
            while not state.is_over() and moves < max_depth:
                legal = state.legal_moves()
                if 3 in legal:
                    state.play(3)
                else:
                    state.play(random.choice(legal))
                turn *= -1
                moves += 1

            if state.is_over():
                return 1.0 if turn == -1 else -1.0
            return 0.0

        def backpropagation(node: Node, reward: float, turn: int) -> None:
            while node is not None:
                node.visits += 1
                node.reward -= turn * reward
                node = node.parent
                turn *= -1

        def best_child(node: Node) -> Node:
            best_score = -float("inf")
            best_node = None
            log_visits = math.log(node.visits + 1)
            for child in node.children:
                exploit = child.reward / (child.visits + 1e-8)
                explore = math.sqrt(log_visits / (child.visits + 1e-8))
                score = exploit + exploration * explore

                if score == best_score:
                    if child.visits >= best_node.visits:
                        best_node = child
                elif score > best_score:
                    best_score = score
                    best_node = child
            return best_node

        for _ in range(iterations):
            node, turn = selection(root, -1)
            reward = simulation(node.state, turn)
            backpropagation(node, reward, turn)
```

```
        move_stats = {}
        for _, child in enumerate(root.children):
            move = child.state.last_move[1]
            move_stats[move] = (child.reward, child.visits)

        return move_stats


class MonteCarlo:


    def __init__(self, iteration: int = config.ITERATION, exploration:
float = config.EXPLORATION, debug: bool = False):

        self.iteration = iteration
        self.exploration = exploration
        self.cpu_cores = max(1, os.cpu_count() or 1)
        self.debug = debug


        if self.debug:
            print(f"Using {self.cpu_cores} CPU cores for MCTS.")
            print(f"Iterations per worker: {self.iteration //
self.cpu_cores}")
            print(f"Exploration factor: {self.exploration}")
            print(f"Total iterations: {self.iteration}")




    def search(self, root: Node) -> tuple[Any, list[Any]]:
        iterations_per_worker = self.iteration // self.cpu_cores

        with ProcessPoolExecutor(max_workers=self.cpu_cores) as
executor:
            futures = [executor.submit(worker_mcts, root.state,
iterations_per_worker, self.exploration)
                       for _ in range(self.cpu_cores)]

            all_stats = [f.result() for f in futures]

        merged_stats: Dict[int, Tuple[float, int]] = {}

        for stat in all_stats:
            for move, (reward, visits) in stat.items():
                if move not in merged_stats:
                    merged_stats[move] = (reward, visits)
                else:
                    r, v = merged_stats[move]
                    merged_stats[move] = (r + reward, v + visits)

        for move, (reward, visits) in merged_stats.items():
            found = False
            for child in root.children:
                if child.state.last_move[1] == move:
```

```
                    child.reward += reward
                    child.visits += visits
                    found = True
                    break
            if not found:
                new_state = root.state.copy()
                new_state.play(move)
                new_child = Node(new_state, root)
                new_child.reward = reward
                new_child.visits = visits
                root.children.append(new_child)
                root.children_move.append(move)
            root.visits += visits

        prob = [child.visits / root.visits for child in root.children]

        ans = max(root.children, key=lambda c: c.visits)
        return ans.state.last_move[1], prob

    def best_child(self, node: Node) -> Node:
        best_score = -float("inf")
        best_children = None
        log_parent_visits = math.log(node.visits + 1)

        for child in node.children:
            exploitation = child.reward / (child.visits + 1e-8)
            exploration = math.sqrt(log_parent_visits / (child.visits +
1e-8))
            score = exploitation + self.exploration * exploration

            if score == best_score:
                if child.visits >= best_children.visits:
                    best_children = child
            elif score > best_score:
                best_score = score
                best_children = child

        return best_children
```

Mesmo assim, provou-se que *paralel processing* não é sempre a solução mais rápida. Isto acontece porque, essencialmente cada *core* do CPU está a realizar MCTS na sua própria árvore e, apenas quando **todos os *cores*** já tenham acabado as suas tarefas é que todas as árvores podem ser unidas, o que computacionalmente é trabalhoso.

Assim, estudando este tema mais a fundo, chegou-se à conclusão que a classe `MonteCarlo` é mais eficiente que a classe `MonteCarlo_Single` a partir das **5000 iterações**. Assim sendo, dependedo do modo de jogo selecionado, e dependendo da dificuldade selecionada (ou do número de iterações selecionadas no caso de AI vs AI) seleciona-se o modelo mais rápido para a situação.

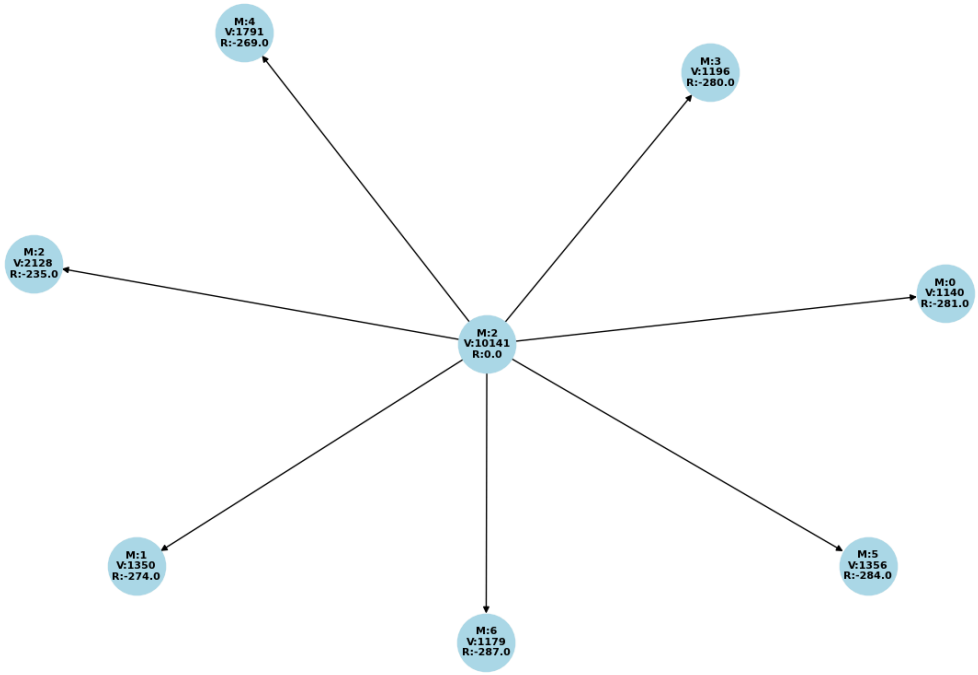
2.4 Visualização da árvore

Finalmente, no sentido de melhor compreender o funcionamento do modelo, e mesmo para *debugging*, criou-se um ficheiro Visualize_MCtree.py que, ao ser implementado na GameMain permite ao jogador ativar uma nova funcionalidade do **Modo de Debugging**.

Este modo ativa prints de debug, que contêm informações como:

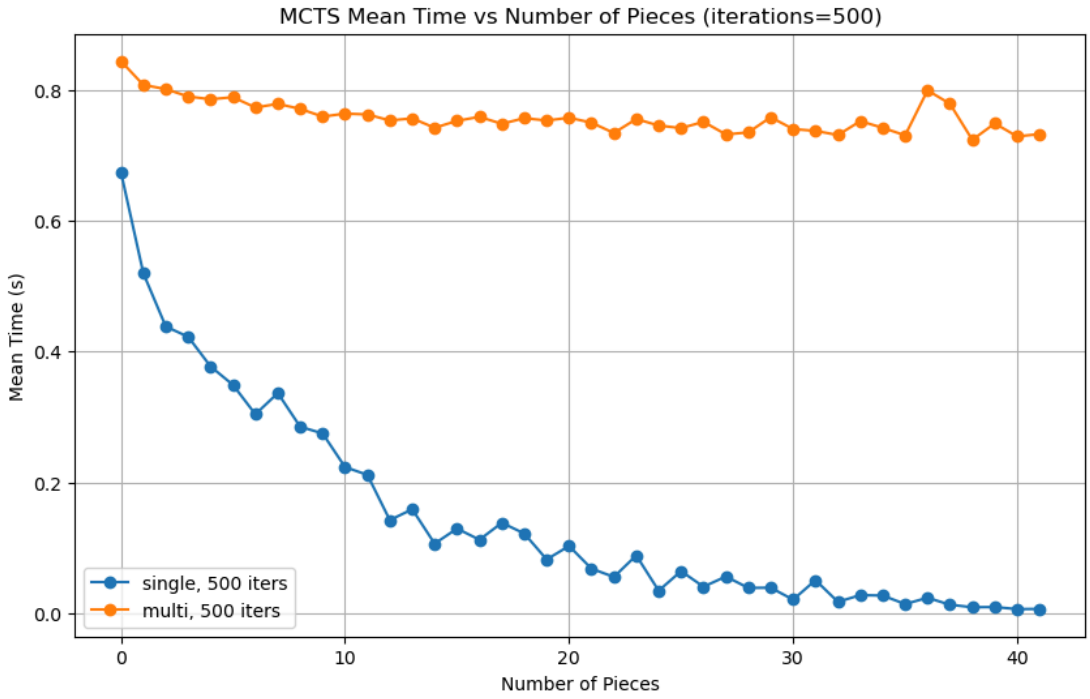
- O tempo de resposta da IA
- Quantos *cores* do CPU estão a ser usados (se estiver a ser usado mais que um)
- O *score* dos vários nós
- Entre outros

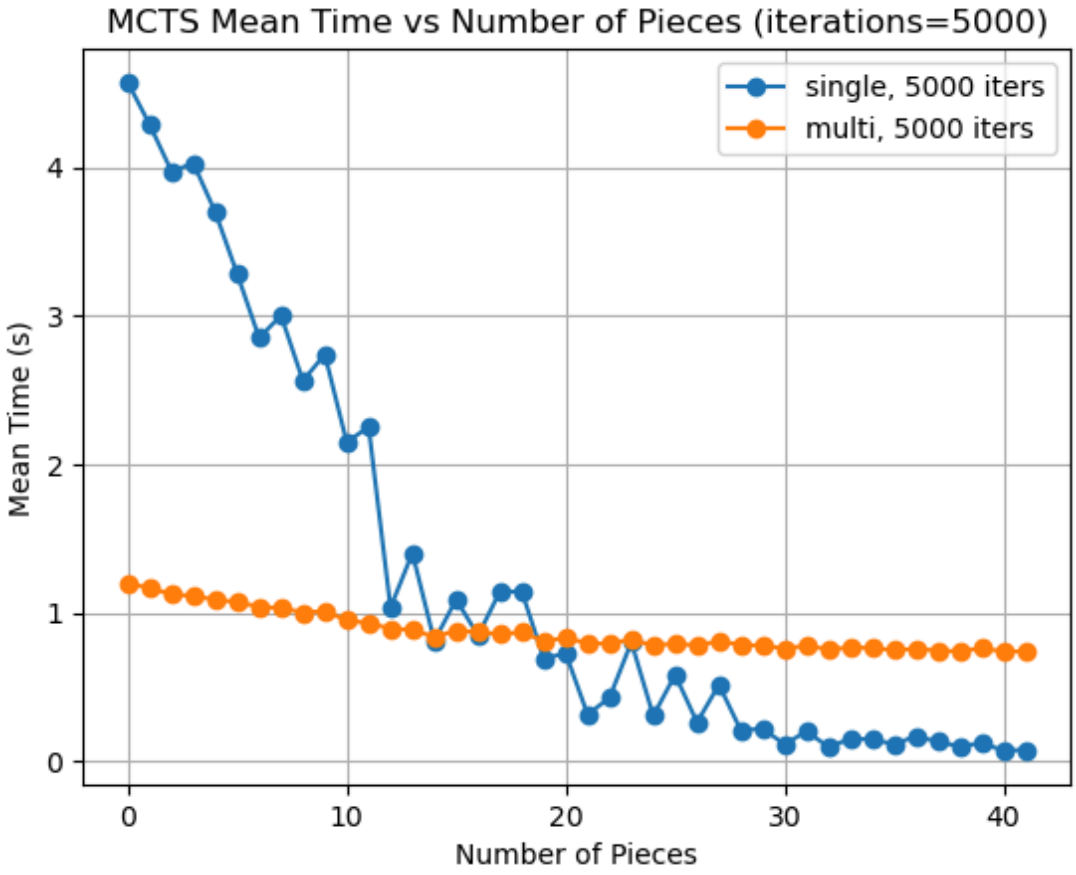
Para além disso, selecionando o modo Player vs AI e jogando contra o modelo, a cada lance feito pelo MTCS, desenhamos um grafo que representa o estado atual e os próximos estados possíveis, bem como as estatísticas a eles associados.
Eis a visulização de uma possível árvore:

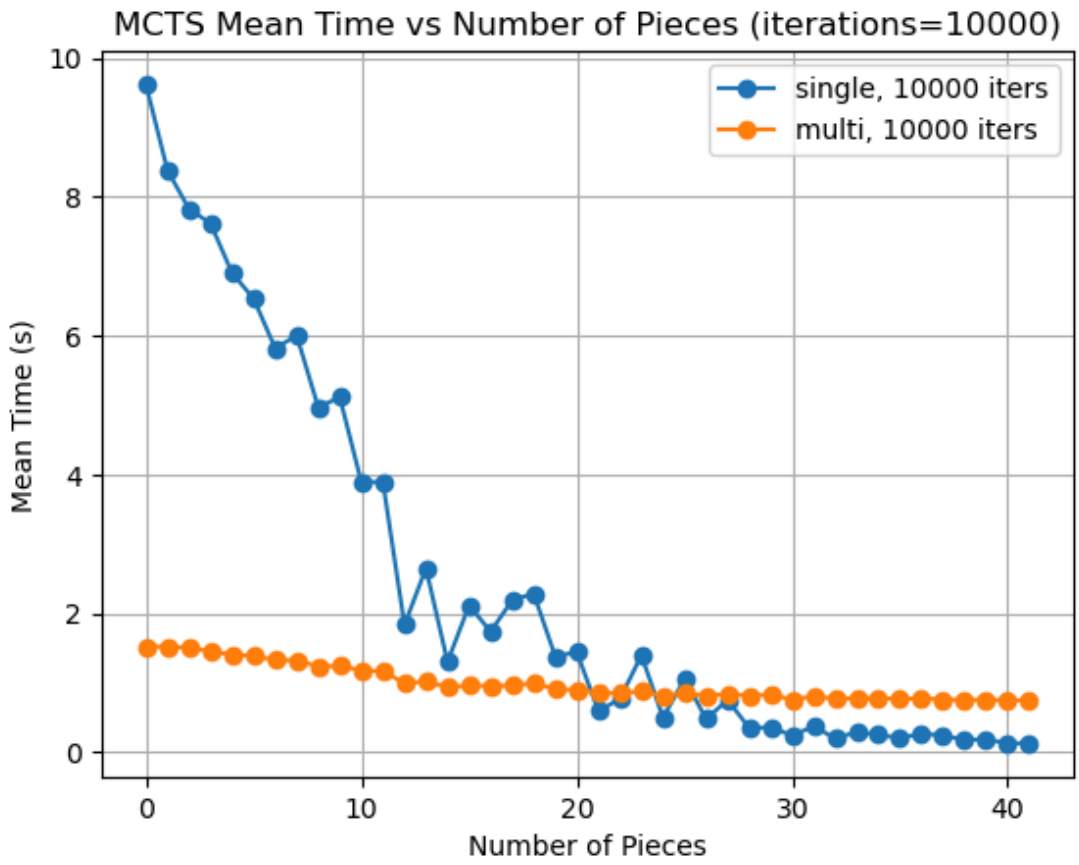


2.5 Monte Carlo Benchmarks

Na tentativa de compreender o tempo gasto por número de iterações pelo Monte Carlo Tree Search singlethread e multithread, obtiveram-se os seguintes resultados:


MCTS Mean Time vs Number of Pieces (iterations=500)
Mean Time (s)
Number of Pieces
single, 500 iters
multi, 500 iters
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40


MCTS Mean Time vs Number of Pieces (iterations=5000)
Mean Time (s)
Number of Pieces
single, 5000 iters
multi, 5000 iters
0
1
2
3
4
0
10
20
30
40



Como se pode oberservar, conclui-se que o **MCTS em singlethread apresenta melhor performance em *baixas iterações***, enquanto o **MCTS em multithread tem melhor desempenho com *altas iterações***.

- Isto deve-se ao facto de que, com poucas iterações, a **complexidade temporal da própria pesquisa** do MCTS é **menor** do que a **complexidade de junção das árvores geradas por cada core**, favorecendo a versão em singlethread.
- Já quando o número de iterações ultrapassa as **5000**, a **complexidade da pesquisa torna-se dominante**, tornando a versão multithread **mais eficiente**.

> Nota: ao executar uma pesquisa MCTS na GUI, os tempos são maiores do que os apresentados nos gráficos acima. Isto deve-se à complexidade temporal das funções auxiliares da GUI, que retardam a pesquisa do algoritmo Monte Carlo.

Por terem sido implementadas soluções de Monte Carlo em singlethread e multithread, surgiram dúvidas quanto à **equidade das decisões entre as versões singlethread e multithread para a mesma posição**, dado que a árvore de decisão multithread seria menos iterada por core.

Foram obtidos os seguintes dados relativamente à diferença percentual das decisões das diferentes versões para a mesma posição:

*__Percentage of different moves for easy level:__ 34.75%
__Percentage of different moves for medium level:__ 25.20%

Percentage of different moves for hard level: 25.73%

Assim, conclui-se que as classificações para as diferents versões singlethread e multithread diferem entre si. Este facto não intui que uma das versões performa pior, apenas que classifica diferente.

(outputs retirados no notebook MCTS_benchmarks.ipynb)*

3. Decision Tree

3.1 Introdução às Decision Trees

As **Decision Trees** são algoritmos de aprendizagem supervisionada utilizados em tarefas de **classificação** e **regressão**. Funcionam através de uma divisão recursiva dos dados com base em regras de decisão simples, formando uma estrutura em forma de árvore.

Cada divisão é guiada por uma métrica de qualidade, sendo a mais comum o **Information Gain**, que mede a redução de **entropia** após uma divisão, ou seja, o quanto uma determinada feature contribui para separar eficazmente as classes.

3.2 Aplicação no Projeto

Neste projeto, utilizamos uma árvore de decisão para **prever a jogada ótima** (i.e., a coluna a jogar) a partir de um estado do tabuleiro no jogo *4 em linha*. A jogada considerada ótima é aquela que seria escolhida por um agente **Monte Carlo Tree Search (MCTS)** com **10.000 iterações**.

3.3 Geração do Dataset

Para treinar a árvore de decisão, foi gerado um **dataset supervisionado** com aproximadamente **500 jogos simulados**, a partir dos quais foram extraídas diversas posições intermédias com respetiva jogada ideal anotada.

3.4 Evolução da Representação do Dataset

3.4.1 Estrutura Inicial

Na fase inicial do projeto, o dataset foi construído com base numa **representação direta do estado do tabuleiro** do jogo *4 em linha*. Cada linha do dataset correspondia a um momento específico de um jogo simulado, e era composta pelos seguintes elementos:

- **Um único vetor** que representava o tabuleiro completo:
  - Cada célula da grelha era codificada como:
    - `0` → célula vazia

    - `1` → peça do jogador 1
    - `-1` → peça do jogador 2
  - A ordem dos elementos era **coluna a coluna**, **do topo para a base**, espelhando a forma se interpreta uma matriz.
- Uma coluna adicional com o **número total de peças** jogadas até ao momento.
- Uma coluna que indicava o **jogador atual** a jogar (1 ou -1).
- A coluna de **output** representava a **coluna ideal** para jogar, determinada por um agente **Monte Carlo Tree Search (MCTS)** com 10.000 iterações.

3.4.2 Nova Estrutura do Dataset

Inspirado por abordagens usadas no treino de agentes para o jogo **Go** (neste artigo), a representação do estado do jogo foi reformulada para melhor refletir a informação posicional de forma explícita e neutra. O estado do tabuleiro é agora representado da seguinte forma:

- **Dois tabuleiros binários**, cada um com as mesmas dimensões do tabuleiro original:
  - Um tabuleiro representa as posições ocupadas pelo **Jogador 1** (1 para peça presente, 0 caso contrário).
  - O outro representa as posições do **Jogador 2**, com a mesma codificação.
- Cada entrada do dataset inclui:
  - Os dois vetores resultantes do **flattening** dos tabuleiros.
  - Uma feature adicional com o **número total de peças** jogadas até ao momento.
  - A **coluna de output** com a jogada ideal sugerida pelo MCTS (com 10.000 simulações).

Esta nova representação não só oferece uma **separação clara da informação entre os dois agentes**, como também permite que o modelo **aprenda padrões estratégicos específicos de cada jogador**.

Eis a estrutura dos nosso dataset:

In [ ]:
```
import pandas as pd
import os

data_file = os.path.join('datasets', 'monte_carlo_AI_VS_AI.csv')
data = pd.read_csv(data_file, delimiter=';')
data['played'] = data['played'].astype(int)
data

cel_columns = [f'cel{i}' for i in range(1, 43)]

# Update the DataFrame
data.loc[data['turn'] == -1, cel_columns] *= -1
data['turn'] = 1  # Change turn to 1
data.drop(columns=['turn'], inplace=True)
data

# Generate column names
c = [f"player1_cel{i}" for i in range(1, 43)]

# Extract and rename columns from the original dataset
```

```
player_1 = data[cel_columns].copy()
player_1.columns = c  # Rename columns to desired format

# Replace -1 with 0
player_1.replace(-1, 0, inplace=True)

c = [f"player2_cel{i}" for i in range(1, 43)]
player_2 = data[cel_columns].copy()
player_2.columns = c  # Rename columns to desired format
# Replace -1 with 0
player_2.replace(1, 0, inplace=True)
player_2.replace(-1, 1, inplace=True)

# Concatenate the two DataFrames
data = pd.concat([player_1, player_2, data['pieces'], data['played']], axis=1)
data
```

Out[ ]:

	player1_cel1	player1_cel2	player1_cel3	player1_cel4	player1_cel5	player1_cel6
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
...	...	...	...	...	...	...
14734	0	0	0	1	0	0
14735	0	0	1	0	0	0
14736	0	1	0	1	0	0
14737	0	0	1	0	0	0
14738	0	1	0	1	1	0

14739 rows × 86 columns

3.4.3 Tentativa de Randomização das Posições

Durante a fase de geração do dataset, foi inicialmente considerada a ideia de **randomizar posições do tabuleiro** em vez de as obter exclusivamente a partir de jogos completos simulados com agentes MCTS.

A motivação por trás desta abordagem era:

- Aumentar a diversidade de posições no dataset
- Reduzir o tempo necessário para simular jogos completos
- Explorar casos de jogo menos prováveis, mas ainda legais

Estratégia Testada

O processo consistia em:

1. Gerar posições aleatórias válidas (respeitando as regras do 4 em linha, como gravidade das peças)
2. Avaliar essas posições usando o agente MCTS com 10.000 iterações
3. Adicionar a posição e a jogada recomendada ao dataset

Apesar de parecer viável, esta abordagem foi **desaconselhada pelo professor da cadeira** a qual foi, então, descartada.

4. Algoritmo ID3 - Implementação da Árvore de Decisão

Neste capítulo será abordada a implementação do algoritmo **ID3** para construção de Decision Trees , conforme desenvolvido no ficheiro ID3Tree.py.

A implementação encontra-se encapsulada numa classe denominada `ID3Tree` , a qual integra os métodos essenciais para:

- Cálculo da entropia de um conjunto de rótulos,
- Determinação do ganho de informação,
- Construção recursiva da árvore de decisão,
- Classificação de novos exemplos.

4.1 Método `entropy`

O método `entropy` é definido na classe `ID3Tree` e tem como função calcular a **entropia** de um conjunto de rótulos, o que corresponde a uma medida quantitativa da **impureza** ou incerteza inerente a esse conjunto.

```
def entropy(self, labels):
    """
    Calculate the entropy of a set of labels.
    - labels: List of class labels.
    """
    total = len(labels)
    counter = Counter(labels)  # Count occurrences of each label
    return -sum((count / total) * math.log2(count / total) for count in
counter.values())
```

4.1.2 Definição e Justificação

A entropia é uma métrica fundamental na teoria da informação, utilizada para quantificar a quantidade de incerteza num conjunto de dados. No contexto do algoritmo ID3, é empregue para medir a heterogeneidade dos rótulos em cada subconjunto de dados, servindo de base para a escolha dos atributos que melhor segmentam a informação.

Formalmente, a entropia $H(S)$ de um conjunto S contendo C classes distintas é dada por:

$$H(S) = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

onde p_i representa a proporção de elementos pertencentes à classe i no conjunto S.

4.1.3 Análise do Método

- **Cálculo do total de amostras**:
  O número total de rótulos no conjunto é determinado através de `total = len(labels)`.
- **Contagem da frequência de cada classe**:
  Utiliza-se a estrutura `Counter` da biblioteca `collections` para obter a frequência absoluta de cada classe.
- **Cálculo da entropia**:

  Para cada classe, calcula-se a frequência relativa $p_i = \frac{\text{count}}{\text{total}}$ e avalia-se o termo $-p_i \log_2(p_i)$. A soma destes termos para todas as classes resulta no valor da entropia do conjunto.

4.1.4 Relevância para o Algoritmo ID3

O cálculo da entropia é indispensável para o cálculo subsequente do **ganho de informação** (information gain), que avalia a eficácia da segmentação do conjunto de dados por cada atributo. O atributo que proporciona a maior redução da entropia é escolhido para a divisão do nó, conduzindo a uma árvore de decisão mais eficaz e informativa.

4.2 Método `id3_continuous`

Esta função avalia o ganho de informação associado a um **atributo contínuo**.

```
def id3_continuous(self, data, attribute):
    """
    Calculate the information gain for a continuous attribute.
    - data: Training data.
    - attribute: The attribute to evaluate.
    """
    idx = self.attributes.index(attribute)
    values = sorted(set(row[idx] for row in data))  # Unique sorted
values of the attribute
    if len(values) == 1:
        return -1, None  # No split possible if only one unique value

    # Calculate potential thresholds
    thresholds = [(values[i] + values[i + 1]) / 2 for i in
```

```
range(len(values) - 1)]
    base_entropy = self.entropy([row[-1] for row in data])  # Entropy
of the entire dataset

    best_gain, best_thresh = -1, None
    for t in thresholds:
        # Split data into above and below threshold
        above = [row for row in data if row[idx] >= t]
        below = [row for row in data if row[idx] < t]
        p, n = len(above) / len(data), len(below) / len(data)
        # Calculate information gain
        gain = base_entropy - p * self.entropy([r[-1] for r in above])
- n * self.entropy([r[-1] for r in below])
        if gain > best_gain:
            best_gain, best_thresh = gain, t
    return best_gain, best_thresh
```

Explicação técnica

1. **Extração e ordenação dos valores do atributo**

   `values = sorted(set(row[idx] for row in data))`
   São considerados apenas os valores únicos e ordenados do atributo contínuo.

2. **Geração de limiares candidatos**

   São gerados todos os possíveis pontos médios entre pares consecutivos de valores:

   $$\text{threshold}_i = \frac{v_i + v_{i+1}}{2}$$

   Estes limiares são os pontos de corte candidatos para dividir o conjunto de dados.

3. **Cálculo do ganho de informação**

   Para cada limiar t, divide-se o conjunto em duas partes:

   - Acima do limiar: $D_{\geq t}$
   - Abaixo do limiar: $D_{<t}$

   O ganho de informação é então calculado como:

   $$\text{Gain}(t) = H(D) - p \cdot H(D_{\geq t}) - (1 - p) \cdot H(D_{<t})$$

   onde $H(D)$ é a entropia do conjunto total e p é a proporção de dados em $D_{\geq t}$.

4. **Resultado**

   A função devolve o limiar t que gera o maior ganho de informação, juntamente com o valor desse ganho.

4.3 Método `id3_discrete`

Esta função calcula o ganho de informação para atributos **discretos**, ou seja, com um número finito de categorias.

```
def id3_discrete(self, data, attribute):
    """
    Calculate the information gain for a discrete attribute.
    - data: Training data.
    - attribute: The attribute to evaluate.
    """
    idx = self.attributes.index(attribute)
    base_entropy = self.entropy([row[-1] for row in data])  # Entropy
of the entire dataset
    values = set(row[idx] for row in data)  # Unique values of the
attribute

    remainder = 0
    for val in values:
        # Subset of data where the attribute equals the current value
        subset = [row for row in data if row[idx] == val]
        remainder += (len(subset) / len(data)) * self.entropy([row[-1]
for row in subset])

    return base_entropy - remainder, None
```

Explicação técnica

Neste caso, o conjunto de dados é particionado em subconjuntos distintos consoante os valores únicos do atributo.

O ganho de informação é calculado da seguinte forma:

- Entropia inicial: $H(D)$
- Resto (*remainder*):

$$\text{Remainder}(A) = \sum_{v \in \text{Values}(A)} \frac{|D_v|}{|D|} \cdot H(D_v)$$

  onde D_v representa o subconjunto de dados para os quais o atributo $A = v$.

- Finalmente, o ganho é:

$$\text{Gain}(A) = H(D) - \text{Remainder}(A)$$

Como não há limiar em atributos discretos, o segundo valor devolvido pela função é `None`.

4.4 Método `build_rules`

Uma das grandes vantagens da utilização de Decision Trees é a sua **capacidade de explicação**. A função `build_rules` tem como objetivo transformar a árvore gerada pelo algoritmo ID3 numa **lista de regras legíveis**, onde cada regra corresponde a um caminho da raiz até uma folha, com as respetivas condições e classificação final.

```
def build_rules(self, tree=None, premises=None):
    """
```

```
        Build a list of rules from the decision tree.
        - tree: The decision tree (default is the trained tree).
        - premises: List of premises leading to the current node.
        """
        tree = self.tree if tree is None else tree
        premises = premises or []
        rules = []

        for node, branches in tree.items():
            for value, subtree in branches.items():
                # Add the current condition to the premises
                new_premise = premises + [(node.attribute, value,
node.threshold) if node.threshold is not None else (node.attribute,
'=', value)]
                if isinstance(subtree, dict):
                    # Recursively build rules for subtrees
                    rules.extend(self.build_rules(subtree, new_premise))
                else:
                    # Create a rule for a leaf node
                    rules.append(Rule(self.attributes, new_premise,
subtree))
        return rules
```

Explicação Técnica

Parâmetros:

- `tree` : Subárvore atual. Caso não seja fornecida, utiliza-se a árvore completa treinada (`self.tree`).
- `premises` : Lista de condições acumuladas ao longo do caminho da raiz até ao nó atual. Cada condição é armazenada como uma tupla.

Objetivo:

Gerar uma lista de objetos da classe `Rule` , onde cada objeto representa uma regra da forma:

> **SE** (atributo1 = valor1) **E** (atributo2 ≥ threshold2) **ENTÃO** classe = X

Processo Recursivo

1. **Iterar sobre os nós da árvore**: A árvore é representada como um dicionário onde cada chave é um `Node` e os valores são os ramos descendentes desse nó.

2. **Construção de condições (premises)**: Cada nó adiciona uma nova condição à lista de `premises` . A condição é construída de forma diferente consoante se trata de um atributo contínuo ou discreto.

    - Para contínuos:

      `(atributo, operador, threshold)`
      onde `operador` será `'<'` ou `'>='` consoante o ramo.

- Para discretos:

```
(atributo, '=', valor)
```

3. **Verificação do tipo de ramo**:

   - Se o ramo ainda for um dicionário (`dict`), significa que há mais subdivisões, e a função é chamada recursivamente.
   - Se for um valor (rótulo), significa que foi alcançada uma **folha**, e uma nova regra é criada com o conjunto atual de premissas.

Exemplo de Regra Gerada

Suponhamos que a árvore contenha os seguintes ramos:

- `Node(attribute='coluna_1', threshold=0.5)` :
  - Ramo `>=` : vai para `Node(attribute='coluna_3', threshold=None)`
    - Ramo `'=': 1` → classe `Jogador_1`
    - Ramo `'=': 0` → classe `Jogador_2`

Neste caso, as regras geradas seriam algo do género:

- SE `coluna_1 ≥ 0.5` E `coluna_3 = 1` → `Classe = Jogador_1`
- SE `coluna_1 ≥ 0.5` E `coluna_3 = 0` → `Classe = Jogador_2`

4.5 Conclusão - Algoritmo ID3

A aplicação do algoritmo **ID3** no contexto do jogo *4 em linha* permitiu uma primeira aproximação à criação de um modelo supervisionado com base em regras explícitas. Apesar da sua simplicidade, o ID3 revelou-se pouco eficaz quando aplicado diretamente sobre o **dataset derivado de estados de jogo simulados por MCTS**.

4.5.1 Resultados e Observações com a primeira abordagem

Durante os testes realizados na primeira abordagem do dataset, o modelo alcançou uma **accuracy entre 40% a 60% em dados de teste**, enquanto apresentava **valores superiores a 100% nos dados de treino**. Este comportamento revela a ocorrência de dois problemas fundamentais:

- **Overfitting (Alta Variância)**: O modelo ajusta-se altamente aos dados de treino, perdendo a capacidade de generalizar para novas situações.
- **Bias estrutural**: A simplicidade do ID3 impossibilita uma estratégia mais complexa necessária para uma boa árvore de decisão no jogo de 4 em linha, especialmente em cenários com múltiplas interdependências entre jogadas.

> Nota: Devido ao progresso do projeto, a alteração do formato do dataset modificou as respetivas métricas, as quais acabam por revelar resultados bastantes diferentes a estes aqui referidos. Não obstante, para seguir a

> linha temporal do nosso projeto é importante referir os nossos motivos na implementação das novas estratégias.

4.5.2 Considerações sobre a Natureza do Problema

Estes resultados eram **esperados**, dado que:

- O jogo *4 em linha* é altamente estratégico e **não-linear**, com muitas combinações possíveis de jogadas dependentes do contexto.
- O ID3 **não tem memória nem profundidade estratégica**, baseando-se apenas em partições de dados locais, sem considerar consequências futuras.

4.5.3 Caminhos Futuros

De forma a **ultrapassar as limitações** observadas com o ID3, foram exploradas outras abordagens mais robustas e adequadas à natureza do problema:

- **Bagging (Bootstrap Aggregation)**
- **RuleSet Generalization**

5. Ruleset (ID3 Tree com Pruning)

5.1 Introdução Teórica

Um **Ruleset** é uma representação de um modelo de decisão na forma de um conjunto explícito de **regras if-then**, derivadas geralmente de modelos base como **Decision Trees**.

Vantagens desta Implementação

- Melhor capacidade de **generalização** por redução do overfitting ao excluir ramos com muito baixo ganho
- **Redução** da largura e profundidade da Decision Tree gerada
- **Melhoria** da perfomance de classificação

Problemas do Rulesets Implementado

- **Custo computacional elevado** - necessário iterar por todas as regras geradas e aferir se o pruning dessa regra afeta, ou não, o ganho de informação da árvore;
- **Complexidade Temporal elevada** - quanto maior a árvore gerada, mais tempo demorará a sua poda;

5.2 Pruning (Poda)

A técnica de **poda** consiste em **remover condições que não gerem ganhos de informação** das regras extraídas, com o objetivo de **simplificar o Ruleset** não

compromentedo - e geralmente melhorando - a sua performance. Para isso, o conjunto de dados de treino é divido em train/validation, com ratio 0.67/0.33

A poda atua como um mecanismo de **regularização**, reduzindo a complexidade do modelo e prevenindo o overfitting. Os cortes e a simplificação da árvore de decisão é feita através da seguinte lógica:

```
for rule in self.rules:
    for _ in range(len(rule.premises)):
        acc_before = rule.accuracy(self.prune_data)
        removed = rule.premises.pop()  # Try removing the last premise
        if acc_before > rule.get_accuracy(self.prune_data):
            rule.premises.append(removed)  # Restore if accuracy drops
            break
```

Explicação da Implementação:

- Precorre-se as todas as regras (ramos) da árvore de decisão que queremos podar;
- Precorre-se todas as premissas (nós) de uma dada regra;
- Calcula-se a accuracy das previsões e, posteriormente, remove-se essa premissa e calcula-se a accuracy pós-remoção;
- Se a accuracy antes da remoção for maior do que depois da remoção, volta-se a adcionar a premissa (a premissa tem elevado ganho de informação para a classificação em validação);
- Se a accuracy antes da remoção não for maior, salta-se para a próxima regra (as premissas subsequentes não garantiam ganhos de informação);

Esta abordagem procura um equilíbrio entre **simplicidade** e **eficácia**, e será detalhada nos tópicos seguintes com o respetivo código de implementação e análise dos resultados obtidos.

6. Bagging (Bootstrap Aggregation)

6.1 Introdução Teórica

O **Bagging** (*Bootstrap Aggregation*) é uma técnica de aprendizagem em conjunto (**ensemble learning**) cujo principal objetivo é **reduzir a variância** de modelos de machine learning instáveis, como as Decision Trees.

A ideia central do Bagging consiste em:

1. **Gerar múltiplos subconjuntos de dados** a partir do conjunto de treino original, usando amostragem com reposição (bootstrap).
2. **Treinar modelos independentes** sobre cada um desses subconjuntos.
3. **Combinar os resultados** das classificações dos modelos, por maioria de voto.

Esta abordagem promove a mitigação do problema de **overfitting** típico, em modelos altamente sensíveis aos dados, como é o caso do ID3, que ao agregar vários modelos

individualmente imperfeitos, se complementam mutuamente.

6.2 Justificação da Aplicação no Projeto

Como demonstrado no capítulo anterior, a aplicação direta do algoritmo **ID3** ao problema do jogo *4 em linha* resultou numa performance limitada, com clara evidência de **alta variância** - accuracy de treino elevada, mas fraca generalização nos dados de teste.

Assim, a técnica de Bagging surge como uma **tentativa natural de aumentar a robustez do modelo** sem alterar o classificador base. Através da combinação de várias árvores ID3 treinadas em subconjuntos diferentes do dataset, esperamos atingir uma **melhoria significativa da performance**, reduzindo a variância sem comprometer em demasia o viés.

7. Resultados

Seguem, abaixo, os resultados retirados dos processos de treino e teste para cada implementação, os quais já foram referidos no relatório. Este resultados já derivam da implementação final da criação do dataset:

Estatísticas de treino:

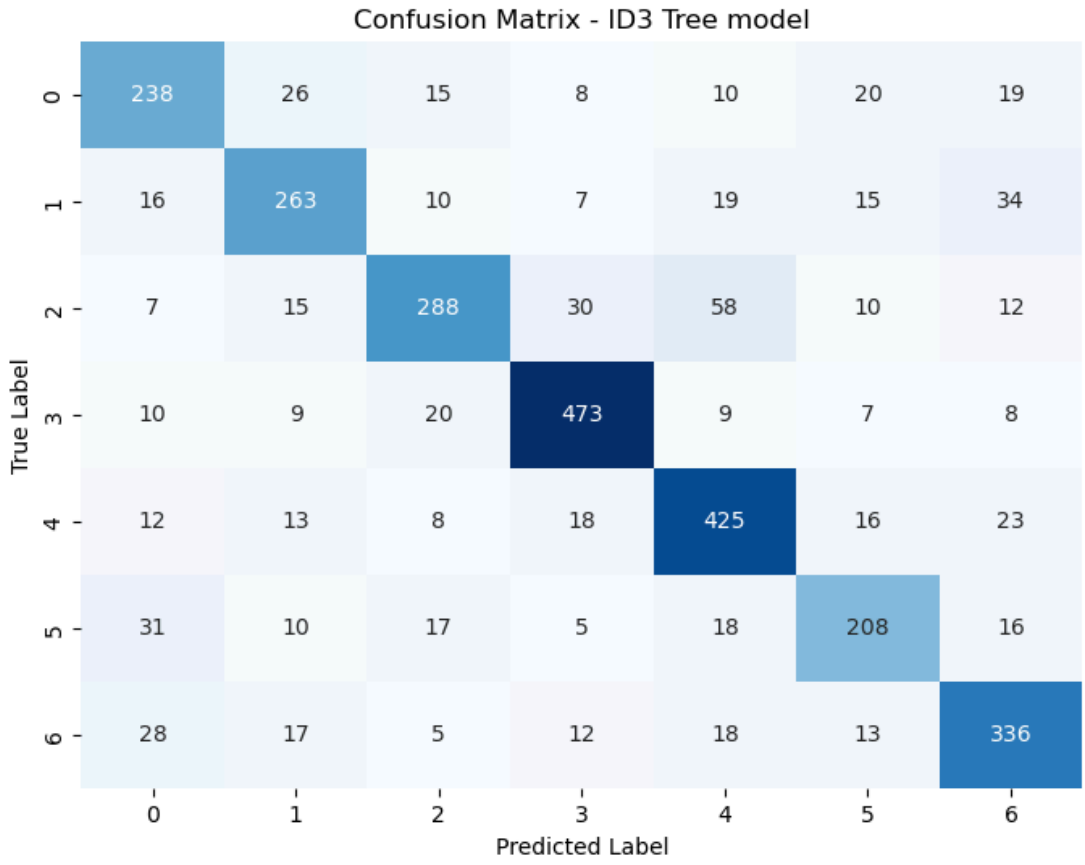
Train metrics for ID3 Tree model:
*__Accuracy__:0.9092, __Precision__: 0.9098, __Recall__: 0.9092, __F1 Score:__ 0.9086

Train metrics for Ruleset model:
__Accuracy__: 0.8223, __Precision__: 0.8239, __Recall__: 0.8223, __F1 Score__: 0.8219

Train metrics for Bagging model:
__Accuracy__: 0.8112, __Precision__: 0.8214, __Recall__: 0.8112, __F1 Score__: 0.8109*

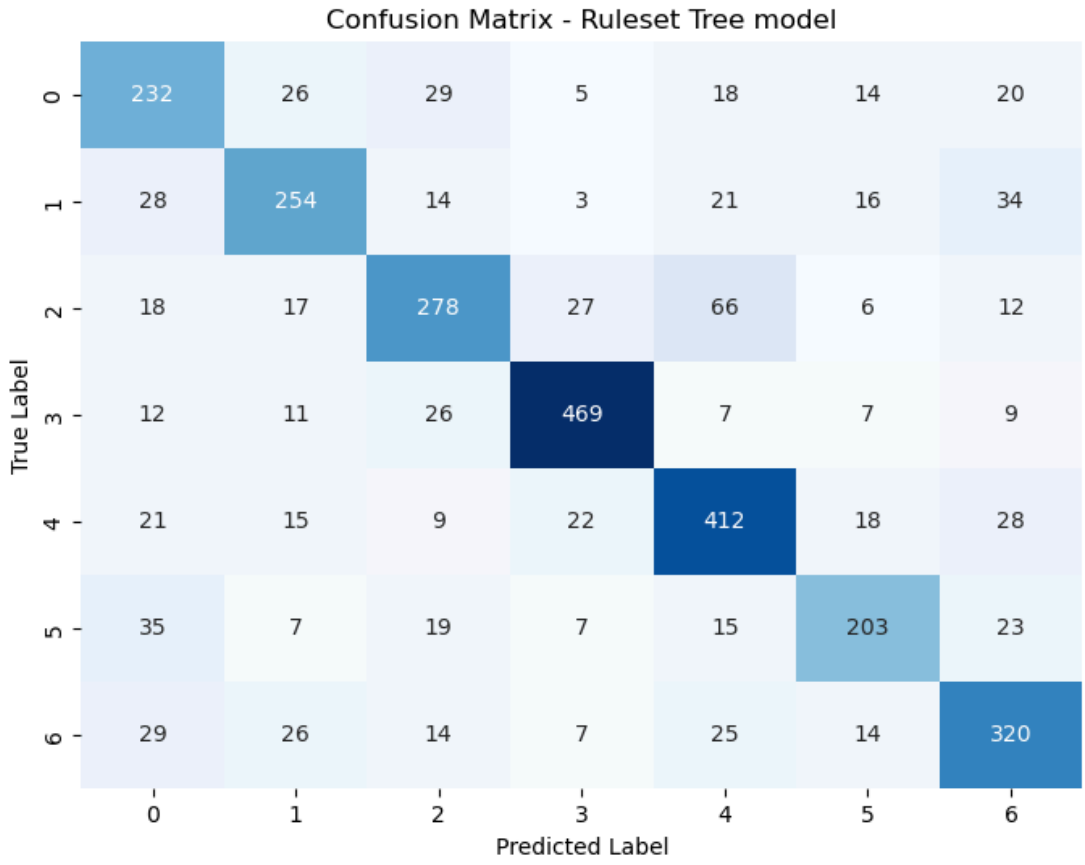
Estatísticas de teste:

Test metrics for ID3 Tree model:
*__Accuracy__: 0.7680, __Precision__: 0.7680, __Recall__: 0.7680, __F1 Score:__ 0.7670

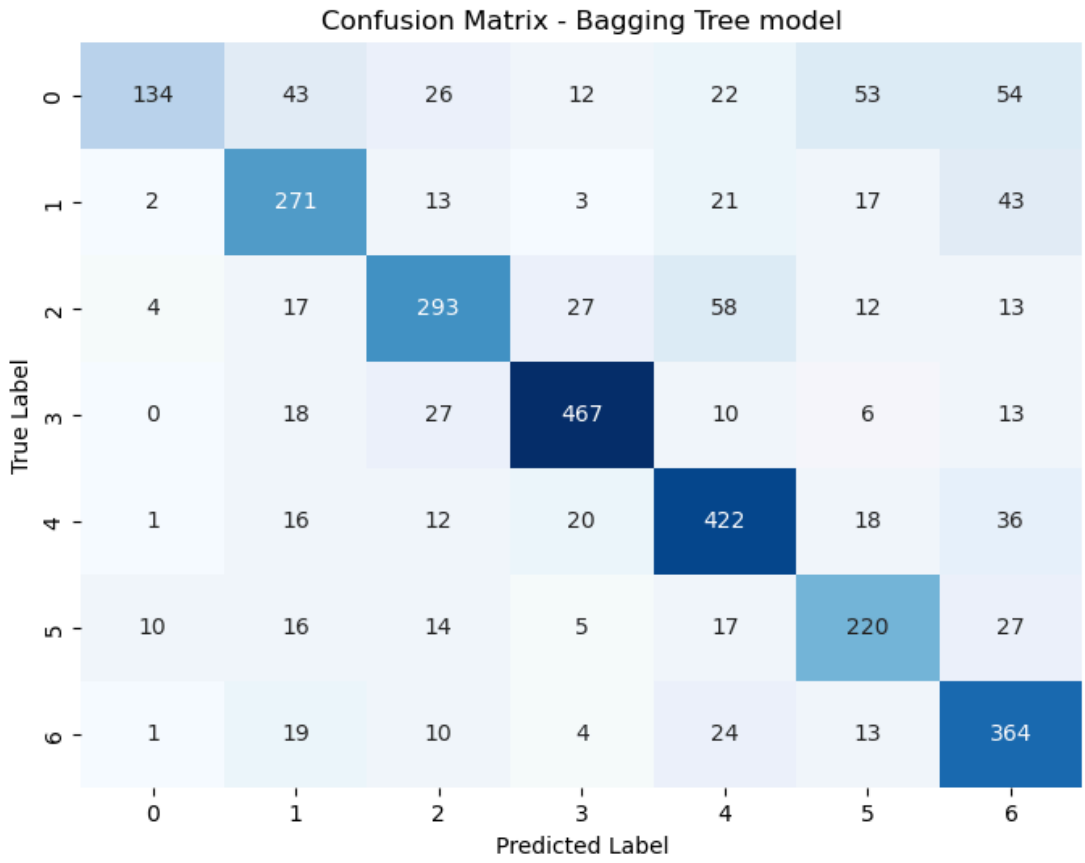


Test metrics for Ruleset model:

Accuracy: 0.7354, **Precision**: 0.7364, **Recall**: 0.7354, **F1 Score:** 0.7352



Test metrics for Bagging model:
Accuracy: 0.7364, **Precision**: 0.7505, **Recall**: 0.7364, **F1 Score:** 0.7302*



(output retirado do notebook analize_decision_tree.ipynb)

Feature importance

Na tentativa de compreender melhor o comportamento dos modelos de Decision Tree implementados e, consequentemente, compreender melhor as decisões do MCTS, estudou-se a `feature importance` dos modelos de Decision Tree.
Obtiveram-se os seguintes resultados:

Feature Importance for ID3 model:

Feature	Score
pieces	0.30871352792495566
cel40	0.021652094021574052
cel31	0.020484247929462285
cel11	0.014016171468572088
cel18	0.013098982612913476

Cells Rank Importance:

37	36	20	8	16	41	30
33	25	23	3	15	39	31
28	26	17	4	5	38	19
35	22	18	13	21	32	27
29	9	2	40	7	34	10
12	11	6	42	1	24	14

Column Importances:

- **Column 5:** 0.0736
- **Column 3:** 0.0670
- **Column 4:** 0.0495
- **Column 2:** 0.0473
- **Column 7:** 0.0472
- **Column 1:** 0.0371
- **Column 6:** 0.0240

Row Importances:

- **Row 6:** 0.0721
- **Row 5:** 0.0663
- **Row 3:** 0.0603
- **Row 4:** 0.0527
- **Row 2:** 0.0513
- **Row 1:** 0.0430

Feature Importance for Ruleset model:

Feature	Score
pieces	0.27306077053768113
cel19	0.028941029940628915
cel26	0.02710772224424464
cel3	0.026707337101747504
cel11	0.02051024195575904

Cells Rank Importance:

22	39	3	6	13	41	38
20	23	17	4	7	33	25
31	32	9	24	1	28	12
35	27	15	14	2	40	30
34	26	10	42	19	21	8
16	11	37	36	5	29	18

Column Importances:

- **Column 5:** 0.1045
- **Column 3:** 0.0687
- **Column 4:** 0.0544
- **Column 7:** 0.0443
- **Column 2:** 0.0341
- **Column 1:** 0.0328
- **Column 6:** 0.0246

Row Importances:

- **Row 3:** 0.0715
- **Row 1:** 0.0642
- **Row 2:** 0.0636
- **Row 4:** 0.0599
- **Row 6:** 0.0560
- **Row 5:** 0.0482

Feature Importance for Bagging model:

Feature	Score
pieces	0.25732551076858157
cel3	0.04390351036360313
cel11	0.019909721360264344
cel40	0.01385043367470451
cel21	0.013740461486513198

Cells Rank Importance:

27	36	1	7	23	40	21
35	22	26	2	13	37	15
16	34	18	25	5	39	4
32	31	14	28	8	38	30
33	10	12	41	11	29	20
17	6	19	42	3	24	9

Column Importances:

- **Column 3:** 0.0875
- **Column 5:** 0.0684
- **Column 7:** 0.0561
- **Column 4:** 0.0484
- **Column 2:** 0.0476
- **Column 1:** 0.0394

- **Column 6:** 0.0238

Row Importances:

- **Row 1:** 0.0852
- **Row 6:** 0.0641
- **Row 2:** 0.0614
- **Row 3:** 0.0587
- **Row 5:** 0.0544
- **Row 4:** 0.0475

(outputs retirados do notebook DT_feature_importance.ipynb)

8. Validação da Implementação da Árvore de Decisão com o Dataset Iris

8.1 O que é o Dataset Iris?

O **Iris Dataset** é um dos conjuntos de dados mais utilizados para tarefas de **classificação supervisionada**. Foi introduzido por Ronald Fisher em 1936 e contém **150 amostras** de flores da espécie *Iris*, distribuídas por três classes:

- *Iris setosa*
- *Iris versicolor*
- *Iris virginica*

Cada amostra é caracterizada por **quatro atributos contínuos**:

- Comprimento da sépala
- Largura da sépala
- Comprimento da pétala
- Largura da pétala

O objetivo é prever a **classe da flor** com base nestes atributos. Dada a sua estrutura simples mas não trivial, o Iris é frequentemente utilizado para **validar implementações de algoritmos de machine learning**.

8.2 Resultados Obtidos (Iris)

Para validar a implementação da árvore de decisão construída com base no algoritmo **ID3**, foram realizados testes no dataset Iris com os seguintes métodos:

1. ID3Tree

```
plaintext
Accuracy: 0.91
```

2. Ruleset

```
plaintext
Accuracy: 0.96
```

3. Bagging

```
plaintext
Training classifier #1
Training classifier #2
Training classifier #3
Training classifier #4
Training classifier #5
Training classifier #6
Training classifier #7
Training classifier #8
Training classifier #9
Training classifier #10
Accuracy: 1.00
```

(outputs retirados do notebook test_DT_iris_dataset.ipynb)

8.3 Conclusão (Iris)

Os resultados obtidos com o dataset Iris confirmam a **correta implementação** da DT. A performance alcançada está alinhada com a esperada para este tipo de modelo e conjunto de dados. Portanto, pode-se concluir com segurança que a implementação desenvolvida está **funcional**, sendo adequada para aplicação noutros problemas, incluindo o jogo *4 em linha*.

9. Problemas e *Misconceptions*

Durante o desenvolvimento e análise do projeto, surgiram algumas questões frequentes e conceções erradas em torno do comportamento do agente de Monte Carlo e da definição de iterações. Esta secção procura clarificar esses pontos.

9.1 O agente fica mais inteligente ao longo do jogo?

Não... e Sim.

Existe uma ideia comum de que o agente de Monte Carlo parece "ficar mais inteligente" à medida que o jogo avança. No entanto, essa perceção não está relacionada com uma melhoria efetiva da IA, mas sim com a **natureza do próprio jogo** e da informação disponível em cada jogada.

Porquê?

- O número de **iterações** do algoritmo Monte Carlo é fixo (ex: 500, 5000, 10000), dependendo da dificuldade escolhida.
- Nas **primeiras jogadas**, o número de possibilidades é muito elevado. Um agente com poucas iterações (ex: dificuldade "fácil") tem pouca capacidade de explorar esse espaço eficientemente, ao contrário de um agente "difícil", que consegue analisar muito mais cenários devido ao maior número de simulações.
- No **final do jogo**, o número de possibilidades é naturalmente muito mais reduzido. Mesmo com poucas iterações, o agente consegue cobrir uma porção significativa (ou total) do espaço de jogo e, por isso, **tomar decisões ótimas**, semelhantes às de um agente difícil.

Conclusão:

O agente **não evolui** ao longo do jogo - ele mantém a mesma "inteligência" (número de simulações permitidas). O que muda é que o jogo se torna **exponencialmente mais simples**, e até um agente com baixa capacidade de exploração consegue tomar decisões quase perfeitas nas últimas jogadas.

9.2 O que são iterações?

Outro ponto de confusão comum é o conceito de **iterações** no contexto do algoritmo **Monte Carlo Tree Search (MCTS)**.

Possíveis interpretações:

- Cada **nó explorado** na árvore de pesquisa.
- Cada **simulação completa de jogo** (do estado atual até ao fim).

Neste projeto:

As **iterações** são consideradas como **simulações completas de jogos** a partir do estado atual do tabuleiro.
Ou seja, uma iteração corresponde a:

> "Simular um jogo completo aleatório até ao fim, a partir da jogada candidata, e usar o resultado para avaliar essa jogada."

Esta abordagem é consistente com uma visão mais prática e direta do MCTS e permite associar facilmente a dificuldade do agente ao número de jogos simulados.

Implicações destes pontos

- É importante não interpretar uma melhoria de desempenho do agente no fim do jogo como aumento de inteligência.
- A escolha do número de iterações deve considerar que os ganhos em jogadas iniciais são maiores, pois é onde as decisões são mais difíceis.

10. Conclusões Finais

10.1 Monte Carlo Tree Search (MCTS)

Ao fim deste trabalho, conclui-se que:

- O algoritmo de **Monte Carlo Tree Search** implementado revelou-se ***muito eficiente*** na pesquisa do melhor lance para uma dada posição, **necessitando de pouco tempo por decisão para vencer um humano**.
- O **MCTS em singlethread apresenta melhor performance em *baixas iterações***, enquanto o **MCTS em multithread tem melhor desempenho com *altas iterações***.
    - Isto deve-se ao facto de que, com poucas iterações, a **complexidade temporal da própria pesquisa** do MCTS é **menor** do que a **complexidade de junção das árvores geradas por cada core**, favorecendo a versão em singlethread.
    - Já quando o número de iterações ultrapassa as **5000**, a **complexidade da pesquisa torna-se dominante**, tornando a versão multithread **mais eficiente**.
- No caso do MCTS em multithread, ao **dividir o número total de iterações pelo número de cores do CPU**, surgiram dúvidas quanto à **equidade da comparação entre as versões singlethread e multithread**, dado que a árvore de decisão multithread seria menos iterada por core.
    - Esta hipótese foi **confirmada**: observou-se que **aproximadamente 30% das decisões tomadas para a mesma posição diferem entre a versão singlethread e a multithread**.

10.2 **Decision Tree nos Datasets Iris e AI vs AI**

Ao longo desta análise, verificou-se que:

- Tanto a implementação da Decision Tree ID3, quanto a Decision Tree com Prunning (Ruleset) e o ensemble por Bootstrap Aggregating (Bagging) apresentaram **accuracy superior a 90% no dataset Iris**.
- Embora o modelo Bagging tenha demorado mais tempo em treino, foi o que alcançou a **melhor performance, atingindo 100% de accuracy no iris dataset**.
- No dataset gerado pelos jogos de AI vs AI, observou-se uma correlação direta entre a quantidade de exemplos de treino e a performance dos modelos, devido ao comportamento do algoritmo MCTS, que prefere jogar nas colunas centrais e inicia as partidas com as mesmas aberturas que gera posições repetidas enviesando os resultados da métricas artificialmente.
- Este comportamento gerou um défice de dados nas colunas extremas, refletindo um menor número de exemplos de treino nessas regiões, o que resultou em

overfitting, evidenciado pelo desempenho superior no conjunto de treino em comparação ao conjunto de teste.

- As métricas de desempenho no conjunto de teste ficaram entre **60% e 75%**, indicando uma queda significativa em relação ao desempenho observado no treino.
- Os modelos revelam, portanto uma **bias** (com os lances repetidos) considerável - entre 10% e 25% no treino e entre 25% e 30% no teste.
- Já na **variance** o ID3 revela maior diferença entre os splits - cerca de 25% - enquanto o bagging e o ruleset apresentam uma melhor performance - cerca de 10% - o que indica que estas técnicas serviram o seu propósito.