TRABALHO AKINATOR - ÁRVORE DE DECISÃO SIMPLIFICADA

Bernardo De Marco Gonçalves - 22102557

1. INTRODUÇÃO

Esse documento refere-se ao relatório do trabalho Akinator da disciplina Estrutura de Dados (INE 5609), realizado pelo aluno Bernardo Gonçalves. O tema definido para o jogo foi times de futebol.

2. ESTRUTURA DO PROGRAMA

Para fins de organização, o programa foi estruturado com base no padrão Model-View-Controller (MVC). Segue uma breve descrição do que cada arquivo contém:

- **“View.py”:** implementação da classe View, assumindo o papel de tela no padrão MVC.
- **“Node.py” e “DecisionTree.py”:** implementação das classes Node e DecisionTree, respectivamente. Elas são os modelos do sistema.
- **“Controller.py”**: implementação da classe Controller, responsável por ser o controlador do sistema.
- **“main.py”:** arquivo que instancia o controlador e inicia o jogo.
- **“enums.py” e “tree-data.pkl”:** O primeiro contém duas enumerações que são úteis na inserção de elementos. O segundo é o arquivo que contém os dados persistidos. Inicialmente ele já contém uma árvore definida para fins de exemplificação. Caso queira, o arquivo pode ser excluído para iniciar um jogo do zero. Ao finalizar o sistema o arquivo será automaticamente recriado com os dados do jogo recém finalizado.

3. DOCUMENTAÇÃO DOS MÉTODOS

3.1. CLASSE VIEW

A classe View não possui nenhum atributo, apenas métodos. Segue a assinatura dos métodos e suas respectivas descrições.

3.1.1. `def read_answer(self, message: str = '') -> str:`

Retorna entrada do usuário. Aceita um parâmetro opcional, “message”, que é mostrado no terminal ao pedir a entrada. Ela é tratada caso seja uma string vazia, e o usuário é devidamente notificado.

3.1.2. `def show_options(self) -> int:`

Pede ao usuário qual opção ele deseja escolher e retorna o número inteiro correspondente (0 ou 1). Existem duas disponíveis: encerrar o programa (0) e jogar (1). Entradas que sejam diferentes dessas duas opções são devidamente tratadas.

3.1.3. `def yes_or_no_question(self, question: str) -> str:`

Faz uma pergunta ao usuário que tenha como resposta “s” (sim) ou “n” (não) e retorna a resposta. Aceita como parâmetro “question” (string), que é a pergunta que será realizada.

3.1.4. `def guess_answer(self, team_name: str) -> str:`

Pede ao usuário se ele pensou em um dado time e retorna a resposta. Aceita como parâmetro “team_name” (string), que é o time que será pedido.

3.1.5. `def ask_question(self, question: str) -> str:`

Faz uma pergunta ao usuário e retorna a resposta. Aceita como parâmetro “question” (string), que é a pergunta a ser feita.

3.1.6. `def ask_difference(self, team_name: str) -> tuple:`

Pede qual time em que o usuário pensou e as diferenças entre o “team_name” e o time pensado. Aceita como parâmetro “team_name” (string). Retorna uma tupla, na qual o primeiro item é o time pensado pelo usuário e o segundo é a diferença apontada por ele.

3.1.6. `def show_message(self, message: str) -> None:`

Mostra uma mensagem no terminal. Aceita como parâmetro “message” (string), que é a mensagem a ser mostrada. Não tem retorno.

3.2. CLASSE NODE

A classe Node representa cada nó da árvore binária. Cada nó possui um valor (string), que pode ser uma pergunta ou resposta, um ponteiro “yes” que aponta para um nó quando a resposta for sim e um ponteiro “no” que aponta para um nó quando a resposta for não. Caso o nó seja uma resposta, “yes” e “no” serão nulos. Já se ele for uma pergunta, obrigatoriamente os atributos apontarão para algum outro nó. Segue a assinatura dos métodos e suas respectivas descrições.

3.2.1. `def __init__(self, value: str) -> None:`

Construtor da classe Node. Aceita como parâmetro “value” (string). Instancia um objeto Node, com o atributo “value” inicializado com o valor do parâmetro e os atributos “yes” e “no” inicialmente nulos.

3.2.2.

```
    @staticmethod
    def build_question(question: str, correct_answer: str, wrong_answer:
str):
        new_node = Node(question)
        new_node.yes = Node(correct_answer)
        new_node.no = Node(wrong_answer)
        return new_node
```

Método estático que instancia e retorna um nó representando uma pergunta. Aceita como parâmetro “question” (string), “correct_answer” (string), e “wrong_answer” (string).

3.2.3. `def is_leaf(self) -> bool:`

Retorna um booleano indicando se o nó é uma folha (resposta) ou não.

3.3. CLASSE DECISION TREE

Classe que representa a árvore binária. Cada árvore possui um atributo “root”, que é um objeto Node representando a raiz dela. Segue a assinatura dos métodos e suas respectivas descrições.

3.3.1. `def __init__(self, initial_team: str) -> None:`

Construtor da classe DecisionTree. Aceita como parâmetro “initial_team” (string). Instancia objeto, com o atributo root sendo uma instância da classe Node, representando o time inicial.

3.3.2.

```
def insert(self, new_node, parent, direction):
        if direction is None:
            self.__root = new_node
            return

        if direction is Direction.CORRECT_ANSWER:
            parent.yes = new_node
        elif direction is Direction.WRONG_ANSWER:
            parent.no = new_node
```

Insere nó na árvore. Aceita como parâmetro “new_node” (Node), representando uma pergunta a ser inserida, “parent” (Node ou nulo) representando o

nó pai no qual o novo nó será inserido e “direction” (“CORRECT_ANSWER”, “WRONG_ANSWER” ou nulo).

Se “direction” for nulo, a raiz recebe o novo nó. Caso contrário, se “direction” for “CORRECT_ANSWER”, então o atributo “yes” do “parent” aponta para novo nó. Se não, se “direction” for “WRONG_ANSWER”, então o atributo “no” do “parent” aponta para o novo nó.

3.3.3. `def __pre_order(self, root):` e `def print_pre_order(self):`

O primeiro é um método privado que realiza a caminhada “pre order” na árvore recursivamente. O segundo apenas chama o primeiro, passando como parâmetro a raiz da árvore. Eles foram implementados para, exclusivamente, checar se a árvore estava sendo implementada corretamente.

3.4. CLASSE CONTROLLER

Classe que representa o controlador do sistema. Ele tem como atributo uma “view” (View) e uma “tree” (DecisionTree). Segue a assinatura dos métodos e suas respectivas descrições.

3.4.1.

```
    def __init__(self) -> None:
            self.__view = View()
            self.__tree = None
            try:
                self.__tree = self.__load()
            except FileNotFoundError:
                self.__tree = DecisionTree('Chapecoense')
```

Construtor da classe Controller. Cria um objeto da classe View e define o atributo “view” para recebê-lo. Também, tenta executar o método load, que carrega e retorna os dados do arquivo “tree-data.pkl”. Porém se esse arquivo não existe uma exceção do tipo “FileNotFoundError” é levantada. Assim, ela é devidamente tratada, criando um objeto da classe DecisionTree e definindo o atributo “tree” para recebê-lo.

3.4.2.

```
        def __load(self):
            with open(FILENAME, 'rb') as f:
                return pickle.load(f)

        def __dump(self):
```

```
            with open(FILENAME, 'wb') as f:
                pickle.dump(self.__tree, f)
```

Ambos métodos privados estão relacionados com a persistência dos dados. O primeiro, abre o arquivo “tree-data.pkl” em modo de leitura binária e retorna seu conteúdo. O segundo, abre o arquivo no modo de escrita binária e serializa o atributo “tree” no arquivo. Esse processo de persistência foi realizado utilizando o módulo pickle.

3.4.3.

```
        def run(self):
            switcher = {
                0: self.end_game,
                1: self.play_game,
            }
            while True:
                switcher[self.__view.show_options()]()
```

Método executado pelo arquivo “main.py” logo após a criação de um objeto da classe Controller. Executa o método “show_options” do atributo “view”. Se o retorno desse método for o inteiro 0, então é executado o método “end_game” do controlador. Se não, se for retornado 1, então é executado o método “play_game”.

3.4.4.

```
        def end_game(self):
            answer = self.__view.yes_or_no_question(
                'Jogo finalizado! Deseja recomeçar? (s/n)\n')
            if answer == Answer.YES.value:
                self.play_game()
            elif answer == Answer.NO.value:
                self.__dump()
                exit(0)
```

Método utilizado para encerrar o programa. Primeiro pede ao usuário se deseja recomeçar a partida. Se a resposta for sim, então o método “play_game” é executado. Se for não, então o método “dump” é executado, persistindo os dados da árvore, e é executada a função interna do Python “exit” que encerra a execução do programa.

3.4.5.

```
    def play_game(self):
        self.__view.show_message('Pense em um time de futebol!')

        iterator = self.__tree.root
```

```
            parent = None
            direction = None

            while not iterator.is_leaf():
                answer = self.__view.ask_question(iterator.value)
                parent = iterator
                if answer == Answer.YES.value:
                    direction = Direction.CORRECT_ANSWER
                    iterator = iterator.yes
                elif answer == Answer.NO.value:
                    direction = Direction.WRONG_ANSWER
                    iterator = iterator.no

            answer = self.__view.guess_answer(iterator.value)
            if answer == Answer.YES.value:
                self.end_game()
            elif answer == Answer.NO.value
                (new_team, new_question) = self.__view.ask_difference(
                                                            iterator.value)

                new_question_node = Node.build_question(
                    new_question, iterator.value, new_team)

                self.__tree.insert(new_question_node, parent, direction)
```

Método percorre a árvore fazendo as perguntas até chegar em uma folha (resposta).

Primeiramente, são inicializadas três variáveis auxiliares: “iterator”, “parent” e “direction”. Então, é criado um loop que é executado enquanto o iterador não é uma folha. Assim, dentro da execução dele, sabe-se que o iterador sempre aponta para uma pergunta. Ela, então, é feita ao usuário, e a variável “parent” aponta para o iterador (abordagem semelhante ao pé atrás da lista encadeada). Se a resposta for sim, então a direção recebe “CORRECT_ANSWER” e o iterador passa a apontar para o atributo “yes” dele. Se for não, a direção recebe “WRONG_ANSWER” e o iterador aponta para o atributo “no” dele.

Portanto, ao sair do loop, o iterador aponta para alguma resposta, o “parent“ para o pai da resposta (uma pergunta) e a direção indica se a resposta é a correta ou errada do seu pai. Com isso, é feita a pergunta ao usuário se ele estava pensando na determinada resposta (time de futebol). Se sim, então é executado o

método “end_game”. Se não, é solicitado ao usuário qual time ele estava pensando e as diferenças entre o time pensado e o que já estava na árvore. Com isso, é executado o método estático da classe Node “build_question”, retornando um nó com a pergunta e suas respectivas respostas. Por fim, esse nó é inserido na árvore.