Exercicio pratico - Agente de resolução de problemas - Parte 03

Alunos: Antônio Carlos Ramos Filho, Fabiano Amaral Alves, Pablo Junio Souza Santos

Formulação do Problema

A representação de um problema consiste em um estado inicial, uma função ao qual se obtém os próximos estados possíveis a partir do estado atual e a definição de um estado objetivo em termos de um método Booleno que retornará `True` caso o estado atual seja o objetivo.

Com base na definição acima, resolvemos por implementar a definição do problema das cidades da forma mais simples possível.

Começamos por definir uma nova classe `Problema_Cidades` ao qual possui os atributos de cidade inicial `self.inicio` e a cidade objetivo `self.objetivo`, também possui métodos abstratos que serão sobrescritos na classe que herdará de `Problema_cidades`.

```
# Arquivo: problema.py

class Problema_Cidades(object):
    '''
    O problema de encontrar um caminho mais curto de uma cidade a outra.
    '''

    def __init__(self, c_inicio, c_objetivo):
        '''
        Define a cidade de partida e a cidade de destino.
        '''
        self.inicio = c_inicio
        self.objetivo = c_objetivo

    def acoes(self, estado):
        '''
        As ações em uma cidade corrente, é justamente as cidades vizinhas
        '''
        raise NotImplementedError

    def verifica_objetivo(self, estado_atual):
        '''
        Se a cidade atual for justamente a mesma em self.objetivo, então
        o agente chegou em seu destino
        '''
        raise NotImplementedError
```

> Note que métodos como `custo_do_caminho` e `resultado` não foram implementados pois na classe `Espaco_de_Estados` existe métodos com a mesma função e o resultado de uma ação nesse problema é justamente está na cidade vizinha.

Como já havíamos implementado algumas funções bastante úteis para formulação do problema na classe `Espaco_de_Estados`, optamos por modifica-la e torna-la filha da classe `Problema_Cidades`. Com isso, sobrescrevemos os métodos `acoes` e `verifica_objetivo` no corpo da classe `Espaco_de_Estados` aproveitando os métodos já existentes nela para implementar os sobrescritos.

Vejamos agora o que foi alterado na classe `Espaco_de_Estados`:

> Primeiro alteramos a herança da classe `object` para a classe `Problema_Cidades`. Em seguida adicionamos mais dois parâmetros ao construtor para definir quais serão as cidades de inicio e cidade objetivo na formulação do problema.

```
# Arquivo: espaco_estado.py
from problema.problema import Problema_Cidades


class Espaco_de_Estados(Problema_Cidades):

    def __init__(self, inicio, objetivo ,grafo_estados = None):
        '''
        Essa classe tratará o espaço de estados como um dicionário \
        onde receberá um dicionário, se nada for passado ao construtor \
        será criado um dicionário vazio.
        Caso contrario a classe receberá o grafo extraído do arquivo.
        '''
        super.__init__(inicio, objetivo)
        if grafo_estados == None:
            grafo_estados = {}
        self.espaco_de_estados = grafo_estados
```

> Então sobrescrevemos os métodos da classe `Problema_Cidades`.

```
    def verifica_objetivo(self, estado_atual):
        if estado_atual == self.objetivo:
            return True
        return False

    def acoes(self, estado_atual):
        '''
        Retorna os próximos estados acessíveis a partir do estado atual
        '''
        possiveis_estados = []
        for l in self.get_vizinhancas():
            for i, j in l:
                if estado_atual == i:
                    possiveis_estados.append(j)
                if estado_atual == j:
```

```
                    possiveis_estados.append(i)
        return possiveis_estados
```

> Note que em `acoes` utilizamos o método `get_vizinhancas()` que é um método de `Espaco_de_Estados` que entrega uma lista de vizinhos para o método `acoes` que então recupera todos os vizinhos da cidade atual, retornando as em uma lista.

A seguir a classe `Espaco_de_Estados` completa após a modificação.

```
# Arquivo: espaco_estado.py
from problema.problema import Problema_Cidades


class Espaco_de_Estados(Problema_Cidades):

    def __init__(self, inicio, objetivo ,grafo_estados = None):
        '''
        Essa classe tratará o espaço de estados como um dicionário \
        onde receberá um dicionário, se nada for passado ao construtor \
        será criado um dicionário vazio.
        Caso contrario a classe receberá o grafo extraido do arquivo.
        '''
        super.__init__(inicio, objetivo)
        if grafo_estados == None:
            grafo_estados = {}
        self.espaco_de_estados = grafo_estados

    def verifica_objetivo(self, estado_atual):
        if estado_atual == self.objetivo:
            return True
        return False

    def acoes(self, estado_atual):
        '''
        Retorna os próximos estados acessíveis a partir do estado atual
        '''
        possiveis_estados = []
        for l in self.get_vizinhancas():
            for i, j in l:
                if estado_atual == i:
                    possiveis_estados.append(j)
                if estado_atual == j:
                    possiveis_estados.append(i)
        return possiveis_estados

    def get_cidades(self):
        '''
        Retorna a lista de cidades
        '''
        return [self.espaco_de_estados.keys()]
```

```
        # ou return list(self.espaco_de_estados)

    def get_vizinhancas(self):
        '''
        Retorna uma lista de vizinhanças, ou seja, todas as arestas do
grafo
        '''
        return self.monta_vizinhanca()

    def get_vizinhancas_com_peso(self):
        '''
        Retorna uma lista de vizinhanças com as distancias, ou seja, todas
as arestas do grafo + peso
        '''
        return self.monta_vizinhaca_com_peso()

    def add_cidade(self, vertice):
        '''
        Adiciona uma nova cidade ao espaço de estados
        A principio ela fica sem cidades vizinhas
        '''
        if vertice not in self.espaco_de_estados:
            self.espaco_de_estados[vertice] = []

    def add_vizinhanca(self, aresta):
        '''
        Adiciona uma nova aresta de cidades no espaço de estados
        :param aresta: pode ser uma lista, tupla ou conjunto contendo as
cidades e a distancia entre elas
        :return: None
        '''
        aresta = set(aresta)

        (cidade1, cidade2, peso) = tuple(aresta)

        if cidade1 in self.espaco_de_estados:
            self.espaco_de_estados[cidade1].append({cidade2: int(peso)})
        else:
            self.espaco_de_estados[cidade1] = [{cidade2: int(peso)}]

    def monta_vizinhanca(self):
        '''
        Esse método percorre o grafo e cria uma lista com das vizinhanças
entre as cidades
        :return: a lista de vizinhos
        '''
        borda = []

        for cidade, vizinho in self.espaco_de_estados.items():
            for k in vizinho:
                for c, p in k.items():
                    if (c, cidade) not in borda:
                        borda.append((cidade, c))
        return borda
```

```
    def monta_vizinhaca_com_peso(self):
        '''
        cria a lista de vizinhos, mas acrescentando a distancia entre cada
uma
        :return: lista de vizinhos
        '''
        borda = []

        for cidade, vizinho in self.espaco_de_estados.items():
            for k in vizinho:
                for c, p in k.items():
                    if (c, cidade, p) not in borda:
                        borda.append((cidade, c, p))
        return borda

    def encontrar_caminho(self, inicio, fim, caminho=None):
        '''
        Método auxiliar para obter o percurso de um cidade a outra
        :param inicio: cidade inicial
        :param fim: cidade objetivo
        :param caminho: lista com as cidades que foram visitadas até chegar
no destino
        :return: o caminho que foi percorrido
        '''
        if caminho==None:
            caminho = []

        cidades = self.espaco_de_estados
        caminho = caminho+[inicio]
        if inicio == fim:
            return caminho
        if inicio not in cidades:
            return None
        for cidade in cidades[inicio]:
            for c in cidade.keys():
                if c not in caminho:
                    caminho_extendido = self.encontrar_caminho(c, fim,
caminho)
                    if caminho_extendido:
                        return caminho_extendido
        return None

    def encontrar_caminho_com_custo(self, inicio, fim, caminho=None,
peso=0):
        '''
        Método auxiliar para obter o percurso de um cidade a outra e a
distancia percorrida
        :param inicio: cidade de partida
        :param fim: cidade objetivo
        :param caminho: percurso percorrido
        :param peso: distancia percorrida
        :return: o caminho + custo ou none caso falha
        '''
```

```
        if caminho == None:
            caminho = []
        cidades = self.espaco_de_estados
        caminho = caminho + [inicio]
        custo = peso
        if inicio == fim:
            return (caminho, custo)
        if inicio not in cidades:
            return None
        for cidade in cidades[inicio]:
            for c, v in cidade.items():
                if c not in caminho:
                    caminho_extendido = self.encontrar_caminho_com_custo(c,
fim, caminho, custo+v)
                    if caminho_extendido:
                        return caminho_extendido
        return None

    def __repr__(self):
        '''
        representação do objeto como string
        :return: o objeto em formato string
        '''
        return f'{self.espaco_de_estados}'

    def __str__(self):
        '''
        Representação personalizada do objeto
        :return: o objeto como string de forma mais legível
        '''
        string = 'Cidades: '
        for cidades in self.espaco_de_estados:
            string += str(cidades) + ' '
        string += '\nVizinhanças: '
        for arestas in self.monta_visinhanca():
            string += str(arestas) + ' '
        return string
```