Algoritmo de Busca A*

O que é Algoritmo de Busca A*

Encontra o caminho de menor custo entre dois pontos em um gráfico.

Busca gulosa + busca de custo uniforme.

Usado em mapas jogos e robôs

Função de avaliação

g(n): Custo real do início até o nó atual.

h(n): Heurística (estimativa do custo até o objetivo).

f(n): Soma dos dois, escolhendo sempre o menor.

$$f(n)=g(n)+h(n)$$

Função p/ Obter Heurísticas

```
import queue
import matplotlib.pyplot as plt


# Função para obter heurísticas do arquivo
def obter_heuristicas():
    heuristicas = {
    }  # Dicionário para armazenar as heurísticas (distâncias estimadas)
    arquivo = open(
        "heuristicas.txt")

    for linha in arquivo.readlines():
        valores = linha.split(a
        )  # Divide a linha em duas partes: nome da cidade e valor heurístico
        heuristicas[valores[0]] = int(
            valores[1]
        )  # Adiciona ao dicionário: cidade como chave e heurística como valor (convertido para inteiro)
    return heuristicas
```

Função p/ Obter Coordenadas

```
# Função para obter as cidades e suas coordenadas a partir de um arquivo
def obter_cidades():
    cidade = {}
    codigos_cidades = {}  # Dicionário para mapear códigos numéricos às cidades
    arquivo = open(
        "cidades.txt")
    indice = 1

    for linha in arquivo.readlines():
        valores = linha.split(
        )  # Divide a linha em partes: nome da cidade, coordenada x e coordenada y
        cidade[valores[0]] = [
            int(valores[1]), int(valores[2])
        ]  # Adiciona ao dicionário: cidade como chave e coordenadas como valor
        codigos_cidades[indice] = valores[
            0]
        indice += 1
    return cidade, codigos_cidades
```

Função p/ Criar Grafo

```python
# Função para criar o grafo a partir de um arquivo
def criar_grafo():
    grafo = {}
    arquivo = open("grafoCidades.txt"
                   )

    for linha in arquivo.readlines():
        valores = linha.split(
        )  # Divide a linha em partes: cidade1, cidade2 e distância

        # Verifica se ambas as cidades já estão no grafo
        if valores[0] in grafo and valores[1] in grafo:
            conexoes = grafo.get(
                valores[0])  # Obtém as conexões da primeira cidade
            conexoes.append([valores[1], valores[2]
                             ])  # Adiciona a segunda cidade e a distância
            grafo.update({valores[0]: conexoes})

            conexoes = grafo.get(
                valores[1])
            conexoes.append([valores[0], valores[2]
                             ])
            grafo.update({valores[1]: conexoes})
```

Função p/ Criar Grafo

```
    # Verifica se apenas a primeira cidade está no grafo
    elif valores[0] in grafo:
        conexoes = grafo.get(
            valores[0])  # Obtém as conexões da primeira cidade
        conexoes.append([valores[1], valores[2]
                         ])  # Adiciona a segunda cidade e a distância
        grafo.update({valores[0]: conexoes})

        grafo[valores[1]] = [[valores[0], valores[2]]
                             ]  # Adiciona a segunda cidade ao grafo

    # Verifica se apenas a segunda cidade está no grafo
    elif valores[1] in grafo:
        conexoes = grafo.get(
            valores[1])
        conexoes.append([valores[0], valores[2]
                         ])
        grafo.update({valores[1]: conexoes})

        grafo[valores[0]] = [[valores[1], valores[2]]
                             ]

    # Caso nenhuma das cidades esteja no grafo
    else:
        grafo[valores[0]] = [[valores[1], valores[2]]
                             ]  # Adiciona a primeira cidade e sua conexão
        grafo[valores[1]] = [[valores[0], valores[2]]
                             ]  # Adiciona a segunda cidade e sua conexão

return grafo
```

Algoritmo de Busca A*

```python
# Algoritmo A* (Astar)
def algoritmo_a_estrela(cidade_inicio,
                        heuristicas,
                        grafo,
                        cidade_destino="Bucharest"):
    fila_prioridade = queue.PriorityQueue()
    distancia_total = 0  # Variável para armazenar a distância acumulada
    caminho = []  # Lista para armazenar o caminho encontrado

    fila_prioridade.put(
        (heuristicas[cidade_inicio] + distancia_total,
         [cidade_inicio,
          0]))  # Adiciona o nó inicial à fila com sua prioridade
```

Algoritmo de Busca A*

```
while fila_prioridade.empty(
) == False:  # Enquanto a fila não estiver vazia
    atual = fila_prioridade.get()[1]  # Retira o nó com menor prioridade
    caminho.append(atual[0])  # Adiciona a cidade ao caminho
    distancia_total += int(atual[1])  # Atualiza a distância acumulada

    if atual[
            0] == cidade_destino:  # Verifica se a cidade atual é o destino
        break

    fila_prioridade = queue.PriorityQueue(
    )

    for vizinho in grafo[atual[0]]:
        if vizinho[
                0] not in caminho:
            fila_prioridade.put(
                (heuristicas[vizinho[0]] + int(vizinho[1]) +
                 distancia_total,
                 vizinho))  # Adiciona à fila com nova prioridade

return caminho, distancia_total
```

Função p/ Desenhar Mapa

```
# Função para desenhar o mapa e os caminhos
def desenhar_mapa(cidades, caminho_a_estrela, grafo):
    for nome_cidade, coordenadas in cidades.items(
    ):  # Percorre cada cidade e suas coordenadas
        plt.plot(coordenadas[0], coordenadas[1],
                 "ro")  # Desenha um ponto vermelho para cada cidade
        plt.annotate(
            nome_cidade,
            (coordenadas[0] + 5,
             coordenadas[1]))  # Adiciona o nome da cidade próximo ao ponto

        for conexao in grafo[
                nome_cidade]:  # Percorre as conexões de cada cidade
            cidade_conectada = cidades[
                conexao[0]]  # Obtém as coordenadas da cidade conectada
            plt.plot([coordenadas[0], cidade_conectada[0]],
                     [coordenadas[1], cidade_conectada[1]],
                     "gray")  # Desenha as conexões entre as cidades
```

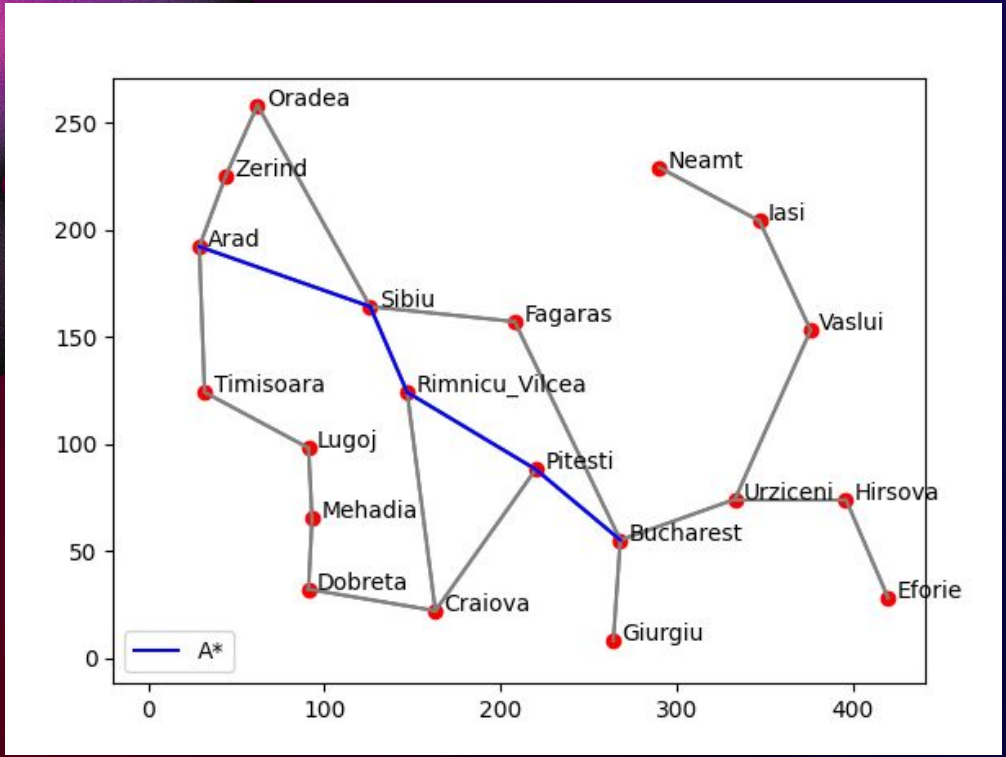
Função p/ Desenhar Mapa

```
for i in range(
        len(caminho_a_estrela)):  # Percorre o caminho encontrado pelo A*
    try:
        cidade_atual = cidades[
            caminho_a_estrela[i]]  # Coordenadas da cidade atual
        proxima_cidade = cidades[caminho_a_estrela[
            i + 1]]  # Coordenadas da próxima cidade

        plt.plot([cidade_atual[0], proxima_cidade[0]],
                 [cidade_atual[1], proxima_cidade[1]],
                 "blue")
    except:
        continue

plt.errorbar(1, 1, label="A*",
             color="blue")
plt.legend(
    loc="lower left")
plt.show()
```

Função Principal

```
# Função principal para executar o programa
def main():
    heuristicas = obter_heuristicas()  # Obtem as heurísticas do arquivo
    grafo = criar_grafo()  # Cria o grafo das cidades
    cidades, codigos_cidades = obter_cidades(
    )

    for codigo, nome in codigos_cidades.items(
    ):
        print(codigo, nome)

    while True:  # Loop para interação com o usuário
        codigo_cidade = int(
            input("Digite o código da cidade desejada (0 para sair): ")
        )  # Solicita o código da cidade
        if codigo_cidade == 0:  # Verifica se o usuário deseja sair
            break
        nome_cidade = codigos_cidades[
            codigo_cidade]  # Obtém o nome da cidade com base no código

        a_estrela, distancia_total = algoritmo_a_estrela(
            nome_cidade, heuristicas, grafo)
        print("ASTAR => ", a_estrela)
        print("Distância total percorrida: ", distancia_total, "km")
        desenhar_mapa(cidades, a_estrela, grafo)

main()
```

Arad - Bucharest



Repositório - GitHub

https://github.com/Laysabernardes/INTA