INSTITUTO FEDERAL DO ESPÍRITO SANTO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA – PPCOMP

BRUNA RUPP RUELA

TRABALHO 01 - IMPLEMENTAÇÃO DOS ALGORITMOS DE BUSCA - DFS - UCS - A-STAR

Serra
2023

BRUNA RUPP RUELA

TRABALHO 01 - IMPLEMENTAÇÃO DOS ALGORITMOS DE BUSCA - DFS - UCS - A-STAR

Trabalho apresentado ao Programa de Pós-Graduação em Computação Aplicada – PPCOMP do Instituto Federal do Espírito Santo, como requisito parcial para aprovação da Disciplina de Inteligencia Artificial.

Professor: Prof. Dr. Sergio Nery Simões

Serra
2023

LISTA DE FIGURAS

SUMÁRIO

1 PROBLEMAS PROPOSTOS

Algoritmos de busca desempenham um papel fundamental em ciência da computação e inteligência artificial, permitindo encontrar soluções para uma ampla gama de problemas, desde a roteirização de veículos até a solução de quebra-cabeças. Eles operam em estruturas de dados como grafos e árvores, explorando sistematicamente possíveis estados ou caminhos para encontrar uma solução ótima ou satisfatória. Esses algoritmos são essenciais em muitas aplicações práticas, incluindo sistemas de planejamento automatizado, jogos, robótica, mídias sociais (GILLESPIE, 2018) e muito mais (CORMEN et al., 2009).

1.0.1 Breadth-first search

O algoritmo de busca em largura é uma técnica de busca não informada que explora todos os nós vizinhos de um nó antes de seguir para os nós mais distantes. Ele inicia a busca a partir do nó raiz e avança em camadas, visitando todos os nós em uma camada antes de passar para a próxima. Para fazer isso, o algoritmo usa a estrutura de dados fila (= queue) de vértices. Esse método garante que o algoritmo encontre a solução mais próxima do nó inicial, sendo ideal para encontrar o caminho mais curto em um grafo não ponderado (RUSSELL et al., 2016).

1.0.2 Depth-first search

Ao contrário da busca em largura, o algoritmo de busca em profundidade explora o máximo possível ao longo de cada ramificação antes de retroceder. Ele percorre tão profundamente quanto possível ao longo de cada caminho antes de retroceder quando não há mais nós a serem explorados. Esse método pode consumir menos memória que a busca em largura, mas pode não encontrar a solução mais próxima, sendo mais adequado para encontrar soluções profundas em grafos grandes (RUSSELL et al., 2016).

1.0.3 Uniform cost search

Este é um algoritmo de busca informada que prioriza a exploração dos nós com menor custo acumulado g(n) até o momento. Ele expande os nós em ordem crescente de custo do caminho, garantindo que a solução encontrada seja a de menor custo. Isso o torna especialmente útil em problemas onde o custo do caminho entre os nós é variável (RUSSELL et al., 2016).

1.0.4 A* search

A* é um dos algoritmos de busca mais populares e eficientes, combinando as vantagens da busca informada e não informada. Ele utiliza uma função heurística g(h) para estimar o custo do caminho restante a partir de um nó até o objetivo, permitindo que ele se

concentre na exploração das áreas mais promissoras do espaço de busca. A* é completo e ótimo, desde que a heurística seja admissível e consistente (RUSSELL et al., 2016).

A Tabela 1 demonstra as complexidades de cada algoritmo. Onde V representa o número de vértices do grafo, E representa o numero de arestas, b representa o fator de ramificação da árvore de pesquisa e d representa a profundidade da solução ótima (GRASSI; PAIVA; ROCHA, 2022).

Algoritmo	Complexidade de tempo	Complexidade de espaço
BFS	$O(V + E)$	$O(V)$
DFS	$O(V + E)$	$O(V)$
UCS	$O(b^d)$	$O(b^d)$
A*	$O(b^d)$	$O(b^d)$

Tabela 1 – Complexidades de tempo e espaço dos algoritmos de busca

1.1 PROBLEMAS PROPOSTOS

1.1.1 QUESTÃO 01

Implementar e comparar o desempenho dos algoritmos de busca: *(i) Depth-first search*, *(ii) Uniform cost search* e *(iii) A* search*. Os algoritmos de busca podem ser úteis em entrevistas de programação em empresas de grande porte, então considere este tempo de implementação como um investimento na sua carreira. Para ajudá-los a começar e prover um ambiente para testes dos algoritmos, é fornecida uma implementação do algoritmo *breadth-first search* para busca de caminho em um labirinto em anexo à esta especificação. Você deve utilizar a estrutura de dados fornecida por essa implementação para implementar os demais algoritmos solicitados neste trabalho. Para o desenvolvimento do trabalho não é permitido o uso de bibliotecas que implementem os algoritmos, mas é permitido usar bibliotecas auxiliares, e.g., que implementem estruturas de dados (ex: *NetworkX*, etc). Para comparar os algoritmos, deve ser usado um labirinto com tamanho 200x200 com percentual de bloqueio 40%, e as métricas de comparação são:

1. tempo de execução

2. número de nós expandidos

3. custo do caminho

4. tamanho do caminho.

Você deverá criar uma Tabela comparativa com os algoritmos de busca (nas linhas) e as respectivas métricas resultantes (nas colunas) para cada algoritmo. Ao medir o tempo de

execução, desligue a visualização porque o custo de atualização da visualização é maior que o custo do algoritmo. Para a comparação ser justa, tome o cuidado de usar o mesmo labirinto em todos os casos (e.g., fixe o parâmetro seed=21 da classe MazeProblem).

1.1.2 QUESTÃO 02

Implementar os algoritmos *UCS* e *A* search* para o problema de roteamento entre cidades descrito no livro-texto da disciplina. O roteamento deve ser feito entre as cidades 'Arad' e 'Bucharest'. Para auxiliá-los, é fornecida uma implementação inicial do algoritmo no link abaixo:

https://colab.research.google.com/drive/15iUnVYFc5uA-Q2SQS7VpzQfAlO4r6ASX

Compare os resultados obtidos pela aplicação dos algoritmos UCS e A* search em relação ao (i) caminho obtido e (ii) custo do caminho através de uma tabela.

1.2 EXPERIMENTOS

Para testar os algoritmos foi utilizado o Vs Code para a questão numero 01 e o google colab para a questão numero 02. Inicialmente foi validado os algoritmos da questão 1 considerando uma matriz menor de 10 x 10 com a visualização ativada afim de acompanhar a evolução dos algoritmos. A versão python utilizada foi a 3.7. Para a questão 2 bastou apenas testar algumas cidades diferentes e comparar os resultados entre o algoritmo UCS e Astar no ambiente virtual oferecido pela plataforma Colab.

Para os experimentos foi utilizado com um laptop Macbook Air 2021 com 8gb de memória ram e processador M1.

1.3 RESULTADOS

Questão 01

A Tabela 2 compara os resultados ao executar os algoritmos BFS, DFS e A estrela numa matriz de 200x200 e um seed=21. Foi utilizado também um bloqueio de 40% para todos os algoritmos.

Tabela 2 – Comparação de Algoritmos de Busca

Algoritmo	Tempo de Execução	Nós Expandidos	Custo do Caminho
BFS	O(V+E)	357	359
DFS	O(V+E)	365	399
A*	O(bd)	3327	340.93

Na evolução de cada algoritmo, o tempo de execução do algoritmo A* foi menor em comparação com o BFS e o DFS, especialmente em labirintos grandes ou com caminhos

complexos, porque A* usa uma heurística para orientar a busca em direção ao objetivo. O BFS e o DFS exploram todas as opções possíveis sem levar em consideração a direção do objetivo. Essa característica pode ser observada comparando a execução de cada algoritmo.

A Figura 1 mostra o BFS explorando todas as fronteiras começando pela raiz e explorando todos os seus vizinhos antes de passar para os vizinhos dos vizinhos. Isso significa que ele explora em largura, camada por camada. Na Figura 2 o DFS explora em profundidade, seguindo um único caminho até que não haja mais nós a serem explorados antes de retroceder e na Figura 3 o A* tenta encontrar o caminho mais curto do ponto inicial ao ponto de destino (goal).

Figura 1 – Algoritmo BFS

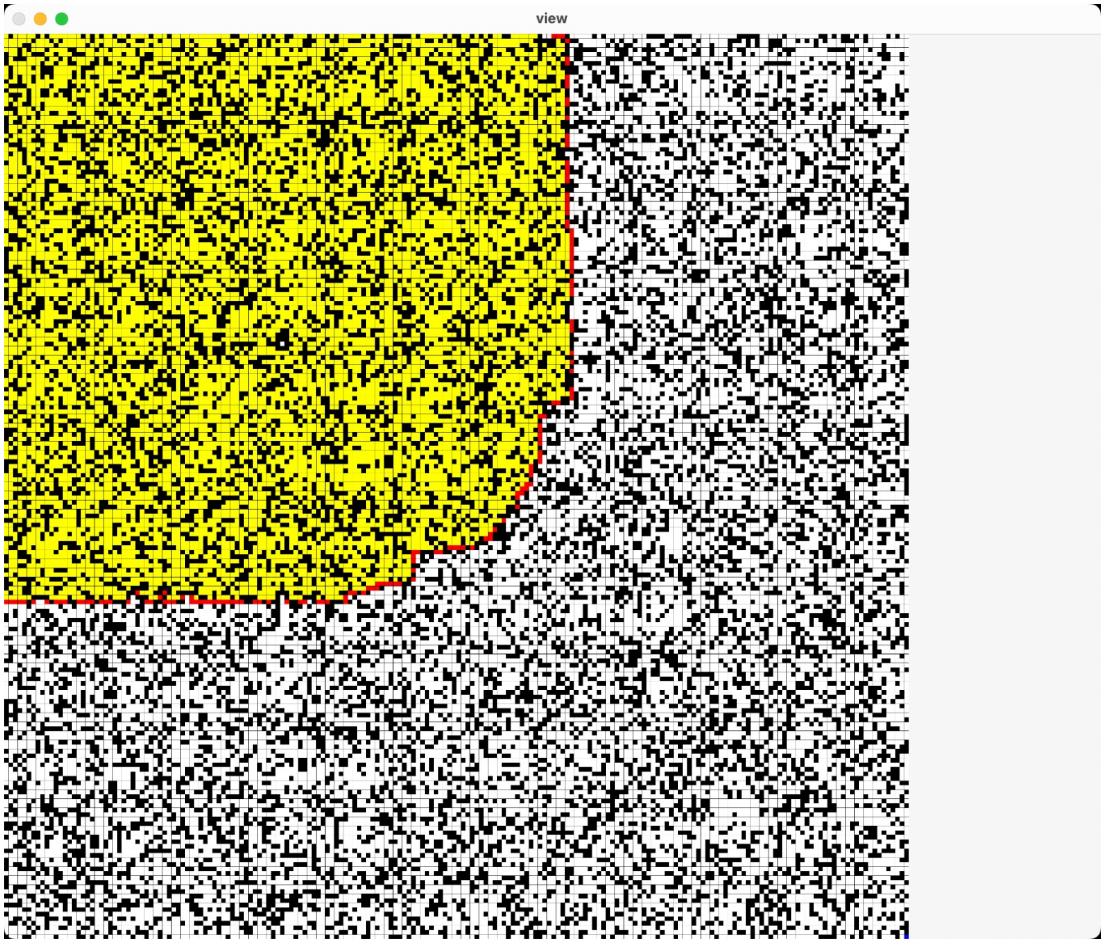
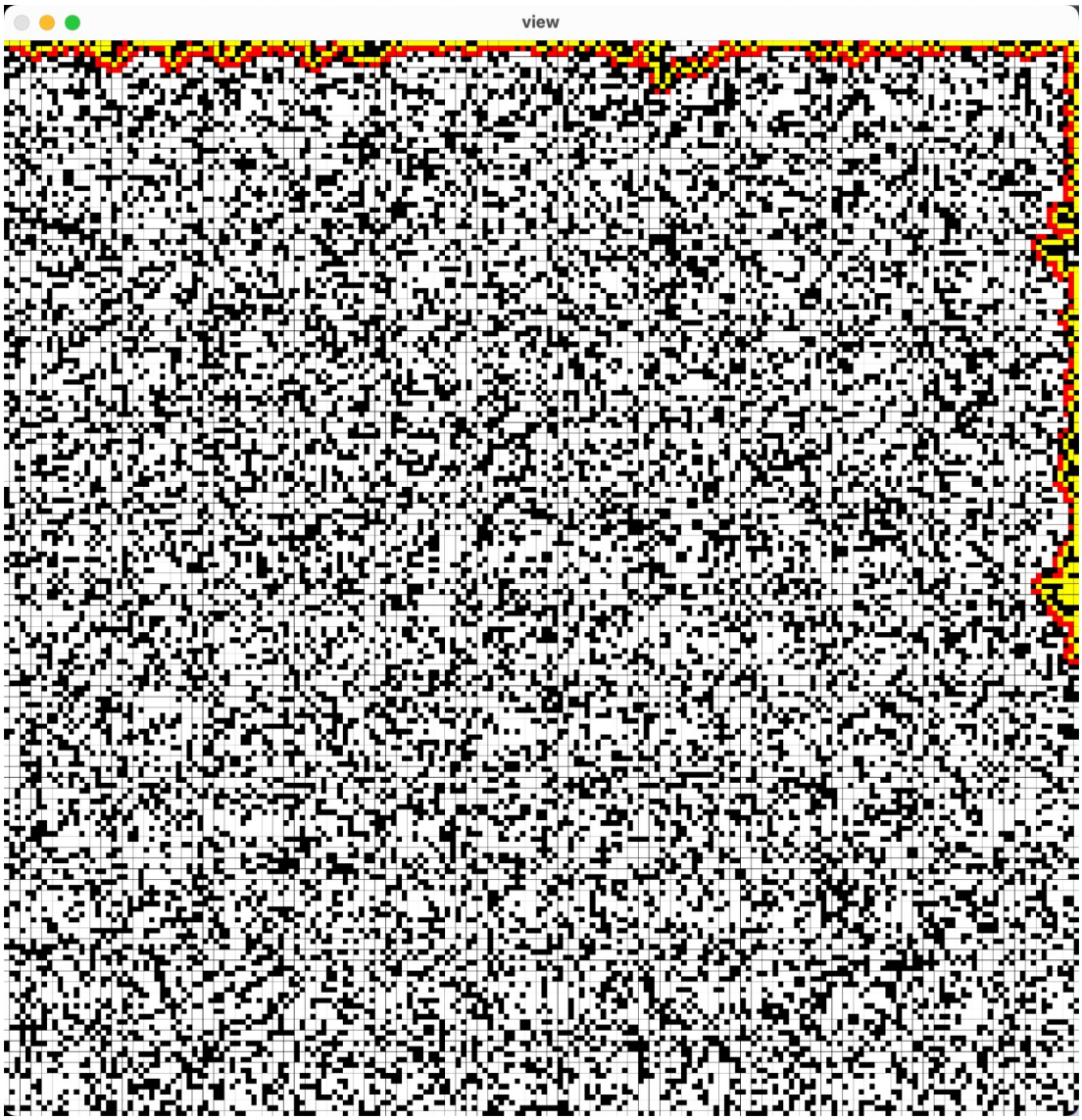


Figura 2 – Algoritmo DFS



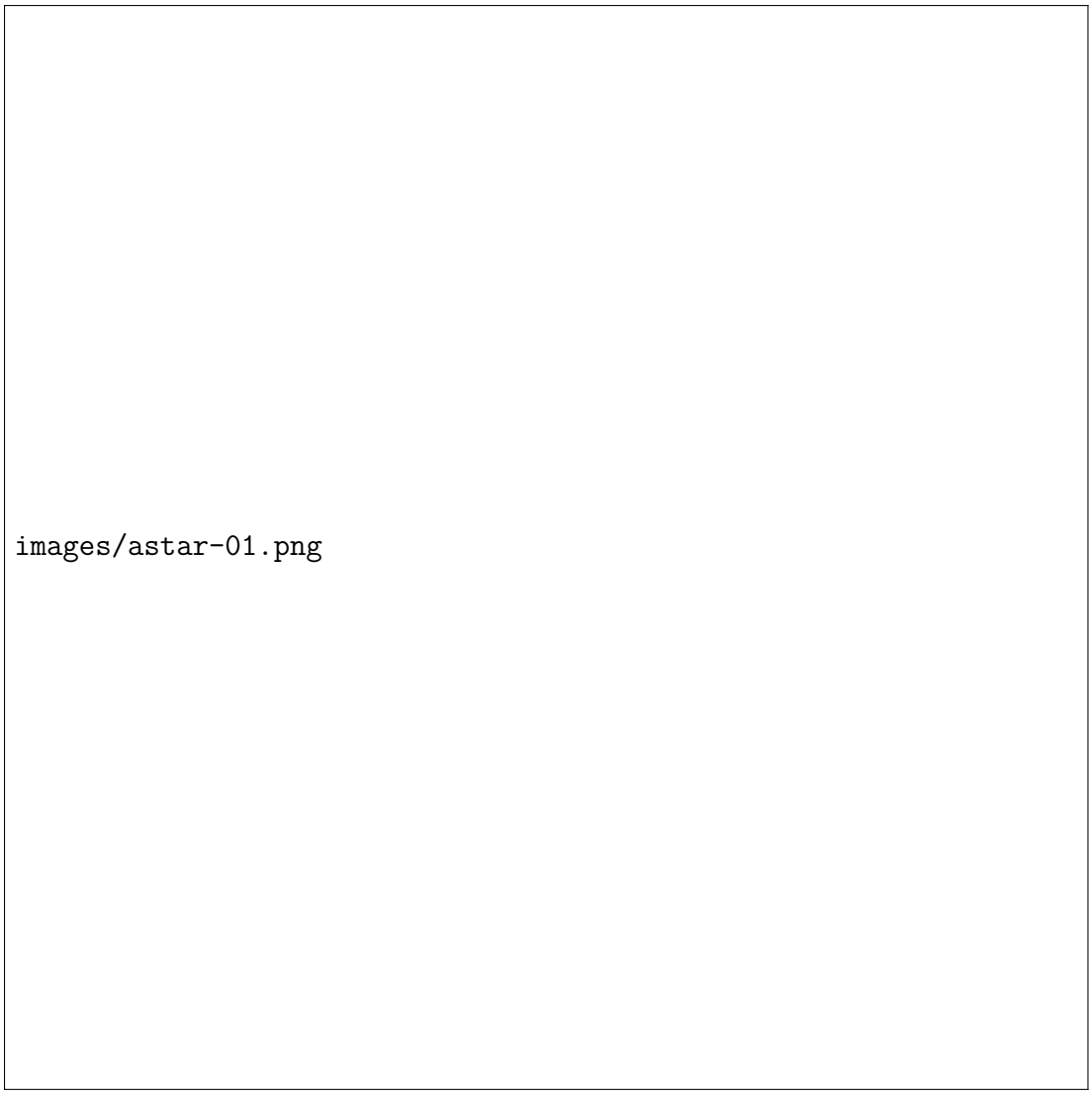
Questão 02

1.3.1 UCS

Para o algoritmo de Custo Uniforme, a fila de prioridade é mantida ordenada para que o nó com o menor custo seja sempre o primeiro a ser removido. Além disso, ao contrário do BFS, o UCS leva em consideração o peso das arestas ao calcular a distância. Portanto, a distância de um nó é a soma dos pesos das arestas no caminho do nó inicial até ele. A fila de prioridade é atualizada sempre que encontramos um caminho mais curto para um nó. Por fim, o algoritmo UCS também mantém um registro do nó predecessor de cada nó para que possamos reconstruir o caminho mínimo do nó inicial até qualquer outro nó.

Ao executarmos o algoritmo da Figura 4 o custo do caminho percorrido é de 450 e o caminho encontrado para sair de Arad e chegar a Bucharest foi: Caminho: 'Arad' para 'Sibiu' para 'Fagaras' para 'Bucharest'.

Figura 3 – Algoritmo A Star



1.3.2 A Star

Para o algoritmo de A estrela, além de considerar o mesmo custo do caminho como no UCS, Tabela 3, ele usa uma heurística para estimar o caminho até o destino. Foi utilizada a biblioteca PriorityQueue para gerenciar a fila de prioridade e a networkx para manipulação de grafos. O algoritmo faz as inicializações, em seguida, enquanto a fila de prioridade não estiver vazia, ele seleciona o nó com menor custo total estimado. Para cada vizinho do nó selecionado, atualiza o custo acumulado e a estimativa do custo restante se um caminho mais curto for encontrado. O processo continua até que o nó de destino seja alcançado ou até que todos os nós sejam explorados. Para o caminho mínimo, após encontrar o nó de destino, o código reconstrói o caminho mínimo percorrendo os predecessores dos nós a partir do nó de destino até o nó de origem.

Ao executarmos o algoritmo da Figura 5 o custo do caminho percorrido é de 418 e ele percorre um caminho com menos cidades para chegar ao destino 'Bucharest'.

Figura 4 – Algoritmo Uniform Cost Search

```python
from collections import deque

def UCS(G_inicial, s):
    G = G_inicial.copy()

    # INICIALIZACAO
    for v in G.nodes() - {s}:
        G.nodes[v]['cor'] = 'branco'
        G.nodes[v]['dis'] = np.inf

    G.nodes[s]['cor'] = 'cinza'
    G.nodes[s]['dis'] = 0

    # Fila de prioridade (deque)
    Q = deque()
    Q.append((0, s))  # Nó inicial adicionado com uma distancia 0

    #enquanto a fila não está vazia
    while len(Q) != 0:
        u = Q.popleft()[1]  # Pega e remove o nó com a menor distancia da fila

        for v in G.neighbors(u):
            # Se o nó não foi visitado ou se encontramos um caminho mais curto
            if G.nodes[v]['cor'] == 'branco' or G.nodes[v]['dis'] > G.nodes[u]['dis'] + G.edges[u, v]['weight']:
                G.nodes[v]['cor'] = 'cinza'
                G.nodes[v]['dis'] = G.nodes[u]['dis'] + G.edges[u, v]['weight']
                # define u como predecessor de v
                G.nodes[v]['pre'] = u

                # V é adicionado a fila e atualiza a fila de prioridade
                Q.append((G.nodes[v]['dis'], v))
                Q = deque(sorted(Q))  # Mantém a fila ordenada

        G.nodes[u]['cor'] = 'preto' #u visitado

        print(u, G.nodes[u]['dis'], G.nodes[u]['cor'])

    # Grafo G retornado contem as informações de distância
    # e cores desde o nó origem a todos os demais nós
    return G
```

Figura 5 – Algoritmo A estrela

```
[34] from queue import PriorityQueue

     def A_star(G, s, t, Estimation):
         # INICIALIZAÇÃO
         for v in G.nodes():
             G.nodes[v]['cor'] = 'branco'
             G.nodes[v]['dis'] = float('inf')

         G.nodes[s]['cor'] = 'cinza'
         G.nodes[s]['dis'] = 0

         # Fila de prioridade
         Q = PriorityQueue()
         Q.put((0, s))  # Nó inicial adicionado com prioridade 0

         while not Q.empty():
             _, u = Q.get()  # Pega e remove o nó com a menor distância da fila

             if u == t:  # Verifica se alcançou o destino
                 break

             for v in G.neighbors(u):
                 # Calcula o custo acumulado até o momento
                 custo_g = G.nodes[u]['dis'] + G.edges[u, v]['weight']
                 # Estima o custo do caminho até o destino
                 custo_h = estima_custo_h(v, Estimation)

                 # Se o nó não foi visitado ou encontramos um caminho mais curto
                 if G.nodes[v]['cor'] == 'branco' or custo_g + custo_h < G.nodes[v]['dis']:
                     G.nodes[v]['dis'] = custo_g + custo_h
                     # Define u como predecessor de v
                     G.nodes[v]['pre'] = u
                     # Atualiza a prioridade na fila de prioridade
                     Q.put((G.nodes[v]['dis'], v))

             G.nodes[u]['cor'] = 'preto'  # u visitado

             print(u, G.nodes[u]['dis'], G.nodes[u]['cor'])
```

Algoritmo	Caminho obtido	Custo do caminho
UCS	Arad 0 preto Zerind 75 preto Timisoara 118 preto Sibiu 140 preto Oradea 146 preto Rimnicu_Vilcea 220 preto Lugoj 229 preto Fagaras 239 preto Mehadia 299 preto Pitesti 317 preto Craiova 366 preto Dobreta 374 preto Bucharest 418 preto Bucharest 418 preto Urziceni 503 preto Giurgiu 508 preto Hirsova 601 preto Vaslui 645 preto Eforie 687 preto Iasi 737 preto Neamt 824 preto	418
A*	Arad 0 preto Sibiu 393 preto Timisoara 447 preto Zerind 449 preto Rimnicu_Vilcea 666 preto Fagaras 670 preto Lugoj 802 preto Pitesti 861 preto	418
BFS	Arad 0 preto Sibiu 1 preto Timisoara 1 preto Zerind 1 preto Fagaras 2 preto Oradea 2 preto Rimnicu_Vilcea 2 preto Lugoj 2 preto Bucharest 3 preto Pitesti 3 preto Craiova 3 preto Mehadia 3 preto Giurgiu 4 preto Urziceni 4 preto Dobreta 4 preto Hirsova 5 preto Vaslui 5 preto Eforie 6 preto Iasi 6 preto Neamt 7 preto	450

Tabela 3 – Caminhos e custos dos algoritmos

REFERÊNCIAS

CORMEN, Thomas H. et al. *Introduction to Algorithms*. [S.l.]: The MIT Press, 2009. ISBN 0262033844.

GILLESPIE, Tarleton. A relevância dos algoritmos. *Parágrafo*, v. 6, n. 1, p. 95–121, 2018.

GRASSI, Larissa; PAIVA, Marcia Helena Moreira; ROCHA, Helder. Redução de complexidade em algoritmo de busca de medidas críticas em estimação de estado via grafos. Galoá, 2022.

RUSSELL, Stuart et al. *Artificial Intelligence: A Modern Approach*. [S.l.]: Pearson, 2016.