Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Inteligência Artificial (2º semestre de 2022)

Projeto 1 - Algoritmos de busca

Gustavo Henrique Brunelli	NUSP: 11801053
Matheus Henrique de Cerqueira Pinto	NUSP: 11911104
Matheus Ventura de Sousa	NUSP: 11345541
Pedro Lucas de Moliner de Castro	NUSP: 11795784

Prof. Dr. Alneu de Andrade Lopes

São Carlos, Outubro de 2022

1 Introdução

Uma questão enfrentada com muita frequência no domínio de problemas que envolvem modelagem com grafos é explorar um grafo a partir de um vértice. Os algoritmos de busca, ou de varredura, oferecem uma solução para esse problema, cada um usando diferentes métodos e proporcionando tempo de solução distintos.

Este trabalho visa comparar os algoritmos de varredura mais utilizados, analisando as vantagens e desvantagens em relação aos outros algoritmos apresentados, a fim de entender com mais profundidade as ferramentas e técnicas que existem hoje para solução de problemas de grafos.

2 Implementação

2.1 Criação do grafo

Por ter de representar grafos com um número de arestas muito pequeno em relação ao número de vértices, o grupo optou por utilizar a estrutura de dados de **lista de adjacência**. Além disso, apesar de não ser a definição original de um grafo-knn, decidiu-se conectar cada vértice à outros k vértices aleatórios para seguir o algoritmo de criação descrito no enunciado do trabalho.

No código, foram utilizadas duas classes auxiliares, a `Coord` que armazena as coordenadas x e y de um vértice e a `Edge` que representa uma aresta armazenando seu vértice de destino e peso. Uma função auxiliar `dist()` também foi definida para calcular a distância euclidiana entre dois vértices e a geração do

grafo aleatória é feita na construção de uma classe pai `Graph`.

```
def __init__(self, vertices: int, neighbors: int):
    self.vertices = vertices
    self.neighbors = neighbors

    # Generate the vertices x and y coordinates
    self.coords = [self.Coord(random.uniform(0, vertices), random.uniform(0, vertices)) for _ in range(vertices)]

    # Generate the vertices neighbors
    population = set(range(vertices))
    self.adj = [[self.Edge(neighbor, self.dist(vertex, neighbor)) for neighbor in random.sample(population - {vertex}, neighbors)] for vertex in range(vertices)]
```

2.2 Algoritmos de busca

Para os algoritmos DFS, BFS e Best First, foi implementada uma versão genérica de algoritmos de busca baseada em agenda nomeada no código como `_agenda()`. Com esse algoritmo base, é necessário apenas alterar a agenda usada para trocar o algoritmo utilizado, sendo uma **pilha** para o DFS, uma **fila** para o BFS e uma **fila de prioridade** para o Best First.

```
def _agenda(graph: Graph, src: int, goal: int, agenda) -> tuple[float, list[int], set[int]]:
    visited: set[int] = set()
    agenda.insert(0.0, [src])

    while agenda:
        dist, path = agenda.remove()
        curr = path[-1]

        if curr == goal:
            return dist, path, visited

        if curr in visited: # Ignore redundant entries
            continue

        visited.add(curr)

        for neighbor, weight in graph[curr]:
            if neighbor not in visited:
                agenda.insert(dist + weight, path + [neighbor])

    return math.inf, [], visited
```

Já na implementação dos algoritmos de Dijkstra, A e A*, mais uma vez foi criado uma versão base nomeada `_astar()`, que aceita uma heurística como parâmetro. Dessa forma, basta utilizar a heurística **zero** para o Dijkstra, a

distância euclidiana no A* e uma heurística não otimista para o A, tendo sido utilizada **100 vezes a distância geométrica** para exemplificar facilmente a não otimalidade desse algoritmo.

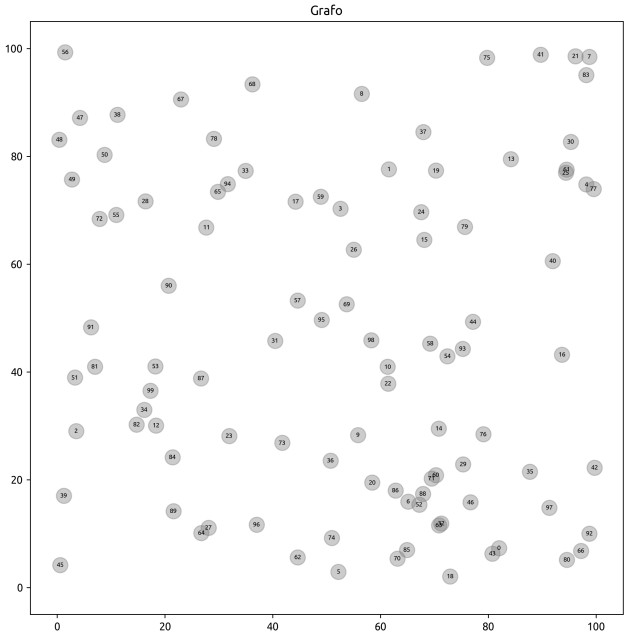
```python
def _astar(graph: Graph, src: int, goal: int, heuristic: Callable[[int, int], float]):
    parents: dict[int, int] = {}
    visited: set[int] = set()

    queue = [(0.0, 0.0, src)]

    dists = [math.inf for _ in range(len(graph))]
    dists[src] = 0.0

    while queue:
        _, dist, curr = heapq.heappop(queue)
        visited.add(curr)

        if curr == goal:
            return dists[goal], _traceback(parents, goal), visited

        if dist > dists[curr]: # Ignore redundant entries
            continue

        for neighbor, weight in graph[curr]:
            dist = dists[curr] + weight

            if dist < dists[neighbor]:
                dists[neighbor] = dist
                parents[neighbor] = curr
                heapq.heappush(queue, (dist + heuristic(neighbor, goal), dist, neighbor))

    return math.inf, [], visited
```

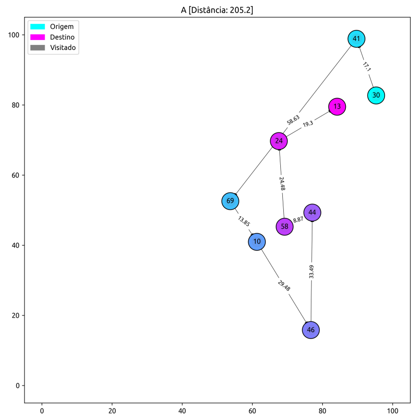
Nota: A forma de reconstrução do caminho percorrido nos algoritmos é diferente, sendo que o `_agenda()` utiliza múltiplas listas enquanto o `_astar()` usa um dicionário de vértices pais, sendo mais eficiente. Devido a isso, o grupo decidiu implementar duas versões do mesmo algoritmo de busca informada com heurística zero, a primeira sendo uma implementação por agenda com uma fila de prioridade nomeada Best First e a segunda a utilização do A* passando a heurística zero chamada de Dijkstra.

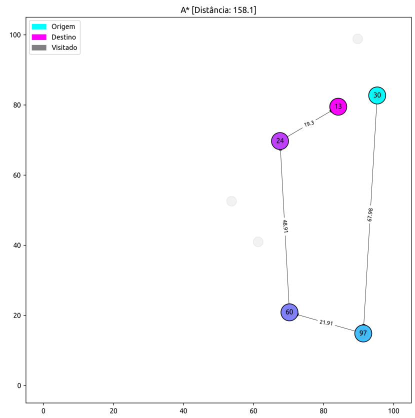
3 Experimentos

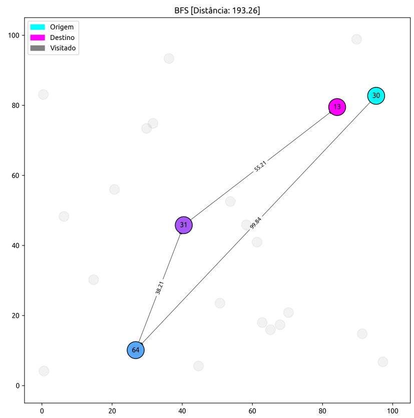
3.1 Visualização dos caminhos

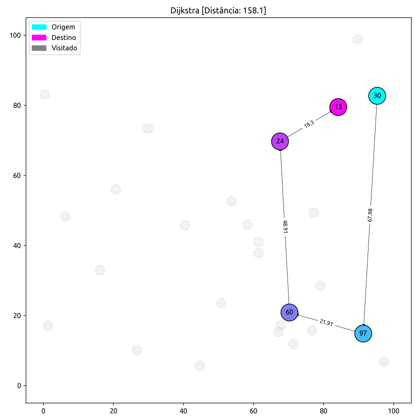
Para fácil visualização de exemplos dos caminhos encontrados pelos algoritmos, foi utilizado um grafo com 100 vértices e 3 arestas por vértice partindo do vértice 30 e chegando em 13.

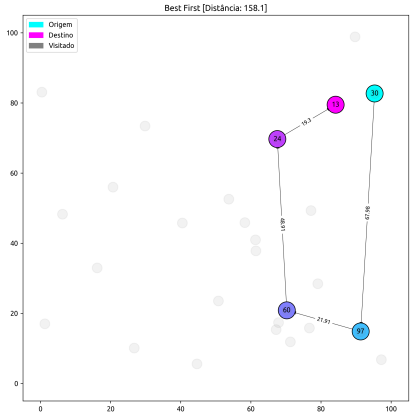


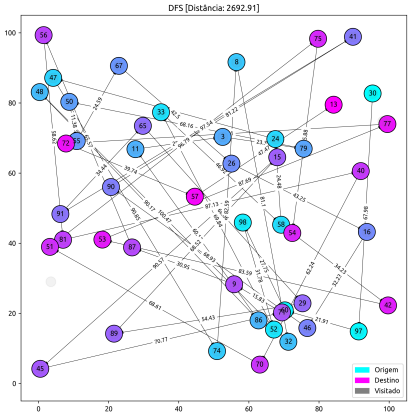

A [Distância: 205.2]
Origem
Destino
Visitado


A* [Distância: 158.1]
Origem
Destino
Visitado


BFS [Distância: 193.26]
Origem
Destino
Visitado


Dijkstra [Distância: 158.1]
Origem
Destino
Visitado





Nota: O código construído é capaz de gerar imagens para grafos de qualquer tamanho, porém a visualização do caminho pode não ser tão boa nesses casos, principalmente para o DFS.

3.2 Comparação dos algoritmos

A comparação mais aprofundada dos algoritmos foi feita com três configurações de grafos com 5000 vértices e 3, 5 e 7 arestas por vértice respectivamente. Para cada uma delas, foram gerados vinte pares aleatórios válidos de origem e destino e calculadas informações sobre o tempo de execução, distância do caminho percorrido, número de vértices no caminho e número de vértices visitados.

Tabela 1: Grafo de 5000 vértices e 3 arestas

	Tempo de Execução	Distância	Vértices no Caminho	Vértices Visitados
DFS	40.66ms	4291144.63	1669.80	2587.40
BFS	4.06ms	17893.94	8.20	2066.65
Best First	4.81ms	15781.99	9.25	1899.25
Dijkstra	3.72ms	15781.99	9.25	1900.25
A	5.28ms	17695.02	10.75	1119.05
A*	3.39ms	15781.99	9.25	1008.00

Tabela 2: Grafo de 5000 vértices e 5 arestas

	Tempo de Execução	Distância	Vértices no Caminho	Vértices Visitados
DFS	80.93ms	5455053.23	2108.40	2422.55
BFS	10.24ms	13868.42	6.35	2291.85
Best First	9.86ms	10645.35	7.70	2148.45
Dijkstra	7.07ms	10645.35	7.70	2149.45
A	7.85ms	13790.31	9.45	1000.35
A*	3.82ms	10645.35	7.70	693.40

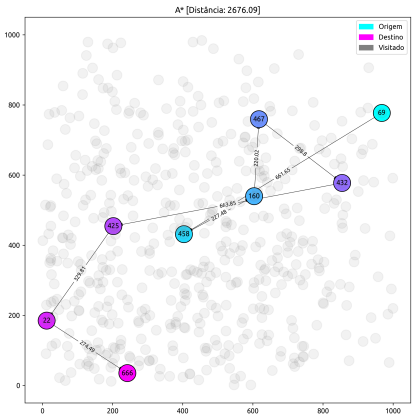
Tabela 3: Grafo de 5000 vértices e 7 arestas

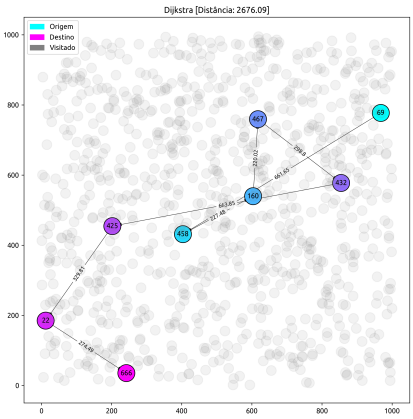
	Tempo de Execução	Distância	Vértices no Caminho	Vértices Visitados
DFS	134.72ms	5766864.78	2190.60	2266.95
BFS	13.29ms	11748.21	5.55	2594.75
Best First	16.10ms	8507.42	6.85	2618.40
Dijkstra	8.68ms	8507.42	6.85	2619.40
A	8.08ms	10259.95	8.50	826.00
A*	4.77ms	8507.42	6.85	728.65

3.3 Comparação entre A* e Dijkstra

Analisando as tabelas anteriores, é possível perceber que os algoritmos de Dijkstra e A* sempre retornam o mesmo resultado, sendo ele o caminho ótimo, no entanto, como o A* se utiliza de uma heurística otimista mais próxima da

realidade, ele visita menos vértices e, assim, leva menos tempo para executar. A diferença fica clara ao visualizar um grafo com 1000 vértices e 5 vizinhos.





4 Conclusão

Ao comparar os resultados, é possível notar que o algoritmo DFS não é uma boa escolha para esse tipo de problema, retornando caminhos longos e tomando muito tempo, já o BFS seria a melhor e mais simples opção em grafos sem peso, porém, como esse não é o caso, sua solução é sub ótima. Dentre o Best First e o Dijkstra, a implementação referida pelo último é mais eficiente e recomendada caso não se tenha uma heurística confiável para o problema, pois aplicar o A* pode aumentar bastante o desempenho, mas caso uma heurística incorreta seja formulada ele pode cair no algoritmo A, que não fornece a solução ótima e pode demorar tanto quanto ou mais que o Dijkstra.