

Análise e implementação do algoritmo de Dijkstra

Lucas Mateus da Silva*

Joao Paulo de Souza Medeiros[†]

Recebido em 19 de julho de 2023, aceito em 19 de julho de 2023.

Resumo

Relatório remetente a implementação do algoritmo de Dijkstra, como forma de avaliação extra referente a 3ª unidade da disciplina de Estrutura de Dados.

1 Introdução

O algoritmo de Dijkstra é amplamente usado para encontrar o caminho mais curto entre um ponto inicial e todos os outros pontos em um grafo ponderado não direcionado. Essa técnica segue uma abordagem gananciosa, o que significa que em cada etapa, ele seleciona o vértice com a menor distância conhecida até o momento.

O conceito central por trás do algoritmo é manter um conjunto de vértices cujos caminhos mais curtos já foram determinados. Inicialmente, todos os vértices são considerados não visitados, exceto o vértice inicial, que possui uma distância mínima de zero. A cada iteração, o algoritmo escolhe o vértice com a menor distância acumulada e atualiza as distâncias dos vizinhos não visitados, caso exista uma rota mais curta passando por esse vértice. Esse processo continua até que todos os vértices tenham sido visitados.

Ao concluir o algoritmo, teremos determinado as distâncias mínimas a partir do vértice inicial para todos os outros vértices do grafo. Além disso, o caminho mais curto entre o vértice inicial e qualquer outro vértice também pode ser reconstruído, seguindo as arestas com as menores distâncias acumuladas.

2 Complexidade

A complexidade do algoritmo de Dijkstra depende da forma como o grafo é representado. Se optarmos por uma matriz de adjacência, a complexidade será $O(V^2)$, onde V é o número de vértices no grafo. Nesse caso, encontrar o vértice não visitado com a menor distância leva $O(V)$ operações, repetidas V vezes.

Por outro lado, ao usar uma lista de adjacência, o algoritmo pode ser otimizado para uma complexidade de tempo de $O((V + E)\log V)$, em que E é o número de arestas no grafo. Essa otimização é possível utilizando uma estrutura de dados como uma fila de prioridade (**heap**) para selecionar eficientemente o próximo vértice com a menor distância.

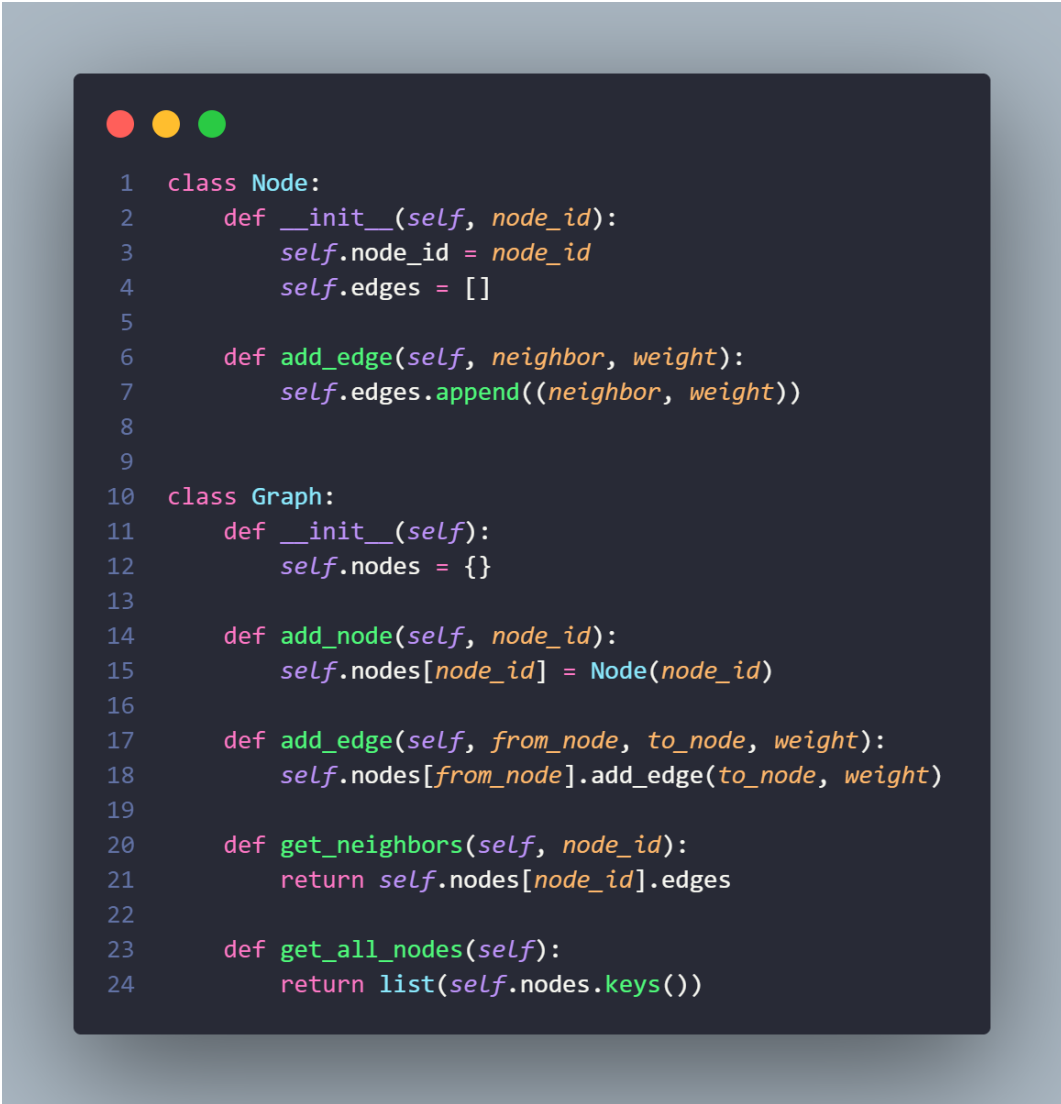
Em resumo, a complexidade do algoritmo de Dijkstra com matriz de adjacência é $O(V^2)$, e com lista de adjacência é $O((V + E)\log V)$. A escolha da representação adequada do grafo é importante para obter um melhor desempenho do algoritmo.

*Aluno do Bacharelado em Sistemas de Informação da Universidade Federal do Rio Grande do Norte. (e-mail: lucas.mateus.130@ufrn.edu.br)

[†]Professor do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte. Coordenador do Laboratório de Elementos do Processamento da Informação. (e-mail: joao.paulo.medeiros@ufrn.br)

3 Implementação

Para a implementação do algoritmo foi utilizada a linguagem python, utilizando orientação a objetos com a ajuda do modulo **heapq** que implementa a fila de prioridade que será utilizada no algoritmo. O algoritmo implementa o grafo utilizando OO, com a classe **Graph** que contem classes **Node**, como visto em 1. Após a criação do grafo usando o arquivo fornecido, é aplicada o algoritmo de Dijkstra, para encontrar todos os possiveis menores caminhos para cada vertice a partir do nó inicial, visto em 2 e logo após pode ser verificado a menor distancia entre dois nós usando a função vista em 3.



```
1 class Node:
2     def __init__(self, node_id):
3         self.node_id = node_id
4         self.edges = []
5
6     def add_edge(self, neighbor, weight):
7         self.edges.append((neighbor, weight))
8
9
10 class Graph:
11     def __init__(self):
12         self.nodes = {}
13
14     def add_node(self, node_id):
15         self.nodes[node_id] = Node(node_id)
16
17     def add_edge(self, from_node, to_node, weight):
18         self.nodes[from_node].add_edge(to_node, weight)
19
20     def get_neighbors(self, node_id):
21         return self.nodes[node_id].edges
22
23     def get_all_nodes(self):
24         return list(self.nodes.keys())
```

Figura 1: Classes Graph e Node implementadas em python

4 Links Externos

Os algoritmos que foram usados para os testes contidos nesse documento podem ser encontrados no seguinte repositório remoto no github: [data-structure](#)

```
1 def dijkstra(G, start):
2     dists = {node: float('inf') for node in G.get_all_nodes()}
3     predecessors = {node: [] for node in G.get_all_nodes()}
4     dists[start] = 0
5
6     fila = [(0, start)]
7
8     while fila:
9         dist_atual, no_atual = heapq.heappop(fila)
10        if dist_atual > dists[no_atual]:
11            continue
12
13        for neighbor, peso in G.get_neighbors(no_atual):
14            dist = dists[no_atual] + peso
15            if dist < dists[neighbor]:
16                dists[neighbor] = dist
17                predecessors[neighbor] = [no_atual]
18                heapq.heappush(fila, (dist, neighbor))
19            elif dist == dists[neighbor]:
20                predecessors[neighbor].append(no_atual)
21
22    return dists, predecessors
```

Figura 2: Algoritmo de Dijkstra implementado em python

```
1 def find_shortest_path(predecessors, start, end):
2     path = [end]
3     while path[-1] != start:
4         path.append(predecessors[path[-1]][0])
5     return path[::-1]
6
```

Figura 3: Função para apontar o menor caminho entre dois vertices