Implementação Web do Algoritmo de Dijkstra para Cálculo de Caminhos Mínimos em Dígrafos

Diogo Francis Belshoff[1] Eduardo Godio Rodrigues[2]
Estevão da Silva Florencio[3] Giovanni Persio Gonçalves[4]
Robert Lopes Terra[5] Werisder de Souza Bertoli[6]

2025

Resumo

Este trabalho apresenta o desenvolvimento de uma aplicação web capaz de calcular caminhos mínimos em grafos direcionados, utilizando o algoritmo de *Dijkstra*. A solução inclui uma interface interativa conectada a uma API REST, que permite ao usuário carregar a estrutura do grafo, selecionar o vértice de origem e visualizar os resultados em tempo real. Os testes demonstraram que o sistema é eficaz na resolução do problema proposto, oferecendo uma ferramenta útil para visualização e análise de rotas em redes direcionadas.

Palavras-chaves: Algoritmo de *Dijkstra*. Grafos direcionados. API REST. Caminho mínimo. Visualização de rotas.

Introdução

O algoritmo de *Dijkstra* é um dos algoritmos mais fundamentais, conhecidos e importantes na área da computação no geral. O algoritmo foi desenvolvido em 1956 pelo cientista da computação *Edsger W. Dijkstra*. O algoritmo tem como objetivo encontrar o caminho com menor custo em um grafo ponderado. O algoritmo se tornou base para várias áreas do conhecimento, além de tecnologias presentes no dia-a-dia como sistemas de navegação, logística, aplicativos de transporte (Uber e 99), redes de computadores e entre outras aplicações. Neste trabalho, buscamos implementar uma solução prática que integre esse algoritmo a uma aplicação *web*, possibilitando o cálculo automático do custo de caminhamento em um dígrafo. A proposta consiste em desenvolver uma interface que permita ao usuário carregar um arquivo contendo a estrutura do grafo, selecionar o vértice de partida e, a partir disso, visualizar tanto os caminhos mínimos quanto seus respectivos custos para todos os demais vértices. Essa abordagem visa tornar mais acessível a compreensão do funcionamento do algoritmo de *Dijkstra*, ao mesmo tempo em que oferece uma ferramenta funcional para análise de rotas e estrutura de grafos

direcionados. A implementação proposta permite explorar, de forma aplicada, os conceitos fundamentais dos grafos e do algoritmo, destacando sua relevância em contextos práticos como planejamento de trajetos, redes de transporte e sistemas de recomendação.

Desenvolvimento

Implementação do algoritmo de *Dijkstra* em python

A lógica foi estruturada em duas classes principais: *Grafo*, responsável pela leitura e interpretação da estrutura do grafo a partir de um arquivo de entrada, e *AlgorithmDijkstra*, encarregada do cálculo dos caminhos mínimos a partir de um vértice fonte. A classe `Grafo` processa um arquivo *.txt* fornecido pelo usuário, no qual cada linha descreve as arestas e seus respectivos pesos em um formato previamente estabelecido. A construção do grafo é realizada por meio de uma lista de adjacência baseada em dicionários aninhados, garantindo flexibilidade e eficiência no acesso aos vértices e suas conexões. Uma vez estruturado o grafo, a classe *AlgorithmDijkstra* aplica o algoritmo clássico de *Dijkstra* utilizando uma fila de prioridade (*heap*), com o objetivo de calcular os menores custos e reconstruir os caminhos mínimos entre o vértice inicial e os demais vértices alcançáveis. A resposta é então formatada em um objeto *JSON*, compatível com a interface de *front-end* da aplicação.

Método construtor da classe

O ponto de partida da estruturação do grafo está no método construtor da classe, conforme apresentado a seguir:

Código 1: Método *___init___* da classe *Grafo*

```
def __init__(self, arquivo: UploadFile | None = None):
    self.lista_adjacencia: Dict[str, Dict] = {}
    self.__validar_arquivo(arquivo=arquivo)
```

Fonte: elaborado pelo autor.

Nesse trecho, observa-se que o atributo *lista_adjacência* é inicializado como um dicionário vazio, no qual cada chave representa um vértice (identificado por uma cadeia de caracteres, usualmente uma letra), e cada valor é outro dicionário que associa vértices adjacentes aos pesos das arestas correspondentes. Em seguida, o método privado *___validar_arquivo* é invocado com o parâmetro *arquivo*, sendo responsável por interpretar o conteúdo de um arquivo de entrada no formato *.txt* e, a partir disso, construir a estrutura do grafo. Tal abordagem assegura que a representação do grafo esteja completamente definida logo após a instanciação da classe.

Processamento do arquivo de entrada

Após, o método privado *___validar_arquivo* é invocado com o objetivo de interpretar o conteúdo de um arquivo de entrada no formato *.txt* e construir a estrutura de adjacência correspondente. Esse método realiza a leitura do arquivo utilizando o objeto *UploadFile* fornecido pelo FastAPI, decodificando seu conteúdo em UTF-8 e separando-o em linhas individuais

Código 2: Método *___validar_arquivo* da classe *Grafo*

```
def __validar_arquivo(self, arquivo: UploadFile) -> None:
    if arquivo is None:
        raise FileNotFoundError("O arquivo não pode ser nulo")
    linhas = arquivo.file.read().decode('utf-8').splitlines()
    # Inicializar todos os vértices (origem e destino)
    for linha in linhas:
        if not linha.strip():
            continue
        parts = re.split(r":\s*", linha, 1)
        if len(parts) != 2:
            continue
        origem = parts[0]
        if origem and re.match(r"^[A-Za-z]$", origem):
            self.lista_adjacencia[origem] =
            self.lista_adjacencia.get(origem, {})
            destinos = re.split(r",\s*", parts[1]) if parts[1] else []
            for destino in destinos:
                if re.match(r"^[A-Za-z][0-9]+$", destino):
                    vertice_ligado = destino[0]
                    self.lista_adjacencia[vertice_ligado] =
                    self.lista_adjacencia.get(vertice_ligado, {})
```

Fonte: elaborado pelo autor.

O processamento do arquivo ocorre em duas etapas principais. Na primeira, são identificados todos os vértices presentes, tanto de origem quanto de destino, garantindo que todos estejam representados na estrutura de adjacência, mesmo que inicialmente não tenham arestas associadas. Essa etapa evita erros de chave ausente durante o preenchimento das conexões. Na segunda etapa, as arestas são efetivamente adicionadas ao grafo por meio do método auxiliar *___add_edge*, que associa cada vértice de origem aos seus destinos e respectivos pesos. A extração do vértice de destino e do peso é feita a partir de uma *string* no formato Xn, onde X é o nome do vértice e n é o peso da aresta

Código 3: Método *___add__edge* da classe *Grafo*

```
# Adicionar arestas
for linha in linhas:
    if not linha.strip():
        continue
    parts = re.split(r":\s*", linha, 1)
    if len(parts) != 2:
        continue
    origem = parts[0]
    if not (origem and re.match(r"^[A-Za-z]$", origem)):
        continue
    destinos = re.split(r",\s*", parts[1]) if parts[1] else []
    lista_vertice = [item for item in destinos if re.match
    (r"^[A-Za-z][0-9]+$", item)]
    self.__add_edge(origem, lista_vertice)

    def __add_edge(self, principal: str, lista_vertice: List[str])
    -> None: if not lista_vertice:
    return
for vertice in lista_vertice:
    vertice_ligado = str(vertice[0])
    try:
        peso = int(vertice[1:])
        self.lista_adjacencia[principal][vertice_ligado] = peso
    except ValueError:
        pass
```

Fonte: elaborado pelo autor.

Cada vértice é representado por uma chave no dicionário principal, e os valores correspondem a dicionários secundários que mapeiam os vértices adjacentes aos pesos das arestas. Essa estrutura permite acesso rápido aos vizinhos de um vértice e facilita a atualização dinâmica das conexões, o que é essencial para algoritmos como o de Dijkstra, que requerem a exploração eficiente dos vizinhos de cada nó

Dijkstra

A seguir, apresentamos a implementação do algoritmo de *Dijkstra* em Python, conforme desenvolvida na classe *AlgorithmDijkstra*. A classe AlgorithmDijkstra é responsável por calcular os caminhos mínimos em um grafo ponderado, utilizando o algoritmo de *Dijkstra*. O método *obter_caminho_vertice* recebe um vértice fonte e determina o menor custo para alcançar todos os outros vértices do grafo, além de reconstruir os caminhos correspondentes.

Inicialmente, os dicionários custo e predecessores são configurados: custo armazena o custo mínimo conhecido para alcançar cada vértice a partir da fonte, sendo todos inicializados com infinito, exceto o vértice fonte, que recebe zero; predecessores mantém o rastreamento do caminho percorrido até cada vértice.

Uma fila de prioridade (implementada com *heapq*) é utilizada para selecionar o próximo vértice a ser processado, sempre escolhendo aquele com o menor custo acumulado. Durante a iteração, para cada vizinho do vértice atual, calcula-se a distância total desde a fonte. Se essa distância for menor que a previamente registrada, atualiza-se o custo e o predecessor do vizinho, além de inseri-lo na fila de prioridade.

Código 4: Método *obter_caminho_vertice* da classe *Dijkstra*

```
def obter_caminho_vertice(self, fonte: str) -> Dict[str, Any]:
    custo = {vertices: float("inf") for vertices in self.grafo.li
    sta_adjacencia}
    custo[fonte] = 0
    predecessores = {vertices: None for vertices in self.grafo.li
    sta_adjacencia}
    fila = [(0, fonte)]
    heapq.heapify(fila)
    no_visitados = set()
    while fila:
        custo_atual, no_atual = heapq.heappop(fila)
        if no_atual in no_visitados:
            continue
        no_visitados.add(no_atual)
        for neighbor, weight in self.grafo.lista_adjacencia[no_at
        ual].items():
            tentative_distance = custo_atual + weight
            if tentative_distance < custo[neighbor]:
                custo[neighbor] = tentative_distance
                predecessores[neighbor] = no_atual
                heapq.heappush(fila, (tentative_distance, neighbo
                r))
    caminhos = self.reconstruir_todos_caminhos(predecessores=pre
    decessores, fonte=fonte)
    response = {
        "startNode": fonte,
        "paths": []
    }
    for target, path in caminhos.items():
        if path and target != fonte:
            response["paths"].append({
                "target": target,
                "path": path,
                "cost": custo[target] if custo[target] != float
                ("inf") else None
            })
    return response
```

Fonte: elaborado pelo autor.

Após processar todos os vértices, o método *reconstruir_todos_caminhos* é chamado

para reconstruir os caminhos mínimos, utilizando as informações armazenadas em predecessores. O resultado final é estruturado em formato *JSON*, contendo o vértice inicial e uma lista de caminhos para cada destino, incluindo o caminho percorrido e o custo associado.

Código 5: Método *reconstruir_todo_caminhos* da classe *Dijkstra*

```
def reconstruir_todos_caminhos(self, predecessores: Dict[str, O
ptional[str]], fonte: str) -> Dict[str, Optional[List[str]]]:
    caminhos = {}

    for destino in predecessores:
        caminho = []
        atual = destino

        while atual is not None:
            caminho.append(atual)
            atual = predecessores[atual]
        caminho.reverse()

        if caminho and caminho[0] == fonte:
            caminhos[destino] = caminho
        else:
            caminhos[destino] = None

    return caminhos
```

Fonte: elaborado pelo autor.

Essa implementação segue a abordagem clássica do algoritmo de Dijkstra, adequada para grafos com pesos não negativos, e é eficiente para encontrar os caminhos mínimos a partir de uma única fonte.

Resultados

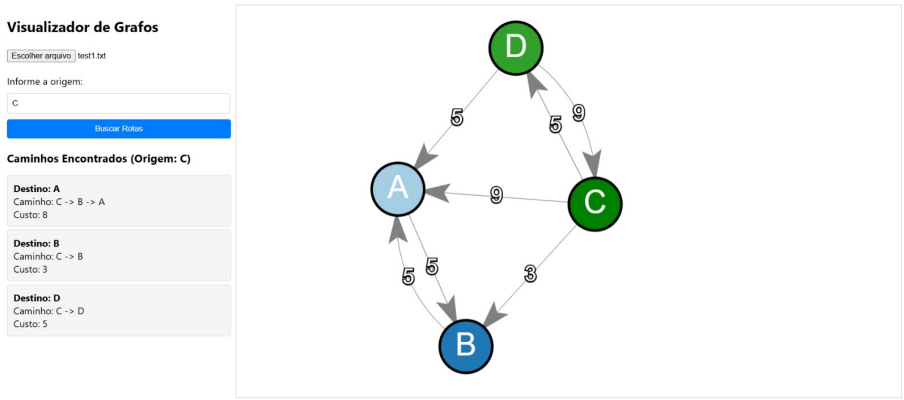
Nesta seção, são apresentados os resultados obtidos a partir da execução do algoritmo de Dijkstra, conforme descrito na seção de desenvolvimento. Para os testes, foram utilizados arquivos de texto contendo a estrutura dos grafos, enquanto o vértice de origem foi informado pelo usuário por meio da interface da aplicação.

Teste 1 – Grafo com ciclos

O primeiro teste utilizou um grafo direcionado com estrutura mais complexa, incluindo ciclos e múltiplos caminhos possíveis entre os vértices. Esse cenário teve como objetivo verificar a robustez da implementação em situações com maior complexidade. A Figura 1 ilustra visualmente esse grafo, bem como os pesos atribuídos às arestas.

Os caminhos mínimos esperados a partir do vértice C foram:

Figura 1 – Visualização do primeiro teste



Fonte: Elaborado pelo autor.

- C → B, com custo 3 (caminho direto).
- C → A, com custo 8, via C → B → A.
- C → D, com custo 5 (caminho direto).

A aplicação retornou corretamente os menores caminhos e custos, demonstrando que o algoritmo foi capaz de lidar adequadamente com a presença de ciclos, sem entrar em laços infinitos, e optando sempre pelas rotas mais eficientes. A resposta da API refletiu os valores esperados, comprovando a eficiência da solução proposta.

Teste 2 – Grafo simples com pesos distintos

O segundo teste avaliou o comportamento do algoritmo em um grafo de menor complexidade, composto por apenas três vértices (A, B e C) e três arestas direcionadas com pesos distintos. A estrutura foi projetada para verificar se a aplicação identifica corretamente o caminho de menor custo quando há múltiplas opções. A Figura 2 apresenta a visualização desse grafo.

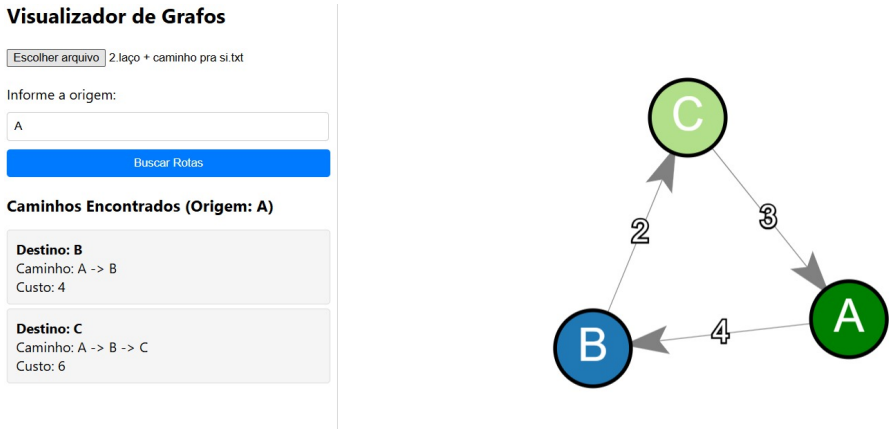
A configuração das arestas foi a seguinte:

- A → B com peso 4;
- B → C com peso 2;
- A → C com peso 3.

Com o vértice A definido como origem, os caminhos mínimos encontrados foram:

- Para o vértice B: caminho direto A → B, com custo 4;
- Para o vértice C: caminho direto A → C, com custo 3.

Figura 2 – Visualização do segundo teste



Fonte: Elaborado pelo autor.

A aplicação identificou corretamente os menores caminhos, optando pela aresta direta entre A e C em vez do trajeto A $\rightarrow$ B $\rightarrow$ C (que teria custo total 6). Isso demonstra que a lógica de escolha do menor caminho foi corretamente implementada e que o algoritmo respondeu de forma precisa ao cenário proposto.

Conclusão

Este trabalho teve como objetivo principal implementar e disponibilizar uma aplicação web capaz de calcular caminhos mínimos em grafos direcionados utilizando o algoritmo de Dijkstra. A proposta foi concretizada por meio do desenvolvimento de uma aplicação Web, que permite ao usuário carregar grafos a partir de arquivos de texto, definir o vértice de origem e visualizar os resultados.

A implementação demonstrou-se eficaz na resolução do problema proposto, sendo capaz de processar diferentes estruturas de grafos, inclusive com ciclos e laços, e retornar corretamente os menores caminhos e seus respectivos custos. Os testes realizados confirmaram a solução, bem como sua aplicabilidade em otimização de trajetos.

Além de contribuir para o entendimento prático do algoritmo de Dijkstra, a aplicação desenvolvida pode servir como base para futuras extensões, como o suporte a grafos com pesos negativos (por meio do algoritmo de Bellman-Ford), a adição de visualizações gráficas interativas dos caminhos e a persistência de dados em banco para histórico de análises.

Concluímos, portanto, que a aplicação atende de forma satisfatória aos objetivos propostos, oferecendo uma ferramenta didática, funcional e extensível para análise de grafos direcionados e estudo de algoritmos de caminhos mínimos.

Referências

CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. 3. ed. Rio de Janeiro: LTC, 2012. Nenhuma citação no texto.

DIJKSTRA, Edsger W. A note on two problems in connexion with graphs. *Numerische Mathematik*, v. 1, n. 1, p. 269–271, 1959. Disponível em: <https://ir.cwi.nl/pub/9256/9256D.pdf>. Acesso em: mai. 2025. Nenhuma citação no texto.

WEST, Douglas B. **Introdução à teoria dos grafos**. 2. ed. São Paulo: Prentice Hall, 2001. Nenhuma citação no texto.