ENTENDENDO O ALGORITMO DE DIJKSTRA TEORIA E APLICAÇÕES

Luiz Gustavo Lessa, Arthur Rocco, Rafael Melo, João Antônio Napoli
Ciência da Computação
Universidade Vila Velha (UVV) – Vila Velha, ES – Brasil

Resumo

Este artigo tem como objetivo apresentar o funcionamento do algoritmo de Dijkstra, amplamente utilizado para encontrar o caminho de menor custo em grafos direcionados e ponderados. Inicialmente, são abordados os conceitos teóricos que fundamentam o algoritmo, explicando sua lógica e aplicações. Em seguida, é apresentada uma implementação prática, na qual o usuário pode construir manualmente o grafo, definindo vértices e arestas com pesos. Após a definição do grafo e a escolha de um vértice inicial, o sistema calcula os menores caminhos até os demais vértices e exibe os custos e rotas correspondentes. Dessa forma, o artigo busca unir teoria e prática para proporcionar uma compreensão completa do algoritmo de Dijkstra.

1 Introdução

Figura 1: Edsger W. Dijkstra, criador do algoritmo de caminho mínimo que leva seu nome.

Fonte:
`https://upload.wikimedia.org/wikipedia/commons/d/d9/Edsger_Wybe_Dijkstra.jpg`

Em diversas áreas da ciência da computação e engenharia, a resolução do problema do caminho de menor custo em grafos é essencial para otimizar rotas, minimizar custos e melhorar a eficiência de sistemas complexos. Grafos são estruturas matemáticas que representam conjuntos de objetos (vértices) conectados por ligações (arestas), as quais podem possuir diferentes pesos associados, como distância, tempo ou custo. O algoritmo de Dijkstra, desenvolvido por Edsger W. Dijkstra em 1956, é uma das soluções mais conhecidas e utilizadas para encontrar o caminho mais curto em grafos direcionados e ponderados, desde que não possuam pesos negativos.

Este artigo tem como objetivo apresentar uma abordagem completa do algoritmo de Dijkstra, unindo a teoria que fundamenta seu funcionamento às etapas práticas de sua implementação. Para isso, será possível construir manualmente um grafo, definindo seus vértices e arestas com pesos, e, a partir de um vértice inicial, calcular as rotas de menor custo para os demais pontos do grafo. Dessa forma, busca-se facilitar a compreensão do algoritmo e evidenciar sua importância em aplicações reais, como sistemas de navegação, redes de comunicação e planejamento logístico.

2 Desenvolvimento

Nesta seção, abordamos os fundamentos teóricos e práticos que sustentam o funcionamento do algoritmo de Dijkstra. Inicialmente, são apresentados os conceitos de grafos, suas variações e representações. Em seguida, discutimos os princípios do algoritmo, suas estruturas de dados e, por fim, sua implementação em Python, integrando teoria e prática para facilitar a compreensão.

2.1 Grafos: conceitos básicos

Grafos são estruturas matemáticas compostas por um conjunto de vértices (ou nós) e um conjunto de arestas que conectam pares desses vértices. Eles são amplamente utilizados na ciência da computação para representar relações e conexões entre elementos.

Quando o grafo é *direcionado*, cada aresta possui um sentido, ou seja, a conexão entre dois vértices tem uma direção específica, como em uma via de mão única. Essa característica é essencial para aplicações como sistemas de navegação, fluxos de trabalho e redes sociais.

2.2 Grafos ponderados

Nos grafos ponderados, cada aresta possui um peso associado, que pode representar diversos fatores, como distância, tempo, custo ou capacidade. Esses pesos são cruciais para algoritmos que buscam otimizar rotas ou recursos, pois determinam quais caminhos são mais vantajosos em determinado contexto.

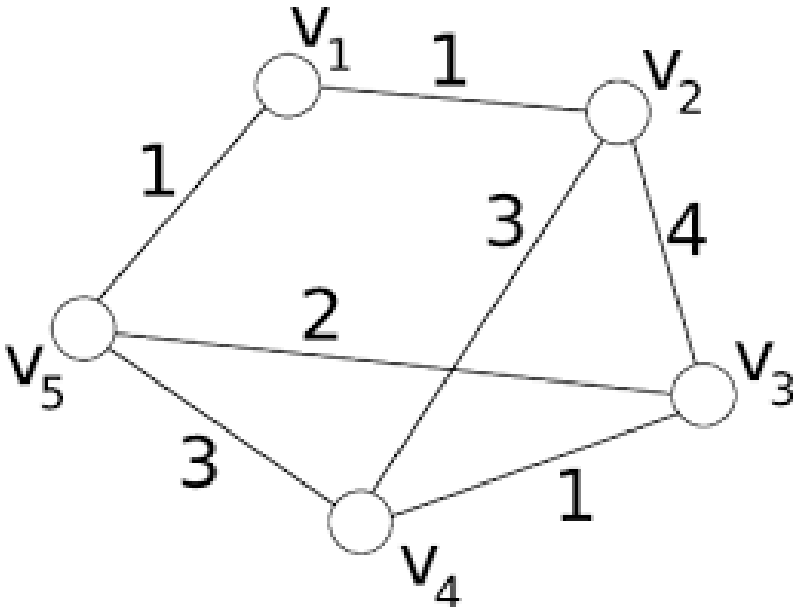

Figura 2: Exemplo de grafo ponderado com diferentes pesos nas arestas.

Fonte: `https://upload.wikimedia.org/wikipedia/commons/f/f0/Weighted_network.svg`

É justamente essa característica que permite ao algoritmo de Dijkstra calcular os

caminhos mais curtos com base em critérios definidos pelo peso das arestas.

2.3 Algoritmo de Dijkstra: princípios gerais

O algoritmo de Dijkstra é uma solução eficiente para encontrar o caminho de menor custo entre um vértice inicial e todos os demais vértices de um grafo direcionado e ponderado, desde que todas as arestas tenham pesos não negativos.

O funcionamento do algoritmo é iterativo. Ele parte do vértice inicial, calcula as menores distâncias provisórias para os demais vértices e atualiza essas distâncias à medida que encontra rotas mais curtas. Esse processo continua até que todas as rotas definitivas estejam determinadas.

2.4 Estruturas de dados utilizadas

A implementação do algoritmo depende de estruturas de dados eficientes para garantir bom desempenho. Em geral, são utilizadas listas para armazenar os vértices e arestas, além de um dicionário de distâncias e predecessores para reconstrução dos caminhos.

Embora versões otimizadas do algoritmo utilizem filas de prioridade (como `min-heaps`), nesta implementação optamos por uma abordagem didática, sem uso de bibliotecas externas, permitindo que os conceitos sejam explorados com maior clareza.

2.5 Implementação e Algoritmos

A implementação prática do algoritmo de Dijkstra foi desenvolvida na linguagem de programação `Python`, que oferece simplicidade e legibilidade, tornando o código acessível para fins educacionais.

A aplicação permite ao usuário criar manualmente um grafo direcionado e ponderado, adicionando vértices e arestas com pesos definidos. Após a construção do grafo, é possível selecionar um vértice de origem para que o sistema calcule os menores caminhos até os demais vértices, exibindo as rotas e os respectivos custos.

O algoritmo foi implementado de forma manual, sem recorrer a bibliotecas externas específicas para grafos, com exceção do `Tkinter`, utilizado exclusivamente para a criação da interface gráfica da aplicação. Essa escolha favorece a compreensão do funcionamento interno do algoritmo.

A seguir, apresentamos um trecho inicial do código da aplicação:

```
1 #   Implementar o algoritmo de Dijkstra
2 #       O usu rio deve poder criar o grafo direcionado e
    ponderado     Ap s a cria  o , o usu rio pode escolher um
    v rtice inicial e o algoritmo deve calcular o custo para
    todos os v rtices
3 #       Ap s calcular o custo, o algoritmo deve mostrar ao
    usu rio a rota que oferece o menor custo
4 #       N o Utilizar bibliotecas externas (a n o ser para
    UX/UI)
```

```
5
6 import tkinter as tk
7 from tkinter import simpledialog, messagebox, scrolledtext
8
9 class Grafo:
10     """
11     Classe que representa um grafo direcionado/ponderado.
12     - vertices: dicion rio {v rtice: [(vizinho, peso), ...]}
13     - posicoes: coordenadas (x,y) para desenhar cada v rtice no
          canvas
14     - direcionado: se True, arestas unilaterais; se False,
          bidirecionais
15     """
16     def __init__(self, direcionado=True):
17         # Inicializa estruturas de dados
18         self.vertices = {}     # mapeia cada v rtice    sua
              lista de arestas
19         self.posicoes = {}     # armazena coordenadas de
              exibi  o para cada v rtice
20         self.direcionado = direcionado  # tipo de grafo
21
22     def adicionar_vertice(self, v):
23         """Adiciona um v rtice v ao grafo, se ainda n  o
              existir."""
24         if v not in self.vertices:
25             self.vertices[v] = []
26
27     def adicionar_aresta(self, origem, destino, peso):
28         """
29         Cria uma aresta de  o r i g e m destino  com determinado peso.
30         L a n  a KeyError se algum v rtice  n  o existir.
31         Em grafo  n  o -direcionado, adiciona rec proca
              d e s t i n o origem .
32         """
33         if origem not in self.vertices or destino not in
              self.vertices:
34             raise KeyError("V rtice  n  o cadastrado.")
35         # adiciona aresta principal
36         self.vertices[origem].append((destino, peso))
37         # se  n  o direcionado, adiciona aresta de volta
38         if not self.direcionado:
39             self.vertices[destino].append((origem, peso))
40
41     def remover_aresta(self, origem, destino, peso):
42         """
43         Desfaz a  ltima  aresta adicionada removendo uma
              ocorr ncia
44         de (destino,peso) em origem (e rec proca, se
              apropriadamente bidirecional).
45         """
46         if origem in self.vertices:
```

```
47              try:
48                  self.vertices[origem].remove((destino, peso))
49              except ValueError:
50                  pass
51          if not self.direcionado and destino in self.vertices:
52              try:
53                  self.vertices[destino].remove((origem, peso))
54              except ValueError:
55                  pass
56
57      def limpar(self):
58          """Remove todos os v rtices e arestas do grafo."""
59          self.vertices.clear()
60          self.posicoes.clear()
61
62      def dijkstra(self, inicio):
63          """
64          Implementa  o do algoritmo de Dijkstra sem uso de
               heapq:
65          - dist: mapeia v rtice     dist ncia m nima desde
               in cio
66          - prev: armazena antecessor para reconstruir caminho
67          - visitados: conjunto de v rtices j  processados
68          """
69          # inicializa  o das dist ncias
70          dist = {v: float('inf') for v in self.vertices}
71          prev = {v: None for v in self.vertices}
72          dist[inicio] = 0
73          visitados = set()
74
75          # enquanto houver v rtices n o visitados
76          while len(visitados) < len(self.vertices):
77              # escolhe v rtice n o visitado com menor dist[v]
78              u = None
79              menor = float('inf')
80              for v in self.vertices:
81                  if v not in visitados and dist[v] < menor:
82                      menor = dist[v]
83                      u = v
84              # se n o encontrou v rtice alcan vel, encerra
85              if u is None:
86                  break
87              visitados.add(u)
88
89              # relaxa arestas saindo de u
90              for (viz, peso) in self.vertices[u]:
91                  if viz in visitados:
92                      continue
93                  nova = dist[u] + peso
94                  if nova < dist[viz]:
95                      dist[viz] = nova
```

```
96                      prev[viz] = u
97
98          return dist, prev
99
100     def obter_caminhos(self, inicio):
101         """
102         Reconstr i os caminhos a partir de 'inicio' at  cada
                v rtice.
103         Retorna lista de strings descrevendo rotas e custos.
104         """
105         dist, prev = self.dijkstra(inicio)
106         resultados = []
107         for dest in self.vertices:
108             if dest == inicio:
109                 continue  # ignora rota at  si mesmo
110             # se infinita, n o h  caminho
111             if dist[dest] == float('inf'):
112                 resultados.append(f"N o h  caminho de {inicio}
                        para {dest}.")
113             else:
114                 # reconstr i sequ ncia de v rtices do destino
                        ao in cio
115                 seq = []
116                 u = dest
117                 while u is not None:
118                     seq.insert(0, u)
119                     u = prev[u]
120                 resultados.append(
121                     f"Caminho {inicio}  {dest}:
                            {'  '.join(seq)} (custo {dist[dest]:.0f})"
122                 )
123         return resultados
124
125 class Interface:
126     """
127     Interface gr fica usando Tkinter para manipular e
            visualizar o grafo.
128     - Bot es para adicionar/remover arestas, limpar grafo,
            executar Dijkstra.
129     - Canvas interativo: arraste n s e veja atualiza  o em
            tempo real.
130     """
131     def __init__(self, master):
132         self.master = master
133         master.title("Dijkstra Interativo - Sem heapq")
134
135         # Grafo e hist rico de arestas para desfazer
136         self.grafo = Grafo(direcionado=True)
137         self.history = []  # pilha de tuplas (origem, destino,
                peso)
138         self.dragging = None  # v rtice atual sendo movido
```

```
139
140         # Flags de configura  o (direcionado, ponderado)
141         self.var_direc = tk.BooleanVar(value=True)
142         self.var_pond = tk.BooleanVar(value=True)
143
144         # Painel de controle    esquerda
145         ctrl = tk.Frame(master)
146         ctrl.pack(side=tk.LEFT, fill=tk.Y, padx=5, pady=5)
147         # Checkbox para alternar tipo de grafo
148         tk.Checkbutton(ctrl, text="Direcionado",
                variable=self.var_direc,
149                        command=self.toggle_direc).pack(anchor='w')
150         tk.Checkbutton(ctrl, text="Ponderado",
                variable=self.var_pond).pack(anchor='w')
151         # Bot es de a   o
152         tk.Button(ctrl, text="Add Aresta",
                command=self.add_aresta).pack(fill=tk.X, pady=2)
153         tk.Button(ctrl, text="Desfazer  ltima  Aresta",
                command=self.undo_aresta).pack(fill=tk.X, pady=2)
154         tk.Button(ctrl, text="Limpar Grafo",
                command=self.clear_graph).pack(fill=tk.X, pady=2)
155         tk.Button(ctrl, text="Dijkstra",
                command=self.run).pack(fill=tk.X, pady=2)
156         tk.Button(ctrl, text="Vertices",
                command=self.show_verts).pack(fill=tk.X, pady=2)
157         #  rea  de texto para logs e resultados
158         self.text = scrolledtext.ScrolledText(ctrl, width=30,
                height=20)
159         self.text.pack(pady=5)
160
161         # Canvas para desenho do grafo    direita
162         self.canvas = tk.Canvas(master, width=600, height=600,
                bg='white')
163         self.canvas.pack(side=tk.RIGHT, padx=5, pady=5)
164         self.radius = 20  # raio dos n s
165         # Eventos de mouse para arrastar n s
166         self.canvas.bind('<Button-1>', self.on_click)
167         self.canvas.bind('<B1-Motion>', self.on_drag)
168         self.canvas.bind('<ButtonRelease-1>', self.on_release)
169
170     def toggle_direc(self):
171         """Ativa/desativa grafo direcionado e redesenha."""
172         self.grafo.direcionado = self.var_direc.get()
173         self.redraw()
174
175     def layout(self):
176         """
177         Posiciona automaticamente novos n s em grade se ainda
                n o tiverem coordenadas.
178         Distribui  o em filas e colunas baseada na raiz
                quadrada do n mero de v rtices.
```

```
        """
        n = len(self.grafo.vertices)
        if n == 0:
            return
        cols = int(n**0.5) + 1
        spacing_x = 600 / cols
        rows = (n // cols) + 1
        spacing_y = 600 / rows
        for idx, v in enumerate(self.grafo.vertices):
            if v not in self.grafo.posicoes:
                col = idx % cols
                row = idx // cols
                x = (col + 0.5) * spacing_x
                y = (row + 0.5) * spacing_y
                self.grafo.posicoes[v] = (x, y)

    def redraw(self):
        """Limpa e redesenha todas arestas e n s no canvas."""
        self.canvas.delete('all')
        # Desenha arestas com setas e pesos
        for u, adj in self.grafo.vertices.items():
            x1, y1 = self.grafo.posicoes[u]
            for v, p in adj:
                x2, y2 = self.grafo.posicoes[v]
                # Calcula dire  o unit ria para desenhar
                   linha entre per metros
                dx, dy = x2 - x1, y2 - y1
                dist = (dx*dx + dy*dy)**0.5 or 1
                ux, uy = dx/dist, dy/dist
                start = (x1 + ux*self.radius, y1 +
                   uy*self.radius)
                end   = (x2 - ux*self.radius, y2 -
                   uy*self.radius)
                # seta opcional para grafo direcionado
                if self.var_direc.get():
                    self.canvas.create_line(*start, *end,
                                            arrow=tk.LAST,
                                            arrowshape=(10,12,4),
                                            width=2)
                else:
                    self.canvas.create_line(*start, *end,
                       width=2)
                # desenha peso, se habilitado
                if self.var_pond.get():
                    mx, my = (start[0]+end[0])/2,
                       (start[1]+end[1])/2
                    self.canvas.create_text(mx, my-10,
                       text=str(p))
        # Desenha n s (c rculos com r tulo)
        for v in self.grafo.vertices:
            x, y = self.grafo.posicoes[v]
```

```
224             self.canvas.create_oval(x-self.radius, y-self.radius,
225                                     x+self.radius, y+self.radius,
226                                     fill='#eef', outline='#44a',
                                           tags=('node', v))
227             self.canvas.create_text(x, y, text=str(v),
                   tags=('node', v))
228
229     def on_click(self, event):
230         """Detecta clique sobre n  para iniciar arraste."""
231         closest = self.canvas.find_closest(event.x, event.y)
232         for tag in self.canvas.gettags(closest):
233             if tag in self.grafo.vertices:
234                 self.dragging = tag
235                 break
236
237     def on_drag(self, event):
238         """Atualiza posi  o do n  enquanto arrasta e
               redesenha."""
239         if self.dragging:
240             self.grafo.posicoes[self.dragging] = (event.x,
                   event.y)
241             self.redraw()
242
243     def on_release(self, event):
244         """Encerra arraste ao soltar o bot o."""
245         self.dragging = None
246
247     def add_aresta(self):
248         """
249         Dialog para inserir origem, destino e peso. Adiciona a
               aresta ao grafo,
250         registra em history para undo e atualiza layout/desenho.
251         """
252         o = simpledialog.askstring('Origem', 'Origem:')
253         d = simpledialog.askstring('Destino', 'Destino:')
254         if not o or not d:
255             return
256         p = 1
257         if self.var_pond.get():
258             val = simpledialog.askfloat('Peso', 'Peso:')
259             if val is None:
260                 return
261             p = val
262         for v in (o, d):
263             self.grafo.adicionar_vertice(v)
264         self.grafo.adicionar_aresta(o, d, p)
265         self.history.append((o, d, p))
266         self.text.insert(tk.END, f'{o}   {d} (peso {p})\n')
267         self.layout()
268         self.redraw()
269
```

```
270     def undo_aresta(self):
271         """Remove a  ltima  aresta adicionada, atualizando grafo
                e canvas."""
272         if not self.history:
273             messagebox.showinfo('Info', 'Nenhuma aresta para
                    desfazer.')
274             return
275         o, d, p = self.history.pop()
276         self.grafo.remover_aresta(o, d, p)
277         self.text.insert(tk.END, f'Desfeito: {o}   {d} (peso
                {p})\n')
278         self.redraw()
279
280     def clear_graph(self):
281         """Limpa completamente o grafo, hist rico e canvas."""
282         self.grafo.limpar()
283         self.history.clear()
284         self.text.delete('1.0', tk.END)
285         self.canvas.delete('all')
286
287     def run(self):
288         """
289         Solicita v rtice inicial, executa Dijkstra e imprime
290         todos os caminhos e custos no painel de texto.
291         """
292         start = simpledialog.askstring('Start', 'V rtice
                inicial:')
293         if not start or start not in self.grafo.vertices:
294             return
295         self.text.insert(tk.END, '\n--- Dijkstra ---\n')
296         for line in self.grafo.obter_caminhos(start):
297             self.text.insert(tk.END, line + '\n')
298
299     def show_verts(self):
300         """Exibe todos os v rtices cadastrados no painel de
                texto."""
301         verts = ','.join(self.grafo.vertices.keys())
302         self.text.insert(tk.END, 'Vertices: ' + verts + '\n')
303
304 if __name__ == '__main__':
305     root = tk.Tk()
306     app = Interface(root)
307     root.mainloop()
308
309
310
311 """
312 Exemplos:
313
314 Exemplo Simples:
315
```

```
A      B (peso 2)
B      C (peso 9)
A      C (peso 5)
C      D (peso 1)


Grafo com ciclo e v rtice isolado

A      B (peso 4)
A      C (peso 2)
B      C (peso 5)
B      D (peso 10)
C      E (peso 3)
E      D (peso 4)
D      C (peso 1)    # ciclo de retorno
F                  # v rtice isolado (crie com Origem=F,
   Destino=F, Peso=0 mas depois ignore)


Grafo dirigido parcialmente desconectado

1      2 (1)
1      3 (4)
2      3 (2)
2      4 (7)
3      5 (3)
5      4 (2)
6      7 (5)    # componente separado
7      2 (1)
4      1 (6)    # volta para 1, formando ciclo


Grafo     maluco     com pesos zero e m ltiplas rotas

S      A (0)
S      B (2)
A      C (1)
B      C (1)
C      D (3)
A      D (7)
B      D (4)
D      E (0)
E      F (5)
F      D (2)    # ciclo de peso positivo


S   direcionado, sem pesos (ponderado desmarcado)

A      B
B      C
C      D
```

```
D      B    # cria um ciclo     B    CDB
E      F    # componente separado


 S   ponderado, sem dire   o (direcionado desmarcado)
1      2 (3)
2      3 (5)
3      4 (2)
1      4 (10)
4      5 (1)

"""
```

2.6 Implementação da Estrutura de Dados de Grafo

A estrutura central da aplicação é a classe `Grafo`, que modela um grafo direcionado e ponderado. Esta implementação foi desenvolvida com foco em simplicidade, flexibilidade e integração direta com componentes gráficos para visualização.

2.6.1 Visão Geral

A classe `Grafo` representa um grafo orientado e ponderado por meio de dois dicionários principais:

- **vertices:** Armazena as conexões entre vértices, onde cada chave é um vértice e o valor associado é uma lista de tuplas (vizinho, peso).
- **posicoes:** Registra as coordenadas (x, y) de cada vértice para fins de exibição gráfica no canvas da interface.

Adicionalmente, um atributo booleano `direcionado` define o comportamento padrão das arestas: se `True`, o grafo é direcionado; caso contrário, trata-se de um grafo não-direcionado (isto é, com arestas bidirecionais).

2.6.2 Inicialização da Estrutura

```
def __init__(self, direcionado=True):
    self.vertices = {}
    self.posicoes = {}
    self.direcionado = direcionado
```

Na inicialização da classe, os dois dicionários são instanciados como vazios. O parâmetro `direcionado` é definido como `True` por padrão, o que torna o grafo direcionado. Essa flexibilidade permite reutilizar a mesma estrutura em diferentes contextos, como grafos de tráfego (direcionados) ou mapas não-orientados (não-direcionados).

2.6.3 Inserção de Vértices

```
def adicionar_vertice(self, v):
    if v not in self.vertices:
        self.vertices[v] = []
```

A adição de um novo vértice ao grafo verifica se o mesmo já existe. Caso contrário, é criado com uma lista vazia de arestas. Essa operação tem complexidade $O(1)$, considerando acesso a dicionários como constante em tempo médio.

2.6.4 Inserção de Arestas Ponderadas

```
def adicionar_aresta(self, origem, destino, peso):
    if origem not in self.vertices or destino not in
       self.vertices:
        raise KeyError("V rtice  n  o  cadastrado.")
    self.vertices[origem].append((destino, peso))
    if not self.direcionado:
        self.vertices[destino].append((origem, peso))
```

Esse método cria uma aresta com peso entre dois vértices. Ele lança uma exceção (`KeyError`) caso algum dos vértices ainda não tenha sido registrado no grafo. Isso força o uso disciplinado da estrutura.

No caso de grafos não-direcionados, o método adiciona uma segunda aresta de retorno. Isso significa que a estrutura representa arestas bidirecionais por duplicação explícita nas duas direções, o que facilita o uso de algoritmos de busca e caminhos mínimos que requerem iteração direta sobre os vizinhos.

2.6.5 Remoção de Arestas

```
def remover_aresta(self, origem, destino, peso):
    if origem in self.vertices:
        try:
            self.vertices[origem].remove((destino, peso))
        except ValueError:
            pass
    if not self.direcionado and destino in self.vertices:
        try:
            self.vertices[destino].remove((origem, peso))
        except ValueError:
            pass
```

Este método remove uma única ocorrência da aresta entre dois vértices, caso ela exista. A operação é tolerante a inconsistências: se a aresta não for encontrada, nenhuma exceção será lançada.

Em grafos não-direcionados, a remoção é simétrica — a aresta recíproca também é eliminada, garantindo que o grafo permaneça consistente.

2.6.6 Reset da Estrutura

```
def limpar(self):
    self.vertices.clear()
    self.posicoes.clear()
```

O método `limpar` reinicializa completamente a estrutura, apagando todos os vértices, arestas e posições. Isso é útil, por exemplo, ao reiniciar a visualização ou trocar de cenário na interface gráfica.

2.6.7 Considerações sobre a Eficiência

Todas as operações básicas (`adicionar_vertice`, `adicionar_aresta`, `remover_aresta`) são eficientes em tempo médio, com complexidade $O(1)$ para inserção/remoção em dicionários e listas curtas.

A escolha por listas de tuplas como representação de arestas torna a leitura e visualização mais simples, mas pode ter limitações de performance em grafos extremamente densos.

2.6.8 Integração com Interface Gráfica

Embora esta seção foque na lógica do grafo, vale destacar que a estrutura foi pensada para integração direta com visualização interativa. O dicionário `posicoes` permite associar cada vértice a uma coordenada, facilitando o desenho e manipulação no canvas `Tkinter` da interface.

2.7 Algoritmo de Dijkstra e Reconstrução de Caminhos

A classe `Grafo` implementa uma versão clássica do algoritmo de Dijkstra para encontrar os caminhos mais curtos a partir de um vértice de origem. Esta seção detalha a implementação, as decisões de projeto e os métodos auxiliares.

2.7.1 Implementação do Algoritmo de Dijkstra

```
def dijkstra(self, inicio):
    ...
```

Este método executa o algoritmo de Dijkstra sem uso de estruturas de prioridade como `heapq`, o que o torna mais simples de entender e adequado para grafos de tamanho pequeno a médio, típicos em contextos educacionais ou aplicações visuais interativas.

2.7.2 Inicialização

```
dist = {v: float('inf') for v in self.vertices}
prev = {v: None for v in self.vertices}
dist[inicio] = 0
```

```
visitados = set()
```

A distância de cada vértice é inicialmente infinita, exceto para o vértice inicial, que é definido com distância zero. O dicionário `prev` é utilizado para rastrear os antecessores de cada vértice no caminho mais curto, e o conjunto `visitados` impede o reprocessamento de vértices.

2.7.3 Seleção do Vértice com Menor Distância

```
u = None
menor = float('inf')
for v in self.vertices:
    if v not in visitados and dist[v] < menor:
        menor = dist[v]
        u = v
```

A cada iteração, o algoritmo busca o vértice não visitado com menor valor de `dist[v]`. Essa abordagem tem custo $O(V)$ por iteração, o que resulta em complexidade total $O(V^2)$, adequada para grafos pequenos.

2.7.4 Relaxamento das Arestas

```
for (viz, peso) in self.vertices[u]:
    if viz in visitados:
        continue
    nova = dist[u] + peso
    if nova < dist[viz]:
        dist[viz] = nova
        prev[viz] = u
```

O passo de relaxamento atualiza a menor distância conhecida até os vizinhos de `u`. Se uma nova distância menor é encontrada, os valores nos dicionários `dist` e `prev` são atualizados.

2.7.5 Finalização

O algoritmo termina quando todos os vértices forem visitados ou quando não houver mais vértices alcançáveis a partir do inicial. O método retorna os dicionários `dist` e `prev`, que serão usados para reconstrução de caminhos.

2.7.6 Reconstrução dos Caminhos Mínimos

```
def obter_caminhos(self, inicio):
    ...
```

Este método utiliza o resultado de `dijkstra()` para reconstruir e apresentar os caminhos mínimos a partir de um vértice inicial. A saída é uma lista de strings legíveis para o usuário, indicando o percurso e o custo.

- **Tratamento de Caminhos Inexistentes:**

```
if dist[dest] == float('inf'):
    resultados.append(f"N o h caminho de {inicio} para
       {dest}.")
```

Quando a distância até um vértice permanece infinita, isso indica que ele não é alcançável a partir do vértice inicial.

- **Reconstrução do Caminho:**

```
seq = []
u = dest
while u is not None:
    seq.insert(0, u)
    u = prev[u]
```

O caminho é reconstruído em ordem reversa, partindo do destino e seguindo os antecessores até o vértice inicial.

- **Apresentação Final:**

```
resultados.append(
    f"Caminho {inicio} {dest}: {' '.join(seq)} (custo
       {dist[dest]:.0f})"
)
```

Cada caminho é formatado como uma string contendo a rota e seu custo total.

2.7.7 Considerações de Desempenho

Embora a ausência de uma fila de prioridade torne a implementação menos eficiente para grafos grandes (complexidade $O(V^2)$), a simplicidade do código facilita a depuração, o ensino e a integração com interfaces gráficas. Para grafos pequenos ou moderados (até algumas centenas de vértices), o impacto é geralmente aceitável.

2.7.8 Aplicação Prática

Este método é integrado à interface do usuário da aplicação, permitindo que os usuários selecionem um vértice inicial e visualizem, de forma textual ou gráfica, os caminhos mais curtos até os demais vértices. Isso é útil em simulações, jogos, redes urbanas e contextos educacionais.

2.8 Interface Gráfica Interativa com Tkinter

A classe `Interface` representa a camada de interação visual do sistema, permitindo ao usuário criar, modificar e visualizar grafos, além de executar o algoritmo de Dijkstra com feedback visual e textual. Essa interface é construída com a biblioteca `Tkinter`, padrão do Python para aplicações GUI.

2.8.1 Estrutura Geral da Interface

A janela é dividida em duas partes principais:

- **Painel de Controle (esquerda):** contém botões, checkboxes e uma área de texto para exibição de logs e resultados.
- **Canvas Gráfico (direita):** área de desenho onde os nós e arestas são visualizados e manipulados.

O construtor da interface realiza a criação desses componentes e a associação dos eventos de clique, arraste e soltar.

2.8.2 Componentes Interativos

O painel de controle contém:

- Checkbox **"Direcionado"**: ativa ou desativa o comportamento direcionado das arestas.
- Checkbox **"Ponderado"**: controla a exibição dos pesos nas arestas.
- Botão **"Add Aresta"**: inicia o processo de adição de uma nova aresta.
- Botão **"Desfazer Última Aresta"**: remove a última aresta adicionada.
- Botão **"Limpar Grafo"**: reseta completamente o grafo.
- Botão **"Dijkstra"**: executa o algoritmo de Dijkstra a partir de um vértice escolhido.
- Botão **"Vértices"**: exibe os vértices existentes na área de texto.
- **Área de Texto com Scroll**: utilizada para mostrar o resultado dos caminhos, mensagens ou logs internos.

2.8.3 Posicionamento Automático

```
1 def layout(self):
2     ...
```

O método `layout` distribui automaticamente os vértices recém-criados em um grid espacial no canvas, com base na raiz quadrada do número total de vértices. Essa abordagem garante boa distribuição e legibilidade visual sem sobreposição.

2.8.4 Redesenho Completo

```
1 def redraw(self):
2     ...
```

O método `redraw` limpa o canvas e redesenha todas as arestas e vértices com base no dicionário de posições. Ele também leva em conta:

- Arestas com direção (setas), se `var_direc` estiver ativado.
- Exibição de pesos, se `var_pond` estiver ativado.
- Cálculo vetorial para desenhar linhas entre os perímetros dos nós.
- Representação de cada nó como um círculo com rótulo.

2.8.5 Eventos de Mouse

A interface suporta os seguintes eventos:

- `on_click`: identifica se um vértice foi selecionado para arraste.
- `on_drag`: atualiza dinamicamente a posição do vértice em tempo real.
- `on_release`: finaliza o movimento e força o redesenho.

Esse conjunto de interações permite ao usuário reposicionar nós livremente, facilitando visualizações personalizadas e compreensíveis.

2.8.6 Integração com a Lógica do Grafo

A classe `Interface` se comunica diretamente com a instância `Grafo`, executando métodos como:

- `add_aresta()`
- `clear_graph()`
- `run()` – executa `grafo.obter_caminhos()` e imprime no painel de texto.
- `toggle_direc()` – alterna o modo direcionado e força redesenho.

Além disso, ela mantém um histórico de arestas (`self.history`) para suportar a funcionalidade de desfazer.

3 Continuação da Classe Interface

A continuação da classe `Interface` inclui os métodos de interação com o canvas, diálogos para entrada de dados e os mecanismos para execução e visualização do algoritmo de Dijkstra. Esses métodos garantem a dinamicidade e usabilidade da aplicação.

3.1 Manipulação de Nós com o Mouse

3.1.1 Clique e Arraste

- `on_click(event)`: identifica se o clique ocorreu sobre um nó existente, ativando o estado de "arraste".
- `on_drag(event)`: atualiza dinamicamente a posição do nó arrastado, refletindo em tempo real na interface.
- `on_release(event)`: encerra o movimento do nó.

Esse conjunto de eventos permite ao usuário reposicionar os nós livremente, ajustando o layout visual conforme preferir.

3.2 Adição de Arestas

3.2.1 `add_aresta()`

Abre diálogos para entrada de origem, destino e, se ativado, peso da aresta. Após isso:

- Adiciona os vértices ao grafo (se ainda não existirem).
- Cria a aresta com ou sem peso.
- Atualiza o histórico para possível desfazer.
- Reorganiza o layout automaticamente.
- Redesenha o canvas.

Esse método é a principal via de construção interativa do grafo.

3.3 Gerenciamento do Grafo

- `undo_aresta()`: desfaz a última aresta inserida, com base em uma pilha (`self.history`). Ideal para corrigir erros de inserção.
- `clear_graph()`: limpa completamente o grafo, o histórico e o canvas.
- `show_verts()`: lista todos os vértices no painel de texto.

Essas ações oferecem ao usuário controle completo sobre o estado atual do grafo.

3.4 Execução do Algoritmo de Dijkstra

3.4.1 `run()`

Solicita o vértice de origem via diálogo e, caso válido:

- Executa `grafo.obter_caminhos(inicio)`, que utiliza o algoritmo de Dijkstra sem `heapq`.
- Exibe no painel de texto os caminhos mínimos e seus respectivos custos de forma descritiva.

Essa funcionalidade transforma a interface em uma poderosa ferramenta educacional e de análise de grafos.

3.5 Execução da Interface

O script finaliza com:

```
if __name__ == '__main__':
    root = tk.Tk()
    app = Interface(root)
    root.mainloop()
```

Esse trecho executa a aplicação, instanciando a interface Tkinter e iniciando o loop de eventos da GUI.

4 Exemplos de Uso

A seção final fornece exemplos práticos que podem ser inseridos manualmente na interface para explorar diferentes cenários:

- Grafos simples
- Grafos com ciclos
- Componentes desconexos
- Grafos com múltiplos caminhos
- Grafos não direcionados, com ou sem pesos

Figura 3: Edsger W. Dijkstra lecionando, compartilhando seu conhecimento com alunos em sala de aula.

Fonte:
`https://upload.wikimedia.org/wikipedia/commons/d/d9/Edsger_Wybe_Dijkstra.jpg`

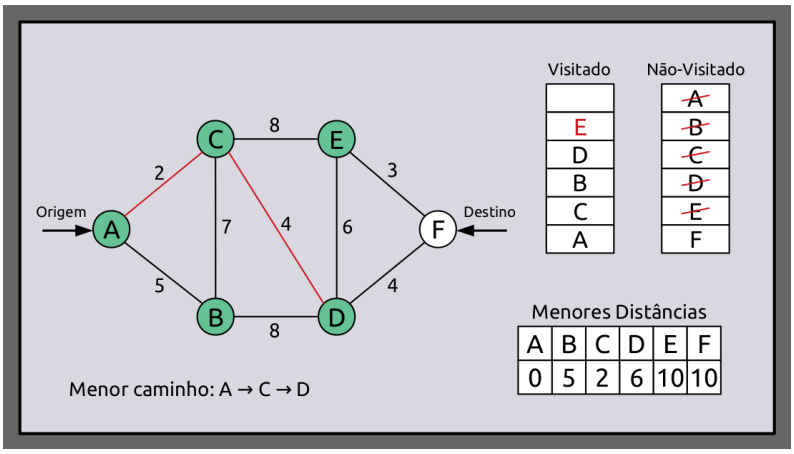

Figura 4: Exemplo visual do algoritmo de Dijkstra aplicado a um grafo com pesos.

Fonte: `https://raw.githubusercontent.com/the-akira/PythonExperimentos/master/Algoritmos/Dijkstra/Imagens/Grafo7.png`

Esses exemplos são essenciais para testes e demonstrações pedagógicas do algoritmo.

5 Bibliografia

1. Elemar Júnior. (s.d.). *Algoritmo de Dijkstra: entendendo o caminho mínimo em grafos ponderados*. Recuperado de `https://elemarjr.com/clube-de-estudos/artigos/algoritmo-de-dijkstra-entendendo-o-caminho-minimo-em-grafos-ponderados/`

2. DataCamp. (s.d.). *Dijkstra Algorithm in Python*. Recuperado de `https://www.datacamp.com/pt/tutorial/dijkstra-algorithm-in-python`

3. GeeksforGeeks. (s.d.). *Dijkstra's Shortest Path Algorithm – Greedy Algo-7*. Recuperado de `https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-alg`