Algoritmos e Estruturas de Dados II

Capítulo das Aulas 20, 21 e 22: Algoritmo de Dijkstra

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

Parte 1: A Mecânica do Algoritmo (Aula 20)

1. O GPS dos Grafos

Se você abre o Google Maps e pede uma rota, ele usa uma variante do algoritmo de **Edsger W. Dijkstra** (1956). Diferente da BFS (que só serve para arestas de peso 1), o Dijkstra encontra o Caminho Mínimo em grafos onde as arestas têm pesos variados **não-negativos**.

Analogia da Água: Imagine que a água começa a fluir da origem. Ela percorre canos (arestas) de comprimentos diferentes. O algoritmo determina em que "momento" a água chega em cada cidade.

Problema Formal

Dado um grafo ponderado $G = (V, E)$ com função de peso $w : E \to \mathbb{R}_{\geq 0}$ e um vértice fonte $s \in V$, encontrar para cada vértice $v \in V$ o menor custo de s até v:

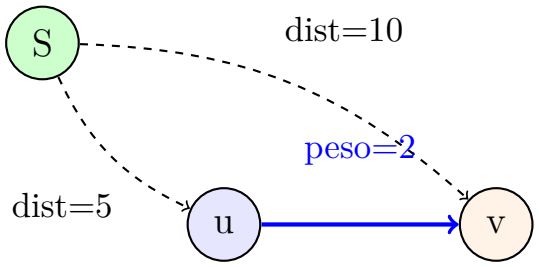
$$\delta(s, v) = \min \left\{ \sum_{e \in P} w(e) \;\middle|\; P \text{ é caminho de } s \text{ a } v \right\}$$

Este é o problema **SSSP** (Single-Source Shortest Path).

2. Conceito de Relaxamento

A base de todos os algoritmos de menor caminho é a operação de **Relaxamento** (Relaxation). Seja `D[v]` a melhor distância conhecida até agora da origem até `v`.

Imagine que conhecemos um caminho para v que custa 10. Descobrimos uma aresta $u \to v$ com peso 2, e sabemos que chegamos em u com custo 5. Novo custo potencial via u: $D[u] + peso(u, v) = 5 + 2 = 7$. Como $7 < 10$, nós **relaxamos** a aresta: atualizamos `D[v] = 7` e dizemos que o pai de v agora é u.



Novo custo via u: $5 + 2 = 7 < 10$
$\Rightarrow$ **Relaxou!** dist[v]=7

Figure 1: Relaxamento: se dist[u] + peso(u,v) < dist[v], atualizar dist[v] e pai[v].

• Teoria

O relaxamento é a operação atômica compartilhada por Dijkstra, Bellman-Ford e outros algoritmos de caminho mínimo. A diferença entre eles está na **ordem** e **frequência** com que as arestas são relaxadas.

3. O Algoritmo Guloso

1. **Inicialização:** `dist[S] = 0`, todos os outros `dist[v] =` ∞. 2. **Priority Queue:** Coloque par `(0, S)` na fila. 3. Enquanto a PQ não estiver vazia:

- Extraia o vértice u com a **menor** distância atual.
- Se essa distância for maior que `dist[u]` (info obsoleta), ignore (**Lazy Deletion**).
- Para cada vizinho v de u, tente **relaxar** a aresta (u, v).
- Se relaxar, empurre `(novo_dist, v)` para a PQ.

Por que Guloso?

O algoritmo assume que, se extraímos u da fila com distância X, **não existe** nenhum outro caminho no mundo que chegue em u com custo menor que X. Isso é verdade porque arestas negativas não existem — qualquer extensão de caminho só pode **aumentar** o custo.

Lazy Deletion

Na implementação com `PriorityQueue`, quando relaxamos uma aresta e melhoramos `dist[v]`, não removemos a entrada antiga de v da fila (isso seria $O(V)$). Em vez disso, inserimos uma nova entrada com a distância atualizada. Ao extrair, verificamos se a distância na fila é obsoleta — se for, descartamos. Isso é chamado de **Lazy Deletion**.

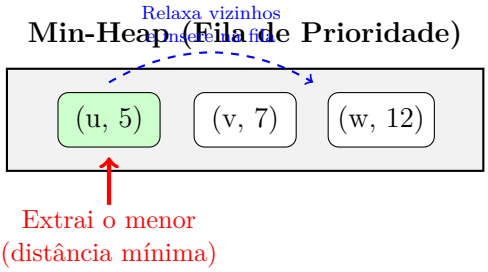


Figure 2: Fila de prioridade: sempre extrair o vértice com menor distância atual.

Parte 2: Implementação e Casos Especiais (Aula 21)

4. Implementação em Java

Atenção: A `PriorityQueue` do Java guarda objetos. Devemos criar uma classe ou usar array.

```
import java.util.*;

class Node implements Comparable<Node> {
    int id;
    int distance;

    public Node(int id, int dist) {
        this.id = id;
        this.distance = dist;
    }

    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.distance, other.distance);
    }
}

public class Dijkstra {

    public static int[] dijkstra(int V, List<List<Edge>> adj, int startNode
        int[] dist = new int[V];
        int[] pai = new int[V];
```

```
        Arrays.fill(dist, Integer.MAX_VALUE);
        Arrays.fill(pai, -1);
        dist[startNode] = 0;

        PriorityQueue<Node> pq = new PriorityQueue<>();
        pq.add(new Node(startNode, 0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
            int u = current.id;
            int d = current.distance;

            // Lazy Deletion: Se j'a achamos um caminho melhor antes, esse '
lixo
            if (d > dist[u]) continue;

            // Explora vizinhos
            for (Edge e : adj.get(u)) {
                int v = e.destino;
                int peso = e.peso;

                // Relaxamento
                if (dist[u] + peso < dist[v]) {
                    dist[v] = dist[u] + peso;
                    pai[v] = u;
                    pq.add(new Node(v, dist[v]));
                }
            }
        }
        return dist;
    }
}
```

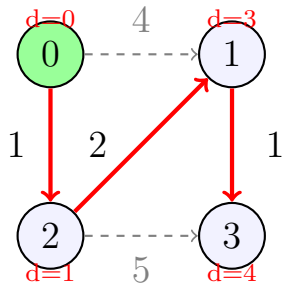


Figure 3: Caminho mínimo da origem a cada vértice; setas vermelhas representam os ponteiros de 'pai'.

5. Reconstrução do Caminho

O vetor `pai[]` permite reconstruir o caminho ótimo de qualquer vértice até a origem:

```
public static List<Integer> reconstruirCaminho(int[] pai, int destino) {
    List<Integer> caminho = new ArrayList<>();
    for (int v = destino; v != -1; v = pai[v])
        caminho.add(v);
    Collections.reverse(caminho);
    return caminho;
}
```

Se `pai[destino] == -1` e `destino` $\neq$ `src`, o destino é **inalcançável**.

6. Limitações e Complexidade

O Problema dos Pesos Negativos

Se houver uma aresta com peso -10, o critério guloso falha. Podemos "voltar no tempo" e achar um caminho melhor depois de ter fechado um vértice. Pior ainda, se houver um **Ciclo Negativo**, podemos ficar girando nele e diminuindo o custo infinitamente ($-\infty$). Para pesos negativos, use **Bellman-Ford**.

Complexidade

- Com **Binary Heap** (PriorityQueue padrão): $O(E \log V)$.
- Com **Fibonacci Heap**: $O(E + V \log V)$ (Teórico, complexo demais para prática).
- Array Simples: $O(V^2)$ (Bom apenas para grafos **muito densos**).

△ Importante

Cuidado com overflow! Usar `Integer.MAX_VALUE` como infinito pode causar overflow ao somar `dist[u] + peso`. Use um valor grande como `999999999` ou verifique antes de somar.

Aplicações

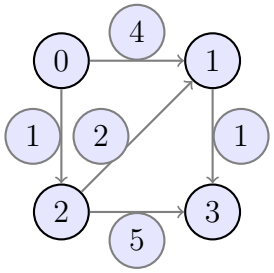
- **Roteamento OSPF (Internet):** O protocolo OSPF calcula a árvore de rotas mais curtas usando Dijkstra.
- **GPS e malhas viárias:** Google Maps, Waze e similares usam variantes de Dijkstra.
- **Games (Pathfinding):** O algoritmo A* é um Dijkstra guiado por heurística.
- **Redes de telecomunicações:** Roteamento de menor custo entre centrais.

• Teoria

Dijkstra é um algoritmo guloso que funciona porque, em grafos com pesos não-negativos, quando um vértice é extraído da fila de prioridade com distância d, não existe caminho mais curto até ele. Isso permite "fechar" vértices permanentemente, diferentemente de algoritmos como Bellman-Ford que podem precisar revisitar vértices.

7. Demonstração Passo a Passo

Considere o seguinte grafo bidirecional / direcionado:



Dijkstra a partir do vértice 0:

Iteração	Vértice	Distâncias Atualizadas	Fila de Prioridade (Min-Heap)
Inicial	-	dist[0]=0	(0, nó 0)
1	0	dist[1]=4, dist[2]=1	(1, nó 2), (4, nó 1)
2	2	dist[1]=3, dist[3]=6	(3, nó 1), (4, nó 1), (6, nó 3)
3	1	dist[3]=4	(4, nó 3), (4, nó 1), (6, nó 3)
4	3 (dist=4)	-	(4, nó 1), (6, nó 3)
5	1 (dist=4)	*Ignora (Lazy Deletion)*	(6, nó 3)
6	3 (dist=6)	*Ignora (Lazy Deletion)*	vazia

Resultado final: distâncias [0, 3, 1, 4].

Parte 3: Variações e Aprofundamento (Aula 22)

8. Dijkstra para Destino Único (Early Termination)

Se precisamos apenas da distância até um destino específico t, podemos parar cedo: quando extraímos t da fila, já temos a distância ótima.

```
while (!pq.isEmpty()) {
    Node current = pq.poll();
    int u = current.id;

    // EARLY TERMINATION: chegou no destino!
    if (u == destino) return dist[destino];
```

```
    if (current.distance > dist[u]) continue;
    for (Edge e : adj.get(u)) { /* relaxamento normal */ }
}
```

9. Dijkstra 0-1 (Deque)

Uma aplicação muito comum em maratonas (como "Labirinto com Portais") ocorre quando um robô pode mover-se com custo 1 (passo normal) ou custo 0 (portal de teletransporte). A Busca em Largura (BFS) clássica não funciona diretamente, pois os portais "pulam a fila" e quebram a ordenação FIFO por distâncias.

Se o grafo tem **apenas pesos 0 e 1**, podemos otimizar o Dijkstra trocando a Heap por um `Deque` (Fila Duplamente Encadeada) e obter tempo linear $O(V + E)$:

- Aresta de peso **0**: o custo não aumentou. Insira na **frente** do Deque para ser processado imediatamente.
- Aresta de peso **1**: o custo aumentou. Insira no **final** do Deque, mantendo a estrutura naturalmente ordenada.

```
Deque<int[]> dq = new ArrayDeque<>();
dq.addFirst(new int[]{src, 0});
while (!dq.isEmpty()) {
    int[] cur = dq.pollFirst();
    int u = cur[0], d = cur[1];
    if (d > dist[u]) continue;
    for (Edge e : adj.get(u)) {
        if (dist[u] + e.peso < dist[e.destino]) {
            dist[e.destino] = dist[u] + e.peso;
            if (e.peso == 0) dq.addFirst(new int[]{e.destino, dist[e.destin
            else             dq.addLast(new int[]{e.destino, dist[e.destino
        }
    }
}
```

10. Dijkstra com Múltiplos Critérios

Em maratonas, é comum desempatar por critérios extras (menor distância, desempatando por menos arestas):

```
// No relaxamento:
int novaDist = dist[u] + e.peso;
if (novaDist < dist[e.destino] ||
    (novaDist == dist[e.destino] && arestas[u]+1 < arestas[e.destino])) {
    dist[e.destino] = novaDist;
    arestas[e.destino] = arestas[u] + 1;
    pai[e.destino] = u;
```

```
    pq.add(new Node(e.destino, novaDist));
}
```

11. Dijkstra em Grid (Pathfinding)

Muitos problemas usam grids 2D. A adaptação é direta:

```
int[] dx = {-1, 1, 0, 0}; // cima, baixo, esq, dir
int[] dy = {0, 0, -1, 1};
int[][] dist = new int[N][M]; // preencher com INF
dist[0][0] = grid[0][0];
PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> a[2]-b[2]);
pq.add(new int[]{0, 0, grid[0][0]});

while (!pq.isEmpty()) {
    int[] cur = pq.poll();
    int x = cur[0], y = cur[1], d = cur[2];
    if (d > dist[x][y]) continue;
    for (int i = 0; i < 4; i++) {
        int nx = x+dx[i], ny = y+dy[i];
        if (nx>=0 && nx<N && ny>=0 && ny<M) {
            int nd = d + grid[nx][ny];
            if (nd < dist[nx][ny]) {
                dist[nx][ny] = nd;
                pq.add(new int[]{nx, ny, nd});
            }
        }
    }
}
```

12. K-ésimo Menor Caminho

Para encontrar não apenas o melhor, mas os K melhores caminhos:

```
int[] cnt = new int[V]; // contagem de visitas
while (!pq.isEmpty()) {
    Node cur = pq.poll();
    cnt[cur.id]++;
    if (cur.id == destino && cnt[cur.id] == K)
        return cur.distance; // K-'esimo menor!
    if (cnt[cur.id] > K) continue;
    for (Edge e : adj.get(cur.id))
        pq.add(new Node(e.destino, cur.distance + e.peso));
}
```

Complexidade: $O(KE \log KV)$. Viável para K pequeno.

13. Modelagem de Problemas: Estado Expandido

Muitas vezes, em problemas estilo maratona de programação, o menor caminho envolve múltiplas restrições.

Exemplo Clássico: Viagem com Limite de Combustível. Você viaja entre N cidades. Seu tanque tem capacidade máxima T. Cada estrada consome gasolina e cada cidade vende gasolina a um preço p_i. Qual o **menor custo em dinheiro** para ir da origem ao destino, começando de tanque vazio?

Em vez de modelar apenas os vértices como posições geográficas (cidades), modelamos o **estado completo** do sistema:

- **Estado:** `(cidade, combustível_atual)`. Cada estado forma um vértice distinto (total de $N \times T$ vértices).
- **Transição 1 (Abastecer):** Mover de `(c, F)` para `(c, F+1)` tem custo do litro p_c.
- **Transição 2 (Viajar):** Mover de `(u, F)` para `(v, F - consumo)` tem custo de dinheiro 0 (a gasolina já foi paga).

O Dijkstra executado sobre esse **Grafo de Estados** resolve problemas de otimização multivariáveis de forma elegante e correta!

14. Exercícios

14.1. Exercícios Conceituais

1 Por que Dijkstra não funciona com arestas de peso negativo? Dê um contraexemplo.

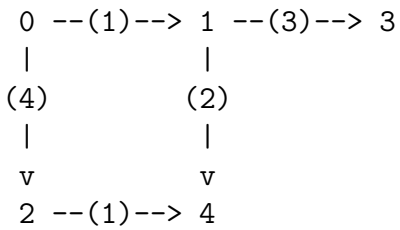
2 Explique o conceito de "lazy deletion" na implementação com Priority Queue. Por que é necessário?

3 Compare Dijkstra com BFS. Quando BFS é suficiente e quando precisamos de Dijkstra?

4 O que acontece se executarmos Dijkstra em um grafo desconexo? Como tratar vértices inalcançáveis?

14.2. Exercícios Analíticos

1 Simule Dijkstra passo a passo no grafo:



A partir do vértice 0, mostre a fila de prioridade e distâncias em cada iteração.

2 Para um grafo completo K_n com pesos unitários, compare o número de operações de Dijkstra vs BFS. Qual é mais eficiente?

3 Analise quando usar Binary Heap vs array simples para a fila. Para qual densidade de grafo cada implementação é preferível?

14.3. Exercícios de Programação

1 Implemente Dijkstra completo com cálculo de distâncias, reconstrução de caminhos e suporte a grafos direcionados e não direcionados.

2 Implemente o Dijkstra 0-1 com Deque e compare com o Dijkstra padrão em um grafo com pesos 0 e 1.

3 **Desafio (Estado Expandido):** Uma cidade tem N cruzamentos e M ruas. Cada rua tem tempo e pedágio. Encontre o menor tempo de 1 a N gastando no máximo P reais. *Dica: estado = (cruzamento, dinheiro_restante).*

4 Resolva problemas de juiz online: caminho mínimo, roteamento em redes, pathfinding em grids.