AED2 - Algoritmos e Estr. de Dados II

Aulas 20, 21 e 22: Algoritmo de Dijkstra

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

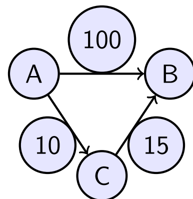
1. O Problema do Caminho Mínimo (O GPS dos Grafos)

Imagine que você precisa ir de uma cidade A para uma cidade B gastando o mínimo possível de combustível (ou tempo).

1. O Problema do Caminho Mínimo (O GPS dos Grafos)

Imagine que você precisa ir de uma cidade A para uma cidade B gastando o mínimo possível de combustível (ou tempo).

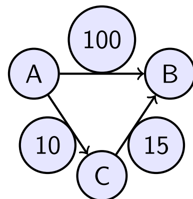
- **Por que não usar BFS?** A Busca em Largura garante o menor número de *arestas*, mas ignora completamente o fato de uma aresta custar muito mais que as outras. BFS só funciona se todas as arestas tiverem peso 1.



1. O Problema do Caminho Mínimo (O GPS dos Grafos)

Imagine que você precisa ir de uma cidade A para uma cidade B gastando o mínimo possível de combustível (ou tempo).

- **Por que não usar BFS?** A Busca em Largura garante o menor número de *arestas*, mas ignora completamente o fato de uma aresta custar muito mais que as outras. BFS só funciona se todas as arestas tiverem peso 1.
- **Dijkstra (1956):** Desenvolvido por Edsger W. Dijkstra, este algoritmo resolve o problema do **Caminho Mínimo de Origem Única** (Single-Source Shortest Path).

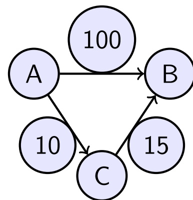


A $\rightarrow$ B direto custa 100.
Mas A $\rightarrow$ C $\rightarrow$ B custa apenas 25!

1. O Problema do Caminho Mínimo (O GPS dos Grafos)

Imagine que você precisa ir de uma cidade A para uma cidade B gastando o mínimo possível de combustível (ou tempo).

- **Por que não usar BFS?** A Busca em Largura garante o menor número de *arestas*, mas ignora completamente o fato de uma aresta custar muito mais que as outras. BFS só funciona se todas as arestas tiverem peso 1.
- **Dijkstra (1956):** Desenvolvido por Edsger W. Dijkstra, este algoritmo resolve o problema do **Caminho Mínimo de Origem Única** (Single-Source Shortest Path).
- **Restrição de Ouro:** Funciona APENAS para grafos onde **todos os pesos são não-negativos** ($w \geq 0$).



A → B direto custa 100.
Mas A → C → B custa apenas 25!

2. A Operação Fundamental: Relaxamento

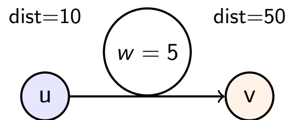
A magia do Dijkstra está em uma única operação chamada **Relaxamento**.

Temos um array `dist[v]` que guarda a "melhor distância conhecida até agora" da origem até v.

Regra do Relaxamento

Dado um caminho de u para v com peso $w(u, v)$: Se descobrir que passar por u é mais barato que o caminho atual que temos para v, nós **atualizamos** a distância!

```
if (dist[u] + w(u,v) < dist[v])
  dist[v] = dist[u] + w(u,v)
  pai[v] = u
```



2. A Operação Fundamental: Relaxamento

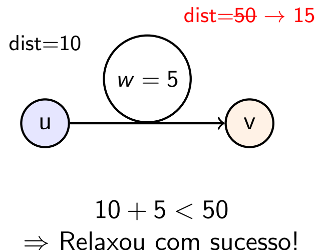
A magia do Dijkstra está em uma única operação chamada **Relaxamento**.

Temos um array `dist[v]` que guarda a "melhor distância conhecida até agora" da origem até v.

Regra do Relaxamento

Dado um caminho de u para v com peso $w(u, v)$: Se descobrir que passar por u é mais barato que o caminho atual que temos para v, nós **atualizamos** a distância!

```
if (dist[u] + w(u,v) < dist[v])
  dist[v] = dist[u] + w(u,v)
  pai[v] = u
```



3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.
3. Enquanto a fila não estiver vazia:
    - Extraia o nó u com a **menor distância** conhecida da fila.

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.
3. Enquanto a fila não estiver vazia:
   - Extraia o nó u com a **menor distância** conhecida da fila.
   - Se essa distância na fila for maior que `dist[u]`, significa que achamos um caminho melhor pra ele antes, então **ignore** (Lazy Deletion).

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.
3. Enquanto a fila não estiver vazia:
    - Extraia o nó u com a **menor distância** conhecida da fila.
    - Se essa distância na fila for maior que `dist[u]`, significa que achamos um caminho melhor pra ele antes, então **ignore** (Lazy Deletion).
    - Para cada vizinho v de u, tente fazer o **Relaxamento**.

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.
3. Enquanto a fila não estiver vazia:
   - Extraia o nó u com a **menor distância** conhecida da fila.
   - Se essa distância na fila for maior que `dist[u]`, significa que achamos um caminho melhor pra ele antes, então **ignore** (Lazy Deletion).
   - Para cada vizinho v de u, tente fazer o **Relaxamento**.
   - Se relaxou com sucesso, adicione `(nova_dist[v], v)` na fila!

3. O Algoritmo Guloso (Passo a Passo)

Dijkstra processa a malha como uma onda de água se expandindo pela rede.

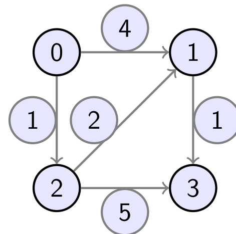
1. **Inicialização:** Defina `dist[Origem] = 0` e `dist[todos_outros] =` ∞.
2. **Fila de Prioridade (Min-Heap):** Insira `(0, Origem)` na fila.
3. Enquanto a fila não estiver vazia:
    - Extraia o nó u com a **menor distância** conhecida da fila.
    - Se essa distância na fila for maior que `dist[u]`, significa que achamos um caminho melhor pra ele antes, então **ignore** (Lazy Deletion).
    - Para cada vizinho v de u, tente fazer o **Relaxamento**.
    - Se relaxou com sucesso, adicione `(nova_dist[v], v)` na fila!

Por que é guloso e correto?

Ao extrair u, não há como descobrir um caminho mais curto para ele no futuro. Por quê? Porque todas as outras opções na fila já são piores que a de u, e como **não existem arestas negativas**, nenhum caminho que passe por eles vai subtrair custo.

Origem $= \mathbf{0}$. Inicializa `dist[0]=0`, `resto=`∞.

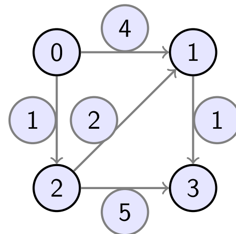
Nó u	**Vizinhos (Relaxamento)**	**PQ**



4. Dijkstra – Trace Animado

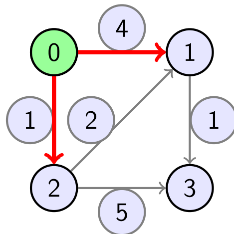
Origem $= \mathbf{0}$. Inicializa `dist[0]=0, resto=`∞.

Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)



Origem = **0**. Inicializa `dist[0]=0, resto=`∞.

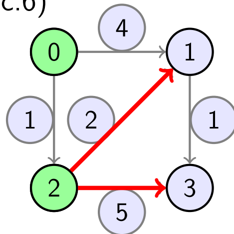
Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)



4. Dijkstra – Trace Animado

Origem $= \mathbf{0}$. Inicializa `dist[0]=0, resto=`∞.

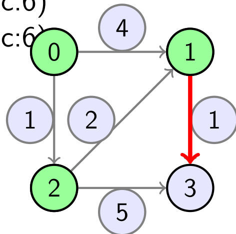
Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)



4. Dijkstra – Trace Animado

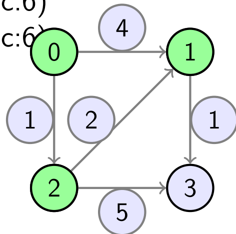
Origem = **0**. Inicializa `dist[0]=0, resto=`∞.

Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)
Visita 1	3 (3+1=4)	(1, c:4), (3, c:4), (3, c:6)



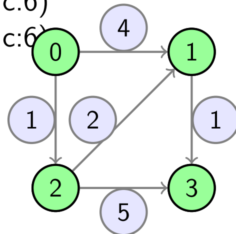
Origem = **0**. Inicializa `dist[0]=0, resto=`∞.

Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)
Visita 1	3 (3+1=4)	(1, c:4), (3, c:4), (3, c:6)
Ignora 1!	(já dist[1]=3 $<$ 4)	(3, c:4), (3, c:6)



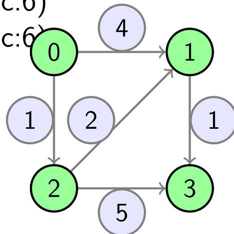
Origem = **0**. Inicializa `dist[0]=0, resto=∞`.

Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)
Visita 1	3 (3+1=4)	(1, c:4), (3, c:4), (3, c:6)
Ignora 1!	(já dist[1]=3 $<$ 4)	(3, c:4), (3, c:6)
Visita 3	nenhum	(3, c:6)



Origem = **0**. Inicializa `dist[0]=0, resto=`∞.

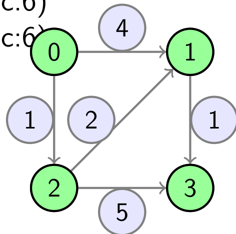
Nó u	**Vizinhos (Relaxamento)**	**PQ**
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)
Visita 1	3 (3+1=4)	(1, c:4), (3, c:4), (3, c:6)
Ignora 1!	(já dist[1]=3 $<$ 4)	(3, c:4), (3, c:6)
Visita 3	nenhum	(3, c:6)
Ignora 3!	(já dist[3]=4 $<$ 6)	vazia



4. Dijkstra – Trace Animado

Origem = **0**. Inicializa `dist[0]=0, resto=`∞.

Nó u	Vizinhos (Relaxamento)	PQ
-	-	(0, c:0)
Visita 0	1 (c:4), 2 (c:1)	(2, c:1), (1, c:4)
Visita 2	1 (1+2=3), 3 (1+5=6)	(1, c:3), (1, c:4), (3, c:6)
Visita 1	3 (3+1=4)	(1, c:4), (3, c:4), (3, c:6)
Ignora 1!	(já dist[1]=3 $<$ 4)	(3, c:4), (3, c:6)
Visita 3	nenhum	(3, c:6)
Ignora 3!	(já dist[3]=4 $<$ 6)	vazia



Resultado Final (`dist[]`)

`dist[0]=0, dist[1]=3, dist[2]=1, dist[3]=4`

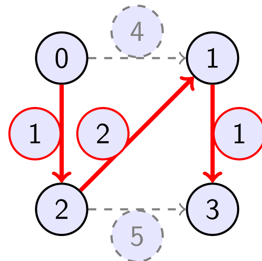
5. O Resultado: A Árvore de Caminhos Mínimos

Ao final da execução, se seguirmos de trás para frente usando o vetor de `pais`, formamos a **Árvore de Caminhos Mínimos** (Shortest Path Tree).

Vetor de Pais

Se no Trace anterior tínhamos:

- `pai[2] = 0`
- `pai[1] = 2`
- `pai[3] = 1`



Apenas as arestas ótimas restam

5. O Resultado: A Árvore de Caminhos Mínimos

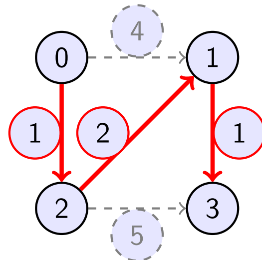
Ao final da execução, se seguirmos de trás para frente usando o vetor de `pais`, formamos a **Árvore de Caminhos Mínimos** (Shortest Path Tree).

Vetor de Pais

Se no Trace anterior tínhamos:

- `pai[2] = 0`
- `pai[1] = 2`
- `pai[3] = 1`

Para chegar no **3**, a rota é lida de trás para frente: $3 \leftarrow 1 \leftarrow 2 \leftarrow 0$. Logo, a rota é $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ com custo total **4**.



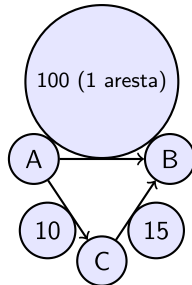
Apenas as arestas ótimas restam

6. Recapitulando: Por que não usar BFS?

Se a Busca em Largura (BFS) também usa fila e explora em ondas, por que precisamos do Dijkstra?

Busca em Largura (BFS)

- Usa uma **Fila Simples (FIFO)**.
- A primeira vez que chega num nó, assume que é o menor caminho.
- **Falha** se as arestas tiverem pesos diferentes.



BFS escolheria $A \to B$ (1 aresta).

6. Recapitulando: Por que não usar BFS?

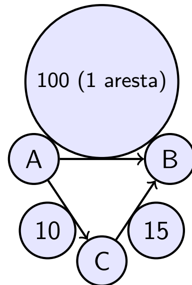
Se a Busca em Largura (BFS) também usa fila e explora em ondas, por que precisamos do Dijkstra?

Busca em Largura (BFS)

- Usa uma **Fila Simples (FIFO)**.
- A primeira vez que chega num nó, assume que é o menor caminho.
- **Falha** se as arestas tiverem pesos diferentes.

Dijkstra

- Usa uma **Fila de Prioridade**.
- Pode descobrir rotas curtas depois e **relaxar** arestas.
- **Garante** a menor soma (pesos



BFS escolheria $A \to B$ (1 aresta).
Dijkstra escolhe $A \to C \to B$ (menor custo).

7a. Implementação em Java – Estruturas

A `PriorityQueue` no Java guarda Objetos e precisa de um comparador para saber quem é o menor.

```
import java.util.*;

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

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

    Overridepublic int compareTo(Node other) return
        Integer.compare(this.distance, other.distance); // Min-Heap
```

7b. Implementação em Java – O Algoritmo

```java
public static int[] dijkstra(int V, List<List<Edge>> adj, int startNode) {
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE); // Tudo infinito no início
    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;

        // Lazy Deletion: Se já achamos um caminho melhor pra ‘u‘ e
        // este nó na fila ficou obsoleto, apenas ignore.
        if (current.distance > dist[u]) continue;

        // Relaxa vizinhos
        for (Edge e : adj.get(u)) {
            if (dist[u] + e.peso < dist[e.destino]) {
                dist[e.destino] = dist[u] + e.peso;
                // Adiciona o novo caminho relaxado na fila
                pq.add(new Node(e.destino, dist[e.destino]));
```

8. O Calcanhar de Aquiles: Pesos Negativos

O que acontece se uma aresta tiver custo -10?

A Quebra do Paradigma Guloso

O Dijkstra **sela** a distância de um vértice quando o tira da fila, assumindo que nenhum caminho futuro será mais barato. Arestas negativas permitem "voltar no tempo" e criar um atalho milagroso, burlando a premissa fundamental do algoritmo e gerando respostas erradas.

O que acontece se uma aresta tiver custo -10?

A Quebra do Paradigma Guloso

O Dijkstra **sela** a distância de um vértice quando o tira da fila, assumindo que nenhum caminho futuro será mais barato. Arestas negativas permitem "voltar no tempo" e criar um atalho milagroso, burlando a premissa fundamental do algoritmo e gerando respostas erradas.

Ciclo Negativo

Pior ainda: se as arestas formarem um **ciclo com soma total negativa**, o algoritmo vai girar nele reduzindo o custo infinitamente rumo ao $-\infty$.

8. O Calcanhar de Aquiles: Pesos Negativos

O que acontece se uma aresta tiver custo -10?

A Quebra do Paradigma Guloso

O Dijkstra **sela** a distância de um vértice quando o tira da fila, assumindo que nenhum caminho futuro será mais barato. Arestas negativas permitem "voltar no tempo" e criar um atalho milagroso, burlando a premissa fundamental do algoritmo e gerando respostas erradas.

Ciclo Negativo

Pior ainda: se as arestas formarem um **ciclo com soma total negativa**, o algoritmo vai girar nele reduzindo o custo infinitamente rumo ao $-\infty$.

A Solução Definitiva

Para grafos com pesos negativos ou para detectar ciclos negativos de forma segura, usamos outro algoritmo clássico:

Bellman-Ford

Complexidade: $\mathcal{O}(V \times E)$

9. Complexidade e Aplicações Reais

Complexidade Computacional

9. Complexidade e Aplicações Reais

Complexidade Computacional

- **Com Binary Heap (PriorityQueue Java):** $\mathcal{O}((V+E)\log V)$. Como em grafos conexos $E \geq V$, simplificamos para $\mathcal{O}(E \log V)$. Perfeito para quase qualquer cenário real de engenharia.

9. Complexidade e Aplicações Reais

Complexidade Computacional

- **Com Binary Heap (PriorityQueue Java):** $\mathcal{O}((V + E) \log V)$. Como em grafos conexos $E \geq V$, simplificamos para $\mathcal{O}(E \log V)$. Perfeito para quase qualquer cenário real de engenharia.
- **Sem Heap (Matriz / Busca Linear):** $\mathcal{O}(V^2)$. Surpreendentemente, é mais rápido que a Heap apenas se o grafo for extremamente denso ($E \approx V^2$).

9. Complexidade e Aplicações Reais

Complexidade Computacional

- **Com Binary Heap (PriorityQueue Java):** $\mathcal{O}((V+E)\log V)$. Como em grafos conexos $E \geq V$, simplificamos para $\mathcal{O}(E \log V)$. Perfeito para quase qualquer cenário real de engenharia.
- **Sem Heap (Matriz / Busca Linear):** $\mathcal{O}(V^2)$. Surpreendentemente, é mais rápido que a Heap apenas se o grafo for extremamente denso ($E \approx V^2$).
- **Com Fibonacci Heap:** $\mathcal{O}(E + V \log V)$. O limite teórico ótimo, mas tem uma constante muito alta, tornando-o complexo demais para usar na prática.

9. Complexidade e Aplicações Reais

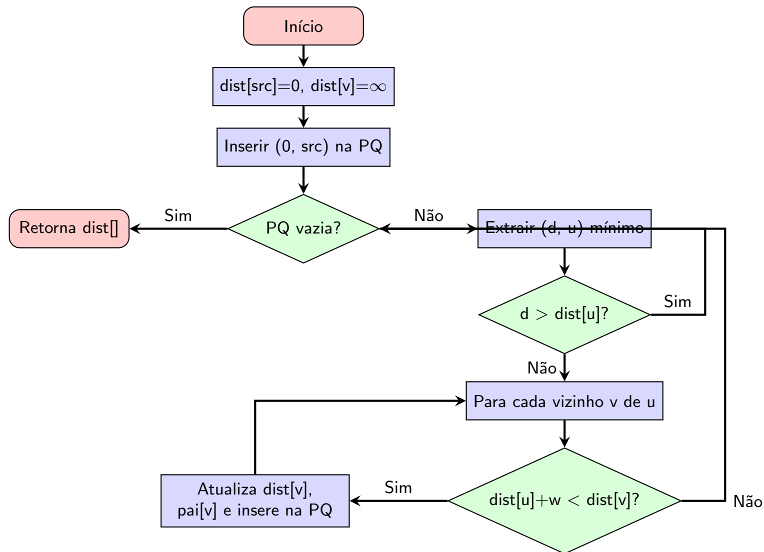
Complexidade Computacional

- **Com Binary Heap (PriorityQueue Java):** $\mathcal{O}((V+E)\log V)$. Como em grafos conexos $E \geq V$, simplificamos para $\mathcal{O}(E\log V)$. Perfeito para quase qualquer cenário real de engenharia.
- **Sem Heap (Matriz / Busca Linear):** $\mathcal{O}(V^2)$. Surpreendentemente, é mais rápido que a Heap apenas se o grafo for extremamente denso ($E \approx V^2$).
- **Com Fibonacci Heap:** $\mathcal{O}(E + V\log V)$. O limite teórico ótimo, mas tem uma constante muito alta, tornando-o complexo demais para usar na prática.

Onde o Dijkstra trabalha hoje?

- **Protocolos de Roteamento de Internet:** OSPF e IS-IS calculam a árvore de rotas mais curtas nas entranhas da Internet usando o Dijkstra.
- **Pathfinding em Mapas e Games:** O motor de rotas do Google Maps e movimentos de I.A. em jogos usam o A* (A-Star), que é matematicamente um Dijkstra guiado por uma Heurística

10. Fluxograma do Dijkstra



11. Dijkstra para Destino Único (Early Termination)

Se precisamos apenas da distância até um **destino específico** t, podemos parar cedo.

Modificação

Quando extraímos t da fila de prioridade, já temos a distância ótima. Basta retornar imediatamente.

```
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 */ }
}
```

Quando usar?

Quando o destino é “próximo” da origem no grafo. Em mapas, isso economiza muito

12. Dijkstra 0-1 (Deque)

E se o grafo tem **apenas pesos 0 e 1**? Podemos trocar a Heap por um Deque e obter $\mathcal{O}(V + E)$!

Ideia

- Se a aresta tem peso **0**, insira na **frente** do Deque.
- Se a aresta tem peso **1**, insira no **final** do Deque.

Assim a estrutura se mantém ordenada naturalmente!

```
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.destino]});
            else             dq.addLast(new int[]{e.destino, dist[e.destino]});
        }
```

13. Reconstrução do Caminho

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

```java
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;
}
```

Cuidado

Inicialize pai[src] = −1 e pai[v] = −1 para todos. Se ao final pai[destino] == −1 e destino $\neq$ src, o destino é **inalcançável**.

14. Dijkstra com Múltiplos Critérios

Em problemas de maratona, é comum precisar desempatar por critérios extras.

Exemplo: Menor distância, desempatando por menos arestas

- Mantemos dois vetores: `dist[]` e `arestas[]`.
- No relaxamento, se `dist[u]+w == dist[v]`, verificamos se `arestas[u]+1 < arestas[v]`.

```java
// 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));
}
```

Variantes comuns: menor custo com mais lucro, menor tempo com menos pedágios.

15. Aplicação 1: Rede de Entregas com Pedágio Variável

Enunciado (Estilo ICPC/OBI)

Uma empresa de entregas opera em N cidades conectadas por M estradas. Cada estrada i tem um custo de combustível c_i. A empresa parte do depósito (cidade 1) e precisa encontrar o **menor custo** para chegar a cada cidade.

Entrada: N, M, seguidos de M triplas (u, v, c).

Saída: N inteiros: custo mínimo do depósito a cada cidade, ou -1 se inalcançável.

15. Aplicação 1: Rede de Entregas com Pedágio Variável

Enunciado (Estilo ICPC/OBI)

Uma empresa de entregas opera em N cidades conectadas por M estradas. Cada estrada i tem um custo de combustível c_i. A empresa parte do depósito (cidade 1) e precisa encontrar o **menor custo** para chegar a cada cidade.

Entrada: N, M, seguidos de M triplas (u, v, c).

Saída: N inteiros: custo mínimo do depósito a cada cidade, ou -1 se inalcançável.

Por que é Dijkstra?

Modelagem direta: cidades = vértices, estradas = arestas, custo = peso. Pesos $\geq 0 \Rightarrow$ Dijkstra clássico.

16. Aplicação 2: Labirinto com Portais (Dijkstra 0-K)

Enunciado

Um robô está num grid $N \times M$. Cada célula pode ser livre (custo 1 para mover) ou conter um portal (custo 0 para teletransportar a uma célula específica). Encontre o **menor número de passos** do canto $(1, 1)$ ao (N, M).

16. Aplicação 2: Labirinto com Portais (Dijkstra 0-K)

Enunciado

Um robô está num grid $N \times M$. Cada célula pode ser livre (custo 1 para mover) ou conter um portal (custo 0 para teletransportar a uma célula específica). Encontre o **menor número de passos** do canto $(1, 1)$ ao (N, M).

Modelagem

- Cada célula (i, j) é um vértice.
- Movimentos adjacentes: arestas de peso $\mathbf{1}$.
- Portais: arestas de peso $\mathbf{0}$.
- Como pesos são 0 e 1 $\Rightarrow$ **Dijkstra 0-1 com Deque**!

16. Aplicação 2: Labirinto com Portais (Dijkstra 0-K)

Enunciado

Um robô está num grid $N \times M$. Cada célula pode ser livre (custo 1 para mover) ou conter um portal (custo 0 para teletransportar a uma célula específica). Encontre o **menor número de passos** do canto $(1, 1)$ ao (N, M).

Modelagem

- Cada célula (i, j) é um vértice.
- Movimentos adjacentes: arestas de peso **1**.
- Portais: arestas de peso **0**.
- Como pesos são 0 e 1 $\Rightarrow$ **Dijkstra 0-1 com Deque**!

Armadilha

BFS simples **não funciona** aqui porque portais têm custo 0 ("pula a fila"). Usar BFS trataria portal como um passo normal.

17. Aplicação 3: Viagem com Limite de Combustível

Enunciado

Você viaja entre N cidades. Seu tanque tem capacidade T. Em cada cidade i, o litro custa p_i. Cada estrada (u, v) consome d_{uv} litros. Qual o **menor custo em dinheiro** para ir da cidade 1 à cidade N, começando com tanque vazio?

Enunciado

Você viaja entre N cidades. Seu tanque tem capacidade T. Em cada cidade i, o litro custa p_i. Cada estrada (u, v) consome d_{uv} litros. Qual o **menor custo em dinheiro** para ir da cidade 1 à cidade N, começando com tanque vazio?

Modelagem com Estado Expandido

- **Estado:** $(cidade,\ combustível_atual)$.
- **Transição – Abastecer:** $(c, f) \to (c, f + 1)$ com custo p_c.
- **Transição – Viajar:** $(u, f) \to (v, f - d_{uv})$ com custo 0, se $f \geq d_{uv}$.
- O grafo de estados tem $N \times T$ vértices $\Rightarrow$ Dijkstra normal nesse grafo!

Enunciado

Você viaja entre N cidades. Seu tanque tem capacidade T. Em cada cidade i, o litro custa p_i. Cada estrada (u, v) consome d_{uv} litros. Qual o **menor custo em dinheiro** para ir da cidade 1 à cidade N, começando com tanque vazio?

Modelagem com Estado Expandido

- **Estado:** $(cidade, \ combustível_atual)$.
- **Transição – Abastecer:** $(c, f) \to (c, f + 1)$ com custo p_c.
- **Transição – Viajar:** $(u, f) \to (v, f - d_{uv})$ com custo 0, se $f \geq d_{uv}$.
- O grafo de estados tem $N \times T$ vértices $\Rightarrow$ Dijkstra normal nesse grafo!

Dica: Expandir o espaço de estados é uma técnica poderosa para adaptar Dijkstra a problemas com restrições adicionais.

18. Adaptação: Dijkstra em Grid (Pathfinding)

Muitos problemas usam grids 2D em vez de grafos explícitos. 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});
            }
        }
    }
```

19. Adaptação: K-ésimo Menor Caminho

Às vezes precisamos não só do melhor caminho, mas dos K melhores.

Ideia

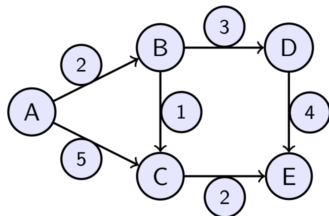
- Permita que cada vértice seja "visitado" até K vezes.
- Use um contador `cnt[v]` que conta quantas vezes v foi extraído.
- Quando `cnt[destino] == K`, encontramos o K-ésimo menor caminho.

```
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-ésimo menor!
    if (cnt[cur.id] > K) continue; // ignora extras
    for (Edge e : adj.get(cur.id))
        pq.add(new Node(e.destino, cur.distance + e.peso));
}
```

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

Exercício 1: Trace Manual

Aplique o algoritmo de Dijkstra **passo a passo** no grafo abaixo, partindo do vértice **A**.

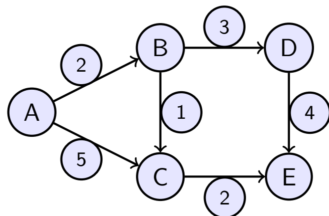


Preencha a tabela:

1. Qual a ordem de visitação dos vértices?
2. Qual o vetor `dist[]` final?
3. Qual o vetor `pai[]` final?
4. Qual o caminho de A até E?
5. Desenhe a Árvore de Caminhos Mínimos.

Exercício 1: Trace Manual

Aplique o algoritmo de Dijkstra **passo a passo** no grafo abaixo, partindo do vértice **A**.



Preencha a tabela:

1. Qual a ordem de visitação dos vértices?
2. Qual o vetor `dist[]` final?
3. Qual o vetor `pai[]` final?
4. Qual o caminho de A até E?
5. Desenhe a Árvore de Caminhos Mínimos.

Resposta

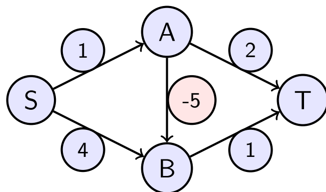
Ordem: A, B, C, D, E.

`dist = [A:0, B:2, C:3, D:5, E:5]`.

Caminho A→E: A→B→C→E (custo 5).

Exercício 2: Quando Dijkstra Falha?

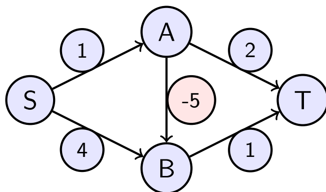
Considere o grafo abaixo com uma aresta de **peso negativo**. Aplique Dijkstra a partir de **S** e mostre onde o algoritmo erra.



1. Qual resultado o Dijkstra produz para `dist[T]`?
2. Qual é o caminho **real** mais curto de S a T?
3. **Explique** por que o algoritmo guloso falha neste caso.

Exercício 2: Quando Dijkstra Falha?

Considere o grafo abaixo com uma aresta de **peso negativo**. Aplique Dijkstra a partir de **S** e mostre onde o algoritmo erra.



1. Qual resultado o Dijkstra produz para `dist[T]`?
2. Qual é o caminho **real** mais curto de S a T?
3. **Explique** por que o algoritmo guloso falha neste caso.

Resposta

Dijkstra "sela" A com dist=1, depois T com dist=3 (via A).
Mas S→A→B→T custa $1 + (-5) + 1 = -3$.
Dijkstra nunca descobre isso pois B foi selado com dist=4 antes.

Prof. Aléssio Miranda Júnior - alessio@cefetmg.br

Exercício 3: Modelagem de Problema

Problema

Uma cidade tem N cruzamentos e M ruas de mão única. Cada rua i tem um tempo de percurso t_i e um pedágio p_i. Você quer ir do cruzamento 1 ao N com o **menor tempo possível**, mas tem no máximo P reais para pedágios.
Dados: $N \leq 1000$, $M \leq 5000$, $P \leq 100$.

1. Como você modelaria o **estado** do Dijkstra? *Dica: pense em estado expandido.*
2. Quantos vértices teria o grafo de estados?
3. Qual seria a complexidade final?

Prof. Aléssio Miranda Júnior - alessio@cefetmg.br

Exercício 3: Modelagem de Problema

Problema

Uma cidade tem N cruzamentos e M ruas de mão única. Cada rua i tem um tempo de percurso t_i e um pedágio p_i. Você quer ir do cruzamento 1 ao N com o **menor tempo possível**, mas tem no máximo P reais para pedágios.
Dados: $N \leq 1000$, $M \leq 5000$, $P \leq 100$.

1. Como você modelaria o **estado** do Dijkstra? *Dica: pense em estado expandido.*
2. Quantos vértices teria o grafo de estados?
3. Qual seria a complexidade final?

Resposta

Estado: $(cruzamento, dinheiro_restante)$.
Vértices: $N \times (P + 1) = 1000 \times 101 = 101.000$.
Transição: $(u, d) \rightarrow (v, d - p_i)$ com custo t_i, se $d \geq p_i$.
Complexidade: $\mathcal{O}(M \cdot P \cdot \log(N \cdot P))$ – viável!

Exercício 4: Implementação Rápida

Desafio de Código

Implemente o Dijkstra para resolver: dado um grafo ponderado não-direcionado com N vértices e M arestas, encontre a menor distância de 1 a N. Se impossível, imprima -1.

Entrada:

- Linha 1: N M
- Próximas M linhas: u v w (aresta bidirecional de peso w)

Saída: Menor distância de 1 a N, ou -1.

Exemplo:

```
4 5
1 2 4 / 1 3 1 / 3 2 2 / 2 4 1 / 3 4 5
```

Saída: 4 (caminho 1→3→2→4)

Resumo Geral e Quando Usar o Quê

Algoritmo	Pesos	Complexidade	Quando usar
BFS	Todos $= 1$	$\mathcal{O}(V + E)$	Grafos sem peso
Dijkstra 0-1	0 ou 1	$\mathcal{O}(V + E)$	Portais, toggle
Dijkstra (Heap)	≥ 0	$\mathcal{O}(E \log V)$	Caso geral
Bellman-Ford	Qualquer	$\mathcal{O}(VE)$	Pesos negativos
Floyd-Warshall	Qualquer	$\mathcal{O}(V^3)$	Todos-para-todos

Checklist de Maratona

1. Pesos são todos iguais a 1? $\Rightarrow$ **BFS**.
2. Pesos são 0 ou 1? $\Rightarrow$ **Dijkstra 0-1 (Deque)**.
3. Pesos ≥ 0? $\Rightarrow$ **Dijkstra com Heap**.
4. Pesos negativos sem ciclo negativo? $\Rightarrow$ **Bellman-Ford**.
5. Precisa de todos os pares? $\Rightarrow$ **Floyd-Warshall**.
6. Tem restrição extra (combustível, dinheiro)? $\Rightarrow$ **Estado expandido**!