# Algoritmos e Estruturas de Dados II

Capítulo da Aula 21: Bellman-Ford e Floyd-Warshall

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

Fevereiro de 2026

## 1. Quando o Dijkstra falha?

Relembrando: o Dijkstra funciona porque, ao extrair um vértice da fila de prioridade, assume que **nenhum caminho futuro pode ser mais barato**. Essa premissa depende de todas as arestas terem peso $\geq 0$.

Se um grafo tem arestas com **pesos negativos**, o Dijkstra não funciona porque uma aresta negativa pode "voltar no tempo" e criar um atalho que o algoritmo guloso já descartou.

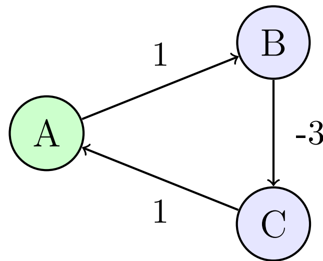
### Onde aparecem pesos negativos na vida real?

Na prática, um custo (peso) negativo geralmente representa um **ganho** ou **bônus**.

- **Veículos Elétricos e Regeneração de Bateria:** Se modelarmos os pesos das ruas como o **consumo de bateria**, subidas gastam bateria (peso $> 0$), mas descidas ativam a frenagem regenerativa e **recarregam** o veículo (peso $< 0$).
- **Arbitragem Financeira (Câmbio):** Ao trocar moedas (ex: USD $\rightarrow$ EUR $\rightarrow$ GBP $\rightarrow$ USD), as flutuações do mercado podem fazer com que você termine com mais dinheiro do que começou. Ao aplicar propriedades de logaritmos nas taxas, esse lucro se transforma em um caminho de peso matemático negativo.
- **Redes de Fluxo e Jogos:** Arestas de desfazer fluxos ou bônus em grafos de estados podem ser modelados com custos negativos.

### O Perigo: Ciclos Negativos

Se houver um **Ciclo Negativo** — um ciclo fechado cuja soma total dos pesos é negativa — não existe "menor caminho". Na vida real, rodar em um ciclo negativo significa **ganho infinito** (recarregar a bateria infinitamente dando voltas no quarteirão, ou gerar dinheiro infinito). Na matemática, significa que podemos percorrer o ciclo repetidamente, reduzindo o custo infinitamente (tendendo a $-\infty$).




Soma do ciclo: $1 + (-3) + 1 = -1 < 0 \Rightarrow$ Ciclo Negativo!

Figure 1: Ciclo negativo: cada volta reduz o custo em 1. Não existe caminho mínimo finito.

**△ Importante**

Se o problema pede "menor caminho" e o grafo pode ter ciclos negativos, você **deve** primeiro verificar se eles existem. Caso existam, a resposta para vértices afetados é $-\infty$ (ou "impossível").

## 2. Algoritmo de Bellman-Ford

Criado por Richard Bellman (pai da Programação Dinâmica) e Lester Ford Jr. em 1958. Resolve o problema de **Single Source Shortest Path** mesmo com pesos negativos e detecta ciclos negativos.

#### Intuição: Por que $V - 1$ Iterações?

Um caminho simples (sem repetir vértices) em um grafo com $V$ vértices pode ter **no máximo $V - 1$ arestas**. Logo:

- Após a **1ª iteração**: caminhos usando no máximo 1 aresta estão corretos.
- Após a **2ª iteração**: caminhos usando no máximo 2 arestas estão corretos.

- ⋮
- Após a $(V-1)$**-ésima iteração**: **todos** os caminhos mínimos estão corretos.

Se na iteração $V$ (a extra) ainda for possível relaxar alguma aresta, significa que existe um caminho que usa mais de $V-1$ arestas — portanto, passa por um **Ciclo Negativo**.

### Algoritmo Passo a Passo

1. **Inicialização:** `dist[S] = 0`, `dist[v] =` $\infty$ para todo $v \neq S$.
2. **Relaxamento ($V-1$ vezes):** Para cada iteração $i$ de 1 a $V-1$:
   - Para cada aresta $(u, v, w)$ do grafo:
   - Se `dist[u] + w < dist[v]`, então `dist[v] = dist[u] + w`.
3. **Detecção de Ciclo Negativo:** Percorra todas as arestas mais uma vez. Se alguma puder ser relaxada, existe ciclo negativo.

> • **Teoria**
>
> A diferença fundamental entre Dijkstra e Bellman-Ford: Dijkstra relaxa cada aresta **uma vez** (guloso), enquanto Bellman-Ford relaxa **todas as arestas** repetidamente ($V-1$ vezes), garantindo que mesmo caminhos com pesos negativos sejam encontrados.

### Demonstração Passo a Passo

Considere o grafo com 5 vértices e origem $S=0$:

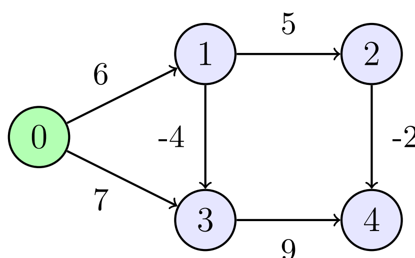



Figure 2: Grafo exemplo para Bellman-Ford com aresta de peso negativo.

| Iter. | d[0] | d[1] | d[2] | d[3] | d[4] | Arestas relaxadas |
|---|---|---|---|---|---|---|
| Início | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | — |
| 1 | 0 | 6 | 11 | 2 | 9 | (0,1), (1,2), (1,3), (2,4), (3,4) |
| 2 | 0 | 6 | 11 | 2 | 9 | nenhuma (estável) |

Resultado final: `dist = [0, 6, 11, 2, 9]`.

### Implementação Java (Edge List)

```java
class Aresta {
    int u, v, w;
    Aresta(int u, int v, int w) { this.u=u; this.v=v; this.w=w; }
}

public static int[] bellmanFord(int V, int start, List<Aresta> arestas) {
    int[] dist = new int[V];
    int[] pai = new int[V];
    final int INF = 999999999;
    Arrays.fill(dist, INF);
    Arrays.fill(pai, -1);
    dist[start] = 0;

    // Passo 1: Relaxar V-1 vezes
    for (int i = 0; i < V - 1; i++) {
        boolean mudou = false; // Otimização: parar cedo
        for (Aresta e : arestas) {
            if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.w;
                pai[e.v] = e.u;
                mudou = true;
            }
        }
        if (!mudou) break; // Convergiu antes de V-1!
    }

    // Passo 2: Detectar Ciclo Negativo
    for (Aresta e : arestas) {
        if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v]) {
            System.out.println("Ciclo Negativo Detectado!");
            return null;
        }
    }
    return dist;
}
```

### Otimização: Early Termination

Se em alguma iteração **nenhuma aresta foi relaxada**, o algoritmo já convergiu. A flag `mudou` no código acima implementa isso. No melhor caso, o algoritmo pode terminar em $O(E)$ ao invés de $O(VE)$.

### Complexidade

- **Tempo:** $O(V \times E)$. No pior caso, precisamos de todas as $V - 1$ iterações.

- **Espaço:** $O(V+E)$ para armazenar distâncias e a lista de arestas.
- **Comparação com Dijkstra:** Muito mais lento ($O(VE)$ vs $O(E \log V)$), mas funciona com pesos negativos.

▶ Prática

**Quando usar Bellman-Ford ao invés de Dijkstra?**

1. Quando o grafo tem arestas com peso negativo.
2. Quando precisamos detectar ciclos negativos.
3. Quando a simplicidade de implementação é mais importante que performance (lista de arestas, sem heap).

#### Aplicações do Mundo Real

- **Protocolo de Roteamento RIP (Redes):** O RIP (*Routing Information Protocol*) da camada de rede usa Bellman-Ford de forma distribuída. Roteadores trocam informações de custo iterativamente com seus vizinhos para descobrir rotas ótimas.
- **Restrições de Diferença e Escalonamento:** Em gerência de projetos, se a Tarefa B deve iniciar no mínimo 2 horas após a Tarefa A, isso forma uma inequação $t_B - t_A \geq 2$. Tais sistemas podem ser representados como grafos. O Bellman-Ford os resolve e usa a detecção de ciclo negativo para provar se as restrições são impossíveis de satisfazer.

## 3. Algoritmo de Floyd-Warshall

O Floyd-Warshall resolve um problema diferente: **All-Pairs Shortest Path** (APSP). Queremos a matriz de distâncias de **todos** para **todos** os vértices.

#### Intuição: Programação Dinâmica

A ideia central é usar **Programação Dinâmica** com vértices intermediários.

Definimos $D^{(k)}[i][j]$ como o menor caminho de $i$ a $j$ usando **apenas vértices** $\{0, 1, \ldots, k\}$ **como intermediários**.

**Recorrência:**

$$D^{(k)}[i][j] = \min\left(D^{(k-1)}[i][j],\quad D^{(k-1)}[i][k] + D^{(k-1)}[k][j]\right)$$

Em palavras: para ir de $i$ a $j$, ou não passamos por $k$ (mantemos o valor anterior), ou passamos por $k$ (somamos $i \to k$ + $k \to j$).

#### Por que a Ordem $k, i, j$ é Crucial?

O loop de $k$ **deve ser o mais externo** porque estamos construindo a solução incrementalmente: primeiro permitimos apenas o vértice 0 como intermediário, depois 0 e 1, depois 0, 1 e 2, e assim por diante.

Se trocarmos a ordem (por exemplo, $i, j, k$), estaríamos tentando usar vértices intermediários antes de calcular os caminhos que passam por eles — gerando resultados incorretos.

### Algoritmo Passo a Passo

1 Crie uma matriz $D[V][V]$:

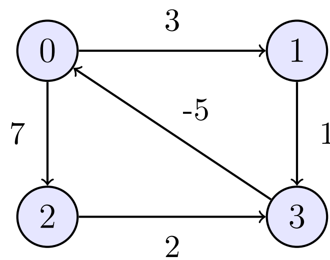
- $D[i][i] = 0$ (distância de um vértice para ele mesmo).
- $D[i][j] = peso(i, j)$ se existe aresta direta.
- $D[i][j] = \infty$ caso contrário.

2 Execute três loops aninhados: $k$ (externo), $i$ (meio), $j$ (interno).

3 Para cada combinação, verifique se passar por $k$ melhora o caminho de $i$ a $j$.

### Demonstração Passo a Passo

Grafo com 4 vértices:




Matriz inicial $D^{(-1)}$:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 3 | 7 | $\infty$ |
| 1 | $\infty$ | 0 | $\infty$ | 1 |
| 2 | $\infty$ | $\infty$ | 0 | 2 |
| 3 | -5 | $\infty$ | $\infty$ | 0 |

Após $k = 3$ (permitindo todos os intermediários), matriz final:

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 3 | 7 | 4 |
| 1 | -4 | 0 | -4 | 1 |
| 2 | -3 | -3 | 0 | 2 |
| 3 | -5 | -2 | -5 | 0 |

### Implementação Java

```java
static final int INF = 999999999;

public static int[][] floydWarshall(int V, int[][] grafo) {
    int[][] dist = new int[V][V];
    int[][] next = new int[V][V]; // Para reconstruir caminhos

    // Inicialização
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = grafo[i][j];
            next[i][j] = (grafo[i][j] != INF && i != j) ? j : -1;
        }
    }

    // O coração do algoritmo
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k]; // Atualiza caminho
                }
            }
        }
    }

    // Detecção de Ciclo Negativo
    for (int i = 0; i < V; i++) {
        if (dist[i][i] < 0) {
            System.out.println("Ciclo negativo envolvendo " + i);
        }
    }
    return dist;
}
```

### Reconstrução de Caminhos

A matriz `next[i][j]` armazena o próximo vértice no caminho ótimo de $i$ a $j$:

```java
public static List<Integer> reconstruirCaminho(int[][] next, int u, int v)
    if (next[u][v] == -1) return null; // Sem caminho
    List<Integer> caminho = new ArrayList<>();
    caminho.add(u);
    while (u != v) {
```

```
        u = next[u][v];
        caminho.add(u);
    }
    return caminho;
}
```

### Características e Limitações

- **Complexidade de Tempo:** $O(V^3)$.
- **Complexidade de Espaço:** $O(V^2)$ para a matriz.
- **Limitação Prática:** Viável para $V \leq 400 \sim 500$ (em maratonas, $\sim 10^8$ operações $\approx$ 1s).
- **Vantagem:** Implementação extremamente simples (3 loops aninhados).
- **Ciclos Negativos:** Se ao final `dist[i][i] < 0`, existe ciclo negativo envolvendo `i`.
- **Funciona com pesos negativos:** Sim, desde que não haja ciclo negativo (ou detecte-os).

## 4. Aplicações no Mundo Real

Além da busca por caminhos mínimos, a Programação Dinâmica do Floyd-Warshall e o algoritmo de Bellman-Ford têm uso extensivo em áreas complexas da computação.

### Arbitragem de Moedas (Câmbio)

Um banco oferece as seguintes taxas de câmbio:

$$\text{USD} \to \text{EUR: } 0.9 \quad \text{EUR} \to \text{GBP: } 0.85 \quad \text{GBP} \to \text{USD: } 1.35$$

Começando com \$1 USD: $1 \times 0.9 \times 0.85 \times 1.35 = 1.03275$ USD. Lucro de 3.27% do nada! Isso é **arbitragem**. Modelamos isso transformando taxas em $-\log(\text{taxa})$. Um ciclo negativo indica lucro! O Bellman-Ford detecta esses atalhos infinitamente lucrativos.

### Fecho Transitivo (Algoritmo de Warshall)

Em compiladores e resolução de dependências, muitas vezes queremos apenas saber: "Existe **algum** caminho de $A$ a $B$?". Substituindo as somas e os testes de mínimo por operadores lógicos booleanos (`AND` e `OR`), criamos a matriz de alcançabilidade:

```
// Fecho Transitivo: reach[i][j] = true se existe caminho de i a j
boolean[][] reach = new boolean[V][V];
for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
```

### Análise de Redes e Logística

- **Redes Sociais:** O Floyd-Warshall é usado para calcular métricas como **Centralidade de Proximidade** e descobrir os "graus de separação" entre todos os indivíduos em uma rede social.
- **Logística e Aviação:** Útil para pré-processar as distâncias mínimas exatas de uma pequena malha aérea/rodoviária. Com a matriz $D$ pré-calculada, o sistema de roteamento não precisa buscar o caminho do zero e consegue responder a consultas entre quaisquer duas cidades instantaneamente, em tempo $O(1)$.

## 5. Comparativo Geral

| Algoritmo | Tipo | Neg.? | Complexidade | Uso Ideal |
|---|---|---|---|---|
| BFS | 1→Todos | Não | $O(V+E)$ | Grafos sem peso |
| Dijkstra | 1→Todos | Não | $O(E \log V)$ | Pesos $\geq 0$ |
| Bellman-Ford | 1→Todos | **Sim** | $O(V \cdot E)$ | Pesos negativos |
| Floyd-Warshall | Todos→Todos | **Sim** | $O(V^3)$ | Grafos pequenos |

• Teoria

**Quando usar qual?**

- Preciso de distância de **uma origem** e pesos $\geq 0$? $\Rightarrow$ **Dijkstra**.
- Preciso de distância de **uma origem** e existem pesos negativos? $\Rightarrow$ **Bellman-Ford**.
- Preciso de distâncias entre **todos os pares** e $V \leq 500$? $\Rightarrow$ **Floyd-Warshall**.
- Preciso de todos os pares mas $V$ é grande? $\Rightarrow$ Rodar **Dijkstra/BF $V$ vezes**.

## 6. Exercícios

### 6.1. Exercícios Conceituais

1 Por que Bellman-Ford precisa de exatamente $V-1$ iterações? O que acontece se pararmos antes? E se continuarmos além disso?

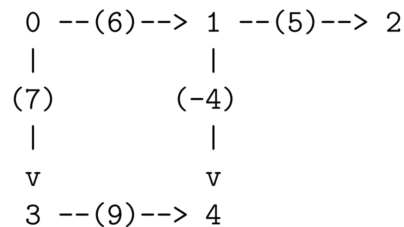
2 Compare Bellman-Ford com Dijkstra: quando cada um é preferível?

3 Explique por que no Floyd-Warshall a ordem dos loops ($k$ externo, depois $i$ e $j$) é crucial. O que acontece se mudarmos para $i, j, k$?

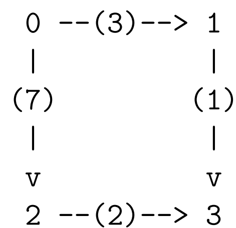
4 O que significa `dist[i][i] < 0` após o Floyd-Warshall? Como isso se relaciona com ciclos negativos?

## 6.2. Exercícios Analíticos

**1** Simule Bellman-Ford no grafo abaixo a partir do vértice 0. Mostre a tabela de distâncias após cada iteração.




**2** Simule Floyd-Warshall no grafo de 4 vértices:




Mostre a matriz $D$ após cada valor de $k$ (0, 1, 2, 3).

**3** Para um grafo com $V = 100$ vértices e $E = 500$ arestas, compare:

- Bellman-Ford (single source): quantas operações?
- Floyd-Warshall (all pairs): quantas operações?
- 100 execuções de Dijkstra: quantas operações?

Qual estratégia é melhor para obter todas as distâncias?

## 6.3. Exercícios de Programação

**1** Implemente Bellman-Ford com detecção de ciclos negativos e reconstrução de caminho via vetor de pais.

**2** Implemente Floyd-Warshall com reconstrução de caminhos usando a matriz `next[][]`.

**3** **Desafio:** Implemente um detector de arbitragem de moedas usando Bellman-Ford. Dada uma tabela de taxas de câmbio, determine se existe oportunidade de lucro.

**4** Resolva problemas de juiz online:

- Detecção de ciclos negativos (Bellman-Ford).
- All-pairs shortest path em grafos pequenos (Floyd-Warshall).
- Fecho transitivo (variante de Warshall).