AED2 - Algoritmos e Estr. de Dados II

Aula 21: Bellman-Ford e Floyd-Warshall

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. O Problema
2. Bellman-Ford
3. Floyd-Warshall
4. Comparativo e Exercícios
5. Aplicações Práticas

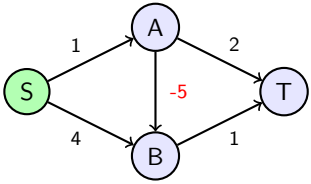
1. Quando o Dijkstra Falha?

O Dijkstra assume que, ao "selar" um vértice, nenhum caminho futuro pode melhorar sua distância. **Arestas negativas quebram essa premissa.**

1. Quando o Dijkstra Falha?

O Dijkstra assume que, ao "selar" um vértice, nenhum caminho futuro pode melhorar sua distância. **Arestas negativas quebram essa premissa.**

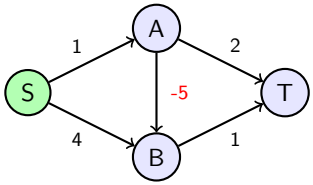
- **Pesos negativos:** Podem criar atalhos que o Dijkstra já descartou.



1. Quando o Dijkstra Falha?

O Dijkstra assume que, ao "selar" um vértice, nenhum caminho futuro pode melhorar sua distância. **Arestas negativas quebram essa premissa.**

- **Pesos negativos:** Podem criar atalhos que o Dijkstra já descartou.
- **Ciclo negativo:** Um ciclo fechado cuja soma dos pesos é < 0.

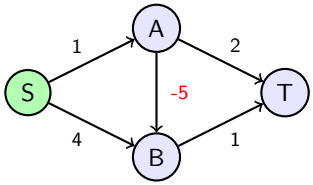


Dijkstra diz dist[T]=3 (S→A→T).
Mas S→A→B→T = $1 + (-5) + 1 = -3$**!**

1. Quando o Dijkstra Falha?

O Dijkstra assume que, ao "selar" um vértice, nenhum caminho futuro pode melhorar sua distância. **Arestas negativas quebram essa premissa.**

- **Pesos negativos:** Podem criar atalhos que o Dijkstra já descartou.
- **Ciclo negativo:** Um ciclo fechado cuja soma dos pesos é < 0.
- Isso permite reduzir o custo infinitamente (tendendo a $-\infty$). Se o custo é negativo, rodar no ciclo "dá lucro", tornando a busca por um "custo mínimo" impossível.



Dijkstra diz dist[T]=3 (S→A→T).
Mas S→A→B→T = $1 + (-5) + 1 = -3$**!**

2. O que representa um "Ciclo Negativo" na Prática?

Na vida real, um custo negativo geralmente representa um **ganho**. Logo, um ciclo negativo significa que ficar rodando nele gera um **ganho infinito**!

Exemplo 1: Veículos Elétricos e Regeneração de Bateria

Imagine um mapa onde os pesos das ruas são o **consumo de bateria**.

- Subidas gastam bateria (peso > 0). Descidas **recarregam** (peso < 0).
- Um **ciclo negativo** seria dar a volta no quarteirão e terminar com *mais* bateria do que começou. Rodar lá carregaria o carro de graça!

2. O que representa um "Ciclo Negativo" na Prática?

Na vida real, um custo negativo geralmente representa um **ganho**. Logo, um ciclo negativo significa que ficar rodando nele gera um **ganho infinito**!

Exemplo 1: Veículos Elétricos e Regeneração de Bateria

Imagine um mapa onde os pesos das ruas são o **consumo de bateria**.

- Subidas gastam bateria (peso > 0). Descidas **recarregam** (peso < 0).
- Um **ciclo negativo** seria dar a volta no quarteirão e terminar com *mais* bateria do que começou. Rodar lá carregaria o carro de graça!

Exemplo 2: Arbitragem Financeira (Câmbio de Moedas)

Imagine trocar seus Reais por Dólar, depois por Euro, e de volta para Reais.

- Devido a flutuações nas taxas de mercado, seus R$ 100 podem voltar R$ 105.
- Convertendo essas taxas para logaritmos, esse lucro vira um ciclo matemático negativo. Repetir o ciclo significa gerar **dinheiro infinito**!

Introdução: O Algoritmo de Bellman-Ford

Visão Geral

O algoritmo de Bellman-Ford é usado para encontrar caminhos mínimos de uma **única origem** para todos os outros vértices, sendo a principal alternativa ao Dijkstra quando o grafo possui **arestas com pesos negativos**.

Introdução: O Algoritmo de Bellman-Ford

Visão Geral

O algoritmo de Bellman-Ford é usado para encontrar caminhos mínimos de uma **única origem** para todos os outros vértices, sendo a principal alternativa ao Dijkstra quando o grafo possui **arestas com pesos negativos**.

- **Como ele trabalha?** Diferente do comportamento "guloso" do Dijkstra (que toma decisões definitivas e sela vértices), o Bellman-Ford **revisa** e tenta melhorar as distâncias de forma iterativa.

Introdução: O Algoritmo de Bellman-Ford

Visão Geral

O algoritmo de Bellman-Ford é usado para encontrar caminhos mínimos de uma **única origem** para todos os outros vértices, sendo a principal alternativa ao Dijkstra quando o grafo possui **arestas com pesos negativos**.

- **Como ele trabalha?** Diferente do comportamento "guloso" do Dijkstra (que toma decisões definitivas e sela vértices), o Bellman-Ford **revisa** e tenta melhorar as distâncias de forma iterativa.
- **Robustez (Força Bruta inteligente):** Ele tenta relaxar **todas as arestas** do grafo repetidas vezes. Assim, mesmo que um atalho com peso negativo seja descoberto mais tarde, a distância será corrigida.

Introdução: O Algoritmo de Bellman-Ford

Visão Geral

O algoritmo de Bellman-Ford é usado para encontrar caminhos mínimos de uma **única origem** para todos os outros vértices, sendo a principal alternativa ao Dijkstra quando o grafo possui **arestas com pesos negativos**.

- **Como ele trabalha?** Diferente do comportamento "guloso" do Dijkstra (que toma decisões definitivas e sela vértices), o Bellman-Ford **revisa** e tenta melhorar as distâncias de forma iterativa.
- **Robustez (Força Bruta inteligente):** Ele tenta relaxar **todas as arestas** do grafo repetidas vezes. Assim, mesmo que um atalho com peso negativo seja descoberto mais tarde, a distância será corrigida.
- **Segurança:** Sua maior vantagem é a capacidade de detectar a presença de **ciclos negativos**, avisando quando a distância mínima pode tender a $-\infty$.

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

- Após **1 iteração**: caminhos com ≤ 1 aresta estão corretos.

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

- Após **1 iteração**: caminhos com ≤ 1 aresta estão corretos.
- Após **2 iterações**: caminhos com ≤ 2 arestas estão corretos.

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

- Após **1 iteração**: caminhos com ≤ 1 aresta estão corretos.
- Após **2 iterações**: caminhos com ≤ 2 arestas estão corretos.
- $\vdots$

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

- Após **1 iteração**: caminhos com ≤ 1 aresta estão corretos.
- Após **2 iterações**: caminhos com ≤ 2 arestas estão corretos.
- $\vdots$
- Após $V - 1$ **iterações**: **todos os caminhos mínimos estão corretos!**

2. A Intuição: Por que $V - 1$ Iterações?

Observação Chave

Um caminho simples (sem repetir vértices) com V vértices tem **no máximo** $V - 1$ **arestas**.

- Após **1 iteração**: caminhos com ≤ 1 aresta estão corretos.
- Após **2 iterações**: caminhos com ≤ 2 arestas estão corretos.
- $\vdots$
- Após $V - 1$ **iterações**: **todos os caminhos mínimos estão corretos!**

Detecção de Ciclo Negativo

Se na iteração V (a extra) ainda for possível relaxar alguma aresta, existe um **ciclo negativo**.

3. Algoritmo de Bellman-Ford – Passo a Passo

3. Algoritmo de Bellman-Ford – Passo a Passo

1. **Inicialização:** `dist[S] = 0`, `dist[v] =` ∞ para $v \neq S$.

3. Algoritmo de Bellman-Ford – Passo a Passo

1. **Inicialização:** `dist[S] = 0`, `dist[v] =` ∞ para $v \neq S$.
2. **Repita** $V - 1$ **vezes:**
   - Para **cada aresta** (u, v, w) do grafo:
   - Se `dist[u] + w < dist[v]`, atualize `dist[v]`.

3. Algoritmo de Bellman-Ford – Passo a Passo

1. **Inicialização:** `dist[S] = 0`, `dist[v] =` ∞ para $v \neq S$.
2. **Repita** $V - 1$ **vezes:**
   - Para **cada aresta** (u, v, w) do grafo:
   - Se `dist[u] + w < dist[v]`, atualize `dist[v]`.
3. **Checagem extra:** Percorra todas as arestas mais uma vez. Se relaxar $\Rightarrow$ ciclo negativo!

3. Algoritmo de Bellman-Ford – Passo a Passo

1. **Inicialização:** `dist[S] = 0`, `dist[v] =` ∞ para $v \neq S$.
2. **Repita** $V - 1$ **vezes:**
   - Para **cada aresta** (u, v, w) do grafo:
   - Se `dist[u] + w < dist[v]`, atualize `dist[v]`.
3. **Checagem extra:** Percorra todas as arestas mais uma vez. Se relaxar $\Rightarrow$ ciclo negativo!

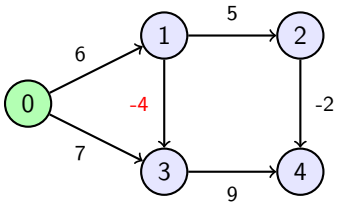
Diferença fundamental vs Dijkstra

- **Dijkstra:** Relaxa cada aresta **uma vez** (guloso, precisa de heap).
- **Bellman-Ford:** Relaxa **todas as arestas** $V - 1$ vezes (força bruta, apenas lista de arestas).

4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

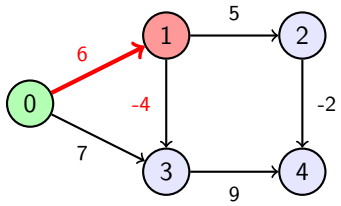
Aresta Testada	**d[0]**	**d[1]**	**d[2]**	**d[3]**	**d[4]**
Início	0	∞	∞	∞	∞



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

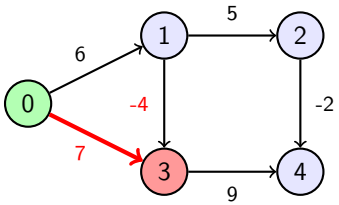
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

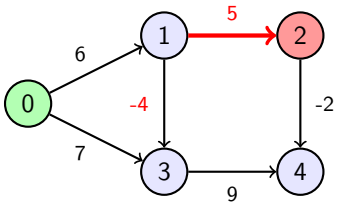
Aresta Testada	**d[0]**	**d[1]**	**d[2]**	**d[3]**	**d[4]**
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

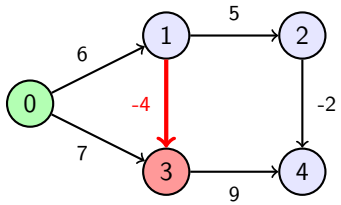
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞
3. (1→2, 5)	0	6	**11**	7	∞



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

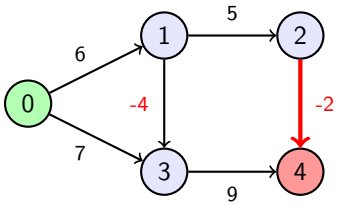
Aresta Testada	**d[0]**	**d[1]**	**d[2]**	**d[3]**	**d[4]**
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞
3. (1→2, 5)	0	6	**11**	7	∞
4. (1→3, -4)	0	6	11	**2**	∞



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

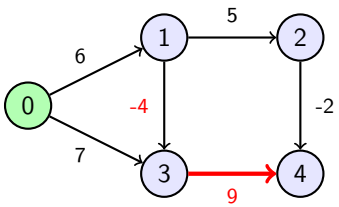
Aresta Testada	**d[0]**	**d[1]**	**d[2]**	**d[3]**	**d[4]**
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞
3. (1→2, 5)	0	6	**11**	7	∞
4. (1→3, -4)	0	6	11	**2**	∞
5. (2→4, -2)	0	6	11	2	**9**



4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

Aresta Testada	**d[0]**	**d[1]**	**d[2]**	**d[3]**	**d[4]**
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞
3. (1→2, 5)	0	6	**11**	7	∞
4. (1→3, -4)	0	6	11	**2**	∞
5. (2→4, -2)	0	6	11	2	**9**
6. (3→4, 9)	0	6	11	2	9 *(ignora)*



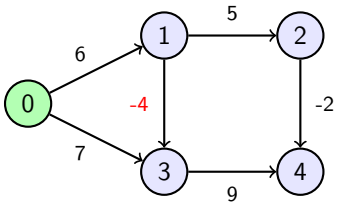
4. Bellman-Ford – Trace Animado (Iteração 1)

Origem = **0**. Relaxando as arestas passo a passo:

Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início	0	∞	∞	∞	∞
1. (0→1, 6)	0	**6**	∞	∞	∞
2. (0→3, 7)	0	6	∞	**7**	∞
3. (1→2, 5)	0	6	**11**	7	∞
4. (1→3, -4)	0	6	11	**2**	∞
5. (2→4, -2)	0	6	11	2	**9**
6. (3→4, 9)	0	6	11	2	*(ignora)*

Próximas Iterações (2 a 4)

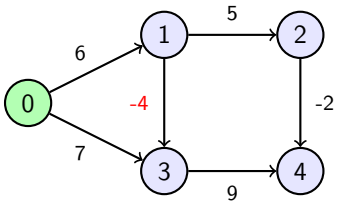
Ao repetir, as distâncias não mudam mais! O algoritmo pode parar precocemente (`early termination`).



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

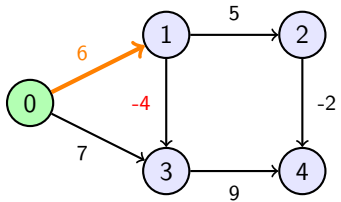
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início da Iter 2	0	6	11	2	9



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

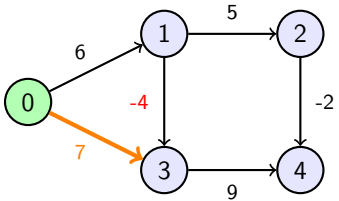
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]	
Início da Iter 2	0	6	11	2		9
1. (0→1, 6)	0	6	11	2	9	$(0 + 6 \not< 6)$



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

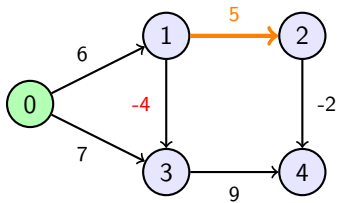
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início da Iter 2	0	6	11	2	9
1. (0→1, 6)	0	6	11	2	9 ($0+6 \not< 6$)
2. (0→3, 7)	0	6	11	2	9 ($0+7 \not< 2$)



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

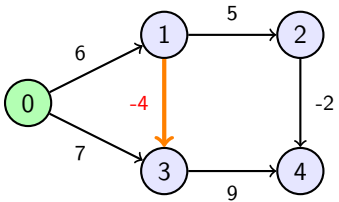
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]
Início da Iter 2	0	6	11	2	9
1. (0→1, 6)	0	6	11	2	9 $(0+6 \not< 6)$
2. (0→3, 7)	0	6	11	2	9 $(0+7 \not< 2)$
3. (1→2, 5)	0	6	11	2	9 $(6+5 \not< 11)$



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

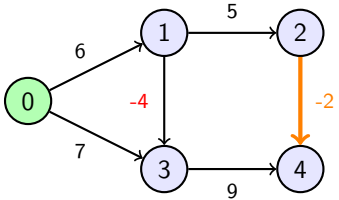
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]	
Início da Iter 2	0	6	11	2		9
1. (0→1, 6)	0	6	11	2	9	$(0+6 \not< 6)$
2. (0→3, 7)	0	6	11	2	9	$(0+7 \not< 2)$
3. (1→2, 5)	0	6	11	2	9	$(6+5 \not< 11)$
4. (1→3, -4)	0	6	11	2	9	$(6-4 \not< 2)$



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

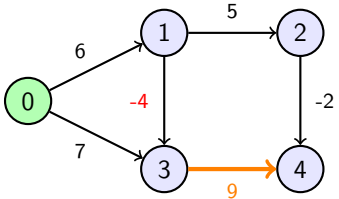
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]	
Início da Iter 2	0	6	11	2		9
1. (0→1, 6)	0	6	11	2	9	$(0+6 \not< 6)$
2. (0→3, 7)	0	6	11	2	9	$(0+7 \not< 2)$
3. (1→2, 5)	0	6	11	2	9	$(6+5 \not< 11)$
4. (1→3, -4)	0	6	11	2	9	$(6-4 \not< 2)$
5. (2→4, -2)	0	6	11	2	9	$(11-2 \not< 9)$



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

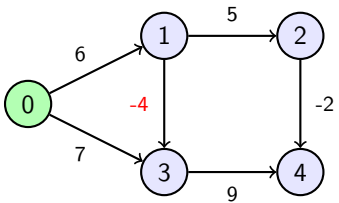
Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]	
Início da Iter 2	0	6	11	2		9
1. (0→1, 6)	0	6	11	2	9	$(0+6 \not< 6)$
2. (0→3, 7)	0	6	11	2	9	$(0+7 \not< 2)$
3. (1→2, 5)	0	6	11	2	9	$(6+5 \not< 11)$
4. (1→3, -4)	0	6	11	2	9	$(6-4 \not< 2)$
5. (2→4, -2)	0	6	11	2	9	$(11-2 \not< 9)$
6. (3→4, 9)	0	6	11	2	9	$(2+9 \not< 9)$



Trace Animado (Iteração 2: Verificação)

Na **Iteração 2**, não escolhemos vértices nem removemos ninguém da fila (não há fila!). Nós testamos **exatamente a mesma lista de arestas** de novo.

Aresta Testada	d[0]	d[1]	d[2]	d[3]	d[4]	
Início da Iter 2	0	6	11	2		9
1. (0→1, 6)	0	6	11	2	9	$(0+6 \not< 6)$
2. (0→3, 7)	0	6	11	2	9	$(0+7 \not< 2)$
3. (1→2, 5)	0	6	11	2	9	$(6+5 \not< 11)$
4. (1→3, -4)	0	6	11	2	9	$(6-4 \not< 2)$
5. (2→4, -2)	0	6	11	2	9	$(11-2 \not< 9)$
6. (3→4, 9)	0	6	11	2	9	$(2+9 \not< 9)$



Fim do Algoritmo

Como a tabela terminou idêntica, nenhuma aresta relaxou (`mudou = false`). Isso prova que já encontramos os caminhos mínimos!

5. Implementação Java – Bellman-Ford

```java
class Aresta { int u, v, w; }

public static int[] bellmanFord(int V, int src, List<Aresta> edges) {
    int[] dist = new int[V];
    Arrays.fill(dist, 999999999);
    dist[src] = 0;

    for (int i = 0; i < V - 1; i++) {
        boolean mudou = false; // Early termination
        for (Aresta e : edges) {
            if (dist[e.u] != 999999999 && dist[e.u] + e.w < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.w;
                mudou = true;
            }
        }
        if (!mudou) break; // Convergiu!
    }
    // Detectar Ciclo Negativo
    for (Aresta e : edges)
        if (dist[e.u] != 999999999 && dist[e.u] + e.w < dist[e.v])
            { System.out.println("Ciclo Negativo!"); return null; }
    return dist;
}
```

O Paradigma da Programação Dinâmica (PD)

O que é Programação Dinâmica?

É uma técnica fundamental no design de algoritmos. A ideia é quebrar um problema grande em **subproblemas menores e sobrepostos**, calculando a resposta de cada subproblema apenas uma vez e **salvando-a em uma tabela** (matriz/array) para uso futuro.

O Paradigma da Programação Dinâmica (PD)

O que é Programação Dinâmica?

É uma técnica fundamental no design de algoritmos. A ideia é quebrar um problema grande em **subproblemas menores e sobrepostos**, calculando a resposta de cada subproblema apenas uma vez e **salvando-a em uma tabela** (matriz/array) para uso futuro.

- **Aplicações:** Roteamento em redes, processamento de texto (distância de edição), problemas de otimização (mochila), etc.
- **Caminhos em Grafos:** Em vez de achar o caminho ótimo diretamente, a PD constrói o caminho perfeito **passo a passo**. Primeiro permitindo caminhos muito curtos/restritos e, a cada iteração, flexibilizando essas restrições.
- O **Algoritmo de Floyd-Warshall** resolve o problema de "Todos-para-Todos" usando exatamente essa essência da Programação Dinâmica!

Resolve **Todos-para-Todos**: distância entre todos os pares.

Ideia: Programação Dinâmica

A cada passo k, testamos se usar o vértice k como "ponte" melhora o caminho entre i e j.

Recorrência:

$$D[i][j] = \min \begin{cases} D[i][j] & (\text{não usa } k) \\ D[i][k] + D[k][j] & (\text{usa } k) \end{cases}$$

6. Floyd-Warshall: O Conceito

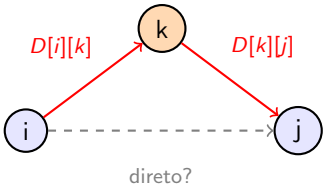
Resolve **Todos-para-Todos**: distância entre todos os pares.

Ideia: Programação Dinâmica

A cada passo k, testamos se usar o vértice k como "ponte" melhora o caminho entre i e j.

Recorrência:

$$D[i][j] = \min \begin{cases} D[i][j] & \text{(não usa } k\text{)} \\ D[i][k] + D[k][j] & \text{(usa } k\text{)} \end{cases}$$

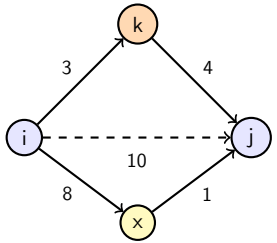


Passar por k é melhor que ir direto de i para j?

7. Entendendo o "Pulo" do Floyd-Warshall

O que significa iterar sobre k?

O algoritmo verifica se adicionar o vértice k no meio do caminho atual entre i e j resulta num atalho.



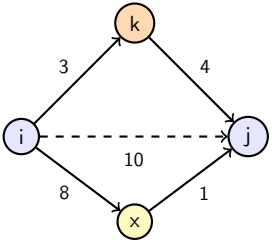
7. Entendendo o "Pulo" do Floyd-Warshall

O que significa iterar sobre k?

O algoritmo verifica se adicionar o vértice k no meio do caminho atual entre i e j resulta num atalho.

Estado Inicial:

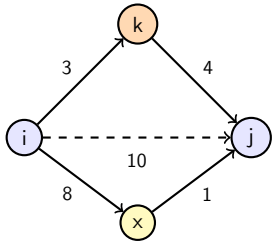
- $D[i][j]$ custa **10**.



7. Entendendo o "Pulo" do Floyd-Warshall

O que significa iterar sobre k?

O algoritmo verifica se adicionar o vértice k no meio do caminho atual entre i e j resulta num atalho.



Estado Inicial:

- $D[i][j]$ custa **10**.

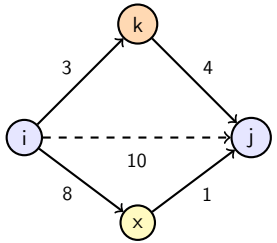
Quando analisa k:

- $i \to k$ custa 3. $k \to j$ custa 4.
- Novo custo: $3 + 4 = 7$.
- $7 < 10 \Rightarrow$ Atualiza $D[i][j] = 7$.

7. Entendendo o "Pulo" do Floyd-Warshall

O que significa iterar sobre k?

O algoritmo verifica se adicionar o vértice k no meio do caminho atual entre i e j resulta num atalho.



Estado Inicial:

- $D[i][j]$ custa **10**.

Quando analisa k:

- $i \to k$ custa 3. $k \to j$ custa 4.
- Novo custo: $3 + 4 = 7$.
- $7 < 10 \Rightarrow$ Atualiza $D[i][j] = 7$.

Quando analisa x (depois):

- $i \to x$ custa 8. $x \to j$ custa 1.
- Novo: $8 + 1 = 9$.
- $9 \not< 7 \Rightarrow$ **Ignora**, mantém 7.

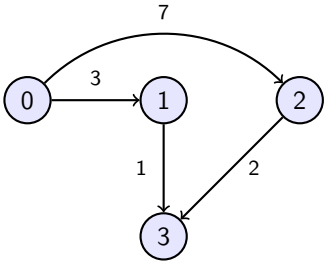
8. Implementação Java – Floyd-Warshall

```java
static final int INF = 999999999;

public static int[][] floydWarshall(int V, int[][] grafo) {
    int[][] dist = new int[V][V];
    // Inicialização
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = grafo[i][j];

    // k DEVE ser o loop externo!
    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];

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

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	3	7	∞
1	∞	0	∞	1
2	∞	∞	0	2
3	∞	∞	∞	0

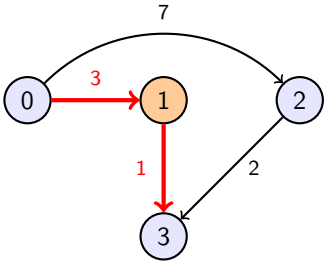


Testando atalhos (Pulos via vértice k)

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	**3**	7	∞
1	∞	0	∞	**1**
2	∞	∞	0	2
3	∞	∞	∞	0



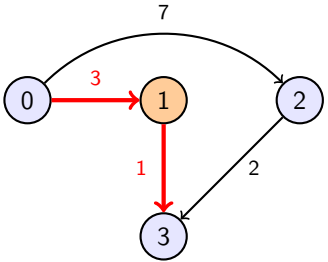
Testando atalhos (Pulos via vértice k)

- **Iteração** $k = 1$**:** $D[0][3] = \min(\infty, D[\mathbf{0}][\mathbf{1}] + D[\mathbf{1}][\mathbf{3}]) = \min(\infty, 3 + 1) = \mathbf{4}$.

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	3	7	**4**
1	∞	0	∞	1
2	∞	∞	0	2
3	∞	∞	∞	0



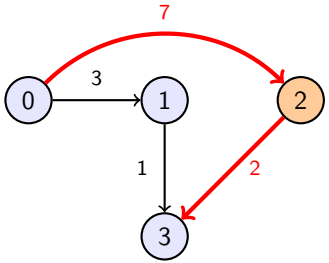
Testando atalhos (Pulos via vértice k)

- **Iteração** $k = 1$**:** $D[0][3] = \min(\infty, D[\mathbf{0}][\mathbf{1}] + D[\mathbf{1}][\mathbf{3}]) = \min(\infty, 3 + 1) = \mathbf{4}$.
- Encontramos um caminho! Atualiza a tabela $D[0][3] \rightarrow \mathbf{4}$.

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	3	**7**	4
1	∞	0	∞	1
2	∞	∞	0	**2**
3	∞	∞	∞	0



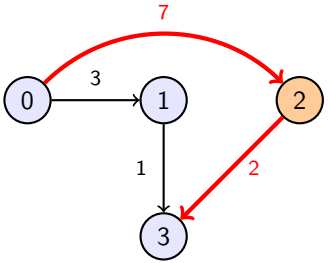
Testando atalhos (Pulos via vértice k)

- **Iteração** $k = 1$**:** $D[0][3] = \min(\infty, D[\mathbf{0}][\mathbf{1}] + D[\mathbf{1}][\mathbf{3}]) = \min(\infty, 3 + 1) = \mathbf{4}$.
- Encontramos um caminho! Atualiza a tabela $D[0][3] \rightarrow \mathbf{4}$.
- **Iteração** $k = 2$**:** $D[0][3] = \min(4, D[\mathbf{0}][\mathbf{2}] + D[\mathbf{2}][\mathbf{3}]) = \min(4, 7 + 2) = \mathbf{9}$.

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	3	7	4
1	∞	0	∞	1
2	∞	∞	0	2
3	∞	∞	∞	0



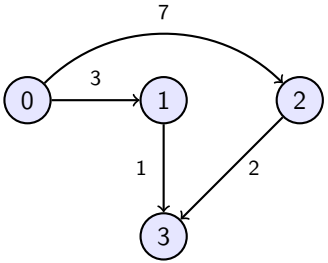
Testando atalhos (Pulos via vértice k)

- **Iteração** $k = 1$**:** $D[0][3] = \min(\infty, D[\mathbf{0}][\mathbf{1}] + D[\mathbf{1}][\mathbf{3}]) = \min(\infty, 3 + 1) = \mathbf{4}$.
- Encontramos um caminho! Atualiza a tabela $D[0][3] \rightarrow \mathbf{4}$.
- **Iteração** $k = 2$**:** $D[0][3] = \min(4, D[\mathbf{0}][\mathbf{2}] + D[\mathbf{2}][\mathbf{3}]) = \min(4, 7 + 2) = \mathbf{9}$.
- O caminho via 1 é melhor ($4 < 9$). A tabela **não muda**!

9. Floyd-Warshall – Trace Animado

Grafo: $0 \xrightarrow{3} 1$, $0 \xrightarrow{7} 2$, $1 \xrightarrow{1} 3$, $2 \xrightarrow{2} 3$. Matriz de distâncias $D[i][j]$:

	0	**1**	**2**	**3**
0	0	3	7	4
1	∞	0	∞	1
2	∞	∞	0	2
3	∞	∞	∞	0

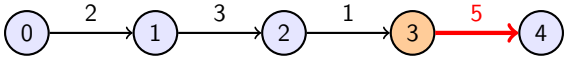


Testando atalhos (Pulos via vértice k)

- **Iteração** $k = 1$**:** $D[0][3] = \min(\infty, D[\mathbf{0}][\mathbf{1}] + D[\mathbf{1}][\mathbf{3}]) = \min(\infty, 3 + 1) = \mathbf{4}$.
- Encontramos um caminho! Atualiza a tabela $D[0][3] \rightarrow \mathbf{4}$.
- **Iteração** $k = 2$**:** $D[0][3] = \min(4, D[\mathbf{0}][\mathbf{2}] + D[\mathbf{2}][\mathbf{3}]) = \min(4, 7 + 2) = \mathbf{9}$.
- O caminho via 1 é melhor ($4 < 9$). A tabela **não muda**!

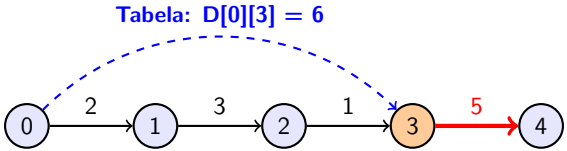
A Mágica da Programação Dinâmica no Floyd-Warshall

O ganho da Programação Dinâmica é **nunca recalcular o que já foi resolvido**.



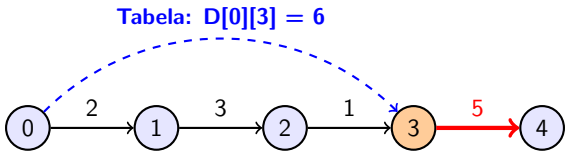
A Mágica da Programação Dinâmica no Floyd-Warshall

O ganho da Programação Dinâmica é **nunca recalcular o que já foi resolvido**.



A Mágica da Programação Dinâmica no Floyd-Warshall

O ganho da Programação Dinâmica é **nunca recalcular o que já foi resolvido**.

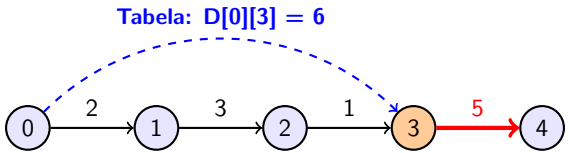


O Pulo do Gato (Reaproveitamento)

- Imagine que estamos na iteração $k = 3$, avaliando se vale a pena ir de $0 \to 4$ passando pelo vértice 3.
- A fórmula faz: $D[0][4] = \min(\ldots, \mathbf{D[0][3]} + D[3][4])$

A Mágica da Programação Dinâmica no Floyd-Warshall

O ganho da Programação Dinâmica é **nunca recalcular o que já foi resolvido**.

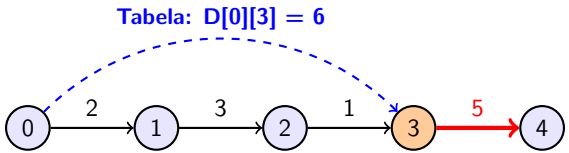


O Pulo do Gato (Reaproveitamento)

- Imagine que estamos na iteração $k = 3$, avaliando se vale a pena ir de $0 \to 4$ passando pelo vértice 3.
- A fórmula faz: $D[0][4] = \min(\ldots, \mathbf{D[0][3]} + D[3][4])$
- **A Mágica:** A tabela **já sabe** que $D[0][3]$ custa 6. Ela não precisa andar no grafo de novo e refazer as contas do trecho interno $(0 \to 1 \to 2 \to 3)$!

A Mágica da Programação Dinâmica no Floyd-Warshall

O ganho da Programação Dinâmica é **nunca recalcular o que já foi resolvido**.



O Pulo do Gato (Reaproveitamento)

- Imagine que estamos na iteração $k = 3$, avaliando se vale a pena ir de $0 \to 4$ passando pelo vértice 3.
- A fórmula faz: $D[0][4] = \min(\ldots, \mathbf{D[0][3]} + D[3][4])$
- **A Mágica:** A tabela **já sabe** que $D[0][3]$ custa 6. Ela não precisa andar no grafo de novo e refazer as contas do trecho interno $(0 \to 1 \to 2 \to 3)$!
- É uma **consulta instantânea** $\mathcal{O}(1)$. O trabalho de descobrir que o melhor atalho até o vértice 3 era via vértices 1 e 2 já foi resolvido nas iterações $k = 1$ e $k = 2$ e salvo na matriz.

10. Aplicação: Arbitragem de Moedas

O Problema

Um banco oferece câmbios: USD→EUR (0.9), EUR→GBP (0.85), GBP→USD (1.35).
Começando com \$1: $1 \times 0.9 \times 0.85 \times 1.35 = 1.0327$ — **lucro de 3.27%!**

10. Aplicação: Arbitragem de Moedas

O Problema

Um banco oferece câmbios: USD→EUR (0.9), EUR→GBP (0.85), GBP→USD (1.35).
Começando com \$1: $1 \times 0.9 \times 0.85 \times 1.35 = 1.0327$ — **lucro de 3.27%!**

Modelagem com Bellman-Ford

- Transforme taxa r em peso $-\log(r)$.
- Produto de taxas $\rightarrow$ soma de $-\log$ (propriedade do logaritmo).
- Ciclo com produto $> 1 \Leftrightarrow$ ciclo com soma negativa no grafo de logs.
- **Bellman-Ford detecta o ciclo negativo = arbitragem!**

11. Comparativo Geral: Qual Algoritmo Usar?

Algoritmo	**Tipo**	**Neg.?**	**Complexidade**	**Quando usar**
BFS	1→Todos	Não	$O(V + E)$	Pesos iguais
Dijkstra 0-1	1→Todos	Não	$O(V + E)$	Pesos 0 ou 1
Dijkstra (Heap)	1→Todos	Não	$O(E \log V)$	Caso geral
Bellman-Ford	1→Todos	**Sim**	$O(V \cdot E)$	Pesos negativos
Floyd-Warshall	Todos→Todos	**Sim**	$O(V^3)$	Grafos pequenos

11. Comparativo Geral: Qual Algoritmo Usar?

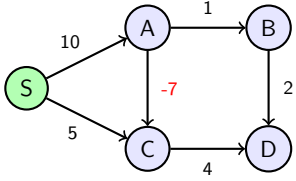
Algoritmo	Tipo	Neg.?	Complexidade	Quando usar
BFS	1→Todos	Não	$O(V+E)$	Pesos iguais
Dijkstra 0-1	1→Todos	Não	$O(V+E)$	Pesos 0 ou 1
Dijkstra (Heap)	1→Todos	Não	$O(E \log V)$	Caso geral
Bellman-Ford	1→Todos	**Sim**	$O(V \cdot E)$	Pesos negativos
Floyd-Warshall	Todos→Todos	**Sim**	$O(V^3)$	Grafos pequenos

Checklist de Decisão

1. Pesos ≥ 0 e uma origem? $\Rightarrow$ **Dijkstra**.
2. Pesos negativos? $\Rightarrow$ **Bellman-Ford**.
3. Todos os pares e $V \leq 500$? $\Rightarrow$ **Floyd-Warshall**.
4. Todos os pares e V grande? $\Rightarrow$ Rodar Dijkstra/BF V vezes.
5. Precisa detectar ciclo negativo? $\Rightarrow$ **Bellman-Ford** ou diagonal de Floyd.

Exercício 1: Trace Manual (Bellman-Ford)

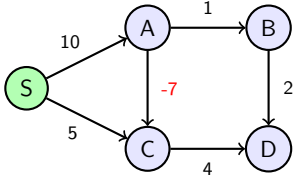
Aplique Bellman-Ford a partir do vértice **S** no grafo abaixo. Mostre `dist[]` após cada iteração.



1. Qual o `dist[]` final?
2. Quantas iterações foram necessárias?
3. Existe ciclo negativo?
4. Qual o caminho de S a D?

Exercício 1: Trace Manual (Bellman-Ford)

Aplique Bellman-Ford a partir do vértice **S** no grafo abaixo. Mostre `dist[]` após cada iteração.



1. Qual o `dist[]` final?
2. Quantas iterações foram necessárias?
3. Existe ciclo negativo?
4. Qual o caminho de S a D?

Resposta

`dist = [S:0, A:10, B:11, C:3, D:7]`.
Caminho S→D: S→A→C→D (custo $10 - 7 + 4 = 7$). Sem ciclo negativo.

Exercício 2: Floyd-Warshall Manual

Problema

Aplique Floyd-Warshall no grafo com 3 vértices:

Arestas: $0 \xrightarrow{4} 1$, $1 \xrightarrow{-2} 2$, $0 \xrightarrow{5} 2$.

Mostre a matriz de distâncias após cada valor de k (0, 1, 2).

Exercício 2: Floyd-Warshall Manual

Problema

Aplique Floyd-Warshall no grafo com 3 vértices:

Arestas: $0 \xrightarrow{4} 1$, $1 \xrightarrow{-2} 2$, $0 \xrightarrow{5} 2$.

Mostre a matriz de distâncias após cada valor de k (0, 1, 2).

Resposta

Inicial: $D[0][1] = 4$, $D[1][2] = -2$, $D[0][2] = 5$.

Após $k = 1$: $D[0][2] = \min(5, 4 + (-2)) = 2$. **Passar por 1 é melhor!**

Final: `D = [[0, 4, 2], [∞, 0, -2], [∞, ∞, 0]]`.

Exercício 3: Modelagem – Arbitragem

Problema

Dadas N moedas e uma tabela $N \times N$ de taxas de câmbio, determine se existe oportunidade de arbitragem (lucro cíclico).

1. Qual algoritmo usar?
2. Como transformar as taxas em pesos de grafo?
3. Qual condição indica arbitragem?

Exercício 3: Modelagem – Arbitragem

Problema

Dadas N moedas e uma tabela $N \times N$ de taxas de câmbio, determine se existe oportunidade de arbitragem (lucro cíclico).

1. Qual algoritmo usar?
2. Como transformar as taxas em pesos de grafo?
3. Qual condição indica arbitragem?

Resposta

Bellman-Ford com pesos $= -\log(\text{taxa})$.
Se detectar ciclo negativo $\Rightarrow$ existe arbitragem.
Alternativa: Floyd-Warshall e verificar se $D[i][i] < 0$.

Aplicações do Mundo Real: Bellman-Ford

Protocolo de Roteamento RIP (Redes)

O **RIP** (Routing Information Protocol) usa Bellman-Ford de forma distribuída. Roteadores trocam informações com os vizinhos para descobrir o caminho mais curto para redes distantes. Se um link cai, as distâncias são recalculadas iterativamente!

Aplicações do Mundo Real: Bellman-Ford

Protocolo de Roteamento RIP (Redes)

O **RIP** (Routing Information Protocol) usa Bellman-Ford de forma distribuída. Roteadores trocam informações com os vizinhos para descobrir o caminho mais curto para redes distantes. Se um link cai, as distâncias são recalculadas iterativamente!

Restrições de Dependência e Escalonamento

Imagine agendar tarefas para um processador ou em uma obra. Se a Tarefa B deve iniciar pelo menos 2 horas após a Tarefa A terminar, isso pode ser traduzido para desigualdades do tipo $t_B - t_A \geq 2$. O Bellman-Ford resolve esse sistema de restrições de diferença, e a existência de ciclos negativos indica que o agendamento é impossível!

Aplicações do Mundo Real: Floyd-Warshall

Fecho Transitivo (Algoritmo de Warshall)

Em compiladores e análise de dependências, queremos apenas saber: "Existe **qualquer caminho** de A a B?". Trocando as somas e o operador `min` por operadores lógicos booleanos (`AND` e `OR`), a matriz final nos dá essa conectividade transitiva com facilidade.

Aplicações do Mundo Real: Floyd-Warshall

Fecho Transitivo (Algoritmo de Warshall)

Em compiladores e análise de dependências, queremos apenas saber: "Existe **qualquer caminho** de A a B?". Trocando as somas e o operador `min` por operadores lógicos booleanos (`AND` e `OR`), a matriz final nos dá essa conectividade transitiva com facilidade.

Análise de Redes e Infraestrutura

- **Redes Sociais:** Calcular a proximidade e o grau de separação entre indivíduos para identificar quem atua como "ponte" de informações (Centralidade de Proximidade).
- **Logística:** Pré-processar as menores distâncias entre todas as cidades de uma malha para consultas instantâneas e otimizadas em tempo real.