Distância em grafos / digrafos ponderados

→ Dados um (di) grafo G, uma função $w: E(G) \to \mathbb{R}$ e dois vértices $u, v \in V(G)$:

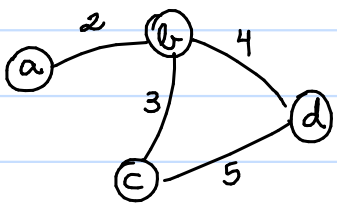
$$dist_G^w(u,v) = \begin{cases} \text{peso de um } uv\text{-caminho de menor peso} & \text{se há } uv\text{-cam.} \\ \infty & \text{c.c.} \end{cases}$$

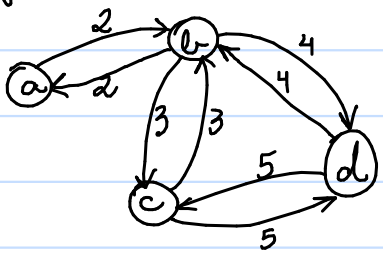
Caminhos mínimos

→ Um uv-caminho é mínimo se tem comprimento / peso igual à distância entre u e v.

Grafos ou digrafos?

→ Todo grafo G tem um digrafo associado $D(G)$





Um uv-caminho mínimo em G é mínimo em $D(G)$ e vice-versa.

→ Se G tem aresta negativa, $D(G)$ terá ciclo negativo.

→ Resolver os problemas de caminhos em qualquer digrafo (que não tenha ciclo negativo) implica em resolver em digrafos que são digrafos associados a grafos ⇒ consideraremos apenas digrafos

Problemas de interesse

Problema *Caminhos Mínimos de Única Fonte*

Entrada: $\langle D, w, s\rangle$, onde D é um digrafo, w é uma função de peso nos arcos e $s \in V(D)$.
Saída: Valor $\text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Problema *Caminhos Mínimos Entre Todos os Pares*

Entrada: $\langle D, w\rangle$, onde D é um digrafo e w é uma função de peso nos arcos.
Saída: Valor $\text{dist}_D^w(u, v)$ para todo par $u, v \in V(D)$.

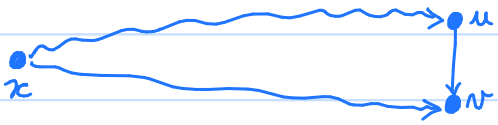
→ Qualquer algoritmo atual (e talvez que venha a ser criado) só resolve esses dois problemas se o digrafo não tiver ciclo com peso negativo.

Observações sobre caminhos mínimos

→ Se $P = (v_1, \ldots, v_k)$ é um v_1v_k-caminho mínimo, então qualquer subcaminho $(v_i, \ldots, v_j)$ de P é um v_iv_j-caminho mínimo.

→ Se uv é um arco, então para qualquer outro vértice x vale que

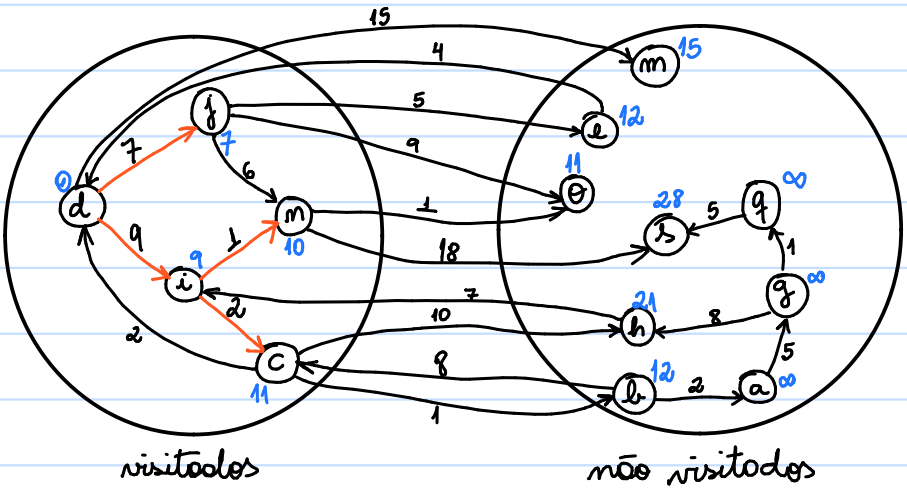
$$\text{dist}^w(x, v) \leq \text{dist}^w(x, u) + w(uv)$$



Algoritmo de Dijkstra

→ Recebe um digrafo D com pesos e $s \in V(D)$.

→ Executa de forma similar à dos algoritmos de busca.



Algoritmo de Dijkstra

→ Cada vértice v terá $v.visitado$, $v.pred$ e $v.dist$ (estimativa)

→ O subgrafo T com

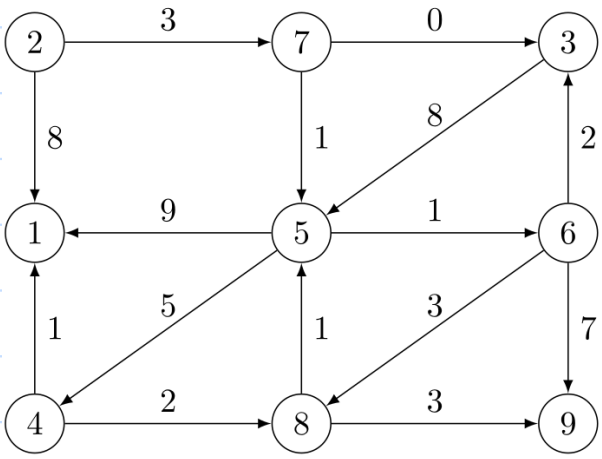
$$V(T) = \{s\} \cup \{ v \in V(D) : v.predecessor \neq NULL \}$$

$$E(T) = \{ (v.predecessor, v) : v \in V(T) \setminus \{s\} \}$$

é uma arborescência e contém um sv-caminho cujo peso é $v.distancia$ para cada $v \in V(T)$.

→ Só funciona <u>corretamente</u> se os pesos forem não-negativos.

Exemplo de execução



	1	2	3	4	5	6	7	8	9
visitado	0	0	0	0	0	0	0	0	0
pred	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
dist	∞	∞	∞	∞	∞	0	∞	∞	∞

Algoritmo de Dijkstra

DIJKSTRA(D, w, s)

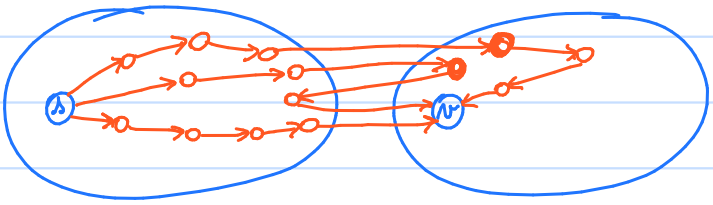
```
1  para todo vértice v ∈ V(D) faça
2      v.pred = null
3      v.dist = ∞
4      v.visitado = 0
5  s.dist = 0
6  enquanto houver vértice u com u.visitado == 0 e u.dist ≠ ∞ faça
7      seja x um vértice não visitado com menor valor x.dist
8      x.visitado = 1
9      para todo vértice y ∈ N⁺(x) faça
10         se y.visitado == 0 então
11             se x.dist + w(xy) < y.dist então
12                 y.dist = x.dist + w(xy)
13                 y.pred = x
```

relaxação do arco xy

TEOREMA: Dado um digrafo D, uma função $w: E(D) \to \mathbb{R}^+$ e um $s \in V(D)$, ao final de DIJKSTRA(D, w, s) vale que

$$v.dist = dist_D^w(s, v) \quad \forall\ v \in V(D).$$

IDEIA DA PROVA: Assim que v é visitado, vamos mostrar que $v.dist = dist_D^w(s,v)$ (uma vez visitado, isso não muda).
Vamos mostrar que qualquer outro sv-caminho tem peso $\geq v.dist$ usando a escolha gulosa: qualquer outro sv-caminho tem vértices não visitados, que poderiam ter sido escolhidos naquele momento



PROVA: Vamos começar mostrando que

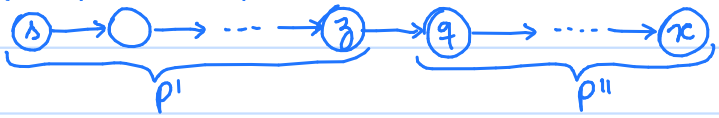
$R(t)$ = "Antes da t-ésima iteração começar, vale que $v.dist = dist_D^w(s,v)$ para todo $v \in V(D)$ com $v.visitado = 1$"

é uma invariante de laço por indução no número t de vezes que o teste do laço é executado.

BASE: Se $t = 1$, o teste executou uma vez. Antes da primeira iteração, nenhum vértice está visitado e portanto $R(1)$ vale trivialmente.
Durante a primeira iteração, claramente s é escolhido e visitado.
Como fizemos $s.dist = 0$ e $dist_D^w(s,s) = 0$, claramente $R(2)$ vale.

PASSO: Seja $t \geq 2$ qualquer e suponha que $R(t)$ vale. Estamos no início da t-ésima iteração e precisamos provar $R(t+1)$.
Seja x o vértice escolhido para ser visitado nessa iteração. ($x \neq s$ pois $t \geq 2$).
Como apenas $x.visitado$ será alterado, basta mostrar $x.dist = dist_D^w(s,x)$ para provar $R(t+1)$.

Seja P um sx-caminho qualquer e seja q o primeiro vértice de P que não está visitado (deve existir, pois $s \neq x$, s é visitado e x não é).
Seja z o vértice que precede q em P.



Pela escolha de q, z é visitado. Logo, por $P(t)$, $z.dist = dist_D^w(s,z)$.
Podemos escrever $w(P)$ como $w(P') + w(zq) + w(P'')$, onde P' é o subcaminho de s a z e P'' é o subcaminho de q a x.
Como P' só tem vértices visitados, $w(P') = dist_D^w(s,z) = z.dist$.
Também note que $w(P'') \geq 0$.

Então $dist_D^w(s,x) = w(P) = w(P') + w(zq) + w(P'') \geq z.dist + w(zq)$.

Quando z foi visitado, os arcos que saem de z foram relaxados, de modo que $q.dist \leq z.dist + w(zq)$.

Quando x foi escolhido, q era uma opção de escolha. Então vale que $x.dist \leq q.dist$.

Juntando tudo, temos $x.dist \leq w(P)$.
Como P é qualquer, o sx-caminho descoberto pelo algoritmo só pode ser ótimo, de forma que $x.dist = dist_D^w(s,x)$ e $R(t+1)$ vale.

Provamos, então, que R é invariante.

Como R é invariante, ao final do laço vale que todo vértice v visitado tem $v.dist = dist_s^w(s, v)$.
Se o laço terminou porque não há vértice não visitado, então o resultado da invariante prova o teorema diretamente.
Então todo vértice u com $u.visitado = 0$ tem $u.dist = \infty$.
Vamos provar que não há su-caminho, o que, junto com o resultado da invariante termina a prova do teorema.

Suponha que há su-caminho P e $u.visitado = 0$ ao fim.
Seja q o primeiro vértice não visitado em P e z seu predecessor.

Então z é visitado e $q.dist = \infty$.
Mas quando z foi visitado, o arco zq foi relaxado, contradição.
C.Q.D.

Qual o tempo de execução?

```
DIJKSTRA(D, w, s)
1  para todo vértice v ∈ V(D) faça
2      v.pred = null
3      v.dist = ∞
4      v.visitado = 0
5  s.dist = 0
6  enquanto houver vértice u com u.visitado == 0 e u.dist ≠ ∞ faça
7      seja x um vértice não visitado com menor valor x.dist
8      x.visitado = 1
9      para todo vértice y ∈ N⁺(x) faça
10         se y.visitado == 0 então
11             se x.dist + w(xy) < y.dist então
12                 y.dist = x.dist + w(xy)
13                 y.pred = x
```

Dá para melhorar?

→ A operação mais custosa é a de encontrar o vértice não visitado com menor campo dist

→ Vamos usar a estrutura de dados heap.
↳ Na heap, serão armazenados os vértices não visitados
↳ A prioridade de um vértice v será $-v.dist$

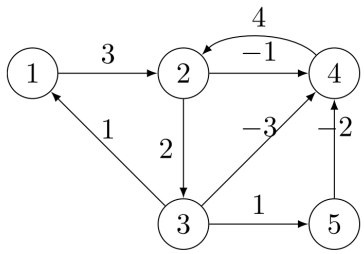
Algoritmo de Dijkstra com heap

DIJKSTRA-HEAP(D, w, s)

```
1  Seja H[1..v(D)] um vetor vazio
2  para todo vértice v ∈ V(D) faça
3      v.pred = null
4      v.prioridade = −∞
5      v.visitado = 0
6      INSERENAHEAP(H, v)
7  ALTERAHEAP(H, s.indice, 0)
8  enquanto u = CONSULTAHEAP(H) e u.prioridade ≠ −∞ faça
9      x = REMOVEDAHEAP(H)
10     x.visitado = 1
11     para todo vértice y ∈ N⁺(x) faça
12         se y.visitado == 0 então
13             se x.prioridade + (−w(xy)) > y.prioridade então
14                 ALTERAHEAP(H, y.indice, x.prioridade + (−w(xy)))
15                 y.pred = x
```

Problema ***Caminhos Mínimos Entre Todos os Pares***

Entrada: $\langle D, w\rangle$, onde D é um digrafo e w é uma função de peso nos arcos.
Saída: Valor $\text{dist}_D^w(u, v)$ para todo par $u, v \in V(D)$.



$\text{dist}_D^w(1,1) = 0$ $\quad$ $\text{dist}_D^w(2,1) = 3$
$\text{dist}_D^w(1,2) = 3$ $\quad$ $\text{dist}_D^w(2,2) = 0$
$\text{dist}_D^w(1,3) = 5$ $\quad$ $\text{dist}_D^w(2,3) = 2$
$\text{dist}_D^w(1,4) = 2$ $\quad$ $\text{dist}_D^w(2,4) = -1$
$\text{dist}_D^w(1,5) = 6$ $\quad$ $\text{dist}_D^w(2,5) = 3$

$\text{dist}_D^w(3,1) = 1$ $\quad$ $\text{dist}_D^w(4,1) = 7$ $\quad$ $\text{dist}_D^w(5,1) = 5$
$\text{dist}_D^w(3,2) = 1$ $\quad$ $\text{dist}_D^w(4,2) = 4$ $\quad$ $\text{dist}_D^w(5,2) = 2$
$\text{dist}_D^w(3,3) = 0$ $\quad$ $\text{dist}_D^w(4,3) = 6$ $\quad$ $\text{dist}_D^w(5,3) = 4$
$\text{dist}_D^w(3,4) = -3$ $\quad$ $\text{dist}_D^w(4,4) = 0$ $\quad$ $\text{dist}_D^w(5,4) = -2$
$\text{dist}_D^w(3,5) = 1$ $\quad$ $\text{dist}_D^w(4,5) = 7$ $\quad$ $\text{dist}_D^w(5,5) = 0$

Como resolver esse problema?

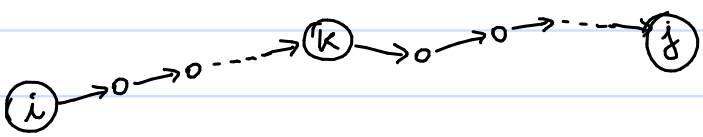
→ Seja $n = v(D)$ e $m = e(D)$.
→ Se os pesos forem todos não-negativos, podemos aplicar Dijkstra a partir de todos os vértices.
↳ Tempo: $n \times O((n+m)\log n) = O((n^2+mn)\log n)$
→ Se não, podemos aplicar Bellman-Ford a partir de todos os vértices.
↳ Tempo: $n \times \Theta(nm) = \Theta(n^2 m)$

Como resolver esse problema?

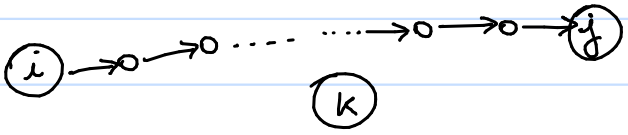
→ Vamos analisar o problema de construir um ij-caminho, com $i, j \in V(D)$ quaisquer.
↳ Note que basta apenas escolher os vértices internos agora

→ novo problema: dados $i, j \in V(D)$ e um conj. X de vértices que podem ser internos, como construir um ij-caminho?

→ Seja $k \in X$. Podemos usar k ou não como interno.



Se usamos, podemos construir um ik-caminho e um kj-caminho e uni-los.



Se não usamos, podemos construir um ij-caminho usando vértices de $X \setminus \{k\}$ como internos.

↳ Temos subproblemas!

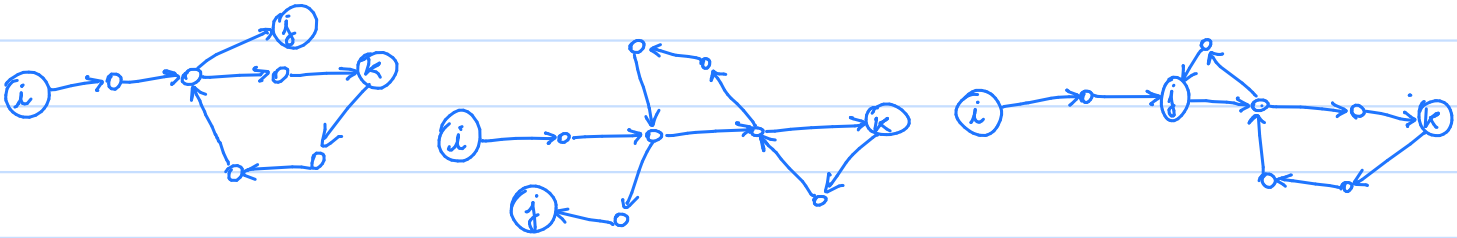
→ Usamos k ou não?
↳ São só duas opções! Teste ambos e fique com a melhor.
→ caso base: qual o menor tamanho de X para o qual é possível resolver o problema facilmente?

```
CAMINHO(D, w, i, j, X)
1 se |X| == 0 então
2     se ij ∈ E(D) então
3         devolve w(ij)
4     devolve ∞
5 Seja k ∈ X
6 nao_usa_k = CAMINHO(D, w, i, j, X \ {k})
7 usa_k = CAMINHO(D, w, i, k, X \ {k}) + CAMINHO(D, w, k, j, X \ {k})
8 devolve min{nao_usa_k, usa_k}
```

Abordagem recursiva

① Esse algoritmo constrói um ij-caminho?

Se o ik-caminho e o kj-caminho tiverem algum vértice em comum, a união deles não formará um caminho.



Mas então o ij caminho construído sem k terá peso menor.

↳ Isso se o digrafo não tiver ciclos de peso negativo!

① Qual k escolher?

Vamos considerar $V(D) = \{1, \dots, n\}$, com $n = v(D)$.

→ Podemos escolher $k \in N^+(i)$ com menor $w(ik)$?
Ou $k \in N^-(j)$ com menor $w(kj)$?

→ Não importa, pois todo vértice em X será escolhido em alguma chamada recursiva.

→ Para simplificar a implementação, vamos escolher o vértice que tiver o maior rótulo.

② Esse algoritmo é ótimo?

Defina $V^k = \{1, \dots, k\}$. Se P^k_{ij} é o peso de um ij-caminho mínimo cujos vértices internos pertencem a V^k, então

$$P^k_{ij} = \min \{ P^{k-1}_{ij}, P^{k-1}_{ik} + P^{k-1}_{kj} \} \quad \text{se } 1 \leq k \leq n$$

e

$$P^0_{ij} = \begin{cases} 0 & \text{se } i = j \\ w(ij) & \text{se } i \neq j \text{ e } ij \in E(D) \\ \infty & \text{se } i \neq j \text{ e } ij \notin E(D) \end{cases}$$

Isso vale pois caminhos mínimos contêm caminhos mínimos.

③ Quanto tempo ele leva?

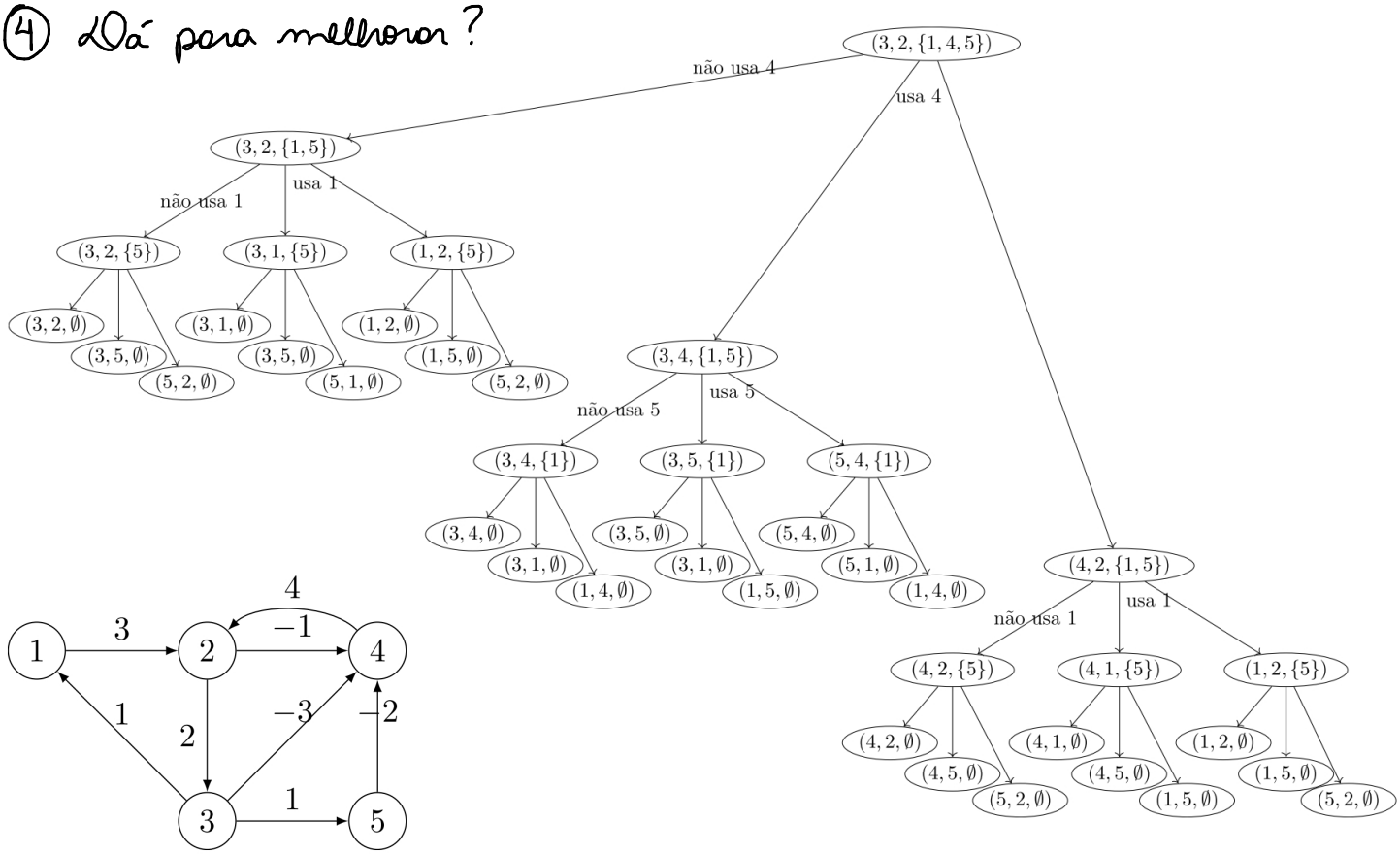
```
CAMINHO(D, w, i, j, X)
1 se |X| == 0 então
2     se ij ∈ E(D) então
3         devolve w(ij)
4     devolve ∞
5 Seja k ∈ X
6 nao_usa_k = CAMINHO(D, w, i, j, X \ {k})
7 usa_k = CAMINHO(D, w, i, k, X \ {k}) + CAMINHO(D, w, k, j, X \ {k})
8 devolve min{nao_usa_k, usa_k}
```

$T(0) = \Theta(1)$ (matriz)

$T(n) = 3T(n-1) + \Theta(1)$
$= \Theta(3^n)$

E ainda precisamos executá-lo para cada par de vértices $\Rightarrow O(n^2 3^n)$

④ Dá para melhorar?



→ Todo subproblema pode ser descrito por (i, j, K), que indica que queremos encontrar ij-caminho tendo K vértices disponíveis.
↳ Assim, $1 \leq i \leq n$, $1 \leq j \leq n$ e $0 \leq K \leq n$
→ Então uma matriz $D[1..n][1..n][0..n]$ consegue guardar todos
↳ Em $D[i][j][K]$ teremos o custo do menor ij-caminho que usa vértices em V^K.
↳ Queremos calcular $D[i][j][n]$ para todo $i, j \in V(D)$.

PD para Caminhos Mínimos entre Todos os Pares (Top-Down)

```
FLOYD-WARSHALL-TOPDOWN(D, w)
1  para i = 1 até n, incrementando faça
2      para j = 1 até n, incrementando faça
3          para k = 0 até n, incrementando faça
4              W[i][j][k] = ∞
5  para i = 1 até n, incrementando faça
6      para j = 1 até n, incrementando faça
7          W[i][j][n] = FLOYD-WARSHALLREC-TOPDOWN(D, w, n, i, j)
8  devolve W
```

```
FLOYD-WARSHALLREC-TOPDOWN(D, w, k, i, j)
1  se W[i][j][k] == ∞ então
2      se k == 0 então
3          se i == j então
4              W[i][j][0] = 0
5              j.pred[i] = i
6          senão se ij ∈ E(D) então
7              W[i][j][0] = w(ij)
8              j.pred[i] = i
9          senão
10             W[i][j][0] = ∞
11             j.pred[i] = null
12     senão
13         nao_usa_k = FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, i, j)
14         usa_k = FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, i, k) +
             FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, k, j)
15         se nao_usa_k < usa_k então
16             W[i][j][k] = nao_usa_k
17         senão
18             W[i][j][k] = usa_k
19             j.pred[i] = j.pred[k]
20 devolve W[i][j][k]
```

PD para Caminhos Mínimos entre Todos os Pares (Bottom-Up)

```
FLOYD-WARSHALL-BOTTOMUP(D, w)
1  Seja W[1..n][1..n][0..n] uma matriz
2  para i = 1 até n, incrementando faça
3      para j = 1 até n, incrementando faça
4          se i == j então
5              W[i][j][0] = 0
6              j.pred[i] = i
7          senão se ij ∈ E(D) então
8              W[i][j][0] = w(ij)
9              j.pred[i] = i
10         senão
11             W[i][j][0] = ∞
12             j.pred[i] = null
13 para k = 1 até n, incrementando faça
14     para i = 1 até n, incrementando faça
15         para j = 1 até n, incrementando faça
16             nao_usa_k = W[i][j][k − 1]
17             usa_k = W[i][k][k − 1] + W[k][j][k − 1]
18             se nao_usa_k < usa_k então
19                 W[i][j][k] = nao_usa_k
20             senão
21                 W[i][j][k] = usa_k
22                 j.pred[i] = j.pred[k]
23 devolve W
```

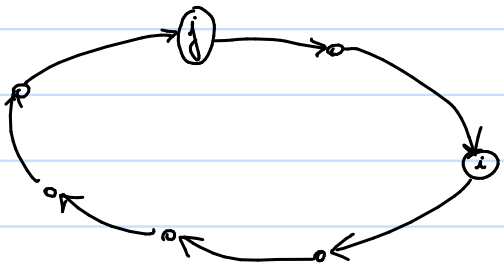
PD para Caminhos Mínimos entre Todos os Pares (Bottom-Up)

```
FLOYD-WARSHALL-MELHORADO(D, w)
1  Seja W[1..n][1..n] uma matriz
2  para i = 1 até n, incrementando faça
3      para j = 1 até n, incrementando faça
4          se i == j então
5              W[i][j] = 0
6              j.pred[i] = i
7          senão se ij ∈ E(D) então
8              W[i][j] = w(ij)
9              j.pred[i] = i
10         senão
11             W[i][j] = ∞
12             j.pred[i] = null
13 para k = 1 até n, incrementando faça
14     para i = 1 até n, incrementando faça
15         para j = 1 até n, incrementando faça
16             se W[i][j] > W[i][k] + W[k][j] então
17                 W[i][j] = W[i][k] + W[k][j]
18                 j.pred[i] = j.pred[k]
19 devolve W
```

Ciclos negativos

→ O algoritmo executa normalmente sobre dígrafos que possuem ciclos negativos.
↳ Porém ele não calcula distâncias corretamente

→ Se algum $W[i][i] < 0$ ao fim, existe ciclo negativo em D.



$W[i][j] + W[j][i]$

Problema do Caminho mínimo entre todos os pares

```
ResolveCaminhosEntreTodosPares(D, w)
1 W = Floyd-Warshall-BottomUp(D, w)
2 para i = 1 até v(G), incrementando faça
3     se W[i][i] < 0 então
4         devolve null
5 devolve W
```

Construindo caminhos

→ Se l é predecessor de j em um ij-caminho mínimo ($l = j.\text{pred}[i]$), então construa o il-caminho e adicione o arco lj ao fim.

```
ConstroiCaminho(D, i, j)
1 Seja L uma lista vazia
2 atual = j
3 enquanto atual ≠ i faça
4     InsereNoInicioLista(L, atual)
5     atual = atual.pred[i]
6 InsereNoInicioLista(L, i)
7 devolve L
```