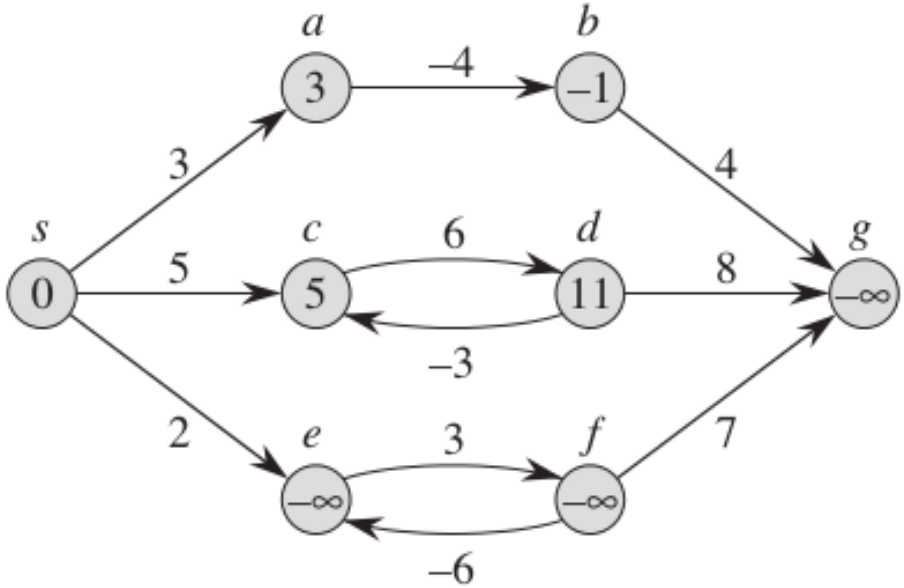
Distâncias Mínimas

Pedro Ribeiro

DCC/FCUP

2016/2017



Distâncias Mínimas

- Uma das aplicações mais típicas em grafos é o cálculo de **distâncias**. Descobrir o **caminho mais curto** entre dois nós de um grafo tem muita utilidade e pode ser usado numa grande variedade de situações
- Um exemplo é descobrir o caminho mais curto entre duas cidadas, dada um grafo representando as estradas disponíveis.
- Para **grafos não pesados** já vimos que uma solução possível é usar uma **pesquisa em largura** (BFS).
- Mas o que se pode usar para **grafos pesados**, sejam eles dirigidos ou não dirigidos?
  - Nota que BFS não funciona em grafos pesados porque o caminho mais curto até pode passar por mais arestas que um caminho mais longo.

Distâncias Mínimas

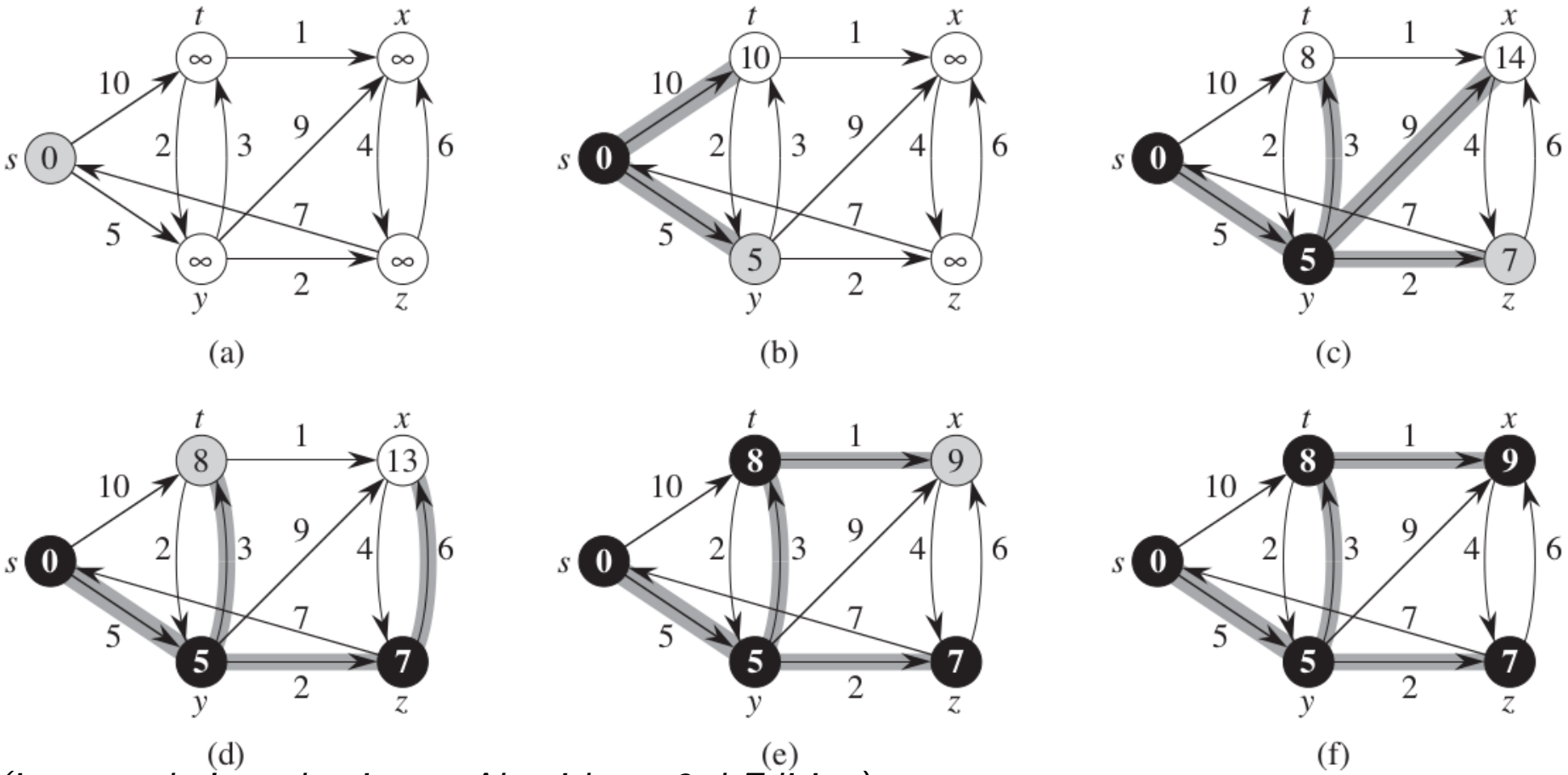
- Vamos falar essencialmente de dois problemas:
  - **SSSP** (*single-source shortest path problem*): descobrir o caminho mais curto entre um nó e todos os outros nós de um grafo.
  - **APSP** (*all-pairs shortest path problem*): descobrir o caminho mais curto entre todos os pares de nós de grafos.
- Pode parecer "estranho" à partida não falarmos do **caminho mais curto entre apenas um único par de nós**, mas o que é facto é que isto é tão difícil como calcular o SSSP, pois dado um caminho mais curto entre u e v, temos de ter o caminho mais curto para todos os nós intermédios desse caminho.

Algoritmo de Dijkstra

- Vamos começar por falar do **Algoritmo de Dijkstra**, para o **SSSP**.
- Este algoritmo serve para **grafos pesados e dirigidos**
  - Também funciona para grafos não dirigidos que são apenas um caso específico de grafos dirigidos
- A ideia principal do algoritmo de Dijkstra é ir "visitando" os nós por ordem crescente de distância ao nó origem.
- Isto é conseguido da seguinte maneira:
  (de certo modo semelhante ao Prim para MSTs)
  - Começar por inicializar a distância de todos os nós ao nó origem como sendo **infinito** e a distância do nó origem a si próprio como sendo **zero**.
  - Em cada passo **descobrir o nó u não processado à distância mínima (escolher melhor: choose_best)**.
  - Verificar se as **arestas do nó u que foi adicionado permitem obter uma nova distância mínima melhor a um nó v** ainda não visitado **relaxar os nós: relax)**.

Algoritmo de Dijkstra

- Vamos ver passo a passo para um grafo pequeno
- Estamos a descobrir os caminhos mínimos a partir do nó s
- Dentro dos nós estão as actuais distâncias mínimas. A cinzento estão as arestas que deram origem ao menor caminho.



(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Dijkstra

Vamos operacionalizar isto em código:

Algoritmo de Dijkstra para calcular distâncias mínimas a partir de s para todos os outros nós no grafo G

Dijkstra(G, s):
  Para todos os nós v de G **fazer**:
    $v.dist \leftarrow \infty$
    $v.visitado \leftarrow falso$
  $s.dist \leftarrow 0$
  Enquanto existirem nós não visitados **fazer**:
    Seleccionar nó u não visitado com menor valor de $dist$ // choose_best
    $u.visitado \leftarrow verdadeiro$
    Para cada aresta (u, v) de G **fazer**:
      Se $v.visitado = falso$ **e** $u.dist + peso(u, v) < v.dist$ **então**
        $v.dist \leftarrow u.dist + peso(u, v)$ // relax

Algoritmo de Dijkstra

Se quisermos saber mesmo o caminho e não só a distância, basta guardar os nós "predecessores" de cada nó (no final podemos reconstruir o caminho)

Algoritmo de Dijkstra para calcular distâncias mínimas a partir de s para todos os outros nós no grafo G - versão com predecessores

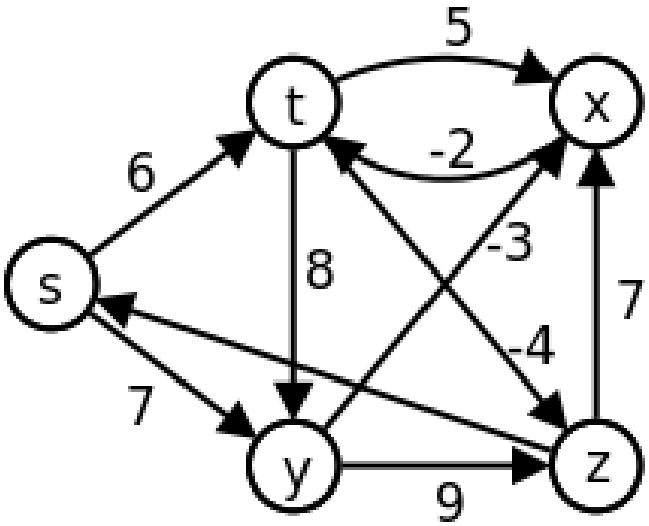
Dijkstra(G, s):
  Para todos os nós v de G **fazer**:
    $v.dist \leftarrow \infty$
    $v.visitado \leftarrow falso$
  $s.dist \leftarrow 0$
  $s.pred \leftarrow s$
  Enquanto existirem nós não visitados **fazer**:
    Seleccionar nó u não visitado com menor valor de $dist$ // choose_best
    $u.visitado \leftarrow verdadeiro$
    Para cada aresta (u, v) de G **fazer**:
      Se $v.visitado = falso$ **e** $u.dist + peso(u, v) < v.dist$ **então**
        $v.dist \leftarrow u.dist + peso(u, v)$ // relax
        $v.pred \leftarrow u$

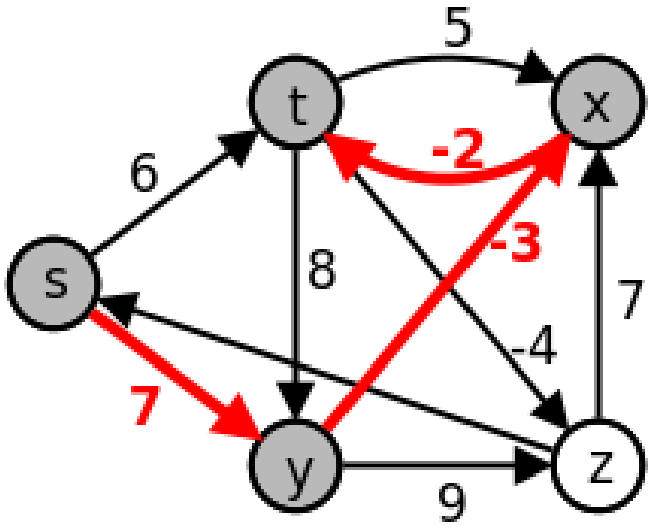
Algoritmo de Dijkstra

- Qual a **complexidade** do algoritmo de Dijkstra?
  - No início fazemos $\mathcal{O}(V)$ inicializações
  - Depois fazemos:
    - $\mathcal{O}(V)$ escolhas de nós mínimos (*choose_best*)
    - $\mathcal{O}(E)$ relaxamentos de arestas (*relax*)
- Gastamos $\mathcal{O}(|V| + |V| \times \textit{choose\_best} + |E|)$ (com lista de adjac.)
- Se usarmos a **forma "naive" de ciclo para descobrir o minimo**, ficamos com complexidade total $\mathcal{O}((|V| + |V|^2) + |E|)$. Como o número de arestas é no máximo $|V|^2$, a complexidade fica $\mathcal{O}(|V^2|)$.
- Podemos melhorar para $\mathcal{O}(|E| \log |V|)$ se usarmos uma **fila de prioridade** (ex: min-heap) para guardar as distâncias. Descobrir o mínimo ou relaxar uma aresta custariam $\mathcal{O}(\log |V|)$
- Tal como no caso do Prim, podemos optar por inserir novamente um elemento na heap com a nova distância, desde que tenhamos o cuidado de depois não processar novamente nós já visitados (cada nó será inserido no máximo tantas vezes quanto o seu grau).

Algoritmo de Dijkstra

- Porque é que o algoritmo de Dijkstra não funciona quando existem **pesos negativos**?



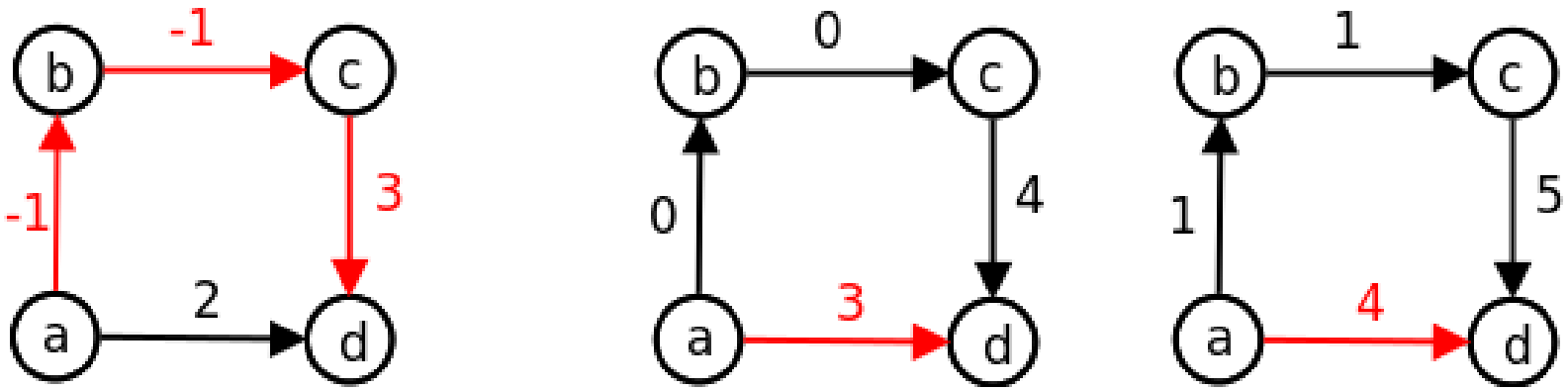


- Por exemplo no grafo indicado, o algoritmo de Dijkstra iria dizer que t está a distância 6, quando existe um caminho (indicado a vermelho) que está a distância 2!

- O problema é que os caminhos podem ficar com menor custo ao acrescentar arestas...

Algoritmo de Dijkstra

- Nota também que não é suficiente acrescentar uma constante a todas as arestas para que fiquem positivas! É que isto penaliza os caminhos de acordo com o número de arestas que têm



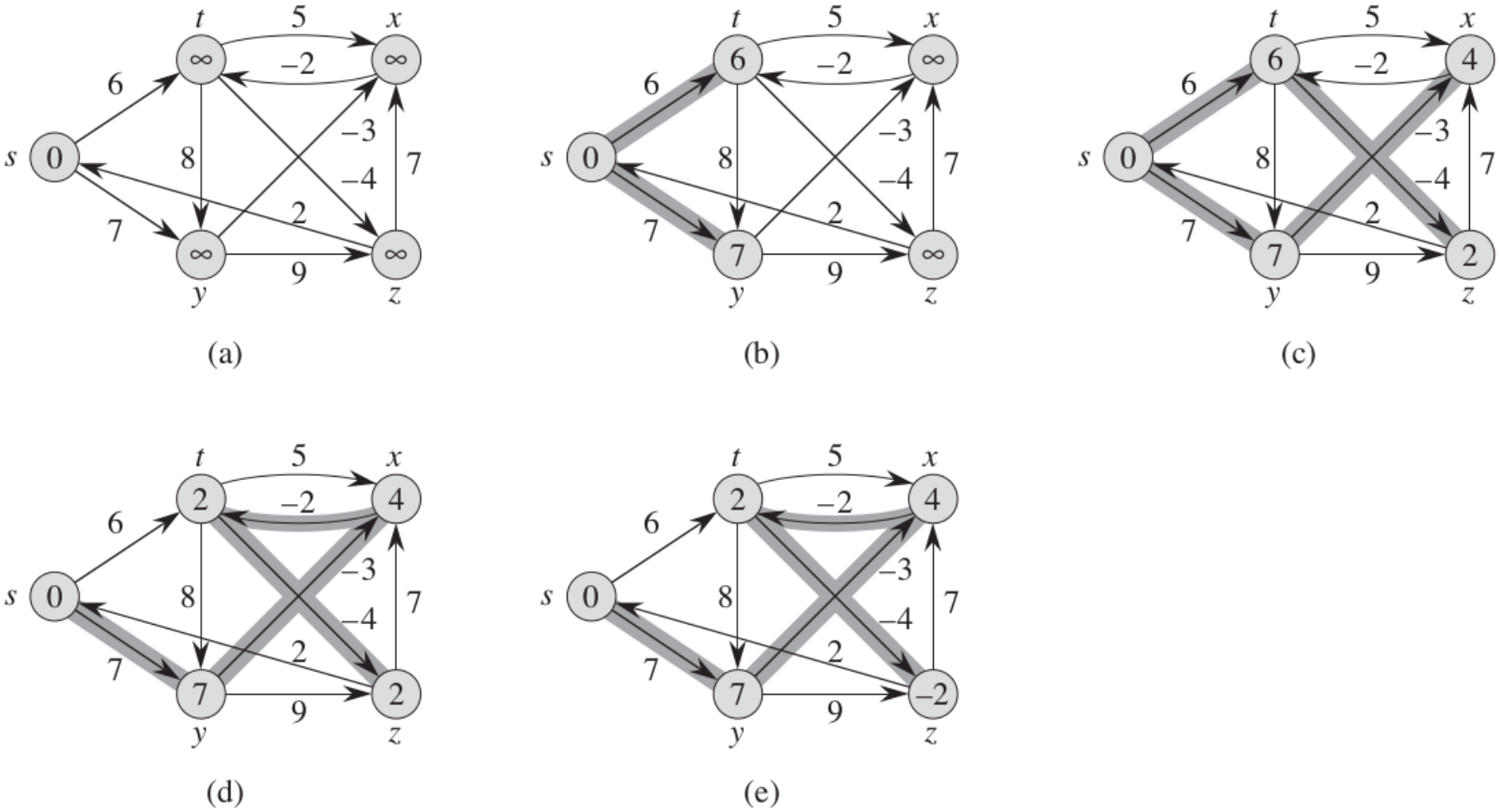
- Para o grafo original de cima, o melhor caminho entre a e d é $a \rightarrow b \rightarrow c \rightarrow d$ (com custo 1). Ao acrescentarmos uma constante a todas as arestas (na figura estão representadas as adições de 1 e de 2), esse caminho vai sofrer uma penalização de $3 \times c$, ao passo que o caminho $a \rightarrow d$ apenas sofre uma penalização de c, pelo que passa a ser esse o novo (e incorrecto) melhor caminho.

Algoritmo de Bellman-Ford

- Para resolver o problema com arestas de **pesos negativos**, podemos usar o algoritmo de **Bellman-Ford**
- Este algoritmo é uma versão mais genérica, mas também mais lenta
- A sua ideia é muito simples: relaxar todas as $|E|$ arestas $|V|-1$ vezes!
- Na sua essência, o Bellman-Ford usa **programação dinâmica**.
  - Depois de relaxar uma vez as arestas, os valores de $v.dist$ reflectem os melhores caminhos usando no máximo uma aresta.
  - Depois de relaxar i vezes as arestas, os valores de $v.dist$ reflectem os melhores caminhos usando no máximo i arestas.
  - Como um caminho simples (sem ciclos) só pode ter no máximo $|V|-1$ arestas, então ao fim de $|V|-1$ relaxamentos, todos os caminhos simples possíveis são tidos em conta!

Algoritmo de Bellman-Ford

Vamos ver um exemplo passo a passo:



(imagem de Introduction to Algorithms, 3rd Edition)

Algoritmo de Bellman-Ford

Vamos operacionalizar isto em código:

Algoritmo de Bellman-Ford para calcular distâncias mínimas a partir de s para todos os outros nós no grafo G

Bellman-Ford(G, s):
  Para todos os nós v de G **fazer**:
    $v.dist \leftarrow \infty$
  $s.dist \leftarrow 0$
  Para $i \leftarrow 1$ **até** $|V| - 1$ **fazer**:
    Para todas as arestas (u, v) de G **fazer**:
      Se $u.dist + peso(u, v) < v.dist$ **então**
        $v.dist \leftarrow u.dist + peso(u, v)$

- A complexidade fica $\mathcal{O}(|V| \cdot |E|)$ se usarmos uma lista de adjacências, ou $\mathcal{O}(V^3)$ se usarmos matriz de adjacências.

Algoritmo de Bellman-Ford

Tal como no Dijkstra, se precisarmos de saber o caminho em si, basta guardar os predecessores:

Algoritmo de Bellman-Ford para calcular distâncias mínimas a partir de s para todos os outros nós no grafo G - versão com predecessores

Bellman-Ford(G, s):
  Para todos os nós v de G **fazer**:
    $v.dist \leftarrow \infty$
  $s.dist \leftarrow 0$
  $s.pred \leftarrow s$
  Para $i \leftarrow 1$ **até** $|V| - 1$ **fazer**:
    Para todas as arestas (u, v) de G **fazer**:
      Se $u.dist + peso(u, v) < v.dist$ **então**
        $v.dist \leftarrow u.dist + peso(u, v)$
        $v.pred \leftarrow u$

Algoritmo de Bellman-Ford

Se quisermos saber se há ciclos negativos basta relaxar mais uma vez todas as arestas:

- Se alguma distância for melhorada então garantidamente temos um ciclo negativo (pois todos os caminhos "simples", sem ciclos, já tinham sido considerados)
- Se nenhuma distância mudou, não existem ciclos negativos

Detectar ciclos negativos depois de executar o Bellman-Ford

Bellman-Ford(G, s):

/* Executar Bellman-Ford como nos slide anteriores */

(..)

Para todas as arestas (u, v) de G **fazer**:

Se $u.dist + peso(u, v) < v.dist$ **então**

erro("Existe ciclo negativo!")

Menor caminho entre todos os pares de nós

- Como resolver o APSP? (*all-pairs shortest path problem*)
- Uma solução "trivial" seria usar um algoritmo de SSSP e executá-lo a partir de todos os nós:
  - Dijkstra (sem heaps): $\mathcal{O}(|V^3|)$
  - Dijkstra (com heaps): $\mathcal{O}(|V| \cdot |E| \log |V|)$
  - Bellman-Ford: $\mathcal{O}(|V^2| \cdot |E|)$ (mas funciona com pesos negativos)
- Existe um algoritmo $\mathbf{O(|V|^3)}$ que é muito fácil de implementar e é mais rápido do que um Dijkstra sem heaps pelo facto de ter um "factor constante" mais baixo.

Algoritmo de Floyd-Warshall

Algoritmo de Floyd-Warshall

Floyd-Warshall(G):
  Seja $dist[][]$ uma matriz $|V| \times |V|$ inicializada com ∞
  Para cada vértice v de G **fazer**:
    $dist[v][v] \leftarrow 0$
  Para todas as arestas (u, v) de G **fazer**:
    $dist[u][v] \leftarrow peso(u, v)$

  Para $k \leftarrow 1$ **até** $|V|$ **fazer**:
    Para $i \leftarrow 1$ **até** $|V|$ **fazer**:
      Para $j \leftarrow 1$ **até** $|V|$ **fazer**:
        Se $dist[i][k] + dist[k][j] < dist[i][j]$ **então**
          $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$

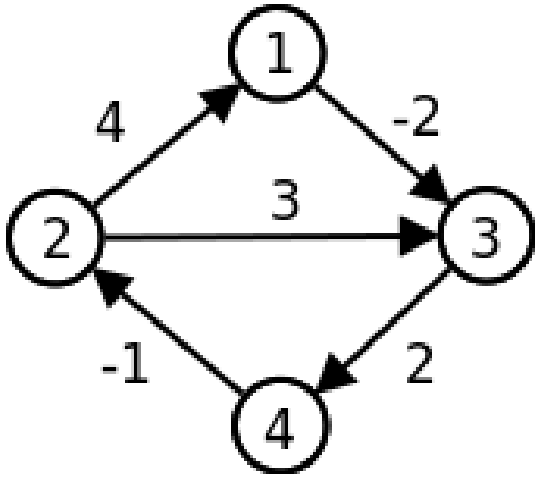
- A complexidade é trivialmente $\mathcal{O}(|V|^3)$ - ver os 3 ciclos!

Algoritmo de Floyd-Warshall

- Tal como o Bellman-Ford, o Floyd-Warshall usa ideias de **programação dinâmica**.
  - No início $dist[][]$ só tem em conta os caminhos directos (usando uma aresta do grafo)
  - No final da primeira iteração (com $k = 1$), tem em conta todos os caminhos directos ou que usem o nó 1 como ponto intermédio
  - No final de i iterações (com $k \leq i$), tem em conta todos os caminhos directos ou que usem quaisquer nós $\leq i$
  - Quando chegamos ao final, todos os caminhos possíveis são tidos em conta!
- Se existir um **ciclo negativo**, vamos ter uma entrada $dist[v][v]$ com valor negativo durante a execução do algoritmo.

Algoritmo de Floyd-Warshall

- Vejamos uma execução do algoritmo:

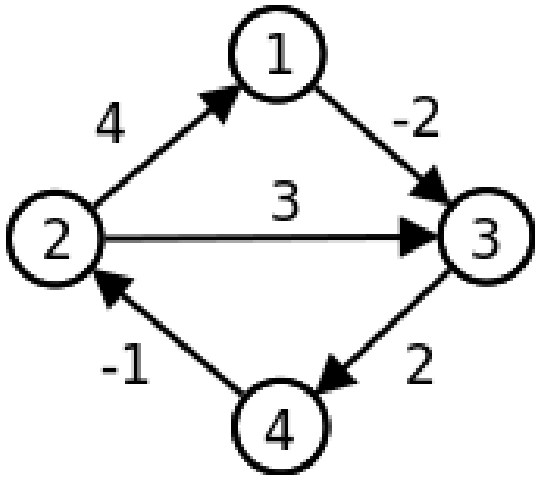


Inicialmente temos a seguinte matriz de distâncias:

	1	2	3	4
1	0	Inf	-2	Inf
2	4	0	3	Inf
3	Inf	Inf	0	2
4	Inf	-1	Inf	0

Algoritmo de Floyd-Warshall

- Vejamos uma execução do algoritmo:

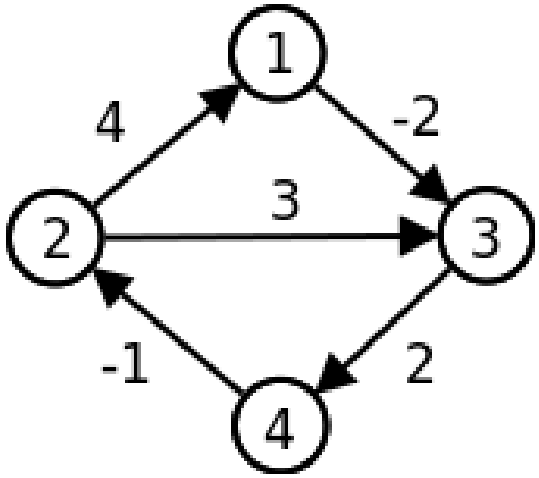


Ao passarmos por $k = 1$ actualizamos os caminhos que passam por 1:
$2 \rightarrow \mathbf{1} \rightarrow 3$

	1	**2**	**3**	**4**
1	0	Inf	-2	Inf
2	4	0	2	Inf
3	Inf	Inf	0	2
4	Inf	-1	Inf	0

Algoritmo de Floyd-Warshall

- Vejamos uma execução do algoritmo:

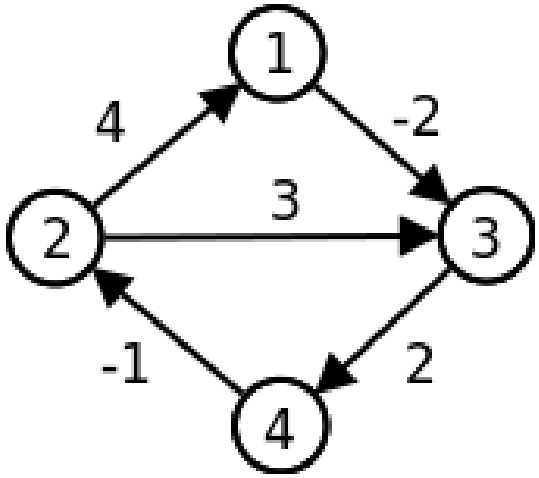


$k = 2$, actualizamos os caminhos que passam por 1 e 2:
$4 \rightarrow \mathbf{2} \rightarrow 1$
$4 \rightarrow \mathbf{2} \rightarrow 1 \rightarrow 3$

	1	**2**	**3**	**4**
1	0	Inf	-2	Inf
2	4	0	2	Inf
3	Inf	Inf	0	2
4	3	-1	1	0

Algoritmo de Floyd-Warshall

- Vejamos uma execução do algoritmo:

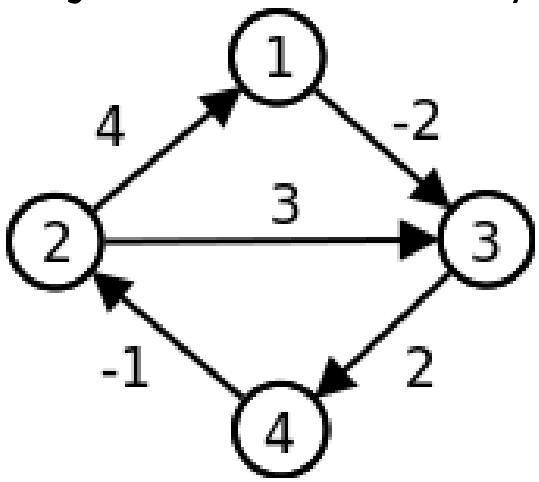


$k = 3$, actualizamos os caminhos que passam por 1, 2 e 3:
$1 \rightarrow \mathbf{3} \rightarrow 4$
$2 \rightarrow 1 \rightarrow \mathbf{3} \rightarrow 4$

	1	**2**	**3**	**4**
1	0	Inf	-2	0
2	4	0	2	4
3	Inf	Inf	0	2
4	3	-1	1	0

Algoritmo de Floyd-Warshall

- Vejamos uma execução do algoritmo:



$k = 4$, actualizamos os caminhos que passam por 1, 2, 3 e 4:

$3 \rightarrow \mathbf{4} \rightarrow 2$

$3 \rightarrow \mathbf{4} \rightarrow 2 \rightarrow 1$

$1 \rightarrow 3 \rightarrow \mathbf{4} \rightarrow 2$

	1	**2**	**3**	**4**
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

Algoritmo de Floyd-Warshall

- O algoritmo de Floyd-Warshall foi também criado a pensar no **fecho transitivo** de um grafo
- O fecho transitivo implica saber se **existe ou não um caminho (seja ele qual for) entre um qualquer par de nós.**
- É equivalente a executar a versão do Floyd de distâncias e verificar quais ficaram diferentes de ∞

Algoritmo de Floyd-Warshall

Algoritmo de Floyd-Warshall - Versão fecho transitivo

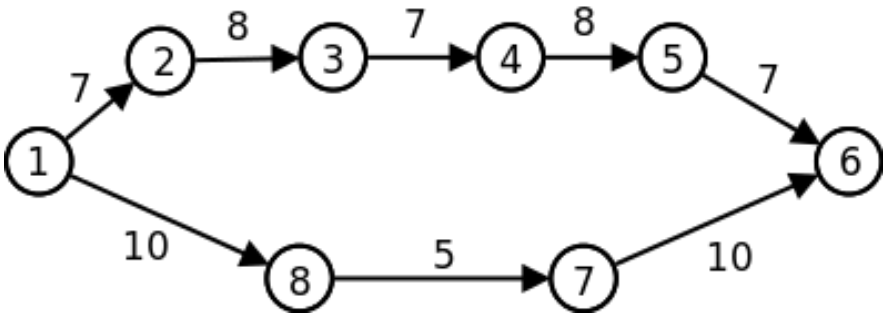
Floyd-Warshall(G):
  Seja *connected*[][] uma matriz booleana $|V| \times |V|$ inicializada a falsos
  Para cada vértice v de G **fazer**:
    connected$[v][v] \leftarrow$ *verdadeiro*
  Para todas as arestas (u, v) de G **fazer**:
    connected$[u][v] \leftarrow$ *verdadeiro*

  Para $k \leftarrow 1$ **até** $|V|$ **fazer**:
    Para $i \leftarrow 1$ **até** $|V|$ **fazer**:
      Para $j \leftarrow 1$ **até** $|V|$ **fazer**:
        Se *connected*$[i][k]$ **e** *connected*$[k][j]$ **então**
          connected$[i][j] \leftarrow$ *verdadeiro*

- A complexidade é novamente $\mathcal{O}(|V|^3)$

Variações

- Existem muitas possíveis **variações** de problemas de distâncias
- Para os resolver é necessário **adequar a parte do relaxamento das arestas (ou a parte de usar um k como nó intermédio)**
- Vejamos como exemplo as distâncias **maximin**: quero o caminho entre u e v que maximize o menor custo que aparece no caminho
  - Ex. de aplicação: peso significa "grau de segurança" e quero o caminho que me garanta mais segurança, mesmo que demore mais a chegar.



Caminho $1 \to 2 \to 3 \to 4 \to 5 \to 6$: peso mínimo $= 7$ (escolhia este)
Caminho $1 \to 8 \to 7 \to 6$: peso mínimo $= 5$

Variações

- Como modificar por exemplo o Dijkstra para **maximin**?
- Vamos assumir que o grafo não tem pesos negativos.
- A ideia é visitar os nós por ordem decrescente de distância **maximin**!

Algoritmo de Dijkstra - versão maximin

Dijkstra(G, s):
  Para todos os nós v de G **fazer**:
    $v.dist \leftarrow -1$
    $v.visitado \leftarrow falso$
  $s.dist \leftarrow \infty$
  Enquanto existirem nós não visitados **fazer**:
    Seleccionar nó u não visitado com maior valor de $dist$ // choose_best
    $u.visitado \leftarrow verdadeiro$
    Para cada aresta (u, v) de G **fazer**:
      Se $v.visitado = falso$ **e** $min(u.dist, peso(u, v)) > v.dist$ **então**
        $v.dist \leftarrow min(u.dist, peso(u, v))$ // relax