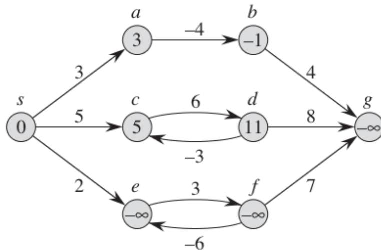
Distâncias Mínimas

Pedro Ribeiro

DCC/FCUP

2020/2021

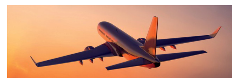


Distâncias Mínimas

- 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:

Descobrir o melhor caminho entre duas localizações, dado um grafo representando as estradas disponíveis (mais curto? com menos trânsito? que custe menos dinheiro?).

Grafo pode representar voos

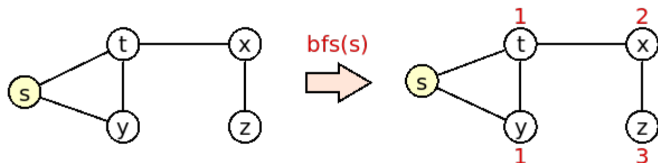


Grafo pode representar um mapa num jogo de computador

E tantas outras coisas mais...

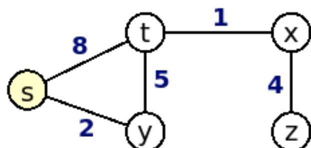
Distâncias Mínimas e BFS

- Para **grafos não pesados** já vimos que uma solução possível é usar uma **pesquisa em largura** (*BFS*).



[a vermelho as distâncias do nós a s]

- Para **grafos pesados** o BFS não funciona porque o caminho mais curto pode passar por mais arestas que um caminho mais longo.



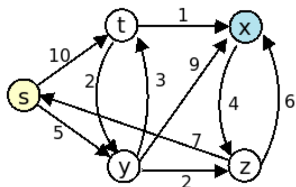
O melhor caminho de s para t não é apenas de uma aresta (de custo 8), mas de duas arestas com custo total 7 (5+2)

Distâncias Mínimas em Grafos Pesados

- O que se pode então usar para **grafos pesados**, sejam eles dirigidos ou não dirigidos?

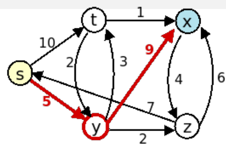
- Comecemos por ver um exemplo de uma instância do problema, para ganhar mais alguma intuição.

  Considere o seguinte grafo e imagine que quer descobrir o **caminho mais curto de s para x**

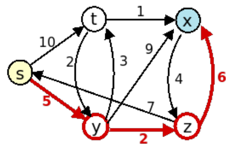


Existem vários caminhos possíveis...

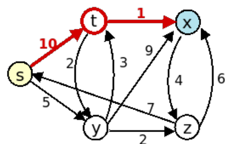
Distâncias Mínimas em Grafos Pesados - Exemplo



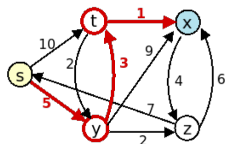
Custo total: 14 = 5 + 9



Custo total: 13 = 5 + 2 + 6



Custo total: 11 = 10 + 1



Custo total: 9 = 5 + 3 + 1 **(é este o melhor)**

Distâncias Mínimas

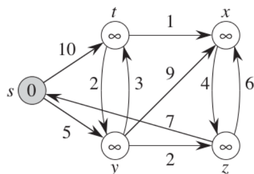
- Vamos falar essencialmente de dois problemas:
  - **SSSP** (*single-source shortest path problem*): descobrir o caminho mais curto entre um nó e todos os outros nós de um grafo.
  - **APSP** (*all-pairs shortest path problem*): descobrir o caminho mais curto entre todos os pares de nós de um grafo.
- 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

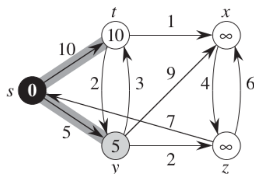
- Vamos começar por falar do **Algoritmo de Dijkstra**, para o **SSSP**.
- Este algoritmo serve para **grafos pesados e dirigidos**
  - Também funciona para grafos **não dirigidos** que são apenas um caso específico de grafos dirigidos
  - Também funciona para grafos **não pesados** que são apenas um caso específico de grafos pesados (todos os pesos $= 1$)
    [mas nesse caso é mais eficiente usar BFS]
  - **Não funciona** no caso de existirem **pesos negativos**
- 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:
  - 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

Algoritmo de Dijkstra

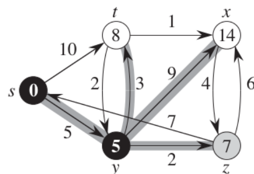
- Vamos ver passo a passo para um grafo pequeno
- Estamos a descobrir os caminhos mínimos a partir do nó s
- Dentro dos nós estão as actuais distâncias mínimas. A cinzento estão as arestas que deram origem ao menor caminho.



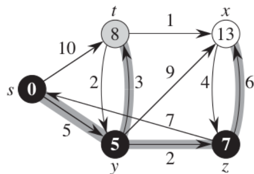
(a)



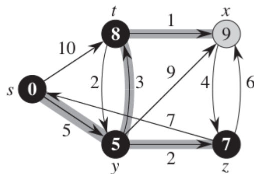
(b)



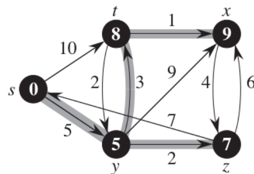
(c)



(d)



(e)



(f)

(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**
                $v.dist \leftarrow u.dist + peso(u, v)$ // relaxamento de uma aresta

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**
                $v.dist \leftarrow u.dist + peso(u, v)$ // relaxamento de uma aresta
                $v.pred \leftarrow u$

Algoritmo de Dijkstra - Complexidade

- 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 (*relaxamentos de arestas*)

- Gastamos $\mathcal{O}(|V| + |V| \times \textit{choose_best} + |E| \times \textit{relaxamento})$
  [assumindo o uso de uma lista de adjacências]

- Consideremos uma implementação *naive* com **um ciclo para descobrir a distância mínima**
  - um choose_best custaria $\mathcal{O}(|V|)$ [ciclo pelos nós]
  - um relaxamento custaria $\mathcal{O}(1)$ [atualizar distância]

  Ficaríamos com complexidade total $\mathcal{O}(|V| + |V|^2) + |E|)$. Como o número de arestas é no máximo $|V|^2$, a complexidade pode ser simplificada para $\mathcal{O}(|V^2|)$.

  Como melhorar?

Algoritmo de Dijkstra - Complexidade

- Dijkstra: $\mathcal{O}(|V| + |V| \times choose_best + |E| \times relaxamento)$
  [assumindo o uso de uma lista de adjacências]

- Consideremos uma implementação com uma **fila de prioridade** (ex: uma **min-heap**)
  - um choose_best custaria $\mathcal{O}(\log |V|)$
    [retirar elemento da fila de prioridade]
  - um relaxamento custaria $\mathcal{O}(\log |V|)$
    [atualizar prioridade fazendo elemento "subir" na heap]

  Ficaríamos no total com $\mathcal{O}(|V| + |V| \times \log |V| + |E| \times \log |V|)$.
  Assumindo que $|E| \geq |V|$, a parte dos relaxamentos vai dominar o tempo e a complexidade pode ser simplificada para $\mathcal{O}(|E| \log |V|)$.

Algoritmo de Dijkstra e Filas de Prioridade

- As linguagens de programação tipicamente trazem já disponível uma fila de prioridade que garante complexidade logarítimica para **inserção** de um novo valor e **remoção do mínimo**:
  - **C++:** `priority_queue`
  - **Java:** `PriorityQueue`
- Estas implementações não trazem tipicamente a parte de actualizar um valor (nem a hipótese de retirar um valor no meio da fila).
- Três possíveis hipóteses para lidar com actualização de valor:
  1. Usar heap "manualmente"
     (podemos chamar *up_heap* em qualquer nó no meio da fila()
     Complexidade do Dijkstra: $\mathcal{O}(|E| \log |V|)$ ou
  2. Usar uma *PriorityQueue* e actualizar ser feito via inserção de novo elemento na heap com a nova distância (ignorar depois nó "repetido")
     (cada nó será inserido no máximo tantas vezes quanto o seu grau)
     Complexidade do Dijkstra: $\mathcal{O}(|E| \log |E|)$ ou
  3. Usamos uma BST (ex: um *set*) e actualizar ser feito via remoção + inserção (ambas as operações em tempo logarítmico)
     Complexidade do Dijkstra: $\mathcal{O}(|E| \log |V|)$

Algoritmo de Dijkstra - Implementação

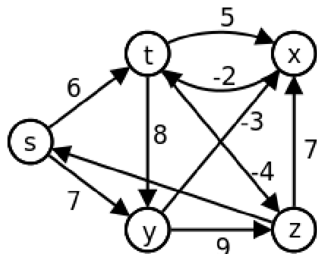
Exemplo de Implementação

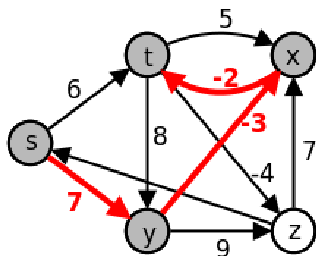
No vídeo temos aqui livecoding com um exemplo de implementação do Dijkstra

[implementação disponível também na aula prática]

Algoritmo de Dijkstra

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



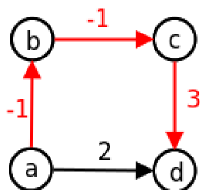


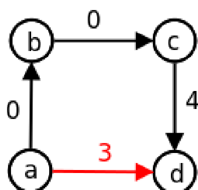
- Por exemplo no grafo indicado, o algoritmo de Dijkstra iria dizer que t está a distância 6, quando existe um caminho (indicado a vermelho) que está a distância 2!

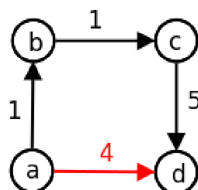
- O problema é que os caminhos podem ficar com menor custo ao acrescentar arestas...

Algoritmo de Dijkstra

- Notem 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...





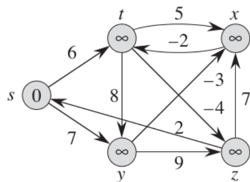


- Para o grafo original de cima, o melhor caminho entre a e d é $a \rightarrow b \rightarrow c \rightarrow d$ (com custo 1). Ao acrescentarmos uma constante a todas as arestas (na figura estão representadas as adições de 1 e de 2), esse caminho vai sofrer uma penalização de $3 \times c$, ao passo que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

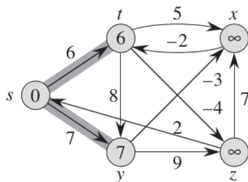
Algoritmo de Bellman-Ford

- 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!

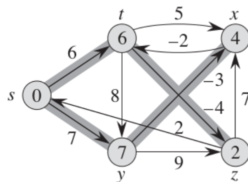
Vamos ver um exemplo passo a passo:



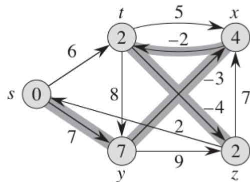
(a)



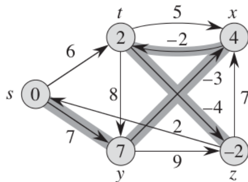
(b)



(c)



(d)



(e)

(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 $\mathcal{O}(|V| \times |E|)$ 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 slides 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 (naive): $\mathcal{O}(|V^3|)$
  - Dijkstra (com fila de prioridade): $\mathcal{O}(|V| \times |E| \log |V|)$
  - Bellman-Ford: $\mathcal{O}(|V^2| \times |E|)$ *(mas funciona com pesos negativos)*
- Existe um algoritmo $\mathcal{O}(|V|^3)$ que é **muito fácil de implementar** e é mais rápido do que um Dijkstra naive 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**
          $dist[i][j] \leftarrow dist[i][k] + dist[k][j]$

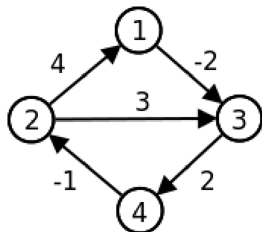
- A complexidade é trivialmente $\mathcal{O}(|V|^3)$ - ver os 3 ciclos!

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:

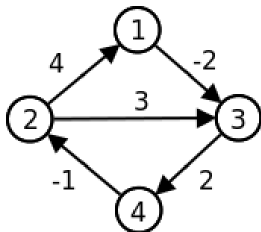


Inicialmente temos a seguinte matriz de distâncias:

	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:

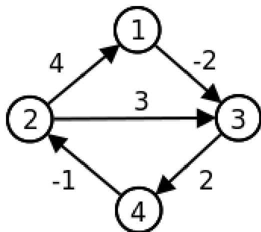


Ao passarmos por $k = 1$ actualizamos os caminhos que passam por 1: $2 \rightarrow \mathbf{1} \rightarrow 3$

	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:

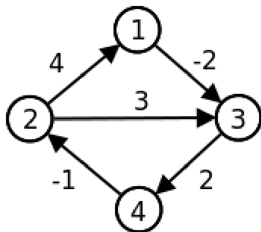
$4 \rightarrow \mathbf{2} \rightarrow 1$

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

	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:

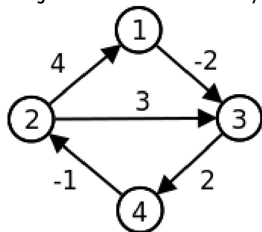
$1 \rightarrow \mathbf{3} \rightarrow 4$

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

	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

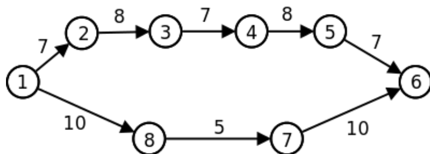
Floyd-Warshall(G):
  Seja *connected*[][] uma matriz booleana $|V| \times |V|$ inicializada a falsos
  Para cada vértice v de G **fazer**:
    $connected[v][v] \leftarrow verdadeiro$
  Para todas as arestas (u, v) de G **fazer**:
    $connected[u][v] \leftarrow verdadeiro$

  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**
        $v.dist \leftarrow min(u.dist, peso(u, v))$ // relaxamento de aresta

Visualizações

- Para finalizar este capítulo fica a recordação que estão disponíveis ligações para **visualizações de algoritmos** que podem ser úteis para ver "em acção" os algoritmos de que vamos falando nesta UC
- **VisuAlgo:** `https://visualgo.net/`
  - Single-Source Shortest Paths (inclui Dijkstra, Bellman-Ford e BFS)
- **David Galles (U San Francisco):** Data Structure Visualizations
  - Dijkstra
  - Floyd-Warshall