Caminhos mínimos de uma única origem
5189-31

Rodrigo Calvo
rcalvo@uem.br

Departamento de Informática – DIN
Universidade Estadual de Maringá – UEM

1° semestre de 2016

Introdução

- Como encontrar o caminho mínimo entre duas cidades?
- O problema para encontrar este caminho é conhecido como **problema de caminho mínimo**
- Entrada
  - Um grafo direcionado $G = (V, A)$
  - Uma função peso $w : A \rightarrow \mathbf{R}$

Introdução

- O **peso do caminho** $p = \langle v_0, v_1, \ldots, v_k \rangle$ é
  - a soma dos pesos das arestas no caminho

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- O **peso do caminho mínimo** de u até v é definido como

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{se existe um caminho de } u \text{ até } v \\ \infty, & \text{caso contrário} \end{cases}$$

- Um **caminho mínimo** do vértice u até o vértice v é qualquer caminho p com peso $w(p) = \delta(u, v)$

Introdução

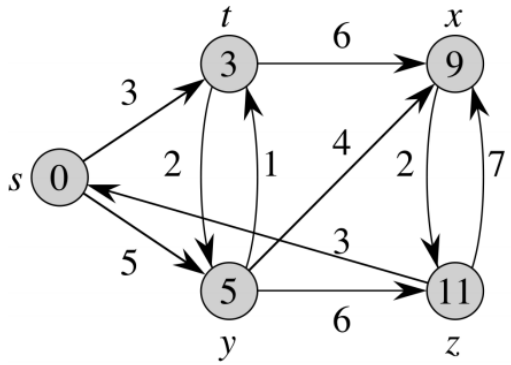
- Os pesos das arestas podem representar outras métricas além da distância, como o tempo, custo, ou outra quantidade que acumule linearmente ao longo de um caminho e que se deseja minimizar

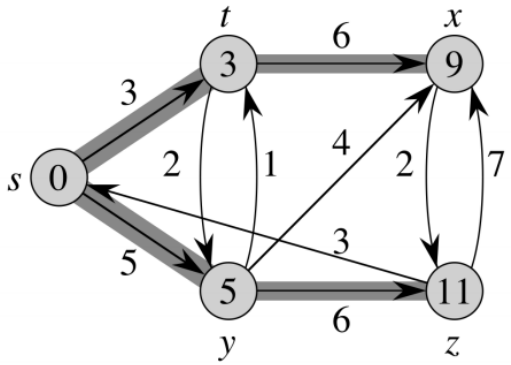
- O algoritmo de busca em largura é um algoritmo de caminhos mínimos que funciona para grafos não valorados (ponderados), isto é, as arestas tem peso unitário

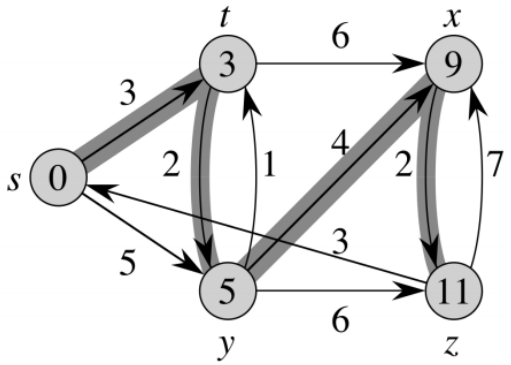
Tipos de problemas de caminhos mínimos

- **Origem única**: Encontrar um caminho mínimo a partir de uma dada origem $s \in V$ até todo vértice $v \in V$
- **Destino único**: Encontrar um caminho mínimo até um determinado vértice de destino t a partir de cada vértice v
- **Par único**: Encontrar o caminho mínimo de u até v
- **Todos os pares**: Encontrar um caminho mínimo de u até v para todo par de vértices u e v

Exemplo de caminhos mínimos de única origem







- Observe que:
  - O caminho mínimo pode não ser único
  - Os caminhos mínimos de uma origem para todos os outros vértices formam uma árvore

Caminhos mínimos

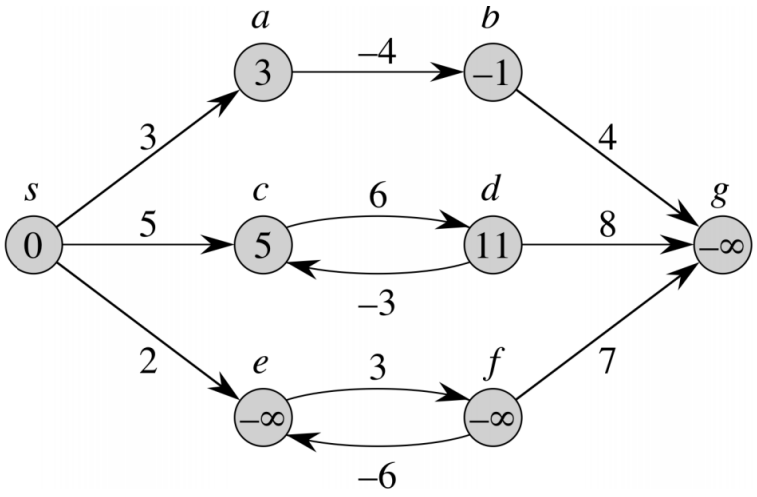
- Características de caminhos mínimos:
  - Subestrutura ótima
  - Ciclos e arestas de pesos negativos
  - Representação
  - Relaxamento
  - Propriedades

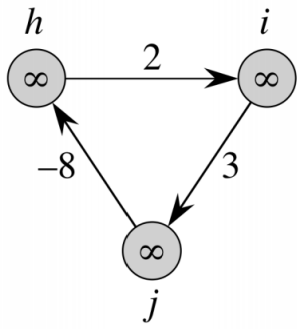
Subestrutura ótima

- Em geral os algoritmos de caminhos mínimos se baseiam na seguinte propriedade
  - **Lema 24.1** Qualquer subcaminho de um caminho mínimo é um caminho mínimo
  - Como provar este lema?

Arestas de pesos negativos

- Não apresentam problemas se nenhum ciclo com peso negativo é alcançável a partir da origem
- Nenhum caminho da origem até um vértice em um ciclo negativo pode ser mínimo
- Se existe um ciclo de peso negativo em algum caminho de s até v, definimos $\delta(s, v) = -\infty$





Ciclos

- Caminhos mínimos podem conter ciclos?

Ciclos

- Caminhos mínimos podem conter ciclos? Não
- **Ciclos de peso negativo**: $\delta(s, v) = -\infty$
- **Ciclos de peso positivo**: Pode-se obter um caminho mínimo eliminando o ciclo
- **Ciclos de peso nulo**: Não existe razão para usar tal ciclo

- Todo caminho mínimo não possui ciclos
- Qualquer caminho acíclico em um grafo $G = (V, A)$ contém no máximo $|V|$ vértices distintos e no máximo $|V| - 1$ arestas

Representação

- Os caminhos mínimos são representados de forma semelhante às árvores primeiro na extensão produzidas pelo algoritmo **`bfs`**
- Para cada vértice $v \in V$, a saída dos algoritmos consiste em
  - $v.d = \delta(s, v)$
    - Inicialmente $v.d = \infty$
    - Diminui conforme o algoritmo progride, mas sempre mantém a propriedade $v.d \geq \delta(s, v)$
    - $v.d$ pode ser chamado de **estimativa do caminho mínimo**
  - $v.\pi$ = predecessor de v no caminho mínimo a partir de s
    - Se não existe predecessor, então $v.\pi = nil$
    - π induz uma árvore, a árvore de caminhos mínimos

Representação

- A partir dos valores de π gerados em algoritmos de caminhos mínimos, têm-se como resultado um grafo G_π que é uma **árvore de caminhos mínimos**
- G_π é uma árvore que contém um caminho mínimo desde a raiz a todo vértice que pode ser alcançado a partir da raiz.
- **Árvore de caminhos mínimos** é semelhante à arvore primeiro na extensão
- Definida como um grafo $G' = (V', A')$, onde $V' \subseteq V$ e $A' \subseteq A$:
  - V' é o conjunto de vértices alcançáveis a partir da raiz
  - G' forma uma árvore enraizada
  - Para todo $v \in V'$, o único caminho simples da raiz até v em G' é um caminho mínimo da raiz até v em G.

Relaxamento

- Sendo os vértices inicializados com a função

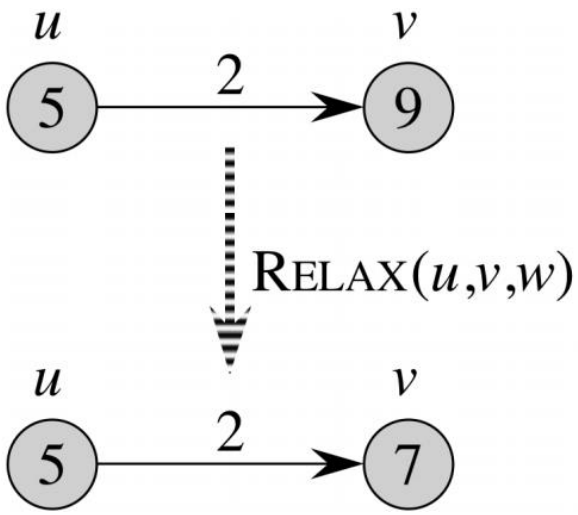
```
initialize-single-source(G, s)
1 for cada vértice v em G.V
2    v.d = ∞
3    v.π = nil
4 s.d = 0
```

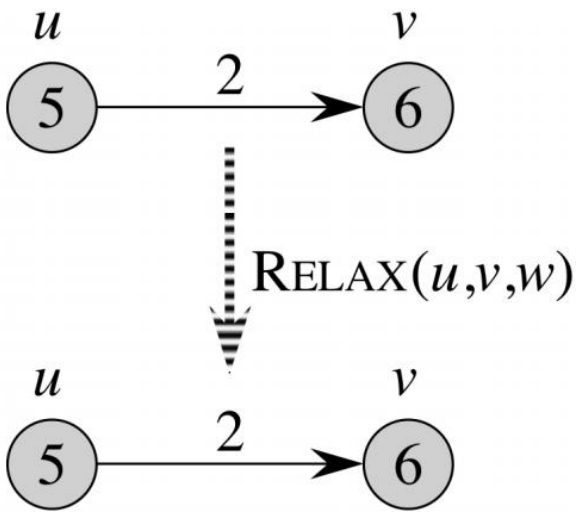
- Podemos melhorar a estimativa do caminho mínimo para *v*, passando por *u* e seguindo (*u*, *v*)?

Relaxamento

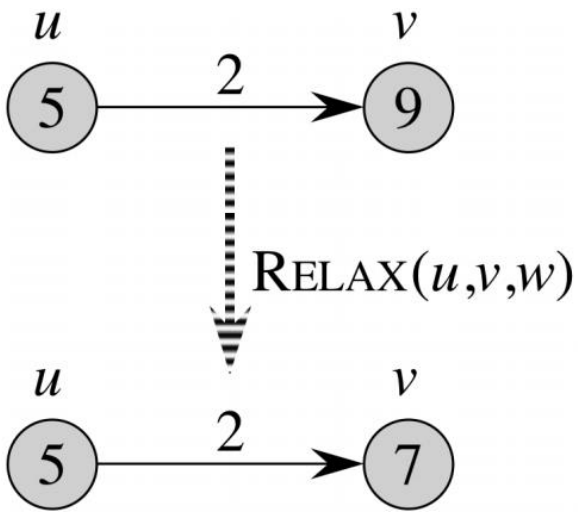
- Relaxamento de uma aresta (u, v) consiste em testar se o caminho mínimo até um vértice v, passando por u, pode ser melhorado. Se for possível, os atributos $v.\pi$ e $v.d$ são atualizados.
- Relaxamento é o único meio de mudar as estimativas de caminhos mínimos e predecessores.
- Os algoritmos para encontrar caminhos mínimos diferem na quantidade de vezes que cada aresta é relaxada e a ordem que tal relaxamento ocorre.

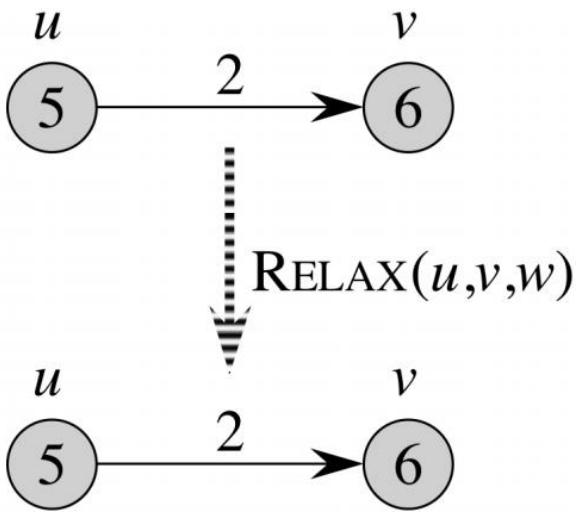
Relaxamento





Relaxamento






```
relax(u, v, w)
1 if v.d > u.d + w(u, v)
2    v.d = u.d + w(u, v)
3    v.π = u
```

Propriedades
Caminhos mínimos e relaxamento

- Desigualdade triangular (Lema 24.10)
  - Para toda aresta $(u, v) \in A$, têm-se que $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Propriedades
Caminhos mínimos e relaxamento

- Desigualdade triangular (Lema 24.10)
  - Para toda aresta $(u, v) \in A$, têm-se que $\delta(s, v) \leq \delta(s, u) + w(u, v)$

- Para as próximas propriedades, é suposto que
  - O grafo é inicializado com uma chamada a **`initialize-single-source`**
  - O único modo de modificar $v.d$ e $v.\pi$ (para qualquer vértice) e pela chamada de **`relax`**

Propriedades
Caminhos mínimos e relaxamento

- Propriedade do limite superior (Lema 24.11)
  - Sempre têm-se que $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda

Propriedades
Caminhos mínimos e relaxamento

- Propriedade do limite superior (Lema 24.11)
  - Sempre têm-se que $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda
- Propriedade de inexistência de caminho (Lema 24.12)
  - Se $\delta(s, v) = \infty$, então sempre $v.d = \infty$

Propriedades
Caminhos mínimos e relaxamento

- Propriedade do limite superior (Lema 24.11)
  - Sempre têm-se que $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda
- Propriedade de inexistência de caminho (Lema 24.12)
  - Se $\delta(s, v) = \infty$, então sempre $v.d = \infty$
- Propriedade de convergência (Lema 24.14)
  - Se $s \rightsquigarrow u \rightarrow v$ é um caminho mínimo e se $u.d = \delta(s, u)$ em qualquer instante antes da chamada relax(u, v, w), então $v.d = \delta(s, v)$ em todos os instantes posteriores.

Propriedades
Caminhos mínimos e relaxamento

- Propriedade do limite superior (Lema 24.11)
  - Sempre têm-se que $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda
- Propriedade de inexistência de caminho (Lema 24.12)
  - Se $\delta(s, v) = \infty$, então sempre $v.d = \infty$
- Propriedade de convergência (Lema 24.14)
  - Se $s \rightsquigarrow u \rightarrow v$ é um caminho mínimo e se $u.d = \delta(s, u)$ em qualquer instante antes da chamada relax(u, v, w), então $v.d = \delta(s, v)$ em todos os instantes posteriores.
- Propriedade de relaxamento de caminho (Lema 24.15)
  - Seja $p = \langle v_0, v_1, \ldots, v_k \rangle$ o caminho mínimo v_0 até v_k, se a função **`relax`** for chamada na ordem $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, mesmo que intercalada com outros relaxamentos, então, então $v_k.d = \delta(s, v_k)$

Algoritmos de caminhos mínimos

- Inicialização dos atributos $v.d$ e $v.\pi$
- Relaxamento das arestas

- Os algoritmos diferem na ordem e na quantidade de vezes que cada aresta é relaxada.

Algoritmo de Bellman-Ford

- Resolve o problema para o caso geral, as arestas podem ter pesos negativos
- Detecta ciclos negativos acessíveis a partir da origem:
  - Retorna ***false***, se encontrar ciclos negativos
  - Retorna ***true***, caso contrário
- Calcula $v.d$ e $v.\pi$ para todo $v \in V$
- Ideia
  - Relaxar todas as arestas, $|V| - 1$ vezes

Algoritmo de Bellman-Ford

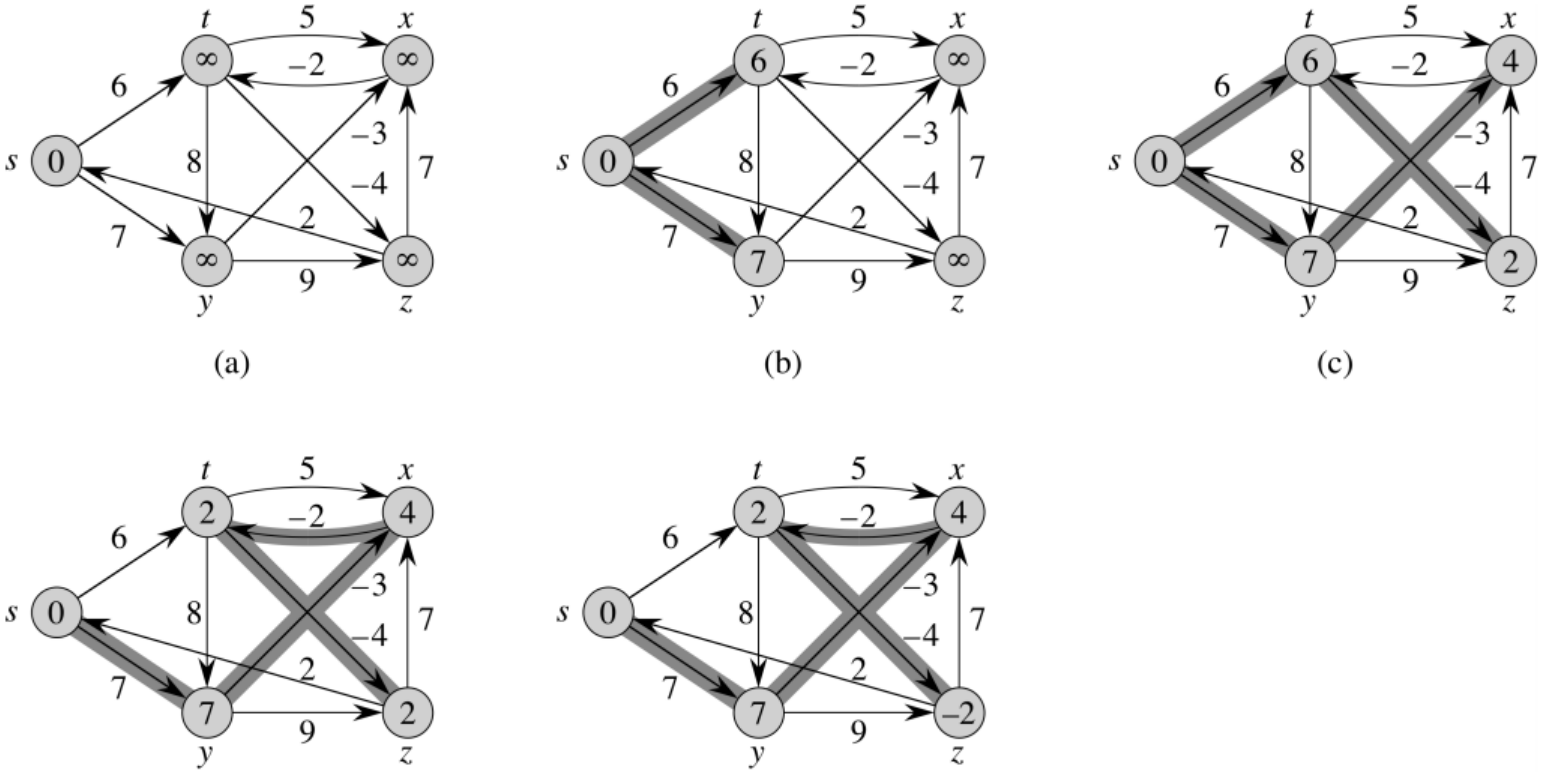
```
bellman-ford(G, w, s)
1 initialize-single-source(G, s)
2 for i = 1 to |G.V| - 1
3   for cada aresta (u, v) em G.A
4     relax(u, v, w)
5 for cada aresta (u, v) em G.A
6   if v.d > u.d + w(u, v)
7     return false
8 return true
```

Algoritmo de Bellman-Ford

```
bellman-ford(G, w, s)
1 initialize-single-source(G, s)
2 for i = 1 to |G.V| - 1
3   for cada aresta (u, v) em G.A
4     relax(u, v, w)
5 for cada aresta (u, v) em G.A
6   if v.d > u.d + w(u, v)
7     return false
8 return true
```

- Análise do tempo de execução:
  - A inicialização na linha 1 demora $\Theta(V)$
  - Cada uma das $|V| - 1$ passagens das linha 2 a 4 demora o tempo $\Theta(A)$, totalizando $O(V \cdot A)$
  - O laço das linha 5 a 7 demora O(A)
  - Tempo de execução do algoritmo $\Theta(V \cdot A)$

Algoritmo de Bellman-Ford



- Ordem em que as arestas foram relaxadas:
  (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

Algoritmo Bellman-Ford

- Por que o algoritmo funciona ?
  - Propriedade de relaxamento de caminho
  - Seja v acessível a partir de s, e seja $p = \langle v_0, v_1, \ldots, v_k \rangle$ um caminho mínimo acíclico entre $s = v_0$ e $v = v_k$. p tem no máximo $|V| - 1$ arestas, e portanto $k \leq |V| - 1$
  - Cada iteração do laço da linha 2 relaxa todas as arestas
  - A primeira iteração relaxa (v_0, v_1)
  - A segunda iteração relaxa (v_1, v_2)
  - . . .
  - A k-ésima iteração relaxa $(v_k{-}1, v_k)$
- Pela propriedade de relaxamento de caminho $v.d = v_k .d = \delta(s, v_k) = \delta(s, v)$

Algoritmo para gad

- Grafo acíclico direcionado (gad) ponderado
- Caminhos mínimos são sempre bem definidos em um gad, pois não existem ciclos (de peso negativo)
- Ideia
  - Relaxar as arestas em uma ordem topológica de seus vértices

Algoritmo para gad

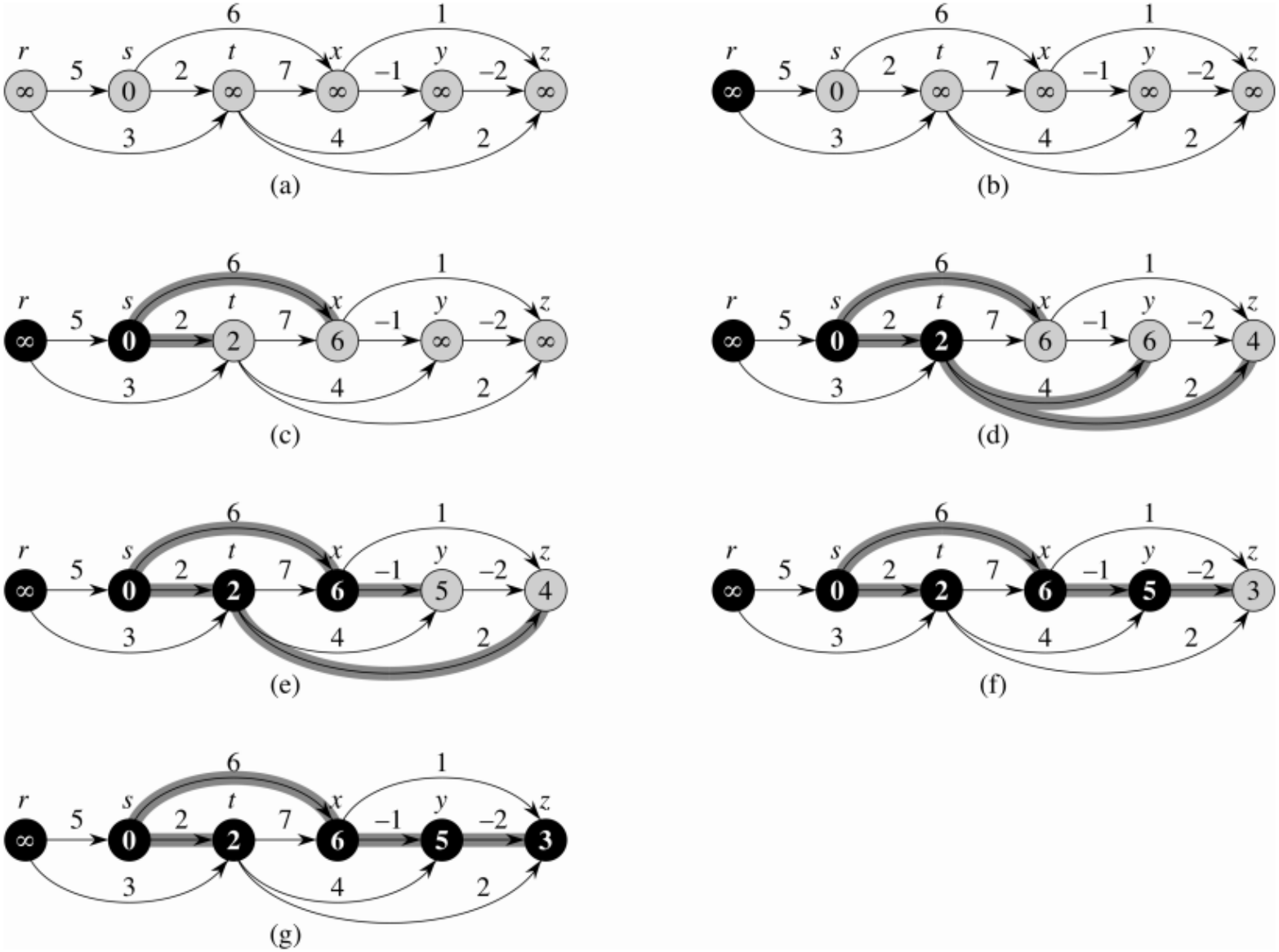
```
dag-shortest-paths(G, w, s)
1 ordenar topologicamente os vértices de G
2 initialize-single-source(G, s)
3 for cada vértice u tomado na ordem topológica
4   for cada vértice v em u.adj
5     relax(u, v, w)
```

Algoritmo para gad

```
dag-shortest-paths(G, w, s)
1 ordenar topologicamente os vértices de G
2 initialize-single-source(G, s)
3 for cada vértice u tomado na ordem topológica
4   for cada vértice v em u.adj
5     relax(u, v, w)
```

- Análise do tempo de execução:
  - A ordenação topológica da linha 1 demora $\Theta(V + A)$
  - **`initialize-single-source`** na linha 2 demora $\Theta(V)$
  - Nos laços das linhas 3 e 4, a lista de adjacências de cada vértices é visitada apenas uma vez, totalizando $V + A$ (análise agregada), como o relaxamento de cada aresta custa $O(1)$, o tempo total é $\Theta(A)$
  - Portanto, o tempo de execução do algoritmo é $\Theta(V + A)$

Caminhos mínimos única origem: gad



Caminhos mínimos única origem: gad

- Por que o algoritmo funciona ?
  - Como os vértices são processados em ordem topológica, as arestas de qualquer caminho são relaxadas na ordem que aparecem no caminho
  - Pela propriedade de relaxamento de caminho, o algoritmo funciona corretamente

Aplicação

- Caminhos críticos na análise de diagramas PERT (*program*
- *evaluation and review technique*)
- As arestas representam serviços a serem executados
- Os pesos de arestas representam os tempos necessários para execução de determinados serviços

  (u, v) , v, (v, x): serviço (u, v) deve ser executado antes do serviço (v, x)
- Um caminho através desse gad: sequencia de serviços
- Caminho crítico: é um caminho mais longo pelo gad
- Tempo mais longo para execução de uma sequencia ordenada
- O peso de um caminho crítico é um limite inferior sobre o tempo total para execução de todos os serviços

Aplicação

- Um caminho crítico pode ser encontrado de duas maneiras:
  - Tornando negativos os pesos das arestas e executando **`dag-shortest-paths`**; ou
  - Executando **`dag-shortest-paths`**, substituindo "∞" por "−∞" na linha 2 de **`initialize-single-source`** e ">" por "<" no procedimento **`relax`**

Algoritmo de Dijkstra

- Caminho mínimo de única origem em um grafo direcionado ponderado
- Todos os pesos de arestas são não negativos, ou seja $w(u, v) \geq 0$ para cada aresta $(u, v) \in A$

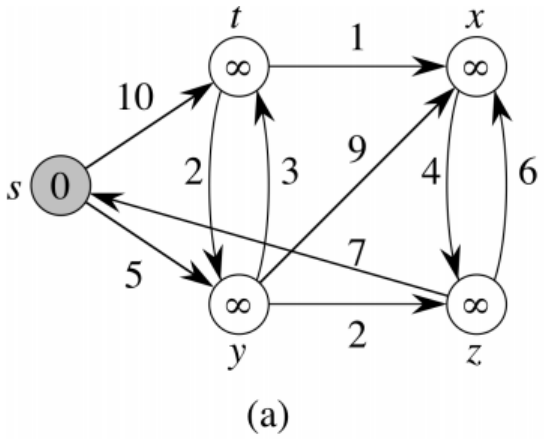
Algoritmo de Dijkstra

- Ideia
  - Essencialmente uma versão ponderada da busca em largura
  - Ao invés de uma fila FIFO, usa uma fila de prioridades
  - As chaves são os valores $v.d$
  - Mantém dois conjuntos de vértices
  - S: vértices cujo caminho mínimo desde a origem já foram determinados
  - $Q = V - S$: fila de prioridades
  - O algoritmo seleciona repetidamente o vértice $u \in Q$ com a mínima estimativa de peso do caminho mínimo, adiciona u a S e relaxa todas as arestas que saem de u

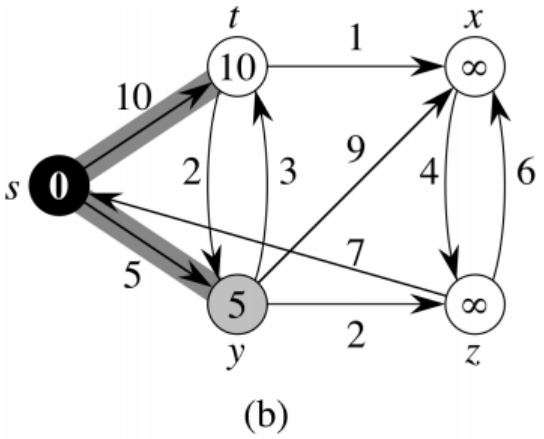
Algoritmo de Dijkstra

```
dijkstra(G, w, s)
1 initialize-single-source(G, s)
2 S = {}
3 Q = G.V
4 while Q != {}
5   u = extract-min(Q)
6   S = S U {u}
7   for cada vértice v em u.adj
8     relax(u, v, w)
```

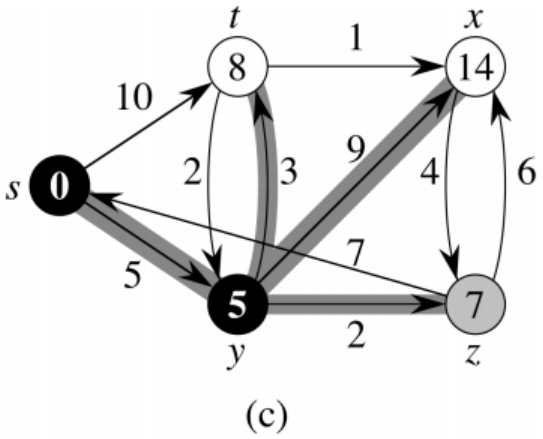
Algoritmo de Dijkstra



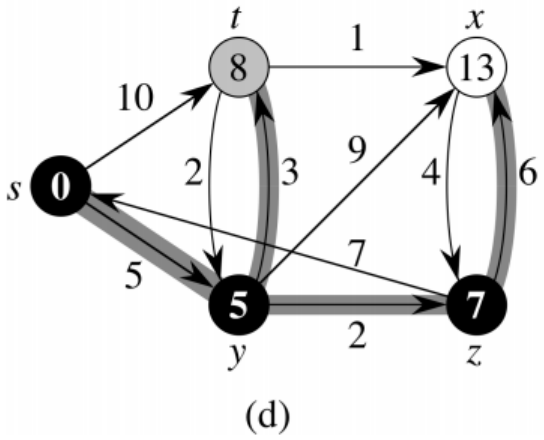
(a)



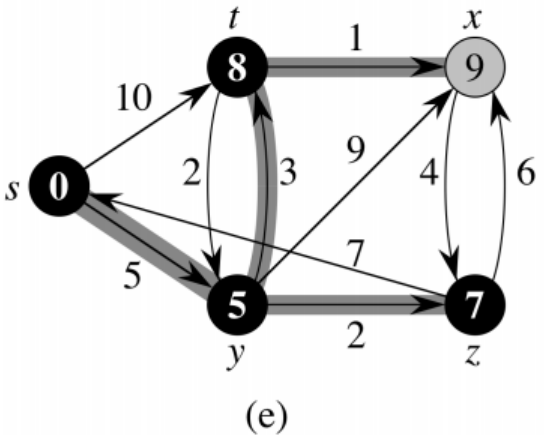
(b)



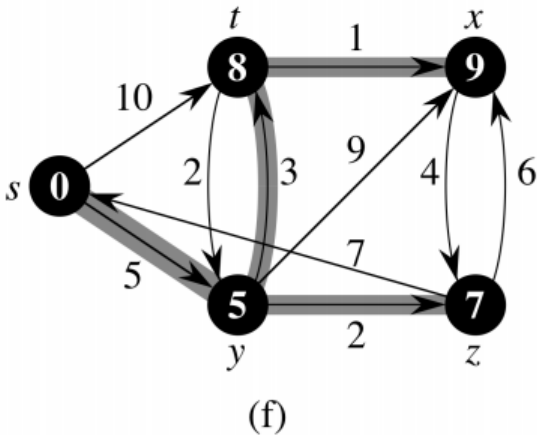
(c)



(d)



(e)



(f)

Algoritmo de Dijkstra

- Análise do tempo de execução
  - Linha 1, $\Theta(V)$
  - Linhas 4 a 8, $O(V + A)$ (sem contar as operações com fila)
  - Operações de fila
  - **`insert`** implícita na linha 3 (executado uma vez para cada vértice)
  - **`extract-min`** na linha 5 (executado uma vez para cada vértice)
  - **`decrease-key`** implícita em relax (executado no máximo de $|A|$ vezes, uma vez para cada aresta relaxada)
  - Depende da implementação da fila de prioridade

Algoritmo de Dijkstra

- Análise do tempo de execução
  - Arranjos simples
    - Como os vértices são enumerados de 1 a $|V|$, o valor $v.d$ é armazenado na v-ésima entrada de um arranjo
    - Cada operação **`insert`** e **`decrease-key`** demora O(*1*)
    - Cada operação **`extract-min`** demora O(V) (pesquisa linear)
    - Tempo total de $O(V^2 + A) = O(V^2)$

Algoritmo de Dijkstra

- Análise do tempo de execução
  - Heap
    - Se o grafo é esparso, em particular, $A = o(V^2 / \lg V)$ é prático utilizar um heap binário
    - O tempo para construir um heap é $O(V)$
    - Cada operação de **`extract-min`** e **`decrease-key`** demora $O(\lg V)$
    - Tempo total de $O((V + A) \lg V + V)$, que é $O(A \lg V)$ se todos os vértices são acessíveis a partir da origem

Algoritmo de Dijkstra

- Análise do tempo de execução
  - Heap de Fibonacci
    - Cada operação **`extract-min`** demora O(lg *V*)
    - Cada operação **`decrease-key`** demora o tempo amortizado de O(*1*)
    - Tempo total de O(*V* lg *V* + *A*)

Algoritmo de Dijkstra

- Por que o algoritmo funciona ?
  - Invariante de laço: no início de cada iteração do laço **`while`**, $v.d = \delta(s, v)$ para todos $v \in S$
  - Inicialização: $S = \emptyset$, então é verdadeiro
  - Término: No final, $Q = \emptyset \Rightarrow S = V \Rightarrow v.d = \delta(s, v)$, para todo $v \in V$
  - Manutenção: é necessário mostrar que $u.d = \delta(s, u)$ quando u é adicionado a S em cada iteração

Bibliografia

- Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulo 24.

- Nivio Ziviani. Projeto de Algoritmos com Implementações em Pascal e C. 3a Edição Revista e Ampliada, Cengage Learning, 2010. Capítulo 7. Seção 7.8