Caminhos mínimos de única origem

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Problema

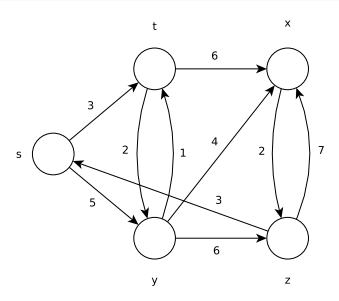
Como determinar a “rota” mais curta entre duas cidades do Brasil?

Que informações nós temos? A cidade de origem, a cidade de destino e um mapa das estradas do Brasil com a distância entre cada par de interseções adjacentes. O mapa e as distâncias podem ser representadas por um grafo:

- Cada interseção é representada por um vértice;
- A distância entre cada par de interseções adjacentes é representada por uma aresta com peso.

Se a nossa entrada é um grafo e dois vértices, qual é a saída?

- Um caminho que inicia no vértice de origem e termina no vértice de destino e tem o peso (soma dos pesos das arestas) mínimo.



Se a cidade de origem é representada por s e a de destino por x

- O caminho $\langle s, y, t \rangle$ representa um "rota" entre as cidades de origem e destino? Não.
- O caminho $\langle s, y, t, x \rangle$ representa um "rota" entre as cidades de origem e destino? Sim. Qual o peso desse caminho? 12. Esta é a "rota" mais curta? Não. Os caminhos $\langle s, y, x \rangle$ e $\langle s, t, x \rangle$ também representam rotas entre as cidades de origem e destino e tem peso 9.

Neste módulo vamos estudar alguns **problemas de caminho mínimo**. Antes de continuarmos precisamos definir com precisão alguns termos.

Seja $G = (V, E)$ um grafo orientado e $w : E \to \mathbb{R}$ uma função peso:

- O **peso do caminho** $p = \langle v_0, v_1, \ldots, v_k \rangle$ é a soma dos pesos das arestas em p

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

- O **peso do caminho mínimo** $\delta(u, v)$ de u até v é

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

- Um **caminho mínimo** de u até v é qualquer caminho p tal que $w(p) = \delta(u, v)$.

Peso ou distância?

Porque usamos o termo “peso” e não “distância?

Porque 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 desejamos minimizar.

Único par: Encontrar o caminho mínimo de u até v.

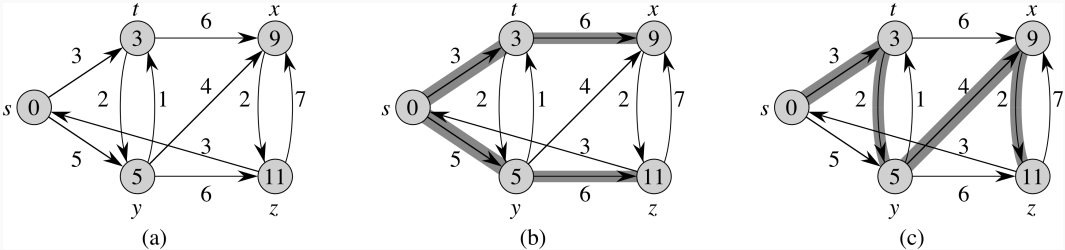
Única origem: Encontrar um caminho mínimo a partir de uma dada origem $s \in V$ até todo vértice $v \in V$. O algoritmo de busca em largura é um algoritmo de caminhos mínimos de única origem que funciona para grafos não valorados, isto é, as arestas tem peso unitário.

Único destino: Encontrar um caminho mínimo até um determinado vértice de destino t a partir de cada vértice v.

Todos os pares: Encontrar um caminho mínimo deste u até v para todo par de vértices u e v.

Exemplo

Vamos focar no **problema do caminho mínimo de única origem**. Aqui está um exemplo de caminhos mínimos de única origem



Antes de pensarmos em como resolver este problema, o que podemos observar sobre caminhos mínimos?

- Um caminho mínimo é formado por outros caminhos mínimos (subestrutura ótima);
- Os caminhos mínimos de única origem formam uma árvore.

Lema 24.1

Qualquer subcaminho de um caminho mínimo é um caminho mínimo.

Prova

Ideia da prova: se um subcaminho não for mínimo podemos trocá-lo por um subcaminho mínimo é obter um caminho de menor peso, o que é uma contradição pois o caminho é mínimo!

Da mesma forma que no BFS, DFS e PRIM:

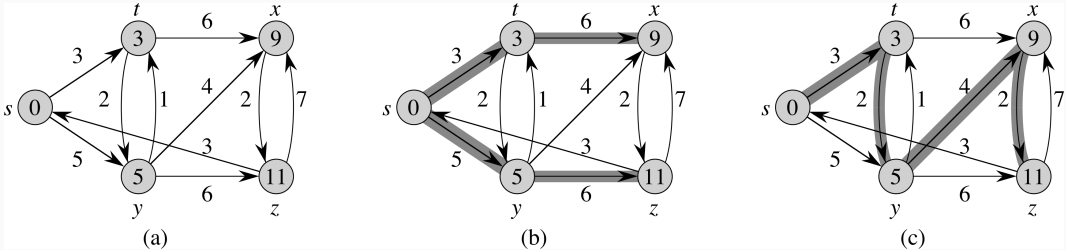
- $v.\pi$ = predecessor de v no caminho mínimo a partir de s
- Se não existe predecessor, então $v.\pi = \text{NIL}$

Qual é o tipo do problema? Otimização.

Que estratégias podemos tentar utilizar?

- Algoritmo guloso
- Programação dinâmica
- Melhoramento iterativo
- Etc

Vamos tentar utilizar estas técnicas para derivar hipóteses de algoritmos.



Como produzir árvores de caminhos mínimos a partir da entrada?

Discutimos em sala como derivamos as hipóteses a seguir.

Hipóteses

- Algoritmo guloso: Começar com o vértice de origem na árvore e ir "ligando" novos vértices à árvore usando os caminhos mínimos (não pode haver arestas de peso negativo).
- Programação dinâmica: Expressar os pesos dos caminhos mínimos que usam no máximo k arestas, em termos dos pesos dos caminhos mínimos que usam no máximo $k - 1$ arestas.
- Melhoramento iterativo: Iniciar com uma árvore de caminhos qualquer e tentar melhorar os caminhos trocando uma aresta da árvore por uma que não está na árvore.

A ideia deste algoritmo guloso para caminhos mínimos de única origem foi inicialmente proposta por Dijkstra.

A ideia do algoritmo de Dijkstra é semelhante a de outro algoritmo guloso: o algoritmo de Prim. Vamos comparar as duas ideias.

Dijkstra (Caminhos mínimos de única origem): Começar com o vértice de origem na árvore e ir "ligando" novos vértices à árvore usando os caminhos de menor peso.

Prim (Árvores geradoras mínimas): Começar com um vértice na árvore e ir "ligando" novos vértices à árvore usando as arestas de menor peso.

Quais são as semelhanças?

- Começam com um vértice na árvore;
- Escolhem o próximo vértice para ligar na árvore usando um critério guloso.

Qual é a diferença? O critério guloso:

- Prim: escolhemos ligar o novo vértice usando **uma aresta** de menor peso.
- Dijkstra: escolhemos ligar um novo vértice usando **um caminho** de menor peso.

Como podemos implementar de forma eficiente o algoritmo de Dijkstra?

Considerando a semelhança entre o algoritmos de Dijkstra e o algoritmo de Prim, vamos lembrar: Qual era o “desafio” para implementar o algoritmo Prim de forma eficiente?

- Escolher de forma eficiente o próximo vértice para ser ligado a árvore.

Como resolvemos esse problema? Usando uma fila de prioridades.

Vamos revisar como isso funcionava no algoritmo de Prim.

Os elementos em uma fila de prioridade são removidos por ordem de prioridade (a *chave* no algoritmo de Prim).

Para cada vértice v fora da árvore, qual é o significado do valor $v.chave$ no algoritmo de Prim? O menor peso entre todas as arestas que podem ser usadas para ligar v a um vértice qualquer da árvore.

Todos os vértices são inicialmente adicionados na fila. O vértice raiz r começa com $r.chave = 0$ e os outros vértices com $chave = \infty$.

Cada vez que um vértice u é removido da fila para ser ligado a árvore, o que é necessário fazer? Verificar é possível ligar algum vértice v adjacente de u de forma "mais eficiente" a árvore, isto é, se podemos mudar o valor de $v.chave$ para $w(u, v)$:

```
if v.chave > w(u, v)
    v.chave = w(u, v)
    v.π = u
```

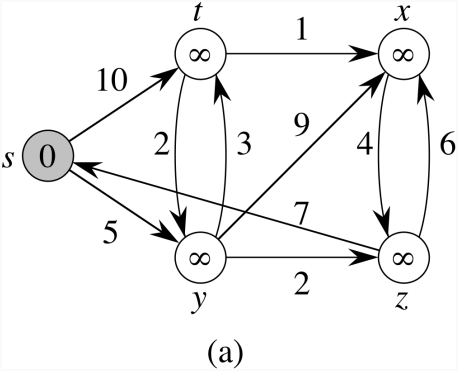
Vamos usar a mesma ideia no algoritmo de Dijkstra.

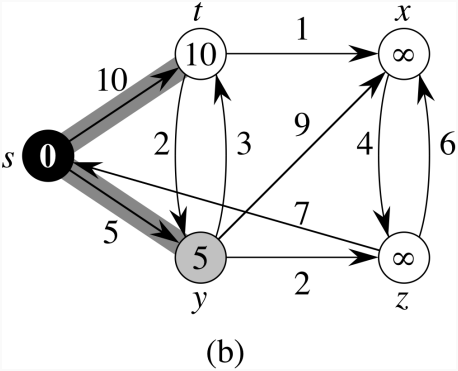
Para cada vértice v do grafo mantemos o atributo $v.d$, a **estimativa de caminho mínimo**, que é usada como prioridade.

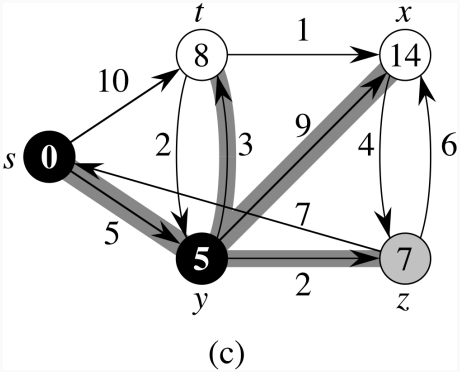
Todos os vértices são inicialmente adicionados na fila. O vértice de origem s começa com $s.d = 0$ e os vértices começam com $d = \infty$.

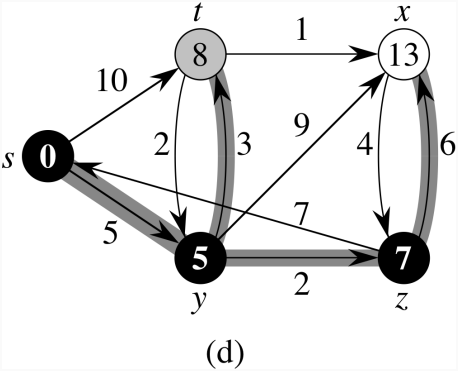
Assim como as chaves no algoritmo de Prim são alteradas conforme o algoritmo progride, os valores das estimativas de caminhos mínimo também são alteradas.

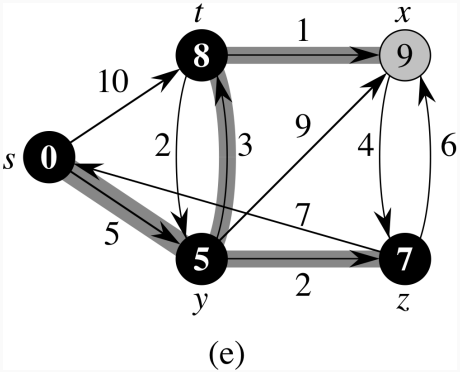
Nos exemplos a seguir as arestas destacadas que levam a vértices brancos mostram a estimativa de caminho mínimo atual.

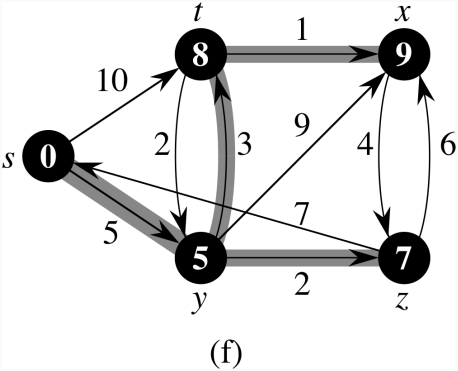

t
x
1
∞
∞
10
9
s
0
2
3
4
6
7
5
∞
∞
2
y
z
(a)


t
x
10
∞
1
10
9
s
0
2
3
4
6
7
5
5
∞
2
y
z
(b)

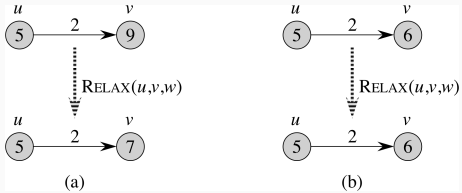

t
x
8
1
14
10
9
s
0
2
3
4
6
7
5
5
7
2
y
z
(c)


t
x
8
13
1
10
9
s
0
2
3
4
6
7
5
5
7
2
y
z
(d)


t
x
1
8
9
10
9
s
0
2
3
4
6
7
5
5
7
2
y
z
(e)



(f)



Como a estimativa de caminho mínimo deve ser atualizada algoritmo de Dijkstra?

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$

Este esquema de atualização também é utilizada em outros algoritmos, por isso definimos um procedimento.

Para manter a propriedade de que o atributo d representa uma estimativa de caminho mínimo, ele deve ser inicializado de forma apropriada e só pode ser alterado utilizando o procedimento RELAX.

Usamos a seguinte função de inicialização em diversos algoritmos:

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** $v \in G.V$
2 $\quad v.d = \infty$
3 $\quad v.\pi = \text{NIL}$
4 $s.d = 0$

Agora que sabemos que o algoritmo de Dijkstra pode ser implementado da mesma forma que o algoritmo de Prim, mudando apenas a forma que as prioridades são computadas, podemos escrever o pseudo código do algoritmo.

Para nos ajudar na análise de corretude do algoritmo, nós vamos utilizar um conjunto S, que conterá os vértices cujo caminho mínimo desde a origem já foram determinados.

```
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}
7        for v ∈ G.adj[u]
8              RELAX(u, v, w)
```

Análise do tempo de execução

É a mesma que a do algoritmo de Prim!

- O tempo de execução abstrato é escrito em termos dos tempos das operações de fila: $O(V) \times O(\text{INSERT}) + O(V) \times O(\text{EXTRACT-MIN}) + O(E) \times O(\text{DECREASE-KEY})$.
- Para determinar o tempo de execução concreto, precisamos considerar como a fila de prioridade é implementada.

Operação	Arranjo	Heap	Heap de Fibonacci
CREATE	$O(V)$	$O(V)$	$O(V)$
EXTRACT-MIN	$O(V)$	$O(\lg V)$	$O(\lg V)$ (amortizado)
DECREASE-KEY	$O(1)$	$O(\lg V)$	$O(1)$ (amortizado)

Tempo do procedimento DIJKSTRA

Tempo	Arranjo	Heap	Heap de Fibonacci
Grafo qualquer	$O(V^2)$	$O(E \lg V)$	$O(E + V \lg V)$
Grafo denso	$O(V^2)$	$O(V^2 \lg V)$	$O(V^2)$

```
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}
7      for v ∈ G.adj[u]
8          RELAX(u, v, w)
```

Análise de corretude

O algoritmo mantém a seguinte invariante:

- No início de cada iteração do laço **while** , $v.d = \delta(s, v)$ para todos $v \in S$

A invariante é verdadeira antes da primeira iteração?

- Sim! $S = \emptyset$, então é verdadeira por nulidade.

Vamos deixar a manutenção de lado por um instante e pensar no término. Quando o laço termina, quais vértices estão em S?

- Todos, então, pela invariante, $v.d = \delta(s, v)$, para todo $v \in V$ e o algoritmo produz a resposta correta!

Agora precisamos mostrar como a invariante é mantida quando um vértice é adicionado a S.

```
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}
7        for v ∈ G.adj[u]
8              RELAX(u, v, w)
```

Análise de corretude

Invariante: no início de cada iteração do laço **while** , $v.d = \delta(s, v)$ para todos $v \in S$.

Manutenção: Temos que mostrar que quando o vértice u é extraído da fila (linha 5) e adicionado ao conjunto S (linha 6) $u.d = \delta(s, u)$. Vamos supor que $u.d > \delta(s, u)$ e derivar uma contradição.

- u pode ser o s? Não, pois $s.d = 0 = \delta(s, s)$;
- Existe algum caminho entre s e u? Sim, pois se não $\delta(s, u) = \infty$ e o algoritmo não poderia ter encontrado $u.d > \delta(s, u)$;
- Seja p um caminho de peso mínimo entre s e u ($w(p) = \delta(s, u)$).
- Como $s \in S$ e $u \notin S$, então no caminho p existe pelo menos uma aresta que conecta um vértice de S com um vértice que não está em S. Seja (x, y) a primeira aresta em p que faz isso ($x \in S$ e $y \notin S$).

```
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}
7      for v ∈ G.adj[u]
8          RELAX(u, v, w)
```

Análise de corretude

Invariante: no início de cada iteração do laço **while** , $v.d = \delta(s, v)$ para todos $v \in S$.

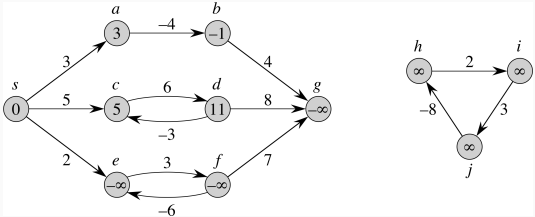
Manutenção: Temos que mostrar que quando o vértice u é extraído da fila (linha 5) e adicionado ao conjunto S (linha 6) $u.d = \delta(s, u)$. Vamos supor que $u.d > \delta(s, u)$ e derivar uma contradição.

- Se não existem arestas de peso negativo no grafo, então $\delta(s, x) + w(x, y) \leq \delta(s, u)$.
- Pela hipótese indutiva $x.d = \delta(s, x)$ e portanto $x.d + w(x, y) \leq \delta(s, u)$.
- A aresta (x, y) foi relaxada quando x foi adicionada a S, portanto $y.d \leq x.d + w(x, y)$.
- Qual a relação entre os valores de d do vértice u e y? $u.d \leq y.d$ porque u foi escolhido primeiro que y.
- $u.d \leq y.d \leq x.d + w(x, y) \leq \delta(s, u)$, uma contradição!

Arestas com pesos negativos

Nós vimos que a corretude do algoritmo de Dijkstra requer que o grafo não tenha arestas de peso negativo. Mas e se o grafo tiver arestas de peso negativos, podemos determinar os caminhos de peso mínimo? Vamos fazer um exemplo e pensar nessa questão.

Arestas com pesos negativos



O que podemos observar em relação as arestas de peso negativo e os caminhos de peso mínimo?

- Arestas de peso negativo podem gerar ciclos de peso negativo;
- Os caminhos de peso mínimo que não envolvem ciclos de peso negativo continuam bem definidos;
- Os ciclos de peso negativo que são acessíveis a partir da origem tornam “impossível” enumerar alguns caminhos de peso mínimo (ex: $\langle s, e, f, e, f, \ldots \rangle$).

Como lidar com caminhos entre s e v que envolvam ciclos de peso negativos?

- Vamos ajustar a definição de peso de caminho mínimo para estes casos para $\delta(s, v) = -\infty$;
- Não podemos enumerar os vértices do caminho, mas será que podemos identificar que ele contém um ciclo de peso negativo?

Vamos voltar para as hipóteses de algoritmos baseadas em programação dinâmica e melhoramento iterativo e verificar se podemos empregá-las para encontrar caminhos mínimos em grafos com arestas de peso negativo.

Programação dinâmica: Expressar os pesos dos caminhos mínimos que usam no máximo k arestas, em termos dos pesos dos caminhos mínimos que usam no máximo $k - 1$ arestas.

Melhoramento iterativo: Iniciar com uma árvore de caminhos qualquer e tentar melhorar os caminhos trocando uma aresta da árvore por uma que não está na árvore.

Existe alguma limitação aparente nessas abordagens que impedem que elas sejam usadas em grafos com arestas de peso negativo (e sem ciclos de peso negativo)?

- Programação dinâmica: encontra todos os caminhos mínimos com até $|V| - 1$ arestas, como os caminhos mínimos, mesmo com arestas de peso negativos (sem ciclos de peso negativo), não podem ter mais que $|V| - 1$ arestas, então as arestas de peso negativo parecem não mudar a hipótese
- Melhoramento iterativo: como o algoritmo só vai parar quando não for possível melhorar mais nenhum caminho, então as arestas de peso negativo parecem não mudar a hipótese

Sabendo que as duas hipóteses ainda são viáveis, agora temos que partir para descrição dos algoritmos e verificação de corretude.

Qual questão ficou em aberto na ideia baseada em melhoramento iterativo?

- Como e em que ordem verificar se as arestas que não estão na árvore podem ser inseridas na árvore e melhorar os caminhos.

Por exemplo, se em um determinado momento tivéssemos os caminhos $s \rightsquigarrow a \rightsquigarrow b$ e de b pra todos os demais vértices (não necessariamente direto), como verificar se podemos "colocar" a aresta (a, b) na árvore?

Verificando se o peso do caminho $s \rightsquigarrow a \to b$ é menor do que o peso do caminho atual $s \rightsquigarrow b$, ou seja, relaxando a aresta (a, b)! Como o peso do caminho $s \rightsquigarrow b$ mudou, temos que mudar o peso de todos os caminhos que começam com b... e como fazer isso? Relaxando as arestas dos caminhos. Então, de fato, não precisamos apenas relaxar as arestas que não estão na árvore, pode ser necessário relaxar as próprias arestas da árvore. A questão continua, em que ordem?

E a ideia baseada em programação dinâmica?

- Nós escrevemos a função recursiva mas não discutimos como implementá-la.

$\delta^k(s, v)$ – peso do caminho mínimo de s para v que utiliza até k arestas

$$\delta^k(s,v) = \begin{cases} 0 & \text{se } s = v \text{ e } k = 0 \\ \infty & \text{se } s \neq v \text{ e } k = 0 \\ \min \begin{cases} \delta^{k-1}(s,v) \\ \min\limits_{(u,v)\in E} (\delta^{k-1}(s,u) + w(u,v)) \end{cases} & \text{caso contrário} \end{cases}$$

O que o "caso contrário" nos diz?

- O caminho mínimo de s para v com até k arestas é o mesmo que o caminho mínimo com até $k-1$ arestas; ou
- O caminho mínimo de s para v com até k arestas tem exatamente k arestas e então pode ser obtido a partir de um caminho mínimo com $k-1$ arestas. Por que isso é verdade? Porque caminhos mínimos tem subestrutura ótima!

Como seria uma implementação *bottom-up*?

- Começamos, de forma trivial, com uma árvore de caminhos mínimos com até 0 aresta;
- Depois, a partir da árvore de caminhos mínimos com até 0 aresta, encontramos a árvore de caminhos mínimos com até 1 aresta;
- Depois, a partir da árvore de caminhos mínimos com até 1 aresta, encontramos a árvore de caminhos mínimos com até 2 aresta;
- E assim por diante até caminhos mínimos com até $|V| - 1$ arestas.

Note de que certa forma esse é um algoritmo de melhoramento iterativo

- Os caminhos mínimos com até $k - 1$ arestas são uma estimativa para os caminhos mínimos com até $|V| - 1$ arestas;
- Na iteração k, buscamos melhorar as estimativas obtidas na iteração $k - 1$.

Esta "visão" de uma abordagem de melhoramento iterativo é mais interessante que a outra, isto porque ela define claramente uma forma de progresso para o algoritmo e o momento de parada.

Vamos seguir com a ideia de programação dinâmica e escrever o algoritmo!

O algoritmo precisa fazer $|V| - 1$ iterações. Em uma iteração $k > 0$ precisamos construir uma nova árvore a partir da árvore da iteração $k - 1$, como fazemos isso? "Aplicando" a equação para cada vértice.

```
// Computar v.d^0 e v.π^0 para todo v ∈ V
for k = 1 to |V| − 1
    for each vertex v ∈ V
        v.d^k = v.d^(k−1)
        v.π^k = v.π^(k−1)
        for each edge (u, v) ∈ E
            if v.d^k > u.d^(k−1) + w(u, v)
                v.d^k = u.d^(k−1) + w(u, v)
                v.π^k = u
```

O que poderia ser difícil de implementar nesse código?

Fazer a repetição usando as arestas que entram em v. A dificuldade existe porque em uma lista de adjacências temos as arestas que saem de um vértice e não as que entram.

Como resolver esse problema? Criando uma lista de adjacências com as arestas que entram nos vértices!

Tem outra maneira? Sim!

// Computar $v.d^0$ e $v.\pi^0$ para todo $v \in V$
for $k = 1$ **to** $|V| - 1$
    for each vertex $v \in V$
        $v.d^k = v.d^{k-1}$
        $v.\pi^k = v.\pi^{k-1}$
        for each edge $(u, v) \in E$
            if $v.d^k > u.d^{k-1} + w(u, v)$
                $v.d^k = u.d^{k-1} + w(u, v)$
                $v.\pi^k = u$

Você consegue identificar algo familiar no código? O relaxamento da aresta (u, v).

Qual é o propósito do relaxamento? Tentar melhorar a estimativa de caminho mínimo para v.

Da forma que o código está escrito, as tentativas de melhora para v são feitas uma após a outro. Isto é necessário ou poderíamos intercalar tentativas? Tentar uma melhora para um vértice, depois tentar para outro e assim por diante até tentar todas as melhoras para todos os vértices?

Podemos tentar as melhoras em qualquer ordem!

// Computar $v.d^0$ e $v.\pi^0$ para todo $v \in V$
for $k = 1$ **to** $|V| - 1$
    for each vertex $v \in V$
        $v.d^k = v.d^{k-1}$
        $v.\pi^k = v.\pi^{k-1}$
    for each edge $(u, v) \in E$
        if $v.d^k > u.d^{k-1} + w(u, v)$
            $v.d^k = u.d^{k-1} + w(u, v)$
            $v.\pi^k = u$

Para facilitar a programação verificamos as arestas na ordem que elas aparecem nas listas de adjacências.

Tem mais alguma coisa que podemos modificar no código para deixar a implementação mais fácil? Ao invés de cada vértice ter atributos d e π para cada valor de k, cada vértice pode ter apenas um atributo d e π (vamos ver a seguir porque isso é possível). Isso permite três simplificações:

- O uso de INITIALIZE-SINGLE-SOURCE para computar os valores iniciais de d e π
- A remoção da repetição que inicializa os atributos d e π da iteração k a partir dos valores da iteração $k - 1$
- O uso de RELAX

O algoritmo que obtemos é chamado de Bellman-Ford.

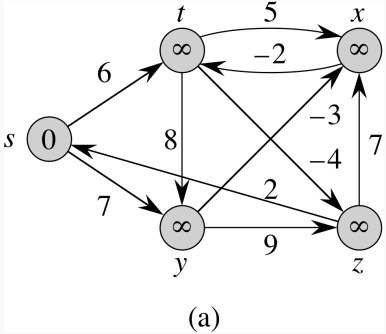
Bellman-Ford(G, w, s)

1 Initialize-Single-Source(G, s)
2 **for** $i = 1$ **to** $|G.V| - 1$
3 **for** each edge $(u, v) \in G.E$
4 Relax(u, v, w)
5 **for** each edge $(u, v) \in G.E$
6 **if** $v.d > u.d + w(u, v)$
7 **return** FALSE
8 **return** TRUE

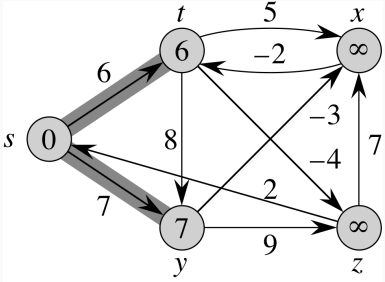
Análise do tempo de execução

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

Relaxação das arestas na ordem $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$

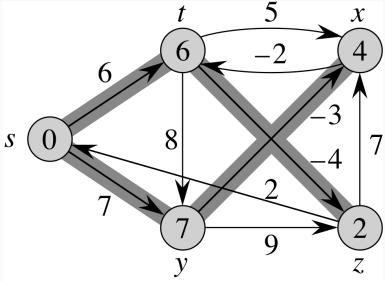


Relaxação das arestas na ordem $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$



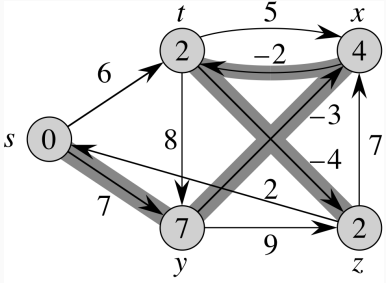
(b)

Relaxação das arestas na ordem $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$



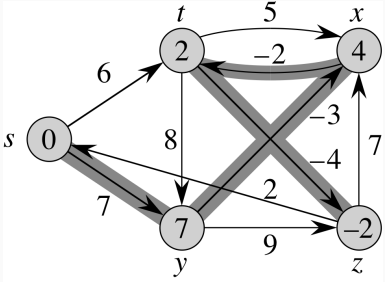
(c)

Relaxação das arestas na ordem $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$



(d)

Relaxação das arestas na ordem $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$



(e)

Nos derivamos o algoritmo a partir do modelo de programação dinâmica. Na iteração k o algoritmo fazia:

- Na primeira versão, selecionava um vértice, inicializa d^k e π^k e relaxava as arestas que "entravam" no vértice;
- Na segunda versão, inicializa d^k e π^k para todos os vértices e depois relaxava todas as arestas;
- Na terceira versão (Bellman-Ford), usava apenas uma "versão" de d e π e relaxada todas as arestas;

Porque o algoritmo de Bellman-Ford funciona?

Após INITIALIZE-SINGLE-SOURCE, qual é a única forma utilizada pelo algoritmo para mudar as estimativas de caminhos mínimos (atributo d)? A função de relaxamento.

Vamos pensar em caminhos mínimos que estão sendo (ou já foram) construídos pelo algoritmo e o uso da função de relaxamento.

Se $s \rightsquigarrow v$ é um caminho mínimo e o algoritmo já encontrou esse caminho mínimo, o que acontece se a função de relaxamento for chamada para qualquer aresta desse caminho? Nada. O caminho continua do jeito que está, ele já é mínimo!

Se $s \rightsquigarrow u \rightarrow v$ é um caminho mínimo e o algoritmo já encontrou o caminho mínimo $s \rightsquigarrow u$, isto é $u.d = \delta(s, u)$, o que podemos afirmar após o relaxamento da aresta (u, v)? Que $v.d = \delta(s, v)$. O relaxamento pode ou não mudar a estimativa para v:

- Se $v.d$ já fosse $\delta(s, v)$, então o algoritmo já tinha encontrado um caminho mínimo para v e o relaxamento não faz nada.
- Por outro lado, se $v.d > \delta(s, v)$, então o relaxamento mudaria $v.d$ para $u.d + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$.

Este é o Lema 24.14, propriedade de convergência, apresentado no livro.

Se $\langle v_0, v_1, \ldots, v_k \rangle$ é uma caminho mínimo de $s = v_0$ até v_k e:

- Em um momento qualquer, mesmo que após relaxamentos diversos, a aresta (v_0, v_1) é relaxada; e
- Em um momento posterior, mesmo que após relaxamentos diversos, a a aresta (v_1, v_2) é relaxada; e
- $\ldots$
- Em um momento posterior, mesmo que após relaxamentos diversos, a aresta (v_{k-1}, v_k) é relaxada.

O que podemos afirmar sobre $v_k.d$ no final desse processo? Que $v_k.d = \delta(s, v_k)$!

Este o Lema 24.15, propriedade de relaxamento de caminho, apresentado no livro.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Análise de corretude

Cada iteração do laço da linha 2 todas as arestas são relaxadas:

- A primeira iteração relaxa todas as primeiras arestas de todos os caminhos mínimos que têm pelo menos uma aresta
- A segunda iteração relaxa todas as segundas arestas de todos os caminhos mínimos que têm pelo menos duas arestas
- ...
- A $|V| - 1$-ésima iteração relaxa todas as $|V| - 1$-ésima arestas de todos os caminhos que têm $|V| - 1$ arestas

Mesmo sem saber os caminhos mínimos, o algoritmo relaxa todas as arestas de todos os caminhos mínimos em sequência, garantindo, pela propriedade de relaxamento de caminho, que os caminhos mínimos serão calculados corretamente.

Tanto o algoritmo de Dijkstra quanto o algoritmo de Bellman-Ford são baseados na mesma ideia

- Inicializar os atributos d e π
- Relaxar as arestas em uma ordem específica

O algoritmo de Dijkstra relaxa cada aresta apenas uma vez e só funciona para grafos sem arestas de peso negativo.

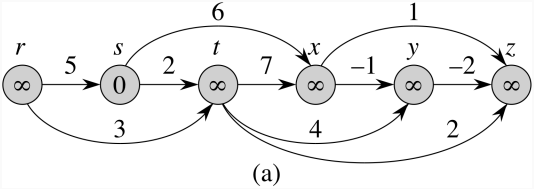
O algoritmo relaxa todas as arestas $|V| - 1$ vezes e funciona para grafos com arestas de peso negativo e identifica grafos com ciclos de peso negativo.

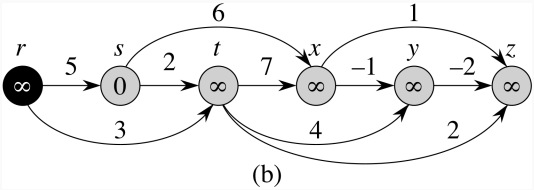
Exercício

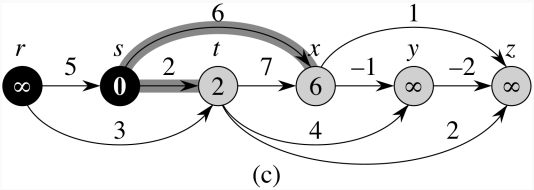
Projete um algoritmo que encontre os caminhos mínimos de única origem em um grafo acíclico orientado.

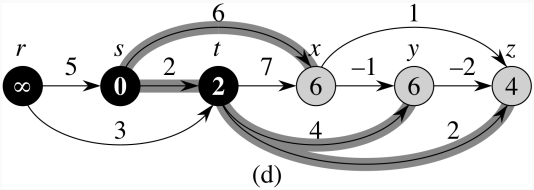
Solução

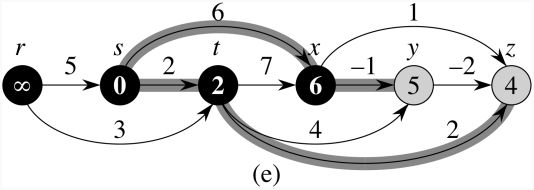
- Relaxar as arestas em uma ordem topológica dos vértices

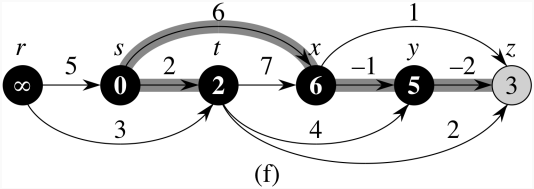

r
s
t
x
y
z
∞
0
∞
∞
∞
∞
5
2
7
−1
−2
6
1
3
4
2
(a)

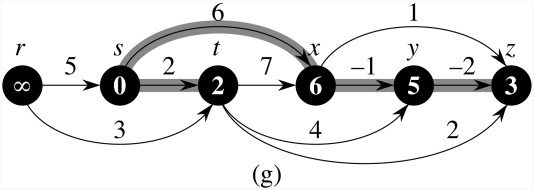

r
s
t
x
y
z
∞
0
∞
∞
∞
∞
5
2
7
−1
−2
6
1
3
4
2
(b)


6
1
r
s
t
x
y
z
5
2
7
−1
−2
∞
0
2
6
∞
∞
3
4
2
(c)


6
1
r
s
t
x
y
z
5
2
7
−1
−2
∞
0
2
6
6
4
3
4
2
(d)


r
s
t
x
y
z
∞
0
2
6
5
4
5
2
7
−1
−2
6
1
3
4
2
(e)


r
s
t
x
y
z
∞
0
2
6
5
3
6
1
5
2
7
−1
−2
3
4
2
(f)


r
s
t
x
y
z
∞
0
2
6
5
3
5
2
7
−1
−2
6
1
3
4
2
(g)

Por que este 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.

DAG-SHORTEST-PATHS(G, w, s)

1 ordenar topologicamente $G.V$
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** vértice u em ordem topológica
4 **for** each vertex $v \in G.adj[u]$
5 RELAX(u, v, w)

Análise do tempo de execução

- A ordenação topológica da linha 1 tem tempo $\Theta(V + E)$
- INITIALIZE-SINGLE-SOURCE na linha 2 tem tempo $\Theta(V)$
- Nos laços das linhas 2 e 3 a lista de adjacências de cada vértices é visitada apenas uma vez, totalizando $V + E$ (análise agregada), como o relaxamento de cada aresta custa $O(1)$, o tempo total é $\Theta(E)$
- Portanto, o tempo de execução do algoritmo é $\Theta(V + E)$

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