Caminhos mínimos de única origem

Algoritmos em Grafos

Marco A L Barbosa

Conteúdo

O estudo utilizando apenas este material **não é suficiente** para o entendimento do conteúdo. Recomendamos a leitura das referências no final deste material e a resolução (por parte do aluno) de todos os exercícios indicados.

Introdução

Introdução

- Como encontrar o caminho mínimo entre duas cidades?

Introdução

- Como encontrar o caminho mínimo entre duas cidades?
- Vamos estudar este tipo de problema, que é conhecido como **problema de caminho mínimo**
- Entrada
  - Um grafo orientado $G = (V, E)$
  - Uma função peso $w : E \to \mathbb{R}$

Introdução

- O **peso do caminho** $p = \langle v_0, v_1, \ldots, v_k \rangle$ é
  - a soma dos pesos das arestas no caminho
  - $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- Definimos o **peso do caminho mínimo** deste u até v como

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

- 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 algoritmos de caminhos mínimos que funciona para grafos não valorados, isto é, as arestas tem peso unitário

Tipos de problemas de caminho mínimo

- **Origem única**: Encontrar um caminho mínimo a partir de uma dada origem $s \in V$ até todo vértice $v \in V$

Tipos de problemas de caminho mínimo

- **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

Tipos de problemas de caminho mínimo

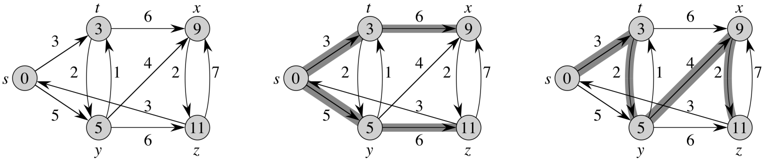
- **Origem única**: Encontrar um caminho mínimo a partir de uma dada origem $s \in V$ até todo vértice $v \in V$
- **Destino único**: Encontrar um caminho mínimo até um determinado vértice de destino t a partir de cada vértice v
- **Par único**: Encontrar o caminho mínimo de u até v

Tipos de problemas de caminho mínimo

- **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 deste u até v para todo par de vértices u e v

Exemplo

- Exemplo de caminhos mínimos de única origem



(a) (b) (c)

- 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

Caminhos mínimos

- Veremos algumas características dos caminho mínimos

Subestrutura ótima

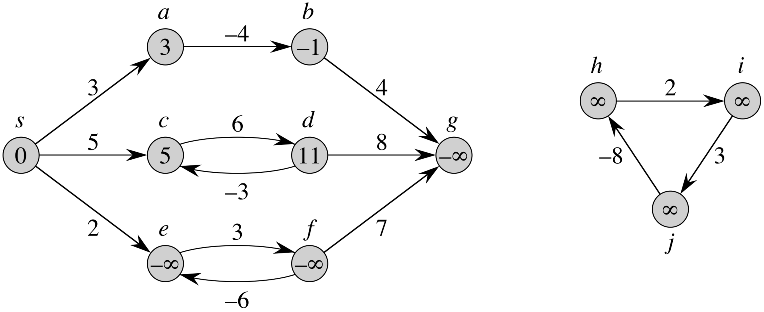
Subestrutura ótima

- 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? (Comentado em sala, veja o livro para a prova)

Ciclos e arestas de pesos negativos

Arestas com 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

- Caminhos mínimos podem conter ciclos? Não
  - Peso negativo, acabamos de descartar
  - Peso positivo, podemos obter um caminho mínimo eliminando o ciclo
  - Peso nulo, não existe razão para usar tal ciclo

Ciclos

- Caminhos mínimos podem conter ciclos? Não
  - Peso negativo, acabamos de descartar
  - Peso positivo, podemos obter um caminho mínimo eliminando o ciclo
  - Peso nulo, não existe razão para usar tal ciclo
- Qualquer caminho acíclico em um grafo $G = (V, E)$ contém no máximo $|V|$ vértices distintos e no máximo $|V| - 1$ arestas
  - Desta forma, vamos restringir a nossa atenção para ciclos com no máximo $|V| - 1$ arestas

Representação

Representação

- Representamos os caminhos mínimos de forma semelhante as árvores primeiro em largura produzidas pelo `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 $v.d \geq \delta(s, v)$
    - Vamos chamar $v.d$ 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

Relaxamento

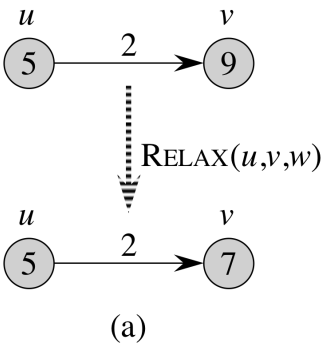
Relaxamento

- Sendo os vértices inicializados com a função

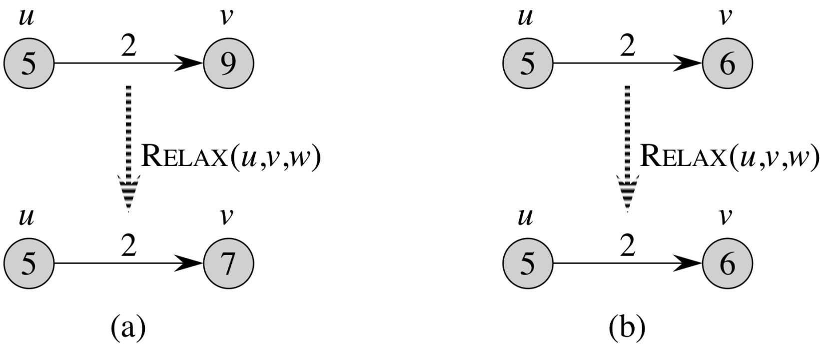
```
initialize-single-source(G, s)
for cada vértice v ∈ G.V
    v.d = ∞
    v.π = nil
s.d = 0
```

- Podemos melhorar a estimativa do caminho mínimo para v, indo através de u e seguindo (u, v)?

Relaxamento



Relaxamento



relax(u, v, w)
if $v.d > u.d + w(u, v)$
    $v.d = u.d + w(u, v)$
    $v.\pi = u$

Propriedades

Propriedades

- Desigualdade de triângulos (Lema 24.10)
  - Para toda $(u, v) \in E$, temos que $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Propriedades

- Desigualdade de triângulos (Lema 24.10)
  - Para toda $(u, v) \in E$, temos que $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- Para as próximas propriedades supomos 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 (continuação)

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

Propriedades (continuação)

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

Propriedades (continuação)

- Propriedade do limite superior (Lema 24.11)
  - Sempre temos $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda
- Propriedade de nenhum caminho (Corolário 24.12)
  - Se $\delta(s, v) = \infty$, então sempre $v.d = \infty$
- Propriedade de convergência (Lema 24.14)
  - Se $s \rightsquigarrow u \to v$ é um caminho mínimo, $u.d = \delta(s, u)$ e `relax(u, v, w)` é chamado, então, em todos os momentos após a chamada, temos $v.d = \delta(s, v)$

Propriedades (continuação)

- Propriedade do limite superior (Lema 24.11)
  - Sempre temos $v.d \geq \delta(s, v)$ para todo v. Uma vez que $v.d = \delta(s, v)$, ele nunca muda
- Propriedade de nenhum caminho (Corolário 24.12)
  - Se $\delta(s, v) = \infty$, então sempre $v.d = \infty$
- Propriedade de convergência (Lema 24.14)
  - Se $s \rightsquigarrow u \to v$ é um caminho mínimo, $u.d = \delta(s, u)$ e `relax(u, v, w)` é chamado, então, em todos os momentos após a chamada, temos $v.d = \delta(s, v)$
- Propriedade de relaxamento de caminho (Lema 24.15)
  - Seja $p = \langle v_0, v_1, \ldots, v_k \rangle$ o caminho mínimo de $s = v_0$ 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 $v_k.d = \delta(s, v_k)$

Algoritmos

Ideia dos algoritmos

- Os algoritmos que veremos usam a mesma ideia
  - inicializar os atributos $v.d$ e $v.\pi$
  - relaxar as arestas
- Eles diferem na ordem e na quantidade de vezes que cada aresta é relaxada

Algoritmo de Bellman-Ford

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 e devolve `false`, caso contrário, devolve `true`
- Calcula $v.d$ e $v.\pi$ para todo $v \in V$
- Ideia
  - Relaxar todas as arestas, $|V| - 1$ vezes

Algoritmo de Bellman-Ford

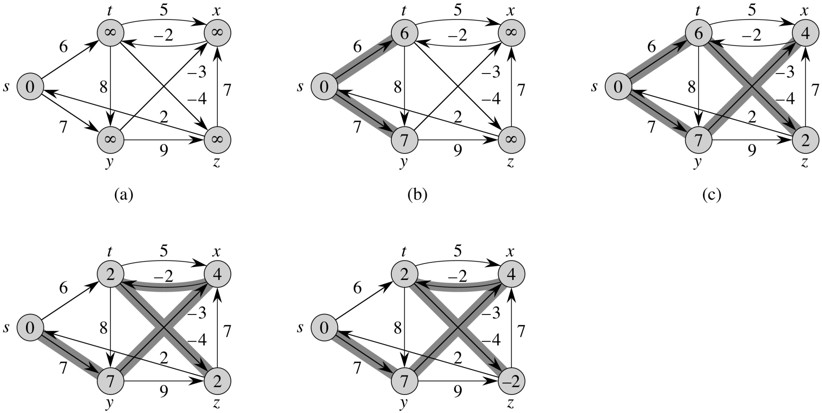
```
bellman-ford(G, w, s)
initialize-single-source(G, s)
for i = 1 to |G.V| - 1
  for cada aresta (u, v) em G.E
    relax(u, v, w)
for cada aresta (u, v) em G.E
  if v.d > u.d + w(u, v)
    return false
return true
```

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.E
4     relax(u, v, w)
5 for cada aresta (u, v) em 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 $O(V \cdot E)$
  - O laço das linha 5 a 7 demora $O(E)$
  - Tempo de execução do algoritmo $\Theta(V \cdot E)$

Algoritmo de Bellman-Ford Exemplo



(a) (b) (c) (d) (e)

As arestas foram relaxadas na ordem
$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$

Algoritmo de Bellman-Ford

- Por que este algoritmo funciona?

Algoritmo de Bellman-Ford

- Por que este 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 gaos

Algoritmo para gaos

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

Caminhos mínimos de única origem em gaos

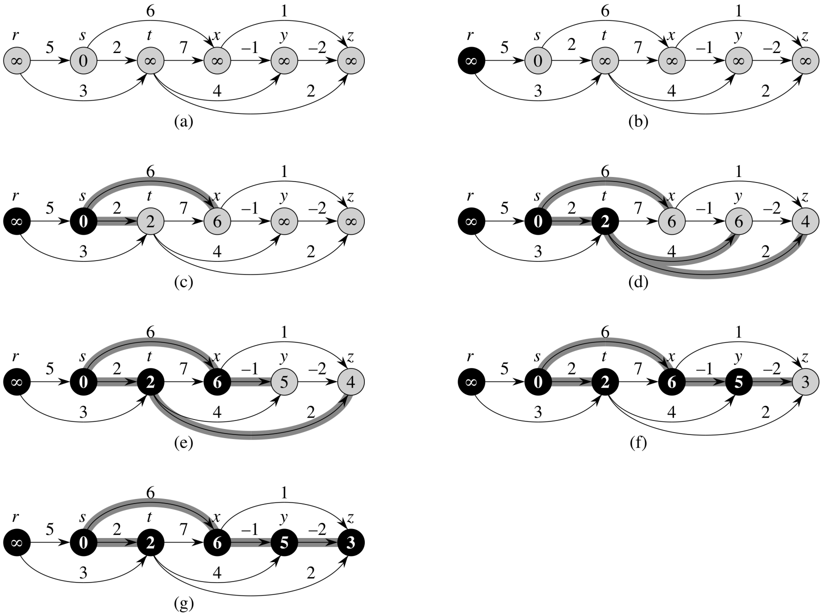
```
dag-shortest-paths(G, w, s)
ordenar topologicamente os vértices de G
initialize-single-source(G, s)
for cada vértice u tomado na ordem topológica
  for cada vértice v em u.adj
    relax(u, v, w)
```

Caminhos mínimos de única origem em gaos

```
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 $\Theta(V + E)$
  - `initialize-single-source` na linha 2 demora $\Theta(V)$
  - Nos laços das linhas 2 e 3 a lista de adjacências de cada vértices é visitada apenas uma, 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 $\Theta(V + E)$

Caminhos mínimos de única origem em gaos



Caminhos mínimos de única origem em gaos

- 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

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 gao: sequencia de serviços
- Caminho crítico: é um caminho mais longo pelo gao
  - 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

- Podemos encontrar um caminho crítico de duas maneiras:

Aplicação

- Podemos encontrar um caminho crítico de duas maneiras:
  - Tornando negativos os pesos das arestas e executando `dag-shortest-paths`; ou

Aplicação

- Podemos encontrar um caminho crítico de duas maneiras:
  - Tornando negativos os pesos das arestas e executando `dag-shortest-paths`; ou
  - Executando `dag-shortest-paths`, substituindo "∞" por "$-\infty$" na linha 2 de `initialize-single-source` e "$>$" por "$<$" no procedimento `relax`

Algoritmo de Dijkstra

Algoritmo de Dijkstra

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

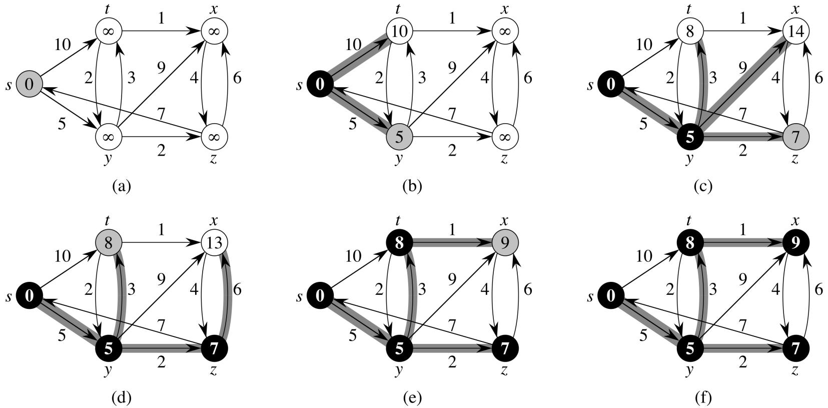
Algoritmo de Dijkstra

- 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)
initialize-single-source(G, s)
S = {}
Q = G.V
while Q != {}
  u = extract-min(Q)
  S = S U {u}
  for cada vértice v em u.adj
    relax(u, v, w)
```

Algoritmo de Dijkstra Exemplo:



Algoritmo de Dijkstra

- Análise do tempo de execução
  - Linha 1 $\Theta(V)$
  - Linhas 3 a 8 $O(V + E)$ (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 $|E|$ 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 $|V|$, armazenamos o valor $v.d$ 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 $O(V^2 + E) = O(V^2)$

Algoritmo de Dijkstra

- Análise do tempo de execução
  - Arranjos simples
    - Como os vértices são enumerados de 1 a $|V|$, armazenamos o valor $v.d$ 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 $O(V^2 + E) = O(V^2)$
  - Heap
    - Se o grafo é esparso, em particular, $E = o(V^2/\lg V)$ 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 $O((V + E)\lg V + V)$, que é $O(E \lg V)$ 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 $O(V \lg V + E)$

Algoritmo de Dijkstra

- Porque este 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: precisamos mostrar que $u.d = \delta(s, u)$ quando u é adicionado a S em cada iteração (Comentado em sala, veja o livro para a prova completa)
  - Feito em sala

Referências

Referências

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