Construção e Análise de Algoritmos

aula 26: Caminhos mais curtos entre todos os pares

1 Introdução

> *Em mais uma tentativa de convencer as pessoas de que o projeto de pavimentação das estradas era uma realidade, o governo do estado anunciou que os correios iriam atualizar as suas bases de dados e recalcular suas tabelas de roteamento.*
>
> *Se uma pessoa em Guaranámiranga, digamos, envia uma encomenda para alguém em Cajuzeiro do Norte, o ideal é que a encomenda siga pelo caminho mais curto possível.*
>
> *De fato, a encomenda não segue direto de Cajuzeiro do Norte, mas é enviada para a agência dos correios na primeira cidade no caminho mais curto, digamos, Pacotin.*
>
> *De Pacotin ela pode ir para Qui Será Meufim, e assim por diante até chegar ao seu destino final.*
>
> *Quer dizer, as agências de cada cidade não precisam saber os caminhos completos até todos os destinos possíveis, mas possuem apenas uma tabela de roteamento que indica a próxima cidade no caminho.*
>
> *Apesar disso, para calcular as tabelas de roteamento, é necessário sim calcular os caminhos mais curtos entre todos os pares de cidades.*
>
> *E isso é uma tarefa da agência central de Fortaleza.*

Uma solução simples para esse problema consiste em repetir a execução do algoritmo que vimos na aula 16, uma vez para cada cidade.

Essa solução leva tempo

$$n \times O(m \log m) \quad = \quad O(nm \log m)$$

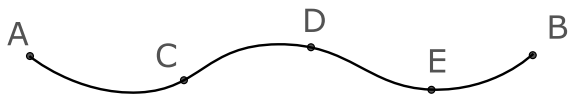
onde n é o número total de cidades, e m é o número total de estradas conectando pares de cidades.

E, levando em conta que m pode ser da ordem de n^2, o tempo de execução pode chegar a $O(n^3 \log m)$.

A seguir, nós vamos ver uma solução mais eficiente, que nos dará a oportunidade de praticar a técnica de programação dinâmica um pouco mais.

2 Visualizando a decomposição do problema

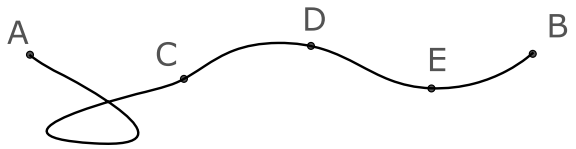
Uma observação simples já nos coloca no caminho da solução.



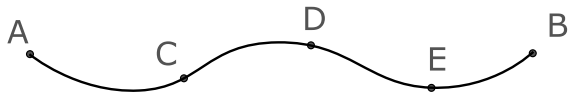
Suponha que esse é um caminho mais curto entre A e B.

Então, nós já sabemos que o trecho entre A e C também é um caminho mais curto.

O raciocínio que fizemos na aula 17 é que, se isso não é o caso



então esse trecho pode ser substituído por um outro mais curto



reduzindo também a distância entre A e B.

Ora, mas se isso é o caso, então nós não tínhamos, pra começo de conversa, um caminho mais curto entre A e B.

Legal.

Suponha, portanto, que nós temos mesmo um caminho mais curto entre A e B.

O que significa isso?

Bom, isso significa que o trecho entre A e C é uma caminho mais curto entre esses dois pontos.

E que o trecho entre C e E também é um caminho mais curto.

E o trecho entre D e B também.

De fato, toda aresta que aparece nesse caminho é também um mais curto.

E todo segmento de 2 arestas também é um caminho mais curto.

E todo segmento com 3 arestas também.

Caramba!

Quando nós descobrimos um caminho mais curto entre dois vértices de um grafo, na verdade nós descobrimos um monte de caminhos mais curtos entre um monte de pares de vértices.

Faz sentido, então, querer reaproveitar esse trabalho e calcular caminhos mais curtos entre todos os pares de vértices do grafo de uma só vez — ao invés de calcular caminhos mais curtos a partir de cada vértice em separado.

Legal.

Mas, como nós podemos fazer isso?

A resposta, é claro, é: Vamos usar programação dinâmica!

Ok, mas como?

Quer dizer, a estratégia que temos utilizado para fazer a programação dinâmica funcionar consiste em

- encontrar uma maneira de decompor o problema em (um ou mais) subproblemas menores
- descobrir como as soluções dos subproblemas podem ser utilizadas para calcular a solução do problema original
- examinar todas as formas de decomposição possíveis para obter a solução ótima

Mas, como é que nós decompomos o problema dos caminhos mais curtos entre todos pares de vértices?

Será que a ideia é quebrar o grafo no meio?

```
< Figura: quebrando o grafo no meio >
```

Ou será que é deixando o vértice w de lado, resolvendo o subproblema G' e depois incorpando o vértice w nessa solução?

```
< Figura: deixando o vértice w de lado >
```

(Você consegue fazer essas ideias funcionarem?)

(A que custo?)

A seguir, nós vamos exercitar uma maneira diferente de pensar a programação dinâmica.

3 Estratégia de composição de soluções

Você lembra que, na discussão (b) da aula 21, nós dissemos que algumas pessoas gostam de ver a programação dinâmica como uma estratégia de composição de soluções — ao invés de decomposição de problemas?

(E você consegue ver que, em certo sentido, as duas coisas são a mesma coisa?)

Pois bem, hoje nós temos um exemplo onde esse ponto de vista nos leva mais facilmente à solução do problema.

E a observação chave já apareceu na seção anterior: *um caminho mais curto é formado por um monte de caminhos mais curtos.*

A ideia, então, é começar com os caminhos mais curtos de todos.

Utilizando esses caminhos, nós montamos caminhos mais curtos um pouquinho mais longos.

A seguir, nós utilizamos esses caminhos para montar caminhos ainda mais longos.

E assim por diante ...

Mas, o que significa isso exatamente?

Quer dizer, o que é um caminho mais curto de todos?

Bom, nós vamos assumir que isto é um caminho mais curto formado por apenas uma aresta.

Isto é,

- nós vamos encontrar primeiro caminhos mais curtos com apenas 1 aresta
- utilizando esses caminhos, nós montamos caminhos mais curtos com (até) 2 arestas
- utilizando esses caminhos, nós montamos caminhos mais curtos com (até) 3 arestas
- e assim por diante ...

Certo.

No início, os caminhos mais curtos com 1 aresta são as arestas do grafo — e se não existe uma aresta entre dois vértices u e v, então esse caminho não existe.

De fato, nós não temos como saber (ainda) se uma aresta $u \to v$ é mesmo o caminho mais curto de u até v.

Mas, nós tomamos isso como a nossa estimativa inicial.

Isso é análogo a quebrar o problema em um certo ponto, sem ter a certeza de que isso vai nos levar à solução ótima do problema.

O próximo passo consiste em verificar como os caminhos de 1 aresta podem ser usados para montar caminhos mais curtos com 2 arestas.

Mas, isso é fácil: caminhos mais curtos com 2 arestas possuem um caminho mais curto com 1 aresta no seu interior

```
< Figura: decomposição de caminhos com 2 arestas >
```

Quer dizer, nós podemos visualizar os caminhos com 2 arestas como um caminho de 1 aresta

que foi extendido com mais uma aresta.

Em outras palavras, basta percorrer todos os caminhos com 1 aresta e extendê-los com uma aresta adicional para obter os caminhos com 2 arestas.

(E assim por diante ... — Você já viu a recursão?)

Na prática, um caminho com 2 arestas pode ser melhor ou não que o caminho com apenas uma aresta

`< Figura: comparação de caminhos de tamanhos 1 e 2 >`

e a ideia é que nós ficamos com aquele que é o melhor dos dois.

Além disso, na prática, nós podemos ter vários caminhos de u até v com 2 arestas

`< Figura: vários caminhos de tamanhos 2 >`

e a ideia é escolher o melhor deles para compará-lo com o caminho de tamanho 1.

Uma vez que os caminhos mais curtos com (até) 2 arestas já foram encontrados, nós estamos em condições de calcular os caiminhos mais curtos com (até) 3 arestas

`< Figura: recursão para caminhos com (até) 3 arestas >`

Isto é, nós examinamos todas as maneiras possíveis de ir do vértice u até o vértice v concatenando caminhos com (até) 2 arestas até um vértice intermediário w_j com a aresta $w_j \to v$.

A seguir, nós comparamos esse resultado com o caminho mais curto com (até) 2 arestas que nós já temos para u e v.

Abaixo nós temos o algoritmo que implementa essa ideia.

```
   Procedimento  Caminhos+Curtos-TP-PD1.0 ( G: grafo c/ n vértices )
    {
 1.    Para cada par de vértices  u,v
        {
 2.        Se a aresta  u -> v  está no grafo $G$
            {
 3.            u.dist[v]  <--  dist(u,v);    u.prox[v]  <--  v
            }
 4.        Senão
            {
 5.            u.dist[v]  <--  infinito;     u.prox[v]  <--  Nulo
       }   }

 6.    Para  k <-- 2  Até  n
        {
 7.        Para cada par de vértices  u,v
            {
 8.            dist-aux  <--  infinito;     prox-aux  <--  Nulo

 9.            Para cada vértice intermediário  w
                {
10.                Se (  dist-aux  >  u.dist[w] + dist(w,v)  )
                    {
11.                    dist-aux  <--  u.dist[w] + dist(w,v)
12.                    prox-aux  <--  u.prox[w]
                }   }

13.            Se (  dist-aux  <  u.dist[v]  )
                {
14.                u.dist[v]  <--  dist-aux;     u.prox[v]  <--  prox-aux
    }   }   }   }
```

Você percebeu que esse algoritmo utiliza as tabelas de roteamento para fazer os seus cálculos?

Quer dizer, cada vértice u possui uma tabela da forma

	dist	prox
...	...	...
x	17	w
y	15	u
...	...	...

implementada por meio dos campos `u.dist` e `u.prox`.

A coluna `dist` indica a estimativa que temos até o momento para a distância de `u` a todos os outros vértices.

E a coluna `prox` indica a primeira cidade em um caminho que tem essa distância.

Por exemplo, a distância estimada de u até x na tabela acima é igual a 17, e para realizar um

caminho com essa distância, nós começamos indo para a cidade `w` — em `w` nós continuamos seguinte para `w.prox[x]`, e assim por diante.

Certo.

Essas tabelas são inicializadas no laço das linhas `1-5`, para refletir os caminhos mais curtos com apenas 1 aresta, entre cada par de cidades.

A seguir, o laço da linha `6` irá aumentar o alcance desse caminhos, uma aresta a cada volta.

Isto é, para cada par de vértices `u, v`, o algoritmo examina todas as possibilidades de extender um caminho `u` até um vértice intermediário `w` com a aresta $w \to v$ (linhas `7-12`).

finalmente, a melhor dessas opções é comparada com a estimativa que tempos, que pode ser atualizada ou não.

Análise de complexidade

Observe primeiramente que todas as instruções do procedimento `Caminhos+Curtos-TP-PD1.0` executam em tempo $O(1)$.

Portanto, tudo o que temos a fazer é estimar a complexidade dos laços aninhados.

É fácil ver que o laço da linha `1` realiza $O(n^2)$ voltas.

Por outro lado,

- o laço da linha `6` realiza $O(n)$ voltas
- o laço da linha `7` realiza $O(n^2)$ voltas
- o laço da linha `9` realiza $O(n)$ voltas

de modo que a estrutura de laços aninhados como um todo tem complexidade $O(n^4)$.

Portanto, o tempo de execução do algoritmo é

$$O(n^2) \quad \times \quad O(n^4) \quad = \quad O(n^4)$$

4 Uma composição um pouco mais esperta

Mas, $O(n^4)$ não é melhor que $O(n^3 \log n)$, não é?

Quer dizer, nós fizemos uma grande confusão, e no final terminamos com um algoritmo menos eficiente.

Mas, uma ideia simples já nos permite ao menos empatar as coisas.

Veja só.

No momento em que nós já temos os caminhos mais curtos com (até) 2 arestas, nós podemos

pular o comprimento e ir direto para os caminhos mais curtos com (até) 4 arestas.

Quer dizer, um caminho mais curto com 4 arestas é a concatenação de dois caminhos mais curtos com 2 arestas

< Figura: concatenação de caminhos com (até) 2 arestas >

E se o caminho mais curto entre u e v só tiver 3 arestas, então ele também é a concatenação de dois caminhos com (até) 2 arestas

< Figura: concatenação de caminhos com (até) 2 arestas de tamanho 3 >

pois nesse caso, por exemplo, a aresta $w_2 \rightarrow v$ é o caminho mais curto com (até) 2 arestas entre w_2 e v.

(Você conseque ver isso?)

A seguir, uma vez que temos todos os caminhos mais curtos com (até) 4 arestas, nós podemos combiná-los para obter os caminhos mais curtos com (até) 8 arestas

< Figura: concatenação de caminhos com (até) 4 arestas >

E assim por diante ...

Para implementar essa ideia, basta modificar as linhas `6, 10` e `11` do código anterior

```
       Procedimento  Caminhos+Curtos-TP-PD2.0 ( G: grafo c/ n vértices )
        {
           ( . . . )

(*)  6.    Para  k <-- 1  Até  log n
            {
     7.        Para cada par de vértices  u,v
               {
     8.           dist-aux  <--  infinito;     prox-aux  <--  Nulo

     9.           Para cada vértice intermediário  w
                  {
(*) 10.              Se (  dist-aux  >  u.dist[w] + w.dist[v]  )
                     {
(*) 11.                 dist-aux  <--  u.dist[w] + w.dist[v]
    12.                 prox-aux  <--  u.prox[w]
                  }   }

           ( . . . )
```

Quer dizer, começando com caminhos de tamanho n, quantas vezes nós precisamos dobrar o tamanho dos caminhos até alcançar o tamanho máximo n?*

A resposta é $\lceil \log_2 n \rceil$.

E isso nos dá o laço da linha `6`.

Já as modificações das linhas `10` e `11` refletem o fato de que agora nós estamos examinando concatenações de caminhos $u \rightsquigarrow w$ e $w \rightsquigarrow v$.

Finalmente, é fácil ver que essas modificações reduzem a complexidade do algoritmo para $O(n^3 \log n)$.

Mas, as observações da Seção 2 indicavam que nós poderíamos fazer melhor que isso.

Como?

5 Uma composição bem esperta

Agora é a hora de resgatar uma ideia que nós já havíamos deixado para trás.

Quer dizer, a ideia de decompor o problema em um vértice w que é deixado de lado, e a parte restante G' do grafo.

`< Figura: deixando o vértice w de lado >`

Suponha, então, que nós resolvemos o subproblema dos caminhos mais curtos entre todos os pares de vértices de G'

`< Figura: solução do subproblema G' >`

Quer dizer, para todo par de vértices u, v em G', nós conhecemos o caminho mais curto de u até v passando apenas por vértices de G'.

Será que esse também é o caminho mais curto entre u e v no grafo G?

Bom, pode ser que sim e pode ser que não.

Para saber a resposta, é preciso examinar o caminho mais curto entre u e v que passa pelo vértice w.

Mas, isso é simplesmente a concatenação de um caminho $u \rightsquigarrow w$ com um caminho $w \rightsquigarrow v$

`< Figura: concatenação de u~>w e w~>v >`

*Na realidade, um caminho em G que não passa duas vezes pelo mesmo vértice pode ter no máximo $n - 1$ arestas.

A boa notícia é que esses dois caminhos só passam por vértices de G' — (Porque?).

A má notícia é que, como w não está em G', esses caminhos não foram calculados quando nós resolvemos o subproblema G'.

Mas, esse é um problema fácil de resolver.

E é aqui que está a esperteza.

A ideia é manter o vértice w no grafo, mas proibir que os caminhos mais curtos passem por ele.

Quer dizer, o novo subproblema G' consiste em

- Encontrar os caminhos entre todos os pares de vértices de G que não passam pelo vértice w (no meio do caminho)

E, basta uma ligeira modificação nas versões anteriores do algoritmo para que eles resolvam esse subproblema

```
7.        Para cada par de vértices  u,v  em  G

          ( . . . )

9.            Para cada vértice intermediário  z  diferente de  w
```

Uma vez que o subproblema foi resolvido, basta comparar a solução obtida com o caminho que passa por w para obter a solução final

```
  x.    Para cada par de vértices  u,v  em  G   (diferentes de w)
         {
x+1.        Se ( u.dist[v] > u.dist[w] + w.dist[v] )
             {
x+2.             u.dist[v] <-- u.dist[w] + w.dist[v] )

x+3.             u.prox[v] <-- u.prox[w]
             }
```

Legal.

Essa solução já pode ser implementada na forma de um algoritmo recursivo.

(Quer tentar?)

Mas, hoje nós estamos praticando a ideia de composição de soluções — e não decomposição de problemas.

Quer dizer, nós devemos começar com as soluções mais simples possíveis.

A seguir, nós combinamos essas soluções para obter soluções um pouquinho mais complicadas.

E continuamos assim até chegar nas soluções do problema original.

Mas, o que são os caminhos mais curtos mais simples de todos nesse caso?

Bom, são os caminhos mais curtos que não passam por nenhum vértice intermediário.

Isto é, as arestas do grafo!

E os caminhos mais curtos um pouquinho mais complicados, quais são?

Bom, esses são os caminhos mais curtos que podem passar apenas por um certo vértice intermediário w_1

`< Figura: concatenações possíveis de arestas de w1 >`

Quer dizer são concatenações de caminhos (ou arestas) $u \rightsquigarrow w_1$ com caminhos (ou arestas) $w_1 \rightsquigarrow v$ — que devem ser comparados com o caminho $u \rightsquigarrow v$ que nós já temos.

E depois?

Bom, depois nós permitimos que os nossos caminhos passem por um outro vértice w_2

`< Figura: concatenações possíveis de caminhos até e a partir de w2 >`

E assim por diante ...

Abaixo nós temos o pseudo-código do algoritmo que implementa essa ideia

```
   Procedimento  Caminhos+Curtos-TP-PD3.0 ( G: grafo c/ n vértices )
    {
 1.    { w1, w2, ..., wn }  <-- ordenação qualquer dos vértices de G

 2.    Para cada par de vértices  u,v
        {
 3.        Se a aresta  u -> v  está no grafo $G$
            {
 4.            u.dist[v]  <--  dist(u,v);    u.prox[v]  <--  v
            }
 5.        Senão
            {
 6.            u.dist[v]  <--  infinito;     u.prox[v]  <--  Nulo
        }   }

 7.    Para  i <-- 1  Até  n
        {
 8.        Para cada par de vértices  u,v   ( diferentes de wi )
            {
 9.            Se (  u.dist[v]  >  u.dist[wi] + wi.dist[v]  )
                {
10.                u.dist[v]  <--  u.dist[wi] + w.dist[v]
11.                u.prox[v]  <--  u.prox[wi]
    }   }   }   }
```

Agora, o laço dos vértices intermediários desapareceu completamente.

E o algoritmo executa em tempo $O(n^3)$.

Não é legal?