Fluxo em Redes

Paulo Feofiloff
http://www.ime.usp.br/~pf/

IME–USP, 30/12/2003

Prefácio

Estas notas de aula foram escritas em 2002 para as disciplinas de Otimização Combinatória (MAC5781 e MAC0325) no IME–USP (Instituto de Matemática e Estatística da Universidade de São Paulo). As notas foram baseadas em dois livros:

- Ahuja–Magnanti–Orlin [AMO93], que chamaremos de "AMO", e
- Cormen–Leiserson–Rivest–Stein [CLRS01], que chamaremos de "CLRS".

Também serviu de referência o excelente livro de Cook, Cunningham, Pulleyblank e Schrijver [CCPS98], que chamaremos de "CCPS".

Ocasionalmente citamos também os números de capítulos do "CLR" [CLR91], que é a primeira edição do CLRS.

Ao contrário do que fazem os livros citados, estas notas identificam as *invariantes* dos algoritmos, permitindo assim uma análise mais rigorosa de sua correção.

Veja o sítio www.ime.usp.br/~pf/mac5781–2002 da edição 2002 da disciplina de Otimização Combinatória.

Sumário

Capítulo 1

Programação linear: resumo

O assunto central de MAC5781 e MAC0325 é o problema do fluxo de custo mínimo em redes. Esse problema é um caso particular do problema de programação linear. Portanto, convém fazer um rápido resumo do assunto, principalmente para introduzir o conceito de dualidade, que é o pano de fundo de todos os capítulos subseqüentes.

1.1 O problema

Eis um problema típico de programação linear: encontrar números x_1, x_2, x_3, x_4, x_5 que minimizem

$$51x_1 + 52x_2 + 53x_3 + 54x_4 + 55x_5$$

enquanto satisfazem as restrições

$$\begin{aligned} 11x_1 + 12x_2 + 13x_3 + 14x_4 + 15x_5 &= 16 \\ 21x_1 + 22x_2 + 23x_3 + 24x_4 + 25x_5 &= 26 \\ 31x_1 + 32x_2 + 33x_3 + 34x_4 + 35x_5 &= 36 \\ 41x_1 + 42x_2 + 43x_3 + 44x_4 + 45x_5 &= 46 \end{aligned}$$

e $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$, $x_4 \geq 0$, $x_5 \geq 0$.

(É claro que o número de linhas poderia ser diferente de 4 e o número de colunas diferente de 5. É claro também que os valores dos coeficientes nada têm de especial: se substituimos cada um por um número racional qualquer, positivo ou negativo, ainda teremos um problema de programação linear.)

Em notação matricial, o problema pode ser escrito assim: dada qualquer matriz M e

quaisquer vetores b e c, encontrar um vetor x que

$$\begin{array}{c} \text{minimize } \ cx \\ \text{sob as restrições} \\ Mx = b \ \text{ e } \ x \geq 0. \end{array} \tag{1.1}$$

Infelizmente, a terminologia tradicional abusa da palavra "solução" e diz que uma **solução viável** é qualquer vetor x que satisfaz as restrições $Mx = b$ e $x \geq 0$. Uma solução viável x é **ótima** se x minimiza cx.

O problema é **viável** se admite ums solução viável. O problema é **ilimitado** se for viável mas cx não tiver mínimo.

1.2 Dualidade

O **dual** do problema acima consiste em encontrar um vetor y que

$$\begin{array}{c} \text{maximize } \ yb \\ \text{sob as restrições} \\ yM \leq c. \end{array} \tag{1.2}$$

Os conceitos de solução viável, problema viável e problema ilimitado são definidos da maneira óbvia.

A relação básica entre os problemas (1.1) e (1.2) é a desigualdade conhecida como "teorema fraco da dualidade": para qualquer x que satisfaz as restrições do primeiro problema e qualquer y que satisfaz as restrições do segundo,

$$yb \leq cx\ . \tag{1.3}$$

A prova dessa desigualdade é muito simples e muito instrutiva:

$$yb \ = \ y(Mx) \ = \ (yM)x \ \leq \ cx\ .$$

Segue daí imediatamente que se $cx = yb$ então x é solução ótima do problema (1.1) e y é solução ótima do problema (1.2). (Em particular, para mostrar que uma solução viável x é ótima, basta exibir uma solução viável y tal que $cx = yb$.) A recíproca dessa observação é garantida pelo "teorema forte" da programação linear:

Teorema 1.1 *Se os problemas* (1.1) *e* (1.2) *são viáveis então existe uma solução viável* x *de* (1.1) *e uma solução viável* y *de* (1.2) *tais que* $cx = yb$.

A prova do teorema é algorítmica: o algoritmo Simplex recebe M, b, c e decide se os dois problemas são viáveis e, em caso afirmativo, devolve x e y.

1.3 Um caso particular importante

Suponha que cada coluna da matriz M tem um componente -1, um componente $+1$, sendo todos os demais nulos. Então a matriz pode ser representada por um grafo: cada linha da matriz é um nó e cada coluna é um arco; se uma coluna tem "$+1$" na linha u e "-1" na linha v então o arco correspondente é uv, ou seja, vai de u para v.

$$\begin{array}{rrrrr} -1 & 0 & +1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ +1 & +1 & 0 & 0 & +1 \\ 0 & 0 & -1 & +1 & -1 \end{array}$$

Cada componente do vetor b fica associada a um nó do grafo. Cada componente de c e cada componente de x fica associada a um arco do grafo.

As restrições $Mx = b$ podem ser formuladas assim: para cada nó v, a soma dos números da forma x_{uv} menos a soma dos números da forma x_{vw} deve ser igual a b_v, ou seja,

$$\sum_{u\,:\,uv\in A} x_{uv} \;-\; \sum_{w\,:\,vw\in A} x_{vw} = b_v \;,$$

onde A é o conjunto de arcos. A expressão cx pode ser escrita como

$$\sum_{uv\in A} c_{uv} x_{uv} \;,$$

onde a soma se estende a todos os arcos do grafo.

Este é o **problema do fluxo viável de custo mínimo** que estudaremos no que segue. O problema é um tanto complexo; por isso começaremos por estudar vários casos do problema.

Exercícios

1.1 O enunciado do teorema 1.1 supõe tacitamente que que os problemas primal e dual são viáveis. Complete o enunciado de modo a cuidar dos problemas inviáveis e dos ilimitados.

Capítulo 2

Conceitos básicos de redes

Este capítulo[1] introduz os conceitos de grafo, rede, caminho e ciclo e establece algumas convenções de notação.

2.1 Grafos

Um **grafo** (= grafo dirigido = grafo orientado) é um par (N, A), onde N é um conjunto finito e A é um conjunto de pares ordenados de elementos de N. Os elementos de N são chamados **nós**. Os elementos de A são chamados **arcos**. Para cada arco (i, j), o nó i é a **ponta negativa**[2] ou **ponta inicial** (= *tail*) de (i, j) e j é a **ponta positiva** ou **ponta final** (= *head*) de (i, j). As pontas inicial e final de cada arco são *distintas*; portanto, nossos grafos não têm "laços" (= *loops*).

Um arco (i, j) também pode ser denotado por ij. Diremos que um tal arco **sai** de i e **entra** em j. O **grau de entrada** de um nó i é o número de arcos que entram em i; o **grau de saída** de i é o número de arcos que saem de i.

De acordo com nossa definição, grafos não têm "arcos paralelos": dois arcos diferentes não podem ter a mesma ponta final e a mesma ponta inicial.[3] Portanto o grau de saída de um nó é no máximo $|N| - 1$. Também o grau de entrada não passa de $|N| - 1$.

Suponha que nosso grafo contém os arcos (i, j) e (j, i); dizemos que esses arcos são **anti-paralelos** ou mutuamente **inversos**. Um grafo é **simétrico** se a presença de um arco implica na presença do arco inverso. Um grafo é **simples** se não tem arcos anti-paralelos. Em outras palavras, um grafo é simples se a presença do arco (i, j) implica na ausência

[1] Trata-se de um resumo da seção 2.1, p.23, de AMO.

[2] Nossa convenção é contrária à de AMO: para eles a ponta inicial é *positiva* e a final é *negativa*.

[3] Em algumas ocasiões, será necessário abusar da definição e permitir a presença de duas ou mais "cópias" de um mesmo arco (i, j). Se tomarmos cuidado, poderemos lidar com isso sem criar confusão.

do arco (j, i).[4]

O número de nós de um grafo (N, A) será denotado por n e o número de arcos por m:

$$n := |N| \qquad \text{e} \qquad m := |A| \,.$$

Como nosssa definição não permite arcos "paralelos" nem "laços", temos

$$m \leq n(n-1) < n^2 \,.$$

Um grafo é **esparso** se $m = \mathrm{O}(n)$ e **denso** em caso contrário (em particular, no caso $m = \Omega(n^2)$).[5]

O conjunto de todos os arcos que saem de um dado nó i será denotado por $A(i)$

$$A(i) \,.$$

Esse conjunto é a **lista de incidência** do nó i. Às vezes trataremos $A(i)$ como uma seqüência e não apenas como um conjunto. Na terminologia da informática, diríamos que $A(i)$ é uma lista encadeada (= *linked list*).

Em algumas raras ocasiões, será necessário dispor do conjunto de arcos que *entram* em um nó j. Denotaremos esse conjunto por $\widetilde{A}(j)$.

É evidente que $|A(i)|$ é o grau de saída de i. É evidente também que

$$\textstyle\sum_{i \in N} |A(i)| = |A| \,.$$

Um grafo (N', A') é **sub-grafo** de um grafo (N, A) se $N' \subseteq N$ e $A' \subseteq A$.

2.2 Matriz de incidências e matriz de adjacências

A **matriz de incidência** de um grafo (N, A) é definida assim: as linhas são indexadas por N e as colunas por A; para cada ij em A, a coluna ij tem um -1 na linha i e um $+1$ na linha j, sendo o resto da coluna igual a 0.w

A **matriz de adjacências** de um grafo (N, A) é definida assim: as linhas e as colunas são indexadas por N; cada componente (i, j) da matriz vale 1 se ij é um arco e vale 0 em caso contrário.

[4] Veja "Working with residual networks", AMO, p.45.

[5] Dizemos que uma função $T(n)$ é $\mathrm{O}(f(n))$ se existe uma constante k e um número n_0 tais que $0 \leq T(n) \leq k\, f(n)$ para todo $n \geq n_0$. Dizemos que $T(n)$ é $\Omega(f(n))$ se existe uma constante k e um número n_0 tais que $0 \leq k\, f(n) \leq T(n)$ para todo $n \geq n_0$. Dizemos que $T(n)$ é $\Theta(f(n))$ se $T(n)$ é $\mathrm{O}(f(n))$ e $\Omega(f(n))$.

2.3 Passeios, caminhos, ciclos

Um **passeio** (= *walk*)[6] num grafo (N, A) é qualquer seqüência $\langle v_0, v_1, \ldots, v_p \rangle$ de nós tal que

$$(v_{k-1}, v_k) \in A$$

para $k = 1, \ldots, p$. Um passeio é um objeto *dirigido* (ou *orientado*): todos os seus arcos apontam do nó anterior para o sequinte. O nó v_0 é a **origem** do passeio e v_p é o **término** do passeio. Dizemos também que o passeio **vai de** v_0 **a** v_p.

Dizemos que um nó t do grafo **está ao alcance de** (= *is reachable from*) um nó s, ou que t **pode ser alcançado** a partir de s, se existe um passeio de s a t.

O **comprimento** de um passeio $\langle v_0, v_1, \ldots, v_p \rangle$ é p. O comprimento de um passeio P será denotado por $|P|$. Um passeio P é **degenerado** se $|P| = 0$.

Um **segmento** de um passeio $\langle v_0, v_1, \ldots, v_p \rangle$ é qualquer passeio $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, com $0 \leq i \leq j \leq p$. Um tal segmento é **inicial** se $i = 0$ e **final** de $j = p$.

Se o término de um passeio $(v_0, \ldots, v_p)$ é igual à origem de um passeio $(w_0, \ldots, w_q)$ (ou seja, se $v_p = w_0$) então a **concatenação** dos dois passeios é o passeio $(v_0, \ldots, v_p, w_1, \ldots, w_q)$. Se um passeio P termina na origem de um passeio Q, a concatenação de P com Q será denotada por $P{\cdot}Q$.

Um **caminho** (= *path* = *simple path*)[7] é um passeio sem nós repetidos (é claro os arcos de um caminho também são distintos dois a dois). Se P é um passeio com origem s e término t então alguma subseqüência de P é um *caminho* de s a t. (Por exemplo, se $\langle a, b, c, d, b, c, e \rangle$ é um passeio então $\langle a, b, c, e \rangle$ é o correspondente caminho.)

Um quase-caminho[8] é um passeio em que somente o primeiro nó é repetido.[9] Assim, um quase-caminho é algo como $\langle d, a, b, c, d, e, f \rangle$, e portanto tem a aparência de um "9" ou um "ρ". Mais formalmente: um **quase-caminho** é um passeio $\langle v_0, v_1, \ldots, v_p \rangle$ em que $v_1, \ldots, v_p$ são distintos dois a dois mas v_0 coincide com um dos outros nós. É claro que os arcos de todo quase-caminho são distintos dois a dois. É claro também que todo quase-caminho tem comprimento maior que 1. quase-caminho

Um **ciclo** (= *cycle*)[10] é um quase-caminho $\langle v_0, v_1, \ldots, v_p \rangle$ em que $v_0 = v_p$.

2.4 Função-predecessor

Suponha que π é uma função parcial de N em N.[11] Se π não está definido em j, diremos π

[6] AMO diz *walk*, mas CLRS diz *path*. Já CCPS, diz *dipath*.
[7] AMO diz *path*, enquanto CLRS diz *simple path* e CCPS diz *simple dipath*.
[8] Nem AMO nem CLRS têm esse conceito. Mas eu acho que ele é muito útil.
[9] Cuidado: eu disse "primeiro" e não "último".
[10] Esta é o termo usado por AMO. CLRS diz *simple cycle*.

que $\pi(j) = \text{NIL}$.[12] Para qualquer j em N, diremos que

$$j\ ,\ \pi(j)\ ,\ \pi(\pi(j))\ ,\ \ldots$$

é a seqüência determinada por π a partir de j. É óbvio que essa seqüência pode ser finita ou infinita. Se for finita, então π não está definida no último termo e a seqüência não tem termos repetidos. Se for infinita, a seqüência é cíclica a partir de um certo ponto (por exemplo, $j, i, h, g, f, h, g, f, h, g, f, \ldots$).

Em um grafo (N, A), uma **função-predecessor**[13] é uma função parcial π de N em N tal que,[14] para todo j em N, pred

$$\pi(j) = \text{NIL} \quad \text{ou} \quad (\pi(j), j) \in A\,.$$

Em particular, $\pi(j) \neq j$ para todo j.

2.5 Grafo de predecessores

Suponha que π é uma função-predecessor. Denotaremos por A_π o conjunto de todos os arcos da forma $(\pi(j), j)$ e diremos que (N, A_π) é o **grafo de predecessores**. É claro que dados dois nós s e t, existe no máximo um caminho de s a t no grafo de predecessores. A_π

O grau de entrada de cada nó do grafo de predecessores é no máximo 1 (o grau de entrada de um nó j é 0 se e só se $\pi(j) = \text{NIL}$). Reciprocamente, para qualquer parte A' de A, se o grau de entrada de todo nó do grafo (N, A') for ≤ 1 então (N, A') é o grafo de predecessores definido por alguma função-predecessor.

Suponha que a seqüência determinada por uma função-predecessor π a partir de um nó j é finita; por exemplo, j, i, h, g, f. Então

$$\langle f, g, h, i, j\rangle$$

é um caminho no grafo de predecessores. Diremos que esse é o caminho **determinado por π a partir de j**.

Suponha agora que a seqüência determinada por π a partir de j é infinita; por exemplo, $j, i, h, g, f, e, g, f, e, g, f, e, \ldots$ Diremos então que o quase-caminho

$$\langle g, e, f, g, h, i, j\rangle$$

é **determinado por π a partir de j**.

[11] Uma **função parcial** de um conjunto X em um conjunto Y é uma função definida sobre um *subconjunto* de X e com valores em Y.

[12] AMO escreve "0" no lugar do meu "NIL".

[13] Predecessores são discutidos nas páginas 137, 139 e 148 do AMO. Também na subseção "Representing shortest paths", p.584, de CLRS.

[14] AMO escreve "pred" no lugar do meu "π".

Eis um algoritmo trivial que calcula o caminho ou o quase-caminho determinado por π a partir de um nó j:

```
0  P ← ⟨j⟩
1  J ← {j}
2  enquanto π(j) ≠ NIL e π(j) ∉ J
3       faça j ← π(j)
4            acrescente j ao início de P
5            J ← J ∪ {j}
6  se π(j) = NIL
7       então P é um caminho
8       senão acrescente π(j) ao início de P
9            P é um quase-caminho
```

Veja exercício 2.1.

2.6 Cortes

Para quaisquer partes S e T de N, vamos denotar por $\nabla(S,T)$ o conjunto de todos os arcos que têm ponta inicial em S e ponta final em T. Ocasionalmente, podemos escrever[15] apenas (S,T) no lugar de $\nabla(S,T)$. Por exemplo, $\nabla(N-T,T)$ é o conjunto de todos os arcos que entram em T. Analogamente, $\nabla(S,N-S)$ é o conjuntos dos arcos que saem de S.

Para qualquer parte T de N, o **corte determinado por** T é o conjunto $\nabla(N-T,T)$.

Para qualquer nó s em $N-T$ e qualquer nó t em T, diremos que o conjunto T **separa** s **de** t; diremos também que o corte $\nabla(N-T,T)$ **separa** s **de** t. Podemos dizer, ainda, que $\nabla(N-T,T)$ é um (s,t)-**corte**.

Se s é um nó, usaremos a abreviatura $\nabla(s,N-s)$ para $\nabla(\{s\},N-\{s\})$. É claro que

$$\nabla(s,N-s) = A(s)\,.$$

2.7 Redes

Nossa definição de rede (= *network*) será um tanto vaga e informal: uma **rede** é um grafo (N,A) juntamente com uma ou mais funções que atribuem números aos arcos e/ou aos nós.

A propósito, o conjunto dos números racionais será denotado por $\mathbb{Q}$, o conjunto dos racionais não-negativos será denotado por $\mathbb{Q}_{\geq}$, o conjunto dos inteiros por $\mathbb{Z}$ e o conjunto

[15] Como faz AMO.

dos inteiros não-negativos por $\mathbb{Z}_{\geq}$. Assim,

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \quad \text{e} \quad \mathbb{Z}_{\geq} = \{0, 1, 2, \ldots\}\,.$$

Se f é uma função que leva A em $\mathbb{Z}$, diremos que (N, A, f) é uma rede. O valor de f num arco (i, j) será usualmente denotado por

$$f_{ij}\,.$$

Dependendo do contexto, f poderá ser chamada **função-custo** ou **pseudofluxo**. Se $f \geq 0$, poderemos dizer (dependendo do contexo) que f é uma **função-capacidade**, um **fluxo** ou um **pré-fluxo**.

Outro exemplo: Se g é uma função que leva N em $\mathbb{Z}$, diremos que (N, A, g) é uma rede. O valor de g num nó i será usualmente denotado por

$$g(i)\,.$$

Dependendo do contexto, g poderá ser chamada **demanda** ou **potencial**.

Para qualquer função g de N em $\mathbb{Z}$ e qualquer parte S de N, vamos denotar por $g(S)$ a soma de todos os $g(i)$ para i em S:

$$g(S) := \sum_{i \in S} g(i)\,.$$

Notação análoga vale para qualquer função f de A em $\mathbb{Z}$ e qualquer parte B de A:

$$f(B) := \sum_{ij \in B} f_{ij}\,.$$

Em particular, se S é uma parte de N então

$$f(S, N - S) := \sum_{ij \in (S, N-S)} f_{ij}\,.$$

Se c e x são funções de A em $\mathbb{Z}$, vamos denotar por cx a soma de todos os produtos $c_{ij}x_{ij}$:

$$cx := \sum_{ij \in A} c_{ij}x_{ij}\,.$$

2.8 Complexidade de algoritmos

Nossa medida do consumo de tempo de um algoritmo será assintótica. Suponha que um algoritmo que opera sobre uma rede (N, A, c) com c definida sobre A. Diremos

que o algoritmo consome $O(f(n, m, C))$ unidades de tempo, onde f é alguma função de $n := |N|$, $m := |A|$ e $C := \max_{ij \in A} |c_{ij}|$.

Vamos adotar um modelo de computação em que o consumo de tempo de cada operação aritmética sobre dois números inteiros não depende do tamanho dos operandos: cada operação aritmética consome $O(1)$ unidades de tempo.[16]

Suponha que um algoritmo $\mathcal{A}$ opera sobre uma rede (N, A, c), onde c é uma função de A em $\mathbb{Z}$. Diremos $\mathcal{A}$ é **fortemente polinomial** (= *strongly polynomial*) se seu consumo de tempo for $O(n^p m^q)$ para algum p em $\mathbb{Z}_{\geq}$ e algum q em $\mathbb{Z}_{\geq}$.

Diremos $\mathcal{A}$ é **fracamente polinomial** (= *weakly polinomial*) se seu consumo de tempo for $O(n^p m^q \log C)$, onde $C := \max_{ij \in A} |c_{ij}|$. (Em geral, o *número de iterações* de tal algoritmo é proporcional a $\log C$.)

Diremos $\mathcal{A}$ é **pseudo-polinomial** se seu consumo de tempo for $O(n^p m^q C)$. (Em geral, o *número de iterações* do algoritmo é proporcional a $\log C$.)

Exercícios

2.1 Suponha que o grafo de predecessores não tem ciclos. Seja ij um arco qualquer do grafo (não necessariamente do grafo de predecessores). Escreva um algoritmo para decidir se a atribuição $\pi(j) \leftarrow i$ vai criar um ciclo orientado no grafo de predecessores.

[16] Num modelo mais realista, uma operação como "$a + b$" consome $O(\log a + \log b)$ unidades de tempo.

Parte I

Caminhos e ciclos

Capítulo 3

Algoritmos de busca

O presente capítulo[1] trata do problema de decidir se um dado nó t de um grafo está ao alcance de um nó s.

Problema 3.1 (de busca) *Dados nós s e t de um grafo, encontrar um caminho[2] de s a t.*

Uma variante do problema: encontrar o conjunto de todos os nós que estão ao alcance de s, isto é, todos os nós que são término de um caminho que tem origem s.

3.1 Condições de inexistência

O problema de busca nem sempre tem solução (isto é, nem sempre é viável). Como é possível demonstrar que uma dada instância do problema não tem solução?

Dizemos que um conjunto T de nós **separa** s **de** t se $s \in N - T$ e $t \in T$; podemos dizer também que $\nabla(N - T, T)$ é um (s,t)**-corte**. Se T separa s de t e

$$(N - T, T) = \emptyset$$

(ou seja, não existe arco ij com $i \in N - T$ e $j \in T$), então é evidente que não existe caminho de s a t. Como veremos adiante, a recíproca é verdadeira: se não existe caminho de s a t então algum corte vazio separa s de t.

Convém repetir todas essas considerações de uma maneira mais sofisticada. Um 0-**potencial** é qualquer função y de N em $\{0, 1\}$ tal que

$$y(j) - y(i) \leq 0 \text{ para todo arco } ij$$

[1] O capítulo é um resumo da seção 3.4, p.73, de AMO. Veja também o capítulo 22 (*Elementary Graph Algorithms*) do CLRS ou capítulo 23 do CLR.

[2] Convém lembrar que nossos caminhos são dirigidos.

(ou seja, não existe arco ij com $y(i) = 0$ e $y(j) = 1$). Qualquer função constante é um 0-potencial; mas não é um 0-potencial muito interessante.

Qualquer 0-potencial y define um corte: se T é o conjunto dos nós j tais que $y(j) = 1$ então $\nabla(N - T, T)$ é o correspondente corte.

Eis uma propriedade básica de qualquer 0-potencial y: se existe um passeio de s a t então

$$y(t) - y(s) \leq 0\,. \tag{3.1}$$

Esboço da prova: Se $P = \langle s, i, j, t\rangle$ então $y(t)-y(s) = y(t)-y(j)+y(j)-y(i)+y(i)-y(s) \leq 0+0+0 = 0$.

Portanto, para mostrar que não existe caminho de s a t basta exibir um 0-potencial y tal que $y(t) - y(s) > 0$.

3.2 Algoritmo genérico

Eis um algoritmo "genérico" para o problema de busca.[3] Ele recebe nós s e t de um grafo (N, A) e devolve um caminho de s a t ou um 0-potencial y tal que $y(t) - y(s) > 0$.

```
BUSCA-GENÉRICO (N, A, s, t)
0  para cada i em N faça
1      y(i) ← 1
2      π(i) ← NIL
3  y(s) ← 0
4  enquanto y(j) > y(i) para algum ij em A faça
5      y(j) ← y(i)
6      π(j) ← i
7  se y(t) = 1
8      então devolva y
9      senão devolva o caminho de s a t no grafo (N, A_π)
```

Para mostrar que o algoritmo faz o que promete fazer é preciso entender a relação entre as variáveis no início de cada iteração. Digamos que cada iteração começa na linha 4, imediatamente antes da verificação da condição "$y(j) > y(i)$ para algum ij em A". No começo de cada iteração, considere o grafo de predecessores (N, A_π) (veja seção 2.5) e observe as seguintes invariantes:[4]

[3] Ele é "genérico" porque é "nú": não tem as estruturas de dados necessárias para uma implementação eficiente.

[4] O que são as invariantes? Eis uma explicação: o algoritmo produzirá resultados corretos se for executado a partir de qualquer configuração que satisfaça as invariantes. Em outras palavras, qualquer configuração que satisfaça as invariantes pode ser usada como "inicialização" do algoritmo.

(i0) para cada arco pq no grafo de predecessores tem-se $y(p) = y(q) = 0$;

(i1) $\pi(s) = \text{NIL}$ e $y(s) = 0$;

(i2) para cada v distinto de s, $\pi(v) \neq \text{NIL}$ se e só se $y(v) = 0$;

(i3) para cada nó v, se $\pi(v) \neq \text{NIL}$ então existe um caminho de s a v no grafo de predecessores.

Eis um esboço da prova das invariantes. É claro que eles valem no início da primeira iteração. Suponha agora que estamos no início de uma iteração em que as invariantes valem; seja ij um arco tal que $y(j) > y(i)$. Vamos mostrar que as invariantes valem no início da iteração seguinte.

Prova de (i0): Durante a iteração, somente o arco ij é acrescentado ao grafo de predecessores e é evidente que $y(j) = y(i) = 0$ no fim da iteração. Como j é o único nó que tem seu potencial alterado, basta verificar que, no início da iteração, nenhum arco no grafo de predecessores tem ponta inicial ou ponta final igual a j.

Digamos que pq é um arco do grafo de predecessores. Em virtude (i0), $y(p) = y(q) = 0$. Por outro lado, com $y(j) > y(i)$ temos necessariamente $y(j) = 1$. Logo, j é diferente de p e de q. (i0)

Prova de (i1): Basta observar que $j \neq s$ no início da iteração. Isso é verdade pois $y(j) > y(i)$ e portanto $y(j) = 1$, enquanto $y(s) = 0$ em virtude de (i1), (i1)

Prova de (i2): Os únicos valores de y e π alterados durante a iteração são $y(j)$ e $\pi(j)$. No início da iteração temos $y(i) = 0$ e portanto no fim da iteração teremos $y(j) = 0$ e $\pi(j) \neq \text{NIL}$.

Prova de (i3): Seja v um nó qualquer tal que $\pi(v) \neq \text{NIL}$ no início da iteração. Por (i3), existe um caminho P de s a v no grafo de predecessores. Por (i0), temos $y(k) = 0$ para cada nó k de P. Como $y(j) = 1$, o nó j não está em P e portanto, no fim da iteração, P continua sendo um caminho de s a v no grafo de predecessores. Resta mostrar que existe um caminho de s a j no grafo de predecessores no fim da iteração. Mas isso é fácil: basta tomar $v = i$ no raciocínio acima, e observar que no fim da iteração $P \cdot \langle i, j \rangle$ é um caminho no grafo de predecessores. (i3) (i0)

(A propósito, as invariantes garantem mais uma propriedade: o grafo de predecessores não tem ciclos.)

Suponha agora que estamos no início da última iteração, e portanto

$$y(j) \leq y(i) \text{ para todo arco } ij\,.$$

Então y é um 0-potencial, como o enunciado do algoritmo prometeu. Se $y(t) = 0$ então, em virtude de (i2), $\pi(t) \neq \text{NIL}$. Logo, em virtude de (i3), existe um caminho de s a t no grafo de predecessores e portanto também no grafo (N, A). Por outro lado, se $y(t) = 1$ então, em virtude de (i1), $y(t) - y(s) > 0$. Em suma, o algoritmo realmente faz o que promete. (i2) (i3) (i1)

A análise do algoritmo BUSCA-GENÉRICO prova o seguinte

Fato 3.2 *Para quaisquer nós s e t em um grafo (N, A), existe um caminho de s a t ou existe um 0-potencial y tal que $y(t) - y(s) > 0$. (A segunda alternativa equivale à existência de um conjunto de nós T que separa s de t e tem $(N - T, T) = \emptyset$.)*

Consumo de tempo. Não está claro, à primeira vista, que a execução de BUSCA-GENÉRICO termina depois de um número finito de iterações. Seja T o conjunto $\{v \in N : y(v) = 1\}$ e observe que $|T|$ diminui a cada iteração. Portanto, o número de iterações (ou seja, o número de execuções do bloco de linhas 4–6) não passa de

$$n - 1 \,,$$

onde $n := |N|$. $n = |N|$

Cada execução da linha 4 consome $O(m)$ unidades de tempo, onde $m := |A|$, e as linhas 5 e 6 consomem $O(1)$ unidades. Cada execução das linhas 0–3 consome $O(n)$ unidades de tempo. Cada execução das linhas 7–9 consome $O(n)$ unidades. Concluímos que o consumo de tempo do algoritmo é $m = |A|$

$$O(nm) \,.$$

Mais grosseiramente, podemos dizer que o consumo de tempo total é $O(n^3)$, uma vez que $m < n^2$.

3.3 Algoritmo de busca

A implementação óbvia da linha 4 do algoritmo BUSCA-GENÉRICO não é muito eficiente. O algoritmo BUSCA abaixo[5] procura remediar a situação. A idéia é manter um conjunto L que contém a ponta inicial de todo arco ij que viola a condição $y(j) - y(i) \leq 0$.

Tal como BUSCA-GENÉRICO, o algoritmo BUSCA recebe nós s e t de um grafo (N, A) e devolve um caminho de s a t ou um 0-potencial y tal que $y(t) - y(s) > 0$.

```
BUSCA (N, A, s, t)
01  para cada i em N faça
02      A'(i) ← A(i)
03      y(i) ← 1
04      π(i) ← NIL
05  y(s) ← 0
06  L ← {s}
07  enquanto L ≠ ∅ faça
```

[5] Esse é, essencialmente, o algoritmo *Search* descrito na figura 3.4, p.74, seção 3.4, do AMO.

```
08      escolha um nó i em L
09      se A'(i) ≠ ∅
10          então retire[6] um arco ij de A'(i)
11                  se y(j) = 1
12                      então y(j) ← 0
13                            π(j) ← i
14                            L ← L ∪ {j}
15          senão L ← L − {i}
16  se y(t) = 1
17      então devolva y
18      senão devolva o caminho de s a t em (N, A_π)
```

Na prática, da lista de adjacência $A(i)$ é implementado como uma seqüência (mais precisamente, como uma lista encadeada) e não como um conjunto. Portanto, não é necessário fazer uma cópia $A'(i)$ de $A(i)$, como fizemos na linha 02: basta manter um ponteiro para o elemento corrente de $A(i)$. O ponteiro começa apontando o primeiro elemento de $A(i)$; na linha 10, o ponteiro é reajustado para o próximo elemento de $A(i)$. Essa estrutura é conhecida como *current arc data structure*[7].

O algoritmo está correto? Observe que no início de cada iteração (ou seja, na linha 07, imediatamente antes da comparação de L com $\emptyset$), além das invariantes (i0) a (i3) de BUSCA-GENÉRICO, valem também os seguintes:

(i4) para cada arco pq, se $y(p) = 0$ e $y(q) = 1$ então $p \in L$;[8]
(i5) $y(p) = 0$ para cada p em L;
(i6) para cada nó p e cada arco pq em $A(p) - A'(p)$, se $y(p) = 0$ então $y(q) = 0$.

Prove essas invariantes! (Veja como a validade de (i6) no início de uma iteração depende da validade de (i4).)

Suponha agora que estamos no início da última iteração, quando $L = \emptyset$. Então, em virtude de (i4), temos $y(q) - y(p) \leq 0$ para todo arco pq; portanto, y é um 0-potencial. Se $y(t) = 1$ então, em virtude de (i1), $y(t) - y(s) = 1 > 0$. Senão, de acordo com (i3), há um caminho de s a t no grafo de predecessores. Em suma, o algoritmo faz o que promete. (i4) (i1) (i3)

Consumo de tempo. O bloco de linhas 01–06 consome $O(n)$ unidades de tempo. O bloco de linhas 16–16 também consome $O(n)$ unidades de tempo. Resta examinar o processo iterativo definido pelas linhas 07-15.

[6] Está implícita aí a operação $A'(i) \leftarrow A'(i) - \{ij\}$.
[7] Veja p.75 do AMO. Veja também a figura 7.7, p.217, de AMO.
[8] Mas pode existir arco pq com p em L e $y(q) = 0$.

Quantas iterações são executadas? Há dois tipos de iteração: o primeiro passa pelo bloco de linhas 10–14 e o segundo pela linha 15. O número de iterações do segundo tipo é no máximo n pois cada nó do grafo pode ser retirado de L no máximo uma vez (pois não pode mais voltar a L, de acordo com o invariante (i5)). O número de iterações do primeiro tipo não passa de (i5)

$$\sum_{k \in N} |A(k)| \,,$$

pois a cada iteração $A'(i)$ diminui e $A'(h)$ não se altera quando $h \neq i$. Como essa soma é igual a m, podemos dizer que o número total de iterações dos dois tipos é no máximo $m := |A|$

$$n + m \,.$$

Cada iteração consome $\mathrm{O}(1)$ undidades de tempo[9]. Logo, o consumo total do bloco de linhas 07-15 é

$$\mathrm{O}(n + m) \,.$$

Nossa conclusão final: o algoritmo consome $\mathrm{O}(n) + \mathrm{O}(n + m) + \mathrm{O}(n)$, ou seja,

$$\mathrm{O}(n + m)$$

unidades de tempo. Podemos dizer, mais grosseiramente, que o consumo de tempo total é $\mathrm{O}(n^2)$, uma vez que $m < n^2$.

Não é difícil verificar que o consumo de tempo do algoritmo também é $\Omega(n + m)$.

3.4 Busca em largura e busca em profundidade

O conjunto L no algoritmo BUSCA é usualmente implementado como uma seqüência. Se a seqüência for manipulada como um fila, ou seja, se elementos forem retirados (linha 15) do início de L e novos elementos forem acrescentados (linha 14) ao final de L, teremos um algoritmo de **busca em largura** (= *breadth-first search* = *BFS*).

Se L for manipulada como um pilha, ou seja, se elementos forem retirados do início de L e novos elementos também forem acrescentados ao início de L, teremos um algoritmo de **busca em profundidade** (= *depth-first search* = *DFS*).

3.5 Versão "capacitada" do problema

Nas aplicações, o problema de busca freqüêntemente ocorre no seguinte contexto. Suponha que uma função u associa um número u_{ij} com cada arco ij de nosso grafo (N, A).

[9] Ou seja, o consumo de tempo é limitado por uma quantidade que não depende de n nem de m.

Digamos que um arco ij é **positivo** se $u_{ij} > 0$ e seja A_u é o conjunto dos arcos positivos. Dados nós s e t, queremos encontrar um caminho de s a t no grafo (N, A_u).

Como mostrar que um tal caminho não existe? Basta usar um 0-potencial no grafo (N, A_u), ou seja, uma função y de N em $\{0, 1\}$ tal que

$$y(j) - y(i) \leq 0 \text{ para todo } ij \text{ tal que } u_{ij} > 0.$$

É fácil verificar que, para um tal y, se P é um caminho de s a t em (N, A_u) então $y(t) - y(s) \leq 0$. Logo, se $y(t) - y(s) > 0$ então o problema não tem solução.

É fácil adaptar o algoritmo BUSCA de modo que ele receba a rede (N, A, u) e devolva (1) um caminho de s a t em (N, A_u) ou (2) um 0-potencial y em (N, A_u) tal que $y(t) - y(s) > 0$.

3.6 Busca "inversa"

Eis uma variante importante do problema de busca: Dado um nó t de um grafo (N, A), encontrar todos os nós que são origem de um passeio que termina em t.[10]

Esse problema poderia ser reduzido ao problema usual de busca mediante inversão de todos os arcos (ou seja, troca de cada arco ij por um arco ji).

Outra possibilidade é introduzir um nova estrutura na descrição do grafo: para cada nó j, seja $\widetilde{A}(j)$ o conjunto de todos os arcos que entram em j.

Qual a definição apropriada de potencial para essa variante do problema? Escreva e analise um algoritmo que resolva o problema.

Exercícios

3.1 Deduza das invariantes (i0) a (i3) que o grafo de predecessores gerado pelo algoritmo BUSCA-GENÉRICO e pelo algoritmo BUSCA não tem ciclos.

3.2 Escreva um algoritmo de busca em largura. Procure simplificar o algoritmo.

3.3 Escreva um algoritmo de busca em profundidade. Procure simplificar o algoritmo.

3.4 [AMO 3.24, p.89] Desenhe as árvores de busca em largura e busca em profundidade do grafo na figura 3.12, p.89, de AMO.

[10] Veja p.76 do AMO.

3.5 Programe os algoritmos de busca usando a estrutura de dados do *Stanford GraphBase*. Faça testes com grafos gerados pelo *Stanford GraphBase*. Inclua no seu módulo uma função que verifique se y é de fato um 0-potencial e se o caminho é de fato um caminho de s a t; é claro que esse função só será usada durante os testes do programa. [Uma das regras do eXtreme Programming: escreva as rotinas de teste *antes* do programa principal!]

3.6 Escreva por extenso o algoritmo de busca "capacitada" descrito na seção 3.5.

3.7 Escreva por extenso o algoritmo de busca "inversa" descrito na seção 3.6.

3.8 [MODELAGEM] Suponha que (N, A) é um grafo e sejam u e u' duas funções de A em $\mathbb{Z}_{\geq}$. Um **pseudo-caminho** é uma seqüência $\langle i_0, i_1, \ldots, i_q \rangle$ de nós tal que, para cada k, tem-se $(i_{k-1}, i_k) \in A$ ou $(i_k, i_{k-1}) \in A$. Os arcos do primeiro tipo são **diretos** e os do segundo são **inversos**. Problema: dados nós s e t, encontrar um pseudo-caminho de s a t tal que $u_{ij} > 0$ para todo arco direto ij e $u'_{ji} > 0$ para todo arco inverso ji. Qual a definição apropriada de função-potencial nesse caso (em termos de u e u')? Formule as condições de existência de solução do problema. Mostre como o problema pode ser transformado em um problema de busca usual (ou seja, um que envolva caminho e não pseudo-caminho). Para resolver esta parte, você pode supor que o grafo é anti-simétrico, ou seja, que se ij é arco então ji não é arco.

Capítulo 4

Ciclos e ordem topológica

O presente capítulo[1] trata do seguinte

Problema 4.1 (do ciclo) *: Encontrar um ciclo num grafo dado.*

Um grafo é **acíclico** (= *DAG* = *directed acyclic graph*) se não tem ciclo. É evidente que o problema não tem solução se e só se o grafo é acíclico.

4.1 Condições de inexistência

Como provar que uma dada instância do problema não tem solução? Em outras palavras, como mostrar que um dado grafo é acíclico?

Um **potencial** em um grafo (N, A) é qualquer função de N em $\mathbb{Z}$. Um -1-**potencial** é um potencial y tal que[2]

$$y(j) - y(i) \leq -1 \text{ para cada } ij \text{ em } A. \tag{4.1}$$

Em outras palavras, $y(j) < y(i)$ para cada arco ij.[3]

Propriedade básica de qualquer -1-potencial: para qualquer passeio P com origem s e término t,

$$y(t) - y(s) \leq -|P| \,.$$

[1] Isso é um resumo da seção 3.4, p.77, de AMO.

[2] Cuidado! Não confunda essa definição com a dos capítulos 5 e 3. Aqui temos "≤ -1" onde (5.1) tinha "≤ 1".

[3] A definição de AMO diz "$\mathrm{order}(i)$" no lugar do nosso "$y(i)$". Além disso, a convenção de AMO é contrária à nossa: $\mathrm{order}(i) < \mathrm{order}(j)$ para cada arco ij.

Esboço da prova: Se $P = \langle s, i, j, t \rangle$ então

$$\begin{aligned} y(t) - y(s) &= y(t) - y(j) + y(j) - y(i) + y(i) - y(s) \\ &\leq -1 - 1 - 1 \\ &= -3 \\ &= -|P| \,. \end{aligned}$$

Se P é um ciclo então $t = s$ e portanto temos $0 \leq -|P|$, ou seja,

$$|P| \leq 0 \,.$$

Mas isso é impossível, pois todo ciclo tem pelo menos 2 arcos. Conclusão: a existência de um -1-potencial é incompatível com a existência de ciclo. Em outras palavras, se há um -1-potencial então não existe ciclo. Portanto, para provar que um dado grafo é acíclico, basta exibir um -1-potencial. Como veremos, isso é sempre possível: todo grafo acíclico admite um -1-potencial.

4.2 Ordem topológica

Há um tipo especial de -1-potencial que vale a pena discutir. Suponha que o conjunto de nós de nosso grafo admite uma enumeração[4] $\langle v_1, v_2, \ldots, v_n \rangle$ tal que

$$p < q \text{ sempre que } v_p v_q \text{ é um arco.}$$

Uma tal enumeração é conhecida como **ordem topológica** (= *topological order*). É fácil ver que qualquer ordem topológica define um -1-potencial: basta fazer

$$y(v_p) := n - p + 1$$

para $p = 1, \ldots, n$, onde $n := |N|$. Reciprocamente, se y é um -1-potencial então qualquer enumeração $\langle v_1, v_2, \ldots, v_n \rangle$ dos nós em ordem não-crescente de valores de y (ou seja, $y(v_1) \geq \cdots \geq y(v_n)$) é uma ordem topológica.

Diante dessa equivalência entre potenciais e ordens topológicas, podemos restringir nossa atenção a potenciais que são uma bijeção de N em $\{1, \ldots, n\}$, onde $n := |N|$.

4.3 Um algoritmo de ordenação topológica

O seguinte algoritmo tem base na seguinte observação: todo grafo acíclico tem pelo menos um nó com grau de entrada nulo. O algoritmo dá uma solução apenas parcial do problema: ele devolve um -1-potencial se o grafo (N, A) for acíclico e não devolve nada se o grafo tem um ciclo.

[4] Isto é, uma seqüência em que cada nó comparece uma e uma só vez.

TOPOLOGICAL-ORDERING[5] (N, A)
01 para cada i em N faça $ge(i) \leftarrow 0$
02 para cada ij em A faça $ge(j) \leftarrow ge(j) + 1$
02 $\triangleright$ $ge(i)$ é o grau de entrada de i
03 *rótulo* $\leftarrow n$
04 $L \leftarrow \emptyset$
05 para cada i em N faça
06 se $ge(i) = 0$ então $L \leftarrow L \cup \{i\}$
07 enquanto $L \neq \emptyset$ faça
08 escolha um nó i em L
09 $L \leftarrow L - \{i\}$
10 $y(i) \leftarrow$ *rótulo*
11 *rótulo* $\leftarrow$ *rótulo* $- 1$
12 para cada ij em $A(i)$ faça
13 $ge(j) \leftarrow ge(j) - 1$
14 se $ge(j) = 0$ então $L \leftarrow L \cup \{j\}$
15 se *rótulo* ≤ 0 então y é um -1-potencial

O algoritmo é prova do seguinte

Fato 4.2 *Um grafo é acíclico se e só se admite um -1-potencial (ou, se preferir, se e só se admite uma ordem topológica).*

O algoritmo consome $O(n+m)$ unidades de tempo, essencialmente porque examina cada arco (linha 12) no máximo uma vez. Se $n = O(m)$, podemos dizer simplesmente que o algoritmo é $O(m)$.

4.4 Um algoritmo melhor

O algoritmo abaixo usa a técnica da busca em profundidade (veja seção 3.4) para resolver o problema. O algoritmo[6] devolve um -1-potencial y se o grafo (N, A) for acíclico e devolve um nó j de um ciclo em caso contrário; no segundo caso, a função predecessor π define um ciclo a partir de j.

DAG (N, A)
01 para cada i em N faça
02 $A'(i) \leftarrow A(i)$
03 $y(i) \leftarrow n + 1$ $\triangleright$ $n + 1$ faz o papel de ∞

[5] Esse é o algoritmo *topological ordering* da figura 3.8, p.79, do AMO.
[6] Veja CLRS seção 22.4, p.550.

```
04      π(i) ← NIL
05  rótulo ← 1
06  enquanto y(s) = n + 1 para algum s em N faça
07      L ← ⟨s⟩
08      enquanto L ≠ ⟨⟩ faça    ▷ L funciona como uma pilha
09          seja i o primeiro elemento de L
10          se A'(i) ≠ ∅
11              então retire[7] um arco ij de A'(i)
12                  se y(j) = n + 1
13                      então π(j) ← i
14                          se j está em L
15                              então devolva j e pare
16                              senão acrescente j ao início de L
17              senão y(i) ← rótulo
18                  rótulo ← rótulo + 1
19                  elimine o primeiro elemento[8] de L
20  devolva y
```

Prove que o algoritmo está correto!

Consumo de tempo. Pode parecer que cada execução da linha 06 do algoritmo consome $O(n)$ unidades de tempo. Mas isso não é assim: todas as execuções da linha 06 somadas consomem $O(n)$ unidades de tempo. De fato, basta percorrer o conjunto de nós uma só vez em uma ordem arbitrária e executar uma de duas ações para cada nó s: se $y(s) = n+1$ então aplique o bloco de linhas 07–19 senão nada faça.

O bloco de linhas 07–19 consome $O(n + m)$ unidades de tempo, essencialmente porque cada arco é examinado no máximo uma vez. Portanto, o consumo de tempo total do algoritmo é

$$O(n + m)\,.$$

Se $n = O(m)$, como é frequentemente o caso, podemos dizer que o algoritmo é $O(m)$.

Não é difícil verificar que o consumo de tempo do algoritmo também é $\Omega(n + m)$.

4.5 Apêndice: algoritmo genérico

O algoritmo DAG é uma implementação concreta do seguinte algoritmo genérico:

[7] Está implícita aí a operação $A'(i) \leftarrow A'(i) - \{ij\}$.
[8] Ou seja, i.

DAG-GENÉRICO (N, A)
1 para cada i em N faça
2 $y(i) \leftarrow n + 1$
3 $\pi(i) \leftarrow$ NIL
4 enquanto $y(j) > y(i) - 1$ para algum ij em A faça
5 $y(j) \leftarrow y(i) - 1$
6 $\pi(j) \leftarrow i$
7 se $y(j) < 1$
8 então devolva j e pare
9 devolva y

Eis as invariantes que explicam o funcionamento do algoritmo. Ao enunciar as invariantes, diremos que um arco vw está **relaxado** ou **folgado** se $y(w) - y(v) \leq -1$, **justo** se $y(w) - y(v) = -1$ e **tenso** se $y(w) - y(v) > -1$. No início de cada iteração (imediatamente antes de verificar a condição "$y(j) > y(i) - 1$ para algum ij em A" na linha 4),

(i1) cada arco no grafo de predecessores (N, A_π) está justo ou tenso (o grafo de predecessores pode ter ciclos);

(i3) para qualquer caminho P no grafo de predecessores, $y(t) \geq n + 1 - |P|$, onde t é o término de P.

Esse algoritmo consome $O(mn^2)$ unidades de tempo (o bloco de linhas 4–8 é repetido $\leq n^2$ vezes). É muito menos eficiente, portanto, que os algoritmos vistos acima.

Voltaremos à análise desse algoritmo genérico, em condições mais gerais, no capítulo 6 e 7.

Exercícios

4.1 Escreva uma variante do algoritmo DAG que use a função-predecessor π no lugar da pilha L.

Capítulo 5

Caminhos de comprimento mínimo

Este capítulo trata de um problema mais geral que o capítulo 3 (Algoritmos de Busca): ele discute o problema do caminho mais curto entre dois nós.[1]

Problema 5.1 (do caminho mais curto) *Dados nós s e t de um grafo (N, A), encontrar um caminho de s a t que tenha comprimento[2] mínimo.*

O problema não tem solução se não existe passeio de s a t. Caso contrário, existe um caminho de s de t. Como o número de tais caminhos é finito, há um (ou mais) caminhos de comprimento mínimo.

5.1 Condições de existência

Como é possível provar que um dado caminho de s a t tem comprimento mínimo? Em outras palavras, como mostrar que nenhum caminho de s a t tem comprimento menor que um dado inteiro λ?

Um 1-**potencial** é uma potencial y (ou seja, uma função de N em $\mathbb{Z}$) tal que

$$y(j) - y(i) \leq 1 \text{ para todo arco } ij\,, \tag{5.1}$$

ou seja, não existe arco ij com $y(j) > y(i) + 1$. (Se y é um 1-potencial, o número $y(i)$ é às vezes chamado **rótulo** (= *label*) do nó i.) É claro que a função nula é um 1-potencial;

[1] Parte desse material está na seção 7.2, p.209, do AMO; outra parte está na seção 3.4, sub-seção "Breadth-First Search", p.76. Veja, em particular, o exercício 3.30, p.90, de AMO. O material também está no capítulo 23 ("Elementary graph algorithms") do CLRS e no capítulo 22 do CLR.

[2] Convém lembrar que o **comprimento** de um passeio $\langle i_0, \ldots, i_p \rangle$ é p e que o comprimento de um passeio P é denotado por $|P|$. Se P é um caminho, então $|P|$ é simplesmente o número de arcos de P.

mas não é um 1-potencial muito interessante. Se y é um 1-potencial e f uma função constante então é evidente que $y - f$ também é um 1-potencial.

Eis uma propriedade básica de qualquer 1-potencial y: para qualquer passeio P com origem s e término t,

$$y(t) - y(s) \leq |P| \,. \tag{5.2}$$

Esboço da prova: Se $P = \langle s, i, j, t\rangle$ então $y(t)-y(s) = y(t)-y(j)+y(j)-y(i)+y(i)-y(s) \leq 1+1+1 = |P|$.

Portanto, para mostrar que não existe caminho de comprimento menor que um determinado número, digamos 99, com origem s e término t basta exibir um 1-potencial y tal que $y(t) - y(s) \geq 99$. Para mostrar que não existe caminho algum de s a t basta exibir um 1-potencial y tal que $y(t)-y(s) \geq n$, onde $n = |N|$, pois o comprimento de qualquer caminho não passa de $n-1$. $n = |N|$

5.2 Algoritmo do caminho mais curto

O algoritmo de busca em largura (veja seção 3.4) resolve nosso problema. O algoritmo recebe nós s e t de um grafo (N, A) e devolve um 1-potencial y; se $y(t)-y(s) < n$ então também devolve um caminho P de s a t tal que $y(t) - y(s) = |P|$.

```
BUSCA-EM-LARGURA (N, A, s, t)
01  para cada i em N faça
02      y(i) ← n      ▷ n faz o papel de ∞
03      π(i) ← NIL
04  y(s) ← 0
05  L ← ⟨s⟩
06  enquanto L ≠ ⟨⟩ faça
07      retire[3] o primeiro elemento, digamos i, de L
08      para cada ij em A(i) faça
09          se y(j) > y(i) + 1      ▷ y(j) = n
10              então y(j) ← y(i) + 1
11                    π(j) ← i
12                    acrescente j ao final de L
13  se y(t) ≥ n
14      então devolva y
15      senão seja P o caminho determinado por π a partir de t
16            devolva y e P
```

[3] Se $L = \langle i_1, i_2, \ldots, i_l\rangle$ antes da operação, então $i = i_1$ e $L = \langle i_2, \ldots, i_l\rangle$ depois da operação.

Eu poderia simplesmente devolve y e π no fim da execução do bloco de linhas 06–12 e deixar que o usuário cuide da execução das linhas 13 a 16. Com isso, o parâmetro t seria suprimido e o usuário poderia escolher qualquer nó para fazer o papel de t.

O algoritmo faz o que promete? Para mostrar que o algoritmo está correto é preciso entender a relação entre as variáveis no início de cada iteração. Digamos que cada iteração começa na linha 06 imediatamente antes da comparação de L com a seqüência vazia. Ao enunciar as invariantes, usaremos a notação $S := \{v : y(v) < n\}$. Diremos que um arco vw está **tenso** se $y(w) - y(v) > 1$, **relaxado** (ou **folgado**) de $y(w) - y(v) \leq 1$ e **justo** se $y(w) - y(v) = 1$.[4] No começo de cada iteração temos as seguinte invariantes:

(i1) cada arco no grafo de predecessores (N, A_π) está justo;

(i2) $\pi(s) = \text{NIL}$ e $y(s) = 0$;

(i3) para cada v em S, existe um caminho de s a v no grafo de predecessores.

A essas invariantes é preciso acrescentar mais algumas para caracterizar L. No começo de cada iteração, seja $i_1, i_2, \ldots, i_l$ a seqüência de elementos de L (isto é, $L =: \langle i_1, i_2, \ldots, i_l \rangle$). Então

(i4) cada arco tenso tem ponta inicial em L;

(i5) $y(i_1) \leq y(i_2) \leq \cdots \leq y(i_l)$ e $y(i_l) \leq y(i_1) + 1$;

(i6) $y(w) \leq y(i_1)$ para cada nó w em $S - \{i_1, \ldots, i_l\}$.

Prove as invariantes! (É verdade também que o grafo de predecessores não tem ciclos; isso pode ser verificado com o mesmo raciocínio que prova (i1) e (i2).)

> É evidente que todas as invariantes valem no início da primeira iteração. Suponha agora que todas valem no início da iteração corrente; vamos mostrar que elas continuam valendo no início da próxima iteração.
>
> Prova de (i1): Todos os novos arcos introduzidos no grafo de predecessores estão em $A(i)$. É evidente que todos eles são justos no fim da iteração corrente e portanto também no início da próxima iteração. Mas isso não prova o invariante! É preciso mostrar que os arcos que já estavam em A_π no início da iteração não deixam de ser justos em virtude das alterações de y na linha 10. Mais especificamente, é preciso provar que na linha 10 *não há arco do grafo de predecessores em* $A(j)$, Façamos isso, então. No início da linha 10, o arco ij está tenso; como $i = i_1$, (i5) e (i6) garante que $y(j) = n$. Por outro lado, o potencial da ponta inicial de cada arco do grafo de predecessores é menor que n, em virtude de (i1). (i5) (i6)
>
> Prova de (i2): Basta mostrar que $j \neq s$ em cada execução da linha 10. Digamos que ij é um arco tenso e $i = i_1$. Por (i5) e (i6), $y(j) = n$. Por outro lado, $y(s) = 0$. Logo $j \neq s$. (i1) (i5)
>
> Prova de (i3): No início da iteração, seja P um caminho com origem s no grafo de predecessores. Em virtude de (i1) e (i2), temos $y(k) < n$ para cada k em P, ou seja, P jamais sai (i6)

[4] Imagine que cada arco vw é um pedaço de barbante de comprimento 1. Imagine também que $y(v)$ e $y(w)$ são as alturas de v e w em relação ao chão. Veja "analog solution", AMO, p.96.

de S. No começo de cada execução da linha 10, temos $y(j) = n$ em virtude de (i5) e (i6); portanto j não está em P. Conclusão: no início da próxima iteração, P continua sendo um caminho no grafo de predcessores. Isso conclui a primeira parte da prova. (i5) (i6)

Falta mostrar que (i3) vale para os nós j que passam a fazer parte de S durante a iteração corrente. Por (i3), há um caminho I de s a i no grafo de predecessores. Suponha que estamos no início de alguma das execuções da linha 10 na iteração corrente. Como já mostramos há pouco, j não está em I. Logo, no fim dessa iteração (e portanto também no início da iteração seguinte), o passeio $I{\cdot}\langle i, j\rangle$ é um caminho de s a j no grafo de predecessores.

Prova de (i4): Durante a iteração corrente, todos os arcos em $A(i)$ deixam de ser tensos. Portanto, ao retirar i de L na linha 07 não estamos comprometendo a validade de (i4). Resta verificar que os arcos que ficam tensos durante a iteração terão sua ponta inicial acrescentada a L. Ora, os únicos arcos que podem ficar tensos são os que estão $A(j)$, para algum j em $A(i)$ tal que $y(j)$ é alterado. Mas os nós j desse tipo são todos acrescentados a L na linha 12.

As provas de (i5) e (i6) são muito fáceis.

Suponha agora que estamos no início da última iteração, quando $L = \langle\rangle$. Então, em virtude de (i4), não há arcos tensos; portanto, y é um 1-potencial. Se $y(t) \geq n$, então $y(t) - y(s) \geq n$ em virtude de (i2). Senão, de acordo com (i3), há um caminho P de s a t no grafo de predecessores. Por (i1) e a desigualdade (5.2), $|P| = y(t) - y(s)$. Portanto, o algoritmo realmente faz o que promete. (i4) (i2) (i3) (i1)

Consumo de tempo. Cada nó entra em L no máximo uma vez (por que?) e cada iteração retira um elemento de L. Logo, teremos no máximo n iterações.

Cada iteração examina a lista de adjacências $A(i)$ e consome $O(|A(i)|)$. Como cada nó do grafo faz o papel de i no máximo uma vez, o consumo total de tempo em todas as execuções do bloco de linhas 08–12 é $O(\sum_i |A(i)|)$. Como essa soma é igual a m, o consumo total de todas as execuções do bloco de linhas 08–12 é $O(m)$ unidades de tempo. $m = |A|$

O consumo de tempo das demais linhas (linhas 01-05 e 13–16) é $O(n)$. Portanto, o consumo de tempo total do algoritmo é

$$O(n + m)\,.$$

Como $m < n^2$, podemos dizer que o algoritmo é $O(n^2)$. Se o grafo é conexo então $m \geq n - 1$ e portanto o consumo de tempo é

$$O(m)\,.$$

5.3 Potencial ótimo

Diremos que um 1-potencial y é $(s, *)$-**ótimo** (ou **ótimo em relação à origem** s) se uma das seguintes alternativas vale para cada nó j:

(1) existe algum caminho P de s a j tal que $|P| = y(j) - y(s)$ e

(2) $y(j) - y(s) \geq n$ e não existe caminho de s a j.

O algoritmo que discutimos na seção anterior produz um 1-potencial $(s, *)$-ótimo.

Não é difícil constatar que a partir de um tal 1-potencial é possível determinar caminhos de comprimento mínimo de s a qualquer outro nó (veja exercício 5.4).

5.4 Caminhos mínimos invertidos

Eis uma variante importante do problema do caminho mais curto:[5]

Problema 5.2 (do caminho mínimo invertido) *Dado um nó t de um grafo (N, A), encontrar, para cada nó i, um caminho de comprimento minimo de i a t.*

Esse problema poderia ser reduzido ao problema usual de busca mediante inversão de todos os arcos (ou seja, troca de cada arco ij por um arco ji). Mas em vez de modificar o grafo é melhor supor que dispomos de uma nova estrutura de dados: para cada nó j, seja

$$\widetilde{A}(j)$$

o conjunto de todos os arcos que *entram* em j. É claro que o conceito de função-predecessor deve ser substituído pelo de **função-sucessor**: uma função parcial $\widetilde{\pi}$ de N em N tal que $ij \in A$ sempre que $\widetilde{\pi}(i) = j$. O **grafo de sucessores** é $(N, A_{\widetilde{\pi}})$ em que $A_{\widetilde{\pi}}$ é o conjunto de todos os arcos da forma $(i, \widetilde{\pi}(i))$.

A definição de 1-potencial continua igual à da seção 5.1: qualquer potencial y tal que $y(j) - y(i) \leq 1$ para todo arco ij. Um 1-potencial y é $(*, t)$**-ótimo** (ou **ótimo em relação ao término** t) se uma das seguintes alternativas vale para cada nó i:

(1) existe algum caminho Q de i a t tal que $|Q| = y(t) - y(i)$ e

(2) $y(t) - y(i) \geq n$ e não existe caminho de i a t.

Um algoritmo, digamos CAMINHOS-MÍNIMOS-INVERTIDOS, para o problema deve devolver um 1-potencial $(*, t)$-ótimo y um caminho Q de s a t tal que $|Q| = y(t) - y(s)$.

É fácil escrever um tal algoritmo de modo que ele consuma $O(n+m)$ unidades de tempo (exercício 5.9). Se o grafo é conexo, teremos $m \geq n - 1$ e portanto o algoritmo consumirá $O(m)$ unidades de tempo.

[5] Veja p.76 do AMO. Também seção 7.2, p.209, AMO.

Exercícios

5.1 [AMO 7.9(g), p.244] Seja s um nó de um grafo (N, A) e y uma função de N em $\mathbb{Z}$ dotada da seguinte propriedade: para cada nó j e qualquer caminho P de s a j, tem-se $|P| \geq y(j) - y(s)$. É verdade que y é um 1-potencial?

5.2 [Bom] Seja s um nó de um grafo (N, A). Para cada nó j, seja P_j um caminho de s a j. Descreva um algoritmo que permita decidir se a coleção de caminhos é ótima, ou seja, se P_j é um caminho de comprimento mínimo de s a j para cada j.

5.3 Prove que no início de cada iteração do algoritmo BUSCA-EM-LARGURA (o início da iteração fica na linha 06, imediatamente antes da comparação de L com a seqüência vazia) vale a seguinte propriedade para todo nó v: se $y(v) < n$ então não existe caminho de s a v de comprimento menor que $y(v)$.

5.4 Escreva uma versão do algoritmo BUSCA-EM-LARGURA que não calcula π mas apenas um 1-potencial $(s, *)$-ótimo y. A partir desse y, calcule um caminho de s a t que tenha comprimento $y(t) - y(s)$ (ou mostre qu um tal caminho não existe). Quanto tempo esse cálculo consome?

5.5 [Versão "capacitada" do problema] Suponha que cada arco ij de nosso grafo tem uma "capacidade" u_{ij}. Um arco ij é **positivo** se $u_{ij} > 0$. Um caminho é **positivo** se todos os seus arcos são positivos. Problema: Dados nós s e t, encontrar um caminho positivo de s a t que tenha comprimento mínimo. Qual a definição apropriada de função-potencial? Escreva e analise um algoritmo que resolva o problema.

5.6 Programe e teste o algoritmo de busca em largura. Sugestão: programe em C (não é preciso escrever em `CWEB`) e use a estrutura de dados do SGB (*Stanford GraphBase*). Inclua no seu módulo pequenas funções de teste que permitam conferir as respostas de sua implementação da busca em largura.

5.7 Brinque com o algoritmo de busca em largura que faz parte do sistema GIDEN (`giden.northwestern.edu/`).

5.8 [AMO 3.19, p.88] Determinar existência de ciclo ímpar passando por um nó i.

5.9 [Caminho mínimo invertidos] Escreva o algoritmo sugerido na seção 5.4.

5.10 Sejam s e t dois nós de um grafo (N, A). Seja y um 1-potencial $(s, *)$-ótimo e z um 1-potencial $(*, t)$-ótimo. (1) Prove que para cada nó i de qualquer caminho

de comprimento mínimo de s a t tem-se $z(i) - z(s) = y(i) - y(s)$ e $y(t) - y(i) = z(t) - z(i)$. (2) Prove que $z(j) - z(i) = 1 = y(j) - y(i)$ para algum arco ij então ij pertence a um caminho de s a t que tem comprimento mínimo.

Capítulo 6

Caminhos de custo mínimo

Este capítulo[1] trata de uma rede (N, A, c) em que c é uma função que atribui um número inteiro a cada arco:

$$c : A \to \mathbb{Z} .$$

Diremos que c é uma **função-custo**. Cada arco ij de nossa rede terá um **custo** inteiro c_{ij}, que pode ser positivo, negativo ou nulo.

O **custo de um passeio** (em particular, o custo de qualquer caminho ou ciclo) na rede é a soma dos custos dos arcos do passeio. O custo de um passeio P será denotado por $c(P)$.

Um caminho P tem **custo mínimo** se $c(P) \leq c(P')$ para todo caminho P' que tenha a mesma origem e o mesmo término que P. Às vezes omitiremos a palavra "custo" e diremos simplesmente que P é um **caminho mínimo**.

> Qual a diferença entre um caminho mínimo e um passeio mínimo? Um passeio mínimo não é necessariamente um caminho, pois pode ter nós repetidos. Um passeio pode "conter" ciclos de custo negativo ou nulo. Um passeio mínimo pode "conter" ciclos de custo nulo. Exemplo: Suponha que $N = \{a, b, c, d, e, f\}$, $A = \{ab, bc, cd, de, ef, be\}$ e que os custos de bc, cd, de e be são nulos. Então $\langle a, b, e, d, c, b, e, f\rangle$ é um passeio de custo mínimo e $\langle a, b, e, f\rangle$ é um caminho de custo mínimo.

6.1 O problema dos caminhos mínimos

O presente capítulo trata do seguinte problema: Dada uma rede (N, A, c) e dois nós s e t, encontrar um caminho de custo mínimo de s a t. É conveniente tratar de um problema

[1] Trata-se de um resumo capítulo 5 (*Shortest Paths: Label-Correcting Algorithms*) do AMO. Tudo isso está muito melhor explicado no cap. 24 (especialmente seções 24.1, 24.2, e 24.5) do CLRS e no cap. 25 do CLR. Veja também o módulo GB_DIJK do *Stanford GraphBase* de Knuth [Knu93].

aparentemente um pouco mais geral:

Problema 6.1 (do caminho de custo mínimo) *Dada uma rede (N, A, c) com função-custo c e um nó s, encontrar, para cada nó t, um caminho de custo mínimo de s a t.*

Este problema é uma generalização do problema de busca (capitulo 3) e do problema do caminho mais curto (capitulo 5): se $c = 0$ temos o primeiro problema e se $c = 1$ temos o segundo. Num certo sentido, também é uma generalização do problema dos ciclos (capitulo 4).

EXEMPLO: Suponha que os nós da rede são s, v, t, que os arcos são sv, vt e st e que os custos são $c_{sv} = 2$, $c_{vt} = -3$ e $c_{st} = 1$. Encontre um caminho de s a t que tenha custo mínimo.

OUTRO EXEMPLO: Suponha que os nós da rede são s, v, w, t, que os arcos são sv, vw, vt, wt e tw e que os custos são $c_{sv} = c_{vw} = c_{vt} = 1$ e $c_{wt} = c_{tw} = -3$. Encontre um caminho de s a t que tenha custo mínimo.

6.2 Redes sem ciclos negativos

Em geral, o problema é computacionalmente muito difícil (NP-difícil).[2] Mas ele fica relativamente simples se a rede não tiver ciclos de custo negativo,[3] ou seja, se

$$c(O) \geq 0 \text{ para todo ciclo } O \text{ na rede.} \tag{6.1}$$

Essa restrição está certamente satisfeita se $c \geq 0$ (trataremos desse caso, em detalhes, no capítulo 8). A restrição também está satisfeita se a rede é acíclica (trataremos desse caso no capítulo 9). O presente capítulo exige apenas a validade de (6.1). A verificação da validade dessa restrição em uma dada rede será considerada no capítulo 7.

Para simplificar a análise dos algoritmos, vamos tratar somente do caso em que o problema tem solução, ou seja, somente do caso em que[4]

$$\text{todos os nós da rede estão ao alcance de } s. \tag{6.2}$$

Não é muito difícil remover essa hipótese (veja exercício 6.19).

[2] Por que o problema é tão difícil em geral? Eis uma pista (veja AMO, p.95). Algoritmos eficientes não sabem procurar caminhos: só sabem procurar *passeios*. Na ausência de ciclos negativos, um passeio de custo mínimo é um caminho (exceto talvez pela presença de ciclos de custo nulo). Mas quando a rede tem ciclos negativos, há passeios de custo arbitrariamente negativo e portanto não existem passeios de custo mínimo.

[3] Mais precisamente, basta que a rede não tenha ciclos negativos ao alcance de s.

[4] Essa hipótese aparece no AMO, p.94, como *Assumption 4.2*.

6.3 Condições de otimalidade: função potencial

Como é possível certificar a minimalidade do custo de um dado caminho que vai de s a t? Em outras palavras, como é possível provar que não existe caminho de s a t que tenha custo menor que um dado número, digamos 99?

Um c-**potencial**[5] é um potencial y (ou seja, uma função y de N em $\mathbb{Z}$) tal que

$$y(j) - y(i) \leq c_{ij} \tag{6.3}$$

para cada arco ij. (Note que o conceito de potencial não envolve o nó s.) Diremos que esta é a **desigualdade triangular** para o arco ij.

Potenciais têm a seguinte propriedade fundamental: o custo de qualquer passeio é limitado inferiormente pela *diferença de potencial* entre seus extremos. Mais precisamente:

Lema 6.2 *Se y é um c-potencial então, para qualquer passeio P com origem s e término t, temos*

$$y(t) - y(s) \leq c(P)\,.$$

Esboço da prova (no caso $|P| = 3$): Se $P = \langle s, i, j, t\rangle$ então

$$\begin{aligned} y(t) - y(s) &= y(t) - y(j) + y(j) - y(i) + y(i) - y(s) \\ &\leq c_{jt} + c_{ij} + c_{si} \\ &= c_{si} + c_{ij} + c_{jt} \\ &= c(P)\,, \end{aligned}$$

como queríamos demonstrar. ∎

> (A propósito, a existência de um c-potencial prova a ausência de ciclos de custo negativo, pois, de acordo com o lema 6.2, $0 \leq c(P)$ para qualquer ciclo P. Voltaremos a esse assunto no próximo capítulo.)

Agora podemos responder a pergunta que abriu esta seção: para provar que *não existe* caminho de s a t que tenha custo menor que, digamos, 99 basta exibir um c-potencial y tal que $y(t) - y(s) \geq 99$.[6]

[5] Lamentavelmente, AMO é confuso a respeito do conceito de função-potencial. Veja seção 5.2, p.135. AMO diz "distance labels" e escreve "d" no lugar do meu "y".

[6] Suponha que c_{ij} é o pedágio que um viajante deve pagar para percorrer o arco ij. (O pedágio de alguns arcos pode ser negativo!) Suponha que temos um "Guia do Viajante Pobre" que atribui um número $y(i)$ a cada nó i da rede e garante a seguinte propriedade: é impossível ir de qualquer nó s a qualquer outro nó t gastando menos que $y(t) - y(s)$ créditos. (Mas o guia não garante que $y(t) - y(s)$ créditos sejam *suficientes* para viajar de s a t!) Nosso guia só serve, então, para dissuadir um viajante pobre de empreender certas viagens.

Portanto, para resolver o problema do caminho mínimo basta exibir um c-potencial y e, para cada t, um caminho P de s a t tal que $y(t) - y(s) = c(P)$. (É claro que nesse caso teremos $y(j) - y(i) = c_{ij}$ para todo arco ij de P.) Diremos que um tal c-potencial y é $(s,*)$-**ótimo**.[7] Como veremos adiante, toda rede sem ciclos negativos admite um c-potencial $(s,*)$-ótimo.

6.4 Algoritmo genérico

Eis um primeiro algoritmo genérico[8] para o problema. O algoritmo recebe uma rede (N, A, c) *sem ciclos de custo negativo* e um nó s. Se a condição (6.2) estiver satisfeita, o algoritmo produz um c-potencial y e uma função-predecessor π dotadas da seguinte propriedade: para cada nó t, a função-predecessor π determina um caminho P de s a t tal que $c(P) = y(t) - y(s)$.

```
FORD (N, A, c, s)     ▷ (N, A, c) não tem ciclos negativos
1   para cada j em N faça
2       y(j) ← ∞
3       π(j) ← NIL
4   y(s) ← 0
5   enquanto y(j) > y(i) + c_ij para algum ij em A faça
6       y(j) ← y(i) + c_ij
7       π(j) ← i
8   devolva y e π
```

Na linha 5, estamos adotando a convenção $\infty + \lambda = \infty$, qualquer que seja λ. Na prática, o ∞ na linha 2 pode ser substituído pelo número $nC + 1$, onde $n := |N|$ e $C := \max_{ij \in A} |c_{ij}|$, pois qualquer caminho na rede tem custo $\leq (n-1)C = nC - C \leq nC$. Se essa substituição for feita, será preciso adotar a convenção $nC + 1 + \lambda = nC + 1$ qualquer que seja λ.

Invariantes. Diremos que um arco vw está **relaxado** se $y(w) - y(v) \leq c_{vw}$, **justo** se $y(w) - y(v) = c_{vw}$ e **tenso** se $y(w) - y(v) > c_{vw}$. No início de cada iteração (imediatamente antes de verificar a condição "$y(j) > y(i) + c_{ij}$ para algum ij em A" na linha 5), valem as seguintes propriedades:

(i1) cada arco em A_π está justo ou tenso;[9]

[7] Veja seção 9.4, p.310, de AMO.

[8] O algoritmo aparece, sob o nome *label-correcting algorithm*, na figura 5.1, p.137, de AMO. Segundo CCPS, este é o "algoritmo de Ford".

[9] Portanto, $y(q) - y(p) \geq c(P)$ para qualquer caminho P de p a q na rede de predecessores.

(i2) para cada arco tenso hk, não existe caminho de k a h no grafo de predecessores (N, A_π);

(i3) $y(s) = 0$ e $\pi(s) = \text{NIL}$;

(i4) se $y(t) < \infty$ então há um caminho de s a t no grafo (N, A_π).

(Prove essas invariantes!) A invariante (i2) é conseqüência de nossa hipótese sobre a ausência de ciclos de custo negativo. Juntas, as invariantes (i1) e (i2) garantem que a rede de predecessores não tem ciclos.

É evidente que todas as invariantes valem no início da primeira iteração. Suponha agora que todas valem no início da iteração corrente; vamos mostrar que elas continuam valendo no início da próxima iteração.

Prova de (i1): O estado do arcos que não têm ponta j não é afetado pela redefinição de $y(j)$. Os arcos que terminam em j serão retirados (pela atribuição $\pi(j) \leftarrow i$) do grafo predecessores. Resta analisar os arcos de A_π que têm ponta inicial j. Com a redefinição de $y(j)$, um tal arco torna-se tenso (se era justo) ou continua tenso (se já era tenso).

Prova de (i2): Seja hk um arco e suponha que existe um caminho P de k a h na rede de predecessores. Em virtude de (i1), temos $y(h) - y(k) \geq c(P)$ por um raciocínio análogo ao (i1)
usado na prova do lema 6.2. Observe que $P \cdot \langle h, k\rangle$ é um ciclo de custo $c(P) + c_{hk}$. Como nossa rede não tem ciclos de custo negativo, devemos ter $y(k) - y(h) \leq c_{hk}$, ou seja, hk não é tenso.

Prova de (i4): Suponha que estamos no início da iteração e já escolhemos o arco ij. Como o arco está tenso, temos $y(i) < \infty$ e portanto existe um caminho — digamos I — de s a i na rede de predecessores. Caso 1: $y(j) = \infty$. Então $\pi(j) = \infty$, donde não há caminho de s a j na rede de predecessores. Portanto, a atribuição $\pi(j) \leftarrow i$ estende até j o caminho I e não altera nenhum dos demais caminhos na rede de predecessores. Caso 2: $y(j) < \infty$. Nesse caso, existe um caminho de s a j – digamos J — na rede de predecessores. Em virtude de (i2), j não está em I. É fácil verificar, então, que depois da atribuição $\pi(j) \leftarrow i$ na linha 07 (i2)
a propriedade (i4) continua válida (ainda que o caminho J não mais pertença à rede de predecessores no fim da iteração).

O algoritmo está correto? Suponha que estamos no início da última iteração. Então $y(j) - y(i) \leq c_{ij}$ para todo arco ij. Portanto y é um c-potencial, como o enunciado do algoritmo promete.

Mostremos em seguida que $y(v) < \infty$ para todo nó v. Se S denota o conjunto dos nós v tais que $y(v) < \infty$ então $(S, N-S) = \emptyset$, uma vez que y é um c-potencial. Diante da hipótese (6.2), isso só é possível se $S = N$. Portanto, $y(v) < \infty$ para todo nó v. Agora (i4) garante que, para todo nó t, existe um caminho P_t de s a t na rede de predecessores. (i4)
Por um raciocínio análogo ao que usamos para provar o lema 6.2, podemos deduzir de (i1) que (i1)

$$y(t) - y(s) \geq c(P_t)\,.$$

Como y é um c-potencial, $y(t) - y(s) = c(P_t)$. Assim, o algoritmo realmente faz o que promete fazer.

Número de iterações. Não está claro que a execução do algoritmo termina depois de um número finito de iterações. Para mostrar isso, é preciso observar mais uma invariante. Se S é o conjunto $\{v : y(v) < \infty\}$, então, no início de cada iteração,

(i5) para cada v em S, temos $y(v) \leq n'C'$, onde $n' := |S|$ e C' é o máximo de $|c_{pq}|$ para todos os arcos pq que têm ambas as pontas em S.

(Prove a invariante!) Segue imediatamente de (i5) que no início de cada iteração temos

$$y(v) \leq nC$$

para todo nó v em S, sendo $C := \max_{ij \in A} |c_{ij}|$.[10] Por outro lado, em virtude de (i1), $y(v) - y(s) \geq c(P_v)$, onde P_v é o caminho de s a v na rede de predecessores. Como $y(s) = 0$ em virtude de (i3), temos $y(v) \geq c(P_v)$. Como P_v tem no máximo $n-1$ arcos, temos $c(P_v) \geq (n-1)(-C) = -nC + C \geq -nC$. Em suma,

(i1) (i3)

$$y(v) \geq -nC\,.$$

Os valores de y são inteiros pois os valores de c são inteiros. Cada iteração reduz o valor de $y(j)$ em pelo menos uma unidade. Como $-nC \leq y(j) \leq nC$, cada nó v da rede pode fazer o papel de j na linha 6 no máximo $2nC$ vezes. Logo, o número de execuções do bloco de linhas 5–7 não passa de

$$(2nC)n = 2n^2C\,. \tag{6.4}$$

Isso mostra, em particular, que a execução do algoritmo termina depois de um número finito de iterações.

Consumo de tempo. Cada teste da condição na linha 5 consome $\mathrm{O}(m)$ unidades de tempo. Logo, o consumo de tempo de cada iteração é $\mathrm{O}(m)$. Como as linhas 1–4 consomem $\mathrm{O}(n)$ unidades de tempo, o algoritmo consome

$$\mathrm{O}(mn^2C)$$

unidades de tempo. (O consumo de tempo é tão elevado porque o algoritmo pode examinar cada arco muitas vezes. Outra maneira de ver isso: o número de iterações é elevado porque, para cada nó i, o valor de $y(i)$ pode diminuir muitas vezes antes de atingir seu valor final.) Como o consumo de tempo depende de c (mais precisamente, o número de iterações depende de c) dizemos que o algoritmo é *pseudo*-polinomial.

6.5 Algoritmo de Ford-Bellman

O algoritmo abaixo é uma implementação genuinamente polinomial do algoritmo FORD.[11] Do ponto de visa do consumo assintótico de tempo, esse é o melhor algoritmo

[10] Em AMO, topo da página 140, a justificativa dessa delimitação está errada.

[11] AMO, p.142, diz que se trata de uma "$\mathrm{O}(nm)$ implementation of the modified label-correcting algorithm". CLRS diz "algoritmo de Bellman-Ford".

conhecido para o problema do caminho mínimo com custos arbitrários.

```
FORD-BELLMAN (N, A, s, c)    ▷ (N, A, c) não tem ciclos negativos
01  para cada i em N faça
02      y(i) ← ∞
03      π(i) ← NIL
04  y(s) ← 0
05  repita n − 1 vezes
06      para cada arco ij em A faça
07          se y(j) > y(i) + c_ij
08              então y(j) ← y(i) + c_ij
09                    π(j) ← i
10  devolva y e π
```

O algoritmo recebe uma rede (N, A, c) sem ciclos de custo negativo e um nó s. Se a condição (6.2) estiver satisfeita, o algoritmo produz um c-potencial y e uma função-predecessor π dotadas da seguinte propriedade: para cada nó t, a função π determina um caminho P de s a t tal que $c(P) = y(t) - y(s)$.

Invariantes. Digamos que o início de cada iteração fica na linha 06 (e não na linha 05), imediatamente antes que um novo arco seja escolhido para fazer o papel de ij. No início de cada iteração, vamos denotar por Γ a seqüência de arcos examinados até agora.[12] Diremos que um caminho é **bem-casado com** Γ se tem origem s e sua seqüência de arcos é uma subseqüência[13] de Γ. (Convém observar que os caminhos na rede de predecessores em geral *não são* bem-casados com Γ.)

Além das invariantes (i1)–(i4), temos a seguinte: no início de cada iteração,

(i6) cada nó w tal que $y(w) < \infty$ é término de um caminho bem-casado com Γ; ademais, $y(w) \leq c(W)$ para qualquer caminho W bem-casado com Γ que termina em w.[14]

É evidente que (i6) vale no início da primeira iteração. Suponha agora que (i6) vale no início de uma iteração qualquer (antes, portanto, da escolha do próximo arco ij a ser examinado). Uma vez escolhido o arco ij, é preciso mostrar que (i6) vale com $\Gamma \cdot \langle ij \rangle$ no lugar de Γ. Suponha que W é um caminho bem-casado em relação a $\Gamma \cdot \langle ij \rangle$ e seja w o término de W. É preciso considerar dois casos. Caso 1: W é bem-casado em relação a Γ. Então temos $y(w) \leq c(W)$ no início da iteração e portanto também $y(w) \leq c(W)$ no

[12] É como se tivéssemos a instrução "$\Gamma \leftarrow \langle\rangle$" entre as linhas 04 e 05 e a instrução "$\Gamma \leftarrow \Gamma \cdot \langle ij \rangle$" entre as linhas 06 e 07. No fim da última iteração, Γ terá exatamente $n - 1$ cópias de cada arco do grafo.

[13] Da mesma forma, por exemplo, que $\langle b, c, e, h \rangle$ é uma subseqüência de $\langle a, b, c, d, e, f, g, h \rangle$ mas $\langle c, a \rangle$ não é subseqüência de $\langle a, b, c, d \rangle$.

[14] O que não significa que todos os arcos do caminho estejam relaxados. Veja o exercício 6.10.

fim da iteração, uma vez que a execução das linhas 07–09 não aumenta o valor de $y(w)$. Caso 2: W não é bem-casado em relação a Γ. Nesse caso, $\langle i, j\rangle$ é segmento terminal de W e portanto $w = j$. Seja W' o segmento inicial de W que termina em i. É claro que W' é bem-casado em relação a Γ. Logo, $y(i) \leq c(W')$ no início da iteração e portanto também $y(i) \leq c(W')$ depois da execução das linhas 07–09. Portanto, no fim da iteração temos $y(w) \equiv y(j) = y(i) + c_{ij} \leq c(W') + c_{ij} = c(W)$. Isso prova (i6).

O algoritmo faz o que promete? Antes de analisar a última iteração, convém estabelecer o seguinte lema:

Lema 6.3 Se a rede (N, A, c) não tem ciclos de custo negativo então, para todo arco vw e para todo caminho V de s a v, existe um caminho W de s a w tal que $c(W) \leq c(V) + c_{vw}$.[15]

DEMONSTRAÇÃO: Se $V{\cdot}\langle v, w\rangle$ é um caminho então tome $W := V{\cdot}\langle v, w\rangle$ e observe que $c(W) = c(V) + c_{vw}$. Suponha agora que $V{\cdot}\langle v, w\rangle$ não é um caminho e portanto V passa por w. Seja V' o segmento inicial de V que termina em w e V'' é o segmento terminal de V que começa em w. Como $V''{\cdot}\langle v, w\rangle$ é um ciclo, temos $c(V'') + c(\langle v, w\rangle) \geq 0$. Observe agora que V' é um caminho de s a w e que $c(V') \leq c(V')+c(V'')+c(\langle v, w\rangle) = c(V)+c_{vw}$. ∎

Suponha agora que estamos no fim da execução do algoritmo. Então Γ é uma composição de $\Gamma_1, \ldots, \Gamma_{n-1}$, sendo cada Γ_k uma enumeração de A. Como todo caminho na rede tem no máximo $n - 1$ arcos, *todo caminho na rede que começa em s é bem-casado com Γ*. Segue daí e de (i6) que (i6)

$$y(w) \leq c(W) \text{ para qualquer caminho } W \text{ de } s \text{ a } w. \tag{6.5}$$

Podemos mostrar agora que y é um c-potencial. Seja vw um arco. Em virtude da hipótese (6.2), existe um caminho, digamos V, de s a v. Em virtude da invariante (i1), temos (i1)
$y(v) - y(s) \geq c(V)$. Como $y(s) = 0$ em virtude de (i3), podemos dizer que $y(v) \geq c(V)$. (i3)
O lema 6.3 garante que existe um caminho W de s a w tal que $c(W) \leq c(V) + c_{vw}$. Finalmente, em virtude de (6.5),

$$y(w) \leq c(W) \leq c(V) + c_{vw} \leq y(v) + c_{vw}\,.$$

Isso mostra que vw é relaxado. Como cada arco está relaxado, y é um c-potencial.

Para concluir a análise, seja t um nó qualquer da rede. Em virtude da hipótese (6.2), existe um caminho T de s a t na rede; em virtude de (6.5), temos $y(t) \leq c(T) < \infty$. Em vista de (i4), existe um caminho de s a t na rede de predecessores; digamos que esse (i4)
caminho é P. Como y é um c-potencial, o lema 6.2 garante que $y(t) - y(s) \leq c(P)$. Por outro lado, $y(t) - y(s) \geq c(P)$ em virtude de (i1). Logo, $y(t) - y(s) = c(P)$, como promete (i1)
o enunciado do algoritmo.

[15] Esse é o Lema 24.10, p.607, no CLRS. Também é a equação (5.1), p.135, em AMO. A demonstração no AMO está errada.

Consumo de tempo. É evidente que o algoritmo consome $O(nm)$ unidades de tempo. Como esse consumo depende apenas de n e m, dizemos que algoritmo é fortemente polinomial.

Como $m = O(n^2)$, pode-se dizer que o algoritmo é $O(n^3)$.

6.6 Implementação FIFO do Ford-Bellman

O algoritmo FORD-BELLMAN examinha os arcos da rede em uma ordem arbitrária. Convém fazer isso de maneira um pouco mais organizada (ainda que isso não reduza o consumo assintótico de tempo):[16]

```
FIFO-FORD-BELLMAN (N, A, c, s)    ▷ (N, A, c) não tem ciclos negativos
01  para cada i em N faça
02      y(i) ← ∞
03      π(i) ← NIL
04  y(s) ← 0
05  L ← ⟨s⟩
06  enquanto L ≠ ⟨⟩ faça
07      retire o primeiro elemento, digamos i, de L
08      para cada ij em A(i) faça
09          se y(j) > y(i) + c_ij
10              então y(j) ← y(i) + c_ij
11                    π(j) ← i
12                    se j ∉ L
13                        então acrescente j ao final de L
14  devolva y e π
```

A seqüência L é manipulada de acordo com a política FIFO: o primeiro nó a entrar na fila é também o primeiro a sair. No começo de cada iteração, L contém todos os nós que são ponta inicial de arcos tensos (mas pode também conter a pontas iniciais de arcos relaxados).

Para mostrar que o algoritmo funciona corretamente, basta refazer a prova da correção do algoritmo FORD-BELLMAN supondo que na seqüência Γ todos os arcos que saem de um mesmo nó são consecutivos.

Graças à ausência de ciclos de custo negativo, cada arco da rede é examinado (na linha 09) no máximo $n - 1$ vezes. Portanto, o algoritmo tem a mesma delimitação assintótica que

[16] A seção 5.3, p.142, de AMO chama essa versão de "FIFO implementation of the label-correcting algorithm" (veja figura 5.5, p.141). FIFO é a sigla da política First-In-First-Out que caracteriza a operação de uma fila.

o algoritmo de FORD-BELLMAN, ou seja, consome

$$O(mn)$$

unidades de tempo.

6.7 Apêndice: Custos reduzidos

Suponha dado um c-potencial y (não necessariamente ótimo) para uma rede (N, A, c). Seja c' a função-custo definida por $c'_{ij} = c_{ij} - y(j) + y(i)$ para cada arco ij. Diz-se que c' é um **custo reduzido**. Observe que

$$c'_{ij} \geq 0$$

para cada ij. Não é difícil mostrar que qualquer caminho que tenha custo mínimo na rede (N, A, c') também tem custo mínimo na rede (N, A, c). Essa observação é útil pois há algoritmos muito mais eficientes (veja capítulo 8) que o de FORD-BELLMAN no caso em que os arcos têm custos não-negativos. A propósito, veja o módulo GB_DIJK do *Stanford GraphBase* de Knuth [Knu93].

Exercícios

6.1 [CCPS 2.19, p.34. CLRS lema 24.1, p.582. AMO property 4.1, p.106.] Mostre (por meio de um exemplo) que um segmento inicial de um caminho de custo mínimo pode não ser um caminho de custo minimo se a rede tiver um ciclo de custo negativo.

6.2 Para cada nó v da rede, seja $\delta(s, v)$ o custo de um caminho de custo mínimo de s a v. Se a rede (N, A, c) não tem ciclos de custo negativo então $\delta(s, w) \leq \delta(s, v) + c_{vw}$ para todo arco vw.[17]

6.3 [Bom] Seja s um nó de uma rede (N, A, c). Para cada nó j, seja P_j um caminho de s a j. Descreva um algoritmo que permita decidir se a coleção de caminhos é ótima, ou seja, se $c(P_j)$ é mínimo para cada j.

6.4 [Parte de AMO 4.45] Suponha dada uma rede (N, A, c) com função-custo $c \geq 0$. Suponha que s, t e p são três nós da rede. Problema: encontrar um passeio[18] de custo mínimo dentre os que começam em s, passam por p, e terminam em t.

[17] Esse lema consta como Lema 24.10, p.607, no CLRS e também como equação (5.1), p.135, em AMO. A demonstração no AMO está errada.

[18] um passeio, ao contrário de um caminho, pode passar *mais de uma vez* por um mesmo nó

1. Uma solução do problem é necessariamente um caminho? 2. Descreva informalmente um algoritmo para resolver o problema. Prove que o seu algoritmo resolve o problema.

6.5 [CCPS 2.29, p.35. Generalização do lema 6.2.] Suponha que para cada nó i temos um caminho s a i de custo z_i. Suponha que para cada arco ij temos $z_j - z_i \leq c_{ij} + K$. Prove que, para cada i, a diferença entre z_i e o custo de um caminho mínimo de s a i é $\leq Kn$.

6.6 [IMPORTANTE. AMO 5.21, p.160] Suponha que a função-custo c de uma rede (N, A, c) tem valores negativos. Digamos que $\lambda = \min_{ij \in A} c_{ij}$. Some $-\lambda$ ao custo de cada arco. Agora os custos são não-negativos. Aplique o algoritmo de Dijkstra (capítulo 8) que é muito mais eficiente que os algoritmos do presente capítulo. Os caminhos produzidos pelo algoritmo têm custo mínimo em relação à função c original?

6.7 Que acontece se o algoritmo FORD for aplicado a uma rede dotada de um ciclo de custo negativo?

6.8 [AMO 5.1, 5.5, p.157] Resolva o problema do caminho mínimo nas redes da figura 5.10, p.158 de AMO.

6.9 [AMO 5.4, p.158] Resolva o problema do caminho mínimo nas redes da figura 5.10, p.158, de AMO.

6.10 Mostre que está errada a seguinte alternativa para a invariante (i6) do algoritmo FORD-BELLMAN: para cada nó x tal que $y(x) < \infty$ tem-se $y(x) - y(s) = \delta_k(s, x)$, onde $\delta_k(s, x)$ é o custo de um caminho de custo mínimo na rede dentre os que caminhos que vão de s a x e têm comprimento $< k$. (i6)

6.11 Considere uma iteração qualquer do algoritmo FORD-BELLMAN. Seja P um caminho na rede de predecessores. Mostre que a seqüência de arcos de P não é necessariamente uma subseqüência de Γ (onde Γ é a seqüência de arcos examinados até o momento).

6.12 Aplique o algoritmo FORD-BELLMAN a uma rede. Interrompa a aplicação do algoritmo em uma iteração qualquer e exiba, para cada nó w, um caminho W com término w que seja bem-casado com Γ (veja invariante (i6)). Construa o seu exemplo de modo que um ou mais desses caminhos não estejam na rede de predecessores.

6.13 Prove que o algoritmo FIFO-FORD-BELLMAN funciona corretamente.

6.14 Suponha que todos os arcos de uma rede têm o mesmo custo Seja s um nó de uma rede. Qual o algoritmo mais rápido para determinar caminhos de custo mínimo de s a cada um dos outros nós?

6.15 [Análise de sensibilidade] Seja s um nó de uma rede (N, A, c) sem ciclos de custo negativo. Para cada nó v, digamos que $\delta(v)$ é a "distância" de s a v, ou seja, o custo de um caminho de custo mínimo de s a v. De quanto podemos diminuir o custo de um certo arco pq sem que os valores da função δ se alterem?

6.16 Digamos que o custo de um caminho é definido como o mínimo dos custos de seus arcos. (Seria mais sugestivo, nesse caso, trocar o termo "custo" por "capacidade".) Problema: Determine um caminho de custo mínimo dentre os vão de s a t.

6.17 [CCPS 2.42, p.36.] Considere o seguinte refinamento do algoritmo FORD. Seja $v_1, \ldots, v_n$ uma enumeração de N tal que $v_1 = s$. Seja $A_1 := \{v_iv_j \in A : i < j\}$ e Γ_1 uma enumeração de A_1 tal que v_iv_j precede v_pv_q se $i < p$. Seja $A_2 := \{v_iv_j \in A : i > j\}$ e Γ_2 uma enumeração de A_2 tal que v_iv_j precede v_pv_q se $i > p$. Faça com que o algoritmo FORD examine os arcos na ordem $\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_1 \cdot \Gamma_2 \cdots$. Compare o consumo de tempo desse algoritmo com o de FORD-BELLMAN.

6.18 Complete os detalhes da seção 6.7.

6.19 Refaça o capítulo todo sem a hipótese (6.2). Procure resolver o problema sem introduzir novos arcos na rede.

6.20 [Scaling. CCPS 2.30, p.35.] Detalhe e discuta a seguinte variante do algoritmo FORD. Para algum inteiro p, suponha que durante o "estágio p" do algoritmo executamos a relaxação $y(j) \leftarrow y(i) - c_{ij}$ somente para arcos ij que satisfazem $y(j) \geq y(i) + c_{ij} + 2^p$. Quando não houver mais arcos desse tipo, o valor de p é diminuído de 1. Se fizermos isso com $p = 0$, teremos o algoritmo FORD original. Deduza do exercício 6.5 uma delimitação do número de iterações em cada "estágio" depois do primeiro. Em seguida, escolha o valor inicial de p de modo que a delimitação também se aplique ao primeiro "estágio". Qual o consumo de tempo total do algoritmo?

6.21 [EXERCÍCIO DE MODELAGEM] Um **pseudo-caminho** é uma seqüência $\langle i_0, i_1, \ldots, i_q\rangle$ de nós tal que, para cada k, tem-se $i_{k-1}i_k \in A$ ou $i_ki_{k-1} \in A$. Os arcos do primeiro tipo são **diretos** e os do segundo são **inversos**. Suponha que cada arco ij da rede tem dois custos, c'_{ij} e c''_{ij}, ambos em $\mathbb{Z}$. O custo de um pseudo-caminho P é $\sum c'_{hi} + \sum c''_{jk}$, onde a primeira soma se estende a todos os arcos diretos de P e a segunda a todos os arcos inversos de P. Problema: Determinar um pseudo-caminho de custo mínimo dentre os que vão de s a t. Enuncie uma condição de otimalidade apropriada.

Capítulo 7

Ciclos negativos

Este capítulo[1] trata de uma generalização do problema dos ciclos que estudamos no capítulo 4. Por outro lado, pode-se dizer que este capítulo reexamina, de um ponto de vista ligeiramente diferente, o capítulo 6 (Caminhos de Custo Mínimo). Como naquele capítulo, temos uma rede (N, A, c) em que c é uma **função-custo**,

$$c : A \to \mathbb{Z} \,,$$

que atribui um custo inteiro arbitrário (positivo, negativo ou nulo) a cada arco.

Problema 7.1 (do ciclo negativo) *Dada uma rede (N, A, c) com função-custo c, encontrar um ciclo de custo negativo.*

Este problema é uma generalização do problema dos ciclos (capitulo 4).

7.1 Condições de inexistência

Um c-**potencial** é qualquer potencial y tal que

$$y(j) - y(i) \leq c_{ij} \tag{7.1}$$

para cada arco ij. A função nula não é, em geral, um c-potencial (a menos que $c \geq 0$). Convém lembrar o lema 6.2: Se y é um c-potencial então, para qualquer passeio P com origem s e término t,

$$y(t) - y(s) \leq c(P) \,.$$

Segue daí imediatamente o seguinte

[1] Trata-se de um resumo da seção 5.5 do AMO. O material também está muito bem explicado no cap.24 (especialmente seções 24.1, 24.2 e 24.5) do CLRS.

Corolário 7.2 *Se uma rede (N, A, c) admite um c-potencial então $c(O) \geq 0$ para todo ciclo O.*

Portanto, para provar que uma rede não tem ciclos de custo negativo, basta exibir um c-potencial. Em outras palavras, um c-potencial é um certificado da inexistência de ciclos negativos em uma rede.

EXERCÍCIO: Prove que a rede abaixo não tem ciclo de custo negativo.

arco	qp	qr	sr	sp	pt	tq	rt	ts
custo	-1	-2	-3	-4	$+3$	$+0$	$+2$	$+1$

7.2 Algoritmo genérico

O seguinte algoritmo resolve o problema do ciclo negativo.[2] Ele é essencialmente idêntico ao algoritmo FORD do capítulo 6 (mas note que o valor inicial de y é 0 em todos os nós).[3]

Para não sobrecarregar o algoritmo com detalhes secundários, vamos permitir que ele devolva não um ciclo mas um quase-caminho (veja seção 2.3) que tem um ciclo de custo negativo como segmento inicial. (Veja exercícios 7.3 e 7.5).

```
FORD (N, A, c)
01  C ← max_{ij∈A} |c_ij|
02  para cada i em N faça
03      y(i) ← 0
04      π(i) ← NIL
05  enquanto y(j) > y(i) + c_ij para algum ij em A faça
06      y(j) ← y(i) + c_ij
07      π(j) ← i
08      se y(j) < −nC
09          então devolva j
10  devolva y
```

O algoritmo recebe uma rede (N, A, c), sendo c uma função que associa custos inteiros aos arcos. O algoritmo devolve um nó j ou um c-potencial y. No primeiro caso, a função-predecessor π determina, a partir de j, um quase-caminho que tem um ciclo de custo negativo como segmento inicial. No segundo caso, o c-potencial prova que a rede não tem ciclo de custo negativo.

[2] O algoritmo é solução do exercício CCPS 2.20, p.34.

[3] Isso é um pouco diferente de AMO e CLR, que começam o algoritmo com $y(i) \leftarrow \infty$ para todo i diferente de s.

Eis as invariantes válidas no início de cada iteração (como de hábito, diremos que um arco vw está **tenso** se $y(w) - y(v) > c_{vw}$, **relaxado** se $y(w) - y(v) \leq c_{vw}$ e **justo** se $y(w) - y(v) = c_{vw}$):

(i1) cada arco em A_π está justo ou tenso;[4]

(i2) $c(O) < 0$ para todo ciclo O na rede de predecessores;

(i3) para cada nó w, se $\pi(w) = \text{NIL}$ então $y(w) = 0$.

Prove essas invariantes!

O algoritmo está correto? Se a execução do algoritmo termina na linha 10 então é evidente que y é um c-potencial. Suponha agora que a execução do algoritmo termina na linha 09. Suponha por um momento que a seqüência

$$\langle j, \pi(j), \pi(\pi(j)), \ldots \rangle \tag{7.2}$$

é finita e seja w o seu último nó. O inverso da seqüência (7.2) é um caminho de w a j, digamos P, na rede de predecessores. De acordo com (i1), $y(j) - y(w) \geq c(P)$. Como (i1)
$y(w) = 0$ em virtude de (i3), temos $y(j) \geq c(P)$. Mas $c(P) \geq (n-1)(-C) = -nC + C \geq$ (i3)
$-nC$. Logo, $y(j) \geq -nC$, o que contradiz a condição da linha 08. Concluímos assim que a seqüência (7.2) é infinita. Portanto, a seqüência inversa é um quase-caminho. Algum segmento inicial do quase-caminho é um ciclo. Por (i2), esse ciclo tem custo negativo. (i2)

Consumo de tempo. Pelas razões que já discutimos na seção 6.4, o algoritmo executa $\leq n^2 C$ iterações. Portanto, o consumo total de tempo é $O(mn^2C)$.

7.3 Algoritmo de Ford-Bellman

O algoritmo de Ford-Bellman é essencialmente o mesmo do capítulo 6. Ele é um aperfeiçoamento de FORD.

```
FORD-BELLMAN (N, A, c)
01  para cada i em N faça
02      y(i) ← 0
03      π(i) ← NIL
04  repita n − 1 vezes    ▷ n := |N|
05      para cada arco ij em A faça
06          se y(j) > y(i) + c_ij
07              então y(j) ← y(i) + c_ij
```

[4] Portanto, $y(q) - y(p) \geq c(P)$ para qualquer caminho P de p a q na rede de predecessores.

```
08                          π(j) ← i
09    para cada arco ij em A faça
10        se y(j) > y(i) + c_ij
11                então π(j) ← i
12                      devolva j
13    devolva y
```

O algoritmo está correto pelos mesmos motivos que o algoritmo FORD-BELLMAN do capítulo 6. O consumo de tempo é, obviamente, $O(mn)$.

7.4 Implementação FIFO

Como no capítulo 6, podemos manter em uma fila os arcos que o algoritmo de Ford-Bellman deve ainda examinar:

```
FIFO-FORD-BELLMAN (N, A, c)
01  para cada i em N faça
02      y(i) ← 0
03      π(i) ← NIL
04      γ(i) ← 0
05  seja L uma seqüência de todos os nós
06  enquanto L ≠ ⟨⟩ faça
07      retire o primeiro elemento, digamos i, de L
08      para cada ij em A(i) faça
09          se y(j) > y(i) + c_ij
10              então y(j) ← y(i) + c_ij
11                    π(j) ← i
12                    se j ∉ L
13                        então acrescente j ao final de L
14      γ(i) ← γ(i) + 1     ▷ i já foi examinado γ(i) vezes
15      se γ(i) > n − 1
16          então devolva j
17  devolva y
```

O consumo de tempo é $O(mn)$.

Exercícios

7.1 Escreva um algoritmo que receba um nó s de uma rede (N, A, c) com $c : A \to \mathbb{Z}$ e devolva (1) um nó de um ciclo de custo negativo que esteja ao alcance de s

ou (2) um c-potencial y e, para cada nó t ao alcance de s, um caminho P de s a t tal que $c(P) = y(t) - y(s)$. O ciclo em (1) e os caminhos em (2) podem ser representados por uma função-predecessor. Implemente o algoritmo usando a estrutura de dados do *Stanford GraphBase*. Escreva rotinas de teste.

7.2 A seguinte versão do algoritmo FORD está correta?

```
FORD-II (N, A, c)
1  para cada i em N faça
2       y(i) ← 0
3       π(i) ← NIL
4  enquanto y(j) > y(i) + c_ij para algum ij em A faça
5       y(j) ← y(i) + c_ij
6       se π(j) ≠ i
7            então π(j) ← i
8            senão devolva j
9  devolva y
```

7.3 Suponha que o algoritmo FORD devolve um nó j. Mostre que j pode não pertencer a um ciclo.

7.4 No fim da execução do algoritmo FORD, é verdade que todo arco ij que não pertence a um ciclo de custo negativo está relaxado?

7.5 Acrescente comandos ao algoritmo FORD para que ele devolva um nó de um ciclo de custo negativo (quando tal ciclo existe). O ciclo propriamente dito é definido pela função-predecessor.

7.6 [Ciclos de custo nulo. AMO 5.19, p.160] Suponha que uma rede (N, A, c) com função-custo c não tem ciclos de custo negativo. Seja y é um c-potencial $(s, *)$-ótimo para algum nó s. Seja E o conjunto dos arcos ij para os quais $y(j) - y(i) = c_{ij}$. Mostre que há uma correspondência biunívoca entre ciclos de custo 0 em (N, A, c) e ciclos em (N, E). Mostre como encontrar um ciclo em (N, E) em $O(|A|)$ unidades de tempo.

7.7 [Grafo tricolorido. AMO 6.29, p.203] Suponha que cada arco de um grafo (N, A) é verde, amarelo ou vermelho. Seja st um arco amarelo. Mostre que uma e apenas uma das seguintes afirmações é verdadeira: (1) st pertence a um ciclo de arcos amarelos e verdes em que todos os arcos amarelos têm uma mesma orientação; (2) st pertence a um corte cujos arcos são amarelos e vermelhos e todos os arcos amarelos têm a mesma orientação.

7.8 [Ciclos negativos não-orientados] Suponha que cada arco de um grafo (N, A) é verde, amarelo ou vermelho. Suponha que cada arco ij tem um custo inteiro c_{ij}

(positivo, nulo ou negativo). Um ciclo **bom** seus arcos diretos são verdes ou amarelos, e seus arcos inversos são vermelhos ou amarelos. O custo de um ciclo é $\sum c_{ij} - \sum c_{kl}$, onde a primeira soma é sobre os arcos diretos do ciclo e a segund sobre os inversos. Especifique uma função-potencial que possa ser usada para provar que não há ciclos bons de custo negativo.

Capítulo 8

Caminhos mínimos sob custos não-negativos

Este capítulo[1] trata do problema dos caminhos mínimos (veja capítulo 6) no caso em que a função-custo é não-negativa: cada arco da rede tem um custo inteiro

$$c_{ij} \geq 0 \,.$$

Nesse caso, a hipótese (6.1) está trivialmente satisfeita e existem algoritmos bem mais eficientes que os do capítulo 6.

Problema 8.1 (do caminho mínimo sob custo não-negativo) *Dada uma rede* (N, A, c) *com função-custo* $c : A \to \mathbb{Z}_{\geq}$ *e um nó* s*, encontrar, para cada nó* t*, um caminho de custo mínimo*[2] *de* s *a* t*.*

Para simplificar a análise dos algoritmos, vamos tratar somente do caso em que o problema tem solução, ou seja, somente do caso em que

todos os nós da rede estão ao alcance de s.[3] (8.1)

Não é muito difícil remover essa hipótese, como sugere o exercício 8.2.

[1] Veja capítulo 4 (*Shortest Paths: Label-Setting Algorithms*) do AMO. Melhor: veja no capítulo 24 (especialmente seções 24.3 e 24.5) do CLRS ou o capítulo 25 do CLR.

[2] Há quem diga que c_{ij} é o *comprimento* do arco ij e que o custo de um caminho mínimo de s a t é a *distância* de s a t. Prefiro não fazer isso e reservar a palavra *comprimento* para designar o número de arcos do caminho.

[3] No AMO, p.94, essa hipótese é registrada como *Assumption 4.2*.

8.1 Condições de otimalidade

Como já vimos no capítulo 6, um c-**potencial** é qualquer função y de N em $\mathbb{Z}$ tal que

$$y(j) - y(i) \leq c_{ij} \tag{8.2}$$

para cada arco ij. Diremos que esta é a **desigualdade triangular** para o arco ij.

Exemplo 1: A função nula ($y = 0$) é um c-potencial, uma vez que $c \geq 0$. Exemplo 2: Se y é um c-potencial e f uma função constante então é evidente que $y - f$ também é um c-potencial. Exemplo 3: Se y mede as "distâncias" a partir de s, ou seja, se $y(i)$ é o custo de um caminho mínimo de s a i, então y é um c-potencial. A prova desse fato não é trivial (veja exercício 8.4).[4]

De acordo com o lema 6.2, se y é um c-potencial então, para qualquer passeio P temos

$$y(k) - y(h) \leq c(P)\,, \tag{8.3}$$

sendo h a origem e k o término de P. Logo, para provar a minimalidade do custo de um caminho P de s a t, basta exibir um c-potencial y tal que $y(t) - y(s) = c(P)$.

Assim, para resolver o problema do caminho mínimo basta exibir um c-potencial y e, para cada t, um caminho P de s a t tal que $y(t) - y(s) = c(P)$. Diremos que um tal potencial y é $(s, *)$-**ótimo**. Como veremos adiante, um c-potencial $(s, *)$-ótimo existe e pode ser facilmente calculado.

8.2 Algoritmo de Dijkstra

O seguinte algoritmo, atribuído a Edsger W. Dijkstra, resolve o problema.[5] Descreveremos aqui a versão "genérica" do algoritmo; algumas variantes de implementação serão discutidas nas próximas seções.

```
DIJKSTRA (N, A, c, s)     ▷ c ≥ 0
01  para cada i em N faça
02        y(i) ← ∞
03        π(i) ← NIL
04  y(s) ← 0
05  Q ← N
06  enquanto Q ≠ ∅ faça
07        escolha i em Q de modo que y(i) seja mínimo
```

[4] Veja propriedade 4.2, p.107, do AMO.

[5] Veja figura 4.6, seção 4.5, p.109, do AMO. Veja exemplo na figura 4.7, p.110, do AMO. Também seção 24.3 do CLRS.

```
08      Q ← Q − {i}
09      para cada ij em A(i) faça
10          se y(j) > y(i) + c_ij      ▷ j ∈ Q
11              então y(j) ← y(i) + c_ij
12                  π(j) ← i
```

Na prática, o ∞ na linha 02 pode ser substituído pelo número $nC + 1$, onde $n := |N|$ e $C := \max_{ij \in A} c_{ij}$, pois qualquer caminho na rede tem custo $\leq (n-1)C = nC - C \leq nC$.

O algoritmo DIJKSTRA recebe uma rede (N, A, c), sendo c uma função-custo não-negativa, e um nó s. Se a hipótese (64) estiver satisfeita,[6] o algoritmo produz uma função-predecessor π e um c-potencial y dotadas da seguinte propriedade: para cada nó t, a função π determina um caminho P de s a t tal que $y(t) - y(s) = c(P)$.

O algoritmo faz o que promete? Para provar que o algoritmo faz o que promete, basta verificar quatro propriedades invariantes. Ao enunciar essas propriedades, diremos que um arco vw está **tenso** se $y(w) - y(v) > c_{vw}$, **relaxado** se $y(w) - y(v) \leq c_{vw}$ e **justo** se $y(w) - y(v) = c_{vw}$.[7]

Eis as invariantes, válidos em cada iteração imediatamente antes da comparação de Q com $\emptyset$ na linha 06:

(i1) cada arco tenso pq tem ambas as pontas em Q;

(i2) cada arco na rede de predecessores é justo;[8]

(i3) se $y(w) < \infty$ então existe um caminho de s a w na rede de predecessores;

(i4) para todo arco vw da rede de predecessores tem-se $v \notin Q$.

Prove essas invariantes!

Suponha agora que estamos no início da última iteração, quando $Q = \langle\rangle$. Então, em virtude de (i1), não há arcos tensos; portanto, y é um c-potencial. O lema 6.2 garante (i1) então que se $y(v) = \infty$ então v não está ao alcance de s, o que contradiz a hipótese (64). Logo, $y(v) < \infty$ para todo v. Por (i3), para todo t existe um caminho P de s a t na rede (i3) de predecessores. Por (i2), $y(t) - y(s) = c(P)$. Portanto, o algoritmo de fato faz o que (i2) promete.

Consumo de tempo do algoritmo. O tempo gasto com a execução do bloco de linhas 01–05 é $O(n)$. O consumo de tempo de todas as execuções da linha 06 também é $O(n)$,

[6] Mesmo que a hipótese não esteja satisfeita, o algoritmo produz resultados corretos: basta interpretá-los de maneira apropriada.

[7] Imagine que cada arco vw é um pedaço de barbante de comprimento c_{vw}. Imagine também que $y(v)$ e $y(w)$ são as alturas de v e w em relação ao chão. Veja "analog solution", AMO, p.96.

[8] Esta é a "condição de folgas complementares" da programação linear.

pois Q diminui a cada iteração. O consumo de tempo de cada execução da linha 07 é $O(|Q|)$ e portanto o consumo de todas as execuções é

$$O(n + (n-1) + \cdots + 2 + 1) = O(n(n+1)/2) = O(n^2)\,.$$

O consumo de tempo de todas as execuções da linha 08 é $O(n)$. O consumo de tempo do bloco de linhas 09-12 para cada i fixo é $O(|A(i)|)$. Logo, o consumo de todas as execuções daquele bloco de linhas é $O(\sum_i |A(i)|) = O(m)$. Em suma, o consumo total de tempo do algoritmo é

$$O(n) + O(n) + O(n^2) + O(n) + O(m) = O(n^2 + m) = O(n^2)$$

unidades de tempo, uma vez que $m < n^2$. Observe que a maior parte do tempo é consumida pelo repetido cálculo do mínimo na linha 07.

> Eis uma versão mais simples dessa mesma análise. O número de iterações do bloco de linhas 07–12 não passa de n pois Q diminui a cada iteração. Em cada iteração, o tempo gasto por cada execução da linha 07 é $O(|Q|)$ e o tempo gasto com cada execução do bloco de linhas 09–12 é $O(|A(i)|)$. Como $|Q| \leq n$ e $|A(i)| \leq n-1$, o consumo total de tempo do algoritmo é $O(n^2)$ unidades de tempo.[9]

Se o grafo é denso, isto é, se $m = \Omega(n^2)$, não se conhece algoritmo assintoticamente melhor que esse. Para grafos não-densos, veja abaixo a implementação HEAP-DIJKSTRA.

8.3 Implementação de Dial

A operação crítica do algoritmo DIJKSTRA, do ponto de vista do consumo de tempo, está na escolha de i em Q (linha 07). Eis uma implementação curiosa que tenta (com sucesso duvidoso) tornar a linha 07 mais eficiente.[10]

Essa implementação explora o fato de que os custos dos arcos são números inteiros. (Note que o algoritmo DIJKSTRA funciona mesmo que os custos sejam números não-inteiros.) Ela usa "baldes" (*buckets*) $Q_0, Q_1, Q_2, \ldots$ para armazenar nós.

```
DIAL-DIJKSTRA (N, A, c, s)
01   C ← max_{ij∈A} c_ij
02   Q_0 ← {s}
03   para k ← 1 até nC faça
04        Q_k ← ∅
05   Q_{nC+1} ← N − {s}
```

[9] Esse é o teorema 4.4, p.111, do AMO.
[10] Veja seção 4.6, p.113, de AMO.

```
06  para cada i em N faça
07      y(i) ← ∞
08      π(i) ← NIL
09  y(s) ← 0
10  enquanto Q_k ≠ ∅ para algum k faça
11      escolha o menor k tal que Q_k ≠ ∅
12      escolha qualquer nó i em Q_k
13      Q_k ← Q_k − {i}
14      para cada ij em A(i) faça
15          se y(j) > y(i) + c_ij
16              então Q_y(j) ← Q_y(j) − {j}
17                    y(j) ← y(i) + c_ij
18                    π(j) ← i
19                    Q_y(j) ← Q_y(j) ∪ {j}
```

Nessa implementação, é claro que a união $Q_0 \cup \cdots \cup Q_{nC+1}$ dos *buckets* corresponde ao conjunto Q do algoritmo DIJKSTRA. A invariante principal que envolve os *buckets* é o seguinte:

(i5) se $v \in Q_k$ então $y(v) = k$.

O consumo de tempo de DIAL-DIJKSTRA é $O(m + nC)$ (mas na prática o consumo fica longe desse limite assintótico). Portanto, o algoritmo é pseudo-polinomial.

8.4 Implementação com heap

O ponto fraco do algoritmo DIJKSTRA em relação ao consumo de tempo está na escolha de i em Q (linha 07). Para tornar o algoritmo mais eficiente, podemos tratar Q como um min-heap com chave y.[11]

```
HEAP-DIJKSTRA (N, A, c, s)
01  para cada i em N faça
02      y(i) ← ∞
03      π(i) ← NIL
04  y(s) ← 0
05  seja Q o conjunto N organizado como um min-heap[12] com chave y
06  enquanto Q ≠ ∅ faça
07      i ← EXTRACT-MIN (Q)
```

[11] Veja seção 24.3, p.595, do CLRS. O algoritmo também está na figura 4.10, p.115, seção 4.7 do AMO. Mas AMO descreve o algoritmo de maneira um tanto suja e inconsistente com o algoritmo DIJKSTRA.

[12] O elemento com menor chave fica no topo do heap.

```
08      para cada ij em A(i) faça
09          valor ← y(i) + c_ij
10          se y(j) > valor
11              então DECREASE-KEY (valor, j, Q)
12                  π(j) ← i
```

A operação EXTRACT-MIN (Q) retira do min-heap Q um nó i que minimiza $y(i)$. e rearranja o que sobrou de modo a restaurar a estrutura de min-heap.

A operação DECREASE-KEY $(valor, j, Q)$ restaura a estrutura de min-heap depois de fazer $y(j) \leftarrow valor$.

Suponha que nosso min-heap é binário (cada pai tem no máximo dois filhos). Então cada execução das linhas 07 e 11 consome $\mathrm{O}(\log_2 |Q|) = \mathrm{O}(\log_2 n)$ unidades de tempo. Logo, o consumo de todas as execuções da linha 07 é $\mathrm{O}(n \log_2 n)$ e o consumo de todas as execuções do bloco de linhas 08–12 é $\mathrm{O}(m \log_2 n)$. Como o consumo das linhas 01–05 é $\mathrm{O}(n)$, podemos dizer que o consumo de tempo total do algoritmo é

$$\mathrm{O}(n) + \mathrm{O}(n) + \mathrm{O}(n \log_2 n) + \mathrm{O}(m \log_2 n) \,.$$

Como $m \geq n - 1$ em virtude da hipótese (64), podemos dizer que o consumo de tempo é

$$\mathrm{O}(m \log_2 n) \,.$$

Isso é melhor que o consumo de tempo do algoritmo DIJKSTRA quando $m = \mathrm{O}(n^2 / \log_2 n)$.

Heap d-ário e valor ótimo de d

Podemos implementar Q como um min-heap d-ário, ou seja, um min-heap em que cada pai tem no máximo d filhos.[13] Então cada execução de DECREASE-KEY consome $\mathrm{O}(\log_d n)$ unidades de tempo e cada execução de EXTRACT-MIN consome $\mathrm{O}(d \log_d n)$ unidades de tempo. Logo, o consumo de todas as execuções da linha 07 é $\mathrm{O}(nd \log_d n)$ e o consumo de todas as execuções do bloco de linhas 08–12 é $\mathrm{O}(m \log_d n)$. O consumo total de tempo do algoritmo é

$$\mathrm{O}(nd \log_d n + m \log_d n) \,.$$

Qual o valor ótimo de d, ou seja, o valor que minimiza $nd \log_d n + m \log_d n$? Pode ser difícil determinar o valor ótimo exato; mas o valor ótimo aproximado, assintótico, é fácil: basta determinar o valor de d que torna as duas parcelas iguais.[14] Assim, o valor ótimo de d é aproximadamente m/n (desde que esse número não seja menor que 2):

$$d = \max(2, \lceil m/n \rceil) \,.$$

[13] Veja p.116 e seção A.2, p.774, do AMO.
[14] Seção 3.2, p.65–66 (*Parameter Balancing*), do AMO.

Para esse valor de d, o consumo de tempo se reduz a

$$O(m \log_d n) .$$

Se $m = \Omega(n^{1+\epsilon})$ para algum $\epsilon > 0$ então $O(m \log_d n) = O(m)$.

8.5 Apêndice: Reverse-Dijkstra

Considere a seguinte variação do problema do caminho mínimo sob custo não-negativo: Dada uma rede (N, A, c) com função-custo $c : A \to \mathbb{Z}_{\geq}$ e um nó t, encontrar, para cada nó s, um caminho de custo mínimo de s a t.[15]

Imagine que a rede é representada pelos "leques de entrada" dos nós: para cada nó i, é dado o conjunto $\widetilde{A}(i)$ dos arcos que entram em i. Resta agora definir uma variante apropriada do conceito de potencial e escrever uma variação apropriada do algoritmo DIJKSTRA (essa variação é conhecida como REVERSE-DIJKSTRA). Veja exercício 8.11.

Exercícios

8.1 [IMPORTANTE. AMO 4.19, p.128. CCPS 2.36, p.36] Considere o problema do caminho de custo mínimo em que alguns arcos têm custo negativo. Construa um exemplo desse tipo que o algoritmo de Dijkstra resolve corretamente. Construa um exemplo que o algoritmo de Dijkstra *não* resolve corretamente.

8.2 Refaça o capítulo todo sem a hipótese (64). Procure resolver o problema sem introduzir novos arcos na rede.

8.3 Seja (N, A, c) uma rede com função-custo $c \geq 0$. Prove que segmentos iniciais de caminhos mínimos são caminhos mínimos. Mais precisamente, mostre que se $\langle i_1, \ldots, i_p, \ldots, i_q \rangle$ é um caminho de custo mínimo então $\langle i_1, \ldots, i_p \rangle$ também é um caminho de custo mínimo.[16]

8.4 Seja s um nó de uma rede (N, A, c) com função-custo $c \geq 0$. Prove a *propriedade triangular* $\delta(s, w) \leq \delta(s, v) + c_{vw}$ onde $\delta(s, x)$ é o custo de um caminho de custo mínimo dentre os que começam em s e terminam em x.[17]

8.5 Inventei o algoritmo abaixo para competir com o algoritmo de Dijkstra. O algoritmo funciona bem?

[15] Veja seção 4.5, p.112, de AMO.
[16] CLRS lema 24.1, p.582. AMO property 4.1, p.106.
[17] CLRS lema 24.10, p.607. AMO theorem 5.1, p.136.

```
1   y(s) ← 0
2   S ← {s}
3   enquanto S ≠ N faça
4       escolha i em S tal que y(i) é mínimo
5       escolha ij em A(i) tal que c_ij é mínimo
6       y(j) ← y(i) + c_ij
7       S ← S ∪ {j}
```

8.6 Inventei o algoritmo abaixo para competir com o de Dijkstra:

```
01  y(s) ← 0
02  S ← {s}
03  se A(s) ≠ ∅
04      então X ← {s}
05      senão X ← ∅
06  enquanto X ≠ ∅ faça
07      escolha i em X tal que y(i) é mínimo
08      escolha ij em A(i) tal que c_ij é mínimo
09      y(j) ← y(i) + c_ij
10      S ← S ∪ {j}
11      X ← X ∪ {j}
12      para cada p em X faça
13          retire de A(p) todos os arcos da forma pq com q ∈ S
14          se A(p) = ∅
15              entao X ← X − {p}
```

O algoritmo funciona bem? Justifique.

8.7 Considere a seguinte variante do algoritmo de Dijkstra.

```
1   y(s) ← 0
2   S ← {s}
3   B ← A(s)
4   enquanto B ≠ ∅ faça
5       escolha ij em B tal que y(i) + c_ij é mínimo
6       retire de B todos os arcos com ponta final j
7       acrescente a B todos os arcos da forma jq com q em N − S
8       y(j) ← y(i) + c_ij
9       S ← S ∪ {j}
```

O algoritmo funciona bem? Que cara tem B no início de cada iteração? Quanto tempo o algoritmo consome no pior caso?

8.8 [CCPS 2.22, p.34] Seja (N, A, c) uma rede com função-custo $c \geq 0$. Sejam S e T subconjuntos de N, sendo $S \cap T = \emptyset$. Queremos encontrar um caminho de custo

mínimo dentre os começam em S e terminam em T. Mostre como reduzir esse problema a um problema ordinário de caminho de custo mínimo.

8.9 [AMO 4.44, p.130. Análise de sensibilidade.] Seja (N, A, c) uma rede com função-custo $c \geq 0$. Seja π uma função-predecessor que descreve caminhos de custo mínimo a partir de um nó fixo s.
1. Seja pq é um arco fora da rede de predecessores (ou seja, $\pi(q) \neq p$). Mostre como calcular o maior número $\beta \leq c_{pq}$ dotado da seguinte propriedade: se trocarmos c_{pq} por $c_{pq} - \beta$ então π continuará descrevendo caminhos de custo mínimo.
2. Suponha que ij é um arco na rede de predecessores (ou seja, $\pi(j) = i$). Mostre como calcular, em tempo $O(|A|)$, o maior número α com a seguinte propriedade: se trocarmos c_{ij} por $c_{ij} + \alpha$ a função π continuará descrevendo caminhos de custo mínimo.

8.10 [CCPS 2.38, p.36] Seja (N, A, c) uma rede com função-custo $c \geq 0$. Considere o problema de encontrar um passeio de custo mínimo dentre os passeios de *comprimento par* que ligam um nó s a um nó t. (Um passeio de custo mínimo pode não ser um caminho.) Mostre como o problema pode ser reduzido a um problema ordinário de caminho de custo mínimo mediate a substituição de cada nó distinto de s e t por dois nós. Repita com "impar" no lugar de "par".

8.11 Complete os detalhes da seção 8.5.

8.12 [Minimum-capacity path. AMO 4.37, p.129] Suponha dada uma rede (N, A, u) sendo u uma função de A em $\mathbb{Z}_{\geq}$. Para cada arco ij, interprete u_{ij} como a *capacidade* do arco. A capacidade de um caminho é então o mínimo de u_{ij} sobre todos os arcos ij do caminho. Probema: Dados nós s e t, determinar um caminho de s a t que tenha capacidade máxima.

Capítulo 9

Caminhos mínimos em redes acíclicas

Este capítulo[1] volta a tratar do problema dos caminhos mínimos (capítulo 6) mas restringe a atenção a grafos acíclicos. Se (N, A) é um grafo acíclico (= *DAG*) então, qualquer que seja a função-custo c, a rede (N, A, c) não tem ciclos de custo negativo e portanto a condição (6.1) está satisfeita.

É fácil resolver o problema dos caminhos de custo mínimo em redes acíclicas; nem é preciso adotar a hipótese (6.2).

9.1 Algoritmo

Como se sabe (veja capítulo 4), um grafo (N, A) é acíclico se e só se admite uma ordem topológica, ou seja, uma enumeração $\langle v_1, v_2, \ldots, v_n \rangle$ dos nós tal que todo arco $v_i v_j$ tem $i < j$. Eis um algoritmo[2] que recebe uma rede acíclica (N, A, c), uma ordem topológica dos nós e um índice o (não confunda "o" com "0") e produz um c-potencial y e, para cada t, um caminho P de v_o a t tal que $c(P) = y(t) - y(v_o)$.

MIN-COST-PATH-IN-DAG $(N, A, c, \langle v_1, \ldots, v_n \rangle, o)$ ▷ $\langle v_1, \ldots, v_n \rangle$ é ordem topológica
1 para cada j em N faça
2 $\quad y(j) \leftarrow \infty$
3 $\quad \pi(j) \leftarrow$ NIL
4 $y(v_o) \leftarrow 0$

[1] Trata-se de um resumo da seção 4.4, p.107, do AMO. Também da seção 24.2, p.592, do CLRS.
[2] O algoritmo usa o paradigma da programação dinâmica.

```
5   para h ← o até n faça
6       para cada ij em A(v_h) faça
7           se y(j) > y(i) + c_ij
8               então y(j) ← y(i) + c_ij
9                     π(j) ← i
```

Na prática, o ∞ na linha 2 pode ser substituído pelo número $nC + 1$, onde $n := |N|$ e $C := \max_{ij \in A} |c_{ij}|$, pois qualquer caminho na rede tem custo $\leq (n-1)C = nC - C \leq nC$.

No início de cada iteração (linha 5, imediatamente antes que i seja comparado com n) temos as seguintes propriedades invariantes:

(i0) $y(v_o) = 0$;

(i1) para cada p, se $y(v_p) < \infty$ então existe um caminho de s a v_p na rede de predecessores;

(i2) cada arco da rede de predecessores é justo;

(i3) todo arco com ponta inicial v_p tal que $p < h$, está relaxado (se convencionarmos que todo arco com ∞ na ponta inicial está relaxado).

Prove essas invariantes!

Consumo de tempo. Durante a execução do algoritmo, cada arco é examinado no máximo uma vez. Logo, o algoritmo consome

$$O(m)$$

unidades de tempo. Portanto, fortemente polinomial.

Exercícios

9.1 [CCPS 2.35, p.36] Suponha dadas tarefas $t_1, \ldots, t_k$. A execução de cada tarefa t_i exige um tempo p_i. Para certos pares (i, j), a execução de t_i deve preceder a execução de t_j. Queremos planejar a execução das tarefas de modo que as restrições de precedência sejam respeitadas e todas as tarefas sejam terminadas tão cedo quanto possível (várias tarefas podem ser executadas ao mesmo tempo).

Parte II

Fluxo máximo entre dois nós

Capítulo 10

Fluxo: introdução

Este capítulo[1] introduz o conceito de fluxo entre dois nós de uma rede. O capítulo serve também para introduzir algumas importantes convenções de notação e discutir algumas propriedades básicas de fluxos.

10.1 Fluxo

Um **fluxo**[2] em uma rede (N, A) é qualquer função de A em $\mathbb{Z}_{\geq}$.[3] Em outras palavras, um fluxo é uma função que atribui um inteiro não-negativo a cada arco da rede.

A coisa mais fundamental a respeito de um fluxo é o seu "excesso" em cada nó. Para definir esse conceito, precisamos introduzir uma convenção de notação. Suponha que x é um fluxo e T uma parte de N. Denote por $\overline{T}$ o complemento de T (ou seja, $\overline{T} := N - T$) e por $x(\overline{T}, T)$ a soma dos valores de x sobre os arcos que entram em T:

$$x(\overline{T}, T) := \sum_{ij \in (\overline{T}, T)} x_{ij} \, .$$

É óbvio que $x(T, \overline{T})$ é a soma dos valores de x nos arcos que saem de T.

O **excesso**, ou **acúmulo**, de x em T é a diferença entre o que *entra* de T e o que *sai* de T:[4]

$$x(\overline{T}, T) \; - \; x(T, \overline{T}) \, .$$

[1] Corresponde às seções 6.1 e 3.5 do AMO. Também à seção 26.1 do CLRS.

[2] AMO às vezes diz *pseudoflow* para se referir a esse conceito muito geral. Alguns livros exigem que um fluxo tenha excesso nulo em quase todos os nós. No presente texto, vamos associar essa condição apenas a expressões como "fluxo de s a t" ou "(s, t)-fluxo".

[3] Em geral, permite-se que um fluxo tenha valores não-inteiros. Mas no presente texto vamos nos restringir aos valores inteiros não-negativos. Veja exercício 11.16.

[4] AMO dá ênfase ao conceito contrário: o que sai de T menos o que entra em T.

Para qualquer nó t, usaremos a abreviatura $x(\bar{t}, t)$ para a expressão $x(\overline{\{t\}}, \{t\})$. Assim, $x(\bar{t}, t) - x(t, \bar{t})$.[5] é o excesso de x em t.

Lema 10.1 (da soma de excessos) *Para qualquer fluxo x em uma rede (N, A) e qualquer parte T de N,*

$$\sum_{t \in T}(x(\bar{t}, t) - x(t, \bar{t})) = x(\overline{T}, T) - x(T, \overline{T})\,.$$

DEMONSTRAÇÃO: Considere os arcos que têm ambas as pontas em T. Cada arco pq desse tipo participa exatamente duas vezes da soma $\sum_{t \in T}(x(\bar{t}, t) - x(t, \bar{t}))$: uma vez (quando q faz o papel de t) ele contribui com x_{pq} e outra vez (quando p faz o papel de t) ele contribui com $-x_{pq}$. Portanto, a contribuição dos arcos que têm ambas as pontas em T é nula. Sobra a contribuição dos arcos que têm uma ponta em $\overline{T}$ e outra em T, ou seja, a contribuição dos arcos em $(\overline{T}, T) \cup (T, \overline{T})$. A contribuição desses arcos é exatamente $x(\overline{T}, T) - x(T, \overline{T})$. ∎

Uma conseqüência óbvia mas importante do lema:

$$\sum_{t \in N}(x(\bar{t}, t) - x(t, \bar{t})) = 0\,. \tag{10.1}$$

10.2 Circulação

Uma **circulação** em uma rede é qualquer fluxo que tenha acúmulo nulo em cada nó. Mais precisamente, uma circulação é qualquer fluxo x tal que

$$x(\bar{j}, j) = x(j, \bar{j})$$

para todo j em N. É claro que o fluxo nulo é uma circulação, ainda que não seja uma circulação muito interessante.

Decomposição em ciclos. Toda circulação pode ser representada por uma coleção de (não mais que m) ciclos, como mostraremos no restante da seção.[6] Seja C um ciclo e ε um inteiro não-negativo. Agora considere o fluxo x definido por

$$x_{ij} := \begin{array}{ll} \varepsilon & \text{se } C \text{ passa por } ij \\ 0 & \text{em caso contrário}\,. \end{array}$$

É fácil perceber que x é uma circulação. Diremos que essa é a circulação **definida por** C **e** ε. Diremos que circulações desse tipo são **elementares**[7].

[5] Seria mais consistente com a notacão anterior escrever $x(\widetilde{A}(t)) - x(A(t))$.
[6] Veja seção 3.5 (*Flow decomposition algorithms*), p.79, do AMO.
[7] AMO não usa esse termo.

Agora suponha que $\mathcal{C}$ é uma coleção de ciclos e que temos um inteiro não-negativo ε_C associado a cada ciclo C em $\mathcal{C}$; diremos que ε_C é a **quantidade de fluxo conduzida por** C. Para cada arco ij, defina

$$x_{ij} := \sum_{C \in \mathcal{C}_{ij}} \varepsilon_C \,,$$

onde $\mathcal{C}_{ij}$ é a coleção de ciclos em $\mathcal{C}$ que passam pelo arco ij. É fácil perceber que x é uma circulação. A recíproca (toda ciculação pode ser descrita por uma coleção de ciclos) é um pouco menos evidente.[8]

Lema 10.2 (da decomposição de circulações) *Para toda circulação x, existe uma coleção $\mathcal{C}$ de ciclos e uma função $\varepsilon : \mathcal{C} \to \mathbb{Z}_{\geq}$ tal que $x_{ij} = \sum_{C \in \mathcal{C}_{ij}} \varepsilon_C$ para cada arco ij. Ademais, $\mathcal{C}$ pode ser escolhido de modo que $|\mathcal{C}| \leq m$.*

A prova do teorema é algorítmica. O algoritmo que esboçaremos a seguir recebe uma circulação x e devolve $\mathcal{C}$ e ε.

```
DECOMPOSIÇÃO-DE-CIRCULAÇÃO (N, A, x)
01  C ← ∅
02  A_x ← {ij ∈ A : x_ij > 0}
03  enquanto A_x ≠ ∅ faça
04      escolha pq em A_x
05      P ← BUSCA (N, A_x, q, p)
06      C ← P · ⟨p, q⟩
07      C ← C ∪ {C}
08      ε_C ← min{x_ij : ij é arco de C}
09      para cada arco ij de C faça
10          x_ij ← x_ij − ε_C
11          se x_ij = 0
12              então A_x ← A_x − {ij}
13  devolva C e ε
```

Na linha 05, o algoritmo BUSCA produz um caminho de q a p no grafo (N, A_x). O algoritmo é o mesmo discutido na seção 3.3. Se não existe caminho de q a p, o algoritmo devolve uma prova desse fato; mas no presente caso, isso não pode acontecer [por que?].

10.3 Fluxo entre dois nós

Dados dois nós s e t do grafo, um fluxo de s a t é qualquer fluxo que tenha excesso nulo em todos os nós distintos de s e t e excesso não-negativo em t. Mais precisamente, um

[8] Veja teorema 3.5 de AMO, p.80.

fluxo de s **a** t ou (s,t)**-fluxo** é qualquer fluxo x tal que

$$x(\bar{j},j) - x(j,\bar{j}) = 0 \text{ para todo } j \text{ em } \overline{\{s,t\}}$$

e $x(\bar{t},t) - x(t,\bar{t}) \geq 0$. O excesso de x em t é o **valor** de x e será denotado por $\text{val}(x)$:

$$\text{val}(x) \;:=\; x(\bar{t},t) - x(t,\bar{t})\ .y$$

De acordo com (10.1), $\text{val}(x) = -x(\overline{s},s) + x(s,\overline{s})$.

Um conjunto T de nós **separa** s de t se $t \in T$ e $s \in \overline{T}$. Diremos também, nessas circunstâncias, que o corte $\nabla(\overline{T},T)$ **separa** s de t ou ainda que $\nabla(\overline{T},T)$ é um (s,t)-**corte**. Como conseqüência do lema 10.1, o valor de um (s,t)-fluxo pode ser medido em qualquer desses cortes:

Corolário 10.3 *Para qualquer* (s,t)*-fluxo* x *e qualquer* (s,t)*-corte* $\nabla(\overline{T},T)$*,*

$$\text{val}(x) = x(\overline{T},T) - x(T,\overline{T})\ .$$

Decomposição em caminhos e ciclos. Se P é um caminho de s a t e ε um inteiro não-negativo então o fluxo x definido por

$$x_{ij} := \begin{array}{ll} \varepsilon & \text{se } P \text{ passa por } ij \\ 0 & \text{em caso contrário} \end{array}$$

é um (s,t)-fluxo e que $\text{val}(x) = \varepsilon$. Diremos que esse é o fluxo **definido por** P **e** ε. Diremos que um (s,t)-fluxo desse tipo é **elementar**[9].

Suponha agora que $\mathcal{P}$ é uma coleção de caminhos, todos de s a t.[10] Suponha também que há um inteiro não-negativo ε_P associado a cada caminho P em $\mathcal{P}$; diremos que ε_P é a **quantidade de fluxo conduzida por** P. Para cada arco ij, defina

$$x_{ij} := \textstyle\sum_{P \in \mathcal{P}_{ij}} \varepsilon_P\ ,$$

onde $\mathcal{P}_{ij}$ é a coleção de caminhos em $\mathcal{P}$ que passam pelo arco ij. É fácil perceber que x é um (s,t)-fluxo e que $\text{val}(x) = \sum_{P \in \mathcal{P}} \varepsilon_P$. A recíproca (todo fluxo de um nó a outro pode ser descrito por uma coleção de caminhos) é um pouco menos evidente.[11] Dizemos que uma coleção $\mathcal{P}$ e uma função $\varepsilon : \mathcal{P} \to \mathbb{Z}_{\geq}$ **representam** um fluxo x se $\text{val}(x) = \sum_P \varepsilon_P$ e nenhum elemento de $\mathcal{P}$ passa por um arco ij tal que $x_{ij} = 0$.

Lema 10.4 (da decomposição de fluxo) *Para todo* (s,t)*-fluxo* x*, existe uma coleção* $\mathcal{P}$ *de caminhos de* s *a* t *e uma função* $\varepsilon : \mathcal{P} \to \mathbb{Z}_{\geq}$ *que, juntas, representam* x*. Ademais,* $\mathcal{P}$ *pode ser escolhido de modo que* $|\mathcal{P}| \leq m$*.*

[9] AMO não usa esse termo.
[10] Veja seção 3.5 (*Flow decomposition algorithms*), p.79, do AMO.
[11] Veja teorema 3.5 de AMO, p.80.

A prova do teorema é algorítmica. O algoritmo que esboçaremos[12] a seguir recebe um (s,t)-fluxo x e devolve $\mathcal{P}$ e ε. Para fazer o serviço,

```
DECOMPOSIÇÃO (N, A, s, t, x)
1  P ← ∅
2  enquanto val(x) > 0 faça
3      A_x ← {ij ∈ A : x_ij > 0}
4      P ← BUSCA (N, A_x, s, t)
5      P ← P ∪ {P}
6      ε_P ← min{x_ij : ij é arco de P}
7      para cada arco ij de P faça
8          x_ij ← x_ij − ε_P
9  devolva P e ε
```

No fim da execução do algoritmo, $x(\bar{i},i) - x(i,\bar{i}) = 0$ para todo nó i, inclusive s e t. Mas não é necessariamente verdade que $x = 0$. Podemos completar o serviço com uma coleção $\mathcal{C}$ de ciclos e uma função e uma função $\varepsilon : \mathcal{C} \to \mathbb{Z}_{\geq}$ tal que $x_{ij} = \sum_{P \in \mathcal{P}_{ij}} \varepsilon_P + \sum_{C \in \mathcal{C}_{ij}} \varepsilon_C$ para cada arco ij.

Exercícios

10.1 Seja x um fluxo em um grafo (N,A) e T um subconjunto de N. É verdade que $\sum_{j \in T} x(\bar{j},j) = x(\overline{T},T)$?

10.2 Seja x um fluxo em um grafo (N,A). Seja X o conjunto dos nós j para os quais $x(\bar{j},j) \neq x(j,\bar{j})$. Mostre que se $|X| \leq 2$ então x é um (s,t)-fluxo para algum par (s,t) de nós.

10.3 Mostre que a soma de duas circulações e uma circulação.

10.4 Seja x uma ciculação e x' um (s,t)-fluxo. Mostre que $x + x'$ é um (s,t)-fluxo e que $\text{val}(x + x') = \text{val}(x')$.

10.5 Suponha que x e x' são (s,t)-fluxos. Mostre que $x + x'$ é um (s,t)-fluxo. Mostre que $\text{val}(x + x') = \text{val}(x) + \text{val}(x')$.

10.6 Seja P um caminho de s a t. Defina e da seguinte maneira: se ij é um arco de P então $e_{ij} = 1$, senão $e_{ij} = 0$. Mostre que e é um (s,t)-fluxo. Mostre que $\text{val}(e) = 1$.

[12] O exercício 10.8 sugere uma versão detalhada do algoritmo.

10.7 Prove que na linha 04 do algoritmo DECOMPOSIÇÃO, existe caminho de s a t no grafo (N, A_x).

10.8 Escreva o algoritmo de decomposição de fluxo de maneira detalhada. O seu algoritmo não deve chamar outros algoritmos. Faça uma análise da correção e do consumo de tempo do seu algoritmo.

Capítulo 11

Fluxo máximo

Este capítulo[1] introduz o problema do fluxo máximo entre dois nós de uma rede com restrições de capacidade. O capítulo culmina com o teorema do fluxo máximo e corte mínimo (= *max-flow min-cut*) de Ford-Fulkerson e Kotzig.

11.1 Problema do fluxo máximo

Um **função-capacidade** em um grafo (N, A) é qualquer função de A em $\mathbb{Z}_{\geq}$:

$$u : A \to \mathbb{Z}_{\geq} .$$

Para cada arco ij, o inteiro não-negativo u_{ij} é a **capacidade** do arco ij na rede (N, A, u). Diremos que uma função x de A em $\mathbb{Z}_{\geq}$ **respeita** u se

$$x \leq u ,$$

ou seja, se $x_{ij} \leq u_{ij}$ para cada arco ij.[2]

Problema 11.1 (do fluxo máximo) *Dados nós s e t de uma rede (N, A, u) com função-capacidade u, encontrar um (s, t)-fluxo que respeite u e tenha valor máximo.*

Em outras palavras, queremos encontrar uma função x de A em $\mathbb{Z}_{\geq}$ tal que $x \leq u$, $x(\overline{j}, j) - x(j, \overline{j}) = 0$ para todo i distinto de s e de t e $x(\overline{t}, t) - x(t, \overline{t})$ é máximo. Para simplificar, dizemos "fluxo máximo" no lugar de "fluxo de valor máximo".

Quaisquer que sejam u, s e t, existe um (s, t)-fluxo que respeita: o fluxo nulo, por exemplo, tem essa propriedade. Assim, todas as instâncias do problema de fluxo máximo são

[1] Corresponde ao capítulo 6 (Maximum Flow: Basic Ideas) do AMO.
[2] A letra "u" é a inicial de "upper bound".

viáveis. É fácil verificar que não há problemas ilimitados de fluxo máximo: o valor de qualquer (s,t)-fluxo x não passa de mU, onde $m = |A|$ e $U = \max_{ij \in A} u_{ij}$. Como veremos adiante, o valor de qualquer (s,t)-fluxo x também não passa de nU (uma vez que nossos grafos não têm arcos paralelos).

Os nós s e t são às vezes chamados **fonte** (= *source*) e **sorvedouro** (= *sink*), respectivamente, da rede. Mas é preciso cuidado com essa terminologia porque em outros contextos uma fonte é um nó com grau de entrada nulo e um sorvedouro é um nó com grau de saída nulo. No presente contexto, não há qualquer restrição ou hipótese sobre os graus da fonte s e do sorvedouro t.

É fácil perceber que o problema do fluxo máximo poderia ter sido formulado assim: *Dado um arco ts em uma rede (N, A, u) com função-capacidade u, encontrar uma circulação $x \leq u$ que maximize x_{ts}.*

11.2 Condições de otimalidade

A **capacidade** de um corte[3] $\nabla(\overline{T}, T)$ é o número[4]

$$u(\overline{T}, T) := \sum_{ij \in (\overline{T},T)} u_{ij} \, .$$

O seguinte lema, muito simples mas muito importante, mostra que o valor de qualquer fluxo é limitado pela capacidade de qualquer corte.[5]

Lema 11.2 (fluxo-versus-corte) *Se x é um (s,t)-fluxo que respeita u e $\nabla(\overline{T}, T)$ é um (s,t)-corte então* $\text{val}(x) \leq u(\overline{T}, T)$.[6]

DEMONSTRAÇÃO: Considere a identidade $\text{val}(x) = x(\overline{T}, T) - x(T, \overline{T})$ que discutimos no corolário 10.3. Então

$$\begin{aligned} \text{val}(x) &= x(\overline{T}, T) - x(T, \overline{T}) \\ &= \sum_{ij \in (\overline{T},T)} x_{ij} \; - \; \sum_{ij \in (T,\overline{T})} x_{ij} \\ &\leq \sum_{ij \in (\overline{T},T)} u_{ij} \, , \end{aligned}$$

pois $x \leq u$ e $x \geq 0$. Em suma, $\text{val}(x) \leq u(\overline{T}, T)$. ∎

Portanto, para mostrar que o valor de qualquer (s,t)-fluxo que respeita u não pode ser maior que um dado número, por exemplo 99, basta exibir um (s,t)-corte $\nabla(\overline{T}, T)$ tal que $u(\overline{T}, T) \leq 99$. Em outras palavras, para mostrar que um (s,t)-fluxo x que respeita u é máximo basta exibir um (s,t)-corte $\nabla(\overline{T}, T)$ que tem capacidade igual a $\text{val}(x)$.

[3] AMO escreve $(S, \overline{S})$ no lugar de $(\overline{T}, T)$.
[4] Atenção! Não confunda com $u(\overline{T}, T) - u(T, \overline{T})$!
[5] Esse lema é um caso particular do Teorema Fraco da Dualidade de programação linear.
[6] Esta é a propriedade 6.1, p.179, de AMO.

Corolário 11.3 *Se x é um (s,t)-fluxo que respeita u e* $\text{val}(x) \leq u(\overline{T},T)$ *para algum (s,t)-corte $\nabla(\overline{T},T)$ então x tem valor máximo.*

A propósito, nessas mesmas circunstâncias concluímos que $\nabla(\overline{T},T)$ é um (s,t)-corte de capacidade mínima. Diremos, simplesmente, que $(\overline{T},T)$ é um **corte mínimo**.

11.3 Teorema do fluxo máximo e corte mínimo

A recíproca do corolário 11.3 é verdadeira, como mostra o teorema a seguir. A prova do teorema contém o germe de todos os algoritmos para o problema do fluxo máximo (problema 11.1).

Teorema 11.4 *Para quaisquer dois nós s e t em uma rede (N,A,u) com função-capacidade u, existe um (s,t)-fluxo x que respeita u tal que*

$$\text{val}(x) = u(\overline{T},T)$$

para algum (s,t)-corte $\nabla(\overline{T},T)$.

DEMONSTRAÇÃO: Digamos que um **pseudo-caminho** é uma seqüência $\langle i_0, a_1, i_1, \ldots, a_q, i_q\rangle$ em que $i_0,\ldots,i_q$ são nós distintos dois a dois, e, para cada k, a_k é o arco $i_{k-1}i_k$ ou o arco i_ki_{k-1}. Os arcos do primeiro tipo são **diretos** e os do segundo são **inversos**. Denotaremos por $\acute{P}$ o conjunto dos arcos diretos de P e por $\grave{P}$ o conjunto dos arcos inversos.

Seja x um (s,t)-fluxo máximo dentre os que respeitam u. Diremos que pseudo-caminho P é **de incremento** se[7]

$$x_{ij} < u_{ij} \text{ se } ij \in \acute{P} \quad \text{e} \quad x_{kl} > 0 \text{ se } kl \in \grave{P}.$$

Seja S o conjunto de todos os nós que são término de algum caminho de incremento que começa em s. Suponha por um instante que $t \in S$. Seja P um pseudo-caminho de incremento que começa em s e termina em t. Escolha o maior número δ tal que $\delta \leq u_{ij} - x_{ij}$ para cada ij em $\acute{P}$ e $\delta \leq x_{ij}$ para cada ij em $\grave{P}$. Seja x' o fluxo definido a partir de x como segue:

$$x'_{ij} = \begin{cases} x_{ij} + \delta & \text{se } ij \in \acute{P} \\ x_{ij} - \delta & \text{se } ij \in \grave{P} \\ x_{ij} & \text{em qualquer outro caso.} \end{cases}$$

É claro que x' é um (s,t)-fluxo que respeita u. É claro também que $\text{val}(x') = \text{val}(x) + \delta$. Como $\delta > 0$, a existência de x' é incompatível com a maximalidade de x.

[7] Tudo isso está discutido no fim da seção 6.4, p.181–183, de AMO.

Devemos concluir, portanto, que $t \notin S$. Assim, o (s,t)-corte $\nabla(\overline{S}, S)$. Mas a definição de S garante que

$$x(S,\overline{S}) = u(S,\overline{S}) \quad \text{e} \quad x(\overline{S},S) = 0\,. \tag{11.1}$$

O corolário 10.3 garante então que $\text{val}(x) = u(S,\overline{S})$, como queríamos demonstrar. ∎ cor 10.3

O teorema continua válido mesmo se nossa definição de fluxo permitir valores não-inteiros (ou seja, mesmo que fluxo seja uma atribuição de números *racionais* não-negativos aos arcos). De acordo com o teorema, o valor máximo de tal fluxo é igual à capacidade de um (s,t)-corte de capacidade mínima. Portanto, o valor de um fluxo de valor máximo continua inteiro, desde que as capacidades dos arcos sejam todas inteiras.[8]

O teorema 11.4, juntamente com o lema 11.2, levam ao célebre teorema do fluxo máximo e corte mínimo (*max-flow min-cut*). O teorema foi publicado em 1956 por Ford e Fulkerson e, independentemente, por Kotzig.

Teorema 11.5 (do fluxo máximo e corte mínimo) *Para quaisquer dois nós s e t em uma rede (N,A,u) com função-capacidade u tem-se*

$$\max_{x \le u} \text{val}(x) = \min_{T} u(\overline{T},T)\,,$$

onde o máximo é tomado sobre todos os (s,t)-fluxos que respeitam u e o mínimo é tomado sobre todos os (s,t)-cortes.

Exercícios

11.1 Mostre que o problema do fluxo máximo é equivalente ao seguinte: dado um arco ts em uma rede (N,A,u), encontrar uma circulação x que respeite u e maximize x_{ts}.

11.2 [Importante!] Considere a seguinte tentativa de inventar um algoritmo para o problema do fluxo máximo. Comece com o fluxo nulo e faça incrementos sucessivos: em cada iteração, acrescente um fluxo elementar (veja seção 10.3) ao fluxo já existente. Assim, cada iteração começa com um (s,t)-fluxo x que respeita u e consiste no seguinte: (1) encontre um fluxo elementar e que respeita $u - x$ e tem valor não-nulo; (2) comece nova iteração com o fluxo $x + e$ no lugar de x. Esse algoritmo resolve o problema?

11.3 Um (s,t)-fluxo x que respeita uma função-capacidade u é **maximal** se não existe (s,t)-fluxo elementar e tal que $\text{val}(e) > 0$ e $x + e$ respeita u. Escreva um algoritmo que calcule um (s,t)-fluxo maximal. Mostre que um fluxo maximal não é necessariamente máximo.

[8] Veja exercício 11.16.

11.4 [AMO 6.14, p.201, fig.6.23] A matriz abaixo dá as capacidades dos arcos de uma rede simétrica (veja figura 6.23, p.201, de AMO). Encontre um (s,t)-fluxo de valor máximo. Exiba o estado da rede no início de cada iteração. Mostre um (s,t)-corte de capacidade mínima.

	s	a	b	c	d	e	t
s	–	4	3	–	–	–	–
a	4	–	–	2	3	–	–
b	3	–	–	2	–	–	–
c	–	2	2	–	–	5	–
d	–	3	–	–	–	4	–
e	–	–	–	5	4	–	8
t	–	–	–	–	–	8	–

11.5 [AMO 6.15, p.201, fig.6.24] A matriz abaixo dá as capacidades dos arcos de uma rede (veja figura 6.24, p.201, de AMO).

	s	2	3	4	5	t
s	–	1	1	1	–	–
2	–	–	1	–	–	–
3	–	–	–	1	–	1
4	–	–	–	–	1	1
5	–	–	–	–	–	1
t	–	–	–	–	–	–

Encontre uma coleção máxima de caminhos de s a t disjuntos nos arcos. Enumere todos os (s,t)-cortes. Observe que o número máximo de caminho de s a t disjuntos nos arcos é igual à cardinalidade minima de um corte do tipo $\nabla(\overline{T},T)$.

11.6 [Programa linear inteiro] Formule o problema do fluxo máximo (problema 11.1) como um programa linear inteiro. Escreva o dual da relaxação linear do programa. Prove o teorema fraco da dualidade para esse par de problemas. Compare com o lema 11.2.

11.7 Considere o seguinte problema, que é equivalente ao problema do fluxo máximo: dado um arco ts em uma rede (N,A,u) com função-capacidade u, encontrar uma circulação $x \leq u$ que maximize x_{ts}. Formule o problema como um programa linear inteiro. Escreva o dual da relaxação linear do programa. Prove o teormea fraco da dualidade para esse par de programas lineares.

11.8 Sejam s e t dois nós de uma rede (N,A,u) em que u é uma função-capacidade. Suponha que y é uma função de N em $\mathbb{Z}$ e z uma função de A em $\mathbb{Z}$ tais que

$$y(j) - y(i) \leq z_{ij} \tag{11.2}$$

para cada arco ij. Mostre que para qualquer (s,t)-fluxo tem-se

$$\text{val}(x)\,(y(t) - y(s)) \leq zu\,, \tag{11.3}$$

onde $zu := \sum_{ij \in A} z_{ij} u_{ij}$. Deduza daí que se vale a igualdade em (11.3) então x é um fluxo de valor máximo.

11.9 Nas condições do exercício 11.8, mostre que existe um (s,t)-fluxo x e um par (y,z) que satisfaz (11.2) bem como a igualdade

$$\text{val}(x)\,(y(t) - y(s)) = zu\,.$$

Sugestão: use o teorema do fluxo máximo e corte mínimo (teorema 11.5); use y e z com valores em $\{0,1\}$.

11.10 [AMO 6.26, p.203] Queremos resolver o problema do (s,t)-fluxo máximo em uma rede (N,A,u). Suponha que um nó i distinto de s e de t tem grau de entrada nulo. Podemos remover esse nó (sem afetar o valor de um fluxo de valor máximo)? E se i tem grau de saída nulo?

11.11 [Grafo tricolorido. AMO 6.29, p.203] Suponha que cada arco de um grafo (N,A) é verde, amarelo ou vermelho. Seja st um arco amarelo. Mostre que uma e apenas uma das seguintes afirmações é verdadeira: (1) st pertence a um ciclo de arcos amarelos e verdes em que todos os arcos amarelos têm uma mesma orientação; (2) st pertence a um corte cujos arcos são amarelos e vermelhos e todos os arcos amarelos têm a mesma orientação.

11.12 [AMO 6.34, p.204] Quais das seguinte afirmações são verdadeiras e quais são falsas? Todas têm como contexto uma rede (N,A,u) e nós s e t. Os fluxos são sempre de s a t e os cortes são sempre os que separam s de t.

1. Se um fluxo x é máximo então $x_{ij} = 0$ ou $x_{ji} = 0$ para cada arco ij.
2. Existe um fluxo de valor máximo x tal que $x_{ij} = 0$ ou $x_{ji} = 0$ para cada arco ij.
3. Se as capacidades dos arcos forem todas diferentes, então há um único corte de capacidade mínima.

11.13 [AMO 6.28, p.203] Suponha dado um (s,t)-fluxo de valor máximo em uma rede (N,A,u). Mostre como é possível encontrar um corte de capacidade mínima em tempo $O(m)$. Agora suponha dado um (s,t)-corte de capacidade mínima. Esse corte pode ser usado para obter um fluxo de valor máximo rapidamente (ou seja, em tempo menor que o necessário para resolver o problema sem qualquer informação adicional)?

11.14 [Fluxo ímpar. AMO 6.33, p.204] Esta questão se refere a (s,t)-fluxos em uma rede com função-capacidade u. Digamos que u é **par** se u_{ij} é par para todo arco ij e é **ímpar** se u_{ij} é impar para todo ij. Definições análogas valem para os conceitos de fluxo par e fluxo ímpar. Se todos os arcos têm capacidade par, é verdade que qualquer fluxo de valor máximo é par? Nas mesmas condições, é verdade que o valor de qualquer fluxo de valor máximo é par? Se todos os arcos têm capacidade ímpar, é verdade que qualquer fluxo de valor máximo é ímpar? Nas mesmas condições, é verdade que o valor de qualquer fluxo de valor máximo é ímpar?

11.15 [Rede bipartida, capacidade unitária] Suponha que (N,A) é um grafo bipartido, ou seja, $N = N_1 \cup N_2$, $N_1 \cap N_2 = \emptyset$ e e todo arco tem uma das pontas (inicial ou final) em N_1 e a outra em N_2. Sejam s e t dois nós em N_1 e seja x um (s,t)-fluxo tal que $x_{ij} \leq 1$ para cada arco ij. Supondo que $|N_1| \leq 10|N_2|$, dê uma delimitação superior para o valor de x.

11.16 [Importante: Fluxo não-inteiro. AMO 6.40, p.205] Seja (N,A,u) uma rede com função-capacidade u (como de hábito, os valores de u são inteiros não-negativos). Sejam s e t dois nós da rede. Relaxe nossa definição oficial de fluxo: permita que um fluxo "relaxado" tenha valores reais não-negativos arbitrários. Suponha agora que x é um fluxo relaxado de s a t que tem valor máximo. Sugira um algoritmo que converta x em um fluxo usual (ou seja, com valores em $\mathbb{Z}_{\geq}$) de mesmo valor. Qual o consumo de tempo do seu algoritmo?

11.17 [Arco universal. AMO 6.30, p.203] Seja ij um arco de uma rede (N,A,u). Suponha que para todo fluxo de valor máximo x de um nó s a um nó t tem-se $x_{ij} = u_{ij}$. Mostre que ij pertence a algum (s,t)-corte $(\overline{T},T)$ que minimiza $u(\overline{T},T)$.

11.18 [Importante: Submodularidade. AMO 6.38–6.39, p.204] Sejam s e t dois nós de uma rede (N,A,u) com função-capacidade u. Esta questão se refere às capacidades dos cortes que separam s de t. Prove que se $(\overline{X},X)$ e $(\overline{Y},Y)$ são cortes de capacidade mínima então também $(\overline{X \cap Y}, X \cap Y)$ e $(\overline{X \cup Y}, X \cup Y)$ têm capacidade mínima. Sugestão: prove preliminarmente a seguinte "propriedade submodular": $u(\overline{X \cap Y}, X \cap Y) + u(\overline{X \cup Y}, X \cup Y) \leq u(\overline{X},X) + u(\overline{Y},Y)$.

11.19 [Corte mínimo com número mínimo de arcos. AMO 7.27, p.247] Sejam s e t dois nós de uma rede (N,A,u) com função-capacidade u. Queremos determinar um (s,t)-corte de capacidade mínima. que tenha o menor número possível de arcos. Para cada arco ij, seja $u'_{ij} := mu_{ij} + 1$, onde $m := |A|$. Mostre que um corte de capacidade mínima na rede (N,A,u') resolve nosso problema.

11.20 [Bom] Discuta o seguinte problema: dados nós s e t de uma rede (N,A,u) e um número inteiro φ, encontrar um (s,t)-fluxo x tal que $\text{val}(x) = \varphi$.

11.21 [Capacidades nos nós. AMO 6.25, p.203] Suponha que em nossa rede não só cada arco tem uma capacidade como também cada nó i distinto de s e t tem uma capacidade não-negativa $a(i)$ que limita a quantidade de fluxo que passa por i. Como resolver o problema de determinar um fluxo de um nó s a um nó t que respeite todas as capacidades e seja máximo? Transforme esse problema em um problema padrão de fluxo máximo (que só tem capacidades nos arcos). Do ponto de vista de complexidade de pior caso, o problema com capacidades nos nós é mais difícil que o problema padrão?

11.22 [Caminhos disjuntos. AMO 6.45, p.205] Seja (N, A) um grafo e sejam S e T dois subconjuntos de N, ambos não-vazios. Uma coleção de caminhos de S a T é **disjunta** se cada arco do grafo pertence a no máximo um dos caminhos da coleção. Descreva um algoritmo que encontre uma coleção disjunta máxima.

Mais exercícios

Esta série de exercícios discute (1) o efeito de alterações na rede e (2) algumas aplicações.

11.23 [Capacidades infinitas. AMO 6.23, p.203] Suponha que alguns arcos de uma rede (N, A, u) têm capacidade infinita (ao contrário de nossas definições usuais). Digamos que B é o conjunto de arcos com capacidade infinita. Suponha também que não há caminho de s a t cujos arcos estão todos em B. Mostre que, do ponto de vista do problema do fluxo máximo, podemos substituir a capacidade de cada arco em B pelo número $\sum_{ij \in A-B} u_{ij}$.

11.24 [Arcos paralelos. AMO 6.24, p.203] Como podemos aplicar nossa teoria e algoritmos a um grafo com arcos paralelos (dado que em nossas definições usuais grafos não têm arcos paralelos).

11.25 [Arcos vitais. AMO 7.7, p.244] Sejam s e t dois nós de uma rede (N, A, u). Nesse exercício, todo fluxo é um (s,t)-fluxo que respeita u e todo corte é um (s,t)-corte. Um arco pq é **vital** se a redução de u_{pq} a 0 causa a maior redução possível no valor do fluxo de valor máximo. Prove ou desprove cada uma das afirmação a seguir:

a. Se pq é vital então u_{pq} é máximo.

b. Seja x um fluxo de valor máximo. Se pq é vital então x_{pq} é máximo.

c. Seja x um fluxo de valor máximo. Se pq é vital então existe um corte de capacidade mínima tal que $x_{pq} \geq x_{ij}$ para todo ij no corte.

d. Se pq é vital então pq pertence a algum corte de capacidade mínima.

e. Um arco vital pode não ser único.

11.26 [Arco menos vital. AMO 7.8, p.244] Sejam s e t dois nós de uma rede (N, A, u). Todo fluxo nesse exercício é um (s,t)-fluxo que respeita u e todo corte é um (s,t)-corte. Um arco kl é **secundário** (= *least vital*) se a redução de u_{kl} a 0 causa a menor redução possível no valor do fluxo de valor máximo. Prove ou desprove cada uma das afirmação a seguir:

a. Seja x um fluxo de valor máximo. Se $x_{kl} = 0$ então kl é secundário.

b. Se x é um fluxo de valor máximo Se x_{kl} é mínimo então kl é secundário.

c. Se ij pertence a um corte de capacidade mínima então ij não é secundário.

11.27 [AMO 6.34, p.204] Quais das seguinte afirmações são verdadeiras e quais são falsas? Todas têm como contexto uma rede (N, A, u) e nós s e t. Os fluxos são sempre de s a t e respeitam u; os cortes são sempre os que separam s de t.

1. Se multiplicarmos a capacidade de cada arco por um número positivo λ, um corte de capacidade mínima continua tendo capacidade mínima.
2. Se somarmos um número positivo λ à capacidade de cada arco, um corte de capacidade mínima continua tendo capacidade mínima.

11.28 [AMO 7.9(a–b), p.244] Sejam s e t dois nós de uma rede capacitada (N, A, u) e x um (s,t)-fluxo de valor máximo dentre ps que respeitam u. Quais das seguinte afirmações são verdadeiras?

a. Se a capacidade de cada arco é um múltiplo de α, então x_{ij} é múltiplo de α para cada ij.

b. Se somarmos α à capacidade de cada arco, estaremos somando um múltiplo de α ao valor do fluxo máximo.

11.29 [AMO 7.26, p.247] Sejam s e t dois nós de uma rede capacitada (N, A, u) e considere o valor v^* de um (s,t)-fluxo de valor máximo dentre os que respeitam u. Para cada par (i, j) de nós, seja $\alpha[i, j]$ o aumento no valor de v^* que obteríamos se u_{ij} fosse ∞ (se ij não for um arco, introduza um tal arco no grafo).

a. Mostre que $\alpha[i, j] \leq \alpha[s, j]$ e $\alpha[i, j] \leq \alpha[i, t]$.

b. Mostre que $\alpha[i, j] = \min\{\alpha[s, j], \alpha[i, t]\}$.

b. Mostre como $\alpha[i, j]$ pode ser calculado para todo par (i, j) mediante resolução de $\mathrm{O}(n)$ problemas de fluxo máximo.

11.30 [Re-otimização. AMO 6.35, p.204] Suponha dado um (s,t)-fluxo x, de valor máximo, em uma rede (N, A, u). Suponha agora que um número $k > 0$ foi somado à capacidade de um determinado arco pq. Mostre como resolver o problema em tempo $\mathrm{O}(km)$. Agora suponha que um número $k > 0$ foi subtraído da capacidade de um determinado arco pq; é possível re-otimizar o problema em tempo $\mathrm{O}(km)$?

11.31 [Escalonamento de aviões. AMO 6.32, p.204] Uma companhia de aviação quer servir p vôos com o menor número possível de aviões. Para isso, ela precisa combinar esses vôos da maneira mais eficiente possível. Cada vôo i deve começar no instante a_i e termina no instante b_i. Um avião precisa de r_{ij} horas para retornar do destino do vôo i à origem do vôo j. Sugira uma maneira de resolver o problema.

11.32 [Escalonamento em máquinas paralelas uniformes. AMO 6.17, p.201] A tabela abaixo descreve um problema de escalonamento em máquinas paralelas uniformes (veja *Application 6.4*, p.172, AMO). Formule o problema como um problema de fluxo máximo. Resolva o problema supondo que há duas máquinas disponíveis diariamente.

tarefa (j)	1	2	3	4
tempo de processamento (p_j dias)	2.1	3.1	5.0	1.8
tempo de liberação (r_j)	1	5	0	2
prazo (d_j)	3	7	6	5

Capítulo 12

Redes simétricas e pseudofluxo

É muito desconfortável escrever algoritmos que envolvam pseudo-caminhos como os que usamos na seção 11.3. Para evitar esse incômodo, vamos nos restringir a grafos simétricos e trabalhar com pseudofluxo.

12.1 Fluxo em redes simétricas

Um grafo (N, A) é **simétrico** se $ji \in A$ sempre que $ij \in A$. Ao resolver o problema do fluxo máximo podemos nos restringir a redes simétricas:[1]

Fato 12.1 *É suficiente resolver o problema 11.1 do fluxo máximo em grafos simétricos.*

DEMONSTRAÇÃO: Seja (N, A, u) uma rede arbitrária com função-capacidade u. Para cada arco ij, se ji não é um arco então acrescente ji ao grafo e defina

$$u_{ji} := 0\,.$$

Seja (N, A', u') é a rede simétrica resultante dessas operações.

Suponha dado um fluxo x' na nova rede. É claro que a restrição x de x' a A tem os mesmos excessos em cada nó que x':

$$x(\bar{i}, i) - x(i, \bar{i}) = x'(\bar{i}, i) - x'(i, \bar{i})$$

para cada nó i. Em particular, se x' é um (s, t)-fluxo então x também é um (s, t)-fluxo e $\mathrm{val}(x) = \mathrm{val}(x')$. ∎

[1] Essa é a *Assumption 6.4* na seção 6.1, p.168, de AMO.

Suporemos que a estrutura de dados que representa o grafo permite acesso rápido so "irmão" de cada arco: dado um arco ij, é possível obter o arco ji em tempo $O(1)$.

Numa rede simétrica, um fluxo x é **normalizado**[2] se, para cada arco ij, tem-se $x_{ij} = 0$ ou $x_{ji} = 0$. Qualquer fluxo x pode ser transformado facilmente num fluxo normalizado x' de mesmo valor: para cada arco ij, se $x_{ij} \geq x_{ji}$ então faça $x'_{ij} = x_{ij} - x_{ji}$ e $x'_{ji} = 0$; senão, faça $x'_{ji} = x_{ji} - x_{ij}$ e $x'_{ij} = 0$.

12.2 Pseudofluxo

Numa rede simétrica, cada par (ij, ji) de arcos será tratado como um único objeto. Dado um fluxo x, podemos entender o número $x_{ij} - x_{ji}$ como a "intensidade do fluxo de i a j" ao longo do par (ij, ji). Analogamente, o número $x_{ji} - x_{ij}$ pode ser entendido como a "intensidade do fluxo de j a i" ao longo do par de arcos. Dois fluxos x e x' serão considerados equivalentes se tivermos

$$x'_{ij} - x'_{ji} = x_{ij} - x_{ji}$$

para cada arco ij. É evidente que se x' e x são equivalentes nesse sentido então $x'(\bar{i}, i) - x'(i, \bar{i}) = x(\bar{i}, i) - x(i, \bar{i})$ para cada nó i.

Essa considerações levam ao conceito de pseudofluxo. Um **pseudofluxo**[3] em um grafo (N, A) é qualquer função de A em $\mathbb{Z}$. Em outras palavras, um pseudofluxo associa números inteiros (positivos, nulos ou negativos) aos arcos do grafo. Embora o conceito faça sentido em qualquer grafo, ele só é usado em grafos simétricos.

Um pseudofluxo será usualmente denotado por $\check{x}$. A notação não é muito feliz, porque dá a falsa impressão de que $\check{}$ é um operador aplicado a algum objeto x previamente definido. Mas teremos que conviver com esse desconforto.

O **excesso**, ou **acúmulo**, de um pseudofluxo $\check{x}$ em um subconjunto T de N é o número $\check{x}(\overline{T}, T) := \sum_{ij \in A} \check{x}_{ij}$. Se $T = \{t\}$, o excesso de $\check{x}$ em t é denotado por $\check{x}(\bar{t}, t)$. O lema 10.1 pode ser reformulado assim: para qualquer pseudofluxo $\check{x}$ em uma rede (N, A) e qualquer parte T de N,

$$\sum_{t \in T} \check{x}(\bar{t}, t) = \check{x}(\overline{T}, T)\,.$$

Segue daí imediatamente a seguinte definição: dados dois nós s e t de um grafo simétrico (N, A), um (s,t)-**pseudofluxo** é qualquer pseudofluxo $\check{x}$ tal que $\check{x}(\bar{i}, i) = 0$ para cada nó i em $N - \{s, t\}$ e $\check{x}(\bar{t}, t) \geq 0$.

Numa rede simétrica (N, A, u) com função-capacidade u, dizemos que um pseudofluxo $\check{x}$ **respeita** u se

$$-u_{ji} \leq \check{x}_{ij} \leq u_{ij}$$

[2] AMO não usa esse termo.

[3] AMO usa o termo *pseudoflow* para designar um conceito diferente.

para cada arco ij. O lema 11.2 pode então ser reformulado assim: se $\check{x}$ é um (s,t)-pseudofluxo que respeita u e $\nabla(\overline{T},T)$ é um (s,t)-corte então $\check{x}(\bar{t},t) \leq u(\overline{T},T)$.

12.3 Pseudofluxo versus fluxo

Em grafos simétricos, pseudofluxos são equivalentes a fluxos num sentido que passamos a discutir. Dado um fluxo x em um grafo simétrico (N,A), o correspondente pseudofluxo $\check{x}$ é definido da seguinte maneira:

$$\check{x}_{ij} = x_{ij} - x_{ji}$$

para cada arco ij.[4] Observe que $\check{x}_{ij} = -\check{x}_{ji}$ para cada arco ij. Eis um pequeno algoritmo que recebe um fluxo x e devolve, em tempo $O(m)$, o correspondente pseudofluxo:

```
PSEUDOFLUXO (x)
1  para cada ij em A faça
2        x̌_ij ← x_ij − x_ji
3  devolva x̌
```

Eis uma relação fundamental entre um fluxo x e o correspondente pseudofluxo $\check{x}$ num grafo simétrico: os excessos de x e $\check{x}$ em cada nó são os mesmos, ou seja,

$$x(\bar{t},t) - x(t,\bar{t}) \;=\; \check{x}(\bar{t},t)$$

para cada nó j (veja exercício 12.1). Segue daí que

$$x \text{ é um } (s,t)\text{-fluxo se e só se } \check{x} \text{ é um } (s,t)\text{-pseudofluxo.} \tag{12.1}$$

É fácil verificar também que $\check{x}(\overline{T},T) = -\check{x}(T,\overline{T})$ para qualquer conjunto T de nós.

Outra relação fundamental entre um fluxo x e o correspondente pseudofluxo $\check{x}$ numa rede simétrica com função-capacidade u:

$$x \text{ respeita } u \text{ se e só se } \check{x} \text{ respeita } u. \tag{12.2}$$

(Veja exercício 12.2.)

Agora considere a transformação de um pseudofluxo em um fluxo. Dado qualquer pseudofluxo $\check{x}$ em uma rede simétrica (N,A) o correspondente fluxo x é definido por

$$x_{ij} := \max\{0, \check{x}_{ij}\}$$

para cada arco ij. É evidente que x é normalizado. Eis um pequeno algoritmo que faz a transformação, em tempo $O(m)$:

[4] No lugar de pseudofluxo, AMO usa o conceito de capacidade residual. Esse conceito está na seção 6.3, p.177. Também é discutido sob "Working with Residual Networks", na seção 2.4, p.44, do AMO.

```
FLUXO (x̆)
1  para cada ij em A faça
2      se x̆_ij ≥ 0
3          então x_ij ← x̆_ij
4          senão x_ij ← 0
5  devolva x
```

É fácil verificar que para qualquer peudofluxo $\breve{x}$ em um grafo simétrico, PSEUDOFLUXO (FLUXO ($\breve{x}$)) $= \breve{x}$.

12.4 Caminhos de incremento

Suponha que x é um fluxo que respeita u numa rede simétrica (N, A, u). Seja $\breve{x}$ o correspondente pseudofluxo. Seja $A_{\breve{x}}$ o conjunto dos arcos ij que têm $\breve{x}_{ij} < u_{ij}$ não nulo:

$$A_{\breve{x}} := \{ij \in A : \breve{x}_{ij} < u_{ij}\}\,.$$

Diremos $(N, A_{\breve{x}})$ é o **grafo residual** (= *residual graph*) correspondente a x.

Numa rede simétrica, os pseudo-caminhos de incremento introduzidos na prova do teorema 11.4 são substituídos por caminhos de incremento (= *augmenting paths*). Um **caminho de incremento** é qualquer caminho (dirigido) no grafo residual.

No contexto de (s, t)-fluxos, estamos particularmente interessados em caminhos de incremento que começam em s e, mais ainda, nos que começam em s e terminam em t. Tanto assim que, às vezes, o termo "caminho de incremento" supõe implicitamente que o caminho começa em s e até mesmo que termina em t.

Exercícios

12.1 [Importante] Suponha que x é um fluxo e $\breve{x}$ o correspondente pseudofluxo num grafo simétrico. Mostre que $x(\overline{T}, T) - x(T, \overline{T}) = \breve{x}(\overline{T}, T)$ para qualquer conjunto T de nós.

12.2 [Importante] Suponha que x é um fluxo e $\breve{x}$ o correspondente pseudofluxo numa rede simétrica (N, A, u). Mostre que x respeita u se e só se $\breve{x}$ respeita u.

12.3 Se $\breve{x}$ é um pseudofluxo, é verdade que PSEUDOFLUXO (FLUXO ($\breve{x}$)) $= \breve{x}$? Se x é um fluxo, é verdade que FLUXO (PSEUDOFLUXO (x)) $= x$?

12.4 Seja $\breve{x}$ um pseudofluxo numa rede simétrica. Considere arcos ij e ji. Se $\breve{x}_{ij} < u_{ij}$ e $\breve{x}_{ji} = u_{ji}$, o que posso concluir sobre o valor de $x :=$ FLUXO ($\breve{x}$) em ij e ji?

Capítulo 13

Algoritmo de Ford-Fulkerson

Este capítulo discute o célebre algoritmo de Ford e Fulkerson (ou algoritmo dos caminhos de incremento) para o problema do fluxo máximo de um nó dado a outro em uma rede com função-capacidade. Trata-se, simplesmente, da formalização do algoritmo implícito na prova do teorema 11.4. A formalização se restringe a redes simétricas, no espírito do capítulo 12 e usa o conceito de pseudofluxo, evitando assim os pseudo-caminhos usados na prova do teorema 11.4.

13.1 Um esboço do algoritmo

Eis um primeiro esboço do "algoritmo dos caminhos de incremento" de Ford e Fulkerson.[1] Ele supõe que o grafo (N, A) é simétrico e procura, em cada iteração, por um caminho de incremento, ou seja, um caminho de s a t no grafo residual.

```
01  x ← 0
02  x̌ ← PSEUDOFLUXO (x)
03  repita
04      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
05      seja P um caminho de s a t no grafo (N, A_x̌)
06      se tal caminho não existe
07          então devolva FLUXO (x̌)
08      δ ← min {u_ij − x̌_ij : ij é arco de P}     ▷ δ > 0
```

[1] Esse algoritmo aparece na figura 6.12, p.180, seção 6.4, do AMO sob o nome *(Generic) Augmenting Path algorithm*.

```
09      para cada arco ij de P faça
10          se x_ji = 0
11              então x_ij ← x_ij + δ
12              senão x_ji ← x_ji − δ
13      x̌ ← PSEUDOFLUXO (x)
```

No começo de cada iteração, x é um (s,t)-fluxo normalizado que respeita u. No fim do bloco de linhas 09–12, o valor do fluxo x terá aumentado de δ.

As conversões de fluxo em pseudofluxo nas linhas 02 e 13 são, na verdade, desnecessárias: é melhor operar o tempo todo com pseudofluxos e voltar ao fluxo somente no fim do processo iterativo.

```
01  x̌ ← 0
02  repita
03      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
04      seja P um caminho de s a t no grafo (N, A_x̌)
05      se tal caminho não existe
06          então devolva FLUXO (x̌)
07      δ ← min {u_ij − x̌_ij : ij é arco de P}     ▷ δ > 0
08      para cada arco ij de P faça
09          x̌_ij ← x̌_ij + δ
10          x̌_ji ← x̌_ji − δ
```

No começo de cada iteração, $\check{x}$ é o (s,t)-pseudofluxo que respeita u e portanto $x :=$ FLUXO $(\check{x})$ é um (s,t)-fluxo que respeita u.[2]

13.2 Algoritmo de Ford e Fulkerson

O esboço da seção anterior não detalha a maneira de procurar um caminho de s a t no grafo residual. Se detalharmos essa busca, teremos o algoritmo descrito abaixo.[3] Ele recebe uma rede simétrica (N, A, u) e devolve um (s,t)-fluxo x e um conjunto T que separa s de t tais que $\text{val}(x) = u(\overline{T}, T)$.

[2] Veja "the effect of augmentation" na figura 6.14, p.182, de AMO.

[3] Esse é, com pequenas modificações, o algoritmo que consta da figura 6.17, p.185, seção 6.5 do AMO, sob o nome *Labelling algorithm*.

```
FORD-FULKERSON (N, A, u, s, t)     ▷ (N, A) simétrico
01  x̌ ← 0
02  repita
03      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
04      ⟨y, P⟩ ← BUSCA (N, A_x̌, s, t)
05      se y(t) − y(s) ≤ 0
06          então x̌ ← INCREMENTE-FLUXO (x̌, P)
07  até que y(t) − y(s) > 0
08  x ← FLUXO (x̌)
09  T ← {j : y(j) − y(s) > 0}
10  devolva x e T
```

O algoritmo BUSCA na linha 04 já foi descrito na seção 3.3. Ao receber um grafo (N, E) e nós s e t, ele devolve um 0-potencial y (ou seja, uma função y de E em $\{0, 1\}$ tal que $y(j) - y(i) \leq 0$ para todo arco ij); se $y(t) - y(s) \leq 0$ então o algoritmo também devolve um caminho P de s a t.[4]

A rotina INCREMENTE-FLUXO usa o caminho P calculado na linha 04 para produzir um incremento (= *augmentation*) do valor do fluxo. Podemos dizer, informalmente, que ela "envia δ unidades de fluxo ao longo do caminho P".

```
INCREMENTE-FLUXO (x̌, P)
1  δ ← min {u_ij − x̌_ij : ij é arco de P}     ▷ δ > 0
2  para cada arco ij de P faça
3      x̌_ij ← x̌_ij + δ
4      x̌_ji ← x̌_ji − δ
5  devolva x̌
```

O algoritmo faz o que promete? Para entender o funcionamento do algoritmo é preciso entender as invariantes do processo iterativo codificado no bloco de linhas 02–07. No começo de cada iteração,

(i1) $\check{x}$ é inteiro;

(i2) $\check{x}$ é um (s, t)-pseudofluxo;

(i3) $\check{x}$ respeita u (ou seja, $-u_{ji} \leq \check{x}_{ij} \leq u_{ij}$ para cada arco ij).

É fácil verificar essas invariantes. Em particular, no início da linha 06 temos $\check{x}_{ij} < u_{ij}$ para todo arco ij do caminho P. Esse fato, aliado a (i1), garante que INCREMENTE-FLUXO produz um incremento de pelo menos 1 unidade no valor do fluxo $x :=$ FLUXO $(\check{x})$.

[4] Na prática, é melhor trocar a linhas 04–05 por uma adaptação apropriada do código de BUSCA que se encontra na seção 3.3. Com isso, não será necessário calcular $A_{\check{x}}$ explicitamente: basta ignorar os arcos ij que têm $\check{x}_{ij} = u_{ij}$, como já sugerimos na seção 3.5.

Garante também que $\check{x}$ permanece inteiro depois da execução de INCREMENTE-FLUXO e assim (i1) continua válida no início da próxima iteração.

Depois da última iteração, o conjunto T definido na linha 20 separa s de t. Ademais, como y é um 0-potencial em $(N, A_{\check{x}})$, temos $\check{x}_{ij} \geq u_{ij}$ para cada arco ij em $\nabla(\overline{T}, T)$. Em virtude de (i3), $\check{x}_{ij} = u_{ij}$ para cada arco ij em $\nabla(\overline{T}, T)$. Logo, depois da linha 08,

$$x_{ij} = u_{ij} \quad \text{e} \quad x_{ji} = 0$$

para cada $ij \in \nabla(\overline{T}, T)$. Portanto, o valor do fluxo x é (veja lema 11.2) $\text{val}(x) = x(\overline{T}, T) - x(T, \overline{T}) = u(\overline{T}, T)$, e assim, o conjunto T devolvido pelo algoritmo de fato tem a propriedade prometida.

13.3 Consumo de tempo

Não é óbvio que a execução do algoritmo termina depois de um número finito de iterações. Mostraremos em seguida que o número de iterações é de fato finito; mostraremos que não passa de nU, onde $U = \max_{ij \in A} u_{ij}$.

Cada iteração do bloco de linhas 03–06 produz um incremento do valor do fluxo. Quantos incrementos (ou seja, quantas iterações) o algoritmo pode fazer no máximo? De acordo com a invariante (i1), o incremento produzido por INCREMENTE-FLUXO é inteiro; portanto, o valor do fluxo $\text{FLUXO}\,(\check{x})$ aumenta de uma unidade, pelo menos, em cada iteração.

Por outro lado, como nossos grafos não têm arcos paralelos, o lema 11.2 garante que o valor do fluxo não pode passar de $u(\bar{t}, t)$, que por sua vez não passa de nU:

$$\text{val}(x) \leq u(\bar{t}, t) \leq nU\,.$$

Assim, o número de iterações não passa de nU e o consumo total de tempo é $O((n + m)nU)$. Se a rede é conexa, temos $m \geq n - 1$ e portanto podemos dizer que o consumo de tempo do algoritmo é

$$O(nmU)\,.$$

No pior caso, o consumo de tempo é $\Omega(nmU)$.[5] Em suma, o algoritmo FORD-FULKERSON é apenas *pseudo*-polinomial.

13.4 Fluxo máximo e caminho de incremento

A análise do algoritmo FORD-FULKERSON prova o seguinte lema:

[5] Veja "Pathological example", figura 6.18, p.187, de AMO.

Lema 13.1 (dos caminhos de incremento) *Para quaisquer dois nós s e t em uma rede simétrica (N, A, u), um (s,t)-fluxo x é máximo se e só se não existe caminho de incremento de s a t.*

Talvez seja apropriado lembrar que um caminho de incremento é qualquer caminho no grafo residual, ou seja, qualquer caminho cada um de cujos arcos ij tem $\check{x}_{ij} < u_{ij}$.

Exercícios

13.1 [Sutilezas do INCREMENTE-FLUXO] A expressão INCREMENTE-FLUXO $(\check{x}, P)$ pode ter efeitos curiosos se P for um passeio que passa por um arco ij e pelo seu "irmão siamês" ji. Suponha, por exemplo, que $P = \langle i, j \rangle$. Como a expressão INCREMENTE-FLUXO $(\check{x}, P)$ afeta $\check{x}$?

13.2 [Upward/downward critical arcs. AMO 7.12, p.245] Sejam s a t dois nós de uma rede capacitada (N, A, u) (se isso for conveniente, suponha a rede é simétrica).

1. Um arco pq é "crítico-para-cima" (*upward critical*) se o aumento do valor de u_{pq} produz um aumento do valor máximo do (s,t)-fluxo que respeita u. É verdade que toda rede tem um arco crítico-para-cima? Descreva um algoritmo que determine todos os arcos críticos-para-cima em uma rede. (O consumo de tempo do seu algoritmo no pior caso deve ser bem melhor que o da solução de m problemas de fluxo máximo.)
2. Um arco kl é "crítico-para-baixo" (*downward critical*) se a redução do valor de u_{pq} produz uma redução do valor máximo do (s,t)-fluxo que respeita u. O conjunto dos arcos críticos-para-baixo coicide com o conjunto dos arcos críticos-para-cima? Se não for, descreva um algoritmo que determine todos os arcos críticos-para-baixo. Qual o consumo de tempo do seu algoritmo no pior?

13.3 [AMO 7.10, p.245] Sejam s e t dois nós de uma rede capacitada simétrica (N, A, u). Sejam α e K dois números inteiros positivos e suponha que a capacidade de cada arco pertence ao conjunto $\{\alpha, 2\alpha, 3\alpha, \ldots, K\alpha\}$. Mostre que o consumo de tempo do algoritmo FORD-FULKERSON para esse tipo de rede é $O(m^2 K)$.

13.4 [Capacidades unitárias] Qual o consumo de tempo do algoritmo FORD-FULKERSON se a função-capacidade é constante?

Capítulo 14

Fluxo: capacity-scaling

O algoritmo FORD-FULKERSON é apenas *pseudo*-polinomial. O presente capítulo[1] elabora modifica aquele algoritmo de modo a torná-lo polinomial, ainda que apenas *fracamente* polinomial. A idéia é dar prioridade aos *grandes* incrementos de fluxo.

14.1 Grandes incrementos de fluxo

Eis um esboço de um algoritmo[2] que resolve o problema do fluxo máximo dando preferência aos grandes incrementos de fluxo. Continuamos supondo, com base no fato 12.1, que nosso grafo é simétrico.

```
CAPACITY-SCALING (N, A, u, s, t)      ▷ (N, A) simétrico
01  U ← max_{ij∈A} u_ij
02  x̌ ← 0
03  Δ ← 2^⌊lg U⌋
04  enquanto Δ ≥ 1 faça
05      A_x̌ ← {ij ∈ A : x̌_ij + Δ ≤ u_ij}
06      ⟨y, P⟩ ← BUSCA (N, A_x̌, s, t)
07      se y(t) − y(s) ≤ 0
08          então x̌ ← INCREMENTE-FLUXO (x̌, P)
09          senão Δ ← Δ/2
10  T ← {j : y(j) − y(s) > 0}
11  x ← FLUXO (x̌)
12  devolva x e T
```

[1] AMO, secção 7.3, p.211. Também exercício 3.11, p.46, no CCPS. Acho que não tem no CLRS.
[2] Este é o algoritmo que está na figura 7.3, p.212, do AMO.

O algoritmo BUSCA na linha 06 já foi descrito na seção 3.3. Ao receber um grafo (N, E) e nós s e t, ele devolve um 0-potencial y; se $y(t) - y(s) \leq 0$ então o algoritmo também devolve um caminho P de s a t.[3]

14.2 Consumo de tempo

O valor do fluxo FLUXO $(\check{x})$ aumenta, durante a execução da linha 08, de pelo menos Δ unidades. O valor de Δ permanece constante durante a execução do bloco de linhas 05–08. Diremos que cada execução desse bloco de linhas (que envolve várias iterações) é uma **fase** (= *scaling phase*). O fim de cada fase é marcado pela execução da linha 09. É claro que o algoritmo executa não mais que

$$1 + \lfloor \lg U \rfloor$$

fases. Cada fase tem

$$\text{no máximo } 2m \text{ iterações,} \tag{14.1}$$

como mostraremos a seguir. Suponha que estamos no fim de uma fase e portanto prestes a executar a linha 09. Seja $x := \text{FLUXO}\,(\check{x})$. De acordo com o teorema do fluxo máximo e corte mínimo (teorema 11.5), existe T separando s de t tal que $\check{x}_{ij} + \Delta > u_{ij}$ para cada arco ij em $\nabla(\overline{T}, T)$. Logo, $\check{x}(\overline{T}, T) + m\Delta > u(\overline{T}, T)$ no início da linha 09. Se x^* é um fluxo de valor máximo então não é difícil concluir que

$$\text{val}(x^*) \leq \text{val}(x) + (u - \check{x})(\overline{T}, T) < \text{val}(x) + m\Delta = \text{val}(x) + 2m(\Delta/2)$$

no início da linha 09. Cada iteração da próxima fase incrementará o valor de fluxo de $\Delta/2$. Logo. a próxima fase terá menos que $2m$ iterações. Isso prova (14.1). Resumindo: o algoritmo excuta não mais que

$$2m(1 + \lfloor \lg U \rfloor)$$

operações de incremento do fluxo.

O consumo de tempo de cada execução de BUSCA é $O(n + m)$ e o de cada execução de INCREMENTE-FLUXO é $O(n)$. Logo, o consumo de tempo total do algoritmo é $O((n + m)m \lg U)$. Se a rede é conexa então $m \geq n - 1$ e portanto podemos dizer que o algoritmo consome

$$O(m^2 \lg U)$$

unidades de tempo. Assim, o algoritmo é polinomial. ainda que apenas fracamente polinomial. Mas isso já é um avanço em relação ao algoritmo pseudo-polinomial FORD-FULKERSON.

[3] Na prática, é melhor trocar a linhas 06–07 por uma adaptação apropriada do código de BUSCA que se encontra na seção 3.3. Com isso, não será necessário calcular $A_{\check{x}}$ explicitamente: basta ignorar os arcos ij que têm $\check{x}_{ij} + \Delta > u_{ij}$.

Exercícios

14.1 [Capacidades unitárias] Qual o consumo de tempo do algoritmo CAPACITY-SCALING se a função-capacidade é constante?

14.2 [AMO 7.17, p.246] Decreva uma generalização do algoritmo CAPACITY-SCALING em que Δ é dividido por um inteiro $\beta \geq 2$ em cada fse. Inicialmente, $\Delta = \beta^{\lceil \log_\beta U \rceil}$; Analise a relação entre β e o consumo de tempo do seu algoritmo.

Capítulo 15

Fluxo: algoritmo de Edmonds-Karp

Os algoritmos para o problema do fluxo máximo (problema 11.1) estudados nos capítulos anteriores são ineficientes porque o *número de incrementos* de fluxo[1] que executam depende de U (mais precisamente, é proporcional a U ou a $\lg U$ no pior caso). Mas é supreendentemente fácil[2] criar um algoritmo em que o número de incrementos de fluxo não depende de U: basta usar caminhos de incremento que tenham comprimento minimo! A estratégia é simples mas sua justificativa é delicada. Ela foi descoberta por Edmonds e Karp (1972).

15.1 Caminhos de incremento mínimos

Com base no fato 12.1, suporemos que nosso grafo é simétrico. para que possamos operar com pseudofluxos. Eis o algoritmo de Edmonds e Karp (1972), também conhecido como *algoritmo dos caminhos de incremento mínimos* (= *shortest augmenting path algorithm*):

[1] Ou seja, o número chamadas da rotina INCREMENTE-FLUXO.
[2] Veja seções 7.2 e 7.4 de AMO. Veja seção 3.2, p.44, de CCPS.

```
EDMONDS-KARP (N, A, u, s, t)    ▷ (N, A) simétrico
1   x̌ ← 0
2   repita
3       A_x̌ ← {ij : x̌_ij < u_ij}
4       (y, P) ← CAMINHO-MÍNIMO (N, A_x̌, s, t)
5       se y(t) − y(s) < n
6           então x̌ ← INCREMENTE-FLUXO (x̌, P)
7           senão devolva FLUXO (x̌)
```

O algoritmo CAMINHO-MÍNIMO é essencialmente o mesmo que BUSCA-EM-LARGURA do capítulo 5. Ele devolve um 1-potencial $(s,*)$-ótimo (veja seção 5.3) y no grafo $(N, A_{\check{x}})$. Portanto, $y(j) - y(i) \leq 1$ para cada arco ij em $A_{\check{x}}$ e uma das seguinte alternativas vale para cada nó j:

(1) existe um caminho Q de s a j em $(N, A_{\check{x}})$ tal que $|Q| = y(j) - y(s)$ ou
(2) $y(t) - y(s) \geq n$ e não existe caminho de s a t em $(N, A_{\check{x}})$.

Se $y(t) - y(s) < n$ então CAMINHO-MÍNIMO devolve também um caminho P de s a t tal que $y(t) - y(s) = |P|$. É claro que P é um caminho de comprimento mínimo de s a t e

$$y(j) - y(i) = 1 \tag{15.1}$$

para cada arco ij em P.

15.2 Número de iterações

O número de iterações do algoritmo não passa de nm. A demonstração desse fato será feita em três passos, representados pelos três lemas a seguir.

Lema 15.1 *Para cada nó j, a diferença de potencial $y(j) - y(s)$ não decresce de uma iteração para a seguinte.*

DEMONSTRAÇÃO: Considere as variáveis $\check{x}$, y e P no começo de uma iteração. Sejam $\dot{r}$, $\dot{y}$ e $\dot{P}$ os valores dessas variáveis no começo da iteração *anterior*. Vamos mostrar que

$$y(j) - y(s) \geq \dot{y}(j) - \dot{y}(s) \text{ para cada nó } j\,.$$

Suponha por um instante que $y(j) - y(s) < \dot{y}(j) - \dot{y}(s)$ para algum nó j e escolha j de modo que $y(j)$ seja mínimo. Como y é um 1-potencial $(s,*)$-ótimo, existe um caminho Q de s a j em (N, A_r) tal que $|Q| = y(j) - y(s)$. É claro que $j \neq s$ e portanto Q tem um penúltimo nó, digamos i. Como $y(j) - y(i) = 1$, temos $y(i) < y(j)$ e portanto

$y(i) - y(s) \geq \dot{y}(i) - \dot{y}(s)$ em virtude da maneira como j foi escolhido. Em suma,

$$\begin{aligned} \dot{y}(j) - \dot{y}(s) &> y(j) - y(s) \\ &= y(i) + 1 - y(s) \\ &= y(i) - y(s) + 1 \\ &\geq \dot{y}(i) - \dot{y}(s) + 1 \,. \end{aligned}$$

Portanto, a diferença de potencial entre as pontas do arco ij era maior que 1:

$$\dot{y}(j) - \dot{y}(i) > 1 \,. \tag{15.2}$$

Como $\dot{y}$ é um 1-potencial em $(N, A_{\dot{r}})$, concluímos que $ij \notin A_{\dot{r}}$, ou seja, que $\dot{r}_{ij} = 0$. Mas $ij \in A_r$, ou seja $\check{x}_{ij} < u_{ij}$, e portanto (veja a rotina INCREMENTE-FLUXO na linha 6) ji pertence ao caminho de incremento $\dot{P}$. De acordo com (15.1), temos $\dot{y}(i) - \dot{y}(j) = 1$. Mas isso é inconsistente com (15.2). A contradição prova o lema. ∎ (15.1)

O 1-potencial y mede distâncias a partir de s no grafo $(N, A_{\check{x}})$: a distância (em número de arcos) de s a qualquer nó j é $y(j) - y(s)$. Será necessário também medir a distância de qualquer nó até t. Para isso, convém imaginar que no início de cada iteração temos, além de y, um 1-potencial z que é $(*, t)$-ótimo (veja seção 5.4) no grafo $(N, A_{\check{x}})$. É evidente que

$$z(j) - z(i) = 1 = y(j) - y(i) \tag{15.3}$$

para qualquer arco ij de qualquer caminho de s a t que tenha comprimento mínimo (pois o comprimento de tal caminho é igual a $z(t) - z(s)$ e também a $y(t) - y(s)$). Segue daí imediatamente que

$$z(i) - z(s) = y(i) - y(s) \quad \text{e} \quad y(t) - y(i) = z(t) - z(i) \tag{15.4}$$

para qualquer nó i de qualquer caminho mínimo de s a t.

Lema 15.2 *Para cada nó i, a diferença de potencial $z(t) - z(i)$ não decresce de uma iteração para a seguinte.*

DEMONSTRAÇÃO: Análoga à demonstração do lema 15.1. ∎

Os lemas 15.1 e 15.2 têm a seguinte conseqüência: o comprimento do caminho de incremento P em uma iteração é pelo menos tão grande quanto o comprimento do caminho de incremento na iteração anterior. As iterações do algoritmo podem, portanto, ser agrupadas em **fases**: duas iterações pertencem à mesma fase se os correspondentes caminhos de incremento têm o mesmo comprimento. É claro que o número de fases não passa de $n - 1$, uma vez que o comprimento de qualquer caminho é menor que n. 15.1 15.2

Resta estimar o número de iterações em cada fase. Isso equivale a estimar o número de execuções da rotina INCREMENTE-FLUXO em cada fase. Cada execução dessa rotina faz

com que $\check{x}_{ij}$ fique igual a u_{ij} para algum arco ij; dizemos que um tal arco sofre uma **saturação**. Para mostrar que INCREMENTE-FLUXO é executada no máximo m vezes em cada fase, basta mostrar que cada arco sofre no máximo uma saturação durante uma fase.

Lema 15.3 *Em cada fase, cada arco sofre no máximo uma saturação.*

DEMONSTRAÇÃO: Suponha que um arco ij sofre uma saturação durante a iteração corrente. Portanto, $\check{x}_{ij} < u_{ij}$ e ij pertence ao caminho P no início da iteração corrente.

Digamos que a iteração corrente é a γ-ésima e suponha que o arco ij já sofreu uma saturação durante a α-ésima iteração, para algum $\alpha < \gamma$. É evidente então que, para algum β entre α e γ, o arco ji pertenceu ao caminho de incremento da β-ésima iteração. Mostraremos abaixo que β e γ não pertencem à mesma fase. Deduziremos daí imediatamente que α e γ também não pertencem à mesma fase.

Sejam $\dot{y}$, $\dot{z}$ e $\dot{P}$ os valores das variáveis y, z e P no início da β-ésima iteração. Queremos mostrar que $|\dot{P}| < |P|$. Como ji está em $\dot{P}$ e ij está em P, (15.1) garante que $\dot{y}(i) - \dot{y}(j) = 1$ e $y(j) - y(i) = 1$. Segue daí, em virtude de (15.4) e dos lemas 15.1 e 15.2, que (15.1) (15.4) 15.1 15.2

$$\begin{aligned}
|P| &= y(t) - y(s) \\
&= y(t) - y(j) + y(j) - y(i) + y(i) - y(s) \\
&= y(t) - y(j) + 1 + y(i) - y(s) \\
&= z(t) - z(j) + 1 + y(i) - y(s) \\
&\geq \dot{z}(t) - \dot{z}(j) + 1 + \dot{y}(i) - \dot{y}(s) \\
&= \dot{y}(t) - \dot{y}(j) + 1 + \dot{y}(i) - \dot{y}(s) \\
&= \dot{y}(t) + \dot{y}(i) - \dot{y}(j) + 1 - \dot{y}(s) \\
&= \dot{y}(t) + 1 + 1 - \dot{y}(s) \\
&= \dot{y}(t) - \dot{y}(s) + 2 \\
&= |\dot{P}| + 2 \,,
\end{aligned}$$

como queríamos demonstrar. ∎

15.3 Consumo de tempo

O lema 15.3 mostra que a rotina INCREMENTE-FLUXO é executada no máximo m vezes em cada fase e portanto no máximo nm vezes ao longo da execução do algoritmo. Cada execução de INCREMENTE-FLUXO consome $O(n)$ unidades de tempo e cada execução de CAMINHO-MÍNIMO consome $O(n + m)$ unidades de tempo. Logo, o consumo do algoritmo EDMONDS-KARP é 15.3

$$O(n^2 m + nm^2) \,.$$

O algoritmo é, portanto, fortemente polinomial. Se o grafo é conexo então $m \geq n - 1$ e podemos dizer que o consumo de tempo do algoritmo é

$$\mathrm{O}(nm^2)\,.$$

Exercícios

15.1 [AMO 7.3, fig.7.21(a), p.243] Use o algoritmo EDMONDS-KARP para resolver o problema do fluxo máximo descrito na figura 7.21(a), p.243, de AMO. Conte o número de incrementos de fluxo.

15.2 Sejam s a t dois de um grafo (N, A). Seja y um 1-potencial $(s, *)$-ótimo e z um 1-potencial $(*, t)$-ótimo. (a) Suponha que P é um caminho de comprimento mínimo de s a t. Mostre que $z(j) - z(i) = 1 = y(j) - y(i)$ para cada arco ij de P. (b) Suponha que $z(j) - z(i) = 1 = y(j) - y(i)$ para um arco ij do grafo. É verdade que ij pertence a um caminho de s a t que tem comprimento mínimo?

15.3 [Capacidades constantes. AMO 8.4, p.288] Seja (N, A, u) uma rede com função capacidade u constante, digamos $u_{ij} = 999$ para cada arco ij. Dados nós s e t, queremos encontrar um (s, t)-fluxo de valor máximo que respeita u. Descreva informalmente o algoritmo mais eficiente (em termos de notação O) que você conhece para resolver o problema.

15.4 [Grafo bipartido.] Uma **bipartição** de um grafo (N, A) é uma partição (N_1, N_2) de N tal que todo arco tem uma das pontas (inicial ou final) em N_1 e a outra em N_2. Suponha que o algoritmo EDMONDS-KARP é aplicado a um grafo dotado de uma bipartição (N_1, N_2). Mostre que o algoritmo faz no máximo $2n_1m$ incrementos de fluxo, sendo $n := |N_1|$.

Capítulo 16

Fluxo: algoritmo de Dinits

Depois de cada incremento de fluxo, o algoritmo EDMONDS-KARP descarta os resultados da busca em largura que produziu um caminho mínimo de incremento. Em geral, entretanto, outros caminhos mínimos de incremento podem ser obtidos sem fazer uma nova busca em largura (usando o 1-potencial z implicitamente calculado durante a busca).[1] O algoritmo de Dinits explora essa possibilidade e obtém um fluxo máximo em $O(n^2m)$ unidades de tempo.

16.1 Uma rotina auxiliar

O algoritmo de Dinits começa com um 1-potencial $(*,t)$-ótimo.[2] O conceito já foi discutido na seção 5.4 e novamente na seção 15.1, mas vamos repetir as definições uma vez mais. Dado um nó t, um 1-potencial z é $(*,t)$-ótimo se uma das seguintes alternativas vale para cada nó i:

(1) existe algum caminho Q de i a t tal que $|Q| = z(t) - z(i)$ ou
(2) $z(t) - z(i) \geq n$ e não existe caminho de i a t.

(O número $z(i)$ é às vezes chamado **rótulo** (= *label*)[3] do nó i.) É fácil calcular um 1-potencial $(*,t)$-ótimo:

```
POTENCIAL-ÓTIMO-TÉRMINO (N, A, t)
01  para cada i em N faça
02      z(i) ← 0
```

[1] Veja seção 7.5, p.221, de AMO.
[2] Segundo AMO, p.242, esse conceito passou a ser usado no contexto de algoritmos de fluxo máximo a partir de Goldberg [1985].
[3] AMO, p.209, usa a expressão "distance labels" para designar potenciais $(*,t)$-ótimos.

```
03   z(t) ← n
04   L ← ⟨t⟩
05   enquanto L ≠ ⟨⟩ faça
06        retire o primeiro elemento, digamos j, de L
07        para cada ij em Ã(j) faça    ▷ arco ij entra em j
08             se z(i) < z(j) − 1    ▷ z(i) = 0
09                  então z(i) ← z(j) − 1    ▷ z(i) aumenta
10                       acrescente i ao final de L
11   devolva z
```

Na linha 07, $\widetilde{A}(j)$ é o conjunto de todos os arcos que entram em j. É claro que no fim da execução do algoritmo temos

$$0 \le z(t) - z(i) \le n \tag{16.1}$$

para todo nó i. Ademais, $z(t) - z(i) = n$ se e só se não existe caminho de i a t no grafo.

16.2 Algoritmo de Dinits

A versão original do algoritmo de Dinits[4] [1970] usava o conceito de "redes em camadas" (= *layered networks*)[5]. Aqui, vamos representar as camadas por um 1-potencial $(*, t)$-ótimo no grafo residual $(N, A_{\check{x}})$.[6] Esse 1-potencial permite que um caminho de incremento seja encontrado rapidamente (em tempo amortizado $O(n)$).

No início de cada iteração teremos um pseudofluxo $\check{x}$ e um 1-potencial z no grafo $(N, A_{\check{x}})$. Diremos que um arco ij é **justo** se $z(j) - z(i) = 1$.[7]

```
DINITS (N, A, u, s, t)    ▷ (N, A) é simétrico
01   x̌ ← 0
02   A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
03   repita
04        z ← POTENCIAL-ÓTIMO-TÉRMINO (N, A_x̌, t)
05        se z(t) − z(s) ≥ n
06             então x ← FLUXO (x̌)
07                  devolva x
08        N^⊗ ← ∅
09        P ← ⟨s⟩
```

[4] AMO escreve "Dinic".

[5] Cada "camada" é o conjunto dos nós i para os quais $z(i)$ tem um determinado valor, sendo z um potencial. A "rede em camadas" é $(N, A^z_{\check{x}})$, onde $A^z_{\check{x}}$ é o conjunto dos arcos ij tais que $\check{x}_{ij} < u_{ij}$ e $z(j) - z(i) = 1$.

[6] É o que AMO faz nas seções 7.4 e 7.5, p.213–223.

[7] AMO chama *admissible* um arco justo no grafo residual. Veja seção 7.2, p.210.

```
10      i ← s
11      enquanto i ∉ N⊗ faça
12          se i = t
13              então x̌ ← INCREMENTE-FLUXO (x̌, P)
14                      A_x̌ ← {ij : x̌_ij < u_ij}
15                      P ← ⟨s⟩
16                      i ← s
17          se algum ij em A_x̌(i) é justo e j ∉ N⊗
20              então AVANCE (ij)
22              senão RETROCEDA (i)

AVANCE (ij)
23  acrescente j ao final de P
24  i ← j

RETROCEDA (i)
25  N⊗ ← N⊗ ∪ {i}     ▷ nó i está "bloqueado"
26  se i ≠ s
27      então remova o último nó de P
28            i ← último nó de P
```

Eis as principais invariantes do algoritmo DINITS: no começo de cada iteração,

(i1) z é um 1-potencial no grafo $(N, A_{\check{x}})$;

(i2) para cada i em $N^{\otimes}$ não existe caminho de i a t em $(N, A_{\check{x}})$ cujos arcos sejam justos;

(i3) P é um caminho de s a i em $(N, A_{\check{x}})$;

(i4) todos os arcos de P são justos;

(i5) todos os nós de P estão em $N - N^{\otimes}$, exceto talvez se $P = \langle s \rangle$;

(i6) FLUXO $(\check{x})$ é um (s,t)-fluxo e respeita u.

Em virtude da invariante (i2), dizemos que os nós em $N^{\otimes}$ estão "bloqueados". Em conseqüência da invariante (i4), o comprimento do caminho P é $z(t) - z(s)$. Assim, em virtude da invariante (i1), P é um caminho de comprimento mínimo na rede $(N, A_{\check{x}})$. Nesse aspecto, o algoritmo tem o mesmo comportamento que EDMONDS-KARP: os caminhos de incremento têm comprimento mínimo no grafo residual $(N, A_{\check{x}})$. (i2) (i4) (i1)

Na última iteração temos $z(t) - z(s) \geq n$ na linha 05. Como z é um 1-potencial, concluímos que não existe caminho de s a t no grafo $(N, A_{\check{x}})$. Portanto, não existe caminho de incremento e assim o fluxo FLUXO$(\check{x})$ é máximo, como já mostramos ao discutir o algoritmo FORD-FULKERSON.

O algoritmo DINITS poderia devolver um corte de capacidade mínima juntamente com

o fluxo de valor máximo: na linha 06, se T é o conjunto dos nós i tais que $z(t) - z(i) < n$ então $(\overline{T}, T)$ é um tal corte.

16.3 Número de incrementos de fluxo

Digamos que uma **fase** é uma uma execução do bloco de linhas 04–22, que define o processo iterativo externo. Durante cada fase, o 1-potencial z permanece constante.

Lema 16.1 *O número de fases não passa de n.*

DEMONSTRAÇÃO: No fim de cada fase temos $i \in N^{\otimes}$. Nessa ocasião, $i = s$ em virtude de (i5). Portanto, de acordo com (i2), todo caminho de s a t em $(N, A_{\check{x}})$ tem comprimento maior que $z(t) - z(s)$. Segue daí que a próxima excução de POTENCIAL-ÓTIMO-TÉRMINO vai resultar num aumento do valor de $z(t) - z(s)$. Assim, o número total de execuções de POTENCIAL-ÓTIMO-TÉRMINO não passa de n. Portanto, o número de fases também não passa de n. ∎ (i5) (i2)

Lema 16.2 *Em cada fase, a rotina* INCREMENTE-FLUXO *é executada no máximo m vezes.*[8]

DEMONSTRAÇÃO: Cada execução de INCREMENTE-FLUXO satura algum arco ij, ou seja, faz com que $\check{x}_{ij}$ fique igual a u_{ij}. Como ij é justo, o arco inverso ji não é justo e portanto não pode fazer parte de um caminho de incremento em alguma iteração futura da fase corrente. Com isso, $\check{x}_{ij}$ permanece igual a u_{ij} e assim ij não pode voltar a ser usado por um caminho de incremento.

Em suma, cada arco pode ser saturado no máximo uma vez em cada fase. Segue daí imediatamente que o número de incrementos não passa de m. ∎

Lema 16.3 *Em cada fase, a rotina* RETROCEDA *é executada no máximo n vezes.*

DEMONSTRAÇÃO: Cada execução da rotina acrescenta um novo nó a $N^{\otimes}$. Cada nó pode ser acrescentado a $N^{\otimes}$ no máximo uma vez, pois nenhum nó sai de $N^{\otimes}$ até o fim da fase. Portanto, RETROCEDA é executada no máximo uma vez para cada nó. ∎

Lema 16.4 *Em cada fase, a rotina* AVANCE *é executada no máximo nm vezes.*

DEMONSTRAÇÃO: De acordo com o lema 16.2, INCREMENTE-FLUXO é executada no má- 16.2

[8] Lema 7.8 e 7.9, p.217–218, de AMO.

ximo m vezes em cada fase. Portanto, basta mostrar que cada nó pode fazer o papel de j na linha 20 no máximo uma vez entre cada duas ocorrências de INCREMENTE-FLUXO.

Considere, pois, um nó q e suponha que AVANCE é executado sobre algum arco da forma iq. O nó q passa a fazer parte do caminho P. Enquanto estiver em P, o nó q não pode fazer o papel de j na linha 20, essencialmente em virtude de (i1) e (i4). Suponha agora (i1)
que o nó q deixa de fazer parte de P antes que ocorra o próximo INCREMENTE-FLUXO. (i4)
Isso só pode acontecer se RETROCEDA for executado sobre q, ou seja, se a linha 22 for executada com q no papel de i. Nesse caso, q passa a fazer parte de $N^{\otimes}$ e portanto não mais pode fazer o papel de j na linha 20 até o fim da fase. ∎

Como o número de fases não passa de n, os números totais de incrementos de fluxo, de execuções de RETROCEDA e de execuções de AVANCE não passam de nm, n^2 e n^2 respectivamente.

rotina	número máximo de execuções
POTENCIAL-ÓTIMO-TÉRMINO	n
INCREMENTE-FLUXO	$n \cdot m$
RETROCEDA	$n \cdot n$
AVANCE	$n \cdot nm$

16.4 Consumo de tempo do algoritmo

Para que possamos analisar o consumo de tempo do algoritmo, é preciso reescrevê-lo de maneira um pouco mais detalhada, com a introdução do *current-arc data structure*.

```
DINITS (N, A, u, s, t)
01  x̌ ← 0
02  repita
03      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
04      z ← POTENCIAL-ÓTIMO-TÉRMINO (N, A_x̌, t)
05      se z(t) − z(s) ≥ n
06          então devolva FLUXO (x̌)
07      para cada i em N faça
08          A′(i) ← A(i)     ▷ current-arc data structure
09      N^⊗ ← ∅
10      P ← ⟨s⟩
11      i ← s
12      enquanto i ∉ N^⊗ faça
13          se i = t
14              então x̌ ← INCREMENTE-FLUXO (x̌, P)
```

```
15                  P ← ⟨s⟩
16                  i ← s
17          se A′(i) ≠ ∅
18              então escolha um arco ij de A′(i)
19                  se x̌ij < uij e z(j) − z(i) = 1 e j ∉ N⊗
20                      então AVANCE (ij)
21                      senão retire ij de A′(i)
22              senão RETROCEDA (i)
```

Além das invariantes (i1) a (i6), temos a seguinte:

(i7) para cada pq em $A(p) - A'(p)$, $\check{x}_{pq} = u_{pq}$ ou $z(q) - z(p) < 1$ ou $q \in N^{\otimes}$.

(Veja exercício 16.4.) Ela é necessária para mostrar que (i2) continua valendo no início da próxima iteração. (i2)

De acordo com o lema 16.4, o número de execuções da linha 20 não passa de nm em cada fase. Cada arco pode fazer o papel de ij na linha 21 uma só vez em cada fase e portanto a linha 21 é executada no máximo m vezes por fase. Logo, a linha 19, e portanto também a linha 18, será executada no máximo $nm + m$ vezes durante uma fase. 16.4

De acordo com o lema 16.3, a linha 22 será executada no máximo n vezes em cada fase. Portanto, a linha 17 será executada no máximo $nm + m + n$ vezes em cada fase. Segue daí que a linha 13 não pode ser executada mais que $nm + m + n$ vezes em cada fase. 16.3

Assim, se levarmos em conta os lemas 16.1 e 16.2, o consumo total de tempo do algoritmo DINITS é 16.1 16.2

$$O(n^2 m) .$$

(Compare com o consumo de tempo $O(nm^2)$ do algoritmo EDMONDS-KARP.)

linha	consumo de tempo
01	$\mathrm{O}(1)$
02	$\mathrm{O}(n)$
03–04	$n\,\mathrm{O}(n+m)$
05	$n\,\mathrm{O}(1)$
06	$n\,\mathrm{O}(m)$
07–08	$n\,\mathrm{O}(n)$
09–11	$n\,\mathrm{O}(1)$
12–13	$n\,(nm+m+n)\,\mathrm{O}(1)$
14	$n\,m\,\mathrm{O}(n)$
15–16	$n\,m\,\mathrm{O}(1)$
17	$n\,(nm+m+n)\,\mathrm{O}(1)$
18–19	$n\,(nm+m)\,\mathrm{O}(1)$
20	$n\,nm\,\mathrm{O}(1)$
21	$n\,m\,\mathrm{O}(1)$
22	$n\,n\,\mathrm{O}(1)$

Como se vê, a linha 14 é crítica em termos de consumo de tempo. Isso serve de motivação para o algoritmo GENERIC-PREFLOW-PUSH, a ser examinhado no próximo capítulo.

Exercícios

16.1 [Bom. AMO 7.1, fig.7.20, p.243] A figura 7.20, p.243, de AMO especifica uma rede (N, A, u) e um (s,t)-fluxo x. Seja $\check{x}$ o pseudofluxo PSEUDOFLUXO(x). (1) Calcule um 1-potencial $(*,t)$-ótimo z no grafo $(N, A_{\check{x}})$, onde $A_{\check{x}}$ o conjunto dos arcos ij com $\check{x}_{ij} < u_{ij}$. (2) Modifique x, sem alterar val(x), de modo que $z(s)$ aumente em uma unidade. (3) Modifique x, sem alterar val(x), de modo que $z(s)$ diminua em uma unidade.

16.2 [AMO 7.9(g), p.244] Seja t um nó de um grafo (N, A) e z uma função de N em $\mathbb{Z}$ dotada da seguinte propriedade: para cada nó i e qualquer caminho P de i a t, tem-se $|P| \geq z(t) - z(i)$. É verdade que z é um 1-potencial (ou seja, que $z(j) - z(i) \leq 1$ para cada arco ij)?

16.3 Prove as invariantes (i1) a (i6) do algoritmo DINITS.

16.4 [Property 7.7, p.217, de AMO] Prove a invariante (i7).

16.5 Suponha que a rotina POTENCIAL-ÓTIMO-TÉRMINO na linha 04 do algoritmo DINITS seja trocada por outra que devolve uma função z de N em $\mathbb{Z}$ tal que

$$z(t) - z(i) = \begin{array}{ll} 0 & \text{se existe caminho de } i \text{ a } t \text{ no grafo } (N, A_{\check{x}}) \\ n & \text{caso contrário.} \end{array}$$

O algoritmo continua correto? Qual o efeito sobre o consumo assintótico de tempo?

Capítulo 17

Preflow-push: algoritmo básico

Em todos os algoritmos para o problema do fluxo máximo (problema 11.1) vistos até aqui, cada iteração começa com um (s,t)-fluxo. O ponto crítico desses algoritmos está no consumo de tempo de cada incremento de fluxo (a linha 14 do algoritmo DINITS descrito na seção 16.4 mostra isso claramente).

Para obter algoritmos mais eficientes, é preciso desistir da idéia de manter um (s,t)-fluxo em cada iteração e contentar-se com o conceito mais fraco de pré-fluxo.[1]

17.1 Pré-fluxo

Suponha que s é um nó de uma rede (N, A, u) com função-capacidade u. Um fluxo x é um **pré-fluxo** (= *preflow*) **com fonte** s se tem excesso não-negativo em cada nó distinto de s, ou seja, se

$$x(\bar{i}, i) \geq x(i, \bar{i})$$

para todo i distinto de s. Em virtude de (10.1), $x(\bar{s}, s) \leq x(s, \bar{s})$.

Eis uma conseqüência simples do lema 10.1:

Lema 17.1 *Para qualquer pré-fluxo x com fonte s, qualquer nó t e qualquer conjunto T que separa s de t, se $x(\bar{t}, t) > x(t, \bar{t})$ então $x(\overline{T}, T) > 0$.*

DEMONSTRAÇÃO: $x(\overline{T}, T) \geq x(\overline{T}, T) - x(T, \overline{T}) = \sum_{j \in T}(x(\bar{j}, j) - x(j, \bar{j})) \geq x(\bar{t}, t) - x(t, \bar{t}) > 0$. ∎

[1] A figura 7.10, p.224, de AMO dá uma boa motivação para os algoritmos do presente capítulo.

17.2 Algoritmo preflow-push básico

O seguinte algoritmo para o problema do fluxo máximo pode ser atribuído a Karzanov [1974], Shiloach–Vishkin [1982] e Goldberg–Tarjan [1986]. O algoritmo recebe nós s e t de uma rede (N, A, u) e produz um fluxo de valor máximo de s a t dentre os que respeitam u. Como de hábito, o algoritmo supõe que o grafo (N, A) é simétrico para que seja possível operar com pseudofluxos.

O algoritmo começa com um pré-processamento que inunda com fluxo os arcos que saem de s, calcula o excesso $e(j)$ em cada nó j e em seguida calcula um 1-potencial $(*, t)$-ótimo.

```
PRÉ-PROCESSAMENTO ()
01  x̌ ← 0
02  e ← 0
03  para cada sj em A(s) faça
04      x̌_sj ← u_sj
05      x̌_js ← −u_sj
06      e(j) ← e(j) + u_sj
07      e(s) ← e(s) − u_sj
08  A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
09  z ← POTENCIAL-ÓTIMO-TÉRMINO (N, A_x̌, t)     ▷ z(t) − z(s) = n
```

No início de cada iteração temos um pré-fluxo $x := \text{FLUXO}\,(\check{x})$. Diremos que um nó é **ativo** se tiver excesso estritamente positivo. Se t é o único nó ativo então x é um fluxo de s a t.

```
GENERIC-PREFLOW-PUSH[2] (N, A, u, s, t)     ▷ (N, A) é simétrico
00  PRÉ-PROCESSAMENTO ()
10  enquanto e(i) > 0 para algum i em N − {t} faça     ▷ i é ativo
11      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
12      se algum ij em A_x̌(i) é justo
13          então PUSH (ij)
14          senão RELABEL (i)
15  x ← FLUXO (x̌)
16  devolva x
```

Em cada iteração, o algoritmo escolhe um nó ativo i e procura "empurrar" o excesso de x em i "em direção a" t (ou "em direção a" s, no caso das últimas iterações). A "direção" é determinado pelo 1-potencial z (que na primeira iteração é $(*, t)$-ótimo).

[2] O algoritmo está nas figuras 7.11 e 7.12, p.225, de AMO. O nome do algoritmo poderia ser traduzido como Empurre-Pré-Fluxo.

Mais precisamente, o excesso em i é "empurrado" ao longo de um arco **justo**, ou seja, um arco ij tal que $z(j) - z(i) = 1$.

PUSH[3] (ij)
17 $\quad \delta \leftarrow \min\{e(i), u_{ij} - \check{x}_{ij}\}$ $\quad \triangleright\ \delta > 0$
18 $\quad \check{x}_{ij} \leftarrow \check{x}_{ij} + \delta$
19 $\quad \check{x}_{ji} \leftarrow \check{x}_{ji} - \delta$
20 $\quad e(i) \leftarrow e(i) - \delta$
21 $\quad e(j) \leftarrow e(j) + \delta$

RELABEL[4] (i)
22 $\quad z(i) \leftarrow \max\{z(j)-1 : ij \in A_{\check{x}}(i)\}$ $\quad \triangleright\ z(i)$ decresce

No início de cada iteração do bloco de linhas 11–14, seja $x := \text{FLUXO}(\check{x})$. Então

(i1) x é pré-fluxo com fonte s;
(i2) $e(i) = x(\bar{i}, i) - x(i, \bar{i})$ para cada i em N;
(i3) x respeita u;
(i4) z é um 1-potencial em $(N, A_{\check{x}})$;
(i5) $z(t) - z(s) = n$;
(i6) $z(t) - z(i) \geq 0$ para todo i;
(i7) se $e(p) > 0$ então existe um caminho de p a s em $(N, A_{\check{x}})$.[5]

Prove essas invariantes!

Uma conseqüência imediata da invariante (i7): se $e(j) > 0$ então o conjunto $\{z(j)-1 : ij \in A_{\check{x}}(i)\}$ na linha 22 não é vazio, e portanto $z(i)$ está bem definido depois da execução dessa linha. (i7)

As invariantes (i4), (i5) e (i7) têm a seguinte conseqüência, fundamental para a estimativa do consumo de tempo do algoritmo:[6] no início de cada iteração, (i4) (i5) (i7)

$$z(t) - z(p) < 2n \text{ para cada nó } p. \tag{17.1}$$

Eis um esboço da prova: $z(t) - z(p) = z(t) - z(s) + z(s) - z(p) = n + z(s) - z(p) < n + n$.

No início da última iteração temos $e(i) = 0$ para todo i em $N - \{s, t\}$. Portanto, x é um fluxo de s a t e $\text{val}(x) = e(t)$. Como z é um 1-potencial em $(N, A_{\check{x}})$ e $z(t)-z(s) \geq n$, não existe caminho de incremento para o fluxo x que comece em s e termine em t. Portanto, x é um fluxo de valor máximo.

[3] Veja figura 7.11, p.225, AMO.
[4] Veja figura 7.11, p.225, AMO.
[5] Atenção: eu disse "de p a s" e não o contrário. Esse é o lema 7.11, p.227, de AMO.
[6] Trata-se do lema 7.12, p.228, de AMO.

Número de iterações. Não é evidente que a execução do algoritmo termina depois de um número finito de iterações. Para verificar esse fato, observe a evolução das somas

$$\sum_{p\neq s}(z(t)-z(p)) \quad \text{e} \quad \sum_{p\neq s} e(p)(z(t)-z(p))\,. \tag{17.2}$$

Em virtude de (17.1), temos $z(t)-z(p) < 2n$, donde o valor da primeira soma não passa de

$$2n^2$$

e o valor da segunda não passa de

$$2n\sum_{p\neq s} e(p) = 2nE\,,$$

onde E é o "excesso total" $\sum_{sj\in A(s)} u_{sj}$. Por outro lado, em virtude de (i1), (i2) e (i6), o valor da segunda soma nunca é negativo: $\sum_{p\neq s} e(p)(z(t)-z(p)) \geq 0$.

Agora observe como o valor das duas somas varia em cada iteração. A cada execução de RELABEL (linha 14), o valor da primeira soma aumenta. E a cada execução de PUSH (linha 13), o valor da primeira soma não se altera mas o valor da segunda diminui (pois $z(t)-z(j)$ é menor que $z(t)-z(i)$ na linha 13). Assim, o número total de iterações não passa de

$$2n^2 \cdot 2nE\,.$$

Na próxima seção, cálculo bem mais delicado mostrará que o número de iterações não passa de $2n^2+nm+2n^2m$.

17.3 Número de relabels e pushes

Em cada iteração do bloco de linhas 11–14 ocorre uma execução de PUSH ou uma execução de RELABEL.

Lema 17.2 *A rotina* RELABEL *é executada menos que* $2n$ *vezes para cada nó.*

DEMONSTRAÇÃO: Seja p um nó qualquer e considere as execuções de RELABEL com p no papel de i. A cada execução, o valor de $z(p)$ descresce de pelo menos uma unidade. Em virtude de (i6), temos $z(p) \leq z(t)$ antes da primeira execução. Em virtude de (17.1), temos $z(p) > z(t) - 2n$ depois da última execução. Logo, o número de execuções de RELABEL com p no papel de i é menor que $2n$. ∎ (i6) (17.1)

Um arco ij sofre uma **saturação** quando $\check{x}_{ij}$ fica igual a u_{ij}. Diremos que uma execução de PUSH é **saturante** (*saturating*) se $\check{x}_{ij} = u_{ij}$ depois da linha 18 e **não-saturante** (*non-saturating*) em caso contrário.

Lema 17.3 [7] *O número de execuções saturantes de* PUSH *não passa de* nm.

[7] Lema 7.8, p.217, de AMO.

DEMONSTRAÇÃO: Seja pq um arco qualquer e considere as execuções saturantes de PUSH com pq no papel de ij. Toda vez que pq sofre uma saturação, pq é justo, ou seja, $z(q) - z(p) = 1$.

Depois da saturação, pq sai de $A_{\check{x}}$. Antes que pq volte a $A_{\check{x}}$, é preciso que a rotina PUSH seja executada sobre o arco inverso qp. Para que isso aconteça, é preciso que o arco qp se torne justo. Para isso é necessário que o valor de $z(q)$ decresça e, mais tarde, que o valor de $z(p)$ decresça. As alterações do potencial z somente ocorrem na rotina RELABEL. Como RELABEL é executada menos que $2n$ vezes com q no papel de i e menos que $2n$ vezes com p no papel de i, o número de execuções saturantes de PUSH com pq no papel de ij deve ser menor que n. ∎

Como temos n nós, o número total de execuções de RELABEL é limitado por $2n^2$. Como temos m arcos, o número total de execuções saturantes de PUSH não passa de nm.

Lema 17.4 *O número total de execuções não-saturantes de* PUSH *não passa de* $2n^2(m + 2)$.

DEMONSTRAÇÃO: Vamos recorrer a uma técnica de análise amortizada. Antes de cada execução do bloco de linhas 11–14, digamos que a **carga** de um nó p é o número

$$\chi(p) = \begin{cases} z(t) - z(p) & \text{se } e(p) > 0 \\ 0 & \text{em caso contrário.} \end{cases}$$

Digamos que a **carga**[8] da rede é a soma das cargas dos nós: $\sum_{p \in N} \chi(p)$. É claro que esse número é não-negativo.

Digamos que $\mathcal{R}$ é o conjunto dos naturais ι tais que um RELABEL ocorre durante a ι-ésima iteração. Analogamente, seja $\mathcal{S}$ o conjunto das iterações em que ocorre um PUSH saturante e $\mathcal{P}$ o conjunto das iterações em que ocorre um PUSH não-saturante.

Digamos que Δ_ι é a variação da carga da rede durante a ι-ésima iteração, ou seja, a diferença entre a carga no fim e a carga no início da ι-ésima iteração. Podemos dizer que a variação total da carga da rede entre a primeira e a última iterações é

$$\textstyle\sum_{\iota \in \mathcal{R}} \Delta_\iota + \sum_{\iota \in \mathcal{S}} \Delta_\iota + \sum_{\iota \in \mathcal{P}} \Delta_\iota \, .$$

Em virtude de (17.1), a carga da rede no início da primeira iteração é menor que $2n^2$, e portanto a carga da rede no fim da última iteração será menor que (17.1)

$$2n^2 + \textstyle\sum_{\iota \in \mathcal{R}} \Delta_\iota + \sum_{\iota \in \mathcal{S}} \Delta_\iota + \sum_{\iota \in \mathcal{P}} \Delta_\iota \, .$$

Agora considere o efeito de uma execução das rotinas PUSH e RELABEL sobre a carga da rede:

[8] O termo mais usual para esse conceito é "potencial" e não "carga". Mas esse "potencial" não deve ser confundido com o potencial z.

A cada RELABEL, a carga de i aumenta e a carga dos demais nós não se altera. Como $0 \le \chi(i) = z(t) - z(i) < 2n$ em virtude de (i6) e de (17.1), temos

$$\textstyle\sum_{\iota \in \mathcal{R}_i} \Delta_\iota < 2n \,.$$

Durante um PUSH saturante, somente as cargas dos nós i e j se alteram. A carga de i pode se tornar nula (se i deixar de ser ativo) e a carga de j pode se tornar positiva (se j se tornar ativo e for diferente de j). De qualquer forma, em virtude de (17.1) a carga total da rede aumenta em no máximo $2n$ unidades. Portanto, se $\iota \in \mathcal{S}$ então (17.1)

$$\Delta_\iota < 2n \,.$$

A cada PUSH não-saturante, a carga do nó i é reduzida de $z(t) - z(i)$ a 0. A carga do nó j pode aumentar para $z(t) - z(j)$ (se o nó j estava inativo no início da iteração) e as cargas dos demais nós permanecem inalteradas. Portanto, se $\iota \in \mathcal{P}$ então

$$\Delta_\iota \le (z(t) - z(j)) - (z(t) - z(i)) = -(z(j) - z(i)) = -1 \,.$$

O lema 17.2 garante que $|\mathcal{R}_p| < 2n$ para cada p, sendo $\mathcal{R}_p$ o conjunto das iterações em que ocorre RELABEL com p no papel de i. O lema 17.3 garante que $|\mathcal{S}| < nm$. Logo, no início da última iteração a carga da rede será menor que 17.2 17.3

$$\begin{aligned}
&2n^2 + \textstyle\sum_{\iota \in \mathcal{R}} \Delta_\iota + \sum_{\iota \in \mathcal{S}} \Delta_\iota + \sum_{\iota \in \mathcal{P}} \Delta_\iota \\
&= \; 2n^2 + \textstyle\sum_p \sum_{\iota \in \mathcal{R}_p} \Delta_\iota + \sum_{\iota \in \mathcal{S}} \Delta_\iota + \sum_{\iota \in \mathcal{P}} \Delta_\iota \\
&< \; 2n^2 + \textstyle\sum_p 2n + 2n|\mathcal{S}| - |\mathcal{P}| \\
&< \; 2n^2 + 2n^2 + 2n^2 m - |\mathcal{P}| \\
&= \; 2n^2(m+2) - |\mathcal{P}| \,.
\end{aligned}$$

Como a carga da rede nunca é negativa, temos $|\mathcal{P}| < 2n^2(m+2)$ como queríamos demonstrar. ∎

Podemos supor que o número de execuções não-saturantes de PUSH não passa de $2n^2 m$, uma vez que estamos interessados apenas na análise assintótica. *No sentido amortizado*, pode-se dizer que o número de execução não-saturantes de PUSH sobre cada arco é menor que $2n^2$.

rotina	número máximo de execuções
RELABEL	$2n^2$
PUSH saturante	nm
PUSH não-saturante	$2n^2 m$

17.4 Consumo de tempo do algoritmo

Para analisar o consumo de tempo do algoritmo, convém reescrevê-lo de maneira um pouco mais detalhada, mostrando como administrar o conjunto de nós ativos e como implementar as linhas 10 e 12 de maneira eficiente.

GENERIC-PREFLOW-PUSH (N, A, u, s, t) $\triangleright$ (N, A) é simétrico
00 PRÉ-PROCESSAMENTO ()
30 $N^\star \leftarrow \emptyset$ $\triangleright$ nós ativos
31 para cada j em N faça
32 se $e(j) > 0$
33 então $N^\star \leftarrow N^\star \cup \{j\}$
34 para cada i em N faça
35 $A'(i) \leftarrow A(i)$
36 enquanto $N^\star - \{t\} \neq \emptyset$ faça
37 escolha um nó i de $N^\star - \{t\}$
38 se $A'(i) \neq \emptyset$
39 então escolha ij em $A'(i)$
40 se $\check{x}_{ij} < u_{ij}$ e $z(j) - z(i) = 1$
41 então PUSH (ij)
42 senão retire ij de $A'(i)$
43 senão RELABEL (i)
44 $A'(i) \leftarrow A(i)$
45 devolva FLUXO $(\check{x})$

PUSH (ij)
46 $\delta \leftarrow \min\{e(i), u_{ij} - \check{x}_{ij}\}$ $\triangleright$ $\delta > 0$
47 $\check{x}_{ij} \leftarrow \check{x}_{ij} + \delta$
48 $\check{x}_{ji} \leftarrow \check{x}_{ji} - \delta$
49 $e(i) \leftarrow e(i) - \delta$
50 se $e(i) = 0$
51 então $N^\star \leftarrow N^\star - \{i\}$
52 $e(j) \leftarrow e(j) + \delta$
53 se $e(j) > 0$ $\triangleright$ $j \neq s$
54 então $N^\star \leftarrow N^\star \cup \{j\}$

RELABEL (i)
55 $z(i) \leftarrow -\infty$
56 para cada ij em $A(i)$ faça
57 se $\check{x}_{ij} < u_{ij}$ e $z(i) < z(j) - 1$

58 então $z(i) \leftarrow z(j) - 1$

Além das invariantes (i1) a (i7), valem também as seguintes:

(i8) para todo nó p, $e(p) > 0$ se e só se $p \in N^\star$;
(i9) para cada arco pq em $A(p) - A'(p)$, $\check{x}_{pq} = u_{pq}$ ou $z(q) - z(p) < 1$.

A invariante (i8) é óbvia. Já a prova de (i9) exige algum esforço (veja exercício 17.5).

Consumo de tempo. Para cada nó p, de acordo com o lema 17.2, a rotina RELABEL na linha 43 é executada menos que $2n$ vezes com p no papel de i. Cada uma dessas execuções consome $O(|A(p)|)$ unidades de tempo. Logo, o consumo total de tempo de RELABEL é 17.2

$$2n\,O(\textstyle\sum_p |A(p)|) = 2n\,O(m)\,.$$

Cada nó p faz o papel de i nas linhas 43 e 44 menos que $2n$ vezes. Portanto, o conjunto de arcos $A(p)$ de cada nó p é examinado menos que $2n$ vezes. Segue daí que o número de execuções da linha 42 é limitado por $2n \sum_p |A(p)| = 2nm$. O consumo de tempo total da linha 42 é, portanto, $2nm\,O(1)$.

De acordo com os lemas 17.3 e 17.4, o número total de execuções da linha 41 (rotina PUSH) é menor que $nm + 2n^2m$. Cada execução da rotina consome $O(1)$ unidades de tempo. Logo, o consumo total de tempo da linha 41 é $(nm + 2n^2m)\,O(1)$. 17.3 17.4

O número de execuções das linhas 39–40 não é maior que a soma dos números de execuções das linhas 41 e 42, ou seja, $nm + 2n^2m + 2nm = 3nm + 2n^2m$. O número de execuções da linha 38 não é maior que o número de execuções da linha 39 mais o número de execuções da linha 43, ou seja, não é maior que $3nm + 2n^2m + 2n^2$.

Conclusão final: o algoritmo GENERIC-PREFLOW-PUSH consome

$$O(n^2m)$$

unidades de tempo.

rotina	linha	consumo de tempo
PRÉ-PROCESSAMENTO	01–07	$nO(1)$
PRÉ-PROCESSAMENTO	08–09	$O(n+m)$
	30–33	$n\,O(1)$
	34–35	$n\,O(1)$
	36–38	$(3nm + 2n^2m + 2n^2)\,O(1)$
	39–40	$(3nm + 2n^2m)\,O(1)$
PUSH	41	$(nm + 2n^2m)\,O(1)$
	42	$2nm\,O(1)$
RELABEL	43	$2n\,O(m)$
	44	$2n^2\,O(1)$
	45	$O(m)$

Exercícios

17.1 Suponha que x é um pré-fluxo com origem s e que $x(\bar{t}, t) - x(t, \bar{t}) > 0$. Prove que existe um caminho de s a t tal que $x_{ij} > 0$ para cada arco ij do caminho.

17.2 [AMO 7.4, fig.7.22, p.243] Use o algoritmo GENERIC-PREFLOW-PUSH para resolver o problema de fluxo máximo descrito na figura 7.22, p.243, de AMO. Ao executar o algoritmo, considere os nós em ordem crescente de número; use essa ordem para escolher o próximo nó ativo; para percorrer o conjunto $A(i)$, use a ordem crescente da ponta final dos arcos. Exiba o estado da rede no início de cada iteração. Conte o número de pushes saturantes, o número de pushes não-saturantes e o número de relabels.

17.3 [AMO 7.14, fig.7.23, p.245] Aplique o algoritmo GENERIC-PREFLOW-PUSH ao grafo da figura 7.23, p.245, de AMO. Ao executar o algoritmo, dê preferência aos nós com maior z e resolva empates em favor de nós com menor número. Conte o número de pushes saturantes, o número de pushes não-saturantes e o número de relabels.

17.4 Prove as invariantes do algoritmo GENERIC-PREFLOW-PUSH.

17.5 [Property 7.7, p.217, de AMO] Prove a invariante (i9).

17.6 [AMO 7.9(c), p.244] Seja v^* o valor de um fluxo de valor máximo de s a t em uma rede capacitada (N, A, u). Prove ou desprove a seguinte afirmação: no início de qualquer iteração do algoritmo GENERIC-PREFLOW-PUSH tem-se $v^* - x(\bar{t}, t) \leq \sum_{i \neq s,\, i \neq t} e(i)$.

17.7 [Conversão de pré-fluxo máximo em fluxo máximo. AMO 7.11, p.245] Sejam s e t dois nós de uma rede capacitada simétrica (N, A, u) e seja x um pré-fluxo com origem s. Diremos que o **valor** de x é o número $\text{val}(x) = x(\bar{t}, t) - x(t, \bar{t})$. Diremos que x é **máximo** se seu valor é máximo (dentre os pré-fluxos com origem s que respeitam u).

a. Suponha que x é um pré-fluxo máximo. Mostre que existe um fluxo máximo x^* de s a t que respeita x (ou seja, $x^*_{ij} \leq x_{ij}$ para cada arco ij) e satisfaz $\text{val}(x^*) = \text{val}(x)$. (Sugestão: Use decomposição de fluxo, seção 10.3.)

b. Esboce um algoritmo que converta um pré-fluxo máximo em um fluxo máximo fazendo não mais que $n + m$ incrementos.

c. Esboce um algoritmo que use caminhos de incremento de comprimento mínimo (veja seção 15) para converter um pré-fluxo máximo em um fluxo máximo. O seu algoritmo deve consumir $O(nm)$ unidades de tempo. (Sugestão: Calcule um 1-potencial $(s, *)$-ótimo e mostre que o algoritmo produzirá no máximo m saturações de arcos.)

17.8 Suponha que a rotina POTENCIAL-ÓTIMO-TÉRMINO na linha 09 do PRÉ-PROCESSAMENTO seja trocada por outra que devolve um 1-potencial z constante (ou seja, uma função z de N em $\mathbb{Z}$ tal que $z(i) = z(t)$ para todo nó i). O algoritmo continua correto? Qual o efeito sobre o consumo assintótico de tempo?

17.9 [AMO 7.22, p.247] Considere as políticas descritas abaixo para escolher nós ativos na linha 10 do algoritmo GENERIC-PREFLOW-PUSH. Descreva os detalhes de implementação em cada caso. Dê uma delimitação para o número de execuções de PUSH e RELABEL em cada caso.

1. Escolhe um nó ativo i que tenha o maior $z(i)$.
2. Escolhe um nó ativo i que tenha maior $e(i)$.
3. Escolhe um nó ativo i que tenha sido usado mais recentemente.
4. Escolhe um nó ativo i que tenha sido usado menos recentemente.

17.10 [Bom! AMO 8.9, p.289] Dê uma boa delimitação assintótica do consumo de tempo do algoritmo GENERIC-PREFLOW-PUSH quando restrito a redes com capacidades unitárias ($u_{ij} = 1$ para cada ij)?

Capítulo 18

Preflow-push: implementação FIFO

Este capítulo trata de uma implementação do algoritmo GENERIC-PREFLOW-PUSH em que o conjunto $N^\star$ de vértices ativos é tratado como uma fila.

18.1 Algoritmo FIFO Preflow-push

O algoritmo abaixo é uma implementação do GENERIC-PREFLOW-PUSH em que $N^\star$ é organizado em uma fila. Ele pode ser atribuído a Goldberg [1985], que baseou o seu trabalho no de Shiloach–Vishkin [1982].

```
FIFO-PREFLOW-PUSH (N, A, u, s, t)     ▷ (N, A) é simétrico
00  PRÉ-PROCESSAMENTO ()
10  L ← ⟨⟩
11  para cada j em N faça
12      se e(j) > 0 e j ≠ t
13          então acrescente j ao final de L
14  enquanto L ≠ ⟨⟩ faça
15      seja i o primeiro nó em L
16      retire i de L
17      NODE-EXAMINATION (i)
18  devolva FLUXO (x̌)
```

```
NODE-EXAMINATION (i)
19  A' ← A(i)
20  enquanto e(i) > 0 e A' ≠ ∅ faça
21      retire um arco ij de A'
22      se x̌_ij < u_ij e z(j) − z(i) = 1
23          então PUSH (ij)
24  se e(i) > 0
25      então RELABEL (i)

PUSH (ij)
26  δ ← min {e(i), u_ij − x̌_ij}    ▷ δ > 0
27  x̌_ij ← x̌_ij + δ
28  x̌_ji ← x̌_ji − δ
29  e(i) ← e(i) − δ
30  e(j) ← e(j) + δ
31  se e(j) > 0 e j ≠ t    ▷ j ≠ s
32      então acrescente j ao final de L

RELABEL (i)
33  z(i) ← −∞
34  para cada ij em A(i) faça
35      se x̌_ij < u_ij e z(i) < z(j) − 1
36          então z(i) ← z(j) − 1
37  acrescente i ao final de L
```

Invariantes: além dos invariantes (i1) a (i7) do algoritmo GENERIC-PREFLOW-PUSH valem também os seguintes

(i8) os nós na seqüência L são distintos dois a dois;

(i9) para todo p em $N - \{t\}$, $e(p) > 0$ se e só se p está em L.

18.2 Consumo de tempo

A análise do consumo de tempo é um refinamento da análise de GENERIC-PREFLOW-PUSH que fizemos na seções 17.3 e 17.4. A análise depende do conceito de fase: uma **fase** é uma seqüência de iterações que tratados nós que estão em L no fim da fase anterior. Para tornar esse conceito mais claro, convém reescrever o algoritmo de modo que L seja a concatenação de duas seqüências, L_1 e L_2.

```
FIFO-PREFLOW-PUSH (N, A, u, s, t)
00  PRÉ-PROCESSAMENTO ()
```

```
10  L_1 ← L_2 ← ⟨⟩
11  para cada j em N faça
12      se e(j) > 0 e j ≠ t
13          então acrescente j ao final de L_1
14  enquanto L_1 ≠ ⟨⟩ ou L_2 ≠ ⟨⟩ faça
15      se L_1 = ⟨⟩
16          então L_1 ← L_2
17                L_2 ← ⟨⟩
18          senão seja i o primeiro nó em L_1
19                retire i de L
20                NODE-EXAMINATION (i)
21  devolva FLUXO (x̌)

PUSH (ij)
26  δ ← min {e(i), u_ij − x̌_ij}     ▷ δ > 0
27  x̌_ij ← x̌_ij + δ
28  x̌_ji ← x̌_ji − δ
29  e(i) ← e(i) − δ
30  e(j) ← e(j) + δ
31  se e(j) > 0 e j ≠ t     ▷ j ≠ s
32      então acrescente j ao final de L_2

RELABEL (i)
33  z(i) ← −∞
34  para cada ij em A(i) faça
35      se x̌_ij < u_ij e z(i) < z(j) − 1
36          então z(i) ← z(j) − 1
37  acrescente i ao final de L_2
```

Cada **fase** é uma seqüência de iterações do processo no bloco de linhas 15–17 entre duas ocorrências consecutivas de $L_1 = \langle\rangle$.

Lema 18.1 *O número de fases não passa de* $2n^2 + n$.

DEMONSTRAÇÃO: Exercício. ∎

Cada nó é submetido à rotina NODE-EXAMINATION no máximo uma vez durante cada fase. Cada NODE-EXAMINATION executa no máximo um PUSH não-saturante (veja seção 17.3). Logo, o número total de execuções não-saturantes de PUSH não passa de

$$2n^3 + n^2 .$$

Segue daí que o consumo total de tempo do algoritmo é

$$\mathrm{O}(n^3)\ .$$

Essa análise do consumo de tempo é baseada em Cheriyan and Maheshwari [1989].

Exercícios

18.1 [AMO 7.5, fig.7.21(a), p.243] Use o algoritmo FIFO-PREFLOW-PUSH para resolver o problema de fluxo máximo descrito na figura 7.21(a). Ao executar o algoritmo, considere os nós em ordem crescente de número; use essa ordem para escolher o próximo nó ativo; para percorrer o conjunto $A(i)$, use a ordem crescente da ponta final dos arcos. Exiba o estado da rede no início de cada iteração. Conte o número de pushes saturantes, o número de pushes não-saturantes e o número de relabels.

18.2 [AMO 7.15, fig.7.24, p.246] Aplique o algoritmo FIFO-PREFLOW-PUSH ao grafo da figura 7.24. Determineo número de pushes em função dos parâmetros L e W (o número de nós e as capacidades dependem desses parâmetros). Para um dado n, que valores de L e W produzem o maior número de pushes?

18.3 [AMO 7.10, p.245] Sejam s e t dois nós de uma rede capacitada simétrica (N, A, u). Sejam α e K dois números inteiros positivos. Suponha que a capacidade de cada arco pertence ao conjunto $\{\alpha, 2\alpha, 3\alpha, \ldots, K\alpha\}$. Mostre que o consumo de tempo do algoritmo FIFO-PREFLOW-PUSH para esse tipo de rede é $\mathrm{O}(\min(Knm, n^3))$.

Parte III

Fluxo viável de custo mínimo

Capítulo 19

Fluxo viável

Este capítulo[1] trata de uma importante generalização do problema do fluxo máximo (problema 11.1). O capítulo culmina com o teorema de Gale, que dá condições necessárias e suficientes para a existência de um fluxo viável, ou seja, um fluxo que respeita restrições de capacidade e satisfaz dadas exigências de excesso em cada nó.

19.1 Nós com demandas

Uma **função-demanda** em um grafo (N, A) é qualquer função que associa um número inteiro $b(i)$ a cada nó i, ou seja, qualquer função

$$b : N \to \mathbb{Z} \,.$$

Como de hábito, $b(T) := \sum_{j \in T} b(j)$ para qualquer subconjunto T de N. Se b é uma função-demanda, diz-se às vezes que $-b$ é uma função-oferta (ou função-suprimento).

Como dissemos na seção 10.1, um **fluxo** é uma função de A em $\mathbb{Z}_{\geq}$ sem quaisquer restrições. Dizemos que um fluxo x **satisfaz** uma função-demanda b se, para cada nó i, o excesso de x em i é igual a $b(i)$, ou seja, se

$$x(\bar{i}, i) - x(i, \bar{i}) \;=\; b(i) \,.$$

Podemos dizer também que um tal x é um b-**fluxo**. Como já fizemos nos capítulos anteriores, diremos que x **respeita** u se $x \leq u$.

Esta seção[2] trata do problema de determinar um fluxo **viável**, ou seja, um fluxo que satisfaz b e respeita u.

[1] Resumo da seção 6.7, p.191, do AMO.
[2] Trata-se de um resumo da subseção *Application 1: Feasible flow problem*, p. 169, de AMO.

Problema 19.1 (do fluxo viável) *Dada uma rede (N, A, u, b) em que u é uma função-capacidade e b uma função-demanda, encontrar um fluxo que satisfaça b e respeite u.*

19.2 Condições de viabilidade

É fácil verificar que uma condição necessária para que o problema tenha solução é $b(N) = 0$ (veja exercício 19.1). Também é fácil verificar outra condição necessária: $b(T) \leq u(\overline{T}, T)$ para todo subconjunto T de N (veja exercício 19.2).

Lema 19.2 *Dada uma rede (N, A, u, b) em que u é uma função-capacidade e b uma função-demanda, se existe fluxo que satisfaz b e respeita u então*

$$-u(T, \overline{T}) \leq b(T) \leq u(\overline{T}, T) \tag{19.1}$$

para todo subconjunto T de N.

DEMONSTRAÇÃO: Suponha que x é um fluxo que respeita u e satisfaz b. Então

$$b(N) = \textstyle\sum_{i \in N} b(i) = \sum_{i \in N} (x(\bar{i}, i) - x(i, \bar{i})) = 0\,,$$

de acordo com o lema 10.1. Por outro lado, para qualquer conjunto T de nós, $b(T) = x(\overline{T}, T) - x(T, \overline{T}) \leq u(\overline{T}, T) - 0 = u(\overline{T}, T)$. ∎

É claro que a condição $b(N)$ e um caso particular de (19.1).

EXEMPLO: O grafo tem nós i, j, k, l. É dada uma função-demanda b. O conjunto $T := \{j, l\}$ viola a condição $b(T) \leq u(\overline{T}, T)$ e assim mostra que não existe fluxo que satisfaz b e respeita u.

nó	b
i	-4
j	0
k	-1
l	$+5$

arco	u
ij	2
jk	2
jk	2
jl	3
kl	2

19.3 Teorema de Gale

A condição necessária discutida no lema 19.2 é também suficiente, como mostraremos a seguir. Nossa demonstração é uma generalização do teorema do fluxo máximo e corte mínimo (teorema 11.5), o que mostra que problema do fluxo viável pode ser considerado um caso particular do problema 11.1 do fluxo máximo.

Teorema 19.3 (de Gale, 1957) *Dada uma rede (N, A, u, b) em que u é uma função-capacidade e b uma função-demanda, se*

$$-u(T, \overline{T}) \leq b(T) \leq u(\overline{T}, T) \tag{19.2}$$

para todo subconjunto T de N então existe fluxo que satisfaz b e respeita u.[3]

DEMONSTRAÇÃO: Digamos que a *discrepância* de um fluxo x é o número $\sum_i |b(i) - e(i)|$, onde $e(i) := x(\bar{i}, i) - x(i, \bar{i})$. É claro que um fluxo satisfaz b se e só se sua discrepância é nula.

Seja x um fluxo de discrepância mínima dentre os que respeitam u. Vamos mostrar que x tem discrepância nula ou as condições (19.2) de Gale estão violadas.

Suponha inicialmente que $e(i) \leq b(i)$ para todo nó i. Então $\sum_{i \in N} b(i) \geq \sum_{i \in N} e(i)$ e portanto, em virtude de (10.1),

$$b(N) \geq x(\emptyset, N) - x(N, \emptyset) = 0 \,.$$

Se $b(N) > 0$ então N viola a condição de Gale. Caso contrário, donde $e(i) = b(i)$ para todo i e portanto x tem discrepância nula.

Suponha agora que $e(s) > b(s)$ para algum nó s. Adote a definição de pseudo-caminho que usamos no capítulo 11. Seja $\acute{P}$ o conjunto dos arcos diretos e $\grave{P}$ o conjunto dos arcos inversos de qualquer pseudo-caminho P. Diremos que um pseudo-caminho P é **positivo** se $x_{ij} < u_{ij}$ para cada ij em $\acute{P}$ e $x_{kl} > 0$ para cada kl em $\grave{P}$. Seja S o conjunto de todos os nós que são término de algum pseudo-caminho positivo que começa em s. A definição de S garante que

$$x(S, \overline{S}) = u(S, \overline{S}) \quad \text{e} \quad x(\overline{S}, S) = 0 \,. \tag{19.3}$$

Suponha agora $e(t) < b(t)$ para algum t em S. Seja P um pseudo-caminho positivo de s a t. Escolha o maior número δ que satisfaça as seguintes restrições: $\delta \leq u_{ij} - x_{ij}$ para cada ij em $\acute{P}$, $\delta \leq x_{ij}$ para cada ij em $\grave{P}$, $\delta \leq e(s) - b(s)$ e $\delta \leq b(t) - e(t)$. Seja x' o fluxo definido a partir de x como segue:

$$x'_{ij} = \begin{cases} x_{ij} + \delta & \text{se } ij \in \acute{P} \\ x_{ij} - \delta & \text{se } ij \in \grave{P} \\ x_{ij} & \text{em qualquer outro caso.} \end{cases}$$

É claro que o fluxo x' respeita u. Como $\delta > 0$, a discrepância de x é estritamente menor que a de x. Isso é inconsistente com nossa escolha de x. Devemos concluir portanto que

$$e(i) \geq b(i)$$

para cada j em S. Como $e(s) > b(s)$, o conjunto S viola a condição de Gale:

$$b(S) = \textstyle\sum_{i \in S} b(i) < \sum_{i \in S} e(i) = x(\overline{S}, S) - x(S, \overline{S}) = -u(S, \overline{S}) \,,$$

em virtude do lema 10.1 e de (19.3). ∎

[3] Veja teorema 6.12, p.196, de AMO. Veja também exercício 6.43, p.205, de AMO.

A prova do teorema é um algoritmo para o problema do fluxo viável. A próxima seção descreve o algoritmo em detalhe, depois de restringir o problema a grafos simétricos.

19.4 Algoritmo do fluxo viável

Com base no fato 12.1, podemos nos restringir a redes simétricas. O algoritmo abaixo recebe uma rede simétrica (N, A, u, b) com função-capacidade u e função-demanda b e devolve (1) um fluxo viável x ou (2) um subconjunto S de N que viola ao condição (19.1). fato 12.1

```
FLUXO-VIÁVEL[4] (N, A, u, b)     ▷ (N, A) simétrico
01  se b(N) ≠ 0
02      então devolva N
03  x̌ ← 0
04  e ← 0
05  enquanto existe s em N tal que e(s) > b(s) faça
06      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
07      ⟨y, π⟩ ← CAMINHO (N, A_x̌, s)
08      se existe t em N tal que y(t) = 0 e e(t) < b(t)
09          então INCREMENTE-FLUXO-VIÁVEL (u, x̌, A_π, s, t)
10          senão S ← {i : y(i) = 0}
11                devolva S
12  x ← FLUXO (x̌)
13  devolva x
```

O algoritmo CAMINHO é uma pequena adaptação do algoritmo BUSCA discutido na seção 3.3: ele recebe um nó s de um grafo (N, E) e devolve um 0-potencial y e uma função-predecessor π tais que, para cada nó t tal que $y(t) - y(s) = 0$, existe um caminho de s a t no grafo (N, E_π). Convém lembrar que o 0-potencial é uma função y de N em $\{0, 1\}$ tal que $y(j) - y(i) \leq 0$ para cada arco ij.

```
INCREMENTE-FLUXO-VIÁVEL (u, x̌, A_π, s, t)
14  seja P um caminho de s a t em (N, A_π)
15  δ ← min {u_ij − x̌_ij : ij é arco de P}
16  δ ← min {δ, e(s) − b(s), b(t) − e(t)}     ▷ δ > 0
17  para cada arco ij de P faça
18      x̌_ij ← x̌_ij + δ
19      x̌_ji ← x̌_ji − δ
20  e(s) ← e(s) − δ
21  e(t) ← e(t) + δ
```

[4] Veja algoritmo *Successive Shortest Paths*, figura 9.9, p.321 do AMO.

Ao longo da execução do algoritmo, seja x o fluxo associado a $\check{x}$, isto é, $x := \text{FLUXO}\,(\check{x})$. No começo de cada iteração do bloco de linhas 06–11,

(i1) $e(i) = x(\bar{i}, i) - x(i, \bar{i})$ para cada nó i;
(i2) $\sum_{i \in N}(b(i) - e(i)) = 0$;
(i4) x respeita u.

Suponha que a execução do algoritmo FLUXO-VIÁVEL termina na linha 11. Seja $S := \{i : y(i) = 0\}$. Como $e(i) \geq b(i)$ para todo i em S e $e(s) > b(s)$, temos $\sum_{i \in S}(b(i) < \sum_{i \in S} e(i)$. Pela invariante (i1) e o lema 10.1,

$$b(S) < \textstyle\sum_{i \in S}(x(\bar{i}, i) - x(i, \bar{i})) = x(\overline{S}, S) - x(S, \overline{S})\,.$$

Como devemos ter $y(j) - y(i) \leq 0$ para cada arco ij em $A_{\check{x}}$, o corte $(S, \overline{S})$ é vazio no grafo $(N, A_{\check{x}})$, donde $\check{x}_{ij} = u_{ij}$ para cada arco ij no corte $(S, \overline{S})$ do grafo (N, A). Logo, $x_{ij} = u_{ij}$ e $x_{ji} = 0$ para cada arco ij no corte $(S, \overline{S})$ do grafo (N, A), donde

$$x(S, \overline{S}) = u(S, \overline{S}) \quad \text{e} \quad x(\overline{S}, S) = 0\,.$$

Segue-se que $b(S) < -u(S, \overline{S})$ e portanto S viola as condições de Gale.

Se a execução do algoritmo termina na linha 13 então temos $e(i) \leq b(i)$ para todo i e portanto, em virtude da invariante (i2), $e(i) = b(i)$ e portanto x satisfaz b.

Consumo de tempo. Em cada iteração, a soma $\sum_i |b(i) - e(i)|$ diminui em pelo menos duas unidades. O valor máximo dessa soma é nB, sendo $B := \max_{i \in N} |b(i)|$.[5] Logo, o número de iterações de FLUXO-VIÁVEL não passa de nB.

O consumo de tempo da rotina CAMINHO é $O(n + m)$ e o consumo de tempo de INCREMENTE-FLUXO é $O(n)$. Logo, o consumo de tempo total de FLUXO-VIÁVEL é

$$O((n + m)nB)\,.$$

Portanto, o algoritmo é apenas pseudo-polinomial.

Algoritmo fortemente polinomial. O algoritmo FLUXO-VIÁVEL pode ser aperfeiçoado e transformado em um algoritmo fortemente polinomial da mesma maneira que FORD-FULKERSON foi transformado em um algoritmo polinomial em capítulos anteriores. Em particular, FLUXO-VIÁVEL pode ser implementado de modo que seu consumo de tempo seja

$$O(n^2 m)$$

Uma maneira alternativa de resolver o problema do fluxo viável em tempo polinomial é aplicar a uma rede auxiliar qualquer algoritmo para o problema do fluxo máximo. Eis

[5] Por algum misterioso motivo, AMO usa a letra "U" no lugar do meu "B". Veja p.323.

como isso pode ser feito. Escolha objetos s' e t' que não estejam em N. Seja N' o conjunto $N \cup \{s', t'\}$. Seja S o conjunto de todos os nós j em N para os quais $b(j) < 0$. Seja T o conjunto dos nós i para os quais $b(i) > 0$. Seja A' o conjunto $A \cup \{s'j : j \in S\} \cup \{it' : i \in T\}$. Defina a função-capacidade u' sobre A' da seguinte maneira:

$$\begin{array}{lcll} u'_{ij} & = & u_{ij} & \text{se } ij \in A \\ u'_{s'j} & = & -b(j) & \text{para cada } j \in S \\ u'_{it'} & = & b(i) & \text{para cada } i \in T\,. \end{array}$$

Qualquer fluxo x que satisfaz b e respeita u na rede (N, A, u, b) corresponde, naturalmente, a um (s', t')-fluxo x' na rede (N', A', u') que respeita u' e satura todos os arcos que saem de s', ou seja, tal que $x'_{s'i} = u'_{s'i}$ para todo j em S. É claro que um tal x' tem valor máximo. Reciprocamente, pode-se verificar que um (s', t')-fluxo na rede (N', A', u') que respeita u' e satura todos os arcos que saem de s' corresponde a um fluxo que satisfaz b e respeita u na rede (N, A, u, b).

Exercícios

19.1 Suponha que x é um fluxo que satisfaz uma função-demanda b em uma rede (N, A, b). Prove diretamente que $b(N) = 0$. Mostre que a recíproca não é verdadeira.

19.2 Seja (N, A, u, b) uma rede em que u é uma função-capacidade e b é uma função-demanda. Seja x um fluxo que satisfaz b e respeita u. Prove diretamente que $b(i) \leq u(\bar{i}, i)$ para cada nó i. Prove diretamente que $b(T) \leq u(\overline{T}, T)$ para cada subconjunto T de N. Mostre que a recíproca não é verdadeira (ou seja, a validada dessas condições não garante a existência de um fluxo viável).

19.3 Seja (N, A, u, b) uma rede em que u é uma função-capacidade e b é uma função-demanda. (1) Suponha que essa rede satisfaz as hipóteses do lema 19.2. Prove que $b(N) = 0$. (2) Agora suponha que $b(N) = 0$ e $b(T) \leq u(\overline{T}, T)$ para todo subconjunto T de N e prove que a rede satisfaz as hipóteses do lema 19.2.

19.4 Seja x' um (s', t')-fluxo de valor máximo na rede (N', A', u') definida na fim da seção 19.4. (1) Suponha que x' satura todos os arcos que saem de s', ou seja, que $x'_{s'i} = u'_{s'i}$ para todo j em S. Mostre que a restrição de x a A é um fluxo que satisfaz b e respeita u. (2) Agora suponha que x' não satura todos os arcos que saem de s'. Seja $(\overline{T}, T)$ um (s', t')-corte de capacidade mínima. Mostre como extrair desse corte um conjunto que viola as condições de Gale na rede original (N, A, u, b).

19.5 Deduza o teorema de Gale (teorema 19.3) do teorema do fluxo máximo e corte mínimo (teorema 11.5).

19.6 Suponha dados números inteiros não-negativos $\alpha_1, \ldots, \alpha_n$ e $\beta_1, \ldots, \beta_n$. Queremos construir um grafo bipartido com bipartição $(\{1', \ldots, n'\}, \{1'', \ldots, n''\})$ tal que cada nó i' tenha grau de saída $\alpha(i)$ e e cada nó i'' tenha grau de entrada $\beta(i)$:

$$|A(i')| = \alpha_i \quad \text{e} \quad |\widetilde{A}(i'')| = \beta_i \,.$$

Mostre como resolver o problema. (*Sugestão:* Reduza a um problema de fluxo máximo.)

19.7 Seja (N, A) um grafo com $N = \{1, \ldots, n\}$. Suponha dados números inteiros positivos $\alpha_1, \ldots, \alpha_n$ e $\beta_1, \ldots, \beta_n$. Queremos encontrar um subconjunto B de A tal que cada nó i em (N, B) tenha grau de saída $\alpha(i)$ e grau de entrada $\beta(i)$:

$$|B(i)| = \alpha_i \quad \text{e} \quad |\widetilde{B}(i)| = \beta_i \,.$$

Mostre como resolver o problema. (*Sugestão:* Veja exercício anterior. Separa cada nó i em um par (i', i''); os arcos que entram em i passam a entrar em i' e os que saem de i passam a sair de i''; não há arcos ligando i' a i''.)

19.8 Formule o problema do fluxo viável (problema 19.1) como um programa linear inteiro. Escreva o dual da relaxação linear do programa. Prove o teorema fraco da dualidade para esse par de programas lineares.

19.9 [Fluxo viável paramétrico. AMO 7.28, p.248] Seja (N, A, u, b) uma rede em que u é uma função-capacidade e b uma função-demanda. Suponha que b varia com o tempo: $b(i) = b^0(i) + \tau b^*(i)$ para cada nó i. Suponha que $\sum_{i \in N} b^0(i) = 0$, que $\sum_{i \in N} b^*(i) = 0$ e que a rede (N, A, u, b^0) é viável. Queremos determinar o maior valor inteiro de τ tal que $(N, A, u, b^0 + \tau b^*)$ é viável. Proponha um algoritmo eficiente para resolver o problema.

Capítulo 20

Fluxo viável de custo mínimo: introdução

Este capítulo introduz o problema do fluxo viável de custo mínimo (*min-cost flow problem*), que é o assunto central do texto e do curso. A solução do problema, a ser estudada nos próximos capítulos, depende dos problemas do caminho mínimo, do ciclo negativo e do fluxo máximo.

20.1 O problema

Uma **função-custo** para um grafo (N, A) é qualquer função de A em $\mathbb{Z}$, ou seja, qualquer função que associa um número inteiro c_{ij} com cada arco ij do grafo. O número c_{ij} é o **custo** do arco e pode ser positivo, negativo ou nulo. O **custo** de um fluxo x na rede (N, A, c) é o número

$$cx := \sum_{ij \in A} c_{ij} x_{ij} \,.$$

Problema 20.1 (do fluxo viável de custo mínimo) *Dada uma rede (N, A, u, b, c) com função-capacidade u, função-demanda b e função-custo c, encontrar um fluxo viável de custo mínimo que satisfaça b e respeite u.*

Como já dissemos no capítulo 19, um fluxo é **viável** se satisfaz b e respeita u. De acordo com o teorema 19.3, um tal fluxo existe se e só se $-u(T, \overline{T}) \le b(T) \le u(\overline{T}, T)$ para todo subconjunto T de N. Podemos enunciar o problema assim:

encontrar um fluxo viável de custo mínimo.

Diremos que um tal fluxo é **ótimo**.

20.2 Condição de otimalidade

Suponha que um fluxo x é solução do problema 20.1; como é possível provar que cx é mínimo? Em outras palavras, dado um número ω, como é possível provar que não existe um fluxo viável de custo menor que ω? A resposta a seguir é baseada na teoria da dualidade de programação linear.

Lema 20.2 *Em qualquer rede, se x é um fluxo viável então*

$$cx \geq yb - wu$$

para qualquer função-custo $w \geq 0$ e qualquer $(c+w)$-potencial y.

Convém lembrar que uma função-custo é qualquer função de A em $\mathbb{Z}$. Convém lembrar também (veja capítulo 6) que um $(c+w)$-potencial é qualquer potencial y tal que $y(j) - y(i) \leq c_{ij} + w_{ij}$ para cada arco ij. Na falta de um nome melhor, podemos dizer que um par (y, w) é uma **solução dual-viável** se w é uma função-custo não-negativa e y é um $(c+w)$-potencial. Uma última observação: o lema adota as abreviaturas $yb := \sum_{i \in N} y(i)b(i)$ e $wu := \sum_{ij \in A} w_{ij}u_{ij}$.

DEMONSTRAÇÃO: Como x satisfaz b, temos

$$\begin{aligned}
yb &= \textstyle\sum_i y(i)\, b(i) \\
&= \textstyle\sum_i y(i) \left(x(\bar{i}, i) - x(i, \bar{i})\right) \\
&= \textstyle\sum_i y(i)x(\bar{i}, i) \;-\; \sum_i y(i)x(i, \bar{i}) \\
&= \textstyle\sum_j y(j)x(\bar{j}, j) \;-\; \sum_i y(i)x(i, \bar{i}) \\
&= \textstyle\sum_i y(j) \sum_{ij} x_{ij} \;-\; \sum_i y(i) \sum_{ij} x_{ij} \\
&= \textstyle\sum_{ij} y(j)x_{ij} \;-\; \sum_{ij} y(i)x_{ij} \\
&= \textstyle\sum_{ij} (y(j) - y(i))\, x_{ij}\,. \qquad (20.1)
\end{aligned}$$

É claro que as expressões da forma $\sum_{ij}$ devem ser entendidas como $\sum_{ij \in A}$ e expressões da forma $\sum_i$ devem ser entendidas como $\sum_{i \in N}$. Então

$$\begin{aligned}
yb &= \textstyle\sum_{ij} (y(j) - y(i))\, x_{ij} \\
&\leq \textstyle\sum_{ij} (c_{ij} + w_{ij})\, x_{ij} \\
&= (c+w)x \\
&\leq cx + wu\,,
\end{aligned}$$

uma vez que x respeita u. ∎

Portanto, para mostrar que uma dada rede (N, A, u, b, c) não admite um fluxo viável de custo menor que um determinado número, digamos 99, basta exibir funções y e $w \geq 0$ tais que y é um $(c+w)$-potencial e $yb - wu \geq 99$.

Corolário 20.3 *Se x é um fluxo viável $cx = yb - wu$ para alguma função-custo $w \geq 0$ e algum $(c + w)$-potencial y então x é fluxo ótimo.*

Como veremos nos próximos capítulos, a recíproca do corolário é verdadeira: se x é um fluxo viável que minimiza cx então existe uma função-custo $w \geq 0$ e um $(c+w)$-potencial y tais que $cx = yb + wu$.

20.3 Folgas complementares

A condição de otimalidade dada no corolário 20.3 pode ser reformulada em termos de "folgas complementares". Seja x um fluxo que respeita u e y um potencial (ou seja, uma função de N em $\mathbb{Z}$). Diremos que **as folgas de x e y são complementares** se, para cada arco ij,[1]

$$\begin{aligned} x_{ij} > 0 \quad &\Rightarrow \quad y(j) - y(i) \geq c_{ij} \quad \text{e} \\ x_{ij} < u_{ij} \quad &\Rightarrow \quad y(j) - y(i) \leq c_{ij} \,. \end{aligned}$$

É claro que essas condições poderiam ter sido igualmente bem formuladas assim: $y(j) - y(i) < c_{ij} \Rightarrow x_{ij} = 0$ e $y(j) - y(i) > c_{ij} \Rightarrow x_{ij} = u_{ij}$. Uma conseqüência imediata da condição de folgas complementares: se $0 < x_{ij} < u_{ij}$ então $y(j) - y(i) = c_{ij}$.

O conceito de folgas complementares permite que o corolário 20.3 seja reformulado da seguinte maneira:

Corolário 20.4 *Se x é um fluxo viável e suas folgas são complementares às de algum potencial y então x é ótimo.*

DEMONSTRAÇÃO: Adote a menor função-custo $w \geq 0$ para a qual y é um $(c + w)$-potencial, ou seja, defina w da seguinte maneira: para cada arco ij,

$$w_{ij} := \max \{0 \,,\, y(j) - y(i) - c_{ij}\} \,.$$

Com essa definição, é evidente que y é um $(c + w)$-potencial. Para provar o corolário, basta mostrar que $cx = yb - wu$. Seja D o conjunto dos arcos ij para os quais $y(j) - y(i) < c_{ij}$. Seja E o conjunto dos arcos ij para os quais $y(j) - y(i) = c_{ij}$. Seja F o conjunto dos arcos ij para os quais $y(j) - y(i) > c_{ij}$. Como as folgas de y são complementares às

[1] Eis a intuição que motiva o conceito. O número $c_{ij} - (y(j) - y(i))$ é o "custo reduzido" do arco ij. Se o custo reduzido de ij for estritamente positivo, então devemos diminuir o valor de x_{ij} para minimizar o custo do fluxo; mas isso só pode ser feito se $x_{ij} > 0$. Por outro lado, se o custo reduzido for negativo então devemos aumentar x_{ij}; mas isso só é possível se $x_{ij} < u_{ij}$. Assim, as condições de folgas complementares parecem caracterizar a otimalidadade do fluxo.

de x,

$$\begin{aligned} x_{ij} = 0 \text{ e } w_{ij} = 0 \quad & \text{para cada } ij \text{ em } D, \\ y(j) - y(i) = c_{ij} \text{ e } w_{ij} = 0 \quad & \text{para cada } ij \text{ em } E, \\ x_{ij} = u_{ij} \text{ e } w_{ij} = y(j) - y(i) - c_{ij} \quad & \text{para cada } ij \text{ em } F. \end{aligned}$$

Logo,

$$\begin{aligned} cx &= \textstyle\sum c_{ij}x_{ij} \\ &= \textstyle\sum_D c_{ij}x_{ij} + \sum_E c_{ij}x_{ij} + \sum_F c_{ij}x_{ij} \\ &= \textstyle\sum_D (y(j) - y(i))x_{ij} + \sum_E (y(j) - y(i))x_{ij} + \sum_F c_{ij}u_{ij} \\ wu &= \textstyle\sum w_{ij}u_{ij} \\ &= \textstyle\sum_F (y(j) - y(i))x_{ij} - \sum_F c_{ij}u_{ij} \\ cx + wu &= \textstyle\sum_D (y(j) - y(i))x_{ij} + \sum_E (y(j) - y(i))x_{ij} + \sum_F (y(j) - y(i)x_{ij} \\ &= \textstyle\sum_{ij} (y(j) - y(i))x_{ij} \\ &= yb \end{aligned}$$

em virtude de (20.1). Como $cx = yb - wu$, o corolário 20.3 garante que cx é mínimo. ∎ (20.1) 20.3

Portanto, para mostrar que um fluxo viável x tem custo mínimo, basta exibir um potencial y que tenha folgas complementares às de x.

A propósito, é fácil mostrar que a recíproca do corolário é verdadeira: se x é um fluxo viável e $cx = yb - wu$ para alguma função-custo $w \geq 0$ e algum $(c+w)$-potencial y então as folgas de y são complementares às de x.

EXEMPLO:[2] O grafo tem nós i, j, k, l. É dado um fluxo viável x. A última tabela registra y e w tais que y é um $(c + w)$-potencial e $cx = 14 = yb - wu$.

nó	b
i	-4
j	0
k	0
l	$+4$

arco	c	x	u
ij	2	2	4
ik	2	2	2
jk	1	2	2
jl	3	0	3
kl	1	4	5

nó	y
i	0
j	2
k	4
l	5

arco	w
ij	0
ik	2
jk	1
jl	0
kl	0

Exercícios

20.1 Seja (N, A, u, c) uma rede em que u uma função-capacidade e c é uma função-custo. Seja y um c-potencial. Especifique um fluxo x que respeite u e tenha folgas complementares com as de y.

[2] Este é o exemplo da fig.9.8, p.318, de AMO.

20.2 [Fluxo de custo mínimo sob capacidades infinitas] Seja b uma função-demanda e c uma função-custo (não necessariamente $c \geq 0$) em um grafo (N, A). Seja x um fluxo que satisfaz b. Seja y um c-potencial. (1) Prove que $cx \geq yb$. (2) Suponha que $y(j) - y(i) = c_{ij}$ para todo arco ij tal que $x_{ij} > 0$. Prove que $cx = yb$.

20.3 Seja (N, A, c) uma rede com função-custo $c \geq 0$. Seja s um nó da rede e seja b a seguinte função-demanda:

$$b(i) = 1 \text{ para cada } i \text{ em } N - \{s\} \text{ e } b(s) = -(n-1),$$

onde $n = |N|$. Suponha que existe um fluxo que satisfaz b. Qual o melhor algoritmo que você conhece para determinar um fluxo x que satisfaça b e um c-potencial y tais que $cx = yb$?

20.4 [Programa linear] Escreva um programa linear para representar o problema do fluxo viável de custo mínimo. Escreva o dual do programa linear. Prove o lema fraco da dualidade para esse par de problemas. Compare o resultado com o lema 20.2.

20.5 Seja x um fluxo que respeita u e y um potencial. Suponha que x e y têm folgas complementares. Se $u_{ij} = 0$ para algum arco ij, que valor pode ter a diferença $y(j) - y(i)$?

20.6 Suponha que x é um fluxo viável e seja y um potencial. Mostre que as folgas de y são complementares às de x se e só se existe uma função-custo $w \geq 0$ tal que y é um $(c + w)$-potencial e $cx = yb - wu$.

20.7 Suponha que y é um c-potencial em uma rede (N, A, c), ou seja, suponha que $y(j) - y(i) \leq c_{ij}$ para cada arco ij. Defina c^y da seguinte maneira: $c^y_{ij} := c_{ij} - (y(j) - y(i))$ para cada arco ij. Mostre que se y' é um c^y-potencial então $y + y'$ é um c-potencial.

Capítulo 21

Fluxo em redes simétricas

Para que possamos ter algum conforto ao escrever algoritmos para o problema do fluxo viável de custo mínimo (ou seja, para possamos escrever os algoritmos sem recorrer a pseudo-caminhos nem pseudo-ciclos), é conveniente restringir a atenção a redes simétricas, ou seja, redes em que a presença de um arco ij implica na presença do arco "inverso" ji. Além disso, em alguns algoritmos convém supor que $c \geq 0$.

Este capítulo mostra que todas essas restrições podem ser feitas sem perda de generalidade. Para interpretar corretamente o "sem perda de generalidade", devemos entender que uma solução completa do problema do fluxo viável de custo mínimo consiste não só em um fluxo viável x mas também em um potencial y cujas folgas são complementares às de x.

21.1 Redes anti-simétricas e custo não-negativo

Nossa primeira providência é mostrar que podemos restringir o problema do fluxo viável de custo mínimo, sem perder generalidade, a grafos *anti*-simétricos, ou seja, grafos em que a presença de um arco ij implica na *ausência* do arco "inverso" ji.

Fato 21.1 *É suficiente resolver o problema 20.1 do fluxo viável de custo mínimo para grafos anti-simétricos.*

DEMONSTRAÇÃO: Seja (N, A, u, b, c) uma rede arbitrária com função-capacidade u, função-demanda b e função-custo c.

Suponha que o grafo (N, A) tem um arco hj e também um arco jh. Subdivida o arco hj por um novo nó i (ou seja, escolha um objeto i que não esteja em N, acrescente i a N e troque hj por hi e ij). Defina

$$b(i) := 0\,,\ u_{hi} := u_{hj}\,,\ u_{ij} := u_{hj}\,,\ c_{hi} := c_{hj}\,,\ c_{ij} := 0\,.$$

Suponha dado um fluxo viável x' na nova rede. Suponha dada também um potencial y' na nova rede que tem folgas complementares às de x'. É claro que x' corresponde, naturalmente, a um fuxo viável x na rede original. É claro também que x' e x têm o mesmo custo. Além disso, a restrição de y' a N tem folgas complementares às de x na rede original. (Verifique!) ∎

Fato 21.2 *É suficiente resolver o problema 20.1 do fluxo viável de custo mínimo em redes anti-simétricas com função-custo não-negativa.*[1]

DEMONSTRAÇÃO: Seja (N, A, u, b, c) uma rede arbitrária com função-capacidade u, função-demanda b e função-custo c. De acordo com o fato 21.1, podemos supor, sem perder generalidade, que o grafo (N, A) é anti-simétrico.[2] fato 21.1

Suponha agora que $c_{pq} < 0$ para algum arco pq. Defina uma nova rede anti-simétrica (N, A', u', b', c') da seguinte maneira: $A' := (A - \{pq\}) \cup \{qp\}$,

$$u'_{qp} := u_{pq}\ ,\ b'(p) := b(p) + u_{pq}\ ,\ b'(q) := b(q) - u_{pq}\ ,\ c'_{qp} := -c_{pq}\ ,$$

$c'_{ij} := c_{ij}$ para $ij \neq qp$, $u'_{ij} := u_{ij}$ para $ij \neq qp$, e $b'(i) := b(i)$ para todo nó i distinto de p e de q.

Suponha dado um fluxo viável x' na nova rede. Suponha dada também um potencial y' que tem folgas complementares às de x'. Para transformar x' num fluxo viável x na rede original, basta definir

$$x_{pq} := u_{pq} - x'_{qp}$$

e $x_{ij} := x'_{ij}$ para os demais arcos ij. O potencial y' terá folgas complementares às de x na rede original. ∎

21.2 Redes simétricas

Fato 21.3 *É suficiente resolver o problema 20.1 do fluxo viável de custo mínimo em redes simétricas. Podemos supor que $u_{ij} = 0$ ou $u_{ji} = 0$ para cada arco ij. Podemos supor, além disso, que $c_{ij} \geq 0$ sempre que $u_{ij} > 0$.*

DEMONSTRAÇÃO: Seja (N, A, u, b, c) uma rede arbitrária com função-capacidade u, função-demanda b e função-custo c. De acordo com o fato 21.1, podemos supor, sem perder generalidade, que o grafo (N, A) é anti-simétrico. De acordo com o fato 21.2, podemos supor também que $c \geq 0$. fato 21.1 fato 21.2

[1] Veja "Arc Reversal" na seção 2.4, p.40, de AMO.

[2] Isso é necessário apenas porque pretendemos introduzir um novo arco ji para cada arco ij e não queremos que essa operação crie arcos paralelos.

Para cada arco ij, acrescente ao grafo um novo arco ji e atribua valor 0 a u_{ji} e valor abitrário a c_{ji}:

$$u_{ji} := 0 \quad \text{e} \quad c_{ji} := \text{arbitrário}\,.$$

É claro que o novo grafo será simétrico e as funções c e u terão a seguinte propriedade: para cada arco ij,

$$u_{ij} > 0 \;\Rightarrow\; c_{ij} \geq 0\,.$$

Agora suponha dado um fluxo viável x' na nova rede. Suponha dado também um potencial y' na nova rede que tem folgas complementares às de x'. É evidente que a restrição de x' à rede original é um fuxo viável; ademais, os dois fluxos têm o mesmo custo. Além disso, y' também tem folgas complementares às da restrição de x' à rede original. ∎

21.3 Custo anti-simétrico

Redes simétricas têm a vantagem de permitir que cada par (ij, ji) de arcos será tratado como um único objeto. Em particular, dado um fluxo x, o número $x_{ij} - x_{ji}$ pode ser tratado como a intensidade do fluxo de i a j ao longo do par (ij, ji). Dois fluxos x e x' serão considerados equivalentes se $x'_{ij} - x'_{ji} = x_{ij} - x_{ji}$ para cada arco ij. É evidente que se x' e x são equivalentes então $x'(\bar{i}, i) - x'(i, \bar{i}) = x(\bar{i}, i) - x(i, \bar{i})$ para cada nó i. Portanto, se x satisfaz b então x' também satisfaz b.

Gostaríamos que fluxos equivalentes x e x' tivessem *o mesmo custo*. Para isso será necessário exigir que a função-custo c seja **anti-simétrica**, isto é, que

$$c_{ji} = -c_{ij}$$

para cada arco ij em A. Os fatos 21.1 e 21.3 garantem que podemos restringir a atenção, sem perder generalidade, a redes em que a função-custo é anti-simétrica.

fato 21.1
fato 21.3

21.4 Folgas complementares em redes simétricas

O conceito de pseudofluxo numa rede simétrica será definida como na seção 12.2: o pseudofluxo associado a um fluxo x é $\check{x} := \textsc{Pseudofluxo}\,(x)$. Se a rede tem custo anti-simétrico, a condição de folgas complementares adquire uma forma particularmente simples.

Lema 21.4 *Seja (N, A, u, c) uma rede simétrica com custo anti-simétrico, x um fluxo que respeita u e y um potencial. As folgas de x e y são complementares se e só se*

$$\check{x}_{ij} < u_{ij} \quad \Rightarrow \quad y(j) - y(i) \leq c_{ij} \tag{21.1}$$

para cada arco ij.

É claro que a condição (21.1) também pode ser escrita assim: se $y(j) - y(i) > c_{ij}$ então $\check{x}_{ij} = u_{ij}$.

DEMONSTRAÇÃO: Suponha que as folgas de x e y são complementares, ou seja, suponha que para cada arco ij

$$\begin{aligned} y(j) - y(i) < c_{ij} &\quad\Rightarrow\quad x_{ij} = 0 \quad \text{e} \\ y(j) - y(i) > c_{ij} &\quad\Rightarrow\quad x_{ij} = u_{ij}\,. \end{aligned}$$

Agora tome um arco ij tal que $y(j) - y(i) > c_{ij}$. Então $y(i) - y(j) = -(y(j) - y(i)) < -c_{ij} = c_{ji}$, uma vez que c é anti-simétrica. A complementaridade das folgas no arco ji garante então que $x_{ji} = 0$. Por outro lado, a complementaridade da folgas em ij garante que $x_{ij} = u_{ij}$. Logo, $\check{x}_{ij} = x_{ij} - x_{ji} = u_{ij}$.

Suponha agora que $\check{x}_{ij} < u_{ij} \Rightarrow y(j) - y(i) \le c_{ij}$ para cada arco ij. Tome qualquer arco ij e suponha que $y(j) - y(i) < c_{ij}$. Então

$$y(i) - y(j) = -(y(j) - y(i)) > -c_{ij} = c_{ji}$$

donde $\check{x}_{ji} = u_{ji}$. Como $0 \le x \le u$, temos $x_{ij} = 0$. Agora suponha que $y(j) - y(i) > c_{ij}$. Então (21.1) garante que $\check{x}_{ij} = u_{ij}$. Como $0 \le x \le u$, temos $x_{ij} = u_{ij}$. ∎ (21.1)

Em vista desse lema, o corolário 20.4 pode ser reformulado assim: corol 20.4

Corolário 21.5 *Se a função-custo c é anti-simétrica, x é um fluxo viável e existe um potencial y tal que $\check{x}_{ij} < u_{ij} \Rightarrow y(j) - y(i) \le c_{ij}$ para cada arco ij então x é ótimo.*

Exercícios

21.1 Complete as demonstrações dos fatos 21.1, 21.2 e 21.3.

Capítulo 22

Algoritmo de Klein

Este capítulo descreve um primeiro algoritmo para o problema do fluxo viável de custo mínimo. O algoritmo revela a importância dos ciclos de custo negativo no grafo residual.

22.1 Algoritmo de Klein

O algoritmo descrito abaixo é atribuído a Klein [1967]. Ao descrever o algoritmo podemos supor, sem perda de generalidade, de acordo com o fato 21.3, que a rede é simétrica. Vamos supor que a função-custo é anti-simétrica, para que possamos usar o lema 21.4.

O algoritmo de Klein recebe uma rede simétrica (N, A, u, b, c) com c anti-simétrico e devolve (1) um subconjunto T de N que viola a condição de Gale (veja (19.1)) ou (2) um fluxo viável x e um potencial y que tem folgas complementares às de x. Conforme o corolário 20.4, a alternativa (2) garante que x é um fluxo ótimo.

```
KLEIN[1] (N, A, u, b, c)      ▷ c_ji = −c_ij
01   ⟨x_0, T⟩ ← FLUXO-VIÁVEL (N, A, u, b)
02   se x_0 não está definido
03        então devolva T
04   r ← PSEUDOFLUXO (x_0)
```

[1] Veja *Cycle-canceling algorithm* na fig.9.7, p.317, de AMO.

```
05  repita
06      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
07      ⟨O, y⟩ ← CICLO-NEGATIVO (N, A_x̌, c)
08      se O está definido
09          então δ ← min {u_ij − x̌_ij : ij é arco de O}
10                para cada arco ij de O faça
11                    x̌_ij ← x̌_ij + δ
12                    x̌_ji ← x̌_ji − δ
13          senão x ← FLUXO (x̌)
14                devolva x, y
```

O algoritmo auxiliar FLUXO-VIÁVEL produz um fluxo viável x_0 se tal existe (seja seção 19.4). O algoritmo auxiliar CICLO-NEGATIVO é qualquer dos algoritmos discutidos no capitulo 7 (FORD-BELLMAN, por exemplo). Ao receber um grafo (N, E) e uma função-custo c, o algoritmo devolve um ciclo (dirigido) O tal que $c(O) < 0$ ou um c-potencial y. Como se sabe, um c-potencial é um potencial y tal que $y(j) - y(i) \leq c_{ij}$ para cada arco ij em E. A existência de um c-potencial prova a inexistência de ciclo de custo negativo.

Eis as invariantes do algoritmo: no começo de cada iteração do bloco de linhas 06–14,

(i1) x satisfaz b,

(i2) x respeita u,

onde $x := \text{FLUXO}\,(\check{x})$. (Prove essas invariantes!)

Na linha 13, y é um c-potencial em $(N, A_{\check{x}})$. Portanto, $\check{x}_{ij} < u_{ij} \Rightarrow y(j) - y(i) \leq c_{ij}$ para cada arco ij em A. De acordo com o lema 21.4, y tem folgas complementares às de x. Assim, o algoritmo comporta-se conforme prometido. lema 21.4

Se adotarmos $w_{ij} := \max\{0\,,\, y(j) - y(i) - c_{ij}\}$ na linha 13, podemos dizer também que y é um $(c + w)$-potencial e $cx = yb - wu$. Isso garante que o fluxo viável x tem custo mínimo, conforme corolários 20.3 e 20.4. cor 20.3 cor 20.4

Consumo de tempo. Digamos que $U := \max_{ij} u_{ij}$ e $C := \max_{ij} |c_{ij}|$. Para qualquer fluxo x que respeita u, é evidente que

$$-mUC \leq cx \leq mUC\,.$$

A cada iteração, cx diminui estritamente, uma vez qeu $\delta > 0$ e r é inteira. Portanto, o número de iterações não passa de

$$2mUC\,.$$

Supondo que cada execução de FLUXO-VIÁVEL consome $\mathrm{O}(nm^2)$ unidades de tempo e que CICLO-NEGATIVO consome $\mathrm{O}(nm)$ unidade de tempo (veja capítulo 7), podemos dizer que o consumo de tempo do algoritmo KLEIN é $\mathrm{O}(nm^2 + nm^2UC)$, ou seja,

$$\mathrm{O}(nm^2UC)\,.$$

Portanto, o algoritmo é apenas pseudo-polinomial.

22.2 Custo mínimo e ciclo negativo

A análise do algoritmo KLEIN prova o seguinte lema:

Lema 22.1 (dos ciclos negativos) *Em qualquer rede (N, A, u, b, c) com função-custo c anti-simétrica, um fluxo viável x tem custo mínimo se e só se não existe ciclo de custo negativo na rede residual.*

Talvez seja apropriado lembrar que a rede residual é $(N, A_{\check{x}}, c)$, sendo $A_{\check{x}} := \{ij \in A : \check{x}_{ij} < u_{ij}\}$ e $\check{x} :=$ PSEUDOFLUXO (x).

22.3 Teorema do fluxo viável de custo mínimo

O algoritmo de Klein e sua análise, aliados ao fato 21.3 e ao lema 21.4, provam a recíproca dos corolários 20.3.

Teorema 22.2 (do fluxo viável de custo mínimo) *Em qualquer rede (N, A, u, b, c), se x é um fluxo que satisfaz b, respeita u e minimiza cx então existe um função-custo $w \geq 0$ e um $(c + w)$-potencial y tais que $cx = yb - wu$.*

Isso também pode ser formulado como a recíproca do corolário 20.4:

Teorema 22.3 *Em qualquer rede (N, A, u, b, c), se x é um fluxo que satisfaz b, respeita u e minimiza cx então existe um potencial y cujas folgas são complementares às de x.*

Exercícios

22.1 Seja (N, A, u, b) um grafo simétrico com função-capacidade u e uma função-demanda b. Suponha que para cada arco ij temos $u_{ij} = 0$ ou $u_{ji} = 0$. Sejam x e x' dois fluxos que respeitam u e satisfazem b. Mostre que existe uma circulação $\check{x}$ tal que $x' = x + \check{x}$.

22.2 Aplique o algoritmo de Klein à rede representada na tabela:

ij	x_{ij}	u_{ij}	c_{ij}
1 2	0	1	2
1 3	2	2	4
1 4	4	5	1
2 3	0	3	3
3 4	0	1	9
4 2	0	5	1
5 1	3	4	1
5 2	2	4	1

22.3 Prove as invariantes do algoritmo KLEIN.

Capítulo 23

Algoritmo de Jewell

Este capítulo trata de uma extensão do algoritmo FLUXO-VIÁVEL (veja seção 19.4) que resolve o problema do custo de fluxo mínimo. O algoritmo é atribuído a Jewell [1958] e Iri [1960], bem como a Busacker e Gowen [1961].

23.1 O algoritmo

O algoritmo JEWELL opera sobre uma rede (N, A, u, b, c) com função-capacidade u, função-demanda b e função-custo c. O algoritmo supõe que o grafo (N, A) é simétrico, que c é anti-simétrica e que

$$u_{ij} > 0 \;\Rightarrow\; c_{ij} \geq 0 \tag{23.1}$$

para cada arco ij. Com mostram os fatos 21.1, 21.2 e 21.3, essas hipóteses não trazem perda de generalidade. O algoritmo de Jewell devolve (1) um subconjunto T de N que viola a condição de Gale (veja (19.1)) ou (2) um fluxo viável x e um potencial y que tem folgas complementares às de x. Conforme o corolário 21.5, a alternativa (2) garante que x é um fluxo viável de custo mínimo.

JEWELL[1] (N, A, u, b, c) $\quad \triangleright\ c_{ji} = -c_{ij}$ e $u_{ij} > 0 \Rightarrow c_{ij} \geq 0$

01 $r \leftarrow u$
02 $e \leftarrow 0$
03 $y \leftarrow 0$
04 enquanto existe s em N tal que $e(s) > b(s)$ faça
05 $\quad A_{\check{x}} \leftarrow \{ij \in A : \check{x}_{ij} < u_{ij}\}$
06 $\quad$ para cada arco ij em $A_{\check{x}}$ faça
07 $\quad\quad c'_{ij} \leftarrow c_{ij} - (y(j) - y(i)) \quad \triangleright\ c'$ é o "custo reduzido"

[1] Veja o algoritmo *Successive Shortest Paths* na figura 9.9, p.321, do AMO.

```
08      ⟨y′, π⟩ ← DIJKSTRA (N, A_x̌, s, c′)
09      se existe t em N tal que π(t) ≠ NIL e e(t) < b(t)
10          então INCREMENTE-FLUXO-VIÁVEL (x̌, A_π, s, t)
11                  y ← y + y′
12          senão T ← {j ∈ N : π(j) = NIL}
13                  devolva T
14  x ← FLUXO (x̌)
15  devolva x e y
```

O algoritmo auxiliar INCREMENTE-FLUXO-VIÁVEL foi descrito na seção 19.4. Ele tem o efeito de enviar a maior quantidade possível de fluxo ao longo de um caminho de s a t no grafo de predecessores (N, A_π).

O algoritmo DIJKSTRA foi descrito no capítulo 8. A invariante (i4) abaixo garante que y é um c-potencial na rede $(N, A_{\check{x}}, c)$; logo, $c'_{ij} = c_{ij} - (y(j) - y(i)) \geq 0$ para cada ij em $A_{\check{x}}$, estando assim asseguradas as condições de aplicabilidade do algoritmo DIJKSTRA. Ao receber um nó s de uma rede (N, E, c') com função-custo $c \geq 0$, o algoritmo DIJKSTRA devolve um c-potencial y (portanto, $y(j) - y(i) \leq c_{ij}$ para cada arco ij em E) e uma função-predecessor π tais que, para cada nó t tal que $y(t) - y(s) < nC$, onde $C = \max_{ij} |c_{ij}|$, existe um caminho de s a t no grafo (N, E_π).

Eis as invariantes do algoritmo, escritas em termos de $x := \text{FLUXO}\,(\check{x})$:

(i1) $e(i) = x(\bar{i}, i) - x(i, \bar{i})$ para cada nó i;
(i2) $\sum_{i \in N}(b(i) - e(i)) = 0$;
(i3) x respeita u;
(i4) $\check{x}_{ij} < u_{ij} \Rightarrow y(j) - y(i) \leq c_{ij}$ para cada arco ij.

Note que (i4) poderia ter sido formulada assim: y é um c-potencial no grafo $(N, A_{\check{x}})$.

Prova da invariante (i4): A invariante vale no início da primeira iteração, pois nessa ocasião temos $y = 0$ e vale a hipótese (23.1). Suponha agora que (i4) vale no início (23.1)
de uma iteração qualquer que não a última. No fim da linha 08 teremos

$$y'(j) - y'(i) \;\leq\; c'_{ij} \;=\; c_{ij} - (y(j) - y(i))$$

para cada arco ij em $A_{\check{x}}$. Segue daí que $(y(j)+y'(j))-(y(i)+y'(i)) \leq c_{ij}$ para cada ij em $A_{\check{x}}$. Logo, no fim da linha 11, $y(j) - y(i) \leq c_{ij}$ para cada ij em $A_{\check{x}}$, ou seja, para cada arco ij que tenha $\check{x}_{ij} < u_{ij}$ antes da execução da rotina INCREMENTE-FLUXO-VIÁVEL.

Mas a execução dessa rotina pode colocar novos arcos em $A_{\check{x}}$: os arcos ij cujo inverso ji pertence ao caminho de s a t em (N, A_π). Cada arco ji desse caminho é justo, ou seja, satisfaz $y'(i) - y'(j) = c'_{ji}$ e portanto

$$y'(j) - y'(i) = -c'_{ji} = c'_{ij}\,.$$

Logo, $(y(j) + y'(j)) - (y(i) + y'(i)) = c_{ij}$ para cada arco ji do caminho. Portanto, depois da linha 11, $y(j) - y(i) = c_{ij}$ para cada arco ji do caminho.

Em suma, (i4) está satisfeita depois da linha 11, e portanto também no início da próxima iteração.

No fim da última iteração (linha 14), de acordo com a invariante (i4) e o lema 21.4, y tem folgas complementares às de x. Assim, ao devolver x e y o algoritmo está se comportando como prometeu.

23.2 Consumo de tempo

Tal como no algoritmo FLUXO-VIÁVEL, o número de iterações de JEWELL não passa de nB, sendo $B := \max_{i \in N} |b(i)|$. O consumo de tempo de cada iteração iteração é dominado pelo consumo de DIJKSTRA, que é $O(n^2)$. Logo, o consumo total do JEWELL é

$$O(n^3 B)$$

unidades de tempo. Portanto, o algoritmo é apenas pseudo-polinomial.

Capítulo 24

Algoritmo Cost Scaling

Este capítulo discute[1] um algoritmo polinomial para o problema do fluxo viável de custo mínimo. O algoritmo *cost scaling* combina idéias do algoritmo Capacity Scaling (capítulo 14) com as do Preflow-Push.

24.1 Folgas complementares relaxadas

O algoritmo Cost Scaling depende da seguinte relaxação do conceito de folgas complementares. Para qualquer número não-negativo (não necessariamente inteiro) ϵ, um vetor y tem folgas ϵ-**complementares**[2] com um fluxo x se

$$\begin{aligned} x_{ij} > 0 \quad &\Rightarrow \quad y(j) - y(i) \geq c_{ij} - \epsilon \\ x_{ij} < u_{ij} \quad &\Rightarrow \quad y(j) - y(i) \leq c_{ij} + \epsilon \end{aligned}$$

para cada arco ij. Essa condição poderia igualmente bem ser formulada assim: $y(j) - y(i) < c_{ij} - \epsilon \Rightarrow x_{ij} = 0$ e $y(j) - y(i) > c_{ij} + \epsilon \Rightarrow x_{ij} = u_{ij}$. Quando $\epsilon = 0$, temos as folgas complementares ordinárias discutidas na seção 20.3.

Se o grafo (N, A) é simétrico e a função-custo c é anti-simétrica, as folgas de y e x são ϵ-complementares se e só se

$$\check{x}_{ij} < u_{ij} \quad \Rightarrow \quad y(j) - y(i) \leq c_{ij} + \epsilon \,.$$

A prova dessa afirmação é análoga à prova do lema 21.4. É claro que essa condição poderia também ser formulada assim: $y(j) - y(i) > c_{ij} + \epsilon \Rightarrow \check{x}_{ij} = u_{ij}$. lema 21.4

[1] Veja seção 10.3, pp.362–372, de AMO.
[2] Veja figura 10.2, p.363, de AMO.

24.2 O algoritmo

Ao contrário de todos os algoritmos anteriores, o algoritmo COST-SCALING usa uma função potencial y cujos valores não são necessariamente inteiros.[3] É fácil verificar que todos os resultados sobre folgas complementares valem para essa generalização.

```
COST-SCALING[4] (N, A, u, b, c)      ▷ c_ji = −c_ij
01  (x_0, T) ← FLUXO-VIÁVEL (N, A, u, b)
02  se x_0 não está definido
03      então devolva T
04  y ← 0
05  ϵ ← C ← max_{ij∈A} |c_ij|
06  enquanto ϵ ≥ 1/n faça
07      para cada ij em A faça
08          se y(j) − y(i) < c_ij
09              então x_ij ← 0
10              senão se y(j) − y(i) > c_ij
11                        então x_ij ← u_ij
12      PUSH-RELABEL ()
13      ϵ ← ϵ/2
14  devolva x

PUSH-RELABEL ()
15  para cada i em N faça
16      e(i) ← x(ī, i) − x(i, ī)
17  r ← PSEUDOFLUXO (x)
18  enquanto e(i) > b(i) para algum i faça
19      se existe arco tenso ij em A(i) tal que x̌_ij < u_ij
20          então δ ← min {e(i) − b(i), u_ij − x̌_ij}
21                x̌_ij ← x̌_ij + δ
22                x̌_ji ← x̌_ji − δ
23                e(i) ← e(i) − δ
24                e(j) ← e(j) + δ
25          senão y(i) ← y(i) − ϵ/2
26  x ← FLUXO (x̌)
```

Na linha 19, um arco ij é **tenso** se $y(j) - y(i) > c_{ij}$.

[3] Mas isso pode ser facilmente evitado: basta substituir c por nc antes de executar o algoritmo e trocar a linha 05 por $\epsilon \leftarrow 2^{\lceil \log_2 C \rceil}$.

[4] Figuras 10.3 e 10.4, pp.364–365, de AMO.

Ao contrário dos algoritmos anteriores, o potencial y não é necessariamente inteiro (ou seja, pode ter valores fracionários) ao longo da execução do algoritmo. Mas isso não tem quaisquer conseqüências,

Seja x o fluxo FLUXO $(\check{x})$. Então, no começo de cada iteração do bloco de linhas 07–13,

(i1) x respeita u;
(i2) x satisfaz b;
(i3) $y(j) - y(i) < c_{ij} - \epsilon \;\Rightarrow\; x_{ij} = 0$ para cada arco ij;
(i4) $y(j) - y(i) > c_{ij} + \epsilon \;\Rightarrow\; x_{ij} = u_{ij}$ para cada arco ij.

No começo de cada execução da rotina PUSH-RELABEL, x pode não satisfazer b mas respeita u e suas folgas são complementares com as de y Portanto (veja lema 21.4), $\check{x}_{ij} < u_{ij} \Rightarrow y(j) - y(i) \le c_{ij}$ para cada arco ij. No começo de cada iteração do bloco de linhas 19–25, 21.4

(i5) x respeita u;
(i6) $\check{x}_{ij} < u_{ij} \;\Rightarrow\; y(j) - y(i) \le c_{ij} + \epsilon/2$ para cada arco ij.

No fim de cada execução de PUSH-RELABEL, o fluxo x satistaz b (pois $e(i) \le b(i)$ para cada i, $e(N) = 0$ em virtude de (10.1) e $b(N) = 0$ uma vez que existe fluxo viável).

Última iteração. No início da última iteração teremos $\epsilon < 1/n$. Seja O um ciclo na rede $(N, A_{\check{x}})$, onde $A_{\check{x}} := \{ij : \check{x}_{ij} < u_{ij}\}$. Digamos que A_O é o conjunto de arcos e N_O o conjunto de nós de O. Então, em virtude das invariantes (i3) e (i4) combinados com o lema 21.4,

$$\begin{aligned}
c(O) &= \textstyle\sum_{ij \in A_O} c_{ij} \\
&= \textstyle\sum_{ij \in A_O} c_{ij} + \sum_{i \in N_O} y(i) - \sum_{j \in N_O} y(j) \\
&= \textstyle\sum_{ij \in A_O} c_{ij} + \sum_{ij \in A_O} (y(i) - y(j)) \\
&= \textstyle\sum_{ij \in A_O} (c_{ij} - (y(j) - y(i))) \\
&\ge \textstyle\sum_{ij \in A_O} (-\epsilon) \\
&= -\epsilon k \\
&\ge -\epsilon n \\
&> -1 \,.
\end{aligned}$$

Como os valores de c são inteiros, podemos concluir que $c(O) \ge 0$. Concluímos assim que todo ciclo na rede $(N, A_{\check{x}})$ tem custo não-negativo. De acordo com o lema 22.1 isso garante que o fluxo x é ótimo. lema 22.1

24.3 Consumo de tempo

O valor de ϵ varia de C a $1/n$ e é dividido por 2 a cada iteração. Logo, o número de iterações é o menor ι tal que $C/2^{\iota-1} < 1/n$. Portanto, o número de iterações não passa de

$$1 + \lfloor \log_2(nC) \rfloor \, .$$

Uma análise não-trivial revela que o consumo de tempo de cada iteração é $\mathrm{O}(n^2 m)$. Logo, o consumo de tempo total de COST-SCALING é

$$\mathrm{O}(n^2 m \log(nC)) \, .$$

Assim, o algoritmo é polinomial.

Exercícios

24.1 Prove as invariantes do algoritmo COST-SCALING.

24.2 Verifique que as linhas 08-11 do algoritmo COST-SCALING podem ser substituídas por

```
08        se y(j) − y(i) > c_ij
09            então x̆_ij ← u_ij
```

Capítulo 25

Algoritmo do ciclo de custo médio mínimo

Este capítulo[1] discute um algoritmo fortemente polinomial para o problema do fluxo viável de custo mínimo (problema 20.1).

25.1 Ciclos de custo médio mínimo

Dada uma rede arbitrária com função-custo c, o **custo médio** de um ciclo (dirigido) O é o número

$$\frac{c(O)}{|O|}\,.$$

Em outras palavras, o custo médio de O é o número (racional) α tal que $c(O) = \alpha|O|$. Exemplo: se O e O' são ciclos e $c(O) = c(O') < 0$ então o ciclo mais curto tem menor custo médio.

Eis uma observação útil. Digamos que α é positivo e $-\alpha$ é o custo médio de um ciclo O. Então $(c+\alpha)(O) = 0$, sendo $c+\alpha$ a função-custo definida da maneira óbvia: $(c+\alpha)_{ij} = c_{ij} + \alpha$. Ademais, $(c+\beta)(O) < 0$ para qualquer $\beta < \alpha$.

Esta seção trata do seguinte problema: Dada uma rede (N, E, c), encontrar um ciclo de custo médio mínimo. Convém denotar por $\mu_*(c)$ o custo médio de tal ciclo:

$$\mu_*(c) := \min_O \frac{c(O)}{|O|}\,.$$

Trataremos do problema apenas no caso em que a rede tem um ciclo de custo estritamente negativo; nesse caso, o problema tem solução e $c(O) < 0$ para qualquer solução O.

[1] Trata-se de um resumo da seção 10.5, pp.376–382, e da seção 5.7, pp.150–154, de AMO.

Além de supor que a rede tem um ciclo de custo negativo, o algoritmo abaixo supõe que c_{ij} é múltiplo de $2n^2$ para cada arco ij;[2] com isso, todos os números gerados pelo algoritmo serão inteiros.

```
MIN-MEAN-CYLE (N, E, c)    ▷ 2n² divide c
01  Δ1 ← 0
02  Δ2 ← max_{ij∈E} |c_ij|
03  enquanto Δ2 − Δ1 > 1 faça
04      Δ ← ⌊(Δ1 + Δ2)/2⌋
05      ⟨O, y⟩ ← CICLO-NEGATIVO (N, E, c + Δ)
06      se O está definido
07          então Δ1 ← Δ
08          senão Δ2 ← Δ
09  ⟨O, y⟩ ← CICLO-NEGATIVO (N, E, c + Δ1)    ▷ O está definido
10  devolva O
```

O algoritmo CICLO-NEGATIVO é qualquer dos algoritmos discutidos no capitulo 7 (FORD-BELLMAN, por exemplo). O algoritmo recebe uma rede (N, E) e uma função-custo (inteira) c' e devolve um ciclo O tal que $c'(O) < 0$ ou um c'-potencial y (que prova a inexistência de ciclo negativo). O algoritmo exige que seu terceiro argumento seja uma função inteira, e isso é garantido pela invariante (i1) abaixo. (i1)

O algoritmo MIN-MEAN-CYLE mantém as seguintes invariantes:

(i1) Δ_1 e Δ_2 estão em $\mathbb{Z}$ e $0 \le \Delta_1 < \Delta_2$;
(i2) $c(O)/|O| \ge -\Delta_2$ para todo ciclo O;
(i3) $c(O)/|O| < -\Delta_1$ para algum ciclo O.

A invariante (i2) pode ser reformulada assim: $(c + \Delta_2)(O) \ge 0$ para todo ciclo O. A invariante (i3) pode ser reformulada assim: $(c + \Delta_1)(O) < 0$ para algum ciclo O.

Última iteração. Digamos que um número (racional) α é *bom* se existe um ciclo cujo custo médio é α. No começo da última iteração, quando $\Delta_2 - \Delta_1 \le 1$, existe um só número bom menor que $-\Delta_1$. Eis a prova desse fato, por contradição. Suponha que existem dois números bons abaixo de $-\Delta_1$. Então existem ciclos O e O' tais que $c(O)/|O|$ e $c(O')/|O'|$ são diferentes e ambos menores que $-\Delta_1$. Como os valores de c são múltiplos de $2n^2$ e todo ciclo tem no máximo n nós,

$$1 \ge \Delta_2 - \Delta_1 \ge \left| \frac{c(O)}{|O|} - \frac{c(O')}{|O'|} \right| = \left| \frac{c(O)|O'| - c(O')|O|}{|O||O'|} \right| \ge \frac{2n^2}{n^2} = 2\,.$$

Essa contradição, juntamente com (i1) e (i3), prova nossa tese. (i1) (i3)

[2] Portanto, multiplique todos os custos por $2n^2$ antes de aplicar o algoritmo.

Como só existe um número bom abaixo de $-\Delta_1$, todos os ciclos negativos na rede $(N, E, c + \Delta_1)$ têm o mesmo custo médio na rede (N, E, c). Isso justifica as linhas 09–10 do algoritmo.

Consumo de tempo. O algoritmo MIN-MEAN-CYLE executa $1 + \lceil \lg C \rceil$ iterações, sendo $C := \max_{ij \in E} |c_{ij}|$. Cada execução de CICLO-NEGATIVO consome $O(nm)$ unidades de tempo (veja capitulo 7). Logo, MIN-MEAN-CYLE consome

$$O(nm \lg C)$$

unidades de tempo.

25.2 Ciclo de custo médio mínimo: programação dinâmica

Há um algoritmo para o problema do ciclo de cuto médio mínimo que é mais eficiente que o da seção anterior.

```
MIN-MEAN-CYLE-DYN-PROG (N, E, c)
01  para cada i em N faça
02      d[0, i] ← ∞
03  escolha s em N
04      d[0, s] ← 0
05  para λ ← 0 até n − 1 faça
06      para cada j em N faça
07          d[λ + 1, j] ← min_{ij ∈ Ã(j)} (d[λ, i] + c_ij))
08  para cada j em N faça
09      m[j] ← max_{0≤λ≤n−1}(d[n, j] − d[λ, j])/(n − λ)
10  μ ← max_{j∈N} m[j]
11  ⟨O, y⟩ ← CICLO-NEGATIVO (N, E, c − μ)     ▷ devolve y
12  E′ ← {ij ∈ E : y(j) − y(i) = c_ij − μ}
13  ⟨O′, y′⟩ ← DAG (N, E′)
14  devolva O′
```

Adote a seguinte notação: para cada λ entre 0 e n e cada nó j,

$$d(\lambda, j, c) = \min_{P \in \mathcal{P}} c(P)\,, \tag{25.1}$$

onde $\mathcal{P}$ é o conjunto de todos os *passeios* de comprimento λ que começam em s e terminam em j. Então é claro que no fim da execução do bloco de linhas 1–7 teremos

$$d[\lambda, j] = d(\lambda, j, c)\,.$$

Assim, é suficiente provar que

$$\mu_*(c) = \min_{j \in N} \max_{0 \le \lambda \le n-1} \frac{d(n,j,c) - d(\lambda,j,c)}{n-\lambda} \,. \tag{25.2}$$

Para qualquer racional Δ, seja $c - \Delta$ a função-custo definida por $(c-\Delta)_{ij} := c_{ij} - \nu$. É claro que $\mu_*(c-\Delta) = \mu_*(c) - \Delta$, É claro também que

$$\frac{d(n,j,c-\Delta) - d(\lambda,j,c-\Delta)}{n-\lambda} = \frac{(d(n,j,c) - d(\lambda,j,c)}{n-\lambda} - \Delta \,,$$

uma vez que $d(\lambda,j,c-\Delta) = d(\lambda,j,c) - \lambda\Delta$ para cada λ e cada j.

Agora tome $\Delta = \mu_*(c)$ e seja c^* a função-custo $c - \Delta$. É suficiente provar que (25.2) vale com c^* no lugar de c. .....

Consumo de tempo. O algoritmo consome

$$O(nm)$$

unidades de tempo. Vamos denotar esse algoritmo por MIN-MEAN-CYLE′.

25.3 Algoritmo para fluxo viável de custo mínimo

Podemos tratar agora do problema do fluxo viável de custo mínimo. Vamos supor, com base no fato 21.3, que nossa rede é simétrica e que a função-custo é anti-simétrica.

```
MIN-MEAN-KLEIN (N, A, u, b, c)     ▷ c_ji = −c_ij
01  ⟨x_0, T⟩ ← FLUXO-VIÁVEL (N, A, u, b)
02  se x_0 não está definido
03      então devolva T
04  r ← PSEUDOFLUXO (x_0)
05  repita
06      A_x̌ ← {ij ∈ A : x̌_ij < u_ij}
07      ⟨O, y⟩ ← CICLO-NEGATIVO (N, A_x̌, c)
08      se O está definido     ▷ c(O) < 0
09          então O ← MIN-MEAN-CYLE′ (N, A_x̌, c)
10                μ ← min {u_ij − x̌_ij : ij é arco de O}
11                para cada arco ij de O faça
12                    x̌_ij ← x̌_ij + μ
13                    x̌_ji ← x̌_ji − μ
14          senão x ← FLUXO (x̌)
15                devolva x e y
```

(Note que o ciclo O produzido por MIN-MEAN-CYLE′ tem comprimento $|O| \geq 3$, pois todos os ciclos de comprimento 2 têm custo nulo, dada a anti-simetria de c.)

A análise da correção do algoritmo é um tanto complexa e será omitida. exceto pelo seguinte esboço vago. Digamos que a **otimalidade** de um fluxo viável x é o menor $\epsilon > 0$ dotado da seguinte propriedade: existe um potencial y tal que

$$\begin{aligned} x_{ij} > 0 &\quad\Rightarrow\quad y(j) - y(i) \geq c_{ij} - \epsilon \\ x_{ij} < u_{ij} &\quad\Rightarrow\quad y(j) - y(i) \leq c_{ij} + \epsilon \end{aligned}$$

para cada arco ij. A correção do algoritmo se apoia no seguinte fato não trivial: no início de cada iteração a otimalidade de x é $-\mu_*$, sendo μ_* o custo de um ciclo de custo médio mínimo na rede $(N, A_{\check{x}}, c)$.

25.4 Consumo de tempo

A análise do consumo de tempo do algoritmo MIN-MEAN-KLEIN não é trivial e será omitida. Diremos apenas que o algoritmo consome

$$\mathrm{O}(n^2 m^3 \log n)$$

unidades de tempo.[3]

[3] Veja seção 10.5, pp.376–382, de AMO.

Capítulo 26

Circulações

Este capítulo[1] trata de uma importante generalização do problema do fluxo máximo (problema 11.1). O capítulo culmina com o teorema de Hoffman.

26.1 Circulações com delimitações inferiores

Como já dissemos no capítulo 10, uma **circulação** em uma rede é um fluxo que tem acúmulo nulo em cada nó. Diremos que uma circulação x **respeita** uma função-capacidade u se $x \leq u$.

Uma **função-limite-inferior** é qualquer função l de A em $\mathbb{Z}_{\geq}$. Uma circulação x **satisfaz** uma função l se $x \geq l$.

Problema 26.1 (da circulação viável) *Dado um arco ts em uma rede (N, A, l, u) com função-limite-inferior l e função-capacidade u, encontrar uma circulação x que respeite u e satisfaça l.*

Esse problema pode ser reduzido ao problema do fluxo máximo como explicaremos a seguir.

Se existe uma circulação que satisfaz l e respeita u então é evidente que $l \leq u$. Além disso, em virtude do lema 10.1, para qualquer parte T de N,

$$l(T, \overline{T}) \;\leq\; u(\overline{T}, T)\,. \tag{26.1}$$

Estas condições são, portanto, necessárias para a existência de uma circulação viável. As condições também são suficientes:

[1] Resumo da seção 6.7, p.191, do AMO.

Teorema 26.2 (Hoffman, 1960) *Uma rede* (N, A, l, u) *tem uma circulação viável se e só se*[2] $l \leq u$ *e*

$$l(T, \overline{T}) \;\leq\; u(\overline{T}, T)$$

para cada parte T *de* N*.*

A prova decorre do teorema do fluxo máximo e corte mínimo (teorema 11.5).

26.2 Apêndice: Fluxos com delimitações inferiores

Suponha que acrescentamos mais uma exigência ao problema do fluxo máximo (problema 11.1): o fluxo em cada arco ij deve valer pelo menos l_{ij}, onde l é uma função-limite-inferior. Como resolver essa generalização do problema?

Exercícios

26.1 [Teorema de Hoffman] Mostre que ambas as condições enunciadas no teorema de Hoffman (teorema 26.2) são necessárias. Mostre que as condições são suficientes.

26.2 Escreva um algoritmo que receba uma rede (N, A, l, u), com função-capacidade u e função-limite-inferior l, e devolva (1) uma circulação que satifaz l e respeita u ou (2) uma prova de que uma tal circulação não existe. Escreva duas versões do algoritmo: uma usa o algoritmo FORD-FULKERSON como "caixa preta" enquanto a outra faz uma adaptação apropriada do FORD-FULKERSON.

26.3 Resolva o problema sugerido na seção 26.2.

[2] Na seção 6.8, teorema 6.11, AMO erra ao omitir a primeira condição.

Bibliografia

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993. 1

[CCPS98] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley, 1998. 1

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1991. 1

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001. 1

[Knu93] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press and Addison-Wesley, 1993. 39, 48

Lista de Tabelas

Índice