Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Marcos Massayuki Kawakami

Algoritmos em redes de fluxo e aplicações

São Paulo
Novembro de 2017

Algoritmos em redes de fluxo e aplicações

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: José Coelho de Pina

São Paulo
Novembro de 2017

Resumo

O tema de fluxos em rede é recorrente em competições de programação. No entanto, a quantidade de material em português sobre o tema é escassa e raramente aborda tanto a parte teórica quanto a parte prática do assunto. Este trabalho tem como objetivo amenizar este problema, servindo de material didático sobre fluxos em rede para os interessados sobre o tema, com ênfase na resolução de problemas de programação competitiva. São apresentados a teoria elementar sobre o tema, algoritmos e suas respectivas implementações, bem como aplicações e extensões do problema do fluxo máximo e resoluções de problemas de competições passadas.

Palavras-chave: rede, fluxo, algoritmo, implementação, aplicação, resolução de problemas, programação competitiva.

Sumário

Capítulo 1

Introdução

1.1 Grafos

Um **grafo** (dirigido) é um par ordenado (V, E), onde V é um conjunto finito e E é um conjunto de pares ordenados de elementos de V. Chamamos de **vértices** os elementos de V e **arestas** os elementos de E.

Dado um grafo $G = (V, E)$, usaremos as expressões "vértice v em G" e "aresta e em G" para significar, respectivamente, v em V e e em E.

Para cada aresta (v, w) em E, chamamos v de **origem** e w de **destino** da aresta. Dizemos que a aresta **entra** (ou chega) em w e **sai** de v. A aresta (w, v) é chamada de **aresta reversa** de (v, w).

Um **caminho não-dirigido** de s a t é uma sequência alternante $(s = v_0, e_1, v_1, e_2, \ldots, e_k, v_k = t)$ de vértices e arestas tal que para cada aresta e_i, temos ou $e_i = (v_{i-1}, v_i)$ ou $e_i = (v_i, v_{i-1})$. No primeiro caso, a aresta é chamada de **direta**. No segundo caso, é chamada de **inversa**. O vértice s é a **origem** do caminho e o vértice t é o seu **destino**. Dizemos que o vértice t é **alcançável** a partir de s. O **comprimento** de um caminho é o número de arestas que ele contém. Um caminho não-dirigido cujas arestas são todas diretas é chamado simplesmente de **caminho** (de v_0 a v_k).

Num contexto que não cause ambiguidade, um caminho $(v_0, e_1, v_1, \ldots, e_k, v_k)$ pode ser representado por sua sequência de vértices $(v_0, v_1, \ldots, v_k)$.

Um caminho é dito **simples** quando seus vértices são todos distintos.

Um **ciclo** é um caminho $(v_0, v_1, \ldots, v_k)$ tal que $v_0 = v_k$ e os vértices $v_0, v_1, \ldots, v_{k-1}$ são distintos dois a dois. Um grafo que não possui ciclos é chamado de **acíclico**.

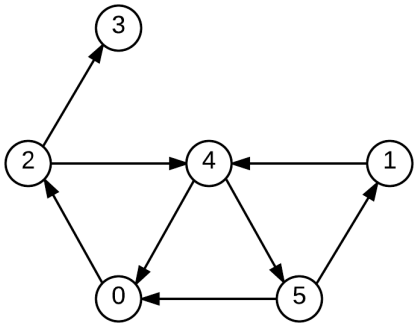

Figura 1.1: Um exemplo de grafo. Como exemplo de caminho e ciclo temos, respectivamente, $(1, 4, 0, 2, 4, 5)$ e $(5, 0, 2, 4, 5)$.

Dado um grafo $G = (V, E)$ e um subconjunto R de V, denotamos por R^c o conjunto

$V \setminus R$. Dizemos que uma aresta (v, w) **entra** em R quando $v \in R^c$ e $w \in R$. Analogamente, (v, w) **sai** de R quando $v \in R$ e $w \in R^c$. Denotamos por $E^-(R)$ o conjunto de todas as arestas que saem de R e por $E^+(R)$ o conjunto de todas as arestas que entram em R.

Para todo vértice v de G, usamos a abreviatura $E^-(v)$ para $E^-(\{v\})$, o conjunto das arestas que saem de v, e $E^+(v)$ para $E^+(\{v\})$, o conjunto das arestas que chegam em v.

Grafos não-dirigidos

Um **grafo não-dirigido** é um par (V, E) em que V é finito e E é um conjunto de pares não ordenados de elementos de V. Assim como em grafos dirigidos, os elementos de V são chamados de **vértices** e os elementos de E de **arestas**. Dada uma aresta $e = (v, w)$, dizemos que v e w são **pontas** de e e que e **incide** em v e w.

As definições de caminho e ciclo em grafos não-dirigidos são análogas às definições para grafos dirigidos.

Em grafos não-dirigidos, a relação "existe caminho de s a t" é de equivalência. As classes de equivalência desta relação são chamadas de **componentes conexas**. Dizemos que um grafo não-dirigido é **conexo** quando consiste de uma única componente conexa.

Uma **árvore** é um grafo não-dirigido conexo e acíclico.

Dado um grafo não-dirigido $G = (V, E)$ e um subconjunto A de V, denotamos por $E(A)$ o conjunto das arestas de G que possuem uma ponta em A e outra em A^c. Se B é outro subconjunto de V disjunto de A, denotamos por $E(A, B)$ o conjunto das arestas que possuem uma ponta em A e outra em B. Se v é elemento de V, então $E(v)$ denota o conjunto das arestas que incidem em v.

1.2 Redes

Chamaremos de **rede** um grafo munido de uma ou mais funções que atribuem valores numéricos a vértices ou arestas do grafo.

Seja $G = (V, E, x)$ uma rede em que x é uma função que atribui valores numéricos a cada aresta de G. Dado um subconjunto A de V, denotaremos por $x^+(A)$ a soma dos valores de x das arestas que entram em A, e por $x^-(A)$ a soma dos valores de x das arestas que saem de A:

$$x^+(A) = \sum_{e \in E^+(A)} x(e), \quad x^-(A) = \sum_{e \in E^-(A)} x(e).$$

Dado um vértice v de G, os valores $x^+(\{v\})$ e $x^-(\{v\})$ também podem ser denotados, respectivamente, por $x^+(v)$ e $x^-(v)$.

Dado dois subconjuntos A e B de V, a soma dos valores de x das arestas que saem de A e entram em B será denotada por $x(A, B)$:

$$x(A, B) = \sum_{e \in E(A,B)} x(e).$$

Aqui também podemos representar um conjunto unitário pelo elemento que ele contém. Por exemplo, se v e w são vértices, então $x(v, w)$ indica o valor de x da aresta (v, w) quando ele existe e 0 caso contrário.

O grafo de uma rede pode ser não-dirigido. Neste caso, dizemos que a rede é **não-dirigida**. Caso contrário, a rede é **dirigida**. Quando não há especificação, subentende-se que a rede é dirigida.

As definições para redes dirigidas, quando aplicáveis, são definidas de forma análoga para redes não-dirigidas. Por exemplo, para redes não-dirigidas, $x(A, B) = x(B, A)$ é a soma dos valores de x das arestas com uma ponta em A e outra em B.

Quando G é uma rede não-dirigida, denotaremos por $x(A)$ a soma dos valores de x das arestas de $E(A)$. Se v é um vértice de G, $x(\{v\})$ também pode ser denotado por $x(v)$.

Cortes

Dada uma rede $G = (V, E, x)$, um **corte** de G é uma partição (S, T) dos vértices de G. Como $T = S^c$, um corte pode ser representado também por um subconjunto S de vértices. Todo corte (S, T) está associado ao conjunto $E(S, T)$, as arestas que saem de S e entram em T.

Se $s \in S$ e $t \in T$, dizemos que s e t estão em **lados opostos** ou **lados diferentes** do corte (S, T). Dizemos também que (S, T) é um s-t **corte**, e que (S, T) separa s de t. Quando $v, w \in S$ ou $v, w \in T$, dizemos que v e w estão do **mesmo lado** do corte (S, T).

As definições acima podem ser estendidas para grafos não-dirigidos de forma análoga.

Contração de vértices

Seja $G = (V, E, x)$ uma rede, sendo x uma função que atribui valores numéricos a cada aresta de G, e seja S um subconjunto não-vazio de V.

A **contração** de S em G é uma rede $G' = (V', E', x')$ resultante da seguinte transformação de G:

- O conjunto de vértices de G' é $V' = (V \setminus S) \cup \{s\}$, sendo s um novo vértice;
- As arestas (v, w) de E, com $v, w \notin S$, permanecem em E' com $x'(v, w) = x(v, w)$;
- As arestas (v, w) de E, com $v, w \in S$, não aparecem em E';
- Para todo vértice v fora de S, todas as arestas (v, w), com $w \in S$, são substituidas em G' por uma aresta (v, s) com $x'(v, s) = x(v, S)$. Similarmente, todas as arestas (w, v), com $w \in S$, são substituidas em G' por uma aresta (s, v) com $x'(s, v) = x(S, v)$.

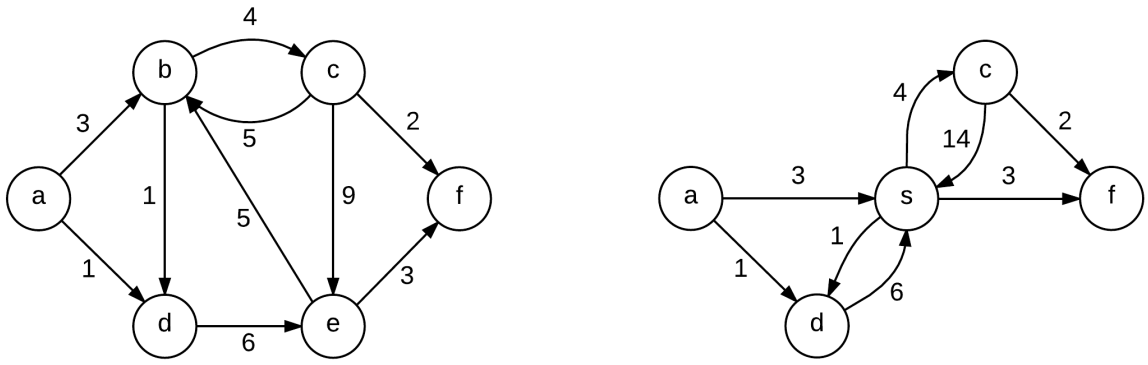

Figura 1.2: À esquerda, uma rede G. À direita, a contração de $S = \{b, e\}$ em G.

Seja (A', B') um corte de G'. O corte (A', B') **induz** o corte (A, B) de G, definido da seguinte maneira:

- Se $v \in S$, então $v \in A \Leftrightarrow s \in A'$;
- Se $v \in V \setminus S$, então $v \in A \Leftrightarrow v \in A'$.

Reciprocamente, dizemos que o corte (A, B) de G **induz** o corte (A', B') de G'.

Se G'' é o resultado de uma sequência de contrações em G e (A'', B'') é um corte de G'', podemos definir o **corte induzido** por (A'', B'') em G de maneira indutiva aplicando a definição anterior.

Para grafos não-dirigidos, a contração de vértices pode ser definida de forma análoga.

Capítulo 2

Fluxos máximos

Iniciaremos este capítulo com a introdução de definições e notações pertinentes ao contexto de redes de fluxo. Em seguida, vamos enunciar o problema do fluxo máximo e obter alguns resultados importantes que formarão a base dos algoritmos a serem estudados.

Nosso objeto de estudo neste capítulo serão redes da forma $G = (V, E, u)$, em que $u : E \to \mathbb{Q}^+$ é uma função que associa um valor racional[1] não-negativo para cada aresta de G, chamada de **função-capacidade**. Uma rede munida de uma função-capacidade será chamada de **rede capacitada**.

Seja $G = (V, E, u)$ uma rede capacitada e sejam s um vértice denominado **fonte** e t um vértice diferente de s denominado **sorvedouro**. Um s-t **fluxo** (ou fluxo de s a t) é uma função $f : E \to \mathbb{Q}^+$ que atribui para cada aresta um valor não-negativo, satisfazendo as seguintes propriedades:

1. (*Condições de capacidade*) Para cada aresta $e \in E$, temos
$$0 \leq f(e) \leq u(e);$$

2. (*Condições de conservação*) Para cada vértice v diferente de s e t, a soma do fluxo das arestas chegando em v deve ser a mesma da soma do fluxo das arestas saindo de v, ou seja,
$$f^+(v) = f^-(v).$$

Em geral, omitiremos a referência a s e t e usaremos somente o termo **fluxo** quando o contexto deixa claro quais são os vértices fonte e sorvedouro.

Para todo vértice v definimos seu **excesso**, denotado por $e(v)$, como a diferença do fluxo entrando e saindo de v, ou seja, $e(v) = f^+(v) - f^-(v)$. Dessa forma, a propriedade de conservação do fluxo pode ser reescrita como $e(v) = 0$, para todo vértice v diferente de s e t.

O **valor** ou **intensidade** de um fluxo f de s a t, denotado por $|f|$, é $e(t) = -e(s)$. Para verificar que estes dois valores são, de fato, iguais, note primeiramente que

$$\sum_{v \in V} e(v) = \sum_{v \in V} \left(\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) \right) = \sum_{e \in E} f(e) - \sum_{e \in E} f(e) = 0.$$

Pela propriedade da conservação do fluxo, $e(v) = 0$ para todo vértice v diferente de s e t. Logo, $e(s) + e(t) = 0$, donde $e(t) = -e(s)$.

[1] Poderia-se definir uma rede de fluxo com uma função c com contra-domínio nos reais não-negativos. Porém, isto trás complicações não relevantes no contexto de competições de programação. Veja, por exemplo, a seção 2.6.

Dada uma aresta e, dizemos que esta aresta está **saturada** quando $f(e) = u(e)$.
Definimos a capacidade $\text{cap}(S)$ de um s-t corte S como sendo

$$\text{cap}(S) := u(S, S^c) = u^-(S).$$

2.1 O problema

O problema do fluxo máximo consiste no seguinte:

Problema (Fluxo máximo). *Dada uma rede capacitada $G = (V, E, u)$ e dois vértices s e t de G, encontrar um fluxo de s a t de valor máximo.*

Um fluxo de valor máximo é chamado de **fluxo máximo**. Note que o fluxo máximo pode não ser único.

Nesta seção, adotaremos a simplificação de que as capacidades das arestas de uma rede são todas inteiras. Note que isto não trás nenhum prejuízo, uma vez que uma instância do problema com capacidades racionais pode ser transformada em outra equivalente com capacidades inteiras multiplicando todos as capacidades da instância original pelo mínimo múltiplo comum de seus denominadores.

Uma primeira abordagem a esse problema é tentar encontrar algum limitante superior para o valor máximo de um fluxo. Dado um s-t corte S, é intuitivo pensar que, como cada unidade de fluxo "flui" de s para t, eventualmente ela deve passar por uma aresta que sai de S. Assim, todo fluxo estaria limitado pelo valor da capacidade de S. As proposições a seguir concretizam esta linha de pensamento.

Proposição 2.1. *Seja $G = (V, E, x)$ uma rede, sendo x uma função qualquer que atribui valores a cada aresta de G. Então, para todo subconjunto $A \subseteq V$, temos*

$$\sum_{v \in A} (x^+(v) - x^-(v)) = x^+(A) - x^-(A).$$

Demonstração. Vamos reescrever esta soma de outra maneira. Seja e uma aresta da rede. Temos quatro casos possíveis:

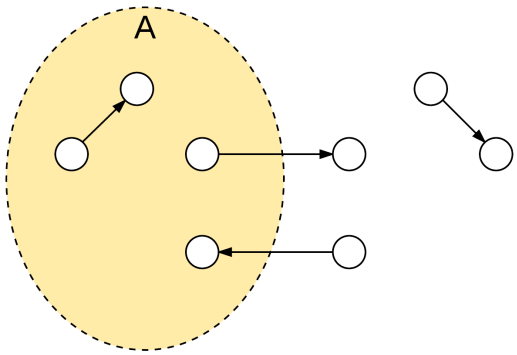

Figura 2.1: Os quatro casos possíveis da proposição 2.1

- Se tanto o vértice-origem quanto o vértice-destino de e estão em A, então $x(e)$ aparece na soma uma vez com sinal positivo e outra com sinal negativo, e portanto estes termos se cancelam;

- Se somente o vértice-origem está em A, então $x(e)$ aparece uma vez na soma com sinal positivo;
- Se somente o vértice-destino está em A, então $x(e)$ aparece uma vez na soma com sinal negativo;
- Se ambos os vértices estão fora de A, então $x(e)$ não aparece na soma.

Assim, a soma pode ser reescrita em termos das arestas da rede:

$$\sum_{v \in A}(x^+(v) - x^-(v)) = \sum_{e \in E^+(A)} x(e) - \sum_{e \in E^-(A)} x(e) = x^+(A) - x^-(A).$$

□

Proposição 2.2. *Seja f um s-t fluxo e S um s-t corte. Então, $|f| = f^-(S) - f^+(S)$.*

Demonstração.

$$\begin{aligned}
|f| &= -e(s) && \text{(definição)} \\
&= -\sum_{v \in S} e(v) && \text{(pois } e(v) = 0, \forall v \in S \setminus \{s\}) \\
&= -\sum_{v \in S}(f^+(v) - f^-(v)) && \text{(definição)} \\
&= -(f^+(S) - f^-(S)) && \text{(proposição 2.1)} \\
&= f^-(S) - f^+(S).
\end{aligned}$$

□

A proposição seguinte dá um limitante superior para o valor de qualquer fluxo em termos dos cortes de uma rede:

Proposição 2.3. *Seja f um s-t fluxo e S um s-t corte. Então, $|f| \leq \text{cap}(S)$.*

Demonstração. Pela proposição anterior, $|f| = f^-(S) - f^+(S)$. Como f é uma função não-negativa, temos que

$$f^-(S) - f^+(S) \leq f^-(S) = \sum_{e \in E^-(S)} f(e).$$

Como f respeita as condições de capacidade, vale que

$$\sum_{e \in E^-(S)} f(e) \leq \sum_{e \in E^-(S)} u(e) = \text{cap}(S).$$

□

A proposição 2.3 nos diz que o valor de *qualquer s-t* fluxo é limitado superiormente pela capacidade de *qualquer s-t* corte. Como consequência, se existir um fluxo f e um corte S tais que a igualdade $|f| = \text{cap}(S)$ vale, então a proposição nos garante que $|f|$ tem valor máximo e $\text{cap}(S)$ tem valor mínimo.

Surge uma questão interessante. Sempre existe um fluxo f e um corte S para os quais a igualdade vale? A resposta é afirmativa, conforme o teorema a seguir.

Teorema 2.1 (do fluxo máximo e corte mínimo). *Seja $G = (V, E, u)$ uma rede com capacidades inteiras e sejam s e t vértices distintos de G. Então, existem um s-t fluxo f e um s-t corte S tais que $|f| = \text{cap}(S)$.*

Provaremos este teorema em duas partes. Primeiro, apresentaremos um algoritmo que recebe uma rede e devolve um fluxo f. Em seguida, mostraremos que f é máximo exibindo um corte de capacidade $|f|$, que implicará na validade do teorema.

2.2 Algoritmo de Ford-Fulkerson

Antes de apresentar o algoritmo, introduziremos um conceito que será fundamental para seu funcionamento.

Seja $G = (V, E, u)$ uma rede capacitada e f um fluxo em G. Um **caminho de aumento** em G com relação a f é um caminho não-dirigido tal que toda aresta direta não está saturada e toda aresta inversa tem fluxo positivo.

O caminho de aumento tem este nome pois caso uma rede possua um caminho de aumento de s a t, o valor do fluxo pode ser aumentado, conforme a proposição a seguir.

Proposição 2.4. *Seja $G = (V, E, u)$ uma rede capacitada com fluxo f e um caminho de aumento P de s a t. Seja ε o maior número tal que $f(e) + \varepsilon \leq u(e)$ para toda aresta direta de P e $\varepsilon \geq f(e)$ para toda aresta inversa. Então, a função $f' : E \to \mathbb{Q}^+$ definida por*

$$f'(e) = \begin{cases} f(e) + \varepsilon & \text{se } e \text{ é aresta direta de } P \\ f(e) - \varepsilon & \text{se } e \text{ é aresta inversa de } P \\ f(e) & \text{caso contrário} \end{cases}$$

um fluxo de G de valor $|f'| = |f| + \varepsilon > |f|$.

Demonstração. Note primeiramente que $\varepsilon > 0$, pois para toda aresta direta temos $f(e) < u(e)$ e para toda aresta inversa temos $f(e) > 0$. Assim, mostraremos que f' é um fluxo, ou seja, respeita as condições de capacidade e conservação, e que $|f'| = |f| + \varepsilon$.

Seja e uma aresta da rede. Se P não passa por e, então $0 \leq f'(e) = f(e) \leq u(e)$. Se e é uma aresta direta, então pela definição de ε, $f'(e) = f(e) + \varepsilon \leq u(e)$. Note que, como P é caminho de aumento, então $\varepsilon > 0$, de modo que $f'(e) = f(e) + \varepsilon > 0$. Similarmente, se e é uma aresta inversa, então $u(e) > f'(e) = f(e) - \varepsilon \geq 0$. Assim, f' satisfaz as condições de capacidade.

Para verificar que f' satisfaz as condições de conservação, suponha que $P = (s = v_0, v_1, \ldots, v_k = t)$. Note que é suficiente verificar a conservação para os vértices de P de v_1 a v_{k-1}.

Seja v_i um desses vértices intermediários de P. Estamos alterando o fluxo de exatamente duas arestas que chegam ou saem de v_i: a que liga v_i com v_{i-1} e a que liga v_i com v_{i+1}. Se a aresta que liga v_i com v_{i-1} for direta, entao o acréscimo de ε nesta aresta contribui com um aumento de ε no excesso de v_i. O mesmo ocorre se a aresta for inversa.

Por outro lado, a aresta que liga v_i com v_{i+1} contribui com uma redução de ε no excesso de v_i. Logo, f' satisfaz as condições de conservação, portanto é fluxo.

Finalmente, o acréscimo/decréscimo de ε na aresta que liga v_{k-1} com $v_k = t$ aumenta o excesso em t em ε. Portanto, $|f'| = |f| + \varepsilon$. □

O valor ε da proposição 2.4 é chamado de **capacidade** do caminho de aumento P.

A ideia de caminhos de aumento sugere uma maneira bastante natural para encontrar um fluxo de valor máximo: partimos de um fluxo qualquer e repetidamente o incrementamos usando caminhos de aumento. A seguir apresentamos um algoritmo que recebe uma rede capacitada $G = (V, E, u)$ e dois vértices s e t e determina um fluxo máximo f utilizando esta ideia.

1. Inicialmente, f é uma função que atribui valor 0 para todas as arestas de G. Note que f é um s-t fluxo.
2. Enquanto houver caminho de aumento de s a t, repita o passo 3.
3. Seja P um caminho de aumento de s a t. Seja f' a função como definida na proposição 2.4. Atualize f com os valores de f'.
4. Devolva f.

O algoritmo anterior é bastante intuitivo, porém não é claro se o fluxo f devolvido é, de fato, máximo. Pior ainda, não é nem evidente que o algoritmo se quer termina! Mostraremos, no entanto, que o algoritmo funciona corretamente quando as capacidades são inteiras. Este é o chamado **algoritmo de Ford-Fulkerson**.

Proposição 2.5. *Seja $G = (V, E, u)$ uma rede com capacidades inteiras. Então, em toda iteração do algoritmo de Ford-Fulkerson aplicado à rede G, o fluxo f e o incremento ε do caminho de aumento são inteiros.*

Demonstração. Suponha que no início de uma iteração do algoritmo o fluxo f seja inteiro, e seja P o caminho de aumento escolhido no passo 3. Seja D o conjunto das arestas diretas de P e I o conjunto das arestas inversas de P. Então,

$$\varepsilon = \min(\{u(e) - f(e) : e \in D\} \cup \{f(e) : e \in I\}).$$

Logo, ε é o mínimo de um conjunto de números inteiros, portanto é inteiro. Por consequência, o fluxo f' também é inteiro. Portanto, f se mantém inteiro na próxima iteração.

Como f é inteiro na primeira iteração do algoritmo, por indução no número de iterações concluimos que em toda iteração do algoritmo, o fluxo f é inteiro, assim como os incrementos ε de cada caminho de aumento. □

Proposição 2.6. *Sejam $G = (V, E, u)$ uma rede com capacidades inteiras e s e t dois vértices distintos de G. Seja $C = u^-(s)$. Então, o algoritmo de Ford-Fulkerson aplicado à rede G com os vértices s e t executa o passo 3 no máximo C vezes.*

Demonstração. Em toda iteração do algoritmo, f é um s-t fluxo. Então, f satisfaz as condições de capacidade. Em particular, temos $f^-(s) \leq u^-(s) = C$. Logo,

$$|f| = f^-(s) - f^+(s) \leq f^-(s) = C.$$

Portanto, $|f|$ está limitado superiormente por C.

Pela proposição 2.5, o incremento ε de cada iteração do algoritmo é inteiro. Além disso, ε é positivo, então o valor de f aumenta em pelo menos uma unidade a cada iteração. Como $|f|$ não excede C, concluimos que o algoritmo deve terminar em no máximo C iterações. □

Proposição 2.7. *Seja $G = (V, E, u)$ uma rede capacitada e s e t dois vértices de G. Se f é um s-t fluxo tal que não existe caminho de aumento de s a t em G, então existe um s-t corte S tal que $|f| = \text{cap}(S)$.*

Demonstração. Seja S o conjunto dos vértices alcançáveis a partir de s por caminhos de aumento. Note que t não está em S, portanto S define um s-t corte.

Note que todas as arestas (v, w) de $E^-(S)$ estão saturadas, caso contrário w estaria em S. Pelo mesmo motivo, todas as arestas de $E^+(S)$ carregam fluxo zero. Logo,

$$\begin{aligned} |f| &= f^-(S) - f^+(S) \quad \text{(proposição 2.2)} \\ &= u^-(S) - 0 \\ &= \text{cap}(S). \end{aligned}$$

□

A proposição 2.6 garante que o algoritmo de Ford-Fulkerson termina sua execução após um número finito de passos. Além disso, como o fluxo f devolvido pelo algoritmo satisfaz as condições da proposição 2.7, existe um s-t corte cuja capacidade é igual ao valor de f, portanto f é máximo, pela proposição 2.3.

A corretude do algoritmo de Ford-Fulkerson trás duas consequências bastante importantes. A primeira consequência é a conclusão da demonstração do Teorema do Fluxo Máximo e Corte Mínimo (teorema 2.1), mencionado anteriormente:

Demonstração do teorema 2.1. Seja f o s-t fluxo devolvido pelo algoritmo de Ford-Fulkerson aplicado à rede G. Como não há caminhos de aumento em G com relação a f, pela proposição 2.7, existe um s-t corte S tal que $|f| = \text{cap}(S)$. □

A segunda consequência é um corolário imediato da proposição 2.5, que garante a existência de um fluxo máximo integral:

Corolário 2.1. *Seja $G = (V, E, u)$ uma rede com capacidades inteiras. Então, G admite um fluxo máximo inteiro.*

A importância do corolário anterior está no fato que ele abre portas para a aplicação de algoritmos de fluxo máximo em problemas cuja solução envolve somente números inteiros. Os capítulos 6 e 7 dedicam-se ao estudo de aplicações de fluxo máximo na resolução de problemas deste tipo.

2.3 Algoritmo de Edmonds-Karp

O algoritmo de Ford-Fulkerson não estabelece um critério para a escolha de um caminho de aumento quando existe mais de um possível. Como consequência, o número de iterações realizadas pode ser muito grande.

Considere, por exemplo, a rede da figura 2.2 (a). É fácil verificar que o fluxo máximo nesta rede tem valor 200. No entanto, escolhendo caminhos de aumento ruins, o algoritmo de Ford-Fulkerson pode demorar bastante para chegar neste valor. Se escolhermos o caminho indicado na figura (b) na primeira iteração, o valor do fluxo aumentará em somente uma unidade. Se na segunda iteração o caminho escolhido for o da figura (c), o fluxo aumentará novamente em uma unidade. Se alternarmos a partir daqui os caminhos indicados por (b) e (c), atingiremos o fluxo máximo somente após 200 iterações! Em contrapartida, se escolhessemos os caminhos de aumento (a, b, d) e (a, c, d), o algoritmo terminaria em apenas 2 iterações.

Uma forma de evitar escolhas ruins de caminhos de aumento é escolher, dentre todos os possíveis, um de comprimento mínimo. Mostraremos que se as escolhas de caminhos de aumento satisfizerem esta condição, o número de iterações do algoritmo não excede $|V||E|$.

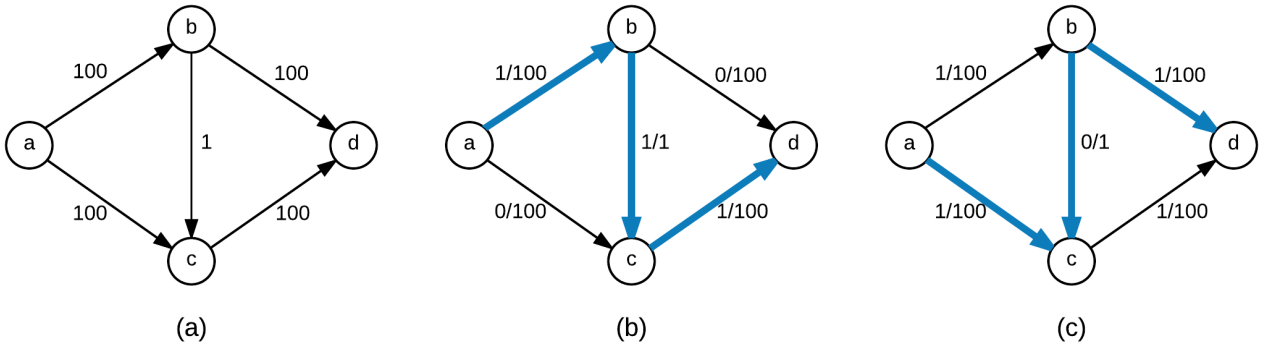

Figura 2.2: Uma rede capacitada e dois caminhos de aumento ruins. Em (a), as capacidades estão indicadas pelos números nas arestas. Em (b) e (c), o fluxo $f(e)$ e a capacidade $u(e)$ de cada aresta estão representados por $f(e)/u(e)$.

O algoritmo de Ford-Fulkerson com esta adaptação é chamado de **algoritmo de Edmonds-Karp**.

Para provar o limite do número de iterações do algoritmo de Edmonds-Karp, iremos utilizar uma ferramenta que será útil também nas implementações dos algoritmos de fluxo máximo baseados em caminhos de aumento.

Sejam $G = (V, E, u)$ uma rede capacitada e f fluxo. A **rede residual** de G com relação a f é uma rede capacitada $G_f = (V, E', u')$, em que u' é uma função chamada **capacidade residual**, construída da seguinte maneira:

- Para toda aresta (v, w) de G, se $f(v, w) < u(v, w)$, então G_f tem uma aresta (v, w) de capacidade residual $u'(v, w) = u(v, w) - f(v, w)$. Estas são chamadas **arestas diretas** de G_f.
- Para toda aresta (v, w) de G, se $f(v, w) > 0$, então G_f tem uma aresta (w, v) de capacidade residual $u'(w, v) = f(v, w)$. Estas são chamadas **arestas inversas** de G_f.

A rede residual possui duas propriedades importantes de fácil verificação:

- Todo caminho de aumento de G com relação a um s-t fluxo f corresponde a um caminho (dirigido) de s a t em G_f de mesmo comprimento e vice-versa;
- A capacidade de um caminho de aumento de G é igual ao mínimo das capacidades residuais das arestas do caminho correspondente em G_f.

Como existe esta correspondência biunívoca, podemos convencionar que um algoritmo que escolhe um caminho de aumento em G escolhe também um caminho de s a t em G_f, e vice-versa.

Proposição 2.8. *Para todo vértice $v \in V$, o comprimento do menor caminho de s a v na rede residual G_f nunca diminui de uma iteração para a seguinte.*

Demonstração. A afirmação é trivialmente verdadeira para $v = s$. Provaremos a proposição para $V \setminus \{s\}$ por contradição. Seja f o s-t fluxo de uma iteração e f' o s-t fluxo da iteração seguinte. Denote por $d_f(v, w)$ o comprimento do menor caminho, em número de arestas, de v a w em G_f. Defina $d_{f'}$ de modo análogo para a rede $G_{f'}$.

Seja w o vértice de menor $d_{f'}(s, w)$ tal que $d_f(s, w) > d_{f'}(s, w)$. Seja v o penúltimo vértice de algum caminho mínimo de s a w em $G_{f'}$. Então

$$d_{f'}(s, v) = d_{f'}(s, w) - 1. \tag{2.1}$$

Pela escolha de w, temos

$$d_{f'}(s,v) \geq d_f(s,v). \tag{2.2}$$

Suponha, por um instante, que (v,w) é uma aresta de G_f. Então, teríamos

$$\begin{aligned} d_f(s,w) &\leq d_f(s,v) + 1 \\ &\leq d_{f'}(s,v) + 1 \quad \text{(por (2.2))} \\ &= d_{f'}(s,w) \quad \text{(por (2.1))}, \end{aligned}$$

contrariando a hipótese de que $d_f(s,w) > d_{f'}(s,w)$.

Portanto, temos $(v,w) \in G_{f'}$ e $(v,w) \notin G_f$. Isto só pode ocorrer de duas formas: $(v,w) \in E$, $f(v,w) = u(v,w)$ e $f'(v,w) < u(v,w)$, ou $(w,v) \in E$, $f(w,v) = 0$ e $f'(w,v) > 0$. Em ambos os casos, o caminho escolhido pelo algoritmo deve passar pela aresta (w,v) em G_f. Como o algoritmo escolhe apenas caminhos de comprimento mínimo, devemos ter

$$\begin{aligned} d_f(s,w) &= d_f(s,v) - 1 \\ &\leq d_{f'}(s,v) - 1 \quad \text{(por (2.2))} \\ &= d_{f'}(s,w) - 2 \quad \text{(por (2.1))}, \end{aligned}$$

novamente contradizendo a hipótese de que $d_f(s,w) > d_{f'}(s,w)$.

Portanto, tal vértice w não existe. Em particular, w não pode ser t, logo o comprimento de um caminho de aumento mínimo não pode diminuir de uma iteração para a seguinte. □

Teorema 2.2. *Sejam $G = (V,E,u)$ uma rede capacitada e s e t dois vértices distintos de G. Então, o algoritmo de Edmonds-Karp aplicado à rede G com os vértices s e t busca por caminho de aumento no máximo $|V||E|$ vezes.*

Demonstração. Sejam f o fluxo em uma iteração do algoritmo e P o caminho em G_f escolhido nesta iteração. Chamaremos uma aresta $(v,w) \in G_f$ de *crítica* se P passa por (v,w) e a capacidade residual de (v,w) é igual à capacidade de P.

Note que, ao aumentar o fluxo ao longo de P, todas as arestas críticas desaparecem de G_f. Além disso, em toda iteração do algoritmo existe ao menos uma aresta crítica. Mostraremos, então, que uma aresta pode ser crítica no máximo $|V|/2$ vezes. Por consequência, como G_f pode ter no máximo $2|E|$ arestas distintas, o algoritmo pode escolher no máximo $|V||E|$ caminhos.

Seja (v,w) uma aresta de G_f. Como o algoritmo escolhe somente caminhos mínimos, quando (v,w) se torna crítica pela primeira vez, temos

$$d_f(s,w) = d_f(s,v) + 1. \tag{2.3}$$

Após o fluxo ser aumentado, a aresta (v,w) é removida de G_f. Esta aresta só volta a fazer parte de G_f se em alguma iteração seguinte o algoritmo escolher um caminho em $G_{f'}$ que passa pela aresta (w,v). Nesta ocasião, temos

$$\begin{aligned} d_{f'}(s,v) &= d_{f'}(s,w) + 1 \\ &\geq d_f(s,w) + 1 \quad \text{(pela proposição 2.8)} \\ &= d_f(s,v) + 2. \quad \text{(por (2.3))} \end{aligned}$$

Isto mostra que, quando (v,w) torna-se crítica novamente, a distância de s a v aumenta em pelo menos 2. Como a distância de s a v é pelo menos 0 inicialmente e não pode

passar de $|V| - 2$ (pois $v \neq t$), temos que a aresta (v, w) pode ser crítica no máximo $(|V| - 2)/2 + 1 = |V|/2$ vezes, como queríamos. □

Como cada caminho de aumento pode ser encontrado em tempo $O(|E|)$ com busca em largura, concluimos que a complexidade de tempo do algoritmo de Edmonds-Karp é $O(|V||E|^2)$.

2.4 Algoritmo de Dinic

A ideia chave do algoritmo de Edmonds-Karp é buscar somente por caminhos de aumento de comprimento mínimo. Focando somente em caminhos mínimos, o algoritmo consegue obter uma complexidade de tempo fortemente polinomial $O(|V||E|^2)$.

Note, no entanto, que o algoritmo realiza repetidamente buscas em largura na rede residual, partindo sempre da fonte e procurando um caminho até o sorvedouro. É razoável pensar que há um certo desperdício aqui. A rede residual não se altera muito de uma iteração para a seguinte, e isto significa que podemos estar recalculando a mesma coisa várias vezes quando reiniciamos a busca.

Nesta seção veremos como podemos reaproveitar informações de buscas anteriores ao procurar por caminhos de aumento na rede residual. O resultado desta melhoria é conhecido como **algoritmo de Dinic**[2].

Assim como no algoritmo de Edmonds-Karp, o algoritmo de Dinic parte de um fluxo nulo e o incrementa iteradamente. Estes incrementos são feitos em fases, caracterizadas pela distância, em número de arestas, da fonte ao sorvedouro.

No início de cada fase, o algoritmo atribui um rótulo $l(v)$ para cada vértice v da rede, que corresponde à distância da fonte s ao vértice v na rede residual atual. Este rótulo é chamado de **nível** do vértice, e será mantido fixo durante toda a fase. Não nos preocuparemos com rótulos de vértices não alcançáveis a partir de s.

Uma aresta (v, w) da rede residual é dita **boa** quando $l(w) = l(v) + 1$. Um **caminho bom** é um caminho que contém somente arestas boas.

A ideia do algoritmo de Dinic é buscar somente por caminhos bons de s a t em cada fase, até que eles se esgotem.

Proposição 2.9. *Suponha que $l(t) = d$. Então, se não existe caminho bom de s a t, então todo caminho de aumento de s a t tem comprimento maior que d.*

Esta proposição pode parecer óbvia, mas não é, uma vez que os rótulos correspondem a distâncias somente no início de cada fase do algoritmo.

Demonstração. Note que, no início da fase, para toda aresta (v, w) da rede residual, vale $l(w) \leq l(v) + 1$, pois neste caso os rótulos representam as distâncias a partir de s.

Considere um caminho bom P de s a t. O aumento do fluxo ao longo de P pode causar alterações na rede residual. Porém, como toda aresta de P é boa, as únicas alterações possíveis são remoção de uma aresta boa ou a adição de uma aresta (v, w) com $l(w) = l(v) - 1$.

Logo, para toda aresta (v, w) da nova rede residual, continua valendo que $l(w) \leq l(v)+1$, ou seja, isto é uma invariante que se mantém por toda a fase.

Agora, suponha que não exista caminhos bons de s a t, e suponha que exista um caminho de aumento P' de s a t. Pela invariante, P' deve ter comprimento pelo menos d, caso contrário possuiria uma aresta (v, w) com $l(w) > l(v) + 1$, violando a invariante.

[2]Também aparece na literatura como Dinits ou Dinitz.

Se P' tem comprimento d, então toda aresta de P' deve ser boa. Isto implica que P' é bom, contradição. Logo, P' deve ter comprimento maior que d. □

Para encontrar os caminhos bons de uma fase, o algoritmo de Dinic faz uma busca em profundidade partindo de s usando somente arestas boas. Se alcançar o vértice t, a busca retorna o caminho encontrado. Quando a busca num vértice v realiza uma chamada para um vizinho w e esta retorna sem encontrar caminho, marcamos a aresta (v, w) para que ela não seja mais considerada em buscas futuras da mesma fase.

Proposição 2.10. *Se $l(t) = d$ e a busca em profundidade partindo de s retorna sem encontrar um caminho bom, então não há caminhos bons na rede residual.*

Demonstração. Basta mostrarmos que, após a busca marcar uma aresta (v, w), então não pode existir um caminho bom de w a t até o fim da fase.

Como foi observado na demonstração da proposição 2.9, o aumento do fluxo ao longo de um caminho bom nunca adiciona novas arestas boas à rede residual. Assim, se em algum momento durante uma fase não há caminho bom de w a t, então t permanecerá inalcançável a partir de w por caminhos bons até o fim da fase. □

Proposição 2.11. *Todos os caminhos bons de uma fase podem ser encontrados em $O(|V||E|)$.*

Demonstração. Mostraremos que o número total de chamadas realizadas pelas buscas em profundidade é $O(|V||E|)$.

Para cada vértice v, cada chamada da busca em profundidade pode retornar um caminho encontrado ou retornar sem encontrar caminho. O último caso pode ocorrer ao total no máximo $|E|$ vezes, pois toda vez que a chamada retorna sem encontrar caminho uma aresta é marcada e não é mais utilizada até o fim da fase.

O primeiro caso ocorre em grupos de $d + 1$, em que d é o comprimento dos caminhos bons da fase atual. Como o aumento do fluxo ao longo de um caminho bom satura uma aresta boa da rede residual, cada fase usa no máximo $|E|$ caminhos bons. Portanto, o total de chamadas que retornam um caminho é no máximo $(d + 1)|E| \leq |V||E|$. □

Proposição 2.12. *O algoritmo de Dinic tem complexidade de tempo $O(|V|^2|E|)$.*

Demonstração. Pela proposição 2.8, a distância de s a t cresce em pelo menos uma unidade de uma fase para a seguinte. Logo, há no máximo $|V|$ fases. Pela proposição 2.11, cada fase leva tempo $O(|V||E|)$. Portanto, o algoritmo de Dinic tem complexidade de tempo $O(|V|^2|E|)$. □

2.5 Algoritmo Push-Relabel

Até o momento nossos algoritmos de fluxo máximo focavam-se na ideia de caminhos de aumento. No entanto, existem algumas técnicas poderosas para o cálculo do fluxo máximo que não se baseiam explicitamente em caminhos de aumento. Nesta seção estudaremos uma delas, o **algoritmo Push-Relabel**.

Os algoritmos vistos até agora mantêm um fluxo f e incrementam seu valor ao longo de caminhos de aumento. O algoritmo Push-Relabel, por outro lado, essencialmente aumentará o fluxo aresta por aresta. O aumento do fluxo em uma única aresta usualmente resulta na violação da condição de conservação. Por este motivo, o algoritmo não manterá um fluxo propriamente dito em cada iteração, e sim algo menos restrito.

Um *s-t* **pré-fluxo** (ou pré-fluxo de s a t) é uma função $f : E \to \mathbb{Q}^+$ que atribui para cada aresta um valor não-negativo. Um pré-fluxo deve satisfazer as condições de capacidade, mas no lugar das condições de conservação, exigimos somente desigualdades:

1. Para cada aresta $e \in E$, temos

$$0 \leq f(e) \leq u(e);$$

2. Para cada vértice v diferente de s, temos

$$e(v) := f^+(v) - f^-(v) \geq 0,$$

em que $e(v)$ é o **excesso** do vértice v.

Note que um pré-fluxo no qual o excesso em todo vértice diferente de s e t é zero é um fluxo de valor $e(t) = -e(s)$.

Assim como em fluxos, omitiremos as referências aos vértices s e t e usaremos somente o termo **pré-fluxo** quando o contexto não deixa ambiguidades.

Podemos definir o conceito de rede residual para pré-fluxos da mesma forma que o definimos para fluxos. O algoritmo, então, "empurrará" fluxo ao longo de arestas da rede residual.

A ideia do algoritmo Push-Relabel é manter um pré-fluxo a cada iteração, visando transformá-lo num fluxo. O algoritmo baseia-se na intuição física de que o fluxo naturalmente flui "para baixo". As "alturas" são rótulos $h(v)$ para cada vértice que o algoritmo mantém. Enviaremos fluxo de vértices de alturas maiores para vértices de alturas menores, seguindo a intuição física. Mais precisamente, os rótulos são definidos por uma **função-altura** $h : V \to \mathbb{Z}^+$ dos vértices para os inteiros não-negativos. O valor $h(v)$ é chamado de **altura** do vértice v.

Dizemos que uma função-altura h e um s-t pré-fluxo f são **compatíveis** quando

1. $h(s) = |V|$ e $h(t) = 0$;

2. (*Condições de inclinação*) Para toda aresta (v, w) da rede residual, temos $h(v) \leq h(w) + 1$.

A propriedade chave de um pré-fluxo e uma função-altura compatíveis é que não pode haver caminho de s a t na rede residual.

Proposição 2.13. *Se um s-t pré-fluxo f é compatível com uma função-altura h, então não existe caminho de s a t na rede residual G_f.*

Demonstração. Provaremos por contradição. Suponha que exista um caminho simples $P = (s = v_0, v_1, \ldots, v_k = t)$ de s a t em G_f. Como h é compatível com f, então $h(s) = |V|$. Como a aresta (v_0, v_1) está na rede residual, pelas condições de inclinação devemos ter $h(v_1) \geq h(v_0) - 1 = |V| - 1$. Por indução em i concluímos que $h(v_i) \geq |V| - i$, implicando $h(t) \geq |V| - k$. Mas P é simples, então $k < |V|$. Portanto, $h(t) > 0$, contradizendo a compatibilidade de f com h. □

Já vimos que, pela proposição 2.7, se não existe caminho de s a t na rede residual de um s-t fluxo f, então f tem valor máximo. Assim, temos o seguinte corolário:

Corolário 2.2. *Se um s-t fluxo f é compatível com uma função-altura h, então f é um fluxo de valor máximo.*

O algoritmo Push-Relabel, então, mantém um pré-fluxo f e uma função-altura compatível h e trabalha em modificar f e h de forma a transformar gradualmente f em um fluxo. Uma vez que f se torna um fluxo, podemos invocar o corolário 2.2 para concluir que f é máximo.

Para começar o algoritmo, temos que definir um pré-fluxo f e uma função-altura h iniciais que sejam compatíveis. Usaremos $h(v) = 0$ para todo vértice v diferente de s e $h(s) = |V|$ como função-altura inicial. Para que f seja compatível com h basta garantirmos que não haja arestas saindo de s na rede residual. Para isto, tomaremos $f(e) = u(e)$ para toda aresta $e = (s, v)$ saindo de s e $f(e) = 0$ para o resto das arestas.

Discutiremos agora os passos que o algoritmo segue para transformar f em um fluxo, mantendo sua compatibilidade com uma função-altura h. Considere um vértice v que tenha excesso positivo. Se existe uma aresta e em G_f que sai de v e chega em um vértice w de altura menor, então podemos modificar f enviando parte do excesso de v para w. Chamaremos esta operação de *push*.

```
1:  PUSH(f, h, v, w)                          ▷ e(v) > 0, h(w) < h(v) e (v, w) ∈ G_f
2:      se (v, w) é uma aresta direta de G_f então
3:          e ← (v, w)
4:          ε ← min(e(v), u(e) − f(e))
5:          f(e) ← f(e) + ε
6:      se (v, w) é uma aresta inversa de G_f então
7:          e ← (w, v)
8:          ε ← min(e(v), f(e))
9:          f(e) ← f(e) − ε
10:     devolva (f, h)
```

Caso não seja possível enviar o excesso de v ao longo de alguma aresta saindo de v, devemos aumentar a altura de v. Chamaremos esta operação de *relabel*.

```
RELABEL(f, h, v)          ▷ e(v) > 0 e h(w) ≥ h(v) para toda aresta (v, w) ∈ G_f
    h(v) ← h(v) + 1
    devolva (f, h)
```

O pseudocódigo a seguir resume o algoritmo Push-Relabel completo.

```
1:  PUSHRELABEL(G, s, t)
2:      para cada v em V faça h(v) ← 0
3:      h(s) ← |V|
4:      para cada e ← (v, w) em G_f faça
5:          se v = s então f(e) ← u(e)
6:          senão f(e) ← 0
7:      enquanto existe vértice v ≠ t com e(v) > 0 faça
8:          seja v um vértice com excesso positivo
9:          se existe aresta (v, w) com h(w) < h(v) então PUSH(f, h, v, w)
10:         senão RELABEL(f, h, v)
11:     devolva f
```

Para provar a corretude do algoritmo, iremos primeiro demonstrar a principal invariante mantida pelo algoritmo.

Proposição 2.14. *Em toda iteração do algoritmo Push-Relabel, f é um pré-fluxo e h é uma função-altura. Ademais, f é compatível com h.*

Demonstração. Vimos que a afirmação é verdadeira no início do algoritmo. Mostraremos que a invariante se mantém após cada operação *push* e *relabel*.

A operação *push* modifica a função f, mas o valor de ε garante que a função continue satisfazendo as condições de capacidade e que o excesso de todo vértice continue não-negativo.

A operação *relabel* somente aumenta as alturas dos vértices. Assim, se h é não-negativo no início de uma iteração, a função continuará não-negativa após uma operação *relabel*.

Para verificar que f e h se mantêm compatíveis, note que uma chamada PUSH(f, h, v, w) pode adicionar uma aresta na rede residual, a aresta inversa (w, v), mas esta aresta satisfaz as condições de inclinação. Uma chamada RELABEL(v) aumenta a altura de v, mas como ela é somente aplicada quando não existem arestas saindo de v e chegando em vértices de altura menor, esta operação não causa a violação das condições de inclinação.

Como a operação *relabel* nunca é chamada com os vértices s e t, concluímos que f e h continuam compatíveis após cada operação *push* e *relabel*. □

Com isso, sabemos que se o algoritmo Push-Relabel terminar, então a função f devolvida pelo algoritmo é um fluxo máximo pelo corolário 2.2. Para provar que o algoritmo, de fato, termina, iremos demonstrar que o número de operações *push* e *relabel* efetuadas são limitadas.

Começaremos provando um fato que será importante para a delimitação da quantidade de operações *relabel* realizadas.

Proposição 2.15. *Seja f um pré-fluxo. Se um vértice v tem excesso positivo, então existe um caminho de v a s na rede residual G_f.*

Demonstração. Seja A o conjunto de todos os vértices v tais que existe um caminho de v a s em G_f, e seja $B = V \setminus A$. Mostraremos que todos os vértices com excesso positivo estão em A.

Note que s está em A. Além disso, nenhuma aresta $e = (v, w)$ saindo de A pode ter fluxo positivo, pois isto implicaria na existência da aresta inversa (w, v) na rede residual e w estaria em A.

Considere agora a soma dos excessos no conjunto B. Como todo vértice em B tem excesso não-negativo, temos

$$\begin{aligned} 0 \leq \sum_{v \in B} e(v) &= \sum_{v \in B} (f^+(v) - f^-(v)) \\ &= f^+(B) - f^-(B) && \text{(proposição 2.1)} \\ &= 0 - f^-(B) \end{aligned}$$

Como f é uma função não-negativa, temos que a soma dos excessos dos vértices em B deve ser 0. Consequentemente, todos os vértices em B tem excesso 0. □

Proposição 2.16. *Durante todo o algoritmo, $h(v) \leq 2|V| - 1$ para todo vértice v.*

Demonstração. As alturas dos vértices s e t não se alteram durante a execução do algoritmo. Considere então um vértice v diferente de s e t. A altura de v só é alterada com a operação *relabel*, então sejam f e h o pré-fluxo e a função-altura devolvidas por uma chamada RELABEL(f, h, v). Pela proposição 2.15, existe um caminho P de v a s na rede residual. P tem no máximo $|V| - 1$ arestas e a altura dos vértices diminui no máximo de unidade em unidade ao longo de P. Portanto, $h(v) - h(s) \leq |V| - 1$, ou seja, $h(v) \leq 2|V| - 1$, como queríamos. □

Como as alturas só podem aumentar durante a execução do algoritmo, o seguinte corolário é imediato.

Corolário 2.3. *Durante toda a execução do algoritmo Push-Relabel a operação relabel é aplicada no máximo* $2|V| - 1$ *vezes em cada vértice, e ao todo o algoritmo realiza menos de* $2|V|^2$ *operações relabel.*

Delimitaremos agora a quantidade de operações *push*. Dividiremos as operações *push* em dois tipos. Uma operação PUSH(f, h, v, w) é dita **saturante** quando $e = (v, w)$ é uma aresta direta e $\varepsilon = u(e) - f(e)$ ou (v, w) é uma aresta inversa com $e = (w, v)$ e $\varepsilon = f(e)$. Em outras palavras, um *push* é saturante quando, após a operação, a aresta (v, w) é removida da rede residual. Todas as outras operações *push* são ditas **não saturantes**.

Proposição 2.17. *O algoritmo Push-Relabel realiza ao todo no máximo* $2|V||E|$ *operações push saturantes.*

Demonstração. Considere uma aresta (v, w) da rede residual G_f. Após uma operação PUSH(f, h, v, w) saturante, temos $h(v) = h(w)+1$ e a aresta (v, w) não se encontra mais em G_f. Antes que seja possível efetuar outra operação *push* por esta aresta, devemos aplicar a operação *push* na aresta (w, v) para que a aresta (v, w) volte a aparecer em G_f. Porém, só podemos aplicar a operação *push* em (w, v) se $h(w) = h(v) + 1$, ou seja, a altura de w deve aumentar em pelo menos 2 unidades.

Como a altura de w pode aumentar em 2 unidades no máximo $|V| - 1$ vezes (pelo corolário 2.3), temos que uma operação *push* saturante pode ser efetuada na aresta (v, w) no máximo $|V|$ vezes. Como cada aresta de E pode dar origem a duas arestas em G_f, temos que o número total de operações *push* saturantes durante todo o algoritmo é no máximo $2|V||E|$. □

Proposição 2.18. *O algoritmo Push-Relabel realiza ao todo no máximo* $2|V|^2(2|E| + 1)$ *operações push não saturantes.*

Demonstração. Para esta demonstração usaremos o método da função potencial. Dados um pré-fluxo f e uma função-altura h compatíveis, definimos a função potencial $\Phi(f, h)$ como a soma das alturas de todos os vértices com excesso positivo:

$$\Phi(f, h) = \sum_{v:e(v)>0} h(v).$$

No início do algoritmo todos os vértices com excesso positivo estão com altura 0, portanto $\Phi(f, h) = 0$. $\Phi(f, h)$ claramente se mantém não-negativo durante todo o algoritmo.

Uma operação PUSH(f, h, v, w) não saturante diminui o valor de $\Phi(f, h)$ em ao menos uma unidade, pois v passa a não ter excesso e w, o único vértice cujo excesso aumenta após a operação, está a uma altura uma unidade abaixo de v. Porém, cada operação *push* saturante e *relabel* pode aumentar o valor de $\Phi(f, h)$.

A operação *relabel* aumenta o valor de $\Phi(f, h)$ em exatamente uma unidade. Como há ao todo no máximo $2|V|^2$ operações *relabel* pelo corolário 2.3, o aumento total de $\Phi(f, h)$ devido a operações *relabel* é de no máximo $2|V|^2$.

Uma operação PUSH(f, h, v, w) saturante não altera alturas, mas pode aumentar o valor de $\Phi(v, w)$, pois o vértice w pode adquirir excesso positivo após a operação. Isto aumentaria o valor de $\Phi(v, w)$ pela altura de w, que é no máximo $2|V| - 1$ (corolário 2.3). Como há ao todo no máximo $2|V||E|$ operações *push* saturantes, o aumento total de $\Phi(f, h)$ devido a operações *push* saturantes é de no máximo $2|E||V|(2|V| - 1)$. Portanto, o valor de $\Phi(f, h)$ pode aumentar ao todo em no máximo $2|V|^2 + 2|E||V|(2|V| - 1) < 2|V|^2 + 4|V|^2|E| = 2|V|^2(2|E| + 1)$ unidades durante todo o algoritmo.

Como $\Phi(f, h)$ se mantém não-negativa durante o algoritmo e diminui em ao menos uma unidade após cada operação *push* não saturante, concluímos que podem haver ao todo no máximo $2|V|^2(2|E| + 1)$ operações *push* não saturantes. □

Mostraremos no capítulo seguinte que o algoritmo Push-Relabel tem consumo de tempo $O(|E||V| + k)$, em que k é o número de operações *push* não saturantes. Como o número de operações *push* não saturantes é $O(|V|^2|E|)$, concluímos que o algoritmo Push-Relabel tem complexidade de tempo $O(|V|^2|E|)$.

Critérios de seleção de vértices ativos

Assim como o algoritmo de Ford-Fulkerson não estabelece critérios para a escolha de caminhos de aumento, o algoritmo Push-Relabel também nos dá total liberdade na escolha do vértice com excesso positivo a ser processado em cada iteração. Assim, é natural pensar que uma escolha sistemática dos vértices poderia levar a uma melhoria na eficiência do algoritmo.

Diremos que um vértice diferente de t é **ativo** quando possui excesso positivo. Estudaremos agora dois critérios de seleção de vértices ativos que resultam numa delimitação menor do número de operações *push* não saturantes realizadas pelo algoritmo.

Critério *first-in first-out*

No critério *first-in first-out* (FIFO), os vértices ativos são mantidos em uma fila. Inicialmente, os vizinhos de s são inseridos na fila em qualquer ordem. Na hora de escolher um vértice ativo, o algoritmo escolhe o vértice que está na frente da fila. Se no fim da iteração o vértice escolhido continuar com excesso positivo, ele permanece na frente da fila. Caso contrário, é removido da fila. Se um outro vértice passa a ter excesso positivo devido a uma operação *push* realizada no vértice escolhido, esse vértice entra no final da fila.

Dessa forma, quando um vértice v é escolhido, o algoritmo repetidamente realizará operações *push* e *relabel* em v até que v fique com excesso nulo. Chamaremos este conjunto de operações de **descarga** do vértice v. Note que, enquanto a última operação *push* de uma descarga pode ter sido não saturante, todas as outras operações *push* anteriores devem ter sido saturantes. Assim, uma delimitação do número de descargas também delimita a quantidade de operações *push* não saturantes feitas pelo algoritmo.

Proposição 2.19. *O número total de descargas feitas pelo algoritmo Push-Relabel com critério de seleção FIFO é* $4|V|^3$.

Demonstração. Note que os vértices são processados em níveis. No primeiro nível estão os vizinhos de s, os vértices inseridos na fila no início do algoritmo. No segundo nível estão os vértices inseridos na fila devido à descarga de algum vértice do primeiro nível. No terceiro nível estão os vértices inseridos na fila devido à descarga de algum vértice do segundo nível, e assim por diante.

Mostraremos que o número de níveis é no máximo $4|V|^2$. Assim, como cada vértice pode sofrer uma descarga no máximo uma vez por nível, concluímos que o número total de descargas feitas pelo algoritmo é limitado por $4|V|^3$.

Para esta demonstração usaremos novamente o método da função potencial. Neste caso, a função potencial é

$$\Phi(f, h) = \max_{v \text{ ativo}} h(v),$$

sendo $\Phi(f, h) = 0$ quando não há vertices ativos.

Vamos analisar o que acontece com o valor da função potencial ao término de cada nível. Chamaremos um nível de *bom* se nenhum vértice deste nível recebeu uma operação de *relabel*. Como as operações *push* só são realizadas de um vértice para outro de altura menor, ao fim de um nível bom o valor da função potencial diminui em ao menos uma unidade.

Como cada operação *relabel* aumenta o valor de $\Phi(f, h)$ em no máximo uma unidade e o total de operações *relabel* é limitado por $2|V|^2$, concluímos que o número de níveis bons não pode ser maior que $2|V|^2$. Por outro lado, a quantidade de níveis não bons também é limitado por $2|V|^2$, pois em cada um destes níveis houve ao menos uma operação *relabel*. Portanto, o número total de níveis é $4|V|^2$, como queríamos. □

Corolário 2.4. *O algoritmo Push-Relabel com critério de seleção FIFO tem consumo de tempo* $O(|V|^3)$.

Critério da maior altura

Assim como no critério FIFO, no critério da maior altura o algoritmo seleciona um vértice ativo e aplica repetidamente operações *push* e *relabel* até que este vértice deixe de ter excesso positivo. Neste caso, o vértice escolhido é aquele que possui maior altura.

Para demonstrar o limite do número de operações *push* não saturantes feitas pelo algoritmo Push-Relabel com critério da maior altura, iremos usar o conceito de fases. Uma fase do algoritmo é uma sequência de operações realizadas em vértices ativos de mesma altura.

Proposição 2.20. *O número total de fases no algoritmo Push-Relabel com critério da maior altura é no máximo* $4|V|^2$.

Demonstração. Seja H o máximo das alturas dentre os vértices ativos. Uma fase do algoritmo pode terminar por dois motivos: ou um vértice ativo de altura máxima sofre um *relabel*, ou o último vértice ativo de altura máxima deixa de ter excesso positivo. No primeiro caso, H aumenta em uma unidade. No segundo, H diminui em ao menos uma unidade.

O valor inicial de H é 0 e pode aumentar no máximo $2|V|^2$ vezes pela proposição 2.3. Como H é não-negativo, seu valor pode diminuir no máximo $2|V|^2$ vezes. Portanto, o número total de fases não pode exceder $4|V|^2$. □

Proposição 2.21. *O número total de operações push não saturantes feitas pelo algoritmo Push-Relabel com critério da maior altura é* $O(|V|^2\sqrt{|E|})$.

Demonstração. Diremos que uma fase é *barata* quando a quantidade de operações *push* não saturantes realizadas nesta fase é menor que K, sendo K uma constante que definiremos posteriormente. Uma fase é dita *cara* quando a quantidade de operações *push* não saturantes é maior ou igual a K.

Pela proposição 2.20, o número fases é no máximo $4|V|^2$. Logo, a quantidade de operações *push* não saturantes feitas em fases baratas é limitada por $4K|V|^2$.

Para delimitarmos a quantidade de operações *push* não saturantes feitas em fases caras, usaremos novamente a ideia de função potencial. Para todo vértice v defina como $\phi(f, h, v)$ como a quantidade de vértices que possuem altura menor ou igual a de v:

$$\phi(f, h, v) = |\{w : h(w) \leq h(v)\}|.$$

Definimos a seguinte função potencial:

$$\Phi(f, h) = \sum_{v \text{ ativo}} \phi(f, h, v).$$

Após uma operação *relabel*, o valor da função potencial aumenta em no máximo $|V|$, pois $\phi(f,h,v) \leq |V|$ para todo vértice v.

Após uma operação *push* saturante numa aresta (v,w), o valor da função potencial pode aumentar em no máximo $|V|$, caso qual o vértice w passa a ser ativo, contribuindo em $\phi(f,h,w) < |V|$ para o valor de $\Phi(f,h)$.

Após uma operação *push* não saturante numa aresta (v,w), o vértice v deixa de ser ativo, causando numa diminuição de $\phi(f,h,v)$ no valor da função potencial. Por outro lado, o vértice w pode passar a ser ativo, contribuindo num acréscimo de $\phi(f,h,w) < \phi(f,h,v)$ em $\Phi(f,h)$. Porém, em fases caras, sabemos que ocorrem ao menos K operações *push* não saturantes. Como cada uma dessas operações é feita em vértices diferentes, sabemos então que há pelo menos K vértices com a mesma altura de v. Portanto, $\phi(f,h,v) - \phi(f,h,w) \geq K$. Assim, um *push* não saturante de uma fase cara reduz em ao menos K o valor da função potencial.

O valor inicial de $\Phi(h,v)$ é menor que $|V|^2$. Assim, $\Phi(h,v)$ pode aumentar ao todo em no máximo $2|V|^3 + 2|V|^2|E| + |V|^2 = O(|V|^2|E|)$ vezes. Como cada *push* não saturante de uma fase cara reduz em ao menos K o valor de $\Phi(f,h)$, concluímos que a quantidade de *push* não saturantes em fases caras é $O(|V|^2|E|/K)$.

Portanto, o número total de *push* não saturantes é $O(K|V|^2 + |V|^2|E|/K)$. Tomando $K = \sqrt{|E|}$ obtemos a delimitação desejada. □

Corolário 2.5. *O algoritmo Push-Relabel com critério de seleção da maior altura tem consumo de tempo* $O(|V|^2\sqrt{|E|})$.

2.6 Redes com capacidades reais

No início deste capítulo mencionamos a possibilidade de definir fluxo como uma função com contra-domínio nos reais não-negativos. De fato, poderíamos refazer toda a teoria com esta hipótese. No entanto, algumas demonstrações precisariam de ajustes e, ainda pior, alguns resultados passariam a ser falsos!

Um exemplo do efeito patológico causado pelos números reais ocorre no algoritmo de Ford-Fulkerson. Surpreendentemente, se permitirmos capacidades de valores reais, o algoritmo de Ford-Fulkerson pode nunca terminar. Note que nossa demonstração de que o algoritmo termina depende fortemente do fato das capacidades serem racionais, pois neste caso é sempre possível garantir que o fluxo aumenta pelo menos de um valor constante ε positivo a cada iteração. No caso de capacidades inteiras, o valor de ε era 1. No entanto, quando as capacidades são reais este valor ε pode não existir.

Considere, por exemplo, a rede da figura 2.3. Nesta rede, M é uma constante maior ou igual a 2, e $r = (\sqrt{5}-1)/2$. O valor de r foi escolhido de modo que $r^2 = 1-r$. Considere uma execução do algoritmo de Ford-Fulkerson que escolhe os caminhos de aumento de acordo com a tabela 2.1.

Note que, se em uma iteração as capacidades residuais de (b,a), (b,c) e (d,c) são, respectivamente, r^k, 0 e r^{k+1} para algum inteiro k, então após a sequência de caminhos de aumento (p_1,p_2,p_1,p_3), as capacidades passam a ser r^{k+2}, 0 e r^{k+3}, sendo o fluxo enviado ao longo de cada caminho r^{k+1}, r^{k+1}, r^{k+2} e r^{k+2}. Assim, podemos repetir esta sequência de caminhos indefinidamente. Pior ainda, o fluxo converge para

$$1 + 2\sum_{k=1}^{\infty} r^k = 3 + 2r,$$

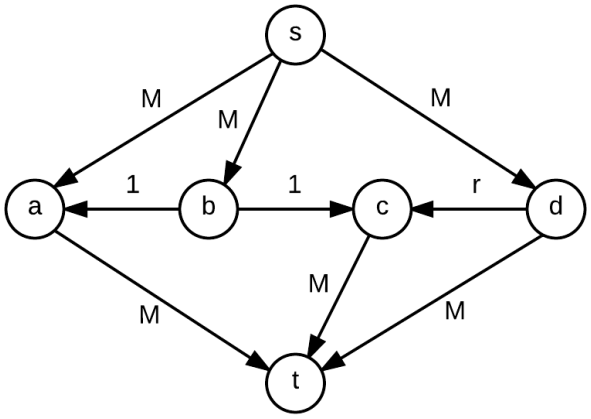

Figura 2.3: Menor exemplo de não terminação do algoritmo de Ford-Fulkerson (Zwick, 1993). Na figura, $M \geq 2$ é uma constante e $r = (\sqrt{5} - 1)/2$.

Iter.	Caminho de aumento	Capacidade do caminho	Capacidade residual		
			(b, a)	(b, c)	(d, c)
0			$r^0 = 1$	1	r^1
1	(s, b, c, t)	1	r^0	0	r^1
2	$p_1 = (s, d, c, b, a, t)$	r^1	r^2	r^1	0
3	$p_2 = (s, b, c, d, t)$	r^1	r^2	0	r^1
4	p_1	r^2	0	r^2	r^3
5	$p_3 = (s, a, b, c, t)$	r^2	r^2	0	r^3

Tabela 2.1: Sequência de caminhos patológica para o algoritmo de Ford-Fulkerson aplicado à rede da figura 2.3.

mas o fluxo máximo é claramente $2M + 1$. Portanto, o algoritmo de Ford-Fulkerson nesta rede com esta sequência patológica de caminhos de aumento não termina e nem converge para o fluxo máximo.

É importante ressaltar, porém, que apesar de nossa demonstração do teorema do fluxo máximo e corte mínimo (teorema 2.1) depender do algoritmo de Ford-Fulkerson terminar, o teorema continua válido mesmo quando as capacidades da rede são reais. De fato, se um fluxo f é máximo, a proposição 2.7 garante a existência de um corte cuja capacidade é igual ao valor de f. Basta, portanto, demonstrar que tal fluxo máximo existe mesmo em redes com capacidades reais. No entanto, a demonstração deste fato foge ao escopo deste trabalho e será omitida.

Capítulo 3

Implementações dos algoritmos de fluxo máximo

Neste capítulo apresentaremos implementações dos algoritmos de fluxo máximo vistos no capítulo anterior. As implementações estão na linguagem C++ e os códigos podem ser obtidos em https://linux.ime.usp.br/~marcosk/mac0499/codigos.html

Embora o foco das implementações seja a resolução de problemas no estilo da Maratona de Programação, foi preferido dar mais atenção à facilidade de entendimento em detrimento à eficiência e redução de linhas de código. Por conta disso, aconselhamos o leitor visando montar sua biblioteca pessoal de algoritmos que reescreva as implementações de acordo com suas necessidades e seus estilos.

3.1 Redes no computador

Em todas as implementações deste texto usaremos um *struct* `Graph` para representar uma rede. Optamos por usar um *struct* ao invés de deixar os atributos da rede como variáveis globais (como seria um código no estilo "maratona") pois esta representação facilita o manuseio de múltiplas redes no mesmo código.

```
1   struct Graph {
```

Queremos representar uma rede capacitada $G = (V, E, u)$ e um fluxo f no computador. Há mais de uma forma de fazer isto. Uma forma simples seria representar o grafo (V, E) por uma lista de adjacências, e armazenar os valores de u e f em uma matriz $|V| \times |V|$. Apesar desta representação funcionar bem na maioria dos casos, ela trás algumas desvantagens. Por exemplo, é necessário tratar o caso em que a rede possui arestas paralelas. Além disso, esta representação ocupa espaço de memória $O(|E| + |V|^2)$. Ainda pior, ela não é capaz de representar redes que possuem mais de um valor associado a cada aresta, por exemplo, um custo c.

Optaremos por uma representação similar que não apresenta estes problemas. O grafo continuará sendo representado por uma lista de adjacências, mas cada entrada da lista conterá, além do destino da aresta correspondente, qualquer outra informação pertinente sobre esta aresta, como sua capacidade e o fluxo que passa por ela.

Em outras palavras, cada aresta será representada por um *struct* contendo informações relevantes sobre ela.

```
2       struct Edge {
3           int v, u;
4           Edge(int _v, int _u) : v(_v), u(_u) {}
```

```
5       };
```

Note que, caso o valor de f em cada aresta não seja relevante para o problema sendo resolvido, podemos, ao invés de armazenar a rede G e o fluxo f, manter somente a rede residual G_f. Isto é prático, pois para aumentar em ε o fluxo por uma aresta de G_f, basta reduzirmos a capacidade residual desta aresta em ε e aumentarmos a capacidade residual da aresta reversa correspondente pelo mesmo valor. Se mantivéssemos G e f, teríamos que separar nos casos em que a aresta é direta ou inversa no caminho de aumento.

Para representar a rede, manteremos uma lista `edges` de todas as arestas da rede e, para cada vértice v, guardaremos uma lista `adj[v]` dos índices das arestas que saem de v. O número de vértices da rede será guardada na variável `n`.

```
6       int n;
7       vector<Edge> edges;
8       vector<vector<int>> adj;
```

O motivo de não usarmos simplesmente um vetor de `vector<Edge>` para representar a rede será explicado em breve.

O construtor do *struct* `Graph` recebe o número de vértices da rede, inicializa a variável `n` e redimensiona `adj` de acordo.

```
9       Graph(int _n) : n(_n) { adj.resize(n); }
```

Por fim, o método `add_edge()` insere uma aresta (v, w) de capacidade u na rede. Adotaremos a convenção de que arestas de capacidade residual 0 serão mantidas na rede. Assim, a adição da aresta (v, w) implica na inserção da aresta (v, w), de capacidade residual u, e da aresta reversa (w, v), de capacidade residual 0, na rede residual.

```
10      void add_edge(int v, int w, int u) {
11          adj[v].push_back(edges.size());
12          edges.push_back(Edge(w, u));
13          adj[w].push_back(edges.size());
14          edges.push_back(Edge(v, 0));
15      }
16  };
```

Caso a rede seja não-dirigida, a capacidade residual da aresta adicionada na linha 14 deve ser substituida por u. Alternativamente, pode-se chamar o método `add_edge()` duas vezes para cada aresta da rede não-dirigida, uma para cada direção.

```
14      edges.push_back(Edge(v, u));
```

Como uma aresta é sempre inserida juntamente com seu par reverso na lista `edges`, o índice de uma aresta e sua reversa diferem somente no seu bit menos significativo na representação binária. Isto é uma vantagem, pois dado o índice de qualquer aresta, digamos `i`, o índice da aresta reversa pode ser facilmente encontrado calculando `i^1`, em que `^` representa a operação xor binário. Não teríamos a mesma facilidade se tivéssemos escolhido representar a rede como um vetor de `vector<Edge>`.

3.2 Implementação de Edmonds-Karp

Conforme visto na seção 2.3, o algoritmo de Edmonds-Karp aplicado a uma rede capacitada G com fonte s e sorvedouro t pode ser descrito pelos seguintes passos:

1. Inicialmente, f é uma função que atribui valor 0 para todas as arestas da rede.

2. Enquanto houver caminho de aumento de s a t, repita o passo 3.

3. Seja P um caminho de aumento de s a t de comprimento mínimo. Incremente f ao longo de P.

4. Devolva f.

Antes de iniciar a implementação do algoritmo, iremos definir uma constante para representar o valor infinito. Um valor popularmente utilizado é $3F3F3F3F_{16} = 1061109567_{10}$. Este valor apresenta algumas vantagens:

- Evita erros de digitação. É muito mais fácil verificar se digitamos "0x3f3f3f3f" corretamente do que contar o número de zeros em "1000000000", por exemplo.
- É um número grande (num contexto envolvendo somente inteiros de 32 bits), mas não tão grande a ponto de precisar de cuidado excessivo com *overflow*. Por exemplo, o dobro deste valor ainda é representável como um inteiro de 32 bits com sinal.
- Como se trata da repetição do byte "3f", ele pode ser usado dentro de um `memset()` para preencher um vetor com o valor infinito.

```
1   const int INF = 0x3f3f3f3f;
```

Descrevemos a seguir uma função que realizará a verificação do passo 2 e o incremento do passo 3. Esta função recebe como argumentos uma rede, o índice da fonte e do sorvedouro, e retorna ε, a capacidade de um caminho de aumento de s a t. Caso não haja caminho de aumento de s a t, a função retorna 0.

```
2   int augment(Graph &g, int s, int t) {
3       vector<int> prv(g.n, -1), flow(g.n);
4       prv[s] = s;
5       flow[s] = INF;
```

Mantemos dois vetores para a implementação desta função. O vetor `prv` guarda, para cada vértice v, o índice da última aresta de um caminho de aumento de s a v. Convencionamos `prv[s] = s` e `prv[w] = -1` caso não haja caminho de aumento de s a w. O vetor `flow` armazena a capacidade do caminho de aumento indicado por `prv`. Convencionamos `flow[s] = INF`. O valor de `flow[w]` quando não existe caminho de aumento de s a w é irrelevante.

Para encontrarmos um caminho de aumento de s a t de comprimento mínimo, realizaremos uma busca em largura na rede residual partindo do vértice s. Para isto, manteremos uma fila `queue` dos vértices a serem processados pela busca, inicialmente contendo o vértice s.

```
6       queue<int> queue({s});
```

As linhas 7-18 contêm o laço principal da busca em largura. Encerramos a busca quando não houver mais vértices a serem processados, ou assim que o sorvedouro for atingido.

```
7       while (!queue.empty() && prv[t] != -1) {
8           int v = queue.front();
9           queue.pop();
10          for (int i: g.adj[v]) {
11              int w = g.edges[i].v, u = g.edges[i].u;
```

Aqui, `i` é o índice de uma aresta que sai de v, chega em w e tem capacidade residual u. É importante verificar se esta aresta tem capacidade residual positiva, uma vez que convencionamos manter arestas de capacidade residual 0 para evitar a necessidade de modificar a estrutura da rede residual.

```
12                  if (prv[w] == -1 && u > 0) {
```

Se esta condição for satisfeita, então encontramos um vértice novo. Neste caso, atualizamos as posições correspondentes nos vetores `prv` e `flow`, e o inserimos na fila.

```
13                      prv[w] = i;
14                      flow[w] = min(flow[v], u);
15                      queue.push(w);
16                  }
17              }
18          }
```

Ao término da busca, se `prv[t]` for -1, então não há caminho de aumento de s a t. Neste caso, a função retorna 0.

```
19          if (prv[t] == -1) return 0;
```

Caso contrário, um caminho de aumento P de s a t de capacidade `flow[t]` foi encontrado, e a função deve atualizar as capacidades residuais das arestas deste caminho, a fim de aumentar o valor do fluxo de s a t. Como estamos enviando `flow[t]` unidades de fluxo ao longo de P, as capacidade residuais das arestas de P se reduzem em `flow[t]`, enquanto as capacidades residuais das arestas reversas correspondentes aumentam pelo mesmo valor. Para fazermos esta atualização, iteramos pelos vértices de P de trás para frente, com o auxílio do vetor `prv`.

```
20          for (int v = t; v != s; v = g.edges[prv[v]^1].v) {
21              g.edges[prv[v]].u -= flow[t];
22              g.edges[prv[v]^1].u += flow[t];
23          }
24          return flow[t];
25      }
```

Com a função principal do algoritmo implementada, a função que determina o valor do fluxo máximo entre s e t torna-se bastante simples. A única coisa que precisamos fazer é chamar repetidamente a função `augment()` enquanto seu valor de retorno for maior que zero. O valor do fluxo máximo será dado pela soma de todos os valores retornados pela função.

```
26      int maxflow(Graph &g, int s, int t) {
27          int total_flow = 0, cur_flow;
28          do {
29              cur_flow = augment(g, s, t);
30              total_flow += cur_flow;
31          } while (cur_flow > 0);
32          return total_flow;
33      }
```

Note que esta implementação determina somente o valor do fluxo máximo. Caso seja necessário determinar o fluxo em cada aresta da rede, basta calcular a diferença entre a capacidade desta aresta e sua capacidade residual.

Como a busca em largura tem complexidade de tempo $O(|E|)$ e, pelo teorema 2.2, o algoritmo busca por no máximo $|V||E|$ caminhos de aumento, esta implementação do algoritmo de Edmonds-Karp tem complexidade de tempo $O(|V||E|^2)$.

3.3 Implementação de Dinic

O algoritmo de Dinic pode ser resumido nos seguintes passos:

1. Inicialmente, f é um fluxo nulo.
2. Determine os níveis dos vértices alcançáveis a partir de s na rede residual.
3. Se t não é alcançável a partir de s, retorne f.
4. Caso contrário, incremente f repetidamente ao longo de caminhos bons até que tais caminhos se esgotem.
5. Volte ao passo 2.

Implementaremos o passo 2 com uma busca em largura a partir de s, de forma bastante similar à função `augment()` do algoritmo de Edmonds-Karp. Manteremos o nível de cada vértice em um vetor `level`. Se um vértice for inalcançável a partir de s, seu nível será -1. A função retornará um valor booleano indicando se t foi alcançado durante a busca ou não.

```
1   vector<int> level;
2   bool bfs(Graph &g, int s, int t) {
3       fill(level.begin(), level.end(), -1);
4       level[s] = 0;
5       queue<int> queue({s});
6       while (!queue.empty()) {
7           int v = queue.front();
8           if (level[v] == level[t]) break;
9           queue.pop();
10          for (int i: g.adj[v]) {
11              int w = g.edges[i].v, u = g.edges[i].u;
12              if (level[w] == -1 && u > 0) {
13                  level[w] = level[v] + 1;
14                  queue.push(w);
15              }
16          }
17      }
18      return level[t] != -1;
19  }
```

Ao invés de construir uma rede contendo só arestas boas para a realização do passo 4, faremos a busca em profundidade na própria rede residual, e marcaremos as arestas não boas assim que as encontrarmos. Note que isto não trás prejuízo à complexidade do algoritmo.

Na realidade, não iremos marcar explicitamente as arestas da rede. Iremos somente garantir de que o algoritmo nunca processará uma aresta marcada. Para isto, manteremos para cada vértice um inteiro `ptr` que indicará qual a próxima aresta a ser processada pela busca. Quando decidirmos "marcar" a aresta que a busca acabou de processar, basta incrementarmos o valor de `ptr`.

Assim como no algoritmo de Edmonds-Karp, chamaremos de `augment()` a função que encontra um caminho de aumento e retorna sua capacidade. Esta função terá quatro argumentos: `g` é uma referência para a rede em questão, `v` é o vértice atual da busca em profundidade, `t` é o sorvedouro e `cap` é a capacidade do caminho de s a v encontrado pela busca.

```
20  vector<int> ptr;
21  int augment(Graph &g, int v, int t, int cap) {
```

Se a busca chegar no vértice t, então um caminho bom foi encontrado. Neste caso, simplesmente retornamos a capacidade do caminho.

```
22        if (v == t) return cap;
```

Caso contrário, processaremos as arestas saindo de v visando aumentar o caminho atual. Ao invés de começarmos o processamento pelo início da lista de adjacências de v, começaremos pela posição indicada por `ptr[v]`, uma vez que arestas em posições menores foram "marcadas", conforme explicado anteriormente.

```
23        for (int &p = ptr[v]; p < (int)g.adj[v].size(); p++) {
24            int i = g.adj[v][p];
25            int w = g.edges[i].v, u = g.edges[i].u;
```

Se a aresta de índice `i` for boa, então chamaremos a função `augment()` recursivamente para o destino desta aresta. Note que a capacidade do novo caminho é o mínimo entre a capacidade do caminho atual e a capacidade residual da aresta.

```
26            if (level[w] == level[v] + 1 && u > 0) {
27                int eps = augment(g, w, t, min(cap, u));
```

Se `eps` for positivo, então encontramos um caminho bom. Neste caso, atualizamos as capacidades residuais da aresta de índice `i` e de sua reversa e retornamos `eps`, a capacidade do caminho encontrado.

```
28                if (eps > 0) {
29                    g.edges[i].u -= eps;
30                    g.edges[i^1].u += eps;
31                    return eps;
32                }
33            }
34        }
```

Caso contrário, temos que "marcar" a aresta de íncide `i`, pois esta aresta não resultou num caminho bom. O laço `for` cuida disto incrementando o valor de `p`. Note que, como `p` é referência para `ptr[v]`, o valor de `ptr[v]` também é alterado.

Se a função sair do laço `for` sem retornar na linha 29, então não há mais caminhos bons de v a t. Neste caso, retornamos 0, e isto conclui o corpo da função.

```
35        return 0;
36    }
```

Por último, implementamos a função `maxflow()`, que retorna o valor do fluxo máximo de s a t. Assim como no algoritmo de Edmonds-Karp, esta função é bastante simples. Repare que esta função basicamente segue os passos 1 a 5 mostrados no início desta seção.

```
37    int maxflow(Graph &g, int s, int t) {
38        level.resize(g.n), ptr.resize(g.n);
39        int total_flow = 0, cur_flow;
40        while (bfs(g, s, t)) {
41            fill(ptr.begin(), ptr.end(), 0);
42            do {
43                cur_flow = augment(g, s, t, INF);
44                total_flow += cur_flow;
45            } while (cur_flow > 0);
46        }
47        return total_flow;
48    }
```

3.4 Implementações de Push-Relabel

Iniciaremos por mostrar como implementar o algoritmo Push-Relabel de forma que seu consumo de tempo seja $O(|V||E|+k)$, em que k é o número de operações *push* não saturantes.

Para cada vértice, manteremos um ponteiro `ptr` que indicará qual a próxima aresta a ser processada pelo algoritmo, similar à implementação do algoritmo de Dinic. Ao efetuarmos uma operação *push* saturante por uma aresta (v, w) apontada por `ptr[v]`, nós avançamos o ponteiro para a próxima aresta da lista de adjacências de v. A ideia é que a operação *push* não poderá ser realizada na aresta (v, w) novamente até que o vértice v sofra um *relabel*. Como consequência, se o ponteiro `ptr[v]` atingir o fim da lista de adjacências de v, então não existirão arestas saindo de v que poderão sofrer um *push*, portanto a operação *relabel* poderá ser aplicada em v. Após realizar uma operação *relabel* num vértice v, movemos o ponteiro `ptr[v]` de volta para o início da lista de adjacências de v.

Cada ponteiro `ptr[v]` pode ser avançado no máximo $O(|E^-(v)|)$ vezes entre duas operações *relabel* no vértice v, e cada vértice v pode sofrer um *relabel* até $2|V|$ vezes. Logo, o tempo gasto avançando os ponteiros durante o algoritmo é $O(|E||V|)$. Como as operações *push* e *relabel* levam tempo constante e a inicialização do algoritmo leva tempo $O(|E|)$, temos que o algoritmo Push-Relabel implementado com as estruturas acima tem tempo de execução $O(|V||E| + k)$, sendo k o número de operações *push* não saturantes.

Heurística *gap relabelling*

Antes de detalhar as implementações dos algoritmos Push-Relabel, iremos explicar uma heurística que será usada nas implementações e que é essencial para que os algoritmos tenham boa performance na prática.

Considere um instante do algoritmo em que existe uma altura $0 < h' < |V|$ tal que não existe vértice v de altura h', e seja w um vértice de altura maior que h'. Como o algoritmo mantém a função-altura sempre compatível com o pré-fluxo, temos que não existem arestas que saem de w e chegam em um vértice de altura menor que h' na rede residual. Por este motivo o excesso que está em w nunca chegará a t, portanto podemos aumentar a altura de w para $|V| + 1$, pois o excesso em w deverá voltar para s.

Mais formalmente, podemos alterar a altura de todo vértice w diferente de s com $h(w) > h'$ para $\max(h(w), |V| + 1)$. Chamaremos esta operação de ***gap relabelling***.

Proposição 3.1. *Uma operação de gap relabelling mantém a função-altura compatível com o pré-fluxo.*

Demonstração. A operação de *gap relabelling* não altera as alturas dos vértices s e t. Temos, então, que verificar que as condições de inclinação continuam satisfeitas.

Sejam h_1 e h_2 as funções-altura antes e depois da operação de *gap relabelling*, e seja (v, w) uma aresta da rede residual. Se $h_1(v) < h'$, então

$$h_2(v) = h_1(v) \leq h_1(w) + 1 \leq h_2(w) + 1.$$

Se $h_1(v) > h'$, então $h_1(w) > h'$, pois $h_1(w) \geq h_1(v) - 1$ e não existem vértices com altura h'. Logo,

$$h_2(v) = \max(h_1(v), |V|+1) \leq \max(h_1(w)+1, |V|+1) \leq \max(h_1(w), |V|+1)+1 = h_2(w)+1.$$

Portanto, a operação de *gap relabelling* não viola as condições de inclinação, logo a função-altura e o pré-fluxo permanecem compatíveis. □

Só precisamos verificar se um *gap relabelling* pode ser feito após realizar um *relabel* em algum vértice. Se neste instante um *gap relabelling* ocorrer, a altura deste vértice passa a ser maior que $|V|$. Com isso, podem ocorrer ao todo no máximo $|V|$ operações de *gap relabelling*. Como cada uma dessas operações tem complexidade de tempo $O(|V|)$, o tempo gasto nestas operações durante a execução do algoritmo é $O(|V|^2)$. Portanto, a implementação desta heurística não trás prejuízos na complexidade assintótica do algoritmo Push-Relabel.

Push-Relabel com critério FIFO

Para a implementação do algoritmo Push-Relabel com o critério de seleção FIFO, manteremos três vetores inteiros: `ptr`, `h` e `e`, que guardam, respectivamente, o ponteiro da aresta atual, a altura e o excesso de cada vértice. Além disso, manteremos um vetor `h_count`, que guardará para cada altura a quantidade de vértices com esta altura. Este vetor será usado na implementação do *gap relabelling*.

```
1   int maxflow(Graph &g, int s, int t) {
2       vector<int> ptr(g.n, 0), h(g.n, 0), e(g.n, 0), h_count(2 * g.n, 0);
```

Começamos a implementação da função inicializando os vetores mencionados.

```
3       h[s] = g.n;
4       h_count[g.n] = 1, h_count[0] = g.n - 1;
```

Usaremos a classe `queue` da *Standard Template Library* (STL) para implementar a fila dos vértices ativos.

```
5       queue<int> queue;
```

Para finalizar a inicialização do algoritmo, iremos saturar as arestas que saem de s e adicionar na fila os vértices com excesso positivo, tomando cuidado para não inserir um mesmo vértice mais de uma vez.

```
6       for (int i: g.adj[s]) {
7           int v = g.edges[i].v, u = g.edges[i].u;
8           if (u == 0) continue;
9           if (e[v] == 0 && v != t) queue.push(v);
10          e[v] += u;
11          e[s] -= u;
12          g.edges[i].u -= u;
13          g.edges[i^1].u += u;
14      }
```

Agora começa o laço principal do algoritmo. Enquanto a fila não estiver vazia, escolhemos o vértice na frente da fila e procuramos uma aresta para aplicar a operação *push*.

```
15      while (!queue.empty()) {
16          int v = queue.front();
17          for (int &p = ptr[v]; p < (int)g.adj[v].size(); p++) {
18              int i = g.adj[v][p];
19              int w = g.edges[i].v, u = g.edges[i].u;
20              if (h[w] < h[v] && u > 0) {
21                  int eps = min(e[v], u);
22                  g.edges[i].u -= eps;
23                  g.edges[i^1].u  += eps;
24                  if (e[w] == 0 && w != t) queue.push(w);
25                  e[w] += eps;
26                  e[v] -= eps;
27                  if (e[v] == 0) break;
```

```
28                  }
29              }
```

A linha 20 contém a condição para a aplicação da operação *push* na aresta. Se a condição não for satisfeita, o algoritmo avança o ponteiro `ptr` e processa a aresta seguinte. Se a condição for satisfeita, então realizamos o *push* de v para w. Se este *push* fizer w passar de não ativo para ativo, então inserimos w na fila, como na linha 24.

A linha 27 verifica se a última operação *push* esgotou o excesso do vértice v. Se isto ocorreu, então v deixou de ser ativo. Assim, saímos do laço `for` sem avançar o ponteiro `ptr`, pois este último *push* pode ter sido não saturante, portanto a aresta atual poderia, a princípio, sofrer outra operação *push*.

Se o excesso de v continuou positivo, então o último *push* foi saturante. Neste caso, a aresta atual não pode mais receber fluxo, portanto avançamos o ponteiro `ptr` para continuar a descarga do vértice v.

Se o ponteiro `ptr` atingir o final da lista de adjacências de v, então o vértice v ainda tem excesso positivo mas não pode "empurrá-lo" por nenhuma aresta. Quando isso ocorre, precisamos aplicar uma operação *relabel* em v. Neste momento, verificamos se pode ocorrer um *gap relabelling*.

```
30              if (e[v] > 0) {
31                  int cur_h = h[v];
32                  if (h_count[cur_h] == 1 && 0 < cur_h && cur_h < g.n) {
33                      for (int w = 0; w < g.n; w++) {
34                          if (h[w] > g.n || h[w] < cur_h || w == s) continue;
35                          ptr[w] = 0;
36                          h_count[h[w]]--;
37                          h[w] = g.n + 1;
38                          h_count[h[w]]++;
39                      }
40                  }
41                  else {
42                      ptr[v] = 0;
43                      h_count[h[v]]--;
44                      h[v]++;
45                      h_count[h[v]]++;
46                  }
47              }
```

Se a condição da linha 32 for satisfeita, então v é o único vértice de altura `cur_h` antes do *relabel*. Portanto, após o *relabel*, nenhum vértice terá altura `cur_h`, logo podemos aplicar o *gap relabelling*. Note que todos os vértices cujas alturas são alteradas devido a esta operação devem ter seus ponteiros movidos de volta ao início das respectivas listas de adjacências.

Se um *gap relabelling* não pode ser aplicado, então realizamos o *relabel* normalmente, incrementando a altura de v e movendo seu ponteiro de volta ao início da lista de adjacências de v, conforme mostram as linhas 42-45.

Se o algoritmo sai do laço `for` da linha 17 após um *push* não saturante, simplesmente removemos v da frente da fila, concluindo uma iteração do algoritmo.

```
48              else queue.pop();
49          }
```

Quando não há mais vértices ativos o algoritmo termina e o pré-fluxo transforma-se num fluxo máximo de s a t. Assim, basta retornarmos o valor deste fluxo, que é igual ao excesso do vértice t.

```
50          return e[t];
```

```
51  }
```

Push-Relabel com critério da maior altura

Para implementar o algoritmo Push-Relabel com o critério de seleção da maior altura, temos que ser capazes de encontrar rapidamente o vértice ativo que possui maior altura.

Manteremos uma lista ligada `active_list` de todos os vértices ativos em cada altura, bem como uma variável `cur_h` que indicará a maior altura que possui um vértice ativo.

Suponha que o algoritmo esteja processando um vértice v com altura h. Se o vértice v sofre um *relabel*, então o valor de `cur_h` passa a ser $h + 1$. Se o algoritmo terminar de processar todos os vértices com altura h e nenhuma operação *relabel* foi aplicada, então a próxima altura a ser processada deve ser menor que h, portanto decrementamos o valor de `cur_h`. Isto nos permite encontrar o vértice ativo de maior altura em tempo amortizado constante.

O resto do algoritmo é bastante similar à implementação com o critério FIFO. Manteremos os mesmos vetores `ptr`, `h`, `e` e `h_count`. A novidade são as listas ligadas, que serão implementadas usando a classe `forward_list` da STL.

```
1   int maxflow(Graph &g, int s, int t) {
2     vector<int> ptr(g.n, 0), h(g.n, 0), e(g.n, 0), h_count(2 * g.n, 0);
3     vector<forward_list<int>> active_list(2 * g.n);
```

As inicializações feitas são análogas às da implementação anterior.

```
4       h[s] = g.n;
5       h_count[g.n] = 1, h_count[0] = g.n - 1;
6       for (int i: g.adj[s]) {
7           int v = g.edges[i].v, u = g.edges[i].u;
8           if (u == 0) continue;
9           if (e[v] == 0 && v != t)
10              active_list[0].push_front(v);
11          e[v] += u;
12          e[s] -= u;
13          g.edges[i].u -= u;
14          g.edges[i^1].u += u;
15      }
```

Identificaremos o fim do algoritmo verificando se o valor de `cur_h` fica negativo, que indica que não há mais vértices ativos a serem processados.

```
16      int cur_h = 0;
17      while (cur_h >= 0) {
```

Se não existem vértices ativos na altura atual, simplesmente decrementamos o valor de `cur_h` e voltamos ao início do `while`.

```
18          if (active_list[cur_h].empty()) {
19              cur_h--;
20              continue;
21          }
```

Após processar um vértice v, este vértice ou sofrerá um *relabel* ou deixará de ser ativo. Em ambos os casos, o vértice v sai da lista de vértices ativos de altura `cur_h`.

```
22          int v = active_list[cur_h].front();
23          active_list[cur_h].pop_front();
```

A implementação da operação de descarga do vértice v é análoga à implementação do critério FIFO.

```
24            for (int &p = ptr[v]; p < (int)g.adj[v].size(); p++) {
25                int i = g.adj[v][p];
26                int w = g.edges[i].v, u = g.edges[i].u;
27                if (h[w] < h[v] && u > 0) {
28                    int eps = min(e[v], u);
29                    g.edges[i].u -= eps;
30                    g.edges[i^1].u  += eps;
31                    if (e[w] == 0 && w != t)
32                        active_list[h[w]].push_front(w);
33                    e[w] += eps;
34                    e[v] -= eps;
35                    if (e[v] == 0) break;
36                }
37            }
```

Ao implementar o *gap relabelling*, temos que lembrar de remover os vértices ativos de suas respectivas listas quando sua altura é alterada. Note que não é necessário remover individualmente cada vértice das listas, pois se um vértice tem sua altura alterada, então todos os vértices de mesma altura também suas alturas alteradas. Assim, podemos simplesmente apagar a lista inteira. Não podemos esquecer também de alterar o valor de `cur_h` para $|V| + 1$ após um *gap relabelling*.

```
38            if (e[v] > 0) {
39                if (h_count[cur_h] == 1 && 0 < cur_h && cur_h < g.n) {
40                    for (int w = 0; w < g.n; w++) {
41                        if (h[w] > g.n || h[w] < cur_h || w == s) continue;
42                        if (e[w] > 0) {
43                            active_list[h[w]].clear();
44                            active_list[g.n + 1].push_front(w);
45                        }
46                        ptr[w] = 0;
47                        h_count[h[w]]--;
48                        h[w] = g.n + 1;
49                        h_count[h[w]]++;
50                    }
51                    cur_h = g.n + 1;
52                }
53                else {
54                    ptr[v] = 0;
55                    h_count[h[v]]--;
56                    h[v]++;
57                    h_count[h[v]]++;
58                    active_list[h[v]].push_front(v);
59                    cur_h++;
60                }
61            }
62        }
63        return e[t];
64    }
```

Capítulo 4

Árvores de Gomory-Hu

Neste capítulo estudaremos cortes em redes não-dirigidas. Podemos estender o conceito de fluxo para este tipo de rede considerando cada aresta não-dirigida (v, w) de capacidade u como um par de arestas dirigidas (v, w) e (w, v), ambas de capacidade u.

Dada uma rede não-dirigida $G = (V, E, u)$ e dois vértices s e t, denotaremos por $f_G(s, t)$ (ou simplesmente $f(s, t)$, quando não houver ambiguidade) o valor de um fluxo máximo entre s e t. A **capacidade** de um corte S será denotada por $\text{cap}(S)$ e corresponde à soma das capacidades das arestas em $E(S, S^c)$. Note que $\text{cap}(S) = \text{cap}(S^c)$.

Estamos particularmente interessados no seguinte problema:

Problema (Corte mínimo entre todos os pares de vértices). *Dada uma rede capacitada não-dirigida, determinar um corte mínimo para todos os pares de vértices de uma rede.*

Este problema tem uma solução trivial: executar o algoritmo do fluxo máximo para todos os $n(n-1)/2$ pares de vértices. Veremos que, para redes não-dirigidas, é possível representar todos estes cortes de maneira bastante compacta.

Considere uma rede não-dirigida $G = (V, E, u)$. Dizemos que uma árvore capacitada $T = (V, E', u')$ é uma **árvore de Gomory-Hu** de G quando possui os mesmos vértices de G e satisfaz as seguintes propriedades:

i. (*Fluxo-equivalência*) Para qualquer par s, t de vértices, $f_G(s, t) = f_T(s, t)$.

ii. (*Cortes mínimos*) Todo s-t corte mínimo de T é também um s-t corte mínimo de G.

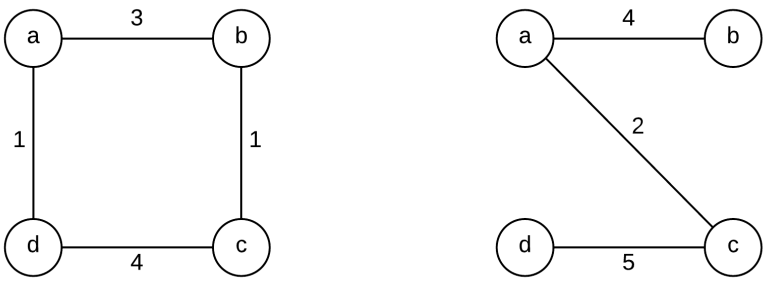

Figura 4.1: Uma rede G e uma árvore de Gomory-Hu de G.

A grande vantagem da árvore de Gomory-Hu está no fato dela ser uma representação compacta dos cortes mínimos de G no seguinte sentido: dados dois vértices s e t, se e é uma

aresta de capacidade mínima no (único) caminho de s a t, então, $f_T(s,t) = u'(e)$ e as duas componentes conexas de $T - e$ formam um s-t corte mínimo.

Além disso, toda rede não-dirigida possui uma árvore de Gomory-Hu. Provaremos este fato primeiro exibindo uma árvore, e depois mostrando que ela satisfaz as propriedades acima.

4.1 Algoritmo de Gomory-Hu

Construiremos a árvore de Gomory-Hu de uma rede $G = (V, E, u)$ forma algorítmica. O algoritmo, denominado algoritmo de Gomory-Hu (Gomory e Hu, 1961), mantém uma partição $\mathcal{P} = \{S_1, \ldots, S_k\}$ de V e uma árvore T sobre $\mathcal{P}$, e consiste nos seguintes passos:

1. Inicialmente, $k = 1, \mathcal{P} = \{S_1 = V\}$ e T consiste de um único "super vértice" contendo todos os vértices de G.
2. Enquanto algum S_i contiver mais de um vértice, repita os passos 3 a 7:
3. Determine as componentes conexas da floresta obtida da remoção do super vértice S_i de T. Cada uma destas componentes corresponde a um subconjunto de V.
4. Seja H a rede obtida da contração de cada uma dessas componentes em um único vértice. Ou seja, para cada componente C_j, remova todas as arestas de G que incidem em dois vértices de C_j, substitua todos os vértices de C_j por um único vértice c_j e redirecione as arestas que incidem em um único vértice de C_j para o vértice c_j, mantendo sua capacidade.
5. Sejam a e b dois vértices distintos de S_i. Encontre um a-b corte mínimo (A, B) em H.
6. Quebre S_i em dois conjuntos $S_i^a = S_i \cap A$ e $S_i^b = S_i \cap B$ e adicione uma aresta (S_i^a, S_i^b) em T com capacidade $u'(S_i^a, S_i^b) = f_H(a, b)$.
7. Redirecione toda aresta (S_x, S_i) de T para (S_x, S_i^a) se $S_x \subseteq A$ ou para (S_x, S_i^b), se $S_x \subseteq B$.

Antes de provar que a árvore gerada pelo algoritmo é, de fato, de Gomory-Hu, mostraremos alguns resultados que serão importantes para a demonstração.

Proposição 4.1. *Para quaisquer vértices $s, t, x \in V$, vale que $f(s,t) \geq \min\{f(s,x), f(x,t)\}$.*

Demonstração. Suponha que existam vértices $s, t, x \in V$ tais que $f(s,t) < \min\{f(s,x), f(x,t)\}$. Então, existe um s-t corte (A, B) cuja capacidade é $f(s,t)$. Se $x \in A$, então (A, B) é também um x-t corte, donde $f(x,t) \leq f(s,t)$. Se $x \in B$, então (A, B) é um s-x corte, donde $f(s,x) \leq f(s,t)$. Em ambos os casos, chegamos numa contradição. □

Aplicando a proposição 4.1 indutivamente, obtemos o seguinte corolário:

Corolário 4.1. *Se $(s = x_1, x_2, \ldots, x_{k-1}, x_k = t)$ é uma sequência de vértices, então*

$$f(s,t) \geq \min\{f(x_1, x_2), f(x_2, x_3), \ldots, f(x_{k-1}, x_k)\}.$$

Proposição 4.2. *Seja X um x-y corte mínimo em G. Sejam $u, v \in X$ e U um u-v corte mínimo em G. Então, existe um u-v corte mínimo W tal que $W \subseteq X$, se $y \in U^c$, ou $W^c \subseteq X$, se $y \in U$, e que separa X da mesma forma que U, ou seja, $X \cap W = X \cap U$.*

Demonstração. Suponha $y \in U$. Mostraremos que $\text{cap}(X \setminus U) + \text{cap}(U \setminus X) \leq \text{cap}(X) + \text{cap}(U)$.

Considere os cortes $X \setminus U$, $U \setminus X$, X e U. Seja e uma aresta de $E(X \setminus U) \cup E(U \setminus X)$. Então e pertence a exatamente um dos seguintes conjuntos: $E(X \setminus U, (X \cup U)^c)$, $E(X \setminus U, X \cap U)$, $E(X \setminus U, U \setminus X)$, $E(U \setminus X, X \cap U)$ ou $E(U \setminus X, (X \cup U)^c)$. Em todos esses casos, a capacidade da aresta e é contabilizada em $\text{cap}(X) + \text{cap}(U)$ ao menos tantas vezes quanto em $\text{cap}(X \setminus U) + \text{cap}(U \setminus X)$, donde decorre a desigualdade.

Como $U \setminus X$ é um y-x corte, temos que $\text{cap}(U \setminus X) \geq \text{cap}(X)$. Logo, $\text{cap}(X \setminus U) \leq \text{cap}(U)$. Mas $X \setminus U$ é um v-u corte, então $\text{cap}(X \setminus U) \geq \text{cap}(U)$, portanto $\text{cap}(X \setminus U) = \text{cap}(U)$. Assim, $W = (X \setminus U)^c$ é um u-v corte mínimo com $W^c \subseteq X$ e tal que $X \cap W = X \cap (X \setminus U)^c = X \cap U$.

Suponha agora que $y \in U^c$. De maneira bastante similar ao caso anterior, pode-se mostrar que $\text{cap}(U \cup X) + \text{cap}(U \cap X) \leq \text{cap}(X) + \text{cap}(U)$. Como $U \cup X$ é um x-y corte, temos que $\text{cap}(U \cup X) \geq \text{cap}(X)$, logo $\text{cap}(U \cap X) \leq \text{cap}(U)$. Mas $U \cap X$ é um u-v corte, então $\text{cap}(U \cap X) \geq \text{cap}(U)$, logo $\text{cap}(U \cap X) = \text{cap}(U)$, portanto $W = U \cap X$ é um u-v corte mínimo com $W \subseteq X$ e tal que $X \cap W = X \cap (U \cap X) = X \cap U$. □

Corolário 4.2. *Seja X um corte mínimo de G, e sejam $u, v \in V \setminus X$. Seja H a rede obtida da contração de X em G. Então, todo u-v corte mínimo em H é um u-v corte mínimo em G.*

Demonstração. Seja U um u-v corte mínimo em G. Pela proposição 4.2, existe um u-v corte mínimo W tal que $W \subseteq X^c$ ou $W^c \subseteq X^c$. Logo, W é um u-v corte em H. Como todo corte em H é também um corte em G e W é mínimo em G, segue que W é mínimo em H e qualquer outro u-v corte mínimo em H é também mínimo em G. □

Estamos prontos para demonstrar a corretude do algoritmo. Começaremos a prova mostrando uma invariante do algoritmo.

Proposição 4.3. *Em qualquer iteração do algoritmo de Gomory-Hu, se (S_i, S_j) é uma aresta de T, então existem vértices $a \in S_i$ e $b \in S_j$, denominados representantes, tais que $u'(S_i, S_j) = f_G(a, b)$ e o corte induzido pela aresta (S_i, S_j) é um a-b corte mínimo em G.*

Demonstração. A invariante é claramente válida no início do algoritmo. Mostraremos que se a invariante é válida no início de uma iteração, então ela se mantém após uma operação de quebra de um super vértice de T.

Seja S_i o super vértice selecionado em uma iteração do algoritmo, e sejam $a, b \in S_i$. Pela invariante, todas as arestas de T que incidem em S_i correspodem a algum corte mínimo de G. Portanto, pelo corolário 4.2, o a-b corte mínimo (A, B) obtido no passo 5 é também um a-b corte mínimo de G. Assim, (a, b) é um par de representantes para a aresta (S_i^a, S_i^b).

Para arestas de T que não incidem nem S_i^a nem em S_i^b pode-se usar o mesmo par de representantes da iteração anterior. Resta encontrar representantes para arestas do tipo (S_i^a, S_j) e (S_i^b, S_j).

Considere uma aresta do tipo (S_i^a, S_j) (o outro caso é análogo). Esta aresta surgiu da aresta (S_i, S_j), que estava presente em T no início da iteração. Seja (x_i, x_j) o par de representantes usados na iteração anterior. Se $x_i \in S_i^a$, então podemos usar o par (x_i, x_j) como representantes na iteração atual. Caso contrário, mostraremos que (a, x_j) pode ser tomado como um par de representantes. Para isto, basta mostrar que $f_G(a, x_j) = f_G(x_i, x_j)$.

Como a invariante era válida no início da iteração, temos que a aresta (S_i, S_j) induzia um corte em G de capacidade $f(x_i, x_j)$. Como a e x_j estão em lados opostos do corte, temos que $f(a, x_j) \leq f(x_i, x_j)$.

Seja G' a rede formada pela contração de B em G, e seja v_B o vértice em G' resultante dessa contração. Pelo corolário 4.2, como $a, x_j \notin B$, temos que $f_G(a, x_j) = f_{G'}(a, x_j)$.

Pela proposição 4.1, $f_{G'}(a, x_j) \geq \min\{f_{G'}(a, v_B), f_{G'}(v_B, x_j)\}$. Como $x_i \in B$, todo v_B-x_j corte em G' induz um x_i-x_j corte em G de mesma capacidade. Logo, $f_{G'}(v_B, x_j) \geq f_G(x_i, x_j)$. De forma similar, como $b \in B$, temos $f_{G'}(a, v_B) \geq f_G(a, b)$, e como B é um x_i-x_j corte, temos que $f_G(a, b) = \text{cap}(B) \geq f_G(x_i, x_j)$.

Daí, $f_{G'}(a, x_j) \geq f_G(a, x_j)$, de onde segue que $f_G(a, x_j) = f_G(x_i, x_j)$, como queríamos. □

Teorema 4.1. *A árvore T ao final do algoritmo é uma árvore de Gomory-Hu de G.*

Demonstração. Sejam s e t dois vértices de G. Seja $(s = x_1, x_2, \ldots, x_k = t)$ o único caminho simples de s a t em T. Pela proposição 4.3, sabemos que $f_G(x_i, x_{i+1}) = u'(x_i, x_{i+1})$ para todo i em $\{1, \ldots, k-1\}$. Logo,

$$\begin{aligned} f_T(s,t) &= \min\{u'(x_i, x_{i+1}) : i \in \{1, \ldots, k-1\}\} \\ &= \min\{f_G(x_i, x_{i+1}) : i \in \{1, \ldots, k-1\}\} \\ &\leq f_G(s,t) \qquad \text{(proposição 4.1).} \end{aligned}$$

Seja (x_j, x_{j+1}) uma aresta de capacidade mínima no caminho de s a t. Novamente pela proposição 4.3, temos que (x_j, x_{j+1}) induz um s-t corte de capacidade $f_G(x_j, x_{j+1})$. Portanto, $f_G(s,t) \leq f_G(x_j, x_{j+1}) = f_T(s,t)$.

Logo, $f_G(s,t) = f_T(s,t)$, portanto T é fluxo-equivalente. Além disso, se (x_j, x_{j+1}) induz um s-t corte mínimo em T, como (x_j, x_{j+1}) também induz um s-t corte S em G de capacidade $f_G(x_j, x_{j+1}) = f_G(s,t)$, temos que S é mínimo. Logo T tem a propriedade dos cortes mínimos, portanto T é uma árvore de Gomory-Hu. □

4.2 Algoritmo de Gusfield

O algoritmo de Gomory-Hu tem complexidade complexidade de tempo $O(|V|(|E| + \tau))$, em que τ é o tempo gasto para resolver o problema do fluxo máximo. No entanto, o algoritmo apresentado depende de operações de contrações de vértices que, apesar de não serem de difícil implementação, deixam o código mais longo e complexo. Mostraremos que é possível modificar o algoritmo para que encontre uma árvore de Gomory-Hu sem a necessidade de contrair vértices. O resultado será um algoritmo curto e de simples implementação (assumindo que já se tenha uma subrotina para o cálculo do fluxo máximo entre dois vértices), conhecido como algoritmo de Gusfield (Gusfield, 1990).

A proposição abaixo mostra que, para determinar a partição do super vértice S_i no passo 6 do algoritmo de Gomory-Hu, podemos tomar um a-b corte mínimo arbitrário em G ao invés de um corte mínimo na rede contraída H.

Proposição 4.4. *Considere uma iteração qualquer do algoritmo de Gomory-Hu. Sejam S um super vértice desta iteração e a e b dois vértices distintos de S. Se (A, B) é qualquer a-b corte mínimo em G, então existe um a-b corte mínimo (X, Y) na rede contraída H tal que $S \cap X = S \cap A$ e $S \cap Y = S \cap B$, e a capacidade de ambos os cortes são iguais.*

Demonstração. Sejam $C_1, \ldots, C_r$ as componentes conexas definidas no passo 4 do algoritmo. Pela proposição 4.3, todo corte $(G - C_j, C_j)$ é um x_j-y_j corte mínimo em G, para algum $x_j \in G - C_j$ e $y_j \in C_j$.

Como a e b estão em $G - C_1$, podemos aplicar a proposição 4.2 aos cortes $(G - C_1, C_1)$ e (A, B) para obter um a-b corte mínimo (A_1, B_1) tal que

i. $A_1 \subseteq G - C_1 \Leftrightarrow C_1 \subseteq B_1$ ou $A_1^c \subseteq G - C_1 \Leftrightarrow C_1 \subseteq A_1$;

ii. $(G - C_1) \cap A = (G - C_1) \cap A_1$.

Como $S \subseteq G - C_1$, segue de (ii) que $S \cap A = S \cap A_1$ e $S \cap B = S \cap B_1$.

Considere agora o corte $(G - C_2, C_2)$. Como C_1 é disjunto de C_2 e $S \subseteq G - C_2$, então $S \cup C_1 \subseteq G - C_2$. Aplicando a novamente proposição 4.2 aos cortes $(G - C_2, C_2)$ e (A_1, B_1), podemos encontrar outro a-b corte mínimo (A_2, B_2) tal que

iii. $C_2 \subseteq B_2$ ou $C_2 \subseteq A_2$;

iv. $(G - C_2) \cap A_2 = (G - C_2) \cap A_1$;

v. $S \cap A_2 = S \cap A_1 = S \cap A$ e $S \cap B_2 = S \cap B_1 = S \cap B$.

Como $C_1 \subseteq G - C_2$, temos de (i) e (iv) que ou $C_1 \subseteq A_2$ ou $C_1 \subseteq B_2$, e que $C_1 \subseteq A_2$ se e somente se $C_1 \subseteq A_1$.

Aplicando indutivamente a proposição 4.2 usando o fato que C_i é disjunto a S e de cada outro C_j com $j \leq i - 1$, podemos obter um a-b corte mínimo (A_i, B_i) em G tal que

vi. $S \cap A_i = S \cap A$ e $S \cap B_i = S \cap B$;

vii. $(G - C_i) \cap A_i = (G - C_i) \cap A_{i-1}$;

De (vii), temos que para todo $j \leq i$, ou $C_j \subseteq A_i$ ou $C_j \subseteq B_i$, e $C_j \subseteq A_i$ se e somente se $C_j \subseteq A_j$.

Concluímos que (A_r, B_r) é um a-b corte mínimo em G tal que $S \cap A_r = S \cap A$ e $S \cap B_r = S \cap B$. Além disso, para todo $i \leq r$, temos ou $C_i \subseteq A_r$ ou $C_i \subseteq B_r$. Logo, toda componente C_i está inteiramente contida em um dos dois lados do corte (A_r, B_r). Assim, (A_r, B_r) induz um a-b corte (X, Y) em H de mesma capacidade de (A, B), e isto termina a demonstração. □

Corolário 4.3. *Para todo j, $C_j \subseteq A_r$ se e somente se $C_j \subseteq A_j$.*

Este corolário é uma das conclusões obtidas na demonstração da proposição 4.4, e será usado mais tarde.

Como já foi mencionado, a proposição 4.4 nos permite usar um a-b corte mínimo em G no passo 6 do algoritmo de Gomory-Hu, evitando a necessidade de computar a rede contraída H. No entanto, o passo 7 passa a ser mal-definido, uma vez que este corte pode "cruzar" outros super vértices de T.

Mostraremos como adaptar o algoritmo de Gomory-Hu para determinar corretamente como as reconexões devem ser feitas usando um a-b corte mínimo em G no passo 7.

Em cada iteração do algoritmo de Gomory-Hu, manteremos para cada super vértice S exatamente um vértice especial chamado de **representante** de S e denotado por $r(S)$. No início do algoritmo, um vértice arbitrário é escolhido como representante do único super vértice da árvore T. No passo 5, iremos impor a restrição de que o par de vértices escolhido deve ser $r(S_i)$ e v para algum $v \in S_i, v \neq r(S_i)$. Ao quebrarmos o conjunto S_i em $S_i^{r(S_i)}$ e S_i^v, definimos $r(S_i)$ como representante de $S_i^{r(S_i)}$ e v como representante de S_i^v.

Com as modificações acima, obtemos a seguinte proposição:

Proposição 4.5. *Em qualquer iteração do algoritmo de Gomory-Hu modificado, se (S_i, S_j) é uma aresta de T, então $u'(S_i, S_j) = f_G(r(S_i), r(S_j))$ e o corte induzido pela aresta (S_i, S_j) é um $r(S_i)$-$r(S_j)$ corte mínimo em G.*

A proposição acima é uma reescrita da proposição 4.3, cuja demonstração pode ser facilmente adaptada para provar a versão dada pela proposição 4.5.

A proposição seguinte exibe uma forma de determinar corretamente as reconexões que devem ser feitas no passo 7 do algoritmo modificado:

Proposição 4.6. *Considere uma iteração do algoritmo de Gomory-Hu modificado e seja S um super vértice desta iteração. Seja v um vértice de S diferente de $r(S)$, e seja (A, B) um $r(S)$-v corte mínimo qualquer de G. Então, a seguinte regra decide corretamente se um vizinho S' de S em T deve ser conectado a $S^{r(S)}$ ou S^v no passo 7: Se $r(S') \in A$, então conectamos S' a $S^{r(S)}$. Caso contrário, conectamos S' a S^v.*

Demonstração. Sejam $S_1, \ldots, S_r$ os vizinhos de S em T, e $C_1, \ldots, C_r$ as componentes conexas correspondentes definitas no passo 4. Pela proposição 4.5, para todo $j \leq r$, temos que $(G - C_j, C_j)$ é um $r(S)$-$r(S_j)$ corte mínimo em G. Seja então (A_i, B_i), $1 \leq i \leq r$, uma sequência de $r(S)$-v cortes mínimos construídos como na demonstração da proposição 4.4. Mostraremos que $C_j \subseteq A_r$ se e somente se $r(S_j) \in A$. Assim, como (A_r, B_r) define um $r(S)$-v corte mínimo na rede contraído H, a regra descrita na proposição torna-se equivalente ao passo 7 do algoritmo original.

Pelo corolário 4.3, temos que $C_j \subseteq A_r$ se e somente se $C_j \subseteq A_j$. Então, basta provarmos que $C_j \subseteq A_j$ se e somente se $r(S_j) \in A$.

Como (A_j, B_j) foi obtido aplicando a proposição 4.2 ao $r(S)$-$r(S_j)$ corte $(G - C_j, C_j)$ e ao $r(S)$-v corte (A_{j-1}, B_{j-1}), então vale que $A_j^c \subseteq (G - C_j)$ se e somente se $r(S_j) \in A_{j-1}$. Mas $A_j^c \subseteq (G - C_j) \Leftrightarrow C_j \subseteq A_j$, então $r(S_j) \in A_j$ se e somente se $r(S_j) \in A_{j-1}$. Além disso, como $r(S_j) \in (G - C_{j-1})$ e $(G - C_{j-1}) \cap A_{j-1} = (G - C_{j-1}) \cap A_{j-2}$, então $r(S_j) \in A_{j-1}$ se e somente se $r(S_j) \in A_{j-2}$. Aplicando o mesmo argumento indutivamente para todo $1 < k < j$, obtemos que $r(S_j) \in A_k$ se e somente se $r(S_j) \in A_{k-1}$. Como $(G - C_1) \cap A_1 = (G - C_1) \cap A$, segue que $r(S_j) \in A_j$ se e somente se $r(S_j) \in A$, como queríamos. □

O algoritmo de Gomory-Hu com as modificações descritas anteriormente é chamado de algoritmo de Gusfield, e apresenta a grande vantagem de não depender de contrações de vértices como no algoritmo original, sem prejudicar sua complexidade assintótica. Abaixo descrevemos o algoritmo de Gusfield como uma adaptação dos passos 1 a 7 do algoritmo de Gomory-Hu.

1. Inicialmente, $\mathcal{P} = \{V\}$ e T consiste de um único "super vértice" contendo todos os vértices de G. Definimos um vértice arbitrário $r(V)$ como representante do único super vértice de T.

2. Enquanto algum S_i contiver mais de um vértice, repita os passos 3 a 7:

3. (Este passo torna-se desnecessário).

4. (Este passo torna-se desnecessário).

5. Seja v um vértice de S_i diferente de $r(S_i)$. Encontre um $r(S_i)$-v corte mínimo (A, B) em G.

6. Quebre S_i em dois conjuntos $S_i^{r(S_i)} = S_i \cap A$ e $S_i^v = S_i \cap B$ e adicione uma aresta $(S_i^{r(S_i)}, S_i^v)$ em T com capacidade $u'(S_i^{r(S_i)}, S_i^v) = f_G(r(S_i), v)$. Defina $r(S_i)$ como representante de $S_i^{r(S_i)}$ e v como representante de S_i^v.

7. Redirecione toda aresta (S_x, S_i) de T para $(S_x, S_i^{r(S_i)})$ se $r(S_x) \in A$ ou para (S_x, S_i^v) se $r(S_x) \in B$.

4.3 Implementação

Nesta seção apresentaremos uma implementação do algoritmo de Gusfield para a determinação de uma árvore de Gomory-Hu de uma rede. Omitiremos a parte do código que trata da representação da rede, assumindo que esta segue o padrão definido na seção 3.1 para redes não-dirigidas. Assumiremos também a existência de uma função `maxflow(g, s, t)` que computa o valor do fluxo máximo de s a t na rede g. A escolha do algoritmo de fluxo máximo definirá a complexidade do algoritmo de Gusfield.

Convencionaremos que a árvore T mantida durante a execução do algoritmo é enraizada no super vértice que contém o vértice de índice 0. Além disso, o vértice de índice 0 é o representante do único super vértice no início do algoritmo.

Implementaremos uma função `gomory_hu()` que recebe como argumento uma rede e retorna uma árvore de Gomory-Hu desta rede.

```
1   Graph gomory_hu(Graph &g) {
```

Manteremos um vetor `p` indexado nos vértices. Para cada vértice v, se v for o representante de seu super vértice S_v, então `p[v]` guardará o índice do representante do super vértice que é pai de S_v em T, com a excessão de `p[0]`, que guardará o valor 0. Se v não for o representante de S_v, então `p[v]` guardará o representante de S_v. Assim, ao término do algoritmo, a árvore de Gomory-Hu será definida pelas arestas $(i, \texttt{p[i]})$, com i variando de 1 a $|V| - 1$. A capacidade da aresta $(i, \texttt{p[i]})$ será `cap[i]`, sendo `cap` um vetor indexado pelos vértices que será computado durante a execução do algoritmo.

Inicializaremos as posições do vetor `p` com o valor 0, pois escolhemos o vértice com este índice para ser o representante do super vértice inicial.

```
2       vector<int> p(g.n, 0), cap(g.n);
```

Processaremos os vértices em ordem de 1 a $|V| - 1$. Assim, temos a seguinte invariante: no momento do processamento do vértice de índice i, os vértices de índices 0 a $i - 1$ são representantes de seus respectivos super vértices, enquanto os vértices de índices i a $|V| - 1$ não são representantes de nenhum super vértice.

Como nossas implementações dos algoritmos de fluxo máximo alteram o vetor `edges`, temos que salvar os valores iniciais para que uma chamada da função `maxflow()` não interfira no resultado de chamadas futuras. Salvaremos a rede original numa variável `original_g` e copiaremos seu conteúdo de volta para `g` ao término de cada iteração.

```
3       Graph original_g = g;
```

Agora iniciamos o laço principal do algoritmo, que corresponde aos passos 5 a 7. Iteraremos nos vértices em ordem de 1 a $|V| - 1$.

```
4       for (int s = 1; s < g.n; s++) {
```

O super vértice escolhido em cada iteração é o super vértice S que contém s, cujo representante é $t = \texttt{p[s]}$. Além disso, s é o vértice escolhido no passo 5. Assim, temos que computar um corte mínimo que deixa s e t em lados opostos. A capacidade deste corte será a capacidade da aresta que ligará o super vértice S_s, representado por s, e o super vértice S_t, representado por t, no fim da iteração. Iremos temporariamente assumir que S_s será filho de S_t em T. Verificaremos posteriormente se isto ocorre de fato, e corrigiremos a árvore se necessário.

```
5           int t = p[s];
6           cap[s] = maxflow(g, s, t);
```

Para definir como os super vértices S_s e S_t serão reconectados à árvore T e como os vértices de S ficarão distribuídos após sua quebra, temos que determinar os vértices que se encontram do lado de s no corte mínimo computado. Para isto, faremos uma busca em largura na rede residual partindo de s. O conjunto dos vértices alcançados pela busca é um s-t corte mínimo, como visto na demonstração da proposição 2.7. Os elementos deste conjunto serão identificados com o valor `true` no vetor `in_cut`.

```
 7           vector<bool> in_cut(g.n, 0);
 8           queue<int> queue({s});
 9           in_cut[s] = true;
10           while (!queue.empty()) {
11               int v = queue.front();
12               queue.pop();
13               for (int i: g.adj[v]) {
14                   int w = g.edges[i].v, u = g.edges[i].u;
15                   if (u > 0 && !in_cut[w]) {
16                       in_cut[w] = true;
17                       queue.push(w);
18                   }
19               }
20           }
```

Temos agora que atualizar o vetor `p` para que represente corretamente a árvore T após a quebra do super vértice S. A figura 4.3 mostra um exemplo das alterações que ocorrem em uma iteração do algoritmo. Em resumo, os seguintes vértices devem ter seu valor de `p` alterados:

- Vértices de S diferentes de s que estão do lado de s do corte mínimo;
- Representantes de super vértices que são vizinhos de S.

As linhas 21-23 tratam da maioria destes vértices. Resta considerar o representante do super vértice que é pai de S em T, caso qual exige mais cuidado.

```
21           for (int v = 0; v < g.n; v++)
22               if (v != s && in_cut[v] && p[v] == t)
23                   p[v] = s;
```

Quando o representante do super vértice que é pai de S em T está do lado de s do corte mínimo, temos que conectar este super vértice a S_s. Neste caso, S_t passa a ser filho de S_s. Assim, temos que mudar o valor de `p[s]` para `p[t]`, em seguida mudar o valor de `p[t]` para s e trocar os valores de `cap[s]` e `cap[t]`, conforme a figura 4.2.

```
24           if (in_cut[p[t]]) {
25               p[s] = p[t];
26               p[t] = s;
27               swap(cap[s], cap[t]);
28           }
```

Ao término da iteração, temos que mudar os elementos de `edges` para seus valores iniciais. Basta copiarmos o conteúdo de `original_g` para `g`.

```
29           g = original_g;
30       }
```

Por fim, temos que construir a árvore de Gomory-Hu. Como já vimos, as arestas que constituem esta árvore são da forma $(i, \texttt{p[i]})$, com i variando de 1 a $|V| - 1$.

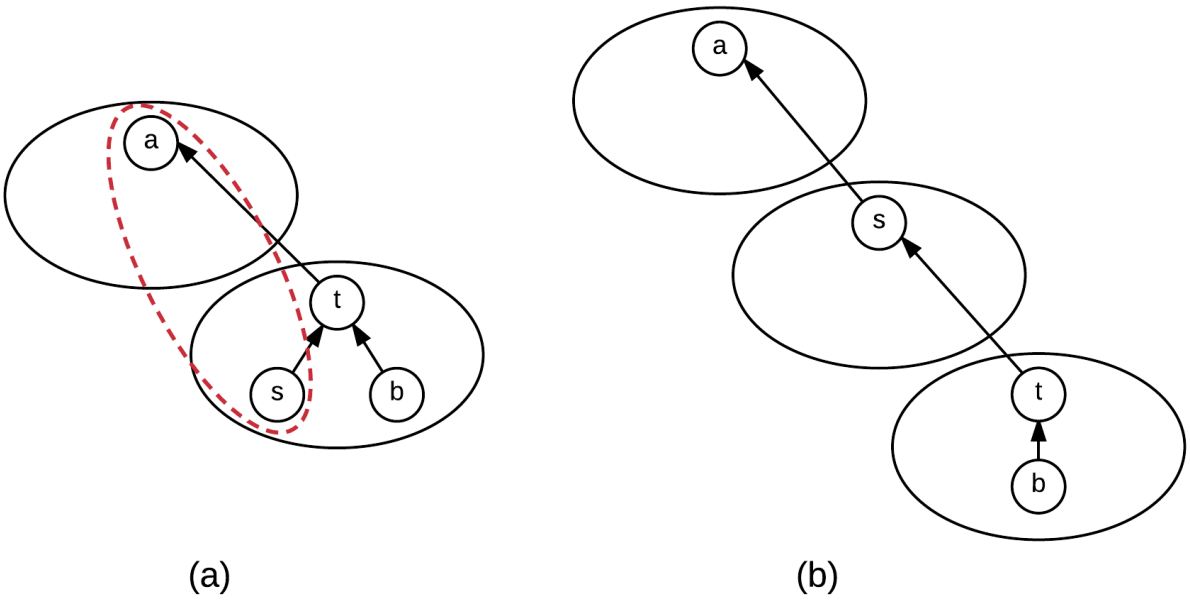

Figura 4.2: Resultado de uma iteração quando o representante do super vértice que é pai de S em T, indicado pelo rótulo 'a' na figura, está do lado de s no corte mínimo, indicado pela linha tracejada. Uma seta de i para j indica `p[i] = j`.

```
31        Graph tree(g.n);
32        for (int v = 1; v < g.n; v++)
33            tree.add_edge(v, p[v], cap[v]);
34        return tree;
35    }
```

A inicialização feita das linhas 4-5 consome tempo $O(|E|)$. Dentro do laço `for` da linha 6, que repete $|V|-1$ vezes, o cálculo do fluxo máximo leva tempo τ, a busca em largura $O(|E|)$ e a atualização do vetor `p` leva tempo $O(|V|)$. Assim, esta implementação do algoritmo de Gusfield tem complexidade de tempo $O(|V|(|E|+\tau))$, em que τ é o tempo gasto no cálculo de um fluxo máximo.

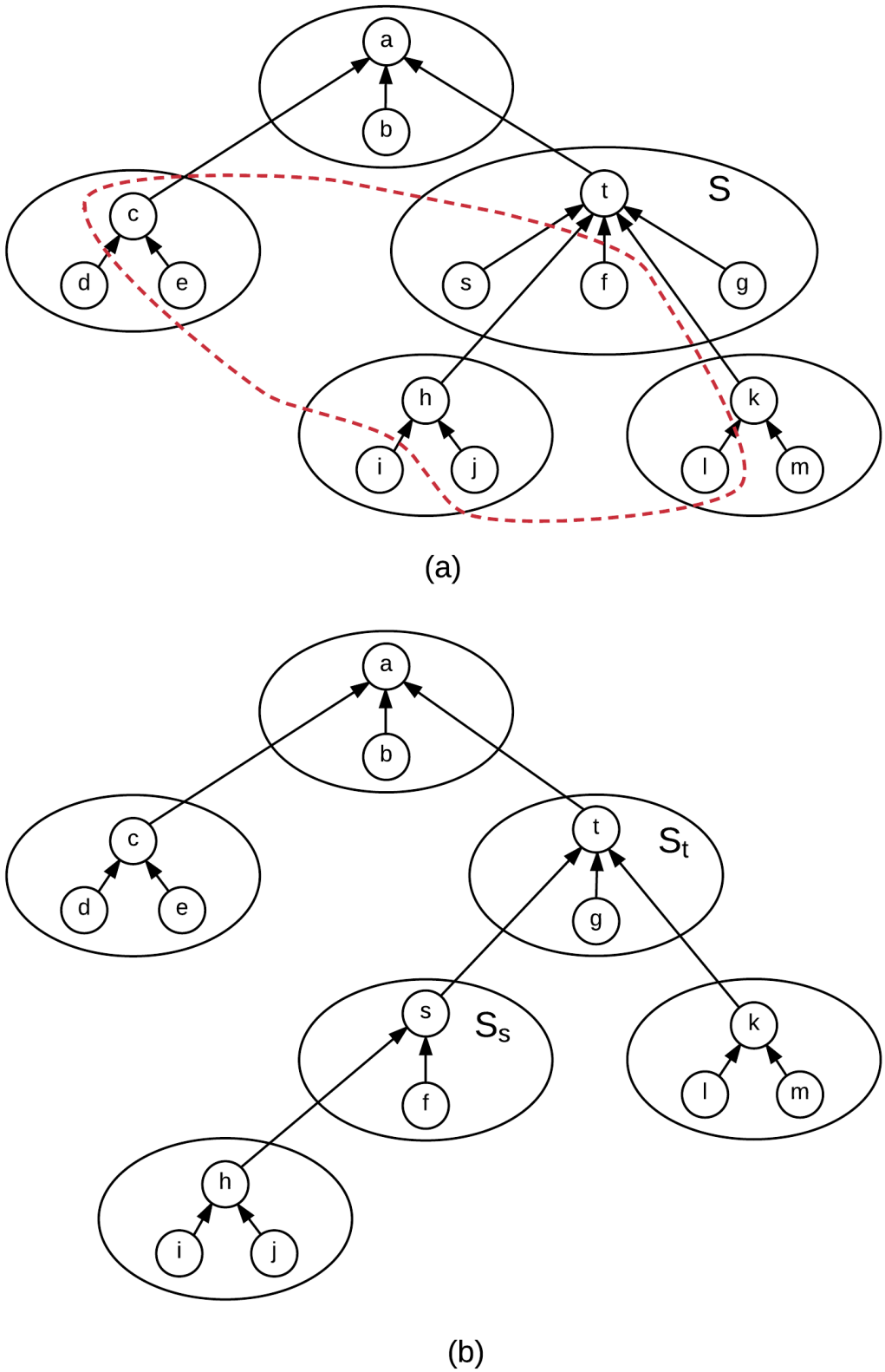

Figura 4.3: Exemplo de uma iteração do algoritmo de Gusfield. A figura (a) mostra a árvore T antes da quebra do super vértice S com o corte mínimo computado representado pela linha tracejada. Uma seta de i para j indica `p[i] = j`. A figura (b) mostra a árvore T ao término da iteração.

Capítulo 5

Cortes mínimos globais

Neste capítulo estudaremos mais um problema restrito a redes não-dirigidas:

Problema (Corte mínimo global). *Dada uma rede capacitada não-dirigida $G = (V, E, u)$, determinar o corte de capacidade mínima dentre todos os cortes de G, denominado* ***corte mínimo global****.*

Um modo equivalente de enunciar o problema é: dado um grafo não-dirigido com custos em suas arestas, encontrar quais arestas devem ser removidas para que o grafo se torne desconexo e a soma dos custos das arestas removidas seja a menor possível.

A forma direta de se resolver este problema seria calcular o corte mínimo entre todos os pares de vértices. Isto exigiria $|V|(|V|-1)/2$ execuções do algoritmo de fluxo máximo entre dois vértices. Uma forma um pouco mais eficiente seria determinar a árvore de Gomory-Hu da rede e tomar a aresta de capacidade mínima desta árvore, exigindo somente $|V|-1$ computações de fluxo máximo. Neste capítulo, estudaremos uma solução simples e eficiente que, surpreendentemente, não faz uso de nenhum algoritmo externo para o cálculo de fluxos máximos.

5.1 Algoritmo de Stoer-Wagner

O algoritmo de Stoer-Wagner (Stoer e Wagner, 1997) para a determinação do corte mínimo global baseia-se na seguinte ideia: dados dois vértices distintos s e t de G, ou o s-t corte mínimo é também um corte mínimo global, ou s e t estão do mesmo lado do corte mínimo global. No caso de s e t estarem do mesmo lado do corte, podemos contrair s e t em um único vértice e repetir o processo, tomando outro par de vértices.

O algoritmo pode ser resumido nos seguintes passos:

1. Repetir os passos 2 e 3 $|V|-1$ vezes:
2. Encontrar um s-t corte mínimo em G, para algum par de vértices (s, t).
3. Contrair os vértices s e t em G. Ou seja, substituir os vértices s e t por um vértice w, e redirecionar toda aresta (v, s) ou (v, t), com $v \neq s$ e $v \neq t$, para (v, w).
4. Devolver o corte de menor capacidade obtido no passo 2.

A parte importante deste algoritmo está em realizar o passo 2 de forma eficiente. Veremos que é possível usar o fato de que s e t não são fixos para para encontrar um s-t corte mínimo de forma bastante similar ao algoritmo de Prim.

O passo 2 do algoritmo é detalhado a seguir.

2.1. Crie um conjunto A de vértices, inicialmente contendo um vértice v arbitrário.

2.2. Enquanto $A \neq V$, repita o passo 2.3:

2.3. Para todo vértice v em $V \setminus A$, defina o *peso* de v por $u(v, A)$, ou seja, a soma das capacidades das arestas que ligam v a algum vértice de A. Insira em A o vértice de maior peso.

2.4. Seja t o último vértice inserido em A. Então, tome $(G \setminus \{t\}, \{t\})$ como corte mínimo.

Aqueles familiarizados com o algoritmo de Prim para árvores geradores mínimas irão notar bastante similaridade entre os dois algoritmos. De fato, se alterássemos a definição de peso para $\max\{u(v, a) : (v, a) \in E(\{v\}, A)\}$ e guardássemos as arestas cuja capacidade atinge esse máximo, o algoritmo acima se transforma no algoritmo de Prim, sendo A o conjunto dos vértices na árvore geradora a cada iteração do algoritmo.

Devido a grande similaridade entre os dois algoritmos, é de se esperar que as implementações e complexidades também sejam parecidas. Veremos em breve que isto realmente ocorre. Antes disso, precisamos primeiro nos convencer de que o algoritmo funciona.

Proposição 5.1. *Sejam s e t dois vértices distintos de G, e seja G' a rede resultante da contração dos vértices s e t. Então, pelo menos uma das seguintes afirmações é verdadeira:*

- *Todo s-t corte mínimo em G é um corte mínimo global de G;*
- *Todo corte mínimo global de G' é um corte mínimo global de G.*

Demonstração. Seja (X, Y) um corte mínimo global de G. Se s e t estão em lados diferentes de (X, Y), então qualquer s-t corte mínimo em G tem a mesma capacidade de (X, Y), e portanto é mínimo global em G.

Caso contrário, (X, Y) induz um corte em G'. Como (X, Y) é mínimo em G e todo corte em G' induz um corte em G de mesma capacidade, então (X, Y) é corte mínimo global em G'. Como todo outro corte mínimo global de G' tem a mesma capacidade de (X, Y), segue que todo corte mínimo global de G' é também corte mínimo global de G. □

A proposição anterior pode ser aplicada indutivamente para provar a corretude do algoritmo de Stoer-Wagner dado pelos passos 1 a 4. Resta demonstrar que o corte encontrado no passo 2 é, de fato, um s-t corte mínimo, para algum par de vértices s e t de G.

Proposição 5.2. *Sejam s e t os dois últimos vértices inseridos em A durante o passo 2.3. Então, $(G \setminus \{t\}, \{t\})$ é um s-t corte mínimo.*

Demonstração. Seja (S, T) um s-t corte qualquer de G. Mostraremos que $(G \setminus \{t\}, \{t\})$ tem capacidade menor ou igual à capacidade de (S, T).

Diremos que um vértice v é *ativo* (com respeito ao corte (S, T)) quando v e o vértice inserido em A imediatamente antes de v no passo 2.3 estão em lados diferentes de (S, T).

Seja A_v o conjunto de todos os vértices inseridos antes de v (excluindo v) e (S_v, T_v) a partição de $A_v \cup \{v\}$ induzida pelo corte (S, T), ou seja, $S_v = S \cap (A_v \cup \{v\})$ e $T_v = T \cap (A_v \cup \{v\})$. Mostraremos que para todo vértice ativo v, vale

$$u(A_v, v) = u(S_v, T_v),$$

por indução no conjunto de vértices ativo.

Para o primeiro vértice ativo, a desigualdade é satisfeita com igualdade, pois neste caso (S_v, T_v) e $(A_v, \{v\})$ particionam $A_v \cup \{v\}$ da mesma maneira. Suponha, então, que a igualdade

seja valida para todos os vértices ativos adicionados até o vértice ativo v, e seja w o próximo vértice ativo a ser adicionado. Então,

$$u(A_w, w) = u(A_v, w) + u(A_w \setminus A_v, w).$$

No momento da escolha de v, v era o vértice de maior peso, portanto $u(A_v, w) \leq u(A_v, v)$. Pela hipótese de indução, $u(A_v, v) \leq u(S_v, T_v)$. Logo,

$$u(A_w, w) \leq u(S_v, T_v) + u(A_w \setminus A_v, w).$$

Repare que $E(X_v, Y_v) \bigcap E(A_w \setminus A_v, w) = \varnothing$. Como todas as arestas nestes conjuntos pertencem à $E(S_w, T_w)$, temos que

$$u(A_w, w) \leq u(S_v, T_v) + u(A_w \setminus A_v, w) \leq u(S_w, T_w).$$

Como s é o vértice inserido antes de t, então t é sempre um vértice ativo. Logo, por indução,

$$u(A_t, t) \leq u(S_t, T_t) = u(S, T),$$

ou seja, $\text{cap}(G \setminus \{t\}, \{t\}) \leq \text{cap}(S, T)$, como queríamos. □

5.2 Implementação

Assim como foi feito na seção 4.3, omitiremos a parte do código que trata da representação da rede em questão, assumindo que foi utilizada a implementação da seção 3.1 para redes não-dirigidas.

Começaremos pela implementação de uma função `contract(g, s, t)`, que recebe uma rede g e dois vértices s e t e devolve a rede resultante da contração destes dois vértices.

```
1   Graph contract(Graph &g, int s, int t) {
2       Graph contracted(g.n - 1);
```

Os índices dos vértices na rede original serão os mesmos na rede contraída. O vértice resultante da contração de s e t terá o índice de s. Um problema que surge é que os índices dos vértices deixam de formar uma sequência contígua devido ao desaparecimento do vértice t. Para contornar isto, faremos com que o vértice de índice $|V| - 1$ passe a ter o índice que t tinha antes.

```
3       vector<int> label(g.n);
4       for (int v = 0; v < g.n; v++) label[v] = v;
5       swap(label[t], label[g.n - 1]);
```

Como cada aresta da rede original corresponde a duas arestas na rede residual, temos que tomar cuidado para não inserir arestas duplicadas na rede contraída. Assim, construiremos a rede contraída da seguinte forma:

- Para cada aresta (v, w) da rede residual de capacidade u, com $v, w \neq s, t$ e $v < w$, adicionaremos a aresta (v, w) na rede contraída com capacidade u;
- Para cada $v \neq s, t$, adicionaremos uma aresta (v, s) de capacidade $u(v, s) + u(v, t)$ na rede contraída, caso esta soma seja positiva.

```
 6       for (int v = 0; v < g.n; v++) {
 7           if (v == s || v == t) continue;
 8           int cap = 0;
 9           for (int i: g.adj[v]) {
10               int w = g.edges[i].v, u = g.edges[i].u;
11               if (w == s || w == t) cap += u;
12               else if (v < w) contracted.add_edge(label[v], label[w], u);
13           }
14           if (cap > 0) contracted.add_edge(label[v], label[s], cap);
15       }
16       return contracted;
17   }
```

Passamos agora a implementar a função `mincut(g)`, que recebe uma rede g e retorna a capacidade de um corte mínimo global de g.

```
18   int mincut(Graph g) {
```

A variável `mincap` guardará o mínimo das capacidades dos cortes encontrados em cada iteração do algoritmo.

```
19       int mincap = INF;
```

Repetiremos o laço principal do algoritmo enquanto a rede atual tiver mais de um vértice.

```
20       while (g.n > 1) {
```

Manteremos os pesos de cada vértice em um vetor `weight`, e os vértices dentro do conjunto A serão identificados pelo valor `true` no vetor `in_A`. As variáveis s e t guardarão o penúltimo e o último vértices a entrarem em A.

```
21           vector<int> weight(g.n, 0);
22           vector<bool> in_A(g.n, false);
23           int s = -1, t = -1;
```

Agora, iremos inserir todos os vértices em A em ordem decrescente de peso. Esta parte se assemelha bastante aos algoritmos de Dijkstra e Prim. Começamos por encontrar o vértice que não está em A e que tem maior peso.

```
24           for (int i = 0; i < g.n; i++) {
25               int v = -1;
26               for (int w = 0; w < g.n; w++) {
27                   if (in_A[w]) continue;
28                   if (v == -1 || weight[w] > weight[v]) v = w;
29               }
30               s = t;
31               t = v;
```

Optamos por uma implementação $O(|V|^2)$ por iteração do laço principal. Porém, assim como nos algoritmos de Dijkstra e Prim, podemos obter uma complexidade $O(|E| \log |V|)$ se usarmos uma fila de prioridade para determinar o vértice de maior peso.

Ao inserir o vértice v em A, temos que atualizar os pesos dos vértices vizinhos de v que estão fora de A.

```
32               for (int j: g.adj[v]) {
33                   int w = g.edges[j].v, u = g.edges[j].u;
34                   if (!in_A[w]) weight[w] += u;
35               }
36               in_A[v] = true;
37           }
```

Após inserir todos os vértices em A, temos que $(V \setminus \{t\}, \{t\})$ é um s-t corte mínimo cuja capacidade é igual ao peso de t.

```
38              mincap = min(mincap, weight[t]);
```

Terminamos uma iteração do laço principal do algoritmo contraíndo os vértices s e t. Quando o laço principal termina, a variável `mincap` contém a capacidade de um corte mínimo global da rede original.

```
39              g = contract(g, s, t);
40          }
41          return mincap;
42      }
```

O laço principal do algoritmo executa $|V| - 1$ vezes. Como a contração da rede leva tempo $O(|V| + |E|)$, cada iteração do algoritmo consome tempo $O(|V|^2)$, resultando numa complexidade total $O(|V|^3)$. Usando uma fila de prioridade na implementação do algoritmo, a complexidade passa a ser $O(|V||E| \log |V|)$.

Capítulo 6

Aplicações de fluxos máximos

6.1 Emparelhamentos e coberturas em grafos bipartidos

Um grafo não-dirigido (V, E) é dito **bipartido** quando existe uma bipartição (P, Q) do conjunto V tal que toda aresta de E tem uma ponta em P e outra em Q.

Dado um grafo bipartido G, um **emparelhamento** em G é um subconjunto M das arestas de G tal que todo vértice de G é ponta de no máximo uma aresta de M.

Nesta seção, apresentaremos uma solução para o seguinte problema:

Problema (Emparelhamento bipartido máximo). *Dado um grafo bipartido, encontrar um emparelhamento de cardinalidade máxima.*

Dado um grafo $G = (V, E)$ com bipartição (P, Q), considere a rede capacitada $G' = (V', E', u)$ construída da seguinte maneira:

- $V' = V \cup \{s, t\}$, sendo s e t dois novos vértices;
- Para cada aresta (p, q) de E, com $p \in P$ e $q \in Q$, temos uma aresta (p, q) em E' de capacidade infinita;
- Para cada vértice $p \in P$ temos uma aresta (s, p) em E' de capacidade unitária;
- Para cada vértice $q \in Q$ temos uma aresta (q, t) em E' de capacidade unitária.

Proposição 6.1. *O problema do emparelhamento bipartido máximo em G é equivalente ao problema do fluxo máximo de s a t em G'.*

Demonstração. Considere um emparelhamento máximo M em G. Construa um fluxo f em G' da seguinte maneira:

- Para cada $p \in P$, se existe uma aresta em M que incide em p, então tome $f(s, p) = 1$. Caso contrário, $f(s, p) = 0$.
- Para cada $q \in Q$, se existe uma aresta em M que incide em q, então tome $f(q, t) = 1$. Caso contrário, $f(q, t) = 0$.
- Para cada aresta e em E, se $e \in M$ então $f(e) = 1$. Caso contrário, $f(e) = 0$.

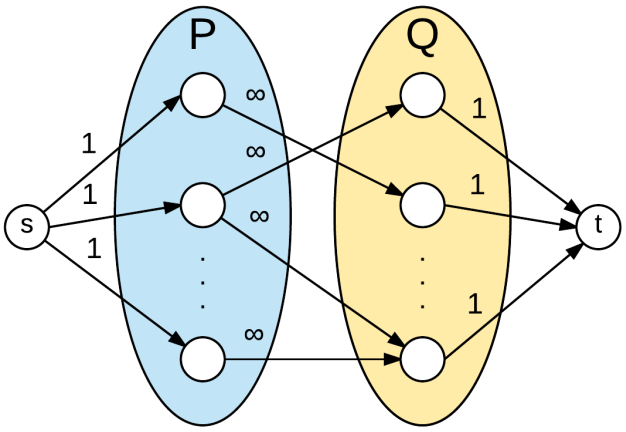

Figura 6.1: Exemplo de uma construção da rede G'. A restrição de G' ao conjunto $P \cup Q$ corresponde ao grafo original G.

A função f claramente satisfaz as condições de capacidade. Além disso, para cada vértice v diferente de s e t, se existe aresta em M que incide em v, então $f^+(v) = f^-(v) = 1$. Caso contrário, temos $f^+(v) = f^-(v) = 0$. Portanto, f é, de fato, um fluxo de s a t em G' cujo valor coincide com a cardinalidade de M.

Por outro lado, considere um fluxo máximo f em G'. Como as capacidades de G' são inteiras, podemos assumir que f é inteiro pelo corolário 2.1. Assim, o conjunto

$$M = \{(p, q) : p \in P, q \in Q, f(p, q) = 1\}$$

é um emparelhamento em G de cardinalidade $|f|$, pois para todo $p \in P$ e $q \in Q$, temos $f^-(p) = f^+(p) \leq u^+(p) = 1$ e $f^+(q) = f^-(q) \leq u^-(q) = 1$, portanto cada vértice de $P \cup Q$ pode ser ponta de no máximo uma aresta de M, e $|f| = f^+(t) = f^-(Q) = f^+(Q) = |M|$.

Como consequência, a cardinalidade de um emparelhamento máximo em G é igual ao valor de um fluxo máximo em G'. Como um pode ser obtido a partir do outro, concluímos que os dois problemas são equivalentes. □

Algoritmo de Hopcroft-Karp

Como o problema do emparelhamento máximo bipartido reduz para um problema de fluxo máximo, é natural pensarmos que o algoritmo mais apropriado para resolvê-lo é, por exemplo, o Push-Relabel, que possui complexidade menor que os outros algoritmos vistos. No entanto, o algoritmo de Dinic apresenta complexidade ainda menor quando aplicado a grafos bipartidos, como mostraremos a seguir. O algoritmo de Dinic aplicado à resolução do problema do emparelhamento bipartido máximo é conhecido como **algoritmo de Hopcroft-Karp**.

Proposição 6.2. *Seja (V, E) um grafo bipartido e G' a rede construída a partir de (V, E) para resolver o problema do emparelhamento máximo bipartido. Então, o algoritmo de Dinic encontra o fluxo máximo de s a t em G' em $O(|E|\sqrt{|V|})$.*

Demonstração. Considere uma fase do algoritmo de Dinic. Como todas as arestas da rede têm capacidade 1, o aumento do fluxo ao longo de um caminho satura todas as arestas deste caminho. Por consequência, cada fase leva tempo $O(|E|)$. Mostraremos, então, que o número de fases é $O(\sqrt{|V|})$.

Pela proposição 2.8, a distância de uma fase para a seguinte sempre aumenta. Portanto, após $\sqrt{|V|}$ fases, o comprimento de qualquer caminho de aumento é pelo menos $\sqrt{|V|}$.

Seja M o emparelhamento encontrado pelo algoritmo após $\sqrt{|V|}$ fases, e seja M^* um emparelhamento máximo. Considere o grafo $G_\triangle$ induzido pela diferença simétrica

$$M \triangle M^* = (M \cup M^*) \setminus (M \cap M^*).$$

Como cada vértice pertence a no máximo uma aresta em M e uma aresta em M^*, então o grau de cada vértice de $G_\triangle$ é 2. Portanto, todas as componentes conexas de $G_\triangle$ são ciclos ou caminhos simples.

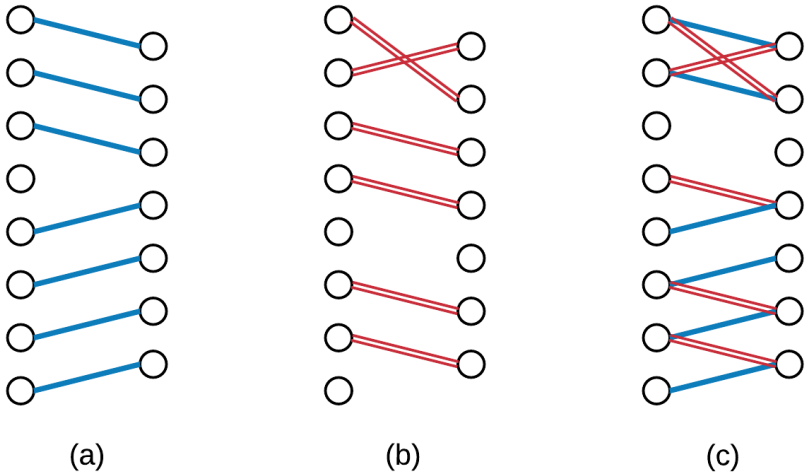

Figura 6.2: As figuras (a) e (b) mostram dois emparelhamentos M^* e M. A figura (c) mostra a diferença simétrica $M \triangle M^*$.

Note que não existem ciclos de comprimento ímpar em $G_\triangle$, pois (V, E) é bipartido. Como todas os ciclos e caminhos alternam entre arestas de M e M^*, então uma componente pode ter no máximo uma aresta de M^* a mais que M. Por consequência, há pelo menos $|M^*| - |M|$ destas componentes, que são caminhos de comprimento ímpar. Por fim, note que cada um destes caminhos corresponde a um caminho de aumento em G', portanto seu comprimento é pelo menos $\sqrt{|V|}$. Como o grafo tem $|V|$ vértices, não pode haver mais do que $\sqrt{|V|}$ caminhos deste tipo. Logo, $|M^*| - |M| \leq \sqrt{|V|}$. Assim, o algoritmo de Dinic precisa encontrar no máximo mais $\sqrt{|V|}$ caminhos de aumento. Como cada fase encontra ao menos um caminho de aumento, concluímos que o algoritmo termina em não mais do que $\sqrt{|V|}$ fases, totalizando no máximo $2\sqrt{|V|}$ fases durante todo o algoritmo. □

Coberturas mínimas

Dado um grafo bipartido G, uma **cobertura** é um subconjunto C dos vértices de G tal que toda aresta tem pelo menos uma ponta em C. Um problema que surge naturalmente é o seguinte:

Problema (Cobertura mínima). *Dado um grafo bipartido, encontrar uma cobertura de cardinalidade mínima.*

O problema da cobertura mínima e do emparelhamento máximo em grafos bipartidos estão bastante relacionados. De fato, é fácil verificar que toda cobertura tem cardinalidade maior que qualquer emparelhamento do grafo. Além disso, sempre existe uma cobertura e um emparelhamento de mesmas cardinalidades, conforme o teorema a seguir. Como consequência, temos que o tamanho de um emparelhamento máximo é igual ao tamanho de uma cobertura mínima. Esta relação é muito similar à relação entre fluxos e cortes em redes capacitadas.

Teorema 6.1 (König). *Seja G um grafo bipartido. Então existe uma cobertura e um emparelhamento em G de mesmas cardinalidades.*

Demonstração. Seja (P, Q) uma bipartição de G. Seja G' a rede construída como na proposição 6.1, e seja (S, T) um s-t corte mínimo em G'. Note que a capacidade deste corte é igual à cardinalidade de um emparelhamento máximo em G. Defina

$$C = (P \cap T) \cup (Q \cap S).$$

Então, C é uma cobertura de G. De fato, se existisse uma aresta (p, q) não coberta por C, com $p \in P$ e $q \in Q$, então teríamos $p \in S$ e $q \in T$. Mas isto implica que a capacidade de (S, T) seria infinita, contradizendo a minimalidade do corte.

Note, portanto, que somente arestas do tipo (s, p) e (q, t) contribuem para a capacidade do corte (S, T). Assim, $\text{cap}(S, T) = |P \cap T| + |Q \cap S| = |C|$. Como a capacidade de (S, T) é igual à cardinalidade de um emparelhamento máximo, temos que C é uma cobertura de mesma capacidade de um emparelhamento em G, como queríamos. □

Note que além de provar a igualdade entre a cardinalidade de um emparelhamento máximo e de uma cobertura mínima em um grafo bipartido, a demonstração do teorema de König também nos dá um algoritmo para determinar uma cobertura mínima dado um emparelhamento máximo.

6.2 Vértices com capacidades

Considere uma rede $G = (V, E, u_E, u_V)$ na qual u_E determina capacidades não negativas para as arestas da rede e u_V determina capacidades não-negativas para os vértices. Neste contexto, um **fluxo** de s a t é uma função f que atribui números não-negativos para cada aresta da rede e que satisfaz as seguintes restrições:

1. (*Condições de conservação*) Para cada vértice v diferente de s e t, temos

$$f^+(v) = f^-(v).$$

2. (*Condições de capacidade para arestas*) Para cada aresta $e \in E$, temos

$$0 \leq f(e) \leq u_E(e);$$

3. (*Condições de capacidade para vértices*) Para cada vértice $v \in V$ diferente de s e t, temos

$$f^+(v) = f^-(v) \leq u_V(v);$$

Além disso,

$$f^-(s) \leq u_V(s) \quad \text{e} \quad f^+(t) \leq u_V(t).$$

O **valor** do fluxo f é definido de forma usual:

$$|f| = f^+(t) - f^-(t) = f^-(s) - f^+(s).$$

Naturalmente, estamos interessados no seguinte problema:

Problema (Fluxo máximo com capacidades nos vértices). *Dada uma rede $G = (V, E, u_E, u_V)$ e dois vértices s e t de G, encontrar um fluxo de s a t de valor máximo.*

Note que, se ignorarmos a restrição (3), o problema se transforma no problema tradicional do fluxo máximo, estudado no capítulo 2. As condições de capacidade para vértices limitam a quantidade de fluxo que pode "fluir" por cada vértice, assim como as condições para arestas limitam a quantidade de fluxo que pode "fluir" por cada aresta.

O problema do fluxo máximo com capacidades nos vértices pode ser reduzido para o problema do fluxo máximo tradicional de maneira bastante simples. Considere uma rede $G' = (V', E', u')$ obtida da seguinte transformação de G:

i. Para cada vértice $v \in V$, temos dois vértices v_i e v_o em V';

ii. Para cada aresta (v, w) de E, temos uma aresta (v_o, w_i) em E' com $u'(v_o, w_i) = u_E(v, w)$;

iii. Para cada vértice $v \in V$, temos uma aresta (v_i, v_o) em E' com $u'(v_i, v_o) = u_V(v)$.

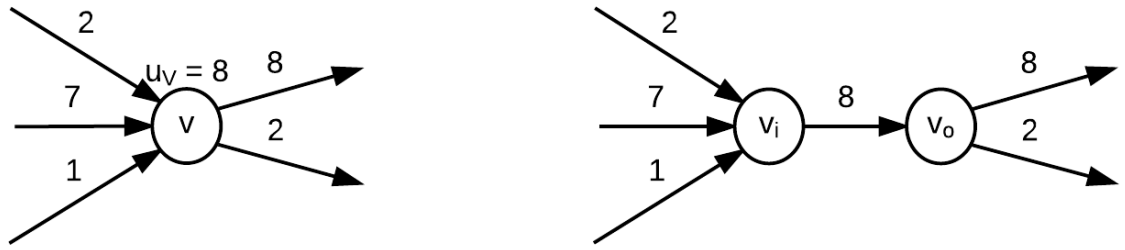

Figura 6.3: Um vértice v em G e sua respectiva transformação em G'.

Todo fluxo f de s a t em G pode ser transformado em um fluxo (usual) f' de s_i a t_o em G' de mesmo valor. De fato, podemos construir f' da seguinte forma:

- $f'(v_o, w_i) = f(v, w)$ para toda aresta $(v, w) \in E$;
- $f'(v_i, v_o) = f^-(v)$ para todo vértice $v \in V$ diferente de t;
- $f'(t_i, t_o) = f^+(t)$.

Por construção, para todo vértice $v \in V$, vale $f^+(v) = f'^+(v_i)$ e $f^-(v) = f'^-(v_o)$. Como f respeita as condições de conservação, temos $f^+(v) = f^-(v)$ para todo vértice v diferente de s e t, logo

$$f'^+(v_i) - f'^-(v_i) = f^+(v) - f^-(v) = 0 \quad \text{e} \quad f'^+(v_o) - f'^-(v_o) = f^-(v) - f^-(v) = 0.$$

Além disso, para o vértice s_o, temos

$$f'^+(s_o) - f'^-(s_o) = f^-(s) - f^-(s) = 0$$

e para o vértice t_i, temos

$$f'^{+}(t_i) - f'^{-}(t_i) = f^{+}(t) - f^{+}(t) = 0.$$

Portanto, f' satisfaz as condições de conservação.

A função f' também satisfaz as condições de capacidade, pois para arestas do tipo (ii), temos

$$f'(v_o, w_i) = f(v, w) \leq u_E(v, w) = u'(v_o, w_i)$$

e para arestas do tipo (iii), temos

$$f'(v_i, v_o) = f^{-}(v) \leq u_V(v) = u'(v_i, v_o), \text{ se } v \neq t$$
$$f'(t_i, t_o) = f^{+}(t) \leq u_V(t) = u'(t_i, t_o), \text{ se } v = t.$$

Portanto, f' é, de fato, um fluxo de s_i a t_o em G' de valor

$$|f'| = f'^{+}(t_o) - f'^{-}(t_o) = f^{+}(t) - f^{-}(t) = |f|.$$

Reciprocamente, todo fluxo f' de s_i a t_o em G' pode ser transformado em um fluxo f de s a t em G de mesma capacidade, bastando tomar $f(v, w) = f'(v_o, w_i)$, para toda aresta (v, w) em E. Isto pode ser verificado de forma bastante similar ao caso anterior.

A conclusão é a seguinte: o problema do fluxo máximo de s a t na rede G com capacidade nos vértices é equivalente ao problema do fluxo máximo usual de s_i a t_o na rede G'.

6.3 Vértices com demandas

Considere uma rede capacitada $G = (V, E, u)$ na qual temos para cada vértice v um valor $b(v)$ associado chamado de **demanda**. Neste contexto, uma função f é chamada de **fluxo viável** quando atribui para cada aresta um valor não-negativo e satisfaz

i. $f(e) \leq u(e)$, para toda aresta $e \in E$;

ii. $f^{+}(v) - f^{-}(v) = b(v)$, para todo vértice $v \in V$.

Estamos interessados no seguinte problema:

Problema (Fluxo viável). *Dada uma rede $G = (V, E, u, b)$, em que u representa as capacidades das arestas e b representa as demandas dos vértices, encontrar um fluxo viável em G.*

A proposição abaixo dá uma condição necessária para a existência de fluxo viável.

Proposição 6.3 (Condições de Gale). *Se uma rede G admite fluxo viável, então*

i. Para todo subconjunto $A \subset V$, $\sum_{v \in A} b(v) \leq u^{+}(A)$;

ii. $\sum_{v \in V} b(v) = 0.$

Demonstração. Seja A um subconjunto de V e f um fluxo viável. Então

$$\begin{aligned}\sum_{v\in A} b(v) &= \sum_{v\in A}(f^+(v) - f^-(v)) && (f \text{ é viável})\\ &= f^+(A) - f^-(A) && (\text{proposição } 2.1)\\ &\leq f^+(A) && (f \text{ é não-negativa})\\ &\leq u^+(A) && (f \text{ é viável}).\end{aligned}$$

Pela proposição 2.1, também temos

$$\sum_{v\in V} b(v) = \sum_{v\in V}(f^+(v) - f^-(v)) = f^+(V) - f^-(V) = 0.$$

□

O teorema a seguir mostra que as condições de Gale são, além de necessárias, suficientes.

Teorema 6.2 (Gale). *Se uma rede $G = (V, E, u, b)$ satisfaz as condições de Gale, então existe um fluxo viável em G.*

Demonstração. Seja P o conjunto dos vértices com b negativo e Q o conjunto dos vértices com b positivo. Considere a rede $G' = (V', E', u')$ construida da seguinte forma:

- V' consiste dos vértices de V e dois vértices adicionais s e t;
- Para cada v em P, há uma aresta $e = (s, v)$ com capacidade $u'(e) = -b(v)$;
- Para cada v em Q, há uma aresta $e = (v, t)$ com capacidade $u'(e) = b(v)$;
- Para cada aresta $e = (v, w)$ de E, há uma aresta $e' = (v, w)$ com $u'(e') = u(e)$.

Seja R um s-t corte de G'. Seja $B = R \setminus \{r\}$. A capacidade deste corte é a soma de três parcelas: as capacidades das arestas que vão de s a $P \setminus B$, as capacidades das arestas que vão de $Q \cup B$ a t e as capacidades das arestas de G que saem de B. Assim,

$$\begin{aligned}\text{cap}(R) &= \sum_{v\in P\setminus B} u(e_{sv}) + \sum_{v\in Q\cup B} u(e_{vt}) + u^-(B)\\ &= \sum_{v\in P\setminus B} -b(v) + \sum_{v\in Q\cup B} b(v) + u^+(V \setminus B)\\ &\geq \sum_{v\in P\setminus B} -b(v) + \sum_{v\in Q\cup B} b(v) + \sum_{v\in V\setminus B} b(v) && (\text{condição (i) de Gale})\\ &= \sum_{v\in P\setminus B} -b(v) + \sum_{v\in Q\cup B} b(v) + \sum_{v\in P\setminus B} b(v) + \sum_{v\in Q\setminus B} b(v)\\ &= \sum_{v\in Q\cup B} b(v) + \sum_{v\in Q\setminus B} b(v)\\ &= \sum_{v\in Q} b(v).\end{aligned}$$

Logo, todo s-t corte de G' tem capacidade maior ou igual a $\sum_{v\in Q} b(v)$. Pelo teorema do fluxo máximo e corte mínimo (teorema 2.1), temos que existe em G' um fluxo f' de s a t tal que $|f'| \geq \sum_{v\in Q} b(v)$. Mas, aplicando a proposiçao 2.3 ao corte $V' \setminus \{t\}$, temos

$|f'| \leq \text{cap}(V' \setminus \{t\}) = \sum_{v \in Q} b(v)$, logo $|f'| = \sum_{v \in Q} b(v)$ e, portanto, f' satura todas as arestas que entram em t.

Como G satisfaz a condição (ii) de Gale, temos que

$$\sum_{v \in P} b(v) = \sum_{v \in V} b(v) - \sum_{v \in Q} b(v) = -\sum_{v \in Q} b(v).$$

Logo, f' também satura todas as arestas que saem de s. Portanto, a restrição f de f' a G é um fluxo viável. □

A prova acima nos dá um algoritmo para encontrar um fluxo viável em G ou determinar que não existe nenhum. Basta construirmos a rede G' descrita na demonstração e encontrar um fluxo máximo f' em G'. Se $|f'| = \sum_{v \in Q} b(v)$, então $f = f'|_E$ é um fluxo viável em G. Caso contrário, G não admite fluxo viável.

6.4 Circulações viáveis

Na seção anterior, vimos como encontrar uma fluxo viável que satisfaz uma dada demanda para cada vértice. Agora, estudaremos um problema com restrições similares, mas desta vez nas arestas.

Considere uma rede $G = (V, E, l, u)$ na qual temos dois valores associados a cada aresta e: um limite inferior $l(e)$ e um limite superior $u(e)$. Uma **circulação** é uma função f que associa cada aresta a um número e satisfaz $f^+(v) - f^-(v) = 0$ para todo vértice v de V. Uma circulação é chamada de **viável** quando, para cada aresta e, tem-se $l(e) \leq f(e) \leq u(e)$.

Naturalmente, o problema que estudaremos nesta seção consiste no seguinte:

Problema (Circulação viável). *Dada uma rede $G = (V, E, l, u)$, encontrar um circulação viável.*

Para resolver este problema, vamos inicialmente tomar uma função $f_1(e) = l(e)$ para toda aresta e da rede. É claro que f_1 pode não ser um fluxo válido, pois um vértice pode ter excesso diferente de zero. Para corrigir f_1, temos que encontrar valores não-negativos $f_2(e)$ de forma que $f_1 + f_2$ seja um fluxo válido e que satisfaça os limites superiores, ou seja:

i. $f_1^+(v) + f_2^+(v) = f_1^-(v) + f_2^-(v)$, para todo vértice $v \in V$

ii. $0 \leq f_2(e) \leq u(e) - f_1(e)$, para toda aresta $e \in E$

Reescrevendo a primeira restrição, temos $f_2^+(v) - f_2^-(v) = f_1^-(v) - f_1^+(v)$. Repare que, se interpretarmos o valor $f_1^-(v) - f_1^+(v)$ como uma demanda $b(v)$ do vértice v, as restrições acima indicam que f_2 é um fluxo viável numa rede com demandas. Assim, podemos usar o algoritmo descrito na seção anterior para encontrar f_2 e então somá-lo ao valor de $f_1 = l$ para obter uma circulação viável no problema original. Caso o fluxo f_2 não exista, o problema não tem solução.

6.5 Arestas com demandas

Considere agora um problema bastante similar ao anterior. Seja $G = (V, E, l, u)$ uma rede com limites inferiores e superiores para cada aresta. Dados dois vértices distintos s e t, chamaremos de **fluxo viável** um fluxo f de s a t que satisfaça as condições de conservação e respeite os limites de cada aresta, ou seja, $f^+(v) - f^-(v) = 0$ para todo vértice em $V \setminus \{s, t\}$ e $l(e) \leq f(e) \leq u(e)$ para toda aresta e de E. Estamos interessados no seguinte problema:

Problema (Fluxo mínimo). *Dada uma rede $G = (V, E, l, u)$, encontrar um fluxo viável de valor mínimo.*

Resolveremos este problemas em duas etapas. Primeiro, encontraremos um fluxo viável e, em seguida, transformaremos este fluxo num fluxo de intensidade mínima.

Para encontrar um fluxo viável, considere a rede $G' = (V, E \cup \{e_{ts}\}, l, u)$ resultante da adição em G de uma aresta $e_{ts} = (t, s)$ com $l(e_{ts}) = 0$ e $u(e_{ts}) = \infty$. Note que todo fluxo f de s a t em G corresponde a uma circulação f' em G' com $|f| = f'(e_{ts})$ e que f é viável em G se e somente se f' for viável em G'. Assim, podemos usar o algoritmo visto na seção anterior para encontrar um fluxo viável f em G.

Para transformar f num fluxo mínimo, usaremos a mesma ideia do algoritmo de Ford-Fulkerson. Reduziremos o valor de f por meio de caminhos de aumento de t a s em G até que não haja mais tal caminho. Neste momento, o valor de f será mínimo. A prova deste fato também será similar àquela do algoritmo de Ford-Fulkerson.

Para a rede G, definiremos a capacidade $\text{cap}(R)$ de um s-t corte R como

$$\text{cap}(R) = \sum_{e \in E^-(R)} l(e) - \sum_{e \in E^+(R)} u(e).$$

Proposição 6.4. *Seja f um fluxo viável em G e R um s-t corte. Então, $|f| \geq \text{cap}(R)$.*

Demonstração. Se f é um fluxo viável, então

$$\begin{aligned}
|f| &= f^-(R) - f^+(R) && \text{(Proposição 2.2)} \\
&= \sum_{e \in E^-(R)} f(e) - \sum_{e \in E^+(R)} f(e) \\
&\geq \sum_{e \in E^-(R)} l(e) - \sum_{e \in E^+(R)} u(e) && (f \text{ é viável}) \\
&= \text{cap}(R)
\end{aligned}$$

□

A definição de caminho de aumento aqui será ligeiramente diferente. No contexto de circulações viáveis, um **caminho de aumento** é um caminho não dirigido tal que para toda aresta e, temos $f(e) < u(e)$ se e for direta ou $f(e) > l(e)$ se e for inversa. Esta definição mantém a ideia de que o fluxo pode ser aumentado ao longo do caminho.

Proposição 6.5. *Seja f um fluxo viável em G e suponha que não haja caminho de aumento de t a s. Então existe um s-t corte R tal que* $\text{cap}(R) = |f|$.

Demonstração. Seja R^c o conjunto dos vértices alcançáveis a partir de t por caminhos de aumento. Note que $s \notin R^c$ e $t \in R^c$, então o complementar de R^c é um s-t corte.

Se $e = (v, w)$ é uma aresta que sai de R^c, então $f(e) = u(e)$, caso contrário w estaria em R^c. Da mesma forma, se e é uma aresta que entra em R^c, então $f(e) = l(e)$. Portanto,

tomando R como o complementar de R^c, temos que R é um s-t corte e

$$\begin{aligned}\text{cap}(R) &= \sum_{e \in E^-(R)} l(e) - \sum_{e \in E^+(R)} u(e) \\ &= \sum_{e \in E^-(R)} f(e) - \sum_{e \in E^+(R)} f(e) \\ &= f^-(R) - f^+(R) \\ &= |f|, \qquad \text{(Proposição 2.2)}\end{aligned}$$

como queríamos. □

Da proposição anterior segue, imediatamente, o seguinte corolário:

Corolário 6.1. *Se não existe caminho de aumento de t a s, então f é mínimo.*

A segunda parte do problema pode ser resolvida com qualquer algoritmo de fluxo máximo baseado em caminhos de aumento, como o algoritmo de Edmonds-Karp ou o algoritmo de Dinic. Basta adaptar a rede residual para comportar ambos os limites inferiores e superiores das arestas, ou seja, para cada aresta $e = (v, w)$ de E, a rede residual terá uma aresta $e_d = (v, w)$ de capacidade $u(e) - f(e)$ e uma aresta $e_r = (w, v)$ de capacidade $f(e) - l(e)$.

6.6 Coberturas mínimas de grafos acíclicos por caminhos

Seja G um grafo. Uma **cobertura por caminhos** de G é um conjunto $\mathcal{P}$ de caminhos em G tal que todo vértice pertence a pelo menos um caminho em $\mathcal{P}$.

Nesta seção, estudaremos o seguinte problema:

Problema (Cobertura por caminhos mínima). *Dado um grafo acíclico $G = (V, E)$, encontrar uma cobertura por caminhos de G de cardinalidade mínima.*

Estudaremos duas variações deste problema. A primeira exige que os caminhos da cobertura sejam disjuntos dois a dois. A segunda variação não apresenta tal restrição.

Cobertura por caminhos disjuntos

Ao exigirmos que os caminhos da cobertura sejam disjuntos, temos que para todo vértice v do grafo, se C é o conjunto das arestas que pertencem à cobertura, então $|C^+(v)| \leq 1$ e $|C^-(v)| \leq 1$. A volta também é verdadeira: se C é um conjunto de arestas tal que $|C^+(v)| \leq 1$ e $|C^-(v)| \leq 1$, então C é uma cobertura por caminhos. Os caminhos que compõem esta cobertura podem ser encontrados partindo-se de um vértice com $|C^+(v)| = 0$ e seguindo a única aresta de C que sai de v até chegar num vértice w com $|C^-(w)| = 0$. O número de caminhos nesta cobertura é $|V| - |C|$, que coincide com a quantidade de vértices que são origem de algum caminho da cobertura.

Isto sugere a seguinte ideia para resolver o problema: tentaremos encontrar o conjunto C de cardinalidade máxima tal que $|C^+(v)| \leq 1$ e $|C^-(v)| \leq 1$ para todo vértice v.

Como podemos encontrar tal conjunto? Perceba que toda aresta do grafo sai de um vértice e chega em outro. Assim, podemos pensar que um conjunto C que satisfaz as condições mencionadas anteriormente corresponde a um emparelhamento das "saídas" de cada vértice com as "entradas" de cada vértice.

Mais formalmente, considere os conjuntos V_{in} e V_{out} que são cópias de V, ou seja, para cada vértice v em V temos um vértice v_{in} em V_{in} e um vértice v_{out} em V_{out}. Sejam

$$V' = V_{\text{in}} \cup V_{\text{out}} \quad \text{e} \quad E' = \{(v_{\text{out}}, w_{\text{in}}) : (v, w) \in E\}.$$

Então, é fácil notar que C satisfaz $|C^+(v)| \leq 1$ e $|C^-(v)| \leq 1$ para todo $v \in V$ se e somente se C induz um emparelhamento no grafo bipartido $G' = (V', E')$.

Portanto, para encontrar a cardinalidade da cobertura por caminhos mínima de G basta calcularmos $|V| - |M|$, sendo M um emparelhamento máximo em G'. Para determinar os caminhos de uma cobertura mínima, tomamos o conjunto C de arestas de G induzido por M e construímos cada caminho conforme explicado anteriormente.

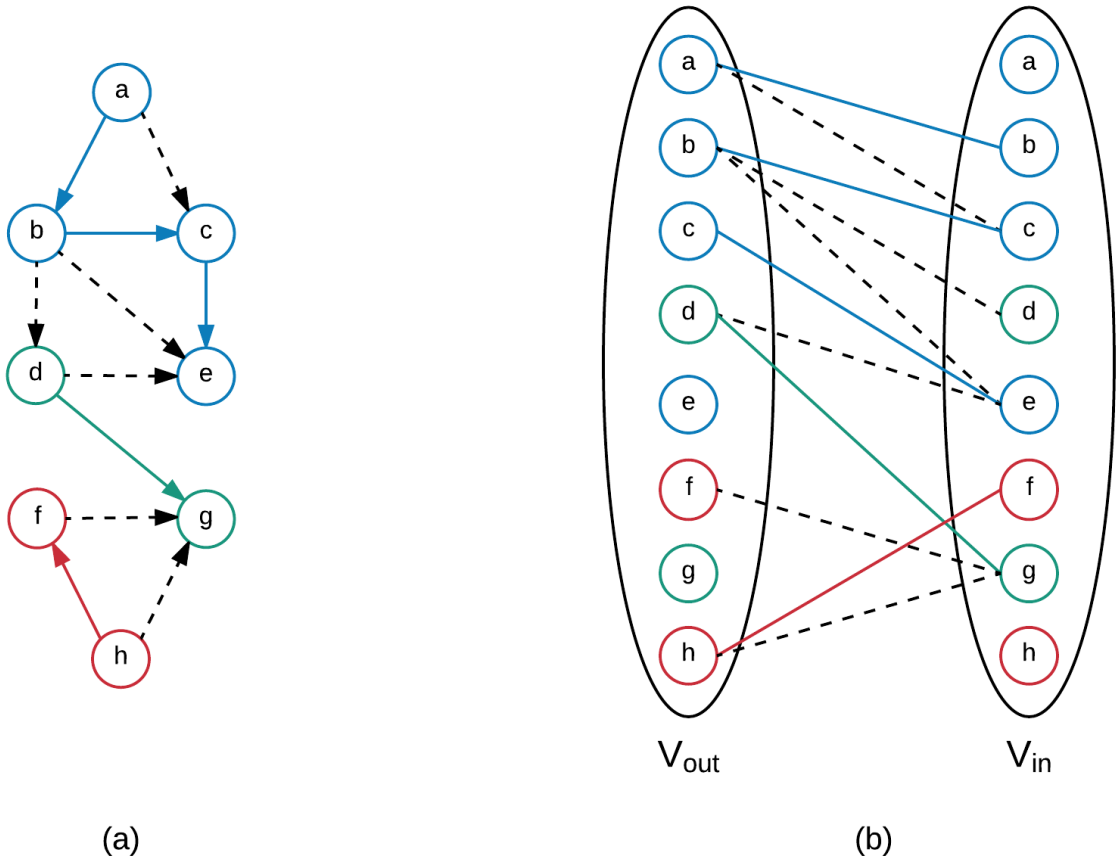

Figura 6.4: A figura (a) mostra um grafo G e uma cobertura por caminhos mínima $\mathcal{P} = \{(a, b, c, e), (d, g), (h, f)\}$ indicada por arestas de linha cheia. A figura (b) mostra o grafo G' com o respectivo emparelhamento máximo M, indicado também por linhas cheias. Note que $|V| = 8, |M| = 5$ e $|\mathcal{P}| = |V| - |M| = 3$.

Cobertura por caminhos não necessariamente disjuntos

Para resolver o problema sem a restrição dos caminhos da cobertura serem disjuntos, temos que usar uma abordagem um pouco diferente.

Considere a adição de dois novos vértices s e t e arestas (s, v) e (v, t) para todo vértice v no grafo G. Note que podemos assumir agora que todos os caminhos da cobertura têm origem s e destino t.

Seja $\mathcal{P}$ uma cobertura por caminhos de G. Considere uma função-capacidade u que atribui capacidade infinita para todas as arestas do grafo G, e uma função $f : E \to \mathbb{Q}^+$ tal que para toda aresta e, $f(e)$ é a quantidade de caminhos em $\mathcal{P}$ que passam por e. É claro que f é um fluxo de s a t. Além disso, o valor de f é igual a quantidade de caminhos em $\mathcal{P}$.

Não é verdade, porém, que todo fluxo f de s a t corresponde a uma cobertura por caminhos de G. No entanto, se f é inteiro e o grafo é acíclico, sempre é possível decompor f em caminhos de s a t, conforme a proposição a seguir.

Proposição 6.6. *Seja (V, E) um grafo acíclico com dois vértices s e t e seja f um s-t fluxo inteiro numa rede capacitada (V, E, u). Então, existe uma coleção $\mathcal{P}$ de $|f|$ caminhos de s a t tal que para toda aresta $e \in E$, $f(e)$ é a quantidade de caminhos em $\mathcal{P}$ que passam por e.*

Chamaremos tal coleção de **decomposição** de f em caminhos.

Demonstração. Provaremos por indução em $|f|$.

Suponha $|f| = 1$. Considere um caminho P que tem origem em s e segue por arestas de fluxo unitário até não ser mais possível. Como o grafo é acíclico, este caminho é finito e não termina em s. Note que o caminho também não pode terminar em um vértice $v \neq t$, pois se o caminho chega em v por uma aresta com fluxo unitário, como $f^+(v) - f^-(v) = 0$, então deve existir uma aresta que sai de v e também tem fluxo unitário.

Portanto, P é um caminho de s a t. Considere agora a função f' que é igual a f exceto nas arestas que pertencem a P, nas quais o valor do fluxo diminui em uma unidade. Claramente f' é um fluxo de s a t. Mostraremos que f' é o fluxo nulo.

É claro que $|f'| = 0$. Portanto, $f^+(v) - f^-(v) = 0$, para todo vértice v. Suponha que $f'(e) > 0$ para alguma aresta (v, w). Construa um caminho P' com origem v que segue por arestas de fluxo unitário até não ser mais possível. Como $f'^+(v) - f'^-(v) = 0$ para todo vértice, o caminho P' só pode terminar em v. Isto contradiz a hipótese do grafo ser acíclico.

Logo, $f' = 0$, e $\mathcal{P} = \{P\}$ é uma coleção de caminhos de s a t que satisfaz as condições do enunciado.

Suponha agora que $|f| > 1$. Construa um caminho P com origem s da mesma forma como foi feito no caso anterior. Pelos mesmos argumentos, este caminho termina em t. Tomando f' como a função igual a f exceto nas arestas de P, nas quais o valor do fluxo diminui em uma unidade, temos que f' é um fluxo de s a t de valor $|f| - 1$. Pela hipótese de indução, podemos decompor f' numa coleção $\mathcal{P}'$ de $|f| - 1$ caminhos de s a t. Assim, a coleção $\mathcal{P}' \cup \{P\}$ é uma decomposição de f em $|f|$ caminhos. □

Nosso problema, então, se reduz a encontrar um fluxo de valor mínimo tal que para todo vértice v diferente de s e t, temos $f^+(v) > 0$ (e, consequentemente, $f^-(v) > 0$). Isto garante que a decomposição de f produz caminhos que cobrem todos os vértices. Chamaremos um fluxo que satisfaz esta restrição de *bom*. O problema de encontrar um fluxo bom é bastante parecido ao problema do fluxo mínimo (seção 6.5), porém com limites inferiores nos vértices, e não nas arestas. Isto pode ser contornado com a mesma transformação usada para resolver o problema do fluxo máximo com capacidades nos vértices (seção 6.2).

Considere a rede (V', E', l', u'), sendo (V', E') o grafo resultante da transformação mencionada, $u'(e) = \infty$ para toda aresta e e $l'(e) = 0$ para toda aresta e exceto para arestas do tipo (v_i, v_o), cujo valor é $l'(e) = 1$. Os seguintes fatos podem ser verificados da mesma forma como foi feito na seção 6.3:

- Todo fluxo bom de s a t na rede original corresponde a um fluxo viável em (V', E', l', u') de mesmo valor;

- Todo fluxo viável em (V', E', l', u') corresponde a um fluxo bom na rede original de mesmo valor.

Portanto, para resolver o problema da cobertura por caminhos mínima do grafo (V, E), seguimos os seguinte passos:

1. Construa a rede (V', E', l', u') como descrito anteriormente e encontre o fluxo mínimo f' de s_i a t_o nesta rede.

2. Encontre o fluxo f correspondente a f' na rede (V, E, u). O fluxo f será um fluxo bom de valor mínimo.

3. Decomponha f em caminhos. Esta decomposição será uma cobertura mínima de (V, E) ao removermos os vértices s e t dos caminhos.

É claro que, se estamos interessados somente na quantidade de caminhos em uma cobertura mínima, basta tomar o valor do fluxo obtido no passo 1.

Capítulo 7

Exemplos de problemas

Neste capítulo apresentamos vários exemplos de problemas de programação competitiva cuja solução envolve o uso de algoritmos de fluxo máximo. Os primeiros problemas da lista são aplicações diretas dos algoritmos vistos, e para estes apresentamos somente um breve comentário sobre o que o problema se trata e algumas informações sobre o juiz, quando relevante. O leitor pode usar estes problemas para verificar seu entendimento sobre o assunto e se suas implementações dos algoritmos estão corretas. Os demais problemas foram escolhidos visando diversificar a forma como o problema é modelado para a aplicação dos algoritmos de fluxo. Para estes, apresentamos um rascunho de solução. Estes problemas não estão listados em ordem de dificuldade.

Todos os problemas possuem um link para um juiz online com o respectivo problema, para que o leitor possa submeter e verificar sua própria solução. Além disso, uma implementação em C++ da solução de cada problema está disponível em https://linux.ime.usp.br/~marcosk/mac0499/codigos.html.

7.1 Total flow

URL: http://www.spoj.com/problems/MTOTALF/

Comentários

O problema se trata da aplicação direta do algoritmo de fluxo máximo em redes não-dirigidas. Como a rede é pequena, qualquer um dos algoritmos vistos resolvem o problema dentro do tempo limite. Assim, o problema pode ser usado para verificar a implementação de um algoritmo de fluxo máximo.

7.2 Fast flow

URL: http://www.spoj.com/problems/FASTFLOW/

Comentários

Este problema também se trata da aplicação direta do algoritmo de fluxo máximo, porém a rede é muito maior comparada ao problema anterior. Assim como o nome sugere, este problema requer uma implementação mais eficiente para não exceder o limite de tempo. A solução disponibilizada usa o algoritmo Push-Relabel com critério de seleção da maior

altura. No entanto, uma solução com uma boa implementação do algoritmo de Dinic também consegue executar dentro do tempo limite.

É necessário tomar cuidado para fazer as modificações necessárias nos códigos para acomodar fluxos cujo valor excedem o valor máximo de uma variável inteira de 32 bits.

7.3 Air raid

URL: https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=697

Comentários

O problema basicamente pede para determinar o número mínimo de caminhos em um grafo acíclico necessário para que cada vértice do grafo pertença a exatamente um destes caminhos, e pode ser resolvido aplicando diretamente a solução apresentada na seção 6.6 para o caso de caminhos disjuntos.

7.4 Viagens no tempo

URL: http://br.spoj.com/problems/EINSTEIN/

Comentários

O problema se trata da aplicação direta do algoritmo de Stoer-Wagner para a determinação do corte mínimo global. Note que, como o grafo pode ser denso, é recomendável a implementação do algoritmo em $O(|V|^3)$ ao invés de $O(|E||V|^2 \log |V|)$. De fato, algumas implementações do algoritmo de Stoer-Wagner com fila de prioridade podem não executar dentro do limite de tempo do juiz.

Cuidado com o formato da entrada. O enunciado omite, mas a entrada começa com um inteiro informando a quantidade de instâncias. Além disso, podem haver arestas repetidas na entrada. Neste caso, considere somente o custo da última aresta da entrada para cada par de vértices (v, w), e ignore todas as outras arestas anteriores que conectam o mesmo par de vértices.

7.5 Dungeon of death

URL: http://www.spoj.com/problems/QUEST4/

Resumo

Em um tabuleiro 120×120, temos N casas que precisam ser cobertas. A cobertura deve ser feita utilizando tábuas de dimensões 1×120, que podem ser posicionadas tanto na vertical quanto na horizontal. Tábuas podem ser sobrepostas. Determinar o número mínimo de tábuas necessário para cobrir todas as N casas.

Solução

Em problemas envolvendo tabuleiros é sempre possível abstrair dois grafos bipartidos: Um que biparticiona as casas pela sua cor (se pintarmos o tabuleiro como um tabuleiro de xadrez), e outro que biparticiona as linhas e colunas do tabuleiro. Neste problema, usaremos a segunda abstração.

As arestas do grafo serão as casas a serem cobertas: uma casa de coordenada (x, y) corresponderá a uma aresta ligando a coluna x com a linha y. Repare que o problema passa a ser equivalente ao problema de cobertura mínima na rede construída. Como o problema só pede a cardinalidade da solução, basta calcularmos o tamanho de um emparelhamento máximo.

Análise. O problema do emparelhamento máximo em grafos bipartidos pode ser resolvido em tempo $O(|E|\sqrt{|V|})$ usando o algoritmo de Hopcroft-Karp. Porém, como os limites das variáveis de entrada do problema são pequenos, qualquer um dos algoritmos de fluxo máximo vistos executa dentro do limite de tempo. Isto porque o algoritmo de Ford-Fulkerson tem complexidade $O(|E||V|)$ quando aplicado ao problema do emparelhamento máximo bipartido, pois o fluxo máximo é limitado por $|V|$, e cada caminho de aumento é encontrado em tempo $O(|E|)$. A solução disponibilizada usa o algoritmo de Edmonds-Karp.

7.6 No cheating

URL: https://code.google.com/codejam/contest/32002/dashboard#s=p2

Resumo

Queremos aplicar uma prova em uma sala. A sala é um retângulo formado por M linhas e N colunas. Cada casa deste retângulo corresponde a uma carteira.

Um aluno é capaz de olhar as respostas dos alunos sentados imediatamente à sua direita, sua esquerda, sua direita à frente e sua esquerda à frente. Queremos determinar o número máximo de alunos que podem fazer a prova nesta sala de modo que nenhum aluno possa colar, sendo que algumas carteiras da sala não podem ser usadas.

Solução

Considere o grafo cujos vértices são as carteiras da sala, e dois vértices são conectados por uma aresta se um aluno sentado em uma das carteiras é capaz de colar de um aluno sentado na outra carteira. O problema se reduz, então, a encontrar um conjunto de vértices de tamanho máximo neste grafo de modo que não haja uma aresta entre dois vértices deste conjunto. Um conjunto com esta propriedade é chamado de **conjunto independente**.

A observação chave para resolver o problema é a seguinte. Seja I um conjunto independente. Então, toda aresta do grafo incide em algum vértice que está fora de I. Em outras palavras, o complementar de um conjunto independente é uma cobertura, e vice-versa. Por consequência, todo conjunto independente máximo é o complementar de uma cobertura mínima.

Infelizmente, só vimos como resolver o problema da cobertura mínima em grafos bipartidos. Por sorte, o grafo do problema é bipartido! De fato, cada vértice só está conectado a vértices de fileiras adjacentes. Portanto, o conjunto dos vértices em fileiras ímpares e dos vértices em fileiras pares formam uma bipartição. Assim, a resposta de uma instância do pro-

blema pode ser obtida calculando a diferença entre o total de vértices do grafo e o tamanho de uma cobertura mínima.

Análise. No *Google Code Jam*, de onde este problema foi retirado, a saída para o caso de teste é gerada pelo próprio competidor em sua máquina. Assim, não é necessário que a solução seja extremamente eficiente, basta que compute a resposta de todas as instâncias dentro de alguns minutos. Neste problema, como os valores de N e M são pequenos, o algoritmo de Edmonds-Karp é suficiente. Com este algoritmo, o problema é resolvido em tempo $O((NM)^2)$ por caso de teste.

7.7 Intelligence quotient

URL: https://szkopul.edu.pl/problemset/problem/7Ty-tp-ihfAOSrBJIPbVwR-z/site/?key=statement

Resumo

Temos um conjunto de $N + M$ alunos, dos quais N estudam matemática e M estudam ciência da computação. Nenhum aluno estuda ambas as matérias.

Todos os alunos que estudam matemática se conhecem. Similarmente, todos os alunos de computação se conhecem. Além disso, são dados mais K pares de alunos, um da matemática e outro da computação, indicando que ambos se conhecem.

Cada aluno possui um QI, um valor inteiro positivo. Queremos determinar um grupo de alunos em que todos se conhecem e que a soma de seus QIs seja a maior possível.

Solução

Problemas deste estilo podem ser atacados de duas maneiras: ou maximizamos o valor do conjunto resposta, ou minimizamos o valor de seu complementar. Neste problema, usaremos a segunda abordagem. Ou seja, partiremos de um conjunto contendo todos os alunos e procuraremos o conjunto cuja soma dos QIs seja mínima e cuja remoção resulte num conjunto em que todos se conheçam.

Como estamos agora resolvendo um problema de minimização, é natural tentarmos modelar o problema como um problema de corte mínimo. Sendo assim, considere uma rede G cujos vértices são os alunos. Seja A o conjunto dos alunos que estudam matemática e B o conjunto dos que estudam computação. Criamos uma aresta de um aluno de A para um aluno de B se ambos não se conhecem.

Vamos adicionar à rede um vértice s com arestas chegando em todos os alunos de A e um vértice t com arestas saindo de todos os alunos de B.

Esta rede possui uma propriedade intessante: Considere um conjunto S de alunos e seja G' a rede resultante da remoção de todos os vértices que não estão em S. Então, existe caminho de s a t em G' se e somente um par de alunos de S não se conhecem.

É fácil verificar a propriedade. Sejam $u \in A$ e $v \in B$ dois alunos de S. Então u e v não se conhecem $\Leftrightarrow$ existe aresta de u para v em $G' \Leftrightarrow$ existe caminho de s a t em G', pois s está ligado a u e v está ligado a t.

Note que, se u pertence a A, remover u de G é equivalente a remover a aresta (s, u) no contexto de manter t alcançável a partir de s. Similarmente, se v pertence a B, remover v é equivalente a remover a aresta (v, t).

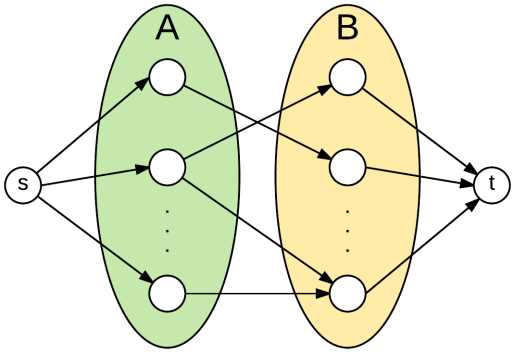

Figura 7.1: A rede G.

Ao removermos um aluno da rede, a soma dos QIs se reduz pelo QI deste aluno. Assim, é natural atribuirmos as arestas que ligam cada aluno a s ou t uma capacidade igual ao valor do QI deste aluno.

Estamos quase prontos para aplicar o algoritmo de corte mínimo. O que falta fazer é evitar que o algoritmo decida cortar uma aresta entre dois alunos, pois caso haja dois alunos em S que não se conheçam, queremos remover ou um ou o outro, e não a restrição que impede ambos de estarem no mesmo conjunto. Isto pode ser feito atribuindo a cada uma dessas arestas uma capacidade infinita.

Modelando a rede G desta maneira, o corte mínimo de G que separa s de t indicará quais os alunos devemos remover de forma que a soma dos QIs dos alunos restantes seja máxima e que todos eles se conheçam.

Análise. A rede G tem $O(NM)$ arestas e $O(N + M)$ vértices. Como cada caminho de aumento satura ou uma aresta de $E^-(s)$ ou uma aresta de $E^+(t)$, o algoritmo de Edmonds-Karp termina buscando por no máximo $O(N + M)$ caminhos de aumento, resultando numa complexidade de tempo $O(NM(N + M))$, suficiente para o programa executar dentro do tempo limite para os casos de teste do problema.

7.8 Containment

URL: https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=668&page=show_problem&problem=5114

Resumo

Queremos isolar as células defeituosas de um grid $10 \times 10 \times 10$. O isolamento é feito colocando placas que cobrem uma face de uma célula. Todas as células defeituosas devem estar no interior de algum politopo formado pelas placas. Queremos determinar o menor número de placas necessário para realizar a tarefa.

Solução

Considere um grafo não-direcionado cujos vértices são as células e dois vértices são conectados por uma aresta se as células correspondentes possuem uma face em comum. Adicione um vértice s que representará o exterior do grid. Para cada vértice, adicione uma aresta entre este vértice e s para cada face exposta que a célula correspondente possui.

O problema se reduz a remover o menor número de arestas deste grafo de modo que não haja caminho do vértice exterior até o vértice de alguma célula defeituosa.

Se precisássemos isolar somente uma única célula, a resposta seria dada pelo corte mínimo que separa este vértice do vértice exterior na rede onde as capacidades são todas unitárias. Para adaptar este problema para o caso em que mais de uma célula precisa ser isolada, basta adicionarmos um novo vértice t à rede e ligarmos este vértice a todos os vértices de células defeituosas com uma aresta de capacidade infinita. Isto força todos as células defeituosas estarem do mesmo lado de t em qualquer s-t corte mínimo.

Análise. A rede possui $|V| = 10^3$ vértices, e o número de arestas é $O(|V|)$. Como o fluxo máximo é limitado pela quantidade de células na superfície do grid, qualquer um dos algoritmos de fluxo máximo vistos resolve o problema dentro do tempo limite. O algoritmo de Edmonds-Karp, por exemplo, tem complexidade $O(|E||V|)$ neste problema.

7.9 Travessia

URL: https://www.urionlinejudge.com.br/judge/pt/problems/view/1502

Resumo

Queremos atravessar um corredor muito longo de largura W. Neste corredor há N postes com uma energia P_i. Cada poste i produz um quadrado de lado igual a $2P_i^2$ centrado neste poste. Como não podemos caminhar no interior de quadrados, pode não ser possível realizar a travessia.

Para tornar a travessia possível, podemos alterar (tanto aumentar quanto diminuir) os valores das energias de cada poste. Diremos que uma configuração de postes é *viável* se permite a travessia. Definiremos a *energia total* de uma configuração como a soma das energias de cada poste nesta configuração. Queremos, então, determinar o menor valor absoluto possível da diferença entre a energia total inicial e a energia total de uma configuração viável.

Solução

À primeira vista, a solução deste problema parece envolver muita geometria e a ideia de redes parece ser pouco relacionada com este problema. O que faz deste problema bastante interessante é que a solução esperada é exatamente o contrário: faremos fortemente o uso de redes, e a única coisa geométrica que usaremos será verificar se dois quadrados se intersectam.

A primeira coisa que devemos pensar sobre este problema é como determinar se uma dada configuração é viável. Uma forma simples de decidir isto é verificar se existe uma sequência de quadrados $(Q_1, \ldots, Q_k)$ tais todo par de quadrados adjacentes na sequência se intersectam, Q_1 intersecta a parede da esquerda e Q_k intersecta a parede da direita. Em outras palavras, temos que verificar se as duas paredes estão *conectadas* por uma sequência de quadrados.

Quando pensamos em conexidade, é natural pensarmos em grafos. Se considerarmos o grafo não-dirigido cujos vértices são os quadrados e as paredes, e dois vértices são conectados por uma aresta se as figuras correspondentes se intersectam, temos que a travessia é possível se e somente se as duas paredes estão na mesma componente conexa do grafo.

Resolvemos uma parte do problema, mas ainda não sabemos como tratar das várias configurações possíveis de energia. A quantidade de tais configurações é grande demais para verificarmos cada uma individualmente. Precisamos ser mais espertos.

Uma observação importante é que, se uma configuração de energia total x é viável, então para todo inteiro $0 \leq y \leq x$, existe uma configuração viável de energia total y. Isto é claro, pois podemos tomar a configuração de energial total x e reduzir as energias dos postes uma unidade por vez, mantendo a viabilidade da configuração.

Logo, para determinar a resposta do problema, basta encontrarmos uma configuração viável de energia total máxima, pois se a energia total inicial for x e a energia total máxima for y, a resposta será o valor mínimo de $|x - z|$ restrito a z inteiro e $0 \leq z \leq y$.

Para encontrar uma configuração viável de energia total máxima, iremos começar com uma configuração em que todos os postes têm um grande valor de energia e tentaremos reduzi-las o mínimo possível de forma a tornar a configuração viável. É claro que em qualquer configuração viável não existe um poste cujo quadrado intersecta ambas as paredes. Logo, podemos tomar como energia de cada poste o maior valor tal que isto não acontece.

Outra observação importante é que podemos modelar o problema de uma maneira ligeiramente diferente: ao invés de considerar que um poste com energia P_i produz um único quadrado de lado $2P_i^2$, iremos considerar que este poste produz P_i quadrados de lados $2 \cdot 1^2, 2 \cdot 2^2, \ldots, 2 \cdot P_i^2$. Isto claramente não altera a viabilidade de uma configuração, e traz uma grande vantagem: o problema de determinar a configuração viável de energia total máxima torna-se um problema de fluxo máximo!

De fato, partindo da configuração em que cada poste tem energia máxima, o que queremos neste novo modelo é remover o número mínimo possível de quadrados de forma tornar a travessia possível. Pensando no grafo que construimos anteriormente, queremos encontrar um conjunto de vértices de cardinalidade mínima cuja remoção desconecta as duas paredes. Este problema pode ser resolvido encontrando-se o fluxo máximo entre as duas paredes na rede em que a capacidade de cada quadrado é 1.

Análise. O problema se resume ao cálculo do fluxo máximo em uma rede. Como temos N postes e cada um gera $O(\sqrt{W})$ quadrados, a quantidade de vértices nesta rede é $V = O(N\sqrt{W})$. O número de arestas nesta rede é limitado por V^2, então $E = O(N^2W)$. Portanto, se usarmos o algoritmo de Dinic (seção 2.4), teríamos, a princípio, uma complexidade total $O(EV^2) = O(N^4W^2)$, que parece muito alta. Porém, como as capacidades dos vértices são todas unitárias, o fluxo em cada fase do algoritmo é encontrado em tempo $O(E)$ ao invés de $O(EV)$, reduzindo a complexidade para $O(EV) = O(N^3W^{3/2})$, suficiente para a solução executar dentro do tempo limite.

7.10 Pumping stations

URL: http://codeforces.com/contest/343/problem/E

Resumo

O problema se resume a encontrar um caminho hamiltoniano de custo máximo (um caminho que passa por todos os vértices de uma rede exatamente uma vez e tal que a soma dos custos das arestas deste caminho é máxima) em uma rede na qual o custo de uma aresta (v, w) é igual ao fluxo máximo de v a w em uma outra rede não-dirigida dada.

Solução

Seja G a rede dada, e G' a rede na qual procuramos um caminho hamiltoniano de custo máximo. Assim, o custo de qualquer aresta (v, w) de G' é igual ao fluxo máximo de v a w em G.

Considere uma árvore de Gomory-Hu (veja o capítulo 4) de G. Provaremos, por indução no número de vértices, que o custo máximo de um caminho hamiltoniano em G' é igual à soma das capacidades das arestas da árvore.

Seja $P = (v_1, v_2, \ldots, v_n)$ um caminho hamiltoniano de custo máximo. Seja (r, s) uma aresta de capacidade mínima da árvore de Gomory-Hu, e seja (R, S) o corte induzido por esta aresta. Assuma, sem perda de generalidade, que v_n está em S.

Sejam $(r_1, r_2, \ldots, r_k)$ e $(s_1, s_2, \ldots, s_l)$ as subsequências do caminho hamiltoniano que consistem somente de vértices de R e S, respectivamente. Afirmamos que

$$P' = (r_1, r_2, \ldots, r_k, s_1, s_2, \ldots, s_l)$$

é um caminho hamiltoniano de custo máximo.

De fato, seja v um vértice da rede diferente de v_n. Seja w o vértice que sucede v em P, e w' o vértice que sucede v em P'. Temos três casos:

- $v \in R$, $w \in R$ e $w' \in R$: Neste caso, $w = w'$, portanto as arestas (v, w) e (v, w') têm o mesmo custo em G'.
- $v \in R$, $w \in S$ e $w' \in R$: Neste caso, como (r, s) tem capacidade mínima e a árvore é de Gomory-Hu, temos que o custo da aresta (v, w') é maior ou igual ao custo da aresta (v, w) em G'.
- $v \in S$: Neste caso, também temos $w = w'$, logo o custo da aresta (v, w) é igual ao custo da aresta (v, w') em G'.

Assim, o custo de P' é maior ou igual ao custo de P. Como P é máximo, temos que P' também é. Como P' é a concatenação de vértices de R e vértices de S, com $R \cap S = \varnothing$, temos que o custo de P' é igual à soma de três parcelas: o custo de um caminho hamiltoniano máximo no grafo induzido por R, o custo de um caminho hamiltoniano máximo no grafo induzido por S, e a capacidade da aresta (r, s). As árvores de Gomory-Hu destes dois grafos são as sub-árvores resultantes da remoção da aresta (r, s) da árvore original. Aplicando a hipótese de indução em ambos os grafos a afirmação segue.

A demonstração acima também nos dá um algoritmo para encontrar um caminho hamiltoniano máximo. Basta encontrar uma aresta de capacidade mínima na árvore de Gomory-Hu, resolver o problema recursivamente para ambas as sub-árvores, e concatenar os caminhos encontrados.

Análise. A árvore de Gomory-Hu pode ser encontrada em $O(|V|(|E| + \tau))$, sendo τ o tempo necessário para calcular o fluxo máximo entre dois vértices. Após computar a árvore, o caminho hamiltoniano máximo pode ser encontrado em tempo $O(|V|^2)$, pois basta encontrar a aresta de capacidade mínima na árvore e resolver o problema recursivamente nas duas sub-árvores. Assim, o gargalo da complexidade está no cálculo da árvore de Gomory-Hu. A solução disponibilizada implementa o algoritmo Push-Relabel, mas como as capacidades da rede do problema são pequenas, uma solução que implementa Edmonds-Karp também executa dentro do tempo limite.

Parte subjetiva

Apreciação pessoal e crítica

Criar um material didático sobre um tema tão extenso foi um grande desafio na elaboração deste trabalho. Juntar o conteúdo de várias fontes diferentes e resumir em um único texto mostrou-se uma tarefa muito mais difícil que eu esperava, principalmente ao "traduzir" demonstrações de artigos numa linguagem mais didática. Além disso, eu não tinha ciência da quantidade de conteúdo que eu queria abordar até escrever as primeiras dezenas de páginas da monografia. Ocorreu que foi necessário deixar de cobrir temas antes presentes no planejamento, como fluxos de custo mínimo e experimentos com os tempos de execução dos algoritmos, a fim de evitar um trabalho excessivamente longo.

Por outro lado, realizar este trabalho foi uma experiência academicamente enriquecedora. Sendo um veterano de competições de programação, eu já estava familiarizado com muitos dos temas abordados, no entanto de forma bastante informal. Por este motivo, uma das coisas que me surpreendeu foi o quão diversificado foram as técnicas de prova empregadas ao longo do texto. Foram demonstrações por algoritmos, por construções, por funções potenciais, por "a primeira metade fazemos de um jeito e a segunda de outro", enfim. A exposição a estes diversos métodos de demonstração certamente contribuirá para a minha vida acadêmica.

Por fim, concluo o trabalho com grande satisfação, na esperança que ele seja de grande ajuda para participantes da Maratona de Programação interessados em conhecer e se aprofundar no tema de fluxos em redes.

Agradecimentos

Agradeço, em primeiro lugar, ao meu orientador Coelho, pela oportunidade em trabalhar neste tema pelo qual tenho bastante interesse, e por ter me acompanhado pacientemente durante todo este ano, sempre interessado em discutir os assuntos abordados no trabalho e dar sugestões.

Agradeço a meu ex-companheiro de equipe Stefano Tommasini, por me ajudar com o tema de fluxos quando eu estava aprendendo sobre o assunto e me apresentar uma lista ampla de problemas, alguns dos quais estão presentes neste trabalho.

Agradeço também aos membros do grupo de extensão MaratonIME, que foram uma das motivações para a elaboração deste texto. Em especial, agradeço a Gabriel Russo e Matheus Oliveira por ajudarem na revisão da monografia, e a Renzo Gómez pela indicação de problemas de juízes online.

Referências Bibliográficas

Blum e Gupta(2013) Avrim Blum e Anupam Gupta. Design & analysis of algorithms, notas de aula. https://www.cs.cmu.edu/~avrim/451f13/lectures/lect1010.pdf, 2013.

Cormen *et al.***(2009)** Thomas Cormen, Charles Leiserson, Ronald Rivest e Clifford Stein. *Introduction to Algorithms*. The MIT Press, terceira edição.

Feofiloff(2017) Paulo Feofiloff. Otimização combinatória, notas de aula. https://www.ime.usp.br/~pf/otimizacao-combinatoria/mynotes/OtimizacaoCombinatoria.pdf, 2017.

Gomory e Hu(1961) R. E. Gomory e T. C. Hu. Multi-terminal network flows. *SIAM J. Appl. Math.*, 9(4):551–570.

Gupta e Sleator(2014) Anupam Gupta e Danny Sleator. Dinic's algorithm, notas de aula. http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15451-f14/www/lectures/lec11/dinic.pdf, 2014.

Gusfield(1990) Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.*, 19(1):143–155.

Kleinberg e Tardos(2006) Jon Kleinberg e Éva Tardos. *Algorithm Design*. Addison Wesley, primeira edição.

Mayr e Räcke(2016) Erns Mayr e Harald Räcke. Gomory hu trees, slides de aula. http://www14.in.tum.de/lehre/2016WS/ea/split/sec-Gomory-Hu-Trees.pdf, 2016.

Mestre(2009) Julián Mestre. Optimization II, notas de aula. https://resources.mpi-inf.mpg.de/departments/d1/teaching/ws09_10/Opt2/handouts/lecture4.pdf, 2009.

Stoer e Wagner(1997) Mechthild Stoer e Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591.

Zwick(1993) Uri Zwick. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148(1):165–170.

Índice Remissivo