Métodos e Técnicas para Resolução do Problema de Fluxo Máximo em Redes

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Patrícia Helaine Landim Nascimento (paty@lia.ufc.br) e aprovada pela Banca Examinadora.

Fortaleza, 2 de dezembro de 2002.

Ricardo Cordeiro Corrêa (DC/UFC)

Dissertação apresentada ao Mestrado em Ciência da Computação, UFC, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Mestrado em Ciência da Computação

Universidade Federal do Ceará

Métodos e Técnicas para Resolução do Problema de Fluxo Máximo em Redes

Patrícia Helaine Landim Nascimento

(paty@lia.ufc.br)[1]

agosto de 2002

Banca Examinadora:

- Ricardo Cordeiro Corrêa (DC/UFC)
- Luiz Satoru Ochi (DCC/UFF)
- Manoel Bezerra Campêlo Neto (DEMA/UFC)

[1]**Pesquisa apoiada pela FUNCAP — Fundação Cearense de Amparo à Pesquisa**

Agradecimentos

Agradeço à Deus, sobre todas as coisas, por tudo.

Agradeço aos meus pais e meus irmãos por todo amor e apoio. Ao meu querido Francisco Torres por seu amor e conselhos, por ter aparecido na hora certa e me dar forças para não desistir.

Agradeço também ao meu amigo Yuri Abitbol por toda a ajuda, dicas, brincadeiras, por não deixar eu desanimar e acreditar em mim. E a todos os meus amigos e parentes que direta ou indiretamente me ajudaram, confiaram e acreditaram em mim. Ao pessoal da RNP, do Cenapdne, do LIA. À peleja, aos meus colegas de mestrado e ao Orley.

E agradeço ao professor Ricardo por sua orientação e paciência.

"A liberdade/independência é diretamente proporcional ao conhecimento."

Sumário

Lista de Tabelas

Lista de Figuras

1

Introdução

A importância da otimização combinatória pode ser constatada em diversos domínios das ciências, engenharias e até mesmo da vida cotidiana. Por isto, diversas são as aplicações envolvendo a produtividade industrial ou comercial, tais como roteamento de veículos, atribuição de tripulação em vôos comerciais, projetos VLSI, controle de estoque, sistemas de comunicação, de produção, planejamento de distribuição, seqüenciamento, entre muitos outros [1].

O problema de fluxo máximo é de fácil definição: em uma rede ponderada, deseja-se enviar tanto fluxo quanto possível entre dois vértices especiais da rede (de uma fonte s para um sumidouro t), sem exceder a capacidade de nenhum arco. Ele pode ser classificado como um problema de otimização que surge em vários cenários da vida cotidiana. Deseja-se, por exemplo, enviar energia elétrica de um ponto a outro da melhor maneira possível em termos de custo e quantidade de energia enviada.

A Tabela 1.1 mostra algumas associações típicas do problema de fluxo máximo com problemas da vida real. Muitas outras aplicações são descritas em [1].

Como o problema de encontrar o fluxo máximo surge em diversos campos de pesquisa, incluindo aplicações matemáticas, ciência da computação, engenharia, gerenciamento e pesquisa operacional, algoritmos eficientes têm sido estudados extensivamente nos últimos

Tabela 1.1. *Ingredientes de Algumas Redes de Fluxo Físicas*

Aplicações	Nós	Arcos	Fluxo
Sistemas de Comunicação	Telefones, Computadores, Satélites	Cabos, Fibra Ótica, *Links*	Mensagens de voz, Dados, Transmissão de vídeos
Sistemas Hidráulicos	Estações, Reservatórios, Lagos	Tubulações	Água, gás, óleo Fluidos hidráulicos
Circuitos Integrados de Computador	Portas, registradores Processadores	Conexões	Corrente elétrica
Sistemas Mecânicos	Articulações	Hastes, vigas, molas	Calor, energia
Sistemas de Transporte	Estações de trem, Aeroportos	Rodovias, Linhas Aéreas	Passageiros, Veículos, Cargas

50 anos.

A Tabela 1.2 ([2, 3]) resume os algoritmos polinomiais conhecidos para o problema. Esses algoritmos são basicamente de dois tipos:

1. Algoritmos de *decomposição de fluxo em caminhos* e

2. Algoritmos de *manipulação de pré-fluxo*.

As complexidades de pior caso dos algoritmos são colocadas em termos do número n de vértices, do número m de arcos e, em alguns casos, em termos do limite superior U da capacidade dos arcos.

Antes destes, duas tentativas de resolução do problema já haviam aparecido. A primeira foi o *método simplex para redes*, de Dantzig, resolvendo o problema de fluxo máximo modelado como um caso especial do problema de transporte ([4]). A complexidade deste método é $O(n^2mU)$ e por este motivo ele não é citado na tabela, pois este valor não é polinomial. O segundo, descoberto por *Ford e Fulkerson*, apesar de ter melhorado a complexidade do primeiro, também não é citado na tabela por ainda se tratar de um algoritmo não polinomial em seu pior caso ($O(nmU)$). Alguns autores preferem chamar este algoritmo de *método* (assim como o primeiro, de Dantzig), uma vez que ele engloba várias implementações de diferentes tempos de execução. Este algoritmo trabalha achando caminhos da fonte para

Tabela 1.2. *Algoritmos de tempo polinomial para o problema de fluxo máximo*

Algoritmo	Ano	Autor	Tempo de Execução
1	1969	Edmonds e Karp	$O(nm^2)$
2	1970	Dinic	$O(n^2m)$
3	1974	Karzanov	$O(n^3)$
4	1977	Cherkasky	$O(n^2m^{1/2})$
5	1978	Malhotra, Pramodh Kumar e Maheshwari	$O(n^3)$
6	1978	Galil	$O(n^{5/3}m^{2/3})$
7	1978	Galil e Naamad; Shiloach	$O(nm(\log n)^2)$
8	1980	Sleator e Tarjan	$O(nm\log n)$
9	1982	Shiloach e Vishkin	$O(n^3)$
10	1983	Gabow	$O(nm\log U)$
11	1984	Tarjan	$O(n^3)$
12	1985	Goldberg	$O(n^3)$
13	1986	Goldberg e Tarjan	$O(nm\log(n^2/m))$
14	1986	Ahuja e Orlin	$O(nm + n^2\log U)$
15	1990	Cheriyan	$O(n^3/\log n)$
16	1997	Hochbaum	$O(mn\log n)$
17	1998	Goldberg e Rao	$O(\min(n^{\frac{2}{3}}, m^{\frac{1}{2}})m\log(n^2/m)\log U)$

o sumidouro (qualquer deles, um por vez), no qual é possível enviar fluxo ([4, 5]). Apesar da sua complexidade, este influenciou muitos algoritmos posteriores.

Assim, Edmonds e Karp observaram que enviar fluxo sobre caminhos mínimos geram um algoritmo polinomial (algoritmo (1)). Para aumentar a eficiência, Dinic propôs um método de achar todos os caminhos mínimos, capazes de enviar fluxo, em uma única fase, utilizando uma rede em camadas. Os algoritmos (2)-(11) usam este método. Uma outra classe de algoritmos são os que utilizam o conceito de pré-fluxo. Estes são, na prática, os mais rápidos para o problema. Assim como os algoritmos que utilizam a idéia de decomposição em caminhos derivam do algoritmo de Ford e Fulkerson, o primeiro a utilizar a idéia de pré-fluxo foi o algoritmo (3) da tabela. Mais tarde, modificações de um algoritmo genérico, também conhecido como *algoritmo preflow-push* ou, ainda, *algoritmo push-relabel*, proposto por Goldberg e Tarjan, ganharam o *status* de algoritmos mais rápidos na prática. Estes algoritmos possuem uma idéia simples e intuitiva e possuem implementações naturais tanto em modelos de computações seqüenciais como em computações paralelas. Os algoritmos (12)-(14) utilizam tal método.

Neste texto, serão tratados os algoritmos que resolvem o problema de fluxo máximo,

iniciando pelo algoritmo de Ford e Fulkerson. O objetivo é mostrar a intenção de cada autor: o que foi feito para melhorar o algoritmo anterior - quais os pontos fracos que foram eliminados e quais apareceram, que operações do algoritmo são determinantes no cálculo da complexidade e quais deles são realmente eficientes na prática.

Este texto tem uma abordagem bastante parecida com a encontrada em [1], no sentido de mostrar o estado da arte dos algoritmos existentes para o problema, comparando-os de forma teórica e prática com seus antecessores. Para a comparação teórica é utilizada as complexidades de pior caso. Para as comparações práticas, vários artigos, que tratam de estudos empíricos de alguns destes algoritmos, forão estudados e reunidos neste texto, tentando dar ao leitor uma visão global e o mais conclusiva possível. Ao final, será possível observar que, muitas vezes, as complexidades de pior caso encontradas na análise teórica nem sempre refletem estes resultados em suas implementações.

Dois novos algoritmos que não constam em [1] são tratados - um dentro de decomposição de fluxo em caminhos, considerado o mais rápido na teoria (algoritmo (17) da tabela, [3]) e o outro, dentro da abordagem de manipulação de pré-fluxos, considerado, em [6], o de melhor eficiência em termos experimentais (na prática) (algoritmo (16)). O texto tenta dar uma visão mais intuitiva e didática do funcionamento dos algoritmos, deixando os detalhes técnicos para os textos originais de seus autores.

A divisão dos algoritmos em apenas duas abordagens – *algoritmos de decomposição de fluxo em caminhos* e *algoritmos de manipulação de pré-fluxos* – é uma visão um pouco diferente da encontrada na maioria dos textos sobre o assunto. Deseja-se portanto, produzir um documento atualizado, reunindo o maior número de algoritmos para resolver o problema de fluxo máximo, apresentando-os e comparando-os de forma clara. Os algoritmos aqui mostrados estão longe de representar o número real de algoritmos existentes, contudo, foram escolhidos os que de alguma forma incluíram uma nova idéia ou que realmente melhoraram seus antecessores. Outras referências sobre fluxo máximo podem ser encontradas em livros que tratam extensivamente do assunto ([1, 7, 8]) e em capítulos de livros mais gerais ([4, 5, 9]).

Este texto se divide portanto em 6 capítulos a partir deste. O próximo trata de definições e algoritmos introdutórios básicas,referentes a grafos e análise de complexidade, necessárias para que o leitor entenda melhor os capítulos seguintes. No Capítulo 3, tem-se os conceitos e aspectos algorítmicos referentes ao problema de fluxo máximo, mais especificamente. Neste capítulo são tambem introduzidas as duas abordagens de solução para classificação dos algoritmos. Os Capítulos 4 e 5 trazem os algoritmos que se encaixam nas abordagens de decomposição de fluxo em caminhos e manipulação de pré-fluxos, respectivamente. São mostradas as idéias dos algoritmos e suas complexidades teóricas. O Capítulo 6 trata da comparação prática dos algoritmos aqui apresentados e finalmente, a conclusão e considerações finais são encontradas no último capítulo.

2

Definições e Algoritmos Básicos

Este capítulo tem como objetivos mostrar as definições básicas e propriedades elementares de Teoria dos Grafos que serão necessárias ao leitor e apresentar também algumas notações que serão usadas no decorrer deste texto. Os conceitos de Teoria dos Grafos foram retirados de várias fontes, mas compartilham a convenção adotada em [10] e [11]. Um estudo mais aprofundado pode ser encontrado em [12].

Outro ponto, aqui abordado, será a breve apresentação de diferentes estruturas de dados usadas para representar, em um computador, tanto redes (grafos) quanto outros dados manipulados pelos algoritmos. Uma explicação mais aprofundada do funcionamento e operações aplicadas a estas estruturas não faz parte do escopo deste texto (estas são citadas aqui apenas para melhor elucidação do entendimento de certos algoritmos). Contudo, é possível encontrar vasta literatura sobre o assunto ([1, 5, 13, 14]). O leitor perceberá mais adiante que a escolha de uma estrutura de dados reflete diretamente na eficiência dos algoritmos para fluxo em redes, tanto na prática como na teoria.

A terceira seção dedica-se a mostrar maneiras de medir tal eficiência (ou complexidade dos algoritmos), apresentando conceitos que permitem comparar algoritmos de forma teórica e prática. E para finalizar este capítulo introdutório, são apresentadas as idéias intuitivas de algoritmos em grafos – busca em largura, em profundidade e problema de

encontrar um caminho mínimo – considerados importantes, por serem bastante utilizados como passos intermediários nos algoritmos para o problema de fluxo máximo.

2.1. Teoria dos grafos

Muitos são os conceitos existentes em teoria dos grafos. Nesta seção são citadas apenas as definições e notações intrinsecamente ligadas ao problema de fluxo máximo em redes e utilizadas nas descrições de seus algoritmos.

2.1.1. Grafos e subgrafos

Um *grafo* $G = (V, E)$ é uma dupla ordenada de conjuntos V e E, onde V é um *conjunto de vértices* e E é um conjunto de pares não ordenados de V, chamados de *arestas* (Figura 2.1.a). Por todo este texto, denota-se por n a cardinalidade de V ($|V| = n$) e por m a cardinalidade de E ($|E| = m$). Quando houver possibilidade de confusão, será usado $V(G)$ e $E(G)$ para o grafo $G = (V, E)$. Os vértices v e w são chamados de *extremidades* da aresta vw.

Se as relações entre os vértices de V são *ordenadas*, então as arestas são denominadas *arcos*, os vértices são chamados *nós* e o grafo G passa a ser *orientado*, como na Figura 2.1.b. Um arco cujo primeiro elemento é v e o segundo w é chamado de *arco de v para w*, e representado vw, onde v é a *origem* do arco e w o *destino*. Vale notar que dois arcos vw e wv são diferentes, ao contrário do mesmo caso para grafos não orientados.

Sejam $G = (V, E)$ um grafo não orientado e $\vec{G} = (V', A)$, orientado. Dois vértices u e v são *adjacentes* em G, se $uv \in E$. Se u e v são adjacentes, então u é *vizinho* de v e v é *vizinho* de u. Diz-se que uma aresta é *incidente* a um vértice (ou um vértice v incide sobre uma aresta e) se $e = vx$ para algum vértice x. De forma análoga, dois nós u e v são adjacentes em $\vec{G}$, se o arco $uv \in A$ ou $vu \in A$.

O grafo G é *simples* se ele não admite arestas múltiplas ou laços, onde duas arestas são

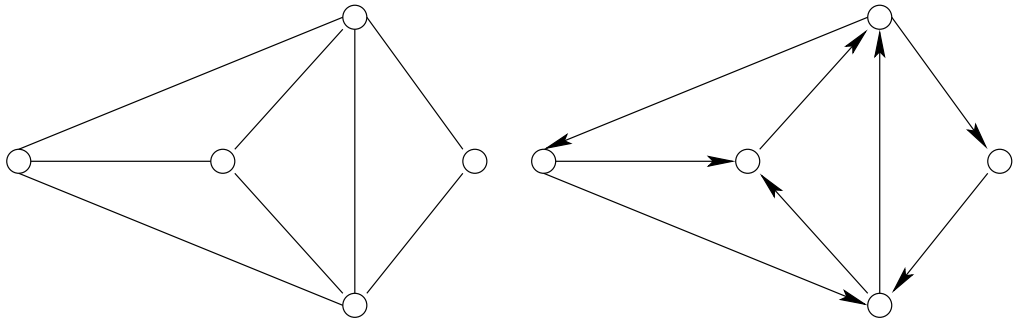


Figura 2.1. *Exemplos de grafos. (a) Grafo G não orientado. (b) Grafo $\vec{G}$ orientado.*

múltiplas se elas coincidem em ambas as extremidades e onde um *laço* é uma aresta cujas extremidades são iguais (Figura 2.2). Vale observar que dois arcos, em $\vec{G}$, são múltiplos se suas origens e destinos são iguais, ou seja, se os arcos possuem a mesma orientação. Para um vértice v, o *grau* de v em G é o número de vizinhos de v. De forma semelhante, definem-se os graus *de entrada* de um vértice v em $\vec{G}$ como sendo o número de arcos dos quais v é o destino, e *de saída* como sendo o número de arcos cuja origem é v. Se o grau de entrada de v é nulo, então v é chamdo de *fonte* de $\vec{G}$. Por outro lado, se o grau de saída de v é igual a zero, então v é chamado de *sumidouro* de $\vec{G}$. O grafo $\vec{G}$ é uma *rede* se ele contém exatamente um vértice s do tipo fonte e um vértice t do tipo sumidouro.

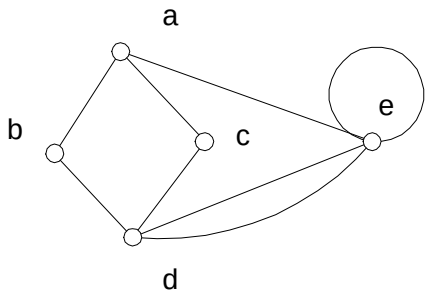


Figura 2.2. *Grafo G não simples com arestas múltiplas de, de e laço ee.*

Um grafo $H = (V', E')$ é um *subgrafo* de G se $V' \subseteq V$ e $E' \subseteq E$ e é um *subgrafo próprio* de G se $V' \subset V$ ou $E' \subset E$. Um subgrafo $H \subseteq G$ é *gerador* se $V' = V$. E ainda, um subgrafo $H \subseteq G$ é um *subgrafo induzido* de G se $V' \subseteq V$ e $E' = \{uv \in E : u, v \in V'\}$. Se

$U \subseteq V(G)$, então $G[U]$ é o subgrafo de G induzido pelos vértices de U.

A *união* de dois grafos G_1 e G_2 é o grafo G com $V = V(G_1) \cup V(G_2)$ e $E = E(G_1) \cup E(G_2)$.

O grafo G é dito *ponderado* se suas arestas recebem pesos, ou seja, existe a função $w : E(G) \longrightarrow \mathbb{R}^+$. O *peso* $w(G)$ é a soma dos pesos $\sum_{e \in E(G)} w(e)$ nas arestas. A mesma definição pode ser aplicada em $\vec{G}$, se seus arcos recebem pesos.

2.1.2. Caminhos

Um *passeio* em G é uma seqüência $C = \langle v_0 v_1 v_2 \dots v_k \rangle$ de vértices de G, tal que $v_i v_{i+1} \in E$, para todo $0 \leq i < k$ (Figura 2.3.a). Se C é um passeio sem repetição de vértices, então C é um *caminho* em G. Neste caso, o vértice v_0 é a *origem* do caminho e o vértice v_k é o seu *destino*. Ambos, v_0 e v_k, são as *extremidades* de C. Os outros vértices do caminho, ou seja $\langle v_1 v_2 \dots v_{k-1} \rangle$, são os *vértices internos*. Os conceitos de passeio e caminho continuam a existir em $\vec{G}$. Entretanto, tem-se também os conceitos de *passeio orientado* $\vec{C} = \langle v_0 v_1 v_2 \dots v_k \rangle$, onde $v_i v_{i+1} \in A$. E de forma análoga, tem-se *caminho orientado* (Figura 2.3.b). Quando não houver confusão, um caminho orientado em $\vec{G}$ será chamado simplemente de caminho.

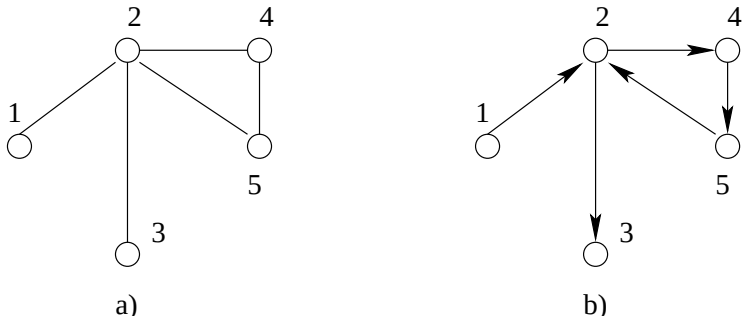


Figura 2.3. *Exemplos de passeios e caminhos: a)Passeio (1-2-5-7) e caminho (5-2-1); b)Passeio orientado (1-2-4-5-2-3) e caminho orientado (1-2-4-5).*

O *comprimento* de C é dado pelo número de arestas que o compõem (assim como o comprimento de $\vec{C}$ é o número de arcos). No caso de um grafo ponderado, este comprimento

é dado pela soma dos pesos das arestas (ou arcos) que compõem tal caminho. Um *ciclo* é um caminho em que as extremidades coincidem ($v_0 = v_k$), conforme ilustrado na Figura 2.4.a). Em $\vec{G}$, há também o conceito *ciclo orientado*, composto por arcos, como mostrado na Figura 2.4.b). O comprimento de um ciclo é dado pelo número de arestas do ciclo (que é igual ao número de vértices). Um grafo é dito *acíclico* se ele não contém ciclos.

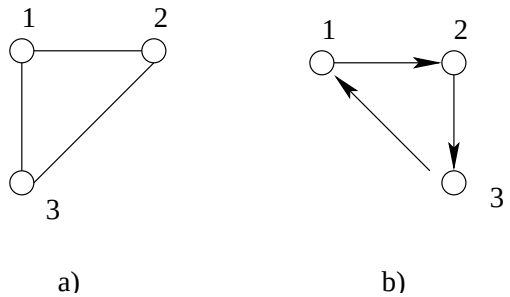


Figura 2.4. *Exemplos de ciclos: a)Ciclo (1-2-3-1); b)Ciclo orientado (1-2-3-1).*

A *distância* entre dois vértices u e v de um grafo G é o comprimento do menor caminho entre eles, denotado por $d_G(u, v)$. Quando não houver confusão, será usado apenas $d(u, v)$. Um caminho C *realiza* a distância entre u e v se o comprimento de C é igual a $d(u, v)$. Se não existe caminho entre eles, então $d_G(u, v) = \infty$.

2.1.3. Conectividade

Dois vértices são ditos *conectados* em G quando existe um passeio entre eles. O grafo G é dito *conexo* se quaisquer dois vértices do grafo são conectados. Por outro lado, se G não é conexo, então ele é chamado de *desconexo* (Figura 2.5). Uma *componente* de G é um subgrafo conexo maximal em vértices de G. Denota-se por $\mathcal{C}(G)$ o número de componentes de G. Se G é conexo, $\mathcal{C}(G) = 1$. De forma semelhante, um par a, b de nós de $\vec{G}$ são ditos *fortemente conectados* se existem caminhos orientados unindo a a b e b a a. Com isto, diz-se que $\vec{G}$ é *fortemente conexo* se todo par de nós de $\vec{G}$ é fortemente conectado.

Para subconjuntos disjuntos S e S' de $V(G)$, denota-se por $[S, S']$ o conjunto de arestas com uma extremidade em S e outra em S'. Um *corte de arestas* de G é um subconjunto de E da forma $[S, \bar{S}]$, onde S é um subconjunto próprio não-vazio de V e $\bar{S} = V - S$. Se

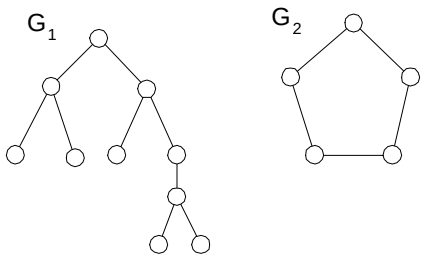


Figura 2.5. *G_1 e G_2 são conexos, mas $G_1 \cup G_2$ é desconexo.*

E' é um corte de arestas de G, então $G - E'$ é desconexo. A definição de corte de arestas também é válida para $\vec{G}$. No caso de uma rede $\vec{G}$ ponderada, um corte onde $s \in S$ e $t \in \bar{S}$ é chamado de *corte* $s-t$.Seja $c(uv)$ o peso associado a todo arco uv de $\vec{G}$, a *capacidade* do corte é:

$$c(S, \bar{S}) = \sum_{u \in S, v \in \bar{S}} c(uv)$$

Um corte é *mínimo* se possui a menor capacidade dentre todos os cortes possíveis.

Alguns possíveis cortes $s - t$ da rede da Figura 2.6 são:

1	$S = \{1\}$	$\bar{S} = \{2,3,4,5,6\}$	$(S, \bar{S}) = \{12, 13\}$	$c(S, \bar{S}) = 6$
2	$S = \{1,2\}$	$\bar{S} = \{3,4,5,6\}$	$(S, \bar{S}) = \{13, 23, 24\}$	$c(S, \bar{S}) = 10$
3	$S = \{1,3,5\}$	$\bar{S} = \{2,4,6\}$	$(S, \bar{S}) = \{12, 34, 56\}$	$c(S, \bar{S}) = 8$
4	$S = \{1,2,3,4\}$	$\bar{S} = \{5,6\}$	$(S, \bar{S}) = \{35, 45, 46\}$	$c(S, \bar{S}) = 10$
5	$S = \{1,2,3,4,5\}$	$\bar{S} = \{6\}$	$(S, \bar{S}) = \{46, 56\}$	$c(S, \bar{S}) = 6$

Os arcos pontilhados da figura ilustram o corte 3 mostrado acima. Os arcos $12, 34, 56$ são *arcos para frente*. Os arcos $23, 45$ são *arcos para trás* (Figura 2.6).

Assim, um corte $s - t$, $[S, \bar{S}]$, pode ser visto como mostrado na Figura 2.7, ou seja, um conjunto de arcos para frente vw e arcos para trás wv, com $v \in S$ e $w \in \bar{S}$.

Seja $d(v)$ a menor distância de um vértice v para o sumidouro t, cortes do tipo $[S_k, T_k]$, onde $S_k = \{v \in V : d(v) \geq k\}$ e $T_k = V - S_k$, para $k = 1, 2, \ldots, d(s)$ (s é a fonte), são ditos *cortes canônicos*.

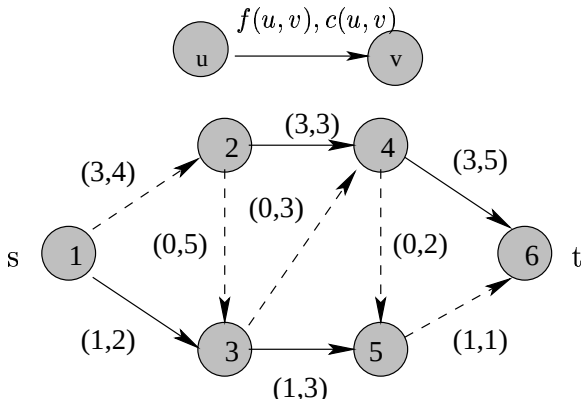


Figura 2.6. *Grafo ponderado e exemplo de um corte $s-t$ (linhas pontilhadas). A capacidade total do corte é dada pela soma das capacidades individuais dos arcos para frente.*

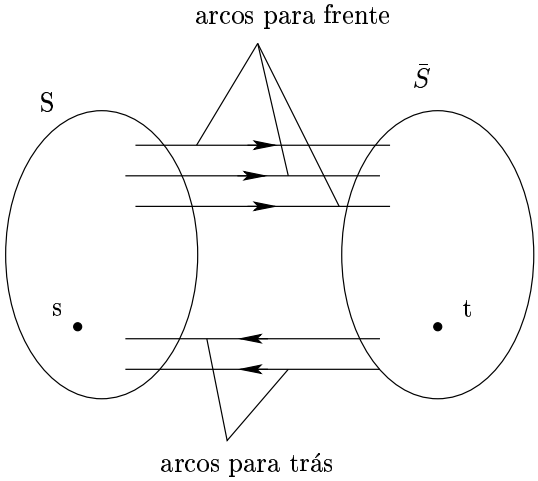


Figura 2.7. *Exemplo de um corte $s-t$, $[S,\bar{S}]$ visto como um conjunto de arcos para frente e para trás.*

2.1.4. Árvores

O grafo G é uma *árvore* se ele é um grafo acíclico e conexo. Uma coleção de árvores é chamada de *floresta*.

Toda árvore possui um vértice especial denominado *raiz*. As arestas uv de uma árvore são relações do tipo *pai-filho*, onde u é pai de v ($pai(v) = u$) e v é filho de u. Cada vértice possui um único pai, com exceção da raiz que não tem nenhum vértice pai. Os vértices que não possuem filhos são chamados de *folhas*.

Se o grafo $\vec{G}$ é uma árvore, considera-se a relação pai-filho como sendo um arco vu, onde u é pai de v e v é filho de u.

2.2. Estruturas de dados

As estruturas de dados que serão primeiramente descritas são consideradas elementares são utilizadas pelos algoritmos descritos ao longo do texto. Elas servem para manipular dados intermediários, diferentes da rede de entrada e, de alguma forma, influenciar a ordem de execução dos algoritmos.

Uma *lista linear* (ou simplesmente *lista*) pode ser utilizada para representar um conjunto ordenado de elementos. Existem várias maneiras de se armazenar uma lista no computador; desde formas bastante triviais utilizando *vetores*, até o emprego de *ponteiros* como em *listas simplesmente e duplamente encadeadas*. Como as listas representam conjuntos ordenados, estas possuem um *início* e um *fim*. As operações mais frequentes em listas são: *inclusão*, *exclusão* e *busca* de um determinado elemento. Os algoritmos destas operações podem ser encontrados em vários livros ([5, 13, 14]).

As *pilhas* são consideradas um tipo especial de listas, na qual as inclusões e exclusões só podem ser feitas de um único lado da lista, chamado *topo*. É fácil ver que esta estrutura executa na forma conhecida como *último a entrar, primeiro a sair - LIFO* (do inglês – "Last In, First Out"), pois o elemento que é inserido por último, ficará sempre situado no

topo da pilha.

A *fila* é outro caso particular de listas lineares. Diferentemente da pilha, esta se utiliza dos dois lados da lista. Porém, existe uma restrição com relação as operações de inclusão e exclusão. Nas filas, as inclusões são sempre feitas no fim da lista que passa a ser chamado de *cauda*. E as exclusões, executadas nos elementos que estão no início da lista, chamado de *cabeça*. Assim, se realizarmos uma exclusão, o elemento a ser retirado será aquele que foi incluído primeiro. Esta técnica é conhecida como *primeiro a entrar, primeiro a sair - FIFO* (do inglês – "Last In, First Out").

As *listas de prioridades* podem ser definidas como uma tabela na qual a cada um de seus elementos está associada uma prioridade. Esta prioridade é, em geral, definida através de um valor numérico.

Além das estruturas ditas elementares, existem diversas formas de representar G e $\vec{G}$ para processamento computacional. Estas incluem, entre outras:

1. Matriz de Adjacências: uma matriz $A_{n\times n}$, na qual $a_{ij} = 1$ se os vértices i e j são adjacentes, e 0 de modo contrário.

2. Matriz de Incidências: é uma matriz N com n linhas, uma para cada vértice, e m colunas, uma para cada aresta. Então tem-se que $n_{ij} = 1$ se a aresta j é incidente ao vértice i, e zero caso contrário. Para grafos orientados, os valores de n_{ij} dependem da orientação:

   - $n_{ij} = +1$, se o vértice i é o destino do arco associado à coluna j;
   - $n_{ij} = -1$, se o vértice i é a origem do arco j;
   - $n_{ij} = 0$, de outra forma.

3. Lista de Adjacências: uma lista que mantém um ponteiro ou índice para uma lista de vértices vizinhos a um outro, dado.

Na implementação dos algoritmos que serão apresentados, a escolha da estrutura lista de adjacências para representar as redes é majoritária.

2.3. Eficiência dos algoritmos

Ao se projetar algoritmos para resolver um determinado problema, normalmente procura-se analisar o desempenho ou os custos incorridos (tempo de processamento, recursos), de modo a melhorar a eficiência. Neste caso, costuma-se interpretar a *complexidade* de um algoritmo como uma forma quantitativa de tal medida, em três níveis: o problema, o algoritmo e a implementação.

Um *problema* consiste em uma questão a ser respondida, um requisito a ser preenchido, ou uma melhor situação possível a ser encontrada, chamada *solução*, normalmente em resposta a vários *parâmetros* ou *variáveis* de entrada, que são descritos mas não especificados.

Uma *instância* de um problema Π é uma especificação particular de valores para seus parâmetros. Informalmente, um *algoritmo* pode ser visto como uma seqüência de passos *bem-definidos* para transformar uma *entrada* ou instância em uma *saída* ou solução, ou seja, um *algoritmo* para Π é um procedimento passo a passo que, quando aplicado a qualquer instância de Π, produz uma solução.

É comum expressar complexidade como *uma função do tamanho da entrada.* A noção para o valor do *tamanho da entrada* depende do problema a ser estudado. A medida mais natural, é o número de parâmetros, como por exemplo, um vetor de tamanho n para ordenação. No caso do problema de fluxo máximo, como a entrada é um grafo, então o tamanho da entrada é vista, normalmente, como o número n de vértices e m de arestas.

2.3.1. Notação assintótica

Um algoritmo $\mathcal{A}$ para Π executa em tempo $O(f(n))$ se para alguma constante $c > 0$ existe uma implementação de $\mathcal{A}$ que termina após $c.f(n)$ passos, no máximo, para todas as instâncias de tamanho n. A *complexidade de um algoritmo* $\mathcal{A}$ é a menor função f tal que A executa em $O(f(n))$. *A complexidade de um problema* Π é menor função f para qual existe um algoritmo $\mathcal{A}$ de tempo $O(f(n))$ para Π, ou seja, a menor complexidade

considerando todos os possíveis algoritmos que resolvem Π.

Matematicamente, sejam $f(n)$, $g(n)$ funções de inteiros positivos para reais positivos: diz-se que $f(n) = O(g(n))$ se existe uma constante $c > 0$ tal que, para n grande o suficiente, $f(n) \leq cg(n)$.

A Figura 2.8 ilustra, graficamente, esta medida [5].

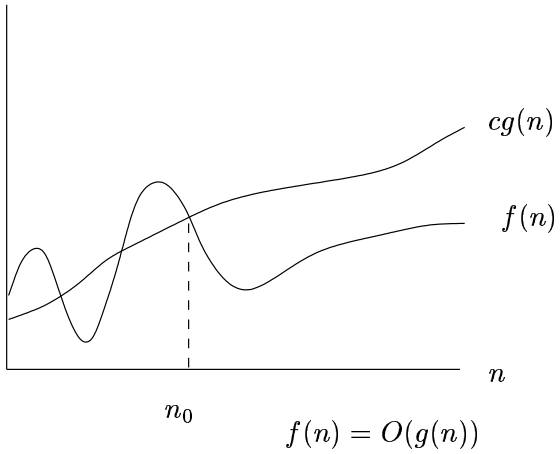


Figura 2.8. *Exemplo gráfico da notação O.*

Como a notação $O(g(n))$ é usada para representar *complexidades de pior caso* de um determinado algoritmo $\mathcal{A}$, ou seja, quando as saídas de $\mathcal{A}$ são simuladas de modo a caracterizar o maior número possível de passos, tempo ou outro custo, esta medida é então considerada um *limite superior assintótico* de $f(n)$. Existe também a notação $\Omega(g(n))$ que representa *complexidades de melhor caso*, sendo um *limite inferior assintótico*. Matematicamente, tem-se que $f(n) = \Omega(g(n))$ se existe uma constante $c' > 0$ tal que, para n grande o suficiente, $f(n) \geq c'g(n)$. É importante observar que, se $\mathcal{A}$ executa em tempo $O(f(n))$, então para *toda* instância do problema que $\mathcal{A}$ resolve, este gastará no máximo $cf(n)$ passos. Por outro lado, se um algoritmo executa em $\Omega(f(n))$ passos, *algumas* instâncias de tamanho n, serão executadas em, pelo menos, $c'f(n)$ passos.

2.3.2. Análise de algoritmos

Uma algoritmo para um determinado problema pode apresentar-se muito "bom" para algumas instâncias, resolvendo-as rapidamente, e muito "ruim" para outras. Esta diferença conduz à questão de como medir a eficiência de um algoritmo e de escolher o *melhor* entre os vários existentes que resolvem um determinado problema.

Duas formas de medir a eficiência de um algoritmo e compará-lo com outros que solucionam um mesmo problema são:

1. *Análise de pior caso* - já citada anteriormente, provê um limite superior para o número de passos que o algoritmo irá executar diante de qualquer instância. É bastante utilizada pois não depende dos recursos computacionais e, por este motivo, mais fácil para se comparar dois algoritmos diferentes, principalmente na teoria. Contudo, pode ser uma medida distante da ocorrida na prática.

2. *Análise empírica* - com o objetivo de verificar como o algoritmo funciona na prática. Neste caso, o algoritmo é implementado e testado com algumas classes de instâncias do problema. Contudo, esta medida depende de vários fatores, como arquitetura do computador utilizado, linguagem de programação, habilidades do programador, entre outras. Ou seja, análises de algoritmos sob esta visão podem ser não conclusivas, principalmente se os algoritmos foram implementados por pessoas diferentes, com linguagens de programação diversas e testados em máquinas com arquiteturas distintas.

   O que acontece na prática são vários trabalhos independentes, com objetivos semelhantes, encontrarem resultados de eficiência parecidos, então é possível constatar, de forma mais concreta e conclusiva, que realmente determinado algoritmo é melhor que outro.

   As conclusões de caráter prático deste trabalho baseiam-se neste enfoque.

2.3.3. Análise de problemas

Determinar a complexidade de um problema Π requer duas medidas:

1. *Limite superior* - a menor complexidade de pior caso dentre todos os algoritmos conhecidos que resolvem Π;

2. *Limite inferior* - a maior função f para qual foi provado (matematicamente) que todos os algoritmos possíveis que resolvem Π requerem complexidade, no mínimo, tão alta quanto f.

Um algoritmo $\mathcal{A}$ para um problema Π é dito *ótimo* se este possui sua complexidade de pior caso igual ao limite inferior de Π.

As idéias de limite inferior e limite superior estendem-se além de complexidade de problemas. Isto é, um parâmetro, por exemplo, pode ter um valor que represente seu limite inferior e outro que represente o limite superior. Muitos algoritmos de fluxo utilizam tais limites para parâmetros em seus projetos.

Assim, uma medida para saber se um algoritmo é "bom" (ou eficiente) pode ser então obtida a partir de sua complexidade de pior caso. Diz-se que um algoritmo é eficiente, se a complexidade de pior caso é limitada superiormente por uma função *polinomial* dos parâmetros de entrada. Tais algoritmos são chamados de *algoritmos polinomiais*. Exemplos de complexidades deste tipo são: $O(n^2)$, $O(nm)$, $O(nm + n^2 \log U)$, onde n, m e U são parâmetros de entrada (n número de vértices, m número de arestas e U maior valor dos pesos das arestas).

Algoritmos polinomiais que dependem apenas dos valores de n e m são ditos *fortemente polinomiais*. Caso contrário, são *fracamente polinomiais*.

Algoritmos com complexidades $O(nmU)$ e $O(m + nU)$, por exemplo, são ditos *pseudo-polinomiais*, pois para valores muito elevados de U o tempo de execução do algoritmo na prática pode ser muito alto.

Este é um assunto amplamente tratado em livros sobre algoritmos ([1, 4, 5, 13, 15, 16]).

2.4. Problemas de otimização

Como citado na seção anterior, a definição de problema está diretamente relacionada ao conceito de instância. Uma *instância de um problema de otimização* S, é um par (X, c), onde X é um conjunto de *pontos viáveis* e $c : X \to R$ é uma *função custo*; $X(S)$ indica o conjunto de pontos viáveis de S. O objetivo de um problema de otimização consiste em encontrar a *melhor configuração* de X de modo que se possa *minimizar* ou *maximizar* a função de custo c, que passa então a ser referenciada como *função objetivo.*

No caso de problemas de *maximização*, deseja-se encontrar um ponto $x^* \in X$ tal que

$$c(x^*) \geq c(y), \forall y \in X,$$

enquanto que no caso de problemas de *minimização*, o objetivo é

$$c(x^*) \leq c(y), \forall y \in X.$$

Em ambos os casos, o ponto x^* é chamado de *ótimo global* do problema.

Certos problemas de otimização são conhecidos como *problemas de programação linear.* Um problema de maximização, por exemplo, modelado em programação linear é da forma:

$$\begin{array}{rrcll} \text{Maximizar} & z & = & Cx \quad (\text{P}) & (1) \\ \text{Sujeito a} & Ax & \leq & B & (2) \\ & x & \geq & 0 & \end{array}$$

onde A é uma matriz de inteiros, B e C vetores de inteiros, (1) é a função objetivo e (2) as chamadas *restrições do problema.* Assim, os valores de x que atendem a todas as restrições são ditas soluções viáveis e, dentre todas as soluções viáveis, a que maximiza o valor de z (para este exemplo) é a solução (ótima) do problema de programação linear. Tem-se uma modelagem análoga para problemas de minimização. Contudo, como o problema tratado neste texto é de maximização, foi escolhido exemplificar com problemas deste tipo. Uma forma de resolver problemas de programação linear é utilizar o chamado *método simplex* que pode ser encontrado em [4, 17].

Um resultado importante em programação linear é o da *dualidade* entre dois problemas. Um problema (D) é dito *dual* de um P, se ele é da forma:

$$\begin{array}{rrcll} \text{Minimizar} & w & = & By \quad \text{(D)} & (1) \\ \text{Sujeito a} & A^T y & \geq & C & (2) \\ & y & \geq & 0 & \end{array}$$

e a solução ótima de (P) é no máximo o valor da solução ótima de (D). Em outras palavras, sejam x^* e y^* as soluções ótimas de (P) e (D), respectivamente. Então:

$$Cx^* \leq Cy^* \tag{2.1}$$

O problema (P) é chamado de *primal*.

Se os valores da solução tiverem de ser inteiros, então temos um *problema de programação inteira*. Um exemplo de modelagem de tal problema seria:

$$\begin{array}{rrcll} \text{Maximizar} & z & = & Cx \quad \text{(PI)} & (1) \\ \text{Sujeito a} & Ax & \leq & B & (2) \\ & x & \geq & 0 & \\ & & x & \text{inteiro} & (3) \end{array}$$

Os problemas de programação inteiras (PI) podem ser resolvidos como sendo de programação linear (PL) (através do mesmo método). Para isto, as restrições de integralidade (3) são então *relaxadas*, ou seja, omitidas do problema. Assim, se x^* é a solução ótima obtida a partir do problema inteiro relaxado, esta deve conter a solução x'^* do problema inteiro original (PI), isto é:

$$x^* \geq x'^*$$

O problema de fluxo máximo é caracterizado como um problema de otimização. Seu dual é o problema de corte mínimo, como será visto no Capítulo 3.

2.5. Algoritmos em grafos

Alguns problemas são frequentemente encontrados como passos intermediários de outros algoritmos para problemas mais complexos. Muitos dos algoritmos que resolvem o problema de fluxo máximo, por exemplo, necessita saber o comprimento do menor caminho entre dois nós. Percorrer um grafo é outro procedimento elementar em problemas que tem este tipo de estrutura como entrada. Por este motivo, serão mostrados as duas formas mais comuns para se visitar os vértices (ou nós) de um grafo - busca em largura e em profundidade.

É importante salientar que este não é propriamente o escopo do texto e que, por este motivo, os procedimentos serão apresentados de forma breve e intuitiva. O objetivo é clarear as idéias do leitor quando tais passos forem utilizados posteriormente.

2.5.1. Buscas em Largura e Profundidade

Como já citado, as buscas em grafos são consideradas procedimentos básicos para algoritmos que tem como entrada uma estrutura deste tipo. Uma busca pode ser vista como uma visita aos vértices do grafos seguindo suas arestas. Normalmente, utiliza-se uma busca para conhecer a estrutura do grafo. Essas buscas são basicamente de dois tipos - em *largura* (BFS - "Breadth-First-Search") e em *profundidade* (DFS - "Deapth-First-Search").

A idéia da busca em largura é bastante simples. Inicialmente, escolhe-se um vértice v e visita-se todos os seus vizinhos. Estes, por sua vez, a medida que são visitados, são marcados e colocados em uma fila. Quando este procedimento inicial termina, o primeiro da fila é escolhido como o novo v e são realizadas as mesmas operações de visita aos vizinhos, ainda não marcados, e colocação destes na fila. O procedimento termina, quando a fila ficar vazia. O algoritmo da Figura 2.9 ilustra a sequência de passos. É fácil ver que a complexidade do algoritmo é $O(n + m)$, pois todos os vértices e todas as arestas serão visitadas.

A busca em profundidade difere da anterior, no sentido de que deseja-se ir o mais longe

```
     Algoritmo: BFS
     Entrada: grafo G, vértice inicial v_0
1.   início
2.       visitado(v) := 0, ∀v ∈ V;
3.       fila := {v_0};
4.       enquanto fila ≠ ∅ faça
5.           v :=cabeça(fila);              % retira o primeiro elemento da fila
6.           para cada vizinho w de v faça
7.               se visitado(w) = 0 então
8.                   fila := fila ∪ {w};    % insere w no fim da fila
9.                   visitado(w) = 1;
10.              fim se
11.          fim para
12.      fim enquanto
13.  fim
```

Figura 2.9. *Algoritmo de busca em largura em um grafo G, dado um vértice inicial v_0.*

de um determinado vértice quanto possível. Assim, o procedimento pode ser resumido da seguinte forma: escolhe-se um vértice inicial v e visita-se um vizinho u. v passa a ser o *antecessor de* u (o antecessor de v é vazio), u é marcado como *visitado* e passa a ser o vértice atual. No momento em que um vértice não possuir mais vizinhos *não visitado*, este retorna ao seu antecessor que buscará novos vértices não visitados. O procedimento então acaba quando, neste retrocesso, alcança-se um vértice que não possui antecessor. O algoritmo da Figura 2.10 mostra o procedimento utilizando uma pilha, a qual se encaixa perfeitamente no controle dos vértices antecessores. Em [5, 16] podem ser encontradas versões recursivas desta busca. Sua complexidade também é $O(n+m)$ pelo mesmo motivo citado anteriormente.

No caso de uma rede de fluxo, uma busca em profundidade partindo da fonte s pode ser vista da seguinte maneira: a cada passo, o algoritmo mantém dois grupos de nós: os *rotulados*, alcançados a partir de s; e os *não rotulados*, que ainda não foram alcançados (visitados) a partir da fonte. Além disso, cada nó v que é rotulado, guarda na variável *antecessor* o valor do nó imediatamente anterior a ele no caminho escolhido de s a v. Esta variável serve para reconstruir o caminho $s-t$ e enviar fluxo através dele, caso ele exista.

```
     Algoritmo: DFS
     Entrada: grafo G, vértice inicial v_0
1.   início
2.       visitado(v) := 0, ∀v ∈ V;
3.       pilha := {v_0};
4.       enquanto pilha ≠ ∅ faça
5.           v := topo(pilha);                % retira o último elemento colocado
6.           para cada vizinho w de v faça
7.               se visitado(w) = 0 então
8.                   pilha := pilha ∪ {w};    % insere w no topo da pilha
9.                   visitado(w) = 1;
10.                  retorne à Linha 4;
11.              fim se
12.          fim para
13.      fim enquanto
14.  fim
```

Figura 2.10. *Algoritmo de busca em profundidade em um grafo G, dado um vértice inicial v_0.*

2.5.2. Problema do caminho mínimo

O *Problema de Caminho Mínimo* (conhecido na literatura como "Shortest Path Problem") consiste em achar, em um grafo ponderado simples $G = (V, E)$, um caminho de peso mínimo conectando dois vértices especificados u_0 e v_0. Os pesos são não-negativos. Adota-se a convenção de que o peso $w(uv) = \infty$ se $uv \notin E$.

O algoritmo a ser descrito foi descoberto por Dijkstra (1959). Ele encontra não apenas um caminho (u_0, v_0) mínimo, mas caminhos mínimos de u_0 para todos os outros vértices de G.

Seja S um subconjunto próprio de V tal que $u_0 \in S$, e $\bar{S} = V - S$. Se $P = \langle u_0, \ldots, \bar{u}, \bar{v} \rangle$ é um caminho mínimo de u_0 para $\bar{S}$ então, evidentemente, $\bar{u} \in S$. Logo, $d(u_0, \bar{v}) = d(u_0, \bar{u}) + w(\bar{u}\bar{v})$ e a distância de u_0 para $\bar{S}$ é dada pela equação:

$$d(u_0, \bar{S}) = \min_{u \in S, v \in \bar{S}} \{d(u_0, u) + w(uv)\} \tag{2.2}$$

Esta fórmula é a base do algoritmo de Dijkstra. Iniciando com o conjunto $S_0 = \{u_0\}$, uma seqüência crescente $S_0, S_1, \ldots, S_{v-1}$ de subconjuntos de V é construída de forma que,

ao final do estágio i, caminhos mínimos de u_0 para todos os vértices em S_i são conhecidos.

O primeiro passo é determinar o vértice mais próximo a u_0, o que é obtido calculando-se $d(u_0, \bar{S}_0)$ e selecionando-se um vértice $u_1 \in \bar{S}_0$ tal que $d(u_0, u_1) = d(u_0, \bar{S}_0)$. Pela Equação(2.2):

$$d(u_0, \bar{S}_0) = \min_{u \in S_0, v \in \bar{S}_0} \{d(u_0, u) + w(uv)\} = \min_{v \in \bar{S}_0} \{w(u_0 v)\}$$

e assim $d(u_0, \bar{S}_0)$ é facilmente calculado. Ajusta-se então $S_1 = \{u_0, u_1\}$ e $P_1 = \langle u_0, u_1 \rangle$. Generalizando, se o conjunto $S_k = \{u_0, u_1, \ldots, u_k\}$ e os caminhos mínimos correspondentes $P_1, P_2, \ldots, P_k$ já tiverem sido determinados, calcula-se $d(u_0, \bar{S}_k)$, usando(2.2), e seleciona-se um vértice $u_{k+1} \in \bar{S}_k$ tal que $d(u_0, u_{k+1}) = d(u_0, \bar{S}_k)$. Ainda pela Equação(2.2), $d(u_0, u_{k-1}) = d(u_0, u_j) + w(u_j u_{k+1})$ para algum $j \leq k$. Obtem-se então um caminho $\langle u_0, u_{k+1} \rangle$ mínimo adicionando-se a aresta $u_j u_{k+1}$ ao caminho P_j .

O algoritmo de Dijkstra é um refinamento do procedimento acima. Ao longo do algoritmo, cada vértice v possui uma *etiqueta* $l(v)$ representando um limite superior para $d(u_0, v)$. Inicialmente $l(u_0) = 0$ e $l(v) = \infty$ para $v \neq u_0$. A medida que o algoritmo avança, as etiquetas são modificadas para que, ao final do estágio i,

$$l(u) = d(u_0, u) \text{ para } u \in S_i$$

e

$$l(v) = \min_{u \in S_{i-1}} \{d(u_0, u) + w(uv)\} \text{ para } v \in \bar{S}_i$$

Quando o algoritmo termina, a distância de u_0 para v é dada pelo valor final da etiqueta $l(v)$. A complexidade do algoritmo de Dijkstra é $O(n^2)$ ([1, 4, 5, 8]). Em [1, 5, 8, 18] são encontrados outros algoritmos. Recentemente, foi descoberto um algoritmo de complexidade linear para este problema.

Existem variações na saída do algoritmo, ou seja, se, em vez das distâncias aos vértices, os caminhos fossem almejados, bastaria que fossem acrescentadas variáveis em forma de lista para armazenar os predecessores dos vértices ao longo da execução. Se apenas o caminho mínimo até o vértice v_0 fosse desejado, o passo 6 do algoritmo pode ser modificado

Algoritmo: CAMINHOS MÍNIMOS (DIJKSTRA)
Entrada: grafo ponderado G, vértice inicial u_0
Saída: distâncias $d(u_0, v)$

1. **início**
2. $l(u_0) := 0$;
3. $l(v) := \infty$ para $v \neq u_0$;
4. $S_0 := \{u_0\}$;
5. $i := 0$;
6. **enquanto** $\bar{S}_i \neq 0$ **faça** % $i = v - 1$
7. **para** cada $v \in \bar{S}_i$ **faça** % $l(v) = d(u_0, v)$, $\forall v \in V$
8. $l(v) := \min\{l(v), l(u_i) + w(u_i v)\}$;
9. $u_{i+1} := t$, para $l(t) = \min_{v \in \bar{S}_i}\{l(v)\}$;
10. $S_{i+1} := S_i \cup \{u_{i+1}\}$; % $S = S \cup \{u_{i+1}\}$
11. $i := i + 1$;
12. **fim enquanto**
13. **fim**

Figura 2.11. *Algoritmo de Dijkstra para problema de caminhos mínimos.*

para checagem de uma variável contendo o vértice atual, e se essa é igual a v_0 o algoritmo pára. No caso de uma rede de fluxo G, algumas versões para o problema seriam, por exemplo, encontrar um caminho mínimo entre ([8]):

- a fonte s e o sumidouro t;
- a fonte s e todo nó v de G e
- o sumidouro t e todos os outros nós v.

3

Aspectos Algorítmicos do Problema de Fluxo Máximo

Este capítulo é dedicado às características, definições e formulações básicas pertinentes ao problema de fluxo máximo em redes.

São mostrados, portanto, a definição formal do problema e sua modelagem como um problema de programação linear inteira. Dentro desta mesma linha de estudo, é abordada sua relação com o problema de corte mínimo, considerada uma das principais relações, senão a mais importante, no escopo deste problema. Este tópico é tratado aqui, pois este resultado é utilizado para provar a corretude dos algoritmos.

Além disso, outros conceitos e aspectos teóricos que fundamentam os métodos algorítmicos também serão vistos, com as respectivas definições e notações utilizadas nos algoritmos.

Duas abordagens de resolução do problema em questão são apresentadas, com o objetivo de preparar o leitor na classificação que será dada aos diversos algoritmos existentes. Será mostrado, portanto, as estratégias de *decomposição de fluxo em caminhos* e *manipulação de pré-fluxos*. Esta divisão é uma visão deste trabalho.

3.1. Formulação do problema

Dada uma rede ponderada $\vec{G} = (V, E)$ simples e conexa, denota-se por *capacidade* $c(v, w)$, para cada arco $vw \in E$, o peso deste arco (valor inteiro e positivo). Supõe-se que $c(v, w) = 0$ para todo $vw \notin E$. Denomina-se U o maior valor finito para a capacidade dos arcos, ou seja, $U = \max_{(v,w)\in E}\{c(v, w) < \infty\}$. No restante do texto, considera-se ainda que a rede não possui um caminho direcionado da fonte s para o sumidouro t formado apenas por arcos com capacidades infinitas. Se alguma das características da rede (grafo simples, direcionado, capacidades inteiras) não forem respeitadas, é possível transformar tal rede em uma que atenda as exigências da definição [1].

Um *fluxo* f em $\vec{G}$ é uma função inteira definida sobre $V^2 = V \times V$ satisfazendo as seguintes restrições:

$$\begin{aligned}
f(v, w) &\leq c(v, w) && \forall (v, w) \in V^2 && \text{(capacidade)} && (3.1)\\
f(v, w) &= -f(w, v) && \forall (v, w) \in V^2 && \text{(antisimetria)} && (3.2)\\
\textstyle\sum_{u\in V} f(u, v) &= 0 && \forall v \in V - \{s, t\} && \text{(conservação de fluxo)} && (3.3)
\end{aligned}$$

O valor $|f|$ do fluxo f é o fluxo da rede no sumidouro:

$$|f| = \sum_{v\in V} f(v, t) = \sum_{v\in V} f(s, v)$$

O problema de fluxo máximo é, então, encontrar um fluxo de valor máximo. A modelagem como um problema de programação inteira é imediata:

$$\begin{aligned}
&\text{Maximizar} && \textstyle\sum_{vt\in E} f(v, t) && \\
&\text{Sujeito a} && \textstyle\sum_{uv\in E} f(u, v) - \sum_{vw\in E} f(v, w) = 0, && v \in V - \{s, t\}\\
& && f(u, v) \leq c(u, v), && uv \in E\\
& && f(u, v) \text{ inteiro} && uv \in E
\end{aligned}$$

Com relação as capacidades dos arcos serem inteiras e positivas, é possível encontrar redes com valores reais (não-inteiras). Para alguns algoritmos esta premissa pode ser

desconsiderada, mas para aqueles cuja complexidade depende do valor de U, então a integralidade é necessária. Além disso, a rede não contém um caminho direcionado da fonte s para o sumidouro t apenas com arcos de capacidades infinitas, pois é fácil ver que, se existisse tal caminho, então o valor de um fluxo máximo seria ilimitado.

Algoritmos específicos para redes não direcionadas, com capacidades reais ou unitárias podem ser encontrados na literatura ([19]). Contudo, estes casos não serão tratados neste texto.

Existe uma relação entre o problema de fluxo máximo e o de corte mínimo, a qual traz resultados importantes para ambos os problemas. A seguir é apresentado tal resultado. Seja f um fluxo em $\vec{G}$ e $[S, \bar{S}]$ um corte $s-t$. Por definição tem-se que:

$$\sum_{v \in S, w \in \bar{S}} f(v,w) \leq \sum_{v \in S, w \in \bar{S}} c(v,w) \tag{3.4}$$

Portanto, a quantidade máxima de fluxo que pode passar através de uma rede é limitada superiormente pela capacidade do corte.

Este resultado pode ser visto como o da Seção 2.4 (Equação(2.1)) sobre problemas primais e duais. Ou seja, se o problema de fluxo máximo pode ser modelado como um problema de programação linear (e já foi visto que isto é possível), então é fácil notar que seu problema dual é o problema de achar um corte mínimo.

Assim, uma modelagem para o dual do problema de fluxo máximo seria:

$$\begin{array}{ll} \text{Minimizar} & \sum_{(u,v) \in E} c(u,v) g_{uv} \\ \text{Sujeito a} & h_u - h_j + g_{uv} \geq 0 \\ & g_{uv} \geq 0 \end{array}$$

onde g e h correspondem as restrições de capacidade e conservação de fluxo, respectivamente. Considere então um corte $s-t$, $[S, \bar{S}]$ e a modelagem do dual do problema de fluxo acima. Se $h_u = 0$ para $u \in S$ e $h_u = 1$, caso contrário; e $g_{uv} = 1$ se uv é um arco para frente, ou seja, $u \in S$ e $v \in \bar{S}$, e $g_{uv} = 0$ se uv é um arco para trás. Logo, estas escolhas

de g e h provêem uma solução viável para o dual, cuja função objetivo é a capacidade do corte ([20]).

Sabe-se que um fluxo máximo f^* é maior que qualquer fluxo f. E um corte $\bar{K} = [S, \bar{S}]$ é mínimo se a capacidade deste é menor que a capacidade de qualquer corte K. Pela Equação (3.4), tem-se um caso particular:

$$|f| \leq |f^*| \leq c(\bar{K}) \leq c(K)$$

Logo, pode-se provar facilmente o teorema a seguir [12].

Teorema 1. *Seja f um fluxo máximo e $K = [S, \bar{S}]$ um corte mínimo, então $|f| = c(K)$.*

Portanto, se o valor do fluxo é igual a capacidade do corte, então o fluxo é máximo e o corte é mínimo. Vale observar que quando este teorema se verifica tem-se (pela solução do problema dual):

1. $f(v, w) = c(v, w), \forall v \in S$ e $w \in \bar{S}$;
2. $c(w, v) = 0, \forall w \in \bar{S}$ e $v \in S$

Este resultado é conhecido como *Teorema de Fluxo Máximo, Corte Mínimo* (*Max-Flow, Min-Cut Theorem*) de Ford e Fulkerson. Considerado um teorema central no estudo de fluxo em redes (talvez o mais importante teorema no domínio deste problema) [1], encontrado na literatura enunciado de várias formas, sendo todas equivalentes entre si [4, 5, 12].

Este teorema tem implicações teóricas nas áreas de matemática discreta, otimização em geral e programação linear. Em [4, 20], os dois problemas (fluxo máximo e corte mínimo) são apresentados com uma modelagem nesta última área (programação linear) e o teorema de maneira mais aprofundada. Este resultado é utilizado nas provas de corretude dos algoritmos projetados para fluxo máximo. O fato de que estes dois problemas são equivalentes também possui implicações práticas, ou seja, a teoria e os algoritmos desenvolvidos para o

problema de fluxo máximo podem também ser aplicados em vários problemas práticos que são naturalmente problemas de corte mínimo, permitindo assim a modelagem de diversas aplicações como um problema de fluxo máximo, mesmo que estas não apresentem uma estrutura de fluxo em redes.

3.2. Fluxo, caminhos e distâncias

As próximas seções se dedicam a conceitos e notações básicas utilizados pelos algoritmos que resolvem o problema.

3.2.1. Rede residual e caminhos $s-t$

Define-se *capacidade residual* $c_f(v,w)$ de um par $(v,w) \in V^2$ como sendo $c(v,w) - f(v,w)$. Este valor significa a quantidade de fluxo que ainda pode ser enviada por um arco sem que exceda a sua capacidade. O par $(v,w) \in V^2$ define um *arco residual* se $c_f(v,w) > 0$. Uma *rede residual* $G_f = (V, E_f)$ é a rede cujo conjunto de nós é o mesmo da rede original e o conjunto de arcos são os residuais.

Um *fluxo residual* é a diferença entre um fluxo máximo f^* e o valor do fluxo f, definido para cada arco a, ou seja, se $f^*(a) \geq f(a)$, então o fluxo residual deste arco será $f^*(a) - f(a)$.

Pode ser observado que existem duas maneiras de um arco possuir capacidade residual positiva: ou (v,w) é um arco com menos fluxo que capacidade, ou (w,v) possui fluxo positivo. No primeiro caso, mover fluxo de v para w aumenta a quantidade de fluxo no arco (v,w), enquanto que no segundo caso, esta operação diminui a quantidade de fluxo em (w,v).

Nos algoritmos que serão apresentados neste texto, a menos que seja dito o contrário, trabalha-se na *rede residual*. Contudo, é possível trabalhar diretamente na rede original, se desejado, como mostrado em [1].

Um *caminho* $s-t$ é um caminho *orientado* de s para t ao longo do qual pode-se enviar

fluxo. A *capacidade residual* de um caminho $s-t$ é definida como a menor das capacidades residuais dos arcos que pertencem ao caminho.

A Figura 3.1.a mostra uma rede $\vec{G}$ com um determinado fluxo f. A Figura 3.1.b apresenta a rede residual G_f e um caminho $s-t$ no qual é possível enviar fluxo $p = (1, 3, 2, 4)$. A capacidade residual do caminho p é $\min\{c_f(1,3), c_f(3,2), c_f(2,4)\} = \min\{1, 2, 1\} = 1$.

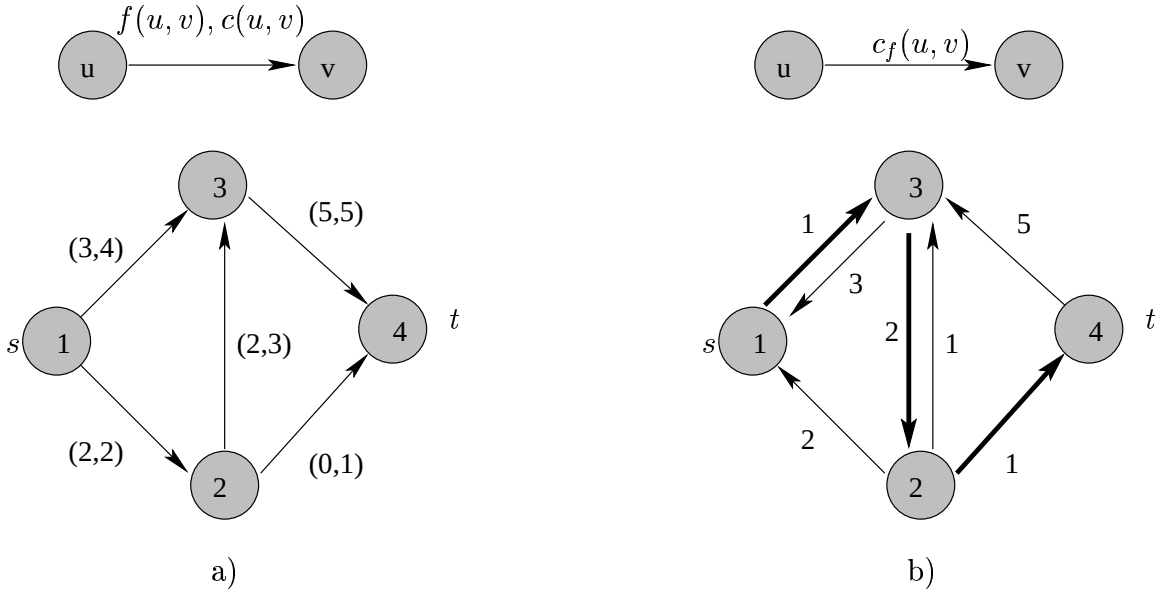


Figura 3.1. *Exemplo de uma rede residual: a) rede original $\vec{G}$ com fluxo f; b) rede residual G_f e caminho p indicado pelas arestas mais escuras.*

3.2.2. Função de comprimento e rótulo de distância

Como será visto adiante, muitos algoritmos utilizam a idéia de distância de um determinado nó até o sumidouro ou até a fonte. Contudo, para determinar este valor para todos os nós da rede, é necessário a definição de comprimento de um arco. Por essa razão, admite-se a existência de uma *função de comprimento*, arbitrária, para os arcos $l : E \to \mathbb{R}^+$. Grande parte dos algoritmos utilizam uma função de comprimento unitária. Contudo, será mostrado um algoritmo que utiliza um valor diferente para esta medida.

Diz-se que uma função $d : V \to \mathbb{R}^+$ é um *rótulo de distância*, com respeito a l, se as duas propriedades a seguir são válidas.

$$d(t) = 0 \tag{3.5}$$

$$l_d(v,w) = l(v,w) + d(w) - d(v) \geq 0 \tag{3.6}$$

Intuitivamente, (3.6) indica que a distância que se percorre em um caminho qualquer de v para t não deve ser menor que o rótulo de distância de v. Denomina-se, $l_d(v,w)$, o *custo reduzido* de um arco $vw \in E$. De uma forma geral, dado um caminho qualquer $p = \langle v_1, v_2, ..., v_n \rangle$, o comprimento de p é dado pelo somatório dos comprimentos dos arcos que o compõem, ou seja, $l(p) = \sum_{v_i v_j \in p} l(v_i, v_j)$. E o custo reduzido deste caminho, $l_d(p)$, pode ser encontrado da seguinte forma:

$$\begin{array}{rrcrcrcr}
 & l_d(v_1,v_2) & = & l(v_1,v_2) & + & \boxed{d(v_2)} & - & d(v_1) \\
+ & l_d(v_2,v_3) & = & l(v_2,v_3) & + & \boxed{d(v_3)} & - & \boxed{d(v_2)} \\
+ & l_d(v_3,v_4) & = & l(v_3,v_4) & + & \boxed{d(v_4)} & - & \boxed{d(v_3)} \\
 & \vdots & & & & & & \\
+ & l_d(v_{n-1},v_n) & = & l(v_{n-1},v_n) & + & d(v_n) & - & \boxed{d(v_{n-1})} \\
\hline
 & \sum_{v_i v_j \in p} l_d(v_i,v_j) & = & \sum_{v_i v_j \in p} l(v_i,v_j & + & d(v_n) & - & d(v_1)
\end{array}$$

Os termos destacados serão cancelados no somatório, obtendo $l_d(p) = l(p) + d(v_n) - d(v_1)$. No caso particular de um caminho p de um nó v até o sumidouro t, por (3.5), tem-se $l_d(p) = l(p) - d(v)$. Como o custo reduzido deve ser não negativo, devido à (3.6), obtem-se $l(p) \geq d(v)$. Portanto, o rótulo de distância de v deve ser menor ou igual ao comprimento de qualquer caminho. Assim, é possível considerar o valor do rótulo de distância $d(v)$ como sendo uma estimativa do valor do menor caminho, de qualquer nó v para o sumidouro t.

As propriedades a seguir, referentes a rótulos de distância, são muito importante nas provas de corretude dos algoritmos que serão vistos adiante.

Propriedade 1. *Se os rótulos das distâncias são válidos, o rótulo $d(v)$ é um limite inferior para o comprimento do menor caminho orientado entre o nó v e o nó t na rede residual.*

Propriedade 2. *Seja um fluxo f em G_f, e $d(v)$ os rótulos de distância. Se $f' - f$ é um fluxo em G'_f, então $d'(v) \geq d(v)$, onde $d'(v)$ é a distância de v em G'_f, para todo $v \in V(G'_f)$.*

Esta propriedade se verifica, porque o rótulo de distância sendo um limite inferior, não pode diminuir.

Propriedade 3. *Se $d(s) \geq n$, a rede residual não contém caminhos orientados da fonte para o sumidouro.*

A partir da definição de rótulo de distância, tem-se também a definição de *arco admissível* como sendo todo arco vwque satisfaz a condição de $d(v) \geq d(w) + l(v, w)$. Todos arco que não é admissível é *inadmissíveis*. Um caminho de s a t composto apenas por arcos admissíveis é referenciado como um *caminho admissível*.

Uma rede construída apenas com os menores caminhos de todo nó v para o sumidouro t é denominada de *rede em camadas*. Se considerarmos os rótulos de distância $d(v)$ como sendo a medida do menor caminho de qualquer nó v para t, é fácil ver porque esta rede recebe o nome de rede em camadas, pois seu conjunto de nós pode ser particionado em camadas $V_0, V_1, V_2, \ldots, V_l$, onde a camada k contém os nós cujos rótulos são iguais a k. Além disso, para cada arco vw da rede, $v \in V_k$ e $w \in V_{k-1}$ para algum k. A Figura 3.2 mostra um exemplo de uma rede residual e a rede em camadas correspondente.

Um fluxo f é dito *bloqueante*, em uma rede em camadas (construída a partir de G_f), se não contém caminhos $s - t$ [1].

A propriedade a seguir relaciona o rótulo de distância da fonte e um fluxo bloqueante em uma rede em camadas.

Propriedade 4. *Seja um fluxo f, $d(s)$ o rótulo de distância da fonte em G_f, a rede em camadas G', construída a partir de G_f, e função de comprimento unitária. Se $f' - f$ é um fluxo bloqueante em G', então $d'(s) > d(s)$, onde $d'(s)$ é a distância $s - t$ em G'_f.*

Esta é facilmente verificada, pois, em uma determinada rede em camadas G', por definição e construção, a distância de qualquer nó para t é a menor. Além disso, se f' é um fluxo bloqueante, então não existe mais caminho $s - t$ em G'. Por este motivo, s não

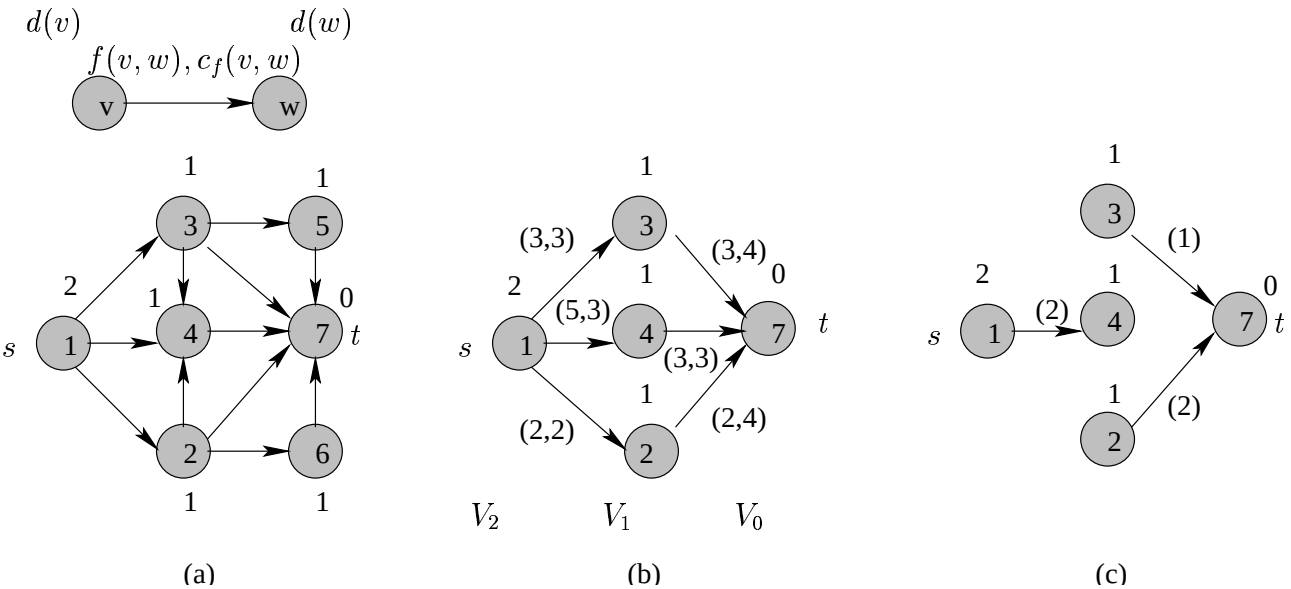


Figura 3.2. *Exemplo de uma rede em camadas. (a) rede residual; (b) rede em camadas resultante; (c) rede em camadas depois do cálculo do fluxo bloqueante $f = 8$.*

alcança t por nenhum caminho que existia na rede de menores caminhos anterior (G'). Portanto, a distância de s para t na rede residual seguinte será estritamente maior que a distância anterior.

3.3. Abordagens algorítmicas

Nas próximas seções serão apresentadas duas estratégias básicas de solução, utilizadas pelos algoritmos para fluxo máximo. Através dos estudos feitos nas características dos diversos algoritmos, observou-se que estes poderiam ser classificados em dois grupos.

Alguns autores costumam fazer uma classificação diferenciada, ou seja, algoritmos de decomposição em caminhos, de escala de capacidades, de fluxo bloqueante e de manipulação de pré-fluxos [3]. Contudo, como será mostrado, é possível reduzir este número de classes, a partir das semelhanças encontradas nos diversos algoritmos.

A classificação deste texto é feita levando em consideração a forma como o algoritmo decompõe um fluxo máximo, ou seja, se o fluxo é formado por vários caminhos da fonte

para o sumidouro, ou entre pares arbitrários de nós.

Assim, um fluxo máximo em uma rede pode ser visto, por exemplo, como a união de caminhos da fonte para o sumidouro. Esta visão é bastante natural. O leitor pode imaginar um exemplo da Tabela 1.1, uma rede formada por estações (reservatórios) e tubulações. A partir de uma estação origem deseja-se enviar o máximo de água, por exemplo, até outra estação destino. A água passa pelas tubulações que possem uma capacidade limitada e também por estações intermediárias que apenas repassam o que nelas chega. A água percorre caminhos da origem até o destino e estes caminhos possuem uma capacidade limitada pela menor das capacidades das tubulações que o compõem. Assim, se todos os caminhos possíveis que levam da origem ao destino pudessem ser colocados um ao lado do outro, duplicando tubulações e estações que aparecem em mais de um caminho, teria-se a mesma quantidade de água chegando, esta seria máxima e vista como uma composição dos quantidades passando pelos caminhos.

Um outra forma de visualizar o envio de fluxo através da rede seria, para o mesmo exemplo de reservatórios, imaginar que estes encontram-se em alturas diferentes de forma que a origem está no ponto mais alto e o destino no ponto mais baixo. Esta abordagem então trabalha de maneira mais localizada que a anterior, ou seja, envia-se o máximo de fluxo possível para reservatórios em níveis abaixo, mas sem se preocupar se toda a quantidade enviada poderá alcançar o destino. O que chega em um reservatório e não tem como escoar, fica acumulado no reservatório, como um excesso. Assim, um caminho não é construído *a priori* e os reservatórios (nós) e as tubulações (arcos) são analisados isoladamente. No instante em que o máximo de fluxo consegue atingir o reservatório destino, o procedimento pára. Desta forma, o fluxo é enviado por caminhos entre quaisquer dois reservatórios, o que, claramente, não significa dizer que todo fluxo enviado conseguirá atingir o destino.

É importante notar que este procedimento não respeita a restrição de conservação de fluxo (3.3). O que entra pode ser maior que o que sai de um reservatório. Portanto, ao final, para se ter uma solução verdadeira para o problema de fluxo, pode ser necessário

retornar, para a origem, o que ficou a mais nos reservatórios.

3.3.1. Abordagem de decomposição de fluxo em caminhos

Seja f um fluxo passando através da rede, um caminho C qualquer de s para t e δ_C a quantidade de fluxo passando por este caminho. Define-se uma operação *união* ou *soma de caminhos* da seguinte forma:

$$|f| = \sum_{\forall C_{s-t}} \delta_C$$

Contudo, esta ainda não representa uma solução para o problema de fluxo, pois é também necessário conhecer a quantidade de fluxo que passa em cada arco da rede ($f(u, v)$, para todo $uv \in E$). Assim, define-se uma operação *soma de arcos*:

$$f(u, v) = \sum_{\forall C_{s-t}: uv \in C} \delta_C$$

ou seja, o fluxo passando por um arco é igual a soma das capacidades de todos os caminhos $s - t$ para os quais este arco participa.

A partir das operações e definições acima, pode-se afirmar que:

Teorema 2. *Um fluxo máximo em uma rede pode ser encontrado a partir de uma composição de fluxo por caminhos.*

Esta prova pode ser feita por indução no número de caminhos. Sabe-se que a solução de um problema de programação inteira (ou linear) pode ser encontrada, percorrendo-se um espaço de soluções até encontrar uma que atenda o objetivo (maximizar ou minimizar). Portanto, seja f um fluxo formado pela união de vários caminhos ($|f| = \sum_{\forall C_{s-t}} \delta_C$) e um fluxo f' encontrado imediatamente após f. É fácil ver que f' é obtido, com as operações acima, a partir de f e um novo caminho encontrado C' ($|f'| = \sum_{\forall C_{s-t}} \delta_C + \delta_{C'}$).

Quando não existir mais caminhos $s - t$, então o fluxo encontrado é máximo e este foi formado por uma união de caminhos levando fluxo.

É fácil ver que a quantidade máxima de fluxo (δ_C) que pode passar da fonte para o sumidouro em qualquer caminho C, é determinada pela menor das capacidades dos arcos que formam o caminho, tais arcos são chamados de *gargalos*. Portanto, considerando a rede residual (definição de caminho $s-t$):

$$\delta_C = \min_{\forall uv \in C}\{c_f(u,v)\}$$

Com isto, pelo menos um arco do caminho é *saturado*, ou seja, não poderá mais ser usado em outros caminhos. Mas os arcos que não foram utilizados em suas máximas capacidades, poderão participar de outros caminhos com o que *sobrou* de suas capacidades (por isto a preferência de uso da rede residual a rede original). Então, quando não existir mais caminhos da fonte para o sumidouro na rede residual, aplica-se as operações soma de caminhos e soma de arcos, obtendo a solução do problema.

A Figura 3.3 mostra esta idéia em forma de algoritmo.

Algoritmo: DECOMPOSIÇÃO DE FLUXO EM CAMINHOS
1. **início**
2. $f := 0$;
3. $f(u,v) := 0$, para todo arco $uv \in E$;
4. **enquanto** G_f contém um caminho $s-t$ **faça**
5. identifica um caminho $s-t$ (C);
6. $\delta_C := \min\{c_f(u,v)\}$, para todo arco $uv \in C$;
7. $f := f + \delta_C$; % operação *soma de caminhos*
8. $f(u,v) := f(u,v) + \delta_C$, para todo arco $uv \in C$; % operação *soma de arcos*
9. atualiza G_f; % retira arcos saturados em C
10. **fim enquanto**
11. **fim**

Figura 3.3. *Algoritmo de decomposição de fluxo em caminhos*

A corretude deste algoritmo pode ser facilmente provada utilizando o Teorema 1, os conceitos de rede residual e caminhos $s-t$. Este algoritmo pára, pois não existe nenhum caminho $s-t$ apenas com capacidades infinitas e cada caminho satura pelo menos um arco. No momento que não existem mais caminhos direcionados da fonte para o sumidouro, toda possibilidade de enviar fluxo foi utilizada, então o fluxo é máximo.

Esta abordagem é bastante simples e intuitiva e foi a primeira a surgir, vindo a influenciar muitos algoritmos. Vale observar, então que o ponto principal desta estratégia é a maneira de construir e escolher caminhos, de forma a conseguir enviar a maior quantidade de fluxo com o menor número de caminhos escolhidos, ou por caminhos construídos de forma rápida (Linha 5). Um outro ponto importante, mas com menor influência na eficiência dos algoritmos, é determinar que não existem mais caminhos (Linha 4).

3.3.2. Abordagem de manipulação de pré-fluxos

Um ponto negativo dos algoritmos que utilizam a idéia de decomposição de fluxo em caminhos é enviar fluxo muitas vezes por caminhos que possuem arcos em comum. Os algoritmos baseados na idéia de pré-fluxo tentam evitar tal gasto de tempo. Deste modo, os caminhos não são criados antes do envio de fluxo, mas sim enviado pelos arcos, individualmente.

Para apresentar melhor esta abordagem, se faz necessária a definição de *pré-fluxo*. Um *pré-fluxo* é um fluxo no qual a quantidade total de fluxo que *entra* em um nó da rede pode ser maior que a quantidade total que *sai* deste nó, com exceção da fonte s.

Formalmente, tem-se que um pré-fluxo é uma função inteira definida sobre $V \times V$ satisfazendo (3.1) e (3.2), assim como a seguinte relaxação da restrição (3.3):

$$e(v) = \sum_{u \in V} f(u, v) \geq 0, \quad \forall u \in V - \{s, t\} \tag{3.7}$$

Define-se $e(v)$ como sendo o fluxo em *excesso* em um nó v. A partir daí tem-se a definição de *nó ativo*. Um nó v é chamado de *ativo* se $v \in V - \{s, t\}$, $d(v) < \infty$ e $e(v) > 0$.

A idéia de *altura*, de um determinado nó com relação a outro, é empregada em termos de rótulos de distância. A medida que um nó não consegue mais enviar fluxo para baixo, seu nível é elevado para que se crie novas possibilidades de envio.

Esta abordagem, portanto, pode ser dividida em duas fases bem definidas:

Fase 1. tenta-se enviar o máximo de fluxo possível da fonte para o sumidouro, analisando

(um por vez) os nós de excessos positivos e níveis de altura mais baixos que a fonte (escoando fluxo de reservatórios mais altos para mais baixos);

Fase 2. quando não é mais possível descer fluxo até o sumidouro, ou seja, quando não existirem mais tais nós, tem-se então uma solução parcial (a solução ainda não é viável), pois o fluxo em alguns nós, provavelmente não estão respeitando a restrição de conservação, logo, é preciso retornar os excessos restantes para a fonte.

Pelas propriedades de rótulos válidos, vistos na Seção 3.2.2, considerando o rótulo do sumidouro $d(t) = 0$, para a fonte, coloca-se como valor de rótulo, um limite superior do menor caminho de s para t, ou seja, $d(s) = n$; e $d(v) < n$, para todo nó $v \in V - \{s, t\}$. Então, é importante observar que ao final da Fase 1, existirá um corte $s - t$, $[S, \bar{S}]$, onde os nós com rótulos de distância maiores ou iguais a n, integrarão a partição S e os outros, a partição $\bar{S}$. É fácil ver que este é realmente um corte onde $s \in S$ e $t \in \bar{S}$.

Os algoritmos desta abordagem então executam, repetidamente, em qualquer ordem, as *operações básicas* - *descarga* e *rotulamento*. A cada operação básica realizada, um pré-fluxo é transformado em outro.

Na segunda fase, tem-se, basicamente, duas maneiras de realizar este processo de retroceder fluxo. Uma delas é apenas cancelar os excessos dos nós e diminuir o fluxo nos arcos que incidem sobre estes nós, os quais estão causando tal excesso. Uma outra maneira, seria aplicar a primeira fase, de uma forma *reversa*, ou seja, a fonte se tornaria o destino. Então os reservatórios seriam elevados acima dela. Este procedimento seria mais simples que a primeira fase, pois os fluxos nos arcos apenas diminuem.

Um *pré-processamento* é feito para que se tenha nós a analisar no início do procedimento. Uma forma bem simples é utilizar todos os arcos que partem da fonte em suas máximas capacidades (criando assim, nós com excessos positivos).

Dois pontos ainda precisam ser esclarecidos: como um fluxo é enviado de um nó v, mais alto, para outro w, mais baixo; e como o nível de um nó u é elevado (para u com $e(u) > 0$ e $d(u) < n$).

Para o primeiro ponto, seja $C = \langle v - a - b - w \rangle$ um caminho direcionado qualquer (na rede residual) de v para w. É fácil ver que o fluxo $f(v, a)$ é igual ao mínimo entre o que se deseja enviar ($e(v)$) e o que se pode enviar ($c_f(v, a)$), além do que já pode estar passando pelo arco). Se v conseguir enviar todo fluxo para a ($f(v, a) = e(v)$, então a repetirá o mesmo processo para repassar a quantidade de fluxo para b, e assim sucessivamente, até chegar em w. Se em algum nó do caminho, todo excesso não for enviado, então a quantidade, que pode continuar, vai até o destino e o que sobrou fica em excesso neste nó. Portanto, é possível definir uma função DESCARREGAR FLUXO(C), que recebe como parâmetro de entrada um determinado caminho C. A escolha de C é que diferencia os algoritmos que se classificam dentro desta abordagem. Esta função deve ser aplicada a todo nó em que é criado excesso. Assim, a operação DESCARREGAR FLUXO(C) é feita da seguinte maneira:

- $f'(u, v) = f(u, v) + \min\{e(u), c_f(u, v)\}$ para todo arco $uv \in C$, onde $f(u, v)$ é o fluxo que já passa no arco uv;
- se $f(u, v) = c_f(u, v)$, então, $e'(u) = e(u) - c_f(u, v)$, ou seja, o nó continua com excesso, o qual é atualizado.

  Estas duas operações são feitas para todos os arcos que pertencem ao caminho.

No segundo ponto, isto é, para criar possibilidades de envio, deve-se aumentar o nível do nó. Por exemplo, elevar seu nível uma unidade acima do mais baixo de seus vizinhos. Esta operação é repetida enquanto o reservatório ainda está mais baixo que a fonte.

O algoritmo desta abordagem é mostrado na Figura 3.4.

Pode-se afirmar o seguinte teorema.

Teorema 3. *Suponha que o algoritmo termina e todos os rótulos de distância são finitos quando isto acontece. Então o pré-fluxo f é um fluxo máximo, isto é, o algoritmo é correto.*

É fácil ver que este algoritmo termina, pois, na Fase 1, ou uma operação de descarga ou uma elevação de nível é feita em cada um dos nós com excesso positivo. O número de

```
     Algoritmo: MANIPULAÇÃO DE PRÉ-FLUXO
1.   início
2.       f := 0;
3.       PRÉ-PROCESSAMENTO(G);                          % gerar nós
4.       determinar níveis ou alturas dos nós;          % com excessos

5.       % início da fase 1
6.       enquanto a rede residual está conexa faça
7.           seleciona um nó i com excesso;             % i ≠ {s,t}
8.           se existe um caminho C_{i-j} para um nó mais baixo j então
9.               DESCARREGAR FLUXO(C_{i-j});
10.          senão eleve nível do nó i;
11.      fim enquanto

12.      % início da fase 2
13.      para todo v com e(v) > 0 e d(v) ≥ n faça
14.          seleciona um nó v;
15.          determine um caminho C de v para a fonte s;
16.          DESCARREGAR FLUXO(C_{v-s});
17.      fim para
18.      f := e(t);
19.  fim
```

Figura 3.4. *Algoritmo de manipulação de pré-fluxos*

operações de descarga é limitado pelo número de caminhos que levam do nó com excesso até t, ou seja, o mesmo argumento da estratégia de decomposição em caminhos pode se utilizado. Já o número de vezes que é possível elevar um fluxo é limitado pela altura máxima da rede, que é o nível da fonte. Por construção, não é possível elevar a fonte, nem descarregar de nós mais baixos para nós mais altos. Portanto a Fase 1 termina.

Outro importante lema, para a Fase 2, é enunciado a seguir.

Lema 1. *Se f é um pré-fluxo e v é um nó com excesso positivo, então s é alcançável a partir de v no grafo residual G_f.*

Então a Fase 2 também termina, pois todos aqueles nós com excesso , estão neste momento com rótulo maior que n, ou seja, constituem a mesma partição de s no corte.

É também fácil provar que a solução ao final da segunda fase desta estratégia é um fluxo máximo. Se um nó v possui $d(v) > n$ então não existe um caminho deste para o sumidouro,

pela definição de rótulos de distância (Seção 3.2.2). Se todos os nós, com excesso positivo, não puderem alcançar t, então não é mais possível descarregar fluxo. Neste instante o fluxo é máximo. A segunda fase, então, torna o fluxo uma solução viável. Na estratégia anterior, o modo de construção e/ou escolha dos caminhos de s para t é ponto importante a considerar. Neste caso, tem-se muita flexibilidade de execução. Sem dúvida, pode-se observar que a maneira de analisar (escolher) os nós com excesso positivos e construir caminhos a partir destes para nós mais baixos, influencia na boa execução do procedimento. Contudo, a rotina de inicialização (PRÉ-PROCESSAMENTO(G)), as formas de elevar o nível de um nó e retornar fluxo (Fase 2), também podem ser diferentes em cada algoritmo, porém, atendem os mesmos objetivos. O leitor perceberá nos capítulos a seguir que estes pontos - escolha do caminho $s-t$ para abordagem de decomposição e escolha do nó com excesso positivo e construção de um caminho a partir dele, na estratégia de manipulação de pré-fluxos - são determinantes na eficiência dos algoritmos e os diferenciam entre si.

4

Algoritmos de Decomposição de Fluxo em Caminhos

Como visto no capítulo anterior, tem-se duas abordagens básicas utilizadas nos algoritmos de fluxo máximo. Este capítulo é dedicado a apresentar os principais algoritmos que utilizam a estratégia de decomposição de um fluxo máximo, em caminhos que levam fluxo da fonte s para o sumidouro t. Estes são apresentados relevando algumas particularidades técnicas, procurando dar uma visão geral das características de cada um, com a intenção de facilitar, ao leitor, o entendimento de seus funcionamentos. A medida que estes vão sendo mostrados, uma comparação de suas complexidades de pior caso é feita, a fim de mostrar a evolução das melhorias alcançadas. Foram escolhidos apenas os algoritmos que, de alguma forma, melhoraram a eficiência de seus antecessores. Em alguns casos, a ordem cronológica das descobertas não é respeitada. Isto porque optou-se pela ordem de melhoria das complexidades, objetivando uma melhor visão comparativa de eficiência teórica. Este é um trabalho semelhante ao encontrado em [1].

É também apresentado um algoritmo recente(*fluxo bloqueante binário* - *BBF*), considerado, atualmente, o de melhor complexidade de pior caso [3].

4.1. Algoritmo de Rotulamento

O algoritmo de Ford e Fulkerson foi o primeiro criado especificamente para o problema de fluxo máximo [7]. É, em essência, um algoritmo de decomposição de fluxo em caminhos, o qual influenciou vários de seus sucessores. Apesar de o algoritmo possuir uma idéia simples, é preciso saber como identificar e escolher um caminho $s-t$, dentre os vários que podem existir na rede; ou dizer que não existe mais nenhum caminho. Em outras palavras, é preciso determinar um método para resolver as linhas 4 e 5 do Algoritmo da Figura 3.3. Como foi visto, estes são, justamente, os dois passos que diferenciam os algoritmos dentro desta abordagem.

Para resolver a linha 5, o algoritmo acha um caminho orientado qualquer, partindo da fonte s para todos os nós que são alcançáveis a partir dela, utilizando uma busca em profundidade, realizando o procedimento descrito na Seção 2.5.1.

Assim que o sumidouro é visitado, a capacidade do caminho é enviada de s para t, ou seja, verifica-se as capacidades residuais dos arcos que compõem o caminho $s-t$ encontrado e é, então, enviado uma quantidade de fluxo igual a menor destas capacidades, através deste caminho, procedimento este, semelhante as linhas 6 à 8 do algoritmo da Figura 3.3. A rede residual é atualizada e os rótulos são apagados. Então uma nova fase é iniciada. Ou seja, um novo caminho será identificado.

O algoritmo termina quando t não consegue mais ser rotulado, ou seja, a rede está desconexa. Neste ponto não existem mais caminhos $s-t$.

A Figura 4.1 mostra um exemplo da execução deste procedimento. O item (a) mostra a rede com fluxo zero. No início de cada fase, apenas s é rotulado.

Iteração 1. (Fase 1) A partir da fonte s (nó 1), suponha que o algoritmo selecione o nó 3 para visitar. Então este é rotulado e $antecessor(3) = 1$ (item (b)).

Iteração 2. A partir de 3, apenas t pode ser rotulado ($antecessor(4) = 3$) e então um caminho foi encontrado (item (c)).

Iteração 3. O item (d) mostra a rede após o envio de 4 unidades de fluxo através do caminho $1-3-4$. Os rótulos são apagados e a Fase 2 é iniciada (apenas s rotulado).

Iterações 4, 5, 6 e 7. O procedimento anterior é repetido e tem-se a rede depois do envio de 1 unidade de fluxo ao longo do caminho $1-2-3-4$ (item (e)). Os rótulos são apagados e a Fase 3 é iniciada.

Iterações 8, 9, 10 e 11. O item (f) mostra a rede resultante, após o envio de 1 unidade de fluxo pelo caminho $1-2-4$. Os rótulos são apagados e quando a Fase 4 se inicia, t não pode mais ser rotulado, então o algoritmo termina e o valor do fluxo máximo é $|f|=6$.

A corretude deste algoritmo, ou seja, mostrar que ele termina e acha um fluxo máximo, utiliza o mesmo argumento da Seção 3.3.1. Como cada caminho leva, pelo menos, uma unidade de fluxo de s para t e as capacidades dos arcos são inteiras e finitas, o número de caminhos que serão encontrados é também finito. É fácil ver que um caminho de s para t é sempre encontrado caso ele exista, pois o procedimento de busca em profundidade utilizado acha um caminho orientado se este existir. Quando todos os caminhos $s-t$ forem encontrados não há possibilidade de enviar mais fluxo, então ele é máximo. É possível também utilizar para esta prova o Teorema Fluxo-Máximo Corte-Mínimo ([1, 4, 5, 12]).

A complexidade deste algoritmo é facilmente mostrada. A cada fase, no máximo m arcos são percorridos para se achar um caminho $s-t$. Em cada caminho encontrado, no mínimo um arco é *saturado*, ou seja, utilizado em sua capacidade máxima, não podendo mais participar de outro caminho. Supondo que as capacidades dos arcos sejam inteiras e que a maior capacidade de cada arco seja U, então a quantidade máxima de fluxo é nU. Cada caminho $s-t$ pode levar no mínimo uma unidade de fluxo. Portanto, no pior caso, serão necessários nU caminhos, cada um tendo que percorrer m arcos para ser determinado. Isto então causa uma complexidade de $O(nmU)$, que é pseudo-polinomial, pois depende do valor U.

Esta complexidade pode se mostrar muito ruim na prática. Isto ocorre, principalmente,

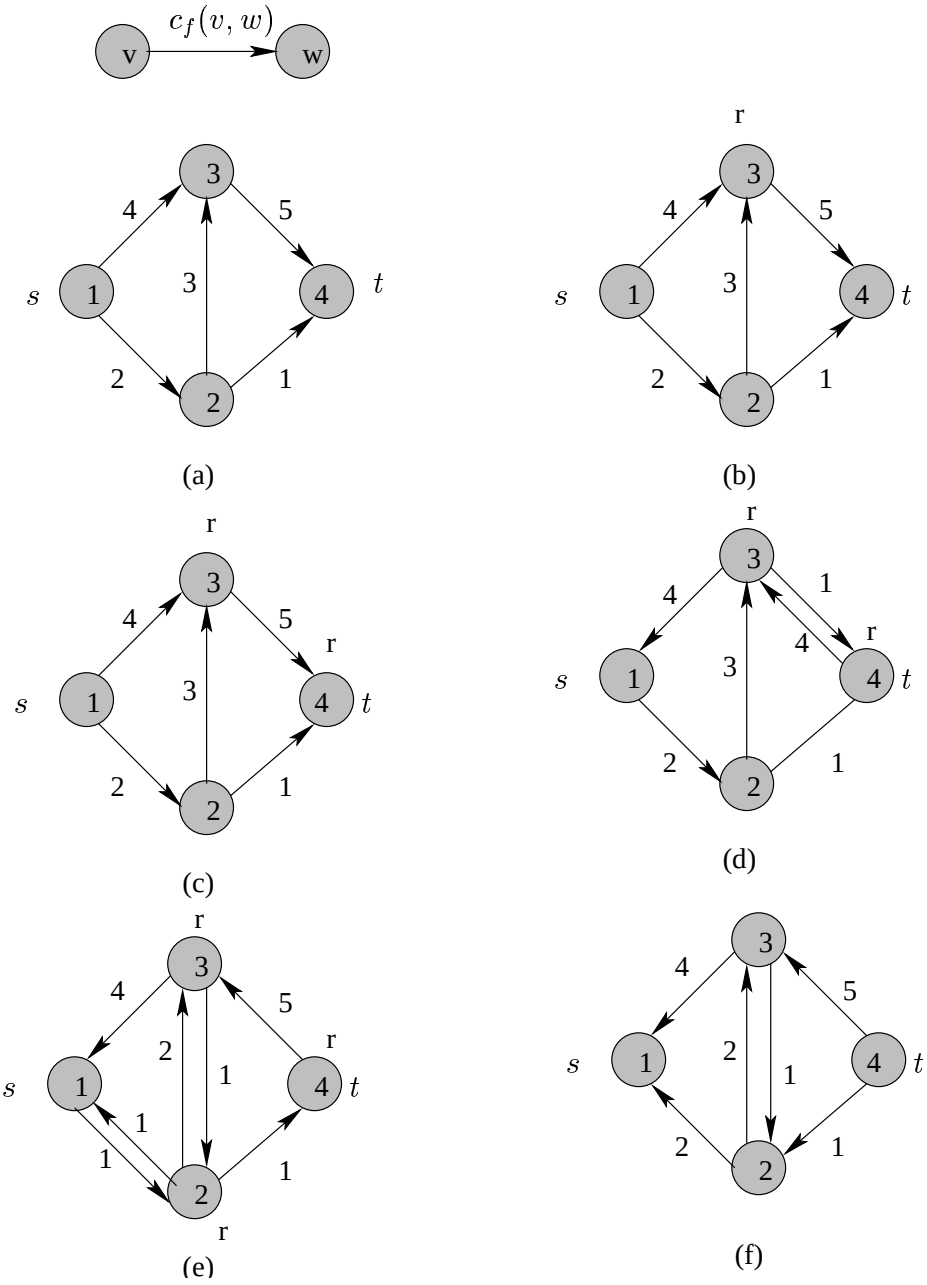


Figura 4.1. *Exemplo de execução do algoritmo de rotulamento (o rótulo r indica quais nós foram visitados).*

quando as capacidades são muito grandes. Outro ponto que também prejudica a eficiência deste algoritmo é a destruição de informação útil para as próximas iterações, pois os rótulos que são apagados no início de cada fase poderiam agilizar os procedimentos das fases seguintes, como será visto adiante, no algoritmo de caminhos mínimos.

A Figura 4.2 mostra um exemplo de como o algoritmo pode funcionar de forma não satisfatória com arcos de capacidades grandes.

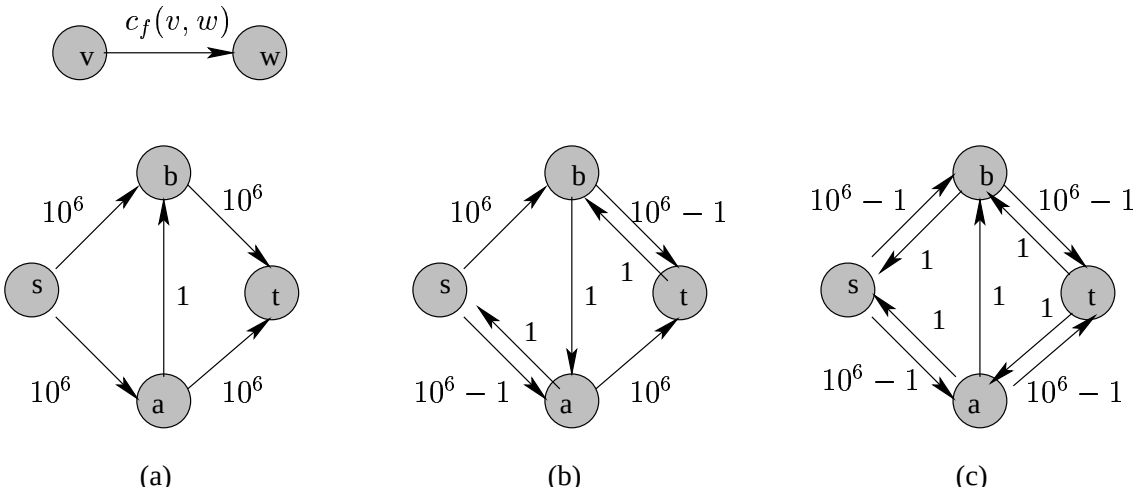


Figura 4.2. *Exemplo do mal funcionamento do algoritmo de rotulamento (o algoritmo pode selecionar os caminhos* $s - a - b - t$ *e* $s - b - a - t$*, alternadamente,* 10^6 *vezes, cada caminho transportando apenas uma unidade de fluxo).*

As seções a seguir mostram outras maneiras utilizadas para construir e escolher um caminho $s - t$, ponto que influencia na complexidade dos algoritmos, como poderá ser comprovado.

4.2. Algoritmo de Escala de Capacidades

Na descrição do algoritmo de rotulamento, verificou-se que um dos problemas que causam um tempo de execução alto é o fato de se enviar fluxo muitas vezes por caminhos $s - t$ de capacidades pequenas. Portanto, uma maneira de melhorar, ou acabar, com esse problema é escolher caminhos com as maiores capacidades possíveis.

Este algoritmo, proposto por Edmonds e Karp (1972) reduz a complexidade de $O(nmU)$ para $O(m^2 \log U)$ [8]. Contudo, ele realiza muitas iterações por ter que achar um caminho de maior capacidade, ou seja, o algoritmo terá que achar todos os caminhos existentes em uma determinada fase e comparar suas capacidades para escolher aquele com a maior delas.

Para evitar tal custo, uma variação deste algoritmo, proposta inicialmente por Gabow, é conhecida como *algoritmo de escala de capacidades*[3, 8]. A idéia é bastante parecida, isto é, ainda deseja-se enviar fluxo por um caminho com grande capacidade, mas não necessariamente a máxima. Para isto, constrói-se uma rede residual *especial*, para que se possa garantir uma quantidade mínima de fluxo passando pelos caminhos que compõem a rede. Isto é, constrói-se a rede residual de forma a garantir uma quantidade mínima de fluxo passando por qualquer caminho $s - t$ escolhido.

Formalmente, seja Δ um *parâmetro* dependente de um dado fluxo f. Define-se *rede* Δ*-residual* $G_f(\Delta)$ uma rede cujos arcos tem capacidade maior ou igual a Δ: $c_f(u, v) \geq \Delta$, para todo $uv \in G_f$. Note que $G_f(1) = G_f$ e $G_f(\Delta)$ é um subgrafo de G_f, conforme exemplificado na Figura 4.3.

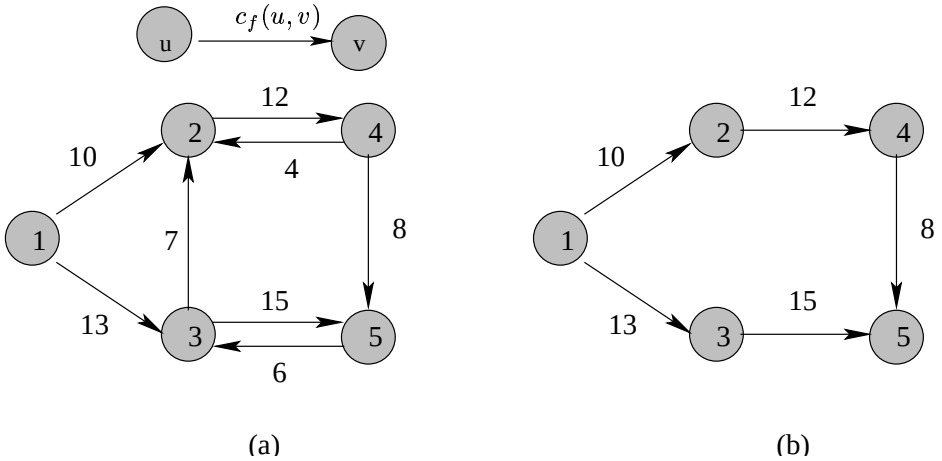


Figura 4.3. *Exemplo de uma rede* Δ*-residual: (a) rede residual* G_f*; (b) rede* Δ*-residual* $G_f(\Delta)$ *para* $\Delta = 8$.

O algoritmo é executado em *fases*, sendo cada uma delas determinada por um valor

constante de Δ. A fase acaba quando não é mais possível encontrar um caminho de s para t na rede Δ-residual. O valor inicial de Δ é $2^{\lfloor \log U \rfloor}$. E este valor é dividido pela metade a cada nova fase até que $\Delta < 1$. Nesse ponto, como as capacidades são inteiras e positivas, não é mais possível formar uma rede residual. Além disso, a rede está desconexa, logo o fluxo encontrado é máximo e o algoritmo termina.

O algoritmo é mostrado na Figura 4.4.

```
     Algoritmo: Escala de Capacidades
1.   início
2.       f := 0
3.       Δ := 2^⌊log U⌋
4.       enquanto Δ ≥ 1 faça
5.           enquanto a rede Δ-residual G_f(Δ) contém um caminho s − t faça   % fase
6.               identifica um caminho p em G_f(Δ);
7.               δ := min{c_f(u, v) : uv ∈ p};
8.               envia δ unidades de fluxo através de p;
9.               atualiza G_f(Δ);
10.          fim enquanto
11.          Δ := Δ/2
12.      fim enquanto
13.  fim
```

Figura 4.4. *Algoritmo de escala de capacidades*

A complexidade deste algoritmo reduz a complexidade do algoritmo de rotulamento de $O(nmU)$ para $O(m^2 \log U)$. Este valor é alcançado porque, a cada fase, a quantidade de fluxo que passa na rede é reduzida à metade. Portanto, são necessários $m \log U$ caminhos $s - t$ (considerando que cada caminho leva uma unidade de fluxo e satura um arco). Contudo, para construir a rede Δ-residual (o que acontece sempre que o valor de Δ é atualizado, ou seja, a cada fase), é preciso percorrer no mínimo m arcos para saber se a capacidade residual de cada um é maior ou igual a Δ.

Esta complexidade é o resultado da aplicação do mesmo método utilizado no algoritmo de rotulamento para a construção e determinação de que não existem mais caminhos (Linhas 5 e 6 da Figura 4.4). A próxima seção mostra uma nova forma de escolher um caminho e esta poderá ser usada neste algoritmo, melhorando um pouco sua complexidade, como

será visto.

4.3. Algoritmo de Menores Caminhos

O algoritmo da seção anterior melhorou o tempo de execução do algoritmo de rotulamento, mas sua complexidade ainda depende do valor U, fato que pode causar um tempo de resposta ruim na prática. Uma complexidade fortemente polinomial é preferível a uma fracamente polinomial (Seção 2.3).

Edmonds e Karp ([1, 5, 6]) imaginaram que enviando fluxo por caminhos mais curtos seria mais rápido. Assim, uma outra maneira de escolher por qual caminho $s - t$ enviar fluxo é através dos menores caminhos, ou seja, aqueles com o menor número de arcos.

Este algoritmo utiliza os conceitos de rótulos de distância, arcos admissíveis e a função de comprimento é unitária para todos os arcos (Ver Capítulo 3).

Inicialmente é realizada uma busca em largura na rede, a partir de t para determinar os rótulos de distância exatos $d(v)$, para todo nó v, onde $d(v)$ é dito *exato* se corresponde ao comprimento de um menor caminho entre v e t em $\vec{G}$. Os caminhos $s - t$, pelos quais o fluxo será enviado, são compostos apenas por arcos admissíveis. O fluxo é então enviado de s para t por um caminho admissível.

Para obter tal caminho, o algoritmo mantém um *caminho admissível parcial*, de s para um nó v qualquer, e executa, iterativamente, uma das duas operações descritas a seguir, sobre o último nó do caminho, chamado de *nó atual ou nó corrente*. Este processo é repetido até que t seja alcançado. Neste momento, a capacidade do caminho $s - t$ construído é enviada e um novo caminho será procurado. Estes passos são executados até que o fluxo seja máximo.

No início de cada passo de determinação de um caminho admissível p, o nó atual inicial é sempre s e $p = \langle s \rangle$.

As duas operações utilizadas no algoritmo são:

- AVANCE(v) - se existe um nó w tal que $d(v) = d(w) + 1$, então adiciona o nó ao caminho admissível (parcial) p, faz $antecessor(w) = v$ e o nó atual passa a ser w. Se $w = t$, então envie fluxo através do caminho $p = \langle s, \ldots, w \rangle$ e faça novamente o nó atual ser s e $p = \langle s \rangle$;

- RETORNE(v) - quando um nó na rede é alcançado, mas não possui arco admissível a partir dele, então o algoritmo utiliza esta função para retornar a um nó da rede que possua tal arco (técnica semelhante a busca em profundidade).

  Matematicamente, se o nó atual v não possui arco admissível, então faz-se $d(v) = \min\{d(w) + 1 : (v, w) \in E_f\}$. A partir daí tem-se três possibilidades:

  1. Se $v = s$ e $d(v) \geq n$, então pare, pois a rede está desconexa e o fluxo é máximo.
  2. Se $v = s$ e $d(v) < n$, então execute AVANCE(v).
  3. Se $v \neq s$, então o nó atual passa a ser o antecessor de v e o nó v é retirado do caminho p.

  Esta operação utiliza a técnica conhecida como *rotulamento*, descrita posteriormente, na Seção 5.2 - algoritmo de Goldberg e Tarjan. De forma geral, a operação RETORNE(v) pode ser vista como uma intenção de criar um arco admissível a partir de v.

A Figura 4.5 mostra o algoritmo e as duas operações acima.

Para que a corretude do algoritmo seja garantida, é preciso recorrer às propriedades referentes a rótulos de distância (Seção 3.2.2). A Propriedade 1 garante que os rótulos de distância servem para construir os caminhos mínimos. Pelo mesmo argumento utilizado para mostrar a corretude do algoritmo de rotulamento, um número finito de caminhos será encontrado. Contudo, a condição de parada do algoritmo é diferente. Então, como garantir que o algoritmo termina e que alcança um fluxo máximo?

A propriedade abaixo garante que os rótulos nunca diminuem.

Propriedade 5. *Para cada nó v, o rótulo de distância $d(v)$ nunca diminui.*

Algoritmo: MENORES CAMINHOS $s - t$
1. **início**
2. $f := 0$;
3. calcule os rótulos de distância exatos $d(v)$, de todo v para t; % busca em largura
4. $p := \langle s \rangle$; % caminho admissível
5. $v := s$; % nó atual
6. **enquanto** $d(s) < n$ **faça** % Propriedade 3
7. **se** existe w tal que $d(v) = d(w) + 1$ **então**
8. AVANCE(v);
9. **se** $v = t$ **então**
10. envie fluxo através do caminho % quantidade de fluxo igual
11. p encontrado; % a $\delta = \min_{(v,w)\in p}\{c_f(v,w)\}$
12. $p := \langle s \rangle$;
13. $v := s$;
14. **fim se**
15. **senão** RETORNE(v); % não existe arco admissível
16. **fim enquanto**
17. **fim**

Procedimento: AVANCE(v)
1. **início**
2. $p := p \cup w$; % adiciona o nó w ao caminho admissível p
3. $v := w$; % nó atual passa a ser o próximo nó
4. $antecessor(w) := v$ % variável para construir o caminho
5. **fim**

Procedimento: RETORNE(v)
1. **início**
2. $d(v) := \min\{d(w) + 1 : (v,w) \in E_f\}$ % rotulamento
3. **se** $v \neq s$ **então** % s não possui antecessor
4. $v := antecessor(v)$ % o nó atual passa a ser o nó antecessor a v
5. **fim**

Figura 4.5. *Algoritmo de menores caminhos $s - t$ e suas operações*

Uma aplicação de uma operação de rotulamento em v - dentro do procedimento RETORNE(v) - aumenta $d(v)$) e quando $d(s) \geq n$, a rede está desconexa, pela Propriedade 3. Logo, pelo Teorema 1, pode-se dizer que, neste instante, o fluxo na rede é máximo.

Vale observar que a técnica utilizada neste algoritmo é bastante parecida com o algoritmo de rotulamento. Contudo, este restringe o número de arcos (apenas os admissíveis) e não "apaga" os rótulos de distância. Pelas propriedades de rótulos válidos, vistos na Seção 3.2.2, as distâncias nunca diminuem e quando $d(s) \geq n$, a rede está desconexa. Portanto, é fácil ver que este algoritmo acha um caminho $s-t$ com menos iterações que o algoritmo de rotulamento. E como os caminhos escolhidos são sempre menores (com menor número de arcos), é mais rápido enviar fluxo de s para t.

Assim como todos os algoritmos de decomposição de fluxo em caminhos, cada envio de fluxo por um caminho encontrado satura, no mínimo, um arco deste caminho. Portanto, após m caminhos, no pior caso, satura-se todos os arcos, não sendo mais possível enviar fluxo. Como o algoritmo executa enquanto $d(s) < n$ e a distância de s até t aumenta de uma unidade (pela Propriedade 5), no mínimo, a cada arco saturado, então depois de no máximo n caminhos encontrados, $d(s) \geq n$. Para achar um caminho mínimo na rede, percorre-se, no máximo m arcos. Desta forma, a complexidade deste algoritmo é $O(nm^2)$, a qual não depende mais do valor de U.

A utilização deste método no algoritmo de escala de capacidades, ou seja, na rede Δ-residual, enviar fluxo pelos menores caminhos (modificação do algoritmo de Gabow, proposta por Ahuja e Orlin [1]), reduz a complexidade do segundo algoritmo de $O(m^2 \log U)$ para $O(nm \log U)$.

4.4. Algoritmo de Rede em Camadas

Este algoritmo, desenvolvido por Dinitz, envia fluxo através de caminhos $s-t$ em uma rede em camadas [8, 21, 6]. A cada *fase* do algoritmo é construída uma rede em camadas e um fluxo bloqueante é encontrado.

Apesar de sua complexidade de pior caso se assemelhar com o algoritmo de menores caminhos (já que pode ser visto como um caso particular deste), na prática ele funciona de forma mais rápida.

Pela própria definição de rede de em camadas, a seguinte propriedade se verifica: se v e w pertencem a mesma camada, então $d(v) = d(w)$. Assim, todos os caminhos que levam de s para t possuem o mesmo comprimento que é mínimo. Dado um fluxo f, é possível construir uma rede em camadas $\hat{G}_f$, a partir de uma rede residual G_f, utilizando rótulos de distância, da seguinte forma:

1. determine as distâncias exatas $d(v)$, de todo nó v até t, na rede residual G_f;
2. a rede em camadas contém, então, apenas os arcos admissíveis;
3. os nós que não participam de nenhum caminho da fonte até o sumidouro são retirados da rede.

Quando um fluxo bloqueante f é encontrado na rede $\hat{G}_f$, o algoritmo recalcula os rótulos de distância exatos na rede residual G_f, constrói uma nova rede em camadas e executa todo o processo novamente. O algoritmo termina quando, formada a rede em camadas, o sumidouro não é mais alcançável a partir da fonte.

O algoritmo é mostrado na Figura 4.6. O procedimento Fluxo Bloqueante é muito parecido com o algoritmo apresentado na Seção 3.3.1, aplicado a uma rede em camadas. Contudo, esta não é a única maneira de se encontrar um fluxo bloqueante em uma rede acíclica. A rotina mais rápida para o problema pode ser encontrada em [22], a qual acha um fluxo deste tipo em $O(m \log n^2/m)$.

A idéia de Dinitz funciona por razões semelhantes às vistas para o algoritmo da seção anterior.

Cada fase deste algoritmo é determinada pela construção de uma rede em camadas, a partir de uma rede residual, e do cálculo de um fluxo bloqueante. A cada envio de fluxo por um caminho na rede em camadas, um arco deve ser saturado. Contudo, neste

```
     Algoritmo: REDE EM CAMADAS
1.   início
2.       f := 0;
3.       f(u,v) := 0, para todo arco uv ∈ E;
4.       calcule as distâncias exatas d(v), de todo v para t, em G_f;      % busca em largura
5.       construa a rede em camadas Ĝ_f;
6.       enquanto Ĝ_f contém um caminho s − t faça
7.           f := f+FLUXO BLOQUEANTE(Ĝ_f);
8.           calcule as distâncias exatas d(v), de todo v para t, em G_f;
9.           construa a rede em camadas Ĝ_f;
10.      fim enquanto
11.  fim
```

```
     Procedimento: FLUXO BLOQUEANTE(Ĝ_f)
1.   início
2.       x := 0;
3.       enquanto Ĝ_f contém um caminho s − t faça
4.           identifica um caminho p em Ĝ_f;
5.           δ := min{c_f(u,v) : ∀uv ∈ p};
6.           x := x + δ;                          % soma de caminhos - Seção 3.3.1
7.           f(u,v) := f(u,v) + δ, ∀uv ∈ p;       % soma de arcos
8.       fim enquanto
9.       retorne x;
10.  fim
```

Figura 4.6. *Algoritmo de rede em camadas e a rotina de fluxo bloqueante*

caso, só é necessário percorrer $n-1$ arcos para achar um caminho $s-t$ nesta rede, pois todos os caminhos tem o mesmo comprimento e arcos para trás não são adicionados a rede. Portanto, em cada fase, $O(nm)$ caminhos são necessários para se calcular um fluxo bloqueante. Como o algoritmo executa até que a rede em camadas construída não possua mais caminhos $s-t$, então no máximo n redes em camadas são construídas até que $d(s) \geq n$, pela Propriedade 5, causando uma complexidade de $O(n^2m)$.

Este algoritmo pode ser visto como uma modificação do algoritmo de menores caminhos, a saber:

1. Na operação RETORNE(v), não haverá mais rotulamento, pois não é possível criar arcos para trás pela própria definição de rede em camadas. O nó que não possuir mais arcos admissíveis partindo dele será *bloqueado*;

2. A definição de arco admissível passa então a ser, no algoritmo adaptado, um arco vw, onde $d(v) = d(w) + 1$, $c_f(v,w) > 0$ e w não está bloqueado;

3. Quando a fonte se tornar bloqueada, os rótulos de distância são então recalculados.

A Figura 4.7 mostra tais modificações no algoritmo de menores caminhos para que este execute como um de fluxo bloqueante. Por este motivo, os algoritmos de fluxo bloqueante podem ser classificados como algoritmos de decomposição de fluxo em caminhos. Como o algoritmo de Dinitz pode ser visto como uma modificação do algoritmo de Edmonds e Karp, que é, naturalmente, um algoritmo de decomposição de fluxo em caminhos, então, optou-se por também classificá-lo dentro desta abordagem.

4.5. Algoritmo de Fluxo Bloqueante Binário

Recentemente, Goldberg e Rao desenvolveram um algoritmo utilizando a mesma técnica de fluxo bloqueante do algoritmo de Dinitz. Mas calculado em uma rede diferente da rede de menores caminhos [3, 23]. Este algoritmo possui uma complexidade de $O(m\Lambda \log(n^2/m) \log U)$,

```
     Algoritmo: MENORES CAMINHOS |= REDE EM CAMADAS
1.   início
2.       f := 0;
3.       ⋮
4.       enquanto d(s) < n faça
5.           se s = bloqueado então
6.               calcule as distâncias corretas d(v), de todo v para t, em G_f;   % busca em largura
7.           se existe w, tal que d(v) = d(w) + 1 e w ≠ bloqueado então
8.               AVANCE(v);
9.               se v = t então
10.                  envie fluxo através do caminho p;
11.                  p := ⟨s⟩;
12.                  v := s;
13.              fim se
14.          senão RETORNE(v);
15.      fim enquanto
16.  fim
```

```
     Procedimento: AVANCE(v)
1.   início
2.       p := p ∪ w;
3.       v := w;
4.       antecessor(w) := v
5.   fim
```

```
     Procedimento: RETORNE(v)
1.   início
2.       v := bloqueado
3.       se v ≠ s então
4.           v := antecessor(v)
5.   fim
```

Figura 4.7. *Modificação do algoritmo de menores caminhos para funcionar como um de rede em camadas*

onde $\Lambda = \min\{m^{\frac{1}{2}}, n^{\frac{2}{3}}\}$. Esta complexidade de pior caso é realmente a menor entre todos os algoritmos conhecidos, até hoje, para resolver o problema de fluxo máximo. Contudo, como será visto Capítulo 6), ele não tem alcançado resultados tão bons na prática.

A idéia geral deste algoritmo é calcular um fluxo bloqueante em uma rede acíclica, construída de forma diferente da rede em camadas, e depois repassar este fluxo para a rede original. As diferenças para o algoritmo de Dinitz residem na rede acíclica sobre a qual é calculada o fluxo bloqueante e na forma como este fluxo é repassado à rede original. É importante observar que o fluxo bloqueante calculado na rede modificada pode não corresponder a um fluxo na rede original. Diferentemente do algoritmo original de Dinitz, o fluxo bloqueante não pode ser simplesmente adicionado ao fluxo anterior, arco por arco. É preciso portanto, um tratamento no fluxo antes e depois de calcular o fluxo bloqueante.

4.5.1. Construção da rede acíclica

Como no algoritmo de Dinitz, é necessário construir uma rede de menores caminhos para que depois seja calculado um fluxo bloqueante nesta. A rede acíclica é construída a partir dos rótulos de distância, mas utiliza uma função de comprimento *binária*, ou seja, os arcos recebem comprimento 0 ou 1, dependendo de um valor τ, chamado que *grau de compressão* da rede, cujo significado será discutido mais adiante.

Então, a definição da função de comprimento binária l, para todo arco vw, é:

- $l(v, w) = 0$ se $c_f(v, w) \geq 3\tau$ ou se $2\tau \leq c_f(v, w) < 3\tau$ e $c_f(w, v) \geq 3\tau$; e
- $l(v, w) = 1$, caso contrário.

A idéia de uma função de comprimento não unitária já havia sido sugerida por Edmonds e Karp [3], mas antes do algoritmo de Goldberg e Rao, nenhum trabalho tinha apresentado melhorias significativas de complexidade.

Depois de definida a função de comprimento dos arcos, são calculados os rótulos de distância. De acordo com a definição de arco admissível dada na Seção 3.2.2, substituindo

o valor de l, um arco é admissível, para este algoritmo, se ele satisfaz uma das condições abaixo:

$$d(v) > d(w) \tag{4.1}$$

$$d(v) = d(w) \quad e \quad l(v,w) = 0 \tag{4.2}$$

A Propriedade 4 vale também para uma função de comprimento binária [3].

4.5.2. Descompressão do fluxo

Depois de construída a rede acíclica, um fluxo bloqueante é calculado, isto modifica o fluxo passando em alguns arcos da rede original. Os arcos de comprimento 0, que parecem essenciais para que seja alcançada a complexidade do algoritmo, causam o problema de nós unificados e então este fluxo precisa ser repassado dentro de tais nós.

Um ponto importante a salientar é o fato de no algoritmo de Dinitz, o que pode causar uma complexidade pior é o fato de existirem arcos com grandes capacidades entre duas camadas diferentes. Portanto este é o ponto que tenta ser evitado no algoritmo de Goldberg e Rao, colocando-se um arco com grande capacidade "dentro" de uma mesma camada. Mas depois de calculado o fluxo bloqueante na rede acíclica é necessário repassar este fluxo na rede original.

Estas operações serão descritas com mais detalhes na Seção 4.5.5.

4.5.3. Grau de compressão

Existe uma estimativa do valor de *fluxo residual* (limite superior da quantidade de fluxo que ainda pode ser enviada pela rede) e a medida que esta estimativa diminui, o parâmetro τ é recalculado e, consequentemente, o valor da função de comprimento e dos rótulos de distância são atualizados. As redes residual e a acíclica são reconstruídas. Neste ponto, calcula-se então um fluxo bloqueante.

Deseja-se enviar, no máximo, τ unidades de fluxo a cada cálculo de fluxo bloqueante, para que a distribuição desta quantidade de fluxo na rede original não se torne uma operação custosa.

Devido ao uso de uma função de comprimento binária, alguns nós são "identificados", ocasionando uma rede diferente da rede residual original. Neste contexto, considera-se *nós* os do grafo original de entrada e *nós contraídos* o conjunto de nós que foram "identificados". É fácil ver que a rede onde será calculada o fluxo bloqueante pode ter ambos os tipos de nós. Contudo, nesta rotina, os dois tipos são tratados da mesma forma.

É importante observar que quanto menor o valor de τ, mais a rede será comprimida, contudo, pouco fluxo passará e cada fase. Caso contrário, ou seja, quanto maior o τ, a execução deste algoritmo e a rede formada ficam mais parecidos com o algoritmo de rede em camadas da seção anterior.

A diferença com o algoritmo de Dinitz está na construção da rede acíclica C', que depende do valor de τ, que por sua vez, depende do valor de fluxo residual $\hat{F}$. A constante de proporcionalidade escolhida, entre estes valores (τ e $\hat{F}$), é α/Λ, onde α é uma constante positiva.

4.5.4. Formulação genérica

Como no algoritmo de Dinitz, o algoritmo BBF (no original inglês - *Binary Blocking Flow*) opera em *iterações* , em cada uma é calculado um fluxo bloqueante g', em uma rede auxiliar acíclica C'; e g' é atualizado na rede original. O valor de τ é mantido constante até que $\hat{F}$ tenha diminuído β vezes, onde β é outra constante positiva. As iterações executados sob um mesmo valor de τ constituem uma *fase*.

Assim, uma fase diminui o valor total de fluxo que ainda pode ser enviado, $\hat{F}$, para $\alpha\hat{F}/\Lambda$. E este valor é alcançado, após β iterações, cada um delas levando τ unidades de fluxo.s

Uma formulação genérica do algoritmo BBF pode ser encontrada na Figura 4.8 [23].

```
     Algoritmo: BBF [Genérico]
1.   início
2.       f := 0;
3.       enquanto Estimativa Fluxo(G_f) ≥ 1 faça        % início de uma fase
4.           F̂ := Estimativa Fluxo(G_f);
5.           τ := ⌈αF̂/Λ⌉;
6.           enquanto Estimativa Fluxo(G_f) ≥ βF̂ faça   % τ constante
7.               Nova Residual(f, τ);                    % cálculo de l e d em G_f
8.               C' := Rede Acíclica(G_f, τ);            % pré-processamento
9.               g' := Fluxo Bloqueante(C');
10.              f := f+Fluxo Transferido(g', C', G);    % pós-processamento
11.          fim enquanto
12.      fim enquanto
13.  fim
```

Figura 4.8. *Algoritmo BBF - Descrição Genérica*

Esta formulação é dita genérica, pois este algoritmo se mostra bastante flexível em vários aspectos:

1. a escolha da estimativa inicial do fluxo residual, tem influência na eficiência do algoritmo, ou seja, quanto mais próximo do verdadeiro valor de fluxo máximo for a estimativa do limite superior, menos iterações serão necessárias até que a estimativa diminua;

2. os valores das constantes α e β servem como uma *parametrização* do algoritmo, ou seja, é possível melhorar o tempo de execução do algoritmo na prática, dependendo do tipo da instância, variando os valores destas constantes. Como já citado, a escolha do valor de τ deve tentar manter o equilíbrio de tempo gasto com as operações de compressão e descompressão da rede (passos de pré e pós-processamento do algoritmo, respectivamente).

3. a forma como os ciclos induzidos pelos arcos de comprimento 0 são tratados (procedimento Rede Acíclica(G_f, τ)), também influenciam na execução do algoritmo. É possível encontrar em [3] e [23] duas maneiras diferentes para formar as redes acíclicas;

4. como o fluxo bloqueante, calculado na rede acíclica, é depois distribuído para a rede residual original. Este ponto tem relação com o item anterior e refere-se ao procedimento FLUXO TRANSFERIDO(g', C', G).

As seções a seguir se dedicam a descrever as operações principais do algoritmo com algum nível de formalismo. A seguir é também mostrada sua complexidade.

4.5.5. Operações

As principais operações do algoritmo da Figura 4.8 são:

- ESTIMATIVA FLUXO(G_f)

  A escolha do valor da estimativa de fluxo residual é bem simples. [3] sugere como valor inicial $\hat{F} = \sum_{s,w} c(s,w)$ ou $\hat{F} = nU$ (o primeiro é mais recomendado). Contudo, é necessário saber se esta estimativa diminuiu, para que seja iniciada uma nova fase.

  Neste ponto, são apresentadas duas possibilidades. A primeira utiliza o conceito de *volume de arco*, ou seja, $vol(a) = c_f(a)l(a)$, para todo arco a na rede residual. É fácil ver que $\sum_{\forall a \in G_f}(c_f(a)l(a))/d(s)$ é um limite superior para o valor de fluxo residual. Portanto, a estimativa de fluxo pode ser atualizada sempre que

  $$\frac{\sum_{\forall a \in G_f}(c_f(a)l(a))}{d(s)} \leq \beta\hat{F}$$

  Uma outra maneira, é manter um corte (S,T) tal que $c_f(S,T) = \hat{F}$. Quando $c_f(S,T) \leq \beta\hat{F}$, a fase atual termina e τ é atualizado. Para esta escolha, utiliza-se o conceito de *cortes canônicos* descrito na Seção 2.1.3. Assim, dentre todos os cortes canônicos (S_k, T_k), para $k = 1, \ldots, d(s)$, escolhe-se aquele de menor capacidade residual $(\bar{S}, \bar{T})$ para atualizar o valor da estimativa de fluxo. No início de cada iteração, é então verificado se o valor da estimativa se tornou menor $(c_f(\bar{S}, \bar{T}) \leq \beta\hat{F})$. Caso isto se verifique, uma nova fase é iniciada, com um novo valor de τ. Esta maneira é mais interessante, pois, a cada fase, a diferença entre a soluções dos problemas primal (fluxo máximo) e dual (corte mínimo) é diminuída.

- NOVA RESIDUAL(f, τ)

  Esta rotina constrói uma nova rede residual G_f a partir do valor de τ. Apesar deste valor ser constante, este procedimento é repetido entre as iterações e não entre fases, pois quando um novo fluxo bloqueante é calculado e distribuído na rede, os fluxos passando nos arcos se modificam, portanto, as funções de comprimento para os arcos podem mudar (já que esta é uma medida dependente da capacidade residual do arco com relação a τ). Os rótulos de distância, por sua vez, também serão atualizados, refletindo na escolha dos arcos admissíveis.

- REDE ACÍCLICA(G_f, τ)

  Para se calcular um fluxo bloqueante (próximo procedimento) é necessário que se tenha uma rede sem ciclos, pois este deve ser calculado em uma rede de menores caminhos.

  Como já citado, o grafo (rede residual) formado pelos arcos admissíveis podem conter ciclos induzidos pelos arcos de comprimento 0. Logo, estes devem ser tratados de forma a *desaparecer* enquanto um fluxo bloqueante é calculado. A seguir, serão apresentadas duas formas de construir tal rede.

  Um forma bem simples de tratar os ciclos é mostrada em [3]. A idéia é *contrair* as componentes fortemente conexas induzidas pelo arcos de comprimento 0. Ou seja, elas são substituídas por um único *nó*.

  Uma segunda maneira é proposta em [23]. Para esta, sejam as definições a seguir. Um arco $(u, v) \in G_f$ é dito *para baixo ou para frente* se $d(u) > d(v)$, *para cima ou para trás* se $d(u) < d(v)$ e *horizontal* se $d(u) = d(v)$. Um *núcleo* C de G_f é um subgrafo de G_f, contendo todos os nós, todos os arcos para baixo e um subconjunto dos arcos horizontais. Tal subconjunto contêm todos os arcos de comprimento 0 e todo arco a com $c_f(a) > 2\tau$, cujo arco reverso já pertence ao núcleo. Além disso, nenhum arco, de algum ciclo do núcleo, possui capacidade menor que 2τ.

  Como citado anteriormente, as componentes fortemente conexas na primeira es-

tratégia, são contraídas em um único nó. Aqui, contudo, cada componente fortemente conexa SCC é substituída por uma *componente acíclica*. Para cada nó v em uma SCC, determina-se (não necessariamente distintos) nós de entrada e nós de saída ($no\_entrada(v)$ e $no\_saida(v)$, respectivamente). A *rede acíclica* C' é portanto formada pela união de todas as componentes acíclicas, unidas por arcos do tipo ($no\_saida(u)$,$no\_entrada(v)$), onde (u, v) é um arco do núcleo de G_f e u e v pertencem a diferentes SCC. O arco ($no\_saida(u)$,$no\_entrada(v)$) possui a mesma capacidade que o arco (u, v) do núcleo.

A seguinte propriedade deve ser atendida neste procedimento:

Propriedade 6. *Para cada componente fortemente conexa SCC do núcleo e para todos os nós u e v de SCC, existe um caminho na componente acíclica de u para v de capacidade pelo menos τ.*

Esta propriedade é facilmente verificada pela própria construção do núcleo e para garantir a correta distribuição do fluxo bloqueante calculado. Para isto, é necessário também que cada arco da componente possua uma capacidade de pelo menos 2τ. Assim, como o algoritmo limita a quantidade de fluxo passando pela rede, e por conseguinte em cada nó contraído, então fica fácil rotear o fluxo dentro destes se as componentes obedecem a propriedade acima.

- Fluxo Bloqueante(C')

  Neste momento, a rede acíclica C' é uma rede de menores caminhos, contendo nós e nós contraídos. Contudo, para o procedimento que calculará o fluxo bloqueante isto é transparente, então uma rotina qualquer para o problema pode ser usada.

- Fluxo Transferido(g', C', G)

  No passo anterior, um fluxo bloqueante foi calculado, isto modifica o fluxo passando em alguns arcos da rede original. A restrição de conservação de fluxo é atendida nos nós da rede contraída, mas pode não ocorrer o mesmo nos nós da rede original, em outras palavras, é preciso rotear (distribuir) o fluxo dentro das componentes conexas.

Considere uma componente que não contem nem a fonte nem o sumidouro. O que entra por esta componente menos o que sai, deve ser igual a zero, já que esta pertence a C' onde foi calculado um fluxo bloqueante. Se o valor de $g' > \tau$, então o primeiro passo desta rotina é enviar $g' - \tau$ unidades de fluxo de volta para s. Isto é feito, colocando excesso no sumidouro igual a $g' - \tau$ unidades e processando uma busca, nos nós da rede ainda contraída, do sumidouro para a fonte, reduzindo fluxo nos arcos para eliminar o excesso.

O restante será então distribuído na forma a seguir:

1. escolhe-se um nó de cada componente para ser a *raiz*;
2. forma-se uma árvore de entrada e uma árvore de saída a partir da raiz;
3. os excessos positivos são distribuídos pela árvore de entrada e os negativos, pela árvore de saída

O fluxo resultante não viola nenhuma capacidade, por causa da Propriedade 6.

Se a componente possui a fonte ou o sumidouro, um processo semelhante é executado, escolhendo estes nós como raízes.

4.5.6. Corretude e complexidade

A prova de que o algoritmo BBF funciona corretamente, baseia-se na propriedade que os rótulos de distância nunca diminuem. Para os lemas a seguir, considera-se que uma iteração inicia com um fluxo f e termina com um fluxo $f + g$, onde g é o resultado do procedimento FLUXO TRANSFERIDO(g', C', G). As provas dos lemas a seguir podem ser encontradas em [3, 23].

Lema 2. *Para todo $u \in V$, vale $d_{f+g}(u) \geq d_f(u)$. Além disso, seja π um caminho mínimo de u para t na rede residual G_{f+g}. Se $d_{f+g}(u) = d_f(u) < \infty$, então, para todo nó v em π, vale $d_{f+g}(v) = d_f(v)$.*

Este lema pode ser provado pela própria escolha dos arcos admissíveis para este algoritmo. A seguir, garante-se que se a quantidade de fluxo que passa em uma determinada iteração é no máximo τ, então, pela construção da rede, é fácil distribuir o fluxo na rede original.

Lema 3. *Seja g um fluxo bloqueante em C'. Se $|g| < \tau$ então g é um fluxo bloqueante em G_f.*

A complexidade deste algoritmo é determinada pelo número de fases, pelo número de iterações em cada fase e pelos custos de cada iteraçao.

A cada iteração, dominada pela computação de um fluxo bloqueante, ou o valor do fluxo aumenta de no mínimo $2\hat{F}/\Lambda$ ou a distância da fonte ao sumidouro aumenta de pelo menos uma unidade. Pode-se provar que quando a distância excede $c\Lambda$, para uma dada constante c, $\hat{F}$ deve diminuir. Depois de $O(\Lambda)$ iterações, $\hat{F}$ deve diminuir, ou porque o fluxo aumentou de pelo menos $\beta\hat{F}$ ou porque a distância entre a fonte e o sumidouro excedeu $c\Lambda$.

Assim, a cada fase, $\beta\hat{F}$ quantidades de fluxo são enviadas (esta quantidade é dividida em *pedaços* de tamanho no máximo τ) e, como os rótulos de distância apenas crescem, em um momento, s não alcançará mais t. Neste momento o fluxo é máximo.

Utilizando cortes canônicos na atualização da estimativa de fluxo, é possível realizá-lo em tempo $O(m)$. A nova rede residual, os comprimentos e rótulos podem ser calculados também em $O(m)$. A contração e descontração da rede acíclica podem ser feitas em $O(n)$ passos.

Como já citado, a rotina mais rápida, proposta por Goldberg e Tarjan ([22]), conhecida para calcular um fluxo bloqueante em uma rede acíclica executa em $O(m \log(n^2/m))$.

A estimativa inicial de $\hat{F}$ é limitada superiormente por nU inicialmente. A partir da função de comprimento binária e das definições de fase e cortes canônicos, são necessárias $O(\Lambda \log U)$ fases para que a estimativa inicial diminua até 1. Como em cada

fase são calculados fluxos bloqueantes, pode-se ver que a complexidade deste algoritmo é $O(m\Lambda \log(n^2/m) \log U)$.

4.5.7. Sumário

Este algoritmo pode ser classificado como um de decomposição de fluxo em caminhos pelas seguintes equivalências com o algoritmo da Seção 3.3.1 (além de ser uma generalização do algoritmo de Dinitz):

1. A condição de executar enquanto a estimativa de fluxo for maior que 1, pode ser vista como equivalente a ainda existir um caminho $s - t$ na rede;

2. As iterações se resumem a identificar um caminho e enviar fluxo através deste. Ou seja, os procedimentos Rede Acíclica, Fluxo Bloqueante e Fluxo Transferido, poderiam ser substituídos por uma única rotina de cálculo de um fluxo bloqueante, caso a função de comprimento fosse unitária para todos os arcos. Então, os caminhos são construídos de forma especial, mas esta é justamente a característica que diferencia os algoritmos desta abordagem.

A Tabela 4.1 mostra um resumo das formas de construir e escolher caminhos nos algoritmos apresentados neste capítulo. As maneiras de finalização do algoritmo, ou seja, determinar que não existem mais caminhos da fonte para o sumidouro são equivalentes entre si, como já mostrado anteriormente.

Tabela 4.1. *Resumo das formas de escolha de caminhos $s-t$ dos algoritmos de decomposição de fluxo em caminhos*

Algoritmo	**Decomposição**	**Terminação**
Rotulamento	Qualquer caminho	t não é mais alcançado a partir de s
Escala de Capacidades	Menores caminhos de grandes capacidades primeiro	Não existe mais unidade de fluxo a ser enviada ou $d(s) \geq n$
Menores Caminhos	Qualquer caminho $s-t$ de menor comprimento	$d(s) > n$
Rede em Camadas	Todos os caminhos $s-t$ de menor comprimento primeiro	$d(s) > n$ ou t não é mais alcançado a partir de s
BBF	Todos os caminhos *contraídos* de menor comprimento primeiro	Não existe mais unidade de fluxo a ser enviada ou $d(s) \geq n$

5

Algoritmos de Manipulação de Pré-fluxos

Os algoritmos de decomposição em caminhos encontram um caminho de s para t e só então enviam fluxo através deste caminho. Esta estratégia, contudo, pode gerar um ponto fraco nestes algoritmos como citado na Seção 3.3.2. Os algoritmos classificados dentro da abordagem de manipulação de pré-fluxo diferem dos primeiros no sentido de enviar fluxo por um caminho entre algum nó com excesso v e um predecessor de t. Isso pode economizar tempo de processamento no caso de vários caminhos de nós com excesso para t se interceptarem em algum nó v. A Figura 5.1 ilustra este caso. Primeiro, todos os nós com excesso u_i descarregam fluxo até v, e depois v descarrega até t.

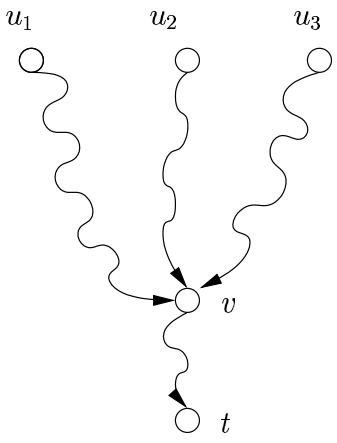


Figura 5.1. *Exemplo da idéia do funcionamento dos algoritmos de manipulação de pré-fluxo*

Por este motivo, a restrição de equilíbrio (ou conservação de fluxo) é relaxada. A idéia de manipulação de pré-fluxo surgiu com o algoritmo de Karzanov [6, 21]. Utilizou uma modificação do algoritmo de rede em camadas, tentando enviar o máximo de fluxo possível de uma camada para a outra, a cada fase; e o que não pudesse mais ser enviado retornaria à camada anterior. Assim, este não seria mais um algoritmo de decomposição de fluxo em caminhos. A idéia de Karzanov ficou também conhecida como *algoritmo de fluxo bloqueante em ondas*.

Anos depois, Goldberg e Tarjan propuseram um novo algoritmo de manipulação de pré-fluxo [2]. Variações deste algoritmo, por muito tempo mostraram ser os melhores e mais robustos na prática, assim como de fácil modelagem em programação paralela ([24, 25]).

Mais recentemente, um algoritmo dentro desta abordagem, desenvolvido por Hochbaum [26], com complexidade de pior caso parecida com a do anterior, mas com rotinas mais complicadas.

Estes, e principalmente variações dos dois últimos, encontram-se entre os algoritmos mais rápidos, na prática, para resolver o problema de fluxo máximo.

5.1. Algoritmo de Pré-fluxo em Camadas

Este algoritmo trabalha com rede em camadas, como o algoritmo de Dinitz, mas não decompõe a rede em caminhos. O algoritmo de Karzanov acha um fluxo bloqueante a cada fase, mas utilizando o conceito de pré-fluxo e não de caminhos [6]. A idéia de seu funcionamento é descrita a seguir.

Inicialmente, a cada fase, é construída a rede em camadas correspondente à rede residual e saturam-se todos os arcos partindo da fonte para criar nós ativos(pré-processamento).

O algoritmo processa todos os nós de uma camada (em qualquer ordem) utilizando dois procedimentos. O primeiro, descarrega os excessos de todos os nós de uma camada para a próxima (mais perto do sumidouro - ordem decrescente das camadas) e continua pelas camadas subsequentes até atingir o sumidouro ou não ter mais como enviar fluxo. Então,

neste algoritmo, se $e(v) > 0$, toma-se um vizinho de v na camada seguinte como candidato a predecessor de t.

O segundo, retorna os excessos dos nós que não puderam atingir a camada vizinha. Isto é feito apenas cancelando o fluxo no arco que chega ao nó com excesso. Quando isto acontece, só volta uma camada para trás, diferente do primeiro procedimento que empurra fluxo para frente até não poder mais. Quando uma operação de retorno de fluxo ocorre, o nó é bloqueado e não poderá mais receber fluxo através da operação de envio (descarga). O fluxo máximo é encontrado quando não existem mais nós ativos, ou seja, a rede em camadas foi construída, mas t não é mais alcançável a partir de s.

Comparando este algoritmo com o apresentado na Seção 3.3.2, vê-se que:

- A operação de pré-processamento (gerar nós com excesso) é feita sempre que uma nova rede em camadas é construída. Pois a cada início de fase não existem nós ativos.
- A fase 1 (descarregar o máximo de fluxo possível) é realizada enquanto existe nó ativo na atual rede em camadas, que, neste caso, devem apenas obedecer a condição $e(v) > 0$.
- A fase 2 (retornar fluxo para a fonte) acontece juntamente com a fase 1, pois, em uma determinada rede em camadas, quando não é mais possível enviar fluxo para o sumidouro, a operação de retorno retira os excessos, da forma citada anteriormente, fazendo com que não existam mais nós ativos ao fim de cada etapa. Por este motivo, ao final de cada fase, o fluxo já está viável, isto é, todos os nós obedecem as restricões de conservação e capacidade.

Assim, a operação DESCARREGUAR FLUXO(C_{uv} do algoritmo da Figura 3.4 não sofre muita alteração, pois o nó mais baixo é um vizinho, ou seja o caminho será composto de apenas um arco. A alturas dos nós correspondem as camadas as quais eles pertencem (quanto mais próxima da fonte, mais alto o nível da camada), então, a operação *eleve nível* fica subentendida no procedimento de retorno, com o cancelamento do fluxo em um arco que chega a determinado nó.

Este algoritmo reduz o tempo de execução do algoritmo de Dinitz, o qual este se baseou, de $O(n^2m)$ para $O(n^3)$. Este valor é alcançado porque, em cada fase, as operações de descarga e retorno de fluxo são executadas $n-2$ vezes, no máximo. A primeira, porque é o número máximo de camadas que ela pode percorrer e a segunda, porque $n-2$ nós podem ser bloqueados (nem a fonte, nem o sumidouro são bloqueados). E uma vez bloqueado, o fluxo não poderá mais ser enviado pelo nó. Como são necessárias n rede em camadas, no pior caso, para que s não alcance mais t, pelo mesmo argumento dado no algoritmo de Dinitz, tem-se portanto que este algoritmo acha um fluxo bloqueante em $O(n^2)$ passos e termina sua execução, achando um fluxo máximo, em $O(n^3)$.

5.2. Algoritmo de Pré-fluxo com Rotulamento

O algoritmo define um rotulo de distância usando uma função de comprimento unittária e utiliza arcos admissíveis para identificar os candidatos a predecessores de t que receberão o fluxo originado dos nós com excessos.

5.2.1. Rótulo de distância

Antes de descrever o algoritmo, duas questões devem ser mostradas. Primeiro, qual a ordem de escolha do nó ativo a enviar fluxo. O caminho C_{vw}, utilizado como parâmetro na função DESCARREGAR FLUXO(C_{vw}), deve ser formado por v com excesso, o qual foi escolhido a partir de uma função que manipula a estrutura de dados para guardar os nós com excesso. E w é um vizinho de v alcançado a partir de um arco admissível. A segunda questão é como estimar a distância de um nó para s ou para t. Para isto, como visto na Seção 3.2.2, utiliza-se *rótulos de distância d*. Se arcos admissíveis *para frente* (na direção do sumidouro) não existem, então aumenta-se o rótulo de distância do nó com a intenção de criar arcos admissíveis *para trás*, isto é, de volta para a fonte.

O lema a seguir apresenta uma importante propriedade sobre rótulos de distância.

Lema 4. *Se f é um pré-fluxo e d é um rótulo válido qualquer para f, então o sumidouro t não é alcançável a partir da fonte s na rede residual.*

5.2.2. Descrição

O algoritmo de fluxo máximo começa com um pré-fluxo f que é igual a capacidade do arco, para aqueles que tem a fonte como origem (pré-processamento), e zero para todos os outros.

```
      Algoritmo: GOLDBERG E TARJAN [GENÉRICO]
1.    início
2.        f := 0;
3.        para cada arco (s,v) ∈ E_f faça                  % pré-processamento
4.            f(s,j) := c(s,j);
5.        d(v) = 0, para todo v ∈ V − {s};
6.        d(s) := n;
7.        enquanto a rede contém um nó ativo v faça        % d(v) < ∞ e e(v) > 0
8.            seleciona um nó ativo v;
9.            se a rede contém um arco admissível (v,w) então   % descarga
10.           descarregue δ := min{e(v), c_f(v,w)} unidades
                  de fluxo do nó v para o nó w;
11.           senão d(v) := min{d(w) + 1 : (v,w) ∈ E_f};   % rotulamento
12.       fim enquanto
13.   fim
```

Figura 5.2. *Algoritmo de Goldberg e Tarjan genérico*

É fácil ver que a linha 10 é equivalente a linha 9 do algoritmo da Seção 3.3.2, considerando o caminho de um nó mais alto para um mais baixo como sendo um único arco. E a linha 11 é a forma de elevar o nível de um nó escolhida por este algoritmo.

Como no algoritmo de Karzanov, as duas fases do algoritmo da Figura 3.4 não estão bem separadas neste. Em [2, 6], pode-se encontrar uma fácil modificação do Algoritmo 5.2 para dividir seu processamento em duas fases bem definidas.

A operação de pré-processamento realiza tarefas importantes, como já visto anteriormente. Primeiro, coloca, em cada nó adjacente a s, excesso de fluxo positivo, assim o algoritmo pode começar escolhendo algum nó ativo. Segundo, como a operação de pré-

processamento satura todos os arcos incidentes a s, nenhum dos arcos é admissível. Terceiro, como $d(s) = n$, a Propriedade 3 implica que não existem caminhos direcionados do nó s para o nó t na rede residual. Como os rótulos das distâncias não diminuem (Propriedade 5), também é garantido que a rede residual nunca irá conter tais caminhos e, portanto, não precisa-se descarregar fluxo a partir de s novamente.

Uma parte do algoritmo que também deve ser especificada é a escolha dos rótulos iniciais. Para facilitar as provas de corretude e complexidade, assume-se uma escolha simplificada com $d(s) = n$ e $d(v) = 0$ para todo $v \in V - \{s\}$. Contudo, uma escolha mais apropriada é fazer $d(v) = \min(d_{G_f}(v,t), d_{G_f}(v,s) + n)$ para todo $v \in V$, onde f é o pré-fluxo inicial. Ou seja, determinar os rótulos de distância exatos, executando uma busca em largura a partir de t.

Para ilustrar a execução deste algoritmo, considere a Figura 5.3. O item (a) da figura representa a rede residual de fluxo com as respectivas capacidades e o rótulo $d(v)$ para todo $v \in V$. Para este exemplo utilizou-se a segunda escolha de rótulos, por ocasionar uma execução mais rápida. O item (b), especifica o pré-fluxo determinado pela operação de pré-processamento.

Iteração 1. Suponha que o algoritmo selecione o nó 2 para uma operação de descarga ou rotulamento. O arco (2,4) é o único admissível e o algoritmo executa a descarga de $\delta = \min\{e(2), c_f(2,4)\} = \min\{2,1\} = 1$. Esta descarga é saturante. O item (c) mostra a situação da rede residual após este passo.

Iteração 2. Suponha que o algoritmo novamente selecione o nó 2. Como não há mais arcos admissíveis com origem no nó 2, o algoritmo executa uma operação de rotulamento e atribui um novo valor para o rótulo da distância de 2, $d(2) = \min\{d(3)+1, d(1)+1\} = \min\{2,5\} = 2$. A rede residual é quase a mesma mostrada no item (c), com a exceção que $d(2) = 2$ ao invés de 1.

Iteração 3. Suponha que desta vez o algoritmo selecione o nó 3. O arco (3,4) é o único admissível a partir de 3. O algoritmo executa a descarga de $\delta = \min\{e(3), c_f(3,4)\} =$

$\min\{4, 5\} = 4$. Esta descarga é não-saturante. O item (d) mostra a situação da rede residual após este passo.

Iteração 4. O algoritmo seleciona o nó 2 e executa uma descarga não-saturante de valor $\delta = \min = \{1, 3\} = 1$, obtendo a rede residual mostrada no item (e).

Iteração 5. O algoritmo seleciona o nó 3 e executa uma descarga saturante de valor $\delta = \min\{1, 1\} = 1$ pelo arco (3,4), obtendo a rede residual dada no item (f) da Figura 5.3.

Então a rede não contém mais nós ativos e o algoritmo termina. O valor do fluxo máximo na rede é $e(4) = 6$ (excesso no sumidouro).

É fácil notar que a cada momento, quando existe um nó ativo, sempre é possível executar uma das duas operações básicas neste nó. A partir disto pode-se afirmar o lema a seguir [1, 2, 5].

Lema 5. *Se f é um pré-fluxo, d é um rótulo válido qualquer para f, e v é um nó ativo qualquer, então ou uma operação de descarga ou de rotulamento pode ser aplicada a v.*

É importante observar que a execução do algoritmo não é única, pois algumas decisões são aleatórias. O leitor irá observar mais adiante que mesmo que as políticas de escolha do nó ativo estejam bem-definidas, ainda assim pode-se ter mais de uma execução possível.

5.2.3. Corretude e complexidade

Nesta seção será mostrado que o algoritmo de pré-fluxo genérico é correto, ou seja, que ele realmente termina e encontra um fluxo máximo. As provas dos lemas e teoremas aqui apresentados podem ser encontradas em [2] e [5].

Primeiro é mostrado que se o algoritmo termina, então o pré-fluxo f encontrado é máximo (Teorema 3). Depois é apresentada a terminação. A prova da corretude utiliza o conceito de caminho $s - t$ e é baseada no Teorema 1.

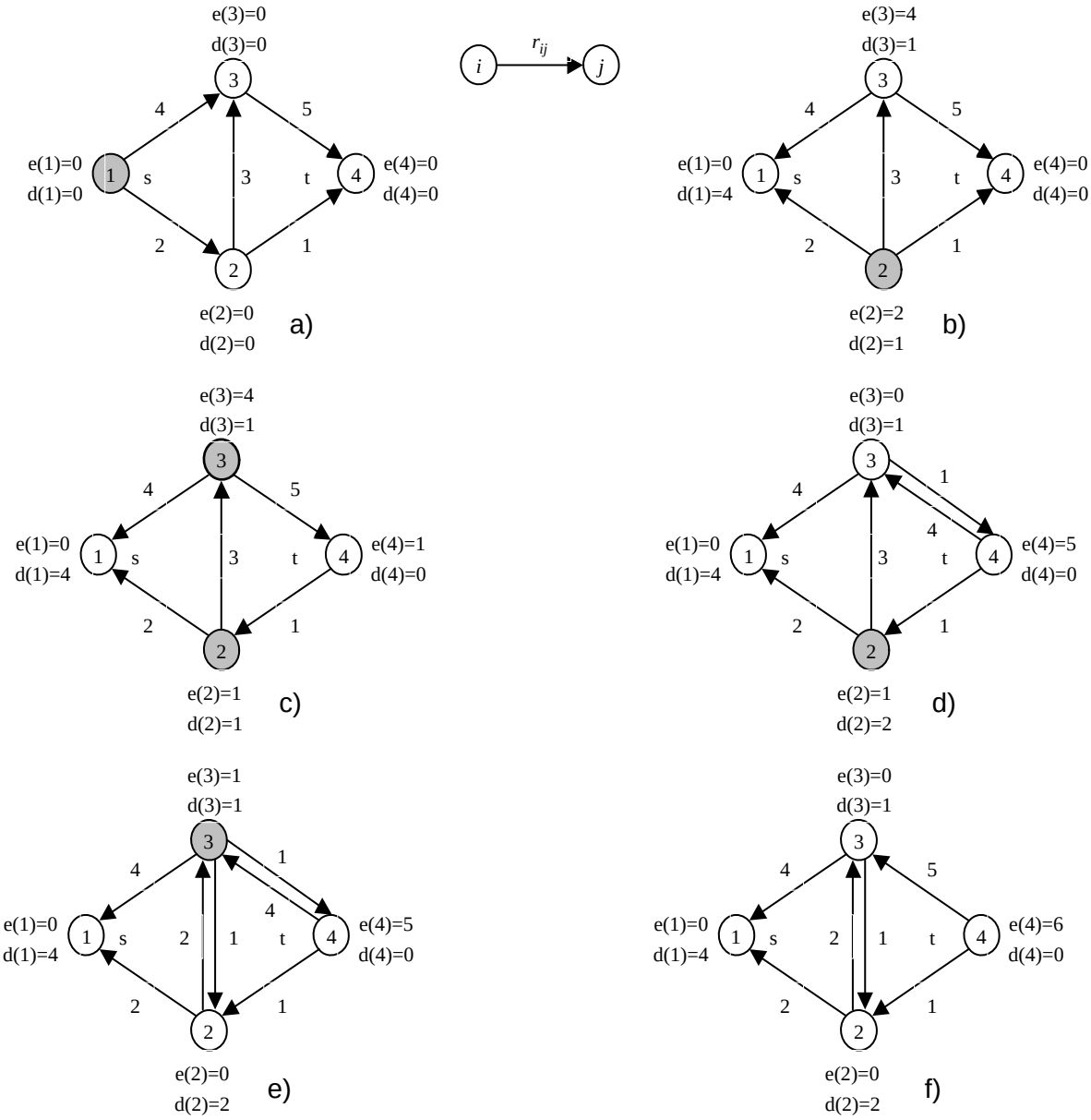


Figura 5.3. *Exemplo de uma execução seqüencial do algoritmo de pré-fluxo genérico.*

Algumas observações sobre os rótulos de distância também são necessárias. Pela Propriedade 5 os rótulos de distância nunca diminuem e os lemas a seguir mostram que eles permanecem finitos durante a execução do algoritmo.

Lema 6. *Uma aplicação de uma operação de rotulamento em v aumenta $d(v)$.*

Portanto, a aplicação da operação de rotulamento em um nó aumenta o valor de seu rótulo. O próximo lema mostra que os rótulos não podem crescer muito, ou seja, permanecem finitos.

Lema 7. *A qualquer momento da execução do algoritmo e para todo nó $v \in V$, $d(v) \leq 2n - 1$.*

O tempo de execução do algoritmo genérico depende da ordem na qual as operações básicas (descarga e rotulamento) são aplicadas e de detalhes da implementação, mas, como será visto a seguir, qualquer implementação seqüencial razoável do algoritmo executa em tempo polinomial.

A complexidade do algoritmo é dada então pelo número total de operações básicas, pois estas não podem ocorrer ao mesmo tempo, ou seja, em um nó ativo v, ou aplica-se rotulamento, ou descargas, onde estas últimas podem ser de dois tipos - saturantes e não-saturantes.

O total de rotulamentos pode ser trivialmente provado a partir do Lema 7 e da suposição que s e t nunca são ativos.

Lema 8. *O número de operações de rotulamento é no máximo $2n-1$ por nó e no máximo $(2n-1)(n-2) < 2n^2$ no total.*

Para que descargas saturantes ocorram em um mesmo arco (v, w), os rótulos válidos de v e w devem alterar-se de forma que $d(v)+d(w) \geq 1$ quando a primeira descarga saturante acontece e $d(v) + d(w) \leq 4n - 3$ quando a última descarga acontece. Assim, para todos os arcos temos um total de descargas saturantes mostrado no lema a seguir.

Lema 9. *O número de operações de descargas saturantes é no máximo* $2nm$.

Cada descarga não-saturante diminui a soma de todos os rótulos dos nós ativos de 1 unidade. Enquanto as descargas saturantes e os rotulamentos aumentam este valor num total de $(2n-1).2nm$ (Lema 9) e $(2n-1)(n-2)$ (Lema 8), respectivamente. Como o algoritmo começa e termina com nenhum nó ativo, esta soma deve possuir valor zero nas duas situações. Logo, o número total de descargas não-saturantes deve ser a soma dos dois valores anteriores, a fim de diminuir o valor da soma.

Lema 10. *O número de operações de descargas não-saturantes é no máximo* $4n^2m$.

O termo dominante na soma das operações básicas é o total de descargas não-saturantes. O teorema abaixo pode então ser provado trivialmente a partir dos Lemas 8, 9 e 10.

Teorema 4. *O algoritmo pré-fluxo genérico termina após* $O(n^2m)$ *operações básicas.*

5.2.4. Variações

Como foi visto no início desta seção, o algoritmo de Goldberg e Tarjan apresentado, foi chamado de genérico, pois a forma de escolher o nó ativo (entre outros fatores), no qual se aplicará uma operação de descarga ou rotulamento (linha 8), possui vários tipos e isto, influencia diretamente na eficiência deste algoritmo. Em [1] e [21] são apresentadas algumas variações do algoritmo com relação a esta escolha e a influência destas decisões nas complexidades. A Tabela 5.1 resume algumas destas variações existentes e exibe suas complexidades de pior caso.

No método LIFO, os nós vão sendo examinados na ordem de uma pilha, ou seja, o último que se tornou ativo, será o primeiro a ser examinado. Este método é bastante ruim na prática.

Quando os nós ativos são examinados na ordem de uma fila, tem-se então o algoritmo conhecido como pré-fluxo FIFO.

Tabela 5.1. *Variações do Algoritmo de pré-fluxo com rotulamento*

Algoritmo	Tempo de Execução	Características
Pré-fluxo LIFO	$O(n^3)$	1.Examina nós ativos em ordem LIFO. 2.Método menos eficiente na prática.
Pré-fluxo FIFO	$O(n^3)$	1.Examina nós ativos em ordem FIFO. 2.Muito eficiente na prática.
Pré-fluxo de maior rótulo	$O(n^2\sqrt{m})$	1.Examina nós ativos com maior rótulo de distância. 2.Possivelmente o algoritmo de fluxo máximo mais eficiente em prática.
Escalonamento de excessos Original	$O(nm + n^2 \log U)$	1.Executa operações descarga/rotulamento em nós com excessos suficientemente grandes e, entre esses nós, seleciona o de menor rótulo de distância. 2.Alcança excelentes tempos de execução sem usar estruturas de dados sofisticadas.
Escalonamento de excessos utilizando pilha	$O(nm + \frac{n^2 \log U}{\log \log U})$	1.Executa operações descarga/rotulamento em nós com excessos suficientemente grandes e, entre esses nós, seleciona o de maior rótulo de distância. 2.O mais rápido entre as três variações de escala de excessos.
Escalonamento de excessos em ondas	$O(nm + n^2\sqrt{\log U})$	1.Mistura os dois métodos anteriores. 2. Não atinge bons tempos na prática.

Para o método maior rótulo, seja $h^* = \max\{d(v) : v \text{ ativo}\}$. Os nós ativos com $d = h^*$ são escolhidos e enviam fluxo para os nós com $d = h^* - 1$. Estes por sua vez enviam fluxo para aqueles com $d = h^* - 2$, assim sucessivamente até que uma das duas possibilidades a seguir aconteça:

1. não existam mais nós ativos, então o algoritmo pára;

2. uma operação de rotulamento é feita, então o algoritmo recomeça.

Esta técnica é considerada e provada em vários estudos computacionais, um dos algoritmos mais rápidos para a maior parte das classes de grafos.

O método de escala de excessos foi originalmente proposto por Ahuja e Orlin [1]. O algoritmo de pré-fluxo, como foi visto, trabalha de forma a produzir os chamados excessos nos nós. Seja $e^* = \max\{e(v) : v \text{ ativo}\}$, o maior valor de excesso. Observando as execuções do algoritmo de pré-fluxo, nota-se que não existe um padrão particular para o valor que e^*

assume. A única certeza é que este valor diminue até 0.

A idéia desta modificação/método é semelhante ao algoritmo de escala de capacidades, ou seja, o segundo melhora a complexidade do algoritmo genérico de caminhos aumentantes, levando uma quantidade de fluxo *suficientemente grande* ao longo de um caminho. Da mesma forma, descargas não-saturantes de pequenos valores são uma das causas (téoricas) de uma complexidade maior nos algoritmos de pré-fluxo. Então, o algoritmo de escala de excessos tenta garantir que a quantidade de fluxo que passará por arcos insaturados sejam grandes o suficiente, para que o número de descargas não saturantes seja pequeno.

Para mostrar o algoritmo (Figura 5.4), seja Υ um limite superior para o valor de e^*, chamado de *excesso dominante*. Se um nó v possui $e(v) \geq \Upsilon/2 \geq e^*/2$, então ele possui *excesso grande*. caso contrário, o nó v possui *excesso pequeno*. O algoritmo escolhe, portanto, sempre os nós ativos com *excessos grandes* e que estão mais próximos do sumidouro (menor rótulo). Além disso, o algoritmo não permite que nenhum excesso seja maior que Υ. Com isto, a operação de descarga, sobre um arco vw, sofre uma modificação, ou seja, $\delta = \min\{e(v), c_f(v,w), \Upsilon - e(w)\}$.

```
     Algoritmo: Escala de Excessos
1.   início
2.       pré-processamento;
3.       Υ := 2^⌈log U⌉
4.       enquanto Υ ≥ 1 faça
5.       % início de uma fase Υ-escalonada
6.           enquanto a rede possui um nó com excesso grande faça
7.               dentre todos os nós com excesso grande,
8.                   escolha um nó v com o menor rótulo de distância;
9.               execute descarga/rotulamento(v), garantindo que
10.                  nenhum e(w) ≥ Υ, para todo nó w;
11.          fim enquanto
12.          Υ := Υ/2
13.      fim enquanto
14.  fim
```

Figura 5.4. *Algoritmo escala de excessos (menor rótulo)*

Além deste algoritmo de escala de excessos mais geral, existem também as técnicas de escala de excessos escolhendo os de maior rótulo e uma que mistura os dois acima, chamado

de escala de excessos em onda. O primeiro, se mostra mais eficiente na prática que os outros dois de escala de excessos. O segundo, utiliza um parâmetro para decidir qual das duas formas anteriores ele irá usar para escolher um nó ativo. Este parâmetro é a soma de todos os excessos ($\sum e(v)$). Se este valor é suficientemente grande ($\sum e(v) > n\Upsilon/\sqrt{\log U}$), então aplicasse o método escala de excessos-pilha, caso contrário, ou seja, quando os excessos diminuírem, escala de excessos original é então aplicado.

Além destes vários métodos de escolha do nó ativo, outros procedimentos e escolhas podem ser aplicados ao algoritmo genérico, para melhorar o tempo de resposta. Por exemplo, a cada vez que um nó ativo é escolhido, pode-se *aplicar apenas uma das duas operações* (descarga ou rotulamento) e então escolher outro nó para trabalhar; ou continuar com este nó, fazendo mais de uma operação de descarga até que seu excesso se torne nulo ou uma operação de rotulamento aconteceu. Neste caso, diz-se que o nó ativo será *examinado*. A maioria dos algoritmos escolhem esta alternativa de trabalhar com os nós ativos, ou seja, só escolhe outro ativo quando o que está sendo utilizado ficou com excesso 0 ou foi rotulado.

Outras estratégias para melhorar o tempo deste algoritmo são as heurísticas *rotulamento global* e *rotulamento de intervalo*. A primeira executa, periodicamente, uma busca em largura nos nós da rede residual, para determinar as distâncias exatas destes até o sumidouro. Para grafos com $m \leq 10n$, o procedimento é executado depois de n operações de descarga. Para o caso contrário, ou seja, $m < 10n$, então a heurística de rotulamento global é chamada depois de $2n$ operações de descarga. Estas freqüências, contudo, podem mudar de acordo com a estrutura do grafo de entrada.

Para a segunda heuristica, se existe um valor l $(0 < l < n)$ tal que nenhum nó tenha rótulo de distância igual a l, então todos os nós v com $l < d(v) < n$ podem ser rotulados para $d(v) = n$. Isto torna o retorno de excessos para a fonte mais rápido. Ou seja, esta técnica se mostra bastante eficiente quando utilizada nos algoritmos cuja divisão de execução em duas fases é bem definida.

5.3. Algoritmo de Hochbaum

Por muito tempo, o algoritmo anterior era considerado o mais rápido para resolver o problema de fluxo máximo, principalmente por seus desempenhos práticos. Em 1997, Hochbaum então desenvolveu um outro método, ainda trabalhando com pré-fluxos, com complexidade de pior caso semelhante ao anterior [6, 26].

O algoritmo de Hochbaum difere do algoritmo de Goldberg e Tarjan, por enviar fluxo não apenas para o próximo vizinho de um nó com excesso. Neste, é mantido um conjunto de nós predecessores de t e, para cada um deles, tem-se um caminho que leva de um nó com excesso até ele.

O algoritmo possui duas fases e um pré-processamento. As variações no pré-processamento e na busca dos caminhos podem melhorar a complexidade do algoritmo.

A seguir tem-se a descrição geral do algoritmo, sua corretude e complexidade, assim como as diversas variações existentes, visando a melhoria do caso geral.

5.3.1. Descrição

A idéia geral do algoritmo é dividir os nós da rede ($V(G) - \{s\}$) em conjuntos R. Estes, por sua vez, devem atender as seguintes propriedades:

1. cada conjunto R de nós possui um nó r chamado de *representante*;
2. cada nó só participa de um único conjunto por vez. Desta forma, a rede de entrada pode ser vista como uma floresta (uma partição do conjunto de nós em árvores);
3. apenas os nós representantes podem excesso positivo ($e(r) \geq 0$) no conjunto. Para todos os outros $e(v) = 0$. Podem existir conjuntos onde todos os nós possuem excesso nulo;
4. é sempre possível encontrar um caminho orientado de qualquer nó de um conjunto até seu representante;

5. o conjunto que contém o sumidouro R_t é chamado de *especial*, e t é seu representante.

O algoritmo trabalha procurando arcos, na rede residual entre nós de qualquer conjunto R para algum nó do conjunto especial R_t. Tais arcos são chamados de *arcos união*, ou seja, se vw é um arco união, então $c_f(v, w) > 0$, $v \in R$ e $w \in R_t$. Os dois conjuntos são então unidos ($R_t = R_t \cup R$). A intenção é enviar o excesso do representante do conjunto R para o sumidouro t, passando pelo arco escolhido. Quando arcos união não mais existirem, o algoritmo pára.

Para mostrar o funcionamento do algoritmo de maneira mais formal, serão apresentadas as operações básicas divididas em inicialização, fase 1 e fase 2. A medida que estas são mostradas, uma comparação com o algoritmo da Seção 3.3.2 será feita.

5.3.1.1. Operações e formulação

A criação dos conjuntos deve ser feita como primeiro passo. Para isso, a fase de inicialização satura os arcos que saem da fonte para criar nós com excessos positivos. A partir daí, são feitos $n - 1 - w$ conjuntos, onde w é o número de vizinhos de t. A Figura 5.5 ilustra o resultado da operação de inicialização sobre uma rede G. Ou seja, o conjunto R_t é formado por t e seus vizinhos; e todos os outros nós, excluindo a fonte, formam conjuntos unitários.

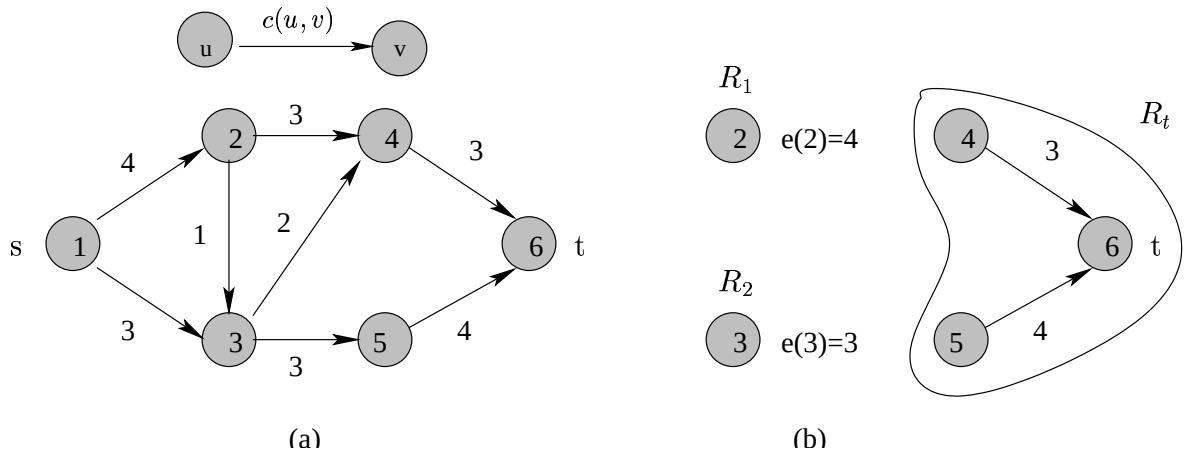


Figura 5.5. *Exemplo de inicialização do algoritmo de Hochbaum: (a) rede original; (b) criação de nós com excesso e determinação dos conjuntos.*

Esta não é a única forma de inicializar os conjuntos, contudo, é a mais simples. Outras maneiras de inicialização serão vistas na Seção 5.3.3.

É fácil ver que esta operação corresponde ao pré-processamento, pois criará nós com excessos positivos.

Depois da inicialização é possível começar a fase 1. Nesta fase, os arcos união serão examinados e os excessos serão enviados para t. A fase termina quando não houver mais estes arcos na rede residual. Neste momento, contudo, alguns nós podem não estar respeitando a restrição de conservação de fluxo, mas este problema será resolvido na fase 2, a qual retornará fluxo em excesso de volta para a fonte.

Para descrever o processo da fase 1, considere os conjuntos $R_1, R_2, \ldots, R_t$ e a rede residual G_f obtida na inicialização. Seja r o nó representante de um conjunto R e t o representante do conjunto especial R_t. E seja, também, vw um arco união ($v \in R$ e $w \in R_t$) escolhido em determinado momento. Quando um arco união é escolhido, o conjunto R é unido ao conjunto especial R_t, formando um novo conjunto R'_t. Assim, tem-se duas possibilidades:

1. Se v *é representante* de R, então o excesso de v é enviado para t passando pelo arco vw. Esta operação é uma aplicação da função a DESCARREGUAR FLUXO(C_{v-t}), visto na Seção 3.3.2, descarregando fluxo por um caminho v para t, passando por w. A diferença é que, quando um nó fica com excesso, então o conjunto R'_t é *partido*, e o novo conjunto terá como representante o nó com excesso. Isto devido a Propriedade 3 que os conjuntos devem obedecer, ou seja, apenas o representante deve possuir excesso positivo.

2. v *não é representante* de R. Pela própria construção dos conjuntos, existe um caminho orientado de qualquer nó até o representante do conjunto ao qual ele participa. Quando um nó v do conjunto R é escolhido e este não é seu representante, se faz então necessário uma operação chamada de *reagrupamento* para que o fluxo possa passar de r para v e deste para t.

A Figura 5.6 mostra as operações para reagrupamento de um conjunto e descaga de fluxo em um caminho. A variável $sucessor(v)$ indica o próximo nó no caminho de um nó qualquer v até o representante do conjunto ao qual v pertence (para um arco vw, diz-se que $sucessor(v) = w$). Assim, se o sucessor de um nó for igual a vazio, então este nó é o representante do conjunto.

```
    Procedimento: REAGRUPAMENTO(R, v)
1.  início
2.      a := sucessor(v);
3.      se a ≠ ∅ então
4.          REAGRUPAMENTO(R, a);
5.          sucessor(a) := v;
6.          sucessor(v) := ∅;
7.      fim se;
8.  fim
```

```
    Procedimento: DESCARREGUAR FLUXO(C_{r-t})
1.  início
2.      δ := e(r);
3.      enquanto sucessor(r) ≠ ∅ faça          % o sucessor só será vazio
4.          v := sucessor(r);                  % quando chegar em t
5.          δ := min{δ, c_f(r, v)};
6.          e(r) := e(r) − δ;
7.          e(v) := e(v) + δ;
8.          f(r, v) := f(r, v) + δ;
9.          se c_f(r, v) = 0 então             % forma outro conjunto
10.             sucessor(r) := ∅;              % onde r é representante
11.         r := v;
12.     fim enquanto;
13. fim
```

Figura 5.6. *Subrotinas de reagrupamento e envio de fluxo.*

Portanto, a rotina REAGRUPAMENTO(R, v) inverte a direção do caminho do representante do conjunto R até o nó v escolhido. Vale ressaltar que as únicas direções afetadas são dos arcos que fazem parte do caminho. E o procedimento DESCARREGUAR FLUXO$(C_{r,t})$ envia o excesso de r, representante de R, até t, pelo caminho $C_{r-v} \cup C_{v-t}$, dividindo o conjunto sempre que necessário para que as propriedades sejam atendidas.

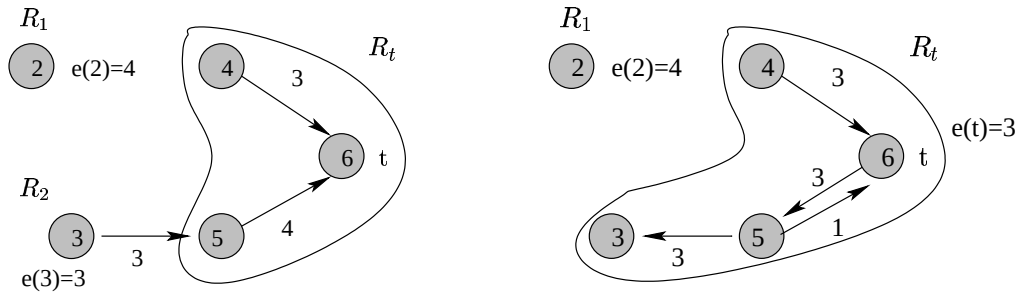
A partir destas duas operações é possível então apresentar o algoritmo de Hochbaum na Figura 5.7.

```
     Algoritmo: HOCHBAUM [GENÉRICO]
1.   início
2.       e(v) := 0, para todo v ∈ V;                    % início do pré-processamento
3.       f(v,w) := 0, para todo vw ∈ E;                 % ou rotina de inicialização
4.       f(s,u) := c(s,u);                              % satura arcos saindo da fonte
5.       e(u) := c(s,u);
6.       sucessor(v) := ∅, para todo v ∈ V − {s};        % cria conjuntos R
7.       sucessor(u) := t, para todo vizinho u de t;    % conjunto especial R_t
8.       se e(u) > 0, para todo vizinho u de t então    % para o caso de u ser vizinho
9.           δ := min{e(u), c(u,t)};                    % de t e de s
10.          e(u) := e(u) − δ;
11.          e(t) := e(t) + δ;
12.          f(u,t) := f(u,t) + δ;
13.          se e(u) > 0 então                          % forma outro conjunto
14.              sucessor(u) := ∅;

15.      enquanto existe um arco união vw faça          % fase 1
16.          REAGRUPAMENTO(R, v);                       % R é o conjunto que contem v
17.          sucessor(v) := w;                          % Une os conjuntos R e R_t
18.          DESCARREGAR FLUXO(C_{r−t});                % r representante de R
19.      fim enquanto;

20.      para todo v ∈ V − {s,t} com e(v) > 0 faça      % fase 2
21.          determine um caminho C em G_f iniciando em v;  % para todo v com e(v) > 0
22.          δ := min{c_f(i,j)}, para todo ij ∈ C;      % descarregar fluxo em C_{v−s}
23.          diminua δ unidades de fluxo em C;
24.          se C é um caminho de v para s então
25.              e(v) := e(v) − δ;
26.      fim para
27.      f := e(t);
28.  fim
```

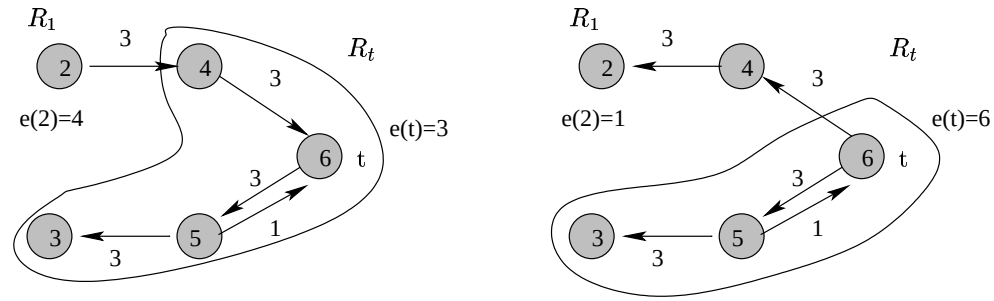
Figura 5.7. *Formulação genérica do algoritmo de Hochbaum.*

A Figura 5.8 mostra um exemplo da execução da fase 1 do algoritmo acima. Considere a inicialização da Figura 5.5.

Assim, este algoritmo também é classificado como um de manipulação de pré-fluxo, pois trabalha com nós com excessos e é dividido nas três fases vistas na Seção 3.3.2. Um pré-processamento criando nós ativos, uma fase onde tenta-se enviar o máximo de fluxo para o sumidouro t e uma outra que ajusta a condição de conservação, ou seja, retorna



Iteração 1 - escolha do arco união 34



Iteração 2 - escolha do arco união 24

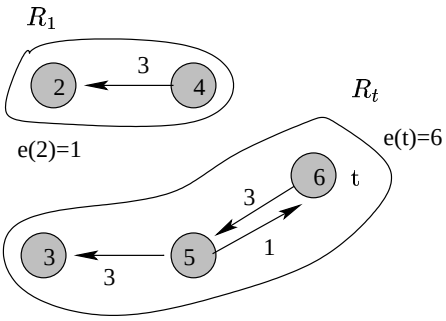


Figura 5.8. *Exemplo de execução da Fase 1 do algoritmo de Hochbaum.*

para a fonte o que não pode ser enviado para t.

A visão dada nesta seção difere da encontrada em [6, 26]. No algoritmo visto aqui, é criado apenas um conjunto especial e somente a fonte não participa de nenhum conjunto. Uma outra forma seria criar $n-2$ conjuntos, dentre estes, w conjuntos especiais (w é o número de vizinhos de t) e considerar a rede sem a fonte, sem o sumidouro, sem os arcos partindo da fonte e sem aqueles chegando ao sumidouro. Então, os nós vizinhos a t teriam um excesso negativo. Contudo, a definição de arco união vw seria a mesma, $c_f(v,w) > 0$, $v \in R$ e $w \in R_t$, onde R é um conjunto qualquer, com representante r de excesso positivo ($e(r) \geq 0$), e R_t é um conjunto especial qualquer, com representante t' de excesso negativo ($e(t') < 0$).

Portanto, a fase 1 do algoritmo da Figura 5.7 continuaria do mesmo jeito. Contudo, a inicialização e a fase 2 sofreriam pequenas modificações como mostra o algoritmo da Figura 5.9.

As duas versões são equivalentes. Como será visto, a eficiência não se altera, pois a fase 1, que determina o valor de complexidade, por ser a mais custosa, não é modificada.

Escolheu-se mostrar a idéia de um único conjunto especial por ser mais simples. Assim, é possível imaginar esta segunda forma de construção dos conjuntos apresentada, como se a rede possuísse mais de um sumidouro.

A seguir, tem-se a corretude e complexidade deste método.

5.3.2. Corretude e complexidade

É preciso mostrar que o algoritmo termina, que ele encontra um fluxo máximo e qual o custo de suas operações. As provas dos lemas e teoremas aqui apresentados podem ser encontradas em [6, 26].

É importante notar que a união de um conjunto R ao conjunto especial pode criar zero ou mais conjuntos R, mas o conjunto especial é sempre único. Além disso, um nó pode alternar entre pertencer a um conjunto R e o conjunto R_t. Então, para a fase 1, pode-se

Algoritmo: HOCHBAUM [MODIFICADO]

```
1.  início
2.      e(v) := 0, para todo v ∈ V;                    % início do pré-processamento
3.      f(v,w) := 0, para todo vw ∈ E;                 % ou rotina de inicialização
4.      f(s,u) := c(s,u);                              % satura arcos saindo da fonte
5.      f(w,t) := c(w,t);                              % satura arcos chegando em t
6.      e(u) := c(s,u);
7.      e(w) := e(w) − c(w,t);
8.      sucessor(v) := ∅, para todo v ∈ V − {s,t};      % cria conjuntos R e R_t

9.      fase 1

10.     para todo v ∈ V − {s,t} com e(v) ≠ 0 faça      % fase 2
11.         se e(v) > 0 então                          % retorna para a fonte
12.             determine um caminho C em G_f iniciando em v;
13.             δ := min{c_f(i,j)}, para todo ij ∈ C;
14.             diminua δ unidades de fluxo em C;
15.             se C é um caminho de v para s então
16.                 e(v) := e(v) − δ;
17.         senão                                      % retorna para a sumidouro
18.             f(v,t) := f(v,t) + e(v);
19.             e(v) := 0;
20.     fim para
21.     f := e(t);
22. fim
```

Figura 5.9. *Visão diferenciada da formação dos conjuntos da versão genérica do algoritmo de Hochbaum.*

afirmar o seguinte lema:

Lema 11. *Cada iteração ou reduz o excesso total dos representantes dos conjuntos R, ou o número de conjuntos com todos os nós com excesso nulo.*

É fácil ver que isto é verdade, pois, como o arco união é escolhido da rede residual, então pelo menos uma unidade de fluxo chegará a t. Por outro lado, se o conjunto escolhido não possui nenhum nó com excesso, então ele será simplesmente adicionado ao conjunto R_t, diminuindo assim o número destes tipos de conjuntos.

Como as capacidades são finitas, os arcos união da rede residual vão acabar. Isto significa que t não é mais alcançável, logo, pelo Teorema 1, o fluxo neste instante é máximo. A fase 2 não modifica o valor escalar deste fluxo e o torna viável.

A complexidade deste algoritmo é facilmente calculada. A *inicialização* pode ser feita em $O(m)$. Na *fase 1*, pode-se encontrar no pior caso $O(m)$ arcos união. Se cada um pode levar uma unidade de fluxo para t, então esta fase termina após $O(nmW)$ iterações, onde $W = \sum_{(s,v)\in E} c(s,v)$, que é a quantidade total de excesso gerada no início do procedimento.

É fácil ver que a *fase 2* retorna fluxo para s em $O(nm)$ iterações ($O(n)$ nós com excesso e $O(m)$ arcos a percorrer até a fonte).

Assim, tem-se o teorema a seguir:

Teorema 5. *A complexidade total do algoritmo genérico de Hochbaum é* $O(nmW)$.

Como já visto, esta é uma complexidade pseudopolinomial, pois depende do valor W (diferentemente do algoritmo de Goldber e Tarjan que possui complexidade polinomial mesmo na versão genérica). Em [6] é possível encontrar variações na fase 1 do algoritmo, a qual é a mais custosa, melhorando a complexidade do algoritmo genérico de $O(nmW)$ para $O(nm \log n)$.

5.3.3. Variações

Assim como o algoritmo da Seção 5.2, este possui variações no método genérico que influenciam na complexidade do algoritmo. Algumas destas variações são descritas nesta seção.

O que pode ser mudado na versão genérica é a forma de escolha de um arco união, assim como a inicialização dos conjuntos. Primeiramente, serão apresentadas duas maneiras de escolha: *menor rótulo* e *maior rótulo*. Ambas as escolhas (menor ou maior rótulo) são modificações apenas na fase 1.

5.3.3.1. Algoritmos Menor e Maior Rótulo

No algoritmo *menor rótulo* associa-se um rótulo $d(v)$ para todo $v \in V - \{s\}$. Durante a inicialização, coloca-se rótulo 0 para todos os nós do conjunto especial e rótulo 1 para os outros. Então, o algoritmo executa procurando por arcos união vw onde $d(v) = d(w) + 1$. Se nenhum arco deste tipo pode ser encontrado, então eleva-se os rótulos dos nós dos conjuntos R ($v \in R$) de uma unidade.

A cada iteração do algoritmo, escolhe-se um conjunto R que possui nós com menor rótulo l. Se algum vizinho destes nós possui rótulo $l-1$, então este, com certeza, participa do conjunto especial R_t. As operações de reagrupamento, união e descarga de fluxo são realizadas como descrito anteriormente. Se nenhum vizinho deste tipo é encontrado, então os rótulos dos nós do conjunto R são aumentados.

Este processo continua até que o menor rótulo dos nós dos conjuntos R seja maior que n, pois neste momento, a rede está desconexa e não é mais possível enviar fluxo para t.

Neste método, é importante observar que a heurística de *rotulamento de intervalo*, descrita na Seção 5.2.4, possui um efeito maior neste algoritmo que no de Goldberg e Tarjan. Isto é, ao invés de deixar de examinar apenas os nós ativos com rótulo maior que um determinado l, não serão examinados todos os conjuntos com rótulo maior que l. Fato que agiliza a passagem para a fase 2.

A Figura 5.10 mostra a fase 1, utilizando esta estratégia.

```
     Algoritmo: MENOR RÓTULO
1.   início
2.       inicialização
3.       d(v) := 1, para todo v ∈ R;
4.       d(v) := 0, para todo v ∈ R_t;
5.       escolhe R com menor rótulo;
6.       l := d(r), onde r é o representante de R;
7.       enquanto l ≤ n faça                          % fase 1
8.           N := nós v de R com d(v) = l;
9.           para todo v ∈ N faça
10.              para todo w vizinho de v faça
11.                  se d(w) = l − 1 então
12.                      REAGRUPAMENTO(R, v);
13.                      sucessor(v) := w;
14.                      DESCARREGUE FLUXO(r, f, R_t);
15.                      retorne enquanto;
16.                  fim se;
17.              fim para;
18.          fim para;
19.          para todo v ∈ N faça                     % não houve nenhuma união
20.              d(v) := l + 1;
21.      fim enquanto;
22.      fase 2
23.  fim
```

Figura 5.10. *Algoritmo de menor rótulo para exame dos arcos união.*

Pelo algoritmo, é fácil ver que o rótulo do representante de um conjunto é menor ou igual ao rótulo dos outros nós participantes deste mesmo conjunto. Isto acontece pela própria formação dos conjuntos. A prova desta característica pode ser encontrada em [6].

Esta estratégia pode ser vista como uma tentativa de agrupar os nós no conjunto especial R_t na *direção do sumidouro para a fonte.*

Uma outra forma seria fazer um procedimento contrário ao descrito acima, ou seja, aumentar o tamanho dos conjuntos R até chegar ao conjunto especial R_t, ou seja, agrupar os conjuntos na *direção da fonte para o sumidouro.* Esta estratégia será vista a seguir.

Como a melhor variação do algoritmo de Goldberg e Tarjan é examinar os nós ativos com maior rótulo (considerado o melhor na prática), Anderson [6] resolveu fazer uma variação no mesmo sentido no algoritmo de Hochbaum. Este é semelhante ao de menor rótulo,

contudo, são escolhidos os conjuntos R com nós de *maior rótulo* l. E, da mesma forma, são procurados vizinhos com rótulo $l-1$. Neste caso, contudo, a união dos conjuntos pode não acontecer entre um conjunto R e o conjunto especial R_t, mas sim entre dois conjuntos R. Portanto, um arco união vw é escolhido levando em consideração apenas a capacidade residual e os rótulos de v e w. Contudo, v e w ainda precisam pertencer a conjuntos distintos, mas não mais necessariamente a definição que tinha-se antes ($v \in R$ e $w \in R'$, onde R' pode ser um conjunto R ou R_t).

Assim, escolhido o conjunto com maior rótulo l, se não existir nenhum arco união como descrito acima, aumenta-se o rótulo destes nós de uma unidade. Vale notar que o mesmo conjunto será escolhido novamente.

É importante observar que apenas os representantes dos conjuntos são analisados neste caso, pois, como dito anteriormente, o rótulo do representante é sempre menor ou igual ao rótulo dos outros nós do conjunto. Então, se não existir um arco união vw, com $d(v) = d(w) + 1$, para o representante, logo também não existirão tais arcos para os outros nós.

A grande diferença deste método para o de menor rótulo está na forma de lidar com os intervalos de rótulos. Isto é, no anterior, assim que um intervalo é encontrado (existe um l tal que nenhum conjunto possui rótulo igual a este valor), então a fase 1 termina imediatamente. No caso de maior rótulo, o processo continua, mesmo existindo um intervalo. No momento que não é possível descarregar o excesso de um conjunto R para t, então este conjunto é *isolado* e não mais processado.

Formalmente, seja R o conjunto de maior rótulo l e $d(r)$ o rótulo do representante de R. Se não existe nenhum arco rw onde $d(r) = d(w) + 1$, então todos os nós deste conjunto R recebem rótulo n. Deste modo, este conjunto não será mais executado. O algoritmo então termina quando não existir mais conjuntos R com rótulos menores n.

A complexidade deste método é a mesma do algoritmo de menor rótulo.

5.3.3.2. Inicialização

Além das possibilidades de escolha dos arcos união existem também outras formas de inicializar os conjuntos.

Na Seção 5.3.1 foi descrita um inicialização *simples*, saturando os arcos partindo da fonte e criando o conjunto especial com t e seus vizinhos. A seguir serão apresentadas outras formas de *inicialização*, para todas elas, a simples é utilizada antes de seus procedimentos:

Caminho Para cada nó v com excesso positivo, procura-se um arco residual para algum vizinho w de v com capacidade para todo o excesso ($c_f(v,w) \geq e(v)$). Caso seja encontrado, w passa a ser representante do conjunto formado por v e w ($sucessor(v) = w$ e $sucessor(w) = \emptyset$). O processo é repetido para w até que nenhum vizinho com as características acima seja encontrado, ou o sumidouro tenha sido atingido. Portanto, a idéia desta inicialização é enviar os excessos *para a frente* o mais possível.

Gulosa No processo anterior, quando um arco vw, v com excesso, não é encontrado, o processo pára e passa para outro com excesso positivo que ainda não foi analizado. Neste caso, se um arco vw do tipo $c_f(v,w) \geq e(v)$ é encontrado, então o mesmo procedimento anterior acontece. Caso contrário, ou seja, não existe arco vw com capacidade residual capaz de suportar todo o excesso de um nó v, então, envia-se para w o que pode ($c_f(v,w)$) e os dois conjuntos resultantes desta operação são: um conjunto R_1, com v representante ($e(v) = e(v) - c_f(v,w)$) e R_2, com representante w ($e(w) = e(w) + c_f(v,w)$.

Menor Caminho Esta estratégia é parecida com a primeira, contudo, em vez de tentar aumentar o tamanho dos conjuntos R, enviando o excesso para nós mais próximos ao sumidouro, tenta-se aumentar o tamanho nó conjunto R_t. Isto é, inicialmente, realiza-se uma busca em largura a partir de t, para determinar as distâncias dos nós. Então, o conjunto R_t será formado por t, seus vizinhos e, para cada um destes

vizinhos (cuja distância $d(v)$ para t é 1), procura-se por nós u com excesso nulo e $d(u) = d(v) + 1$, para adicionar este ao conjunto especial.

Todas essas inicializações podem ser feitas em $O(m)$ passos.

Em [6] é possível encontrar os algoritmos e outras heuristicas para o algoritmo genérico de Hochbaum.

5.3.4. Sumário

Assim como foi possível encontrar uma forma de equivalência dos algoritmos de decomposição de fluxo em caminhos com o algoritmo genérico desta abordagem, descrito na Seção 3.3.1, será apresentada agora uma tabela que resume os três algoritmos, apresentados neste capítulo, dentro da abordagem mostrada na Seção 3.3.2. Isto é, a forma de criar nós ativos,de enviar o máximo de fluxo possível até o sumidouro, ou seja, de escolher um caminho de um nó v com excesso até um predecessor de t; e de retornar excessos para a fonte.

Tabela 5.2. *Resumo das operações dos algoritmos de manipulação de pré-fluxos*

Algoritmo	**Inicialização**	**Fase 1**	**Fase 2**
Pré-fluxo em camadas	1. Ocorre a cada construção da rede em camadas 2. Satura arcos saindo da fonte	Cálculo de fluxo bloqueante. A idéia é "empurrar" o máximo de fluxo de camada em camada. O caminho C_{v-w} escolhido para descarga de fluxo é de um nó v da camada k para um w na camada $k-1$.	Ocorre dentro da fase 1 com o bloqueio de nós e cancelamento de fluxo nos arcos incidentes.
Pré-fluxo com rotulamento	1. Ocorre uma única vez 2. Satura arcos saindo da fonte	Envia fluxo de arco em arco, por caminhos de menor comprimento. Várias formas para escolher um nó com excesso. O caminho C_{v-w}, escolhido, é de um nó v com excesso para um vizinho w com $d(v) = d(w) + 1$.	Duas possibilidades: junto ou separado da fase 1. Retorno por um caminho de menor comprimento para a fonte.
Hochbaum	1. Criação de conjuntos de nós 2. Ocorre uma única vez 3. Satura arcos saindo da fonte	União de conjuntos para envio de fluxo de nós com excesso para o sumidouro, através de caminhos pré-estabelecidos. Envio de fluxo por caminhos mais "longos"	Ocorre depois da fase 1, com o cancelamento de fluxo em um caminho para a fonte.

6

Eficiência

Nos capítulos anteriores, vários algoritmos e suas respectivas complexidades de pior caso foram apresentados. Porém, muitas vezes, um algoritmo funciona melhor na prática que na teoria, pois, como a complexidade calculada é de pior caso, pode ser difícil encontrar uma instância que cause esta possibilidade de execução. Então, devido ao tempo gasto na maioria das instâncias, determinado algoritmo acaba se tornando preferível com relação a outro que, para obter uma complexidade pequena, utiliza recursos de difícil implementação. Isto além dos motivos já citados na Seção 2.3 - habilidade do programador, arquitetura da máquina, consumo de memória por determinada estrutura de dados etc.

A análise experimental, embora não exata em muitos casos, é feita para que se possa identificar o comportamento dos algoritmos na prática. Os resultados contidos neste capítulo foram colhidos principalmente em [6, 23, 27]. Estes foram reunidos com o objetivo de mostrar ao leitor qual algoritmo *vale a pena* implementar ou qual seria mais adequado para determinados tipos de instâncias, ou, ainda, se existe algum algoritmo que se mostre mais rápido para uma maioria de tipos de grafos.

As tabelas comparativas, aqui mostradas, apresentam apenas o tempo total gasto. Quando alguma variação (ou heurística) do algoritmo for utilizada para alcançar determinados tempos, esta será devidamente especificada. Outra abordagem de análise exper-

imental, onde as operações de cada algoritmo são comparadas, e não o tempo de CPU, pode ser encontrada em [21, 23].

Este capítulo se divide em três partes:

1. comparação dos algoritmos de decomposição de fluxo em caminhos;
2. comparação dos algoritmos de manipulação de pré-fluxos;
3. comparação entre os melhores das duas abordagens.

Antes, serão apresentadas as classes de instâncias utilizadas nos testes a fim de mostrar a *densidade* e *forma* dos grafos.

6.1. Classes de Instâncias

As instâncias são criadas a partir de geradores bem conhecidos utilizados pelos *desafios DIMACS*. Todos os grafos são gerados aleatoriamente, a partir de um valor de *semente*. A seguir, tem-se a descrição sumária dos tipos utilizados.

1. **Genrmf-Comprido**(rmfl): as dimensões do grafo são controlados por um parâmetro x. Dois outros parâmetros $a = 2^{x/4}$ e $b = 2^{x/2}$ determinam a forma como o grafo é conectado. Ou seja, o grafo gerado contem b conjuntos com a^2 nós. Os nós são organizados em uma malha. Cada nó de um conjunto possui arcos para 4 vizinhos. Conjuntos vizinhos são conectados por arcos que formam um emparelhamento perfeito entre os nós dos conjuntos. As capacidades dos arcos são escolhidas de acordo com os parâmetros $c_1 = 1$ e $c_2 = 10^4$. Os arcos dentro do mesmo conjunto possuem capacidade $c_2 a^2$ e aqueles entre conjuntos são escolhidos uniformemente do intervalo $[c_1, c_2]$;
2. **Genrmf-Largo**(rmfw): é construído da mesma forma que o anterior apenas com valores diferentes para a e b, isto é $a = 2^{2x/5}$ e $b = 2^{x/5}$;

3. **Acíclico-Denso**(ad): para cada nó k, existe um arco para todos os outros numerados com $k+1, k+2, k+3, \ldots, n$. Pode-se notar que a fonte terá $n-1$ vizinhos e todos os nós do grafo possuem um arco para o sumidouro. As capacidades destes arcos são selecionadas aleatoriamente do intervalo $[1, 10^6]$;

4. **AK**(ak): esta classe não é aleatória, ou seja, para um parâmetro k específico, o grafo é gerado deterministicamente com $4k$ nós. Foi criada para ser uma instância ruim para os algoritmos de Goldberg e Tarjan. Sua descrição detalhada pode ser encontrada em [27];

5. **Washington-Nível-Aleatório-Comprido**(wrlgl): o número total de nós é especificado por um parâmetro x ($n = 2^x$). O grafo é uma malha retangular de 64 linhas e 2^{x-6} colunas. Cada nó é conectado com três nós escolhidos aleatoriamente da coluna mais próxima. Este grafo é acíclico. As capacidades dos arcos entre colunas são números entre zero e 10^5;

6. **Washington-Nível-Aleatório-Largo**(wrlgw): o grafo é construído da mesma forma que o anterior, com 2^{x-6} linhas e 64 colunas;

7. **Washington-Moderado-Linear**(wlm): as dimensões deste grafo são: $n = 2^x$, 2^{x-2} linhas e 4 colunas. Assim, a fonte e o sumidouro possuem 4 vizinhos cada. Cada nó $i \in V - \{s, t\}$ possui mais de $\sqrt{n}/4$ arcos para nós escolhidos no intervalo $[i+1, i+\sqrt{n}]$. As capacidades são escolhidas uniformemente entre $[1, 10^6]$. Este grafo também é acíclico.

6.2. Algoritmos de decomposição de fluxo em caminhos

Dentre os algoritmos que decompõem o fluxo máximo de uma rede em caminhos, os melhores são os que trabalham em uma rede em camadas, como visto no Capítulo 4. Isto se verifica também na prática com os algoritmos de Dinitz e de Goldberg e Rao.

Em [27] é possível verificar que o algoritmo de Dinitz é o melhor entre os algoritmos de decomposição de fluxo em caminhos aqui apresentados. Contudo, não é feita uma comparação com o BBF. Por este motivo, a tabela a seguir mostra os tempos dos algoritmos de Dinitz e Goldberg e Rao (BBF). São mostradas as três maiores instâncias - em número de nós. A tabela completa pode ser encontrada em [23].

Tabela 6.1. *Tempos de execução (em segundos) para os algoritmos de Rede em Camadas e Fluxo Bloqueante Binário (BBF), utilizando as instâncias descritas acima. Os melhores tempos estão destacados*

n	m	Dinitz	BBF
rmfl			
9000	41300	92.34	**6.02**
15488	71687	200.31	**15.00**
30589	143364	695.96	**40.30**
rmfw			
8664	40964	91.69	**3.49**
16807	80262	285.35	**8.18**
32768	157696	783.04	**19.32**
ad			
128	8128	**0.23**	0.47
256	32640	**1.08**	2.92
512	130816	**5.02**	16.43
ak			
8191	12295	709.29	**9.91**
16390	24583	3090.65	**38.77**
32774	49159	12942.90	**151.59**
wrlgl			
4098	12224	**6.16**	63.13
8194	24512	**22.53**	229.95
16386	49088	**81.76**	719.52
wrlgw			
8194	24448	**19.15**	171.29
16386	48896	**44.48**	482.81
wlm			
1026	8069.6	**1.16**	34.08
2050	22293.6	**3.11**	165.10
4098	65020	**9.94**	829.28

A partir da Tabela 6.1 verifica-se que o algoritmo de Goldberg e Rao, apesar de possuir a menor complexidade de pior caso, apresenta-se melhor que o algoritmo de Dinitz, considerado o melhor dentro desta abordagem até o aparecimento do algoritmo BBF, em apenas três das sete classes. O algoritmo de Dinitz mostrou-se melhor nas classes de grafos acíclicos.

6.3. Algoritmos de manipulação de pré-fluxo

O melhor algoritmo de manipulação de pré-fluxo, encontrado nos estudos experimentais, era uma variação do algoritmo de Goldberg e Tarjan, chamada de maior rótulo primeiro, apresentada anteriormente. Em [21] e [27], o leitor pode comprovar este resultado. Junto com a heurística de rotulamento global, este e, em alguns casos, a variação FIFO, eram consideradas as melhores na prática.

Contudo, com o aparecimento da algoritmo de Hochbaum, este se mostrou melhor em alguns casos, principalmente, utilizando-se de heurísticas bem específicas para cada uma das classes de instâncias ([23]). Assim, a tabela a seguir mostra uma comparação entre os melhores algoritmos de manipulação de pré-fluxo.

A partir da Tabela 6.2 nota-se que o algoritmo de maior rótulo é o melhor para a maioria das classes de instâncias. Em [6] é apresentada uma comparação mais detalhada e uma maneira de encontrar o melhor conjunto de variações do algoritmo de Hochbaum para cada classe.

Portanto, pode-se concluir que os algoritmos de manipulação de pré-fluxo, em especial, o algoritmo de maior rótulo primeiro e uma parametrização específica do algoritmo de Hochbaum apresentam melhores tempos, na prática, para resolver o problema de fluxo máximo em redes.

Tabela 6.2. *Tempos de execução (em segundos) para as variações do algoritmo de Goldberg e Tarjan - Maior Rótulo(HL) e FIFO - e do algoritmo de Hochbaum - inicialização simples e maior rótulo (PFS). Os melhores tempos estão destacados*

n	m	HL	FIFO	PFS
rmfl				
15488	71687	**0.12**	0.26	0.28
30589	143364	**0.28**	0.79	0.60
65536	310040	**0.69**	2.40	1.57
rmfw				
16807	80262	**0.73**	1.01	1.67
32768	157696	**1.89**	3.06	5.33
63503	307440	**6.83**	10.88	14.84
ad				
512	130816	**0.12**	0.19	0.13
1024	523776	1.21	1.65	**0.62**
2048	2096128	3.89	6.20	**1.79**
ak				
16390	24583	2.80	4.46	**1.34**
32774	49159	10.41	17.41	**5.28**
64006	96007	**41.27**	71.17	43.50
wrlgl				
32770	98240	**0.17**	0.36	0.50
65538	196544	**0.44**	1.28	1.45
131074	393152	**0.96**	3.32	3.89
wrlgw				
32774	97792	**0.32**	0.55	0.75
65538	195584	**1.38**	2.25	2.00
131074	391168	3.18	3.87	**3.12**
wlm				
16386	522249	**0.22**	0.25	0.94
32770	1470473	**0.66**	0.77	3.05
65538	4186093	**2.16**	2.50	9.69

7

Conclusão

Ao final deste trabalho, o leitor pôde perceber que o problema de fluxo máximo em redes tem várias aplicações práticas. Apesar de ser um problema *fácil*, pois existem algoritmos de tempos polinomiais para ele, é extensivamente estudado há muitas décadas e existem vários algoritmos que o solucionam. Um documento reunindo os algoritmos mais importantes e comparando-os, mostrando o atual estado da arte, é uma boa ferramenta para alunos de graduação e especializações.

Outro ponto é a verificação que o algoritmo de Fluxo Bloqueante Binário transpôs a barreira do limite inferior dos algoritmos que utilizam a decomposição de fluxo em caminhos que era, naturalmente, $\Omega(nm)$. Contudo, sua complexidade é fracamente polinomial, já que depende do valor U, então será possível um algoritmo que quebre esta barreira, mas seja fortemente polinomial?

Na questão prática, a variação do algoritmo de Goldberg e Tarjan, maior rótulo, mostra-se mais fácil de implementar e mais eficiente em grande parte das classes de grafos comumente testadas. Sendo, por estes motivos, preferível para implementações sequenciais e paralelas.

Pretende-se dar continuidade a esta pesquisa no sentido de implementar uma biblioteca de funções, onde as abordagens gerais apresentadas no Capítulo 3 possam ser decompostas

em funções parametrizadas, de tal modo que a implementação de determinado algoritmo seja obtida colocando-se parâmetros nas funções gerais. Espera-se ser também possível, experimentar outras combinações de estratégias para produzir algoritmos diferentes. A construção desta biblioteca objetiva uma implementação mais facilitada também em arquiteturas paralelas.

Referências Bibliográficas

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice–Hall, 1992.

[2] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of Association for Computing Machinery*, vol. 35(4), pp. 921–940, 1988.

[3] A. V. Goldberg and S. Rao, "Beyond the flow decomposition barrier," *Journal of Association for Computing Machinery*, vol. 45(5), pp. 783–797, 1998.

[4] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice–Hall, 1982.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press and McGraw–Hill, 1990.

[6] C. L. Anderson, "Implementations os the pseudoflow algoritm for the maximum flow problem," Master's thesis, 2001.

[7] L. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.

[8] R. E. Tarjan, *Data Structures and Network Algorithms.* Filadelfia, PA: Society for Industrial and Applied Mathematics, 1983.

[9] S. Even, *Graph Algorithms.* Rockville, MD: Computer Science Press, 1989.

[10] R. J. Wilson, *Introduction to Graph Theory.* Nova York, NY: Longman Scientific & Technical, 1985.

[11] R. J. Trudeau, *Introduction to Graph Theory.* Nova York, NY: Dover Publications, 1997.

[12] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications.* Nova York, NY: American Elsevier, 1976.

[13] J. L. Szwarcfiter and L. Markenzon, *Estruturas de Dados e seus Algoritmos.* Rio de Janeiro, RJ: Livros Técnicos e Científicos S.A., 1994.

[14] E. Horowitz and S. Sahni, *Fundamentals of Data Structures Using Pascal.* Rockville, MD: Computer Science Press, 1990b.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* São Francisco, CA: W. H. Freeman, 1979.

[16] G. V. R. Viana, "Meta-heurísticas para solução de problemas de otimização combinatória," Dissertação de mestrado, Curso de Computação — Universidade Federal do Ceará, Fortaleza, CE, 1996.

[17] W. L. Winston, *Operations Research: Applications and Algorithms.* Belmont, CA: Duxbury Press, 1993.

[18] J. R. Evans and E. Minieka, *Optimization Algorithms for Networks and Graphs.* E.U.A.: Marcel Dekker Inc., 1992.

[19] A. V. Goldberg, "Recent developments in maximum flow algorithms," Tech. Rep. 98–045, NEC Recsearch Institute, Inc., Princenton, NJ, 1998.

[20] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. Singapore: John Wiley and Sons, Inc., 1990.

[21] R. K. Ahuja, M. Kodialam, A. K. Mishra, and J. Orlin, "Computational investigations of maximum flow algorithms," *European Journal of Operational Research*, vol. 97(3), pp. 509–542, 1997.

[22] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by sucessive approximation," *Mathematics Operational Research*, vol. 15, pp. 430–466, 1990.

[23] T. Hagerup, P. Sanders, and J. L. Träff, "An implementation of the binary blocking flow algorithm," in *Workshop on Algorithm Engineering*, (Saarbrücken, Alemanha), pp. 143–154, Kurt Mehlhorn, 1998.

[24] V. C. Barbosa, *An Introduction to Distributed Algorithms*. Cambridge, MA: MIT Press, 1996.

[25] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, MA: Addison–Wesley, 1992.

[26] D. S. Hochbaum, "The pseudoflow algorithm: A new algorithm and a new simplex algorithm for the maximum flow problem," 1997.

[27] R. J. Anderson and J. C. Setubal, "Goldberg's algorithm for maximum flow in perspective: a computational study," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 12, pp. 1–18, 1993.

[28] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD: Computer Science Press, 1990a.

[29] Y. Shiloach and U. Vishkin, "An $o(n^2 \log n)$ parallel max-flow algorithm," *Journal of Algorithms*, no. 3, pp. 128–146, 1982.

[30] R. J. Anderson and J. C. Setubal, "A parallel implementation of the push-relabel algorithm for the maximum flow problem," *Journal of Parallel and Distributed Computing*, vol. 29(1), pp. 17–26, 1995.

[31] J. C. Setubal, P. F. Macedo, and E. N. Cárceres, "Solving the maximum flow problem in parallel, with distributed memory, and asynchronously," 1998.

[32] C. L. Sales and R. C. Corrêa, "Aspectos combinatórios de redes de interconexão," in *1a. Escola Regional de Informática da Sociedade Brasileira de Computação*, (Nova Friburgo, RJ/ Vitória, ES), pp. 77–100, 1998.