Algoritmos para Emparelhamentos em Grafos Bipartidos[1]

Herbert Alexander Baier Saip[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Cláudio Leonardo Lucchesi (Orientador)[3]
- Yoshiko Wakabayashi[4]
- João Carlos Setubal[3]
- Célia Picinin de Mello (Suplente)[5]

[1]Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]O autor é Bacharel em Ciência da Computação pela Universidade Federal do Ceará.

[3]Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4]Professora do Departamento de Ciência da Computação - IME - USP.

[5]Professora do Departamento de Ciência da Computação - IMECC - UNICAMP.

Algoritmos para Emparelhamentos em Grafos Bipartidos

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Herbert Alexander Baier Saip e aprovada pela Comissão Julgadora.

Campinas, 03 de março de 1993.

Prof. Dr. _____
Cláudio Leonardo Lucchesi

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de MESTRE em Ciência da Computação.

Aos meus pais, irmãos, Cláudio de Carvalho e ao universo.

Agradecimentos

"A pior das instituições gregárias se intitula exército. Eu o odeio. Se um homem puder sentir qualquer prazer em desfilar aos sons de música, eu desprezo este homem... Não merece um cérebro humano, já que a medula espinhal o satisfaz. Deveríamos fazer desaparecer o mais depressa possível este câncer da civilização. Detesto com todas as forças o heroísmo obrigatório, a violência gratuita e o nacionalismo débil. A guerra é a coisa mais desprezível que existe. Preferiria deixar-me assassinar a participar desta ignomínia."

Albert Einstein

Um especial agradecimento ao meu pai, Alfredo Baier, por ter-me incentivado na realização deste mestrado. A minha mãe, Ingrid Saip, pelo seu apoio nas horas em que eu precisava. Aos meus irmãos que sempre foram amigos e companheiros.

Ao professor Lucchesi pelo seu apoio durante o tempo em que trabalhamos juntos. A sua orientação foi indispensável para mim na realização deste trabalho.

Ao professor Setubal por ter-me ajudado nas pesquisas bibliográficas.

Ao grupo de orientandos do professor Lucchesi: Marcelo Carvalho, Marcus Vinícius e Mário Harada. A ajuda deles na correção deste trabalho foi importante.

Aos companheiros de casa: Galileu Batista (Gal), Alfredo Jackson (Cacão), Leonardo Matos (Leo), José Guimarães (Guima), Renato Tutumi (Tutu), Renato Fileto (Filetinho), Frederico Cavalcanti (Fred) e Juliano de Oliveira (Ramon), com os quais compartilhei muito durante este tempo.

A um grande amigo que se encontra num desafio parecido em Belo Horizonte, José Leite Jr (Coalhada).

Aos companheiros deste desafio da UNICAMP, onde as dificuldades enfrentadas nos fizeram ser bons amigos. Entre estes destaco os do volley.

As famílias Barrueto e Guerrero que me ajudaram durante a minha permanência em Campinas.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), a Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) e ao Fundo de Apoio ao Ensino e à Pesquisa (FAEP) que deram o suporte financeiro à execução do Programa de Mestrado.

Abstract

The matching problem in graphs consists in determining a vertex disjoint set M of edges of the graph. In particular, we are interested in finding maximum matchings, that is, matchings of maximum cardinality. There are many variations around this problem, the graph can be: bipartite or general, weighted or not.

In this work we present the main techniques to design the most efficient algorithms that solve the problem of maximum matching, weighted or not, in bipartite graphs. We also describe the main algorithms, sequential and parallel, to solve this problem.

Chapter 2 contains the most important algorithms to solve the problem for non weighted bipartite graphs, namely, the algorithm of Hopcroft and Karp, the parallel algorithm of Kim and Chwa, and the parallel algorithm of Goldberg, Plotkin and Vaidya.

Chapter 3 contains the most important algorithms to solve the problem for weighted bipartite graphs, namely, the algorithm of Edmonds and Karp, the scaling algorithm of Gabow, the scaling and approximation algorithm of Gabow and Tarjan, the parallel algorithm of Goldberg, Plotkin and Vaidya and the parallel algorithm of Gabow and Tarjan.

In Appendix A it is given a table which describes briefly the most important algorithms for solving the general problem, in which the graph is not bipartite.

Sumário

O problema de emparelhamentos em grafos consiste em determinar um conjunto M de arestas do grafo, onde as arestas são disjuntas nos vértices. Em particular, estamos interessados em determinar emparelhamentos máximos, ou seja, de cardinalidade máxima. Existem muitas variações em torno do tema, o grafo pode ser: bipartido ou não, ponderado ou não.

Neste trabalho apresentamos as principais técnicas para se projetar os algoritmos mais eficientes que resolvem o problema de emparelhamentos máximos, ponderados ou não, em grafos bipartidos. Também descrevemos os principais algoritmos, seqüenciais e paralelos, que resolvem este problema.

O Capítulo 2 apresenta os principais algoritmos para resolver o problema em grafos bipartidos não ponderados: o algoritmo de Hopcroft e Karp, o algoritmo paralelo de Kim e Chwa e o algoritmo paralelo de Goldberg, Plotkin e Vaidya.

O Capítulo 3 apresenta os principais algoritmos para resolver o problema em grafos bipartidos ponderados: o algoritmo de Edmonds e Karp, o algoritmo com escalonamento de Gabow, o algoritmo com escalonamento e aproximação de Gabow e Tarjan, o algoritmo paralelo de Goldberg, Plotkin e Vaidya e o algoritmo paralelo de Gabow e Tarjan.

O Apêndice A contém uma tabela dos principais algoritmos para resolver o problema no caso em que os grafos não são bipartidos.

Conteúdo

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No nosso cotidiano existem muitas situações que podem ser representadas por diagramas consistindo de um conjunto de pontos e linhas que ligam aos pares esses pontos. Por exemplo, os pontos poderiam representar cidades e as linhas as estradas entre pares destas cidades; ou os pontos poderiam representar pessoas e as linhas ligando pares de pessoas que se gostam. Os grafos surgiram na matemática exatamente para dar uma representação abstrata para estas situações, onde os pontos são chamados de *vértices* e as ligações de *arestas*.

Dentre todos os problemas tratados na teoria dos grafos estamos interessados em encontrar um emparelhamento máximo de um grafo bipartido G, onde G pode ser ponderado ou não. Por exemplo, suponhamos que os vértices de um grafo representam as pessoas e as arestas a possibilidade de duas pessoas contraírem matrimônio (caso ortodoxo, ou seja, entre pessoas de sexo diferentes). Estamos interessados em responder a seguinte pergunta:

"De que forma podemos realizar o maior número de casamentos?"

Em 1935, Philip Hall enunciou o seu Teorema que é um dos resultados mais significativos sobre emparelhamentos em grafos bipartidos. Este Teorema será apresentado no próximo capítulo.

A proposta original da tese previa, também, o estudo de emparelhamentos máximos em grafos gerais (ponderados ou não), porém, pelo estudo ter-se estendido muito, optamos somente pelo estudo do caso bipartido.

1.1 Algumas Definições

Um grafo $G = (VG, EG)$ é um conjunto finito de vértices VG e outro de arestas EG, onde cada aresta $\alpha \in EG$ associa um par de vértices (não necessariamente distintos) de V. Os *extremos* de uma aresta são o par de vértices que ela associa. A partir de agora vamos denotar o grafo $G = (VG, EG)$ simplesmente por G, VG por V e EG por E. Também denotaremos a cardinalidade de V por $|V| = n$ e a cardinalidade de E por $|E| = m$.

Seja $X \subseteq E$. Dizemos que $\overline{X}$ é o *conjunto complementar* de X se $\overline{X} := E \setminus X$.

Há ocasiões em que é conveniente orientar as arestas do grafo, associando a cada aresta não um par mas sim um par ordenado de vértices: tais grafos são ditos *orientados*.

Os extremos de uma aresta α são ditos *incidentes* em α, e vice versa. Dois vértices que são incidentes na mesma aresta são chamados de *adjacentes*, como também duas arestas que são incidentes no mesmo vértice. Uma aresta que possui extremos iguais é chamada de *laço* e extremos distintos de *ligação*.

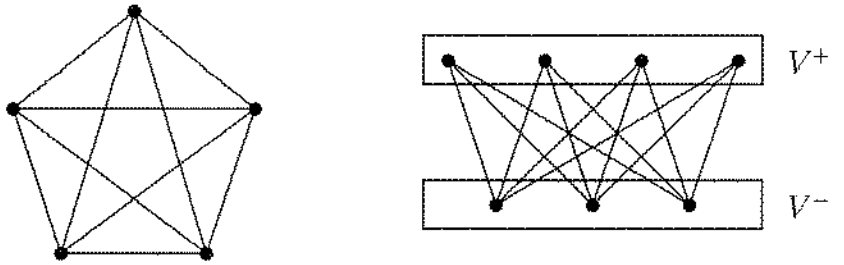
Um grafo H é um *subgrafo* de G se $VH \subseteq V$ e $EH \subseteq E$. Seja $E' \subseteq E$. Dizemos que H é um *grafo gerado* por E', $H := G[E']$, se H é um subgrafo de G, em que o conjunto de vértices são os extremos de E' e o conjunto de arestas E'. De uma forma similar, H pode ser um grafo gerado por V', onde $V' \subseteq V$. O símbolo $\oplus$ denota o *ou exclusivo* (também chamado de *diferença simétrica*) entre conjuntos, ou seja, se X e Y são dois conjuntos, então $X \oplus Y = (X \cup Y) \setminus (X \cap Y)$.

Seja $X \subseteq E$ e $Y \subseteq V$. Os *vizinhos* de Y utilizando as arestas de X, denotado por $X(Y)$, é o conjunto de vértices adjacentes aos vértices de Y no grafo $G[X]$. O subconjunto de arestas de X incidentes nos vértice de Y será denotado por $\widetilde{X}(Y)$. Denotaremos por VX o conjunto de vértices incidentes nas arestas de X.

Definimos o *grau de um vértice* $v \in V$ no grafo G, denotado por $g_G(v)$, como sendo o número de arestas que incidem nele, ou seja, $g_G(v) := |E(\{v\})|$.

Um *passeio* em G é uma seqüência finita não nula $P = (v_0, e_1, v_1, \ldots, e_r, v_r)$, onde:

- $v_j \in V$ e $e_i \in E$.
- $\forall i$, $1 \leq i \leq r$, v_{i-1} e v_i são os extremos de e_i.



K_5 $K_{4,3}$

Figura 1.1: Grafos completos: K_5 e $K_{4,3}$.

Dizemos que v_0 e v_r são os *extremos* de P e o seu *comprimento* é r. Um passeio de comprimento zero é chamado de *degenerado*. Representaremos o conjunto de arestas de P por EP e o de vértices por VP.

Chamamos P de *trilha* se as arestas de P são duas a duas distintas; se, além disso, os vértices também são dois a dois distintos, P é chamado de *caminho*. Porém, se todos os vértices de uma trilha P são dois a dois distintos exceto os extremos que são iguais ($v_0 = v_r$), P é chamado de *circuito*. Uma trilha é dita de *Euler* se e somente se passa por todas as arestas do grafo.

Um grafo G é *bipartido* se o conjunto de vértices V pode ser particionado em dois conjuntos, V^+ e V^-, de modo que toda aresta $\alpha \in E$ possui um extremo em V^+ e o outro em V^-.

Um grafo G é dito *simples*[1] se não possui laços e duas arestas distintas de G não possuem os mesmos extremos. Se G é simples e para cada par de vértices de V existe uma aresta em E que os liga, então denominaremos G de *grafo completo*, denotado por K_n. Se G é simples com bipartição (V^+, V^-), onde cada vértice de V^+ é ligado, através de arestas de G, a todos os vértices de V^-, então denominaremos G de *grafo bipartido completo*, denotado por $K_{|V^+|,|V^-|}$ (veja Figura 1.1).

Um *emparelhamento* no grafo G é um conjunto de arestas $M \subseteq E$ duas a duas não adjacentes e que não são laços. Se alguma aresta de M é incidente a um vértice $v \in V$, v é chamado de *ocupado*; caso contrário, v é chamado

[1]Caso não seja especificado de outra forma, vamos assumir que os grafos utilizados são simples.

de *livre*.

Um conjunto T de arestas *cobre* um conjunto X de vértices se em todo vértice de X incide pelo menos uma aresta de T. Um conjunto X de vértices *cobre* um conjunto T de arestas se em toda aresta de T incide pelo menos um vértice de X. Um conjunto $S \subseteq V$ é chamado de *conjunto independente* em G se nenhuma aresta de G tem os ambos extremos em S.

Seja $\mathcal{C}$ a coleção dos emparelhamentos do grafo G. Um emparelhamento $M \in \mathcal{C}$ é *maximal* se $\forall M' \in \mathcal{C}, M \not\subset M'$; M é *máximo* se $\forall M' \in \mathcal{C}$, $|M| \geq |M'|$. Um emparelhamento é *perfeito* quando todos os vértices de G estão ocupados.

Um grafo G é *ponderado* se existe uma função c sobre as suas arestas, chamada de *custo*, que associa a cada aresta um valor real, ou seja, $c : E \rightarrow \Re$. Seja $X \subseteq E$: definimos o *custo do conjunto* X por

$$c(X) := \sum_{\alpha \in X} c(\alpha).$$

Seja M um emparelhamento de G. Um *caminho M-alternado* em G é um caminho cujas arestas estão alternadamente em $E \setminus M$ e em M. Um *caminho M-aumentante* em G é um caminho M-alternado não degenerado cujos extremos são livres.

Dizemos que dois vértices u e v em G são *conectados* se e somente se existe um caminho entre u e v em G. Seja uma partição de V os subconjuntos não vazios $V_1, \ldots, V_l$ tal que u e v pertencem a V_i se e somente se u e v são conectados. Chamamos de *componentes conexas* de G os subgrafos $G[V_1], \ldots, G[V_l]$. Se G possui uma única componente, G é dito *conexo*; caso contrário, G é dito *desconexo*.

Seja o grafo G com a bipartição V^+ e V^- e um emparelhamento M em G. Dizemos que o sinal de um vértice é *positivo* se pertence a V^+ e *negativo* se pertence a V^-. No caso das arestas, o sinal é *positivo* se ela pertence a M e *negativo* se pertence a $E \setminus M$. Da mesma forma, definimos o sinal de um passeio M-alternado não degenerado como sendo *positivo* (*negativo*) se a sua primeira aresta (e_1) é positiva (negativa). Estendemos esta definição para caminhos degenerados (comprimento zero) dando-lhes o sinal de sua origem.

Em [43] encontram-se outras definições bem como uma pequena lista de livros texto sobre grafos.

1.2 Definições Para Algoritmos Paralelos

O modelo computacional *PRAM* (*Parallel RAM*) foi estendido do modelo *RAM* (*Random Access Machine*) para englobar paralelismo. Este modelo consiste de p ($p \geq 1$) processadores sincronizados tendo acesso a uma memória global comum.

Num modelo PRAM, dizemos que a leitura ou a escrita na memória global é *concorrente* se for permitido o acesso simultâneo pelos processadores a uma mesma posição da memória global; caso contrário é *exclusiva*. Portanto, podemos subdividir o modelo PRAM em (vide [51, 3]):

EREW (*Exclusive Read Exclusive Write*) Neste modelo não são permitidas nem leituras e nem escritas concorrentes na memória global.

CREW (*Concurrent Read Exclusive Write*) Neste modelo só são permitidas leituras concorrentes na memória global.

ERCW (*Exclusive Read Concurrent Write*) Neste modelo só são permitidas escritas concorrentes na memória global.

CRCW (*Concurrent Read Concurrent Write*) Neste modelo são permitidas leituras e escritas concorrentes na memória global.

As operações em paralelo, nos algoritmos, são executadas pelas seguintes instruções [3]:

- Quando for necessário executar muitos passos do algoritmo em paralelo:

  faça em paralelo os passos P_1 **e** P_2 [e P_i]
  início
  P_1: ⟨declarações⟩
  P_2: ⟨declarações⟩
  ⋮
  P_i: ⟨declarações⟩
  fim.

  Ou seja, para cada rótulo P_j, $1 \leq j \leq i$, dentro do escopo do comando, é associado um processador, que executa as instruções, a partir deste rótulo, até encontrar um outro rótulo ou o fim da instrução (em paralelo).

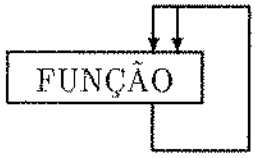


Figura 1.2: Chamada de uma função dividir para conquistar em paralelo.

- Quando muitos processadores executam a mesma instrução simultaneamente, temos dois comandos para representar o paralelismo:

  1. Para cada i, $i \in [j \ldots k]$, é associado um processador P_i:

     para $i \leftarrow j$ **até** k **faça em paralelo.**

  2. Para cada elemento x_i do conjunto X é associado um processador P_i:

     para cada $x_i \in X$ **faça em paralelo.**

O gráfico da Figura 1.2 representa a chamada de uma função dividir para conquistar em paralelo.

1.3 Teorema de Berge

Este Teorema é de fundamental importância para se encontrar emparelhamentos máximos. Muitos algoritmos, como é o caso do algoritmo primitivo do próximo capítulo, o utilizam diretamente para resolver o problema de emparelhamento.

Teorema 1.1 (Berge [7], 1957) *Um emparelhamento M em G é máximo se e somente se G não contém nenhum caminho M-aumentante.*

Demonstração: $\Longrightarrow$ "Se existe um caminho M-aumentante, então M não é um emparelhamento máximo". Seja M um emparelhamento e P um caminho M-aumentante. O conjunto $M^* := M \oplus EP$ é um emparelhamento de cardinalidade $|M| + 1$. Assim M não é um emparelhamento máximo.

$\Longleftarrow$ "Se M não é um emparelhamento máximo, então existe um caminho M-aumentante". Seja N um emparelhamento com $|N| > |M|$. Gere um novo grafo H da seguinte forma:

$$H := G[N \oplus M]$$

Note que o grau dos vértices de H é menor do que ou igual a dois, pois tanto as arestas de N quanto as de M incidem no máximo uma vez em cada vértice de V. Seja $\mathcal{C} := \{K_1, \ldots, K_l\}$ a coleção das componentes conexas de H. Cada componente $K_i \in \mathcal{C}$ pode ser:

1. Um ciclo de comprimento par com arestas alternadamente em $N \setminus M$ e em $M \setminus N$.

2. Um caminho com arestas alternadamente em $N \setminus M$ e em $M \setminus N$.

Como $|N| > |M|$, então $\exists i$, $1 \leq i \leq l$, tal que $|(N \setminus M) \cap EK_i| > |(M \setminus N) \cap EK_i|$, ou seja, existe uma componente K_i que possui mais arestas de N do que de M. Logo, K_i é um caminho que começa e termina com arestas de N; portanto, K_i é um caminho M-aumentante. □

Da demonstração do Teorema 1.1, obtemos os seguintes corolários:

Corolário 1.2 *Sejam M e N emparelhamentos em G, $r = |M|$, $s = |N|$ e $r < s$. Então $G[M \oplus N]$ contém no mínimo $s - r$ caminhos que são M-aumentantes e disjuntos nos vértices; ademais, pelo menos um dos caminhos tem comprimento $\leq 1 + 2\lfloor r/(s-r) \rfloor$.*

Demonstração: Da demonstração do Teorema 1.1, temos que $G[M \oplus N]$ contém no mínimo $s - r$ caminhos que são M-aumentantes disjuntos nos vértices; portanto, os $s - r$ caminhos são disjuntos nas arestas. Todos os caminhos podem possuir no máximo r arestas de M, logo existe um caminho M-aumentante com no máximo $\lfloor r/(s-r) \rfloor$ arestas de M, ou seja, o caminho possui no máximo $1 + 2\lfloor r/(s-r) \rfloor$ arestas. □

Corolário 1.3 *Sejam M um emparelhamento maximal e M^* um emparelhamento máximo, ambos em G. A seguinte desigualdade é válida,*

$$|M^*| \leq 2|M|.$$

Demonstração: A desigualdade é obviamente verdadeira se M for máximo. Portanto, podemos supor que $|M| < |M^*|$. Pela maximalidade de M, todos os caminhos M-aumentantes têm comprimento no mínimo três.

Assim, pelo Corolário 1.2, existe um caminho M-aumentante em $G[M \oplus M^*]$ de comprimento no máximo $1 + 2\left\lfloor \frac{|M|}{|M^*|-|M|} \right\rfloor$. Logo,

$$3 \leq 1 + 2\left\lfloor \frac{|M|}{|M^*| - |M|} \right\rfloor,$$

daí a desigualdade enunciada. □

1.4 Grafos Ponderados

Em grafos ponderados existe uma grande variedade de problemas distintos, entre os quais podemos destacar:

(i) Emparelhamento perfeito de custo mínimo - entre todos os emparelhamentos perfeitos de um grafo, escolhemos um de custo mínimo.

(ii) Emparelhamento de custo mínimo - entre todos os emparelhamentos de um grafo, escolhemos um de custo mínimo.

(iii) Emparelhamento máximo de custo mínimo - entre todos os emparelhamentos máximos de um grafo, escolhemos um de custo mínimo.

(iv) Emparelhamento perfeito de custo máximo - entre todos os emparelhamentos perfeitos de um grafo, escolhemos um de custo máximo.

(v) Emparelhamento de custo máximo - entre todos os emparelhamentos de um grafo, escolhemos um de custo máximo.

(vi) Emparelhamento máximo de custo máximo - entre todos os emparelhamentos máximos de um grafo, escolhemos um de custo máximo.

Proposição 1.4 *Os seguintes problemas são equivalentes: (i) e (iv), (ii) e (v) e também o são (iii) e (vi).*

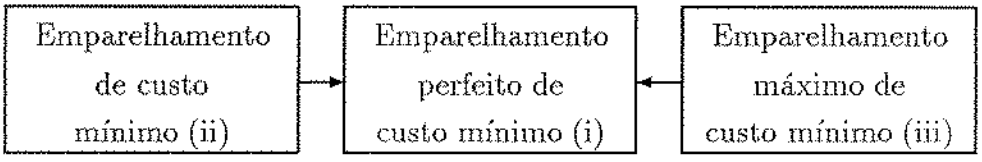


Figura 1.3: Reduções de problemas em grafos ponderados.

Demonstração: A redução que leva um problema a outro é simplesmente a complementação dos custos das arestas (substituição pelo valor simétrico). □

Mostraremos a seguir que os Problemas (ii) e (iii) se reduzem ao Problema (i) (veja Figura 1.3).

Proposição 1.5 *O Problema (ii) é polinomialmente redutível ao Problema (i). Ademais, caso o grafo do Problema (ii) seja bipartido, esta propriedade é preservada pela redução.*

Demonstração: Como o Problema (i) determina um emparelhamento perfeito, logo precisamos garantir que os grafos obtidos nas reduções admitam emparelhamentos perfeitos. Seja G o grafo em questão. Se G é bipartido, com bipartição V^+ e V^-, então imergimos G num grafo bipartido completo $K_{r,r}$, onde r é o valor máximo entre $|V^+|$ e $|V^-|$. Se G não é bipartido, então imergimos G num grafo completo K_{2r}, onde $r := \left\lceil \frac{|V|}{2} \right\rceil$. Por simplicidade, denotaremos K o grafo $K_{r,r}$ ou K_{2r}, conforme o caso.

Definimos a função custo $c_K : EK \to \Re$ da seguinte forma:

$$c_K(\alpha) := \begin{cases} c_G(\alpha) & \text{se } \alpha \in E \text{ e } c_G(\alpha) < 0 \\ 0 & \text{caso contrário.} \end{cases}$$

Seja M_K um emparelhamento perfeito de c_K-custo mínimo em K e $M_G := \{\alpha : \alpha \in M_K \cap E \text{ e } c_G(\alpha) < 0\}$. Vamos mostrar que M_G é um emparelhamento de c_G-custo mínimo em G.

Seja N_G um emparelhamento em G. Como K é um grafo completo, então existe um emparelhamento perfeito N_K em K, onde $N_G \subseteq N_K$. Como N_G

pode possuir arestas com custos positivos (o custo destas arestas é zero em N_K), então $c_K(N_K) \leq c_G(N_G)$. Portanto,

$$c_G(M_G) \overset{\text{def } M_G}{=} c_K(M_K) \overset{\text{def } M_K}{\leq} c_K(N_K) \leq c_G(N_G). \square$$

Proposição 1.6 *O Problema (iii) é polinomialmente redutível ao Problema (i). Ademais, caso o grafo do Problema (iii) seja bipartido, esta propriedade é preservada pela redução.*

Demonstração: O grafo K, com $2r$ vértices, é definido exatamente como na Proposição 1.5.

Definimos a função custo $c_K : EK \to \Re$ da seguinte forma:

$$c_K(\alpha) := \begin{cases} c_G(\alpha) & \text{se } \alpha \in E \\ 2rU & \text{caso contrário,} \end{cases}$$

onde $U := 1 + \max\{|c_G(\alpha)| : \alpha \in E\}$.

Seja M_K um emparelhamento perfeito de c_K-custo mínimo em K e $M_G := M_K \cap E$. Vamos mostrar que M_G é um emparelhamento máximo de c_G-custo mínimo em G.

Como K é um grafo completo, então para todo emparelhamento S_G em G existe um emparelhamento perfeito S_K em K tal que $S_G \subseteq S_K$. Ademais, por definição de c_K,

$$c_K(S_K) = c_G(S_G) + 2rU\,(|S_K| - |S_G|) \tag{1.1}$$

Por outro lado, por definição de U,

$$|c_G(S_G)| < rU. \tag{1.2}$$

Seja N_G um emparelhamento em G. Aplicando (1.1) a N_K e M_K temos:

$$\begin{array}{rll} 0 & \overset{\text{def } M_K}{\leq} & c_K(N_K) - c_K(M_K) \\ & \overset{(1.1)}{=} & c_G(N_G) - c_G(M_G) + 2rU\,(|M_G| - |N_G|)\,. \end{array} \tag{1.3}$$

Aplicando (1.2) a N_K e M_K temos:

$$c_G(N_G) - c_G(M_G) < 2rU. \tag{1.4}$$

De (1.3) e (1.4) temos que

$$0 < 2rU\,(|M_G| - |N_G| + 1)$$

e portanto $|N_G| \leq |M_G|$. De fato, M_G é máximo.

Agora demonstraremos que M_G é de custo mínimo. Suponha que N_G é um emparelhamento máximo em G, ou seja, $|M_G| = |N_G|$. Logo, usando a desigualdade em (1.3) temos que:

$$0 \leq c_G(N_G) - c_G(M_G).$$

De fato, $c_G(M_G) \leq c_G(N_G)$. □

Proposição 1.7 *Seja c' uma função custo definida da seguinte forma:*

$$c'(\alpha) := c(\alpha) - a,\ \forall \alpha \in E,$$

onde c é uma função custo em G que assume valores no intervalo $[a, b]$ (ou seja, $c'(\alpha) \in [0, b-a]$). Se M e N são emparelhamentos de mesma cardinalidade em G, então $c(M) - c(N) = c'(M) - c'(N)$.

Demonstração:

$$c(M) - c(N) = c'(M) + a\,|M| - (c'(N) + a\,|N|) = c'(M) - c'(N).\ \square$$

Corolário 1.8 *Se G é um grafo ponderado onde a função custo c pode assumir valores negativos, então podemos definir uma nova função custo não negativa c' tal que M é um emparelhamento máximo de c-custo mínimo se e somente se M é um emparelhamento máximo de c'-custo mínimo.* □

1.5 Resumo dos Demais Capítulos

Nos Capítulos 2 e 3 são apresentados os principais algoritmos sobre emparelhamentos máximos em grafos bipartidos dando ênfase à sua complexidade, bem como o esclarecimento de alguns conceitos importantes na área.

No Capítulo 2 o estudo se restringirá ao caso de grafos bipartidos não ponderados (a Tabela 1.1 mostra a relação dos algoritmos estudados com as

Algoritmos Seqüenciais			
Data	Autor(es)	Ref.	Complexidade
	Primitivo		$mn + n^2$
1973	Hopcroft e Karp	[35]	$mn^{1/2} + n^{3/2}$

Algoritmos Paralelos				
Data	Autor(es)	Ref.	Complexidade	Processadores
1987	Kim e Chwa	[39]	$n \log n \log\log n$	$n^3/\log n$
1988	Goldberg, Plotkin e Vaidya	[32]	$n^{2/3}\log^3 n$	$n^3/\log n$

Tabela 1.1: Grafos bipartidos não ponderados.

suas respectivas complexidades[2]). No Capítulo 3 será estudado o caso para grafos bipartidos ponderados (Tabela 1.2).

No Apêndice B descrevemos alguns algoritmos paralelos. No Apêndice A relacionamos os principais algoritmos que resolvem o problema de emparelhamentos máximos em grafos gerais (não bipartidos).

[2]Nas Tabelas 1.1 e 1.2, p representa o número de processadores que o algoritmo utiliza e U o maior valor absoluto entre todos os custos das arestas do grafo. Por problemas de espaço nas tabelas, omitimos o símbolo $O()$ nos itens de complexidade e processadores.

Algoritmos Seqüenciais			
Data	Autor(es)	Ref.	Complexidade
1955	Kuhn	[41]	$mn^2 + n^3$
1972	Edmonds e Karp	[17]	$mn \log n$
1984	Fredman e Tarjan	[17, 22]	$mn + n^2 \log n$
1985	Gabow	[25]	$mn^{3/4} \log U + n^{7/4} \log U$
1989	Gabow e Tarjan	[27]	$mn^{1/2} \log nU$

Algoritmos Paralelos				
Data	Autor(es)	Ref.	Complexidade	Processadores
1988	Goldberg, Plotkin e Vaidya	[32]	$n^{2/3} \log^3 n \log nU$	$n^3 / \log n$
1988	Gabow e Tarjan	[26]	$(mn^{1/2}/p) \log p \log nU$	$m/(n^{1/2} \log^2 n)$

Tabela 1.2: Grafos bipartidos ponderados.

Capítulo 2

Grafos Bipartidos Não Ponderados

Neste capítulo apresentaremos alguns algoritmos que encontram emparelhamentos máximos quando os grafos são bipartidos não ponderados. Na Seção 2.1 será apresentado um algoritmo primitivo que usa basicamente o Teorema 1.1 (Berge). Na Seção 2.2 é apresentado o algoritmo de Hopcroft e Karp [35], que desde 1973 continua sendo o mais eficiente conhecido. É importante notar porém que para grafos densos há um algoritmo melhor, de complexidade $O\left(n^{3/2}\sqrt{(m+n)/\log n}\right)$, de Alt, Blum, Mehlhorn e Paul [4]. Nas Seções 2.3, 2.4 e 2.5 são apresentados três algoritmos paralelos, respectivamente o algoritmo de Kim e Chwa [39], o algoritmo de Goldberg, Plotkin e Vaidya [32] e o algoritmo de Goldberg, Tarjan, Plotkin e Vaidya [34, 32]; a propósito, o algoritmo da Seção 2.4 é sublinear.

2.1 Algoritmo Primitivo

- Tipo: seqüencial.
- Complexidade: $O(mn + n^2)$.

Dado que o grafo é bipartido, um algoritmo primitivo se deriva diretamente do Teorema 1.1 (Berge).

Os autores Ford e Fulkerson [21], em 1956, apresentaram um algoritmo que também usa uma noção de caminhos aumentantes para o problema de fluxos máximos.

2.1.1 Fundamentos

Seja G um grafo com bipartição de vértices V^+ e V^-. Estamos interessados em decidir se existe um emparelhamento que cubra V^+. A condição para a existência de tal emparelhamento se encontra a seguir:

Teorema 2.1 (König, 1931 e Hall, 1935) *Seja $G = (V, E)$ um grafo bipartido com bipartição V^+ e V^-. Então, G contém um emparelhamento que cobre V^+ se e somente se*

$$|E(X)| \geq |X|, \forall X \subseteq V^+,$$

onde $|E(X)|$ é o número de vértices adjacentes aos vértices de X no grafo G.

Demonstração: $\Longrightarrow$ Seja M um emparelhamento que cobre V^+ e $X \subseteq V^+$. Como os vértices de X são extremos de arestas (distintas) do emparelhamento M, então $|E(X)|$ é no mínimo $|X|$.

$\Longleftarrow$ Seja M um emparelhamento máximo. Seja U o conjunto dos vértices M-livres de V^+. Seja K_i o conjunto dos vértices conectados, a uma distância i, a U através de caminhos M-alternados. Seja S o conjunto dos vértices que pertencem aos $K_{i's}$ e a V^+ e seja T o conjunto dos vértices que pertencem aos $K_{i's}$ e a V^- (veja Figura 2.1, onde as linhas contínuas representam arestas do emparelhamento e as tracejadas arestas fora do emparelhamento[1]). Observe que não existem arestas do tipo (u, v), onde $u \in S$ e $v \in V^- \setminus T$. Como

[1] De agora em diante será usada esta convenção nas figuras.

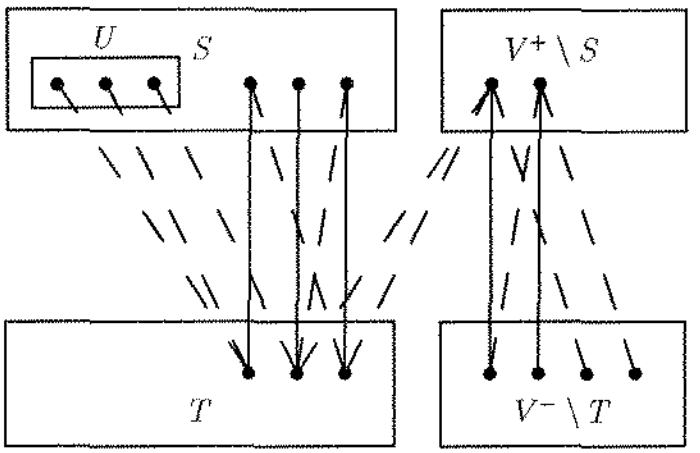


Figura 2.1: Caminhos alternados a partir de U

os únicos vértices livres de todos os $K_{i's}$ são os de K_0 ($K_0 = U$), pois não existem caminhos M-aumentantes, temos que:

$$|S| - |U| = |T|\,.$$

Dado que cada vértice em $E(S)$ é conectado a algum vértice de U através de um caminho M-alternado, então $T = E(S)$. Logo,

$$|E(S)| = |T| = |S| - |U|\,.$$

Por hipótese, $|S| \leq |E(S)|$ e portanto U é vazio. Ou seja, M cobre V^+. □

Na verdade, da demonstração do Teorema 2.1, temos o seguinte corolário, no caso em que não existem mais caminhos M-aumentantes e calculamos S e T, independente da validade ou não da desigualdade enunciada:

Corolário 2.2 *Sejam M^* e C_*, respectivamente, um emparelhamento máximo e uma cobertura mínima das arestas por vértices em G. Então, $|M^*| = |C_*|$. Ademais, se S e T denotam os conjuntos definidos na demonstração do Teorema 2.1, com M^* no papel de M, então $(V^+ \setminus S) \cup T$ é uma cobertura mínima das arestas por vértices.*

Demonstração: Sejam M um emparelhamento e C uma cobertura, então C possui pelo menos uma extremidade das arestas de M. Portanto, para qualquer M e C, temos que:

$$|M| \leq |C|.$$

Em particular, se $|M| = |C|$, então $|M| = |M^*|$ e $|C| = |C_*|$, pois:

$$|M| \leq |M^*| \leq |C_*| \leq |C|.$$

Pela demonstração do Teorema 2.1, $E(S) = T$ e portanto $E(V^- \setminus T) \subseteq V^+ \setminus S$. Logo, $(V^+ \setminus S) \cup T$ é uma cobertura. Além disso, todos os vértices de $V^+ \setminus S$ e de T são ocupados, ou seja, $|(V^+ \setminus S) \cup T| = |M|$. □

2.1.2 Descrição

O algoritmo primitivo que veremos é uma versão algorítmica do Teorema 2.1 (Kõnig e Hall), onde o conjunto $(V^+ \setminus S) \cup T$ nos dá uma certidão da otimalidade do emparelhamento encontrado (Corolário 2.2).

A cada iteração do algoritmo é feita uma busca, a partir dos vértices M-livres de V^+, à procura de um caminho M-aumentante. Quando for encontrado um caminho M-aumentante P, M é substituído por $M \oplus EP$, antes de se procurar o próximo caminho. Quando aumentamos o número de arestas do emparelhamento, estamos também diminuindo o número de vértices livres de V^+ e de V^-.

Portanto, o nosso problema se reduz a encontrar caminhos M-aumentantes. Para encontrar um caminho M-aumentante usamos o fato do grafo ser bipartido, que nos permite utilizar a busca em largura. Esta busca é realizada a partir dos vértices M-livres de V^+, onde nos níveis ímpares da busca vamos utilizar as arestas fora do emparelhamento e nos níveis pares as arestas do emparelhamento (lembre-se que estamos procurando um caminho M-aumentante). Note que como o grafo é bipartido, vértices de mesmo nível nunca são adjacentes. A busca pode terminar por dois motivos:

1. Encontramos um vértice livre de V^-, ou seja, determinamos um caminho M-aumentante. Portanto, pelo Teorema 1.1 (Berge), M não é máximo.

2. A busca terminou e não encontramos um vértice livre de V^-, ou seja, não existem mais caminhos M-aumentantes. Portanto, pelo Teorema 1.1 (Berge), M é máximo.

O Algoritmo 2.1 apresenta a busca em largura. No caso de não se conseguir visitar nenhum vértice livre em V^-, os conjuntos S e T nos fornecem uma certidão da otimalidade de M (Corolário 2.2). No caso de se conseguir visitar um vértice livre em V^-, a busca é interrompida, pois, nesse caso, já foram determinados os elementos para a obtenção do caminho aumentante. De fato o Algoritmo 2.2 determina, nesse caso, um tal caminho.

O Algoritmo 2.3 (primitivo), dado um emparelhamento inicial M_0 (possivelmente vazio) em G, determina um emparelhamento máximo e uma cobertura mínima em G.

Na Figura 2.2 é apresentado um esquema das chamadas de funções deste algoritmo.

/* Dado um grafo G com bipartição (V^+,V^-) e um emparelhamento M em G, determina se existe ou não um caminho M-aumentante através de uma busca em largura.

- Em caso afirmativo, devolve $\langle$sim, K, $r\rangle$, onde K é o vetor das camadas dos vértices visitados na busca em largura e r o nível da última camada.
- Em caso negativo, devolve $\langle$não, S, $T\rangle$, onde S e T são os conjuntos de vértices visitados de V^+ e V^-, respectivamente.

A complexidade do algoritmo é $O(m+n)$. */

```
BuscaLargura (G, M)
  S ← ∅; /* Conjunto dos vértices visitados de V+ */
  T ← ∅; /* Conjunto dos vértices visitados de V- */
  U ← vértices livres de V+;
  K0 ← U;
  i ← 0;
  enquanto Ki ≠ ∅ faça
  início
      se ímpar (i + 1) então /* De V+ para V- */
      início
          Ki+1 ← M̄(Ki) \ T; /* M̄ = E \ M */
          T ← T ∪ Ki+1;
          se Ki+1 contém vértice livre de V- então
              devolva ⟨sim, K, i + 1⟩;
      fim
      senão /* De V- para V+ */
      início
          Ki+1 ← M(Ki) \ S;
          S ← S ∪ Ki+1;
      fim
      i ← i + 1;
  fim
  devolva ⟨não, S, T⟩;
```

Algoritmo 2.1: Busca em largura para determinação da existência ou não de caminho M-aumentante.

/* Dado um grafo G com bipartição (V^+,V^-), um emparelhamento M em G, um vetor K das camadas dos vértices visitados na busca em largura e o nível r da última camada, determina um caminho M-aumentante, de comprimento r, com base na busca em largura efetuada.
A complexidade do algoritmo é $O(m+n)$. */

DetCaminho (G, M, K, r)
  $v_r \leftarrow$ vértice livre em K_r;
  para $i \leftarrow r-1$ **descendo até** 0 **faça**
    se ímpar (i) **então** /* De V^+ para V^- */
    início
      $v_i \leftarrow$ vértice em $K_i \cap M(v_{i+1})$;
      $e_{i+1} \leftarrow$ aresta (v_i, v_{i+1}) em M;
    fim
    senão /* De V^- para V^+ */
    início
      $v_i \leftarrow$ vértice em $K_i \cap \overline{M}(v_{i+1})$;
      $e_{i+1} \leftarrow$ aresta (v_i, v_{i+1}) em $\overline{M}$;
    fim
  devolva $(v_0, e_1, v_1, \ldots, e_r, v_r)$;

Algoritmo 2.2: Determina um caminho M-aumentante com base na busca em largura efetuada.

/* Dado um grafo G com bipartição (V^+,V^-) e um emparelhamento inicial M_0 (possivelmente vazio) em G, é retornado um emparelhamento máximo M e uma cobertura mínima $(V^+ \setminus S) \cup T$ em G. De acordo com a busca em largura, pode ocorrer:

- Encontramos um caminho aumentante, onde usamos K (vetor das camadas dos vértices visitados na busca em largura) e r (nível da última camada da busca em largura) para determinar um caminho aumentante.
- Não encontramos um caminho aumentante, onde $(V^+ \setminus S) \cup T$ nos dá uma certidão da otimalidade de M (S e T são os conjuntos de vértices visitados de V^+ e V^-na busca em largura, respectivamente).

A complexidade do algoritmo é $O((m+n)(|M^*| - |M_0|))$, onde M^* é um emparelhamento máximo em G. */

Primitivo (G, M_0)
  $M \leftarrow M_0$;
  repita
    $\langle \exists$cam, K ou S, r ou $T\rangle \leftarrow$ BuscaLargura (G, M);
    se $\exists$cam **então**
    início
      $P \leftarrow$ DetCaminho (G, M, K, r);
      $M \leftarrow M \oplus EP$;
    fim
  até que $\nexists$cam;
  devolva $(M, (V^+ \setminus S) \cup T)$;

Algoritmo 2.3: Determina um emparelhamento máximo e uma cobertura mínima em G.

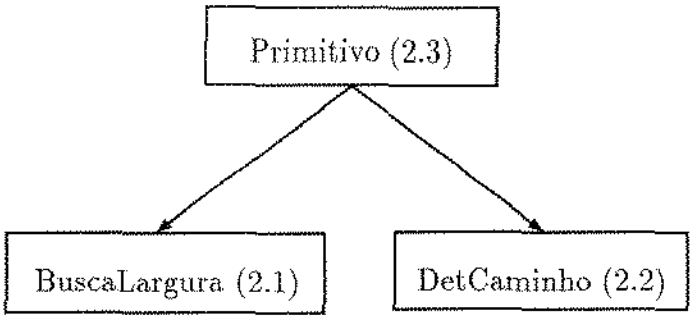


Figura 2.2: Esquema das chamadas das funções do algoritmo primitivo.

2.1.3 Complexidade

A cada caminho aumentante encontrado, a cardinalidade do emparelhamento aumenta de 1 (um). Portanto, o número de buscas em largura é $1 + |M^*| - |M_0|$ e o número de caminhos aumentantes é $|M^*| - |M_0|$, onde M^* é um emparelhamento máximo em G e M_0 é um emparelhamento em G fornecido ao Algoritmo 2.3 (primitivo). Assim, temos $O(1 + |M^*| - |M_0|)$ buscas em largura e determinações de caminhos. Cada busca em largura e cada determinação de caminho gasta $O(m + n)$, portanto, a complexidade do algoritmo é

$$O((m + n)(1 + |M^*| - |M_0|)).$$

Portanto, se o emparelhamento inicial for vazio ($|M_0| = 0$), então a complexidade deste algoritmo será ($|M^*| < n$)

$$O((m + n)|M^*|) = O((m + n)n).$$

2.2 Algoritmo de Hopcroft e Karp

- Autores: Hopcroft e Karp [35].
- Data: 1973.
- Tipo: seqüencial.
- Complexidade: $O\left(mn^{1/2} + n^{3/2}\right)$.

Ao contrário do Algoritmo Primitivo (Seção 2.1), ao invés de em cada iteração determinarmos um único caminho M-aumentante, procuramos uma coleção maximal de caminhos M-aumentantes mínimos disjuntos. Este é o algoritmo seqüencial mais eficiente que se conhece[2].

Os autores Even e Tarjan, em [19], observaram a semelhança deste algoritmo com o algoritmo de Dinic [14].

2.2.1 Fundamentos

Seja $M_0, M_1, \ldots, M_h$ a seqüência de emparelhamentos obtidos no Algoritmo 2.3 (Seção 2.1), onde $M_0 = \emptyset$ e M_h é um emparelhamento máximo, de cardinalidade h ($|M_i| = i$, $i = 0 \ldots h$). Note que a busca em largura (Algoritmo 2.1) determina sempre um caminho aumentante de comprimento mínimo, pois na busca em largura:

1. Os vértices de um nível i da busca se encontram a uma distância i, através de caminhos alternados, de vértices livres de V^+.
2. A busca é interrompida quando encontramos o primeiro nível com vértices livres de V^-.

[2]Os autores Alt, Blum, Mehlhorn e Paul, em [4], apresentam uma variação do Algoritmo de Hopcroft e Karp que possui uma complexidade

$$O\left(n^{3/2}\sqrt{(m+n)/\log n}\right).$$

Assim, para grafos densos ($m = O\left(n^2\right)$), este algoritmo melhora a complexidade do Algoritmo de Hopcroft e Karp num fator de $\sqrt{\log n}$.

Portanto, seja $M_i := M_{i-1} \oplus EP_i$, onde P_i é determinado na busca em largura. Então, P_i é um caminho M_{i-1}-aumentante mínimo.

Nesta seqüência de caminhos aumentantes mínimos temos que $|EP_i| \leq |EP_{i+1}|$, com igualdade se e somente se P_i e P_{i+1} são disjuntos. Veja o teorema a seguir:

Teorema 2.3 *Sejam M um emparelhamento, P um caminho M-aumentante de menor comprimento e P' um caminho $(M \oplus EP)$-aumentante, então:*

$$|EP'| \geq |EP| + 2\,|EP \cap EP'|\,.$$

Demonstração: Observe inicialmente que

$$|EP'| = |EP \oplus EP'| + 2\,|EP \cap EP'| - |EP|\,.$$

Faça $N := M \oplus EP \oplus EP'$. Então $|N| = |M| + 2$, pois $|M \oplus EP| = |M| + 1$. Portanto, pelo Corolário 1.2 temos que $G[M \oplus N]$ contém no mínimo dois caminhos M-aumentantes disjuntos, P_1 e P_2. Ademais $EP \oplus EP' = M \oplus N$, portanto

$$\begin{aligned} |EP \oplus EP'| &= |M \oplus N| \\ &\geq |EP_1 \oplus EP_2| \\ &= |EP_1| + |EP_2|\,. \end{aligned}$$

Como P é um caminho M-aumentante de menor comprimento, então

$$|EP_1| \geq |EP|$$

e

$$|EP_2| \geq |EP|\,.$$

Das desigualdades acima segue o Teorema. □

Das observações feitas acima, Hopcroft e Karp notaram que é possível computar todos os caminhos de um patamar de uma só vez, onde os *caminhos de um patamar* são os caminhos aumentantes de mesmo comprimento. Hopcroft e Karp provaram que o número de patamares no Algoritmo 2.3 (Primitivo) é $O(\sqrt{n})$. Veja o teorema a seguir:

Teorema 2.4 (Hopcroft e Karp, 1973) *Seja h a cardinalidade de um emparelhamento máximo. O número de inteiros distintos na seqüência*

$$|EP_0|, |EP_1|, \ldots, |EP_h|$$

é no máximo $2\left\lfloor\sqrt{h}\right\rfloor$.

Demonstração: Faça $r := \left\lfloor h - \sqrt{h}\right\rfloor$, vamos provar que para o emparelhamento M_r, de cardinalidade r, muitos dos caminhos aumentantes que o geraram possuíam o mesmo tamanho (tamanhos pequenos) e que a partir deste emparelhamento restam poucos caminhos aumentantes.

Pelo Corolário 1.2 temos que:

$$\begin{aligned}
|EP_{r+1}| \leq 1 + 2\lfloor r/(h-r)\rfloor &= 1 + 2\left\lfloor \frac{\left\lfloor h - \sqrt{h}\right\rfloor}{h - \left\lfloor h - \sqrt{h}\right\rfloor}\right\rfloor \\
&= 1 + 2\left\lfloor \frac{\left\lfloor h - \sqrt{h}\right\rfloor}{\left\lceil \sqrt{h}\right\rceil}\right\rfloor \\
&\leq 1 + 2\left\lfloor \frac{h - \sqrt{h}}{\sqrt{h}}\right\rfloor \\
&= 2\left\lfloor\sqrt{h}\right\rfloor - 1.
\end{aligned}$$

Ou seja, para cada $i \leq r+1$, $|EP_i|$ é um dos $\left\lfloor\sqrt{h}\right\rfloor$ inteiros positivos ímpares menores do que ou iguais a $2\left\lfloor\sqrt{h}\right\rfloor - 1$. Porém, o número de inteiros distintos na seqüência $|EP_{r+2}|, \ldots, |EP_h|$ é no máximo $h - (r+2) + 1 = h - \left\lfloor h - \sqrt{h}\right\rfloor - 1 = \left\lceil\sqrt{h}\right\rceil - 1$. Portanto, o número total de inteiros distintos é no máximo $\left\lfloor\sqrt{h}\right\rfloor + \left\lceil\sqrt{h}\right\rceil - 1 \leq 2\left\lfloor\sqrt{h}\right\rfloor$. □

Um importante corolário, obtido a partir do Teorema 2.3, é apresentado a seguir. Nele provamos que os caminhos de um mesmo patamar são disjuntos nos vértices. Isto nos possibilita determinar estes caminhos em tempo linear.

Corolário 2.5 *Se $|EP_i| = |EP_j|$ e $i < j$, então P_i e P_j são disjuntos.*

Demonstração: Prova por absurdo. Suponha que $|EP_i| = |EP_j|$, $i < j$ e P_i e P_j não são disjuntos. Tome k e l, $i \leq k < l \leq j$, de tal forma que:

1. P_k e P_l não são disjuntos.

2. $\forall m$, $k < m < l$, P_m é disjunto de P_k e de P_l.

Então, P_l é um caminho $(M_{k-1} \oplus EP_k)$-aumentante e, pelo Teorema 2.3, $|EP_l| \geq |EP_k| + 2|EP_l \cap EP_k|$.

Por outro lado, como $|EP_i| = |EP_j|$, então pelo Teorema 2.3 temos que $|EP_k| = |EP_l|$. Portanto, P_k e P_l são disjuntos, contradição. □

2.2.2 Descrição

Como a busca em largura (Algoritmo 2.1) não termina quando encontramos o primeiro caminho M-aumentante, mas sim quando terminamos o nível corrente da busca, então o vetor K, da busca em largura, contém todos os possíveis caminhos M-aumentantes de comprimento mínimo. Com base nas camadas K_i obtidas nessa busca em largura, o algoritmo de Hopcroft e Karp determina uma coleção maximal de caminhos M-aumentantes mínimos disjuntos (caminhos de um patamar de comprimento mínimo). Esta determinação é feita através de uma busca em profundidade a partir de vértices livres de V^- visitados na busca em largura. Para garantir a linearidade da complexidade desta busca em profundidade, nenhum vértice é visitado mais do que uma vez. Para isto, utilizamos um conjunto X de vértices ditos *proibidos*. Inicialmente X é vazio. Durante a busca, cada vértice em V^+ visitado é adicionado a X. Esta adição se justifica pois:

- Ou o vértice é utilizado pelo caminho recém encontrado, e portanto, como os caminhos são disjuntos nos vértices, este vértice não pertence a nenhum outro caminho desta coleção maximal.

- Ou é impossível através deste vértice chegar a um vértice livre de V^+, usando um caminho aumentante de comprimento mínimo.

No Algoritmo 2.4 é determinada uma coleção maximal de caminhos M-aumentantes disjuntos de comprimento mínimo, com base na busca em largura efetuada. O Algoritmo 2.5 (Hopcroft e Karp), dado um emparelhamento inicial M_0 (possivelmente vazio) em G, determina um emparelhamento máximo e uma cobertura mínima. Na Figura 2.3 é apresentado um esquema das chamadas de funções deste algoritmo.

/* Dado um grafo G com bipartição (V^+,V^-), um emparelhamento M em G, um vetor K das camadas dos vértices visitados na busca em largura e o nível r da última camada, determina uma coleção maximal de caminhos aumentantes disjuntos, todos de comprimento mínimo (igual a r), com base na busca em largura efetuada.
A complexidade do algoritmo é $O(m+n)$. */

DetColeçãoMaximal (G, M, K, r)
  $\mathcal{P} \leftarrow \emptyset$; /* Coleção maximal de caminhos aumentantes disjuntos,
        de comprimento mínimo. */
  $X \leftarrow \emptyset$; /* vértices visitados na busca em profundidade */
  para cada vértice v_r livre em K_r **faça**
  início
    $\langle \exists$cam, $P\rangle \leftarrow$ BuscaProfundidade (G, M, K, v_r, r, X);
    se $\exists$cam **então acrescente** P a $\mathcal{P}$;
  fim
  devolva $\mathcal{P}$;

BuscaProfundidade (G, M, K, v_i, i, X) /* i é ímpar */
  para cada $v_{i-1} \in \{K_{i-1} \cap \overline{M}(v_i)\} \setminus X$ **faça**
  início
    $e_i \leftarrow$ aresta (v_{i-1}, v_i) em $\overline{M}$; $X \leftarrow X \cup \{v_{i-1}\}$;
    se $i = 1$ **então** /* Encontramos um caminho aumentante */
      devolva $\langle$sim, $(v_0, e_1, v_1)\rangle$
    senão
    início
      $v_{i-2} \leftarrow$ vértice em $\{K_{i-2} \cap M(v_{i-1})\}$;
      $e_{i-1} \leftarrow$ aresta (v_{i-2}, v_{i-1}) em M;
      $\langle \exists$cam, $P\rangle \leftarrow$ BuscaProfundidade $(G, M, K, v_{i-2}, i-2, X)$;
      se $\exists$cam **então devolva** $\langle$sim, $P \circ (v_{i-2}, e_{i-1}, v_{i-1}, e_i, v_i)\rangle$;
    fim
  fim
  devolva $\langle$não, $\emptyset\rangle$;

Algoritmo 2.4: Determina uma coleção maximal de caminhos M-aumentantes com base na busca em largura efetuada.

/* Dado um grafo G com bipartição (V^+,V^-) e um emparelhamento inicial M_0 (possivelmente vazio) em G, é retornado um emparelhamento máximo M em G e uma cobertura mínima $(V^+ \setminus S) \cup T$. De acordo com a busca em largura, pode ocorrer:

- Encontramos um caminho aumentante, onde usamos K (vetor das camadas dos vértices visitados na busca em largura) e r (nível da última camada da busca em largura) para determinar uma coleção maximal de caminhos aumentantes disjuntos e de comprimento mínimo.
- Não encontramos um caminho aumentante, onde $(V^+ \setminus S) \cup T$ nos dá uma certidão da otimalidade de M (S e T são os conjuntos de vértices visitados de V^+ e V^-na busca em largura, respectivamente).

A complexidade do algoritmo é $O((m+n)\min\{\sqrt{n}, 1+|M|-|M_0|\})$. */

HopcroftKarp (G, M_0)
  $M \leftarrow M_0$;
  repita
    $(\exists$cam, K ou S, r ou $T) \leftarrow$ BuscaLargura (G, M);
    se $\exists$cam **então**
    início
      $\mathcal{P} \leftarrow$ DetColeçãoMaximal (G, M, K, r);
      para cada $P \in \mathcal{P}$ **faça**
        $M \leftarrow M \oplus EP$;
    fim
  até que $\nexists$cam;
  devolva $(M, (V^+ \setminus S) \cup T)$;

Algoritmo 2.5: Hopcroft e Karp.

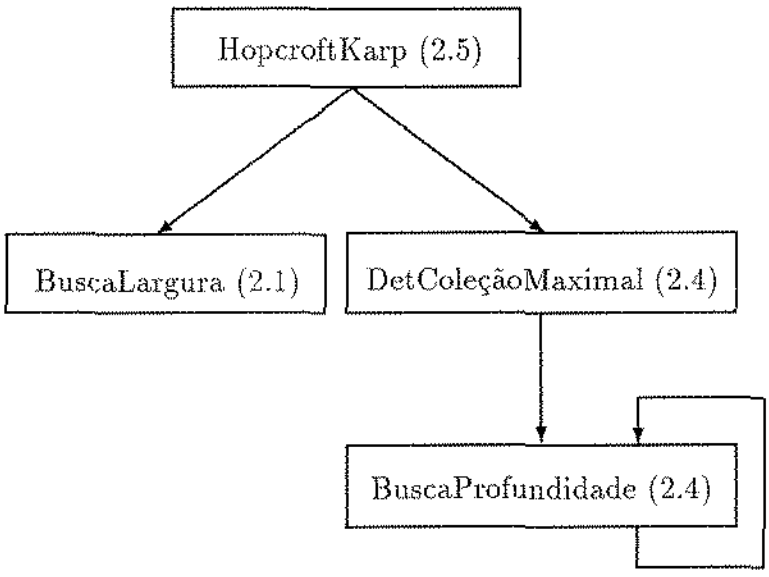


Figura 2.3: Esquema das chamadas das funções do algoritmo de Hopcroft e Karp.

2.2.3 Complexidade

Cada coleção maximal pode ser encontrada em $O(m+n)$, pois:

1. Uma busca em largura (Algoritmo 2.1) gasta $O(m+n)$.
2. Na determinação de uma coleção maximal (Algoritmo 2.4) são realizadas várias buscas em profundidade, porém, nestas buscas cada vértice é visitado no máximo uma única vez e cada aresta no máximo duas vezes, dando, assim, uma complexidade de $O(m+n)$.

Como existem $O(\sqrt{n})$ fases em que procuramos coleções maximais (Teorema 2.4) e em cada fase determinamos no mínimo um caminho aumentante, então a complexidade deste algoritmo é

$$O\left((m+n)\min\{\sqrt{n}, 1+|M^*|-|M_0|\}\right),$$

onde M^* é um emparelhamento máximo e M_0 é o emparelhamento inicial.

2.3 Algoritmo Paralelo de Kim e Chwa

- Autores: Kim e Chwa [39].
- Data: 1987.
- Tipo: Paralelo.
- Complexidade: $O(n \log n \log \log n)$.
- Processadores: $O(n^3/\log n)$.
- Modelo: PRAM e CREW.

Este algoritmo é semelhante ao Primitivo (Seção 2.1), porém, a busca pelo caminho M-aumentante de comprimento mínimo é feita em paralelo. Para tanto, Kim e Chwa transformaram o problema de emparelhamento máximo em grafos bipartidos num problema de fluxo máximo, com capacidades zero ou um nas arestas (transformação padrão). O algoritmo apresentado nesta seção não utiliza esta transformação.

2.3.1 Fundamentos

Seja G um grafo com bipartição (V^+ e V^-) e M um emparelhamento em G. Dizemos que um passeio é *M-estrito* se for M-alternado e tiver sinal igual ao do vértice de sua origem (veja definição de sinal na página 4). A seguir damos algumas características importantes sobre passeios M-estritos:

Lema 2.6 *Se $P \circ Q$ é um passeio M-estrito, então P e Q são passeios M-estritos.*

Demonstração: Claramente P é um passeio M-estrito. De acordo com o comprimento de P, vamos dividir a demonstração em dois casos:

1. Comprimento igual a zero. Neste caso é trivial, pois $P \circ Q = Q$, e portanto Q é um passeio M-estrito.
2. Comprimento maior do que zero. Seja P da seguinte forma

$$P := P' \circ (u, \alpha, v).$$

Por hipótese de indução, (u, α, v) é M-estrito, logo a aresta α possui o mesmo sinal do vértice u e sinal oposto ao do vértice v. Como $P \circ Q$ é um passeio M-alternado, então Q possui o mesmo sinal do vértice v (v é o vértice inicial de Q), e portanto Q é um passeio M-estrito. □

Lema 2.7 *Sejam P e Q dois passeios M-estritos, sendo P do vértice i ao l e Q do vértice l ao j. Então, $P \circ Q$ é um passeio M-estrito de i a j.*

Demonstração: Note que se P e/ou Q for de comprimento zero, a demonstração é trivial. Portanto, assuma que P e Q possuem comprimento maior do que zero. Sejam

$$P := P' \circ (u, \alpha_P, l)$$

e

$$Q := (l, \alpha_Q, v) \circ Q'.$$

Pelo Lema 2.6, l possui sinal oposto ao de α_P; e, por hipótese, l possui o mesmo sinal de α_Q. Portanto, $P \circ Q$ é alternado e conseqüentemente um passeio M-estrito de i a j. □

Lema 2.8 *Se P é um passeio M-estrito de i a j, então existe uma subseqüência Q de P que é um caminho M-estrito de i a j.*

Demonstração: Suponha que P não é um caminho (se for, a subseqüência é o próprio P), então $\exists v \in VP$ que ocorre mais de uma vez em P. Faça (veja Figura 2.4)

$$P := R \circ S \circ T,$$

onde S é um passeio não degenerado de v a v.

Pelo Lema 2.6 temos que R e T são passeios M-estritos. Portanto, pelo Lema 2.7, $P' := R \circ T$ é um passeio M-estrito de i a j. Repetindo-se este processo, obtemos no final uma subseqüência Q de P que é um caminho M-estrito de i a j. □

Seja $D_{i,j}$ o comprimento do menor caminho M-estrito de i a j; caso não exista faça $D_{i,j} := \infty$. Suponha que os vértices de G são numerados de 0 a

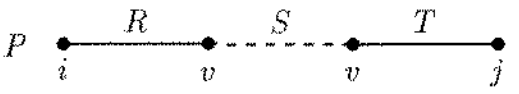


Figura 2.4: Subdivisão do caminho P.

$n-1$. Vamos agora definir duas seqüências de matrizes $n \times n$ ($[0 \ldots n-1] \times [0 \ldots n-1]$)

$$D^0, D^1, \ldots D^{\lceil \log_2 n \rceil}$$

e

$$l^0, l^1, \ldots l^{\lceil \log_2 n \rceil}$$

recursivamente, com $0 \leq i, j \leq n-1$, da seguinte forma:

$$D^0_{i,j} := \begin{cases} 0 & \text{se } i = j \\ 1 & \text{se } (i,j) \in E \text{ e o sinal da aresta } (i,j) \\ & \text{é igual ao sinal do vértice } i \\ \infty & \text{nos demais casos,} \end{cases}$$

$$\underset{(k>0)}{D^k_{i,j}} := \min \left\{ D^{k-1}_{i,l} + D^{k-1}_{l,j} : 0 \leq l \leq n-1 \right\}$$

e

$$l^k_{i,j} := \begin{cases} i & \text{se } k = 0 \\ l & \text{se } k > 0,\ l \in [0 \ldots n-1] \text{ e satisfaz } D^k_{i,j} = D^{k-1}_{i,l} + D^{k-1}_{l,j}. \end{cases}$$

Para cada tripla i, j, k, onde $0 \leq i, j \leq n-1$, $0 \leq k \leq \lceil \log_2 n \rceil$ e $D^k_{i,j} < \infty$, a seqüência $P^k_{i,j}$ é definida recursivamente, da seguinte forma:

$$P^0_{i,j} := \begin{cases} (i) & \text{se } i = j \\ (i,(i,j),j) & \text{caso contrário} \end{cases}$$

e

$$\underset{(k>0)}{P^k_{i,j}} := P^{k-1}_{i,l} \circ P^{k-1}_{l,j}, \text{ com } l := l^k_{i,j}.$$

Um importante lema, obtido das definições acima, é apresentado a seguir:

Lema 2.9 *As seqüências $P^k_{i,j}$ são bem definidas. Ademais, para $0 \leq i,j \leq n-1$ e $0 \leq k \leq \lceil \log n \rceil$ temos que:*

Se $D^k_{i,j} < \infty$, então $P^k_{i,j}$ é um passeio M-estrito, de i a j, e de comprimento $D^k_{i,j}$.

Demonstração: Prova por indução em k. O caso base ($k = 0$) é trivial, direto das definições de P^0 e D^0. Suponha que a afirmação é verdadeira para $k-1$ ($k > 0$). Pela definição de $P^k_{i,j}$, temos que

$$P^k_{i,j} := P^{k-1}_{i,l} \circ P^{k-1}_{l,j}, \text{ onde } l := l^k_{i,j}.$$

Pela definição de $l^k_{i,j}$, $D^k_{i,j} = D^{k-1}_{i,l} + D^{k-1}_{l,j}$, portanto, $D^{k-1}_{i,l} < \infty$ e $D^{k-1}_{l,j} < \infty$. Logo, $P^{k-1}_{i,l}$ e $P^{k-1}_{l,j}$ são definidos. Por hipótese de indução, $P^{k-1}_{i,l}$ e $P^{k-1}_{l,j}$ são passeios M-estritos de i a l e de l a j, respectivamente, e de comprimentos $D^{k-1}_{i,l}$ e $D^{k-1}_{l,j}$, respectivamente. Portanto, pelo Lema 2.7, $P^k_{i,j}$ é um passeio M-estrito de i a j e de comprimento $D^k_{i,j}$. □

Teorema 2.10 *Para $0 \leq i,j \leq n-1$ e $0 \leq k \leq \lceil \log n \rceil$ temos que:*

(i) Se $D^k_{i,j} < \infty$, então $P^k_{i,j}$ é um caminho M-estrito, de i a j, e de comprimento $D_{i,j}$ (mínimo).

(ii) $$D^k_{i,j} = \begin{cases} D_{i,j} & \text{se } D_{i,j} \leq 2^k \\ \infty & \text{caso contrário.} \end{cases}$$

Demonstração: Prova por indução em k. O caso base ($k = 0$) é trivial, direto das definições de P^0 e D^0. Suponha que as afirmações são verdadeiras para $k-1$ ($k > 0$). De acordo com o valor de $D_{i,j}$, dividiremos a demonstração em dois casos:

1. $D_{i,j} \leq 2^k$. Pelo Lema 2.6, $\exists l \in [0 \ldots n-1]$, com $D_{i,l}$, $D_{l,j} \leq 2^{k-1}$, onde $D_{i,j} = D_{i,l} + D_{l,j}$. Por definição, $D^k_{i,j} \leq D^{k-1}_{i,l} + D^{k-1}_{l,j}, \forall l \in [0 \ldots n-1]$. Logo
$$D^k_{i,j} \leq D^{k-1}_{i,l} + D^{k-1}_{l,j} \overset{\text{H.I.}}{=} D_{i,l} + D_{l,j} = D_{i,j}.$$
Pelo Lema 2.9, $P^k_{i,j}$ é M-estrito, de i a j, e de comprimento $D^k_{i,j}$. Como, por definição, $D_{i,j}$ é o comprimento mínimo de um caminho M-estrito de i a j e, pelo Lema 2.8, existe uma subseqüência M-estrita de $P^k_{i,j}$ que é um caminho, então, na realidade, $D^k_{i,j} = D_{i,j}$ e $P^k_{i,j}$ é um caminho M-estrito mínimo de i a j.

2. $D_{i,j} > 2^k$. Seja $l \in [0 \ldots n-1]$. Note que $D_{i,j} \leq D_{i,l} + D_{l,j}$, portanto,

$$\max\{D_{i,l}, D_{l,j}\} > 2^{k-1},$$

pois caso contrário $D_{i,j}$ seria no máximo 2^k, contradição. Logo, por hipótese de indução,

$$\max\left\{D_{i,l}^{k-1}, D_{l,j}^{k-1}\right\} = \infty$$

e portanto $D_{i,l}^{k-1} + D_{l,j}^{k-1} = \infty$. Como esta igualdade é válida $\forall l \in [0 \ldots n-1]$, então, por definição de $D_{i,j}^k$, $D_{i,j}^k = \infty$. □

2.3.2 Descrição

Este algoritmo é uma versão em paralelo do algoritmo primitivo (Seção 2.1), onde a correta determinação de um caminho aumentante é dada pelo Teorema 2.10.

Seja d o comprimento do menor caminho M-aumentante. As matrizes D^0 e l^0 e as seqüências $D^1, D^2, \ldots, D^{\lceil \log d \rceil}$ e $l^1, l^2, \ldots, l^{\lceil \log d \rceil}$ são determinadas no Algoritmo 2.6. Assim que obtemos $l^{\lceil \log d \rceil}$, chamamos o Algoritmo 2.7, onde determinamos um caminho M-aumentante com base nos vértices armazenados nas matrizes da seqüência $l^0, l^1, \ldots, l^{\lceil \log d \rceil}$.

O Algoritmo 2.8 (Kim e Chwa) determina um emparelhamento máximo em paralelo. Na Figura 2.5 é apresentado um esquema das chamadas de funções deste algoritmo.

/* Dado um grafo G e um emparelhamento M em G, determina se existe ou não um caminho M-aumentante.

- Em caso afirmativo, devolve $\langle$sim, $P\rangle$, onde P é um caminho M-aumentante mínimo.
- Em caso negativo, devolve $\langle$não$\rangle$.

Este algoritmo utiliza $O(n^3)$ processadores num tempo $O(\log d \log n)$-CREW, onde d é o menor comprimento entre os caminhos M-aumentantes em G. */

```
EncontraCaminho (G, M)
  para i ← 0 até n − 1 faça em paralelo /* Determina D^0 e l^0 */
      para j ← 0 até n − 1 faça em paralelo
      início
          l^0_{i,j} ← i;
          se i = j então D^0_{i,j} ← 0
          senão
              se (i, j) ∈ E e SinalAresta ((i, j)) = SinalVértice (i) então
                  D^0_{i,j} ← 1
              senão D^0_{i,j} ← ∞;
      fim
  para k ← 1 até ⌈log n⌉ faça /* Determina D^k e l^k, k > 0 */
  início
      para i ← 0 até n − 1 faça em paralelo
          para j ← 0 até n − 1 faça em paralelo
          início
              ⟨D^k_{i,j}, l^k_{i,j}⟩ ← min{D^{k−1}_{i,l} + D^{k−1}_{l,j} : 0 ≤ l ≤ n − 1};
              se i ≠ j e Livre (i) e Livre (j) então w_{i,j} ← D^k_{i,j}
              senão w_{i,j} ← ∞;
          fim
      ⟨d, i, j⟩ ← min{w_{i,j} : 0 ≤ i ≤ n − 1 e 0 ≤ j ≤ n − 1};
      se d < ∞ então devolva ⟨sim, DeterminaCaminho (i, j, l, k, G)⟩
  fim
  devolva ⟨não⟩;
```

Algoritmo 2.6: Determina, se existe, um caminho M-aumentante mínimo.

```
/* Dados s, t, l, ⌈log d⌉ e G, onde s e t são os vértices livres extremos de
um caminho M-aumentante de comprimento d e l é a seqüência de matrizes
l^0, l^1, ..., l^⌈log d⌉, determina um caminho M-aumentante de s a t de compri-
mento d, através dos vértices armazenados em l. Este algoritmo utiliza O(n)
processadores num tempo O(log d)-EREW. */

DeterminaCaminho (s, t, l, k, G) /* k = ⌈log d⌉ */
  se k = 0 então
      se s = t então
          devolva (s)
      senão
          devolva (s, (s, t), t)
  senão
  início
      v ← l^k_{s,t};
      faça em paralelo os passos esquerda e direita
      início
          esquerda: P_e ← DeterminaCaminho (s, v, l, k − 1, G);
          direita:  P_d ← DeterminaCaminho (v, t, l, k − 1, G);
      fim
      devolva P_e ∘ P_d;
  fim
```

Algoritmo 2.7: Determina um caminho M-aumentante com base na seqüência de matrizes $l^0, l^1, \ldots, l^{\lceil \log d \rceil}$.

```
/* Dado um grafo G com bipartição (V+, V-), determina, em paralelo, um
emparelhamento máximo de G. Este algoritmo utiliza O(n^3) processadores
num tempo O(n log n log log n)-CREW. */

KimChwa (G)
  M ← ∅;
  M ← ObtémEmparelhamentoMáximo (G, M);
  devolva M;

ObtémEmparelhamentoMáximo (G, M)
  repita
      ⟨∃cam, P⟩ ← EncontraCaminho (G, M);
      se ∃cam então
          M ← M ⊕ EP
  até que ∄cam;
  devolva M;
```

Algoritmo 2.8: Kim e Chwa (paralelo).

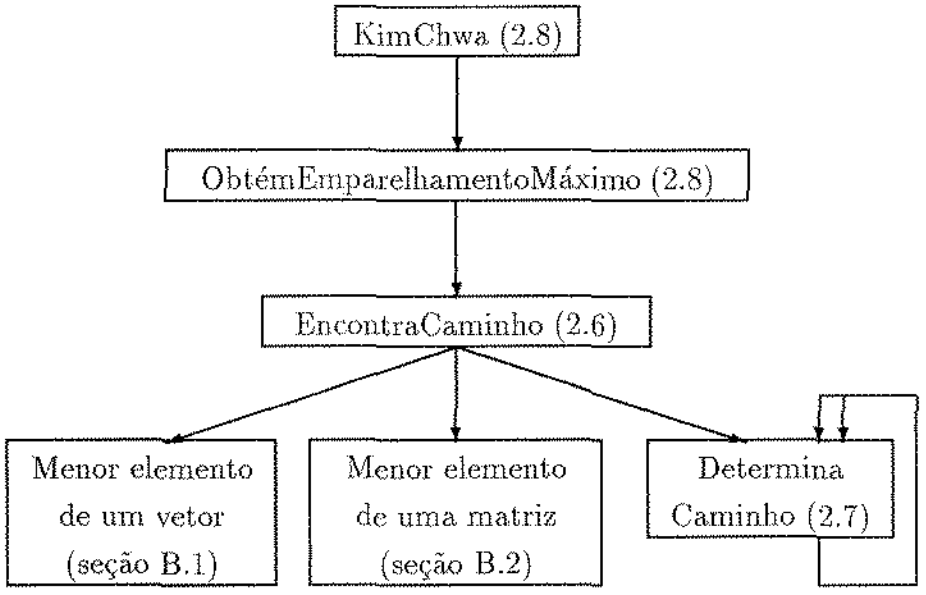


Figura 2.5: Esquema das chamadas das funções do algoritmo de Kim e Chwa.

2.3.3 Complexidade e Número de Processadores

Como no Algoritmo 2.6 existem n^2 valores para serem calculados nas matrizes D^k e l^k $(k > 0)$, logo usamos $O(n^3/\log n)$ processadores para calcular todos os elementos das matrizes num tempo de $O(\log n)$, pois a complexidade para se determinar o menor elemento de um vetor é $O(\log n)$, usando $O(n/\log n)$ processadores (veja Seção B.1). O Algoritmo 2.6, ao calcular D^k e l^k a partir de D^{k-1}, usa leitura concorrente: este, na realidade, é o único ponto do algoritmo de Kim e Chwa onde ocorre leitura concorrente. Na obtenção das matrizes D^0 e l^0, usamos $O(n^2)$ processadores num tempo de $O(1)$. A complexidade para se determinar o menor elemento da matriz $w_{n\times n}$ é $O(\log n)$, usando $O(n^2/\log n)$ processadores (veja Seção B.2).

Na realidade, o "ou exclusivo" $(M \oplus EP)$ não é realizado no Algoritmo 2.8. Ele deverá ser realizado no Algoritmo 2.7, onde temos

devolva $(s,(s,t),t)$,

deverá ser executado

se $(s,t) \in M$ **então**
    remova (s,t) **de** M
senão
    acrescente (s,t) **a** M.

Seja $M_0, M_1, \ldots, M_h$ a seqüência de emparelhamentos obtidos no Algoritmo 2.8, onde $M_0 = \emptyset$ e M_h é um emparelhamento máximo ($|M_i| = i$, $i = 0 \ldots h$). Seja P_i um caminho M_{i-1}-aumentante mínimo $(1 \leq i \leq h)$, portanto $M_i = M_{i-1} \oplus EP_i$. Defina $d_i := |EP_i|$, quando $1 \leq i \leq h$, e $d_{h+1} := n$.

Na obtenção de um caminho M_{i-1}-aumentante a partir da seqüência de matrizes $l^0, l^1, \ldots, l^{\lceil \log d_i \rceil}$ (Algoritmo 2.7), usamos $O(n)$ processadores, um para cada vértice de G (um caminho passa no máximo uma vez por cada vértice), num tempo de $O(\log d_i)$.

Portanto, se existe um caminho M_{i-1}-aumentante, o Algoritmo 2.6 possui uma complexidade $O(\log d_i \log n)$, usando $O(n^3/\log n)$ processadores, pois só precisamos determinar $O(\log d_i)$ matrizes nas seqüências $D^0, \ldots, D^{\lceil \log d_i \rceil}$ e $l^0, \ldots, l^{\lceil \log d_i \rceil}$ para encontrar um caminho M_{i-1}-aumentante. Se M_{i-1}

é um emparelhamento máximo ($i - 1 = h$), ou seja, não existem mais caminhos M_{i-1}-aumentantes, então o algoritmo possui uma complexidade $O(\log n \log n)$, ou seja, $O(\log d_{h+1} \log n)$, usando $O(n^3/\log n)$ processadores. Logo, a complexidade do Algoritmo 2.8 é

$$O\left(\log n \left(\sum_{i=1}^{h+1} \log d_i\right)\right).$$

Para um emparelhamento M_r, onde

$$r := \left\lfloor h\left(1 - \frac{2}{1+\log n}\right)\right\rfloor,$$

temos, pelo Corolário 1.2, que

$$\begin{aligned} d_r &\leq 1 + 2\left\lfloor \frac{r}{h-r} \right\rfloor \\ &\leq 1 + 2\left\lfloor \frac{h\left(1 - \frac{2}{1+\log n}\right)}{h - h\left(1 - \frac{2}{1+\log n}\right)} \right\rfloor \\ &= 1 + 2\left\lfloor \frac{-1+\log n}{2} \right\rfloor \\ &\leq 1 + 2\left(\frac{-1+\log n}{2}\right) \\ &= \log n. \end{aligned}$$

Por outro lado, pelo Teorema 2.3 e dado que $d_{h+1} = n$, então $d_{i-1} \leq d_i \leq n$, onde $2 \leq i \leq h+1$. Portanto,

$$\begin{aligned} \sum_{i=1}^{h+1} \log d_i &\leq \sum_{i=1}^{r} \log d_r + \sum_{i=r+1}^{h+1} \log n \\ &\leq r \log\log n + (h - r + 1)\log n \\ &= r \log\log n + \left\lceil 1 + \frac{2h}{1+\log n} \right\rceil \log n \\ &\leq O(n \log\log n). \end{aligned}$$

Logo, temos a complexidade desejada.

2.3.4 Observação

Podemos reduzir a quantidade de memória utilizada pelo algoritmo, pois:

1. Da seqüência de matrizes $D^0, D^1, \ldots, D^k, \ldots$, basta armazenar a matriz mais recente, pois para calcular D^k, só precisamos de D^{k-1} $(k > 0)$.

2. Da seqüência de matrizes $l^0, l^1, \ldots, l^k, \ldots$, para determinar um caminho aumentante, basta armazenar uma matriz l, onde $l_{i,j} \leftarrow l^k_{i,j}$, para o menor k que satisfaz $D^k_{i,j} < \infty$.

2.4 Algoritmo Paralelo de Goldberg, Plotkin e Vaidya

- Autores: Goldberg, Plotkin e Vaidya [32].
- Data: 1988.
- Tipo: Paralelo.
- Complexidade: $O\left(n^{2/3}\log^3 n\right)$.
- Processadores: $O\left(n^3/\log n\right)$.
- Modelo: PRAM e CRCW.

Este algoritmo trabalha em duas fases. Primeiro é usado um algoritmo, chamado de *algoritmo de pré-emparelhamentos*, que funciona de uma forma eficiente quando existem muitos vértices livres incidindo em caminhos aumentantes de comprimento "pequeno". Depois é usado, até se obter um emparelhamento máximo, o algoritmo de Kim e Chwa. Existe um critério de balanceamento, entre estas duas fases, que torna este algoritmo sublinear.

O Algoritmo de pré-emparelhamentos será descrito na Seção 2.5. O Algoritmo de Kim e Chwa foi visto na Seção 2.3.

2.4.1 Descrição e Complexidade

Goldberg, Plotkin e Vaidya definem dois parâmetros, l e a, em função de n; um caminho é *pequeno* se o seu comprimento é no máximo l; o papel de a é esclarecido em (2.1). Conforme teremos ocasião de ver, é conveniente fazer $l := \left\lfloor n^{1/3} \right\rfloor$ e $a := \left\lfloor n^{2/3} \right\rfloor$. Este algoritmo se divide em duas fases (Algoritmo 2.9):

Primeira fase É utilizado um algoritmo chamado de pré-emparelhamentos (Seção 2.5), que determina um emparelhamento M com a seguinte propriedade:

> "o número de vértices M-livres de V^- incidindo em caminhos M-aumentantes de comprimento pequeno é menor do que a." (2.1)

/* Dado um grafo G com bipartição (V^+, V^-), determina, em paralelo, um emparelhamento máximo de G. Primeiro utilizamos o algoritmo de pré-emparelhamentos e depois o algoritmo de Kim e Chwa. Este algoritmo utiliza $O\left(n^2 \lceil n/\log n \rceil\right)$ processadores num tempo $O\left(n^{2/3}\log^3 n\right)$-CRCW. */

GoldbergPlotkinVaidya (G)
  $M \leftarrow$ Pré-Emparelhamento $\left(G, \left\lfloor n^{1/3} \right\rfloor, \left\lfloor n^{2/3} \right\rfloor\right)$;
  devolva ObtémEmparelhamentoMáximo (G, M); /* Kim e Chwa */

Algoritmo 2.9: Goldberg, Plotkin e Vaidya (paralelo).

Conforme será visto na Seção 2.5 (Corolário 2.22), a complexidade do algoritmo de pré-emparelhamentos é

$$O\left(\frac{nl}{a}\log^3 n\right) \tag{2.2}$$

tempo, usando $O\left(n^2\right)$ processadores (CRCW).

Segunda fase Partindo do emparelhamento M obtido na primeira fase, é utilizado o Algoritmo de Kim e Chwa (Seção 2.3). Conforme foi visto, a complexidade de uma iteração desse algoritmo é

$$O\left(\log d \log n\right) = O\left(\log^2 n\right),$$

onde d é o comprimento do caminho aumentante determinado (obtido).

Seja M^* o emparelhamento máximo fornecido pelo algoritmo de Kim e Chwa: o número de iterações executadas é igual ao número de caminhos M-aumentantes, digamos i, no grafo $G[M^* \oplus M]$. Destes i caminhos, digamos que i_p são pequenos. Por hipótese,

$$i_p \leq a,$$

Dado que os i caminhos são disjuntos nos vértices,

$$i - i_p \leq \frac{n}{l}.$$

Portanto,

$$i \leq a + \frac{n}{l}.$$

Assim, a complexidade da segunda fase é

$$O\left(\left(a + \frac{n}{l}\right)\log^2 n\right) \tag{2.3}$$

tempo, usando $O(n^3/\log n)$ processadores (CREW).

De (2.2) e (2.3) temos que a complexidade do algoritmo é

$$O\left(\frac{nl}{a}\log^3 n + a\log^2 n + \frac{n}{l}\log^2 n\right)$$

tempo, usando $O(n^3/\log n)$ processadores (CRCW), que com $l := \left\lfloor n^{1/3} \right\rfloor$ e $a := \left\lfloor n^{2/3} \right\rfloor$ fornece

$$O\left(n^{2/3}\log^3 n\right).$$

Na Figura 2.6 é apresentado um esquema das chamadas de funções deste algoritmo.

Para justificar a otimalidade dos expoentes de a e l, precisamos do seguinte resultado:

Lema 2.11 *Para $t \in \Re_{>0}$ e $i \in \mathcal{Z}_{\geq 0}$, temos que*

$$\lim_{n\to\infty} \frac{n^t}{\log^i n} = \infty.$$

Demonstração: A demonstração é feita por indução em i, usando a regra de L'Hôpital (vide [20, 52]). □

Teorema 2.12 *Os valores dos expoentes de a (2/3) e l (1/3) são ótimos.*

Demonstração: Para tornar a demonstração mais geral, vamos assumir que as complexidades das duas fases são, respectivamente,

$$O\left(\frac{nl}{a}\log^k n\right) \text{ e } O\left(\left(\frac{n}{l} + a\right)\log^j n\right),$$

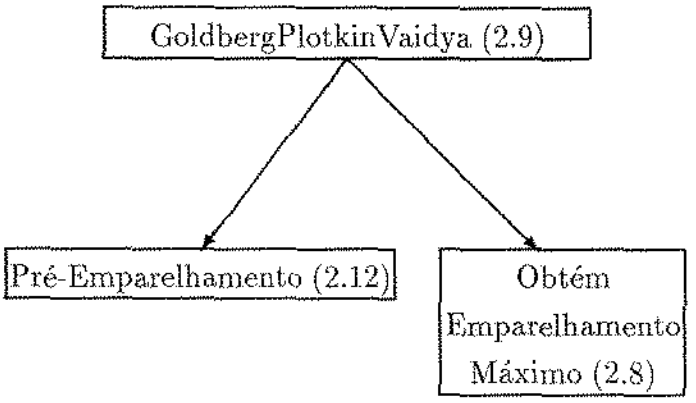


Figura 2.6: Esquema das chamadas das funções do algoritmo de Goldberg, Plotkin e Vaidya.

com k e j inteiros não negativos. Logo, a complexidade do algoritmo, com os valores indicados de l e a, seria

$$O\left(n^{2/3}\log^{\max\{k,j\}} n\right). \tag{2.4}$$

Faça $a := n^x$ e $l := n^y$, onde x e $y \in \Re$. Logo, a complexidade do algoritmo com estes valores ficaria:

$$O\left(n^{y-x+1}\log^k n + \left(n^{1-y} + n^x\right)\log^j n\right).$$

Para que esta nova complexidade seja menor que a obtida em (2.4), os seguintes limites devem assumir valores finitos:

$$\lim_{n\to\infty} \frac{n^{y-x+1}\log^k n}{n^{2/3}\log^{\max\{k,j\}} n} = \lim_{n\to\infty} \frac{n^{y-x+\frac{1}{3}}}{\log^{\max\{0,j-k\}} n}$$

$$\lim_{n\to\infty} \frac{n^{1-y}\log^j n}{n^{2/3}\log^{\max\{k,j\}} n} = \lim_{n\to\infty} \frac{n^{\frac{1}{3}-y}}{\log^{\max\{k-j,0\}} n}$$

$$\lim_{n\to\infty} \frac{n^{x}\log^j n}{n^{2/3}\log^{\max\{k,j\}} n} = \lim_{n\to\infty} \frac{n^{x-\frac{2}{3}}}{\log^{\max\{k-j,0\}} n}$$

Portanto, pelo Lema 2.11, temos que:

(i) $y - x + 1/3 \leq 0$.

(ii) $y \geq 1/3$.

(iii) $x \leq 2/3$.

Por outro lado, aplicando as duas últimas desigualdades na primeira obtemos

$$0 \leq y - x + 1/3 \leq 0.$$

Logo, necessariamente x e y satisfazem: $x = 2/3$ e $y = 1/3$.

Note que nesta demonstração os valores de j e k não são relevantes. □

2.4.2 Observação

Os autores Goldberg, Plotkin, Shmoys e Tardos, em [31], apresentam um algoritmo paralelo, que utiliza o método de ponto interior (programação linear), para resolver o problema de emparelhamento máximo numa complexidade $O\left(m^{1/2}\log^3 n\right)$, usando m^3 processadores (CRCW). Assim, para grafos pouco densos ($m = O\left(n^{4/3}\right)$), este algoritmo possui uma complexidade menor do que o algoritmo de Goldberg, Plotkin e Vaidya.

2.5 Algoritmo de Pré-Emparelhamentos

- Autores: Goldberg, Tarjan, Plotkin e Vaidya [34, 32].
- Data: 1988.
- Tipo: Paralelo.
- Complexidade: $O\left(\frac{nl}{a}\log^3 n\right)$.
  Os parâmetros l e a são definidos logo a seguir.
- Processadores: $O(n^2)$.
- Modelo: PRAM e CRCW.

Este algoritmo determina um emparelhamento M em G satisfazendo a seguinte propriedade:

> "o número de vértices M-livres de V^- incidindo em caminhos M-aumentantes de comprimento no máximo l é menor do que a." (2.5)

Note que esta propriedade é a mesma Propriedade (2.1) apresentada na Seção 2.4.1.

Os autores Goldberg e Tarjan, em [34], apresentaram algumas funções, que executadas de uma forma apropriada, determinam um fluxo máximo entre dois vértices. Os algoritmos que utilizam estas funções são denominados de *pré-fluxos*. Os autores Goldberg, Plotkin e Vaidya, em [32], usaram estas funções e emparelhamentos maximais para determinar um emparelhamento M em G satisfazendo a Propriedade (2.5).

Fizemos uma adaptação do algoritmo de pré-fluxos que, por coerência, chamaremos de *algoritmo de pré-emparelhamentos*. O algoritmo de pré-emparelhamentos é apresentado nesta seção.

2.5.1 Fundamentos

Seja o grafo G com bipartição V^+ e V^-. Suponha que os vértices de G são numerados de 0 a $n-1$. Vamos definir uma seqüência de emparelhamentos M_k $(k \geq 0)$ em G, onde

$$M_0 := \emptyset.$$

Para obter o próximo emparelhamento (M_{k+1}), necessitamos do auxílio das seqüências d_k, T_k, N_k e E_k ($k \geq 0$), onde:

- $d_k(v)$ ($v \in V$), como será demonstrado mais tarde, estima um limite inferior da distância de v a um vértice livre de V^+, através de um caminho M_k-estrito de origem em v.

  Como uma extensão da definição de $d_k(v)$, definimos $d_k(X)$ para um conjunto de vértices $X \subseteq V$, da seguinte forma:

  $$d_k(X) := \min\left(\{n-1\} \cup \{d_k(v) : v \in X\}\right).$$

  Temos então imediatamente o seguinte resultado:

  Lema 2.13 *Se $Y \subseteq X$, onde $X \subseteq V$, então $d_k(Y) \geq d_k(X)$.* □

  A definição de $d_k(v)$ é dada a seguir:

  $$\underset{(v^+ \in V^+)}{d_k(v^+)} := \begin{cases} 0 & \text{se } v^+ \text{ é } M_k\text{-livre} \\ 1 + d_{k-1}(v^-) & \text{caso contrário, onde } v^- \in M_k(\{v^+\}) \end{cases}$$

  e

  $$\underset{(v^- \in V^-)}{d_k(v^-)} := 1 + d_k(\overline{M}_k(\{v^-\})).$$

  Note que esta seqüência está bem definida, pois $M_0 := \emptyset$.

  A seguir apresentamos um resultado facilmente demonstrado por indução:

  Lema 2.14 *Se $v^+ \in V^+$, então $0 \leq d_k(v^+) \leq n+1$. Se $v^- \in V^-$, então $1 \leq d_k(v^-) \leq n$.* □

- T_k é um conjunto de arestas disjunto de M_k:

  $$T_k := \{(v^-, v^+) \in E : v^- \in V^- \setminus VM_k \text{ e } d_k(v^-) = 1 + d_k(v^+)\}.$$

- N_k é um emparelhamento maximal do grafo $G[T_k]$.

- E_k é um emparelhamento e subconjunto de $M_k \cup N_k$:

$$E_k := \{(v^-, v^+) : \left\{ \begin{array}{ll} v^- < u^- & \text{se } d_k(v^-) = d_k(u^-) \\ d_k(v^-) < d_k(u^-) & \text{caso contrário} \end{array} \right\}, \\ v^+ \in V^+ \text{ e } (M_k \cup N_k)(\{v^+\}) = \{v^-, u^-\}\}.$$

O novo emparelhamento M_{k+1} é obtido da seguinte forma:

$$M_{k+1} := (M_k \cup N_k) \setminus E_k.$$

A seguir são apresentadas importantes características sobre M_k.

Lema 2.15 *Para $k \geq 0$, M_k é um emparelhamento em G.*

Demonstração: Prova por indução em M_k. O caso base ($k = 0$) é trivial, pois $M_0 = \emptyset$. Suponha verdadeira para M_k ($k \geq 0$). Vamos provar que para todo vértice $v \in V$, no máximo uma aresta de M_{k+1} incide em v.

Dado que M_k e N_k são emparelhamentos, então no máximo duas arestas de $M_k \cup N_k$ incidem em v.

Pela definição de T_k, todo vértice de V^- incidente em alguma aresta de N_k é M_k-livre. Portanto, podemos supor que $v \in V^+$ e que precisamente duas arestas de $M_k \cup N_k$ incidem em v.

Por definição de E_k, precisamente uma das duas arestas de $M_k \cup N_k$ que incidem em v pertence a E_k.

Em todos os casos considerados, no máximo uma aresta de M_{k+1} incide em v. □

Corolário 2.16 *Sejam $k \geq 0$, $v^- \in V^-$ e $v^+ \in V^+$. Se v^- é M_k-ocupado e M_{k+1}-ocupado, então $M_k(\{v^-\}) = M_{k+1}(\{v^-\})$. Se v^+ é M_k-ocupado, então v^+ é M_{k+1}-ocupado.* □

A seguir são apresentadas importantes características sobre d_k:

Teorema 2.17 *Para todo vértice $v \in V$ e $k \geq 0$, temos que $d_{k+1}(v) \geq d_k(v)$.*

Demonstração: A demonstração será feita por indução em k, da seguinte forma:

Base $v \in V^+$ e $k = 0$.

Passo caso 1 $v \in V^+$ e $k > 0$: adota-se como hipótese de indução que $d_k(v^-) \geq d_{k-1}(v^-)$, $\forall v^- \in V^-$.

Passo caso 2 $v \in V^-$ e $k \geq 0$: adota-se como hipótese de indução que $d_{k+1}(v^+) \geq d_k(v^+)$, $\forall v^+ \in V^+$.

Assim, considere os seguintes casos:

Base e passo caso 1 $v \in V^+$ e $k \geq 0$.

Se v é M_k-livre (inclui o caso base em que $k = 0$) então

$$d_{k+1}(v^+) \overset{\text{Lem 2.14}}{\geq} 0 \overset{\text{def}}{=} d_k(v^+).$$

Podemos portanto supor que $k > 0$ e v é M_k-ocupado. Adotaremos como hipótese de indução que $d_k(v^-) \geq d_{k-1}(v^-)$, $\forall v^- \in V^-$. Pelo Corolário 2.16, v é M_{k+1}-ocupado. Sejam (v^-, v) e (u^-, v) as arestas (não necessariamente distintas) de M_k e M_{k+1}, respectivamente, que incidem em v.

Se $u^- \neq v^-$ então $d_k(u^-) \geq d_k(v^-)$, por definição de E_k. Se $u^- = v^-$ então $d_k(u^-) = d_k(v^-)$. Em ambos casos temos que

$$d_k(u^-) \geq d_k(v^-). \tag{2.6}$$

Portanto,

$$d_{k+1}(v) \overset{\text{def}}{=} 1 + d_k(u^-) \overset{(2.6)}{\geq} 1 + d_k(v^-) \overset{\text{H.I.}}{\geq} 1 + d_{k-1}(v^-) \overset{\text{def}}{=} d_k(v).$$

Passo caso 2 $v \in V^-$ e $k \geq 0$.

Adotaremos como hipótese de indução que $d_{k+1}(v^+) \geq d_k(v^+)$, $\forall v^+ \in V^+$. Consideraremos dois casos:

1. $d_{k+1}(\overline{M}_{k+1}(\{v\})) \geq d_{k+1}(\overline{M}_k(\{v\}))$. Neste caso temos

$$\begin{aligned} d_{k+1}(v) &\overset{\text{def}}{=} 1 + d_{k+1}(\overline{M}_{k+1}(\{v\})) \geq 1 + d_{k+1}(\overline{M}_k(\{v\})) \\ &\overset{\text{H.I.}}{\geq} 1 + d_k(\overline{M}_k(\{v\})) \overset{\text{def}}{=} d_k(v). \end{aligned}$$

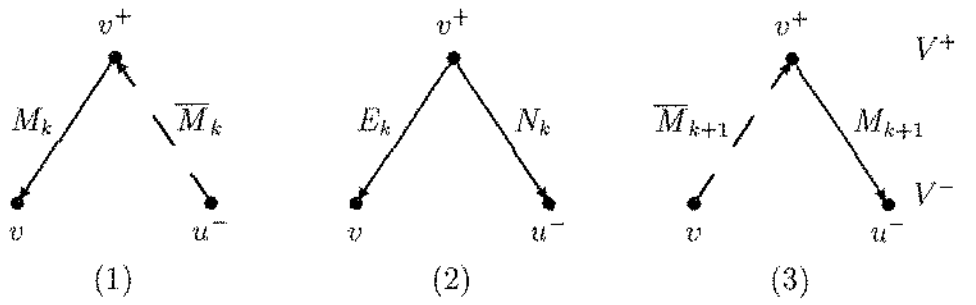


Figura 2.7: Passos para remover a aresta (v, v^+) de M_k.

2. $d_{k+1}(\overline{M}_{k+1}(\{v\})) < d_{k+1}(\overline{M}_k(\{v\}))$. Pelo Lema 2.13,

$$\overline{M}_{k+1}(\{v\}) \not\subseteq \overline{M}_k(\{v\}).$$

Pelo Corolário 2.16, v é M_k-ocupado e M_{k+1}-livre.

Seja (v, v^+) a aresta de M_k incidindo em v. Por definição de M_{k+1}, $(v, v^+) \in E_k$ e portanto para algum vértice $u^- \in V^-$, $(u^-, v^+) \in N_k \cap M_{k+1}$ (veja Figura 2.7). Temos então,

$$\begin{array}{lll} d_{k+1}(v) & \overset{\text{def e hip do item}}{=} & 1 + d_{k+1}(v^+) \overset{\text{H.I.}}{\geq} 1 + d_k(v^+) \\ & \overset{\text{def } T_k}{=} & d_k(u^-) \overset{\text{def } E_k}{\geq} d_k(v). \ \square \end{array}$$

Um corolário importante, obtido do Teorema 2.17, de onde deduzimos que $d_k(v)$ estima um limite inferior da distância de v a um vértice livre de V^+, através de um caminho M_k-estrito de origem em v, é apresentado a seguir (veja a Figura 2.8, onde os valores associados aos vértices representam d_k):

Corolário 2.18 *Sejam $k \geq 0$, $v^- \in V^-$ e $v^+ \in V^+$. Se $(v^-, v^+) \in M_k$, então $d_k(v^+) \leq 1 + d_k(v^-)$. Se $(v^-, v^+) \in \overline{M}_k$, então $d_k(v^-) \leq 1 + d_k(v^+)$.*

Demonstração: Considere inicialmente o caso em que $(v^-, v^+) \in \overline{M}_k$. Temos que

$$d_k(v^-) \overset{\text{def}}{=} 1 + d_k(\overline{M}_k(v^-)) \leq 1 + d_k(v^+).$$

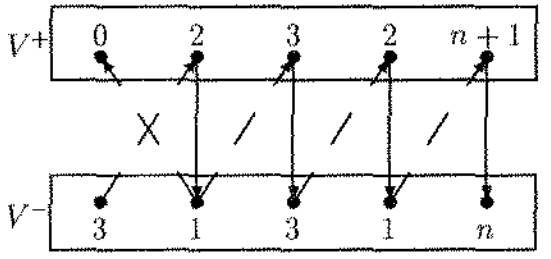


Figura 2.8: Exemplo de grafo com os valores d_k associados aos vértices.

Considere agora o caso em que $(v^-, v^+) \in M_k$. Necessariamente $k > 0$ e

$$d_k(v^+) \stackrel{\text{def}}{=} 1 + d_{k-1}(v^-) \stackrel{\text{Teo 2.17}}{\leq} 1 + d_k(v^-). \square$$

Lema 2.19 *Se $v^+ \in N_k(V^-)$ $(k \geq 0)$, então $d_{k+1}(v^+) \geq 2 + d_k(v^+)$.*

Demonstração: Seja $(v^-, v^+) \in N_k$. Logo, $\exists (u^-, v^+) \in M_{k+1}$, onde v^- e u^- não são necessariamente distintos. Assim como na demonstração do Teorema 2.17 (base e passo 1), temos que

$$d_k(u^-) \geq d_k(v^-). \tag{2.7}$$

Portanto,

$$d_{k+1}(v^+) \stackrel{\text{def}}{=} 1 + d_k(u^-) \stackrel{(2.7)}{\geq} 1 + d_k(v^-) \stackrel{\text{def } T_k}{=} 2 + d_k(v^+). \square$$

Lema 2.20 *Se $v^- \in (T_k \setminus N_k)(V^+)$ $(k \geq 0)$ e $d_k(v^-) < n$, então $d_{k+1}(v^-) > d_k(v^-)$.*

Demonstração: Como v^- é $(M_k \cup N_k)$-livre ($v^- \in (T_k \setminus N_k)(V^+)$), então v^- é M_{k+1}-livre. Logo, $\overline{M}_{k+1}(v^-) = \overline{M}_k(v^-)$ e, portanto,

$$d_{k+1}(v^-) \stackrel{\text{def}}{=} 1 + d_{k+1}(\overline{M}_{k+1}(\{v^-\})) = 1 + d_{k+1}(\overline{M}_k(\{v^-\})). \tag{2.8}$$

Por outro lado, $\forall v^+ \in \overline{M}_k(\{v^-\})$ que satisfaz $d_k(v^-) = 1 + d_k(v^+) = 1 + d_k(\overline{M}_k(\{v^-\}))$, temos que $v^+ \in T_k(\{v^-\})$. Logo, como N_k é um emparelhamento maximal em $G[T_k]$, então $v^+ \in N_k(V^-)$. Assim, pelo Lema 2.19, $d_{k+1}(v^+) \geq 2 + d_k(v^+)$, ou seja, (lembre-se da definição de $d_{k+1}(X)$)

$$d_{k+1}(\overline{M}_k(\{v^-\})) \geq 1 + d_k(\overline{M}_k(\{v^-\})). \tag{2.9}$$

Logo, de (2.8) e (2.9) temos que

$$d_{k+1}(v^-) \geq 2 + d_k(\overline{M}_k(\{v^-\})) \stackrel{\text{def}}{=} 1 + d_k(v^-). \ \square$$

2.5.2 Descrição

Inicialmente apresentaremos a estrutura de dados M_k, utilizada nos algoritmos, que é um vetor de n elementos indexado pelos vértices de V, onde

$$\underset{(v \in V)}{M_k(v)} := \begin{cases} \infty & \text{se } v \text{ é } M_k\text{-livre} \\ \text{elemento de } M_k(\{v\}) & \text{caso contrário.} \end{cases}$$

Uma versão algorítmica para determinar d_k é apresentada no Algoritmo 2.10. O Algoritmo 2.11 determina o novo emparelhamento M_{k+1}. Os autores Israeli e Shiloach, em [37], apresentam um algoritmo que determina emparelhamentos maximais em paralelo.

No Algoritmo 2.12, a função Pré-Emparelhamento usa dois parâmetros, l e a, para determinar um emparelhamento M em G satisfazendo a Propriedade (2.5).

Na Figura 2.9 é apresentado um esquema das chamadas de funções deste algoritmo.

/* Dados os vetores d_{k-1} e M_k, onde

$$\underset{(v \in V)}{M_k(v)} := \begin{cases} \infty & \text{se } v \text{ é } M_k\text{-livre} \\ \text{elemento de } M_k(\{v\}) & \text{caso contrário,} \end{cases}$$

e o grafo G, este algoritmo determina os valores do vetor d_k. Esta função utiliza $O(n^2)$ processadores num tempo $O(\log n)$–CREW. */

```
Rotule (d_{k-1}, M_k, G)
  para cada v ∈ V+ faça em paralelo
      se v é M_k-livre então
          d_k(v) ← 0
      senão
      início
          seja u o elemento de M_k({v});
          d_k(v) ← 1 + d_{k-1}(u)
      fim
  para cada u ∈ V− faça em paralelo
      d_k(u) ← 1 + min({n − 1} ∪ {d_k(v) : (u, v) ∈ E \ M_k});
  devolva d_k;
```

Algoritmo 2.10: Determina o vetor d_k.

/* Dados os vetores d_k e M_k, onde

$$M_k(v)_{(v \in V)} := \begin{cases} \infty & \text{se } v \text{ é } M_k\text{-livre} \\ \text{elemento de } M_k(\{v\}) & \text{caso contrário,} \end{cases}$$

e o grafo G, este algoritmo determina o vetor M_{k+1}. Este algoritmo utiliza $O(m)$ processadores num tempo $O\left(\log^3 n\right)$–CRCW. */

```
CalculaNovoEmparelhamento (d_k, M_k, G)
  M_{k+1} ← M_k;
  T_k ← ∅;
  para cada (u, v) ∈ E \ M_k e u ∈ V⁻ faça em paralelo
      se u é M_k-livre e d_k(u) = d_k(v) + 1 então
          acrescente (u, v) a T_k;
  N_k ← EmparelhamentoMaximal (G[T_k]);
  para cada (u, v) ∈ N_k e u ∈ V⁻ faça em paralelo
      se v é M_k-livre então
          M_{k+1}(u), M_{k+1}(v) ← v, u
      senão
      início
          seja w o elemento de M_k({v});
          se d_k(w) < d_k(u) ou (d_k(w) = d_k(u) e w < u) então
          início
              M_{k+1}(w) ← ∞;
              M_{k+1}(u), M_{k+1}(v) ← v, u
          fim
      fim
  devolva M_{k+1};
```

Algoritmo 2.11: Determina o vetor M_{k+1}.

```
/* Dados um grafo G com bipartição (V+, V-) e os parâmetros l e a, determina um emparelhamento M em G satisfazendo a Propriedade (2.5). Este algoritmo utiliza O(n^2) processadores num tempo O((nl/a) log^3 n)-CRCW. */

Pré-Emparelhamento (G, l, a);
  M_0 ← ∅;
  para cada v ∈ V faça em paralelo
      d_{-1}(v) ← 0;
  k ← 0;
  repita
      d_k ← Rotule (d_{k-1}, M_k, G);
      /* t estima um número de vértices livres de V- incidindo em
         caminhos aumentantes de comprimento no máximo l */
      para cada u ∈ V- faça em paralelo
          se u é M_k-livre e d_k(u) ≤ l então
              x_u ← 1
          senão
              x_u ← 0;
      t ← Σ_{u∈V-} x_u;
      se t ≥ a então
      início
          M_{k+1} ← CalculaNovoEmparelhamento (d_k, M_k, G);
          k ← k + 1;
      fim
  até que t < a;
  devolva M_k;
```

Algoritmo 2.12: Pré-emparelhamento (paralelo).

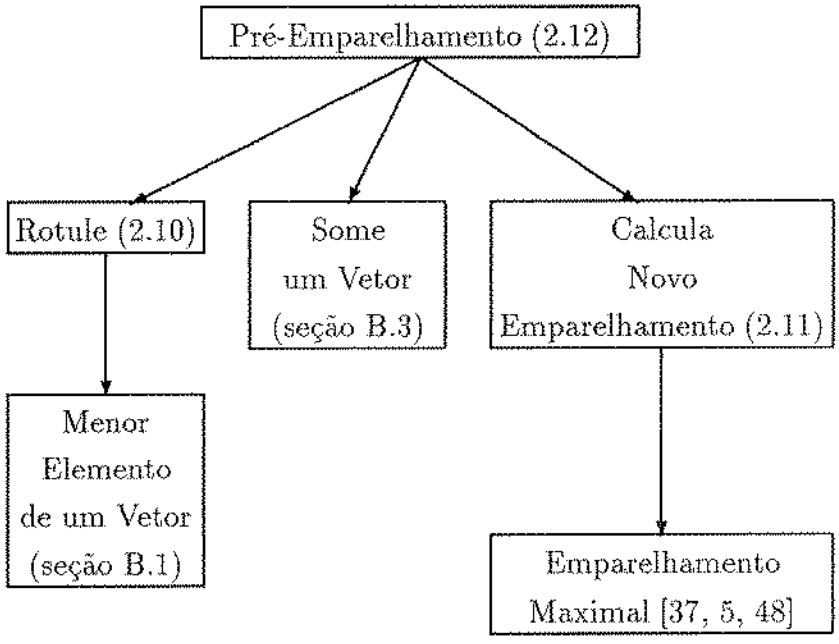


Figura 2.9: Esquema das chamadas das funções do algoritmo de Goldberg, Tarjan, Plotkin e Vaidya.

2.5.3 Complexidade e Número de Processadores

A complexidade do Algoritmo 2.10 (determinação do vetor d_k) é $O(\log n)$, usando $O(m+n^2) = O(n^2)$ processadores (CREW). Esta complexidade se deve à determinação do menor elemento de um vetor (veja Seção B.1).

No Algoritmo 2.11 (determinação do vetor M_{k+1}), as operações, com exceção da chamada da função EmparelhamentoMaximal, gastam $O(1)$ tempo, usando $O(m)$ processadores (CREW). A função EmparelhamentoMaximal, apresentada pelos autores Israeli e Shiloach em [37], possui uma complexidade de $O\left(\log^3 n\right)$ tempo, usando $O(m+n) = O(m)$ processadores (CRCW). Portanto, o Algoritmo 2.11 possui uma complexidade de $O\left(\log^3 n\right)$ tempo, usando $O(m)$ processadores (CRCW).

Assim, um laço na função Pré-Emparelhamento (Algoritmo 2.12) gasta $O\left(\log^3 n\right)$ tempo, usando $O(n^2)$ processadores (CRCW). Como veremos mais adiante, Corolário 2.22, este laço é executado no máximo $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ vezes. Logo, a complexidade desta função é: $O\left(\frac{nl}{a}\log^3 n\right)$ tempo, usando $O(n^2)$ processadores (CRCW).

Lema 2.21 *Seja S_k o seguinte conjunto:*

$$S_k := \{w \in V : d_k(w) \leq l\}.$$

Se $l \leq n-1$ e $a \geq 1$, então no mínimo a vértices de S_k ($k \geq 0$) aumentam o valor de sua função distância estimada na $(k+1)$-ésima iteração da Função Pré-Emparelhamento (Algoritmo 2.12), ou seja,

$$|\{w \in S_k : d_{k+1}(w) > d_k(w)\}| \geq a.$$

Demonstração: Seja esta a $(k+1)$-ésima iteração da função e $W := S_k \cap V^- \setminus VM$.

Como $l \leq n-1$, então, por definição de d_k, $\forall v^- \in W$, $v^- \in T_k(V^+)$ e $\forall v^+ \in T_k(\{v^-\})$, $d_k(v^-) = 1 + d_k(v^+)$, ou seja,

$$T_k(W) \subseteq S_k \cap V^+. \tag{2.10}$$

Portanto, $\forall v^- \in W$, se $v^- \in N_k(V^+)$, então $v^+ \in N_k(\{v^-\})$ satisfaz $d_{k+1}(v^+) \overset{\text{Lem 2.19}}{>} d_k(v^+)$. Pela Equação (2.10) temos que $v^+ \in S_k \cap V^+$. Se $v^- \notin N_k(V^+)$, então $d_{k+1}(v^-) \overset{\text{Lem 2.20}}{>} d_k(v^-)$.

Como, por definição da função (Propriedade (2.5)), $|W| \geq a$, então no mínimo a vértices de S_k aumentam o valor de sua função distância estimada na $(k+1)$-ésima iteração da função. □

Corolário 2.22 *Se $l \leq n-1$ e $a \geq 1$, então o laço principal da função Pré-Emparelhamento (Algoritmo 2.12) é executado no máximo $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ vezes.*

Demonstração: Como os valores da função d nunca diminuem (Teorema 2.17), então no máximo $l+1$ vezes um vértice $v \in V$, com $d(v) \leq l$, teve aumentado o valor de $d(v)$. Como temos n vértices, então no máximo $n(l+1)$ vezes aumentamos o valor da função d de vértices com $d \leq l$. Pelo Lema 2.21, no mínimo a vértices com $d \leq l$ aumentam o seu valor de d por iteração da função. Portanto, o número de iterações deste laço necessárias para satisfazer a Propriedade (2.5) é no máximo $\left\lfloor \frac{n(l+1)}{a} \right\rfloor$. Porém, o número de vértices livres de V^- incidindo em caminhos aumentantes é calculado antes de se determinar o novo emparelhamento M da iteração. Assim, precisamos de mais uma iteração para obtermos o número o de vértices livres de V^- incidindo em caminhos M-aumentantes. □

2.5.4 Observações

A seguir relacionamos algumas observações sobre o algoritmo.

1. Podemos reduzir a quantidade de memória utilizada pelo algoritmo, pois nas seqüências definidas só precisamos armazenar os valores mais recentes.

2. Se fizermos $l := n-1$ e $a := 1$, então o emparelhamento determinado no Algoritmo 2.12 é um emparelhamento máximo em G. As complexidades deste algoritmo, com estes valores para l e a, são: $O\left(n^2 \log^3 n\right)$ tempo, usando $O\left(n^2\right)$ processadores (CRCW).

Capítulo 3

Grafos Bipartidos Ponderados

Neste capítulo determinamos emparelhamentos máximos de custo mínimo em grafos bipartidos ponderados.

Conforme foi visto na Seção 1.1, um grafo G é *ponderado* se existe uma função c sobre as suas arestas, chamada de *custo*, que associa a cada aresta um valor real, ou seja, $c : E \to \Re$. Seja $X \subseteq E$: definimos o *custo do conjunto* X por

$$c(X) := \sum_{\alpha \in X} c(\alpha).$$

Na Seção 3.1 apresentamos um algoritmo (Busca Húngara [41]) que usa uma variável dual nos vértices para determinar um emparelhamento máximo de custo mínimo. Na Seção 3.2 apresentamos um algoritmo (Edmonds e Karp [17]) que utiliza o algoritmo de Dijkstra para realizar os ajustes nas variáveis duais. Neste algoritmo podemos utilizar uma estrutura de dados desenvolvida por Fredman e Tarjan, chamada de *Fibonacci heaps* [22], para diminuir a complexidade do algoritmo. Na Seção 3.3 é apresentado um algoritmo (Gabow [25]) que utiliza escalonamento para determinar um emparelhamento perfeito de custo mínimo. Nas Seções 3.4, 3.5 e 3.6 são apresentados, respectivamente: o algoritmo de Gabow e Tarjan [27] (seqüencial), o algoritmo de Goldberg, Plotkin e Vaidya [32] (paralelo), e o algoritmo de Gabow e Tarjan [26] (paralelo – EREW). Estes três algoritmos utilizam escalonamento e aproximação para determinar um emparelhamento perfeito de custo mínimo. A propósito, o algoritmo de Goldberg, Plotkin e Vaidya é sublinear.

3.1 Busca Húngara

- Autores: Kuhn [41, 42] e Munkres [47].
- Datas: 1955 e 1957.
- Tipo: Seqüencial.
- Complexidade: $O(mn^2 + n^3)$.

Kuhn, em [41], desenvolveu um método para se determinar um emparelhamento máximo de custo mínimo, utilizando caminhos alternados. Em reconhecimento aos trabalhos realizados pelos autores König e Egerváry em emparelhamentos máximos, Kuhn denominou este método de *busca húngara*.

Este algoritmo determina um emparelhamento máximo de custo mínimo num grafo bipartido.

3.1.1 Fundamentos

Seja G um grafo bipartido (V^+ e V^-) e ponderado (função custo $c : E \longrightarrow \Re$ sobre as suas arestas – Seção 1.1). Faça $y : V \longrightarrow \Re$ ser uma *variável dual* nos vértices de G. Dizemos que y é *c-independente* se

$$y(u) + y(v) \leq c(u,v), \forall (u,v) \in E. \tag{3.1}$$

Nota: Se c é constante, igual a 1, e se y, além de ser c-independente, assume apenas valores em $\{0,1\}$, então cada aresta do grafo possui no máximo um vértice extremo v com $y(v) = 1$, ou seja, o conjunto de vértices $v \in V$ com $y(v) = 1$ corresponde a um *conjunto independente*. Assim, a Definição (3.1) generaliza o conceito de independência.

Se y é c-independente, denotamos por $E_{y,c}$ o conjunto de arestas que satisfazem (3.1) com igualdade, chamadas de arestas *justas*, ou seja,

$$E_{y,c} := \{(u,v) \in E : y(u) + y(v) = c(u,v)\}.$$

A (y,c)-*folga* de uma aresta $(u,v) \in E$ é definida da seguinte forma:

$$\Delta_{y,c}(u,v) := c(u,v) - (y(u) + y(v)). \tag{3.2}$$

Portanto, se y é c-independente, então

$$\Delta_{y,c}(\alpha) \overset{(3.1)}{\geq} 0, \forall \alpha \in E, \tag{3.3}$$

com igualdade se e somente se α é uma aresta justa, ou seja, $\alpha \in E_{y,c}$.

Denotaremos por $G_{y,c}$ o grafo

$$G_{y,c} := (V, E_{y,c}).$$

Note que $G_{y,c}$ não é simplesmente um grafo gerado pelas arestas de $E_{y,c}$, mas contém, também, todos os vértices de G, ou seja, $G_{y,c}$ pode possuir vértices isolados.

Seja M um emparelhamento em G e y uma variável dual. Um par (M, y) é *c-admissível* se y é c-independente e as seguintes propriedades forem satisfeitas:

(i) $M \subseteq E_{y,c}$.

(ii) todo vértice M-livre de V^+ (ou de V^-) possui um valor de y no mínimo igual ao valor de y dos vértices restantes de V^+ (ou de V^-, respectivamente).

Nestas definições, sempre que c e/ou y for subentendido será omitido.

A seguir apresentamos um lema que relaciona o par admissível (M, y) com o problema de emparelhamentos máximos de custo mínimo.

Lema 3.1 *Se o par (M, y) é admissível, então, dentre os emparelhamentos de cardinalidade $|M|$ em G, M tem custo mínimo.*

Demonstração: Seja N um emparelhamento em G com a mesma cardinalidade de M. Somando (3.1) para cada aresta em N, temos

$$y(VN) \overset{(3.1)}{\leq} c(N).$$

Por outro lado, como $M \subseteq E_y$, então vale a igualdade em (3.1) para toda aresta em M; somando esta igualdade para toda aresta em M, obtemos

$$y(VM) \overset{\text{def } E_y}{=} c(M).$$

Pela Propriedade (ii) da definição de par admissível, para cada vértice v em $(VM \cap V^+ \setminus VN)$ e para cada vértice w em $(VN \cap V^+ \setminus VM)$, $y(v) \leq y(w)$. Ademais, os dois conjuntos têm o mesmo número de vértices. Logo,

$$y(VM \cap V^+) \leq y(VN \cap V^+).$$

Analogamente $y(VM \cap V^-) \leq y(VN \cap V^-)$. Portanto,

$$y(VM) \leq y(VN).$$

Assim,

$$c(M) = y(VM) \leq y(VN) \leq c(N). \square$$

Corolário 3.2 *Seja M um emparelhamento perfeito em G e y uma variável independente. Se $M \subseteq E_y$, então M é um emparelhamento perfeito em G de custo mínimo.* $\square$

Seja (M, y) um par admissível. Denotaremos por:

- $P(M, y)$ o conjunto dos caminhos M-alternados em G_y com origem em vértices M-livres de V^+.
- $S(M, y)$ e $T(M, y)$, respectivamente, os conjuntos de vértices $V^+ \cap VP(M, y)$ e $V^- \cap VP(M, y)$.
- $J(M, y)$ o conjunto das arestas de G com um extremo em $S(M, y)$ e o outro fora de $T(M, y)$.

Observe que se M for um emparelhamento máximo em G_y, então $J(M, y)$ e E_y são disjuntos e as arestas de M têm ambos os extremos ou nenhum extremo em $S(M, y) \cup T(M, y)$ (veja a Figura 3.1, onde as linhas pontilhadas representam arestas de $J(M, y)$). Esta afirmação segue imediatamente da otimalidade de M em G_y (veja a Figura 2.1, com G_y no papel de G).

Teorema 3.3 *Se $J(M, y) \neq \emptyset$ e M é um emparelhamento máximo em G_y, então existe uma variável dual z tal que (M, z) é admissível e*

$$T(M, y) \subset T(M, z).$$

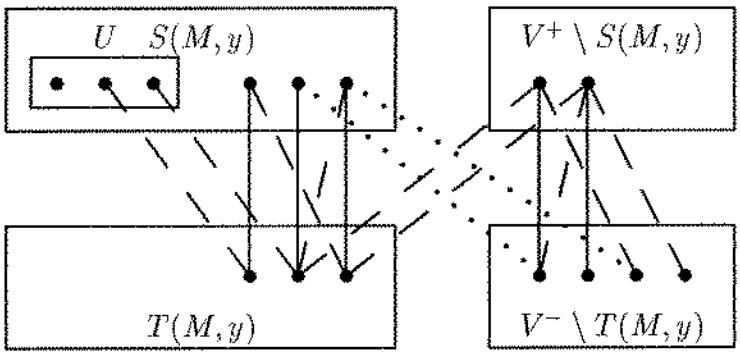


Figura 3.1: Caminhos M-alternados a partir de vértices M-livres de V^+ (denotados por U na figura) no grafo G_y e arestas $J(M,y)$ (pontilhadas). O emparelhamento M é máximo em G_y.

Demonstração: Conforme foi observado anteriormente, da otimalidade de M em G_y segue que $J(M,y)$ e E_y são disjuntos e toda aresta de M tem ambos os extremos ou nenhum extremo em $S(M,y) \cup T(M,y)$ (veja a Figura 3.1, onde as linhas pontilhadas representam arestas de $J(M,y)$).

Definiremos Δ como sendo o menor valor das folgas das arestas de $J(M,y)$ (Equação (3.2)), ou seja,

$$\Delta := \min\{\Delta_y(\alpha) : \alpha \in J(M,y)\}.$$

Note que Δ está bem definido (por hipótese, $J(M,y) \neq \emptyset$). Além disso, $\Delta > 0$, pois y é independente e os conjuntos $J(M,y)$ e E_y são disjuntos.

Seja uma nova variável dual $z : V \to \Re$ definida da seguinte forma:

$$z(v) := \begin{cases} y(v) - \Delta & \text{se } v \in T(M,y) \\ y(v) + \Delta & \text{se } v \in S(M,y) \\ y(v) & \text{nos demais casos.} \end{cases}$$

Com respeito a esta nova variável dual z, note que:

1. As arestas com ambos os extremos ou nenhum extremo em $S(M,y) \cup T(M,y)$ têm sua folga preservada.

2. As arestas com um extremo em $T(M,y)$ e o outro fora de $S(M,y)$ têm sua folga aumentada de Δ.

3. As arestas de $J(M,y)$ têm sua folga diminuída de Δ.

Portanto, z é independente (itens 1, 2 e 3 acima) e $M \subseteq EG_z$ (item 1 acima). Por outro lado, os vértices M-livres de V^+ pertencem a $S(M,y)$ e, portanto, para obter z somamos Δ ao seu y. Os vértices M-livres de V^- não pertencem a $T(M,y)$, ou seja, para obter z não diminuímos de Δ o seu y. Logo, como o par (M,y) é admissível, então z satisfaz a Propriedade (ii) da definição de par admissível. Portanto, o par (M,z) é admissível.

Seja T' o conjunto (não vazio) dos extremos em $V^- \setminus T(M,y)$ das arestas de $J(M,y) \cap E_z$. Logo,

$$T(M,y) \cup T' \subseteq T(M,z).$$

De fato, $T(M,y)$ é uma parte própria de $T(M,z)$. □

3.1.2 Descrição

Este algoritmo determina um emparelhamento máximo de custo mínimo num grafo bipartido.

No Algoritmo 3.1 vamos sempre trabalhar com o par admissível (M,y) (o par admissível (M,y), no Algoritmo 3.1, fica subentendido para os conjuntos S, T e J). Por exemplo, no início do algoritmo podemos ter $M := \emptyset$ e para todo vértice v em V, $y(v) = c_0/2$, onde $c_0 := \min\{c(\alpha) : \alpha \in E\}$.

Durante a execução do laço deste algoritmo podem ocorrer as seguintes situações:

1. M não é um emparelhamento máximo em G_y. Neste caso determinamos um novo emparelhamento M em G_y, onde M é agora um emparelhamento máximo em G_y. Para isto usamos o Algoritmo 2.3 (Primitivo).

2. M é um emparelhamento máximo em G_y e $J(M,y) \neq \emptyset$. Neste caso aplicamos o ajuste sobre a variável dual y feito na demonstração do Teorema 3.3 (y toma, também, o lugar de z na demonstração). Note que M, com a nova variável dual y, pode não ser mais um emparelhamento máximo em G_y.

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo c sobre as arestas de G, retorna $\langle M, y, (V^+ \setminus S) \cup T\rangle$, onde M é um emparelhamento máximo em G de custo mínimo, (M, y) é um par admissível e $(V^+ \setminus S) \cup T$ é uma cobertura mínima em G (Corolário 2.2).
A complexidade do algoritmo é $O((m+n)n^2)$. */

BuscaHúngara (G, c)
  $M \leftarrow \emptyset$;
  $c_0 \leftarrow \min\{c(\alpha) : \alpha \in E\}$;
  para todo $v \in V$ **faça**
    $y(v) \leftarrow c_0/2$;
  repita
    $\langle M, X\rangle \leftarrow$ Primitivo (G_y, M);
    $S \leftarrow V^+ \setminus X$;
    $T \leftarrow V^- \cap X$;
    $J \leftarrow \{(u,v) \in E : u \notin T \text{ e } v \in S\}$;
    se $J \neq \emptyset$ **então**
    início
      $\Delta \leftarrow \min\{c(u,v) - (y(u) + y(v)) : (u,v) \in J\}$;

$$y(v) \underset{(v \in V)}{\leftarrow} \begin{cases} y(v) - \Delta & \text{se } v \in T \\ y(v) + \Delta & \text{se } v \in S \\ y(v) & \text{nos demais casos;} \end{cases}$$

    fim
  até que $J = \emptyset$;
  devolva $\langle M, y, (V^+ \setminus S) \cup T\rangle$;

Algoritmo 3.1: Busca húngara.

3. M é um emparelhamento máximo em G_y e $J(M,y) = \emptyset$. Como $J(M,y)$ é vazio, então M também é um emparelhamento máximo em G; logo, terminamos o laço do algoritmo.

Portanto, o emparelhamento M^* retornado pelo laço é máximo. Pelo Lema 3.1, o seu custo é mínimo, ou seja, M^* é um emparelhamento máximo de custo mínimo.

3.1.3 Complexidade

A complexidade de uma chamada da função Primitivo (Seção 2.1.3) no Algoritmo 3.1 (busca húngara) é

$$O\left((m+n)\left(1+|M|-|M_0|\right)\right),$$

onde M_0 é o emparelhamento com o qual chamamos a função e M é o emparelhamento que retorna da função. Seja x o número de iterações do laço do Algoritmo 3.1. Então, a complexidade global das chamadas da função Primitivo é:

$$O\left((m+n)(x+n)\right).$$

A complexidade das demais operações do algoritmo é:

$$O\left((m+n)x\right).$$

Assim, a complexidade deste algoritmo é:

$$O\left((m+n)(x+n)\right).$$

Pelo Teorema 3.3, fazemos $O(n)$ ajustes da variável dual para o mesmo emparelhamento M; por outro lado, a seqüência de cardinalidades dos emparelhamentos obtidos é crescente e tem portanto tamanho $O(n)$. Logo, $x = O(n^2)$ e, portanto, a complexidade deste algoritmo é:

$$O\left((m+n)n^2\right).$$

3.1.4 Observações

A cada nova iteração no Algoritmo 3.1, em que o emparelhamento M da iteração anterior não aumentou a sua cardinalidade, não precisamos iniciar uma nova busca em largura na função Primitivo, basta continuar a busca, interrompida na iteração anterior, a partir das novas arestas justas (folga zero) entre S e $V^- \setminus T$ (arestas do conjunto $J(M, y) \cap E_z$ na demonstração do Teorema 3.3).

De fato, o nosso algoritmo determina um emparelhamento de cardinalidade k (k fixo) de custo mínimo.

O autor Bertsekas, em [8], apresenta um algoritmo que em alguns aspectos é semelhante ao algoritmo de busca húngara e em outros é bastante distinto. A complexidade média deste algoritmo é significativamente melhor do que do método da busca húngara.

3.2 Algoritmo de Edmonds e Karp

- Autores: Edmonds e Karp [17].
- Data: 1972.
- Tipo: Seqüencial.
- Complexidade usando *fila linear*[1] [1, 44]: $O(mn \log n)$.
- Complexidade usando *fila de Fibonacci* [22]: $O(mn + n^2 \log n)$.

Neste algoritmo é utilizado o algoritmo de Dijkstra [13] para auxiliar no ajuste da variável dual independente (Seção 3.1). A cada laço deste algoritmo é feito um ajuste que garante a existência de pelo menos um caminho aumentante de arestas justas. Assim, este algoritmo possui $O(n)$ iterações. Os autores Fredman e Tarjan, em 1984, desenvolveram uma implementação para fila de prioridade, uma estrutura de dados denominada *fila de Fibonacci* [22], que diminuiu a complexidade deste algoritmo para

$$O\left(mn + n^2 \log n\right).$$

Este algoritmo determina um emparelhamento máximo de custo mínimo num grafo bipartido. Na realidade, este algoritmo é uma generalização do algoritmo da Seção 2.1 (Primitivo).

3.2.1 Fundamentos

Seja G um grafo bipartido (V^+ e V^-) e ponderado (função custo $c : E \longrightarrow \Re$ sobre as suas arestas).

Se $y : V \rightarrow \Re$ é uma variável dual nos vértices de G, então o *y-comprimento* de um caminho P em G é definido da seguinte forma (veja Seção 3.1.1, Equação (3.2)):

$$\Delta_y(P) := \sum_{\alpha \in EP} \Delta_y(\alpha).$$

[1] Do inglês *heap*; vetor usado para representar uma *fila de prioridade*.

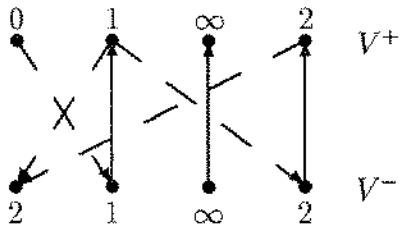


Figura 3.2: Grafo bipartido com folga unitária nas arestas fora do emparelhamento. Os valores associado aos seus vértices representam d.

Se o caminho for degenerado, ou seja, contém apenas um vértice, definimos como sendo zero o seu y-comprimento. Assim, se y é independente, então

$$\Delta_y(P) \overset{(3.3)}{\geq} 0. \tag{3.4}$$

Se o par (M, y) é admissível, então definimos $d : V \to \Re$ como sendo uma variável sobre os vértices de G, onde $d(v)$ $(v \in V)$ é o menor y-comprimento entre os caminhos M-alternados com origem num vértice M-livre de V^+ e destino v; se tal caminho não existe, então $d(v) := \infty$ (veja Figura 3.2).

Seja o par admissível (M, y), onde M não é um emparelhamento máximo em G, e P é um caminho M-aumentante de y-comprimento mínimo. Seja u o vértice M-livre de P em V^- (pela definição de d, $d(u) < \infty$). Então, o y-comprimento de P é $d(u)$. Defina $d_0 := d(u)$ e uma nova variável dual $z : V \to \Re$ da seguinte forma:

$$z(v) := \begin{cases} y(v) - (d_0 - d(v)) & \text{se } v \in V^- \text{ e } d(v) \leq d_0 \\ y(v) + (d_0 - d(v)) & \text{se } v \in V^+ \text{ e } d(v) \leq d_0 \\ y(v) & \text{nos demais casos.} \end{cases} \tag{3.5}$$

Proposição 3.4 *As arestas de M permanecem justas com respeito à variável dual z.*

Demonstração: Seja $(v^-, v^+) \in M$, onde $v^- \in V^-$. Como $\Delta_y(v^-, v^+) = 0$ e em todo caminho M-alternado com origem num vértice M-livre de V^+, a

passagem de um vértice em V^- para outro em V^+ é feita através de uma aresta de M, então $d(v^+) = d(v^-)$. Portanto, por definição de z,

$$\Delta_z(v^-, v^+) = \Delta_y(v^-, v^+) + \begin{cases} (d_0 - d(v^-)) - (d_0 - d(v^+)) & \text{se } d(v^-) \leq d_0 \\ 0 & \text{caso contrário.} \end{cases}$$

Logo, $\Delta_z(v^-, v^+) = \Delta_y(v^-, v^+) = 0$. □

Proposição 3.5 *Para toda aresta (v^-, v^+) em $E \setminus M$ vale a desigualdade*

$$\Delta_z(v^-, v^+) \geq 0,$$

com igualdade se a aresta é usada por P.

Demonstração: Seja $v^- \in V^-$. Provaremos inicialmente a desigualdade. Considere os seguintes casos:

Caso 1 $d(v^+) > d_0$. Por definição de z,

$$\Delta_z(v^-, v^+) = \Delta_y(v^-, v^+) + \begin{cases} d_0 - d(v^-) & \text{se } d(v^-) \leq d_0 \\ 0 & \text{caso contrário.} \end{cases}$$

Portanto, $\Delta_z(v^-, v^+) \geq \Delta_y(v^-, v^+) \geq 0$.

Caso 2 $d(v^+) \leq d_0$. Por definição de z,

$$\Delta_z(v^-, v^+) = \Delta_y(v^-, v^+) + \begin{cases} d(v^+) - d(v^-) & \text{se } d(v^-) \leq d_0 \\ d(v^+) - d_0 & \text{caso contrário.} \end{cases}$$

Portanto,

$$\Delta_z(v^-, v^+) \geq \Delta_y(v^-, v^+) + d(v^+) - d(v^-), \tag{3.6}$$

com igualdade se $d(v^-) \leq d_0$.

Por outro lado, em todo caminho M-alternado com origem num vértice M-livre de V^+, a passagem de um vértice em V^+ para outro em V^- é sempre feita através de uma aresta em $E \setminus M$. Assim,

$$d(v^+) + \Delta_y(v^-, v^+) \geq d(v^-), \tag{3.7}$$

com igualdade se a aresta é usada por P.

De (3.7) e (3.6) temos que

$$\Delta_z(v^-, v^+) \geq 0, \tag{3.8}$$

com igualdade se $d(v^-) \leq d_0$ e a aresta é usada por P.

Assim, a desigualdade enunciada está provada em ambos os casos.

Provaremos agora a igualdade enunciada. Logo, suponha que $(v^-, v^+) \in EP$. Portanto, por definição de d e como todas as arestas de G possuem folga não negativa, então $d(v^-) \leq d_0$ e $d(v^+) \leq d_0$. Assim, o caso 2 se aplica e (3.8) é satisfeito com igualdade. □

Corolário 3.6 *O par $(M \oplus EP, z)$ é admissível.*

Demonstração: Pelas Proposições 3.4 e 3.5, z é independente e $M \oplus EP \subseteq E_z$. Por outro lado, o y dos vértices $M \oplus EP$-livres de V^+ aumenta de d_0 (maior valor admissível) para obter z. O y dos vértices $M \oplus EP$-livres de V^- não são subtraídos para obter z, pois d_0 é um y-comprimento mínimo entre os caminhos M-aumentantes. Logo, como o par (M, y) é admissível, então z satisfaz a Propriedade (ii) da definição de par admissível. Assim, o par $(M \oplus EP, z)$ é admissível. □

3.2.2 Descrição

Como no Algoritmo 3.1 (busca húngara), vamos sempre trabalhar com um par admissível (M, y), usando inclusive a mesma inicialização $(\emptyset, c_0/2)$.

O laço deste algoritmo consiste em determinar um caminho aumentante P de y-comprimento mínimo. Como as arestas de G possuem folga não negativa, então podemos usar o algoritmo de Dijkstra para determinar P e o valor de d para os vértices de G (note que d_0 é zero caso exista um caminho M-aumentante em G_y). O emparelhamento M é então substituído por $M \oplus EP$ (como podemos observar no exemplo da Figura 3.3, existem grafos onde a cada iteração deste algoritmo temos apenas um caminho M-aumentante em G_y) e y por z como definido em (3.5) (Corolário 3.6).

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo c sobre as arestas de G, retorna $\langle M, y, (V^+ \setminus S) \cup T\rangle$, onde M é um emparelhamento máximo em G de custo mínimo, (M,y) é um par admissível e $(V^+ \setminus S) \cup T$ é uma cobertura mínima em G (Corolário 2.2).
A complexidade do algoritmo é $O(mn + n^2 \log n)$. */

EdmondsKarp (G,c)
  $M \leftarrow \emptyset$;
  $c_0 \leftarrow \min\{c(\alpha) : \alpha \in E\}$;
  para todo $v \in V$ **faça**
    $y(v) \leftarrow c_0/2$;
  repita
    $\langle \exists \text{cam}, P, d\rangle \leftarrow$ Dijkstra (G, M, c, y);
    se $\exists$cam **então**
    início
      seja u o último vértice do caminho P;
      $d_0 \leftarrow d(u)$;
      $$y(v) \underset{(v \in V)}{\leftarrow} \begin{cases} y(v) - (d_0 - d(v)) & \text{se } v \in V^- \text{ e } d(v) \leq d_0 \\ y(v) + (d_0 - d(v)) & \text{se } v \in V^+ \text{ e } d(v) \leq d_0 \\ y(v) & \text{nos demais casos;} \end{cases}$$
      $M \leftarrow M \oplus EP$;
    fim
  até que $\nexists$cam;
  $S \leftarrow \{v : v \in V^+ \text{ e } d(v) < \infty\}$;
  $T \leftarrow \{v : v \in V^- \text{ e } d(v) < \infty\}$;
  devolva $\langle M, y, (V^+ \setminus S) \cup T\rangle$;

Algoritmo 3.2: Edmonds e Karp.

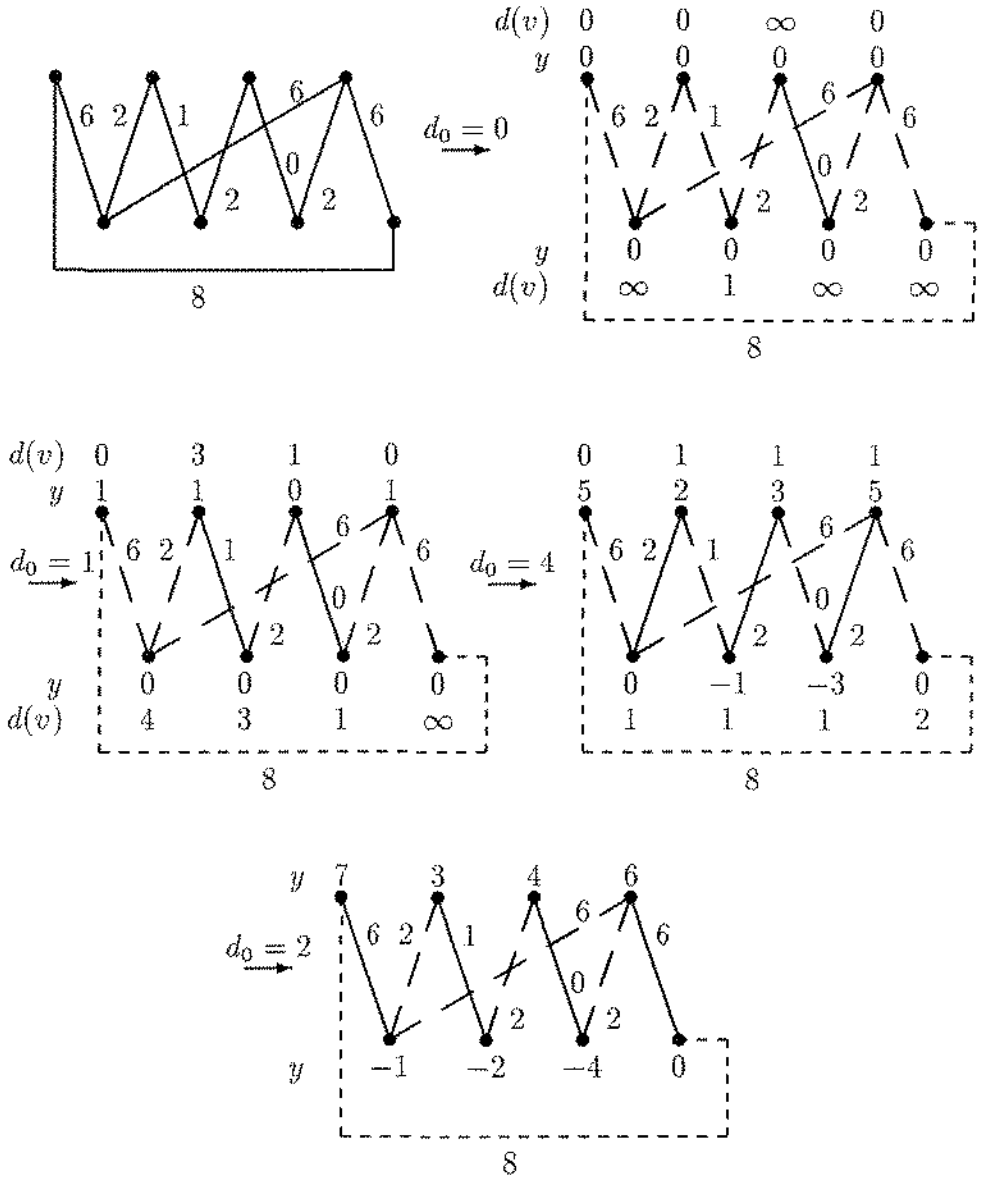


Figura 3.3: Exemplo de um grafo onde a cada iteração do algoritmo de Edmonds e Karp existe apenas um caminho M-aumentante em G_y.

Como executamos este algoritmo enquanto existirem caminhos aumentantes, então o emparelhamento M^* retornado pelo algoritmo é um emparelhamento máximo em G. Pelo Lema 3.1, o seu custo é mínimo, ou seja, M^* é um emparelhamento máximo de custo mínimo.

3.2.3 Complexidade

Seja R a implementação da fila de prioridade usada no algoritmo de Dijkstra [13] para representar os vértices com *rótulo temporário*. Sejam i, r e p, respectivamente, as complexidades de R para inserir um elemento, remover um elemento mínimo e reduzir a prioridade de um elemento. Como no algoritmo Dijkstra fazemos

- exatamente n inserções
- no máximo n remoções de elementos mínimos
- no máximo m reduções de prioridade

em R, então a complexidade do algoritmo de Dijkstra, utilizando R é

$$O(ni + nr + mp) = O(mp + n(i + r)).$$

O laço do Algoritmo 3.2 (Edmonds e Karp) é executado no máximo $O(n)$ vezes, pois a cada iteração diminuímos de dois o número de vértices M-livres. Assim, a complexidade deste algoritmo é

$$O\left(mnp + n^2(i + r)\right).$$

Podemos utilizar uma *fila linear* para implementar a fila de prioridade R [1, 44]. Neste caso, $i = r = p = O(\log h)$, onde h é o tamanho máximo da fila. Portanto, $h = n$ e a complexidade do algoritmo ficaria

$$O\left(mn \log n + n^2 \log n\right).$$

Os autores Fredman e Tarjan desenvolveram uma outra implementação para a fila de prioridade, uma estrutura de dados denominada *fila de Fibonacci* [22], que possui $i = p = O(1)$ e $r = O(\log h)$. As complexidades desta estrutura representam o *custo amortizado* das operações [50]. Portanto, se R é uma fila de Fibonacci, então $h = n$ e a complexidade do algoritmo ficaria

$$O\left(mn + n^2 \log n\right).$$

3.2.4 Observação

Os autores Jonker e Volgenant, em [38], apresentam um algoritmo que contém poucas rotinas de inicialização e uma implementação especial do Algoritmo de Díjkstra. São comparados os resultados computacionais do desempenho deste algoritmo com o de outros algoritmos.

3.3 Algoritmo com Escalonamento – Gabow

- Autor: Gabow [25].
- Data: 1985.
- Tipo: Seqüencial.
- Complexidade:[2] $O\left(mn^{3/4}\log U + n^{7/4}\log U\right)$.
- Restrições: integralidade dos custos das arestas.

É feito um escalonamento sobre os custos das arestas do grafo. A grosso modo, a técnica de escalonamento toma, para cada aresta, um custo igual à metade do custo original e obtém, recursivamente, uma solução para a nova função custo. A partir desta solução e mediante poucos ajustes, uma solução é obtida para o problema original. É evidente que o uso desta técnica pressupõe a integralidade da função custo.

O uso de escalonamento em algoritmos para determinar emparelhamentos perfeitos de custo mínimo em grafos bipartidos foi introduzido, em 1972, por Edmonds e Karp [17]. O algoritmo aqui apresentado é devido a Gabow [25].

Neste algoritmo determinamos emparelhamentos perfeitos de custo mínimo em grafos bipartidos; caso não exista tal emparelhamento perfeito, o algoritmo detecta este fato.

3.3.1 Descrição

Seja G um grafo bipartido (V^+ e V^-) e c a sua função custo. Por causa do escalonamento nos custos das arestas, a função custo deve assumir valores inteiros.

Por simplicidade, assumiremos que os valores da função custo, neste algoritmo, são inteiros não negativos, ou seja, $c : E \to \mathcal{Z}_{\geq 0}$. Note que, se a função custo possui valores negativos, basta subtrair desta função o valor mínimo assumido por ela (Proposição 1.7) que o conjunto solução deste emparelhamento não se altera (Corolário 1.8). O valor de U desta nova função é no máximo duas vezes o valor de U da função original, ou seja, a complexidade fica inalterada.

[2] U é o maior valor absoluto entre todos os custos (inteiros) das arestas do grafo.

Este algoritmo primeiro resolve o problema de emparelhamento perfeito de custo mínimo num grafo cujo custo das arestas são a metade do original. Isto fornece uma variável independente inicial, no novo grafo, "melhor" (com uma folga menor) em relação ao grafo original.

A Função recursiva Escalonamento (Algoritmo 3.3) implementa esta estratégia. Durante a execução deste algoritmo, o par (M, y) retornado nas chamadas recursivas é sempre admissível (Proposição 3.7 a seguir). Na execução deste algoritmo podem ocorrer as seguintes situações:

1. Todas as arestas possuem custo zero. Neste caso fazemos $y := 0$ e utilizamos o Algoritmo 2.5 (Hopcroft e Karp) para determinar um emparelhamento máximo M em G (note que $G_y = G$). Se M for um emparelhamento perfeito, retornamos o par admissível (M, y); caso contrário, interrompemos o algoritmo e devolvemos M e $(V^+ \setminus S) \cup T$ como uma certidão da não existência de um emparelhamento perfeito em G (Corolário 2.2).

2. Existem arestas com custo diferente de zero. Chamamos, recursivamente, este algoritmo com o mesmo grafo G, porém com o custo das arestas valendo $\left\lfloor \frac{c(\alpha)}{2} \right\rfloor$ $(\alpha \in E)$. Isto é feito para podermos obter uma "boa" inicialização na variável independente desta chamada do algoritmo. O restante deste algoritmo é similar ao Algoritmo 3.2 (Edmonds e Karp), porém ao invés de em cada iteração do laço determinarmos apenas um caminho M-aumentante admissível, determinamos um emparelhamento máximo em G_y (algoritmo de Hopcroft e Karp).

Chamamos a atenção para a existência de comandos entre $\langle\langle$ e $\rangle\rangle$. Tais comandos foram colocados apenas para facilitar a análise de complexidade, sendo absolutamente irrelevantes para o algoritmo propriamente dito.

Proposição 3.7 *Se existe um emparelhamento perfeito em G, então o par (M, y) retornado pelo algoritmo é admissível, onde M é perfeito, ou seja, M é um emparelhamento perfeito de custo mínimo em G.*

Demonstração: Prova por indução no número de chamadas do algoritmo. O caso base, quando todas as arestas possuem custo zero, é trivial $(G_y = G)$.

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \to \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y\rangle$, onde (M, y) é um par admissível e M é um emparelhamento perfeito em G de custo mínimo. Caso contrário, interrompe o algoritmo e devolve $\langle \text{não}, M, (V^+ \setminus S) \cup T\rangle$, onde M e $(V^+ \setminus S) \cup T$ é uma certidão da não existência de um emparelhamento perfeito em G (Corolário 2.2).
A complexidade do algoritmo é $O\left(mn^{3/4} \log U + n^{7/4} \log U\right)$. */

Escalonamento (G, c)
  se $\forall \alpha \in E$, $c(\alpha) = 0$ **então início**
    $\langle M, W\rangle \leftarrow$ HopcroftKarp $(G, \emptyset)$;
    se M é um emparelhamento perfeito **então devolva** $\langle \text{sim}, M, y = 0\rangle$
    senão interrompa $\langle \text{não}, M, W\rangle$;
  fim
  senão início
    para todo $\alpha \in E$ **faça**
      $c'(\alpha) \leftarrow \left\lfloor \frac{c(\alpha)}{2} \right\rfloor$;
    $\langle \exists \text{emp}, M', y'\rangle \leftarrow$ Escalonamento (G, c');
    para todo $v \in V$ **faça**
      $y(v) \leftarrow 2y'(v)$;
    $M \leftarrow \emptyset$; $\langle\langle l \leftarrow 0; d_+ \leftarrow 0;\rangle\rangle$
    repita
      $\langle \exists \text{cam}, P, d\rangle \leftarrow$ Dijkstra (G, M, c, y);
      seja u o último vértice do caminho P;
      $d_0 \leftarrow d(u)$;

$$y(v) \underset{(v \in V)}{\leftarrow} \begin{cases} y(v) - (d_0 - d(v)) & \text{se } v \in V^- \text{ e } d(v) \leq d_0 \\ y(v) + (d_0 - d(v)) & \text{se } v \in V^+ \text{ e } d(v) \leq d_0 \\ y(v) & \text{nos demais casos;} \end{cases}$$

      $\langle M, W\rangle \leftarrow$ HopcroftKarp (G_y, M);
      $\langle\langle l \leftarrow l + 1; d_+ \leftarrow d_0 + d_+;\rangle\rangle$ $\langle\langle \lhd \rangle\rangle$
    até que M seja perfeito;
    devolva $\langle \text{sim}, M, y\rangle$;
  fim

Algoritmo 3.3: Escalonamento – Gabow.

Como hipótese de indução suponha que o par (M', y') retornado pela recursão seja admissível. Logo, y no início do laço é independente, pois $\forall (u,v) \in E$:

$$c(u,v) \overset{\text{def } c'}{\geq} 2c'(u,v) \overset{\text{H.I.}}{\geq} 2y'(u) + 2y'(v) \overset{\text{def } y}{=} y(u) + y(v).$$

Pelas Proposições 3.4 e 3.5, mantemos a independência de y durante o laço do algoritmo. Como, pelo enunciado, existe um emparelhamento perfeito em G, este laço termina quando M é perfeito e, além disso, $M \subseteq E_y$. Assim, o par (M, y) é admissível. □

Nota: poderíamos tentar generalizar o algoritmo, mudando a condição de término do laço para **até que** M seja máximo em G. Como podemos observar no exemplo da Figura 3.4, esta modificação não produz um algoritmo correto.

3.3.2 Complexidade

Como os custos são valores inteiros, então o número de chamadas recursivas deste algoritmo, até executarmos a *base da recursão* (quando todas as arestas possuem custo zero), é $O(\log U)$. Assim, para obtermos a complexidade afirmada, resta mostrar que a complexidade de cada *patamar* do algoritmo (isto é, a complexidade obtida ignorando-se a chamada recursiva) é

$$O\left(mn^{3/4} + n^{7/4}\right).$$

Note que a base da recursão possui uma complexidade $O\left(mn^{1/2} + n^{3/2}\right)$, pois é basicamente uma chamada do algoritmo de Hopcroft e Karp. Assim, resta mostrar que a complexidade x do laço **repita** satisfaz

$$x = O\left(mn^{3/4} + n^{7/4}\right). \tag{3.9}$$

A seguir analisaremos a complexidade x num patamar.

Teorema 3.8 *A seguinte afirmação é invariante no laço* **repita**, *no ponto* $\lhd$*:*

$$\left|V^+ \setminus VM\right| d_+ \leq \frac{n}{2}.$$

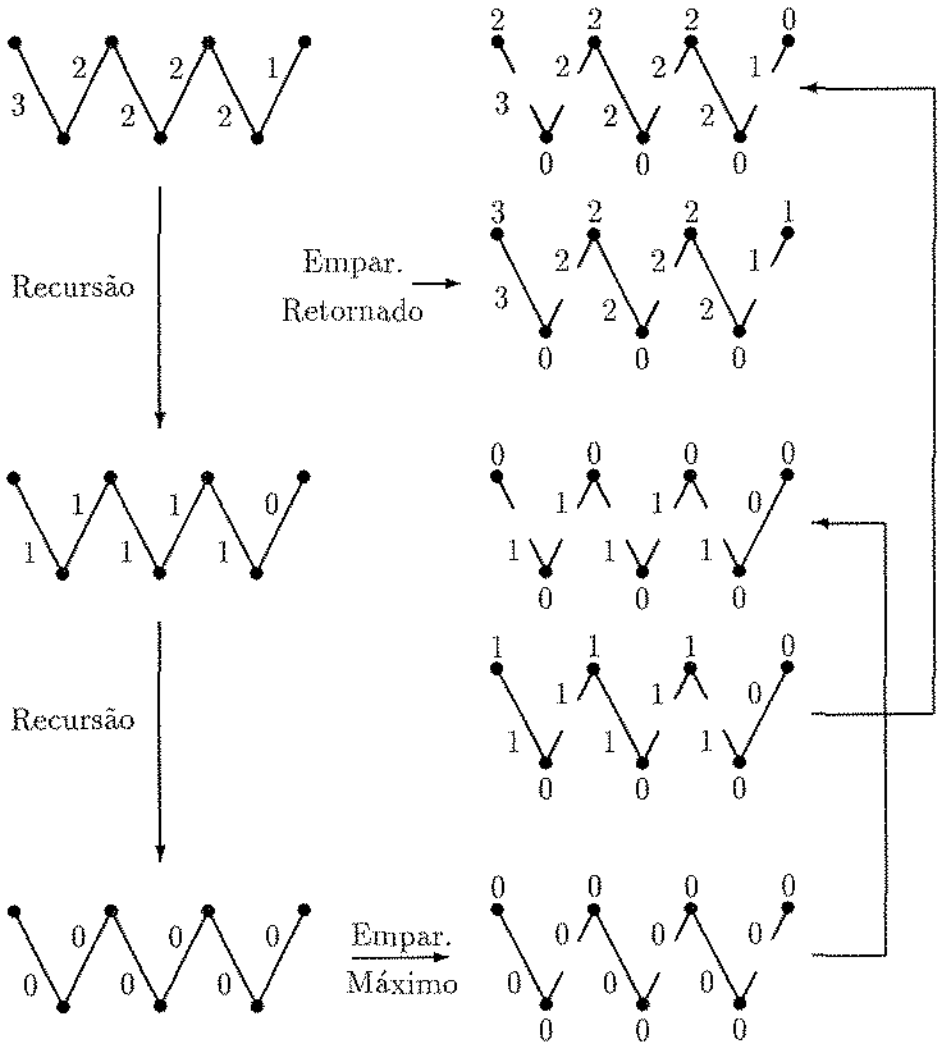


Figura 3.4: Exemplo usando o Algoritmo 3.3 modificado para obter um emparelhamento máximo (não perfeito). O emparelhamento retornado não é de custo mínimo (existe um emparelhamento máximo de custo 5).

Demonstração: Seja y_0 a variável independente inicial (ou seja, $y_0 = 2y'$) do patamar da obtenção de M e y a variável independente obtida junto com M. Logo, para todo vértice $v \in V \setminus VM$,

$$y(v) - y_0(v) = \begin{cases} d_+ & \text{se } v \in V^+ \\ 0 & \text{se } v \in V^- \end{cases}$$

Portanto,

$$y(V \setminus VM) - y_0(V \setminus VM) = \left|V^+ \setminus VM\right| d_+. \tag{3.10}$$

Como y_0 é independente e $M \subseteq E_y$, então

$$y_0(VM) \leq c(M) = y(VM).$$

Logo,

$$0 \leq y(VM) - y_0(VM). \tag{3.11}$$

Como y é independente, M' é perfeito e $M' \subseteq E_{y'}$, então

$$y(V) \leq c(M') \leq 2c'(M') + \frac{n}{2} \overset{\text{H.I.}}{=} 2y'(V) + \frac{n}{2} = y_0(V) + \frac{n}{2},$$

ou seja,

$$\frac{n}{2} \geq y(V) - y_0(V). \tag{3.12}$$

Assim, se fizermos (3.10) − (3.11) + (3.12), obtemos a desigualdade desejada. □

Lema 3.9 *As seguintes afirmações são invariantes no laço* **repita**, *no ponto* $\lhd$*:*

(i) $l \leq 1 + \frac{n}{2|V^+ \setminus VM|}$.

(ii) $l \leq 1 + (2n)^{1/2}$.

Demonstração: Provaremos primeiro (i). É fácil ver que $l \leq 1 + d_+$, pois a cada incremento de l, a variável d_+ é incrementada de d_0, que por sua vez é sempre positivo, exceto talvez no primeiro incremento que l sofre, quando d_0 pode ser zero. Assim, pelo Teorema 3.8,

$$l \leq 1 + \frac{n}{2\left|V^+ \setminus VM\right|}.$$

Para provarmos (ii), seja k um valor no intervalo $1 \leq k \leq n$. Tome l_k como sendo o maior valor de l tal que $|V^+ \setminus VM| \geq k$ e l^* como sendo o último (maior) valor de l. É claro que

$$l^* \leq l_k + k \overset{\text{(i)}}{\leq} 1 + \frac{n}{2k} + k.$$

O valor mínimo de l^* ocorre quando $k = \left(\frac{n}{2}\right)^{1/2}$. Portanto, temos a desigualdade desejada. □

Agora estamos em condições de demonstrar (3.9).

O algoritmo de Dijkstra pode ser implementado em $O(m + n \log n)$, usando uma fila de Fibonacci (Seção 3.2.3). Assim, pelo Lema 3.9 (ii), a complexidade total desta função e do ajuste dual num patamar é

$$O\left(mn^{1/2} + n^{3/2} \log n\right) \leq O\left(mn^{3/4} + n^{7/4}\right).$$

Assim, resta mostrar que a complexidade total de todas as execuções do algoritmo de Hopcroft e Karp no laço é

$$O\left(mn^{3/4} + n^{7/4}\right).$$

Uma execução do algoritmo de Hopcroft e Karp, neste laço, gasta

$$O\left((m+n)\min\{n^{1/2}, a\}\right)$$

tempo, onde a é o número de caminhos aumentantes (veja Seção 2.2.3). Seja k um valor no intervalo $1 \leq k \leq n$. Tome l_k como sendo o maior valor da variável l no ponto $\lhd$ tal que $|V^+ \setminus VM| \geq k$. Assim, a complexidade do total de execuções do algoritmo de Hopcroft e Karp no laço é

$$O\left((m+n)\left(n^{1/2} l_k + k\right)\right).$$

Pelo Lema 3.9 (i),

$$l_k \leq 1 + \frac{n}{2k} = O\left(\frac{n}{k}\right).$$

Assim,

$$O\left((m+n)\left(\frac{n^{3/2}}{k} + k\right)\right).$$

A expressão $\frac{n^{3/2}}{k} + k$ tem valor mínimo para $k = n^{3/4}$, o que fornece a igualdade (3.9). A análise da complexidade está completa.

3.4 Algoritmo com Escalonamento e Aproximação – Gabow e Tarjan

- Autores: Gabow e Tarjan [27].
- Data: 1989.
- Tipo: Seqüencial.
- Complexidade: $O\left(mn^{1/2}\log nU\right)$.
- Restrições: integralidade dos custos das arestas.

Este algoritmo utiliza escalonamento (nos custos das arestas) e aproximação para determinar um emparelhamento perfeito de custo mínimo; caso não exista tal emparelhamento perfeito, o algoritmo detecta este fato. Os algoritmos com aproximação determinam um emparelhamento perfeito que pode não ser um emparelhamento perfeito de custo mínimo, porém possui um custo próximo do mínimo. Logo, se a diferença entre os custos dos emparelhamentos perfeitos em G, a não ser quando possuem o mesmo custo, é "grande", então o emparelhamento perfeito retornado, usando aproximação, é na realidade um de custo mínimo.

3.4.1 Fundamentos

Seja G um grafo bipartido (V^+ e V^-) e c a sua função custo. O uso da técnica de escalonamento pressupõe integralidade da função custo.

Assim como na Seção 3.3 (Gabow), por simplicidade assumiremos que os valores da função custo, neste algoritmo, são inteiros não negativos, ou seja, $c : E \to \mathcal{Z}_{\geq 0}$. Note que, se a função custo possui valores negativos, basta subtrair desta função o valor mínimo assumido por ela (Proposição 1.7) que o conjunto solução deste emparelhamento não se altera (Corolário 1.8). O valor de U desta nova função é no máximo duas vezes o valor de U da função original, ou seja, a complexidade fica inalterada.

Definimos a função custo $c_M : E \to \mathcal{Z}_{\geq 0}$ da seguinte forma:

$$c_M(\alpha) := \begin{cases} c(\alpha) & \text{se } \alpha \in M \\ 1 + c(\alpha) & \text{se } \alpha \in E \setminus M, \end{cases}$$

onde M é um emparelhamento em G.

Proposição 3.10 *Seja M um emparelhamento em G e $k \in \Re$. Então,*

$$(kc)(M) = k(c(M)),$$

ou seja, se multiplicamos a função custo por k, os custos dos emparelhamentos em G ficam multiplicados por k. □

Proposição 3.11 (Bertsekas, 1979 e 1986) *Suponha que existe um inteiro k $(k > \frac{n}{2})$ tal que o c-custo de toda aresta de G é múltiplo de k. Seja M um emparelhamento perfeito em G. Se existe uma variável dual y tal que (M, y) é c_M-admissível, então M tem c-custo mínimo.*

Demonstração: Seja N um emparelhamento perfeito em G. Então,

$$c(M) = c_M(M) = y(VM) = y(VN) \leq c_M(N) \leq \frac{n}{2} + c(N).$$

Portanto, dividindo a desigualdade acima por k obtemos

$$\frac{c(M)}{k} \leq \varepsilon + \frac{c(N)}{k},$$

onde $0 \leq \varepsilon < 1$. Mas, pelo enunciado, $c(M)/k$ e $c(N)/k$ são inteiros. Assim,

$$\frac{c(M)}{k} \leq \frac{c(N)}{k}$$

e portanto $c(M) \leq c(N)$. De fato, M tem c-custo mínimo. □

Corolário 3.12 *Seja $c^{(n)}$ uma função custo definida da seguinte forma*

$$c^{(n)}(\alpha) := \left\lceil \frac{n+1}{2} \right\rceil c(\alpha), \forall \alpha \in E.$$

Se existe um par (M, y) $\left(c^{(n)}\right)_M$-admissível, onde M é um emparelhamento perfeito em G, então M é um emparelhamento perfeito de c-custo mínimo em G. □

Seja M um emparelhamento em G e y uma variável dual. Um par (M, y) é *c-semi-admissível* se y é c-independente e $M \subseteq E_y$. Ou seja, (M, y) é c-semi-admissível se e somente se $\forall (u, v) \in E$

$$y(u) + y(v) \leq c(u, v),$$

com igualdade se $(u, v) \in M$. Nesta definição não temos a restrição, feita para par admissível, sobre os vértices M-livres de G (página 62). Assim, se M for um emparelhamento perfeito, então a definição de par c-semi-admissível é equivalente à definição de par c-admissível.

Seja X um conjunto de arestas e Y um conjunto de vértices. Recordamos que $X(Y)$ é o conjunto de todos os vértices adjacentes dos de Y através das arestas de X.

Proposição 3.13 *Seja (M, y) um par c_M-semi-admissível, $\mathcal{P}$ uma coleção maximal disjunta de caminhos M-aumentantes em G_{y,c_M} e $N := M \oplus E\mathcal{P}$. Então, existe z, tal que o par (N, z) é c_N-semi-admissível e N é um emparelhamento máximo em G_{z,c_N}.*

Demonstração: Seja $z : V \to \mathcal{Z}$ uma variável dual definida da seguinte forma:

$$z(v) := \begin{cases} y(v) - 1 & \text{se } v \in V\mathcal{P} \cap V^- \\ y(v) & \text{caso contrário.} \end{cases}$$

A seguir fazemos observações necessárias para a demonstração da proposição. Seja $(u, v) \in E$, onde $u \in V^-$. Então, considere os seguintes casos:

1. $u \in V^- \setminus V\mathcal{P}$. Como $z(u) = y(u)$ e $N(\{u\}) = M(\{u\})$, então

   $$z(u) + z(v) \leq c_N(u, v)$$

   e, além disso, $E_{z,c_N}(\{u\}) = E_{y,c_M}(\{u\})$, ou seja, o conjunto de arestas de E_{z,c_N} que incidem em u é igual ao conjunto de arestas de E_{y,c_M} que incidem em u.

2. $u \in V^- \cap V\mathcal{P}$. Como $z(u) = y(u) - 1$, então

   $$z(u) + z(v) \leq c_N(u, v),$$

   com igualdade se e somente se $(u, v) \in N$ $(N(\{u\}) \neq M(\{u\}))$. Assim, como u é N-ocupado, então $E_{z,c_N}(\{u\}) = N(\{u\})$, ou seja, do conjunto de arestas de E que incidem em u, precisamente uma pertence a E_{z,c_N}, a saber, a aresta que pertence a N.

Das observações 1 e 2, segue que o par (N, z) é c_N-semi-admissível.

Provaremos agora que N é um emparelhamento máximo em G_{z,c_N}. Suponha, por absurdo, que existe um caminho N-aumentante em G_{z,c_N}, digamos, Q. Como os vértices extremos de Q são N-livres, então estes vértices são M-livres e não pertencem a $V\mathcal{P}$. Pelo item 2, os vértices internos de Q em V^- não pertencem a $V\mathcal{P}$, pois a única aresta de E_{z,c_N} que incide num vértice de $V^- \cap V\mathcal{P}$ pertence a N. Portanto,

$$EQ \cap N \subseteq M. \tag{3.13}$$

Logo, os vértices internos de Q em V^+ não pertencem a $V\mathcal{P}$. Assim, pelo item 1 e pela Equação (3.13), Q é c_M-aumentante em G_{y,c_M}. Além disso, $VQ \cap V\mathcal{P} = \emptyset$, o que contradiz a maximalidade de $\mathcal{P}$. □

3.4.2 Descrição

O Algoritmo 3.4 (GabowTarjan) implementa as idéias abordadas na seção anterior. Para tanto, o algoritmo simplesmente multiplica os custos por $\frac{n}{2} + 1$ (note que $\frac{n}{2}$ é um inteiro; caso contrário não existe um emparelhamento perfeito em G), o que permite aplicar o Corolário 3.12. Em seguida o algoritmo invoca EscAprox (Algoritmo 3.5) que obtém, caso exista um emparelhamento perfeito em G, um par $\left(c^{(n)}\right)_M$-admissível, onde M é perfeito (Proposição 3.14 a seguir). Portanto, pelo Corolário 3.12, o emparelhamento M retornado pelo Algoritmo 3.4 é perfeito e possui c-custo mínimo. Se não existe um emparelhamento perfeito, EscAprox detecta este fato.

Na execução do Algoritmo 3.5 (EscAprox) podem ocorrer as seguintes situações:

1. Todas as arestas possuem custo zero. Neste caso utilizamos o Algoritmo 2.5 (Hopcroft e Karp) para determinar um emparelhamento máximo M em G. Se M for um emparelhamento perfeito, retornamos o par $(M, y = 0)$ 0_M-admissível; caso contrário, interrompemos o algoritmo e devolvemos M e $(V^+ \setminus S) \cup T$ como uma certidão da não existência de um emparelhamento perfeito em G (Corolário 2.2).

2. Existem arestas com custo diferente de zero. Chamamos, recursivamente, este algoritmo com o mesmo grafo G, porém com o custo das

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \to \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y \rangle$, onde M é um emparelhamento perfeito em G de custo mínimo e y é $\left(c^{(n)}\right)_M$-independente. Caso contrário, interrompe o algoritmo e devolve $\langle \text{não}, M, (V^+ \setminus S) \cup T \rangle$, onde M e $(V^+ \setminus S) \cup T$ é uma certidão da não existência de um emparelhamento perfeito em G (Corolário 2.2).
A complexidade do algoritmo é $O\left((mn^{1/2} + n^{3/2}) \log nU\right)$. */

GabowTarjan (G, c)
  para todo $\alpha \in E$ **faça** $c^{(n)}(\alpha) \leftarrow \left(\frac{n}{2} + 1\right) c(\alpha)$;
  devolva EscAprox $(G, c^{(n)})$;

Algoritmo 3.4: Escalonamento e aproximação – Gabow e Tarjan.

arestas valendo $\left\lfloor \frac{c(\alpha)}{2} \right\rfloor$ ($\alpha \in E$), ou seja, fazemos o escalonamento. No laço **repita** do algoritmo, primeiro (até o ponto $\lhd 1$) fazemos um ajuste sobre a variável dual y, mantendo o par (M, y) c_M-semi-admissível, para garantir que existe pelo menos um caminho M-aumentante em G_{y,c_M}. O restante deste laço é a implementação do ajuste da variável dual apresentado na Proposição 3.13.

Assim como no Algoritmo 3.3 (Escalonamento – Gabow), usamos comandos entre $\langle\langle$ e $\rangle\rangle$ para facilitar a análise do algoritmo, sendo absolutamente irrelevantes para o algoritmo propriamente dito.

Proposição 3.14 *Se existe um emparelhamento perfeito, então o par (M, y) durante todo o Algoritmo 3.5 é c_M-semi-admissível. Além disso, o emparelhamento retornado é perfeito, ou seja, o par (M, y) retornado é c_M-admissível.*

Demonstração: Prova por indução no número de chamadas do algoritmo. O caso base, quando todas as arestas possuem custo zero, é trivial ($y = 0$). Como hipótese de indução suponha que o par (M', y') retornado pela recursão seja $c'_{M'}$-admissível. Logo $\forall (u, v) \in E$:

$$y(u) + y(v) = 2y'(u) + 2y'(v) - 1 \leq 2c'_{M'}(u, v) - 1 \leq 1 + 2c'(u, v) \leq 1 + c(u, v).$$

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \to \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y\rangle$, onde (M, y) é um par c_M-admissível e M é um emparelhamento perfeito em G. Caso contrário, interrompe o algoritmo e devolve $\langle \text{não}, M, (V^+ \setminus S) \cup T\rangle$, onde M e $(V^+ \setminus S) \cup T$ é uma certidão da não existência de um emparelhamento perfeito em G (Corolário 2.2).
A complexidade do algoritmo é $O\left((mn^{1/2} + n^{3/2}) \log nU\right)$. */

EscAprox (G, c)
  se $\forall \alpha \in E$, $c(\alpha) = 0$ **então início**
    $\langle M, W\rangle \leftarrow$ HopcroftKarp $(G, \emptyset)$;
    se M é um emparelhamento perfeito **então devolva** $\langle \text{sim}, M, y = 0\rangle$
    senão interrompa $\langle \text{não}, M, W\rangle$;
  fim
  senão início
    para todo $\alpha \in E$ **faça** $c'(\alpha) \leftarrow \left\lfloor \frac{c(\alpha)}{2} \right\rfloor$;
    $\langle \exists \text{emp}, M', y'\rangle \leftarrow$ EscAprox (G, c');
    $y_0(v) \underset{(v \in V)}{\leftarrow} 2y'(v) \begin{cases} -1 & \text{se } v \in V^- \\ +0 & \text{caso contrário;} \end{cases}$
    $M \leftarrow \emptyset$; $y \leftarrow y_0$; $\langle\langle l \leftarrow 0; d_+ \leftarrow 0;\rangle\rangle$
    repita
      $\langle \exists \text{cam}, P, d\rangle \leftarrow$ Dijkstra (G, M, c_M, y);
      seja u o último vértice do caminho P; $d_0 \leftarrow d(u)$;
      $$y(v) \underset{(v \in V)}{\leftarrow} \begin{cases} y(v) - (d_0 - d(v)) & \text{se } v \in V^- \text{ e } d(v) \leq d_0 \\ y(v) + (d_0 - d(v)) & \text{se } v \in V^+ \text{ e } d(v) \leq d_0 \\ y(v) & \text{nos demais casos;} \end{cases} \qquad \langle\langle \triangleleft 1\rangle\rangle$$
      $\mathcal{P} \leftarrow$ ColeçãoMaximalCaminhosAumentantes (G_{y,c_M}, M);
      para cada $P \in \mathcal{P}$ **faça** $M \leftarrow M \oplus EP$;
      para cada $v \in V\mathcal{P} \cap V^-$ **faça** $y(v) \leftarrow y(v) - 1$;
      $\langle\langle l \leftarrow l + 1; d_+ \leftarrow d_0 + d_+;\rangle\rangle$ $\qquad \langle\langle \triangleleft 2\rangle\rangle$
    até que M seja perfeito;
    devolva $\langle \text{sim}, M, y\rangle$;
  fim

Algoritmo 3.5: Escalonamento utilizando par semi-admissível.

Portanto, o par $(\emptyset, y)$ no início do laço é $c_\emptyset$-semi-admissível.

Pelas Proposições 3.4 e 3.5 (Algoritmo de Edmonds e Karp – Seção 3.2.1), o ajuste dual do início do laço (até o ponto $\lhd 1$) mantém o par (M, y) c_M-semi-admissível. Pela Proposição 3.13, o restante do laço preserva o par (M, y) c_M-semi-admissível. Como, pelo enunciado, existe um emparelhamento perfeito em G, este laço termina quando M é perfeito, ou seja, o par (M, y) retornado é c_M-admissível. □

3.4.3 Complexidade

A complexidade do Algoritmo 3.4 é dominada pela chamada do Algoritmo 3.5 (EscAprox). Portanto, o Algoritmo 3.4 possui a mesma complexidade do Algoritmo 3.5.

Analisaremos agora a complexidade do Algoritmo 3.5. Como os custos são valores inteiros, então o número de chamadas recursivas deste algoritmo, até executarmos a *base da recursão* (quando todas as arestas possuem custo zero), é $O(\log nU)$. Assim, para obtermos a complexidade afirmada, resta mostrar que a complexidade de cada *patamar* do algoritmo (isto é, a complexidade obtida ignorando-se a chamada recursiva) é

$$O\left(mn^{1/2} + n^{3/2}\right).$$

Note que a base da recursão possui uma complexidade $O\left(mn^{1/2} + n^{3/2}\right)$, pois é basicamente uma chamada do algoritmo de Hopcroft e Karp. Assim, resta mostrar que a complexidade x do laço **repita** satisfaz

$$x = O\left(mn^{1/2} + n^{3/2}\right).$$

A seguir analisaremos a complexidade x num patamar.

Teorema 3.15 *A seguinte afirmação é invariante no laço* **repita**, *no ponto* $\lhd 2$*:*

$$\left|V^+ \setminus VM\right| d_+ \leq 2n.$$

Demonstração: Seja y_0 a variável $c_\emptyset$-independente imediatamente antes do laço e y a variável c_M-independente. Então, para todo vértice $v \in V \setminus VM$,

$$y(v) - y_0(v) = \begin{cases} d_+ & \text{se } v \in V^+ \\ 0 & \text{se } v \in V^- \end{cases}$$

Portanto,

$$y(V \setminus VM) - y_0(V \setminus VM) = \left|V^+ \setminus VM\right| d_+. \tag{3.14}$$

Como y_0 é $c_\emptyset$-independente e $M \subseteq E_{y,c_M}$, então

$$y_0(VM) \leq \frac{n}{2} + c_M(M) = \frac{n}{2} + y(VM).$$

Logo,

$$-\frac{n}{2} \leq y(VM) - y_0(VM). \tag{3.15}$$

Como y é c_M-independente, M' é perfeito e $M' \subseteq E_{y',c'_{M'}}$, então

$$\begin{aligned} y(V) &\leq c_M(M') \leq |M'| + c(M') \leq 2\,|M'| + 2c'(M') \\ &= 2\,|M'| + 2c'_{M'}(M') \overset{\text{H.I.}}{=} 2\,|M'| + 2y'(V) \\ &= 3\,|M'| + y_0(V). \end{aligned}$$

Portanto,

$$\frac{3}{2}n \geq y(V) - y_0(V). \tag{3.16}$$

Assim, se fizermos (3.14) − (3.15) + (3.16), obtemos a desigualdade desejada. □

Lema 3.16 *O laço* **repita** *da função EscAprox é executado no máximo* $1 + 2^{3/2}n^{1/2}$ *vezes num patamar.*

Demonstração: É fácil ver que $l \leq 1 + d_+$, pois a cada incremento de l, a variável d_+ é incrementada de d_0, que por sua vez é sempre positivo, exceto talvez no primeiro incremento que l sofre, quando d_0 pode ser zero. Assim, pelo Teorema 3.15,

$$l \leq 1 + \frac{2n}{|V^+ \setminus VM|}. \tag{3.17}$$

Seja k um valor no intervalo $1 \leq k \leq n$. Tome l_k como sendo o maior valor de l tal que $|V^+ \setminus VM| \geq k$ e l^* como sendo o último (maior) valor de l. É claro que

$$l^* \leq l_k + k \overset{(3.17)}{\leq} 1 + \frac{2n}{k} + k.$$

O valor mínimo de l^* ocorre quando $k = (2n)^{1/2}$. Portanto, temos a desigualdade desejada. □

Portanto, o laço **repita** é executado $O\left(n^{1/2}\right)$ vezes (Lema 3.16), ou seja, para obtermos a complexidade $x = O\left((m+n)\,n^{1/2}\right)$ afirmada, resta mostrar que a complexidade de uma iteração do laço é $O(m+n)$.

Com exceção da chamada do algoritmo de Dijkstra, é fácil verificar que as operações numa iteração deste laço possuem complexidade $O(m+n)$. Fazemos a seguir uma descrição sucinta da versão do algoritmo de Dijkstra que possui complexidade $O(m+n)$.

A cada iteração do laço **repita** do Algoritmo 3.5 temos pelo menos um vértice M-livre em V^+. Portanto, pelo Teorema 3.15, $d_+ \leq 2n$. Como $d_0 \leq d_+$, podemos, ao invés de utilizarmos uma fila de prioridade no algoritmo de Dijkstra, utilizar um vetor R de tamanho $2n$ para determinar $d(v)$ e d_0, onde o elemento R_r aponta para uma lista de vértices com $d(v) = r$ (note que não precisamos armazenar os vértices que possuem $d(v) > 2n$, pois $d_0 \leq d_+$). Utilizamos, também, um vetor W indexado pelos vértices de G (tamanho n), onde W_v ($v \in V$) representa, caso tenha, o rótulo definitivo $d(v)$ dado pelo algoritmo de Dijkstra ao vértice v.

Assim, inicialmente todos os vértices M-livres de V^+ estão numa lista apontada por R_0. O algoritmo consiste em percorrer o vetor R e as listas apontadas por ele numa forma seqüencial e crescente, ou seja, quando estamos no índice x do vetor R, visitamos todos os vértices na lista apontada por R_x antes de irmos para o índice $x+1$. Isto é feito para determinar o próximo vértice que possui um rótulo temporário em W. Logo, este algoritmo possui uma complexidade $O(m+n)$.

Assim, a análise da complexidade está completa.

3.4.4 Observação

No algoritmo originalmente descrito pelos autores Gabow e Tarjan [27], a invariante do Teorema 3.15 obtida por eles foi

$$\left|V^+ \setminus VM\right| d_+ \leq \frac{5}{2}n.$$

Ou seja, na descrição deste algoritmo conseguimos obter uma invariante mais "justa".

3.5 Algoritmo Paralelo com Escalonamento e Aproximação – Goldberg, Plotkin e Vaidya

- Autores: Goldberg, Plotkin e Vaidya [32].
- Data: 1988.
- Tipo: Paralelo.
- Complexidade: $O\left(n^{2/3}\log^3 n \log nU\right)$.
- Processadores: $O\left(n^3/\log n\right)$.
- Modelo: PRAM e CRCW.
- Restrições: integralidade dos custos das arestas.

Este é um algoritmo paralelo que, assim como o algoritmo da Seção 3.4 (Gabow e Tarjan), utiliza escalonamento (nos custos das arestas) e aproximação para determinar um emparelhamento perfeito de custo mínimo; caso não exista tal emparelhamento perfeito, o algoritmo detecta este fato.

Observação: em certas partes do algoritmo utilizamos um processador para cada aresta e/ou para cada par de vértices do grafo. Isto fornece $O\left(m+n^2\right) = O\left(n^2\right)$ processadores utilizados, ou seja, está dentro do limite de processadores do algoritmo.

3.5.1 Algoritmo Básico: Descrição e Complexidade

Seja G um grafo bipartido (V^+ e V^-) e c a sua função custo. O uso da técnica de escalonamento pressupõe integralidade da função custo.

Assim como na Seção 3.3 (Gabow), por simplicidade assumiremos que os valores da função custo, neste algoritmo, são inteiros não negativos, ou seja, $c : E \rightarrow \mathcal{Z}_{\geq 0}$. Note que, se a função custo possui valores negativos, basta subtrair desta função o valor mínimo assumido por ela (Proposição 1.7) que o conjunto solução deste emparelhamento não se altera (Corolário 1.8). O valor de U desta nova função é no máximo duas vezes o valor de U da função original, ou seja, a complexidade fica inalterada.

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \to \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y \rangle$, onde M é um emparelhamento perfeito em G de custo mínimo e y é $\left(c^{(n)}\right)_M$-independente. Caso contrário, interrompe o algoritmo e devolve não. Este algoritmo utiliza $O\left(n^3/\log n\right)$ processadores num tempo $O\left(n^{2/3}\log^3 n \log nU\right)$-CRCW. */

GoldbergPlotkinVaidyaPonderado (G, c)
  para cada $\alpha \in E$ **faça em paralelo**
    $c^{(n)}(\alpha) \leftarrow \left(\frac{n}{2}+1\right) c(\alpha)$;
  devolva EscAproxParalelo $(G, c^{(n)})$;

Algoritmo 3.6: Goldberg, Plotkin e Vaidya – escalonamento e aproximação (paralelo).

Esta transformação, sobre a função custo, pode ser realizada num tempo $O(\log m) = O(\log n)$, utilizando $O(m/\log m) = O(n^2/\log n)$ processadores (EREW) (veja Seção B.1). Ou seja, esta transformação não altera a complexidade do algoritmo.

A seguir descreveremos o algoritmo básico.

Neste algoritmo utilizamos aproximação para determinar um emparelhamento perfeito de custo mínimo. Para tanto, a Função GoldbergPlotkinVaidyaPonderado (Algoritmo 3.6) simplesmente multiplica os custos por $\frac{n}{2}+1$ (note que $\frac{n}{2}$ é um inteiro; caso contrário não existe um emparelhamento perfeito em G), obtendo uma função custo $c^{(n)}$. Esta multiplicação possui um custo $O(1)$, usando $O(m) = O(n^2)$ processadores (CREW). Pelo Corolário 3.12, se esta função (Algoritmo 3.6) determina um par (M, y) $\left(c^{(n)}\right)_M$-admissível, onde M é perfeito, então M possui c-custo mínimo.

Assim, resta mostrar que a Função EscAproxParalelo (Algoritmo 3.7) satisfaz as seguintes propriedades:

(i) Possui uma complexidade $O\left(n^{2/3}\log^3 n \log nU\right)$, usando $O\left(n^3/\log n\right)$ processadores (CRCW).

(ii) Dado um grafo G e uma função c-custo, retorna um par (M, y) c_M-

admissível, onde M é perfeito; se não existe um emparelhamento perfeito, EscAproxParalelo detecta este fato.

Assim como na Função EscAprox (Algoritmo 3.5 – Seção 3.4), utilizamos escalonamento nos custos das arestas na função EscAproxParalelo. Como os custos são valores inteiros, então o número de chamadas recursivas (escalonamentos) na função EscAproxParalelo, até executarmos a *base da recursão* (quando todas as arestas possuem custo zero), é $O(\log nU)$. Logo, resta mostrar que a complexidade de cada *patamar* da função (isto é, a complexidade obtida ignorando-se a chamada recursiva) é

$$O\left(n^{2/3}\log^3 n\right), \tag{3.18}$$

utilizando

$$O\left(n^3/\log n\right) \tag{3.19}$$

processadores e que a Propriedade (ii) é satisfeita.

Na execução da função EscAproxParalelo podem ocorrer as seguintes situações:

1. Todas as arestas possuem custo zero. Este caso é a base da recursão da função, logo é executada apenas uma vez. Ao contrário da Função EscAprox (Algoritmo 3.5), onde usamos o algoritmo de Hopcroft e Karp, utilizamos o Algoritmo 2.9 (Goldberg, Plotkin e Vaidya – Seção 2.4), paralelo, para determinar um emparelhamento máximo M em G. Se M for um emparelhamento perfeito, retornamos o par $(M, y = 0)$ 0_M-admissível; caso contrário, interrompemos o algoritmo. O Algoritmo 2.9 possui complexidade $O\left(n^{2/3}\log^3 n\right)$, usando $O(n^3/\log n)$ processadores (CRCW). Logo, a complexidade da base da recursão é dominada pela complexidade do Algoritmo 2.9 (satisfaz (3.18) e (3.19)).

2. Existem arestas com custo diferente de zero. Chamamos, recursivamente, este algoritmo com o mesmo grafo G, porém com o custo das arestas valendo $c'(\alpha) := \lfloor c(\alpha)/2 \rfloor$ $(\alpha \in E)$, ou seja, fazemos o escalonamento. Da recursão obtemos um par (M', y') $c'_{M'}$-semi-admissível, onde M' é perfeito. Assim como no Algoritmo 3.5 (EscAprox – Seção 3.4), obtemos y_0 a partir de y' da seguinte forma:

$$\underset{(v \in V)}{y_0(v)} \leftarrow 2y'(v) \begin{cases} -1 & \text{se } v \in V^- \\ +0 & \text{caso contrário.} \end{cases}$$

De uma forma análoga à demonstração da Proposição 3.14 (Seção 3.4), temos que o par $(\emptyset, y_0)$ é $c_\emptyset$-semi-admissível. A partir deste par $(\emptyset, y_0)$ $c_\emptyset$-semi-admissível, obtemos um par (M, y) c_M-admissível, onde M é perfeito.

A determinação deste par (M, y) c_M-admissível é feita em duas fases. Analogamente ao algoritmo paralelo de Goldberg, Plotkin e Vaidya (Seção 2.4), aqui também utilizamos dois parâmetros, a e l. Posteriormente verificaremos que os valores ótimos de a e l são, respectivamente, $\lfloor n^{2/3} \rfloor$ e $\lfloor n^{1/3} \rfloor$.

Primeira fase: usamos a Função PréEmparelhamentoPonderado (Algoritmo 3.8 – Seção 3.5.2). Dado um par $(\emptyset, y_0)$ $c_\emptyset$-semi-admissível, esta função retorna um par (M_1, y_1) c_{M_1}-semi-admissível, onde

$$\left| \{ v \in V^+ \setminus VM_1 : y_1(v) - y_0(v) \leq l \} \right| < a. \tag{3.20}$$

Esta função possui uma complexidade

$$O\left(\frac{nl}{a} \log^3 n \right), \tag{3.21}$$

usando

$$O\left(n^2 \right) \tag{3.22}$$

processadores (CRCW).

Segunda fase: usamos a Função EmparelhamentoPerfeitoPonderado (Algoritmo 3.9 – Seção 3.5.3). Dado um par (M_1, y_1) c_{M_1}-semi-admissível satisfazendo (3.20), esta função retorna um par (M, y) c_M-admissível, onde M é perfeito. Esta função possui uma complexidade

$$O\left(\left(a + \frac{n}{l} \right) \log^2 n \right), \tag{3.23}$$

usando

$$O\left(n^3 / \log n \right) \tag{3.24}$$

processadores (CREW).

Assim, a complexidade de um patamar deste caso é (Equações (3.21) e (3.23))

$$O\left(\frac{nl}{a}\log^3 n + a\log^2 n + \frac{n}{l}\log^2 n\right), \tag{3.25}$$

usando (Equações (3.22) e (3.24))

$$O\left(n^2 + n^3/\log n\right) = O\left(n^3/\log n\right)$$

processadores (CRCW).

Como foi mostrado na Seção 2.4, (3.25) assume um valor mínimo quando $a = \left\lfloor n^{2/3} \right\rfloor$ e $l = \left\lfloor n^{1/3} \right\rfloor$. Portanto, a complexidade de um patamar é

$$O\left(n^{2/3}\log^3 n\right).$$

Assim, um patamar deste caso satisfaz as Equações (3.18) e (3.19).

Portanto, a função EscAproxParalelo satisfaz as Propriedades (i) e (ii). O Algoritmo 3.7 apresenta, em nível macro, a função EscAproxParalelo.

As duas seções a seguir completam a descrição do algoritmo. Na Seção 3.5.2 descrevemos a função PréEmparelhamentoPonderado (primeira fase) e na Seção 3.5.3 descrevemos a função EmparelhamentoPerfeitoPonderado (segunda fase).

3.5.2 Algoritmo de Pré-Emparelhamento Ponderado

- Tipo: Paralelo.
- Complexidade: $O\left(\frac{nl}{a}\log^3 n\right)$.
- Processadores: $O\left(n^2\right)$.
- Modelo: PRAM e CRCW.

Dado um par (M_0, y_0) c_{M_0}-semi-admissível e os parâmetros a e l, este algoritmo determina um par (M, y) c_M-semi-admissível, onde

> "o número de vértices M-livres de V^+ que possuem a sua variável independente incrementada de no máximo l é menor do que a." (3.26)

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \to \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y \rangle$, onde M é um emparelhamento perfeito em G de custo mínimo e y é c_M-independente. Caso contrário, interrompe o algoritmo e devolve não. Este algoritmo utiliza $O(n^3/\log n)$ processadores num tempo $O\left(n^{2/3}\log^3 n \log nU\right)$-CRCW. */

EscAproxParalelo (G, c)
se $\forall \alpha \in E$, $c(\alpha) = 0$ **então**
início
$M \leftarrow$ GoldbergPlotkinVaidya (G); /* Algoritmo 2.9 */
se M é um emparelhamento perfeito **então devolva** $(\text{sim}, M, y = 0)$
senão interrompa não;
fim
senão
início
para cada $\alpha \in E$ **faça em paralelo**
$c'(\alpha) \leftarrow \left\lfloor \frac{c(\alpha)}{2} \right\rfloor$;
$(\exists \text{emp}, M', y') \leftarrow$ EscAproxParalelo (G, c');
$y_0(v) \underset{(v \in V)}{\leftarrow} 2y'(v) \begin{cases} -1 & \text{se } v \in V^- \\ +0 & \text{caso contrário;} \end{cases}$
$M_0 \leftarrow \emptyset$;
$a \leftarrow \left\lfloor n^{2/3} \right\rfloor$; $l \leftarrow \left\lfloor n^{1/3} \right\rfloor$;
$\langle M_1, y_1 \rangle \leftarrow$ PréEmparelhamentoPonderado (G, M_0, c, y_0, a, l);
devolva EmparelhamentoPerfeitoPonderado (G, M_1, c, y_1);
fim

Algoritmo 3.7: Escalonamento utilizando par semi-admissível (paralelo).

Ou seja,

$$\left|\{v \in V^+ \setminus VM : y(v) - y_0(v) \leq l\}\right| < a.$$

A seguir apresentamos proposições necessárias para a descrição do algoritmo.

Proposição 3.17 *Seja (M, y) um par c_M-semi-admissível. Então, existe z tais que o par (M, z) é c_M-semi-admissível e todo vértice em $V^+ \setminus VM$ é extremo de pelo menos uma aresta de E_{z,c_M}.*

Demonstração: Basta definir a variável dual $z : V \to \mathcal{Z}$ da seguinte forma:

$$z(v) := \begin{cases} y(v) + \min\{\Delta_{y,c_M}(u,v) : (u,v) \in E\} & \text{se } v \in V^+ \setminus VM \\ y(v) & \text{caso contrário,} \end{cases}$$

onde $\Delta_{y,c_M}(\alpha)$ é a (y, c_M)-folga da aresta α. □

Proposição 3.18 *Seja (M, y) um par c_M-semi-admissível e $(u, v) \in E_{y,c_M} \setminus M$, onde $v \in V^+ \setminus VM$. Então, existe um emparelhamento N, onde $(u, v) \in N$ e $|N| \geq |M|$, e uma variável dual z, tais que o par (N, z) é c_N-semi-admissível e a única aresta de E_{z,c_N} incidindo em u é (u, v).*

Demonstração: Seja o emparelhamento N definido da seguinte forma:

$$N := M \cup \{(u,v)\} \setminus \begin{cases} \{(u,w)\} & \text{se } u \text{ é } M\text{-ocupado, onde } (u,w) \in M \\ \emptyset & \text{caso contrário.} \end{cases}$$

Como v é M-livre e u é extremo de no máximo uma aresta de M, então N é um emparelhamento e $|N| \geq |M|$.

Vamos agora definir a variável dual $z : V \to \mathcal{Z}$ da seguinte forma:

$$\underset{(w \in V)}{z(w)} := \begin{cases} y(w) - 1 & \text{se } w = u \\ y(w) & \text{caso contrário.} \end{cases}$$

É fácil ver que (N, z) é um par c_N-semi-admissível e (u, v) é a única aresta de E_{z,c_N} que incide em u. □

A seguir apresentamos uma descrição do algoritmo.

O Algoritmo 3.8 (PréEmparelhamentoPonderado) implementa as idéias abordadas nas Proposições acima, obtendo um par semi-admissível satisfazendo a Propriedade (3.26), descrita na página 97. A cada iteração deste algoritmo, primeiro é feito um ajuste da variável dual apresentado na Proposição 3.17 para garantir que todos os vértices M-livres em V^+ possuam pelo menos uma aresta justa incidindo neles. A seguir, contamos o número de vértices M-livres de V^+ satisfazendo a Propriedade (3.26). Se esta propriedade for satisfeita, interrompemos o algoritmo; caso contrário, determinamos um emparelhamento maximal H no grafo gerado pelas arestas justas que incidem em vértices M-livres de V^+. A seguir, para cada aresta de H fazemos a mudança no emparelhamento M e na variável dual y apresentada na Proposição 3.18. Como as arestas de H são disjuntas nos vértices, então podemos fazer estas mudanças para todas as arestas de H em paralelo.

Assim, pelas Proposições 3.17 e 3.18, o par (M, y) retornado pelo algoritmo descrito acima é c_M-semi-admissível. Conforme veremos na análise da complexidade, o algoritmo acaba por obter um par (M, y) que satisfaz a Propriedade (3.26) para os valores de a e l fornecidos.

Novamente usamos comandos entre $\langle\langle$ e $\rangle\rangle$ para facilitar a análise do Algoritmo 3.8, sendo absolutamente irrelevantes para o algoritmo propriamente dito.

A seguir apresentamos a complexidade e o número de processadores utilizados. Esta análise será feita utilizando os parâmetros a e l definidos na Propriedade (3.26). As operações, com exceção da chamada da função EmparelhamentoMaximal, numa iteração deste algoritmo, gastam $O(\log n)$ tempo, usando $O(n^2)$ processadores (CREW). A função EmparelhamentoMaximal, apresentada pelos autores Israeli e Shiloach em [37], possui uma complexidade $O\left(\log^3 n\right)$, usando $O(m+n)$ processadores (CRCW). Portanto, a complexidade de uma iteração do Algoritmo 3.8 é $O\left(\log^3 n\right)$, usando $O(n^2)$ processadores (CRCW). Conforme veremos a seguir (Corolário 3.20), executamos no máximo $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ vezes o laço deste algoritmo. Assim, a complexidade deste algoritmo é

$$O\left(\frac{nl}{a}\log^3 n\right),$$

usando

$$O\left(n^2\right)$$

```
/* Dado um grafo G com bipartição (V+,V-), um par (M0, y0) cM0-semi-
admissível e os valores a e l, retorna um novo par (M, y) cM-semi-admissível
que satisfaz a Propriedade (3.26). Este algoritmo utiliza O(n^2) processadores
num tempo O((nl/a) log^3 n)–CRCW. */

PréEmparelhamentoPonderado (G, M0, c, y0, a, l)
  M ← M0; y ← y0; ⟨⟨k ← 1;⟩⟩
  repita
      para cada v ∈ V+ \ VM faça em paralelo
          y(v) ← y(v) + min{Δ_{y,cM}(u, v) : (u, v) ∈ E};
      ⟨⟨y_k ← y; M_k ← M; k ← k + 1;⟩⟩
      /* w estima o número de vértices de V+ satisfazendo
         a Propriedade (3.26) */
      para cada v ∈ V+ faça em paralelo
          se v é M-livre e y(v) − y0(v) ≤ l então x_v ← 1
          senão x_v ← 0;
      w ← Σ_{v∈V+} x_v;
      se w ≥ a então início
          T ← ∅;
          para cada (u, v) ∈ E \ M e v ∈ V+ \ VM faça em paralelo
              se Δ_{y,cM}(u, v) = 0 então
                  acrescente (u, v) a T;
          H ← EmparelhamentoMaximal (G[T]);
          para cada (u, v) ∈ H e u ∈ V- faça em paralelo início
              se u é M-livre então
                  M ← M ∪ {(u, v)}
              senão início
                  seja (u, w) a aresta de M;
                  M ← M ∪ {(u, v)} \ {(u, w)}
              fim
              y(u) ← y(u) − 1;
          fim
      fim
  até que w < a;
  devolva ⟨M, y⟩;
```

Algoritmo 3.8: PréEmparelhamentoPonderado.

processadores (CRCW).

Lema 3.19 *Defina os seguintes conjuntos:*

$$S_k := \{v \in V : |y_k(v) - y_0(v)| \leq l\}$$

e $W := S_k \cap V^+ \setminus VM_k$. *Se* $|W| \geq a$, *então no mínimo* $|W|$ *vértices* v *de* S_k $(k \geq 1)$ *satisfazem* $|y_k(v) - y_0(v)| < |y_{k+1}(v) - y_0(v)|$.

Demonstração: Provaremos primeiro que toda aresta em $E_{y_k, c_{M_k}}$ que incide num vértice de W possui, também, o seu outro extremo em S_k. Seja (v^-, v^+) uma destas arestas, onde $v^+ \in W$. Como $(v^-, v^+) \in E_{y_k, c_{M_k}} \setminus M_k$ e y_0 é c_{M_0}-independente, então

$$y_k(v^-) + y_k(v^+) = 1 + c(v^-, v^+) \geq y_0(v^-) + y_0(v^+).$$

Logo,

$$l \geq y_k(v^+) - y_0(v^+) \geq y_0(v^-) - y_k(v^-) = \left|y_k(v^-) - y_0(v^-)\right|,$$

pois só aumentamos (diminuímos) a variável independente dos vértices de V^+ (V^-, respectivamente). Portanto, $v^- \in S$.

Vamos agora concluir a demonstração do Lema.

Os vértices em $H(W)$, uma parte de V^-, pertencem todos a S_k, como foi visto acima. Assim, as suas variáveis duais são decrementadas ao final da iteração. Por outro lado, os vértices de $W \setminus VH$, uma parte de V^+, não incidem em arestas de $E_{y_k, c_{M_{k+1}}}$, pela maximalidade de H. Assim, suas variáveis duais são incrementadas ao início da próxima iteração. De fato, pelo menos $|W|$ vértices de S_k têm sua variável dual mais afastada do valor original. □

Corolário 3.20 *O laço da função PréEmparelhamentoPonderado (Algoritmo 3.8) é executado no máximo* $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ *vezes.*

Demonstração: Como os valores de $|y - y_0|$ nunca diminuem, então no máximo $l + 1$ vezes um vértice $v \in V$, com $|y(v) - y_0(v)| \leq l$, teve a diferença $|y(v) - y_0(v)|$ aumentada. Como temos n vértices, então no máximo $n(l + 1)$ vezes aumentamos o valor de $|y(v) - y_0(v)|$ para vértices v tais que

$|y(v) - y_0(v)| \leq l$. Pelo Lema 3.19, no mínimo a vértices com $|y(v) - y_0(v)| \leq l$ têm a diferença aumentada, por iteração da função. Como precisamos de mais uma iteração para calcular o número de vértices satisfazendo a Propriedade (3.26), então o número de iterações deste laço para satisfazer a Propriedade (3.26) é no máximo $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$. □

3.5.3 Algoritmo para Emparelhamento Perfeito Ponderado

- Tipo: Paralelo.
- Complexidade: $O\left(\left(a + \frac{n}{l}\right) \log^2 n\right)$.

  Os parâmetros a e l foram definidos para o Algoritmo 3.8 (Seção 3.5.2).
- Processadores: $O(n^3/\log n)$.
- Modelo: PRAM e CREW.

Dado um par (M_1, y_1) c_{M_1}-semi-admissível, este algoritmo determina um par (M, y) c_M-admissível, onde M é um emparelhamento perfeito em G.

A seguir descrevemos o algoritmo.

O Algoritmo 3.9 (EmparelhamentoPerfeitoPonderado) funciona basicamente como uma versão em paralelo do laço **repita** do Algoritmo 3.5 (EscAprox). Porém, ao invés de determinarmos a cada iteração do laço uma coleção maximal de caminhos M-aumentantes em G_{y,c_M}, determinamos apenas um caminho M-aumentante em G_{y,c_M}, utilizando o Algoritmo 2.6 (EncontraCaminho – Kim e Chwa). O algoritmo DijkstraParalelo determina d para os vértices de G utilizando multiplicação de matrizes de distância (uma versão deste algoritmo pode ser encontrada em [3, pag 257]). Como este algoritmo é basicamente uma versão em paralelo do laço do Algoritmo 3.5 (EscAprox), então este algoritmo também retorna um par (M, y) c_M-admissível, onde M é perfeito (veja a Proposição 3.14 – Seção 3.4).

A seguir apresentamos a complexidade e o número de processadores utilizados pelo algoritmo. Esta análise será feita utilizando os parâmetros a e l definidos para o Algoritmo 3.8 (Seção 3.5.2 – Propriedade (3.26)). Em [3, pag

/* Dado um grafo G com bipartição (V^+,V^-), onde existe um emparelhamento perfeito, e um par (M_1, y_1) c_{M_1}-semi-admissível, retorna um novo par (M, y) c_M-semi-admissível, onde M é um emparelhamento perfeito em G. Este algoritmo utiliza $O(n^3/\log n)$ processadores num tempo $O\left((|M| - |M_1|)\log^2 n\right)$-CREW. */

EmparelhamentoPerfeitoPonderado (G, M_1, c, y_1)
  $M \leftarrow M_1$;
  $y \leftarrow y_1$;
  enquanto M não é um emparelhamento perfeito **faça**
  início
    $\langle \exists \text{cam}, d \rangle \leftarrow$ DijkstraParalelo (G, M, c_M, y);
    $d_0 \leftarrow \min\{d(u) : u \in V^- \setminus VM\}$;

$$y(v) \underset{(v \in V)}{\leftarrow} \begin{cases} y(v) - (d_0 - d(v)) & \text{se } v \in V^- \text{ e } d(v) \leq d_0 \\ y(v) + (d_0 - d(v)) & \text{se } v \in V^+ \text{ e } d(v) \leq d_0 \\ y(v) & \text{nos demais casos;} \end{cases}$$

    $\langle \exists \text{cam}, P \rangle \leftarrow$ EncontraCaminho (G_{y,c_M}, M); /* Algoritmo 2.6 */
    $M \leftarrow M \oplus EP$;
    para cada $v \in VP \cap V^-$ **faça em paralelo**
      $y(v) \leftarrow y(v) - 1$;
  fim
  devolva (sim, M, y);

Algoritmo 3.9: Emparelhamento perfeito (paralelo).

257] temos que a complexidade de uma chamada do algoritmo DijkstraParalelo é $O\left(\log^2 n\right)$, usando $O\left(n^3/\log n\right)$ processadores[3] (CREW). A função EncontraCaminho (Algoritmo 2.6) possui uma complexidade $O\left(\log d \log n\right) = O\left(\log^2 n\right)$, onde d é o comprimento do caminho aumentante determinado, usando $O\left(n^3/\log n\right)$ processadores (CREW). Assim, a complexidade deste algoritmo é $O\left((|M| - |M_1|)\log^2 n\right)$, onde M é o emparelhamento perfeito retornado e M_1 é o emparelhamento com o qual invocamos o algoritmo, usando $O\left(n^3/\log n\right)$ processadores (CREW). Conforme veremos a seguir (Corolário 3.22), temos que $|M| - |M_1| \leq a + \left\lfloor \frac{2n}{l} \right\rfloor$, ou seja, a complexidade do algoritmo é

$$O\left(\left(a + \frac{n}{l}\right)\log^2 n\right),$$

usando $O\left(n^3/\log n\right)$ processadores (CREW).

Lema 3.21 *A seguinte desigualdade é válida na função EscAproxParalelo (Algoritmo 3.7)*

$$y_1(V^+ \setminus VM_1) - y_0(V^+ \setminus VM_1) \leq 2n.$$

Demonstração: Para todo vértice de $v \in V^- \setminus VM_1$, $y_1(v) = y_0(v)$, ou seja, não modificamos o valor de y dos vértices M-livres em V^- na função PréEmparelhamentoPonderado (Algoritmo 3.8). Portanto,

$$y_1(V \setminus VM_1) - y_0(V \setminus VM_1) = y_1(V^+ \setminus VM_1) - y_0(V^+ \setminus VM_1). \quad (3.27)$$

Como y_0 é $c_\emptyset$-independente e $M_1 \subseteq E_{y_1, c_{M_1}}$, então

$$y_0(VM_1) \leq \frac{n}{2} + c_{M_1}(M_1) = \frac{n}{2} + y_1(VM_1).$$

Logo,

$$-\frac{n}{2} \leq y_1(VM_1) - y_0(VM_1). \quad (3.28)$$

[3]Na hora em que multiplicamos a linha da matriz por uma coluna, podemos utilizar um truque semelhante ao apresentado na Seção B.1 para utilizar $O\left(n/\log n\right)$ ao invés de $O\left(n\right)$ processadores nessa multiplicação. Assim, podemos reduzir de $O\left(n^3\right)$ para $O\left(n^3/\log n\right)$ o número de processadores deste algoritmo.

Como y_1 é c_{M_1}-independente, M' é perfeito e $M' \subseteq E_{y',c'_{M'}}$, então

$$\begin{aligned} y_1(V) &\leq c_{M_1}(M') \leq |M'| + c(M') \leq 2|M'| + 2c'(M') \\ &= 2|M'| + 2c'_{M'}(M') \overset{\text{H.I.}}{=} 2|M'| + 2y'(V) \\ &= 3|M'| + y_0(V). \end{aligned}$$

Portanto,

$$\frac{3}{2}n \geq y_1(V) - y_0(V). \tag{3.29}$$

Assim, se fizermos (3.27) − (3.28) + (3.29), obtemos a desigualdade desejada. □

Corolário 3.22 *A seguinte desigualdade é válida na Função EscAproxParalelo (Algoritmo 3.7)*

$$\left|V^+ \setminus VM_1\right| \leq a + \left\lfloor \frac{2n}{l} \right\rfloor.$$

Demonstração: Como só aumentamos a variável independente dos vértices de V^+ na função PréEmparelhamentoPonderado (Algoritmo 3.8), então $\forall v^+ \in V^+ \setminus VM_1$,

$$y_1(v^+) - y_0(v^+) \geq 0.$$

Pela Propriedade (3.26), no máximo a vértices de $V^+ \setminus VM_1$ possuem $y_1(v^+) - y_0(v^+) \leq l$. Por outro lado, no máximo $\left\lfloor \frac{2n}{l} \right\rfloor$ vértices de $V^+ \setminus VM_1$ possuem $y_1(v^+) - y_0(v^+) > l$, pois pelo Lema 3.21,

$$y_1(V^+ \setminus VM_1) - y_0(V^+ \setminus VM_1) \leq 2n.$$

Portanto,

$$\left|V^+ \setminus VM_1\right| \leq a + \left\lfloor \frac{2n}{l} \right\rfloor. \quad \square$$

Na Figura 3.5 é apresentado um esquema das chamadas de funções do Algoritmo 3.6 (GoldbergPlotkinVaidyaPonderado).

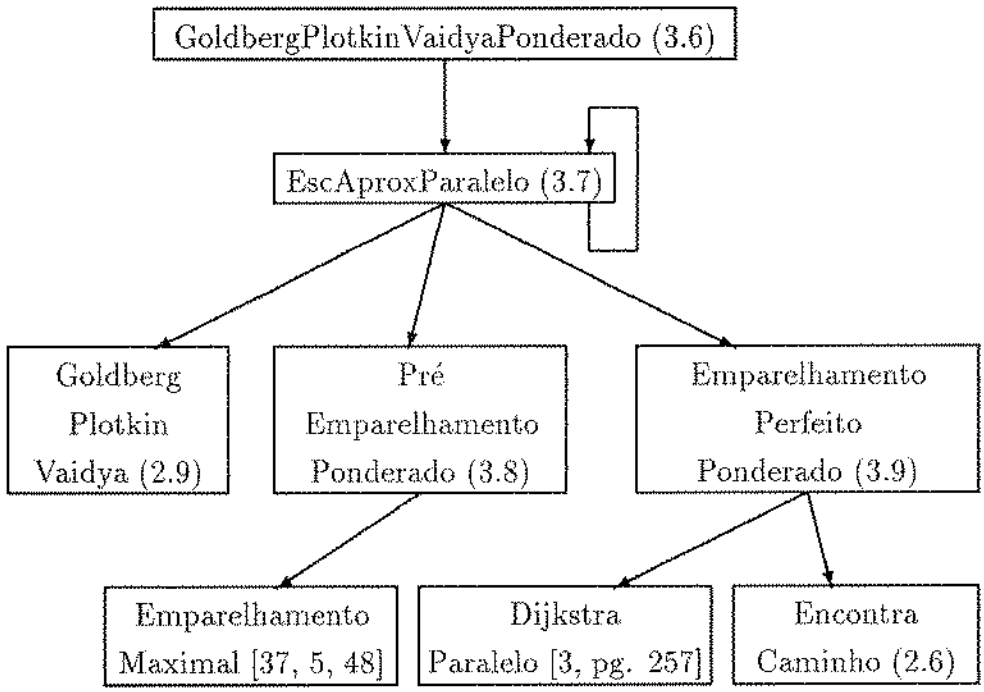


Figura 3.5: Esquema das chamadas das funções do algoritmo de Goldberg, Plotkin e Vaidya (ponderado).

3.5.4 Observação

O laço da Função PréEmparelhamentoPonderado (Algoritmo 3.8) descrito pelos autores Goldberg, Plotkin e Vaidya [32], é executado no máximo $l + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ vezes. A nossa versão da função PréEmparelhamentoPonderado é executada no máximo $1 + \left\lfloor \frac{n(l+1)}{a} \right\rfloor$ vezes (Corolário 3.20).

Note, porém, que os valores de a e l são, respectivamente, $\left\lfloor n^{2/3} \right\rfloor$ e $\left\lfloor n^{1/3} \right\rfloor$. Portanto, a complexidade final do algoritmo não se altera.

3.6 Algoritmo Paralelo com Escalonamento e Aproximação – Gabow e Tarjan

- Autores: Gabow e Tarjan [26].
- Data: 1988.
- Tipo: Paralelo.
- Complexidade: $O\left(\left(mn^{1/2}/p\right)\log p \log nU\right)$.
- Processadores: p, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$.
- Modelo: PRAM e EREW.
- Restrições: integralidade dos custos das arestas.

Este algoritmo é praticamente uma versão em paralelo do algoritmo da Seção 3.4 (Gabow e Tarjan). Assim como o algoritmo da Seção 3.4, este algoritmo utiliza escalonamento (nos custos das arestas) e aproximação para determinar um emparelhamento perfeito de custo mínimo; caso não exista tal emparelhamento perfeito, o algoritmo detecta este fato.

Nota: na análise de complexidade, sempre que o número p de processadores usados e/ou o modelo EREW estiver subentendido, será omitido.

3.6.1 Fundamentos

Seja G um grafo bipartido (V^+ e V^-) e c a sua função custo. O uso da técnica de escalonamento pressupõe integralidade da função custo.

Assim como na Seção 3.4 (Gabow e Tarjan), por simplicidade assumiremos que os valores da função custo, neste algoritmo, são inteiros não negativos, ou seja, $c : E \rightarrow \mathcal{Z}_{\geq 0}$. Note que, se a função custo possui valores negativos, basta subtrair desta função o valor mínimo assumido por ela (Proposição 1.7) que o conjunto solução deste emparelhamento não se altera (Corolário 1.8). O valor de U desta nova função é no máximo duas vezes o valor de U da função original, ou seja, a complexidade fica inalterada.

Esta transformação, sobre a função custo, pode ser realizada num tempo $O(m/p + \log p)$, dominada pela complexidade do restante do algoritmo.

De uma forma análoga à Seção 3.4, definimos a função custo $c_M : E \to \mathcal{Z}_{\geq 0}$ da seguinte forma

$$c_M(\alpha) := \begin{cases} c(\alpha) & \text{se } \alpha \in M \\ 1 + c(\alpha) & \text{se } \alpha \in E \setminus M, \end{cases}$$

onde M é um emparelhamento em G.

Uma quádrupla (M, y, c, r) é *ótima* se as seguintes propriedades forem satisfeitas:

(i) M é um emparelhamento em G.

(ii) y é c_M-independente, ou seja, $\forall (u, v) \in E$, $y(u) + y(v) \leq c_M(u, v)$.

(iii) $c(M) \leq r + y(VM)$.

(iv) r é um inteiro não negativo.

Note que se uma quádrupla $(M, y, c, 0)$ é ótima, então o par (M, y) é c_M-semi-admissível.

Seja (M, y, c, r) uma quádrupla ótima. Definimos por *conjunto de arestas M-justas* ao conjunto de arestas (u, v) de G que pertencem a M e/ou possuem igualdade em $y(u) + y(v) = c_M(u, v)$, ou seja,

$$E_{M,y,c_M} := M \cup E_{y,c_M}.$$

Assim, se a quádrupla $(M, y, c, 0)$ é ótima, então $E_{M,y,c_M} = E_{y,c_M}$.

As Proposições 3.23 e 3.25 são generalizações, respectivamente, das Proposições 3.11 e 3.13 (Seção 3.4.1).

Proposição 3.23 *Suponha que existem inteiros k e r não negativos ($k > \frac{n}{2} + r$) tal que o c-custo de toda aresta de G é múltiplo de k. Seja M um emparelhamento perfeito em G. Se existe uma variável dual y tal que a quádrupla (M, y, c, r) é ótima, então M tem c-custo mínimo.* □

Corolário 3.24 *Seja $c^{(n)}$ uma função custo definida da seguinte forma*

$$c^{(n)}(\alpha) := \left(\left\lceil \frac{n+1}{2} \right\rceil + n \right) c(\alpha), \ \forall \alpha \in E.$$

Se existe uma quádrupla $(M, y, c^{(n)}, r)$ ótima, onde M é um emparelhamento perfeito em G e $r \leq n$, então M é um emparelhamento perfeito de c-custo mínimo em G. □

Proposição 3.25 *Seja (M, y, c, r) uma quádrupla ótima, $\mathcal{P}$ uma coleção maximal disjunta de caminhos M-aumentantes em G_{M,y,c_M} e $N := M \oplus E\mathcal{P}$. Então, existe z, tais que a quádrupla (N, z, c, r) é ótima e N é um emparelhamento máximo em G_{N,z,c_N}.*

Demonstração: Seja a variável dual $z : V \to \mathcal{Z}$ definida da seguinte forma

$$z(v) := \begin{cases} y(v) - 1 & \text{se } v \in V\mathcal{P} \cap V^- \\ y(v) & \text{caso contrário.} \end{cases}$$

O restante da demonstração é semelhante à da Proposição 3.13. □

3.6.2 Algoritmo Básico: Descrição e Complexidade

Este algoritmo é praticamente uma versão em paralelo do algoritmo da Seção 3.4 (Gabow e Tarjan).

Este algoritmo utiliza aproximação. Para tanto, a Função GabowTarjan-Paralelo (Algoritmo 3.10) simplesmente multiplica os custos por $1 + \frac{3n}{2}$ (note que $\frac{n}{2}$ é um inteiro; caso contrário não existe um emparelhamento perfeito em G), obtendo uma função custo $c^{(n)}$. Esta multiplicação possui uma complexidade $O((m/p) + \log p)$ (dentro do limite do algoritmo). Pelo Corolário 3.24, se esta função (Algoritmo 3.10) determina uma quádrupla $(M, y, c^{(n)}, r)$ ótima, onde M é perfeito e $r \leq n$, então M possui c-custo mínimo.

Assim, resta mostrar que a Função EscEmparelhamentoParalelo (Algoritmo 3.11) satisfaz as seguintes propriedades:

(i) Dado um grafo G e uma função c-custo, esta função retorna uma quádrupla (M, y, c, r) ótima, onde M é perfeito e $r \leq n$; se não existe um emparelhamento perfeito, EscEmparelhamento detecta este fato.

(ii) Possui uma complexidade $O\left(\left(mn^{1/2}/p\right) \log p \log nU\right)$.

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \rightarrow \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y, r \rangle$, onde a quádrupla (M, y, c, r) é ótima, M é um emparelhamento perfeito em G de custo mínimo e $r \leq n$. Caso contrário, interrompe o algoritmo e devolve não.
A complexidade do algoritmo é $O\left(\left(mn^{1/2}/p\right) \log p \log nU\right)$, usando p processadores, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$ - EREW. */

GabowTarjanParalelo (G, c)
para todo $\alpha \in E$ **faça em paralelo** $c^{(n)}(\alpha) \leftarrow \left(1 + \frac{3n}{2}\right) c(\alpha)$;
devolva EscEmparelhamentoParalelo $(G, c^{(n)})$;

Algoritmo 3.10: Escalonamento e aproximação - Gabow e Tarjan (paralelo).

Assim como na Função EscAprox (Algoritmo 3.5 – Seção 3.4), utilizamos escalonamento nos custos das arestas na função EscEmparelhamentoParalelo. Como os custos são valores inteiros, então o número de chamadas recursivas (escalonamentos) na função EscEmparelhamentoParalelo, até executarmos a *base da recursão* (quando todas as arestas possuem custo zero), é $O(\log nU)$. Logo, resta mostrar que a complexidade de cada *patamar* da função (isto é, a complexidade obtida ignorando-se a chamada recursiva) é

$$O\left(\left(mn^{1/2}/p\right) \log p\right) \tag{3.30}$$

e que a Propriedade (i) é satisfeita.

Na execução da função EscEmparelhamentoParalelo podem ocorrer as seguintes situações:

1. Todas as arestas possuem custo zero. Então tomamos a quádrupla $(\emptyset, 0, 0, 0)$ que é ótima. Este caso é a base da recursão da função e é executado apenas uma vez.

2. Existem arestas com custo diferente de zero. Chamamos, recursivamente, este algoritmo com o mesmo grafo G, porém com o custo das arestas valendo $c' := \lfloor c/2 \rfloor$, ou seja, fazemos o escalonamento. Da

recursão obtemos uma quádrupla (M', y', c', r') ótima, onde M' é perfeito e $r' \leq n$. Assim como no Algoritmo 3.5 (EscAprox – Seção 3.4), obtemos y_0 a partir de y' da seguinte forma:

$$\underset{(v \in V)}{y_0(v)} \leftarrow 2y'(v) \begin{cases} -1 & \text{se } v \in V^- \\ +0 & \text{caso contrário.} \end{cases}$$

De uma forma análoga à demonstração da Proposição 3.14 (Seção 3.4), temos que a quádrupla $(\emptyset, y_0, c, 0)$ é ótima.

A partir desta quádrupla $(\emptyset, y_0, c, 0)$ ótima (item 1 ou 2 acima, conforme o caso), determinamos uma quádrupla (M, y, c, r) ótima, onde M é um emparelhamento máximo em G e $r \leq n$. Se M não for perfeito, interrompemos o algoritmo. O teste para verificar se o emparelhamento é perfeito só precisa ser realizado na base da recursão, ou seja, quando $c = 0$ (item 1).

Note que:

1. Verificar se o custo de todas as arestas é zero pode ser realizado numa complexidade $O((m/p) + \log p)$.

2. Verificar se o emparelhamento M é perfeito pode ser realizado numa complexidade $O((n/p) + \log p)$.

3. A obtenção de y_0 pode ser realizada numa complexidade $O(n/p)$.

Assim, estas operações estão dentro dos limites de complexidade de um patamar (Equação (3.30)).

Como Obter a Quádrupla (M, y, c, r): Descrição

Na obtenção desta quádrupla (M, y, c, r) ótima utilizamos um grafo denominado *M-simples*, definido da seguinte forma:

$$H(G, M) := G[X],$$

onde X é o conjunto de vértices que pertencem a algum caminho M-aumentante de G. Isto é, o grafo $H(G, M)$ é o grafo gerado a partir de G, usando os vértices que pertencem a algum caminho M-aumentante de G. Por simplicidade, quando o grafo G e o emparelhamento M estiverem subentendidos, usaremos simplesmente H para representar o grafo M-simples $H(G, M)$.

Proposição 3.26 *Um caminho P é M-aumentante em G se e somente se P é um caminho $M \cap EH$-aumentante em H.*

Demonstração: $\Longrightarrow$ Como P é um caminho M-aumentante em G, então, pela definição de H, todos os vértices de P estão em H, ou seja, $VP \subseteq VH$. Assim, P é um caminho $M \cap EH$-aumentante em H.

$\Longleftarrow$ Seja P um caminho $M \cap EH$-aumentante em H. Como H é um subgrafo de G, então basta provar que os vértices extremos de P são M-livres em G. Suponha, por contradição, que um dos extremos de P, digamos o vértice u, é M-ocupado em G. Seja (u, v) a aresta de M que incide em u no grafo G. Como, pela definição de H, todos os vértices que pertencem a algum caminho M-aumentante de G estão em H e $u \in VH$, então v tem que pertencer a H. Portanto, a aresta $(u, v) \in EH$ e, conseqüentemente, u é $M \cap EH$-ocupado em H, contradição. □

Corolário 3.27 *Uma coleção disjunta de caminhos M-aumentantes $\mathcal{P}$ é maximal em G se e somente se $\mathcal{P}$ é uma coleção maximal disjunta de caminhos $M \cap EH$-aumentantes em H.* □

A determinação desta quádrupla (M, y, c, r) ótima é feita dentro de um laço **repita**. Este laço é executado até que M seja máximo (ou seja, não existam mais caminhos M-aumentantes). Cada iteração deste laço se divide em três partes:

Primeira parte: é feito um ajuste na variável dual y e em r, de forma que:

1. Os valores de y para cada vértice de $V^+ \setminus VM$ são acrescidos de um mesmo valor, d_0.
2. Os valores de y para os vértices de $V^- \setminus VM$ permanecem inalterados.
3. O valor de r nunca diminui, mas $r \leq n$.
4. (M, y, c, r) permanece ótima.

Os ajustes descritos acima são efetuados até que um dos seguintes casos ocorra:

- Existe um caminho M-aumentante em G_{M,y,c_M}.
- M é máximo em G.

Além disso, também são obtidos:

- O valor d_0, que é o acréscimo que a variável dual y sofreu em cada vértice M-livre de V^+.
- O grafo $H(G_{M,y,c_M}, M)$ M-simples.

Este ajuste é efetuado pela Função AjusteDualRelaxado (Algoritmo 3.13 – Seção 3.6.4), que recebe como parâmetros G, M, y, c e r.

Convém salientar que o valor de r só é alterado nessa função. A quantidade com que aumentamos r, numa chamada dessa função, depende do valor que uma constante g ($g \in \Re_{>0}$) assume nas chamadas dessa função. Como veremos a seguir (Proposição 3.32), g deve assumir no mínimo $10\left(1 + \log \frac{n}{2}\right)$ para que r seja menor do que ou igual a n.

Segunda parte: caso exista um caminho M-aumentante em G_{M,y,c_M}, ou seja, quando $VH(G_{M,y,c_M}, M) \neq \emptyset$ (Proposição 3.26), uma coleção maximal disjunta $\mathcal{P}$ de tais caminhos é obtida para o grafo G_{M,y,c_M}. Pelo Corolário 3.27, $\mathcal{P}$ é uma coleção maximal disjunta em G_{M,y,c_M} se e somente se $\mathcal{P}$ é uma coleção maximal disjunta em $H(G_{M,y,c_M}, M)$. Assim, para determinar $\mathcal{P}$, utilizamos a Função ColeçãoMaximalParalelo (Algoritmo 3.14 – Seção 3.6.5), passando como parâmetros o grafo bipartido $H(G_{M,y,c_M}, M)$ (obtido na primeira parte) e o emparelhamento $M \cap EH(G_{M,y,c_M}, M)$, onde $H(G_{M,y,c_M}, M)$ não possui ciclos $M \cap EH(G_{M,y,c_M}, M)$-alternados, pois como veremos na Proposição 3.28, G_{M,y,c_M} não possui ciclos M-alternados.

Terceira parte: fazemos $M \leftarrow M \oplus E\mathcal{P}$ e aplicamos o ajuste sobre a variável dual y apresentado na Proposição 3.25. Neste ponto, a quádrupla (M, y, c, r) é ótima, M é máximo em G_{M,y,c_M} e $r \leq n$.

Logo, a função EscEmparelhamentoParalelo satisfaz a Propriedade (i). O Algoritmo 3.11 apresenta, em nível macro, a função EscEmparelhamentoParalelo. Novamente, usamos comandos entre $\langle\langle$ e $\rangle\rangle$ para facilitar a análise do algoritmo, sendo absolutamente irrelevantes para o algoritmo propriamente dito.

/* Dado um grafo G com bipartição (V^+,V^-) e uma função custo $c : E \rightarrow \mathcal{Z}_{\geq 0}$ sobre as suas arestas, retorna, caso exista um emparelhamento perfeito em G, $\langle \text{sim}, M, y, r\rangle$, onde a quádrupla (M, y, c, r) é ótima, M é um emparelhamento perfeito em G de custo mínimo e $r \leq n$. Caso contrário, interrompe o algoritmo e devolve não.
A complexidade do algoritmo é $O\left(\left(mn^{1/2}/p\right)\log p \log nU\right)$, usando p processadores, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$ - EREW. */

EscEmparelhamentoParalelo (G, c)
se $\forall \alpha \in E$, $c(\alpha) = 0$ **então** $y_0 \leftarrow 0$
senão início
para cada $\alpha \in E$ **faça em paralelo** $c'(\alpha) \leftarrow \left\lfloor \frac{c(\alpha)}{2} \right\rfloor$;
$\langle \exists \text{emp}, M', y', r' \rangle \leftarrow$ EscEmparelhamentoParalelo (G, c');
$y_0(v) \underset{(v \in V)}{\leftarrow} 2y'(v) \begin{cases} -1 & \text{se } v \in V^- \\ +0 & \text{caso contrário;} \end{cases}$
fim
$M \leftarrow \emptyset$; $y \leftarrow y_0$; $r \leftarrow 0$; $l \leftarrow 0$; $\langle\langle M_0 \leftarrow \emptyset; r_0 \leftarrow 0; d_+^{(0)} \leftarrow 0; \rangle\rangle$
repita
$l \leftarrow l + 1$;
$\langle y, r, d_0^{(l)}, H\rangle \leftarrow$ AjusteDualRelaxado (G, M, y, c, r);
$\langle\langle d_+^{(l)} \leftarrow d_0^{(l)} + d_+^{(l-1)}; f_l \leftarrow |V^+ \setminus VM|; y_l \leftarrow y; \rangle\rangle$
se $VH \neq \emptyset$ **então início**
$\mathcal{P}_l \leftarrow$ ColeçãoMaximalParalelo $(H, M \cap EH)$; $\langle\langle \lhd 1 \rangle\rangle$
para cada $\alpha \in E\mathcal{P}_l$ **faça em paralelo** $M \leftarrow M \oplus \{\alpha\}$;
para cada $v \in V\mathcal{P}_l \cap V^-$ **faça em paralelo** $y(v) \leftarrow y(v) - 1$;
fim
$\langle\langle M_l \leftarrow M; \rangle\rangle$ $\langle\langle \lhd 2 \rangle\rangle$
até que $|V^+ \setminus VM| = \emptyset$ **ou** $VH = \emptyset$;
se M é perfeito **então devolva** $\langle \text{sim}, M, y, r\rangle$
senão interrompa não;

Algoritmo 3.11: Escalonamento utilizando quádrupla ótima.

Completamos agora a descrição apresentando alguns resultados.

Proposição 3.28 *Durante a execução da função EscEmparelhamentoParalelo, o grafo G_{M,y,c_M} não possui ciclos M-alternados.*

Demonstração: Prova por indução no número de iterações da função. O caso base é trivial, pois $M_0 = \emptyset$. Como hipótese de indução suponha que o grafo não possui ciclos M-alternados no início de uma iteração.

Como veremos na Seção 3.6.4 (Proposição 3.38), a função AjusteDualRelaxado não cria ciclos M-alternados. Por outro lado, a diferença simétrica não cria ciclos M-alternados, pois diminuímos de 1 o valor da variável dual nos vértices de $V\mathcal{P} \cap V^-$. Portanto, as únicas arestas M-justas incidindo nesses vértices são as de M.

Assim, temos que o grafo não possui tais ciclos. □

Pelo ajuste realizado na variável dual y e o valor de d_0 retornado pela função AjusteDualRelaxado, temos a seguinte proposição:

Proposição 3.29 *A seguinte afirmação é verdadeira na l-ésima iteração do laço* **repita**, *nos pontos $\lhd 1$ e $\lhd 2$: para todo vértice $v \in V \setminus VM$,*

$$y(v) - y_0(v) = \begin{cases} d_+^{(l)} & \text{se } v \in V^+ \\ 0 & \text{se } v \in V^-. \end{cases} \square$$

Teorema 3.30 *A seguinte afirmação é verdadeira na l-ésima iteração do laço* **repita**, *nos pontos $\lhd 1$ e $\lhd 2$:*

$$\left|V^+ \setminus VM\right| d_+^{(l)} \leq r + 4n.$$

Demonstração: Seja y_0 a variável $c_\emptyset$-independente imediatamente antes do laço e y a variável c_M-independente. Pela Proposição 3.29, nos pontos $\lhd 1$ e $\lhd 2$ temos que:

$$\left|V^+ \setminus VM\right| d_+^{(l)} = y(V \setminus VM) - y_0(V \setminus VM). \tag{3.31}$$

Como y_0 é $c_\emptyset$-independente e (M, y, c, r) é ótima, então

$$y_0(VM) \leq |M| + c_M(M) \leq |M| + r + y(VM).$$

Logo,

$$-\left(y(VM)-y_0(VM)\right) \leq r+\frac{n}{2}. \tag{3.32}$$

Como y é c_M-independente e M' é perfeito, então

$$y(V) \leq c_M(M') \leq |M'| + c(M') \leq 2\,|M'| + 2c'(M'). \tag{3.33}$$

Ademais, por indução, a quádrupla (M', y', c', r') é ótima e $r' \leq n$, portanto

$$c'(M') \overset{\text{H.I.}}{\leq} r' + y'(V) \leq n + y'(V). \tag{3.34}$$

Finalmente, por definição de y_0,

$$2y'(V) = y_0(V) + |M'|\,. \tag{3.35}$$

Multiplicando ambos os termos da desigualdade (3.34) por 2 e somando-a a (3.33) e (3.35), obtemos

$$y(V) - y_0(V) \leq \frac{7}{2}n. \tag{3.36}$$

Assim, somando (3.31), (3.32) e (3.36), obtemos a desigualdade desejada. □

Seja L o número de iterações do laço **repita** da função EscEmparelhamentoParalelo num patamar. Então, temos a seguinte proposição:

Proposição 3.31 *Num patamar da função EscEmparelhamentoParalelo, temos que*

$$\sum_{l=1}^{L} d_+^{(l)}\,|\mathcal{P}_l| \leq (r+4n)\left(1+\log\frac{n}{2}\right).$$

Demonstração: Sempre que executamos uma iteração da função, no ponto $\lhd 1$ temos pelo menos um vértice M-livre. Assim, pelo Teorema 3.30, na l-ésima iteração, no ponto $\lhd 1$, temos que:

$$d_+^{(l)} \leq \frac{r+4n}{|V^+ \setminus VM_{l-1}|}.$$

Portanto,

$$d_+^{(l)}\,|\mathcal{P}_l| \leq (r+4n)\sum_{i=0}^{|\mathcal{P}_l|-1}\frac{1}{|V^+ \setminus VM_{l-1}|-i}. \tag{3.37}$$

Observe que:

- $|V^+ \setminus VM_0| = |V^+| = n/2$.
- $|V^+ \setminus VM_l| = |V^+ \setminus VM_{l-1}| - |\mathcal{P}_l|$.
- $|V^+ \setminus VM_L| = 0$.

Assim, somando (3.37) para $l = 1, \ldots, L$ temos que

$$\sum_{l=1}^{L} d_+^{(l)} |\mathcal{P}_l| \leq (r + 4n) \sum_{i=1}^{n/2} \frac{1}{i}.$$

Logo, usando um limite superior para o truncamento da série Harmônica (vide [44, pág. 54]),

$$\sum_{i=1}^{m} \frac{1}{i} \leq (1 + \log m),$$

temos a desigualdade desejada. □

A seguir apresentamos uma proposição que relaciona os valores de g e r:

Proposição 3.32 *Se* $g \geq 10\left(1 + \log \frac{n}{2}\right)$, *então* $r \leq n$.

Demonstração: Como veremos a seguir (Proposição 3.39 – Seção 3.6.4), r é aumentado de no máximo $2d_0^{(l)} f_l/g$ na l-ésima iteração da função EscEmparelhamentoParalelo (r só é alterado na função AjusteDualRelaxado). Assim, como $r_0 = 0$, então

$$r \leq \frac{2}{g} \sum_{l=1}^{L} d_0^{(l)} f_l.$$

Por outro lado, $\sum_{l=1}^{L} d_0^{(l)} f_l$ é precisamente $\sum_{l=1}^{L} d_+^{(l)} |\mathcal{P}_l|$. Logo, pela Proposição 3.31,

$$r \leq \frac{2(r + 4n)}{g} \left(1 + \log \frac{n}{2}\right).$$

Como $g \geq 10\left(1 + \log \frac{n}{2}\right)$, então $r \leq n$. □

Complexidade Para Obter a Quádrupla (M, y, c, r)

Recordamos que L é o número de iterações do laço **repita** da função EscEmparelhamentoParalelo num patamar. O Lema 3.33, abaixo, é demonstrado de forma semelhante ao Lema 3.16 (Seção 3.4 - Gabow e Tarjan), usando o Teorema 3.30.

Lema 3.33 *O valor de L é no máximo $1 + 2(r + 4n)^{1/2}$.* □

Proposição 3.34 *Num patamar da função EscEmparelhamentoParalelo, temos que*

$$\sum_{l=1}^{L} |E\mathcal{P}_l| \leq \frac{n}{2} + 2\,(r + 4n)\left(1 + \log \frac{n}{2}\right).$$

Demonstração: Para qualquer caminho M-aumentante P, temos que $|EP| = 2\,|EP \setminus M| - 1$. Por hipótese, cada $\mathcal{P}_l$ $(1 \leq l \leq L)$ é uma coleção disjunta de caminhos M_{l-1}-aumentantes. Portanto, como $M_l := M_{l-1} \oplus E\mathcal{P}_l$, então $|E\mathcal{P}_l| = 2\,|M_l \setminus M_{l-1}| - |\mathcal{P}_l|$. Assim,

$$\sum_{l=1}^{L} |E\mathcal{P}_l| = 2\sum_{l=1}^{L} |M_l \setminus M_{l-1}| - \sum_{l=1}^{L} |\mathcal{P}_l|\,.$$

Por hipótese M_L é perfeito, portanto

$$\sum_{l=1}^{L} |\mathcal{P}_l| = |M_L| = \frac{n}{2}.$$

Logo,

$$\sum_{l=1}^{L} |E\mathcal{P}_l| = 2\sum_{l=1}^{L} |M_l \setminus M_{l-1}| - \frac{n}{2}. \tag{3.38}$$

Ademais, y_0 é $c_\emptyset$-independente, então $y_0(V) \leq |M_L| + c(M_L)$. Assim, como $|M_L| = n/2$ e $c(M_0) = 0$ temos que

$$0 \leq \frac{n}{2} + c(M_L) - c(M_0) - y_0(V).$$

Por outro lado, temos que

- $c(M_L) - c(M_0) = \sum_{l=1}^{L} \Big[c(M_l) - c(M_{l-1})\Big]$.
- $-y_0(V) = \sum_{l=1}^{L} -y_0(VM_l \setminus VM_{l-1})$.

Assim,

$$0 \leq \frac{n}{2} + \sum_{l=1}^{L} \Big[c(M_l) - c(M_{l-1}) - y_0(VM_l \setminus VM_{l-1})\Big]. \tag{3.39}$$

Multiplicando ambos os termos da desigualdade (3.39) por 2 e somando-a (3.38), obtemos

$$\sum_{l=1}^{L} |E\mathcal{P}_l| \leq \frac{n}{2} + 2\sum_{l=1}^{L} \Big[|M_l \setminus M_{l-1}| + c(M_l) - c(M_{l-1}) - y_0(VM_l \setminus VM_{l-1})\Big]. \tag{3.40}$$

Damos abaixo uma desigualdade e quatro igualdades, justificadas a seguir.

$$\begin{aligned}
c(M_l) - c(M_{l-1}) &= c(M_l \setminus M_{l-1}) - c(M_{l-1} \setminus M_l). && (3.41)\\
|M_l \setminus M_{l-1}| + c(M_l \setminus M_{l-1}) &= y_l(V(M_l \setminus M_{l-1})). && (3.42)\\
y_l(V(M_{l-1} \setminus M_l)) &\leq c(M_{l-1} \setminus M_l). && (3.43)\\
y_l(V(M_l \setminus M_{l-1})) - y_l(V(M_{l-1} \setminus M_l)) &= y_l(VM_l \setminus VM_{l-1}). && (3.44)\\
y_l(VM_l \setminus VM_{l-1}) - y_0(VM_l \setminus VM_{l-1}) &= d_+^{(l)} |\mathcal{P}_l|. && (3.45)
\end{aligned}$$

- Para justificar (3.41), note que:
  1. $c(M_l) = c(M_l \setminus M_{l-1}) + c(M_l \cap M_{l-1})$.
  2. $c(M_{l-1}) = c(M_{l-1} \setminus M_l) + c(M_l \cap M_{l-1})$.

  Subtraindo a segunda igualdade da primeira, obtém-se (3.41).
- A igualdade (3.42) segue do fato que $M_l \setminus M_{l-1} \subseteq E_{y_l, c_{M_{l-1}}}$.
- A desigualdade (3.43) segue da $c_{M_{l-1}}$-independência de y_l.
- A igualdade (3.44) é justificada lembrando que $\mathcal{P}_l$ é uma coleção disjunta de caminhos M_{l-1}-aumentantes e que $M_l = M_{l-1} \oplus E\mathcal{P}_l$.
- A igualdade (3.45) segue da Proposição 3.29.

Somando (3.41) até (3.45) obtemos que

$$|M_l \setminus M_{l-1}| + c(M_l) - c(M_{l-1}) - y_0(VM_l \setminus VM_{l-1}) \leq d_+^{(l)} |\mathcal{P}_l| . \qquad (3.46)$$

Assim, pelas Equações (3.40) e (3.46) temos que

$$\sum_{l=1}^{L} |E\mathcal{P}_l| \leq \frac{n}{2} + 2 \sum_{l=1}^{L} d_+^{(l)} |\mathcal{P}_l| .$$

Portanto, pela Proposição 3.31 temos a desigualdade desejada. □

Analisamos agora a complexidade de uma iteração do laço **repita** da função EscEmparelhamentoParalelo.

A complexidade da primeira parte desse laço, uma chamada da Função AjusteDualRelaxado (Algoritmo 3.13 – Seção 3.6.4), na l-ésima iteração é

$$O\left(\left(m/p + d_0^{(l)} + \frac{ng}{f_l}\right) \log p\right) . \qquad (3.47)$$

A complexidade da segunda parte desse laço, uma chamada da Função ColeçãoMaximalParalelo (Algoritmo 3.14 – Seção 3.6.5), na l-ésima iteração é

$$O\left((m/p + |E\mathcal{P}_l|) \log p\right) . \qquad (3.48)$$

A complexidade da terceira parte desse laço é

$$O\left(m/p\right) , \qquad (3.49)$$

pois tanto o ajuste dual como a diferença simétrica podem ser realizados nessa complexidade.

De (3.47), (3.48) e (3.49) temos que a complexidade de uma iteração é

$$O\left(\left(m/p + d_0^{(l)} + \frac{ng}{f_l} + |E\mathcal{P}_l|\right) \log p\right) . \qquad (3.50)$$

A seguir concluímos a análise da complexidade de um patamar. Para tanto, usamos que $r \leq n$ e $g = 10\left(1 + \log \frac{n}{2}\right)$.

Pelo Lema 3.33, temos que

$$L = O\left((r+n)^{1/2}\right) = O\left(n^{1/2}\right) . \qquad (3.51)$$

Por outro lado, mesmo na última iteração existe pelo menos um vértice M-livre em V^+, assim, pelo Teorema 3.30:

$$\sum_{l=1}^{L} d_0^{(l)} = d_+^{(L)} \overset{\text{Teo 3.30}}{=} O(r+n) = O(n).$$

Cada vez que chamamos a função AjusteDualRelaxado, o número de vértices M-livres é menor. Assim,

$$\sum_{l=1}^{L} \frac{ng}{f_l} \leq ng \sum_{f=1}^{n/2} \frac{1}{f} \overset{\text{Sér. Harm.}}{\leq} ng\left(1+\log\frac{n}{2}\right) = O(ng\log n) = O\left(n\log^2 n\right).$$

Pela Proposição 3.34,

$$\sum_{l=1}^{L} |E\mathcal{P}_l| = O((r+n)\log n) = O(n\log n).$$

De (3.50) e pelas observações feitas acima, temos que a complexidade de um patamar é

$$O\left(\left(L + mn^{1/2}/p + n\log^2 n\right)\log p\right).$$

Mas, de (3.51), a complexidade é

$$O\left(\left(mn^{1/2}/p + n\log^2 n\right)\log p\right).$$

Por outro lado, como $p \leq m/\left(n^{1/2}\log^2 n\right)$, então

$$\frac{mn^{1/2}}{p} \geq \frac{mn^{1/2}}{\frac{m}{n^{1/2}\log^2 n}} = n\log^2 n,$$

ou seja, $n\log^2 n = O\left(mn^{1/2}/p\right)$. Portanto, a complexidade de um patamar é

$$O\left(\left(mn^{1/2}/p\right)\log p\right).$$

Assim, a complexidade da função EscEmparelhamentoParalelo satisfaz (3.30).

Na Seção 3.6.3 apresentamos um algoritmo paralelo para realizar uma busca em largura, usando p processadores. Esta busca em largura será necessária nas Seções 3.6.4 e 3.6.5, onde apresentamos, respectivamente, a função AjusteDualRelaxado e a função ColeçãoMaximalParalelo.

3.6.3 Algoritmo Paralelo Para Busca em Largura

- Tipo: Paralelo.
- Complexidades:
  - Obtenção da estrutura de dados apropriada para representar o grafo na busca em largura, a partir de uma razoável representação de G: $O\left(\frac{m}{p}\log p\right)$.
  - Inicialização da busca em largura: $O\left(\frac{n}{p}\right)$.
  - Busca em largura: $O\left(\left(\frac{\overline{m}}{p}+K\right)\log p\right)$, onde $\overline{m}$ é o número de arestas incidentes nos vértices visitados na busca em largura e K é o número de iterações da busca em largura, ou seja, é a profundidade da árvore gerada pela busca.
- Processadores: p.
- Modelo: PRAM e EREW.
- Restrições:
  - O grafo é simples, bipartido e não possui vértices isolados.
  - O conjunto S dos vértices ($S \subseteq V$) dos quais desejamos iniciar a busca em largura é fornecido em p listas, onde:
    1. A diferença entre o número de vértices de quaisquer duas listas é no máximo 1.
    2. Cada vértice de S aparece no máximo numa lista.

Nesta seção apresentamos um algoritmo para realizar uma busca em largura num grafo bipartido.

Estrutura de Dados Para Representar o Grafo G

A estrutura de dados usada para representar o grafo G é fundamental na busca em largura. A seguir fazemos a sua descrição.

A estrutura de dados para representar o grafo na busca em largura deve ter a seguinte característica:

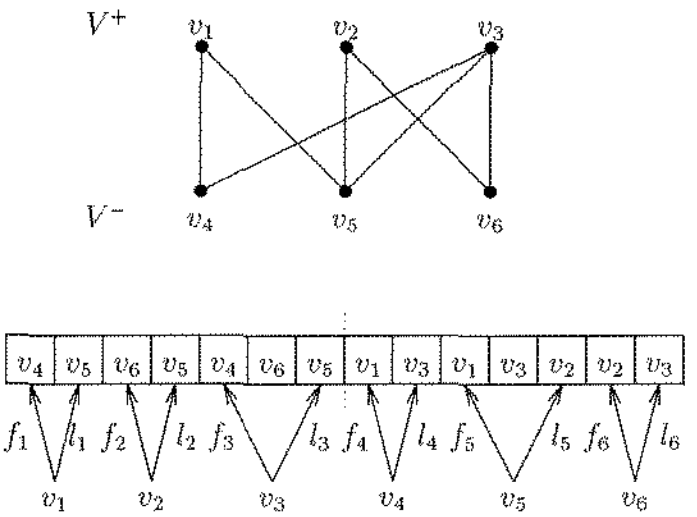


Figura 3.6: Vetor de $2m$ posições para representar o grafo na busca em largura.

> para cada vértice $v \in V$, a estrutura de dados deve fornecer os vértices adjacentes a v.

Em particular, utilizamos um vetor de $2m$ posições, onde para cada vértice $v \in V$, os seus vértices adjacentes estão armazenados consecutivamente no vetor. O conjunto de vértices adjacentes a v_j $(1 \leq j \leq n)$ são indicados, no vetor, pelo par de índices (f_j, l_j). A Figura 3.6 mostra um exemplo.

Esta estrutura de dados pode ser construída, a partir de uma razoável representação de G, em

$$O\left((m/p)\log p\right). \tag{3.52}$$

Suponha, por exemplo, que as arestas de G estão representadas num vetor $Q[1 \ldots m]$, onde cada Q_q $(1 \leq q \leq m)$ é uma aresta do tipo (v^-, v^+) com $v^- \in V^-$ e $v^+ \in V^+$. Descreveremos como obter a representação desejada para o grafo G, a partir de uma representação que usa o vetor Q, numa complexidade dada por (3.52). Suponha que os vértices de G são numerados de 1 a n.

Na obtenção da estrutura de dados desejada, utilizamos um algoritmo de ordenação, denominado ordena vetor paralelo, apresentado na Seção B.7

(Algoritmo B.3 – OrdenaVetorParalelo). Esse algoritmo ordena um vetor de registros, usando como chave para ordenação um dos campos numéricos do registro. Além de retornar um novo vetor com os registros ordenados, para cada valor z dentro do intervalo que a chave de ordenação pode assumir, é também retornado um par de índices que indicam a parte do vetor ordenado onde estão os registros com a chave de ordenação iguais a z. A complexidade desse algoritmo de ordenação é

$$O\left(\frac{x+y}{p}\log p\right), \tag{3.53}$$

onde x é o número de elementos do vetor que desejamos ordenar e y é o tamanho do intervalo que a chave de ordenação pode assumir.

Recordamos que cada elemento do vetor $Q[1\ldots m]$ é um registro (aresta), contendo um vértice de V^+ e um de V^-. Assim, ao usar o algoritmo OrdenaVetorParalelo, passando como parâmetros o vetor Q e os vértices de V^+ (V^-) como chave para ordenação, obtemos um vetor de tamanho m, sendo que para cada $v \in V^+$ ($v \in V^-$, respectivamente), os seus vértices adjacentes estão armazenados consecutivamente no novo vetor e temos um par de índices (f_v, l_v) que indicam a parte do novo vetor onde aparecem os vértices adjacentes a v. Veja por exemplo a Figura 3.7.

Logo, para obter o vetor desejado para representar o grafo G na busca em largura, basta concatenar o vetor obtido ao se executar o algoritmo OrdenaVetorParalelo, passando como parâmetros o vetor Q e os vértices de V^+ como chave para ordenação, com o vetor obtido ao se executar o algoritmo OrdenaVetorParalelo, passando como parâmetros o vetor Q e os vértices de V^- como chave para ordenação (veja a Figura 3.7).

Como, no nosso caso, $x = m$ e $y = n$, então de (3.53) temos que a complexidade para se obter a estrutura desejada é

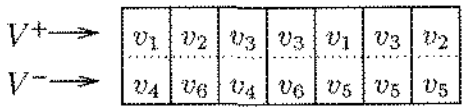
$$O\left(\frac{m+n}{p}\log p\right).$$

Mas $n = O(m)$, pois o grafo não possui vértices isolados. Logo, a complexidade para se obter a estrutura desejada é

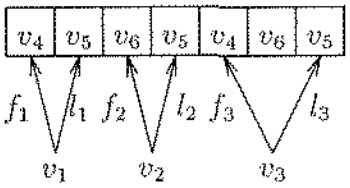
$$O\left(\frac{m}{p}\log p\right),$$

como queríamos.

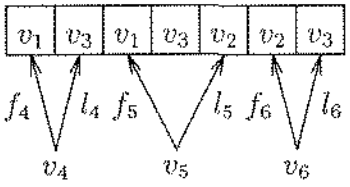
(a) Vetor Q



(b) OrdenaVetorParalelo (Q, m, chave V^+, n)



(c) OrdenaVetorParalelo (Q, m, chave V^-, n)



(d) Vetores (b) e (c) concatenados

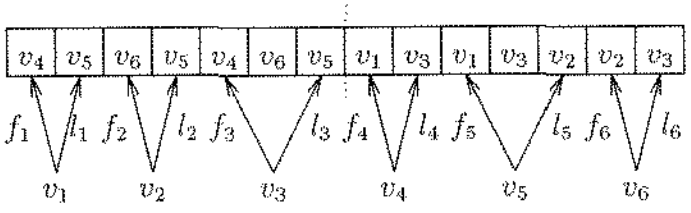


Figura 3.7: Obtenção da estrutura de dados desejada para representar o grafo na busca em largura (d), a partir do vetor de arestas Q (a). As arestas do vetor Q correspondem as arestas do grafo apresentado na Figura 3.6.

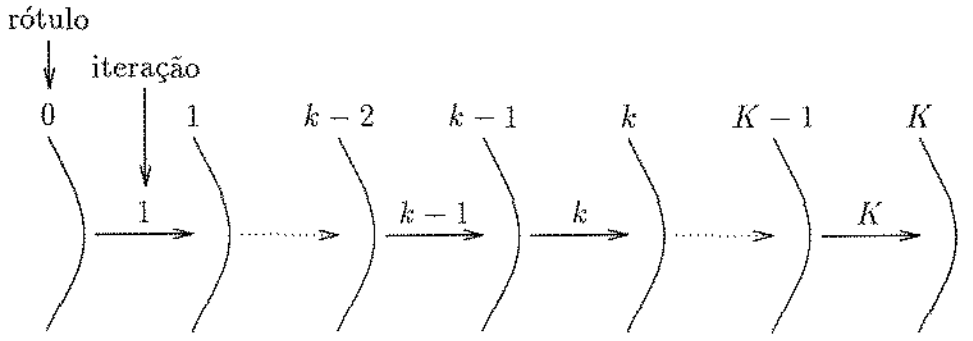


Figura 3.8: Busca em largura – iterações da busca e rótulos dos vértices. Os vértices de onde iniciamos a busca possuem rótulo 0.

Busca em Largura

Na k-ésima iteração ($1 \leq k \leq K$) da busca em largura visitamos as arestas incidentes nos vértices de rótulo $k-1$, obtendo os vértices de rótulo k. Os vértices de onde iniciamos a busca possuem rótulo 0 (veja a Figura 3.8). Defina

$$m_k := \sum \{ \left| \widetilde{E}\left(\{v\}\right) \right| : v \text{ é um vértice de rótulo } k \},$$

ou seja, m_k é o número de arestas que incidem nos vértices de rótulo k. Como veremos a seguir, a complexidade da busca na k-ésima iteração é

$$O\left((1 + (m_{k-2} + m_{k-1})/p) \log p\right), \tag{3.54}$$

onde $m_{-1} = |S|$, ou seja, m_{-1} é o número de vértices com rótulo 0. Logo, como o grafo não possui vértices isolados, então a complexidade do algoritmo é satisfeita, pois

$$\sum_{k=1}^{K} (1 + (m_{k-2} + m_{k-1})/p) \log p = O\left((K + \overline{m}/p) \log p\right).$$

Logo, resta descrever como é realizada a busca numa k-ésima iteração, numa complexidade dada por (3.54). Seja $B[1 \ldots 2m]$ o vetor usado para representar a estrutura de dados na busca em largura e (f_v, l_v) $(1 \leq v \leq n)$ o par de índices, no vetor B, que indicam os vértices adjacentes a v.

Em especial, para cada processador p_i $(1 \leq i \leq p)$ usamos duas listas, W_i e A_i, onde

W_i: contém no máximo $\lceil m_{k-2}/p \rceil$ vértices com rótulo $k-1$.

A_i: contém no máximo $\lceil m_{k-1}/p \rceil$ vértices adjacentes aos vértices de rótulo $k-1$. Uma entrada desta lista corresponde a um par de índices (i, j) que indicam, no vetor B, vértices que são adjacentes a um vértice de rótulo $k-1$.

Entre todas as listas W_i $(1 \leq i \leq p)$, um vértice $v \in V$ de rótulo $k-1$ ocorre exatamente uma vez. As listas A_i $(1 \leq i \leq p)$ contém todos os vértices adjacentes aos vértices de rótulo $k-1$.

As listas W_i $(1 \leq i \leq p)$ dos vértices de rótulo 0, ou seja, dos vértices de onde iniciamos a busca em largura, correspondem as p listas fornecidas à busca em largura.

A k-ésima iteração da busca pode ser dividida em duas partes: na primeira parte, dada as listas W_i $(1 \leq i \leq p)$ dos vértices de rótulo $k-1$, determinamos as listas A_i $(1 \leq i \leq p)$ dos vértices adjacentes aos vértices de rótulo $k-1$; na segunda parte, dada uma lista A_i $(1 \leq i \leq p)$, determinamos a lista W_i da próxima iteração $(k+1)$, isto é, dos vértices com rótulo k. A seguir descrevemos estas partes:

Primeira parte: determinamos as listas A_i $(1 \leq i \leq p)$ dos vértices adjacentes aos vértices das listas W_i $(1 \leq i \leq p)$. Para tanto, cada processador p_i visita a lista de vértices W_i. Inicialmente, as listas A_i $(1 \leq i \leq p)$ estão vazias. A j-ésima visita ocorre da seguinte forma:

(i) Cada um dos p_i $(1 \leq i \leq p)$ processadores extrai um vértice da lista W_i e o armazena num vetor $u[1 \ldots p]$.

(ii) Obtemos um vetor $X^{(j)}$ que contém todos os vértices adjacentes aos vértices do vetor u.

Para tanto, utilizamos um algoritmo denominado seqüência partes vetor, apresentado na Seção B.8 (Algoritmo B.4), que consiste em: dadas p partes disjuntas de um vetor, onde o comprimento destas partes podem ser zero, este algoritmo utiliza os p processadores para construir um novo vetor que é formado pela concatenação de todas essas partes.

a)

i	1	2	3	4	5
$\lvert E_{u_i}\rvert$	1	3	0	2	1

b)

r	1	2	3	4	5	6	7
$X_r^{(j)}$	$B_{f_{u_1}}$	$B_{f_{u_2}}$	$B_{f_{u_2}+1}$	$B_{f_{u_2}+2}$	$B_{f_{u_4}}$	$B_{f_{u_4}+1}$	$B_{f_{u_5}}$

Tabela 3.1: a) Número de vértices adjacentes aos vértices do vetor $u[1 \ldots p]$ $(p = 5)$. b) Vetor $X^{(j)}$ obtido, usando a tabela a).

Assim, para obter o vetor $X^{(j)}$ basta executar o Algoritmo B.4 (SeqüênciaPartesVetor), passando como parâmetros o vetor B e as p partes do vetor B indicadas pelos índices (f_{u_i}, l_{u_i}) $(1 \leq i \leq p)$. Veja por exemplo a Tabela 3.1.

(iii) Distribuímos os vértices do vetor $X^{(j)}$ nas listas A_i $(1 \leq i \leq p)$ de tal forma que a diferença entre o número de vértices de quaisquer duas listas A_i seja no máximo 1.

Seja A_i $(1 \leq i \leq p)$ a última lista a receber um vértice na visita anterior $(j-1)$, então começamos a distribuir os elementos do vetor $X^{(j)}$ a partir da lista A_{i+1} (se $i = p$, então começamos a partir da lista A_1). Veja por exemplo a Tabela 3.2.

Portanto, no final das visitas temos as listas A_i $(1 \leq i \leq p)$ desejadas (cada lista A_i possui no máximo $\left\lceil \frac{m_{k-1}}{p} \right\rceil$ vértices). Observe que na j-ésima visita:

- O item (i) possui uma complexidade $O(1)$.
- A complexidade do item (ii) é $O\left(\frac{\left|X^{(j)}\right|}{p} + \log p\right)$, pois o Algoritmo B.4 (SeqüênciaPartesVetor) possui essa complexidade, onde $\left|X^{(j)}\right|$ é o tamanho do vetor $X^{(j)}$.

A_1	A_2	A_3	A_4	A_5
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
o_1	o_2	o_3	o_4	B_{fu_1}
B_{fu_2}	B_{fu_2+1}	B_{fu_2+2}	B_{fu_4}	B_{fu_4+1}
B_{fu_3}				

Tabela 3.2: Preenchimento das listas A_i $(1 \leq i \leq p)$, usando o vetor $X^{(j)}$ da Tabela 3.1 $(p = 5)$. A lista A_4 foi a última a receber um vértice na visita anterior $(j-1)$. Os simbolos o_j $(1 \leq j \leq 4)$ representam vértices adicionadas as listas A_i $(1 \leq i \leq p)$ nas visitas anteriores.

- A complexidade do item (iii) é $O\left(\log p + \frac{\left|X^{(j)}\right|}{p}\right)$, pois em $O(\log p)$ os p processadores sabem o tamanho de $X^{(j)}$ e o índice i da lista A_i que recebeu o último vértice na visita anterior $(j-1)$ e em $O\left(\frac{\left|X^{(j)}\right|}{p}\right)$ distribuímos os vértices de $X^{(j)}$ pelas listas A_i.

Logo, a complexidade da j-ésima visita é

$$O\left(\frac{\left|X^{(j)}\right|}{p} + \log p\right).$$

Seja J o número de visitas realizadas. Assim, a complexidade desta parte na k-ésima iteração é

$$O\left(\sum_{j=1}^{J} \frac{\left|X^{(j)}\right|}{p} + J \log p\right).$$

Como as listas W_i's possuem no máximo $\left\lceil \frac{m_{k-2}}{p} \right\rceil$ vértices cada, então

$$J = O\left(\frac{m_{k-2}}{p}\right).$$

Por outro lado,

$$\sum_{j=1}^{J} \left| X^{(j)} \right| = O\left(m_{k-1}\right).$$

Assim, temos que a complexidade desta parte na k-ésima iteração é

$$O\left(\frac{m_{k-1}}{p} + \left(1 + \frac{m_{k-2}}{p}\right) \log p\right). \tag{3.55}$$

Segunda parte: cada p_i $(1 \leq i \leq p)$ obtém, a partir da lista A_i, a lista W_i da próxima iteração $(k+1)$. A seguir descrevemos como obter essas novas listas W_i $(1 \leq i \leq p)$. Seja v_i $(1 \leq i \leq p)$ um vértice obtido da lista A_i. Como na lista W_i devemos colocar somente os vértices com rótulo k que ainda não foram visitados e como estas listas devem conter apenas uma ocorrência de um vértice, então podemos utilizar o algoritmo de ordenação de Cole [11] para ordenar os vértices $v_1, \ldots, v_p$, obtendo a seqüência de vértices $(u_1, \ldots, u_p)$ e em seguida, cada processador p_i $(1 \leq i \leq p)$ efetua as seguintes verificações:

1. i $(1 \leq i \leq p)$ é o menor inteiro tal que $u_i = z$, onde z é um elemento da seqüência $(u_1, \ldots, u_p)$. Isto é feito para garantir que os p processadores não possuam vértices repetidos.
2. u_i ainda não foi visitado na busca.
3. u_i não pertence a nenhum W_i $(1 \leq i \leq p)$.

Em caso afirmativo, u_i é acrescentado a W_i. Observe que acrescentamos no máximo $\lceil m_{k-1}/p \rceil$ vértices a cada lista W_i, pois cada A_i possui no máximo $\lceil m_{k-1}/p \rceil$ vértices.

Para verificar se um vértice foi ou não visitado na busca em largura ou se pertence a algum W_i, usamos um vetor de tamanho n, onde o elemento de índice v $(1 \leq v \leq n)$ desse vetor indica se o vértice foi ou não visitado na busca em largura ou se pertence a algum W_i. Daí a complexidade de inicialização da busca em largura ser $O\left(n/p\right)$, pois precisamos inicializar os elementos deste vetor com vértice não visitado.

Assim, a complexidade desta parte na k-ésima iteração é

$$O\left(\left(1 + m_{k-1}/p\right) \log p\right), \tag{3.56}$$

pois cada processador visita no máximo $\lceil m_{k-1}/p \rceil$ vértices e o algoritmo de Cole ordena p elementos em $O(\log p)$.

De (3.55) e (3.56) temos que a complexidade da k-ésima iteração da busca em largura é

$$O\left((1 + (m_{k-2} + m_{k-1})/p) \log p\right),$$

ou seja, satisfaz a complexidade dada por (3.54).

No Algoritmo 3.12 temos uma implementação da busca em largura.

3.6.4 Algoritmo Ajuste Dual Relaxado

- Tipo: Paralelo.
- Complexidade: $O\left(\left(\frac{m}{p} + d_0 + \frac{ng}{f}\right) \log p\right)$.
- Processadores: p, onde p é no máximo $\frac{m}{n^{1/2} \log^2 n}$.
- Modelo: PRAM e EREW.

Dado um grafo G bipartido (V^+ e V^-) e uma quádrupla (M, y, c, r) ótima, este algoritmo realiza ajustes na variável dual y e em r, de tal forma que:

1. Os valores de y para cada vértice de $V^+ \setminus VM$ são acrescidos de um mesmo valor, d_0.
2. Os valores de y para os vértices de $V^- \setminus VM$ permanecem inalterados.
3. O valor de r nunca diminui, mas $r \leq n$.
4. (M, y, c, r) permanece ótima.

Os ajustes descritos acima são efetuados até que um dos seguintes casos ocorra:

- Existe um caminho M-aumentante em G_{M,y,c_M}.
- M é máximo em G.

Além disso, também são obtidos:

```
/* Dado um grafo G e um conjunto de vértices S fornecido em p listas (a
diferença do número de vértices de quaisquer duas listas é no máximo 1), faz
uma busca em largura a partir dos vértices de S.
A complexidade do algoritmo é O((m̄/p + K) log p), onde m̄ é o número de
arestas incidentes nos vértices visitados na busca em largura e K é o número
de iterações da busca, usando p processadores – EREW. */

BuscaLarguraParalela (G, S)
  m_{-1} ← |S|; k ← 1; W ← S; /* W[1...p] */
  repita
      /* primeira parte */
      /* z indica o último índice de A[1...p] onde acrescentamos vértices */
      z ← p; A ← ∅;
      para j ← 1 até ⌈m_{k-2}/p⌉ faça
      início
          para i ← 1 até p faça em paralelo
              ⟨W_i, u_i⟩ ← RemoveVértice (W_i); /* u[1...p] */
          X^(j) ← SeqüênciaPartesVetor (B, [(f_{u_1}, l_{u_1}), ..., (f_{u_p}, l_{u_p})]);
          ⟨A, z⟩ ← InsereVértices (A, X^(j), z);
      fim
      /* segunda parte */
      W ← ∅;
      para j ← 1 até ⌈m_{k-1}/p⌉ faça
      início
          /* se A_i é vazio, então v_i = ∞ */
          para i ← 1 até p faça em paralelo
              ⟨A_i, v_i⟩ ← RemoveVértice (A_i); /* v[1...p] */
          u ← OrdenaçãoCole (v, p); /* u[1...p] */
          u_0 ← 0; /* u_0 é diferente dos demais valores de u[1...p] */
          para i ← 1 até p faça em paralelo
              se u_i ≠ ∞ e u_i ≠ u_{i-1} e u_i não foi visitado e u_i ∉ W_i's
                  então W_i ← W_i ∪ {u_i};
      fim
      k ← 1 + k;
  até que W = ∅;
```

Algoritmo 3.12: Busca em largura (paralelo).

- O valor d_0, que é o acréscimo que a variável dual y sofreu em cada vértice M-livre de V^+.

- O grafo $H(G_{M,y,c_M}, M)$ M-simples.

Neste algoritmo utilizamos uma operação denominada *relaxamento* que consiste em: dado um conjunto $W^+ \subseteq V^+ \cap VM$, para cada $w \in W^+$, $y(w)$ é diminuído de 1. Observe que esta operação preserva a Propriedade (ii) de quádrupla ótima (y é c_M-independente – página 110); porém, esta operação diminui $y(VM)$ de $|W^+|$. Portanto, a cada operação de relaxamento de W^+, aumentamos o valor de r de $|W^+|$, para garantir que a quádrupla (M, y, c, r) permaneça ótima (Propriedade (iii), $c(M) \leq r + y(VM)$). Note que após aplicarmos relaxamento em W^+, as únicas arestas de E_{M,y,c_M} incidindo em vértices de W^+ são precisamente as de M.

A seguir descrevemos a função AjusteDualRelaxado. Na descrição da função vamos utilizar dois parâmetros f e g, onde f é o número de vértices M-livres de V^+ e g é um número maior do que zero. Conforme vimos na Proposição 3.32, g deve assumir o valor $10\left(1 + \log \frac{n}{2}\right)$ para que r seja no máximo n.

De uma forma semelhante à busca húngara (Algoritmo 3.1), nesta função vamos usar os seguintes conjuntos: S, T e $J := \{(u, v) \in E : u \in S \text{ e } v \in V^- \setminus T\}$. Inicialmente fazemos $S := V^+ \setminus VM$ e $T := \emptyset$. A seguir executamos repetidas vezes uma iteração composta por dois passos, **ajuste** e **construção**, que descrevemos a seguir:

1. **Ajuste:** seja $W^- := E_{M,y,c_M}(S) \setminus T$, ou seja, W^- consiste dos extremos, em $V^- \setminus T$, das arestas de E_{M,y,c_M} que incidem em vértices de S. Se $W^- = \emptyset$, podem ocorrer as seguintes situações:

   (a) T contém um vértice M-livre, ou seja, existe um caminho M-aumentante de arestas M-justas em E_{M,y,c_M}.

   (b) T não contém vértices M-livres e $J = \emptyset$. Neste caso M é um emparelhamento máximo em G (Corolário 2.2).

   (c) T não contém vértices M-livres e $J \neq \emptyset$. Neste caso aplicamos o ajuste sobre a variável dual y apresentado na demonstração do Teorema 3.3 (Seção 3.1 – busca húngara). Chamaremos esse ajuste de *padrão*. Note que após este ajuste temos que $W^- \neq \emptyset$.

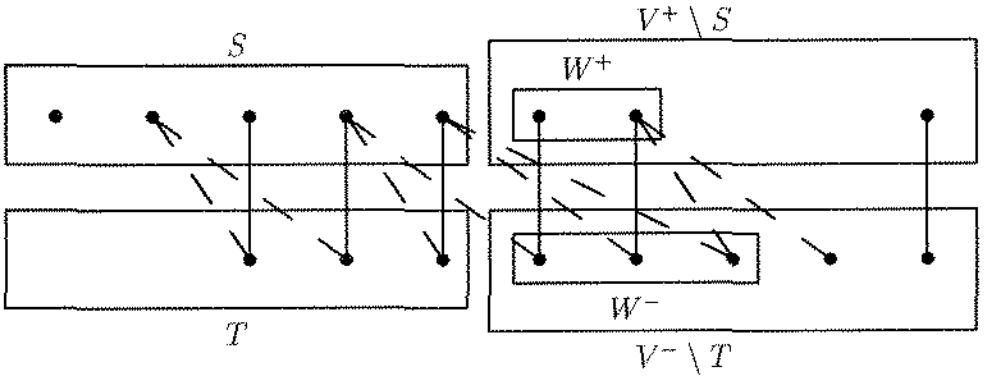


Figura 3.9: Grafo G_{M,y,c_M}, com os conjuntos S, T, W^+ e W^-.

Nos casos 1a (existe um caminho M-aumentante de arestas M-justas em E_{M,y,c_M}) e 1b (M é um emparelhamento máximo em G) interrompemos a função.

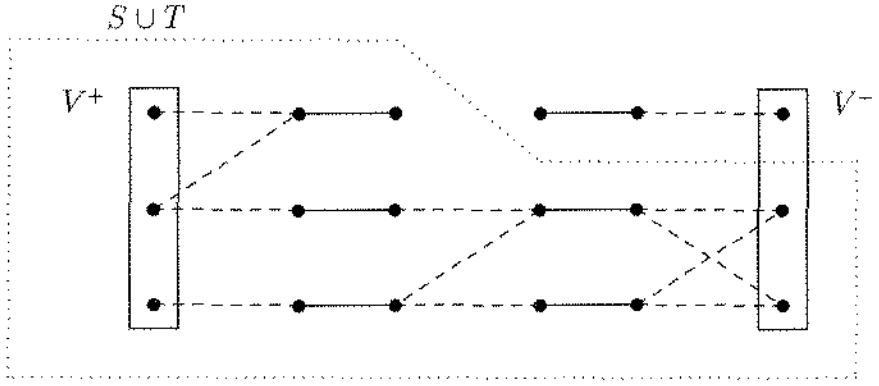
2. **Construção:** seja $W^+ := M(W^-)$ (veja a Figura 3.9, onde temos um grafo G_{M,y,c_M}). Adicionamos os vértices de W^- e W^+, respectivamente, a T e S, ou seja, fazemos $T \leftarrow T \cup W^-$ e $S \leftarrow S \cup W^+$. Se $|W^+| < f/g$, então relaxamos o conjunto W^+. Note que após relaxarmos o conjunto W^+, temos $W^- = \emptyset$.

Como veremos a seguir (Proposição 3.35), no fim do processo descrito acima, S e T são, respectivamente, os conjuntos de vértices de V^+ e V^- que são términos de caminhos M-alternados em G_{M,y,c_M} com origem em vértices M-livres de V^+. Assim, para determinar o grafo $H(G_{M,y,c_M}, M)$ M-simples (veja a definição na página 113) basta fazer, depois que encerrarmos o processo descrito acima, uma busca em largura no grafo $G_{M,y,c_M}[S \cup T]$, a partir dos vértices M-livres de T, usando caminhos M-alternados (veja a Figura 3.10).

No Algoritmo 3.13 temos uma implementação das idéias descritas acima.

Completamos agora a descrição do algoritmo, apresentando mais alguns resultados:

a) Grafo G_{M,y,c_M}



b) Grafo $H(G_{M,y,c_M}, M)$

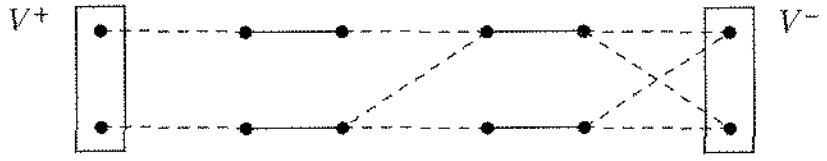


Figura 3.10: Grafo G_{M,y,c_M} e seu respectivo grafo $H(G_{M,y,c_M}, M)$.

/* Dado um grafo G bipartido (V^+,V^-) e uma quádrupla (M, y_0, c, r) ótima, retorna $\langle y, r, d_0, H(G_{M,y,c_M}, M)\rangle$, onde a quádrupla (M, y, c, r) é ótima, d_0 é o acréscimo que a variável dual y sofreu em cada vértice M-livre de V^+ e o grafo $H(G_{M,y,c_M}, M)$ é M-simples.
A complexidade do algoritmo é $O\left(\left(m/p + d_0 + \frac{ng}{f}\right)\log p\right)$, usando p processadores, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$ - EREW. */

AjusteDualRelaxado (G, M, y_0, c, r)
  $S \leftarrow V^+ \setminus VM$; $T \leftarrow \emptyset$; $f \leftarrow |S|$; $g \leftarrow 10\left(1 + \log\frac{n}{2}\right)$;
  $y \leftarrow y_0$; $d_0 \leftarrow 0$; FimLaço $\leftarrow$ **falso**; $i \leftarrow 0$; $\langle\langle W_0^+ \leftarrow S;\rangle\rangle$
  repita
    $W^- \leftarrow E_{M,y,c_M}(S) \setminus T$; $J \leftarrow \{(u,v) \in E : u \in S \text{ e } v \in V^- \setminus T\}$;
    $i \leftarrow 1 + i$; $\Delta_i \leftarrow 0$;
    se $W^- = \emptyset$ **então**
      se $\exists v \in T$, onde v é M-livre **ou** $J = \emptyset$ **então**
        FimLaço $\leftarrow$ **verdade**
      senão início
        $\Delta_i \leftarrow \min\{c_M(u,v) - (y(u) + y(v)) : (u,v) \in J\}$;
        $d_0 \leftarrow d_0 + \Delta_i$;
        $$y(v)\ \underset{(v\in V)}{\leftarrow} \begin{cases} y(v) - \Delta_i & \text{se } v \in T \\ y(v) + \Delta_i & \text{se } v \in S \\ y(v) & \text{nos demais casos;} \end{cases}$$
        $W^- \leftarrow E_{M,y,c_M}(S) \setminus T$;
      fim
    se FimLaço = **falso então início**
      $W_i^+ \leftarrow M(W^-)$; $T \leftarrow T \cup W^-$; $S \leftarrow S \cup W_i^+$;
      se $\left|W_i^+\right| < f/g$ **então início**
        para cada $v \in W_i^+$ **faça em paralelo** $y(v) \leftarrow y(v) - 1$;
        $r \leftarrow r + \left|W_i^+\right|$;
      fim
    fim
  até que FimLaço;
  $H \leftarrow$ BuscaLarguraAlternada $(G_{M,y,c_M}[S \cup T], M, T \setminus VM)$;
  devolva $\langle y, r, d_0, H\rangle$;

Algoritmo 3.13: Ajuste dual usando a operação relaxamento.

Proposição 3.35 *No fim da função AjusteDualRelaxado, S e T são, respectivamente, os conjuntos de vértices de V^+ e V^- que são términos de caminhos M-alternados em G_{M,y,c_M} com origem em vértices M-livres de V^+.*

Demonstração: Durante a execução do algoritmo, todo vértice de $S \cup T$ é término de um caminho M-alternado em G_{M,y,c_M} com origem em $V^+ \setminus VM$.

Ao final da execução W^- é vazio, ou seja, toda aresta de E_{M,y,c_M} que incide em S tem o seu outro extremo em T. Portanto, nenhum outro vértice, além dos vértices de $S \cup T$ é término de tais caminhos. □

Proposição 3.36 *No fim da função AjusteDualRelaxado, para todo vértice $v \in V \setminus VM$:*

$$y(v) - y_0(v) = \begin{cases} d_0 & se\ v \in V^+ \\ 0 & se\ v \in V^-. \end{cases}$$

Demonstração: A operação de relaxamento é aplicada somente sobre vértices M-ocupados, assim esta operação não interfere no valor dos vértices M-livres. Por outro lado, o ajuste padrão (busca húngara) altera os valores da variável dual dos vértices de $S \cup T$: S inclui $V^+ \setminus VM$ e T é disjunto de $V^- \setminus VM$ (quando o ajuste é realizado). Assim, temos a igualdade desejada. □

Proposição 3.37 *A quádrupla (M, y, c, r) retornada por esta função é ótima.*

Demonstração: A variável y é c_M-independente, pois tanto o ajuste padrão como a operação de relaxamento preservam esta propriedade. O ajuste padrão preserva o valor de $y(VM)$ e as operações de relaxamento diminuem $y(VM)$, porém r é aumentado desse valor. Assim, (M, y, c, r) é ótima. □

Apresentamos a seguir as duas proposições que deixamos de apresentar na Seção 3.6.2.

Proposição 3.38 *Dado um grafo G_{M,y,c_M} sem ciclos M-alternados, a função AjusteDualRelaxado não cria tais ciclos.*

Demonstração: Durante a execução do algoritmo temos as seguintes situações:

1. A operação de relaxamento não cria novas arestas M-justas e portanto não cria ciclos M-alternados.

2. O ajuste padrão não cria ciclos M-alternados. Suponha, por contradição, que o ajuste padrão cria um ciclo M-alternado. Então, após o ajuste padrão existem arestas α_1 e α_2 M-justas, tais que α_1 possui extremos em S e $V^- \setminus T$ e α_2 possui extremos em $V^+ \setminus S$ e T. Porém, para tornar α_1 M-justa, somamos Δ à variável dual y dos vértices de S e diminuímos Δ à variável dual y dos vértices de T. Então, α_2 não é M-justa, contradição.

Assim, temos que o grafo não cria tais ciclos. □

Proposição 3.39 *O valor de r aumenta de no máximo $2d_0 f/g$ numa chamada da função AjusteDualRelaxado.*

Demonstração: Como numa chamada da função AjusteDualRelaxado só podemos ter uma operação de relaxamento que não é seguida de um ajuste padrão, então numa chamada temos no máximo $1 + d_0$ operações de relaxamento. Assim, estas operações dimínuem a variável dual de no máximo $(1 + d_0)f/g \leq 2d_0 f/g$. □

Vamos agora analisar a complexidade do algoritmo AjusteDualRelaxado. Seja I o número de iterações efetuadas no algoritmo.

Proposição 3.40 *O valor de I é no máximo $d_0 + \left\lfloor \frac{ng}{2f} \right\rfloor$.*

Demonstração: Existem duas possibilidades, mutuamente exclusivas, na execução de uma iteração:

1. $|W^+| \geq f/g$. Claramente, este caso ocorre no máximo $\left\lfloor \frac{ng}{2f} \right\rfloor$ vezes, pois cada vez que ele ocorre adicionamos no mínimo $\lceil f/g \rceil$ vértices a S.

2. Na próxima iteração executamos o ajuste padrão. Sempre que executamos este ajuste, d_0 é aumentado de Δ, onde $\Delta > 0$. Assim, este caso é executado no máximo d_0 vezes.

Logo, temos o valor desejado. □

Como veremos a seguir, a complexidade da i-ésima iteração do laço **repita** desta função é

$$O\left(\left(1+\left(\left|W_i^+\right|+\left|\widetilde{E}\left(W_i^+\right)\right|\right)/p+\Delta_i\right)\log p\right), \tag{3.57}$$

onde $\left|\widetilde{E}\left(W_i^+\right)\right|$ é o número de arestas incidindo em W_i^+.

Além disso,

1. Pela Proposição 3.40, temos no máximo $O\left(d_0+\left\lfloor\frac{ng}{f}\right\rfloor\right)$ iterações.
2. $\sum_{i=1}^{I}\left(\left|W_i^+\right|+\left|\widetilde{E}\left(W_i^+\right)\right|\right)=O\left(n+m\right)$.
3. $\sum_{i=1}^{I}\Delta_i=d_0$.

Assim, como $n=O\left(m\right)$, então a complexidade deste laço **repita** na função é

$$O\left(\left(d_0+\frac{ng}{f}+\frac{m}{p}\right)\log p\right).$$

Por outro lado, observe que no fim da função AjusteDualRelaxado, o comprimento de um caminho M-aumentante em $G_{M,y,c_M}[S\cup T]$ é no máximo $2I$. Assim, usamos uma implementação semelhante à busca em largura paralela (Seção 3.6.3) na Função BuscaLarguraAlternada, onde a medida que vamos determinando o conjunto T no algoritmo AjusteDualRelaxado, determinamos também as p listas dos vértices de $T\setminus VM$ necessárias na chamada da busca em largura paralela (Seção 3.6.3). Assim, temos que a complexidade da função BuscaLarguraAlternada é

$$O\left(\left(\frac{m}{p}+I\right)\log p\right)=O\left(\left(\frac{m}{p}+d_0+\frac{ng}{f}\right)\log p\right).$$

Logo, temos a complexidade desejada para o algoritmo AjusteDualRelaxado.

Verificamos agora que a complexidade de uma iteração do laço **repita** é dada por (3.57).

Observe que a complexidade para se realizar um ajuste padrão conforme descrito é muito elevada, pois precisamos determinar o valor de Δ e esta

operação possui uma complexidade $O\left(\frac{m}{p}\log p\right)$. Além disso, devemos fazer os ajustes na variável dual y, o que possui uma complexidade $O\left(\frac{n}{p}+\log p\right)$. Ou seja, a complexidade para se realizar um ajuste padrão desta maneira é $O\left(\frac{n}{p}+\frac{m}{p}\log p\right)$.

Para resolver este problema de complexidade, ao invés de determinarmos os Δ's e fazer os vários ajustes duais, vamos adaptar o algoritmo seqüencial de Gabow e Tarjan (apresentado na Seção 3.4) ao nosso problema. Nesse algoritmo seqüencial utilizávamos o algoritmo de Dijkstra para determinar os valores $d(v)$ $(v \in V)$ e o valor de d_0, e obtínhamos o valor da variável dual y dos vértices de V da seguinte forma:

$$\underset{(v\in V)}{y(v)} \leftarrow \begin{cases} y_0(v)-(d_0-d(v)) & \text{se } v\in V^- \text{ e } d(v)\leq d_0 \\ y_0(v)+(d_0-d(v)) & \text{se } v\in V^+ \text{ e } d(v)\leq d_0 \\ y_0(v) & \text{nos demais casos.} \end{cases} \tag{3.58}$$

Lembre-se que o valor de y obtido acima era o definitivo, isto é, não precisávamos mais realizar ajustes. A seguir mostramos como eram obtidos os valores $d(v)$ $(v \in V)$ no algoritmo seqüencial.

No algoritmo seqüencial tínhamos que o valor de d_+ (d_+ é a soma dos valores d_0) era no máximo $2n$. Assim, como $d_0 \leq d_+$, então ao invés de utilizarmos uma fila de prioridade no algoritmo de Dijkstra, utilizávamos um vetor R de tamanho $2n$ para determinar $d(v)$ e d_0, onde o elemento R_x apontava para uma lista de vértices com $d(v) = x$ (não precisávamos armazenar os vértices que possuíam $d(v) > 2n$, pois $d_0 \leq d_+$). Utilizávamos, também, um vetor U indexado pelos vértices de G (tamanho n), onde U_v $(v \in V)$ representava, caso tivesse, o rótulo definitivo $d(v)$ dado pelo algoritmo de Dijkstra ao vértice v.

Assim, inicialmente todos os vértices M-livres de V^+ estavam numa lista apontada por R_0. O algoritmo Dijkstra consistia em percorrer o vetor R e as listas apontadas por ele numa forma seqüencial e crescente, ou seja, quando estávamos no índice x do vetor R, visitávamos todos os vértices na lista apontada por R_x antes de irmos para o índice $x+1$. Isto era feito para determinar o próximo vértice que possuía um rótulo temporário em U. Assim, essa versão do algoritmo de Dijkstra possuía uma complexidade $O(m+n)$ (seqüencial).

A seguir mostramos uma adaptação (em paralelo) do método descrito acima, que devemos utilizar no algoritmo AjusteDualRelaxado para deter-

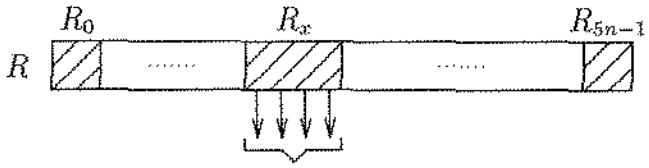


Figura 3.11: Vetor $R[0 \ldots 5n-1]$ para o algoritmo paralelo, onde $p = 4$.

minar d_0 e a variável dual y. Para tanto, precisamos fazer as seguintes modificações no vetor R descrito acima:

1. O tamanho de R deve ser $5n$, pois a cada vez que executamos o algoritmo AjusteDualRelaxado temos pelo menos um vértice M-livre em V^+ e como $r \leq n$, então pelo Teorema 3.30 temos que $d_+ \leq 5n$.

2. Ao invés de cada R_x possuir uma única lista dos vértices com $d(v) = x$, vamos ter p listas, onde cada lista vai conter no máximo $\lceil |R_x| / p \rceil$ vértices ($|R_x|$ é o número de vértices apontados por R_x). Assim, podemos aproveitar melhor o paralelismo, pois ao invés de analisarmos um vértice por vez, agora cada um dos p processadores pode extrair um vértice apontado por R_x (Veja a Figura 3.11).

A seguir relacionamos algumas características que existem entre a versão da Função AjusteDualRelaxado apresentada no Algoritmo 3.13 e a nova versão que estamos propondo:

1. Os vértices de R_x correspondem aos vértices de W^+ de uma iteração da versão anterior do algoritmo AjusteDualRelaxado. Depois de visitarmos os vértices de R_x, determinaremos a próxima posição de R não vazia. Seja x' esta posição. Note que:

   (a) $R_{x'}$ corresponde ao próximo W^+.

   (b) $x' = x + \Delta_i$, onde Δ_i é o valor do ajuste padrão quando visitamos os vértices de R_x.

2. Quando estamos em R_x e temos que $|R_x| < f/g$, então precisamos realizar uma operação de relaxamento sobre os vértices de R_x. Esta

operação de relaxamento corresponde a mover todos os vértices de R_x para R_{x+1}.

Para visitar os vértices apontados por R_x e garantir que as p listas apontadas por $R_{x'}$ $(0 \leq x' \leq 5n-1)$ possuam no máximo $\lceil |R_{x'}|/p \rceil$ vértices, utilizamos uma estratégia semelhante à de numa iteração da busca em largura paralela (Seção 3.6.3).

Analisamos agora a complexidade de uma iteração desta nova versão do algoritmo AjusteDualRelaxado, que, como vimos acima, corresponde a uma iteração da versão da função AjusteDualRelaxado apresentada no Algoritmo 3.13.

Sejam i e x, respectivamente, a i-ésima iteração desta nova versão e o índice em que estamos no vetor R. Se as listas apontadas por R_x são vazias, então determinamos o próximo índice x' $(x < x' \leq d_0)$, tal que $R_{x'} \neq \emptyset$ (ajuste dual). Isto possui uma complexidade

$$O\left((1+\Delta_i)\log p\right). \tag{3.59}$$

Quando temos que $R_x \neq \emptyset$, aplicamos uma estratégia semelhante à uma iteração da busca em largura paralela (Seção 3.6.3) para visitar os vértices apontados por R_x (W_i^+). Isto pode ser realizado numa complexidade

$$O\left(\left(1+\left(\left|W_i^+\right|+\left|\widetilde{E}\left(W_i^+\right)\right|\right)/p\right)\log p\right), \tag{3.60}$$

pois cada uma das p listas de R_x possui no máximo $\left\lceil \left|W_i^+\right|/p \right\rceil$ vértices e visitamos $O\left(\left|\widetilde{E}\left(W_i^+\right)\right|\right)$ arestas. Por outro lado, a operação de relaxamento possui uma complexidade $O(1)$, pois cada um dos p processadores concatena uma lista de vértices de R_x com uma lista de vértices de R_{x+1}. Assim, de (3.59) e (3.60) temos que a complexidade da i-ésima iteração é:

$$O\left(\left(1+\left(\left|W_i^+\right|+\left|\widetilde{E}\left(W_i^+\right)\right|\right)/p+\Delta_i\right)\log p\right),$$

como desejávamos (veja a Equação (3.57)).

No final desta nova versão do algoritmo AjusteDualRelaxado usamos o ajuste dual na variável y apresentado em (3.58) para obter os valores desejados de y. Esse ajuste pode ser realizado numa complexidade $O\left(\frac{n}{p}+\log p\right)$. Como só realizamos uma vez esse ajuste numa chamada do algoritmo AjusteDualRelaxado e $n = O(m)$, então a complexidade desse ajuste não interfere na complexidade final do algoritmo AjusteDualRelaxado.

3.6.5 Algoritmo Coleção Maximal Paralelo

- Tipo: Paralelo.
- Complexidade: $O((m/p + |E\mathcal{P}|)\log p)$.
- Processadores: p, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$.
- Modelo: PRAM e EREW.

Dados um grafo (bipartido) G e um emparelhamento M em G, tais que

- G não possui ciclos M-alternados e
- G é M-simples,

determinamos uma coleção maximal disjunta $\mathcal{P}$ de caminhos M-aumentantes em G.

A seguir apresentamos, sem demonstração, uma proposição útil na obtenção de um caminho M-aumentante em G:

Proposição 3.41 *Seja G um grafo bipartido e M um emparelhamento em G, tais que G não possui ciclos M-alternados e G é M-simples. Então todo caminho M-alternado Q em G pode ser estendido a um caminho M-aumentante P em G ($EQ \subseteq EP$), numa complexidade $O(|EP| - |EQ|)$, usando um processador, através de uma busca em profundidade.* □

A seguir fazemos uma descrição do algoritmo.

Se o grafo G não for vazio, o algoritmo para encontrar uma coleção maximal disjunta $\mathcal{P}$ consiste basicamente dos seguintes passos:

- Determinação de um caminho M-aumentante P em G, através de uma busca em profundidade (Proposição 3.41). Na verdade, conforme será visto posteriormente, P deve satisfazer algumas restrições, além de ser M-aumentante.
- Remoção de um superconjunto minimal W de VP tal que $H := G[V \setminus W]$ seja $M \cap EH$-simples.

- Se H não for vazio, recursivamente determinar uma coleção maximal disjunta $\mathcal{Q}$ de caminhos $M \cap EH$-aumentantes em H.

- Fazer $\mathcal{P} = \mathcal{Q} \cup \{P\}$.

Convém recordar que o grafo H mencionado acima nada mais é do que o grafo

$$H(G[V \setminus VP], M \setminus EP),$$

definido na página 113. Por hipótese, G é M-simples e livre de ciclos M-alternados, portanto W, o conjunto de vértices a remover, pode ser determinado da seguinte maneira:

1. Remova os vértices de P do grafo G e inclua-os em W.

2. Remova do grafo $G[V \setminus W]$ os vértices que satisfazem a seguinte restrição:

   > "vértices isolados ou vértices que eram M-ocupados em G e que possuem apenas arestas de M ou apenas arestas fora de M incidindo neles" (3.61)

   e inclua-os em W.

3. Repita o passo 2 até que não existam mais vértices que satisfaçam a restrição.

Observe que o grafo resultante é o grafo H desejado.

O processo descrito acima pode ser implementado através de duas buscas em largura no grafo G, a primeira a partir dos vértices de P em V^+ e a segunda a partir dos vértices de P em V^-, usando caminhos M-alternados (não visitamos as arestas de P). As arestas visitadas na busca em largura seriam removidas de G e a busca continuaria a partir dos vértices que passassem a satisfazer a Restrição (3.61). Assim, os vértices visitados nas duas buscas correspondem aos vértices de W. Veja por exemplo a Figura 3.12.

Ao determinar W, utilizando o algoritmo de busca em largura, podemos ter um problema de complexidade, pois o número de iterações da busca em largura (ou seja, a profundidade da árvore gerada pela busca) pode ser muito maior do que o comprimento do caminho P (lembre-se de que a complexidade da busca em largura em paralelo leva em consideração o número de iterações – Seção 3.6.3). A seguir explicamos como contornar este problema.

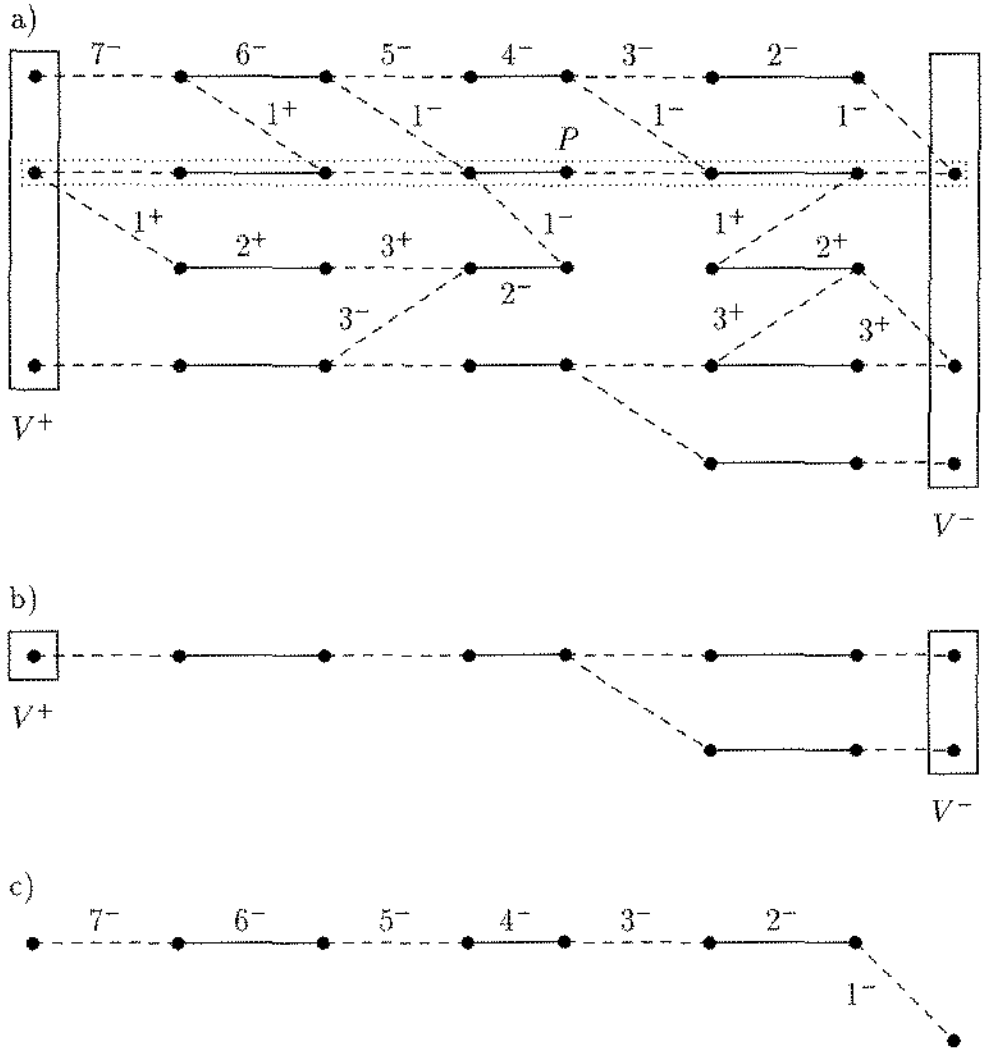


Figura 3.12: a) Grafo G M-simples e um caminho M-aumentante P em G. O número sobre uma aresta representa a iteração da busca em largura na qual ela foi visitada (removida) e o sinal sobre o número informa se a aresta foi visitada na busca em largura a partir dos vértices de V^+ ou V^- de P. b) Grafo $H\,(G[V \setminus VP], M \setminus EP)$. c) Maior caminho M-alternado de arestas removidas numa busca de caminho aceitável.

Sejam K^+ e K^- o número de iterações das buscas em largura efetuadas para determinar W a partir de V^+ e V^- respectivamente, e seja $K = \max\{K^+, K^-\}$. Dizemos que o caminho M-aumentante P é *aceitável* se

$$K \leq 2\left(|EP| + \frac{\left|\tilde{E}(W)\right|}{p}\right), \tag{3.62}$$

onde $\left|\tilde{E}(W)\right|$ é o número de arestas do grafo G que incidem em W. Se o caminho P for aceitável, adicionamos P à coleção maximal $\mathcal{P}$ e aplicamos, recursivamente, este processo no grafo M-simples H. Caso contrário, tentamos utilizar um caminho M-aumentante em G que possua um comprimento no mínimo o dobro de P. Isto garante que vamos ter um caminho M-aumentante aceitável em G.

Explicamos agora como obter um caminho M-aumentante $P^{(d)}$ que possua no mínimo o dobro do comprimento de P, quando P não é aceitável. Quando realizamos as buscas em largura para determinar W, determinamos em cada busca um caminho M-alternado de comprimento máximo, isto é, o caminho M-alternado determinado numa busca em largura possui um comprimento igual ou um a menos do que o número de iterações da busca. Seja Q o caminho M-alternado de maior comprimento obtido nas buscas em largura para determinar W (veja a Figura 3.12c). Assim, para obter $P^{(d)}$ estendemos o caminho M-alternado Q a um caminho M-aumentante no grafo G, isto é, no grafo original (Proposição 3.41). Claramente $\left|EP^{(d)}\right| \geq 2|EP|$, pois $|EQ| + 1 \geq K > 2|EP|$ (Equação (3.62)).

Uma descrição mais detalhada de como realizar esta busca em largura será dada no final desta seção (Busca de Caminho Aceitável).

No Algoritmo 3.14 temos uma implementação das idéias apresentadas acima.

Analisamos agora a complexidade do Algoritmo 3.14 (ColeçãoMaximal-Paralelo).

Seja L o número de caminhos M-aumentantes da coleção $\mathcal{P}$ (ou seja, o número de vezes que executamos o laço **enquanto**). Como veremos na Proposição 3.42, a complexidade para se determinar um caminho M-aumentante aceitável na l-ésima iteração do laço **enquanto** é

$$O\left(\left(\left|\tilde{E}\left(W_l^{(a)}\right)\right|/p + \left|EP_l^{(a)}\right|\right)\log p\right).$$

/* Dados um grafo G M-simples com bipartição (V^+,V^-) e um emparelhamento M em G, onde G não possui ciclos M-alternados, determina uma coleção maximal $\mathcal{P}$ de caminhos M-aumentantes em G.
A complexidade do algoritmo é $O\left((m/p+|E\mathcal{P}|)\log p\right)$, usando p processadores, onde p é no máximo $\frac{m}{n^{1/2}\log^2 n}$ - EREW. */

ColeçãoMaximalParalelo (G,M)
  $\mathcal{P} \leftarrow \emptyset$; $\langle\langle l \leftarrow 0;\rangle\rangle$
  enquanto $V \neq \emptyset$ **faça**
  início
    $\langle\langle l \leftarrow l+1;\rangle\rangle$
    $P_1 \leftarrow$ DeterminaCaminhoAumentante (G,M);
    $j \leftarrow 0$;
    /* determina um caminho aceitável */
    repita
      $j \leftarrow j+1$;
      $(W^+,K^+,Q^+) \leftarrow$
        BuscaDeCaminhoAceitável $(G,M,P_j,VP_j \cap V^+)$;
      $(W^-,K^-,Q^-) \leftarrow$
        BuscaDeCaminhoAceitável $(G[V \setminus W^-],M,P_j,VP_j \cap V^-)$;
      $W_j \leftarrow W^+ \cup W^-$;
      $K_j \leftarrow \max\{K^+,K^-\}$;
      se $K_j > 2\left(|EP_j| + \left|\widetilde{E}\left(W_j\right)\right|/p\right)$ **então**
      início
        $Q \leftarrow$ MaiorCaminho (Q^+,Q^-);
        $P_{j+1} \leftarrow$ EstendeCaminho (Q,G,M);
      fim
    até que $K_j \leq 2\left(|EP_j| + \left|\widetilde{E}\left(W_j\right)\right|/p\right)$;
    $\langle\langle P_l^{(a)} \leftarrow P_j; W_l^{(a)} \leftarrow W_j;\rangle\rangle$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_j\}$;
    $G \leftarrow G[V \setminus W_j]$;
  fim
  devolva $\mathcal{P}$;

Algoritmo 3.14: Coleção maximal disjunta de caminhos M-aumentantes em G.

Além disso, temos que:

1. A complexidade para se determinar um caminho M-aumentante P em G é $O(|EP|)$ (Proposição 3.41).
2. $\sum_{l=1}^{L} \left|\widetilde{E}\left(W_l^{(a)}\right)\right| = O(m)$.
3. $\sum_{l=1}^{L} \left|EP_l^{(a)}\right| = |E\mathcal{P}|$.
4. Como veremos mais adiante (Busca de Caminho Aceitável), o novo grafo G ($G \leftarrow G[V \setminus W]$) é, na realidade, determinado durante a busca de um caminho aceitável.

Assim, a complexidade para se obter uma coleção maximal de caminhos M-aumentantes $\mathcal{P}$ de G é

$$O\left(\left(\frac{m}{p} + |E\mathcal{P}|\right)\log p\right),$$

como desejávamos.

A seguir analisamos a complexidade para se obter um caminho aceitável. Suponha que verificamos J caminhos até obter um aceitável, então P_J é o caminho que é adicionado a $\mathcal{P}$ e W_J são os vértices removidos de G. Como veremos a seguir, a j-ésima tentativa ($1 \leq j \leq J$) possui uma complexidade

$$O\left(\left(\left|\widetilde{E}\left(W_j\right)\right|/p + |EP_{j+1}|\right)\log p\right), \tag{3.63}$$

onde assumimos que $P_{J+1} = P_J$. Assim, podemos estabelecer a seguinte proposição:

Proposição 3.42 *A complexidade para se determinar um caminho M-aumentante aceitável em G é $O\left(\left(\left|\widetilde{E}\left(W_J\right)\right|/p + |EP_J|\right)\log p\right)$.*

Demonstração: Por (3.63) e como $|EP_{j+1}| \geq 1$, temos que a complexidade é

$$O\left(\left(\sum_{j=1}^{J}\left(\frac{\left|\widetilde{E}\left(W_j\right)\right|}{p} + |EP_{j+1}|\right)\right)\log p\right).$$

Note que

$$\sum_{j=1}^{J}\left(\frac{\left|\widetilde{E}\left(W_j\right)\right|}{p}+\left|EP_{j+1}\right|\right)\leq\frac{\left|\widetilde{E}\left(W_J\right)\right|}{p}+2\left|EP_J\right|+\sum_{j=1}^{J-1}\left(\frac{\left|\widetilde{E}\left(W_j\right)\right|}{p}+\left|EP_j\right|\right).$$

Assim, é suficiente mostrar que $\sum_{j=1}^{J-1}\left(\left|\widetilde{E}\left(W_j\right)\right|/p+\left|EP_j\right|\right)=O\left(\left|EP_J\right|\right)$. Mas, pelo teste de aceitação, para $j<J$, $\frac{\left|\widetilde{E}(W_j)\right|}{p}+\left|EP_j\right|<\frac{\left|EP_{j+1}\right|}{2}$. Assim,

$$\sum_{j=1}^{J-1}\left(\frac{\left|\widetilde{E}\left(W_j\right)\right|}{p}+\left|EP_j\right|\right)\leq\sum_{j=2}^{J}\frac{\left|EP_j\right|}{2}\leq\sum_{j=2}^{J}\frac{\left|EP_J\right|}{2^{J-j+1}}<\left|EP_J\right|,$$

onde a penúltima desigualdade vem do fato que $2\left|EP_j\right|\leq\left|EP_{j+1}\right|$ para $j<J$. □

Verificaremos agora que a complexidade de uma tentativa para obter um caminho aceitável é dada por (3.63). Como veremos a seguir (Busca de Caminho Aceitável), a complexidade de uma busca na j-ésima tentativa é

$$O\left(\left(\left|\widetilde{E}\left(W_j\right)\right|/p+K_j\right)\log p\right)+O\left(\left|VP_j\right|\right).$$

O valor de $\left|\widetilde{E}\left(W_j\right)\right|$ $(1\leq j\leq J)$, usado no teste de aceitação, é fornecido pela busca de caminho aceitável. Note que na j-ésima tentativa, onde $1\leq j\leq J-1$, temos que o caminho P_j não é aceitável, logo $K_j=O\left(\left|EP_{j+1}\right|\right)$ e a complexidade de se determinar P_{j+1} é $O\left(\left|EP_{j+1}\right|\right)$ (Proposição 3.41). Por outro lado, na J-ésima tentativa temos que o caminho P_J é aceitável, logo $K_J=O\left(\left|EP_J\right|\right)$ e não precisamos determinar o caminho P_{J+1}. Daí, temos que a complexidade da j-ésima tentativa é

$$O\left(\left(\left|\widetilde{E}\left(W_j\right)\right|/p+\left|EP_{j+1}\right|\right)\log p\right),$$

onde $P_{J+1}=P_J$, como desejávamos.

Busca de Caminho Aceitável

A função BuscaDeCaminhoAceitável é semelhante à busca em largura paralela (Seção 3.6.3). Assim como na busca em largura paralela, nesta busca também precisamos de uma estrutura de dados especial para representar o grafo G. Na realidade, a estrutura de dados usada aqui é praticamente a mesma que é utilizada na busca em largura, só utilizamos um vetor a mais: o vetor $g[1 \ldots n]$, onde o elemento g_v $(1 \leq v \leq n)$ representa o grau do vértice $v \in V$ no grafo G. O vetor g é necessário para decidir quando um vértice $v \in V$ passa a satisfazer a Restrição (3.61), isto é, quando deve ser visitado nesta busca.

A estrutura de dados para representar G necessária na busca em largura é obtida no início do algoritmo ColeçãoMaximalParalelo. Na Seção 3.6.3 (busca em largura paralela) temos que a complexidade da obtenção desta estrutura é

$$O\left(\frac{m}{p}\log p\right).$$

Além disso, temos também uma complexidade $O(n/p)$ de inicialização do vetor g (para cada $v \in [1, \ldots, n]$ basta fazer $g_v = l_v - f_v + 1$, onde os valores f_v e l_v pertencem à nova estrutura de dados para representar G) e inicialização do vetor $b[1 \ldots n]$, onde b_v $(1 \leq v \leq n)$ indica se o vértice v foi ou não visitado na busca (b é o mesmo vetor utilizado na busca em largura - página 132). O vetor b é inicializado marcando cada vértice como não visitado. Portanto, como $n = O(m)$, então esta parte inicial não altera a complexidade do algoritmo ColeçãoMaximalParalelo.

Quando executamos uma busca de caminho aceitável, vamos marcando, no vetor b, os vértices visitados e diminuindo o grau dos vértices, no vetor g, à medida que vamos removendo as arestas que incidem neles.

Suponha que, ao final de uma busca, determinamos que o caminho encontrado é aceitável. Os vetores b e g já estão atualizados, para buscas futuras por novos caminhos aceitáveis. De fato, os vértices marcados como visitados em b não mais fazem parte do grafo G corrente e, além disso, os graus dos vértices, dado pelo veto g, correspondem aos graus dos vértices da versão corrente do grafo G.

A situação é ligeiramente mais complicada no caso em que uma iteração da busca determina que o caminho aumentante não é aceitável. Neste caso, é necessário restaurar os vetores b e g aos valores que tinham imediatamente

antes da busca.

A atualização tanto de b como de g é simples, desde que durante a busca sejam tomadas as seguintes precauções:

1. O processador que marca um vértice em b como visitado anota o vértice, naquele instante, numa lista local, chamada *lista de alterações*.

2. Antes do processador modificar o grau de um vértice, o processador anota, na sua lista de alterações, o vértice e o seu grau.

Ao final da busca, tendo sido determinado que o caminho M-aumentante obtido é inaceitável, cada processador desmarca em b os vértices que marcou como visitado e restaura em g o grau original dos vértices, percorrendo a sua lista de alterações. Observe que na lista de alterações podemos ter mais de um grau para um vértice. Assim, para restaurar o grau correto, basta tomar o maior grau do vértice nas listas de alterações.

A seguir fazemos uma análise da complexidade.

A complexidade de uma busca de caminho aceitável na j-ésima iteração é

$$O\left(\left(\left|\tilde{E}\left(W_j\right)\right|/p + K_j\right)\log p\right) + O\left(|VP_j|\right), \tag{3.64}$$

pois a busca de caminho aceitável é praticamente uma busca em largura paralela (Seção 3.6.3). A complexidade $O\left(|VP_j|\right)$ é necessária, pois precisamos determinar as p listas utilizadas no início da busca em largura paralela (Seção 3.6.3).

A complexidade de manipulação das listas de alteração evidentemente é majorada por (3.64). Assim, temos a complexidade desejada para a busca de caminho aceitável.

3.6.6 Observação

Os autores Balas, Miller, Pekny e Toth, em [6], apresentam um algoritmo paralelo que determina vários caminhos M-aumentantes em paralelo. São apresentados vários resultados computacionais do desempenho deste algoritmo.

Apêndice A

Algoritmos Para Grafos Não Bipartidos

Neste apêndice relacionamos alguns algoritmos que resolvem o problema de encontrar emparelhamentos máximos em grafos não bipartidos. As Tabelas A.1 e A.2 apresentam, respectivamente, o caso não ponderado e o caso ponderado. Vale ressaltar que o algoritmo paralelo da Tabela A.1 (Mulmuley, Vazirani e Vazirani [46]) é probabilístico (Monte Carlo).

Algoritmos Seqüenciais			
Data	Autor(es)	Ref.	Complexidade
1965	Edmonds	[15]	$mn^2 + n^3$
1975	Even e Kariv	[18]	$\min\{n^{2.5}, mn^{1/2}\log n\}$
1976	Gabow	[24]	n^3
1980	Micali e Vazirani	[45]	$mn^{1/2} + n^{3/2}$
1990	Blum	[9]	$(m+n)n^{1/2}$

Algoritmo Paralelo				
Data	Autores	Ref.	Complexidade	Processadores
1987	Mulmuley, Vazirani e Vazirani	[46]	$\log^2 n$	$n^{3.5}m$

Tabela A.1: Grafos gerais (não bipartidos) não ponderados.

Algoritmos Seqüenciais			
Data	Autor(es)	Ref.	Complexidade
1973	Edmonds e Johnson	[16]	$mn^2U + n^3U$
1982	Galil, Micali e Gabow	[29]	$mn \log n$
1990	Gabow	[23]	$mn + n^2 \log n$

Tabela A.2: Grafos gerais (não bipartidos) ponderados.

Apêndice B

Algoritmos Paralelos

Neste apêndice descrevemos alguns algoritmos paralelos. Na análise de complexidade, sempre que o número p de processadores usados e/ou o modelo EREW estiver subentendido, eles serão omitidos.

B.1 Menor Elemento de um Vetor

- Complexidade: $O(\log n)$.
- Processadores: $O(n/\log n)$.
- Modelo: PRAM e EREW.

Dado um vetor $k[1 \ldots n]$ de números, este algoritmo determina o valor e o índice de um elemento de valor mínimo em k. Um algoritmo similar a este, utilizando $O(n)$ processadores, pode ser encontrado em [3, pág. 121] (procura numa árvore).

Existe, porém, um truque utilizado em computação paralela que reduz o número de processadores utilizados no algoritmo citado acima. Este truque reduz de $O(n)$ para $O(n/\log n)$ o número de processadores utilizados, mantendo a complexidade em $O(\log n)$. Uma descrição rápida é apresentada a seguir:

1. Dividimos o vetor de n elementos em $\lceil n/\lceil \log n \rceil \rceil$ faixas de $\lceil \log n \rceil$ elementos cada, onde cada faixa possui um processador associado. Cada

processador percorre a sua faixa, de uma forma seqüencial, para determinar qual é o menor elemento. Como existem $O(\log n)$ elementos por faixa, logo gastamos $O(\log n)$ tempo nesta operação.

2. Depois, quando restam $\lceil n/\lceil \log n \rceil \rceil$ elementos, usamos o algoritmo para determinar o menor elemento de um vetor apresentado acima, numa complexidade $O(\log(n/\log n))$, usando $O(n/\log n)$ processadores.

Portanto, temos a complexidade desejada.

B.2 Menor Elemento de uma Matriz Quadrada

- Complexidade: $O(\log n)$.
- Processadores: $O(n^2/\log n)$.
- Modelo: PRAM e EREW.

Dada uma matriz $W[1 \ldots n, 1 \ldots n]$ de números, este algoritmo determina o valor e os índices de um elemento de valor mínimo em W. Neste algoritmo usamos basicamente o algoritmo para determinar o menor elemento de um vetor (Seção B.1).

B.3 Somar Elementos de um Vetor

- Complexidade: $O(\log p)$.
- Processadores: p.
- Modelo: PRAM e EREW.

Dado um vetor $k[1 \ldots q]$, este algoritmo soma os q elementos de k. Em [3, pág. 364] encontramos este algoritmo.

B.4 Algoritmo Todas as Somas

- Complexidade: $O(\log p)$.
- Processadores: p.
- Modelo: PRAM e EREW.

Dada uma seqüência de p números $A = (a_1, \ldots, a_p)$, o algoritmo todas as somas[1] determina o conjunto $S = \{s_1, \ldots, s_p\}$, onde para cada $i \in [1, \ldots, p]$,

$$s_i = \sum_{j=1}^{i} a_j,$$

ou seja, $s_i = a_1 + \cdots + a_i$. O autor Akl, em [3, pág. 46], apresenta uma implementação deste algoritmo que possui uma complexidade $O(\log p)$. A idéia básica utilizada, nessa implementação, é a propriedade associativa da soma.

B.5 Algoritmo Todas as Somas Parciais

- Complexidade: $O(\log p)$.
- Processadores: p.
- Modelo: PRAM e EREW.

Seja $A = (a_1, \ldots, a_p)$ uma seqüência de números. Considere um restrição na seqüência A para dividi-la em subseqüências de elementos consecutivos. O algoritmo todas as somas parciais determina todas as somas (Seção B.4) em cada uma das subseqüências.

Em particular, suponha que as subseqüências de A são estabelecidas através de rótulos de tal forma que dois elementos de A estão numa mesma subseqüência se e somente se seus rótulos são iguais, isto é, sejam r_i e r_j, respectivamente, os rótulos de a_i e a_j. Assim, a_i e a_j pertencem a uma mesma subseqüência de A se e somente se $r_i = r_j$ (e nesse caso $r_k = r_i$ para todo k

[1] Do inglês: *ALL SUMS*.

```
/* Dado um vetor a[1...p] de números e o vetor r[1...p] de rótulos, determina o vetor s[1...p], tal que s_i = Σ_{j=f}^{i} a_j (1 ≤ i ≤ p), onde f é o menor inteiro (1 ≤ f ≤ p) tal que r_f = r_i.
A complexidade do algoritmo é O(log p), usando p processadores - EREW.
*/

TodasAsSomasParciais (a, r, p)
  para i ← 1 até p faça em paralelo
      s_i ← a_i;
  para l ← 0 até ⌈log p⌉ − 1 faça
      para i ← 2^l + 1 até p faça em paralelo
          se r_{i−2^l} = r_i então
              s_i ← s_{i−2^l} + s_i;
  devolva s;
```

Algoritmo B.1: Determina todas as somas parciais.

tal que $i \leq k \leq j$). Assim, o algoritmo todas as somas parciais corresponde, para todo $i \in [1, \ldots, p]$, a determinar

$$s_i = \sum_{j=f}^{i} a_j,$$

onde f $(1 \leq f \leq p)$ é o menor inteiro tal que $r_f = r_i$.

Este algoritmo é uma generalização trivial do algoritmo todas as somas apresentado pelo autor Akl em [3, pág. 46]. O Algoritmo B.1 apresenta uma implementação da função TodasAsSomasParciais, que possui uma complexidade $O(\log p)$.

B.6 Número de Ocorrências

- Complexidade: $O\left(\frac{y}{p} + \frac{x}{p} \log p\right)$, onde os valores de x e y são explicados abaixo.
- Processadores: p.

i	1	2	3	4	5	6	7	8	9	10	11	12
r_i	1	1	1	3	5	5	5	5	5	6	7	7
a_i	1	1	1	1	1	1	1	1	1	1	1	1
s_i	1	2	3	1	1	2	3	4	5	1	1	2
			↑	↑					↑	↑		↑

Tabela B.1: Valores do vetor $s[1 \ldots p]$ para os respectivos vetores r e a, onde $p = 12$. As setas representam o maior inteiro tal que $r_i = z$.

- Modelo: PRAM e EREW.

Determina o número de ocorrências dos elementos de um vetor.

Seja $X[1 \ldots x]$ um vetor de inteiros, onde cada elemento de X está no intervalo de 1 a y inclusive. O algoritmo Número ocorrências determina um vetor $c[1 \ldots y]$ tal que c_j $(1 \leq j \leq y)$ indica o número de ocorrências do valor j no vetor X. No início do algoritmo temos o vetor c inicializado com zeros. Este algoritmo funciona em $\lceil x/p \rceil$ fases, onde em cada fase:

1. Cada um dos p processadores extrai um elemento do vetor X e armazena num vetor auxiliar $b[1 \ldots p]$.

2. Usando o algoritmo de Cole [11], ordenamos os elementos do vetor b, obtendo o vetor ordenado $r[1 \ldots p]$.

3. Inicializamos os elementos de um vetor $a[1 \ldots p]$ com 1.

4. Executamos o Algoritmo B.1 (TodasAsSomasParciais), passando como parâmetros o vetor de números a e o vetor de rótulos r, para obter o vetor soma $s[1 \ldots p]$.

   Seja i $(1 \leq i \leq p)$ o maior inteiro tal que $r_i = z$, onde z é um elemento do vetor r. Então, s_i é o número de elementos iguais a z no vetor b. A Tabela B.1 ilustra um exemplo.

5. Por último atualizamos o vetor c. Para tanto, cada processador p_i $(1 \leq i \leq p)$ verifica se i é o maior inteiro tal que $r_i = z$. Em caso afirmativo, faz $c_{r_i} \leftarrow c_{r_i} + s_i$.

```
/* Dado um vetor X[1...x], onde cada elemento de X está no intervalo de
1 a y inclusive, determina o vetor c[1...y], tal que c_i (1 ≤ i ≤ y) é o número
de vezes que o elemento i ocorre no vetor X.
A complexidade do algoritmo é O(y/p + (x/p) log p), usando p processadores -
EREW. */

NúmeroOcorrências (X, x, y)
  c ← InicializaVetorZeros (y);
  para j ← 1 até ⌈x/p⌉ faça
  início
      /* se existirem menos de p elementos para extrair,
         então as posições vazias do vetor b são preenchidas com ∞ */
      b ← ExtraiElementos (X);
      r ← OrdenaçãoCole (b, p);
      r_{p+1} ← 0; /* r_{p+1} é diferente dos demais elementos de r */
      para i ← 1 até p faça em paralelo
          a_i ← 1;
      s ← TodasAsSomasParciais (a, r, p);
      para i ← 1 até p faça em paralelo
          se r_i ≠ ∞ e r_i ≠ r_{i+1} então
              c_{r_i} ← c_{r_i} + s_i;
  fim
  devolva c;
```

Algoritmo B.2: Determina o número de ocorrências dos elementos de um vetor.

O Algoritmo B.2 implementa as idéias acima.

Analisamos agora a sua complexidade. A complexidade de uma fase é $O(\log p)$, pois:

- Os itens 1, 3 e 5 possuem uma complexidade $O(1)$.
- Os itens 2 e 4 possuem uma complexidade $O(\log p)$, pois tanto o algoritmo de Cole [11] como o Algoritmo B.1 (TodasAsSomasParciais) possuem essa complexidade.

Assim, como temos $O(x/p)$ fases e a inicialização do vetor c possui uma complexidade $O(y/p)$, então a complexidade deste algoritmo é

$$O\left(\frac{y}{p} + \frac{x}{p}\log p\right).$$

B.7 Ordena Vetor Paralelo

- Complexidade: $O\left(\frac{x+y}{p}\log p\right)$, onde os valores de x e y são explicados abaixo.
- Processadores: p.
- Modelo: PRAM e EREW.

Este algoritmo ordena um vetor de registros, usando como chave para ordenação um dos campos numéricos do registro. Além de retornar um novo vetor com os registros ordenados, para cada valor z dentro do intervalo que a chave de ordenação pode assumir, é também retornado um par de índices que indicam a parte do vetor ordenado onde estão os registros com a chave de ordenação igual a z.

Em outras palavras, dados um vetor $X[1 \ldots x]$ de registros e um campo numérico r desses registros, onde o valor de r é um inteiro no intervalo de 1 a y inclusive, obtemos um vetor $Z[1 \ldots x]$ com os registros do vetor X ordenados pelo campo r. Para cada $j \in [1, \ldots, y]$ também retornamos um par de índices (f_j, l_j) que indicam a parte do vetor Z que possui registros com $r = j$ $(f_j \leq l_j)$. Se não temos $r = j$ no vetor Z, retornamos $f_j = 1 + l_j$.

Este algoritmo é essencialmente um *bucket sort* em paralelo. A seguir fazemos a sua descrição. Quando nos referimos ao campo c do registro K_j (K é um vetor de registros), usaremos a seguinte notação: $K_j.c$.

Dividiremos a descrição do algoritmo ordena vetor paralelo em duas partes: na primeira determinamos os pares de índices (f_j, l_j) $(1 \leq j \leq y)$; na segunda, usando os índices f_j $(1 \leq j \leq y)$ determinados na primeira parte, obtemos o vetor Z. A seguir descrevemos estas partes:

Primeira parte: determinamos os pares de índices (f_j, l_j) $(1 \leq j \leq y)$.

Utilizando o Algoritmo B.2 (NúmeroOcorrências), determinamos o número de ocorrências de cada elemento de $X.r$, isto é, determinamos um vetor $c[1 \ldots y]$ tal que c_j $(1 \leq j \leq y)$ contém o número de registros de X com $r = j$. Por razões de complexidade, não podemos aplicar o Algoritmo todas as somas (Seção B.4) diretamente sobre o vetor c para determinarmos os índices (f_j, l_j) $(1 \leq j \leq y)$. Assim, dividimos o vetor c em $\lceil y/p \rceil$ partes, onde o tamanho de cada parte é no máximo p. Daí, aplicamos o algoritmo todas as somas sobre cada uma dessas partes. Finalmente, fazemos $f_i := 1 + c_{j-1}$ e $l_j := c_j$, onde $c_0 := 0$.

A complexidade desta parte é

$$O\left(\frac{y+x}{p} \log p\right) \tag{B.1}$$

pois:

1. Uma chamada do Algoritmo B.2 (NúmeroOcorrências) possui uma complexidade $O\left(\frac{y}{p} + \frac{x}{p} \log p\right)$.
2. Temos $O(y/p)$ partes e a complexidade do algoritmo todas as somas é $O(\log p)$.
3. A determinação de f e l, a partir de c, possui uma complexidade $O(y/p)$.

Segunda parte: usando os índices f_j $(1 \leq j \leq y)$ determinados na primeira parte, obtemos o vetor Z.

Usamos um vetor $\Delta[1 \ldots y]$, onde inicialmente temos $\Delta_j = f_j$. O valor de Δ_j indica o índice do vetor Z onde devemos inserir o próximo registro de X que possui $r = j$.

i	1	2	3	4	5	6
$h_i.r$	1	1	1	2	2	3
a_i	$\Delta_{h_1.r}$	1	1	$\Delta_{h_4.r}$	1	$\Delta_{h_6.r}$
s_i	$\Delta_{h_1.r}$	$\Delta_{h_1.r}+1$	$\Delta_{h_1.r}+2$	$\Delta_{h_4.r}$	$\Delta_{h_4.r}+1$	$\Delta_{h_6.r}$

Tabela B.2: Valores do vetor $s[1 \ldots p]$ para os respectivos vetores h e a, onde $p = 6$.

Um algoritmo ingênuo para obter Z quando temos $p = 1$ (apenas um processador) seria para cada $k \in [1, \ldots, x]$ fazer $Z_{\Delta_{X_k.r}} \leftarrow X_k$ e adicionar $\Delta_{X_k.r}$ de 1. Porém, quando temos mais de um processador surgem problemas com este algoritmo, pois dois ou mais processadores podem tentar, num mesmo instante, inserir registros distintos em um mesmo Z_k $(1 \leq k \leq x)$.

Para contornar este problema, podemos subdividir a solução em $\lceil x/p \rceil$ fases, onde em cada fase:

1. Cada um dos p processadores extrai um elemento do vetor X e armazena num vetor auxiliar $b[1 \ldots p]$.
2. Usando o algoritmo de Cole [11], ordenamos os registros do vetor b pelo campo r, obtendo o vetor de registros $h[1 \ldots p]$.
3. Inicializamos os elementos de um vetor $a[1 \ldots p]$ da seguinte forma: para cada $i \in [1 \ldots p]$, se i é o menor inteiro tal que $h_i.r = z$, onde z é um elemento do vetor $h.r$, então fazemos $a_i \leftarrow \Delta_{h_i.r}$; caso contrário, $a_i \leftarrow 1$.
4. Executamos o Algoritmo B.1 (TodasAsSomasParciais), passando como parâmetros o vetor de números a e o vetor de registros $d = [h_1.r, \ldots, h_p.r]$, para obter o vetor soma $s[1 \ldots p]$.
5. Atualizamos o vetor Z. Ou seja, para cada $i \in [1, \ldots, p]$ fazemos $Z_{s_i} \leftarrow h_i$.
   Note que a maneira como inicializamos o vetor a evita a concorrência na atualização de Z (veja a Tabela B.2).
6. Por último atualizamos o vetor Z. Para tanto, cada processador p_i $(1 \leq i \leq p)$ verifica se i é o maior inteiro tal que $h_i.r = z$. Em

caso afirmativo, faz $\Delta_{h_i,r} \leftarrow 1 + s_i$.

A complexidade de cada fase é $O(\log p)$ pois:

- Os itens 1, 3, 5 e 6 possuem uma complexidade $O(1)$.
- Os itens 2 e 4 possuem uma complexidade $O(\log p)$, pois tanto o algoritmo de Cole [11] como o Algoritmo B.1 (TodasAsSomasParciais) possuem essa complexidade.

Assim, como temos $O(x/p)$ fases, então a complexidade desta parte é

$$O\left(\frac{x}{p}\log p\right). \tag{B.2}$$

O Algoritmo B.3 (OrdenaVetorParalelo) implementa as idéias das duas partes acima.

De (B.1) e (B.2) temos que a complexidade do Algoritmo B.3 (OrdenaVetorParalelo) é

$$O\left(\frac{x+y}{p}\log p\right),$$

como desejávamos.

B.8 Seqüência Partes Vetor

- Complexidade: $O\left(\frac{|X|}{p} + \log p\right)$, onde $|X|$ é o número de elementos do vetor X explicado a seguir.
- Processadores: p.
- Modelo: PRAM e EREW.

Dadas p partes disjuntas de um vetor (os comprimentos de algumas destas partes podem ser zero), este algoritmo constrói um novo vetor concatenando essas partes. Ou seja, o novo vetor vai conter todos os elementos dessas partes. A Figura B.1 ilustra um exemplo.

Em outras palavras, dado um vetor $Y[1 \ldots y]$ e a i-ésima parte do vetor Y indicada pelo par de índices (f_i, l_i) $(1 \leq i \leq p)$, onde $f_i \leq 1 + l_i$ (se

/* Dados um vetor $X[1 \ldots x]$ de registros e um campo numérico r desses registros, onde o valor de r é um inteiro no intervalo de 1 a y inclusive, obtemos um vetor $Z[1 \ldots x]$ com os registros do vetor X ordenados pelo campo r. Para cada $j \in [1, \ldots, y]$ também retornamos um par de índices (f_j, l_j) que indicam a parte do vetor Z que possui registros com $r = j$ ($f_j \leq l_j$). Se não temos $r = j$ no vetor Z, retornamos $f_j = l_j + 1$.
A complexidade do algoritmo é $O\left(((x + y)/p) \log p\right)$, usando p processadores – EREW. */

```
OrdenaVetorParalelo (X, x, r, y)
  /* primeira parte */
  c ← NúmeroOcorrências ([X_1.r, ..., X_x.r], x, y); a_1 ← 1;
  para j ← 0 até ⌈y/p⌉ − 1 faça início
      para i ← 1 até p − 1 faça em paralelo a_{i+1} ← c_{jp+i};
      s ← TodasAsSomas (a, p);
      para i ← 1 até p faça em paralelo início
          f_{jp+i} ← a_i; l_{jp+i} ← a_i + c_{jp+i}; Δ_{jp+i} ← f_{jp+i};
      fim
      a_1 ← 1 + l_{jp+p};
  fim
  /* segunda parte */
  para j ← 1 até ⌈x/p⌉ faça início
      /* se existirem menos de p elementos para extrair,
         então as posições vazias do vetor b são preenchidas com ∞ */
      b ← ExtraiElementos (X); h ← OrdenaçãoCole ([b_1.r, ..., b_p.r], p);
      h_0.r ← 0; h_{p+1}.r ← 0; /* h_0.r e h_{p+1}.r são diferentes dos demais */
      para i ← 1 até p faça em paralelo
          se h_i.r ≠ ∞ e h_i.r ≠ h_{i−1}.r então a_i ← Δ_{h_i.r}
          senão a_i ← 1;
      s ← TodasAsSomasParciais (a, [h_1.r, ..., h_p.r], p);
      para i ← 1 até p faça em paralelo Z_{s_i} ← h_i;
      para i ← 1 até p faça em paralelo
          se h_i.r ≠ ∞ e h_i.r ≠ h_{i+1}.r então Δ_{h_i.r} ← 1 + s_i;
  fim
  devolva (Z, f, l);
```

Algoritmo B.3: Ordena um vetor em paralelo.

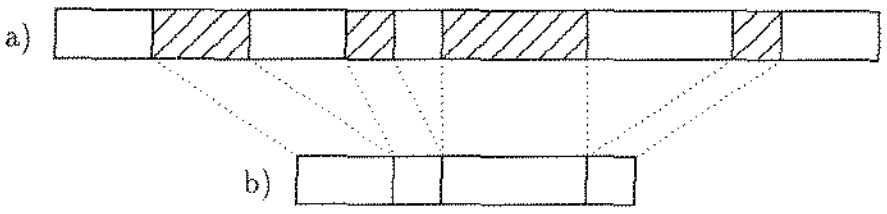


Figura B.1: a) Vetor Y com p partes disjuntas. b) Vetor X com as partes do vetor Y concatenadas.

i	-	1	2	3	4	5
$d_i = 1 + l_i - f_i$	-	1	7	0	2	0
s_i	0	1	8	8	10	10

Tabela B.3: Valores de $d_i = 1 + l_i - f_i$ e s_i, onde $p = 5$.

$f_i = 1 + l_i$ então o comprimento da i-ésima parte é zero), retornamos um vetor X contendo todos os elementos do vetor Y entre os índices f_i e l_i $(1 \leq i \leq p)$. A seguir descrevemos este algoritmo.

Faça $d[1 \ldots p]$ ser o vetor dos comprimentos das p partes, isto é, para todo $i \in [1 \ldots p]$,

$$d_i = 1 + l_i - f_i.$$

Usamos o algoritmo todas as somas (Seção B.4), passando como parâmetro o vetor d, para determinar $(1 \leq i \leq p)$

$$s_i = \sum_{j=1}^{i} d_j,$$

ou seja, determinar $s_i = d_1 + \cdots + d_i$. Assuma que $s_0 = 0$. Veja, por exemplo, a Tabela B.3, onde $p = 5$.

Os elementos da i-ésima parte $(1 \leq i \leq p)$, ou seja, os elementos do vetor Y entre os índices f_i e l_i, serão armazenados a partir do índice $s_{i-1} + 1$ até o índice s_i do vetor X. Note que o tamanho do vetor X corresponde ao valor de s_p, isto é, $|X| = s_p$. Na Tabela B.4 mostramos o preenchimento do vetor X para os valores da Tabela B.3.

r	1	2	3	4	5	6	7	8	9	10
X_r	Y_{f_1}	Y_{f_2}	Y_{f_2+1}	Y_{f_2+2}	Y_{f_2+3}	Y_{f_2+4}	Y_{f_2+5}	Y_{f_2+6}	Y_{f_4}	Y_{f_4+1}

Tabela B.4: O vetor X correspondente aos valores da Tabela B.3.

Mostraremos agora como preencher o vetor X.

Considere a tripla (f, l, j), onde Y_f até Y_l é a parte do vetor Y que deve ser inserida no vetor X, a partir do índice j de X. A tripla (f, l, j) indica uma ação a ser tomada por um processador. A esta tripla chamaremos de *tarefa*.

A *execução* de uma tarefa (f, l, j) consiste em inserir o elemento X_{f+i} na posição Y_{j+i}, onde i varia de 0 a $l - f$.

Inicialmente, para cada processador p_i $(1 \leq i \leq p)$ temos a seguinte tarefa:

$$(f_i, l_i, s_{i-1} + 1). \tag{B.3}$$

Logo, se todos os processadores executarem as suas respectivas tarefas por completo, então obtemos o vetor X totalmente preenchido da forma desejada. Note que o tamanho da tarefa (f, l, s) é $d = 1 + l - f$. Assim, essas tarefas podem ter tamanhos diferentes. Isto não é desejável para a complexidade do algoritmo, pois podemos sobrecarregar alguns processadores enquanto outros ficam praticamente ociosos. Para contornar esse problema, seguimos a seguinte estratégia até que todas as tarefas estejam completas:

(i) Cada processador p_i executa a sua tarefa (f, l, j), obtendo a tarefa (f', l, j'), até que um dos seguintes casos ocorra:

   1. A tarefa esteja completa, ou seja, $f' = l + 1$.
   2. A tarefa é executada $h := \left\lceil \frac{|X|}{p} \right\rceil$ vezes, ou seja, $f' - f = h = j' - j$ vezes.

(ii) A seguir determinamos em quais segmentos a execução da tarefa ainda não foi concluída. Na realidade, vamos determinar para cada tarefa (f', l, j') o número de segmentos de tamanho h que ainda necessitam ser efetuados para concluir a tarefa (f', l, j'). Denotando por g este

número, então

$$g := \left\lceil \frac{1 + l - f'}{h} \right\rceil .$$

Dependendo do valor de g temos que:

1. $g = 0$. A tarefa está completa.
2. $g > 0$. O problema consiste em agrupar todos os segmentos restantes ainda não processados, redistribuindo-os na forma de novas tarefas aos processadores. Observe que se usarmos a estratégia anterior para distribuir as tarefas obtemos uma solução cuja complexidade não é adequada aos nossos objetivos. Diante disso utilizamos a seguinte solução:
   (a) $g = 1$. Repassamos a tarefa (f', l, j') para o processador $p_{\lceil j'/h \rceil}$. Como $g = 1$, então esta tarefa será concluída na próxima iteração.
   (b) $g > 1$. Subdividimos a tarefa (f', l, j') em duas de forma que o tamanho de uma delas seja múltiplo de h e que as duas possuam aproximadamente o mesmo tamanho. Ou seja, definimos $l' := f' + h \left\lfloor \frac{g}{2} \right\rfloor$ e $j'' := j' + h \left\lfloor \frac{g}{2} \right\rfloor$ e criamos as tarefas: $(f', l' - 1, j')$ e (l', l, j''). Estas tarefas são, respectivamente, repassadas aos processadores $p_{\lceil j'/h \rceil}$ e $p_{\lceil j''/h \rceil}$.

A garantia de que não haverá problemas de concorrência na distribuição das tarefas é dada na Proposição B.1, apresentada a seguir:

Proposição B.1 *A cada processador é repassada no máximo uma tarefa, durante toda a execução do algoritmo.*

Demonstração: Sejam $t_1 = (f_1, l_1, j_1)$ e $t_2 = (f_2, l_2, j_2)$ duas tarefas repassadas, respectivamente, aos processadores i_1 e i_2, durante a execução do algoritmo.

Pela regra do repasse,

$$i_1 = \left\lceil \frac{j_1}{h} \right\rceil \quad \text{e} \quad i_2 = \left\lceil \frac{j_2}{h} \right\rceil .$$

Sem perda de generalidade, suponha que $j_1 < j_2$.

Se t_2 foi repassada, então ou t_2 é o resto de uma tarefa ou uma metade do resto de uma tarefa.

Se t_2 for o resto de uma tarefa ou a primeira metade do resto de uma tarefa, as h posições imediatamente anteriores a j_2 são preenchidas pelo processador que repassou t_2.

Por outro lado, se t_2 for a última metade do resto de uma tarefa, a primeira metade do mesmo resto tem tamanho múltiplo de h e todos os repasses oriundos da primeira metade também terão comprimento múltiplo de h.

Em ambos casos, um mesmo processador preencherá as h posições imediatamente anteriores a j_2, seja por repasse, seja diretamente.

Logo, podemos concluir que $j_1 \leq j_2 - h$ e portanto $i_1 < i_2$. □

O Algoritmo B.4 (SeqüênciaPartesVetor) determina o vetor X desejado, usando as idéias apresentadas acima. Como vimos na Proposição B.1, a cada processador é repassada no máximo uma tarefa, isto é, cada processador executa no máximo duas tarefas. Assim, por razões de complexidade, no Algoritmo B.4 (SeqüênciaPartesVetor) primeiro determinamos quais são as tarefas que devem ser executadas por cada processador (Algoritmo B.5 – DistribuiTarefas) e depois cada processador executa as suas respectivas tarefas (Algoritmo B.6 – ExecutarTarefas). Observe que pela maneira como as tarefas são repassadas, então vamos ter que executar no máximo $1 + \lceil \log p \rceil$ vezes o Algoritmo B.5 (DistribuiTarefas) para determinar as tarefas que devem ser executadas por cada processador. O Algoritmo B.7 (AlocaTarefa) aloca uma tarefa ao processador indicado.

A seguir analisamos a complexidade deste algoritmo.

A complexidade de uma chamada das Funções DistribuiTarefas (Algoritmo B.5) e AlocaTarefa (Algoritmo B.7) é $O(1)$.

Pela Proposição B.1, a cada processador é repassada no máximo uma tarefa. Assim, o tamanho total das tarefas executadas por cada processador é no máximo $2h$. Logo, a complexidade da Função ExecutarTarefas (Algoritmo B.6) é $O(h)$.

Como na Função SeqüênciaPartesVetor (Algoritmo B.4) temos que o valor de h é $\lceil |X| / p \rceil$, então temos a complexidade desejada:

$$O\left(\frac{|X|}{p} + \log p\right).$$

```
/* Dado um vetor Y[1...y] e cada uma das p partes do vetor Y indicada
pelo par de índices (f_i, l_i) (1 ≤ i ≤ p), onde f_i ≤ 1 + l_i (se f_i = 1 + l_i então
o comprimento da i-ésima parte é zero), retornamos um vetor X contendo
todos os elementos do vetor Y entre os índices f_i e l_i (1 ≤ i ≤ p).
A complexidade do algoritmo é O((|X| /p) + log p), usando p processadores
- EREW. */

SeqüênciaPartesVetores (Y, f, l)
  para i ← 1 até p faça em paralelo
      d_i ← 1 + l_i − f_i;
  s ← TodasAsSomas (d, p);
  s_0 ← 0;
  X ← CriaVetor (s_p); /* cria o vetor X[1...s_p] - X é global */
  h ← ⌈|X| / p⌉; /* distribui para os p processadores o valor de h */
  /* r_i - indica se o processador p_i possui alguma tarefa alocada
     t_i - é a tarefa alocada ao processador p_i
     c_i - é o número de tarefas que o processador p_i deve executar
     u_i - contém as tarefas que devem ser executadas pelo processador p_i
     r, t, c e u são globais */
  para i ← 1 até p faça em paralelo
  início
      r_i ← ∄tarefa;
      c_i ← 0;
  fim
  para i ← 1 até p faça em paralelo
      p_i.AlocaTarefa (i, (f_i, l_i, 1 + s_{i-1}));
  para q ← 0 até ⌈log p⌉ faça
      para i ← 1 até p faça em paralelo
          se r_i = ∃tarefa então
              p_i.DistribuiTarefas;
  para i ← 1 até p faça em paralelo
      p_i.ExecutarTarefas;
  devolva X;
```

Algoritmo B.4: Algoritmo paralelo para preencher o vetor X.

/* Dada uma tarefa t_i (ou seja, $r_i = \exists$tarefa), o processador p_i registra em u_i a parte da tarefa que deve ser executada por p_i e repassa, se existir, o resto da tarefa a outros processadores.
A complexidade do algoritmo é $O(1)$. */

DistribuiTarefas
  {processador p_i}
  $(f, l, j) \leftarrow t_i$;
  $r_i \leftarrow \nexists$tarefa;
  $c_i \leftarrow 1 + c_i$;
  $d \leftarrow \min\{h, 1 + l - f\}$;
  $u_i[c_i] \leftarrow (f, f + d - 1, j)$;
  $f \leftarrow f + d$; $j \leftarrow j + d$;
  se $f \leq l$ **então**
  início
      $g \leftarrow \left\lceil \frac{1+l-f}{h} \right\rceil$; $k \leftarrow \left\lceil \frac{j}{h} \right\rceil$;
      se $g = 1$ **então**
          AlocaTarefa $(k, (f, l, j))$
      senão /* $g > 1$ */
      início
          $f' \leftarrow f + h \left\lfloor \frac{g}{2} \right\rfloor$;
          AlocaTarefa $(k, (f, f' - 1, j))$;
          AlocaTarefa $(k + \left\lfloor \frac{g}{2} \right\rfloor, (f', l, j + h \left\lfloor \frac{g}{2} \right\rfloor))$
      fim
  fim

Algoritmo B.5: Determina as tarefas que devem ser executadas pelos processadores.

/* Dadas as tarefas u_i que o processador p_i deve executar, o processador p_i executa essas tarefas.
A complexidade do algoritmo é $O(h)$. */

```
ExecutarTarefas
  {processador p_i}
  para k ← 1 até c_i faça
  início
      (f, l, j) ← u_i[k];
      para q ← 0 até l − f faça
          X_{j+q} ← Y_{f+q}
  fim
```

Algoritmo B.6: Executa as tarefas alocadas.

/* Dadas uma tarefa (f, l, j) e um índice i, se essa tarefa não está completa, ou seja, se $f \leq l$, então a tarefa (f, l, j) é alocada ao processador p_i.
A complexidade do algoritmo é $O(1)$. */

```
AlocaTarefa (i, (f, l, j))
  se f ≤ l então
  início
      t_i ← (f, l, j);
      r_i = ∃tarefa;
  fim
```

Algoritmo B.7: Aloca uma tarefa a um processador.

Bibliografia

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design And Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Inc., Massachusetts, 1974.

[2] S. Akl. *Parallel Sorting Algorithms.* Academic Press, Inc., Orlando, Florida, 1985.

[3] S. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall International, Inc., New Jersey, 1989.

[4] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $o(n^{1.5}\sqrt{m/\log n})$. *Inf. Proc. Letters*, 37:237–240, Feb 1991.

[5] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *16th STOC*, pages 249–257, 1984.

[6] E. Balas, D. Miller, J. Pekny, and P. Toth. A parallel shortest augmenting path algorithm for the assignment problem. *J. of the Assoc. for Comput. Mach.*, 38(4):985–1004, Oct 1991.

[7] C. Berge. Two theorems in graph theory. In *Proc. Nat. Acad. Sci. USA*, pages 842–844, 1957.

[8] D. P. Bertsekas. A new algorithm for the assignment problem. *Math. Prog.*, 21:152–171, 1981.

[9] N. Blum. A new approach to maximum matching in general graphs. Technical Report 8546-CS, Universitat Bonn, Feb 1990.

[10] J. A. Bondy and U. S. R. Murty. *Graph Theory With Applications.* MacMillan, New York, 1976.

[11] R. Cole. Parallel merge sort. In *27th FOCS*, pages 511–516, 1986.

[12] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10(4):657–675, Nov 1981.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11(5):1277–1280, Sep 1970.

[15] J. Edmonds. Path, trees and flowers. *Canadian J. Math.*, 17:449–467, 1965.

[16] J. Edmonds and E. J. Johnson. Euler tours and the chinese postman. *Math. Prog.*, 5:88–124, 1973.

[17] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. of the Assoc. for Comput. Mach.*, 19(2):248–264, Apr 1972.

[18] S. Even and Kariv. An $O(n^{2.5})$ - algorithm for maximum matching in general graphs. In *16th FOCS*, pages 100–112, 1975.

[19] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, Dec 1975.

[20] H. Flanders and J. Price. *Calculus With Analytic Geometry.* Academic Press, Inc., New York, 1978.

[21] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian J. Math.*, 8(3):399–404, 1956.

[22] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *25th FOCS*, pages 338–346, 1984.

[23] H. Gabow. Data structures for weigthed matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[24] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. of the Assoc. for Comput. Mach.*, 23(2):221–234, Apr 1976.

[25] H. N. Gabow. Scaling algorithms for network problems. *J. of Comput. and Syst. Sci.*, 31(2):148–168, Oct 1985.

[26] H. N. Gabow and R. E. Tarjan. Almost-optimal speed-ups of algorithms for matching and related problems. In *20th STOC*, pages 514–527, 1988.

[27] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, Oct 1989.

[28] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *Computing Surveys*, 18(1):23–38, Mar 1986.

[29] Z. Galil, S. Micali, and H. Gabow. Priority queues with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. In *23rd FOCS*, pages 255–261, 1982.

[30] O. Garrido, S. Jarominek, A. Lingas, and W. Rytter. A simple randomized parallel algorithm for maximal f-matching. In *LNCS 583*, pages 165–176. Springer-Verlag, 1992.

[31] A. Goldberg, S. Plotkin, D. Shmoys, and E. Tardos. Using interior-point methods for fast parallel algorithms for bipartite matching and related problems. *SIAM J. Comput.*, 21(1):140–150, Feb 1992.

[32] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *29th FOCS*, pages 174–185, 1988.

[33] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *19th STOC*, pages 7–18, 1987.

[34] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. of the Assoc. for Comput. Mach.*, 35(4):921–940, Oct 1988.

[35] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, Dec 1973.

[36] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Proc. Letters*, 22(2):77–80, Jan 1986.

[37] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Inf. Proc. Letters*, 22(2):57–60, Jan 1986.

[38] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problem. *Computing*, 38:325–340, 1987.

[39] T. Kim and K. Y. Chwa. An $O(n \log n \log \log n)$ parallel maximum matching algorithm for bipartite graphs. *Inf. Proc. Letters*, 24(1):15–17, Jan 1987.

[40] D. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley Publishing Company, Inc., Massachusetts, 1973.

[41] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[42] H. W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Res. Logist. Quart.*, 3:253–258, 1956.

[43] C. Lucchesi, I. Simon, I. Simon, J. Simon, and T. Kowaltowski. *Aspectos Teóricos da Computação*. Projeto Euclides, Brazil, 1979.

[44] U. Manber. *Introduction To Algorithms - A Creative Approach*. Addison-Wesley Publishing Company, Inc., Massachusetts, 1989.

[45] S. Micali and V. Vazirani. An $O(\mid V \mid^{0.5} \cdot \mid E \mid)$ algorithm for finding maximum matchings in general graphs. In *21th FOCS*, pages 17–27, 1980.

[46] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. In *19th STOC*, pages 345–354, 1987.

[47] J. Munkres. Algorithms for the assignment and transportation problems. *J. Soc. Indust. Appl. Math.*, 5:32–38, 1957.

[48] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[49] R. E. Tarjan. *Data Structures and Network Algorithms.* Soc. Indust. Appl. Math., Philadelphia, 1983.

[50] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, April 1985.

[51] R. Terada. *Introdução à Complexidade de Algoritmos Paralelos.* IME-USP, São Paulo, 1990.

[52] G. Thomas. *Calculus And Analytic Geometry.* Addison-Wesley Publishing Company, Inc., Massachusetts, 1962.

Índice

T

V