Algoritmos para Emparelhamento em Grafos e uma Implementação Paralela

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Carlos Fernando Bella Cruz e aprovada pela Comissão Julgadora.

Campinas, 17 de abril de 1996.

Prof. João Carlos Setubal
Orientador

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.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Cruz, Carlos Fernando Bella

C889a Algoritmos para emparelhamento em grafos e uma implementação paralela / Carlos Fernando Bella Cruz -- Campinas, [S.P. :s.n.], 1996.

Orientador : João Carlos Setubal

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

1. Teoria dos grafos. 2. Otimização combinatória. 3. Algoritmos paralelos. 4. *Emparelhamento em grafos. I. Setubal, João Carlos. II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Tese de Mestrado defendida e aprovada em 17 de abril de 1996

pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a). ROUTO TERADA

Prof (a). Dr (a). RICARDO DAHAB

Prof (a). Dr (a). JOÃO CARLOS SETUBAL

Algoritmos para Emparelhamento em Grafos e uma Implementação Paralela[1]

Carlos Fernando Bella Cruz[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Routo Terada (IME/USP)
- Ricardo Dahab (DCC/IMECC/UNICAMP)
- João Carlos Setubal (Orientador)
- Cid Carvalho de Souza (Suplente, DCC/IMECC/UNICAMP)

[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. Este trabalho foi financiado, em parte, por auxílios do CNPq e da FAPESP.

[2]O autor é Bacharel em Informática pela Universidade Federal do Paraná.

Aos meus pais Oswaldo e Vera.

Agradecimentos

Os meus sinceros agradecimentos ao professor João Carlos Setubal pelo excelente convívio e principalmente pela liberdade no desenvolvimento deste trabalho, sem a qual muitas idéias não teriam surgido.

Ao CNPq (1994) processo número 131770/94 e à FAPESP (1995) processo número 94/4158-5 pelas bolsas de mestrado.

Agradeço aos colegas Andrey, Edson e João, que se tornaram minha segunda família durante a minha estadia em Campinas.

Resumo

Abordamos os principais algoritmos para o problema de emparelhamento máximo em grafos genéricos e desenvolvemos uma implementação paralela eficiente na prática, baseada no algoritmo seqüencial de Edmonds. Por prática entendemos uma implementação eficiente num multiprocessador de memória compartilhada. A implementação consiste em permitir que cada processador procure caminhos aumentantes no grafo de forma assíncrona e independente dos demais. Embora a busca ocorra de forma paralela, o aumento do emparelhamento é feito por somente 1 processador por vez, o que garante a corretude do algoritmo sem incorrrer em atraso significativo no tempo de execução. O desenvolvimento da implementação teve como antecedente uma experiência negativa de paralelização baseada no algoritmo de Micali e Vazirani.

Abstract

In this work we present the most important matching algorithms for general graphs and develop an efficient parallel implementation in practice based on Edmonds'matching algorithm. By practice we mean an efficient implementation on a shared memory multiprocessor. The implementation allows each processor to find augmenting paths assinchronously and independently of each other. Each matching augmentation is done by only one processor, and this makes the algorithm correct without causing significant delay in the execution time, in practice. The development of this implementation was made after a negative experience of paralelization based on the sequential algorithm of Micali and Vazirani.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O problema do emparelhamento em grafos pode ser definido da seguinte forma: dado um grafo não orientado, queremos encontrar um subconjunto M das arestas tal que nenhuma aresta de M tenha um vértice em comum com outra aresta de M, e a cardinalidade de M seja máxima. Informalmente, trata-se de conseguir "casar" o máximo número de pares de vértices adjacentes do grafo.

Este é um problema clássico que ocupa um lugar especial em otimização combinatória. Além de encontrar diversas aplicações em ciência da computação e pesquisa operacional, a busca de algoritmos eficientes para resolvê-lo contribuiu para lançar os fundamentos da moderna teoria da complexidade de algoritmos, graças ao trabalho de Edmonds [Edm65].

Este trabalho tem dois temas. Por um lado, fazemos um apanhado geral de algoritmos seqüenciais e paralelos para o problema do emparelhamento; por outro lado, apresentamos uma adaptação dos algoritmos existentes que resultou numa implementação paralela com bom desempenho na prática. Consideramos esta a principal contribuição deste trabalho. O desenvolvimento de uma boa implementação paralela é particularmente importante tendo em vista que os algoritmos paralelos existentes na literatura não são adequados para implementação em multiprocessadores atuais.

O trabalho está estruturado da seguinte forma.

No capítulo 2 apresentamos as definições que usamos ao longo do trabalho e a descrição dos principais algoritmos seqüenciais. Dentre estes, nosso foco é sobre o algoritmo de Micali e Vazirani [MV80], o mais rápido que se conhece atualmente.

No capítulo 3 descrevemos o algoritmo paralelo de Mulmuley, Vazirani e Vazirani [MVV87], e mostramos porque sua implementação seria problemática.

No capítulo 4 descrevemos nossa implementação paralela. Inicialmente mostramos as dificuldades que encontramos ao tentar obter uma implementação paralela baseada no algoritmo de Micali e Vazirani. Em seguida, descrevemos nossa nova abordagem, que pode ser vista como híbrida entre os algoritmos de Edmonds [Edm65] e de Micali e Vazirani.

No capítulo 5 apresentamos os resultados computacionais de nossa implementação obtidos numa SPARC Server 1000 com 8 processadores. Estes resultados são comparados a duas implementações seqüenciais: uma implementação do algoritmo de Micali e Vazirani escrita por Mattingly e Ritchey [MR93]; e uma implementação do algoritmo de Gabow [Gab76] escrita por Edward Rothberg [MR93].

Capítulo 2

Emparelhamento Seqüencial

2.1 Definições

Um grafo G é dado por um conjunto finito V de vértices e por um conjunto finito E de arestas, e o denotamos por $G = (V, E)$. Cada aresta é um par de vértices (v, w) e dizemos que ela é *incidente* sobre os vértices v e w.

Um *caminho* no grafo G é dado por uma lista de vértices $(v_1, v_2, ..., v_n)$ onde $n \geq 2$ e (v_i, v_{i+1}) é uma aresta para $1 \leq i < n$. Um caminho é *simples* se nenhum vértice ocorre mais de uma vez na lista. Um *circuito* é um caminho $(v_1, v_2, ..., v_n, v_1)$ tal que $n \geq 3$ e $(v_1, v_2, ..., v_n)$ é um caminho simples.

Seja $G = (V, E)$ um grafo com n vértices e m arestas. Um *emparelhamento* M é um subconjunto das arestas tal que não existem duas arestas de M que incidem no mesmo vértice. Um emparelhamento M é *maximal* se não existe um emparelhamento M' tal que $M \subset M'$. Um emparelhamento é *máximo* se sua cardinalidade for máxima e perfeito se as arestas de M emparelham todos os vértices de V.

Dado um emparelhamento M, um vértice é considerado *livre* se nenhuma das arestas nele incidentes pertence a M. Um *caminho alternado* é um caminho simples cujas arestas se alternam entre M e $E - M$. Um *caminho aumentante* é um caminho alternado entre dois vértices livres. A partir de um emparelhamento M e de um caminho aumentante P podemos obter um emparelhamento M', fazendo $M' = M \oplus P$, onde $M \oplus P = (M \cup P) - (P \cap M)$ é a diferença simétrica entre M e P. O caminho P se chama aumentante porque $|M'| = |M| + 1$.

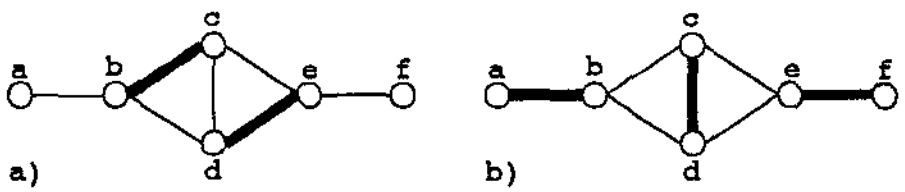

Figura 2.1: a) Emparelhamento. b) Emparelhamento máximo.

Na figura 2.1a é mostrado um grafo com um emparelhamento de cardinalidade 2 (as arestas emparelhadas são representadas por linhas grossas). Os vértices a e f são vértices livres e são os extremos do caminho aumentante (a, b, c, d, e, f). Trocando-se as arestas deste caminho obtemos o emparelhamento da figura 2.1b.

2.2 Histórico

O primeiro resultado relevante para esta dissertação foi obtido em 1957, quando Berge [Ber57], e independentemente, Norman e Rabin [NR59], provaram o seguinte teorema:

Teorema 1 *Um emparelhamento M é máximo se e somente se ele não admite caminho aumentante.*

Este resultado conduz a um esquema para encontrar o emparelhamento máximo. Ele consiste na procura sucessiva de caminhos aumentantes e na subseqüente troca de suas arestas. Entretanto o teorema não dá indicações de como se deve proceder de forma eficiente para procurar caminhos aumentantes. Somente alguns anos depois Edmonds [Edm65] encontrou a solução, propondo um algoritmo de tempo de execução $O(n^4)$. Nesse trabalho foi introduzido o conceito de *flor* (um circuito de comprimento ímpar) e como tratá-la para que seja possível encontrar caminhos aumentantes eficientemente. Vale a pena lembrar que este trabalho também é um marco na história da teoria da computação, pois nele Edmonds sugeriu a definição de um "bom" algoritmo como sendo um algoritmo de tempo de execução polinomial. A seção 2.3 abaixo descreve o algoritmo de Edmonds em detalhe.

Tendo como base o trabalho de Edmonds, vários outros algoritmos, com melhorias na complexidade, foram propostos. Gabow [Gab76], forneceu um algoritmo com complexidade $O(n^3)$. Em seguida Even e Kariv [EK75] apresentaram um algoritmo com tempo $O(n^{2.5})$. Micali e Vazirani [MV80] forneceram um algoritmo com complexidade $O(\sqrt{n}\ m)$. Atualmente este é o algoritmo mais rápido conhecido. Ele se encontra descrito em detalhes na seção 2.5 abaixo.

Um novo enfoque para o algoritmo de Micali e Vazirani foi apresentado por Blum [Blu94]. Este algoritmo não utiliza explicitamente o conceito de caminho aumentante, porém possui a mesma complexidade de tempo. Uma breve descrição deste algoritmo está na seção 2.6.

Na computação paralela, Mulmuley, Vazirani e Vazirani (MVV)[MVV87] apresentaram um algoritmo randomizado para o modelo PRAM baseado em inversão de matrizes, com tempo de execução $O(\log^2 n)$. Este algoritmo se encontra descrito no capítulo 3.

2.3 Algoritmo de Edmonds

Na descrição que se segue fizemos uso do capítulo 12 do livro de Ahuja, Magnanti e Orlin [AMO93]. Para descrevermos o algoritmo de Edmonds utilizaremos a definição de grafos *bipartidos*.

Definição 1 *Um grafo G é bipartido se os seus vértices podem ser particionados em dois conjuntos distintos e não vazios, A e B, tal que todas as arestas do grafo ligam vértices de A a vértices de B. Denotamos um grafo bipartido por $G = (A, B, E)$.*

A estrutura bem definida dos grafos bipartidos fornece algumas propriedades que tornam um algoritmo de emparelhamento máximo para grafos bipartidos menos complexo do que um para grafos genéricos. Portanto, para descrevermos o algoritmo de Edmonds, faremos uma breve explanação sobre o funcionamento de um algoritmo para emparelhamento máximo em grafos bipartidos. Em seguida, descreveremos as modificações que deverão ser feitas neste algoritmo para que seja possível encontrar o emparelhamento máximo em grafos genéricos.

2.3.1 Algoritmo para emparelhamento em grafos bipartidos

Numa visão geral, o algoritmo executa uma busca no grafo bipartido $G = (A, B, E)$. Ela parte de um vértice livre r. O objetivo é encontrar um outro vértice livre q para formar um caminho aumentante. Se ele for encontrado, então as arestas deste caminho são trocadas para aumentar a cardinalidade do emparelhamento. No entanto, se a busca percorreu todo o grafo sem encontrar nenhum outro vértice livre, então significa que não existe caminho aumentante partindo do vértice livre r. Neste caso, não se pode aumentar o emparelhamento a partir de r. Ele então é marcado como *apagado*. O algoritmo prossegue escolhendo um outro vértice livre s que não esteja apagado para executar uma nova busca. Este processo se repete até que não haja mais vértices livres não apagados. Neste ponto o emparelhamento é máximo.

Em cada passo, o algoritmo reduz o número de vértices livres não apagados em pelo menos um. Isto é feito apagando-se um vértice livre, caso a busca não obtenha sucesso; ou aumentando a cardinalidade do emparelhamento, caso seja encontrado um caminho aumentante.

O que resta para completar a explicação do algoritmo é como executar a busca no grafo. Um método utilizado é crescer uma árvore de busca em largura a partir do vértice livre r. Os vértices visitados pela busca recebem um rótulo *par* ou *ímpar*. Um vértice visitado v recebe rótulo par se a distância em número de arestas entre v e r for par e recebe rótulo ímpar se a distância entre v e r for de comprimento ímpar. Sempre que um vértice livre $q \neq r$ recebe um rótulo ímpar então foi identificado um caminho aumentante numa busca a partir de r.

Para coordenar o crescimento da busca o algoritmo mantém um lista R de vértices que são processados em fila para garantir que a busca seja feita em largura. No início da busca, o único vértice que pertence a R é a raiz da árvore r, a qual possui rótulo par. A expansão da busca é feita retirando-se um vértice v da lista R. Se o vértice v possui um rótulo par, então para cada vértice u adjacente a v, tal que a aresta (v,u) não está emparelhada e que o vértice u não tenha sido visitado, é feita a seguinte verificação. Se o vértice u é um vértice livre, então foi identificado um caminho aumentante entre os vértices r e u. No entanto, se o vértice u, adjacente a v, não estiver livre, então a busca deve ser expandida para ele. Isto é feito marcando o vértice v como *predecessor* de u, rotulando u com ímpar e inserindo-o na lista R. Agora se o vértice v, retirado da lista R, tiver rótulo ímpar, então a única ação a ser feita é expandir a busca através da aresta emparelhada (v,u). Isto é feito marcando o vértice v como predecessor de u, atribuindo um rótulo par ao vértice u e inserindo-o na lista R. A busca é expandida enquanto houver vértices na lista R ou enquanto não se identificar um caminho aumentante.

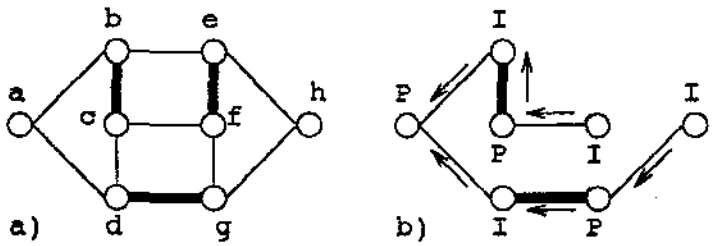

Figura 2.2: a) Grafo bipartido. b) Árvore de busca.

A figura 2.2b ilustra o crescimento da busca em largura sobre o grafo bipartido $G = (A, B, E)$ da figura 2.2a. Os vértices $\{a, c, e, g\}$ pertencem ao conjunto A, os vértices $\{b, d, f, h\}$ pertencem ao conjunto B e as arestas ligam somente vértices do conjunto A com os do conjunto B. Os vértices com rótulo par são representados pela letra **P**, os com rótulo ímpar pela letra **I** e as setas representam as marcações de predecessores. A raiz da árvore de busca é o vértice a. A partir dele a busca se expande para os vértices b e d, pois nenhum deles é vértice livre. Como os vértices b e d possuem rótulo ímpar, a única ação a ser feita é expandir a busca para os vértices c e g respectivamente. A partir do vértice c, a busca somente poderá ser expandida para o vértice f, pois o vértice d já foi visitado. A partir do vértice g é atribuído um rótulo ímpar ao vértice livre h, identificando o caminho aumentante (a, d, g, h). Este caminho é estabelecido pelas marcações de predecessores a partir de h até a raiz a.

A garantia de que o algoritmo acima encontra o emparelhamento máximo reside no fato de que grafos bipartidos possuem a propriedade de rotulagem única dos vértices, ou seja, nunca é o caso de haver ambigüidade na atribuição de um rótulo a um vértice.

Definição 2 *Definimos* ambigüidade *como sendo a possibilidade de um vértice receber um rótulo par e também um rótulo ímpar durante uma busca.*

Em grafos bipartidos se a raiz da árvore de busca estiver no conjunto A então todos os vértices de A receberão rótulos par e todos os vértices de B receberão ímpar. Similarmente, se a raiz estiver em B, os vértices de A receberão rótulos ímpar e os de B receberão par. Isto acontece porque a expansão dos ramos da busca se alternam entre os elementos do conjunto A e do conjunto B.

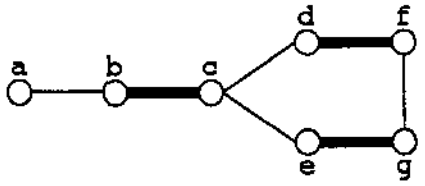

Figura 2.3: Grafo genérico.

Infelizmente, grafos genéricos não possuem a propriedade de rotulagem única dos vértices. A figura 2.3 mostra um caso em que o vértice f pode receber um rótulo par e também um rótulo ímpar. O vértice f pode receber um rótulo par através do ramo da busca (a, b, c, d, f) e pode recebe um rótulo ímpar através do ramo da busca (a, b, c, e, g, f). Esta situação ocorre porque podemos ligar o vértice f à raiz a através de um caminho de comprimento ímpar e por um caminho de comprimento par.

Edmonds [Edm65], em 1965, mostrou como lidar com a ambigüidade nos vértices, introduzindo o conceito de *flor*, que descreveremos na próxima seção.

2.3.2 Modificação no algoritmo para emparelhamento bipartido

Dado um emparelhamento num grafo, definimos uma **flor** como sendo um circuito de comprimento ímpar conectado a uma estrutura chamada *haste* através de um vértice chamado *base* da flor. Formalmente:

Definição 3 *Uma haste é um caminho alternado de comprimento par que parte da raiz da busca em largura r e vai até um vértice w. É permitido que r seja igual a w, onde dizemos que a haste é vazia.*

Definição 4 *Uma flor (blossom em Inglês) é um circuito de comprimento ímpar que começa e termina no vértice w da haste e não tem nenhum outro vértice em comum com ela. O vértice w é a base da flor.*

A figura 2.4a representa uma flor com uma haste vazia e a figura 2.4b representa um flor com haste não vazia.

Uma flor e sua haste possuem três propriedades importantes:

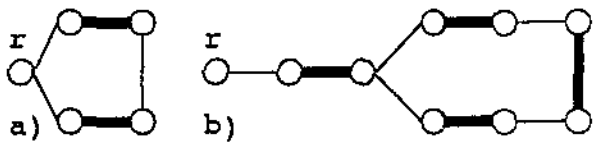

Figura 2.4: a) Flor com haste vazia b) Flor com haste.

1. Uma haste possui $2i + 1$ vértices e contém i arestas emparelhadas para $i \geq 0$.

2. Uma flor possui $2j + 1$ vértices e contém j arestas emparelhadas para $j \geq 1$ e as arestas emparelhadas emparelham todos os vértices da flor exceto a base, a qual é um vértice com rótulo par.

3. Pelo fato de uma flor ser um circuito ímpar, todos os seus vértices alcançam a haste por dois caminhos, um deles de comprimento par e outro de comprimento ímpar.

Estas propriedades nos permitem tirar as seguintes conclusões: uma aresta que não pertence a uma flor, mas que incide num de seus vértices é uma aresta não emparelhada e os caminhos aumentantes que passam por uma tal aresta alcançam a base. Portanto, uma flor não constitui um obstáculo para um caminho que passe através dela, o que conduz à idéia de contrair a flor em um pseudo-vértice sobre a sua base. As arestas que antes eram incidentes sobre vértices da flor agora são incidentes sobre esse pseudo-vértice.

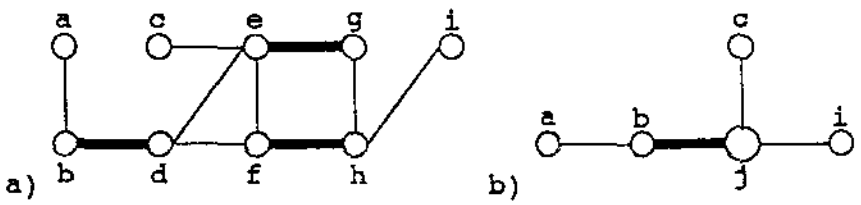

Figura 2.5: a) Identificação de uma flor b) Contração da flor.

Na figura 2.5 observamos a identificação e contração de uma flor. A raiz da árvore de busca é o vértice a. Um ramo da busca segue pelos vértices (a, b, d, e, g) e outro por (a, b, d, f, h, g). O vértice g pode receber tanto rótulo par quanto ímpar. Isto caracteriza a presença de uma flor. Os vértices que a compõem são: $\{d, e, f, g, h\}$. A sua base é o vértice d e a haste é o caminho (a, b, d). A flor é contraída sobre a sua base criando um pseudo-vértice j. O resultado da contração está representado na figura 2.5b.

A modificação feita no algoritmo para emparelhamento bipartido consiste em verificar se, durante a busca a partir de um vértice livre, ocorre ambigüidade no rotulamento de um vértice v. Quando isto ocorrer, a busca é suspensa e são disparadas duas buscas em profundidade a partir dos dois caminhos que chegam

no vértice v. Estas duas buscas seguirão as marcações de predecessores e têm como objetivo, encontrar o primeiro vértice em comum entre elas. Este vértice é a base da flor e os dois caminhos percorridos contém os vértices da flor. O algoritmo contrai a flor sobre a sua base criando um pseudo-vértice, o qual recebe rótulo par e é inserido na fila R para prosseguir a expansão da busca.

Voltando ao exemplo da figura 2.5a um ramo da busca em largura segue pelo caminho (a, b, d, e, g), atribuindo um rótulo par ao vértice g, e um ramo segue pelo caminho (a, b, d, f, h), atribuindo um rótulo par ao vértice h. A partir do vértice h é possível atribuir um rótulo ímpar ao vértice visitado g, o qual já possui um rótulo par. Isto caracteriza uma ambigüidade e portanto implica na presença de uma flor. Neste instante são disparadas duas buscas em profundidade. Uma segue o caminho (g, e, d) e a outra segue (g, h, f, d). O vértice em comum entre elas, d, é a base da flor e os vértices $\{d, e, f, g, h\}$ são os vértices que a compõem. Esta estrutura é contraída num pseudo-vértice como na figura 2.5b.

Várias contrações poderão ocorrer até que seja encontrado um caminho aumentante. Inclusive é possível existir flor dentro de flor. Quando um caminho aumentante é identificado, o algoritmo segue as marcações de predecessores para verificar se existem pseudo-vértices no caminho. Se existir então ele é expandido e é verificado se o caminho até a base segue pelo caminho de comprimento par ou de comprimento ímpar. Isto é feito observando a aresta seguinte de cada um dos dois caminhos, pois somente um deixará o caminho aumentante consistente. Quando todos os pseudo-vértices existentes no caminho forem expandidos, então as suas arestas são trocadas para aumentar a cardinalidade do emparelhamento.

Na figura 2.5b foi identificado o caminho aumentante (a, b, j, c). Seguindo as marcações de predecessores, descobrimos que o vértice j é, na realidade, um pseudo-vértice. Neste ponto, a flor é expandida para a sua forma original, como mostra a figura 2.5a. Então é observado que o caminho segue pelo arco de comprimento par até a base d da flor. Portanto, o caminho aumentante é (a, b, d, f, h, g, e, c).

Um resumo do algoritmo de Edmonds em forma de pseudo-código está descrito abaixo.

ALGORITMO DE EDMONDS

Para cada vértice livre r do grafo faça:
    Retire os rótulos de todos os vértices do grafo
    Faça $r \leftarrow$ Par e $R \leftarrow \{r\}$
    Enquanto $R \neq \emptyset$
        Retire um vértice v de R
        Se v for Par
            Resultado $\leftarrow$ **Examina_Vértice_Par** (v)
            Se Resultado = Encontrou_Caminho_Aumentante
                Aumenta_Caminho (q,r)
        Senão "v é Ímpar"
            Examina_Vértice_Ímpar (v)
Fim do Algoritmo

Sub-rotina: Examina_Vértice_Par (v)

Para cada vértice q adjacente a v faça:
    Se q é um vértice livre
        Predecessor $(q) \leftarrow v$
        Resultado $\leftarrow$ Encontrou_Caminho_Aumentante
        Retorne
    Se q for Par
        Siga os predecessores de q e v para identificar
        os vértices da flor e criar o pseudo-vértice b
        Faça $b \leftarrow$ Par e adicione b a R
    Senão "q não possui rótulo"
        Faça predecessor$(q) \leftarrow v$, $q \leftarrow$Ímpar e adicione q a R
    Resultado $\leftarrow$ Não_Encontrou_Caminho_Aumentante
Fim da sub-rotina

Sub-rotina: Examina_Vértice_Ímpar (v)

Seja (v,q) a aresta emparelhada
Se q for Ímpar
    Siga os predecessores de q e v para identificar
    os vértices da flor e criar o pseudo-vértice b
    Faça $b \leftarrow$ Par e adicione b a R
    Retorne
Senão "q não possui rótulo"
    Faça predecessor $(q) \leftarrow v$, $q \leftarrow$ Par e adicione q a R
Fim da sub-rotina

Sub-rotina: Aumenta_Caminho (q, r)

Percorra o caminho aumentante P entre q e r
seguindo as marcações de predecessores
Se P contém um pseudo-vértice b
    Expanda b para a sua forma anterior e obtenha
    o caminho aumentante no grafo original
Atualize o emparelhamento fazendo $M \leftarrow M \oplus P$
Fim da sub-rotina

2.3.3 Complexidade computacional

A complexidade do algoritmo de Edmonds, como inicialmente foi proposto, é $O(n^4)$. Porém, esta complexidade pode ser melhorada com uma manipulação mais eficiente da estrutura de dados. Esta estrutura de dados é composta por um vetor onde cada elemento possui uma lista ligada. O vetor representa os vértices do grafo e os elementos das listas ligadas representam as arestas incidentes sobre os vértices. Definimos $A(v)$ como sendo a lista de adjacência do vértice v, representada pelos nodos de sua lista ligada.

Um resultado que deve ser observado nesta seção é apresentado através do seguinte lema:

Lema 1 *Durante a execução de uma busca o algoritmo executa no máximo $n/2$ contrações.*

Demonstração: Uma flor possui no mínimo três vértices que contraídos constituem um pseudo-vértice. Portanto, cada contração reduz o número de vértices em pelo menos 2. Tendo um grafo n vértices, é possível executar no máximo $n/2$ contrações durante uma busca.□

O algoritmo, ao identificar uma flor declara os vértices que a compõem como sendo *inativos*, assim eles não precisam ser examinados novamente nesta mesma busca. Estes vértices são contraídos formando um pseudo-vértice, o qual é inserido na lista de adjacência. Como, durante uma busca, é possível contrair no máximo $n/2$ flores, a lista de adjacência não cresce mais do que $3n/2$.

Considerando a complexidade do algoritmo quando não são executadas contrações ou expansões de flores, observamos que para cada vértice i o algoritmo executa uma das seguintes operações abaixo no máximo uma vez:

1. Descobrir que o vértice i é inativo. Portanto, ele é desconsiderado.
2. Expandir a busca por um vértice ímpar. Neste caso, por hipótese, a única opção é expandir a busca pela aresta emparelhada.
3. Expandir a busca por um vértice par. As duas únicas opções possíveis são: ou expandir a busca para todos os vértices adjacentes a i, com exceção do seu predecessor ou identificar um caminho aumentante.

Os dois primeiros passos requerem claramente $O(1)$. O terceiro passo é proporcional ao tamanho da lista de adjacência de i, a qual tem tamanho $|A(i)| \leq 3n/2$. Portanto, a complexidade do algoritmo, ignorando contrações e expansões de flores, é $O(n^2)$.

Para contrair uma flor, primeiramente declaramos todos os vértices como *não marcados*. Em seguida, visitam-se os vértices i da flor e, para cada i, marcam-se os vértices pertencentes a $A(i)$. Numa etapa seguinte, examinam-se os vértices do grafo e, os que estiverem marcados são exatamente os vértices adjacentes ao pseudo-vértice f que está sendo criado. Estes vértices marcados são colocados em $A(f)$ e f é adicionado à lista de adjacência de cada um deles, completando a contração da flor. Uma contração de flor exige $O(n)$ visitas a vértices, pois é necessário essencialmente visitar todos os vértices do grafo para saber quais são adjacentes à flor e quais não são. Como numa busca há no máximo $n/2$ contrações, fazemos no máximo $O(n^2)$ visitas a vértices. Precisamos também contabilizar as consultas às arestas (varredura da lista de adjacência). Aqui observamos que um vértice pertence a uma flor somente 1 vez, portanto numa busca cada aresta é consultada somente 1 vez. Disso resulta que numa busca são consultadas $O(n^2)$ arestas no pior caso. Combinando visitas a vértices com consultas às arestas vemos que numa busca o tempo total gasto é $O(n^2)$.

Para observar a complexidade na expansão de flores, analisaremos um caminho aumentante entre dois vértices: p e q. O algoritmo começa em q e segue as marcações de predecessores até que o vértice p seja encontrado. Durante este processo, pseudo-vértices podem ser encontrados, neste caso eles devem ser expandidos para a sua forma anterior. Vamos supor que j seja o primeiro pseudo-vértice encontrado e que j seja predecessor de um vértice i. Então, por suposição, i não é um pseudo-vértice. Além do mais, i é adjacente a algum vértice k da flor J, a qual é representada pelo pseudo-vértice j. Para expandir o pseudo-vértice j, declaramos todos os vértices do grafo como não marcados. Em seguida, marcam-se os vértices da flor F. A lista de adjacência de i é percorrida até que o vértice marcado k seja encontrado. O vértice k é adjacente a dois ou mais vértices da flor, mas somente um vértice está ligado a ele por uma aresta emparelhada. O caminho até a base segue através desta aresta emparelhada e a expansão do pseudo-vértice está completada. É importante notar que durante a expansão de um pseudo-vértice é possível encontrar um outro pseudo-vértice. Neste caso aplicamos o mesmo método recursivamente. A complexidade da expansão de um pseudo-vértice é $O(n)$. Como é possível haver no máximo $n/2$ pseudo-vértices, a expansão de um caminho aumentante tem complexidade $O(n^2)$.

Através da aplicação destes métodos observa-se que a expansão de uma busca junto com a contração das flores encontradas requer $O(n^2)$. O tratamento de um caminho aumentante também requer $O(n^2)$. Portanto, o tempo total de uma fase é $O(n^2)$. Como são feitas no máximo $O(n)$ fases obtemos o seguinte teorema:

Teorema 2 *O emparelhamento máximo num grafo pode ser encontrado em* $O(n^3)$.

Observamos que este algoritmo foi originalmente apresentado [Edm65] como tendo tempo de execução $O(n^4)$. O tempo $O(n^3)$ acima provém de uma análise mais cuidadosa, tal como aparece em [AMO93].

2.4 Algoritmo de Gabow

Tendo como base o algoritmo de Edmonds, vários outros algoritmos, com melhorias na complexidade, foram propostos. Um destes algoritmos, foi o de Gabow. Um do principais avanços alcançados foi um novo método de rotulagem dos vértices, o qual evita a criação de pseudo-vértices.

De uma maneira geral, o algoritmo de Gabow segue os mesmos passos do algoritmo de Edmonds. Ele também constrói buscas em largura a partir de vértices livres com o objetivo de encontrar caminhos aumentantes. As únicas diferenças entre os dois algoritmos são: a maneira como os vértices são rotulados pela busca e como são identificadas as arestas de caminhos aumentantes que passam ou não por flores.

No algoritmo de Gabow, os vértices do grafo são numerados de 1 a n e as arestas de $n+1$ a $n+m$. Estes valores são os rótulos que são atribuídos aos

vértices para definir caminhos aumentantes. Os rótulos cujos valores são entre 1 e n fazem o papel semelhante ao de predecessor indicando por onde a busca foi expandida. Os valores entre $n+1$ e $n+m$ auxiliam no processo de encontrar caminhos através de flores.

O algoritmo utiliza como estruturas de dados um vetor C de tamanho n cujos elementos são vértices. Ela é responsável pelo armazenamento do emparelhamento. Uma aresta (u, v) está emparelhada se e somente se $C(u) = v$ e $C(v) = u$. Também é usado um vetor M de tamanho m cujos elementos são pares de vértices. Através do número x de uma aresta é possível saber quais os dois vértices sobre os quais ela incide, $M(x) = (u, w)$.

Um vértice v é considerado *externo* se existe um caminho alternado de comprimento par entre v e a raiz da árvore de busca r. Supondo que $P(v) = (v, v_1, v_2, ..., r)$ seja este caminho, então a aresta (v, v_1) está emparelhada. Se uma aresta (q, v) unindo um vértice livre $q \neq r$ a um vértice externo v, for encontrada, então um caminho aumentante $(q, P(v)) = (q, v, v_1, v_2, ..., r)$ foi identificado.

Para definir as arestas que compõem um caminho aumentante utiliza-se uma rotina recursiva. Ela trata vértices com rótulos do tipo *vértice* (valores de 1 até n) de maneira distinta dos vértices com rótulos do tipo *aresta* (valores de $n+1$ até $n+m$). Se um vértice v possui rótulo do tipo vértice então a função que define o caminho até a raiz da árvore da busca é: $P(v) = (v, C(v), P(rótulo(v)))$, onde a aresta $(v, C(v))$ é a aresta emparelhada e $rótulo(v)$ é o valor do rótulo do vértice v. Se um vértice v possui rótulo do tipo aresta então $M(rótulo(v))$ é uma aresta que liga dois vértices externos. Supondo que $M(rótulo(v)) = (u, w)$ então o caminho $P(v)$ é definido em termos de $P(u)$ e $P(w)$. Se o vértice v pertence ao caminho $P(w)$ então a função que define o caminho é: $P(v) = (reverso(P(w, v)), P(u))$, onde $P(w, v)$ define a porção do caminho de $P(w)$ que está entre w e v e a função $reverso(j)$ inverte a ordem dos vértices do caminho j. Agora se v pertence ao caminho $P(u)$ então a função que define o caminho é: $P(v) = (reverso(P(u, v)), P(w))$.

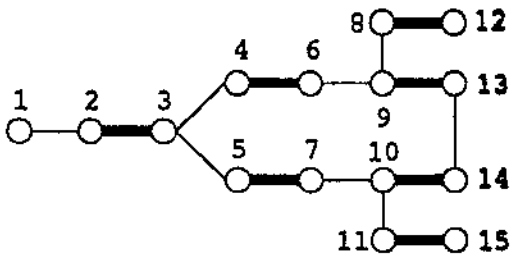

Figura 2.6: Caminhos alternados no algoritmo de Gabow.

Para exemplificar a identificação das arestas que compõem um caminho alternado até a raiz da árvore de busca, utilizaremos o grafo da figura 2.6, cujos vértices possuem rótulos representados na tabela 2.1 (nesta tabela, os rótulos do tipo aresta foram substituídos diretamente pelas respectivas arestas).

v	C(v)	tipo do rótulo	valor do rotulo
1	—	início	—
2	3	—	—
3	2	vértice	1
4	6	aresta	(13,14)
5	7	aresta	(13,14)
6	4	vértice	3
7	5	vértice	3
8	12	—	—
9	13	aresta	(13,14)
10	14	aresta	(13,14)
11	15	—	—
12	8	vértice	9
13	9	vértice	6
14	10	vértice	7
15	11	vértice	10

Tabela 2.1: Rotulagem dos vértices no algoritmo de Gabow.

Na figura 2.6 o vértice 1 é a raiz da busca e recebe um rótulo *início* indicando que a busca foi iniciada a partir dele. O vértice 7 possui rótulo do tipo vértice com valor 3 e o vértice 3 também possui rótulo do tipo vértice, com conteúdo 1. Portanto, o caminho $P(7)$ é definido pela seguinte recursão:

$$P(7) = (7, C(7), P(\textit{rótulo}(7))) = (7, 5, P(3))$$
$$P(3) = (3, C(3), P(\textit{rótulo}(3))) = (3, 2, 1)$$
$$P(7) = (7, 5, 3, 2, 1)$$

Na figura 2.6 o vértice 9 possui rótulo do tipo aresta com o valor da aresta $(13, 14)$. O vértice 9 faz parte do caminho $P(13)$. Seja $P(13, 9)$ a porção do caminho $P(13)$ entre o vértice 13 e o 9. O caminho $P(9)$ é definido por:

$$P(9) = (\textit{reverso}(P(13, 9), P(14)) = (9, 13, 14, C(14), P(7)) =$$
$$(9, 13, 14, 10, 7, 5, 3, 2, 1)$$

Em seguida apresentaremos o algoritmo de Gabow no formato de pseudo-código. O algoritmo é composto de uma rotina principal e mais três sub-rotinas: Busca_Caminho_Aumentante, Trata_Flor e Trata_Caminho_Aumentante. A descrição destas sub-rotinas utiliza um vetor chamado de *primeiro*. Para cada vértice v, *primeiro*(v) representa o primeiro vértice não externo no caminho $P(v)$.

ALGORITMO DE GABOW

A_1 Numerar os vértice de 1 até n e as arestas de $n+1$ até $n+m$
A_2 Para u de 1 até n faça:
A_3 Se u é um vértice livre
A_4 $rótulo(i) \leftarrow Nulo$ para $0 \leq i \leq n$
A_5 $rótulo(u) \leftarrow primeiro(u) \leftarrow 0$
A_6 Busca_Caminho_Aumentante(u)
Fim do Algoritmo

No passo A_4 os rótulos dos vértices são inicializados com *Nulo* (valor -1) e no passo A_5 o vértice livre u é marcado como sendo o único vértice externo do grafo, pois supõe-se que um vértice v é externo se $rótulo(v) \geq 0$.

Sub-rotina: Busca_Caminho_Aumentante(u)

B_1 Para cada aresta (x, y) onde x é um vértice externo faça:
B_2 Se y é um vértice livre e $y \neq u$
B_3 $C(y) \leftarrow x$
B_4 Trata_Caminho_Aumentante(x, y)
B_5 Retorne
B_6 Se y é um vértice externo
B_7 Trata_Flor(x, y)
B_8 $próximo \leftarrow C(y)$
B_9 Se *próximo* não é um vértice externo
B_{10} $rótulo(próximo) \leftarrow x$
B_{11} $primeiro(próximo) \leftarrow y$
Fim da sub-rotina

Os vértices externos podem ser escolhidos em qualquer ordem no passo B_1. Uma ordem possível é em largura, ou seja, escolhe-se um vértice $x = x_1$ e visitam-se as arestas (x, y). O próximo vértice a ser escolhido será o vértice x_2 que foi rotulado logo após o vértice x_1. O processo se repete para $x = x_2$.

A verificação da ocorrência de caminho aumentante é feita no passo B_2. Se existir tal caminho então ele é composto pelo vértice livre y recém descoberto concatenado com caminho alternado de comprimento par $P(x)$. A troca das

arestas do caminho $(y, P(x))$ é iniciada no passo B_3 e completada pela sub-rotina Trata_Caminho_Aumentante.

No passo B_6, se o vértice y já tiver sido visitado então isto caracteriza um encontro de duas buscas que formam uma flor com a mesma estrutura definida na seção 2.3.2. Definimos a aresta de encontro (x, y) como sendo uma *ponte*. O tratamento desta estrutura é feita pela sub-rotina Trata_Flor.

Se o vértice y estiver emparelhado então o passo B_8 será executado e armazenará na variável *próximo* o possível ponto de expansão da busca. O passo B_9 verifica se o vértice *próximo* já foi visitado. Se ele ainda não foi visitado então isto será feito pelos passos B_{10} e B_{11}.

Sub-rotina: Trata_Flor(x, y)

C_1 *direita* $\leftarrow$ *primeiro*(x)
C_2 *esquerda* $\leftarrow$ *primeiro*(y)
C_3 Se *direita* $=$ *esquerda*
C_4 Retorne
C_5 Marque *esquerda*
C_6 Repita
C_7 Marque *direita*
C_8 Se *esquerda* $\neq 0$
C_9 *direita* $\leftrightarrow$ *esquerda* "troca *direita* com *esquerda*"
C_{10} *direita* $\leftarrow$ *primeiro*(*rótulo*$(C($*direita*$)))$
C_{11} Até que *direita* esteja marcado
C_{12} *encontro* $\leftarrow$ *direita*
C_{13} Para $v \in \{$*primeiro*(x), *primeiro*$(y)\}$
C_{14} Enquanto $v \neq$ *encontro* faça:
C_{15} *rótulo*$(v) \leftarrow$ *valor* onde $M($*valor*$) = (x, y)$
C_{16} *primeiro*$(v) \leftarrow$ *encontro*
C_{17} $v \leftarrow$ *primeiro*(*rótulo*$(C(v)))$
C_{18} Para cada vértice externo i faça:
C_{19} Se *primeiro*(i) é um vértice externo
C_{20} *primeiro*$(i) \leftarrow$ *encontro*
Fim da sub-rotina

A sub-rotina Trata_Flor é responsável pela rotulagem correta dos vértices pertencentes a uma flor. Ela recebe como parâmetro a *ponte* recém descoberta (x, y).

Os passos de C_1 a C_{12} são responsáveis pela identificação do primeiro vértice não externo pertencente aos caminhos $P(x)$ e $P(y)$. Este vértice, chamado de *encontro* delimitará os vértices pertencentes à flor. Os passos seguintes serão

responsáveis pela rotulagem dos vértices da flor.

O passo C_3 verifica um caso em que nenhum vértice pode ser rotulado, pois o encontro dos caminhos $P(x)$ e $P(y)$ é imediato. Os passos de C_5 a C_{12} são responsáveis pelo crescimento de duas buscas: a busca *direita*, que parte de *primeiro*(x), e a *esquerda*, que parte de *primeiro*(y). Os vértices visitados por estas buscas são marcados. Um possível método para marcá-los é atribuir para cada vértice v visitado *rótulo*$(v) = -valor$ sendo $M(valor) = (x, y)$, ou seja, o valor negativo da ponte (x, y). Deste modo, cada chamada da sub-rotina Trata_Flor utilizará uma marcação diferente. O crescimento das duas buscas é alternado através dos passos C_9 e C_{10}. Isto é feito trocando entre si os conteúdos das variáveis *direita* e *esquerda*. O objetivo é fazer com que uma busca seja expandida para um vértice marcado, caracterizando assim um ponto de encontro entre elas. A verificação do passo C_8 evita que uma busca vá além do vértice u, pois *primeiro*$(u) = 0$.

Os passos de C_{13} a C_{17} são responsáveis pela rotulagem dos vértices não externos de *primeiro*(x) até *encontro* e de *primeiro*(y) até *encontro*. Os passos C_{18} a C_{20} rotulam os vértices externos.

Sub-rotina: Trata_Caminho_Aumentante(x, y)

D_1 $t \leftarrow C(x)$
D_2 $C(x) \leftarrow y$
D_3 Se $C(t) \neq x$
D_4 Retorne
D_5 Se x tem rótulo do tipo vértice
D_6 $C(t) \leftarrow$ *rótulo*(x)
D_7 Trata_Caminho_Aumentante(*rótulo*$(x), t)$
D_8 Retorne
D_9 Faça k e l os vértices da aresta $M($*rótulo*$(x)) = (k, l)$
D_{10} Trata_Caminho_Aumentante(k, l)
D_{11} Trata_Caminho_Aumentante(l, k)
Fim da sub-rotina

A sub-rotina recursiva Trata_Caminho_Aumentante é responsável pela troca das arestas do caminho para aumentar a cardinalidade do emparelhamento. Os passos de D_5 a D_8 tratam as partes do caminho que passam por vértices com rótulo do tipo vértice e os passos de D_9 a D_{11} tratam as partes do caminho que passam por vértices do tipo aresta. Se o vértice x possuir rótulo do tipo vértice então a sub-rotina prossegue a troca das arestas através do passo D_7. Se o vértice x possui rótulo do tipo aresta então o caminho segue por uma flor e

passa obrigatoriamente pela ponte (k, l) que a gerou (o caminho através de uma flor e que não passa pela ponte que a gerou seguem pelos vértices com rótulo do tipo vértice). Os passos D_{10} e D_{11} trocam as arestas através da ponte. Mais especificamente, uma chamada parte de um extremo da ponte e encontra o vértice x e a outra prossegue a troca através do outro extremo da ponte, contornando a flor.

Com a utilização eficiente de rótulos o algoritmo de Gabow evita a contração das flores sobre a sua base, tornando desnecessária a atualização da estrutura de dados.

2.5 Algoritmo de Micali e Vazirani

O algoritmo de Micali e Vazirani [MV80] depende de um resultado obtido por Hopcroft e Karp [HK73]. Nele, os autores propõem que caminhos aumentantes sejam encontrados em *fases*. Em cada fase seriam encontrados todos os caminhos aumentantes disjuntos de menor comprimento. Eles provaram que, com esta estratégia, são necessárias somente $O(\sqrt{n})$ fases para se encontrar o emparelhamento máximo. Portanto, esta abordagem apresenta uma complexidade menor do que as $O(n)$ iterações necessárias nos algoritmos anteriores. A contribuição de Micali e Vazirani é um algoritmo que permite executar uma fase em $O(m)$, o que resulta numa complexidade final de $O(\sqrt{n}\, m)$.

De uma maneira geral, o algoritmo realiza simultaneamente buscas em largura a partir de todos os vértices livres do grafo. Os ramos destas buscas crescem até que haja um encontro entre eles. Este encontro ocorre sempre em arestas, as quais denominamos de *pontes*. Cada ponte implica na ocorrência de um caminho aumentante ou de uma *florescência* (uma generalização do conceito de flor). Para se distinguir qual dos dois casos ocorreu, são disparadas duas buscas em profundidade, uma para cada extremo da ponte. Elas tentam encontrar dois vértices livres distintos. Se eles forem encontrados então isto caracteriza a ocorrência de um caminho aumentante. Neste caso as suas arestas são trocadas para aumentar a cardinalidade do emparelhamento. Agora, se apenas um vértice livre for encontrado então isto caracteriza a ocorrência de uma florescência, a qual requer um tratamento especial. Uma nova fase é necessária somente se forem encontrados caminhos aumentantes, caso contrário o emparelhamento é máximo e o algoritmo termina.

Em seguida apresentaremos uma descrição das sub-rotinas principais do algoritmo de Micali e Vazirani. Esta descrição está baseada no artigo de Peterson e Loui [PL88].

2.5.1 Descrição das sub-rotinas principais

Para encontrar todos os caminhos aumentantes disjuntos de menor comprimento numa fase, o algoritmo de Micali e Vazirani utiliza três sub-rotinas principais: busca_pontes, procura_caminho e identifica_arestas.

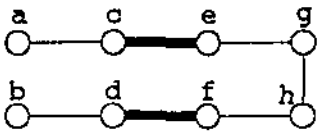

Figura 2.7: A aresta (g, h) é uma ponte.

A sub-rotina busca_pontes é responsável pela construção das buscas em largura a partir dos vértices livres do grafo. Cada um dos ramos da busca é um caminho alternado a partir de um vértice livre. As buscas se expandem até que haja um encontro entre elas. As arestas onde ocorreram os encontros compõem o conjunto de pontes daquele nível de expansão. No exemplo da figura 2.7 as buscas partem dos vértices a e b e se encontram na ponte (g, h). Para cada ponte identificada é chamada a sub-rotina procura_caminho.

A sub-rotina procura_caminho é responsável pela procura de dois vértices livres distintos para formar um caminho aumentante. Para isto, ela executa duas buscas em profundidade, uma a partir de cada vértice da ponte. Na figura 2.7, as duas buscas partem dos vértices g e h. Elas encontrarão simultaneamente os vértices livres a e b. Neste instante, a sub-rotina identifica_arestas é chamada duas vezes. Uma para identificar as arestas que pertencem ao caminho entre os vértices a e g e outra para o caminho entre os vértices b e h. Apesar deste caminho já ter sido percorrido pela sub-rotina procura_caminho, a sub-rotina identifica_arestas é necessária, pois existem casos que existirão florescências no caminho e um tratamento especial será necessário.

A sub-rotina identifica_arestas é responsável pela identificação das arestas que pertencem ao caminho alternado entre dois vértices fornecidos. Para isto, ela executa uma busca simples em profundidade. Na figura 2.7, ela é chamada duas vezes. Uma com a entrada g e a e outra com h e b. O caminho aumentante será a concatenação dos caminhos alternados entre os vértices a e g com a ponte (g, h) e com o caminho alternado entre h e b.

2.5.2 Definições referentes ao algoritmo de Micali e Vazirani

O algoritmo faz uso das seguintes definições referentes aos vértices do grafo.

- *Nível_par*: indica o comprimento do menor caminho alternado de comprimento par até um vértice livre. Caso ele ainda não tenha sido definido o valor do *nível_par* é infinito.

- *Nível_ímpar*: tem a mesma definição do *nível_par* só que se refere a um caminho de comprimento ímpar.
- *Nível*: indica o comprimento do menor caminho até um vértice livre.

$$nível(v) = min\{nível\_par(v), nível\_ímpar(v)\}$$

- *Externo*: um vértice é externo se o seu *nível* for par.
- *Interno*: um vértice é interno se o seu *nível* for ímpar.
- *Outro_nível*: se um vértice for externo (interno) então o *outro_nível* será o seu *nível_ímpar* (*nível_par*).

As definições referentes às arestas são:

- *Ponte*: uma aresta (u, v) é uma ponte quando o $nível\_par(u)$ e $nível\_par(v)$ estiverem definidos (ambos possuem valor diferente de infinito) ou quando o $nível\_ímpar(u)$ e $nível\_ímpar(v)$ estiverem definidos.
- *Tenacidade*: Esta definição é aplicável a uma ponte (u, v).

$$tenacidade(u, v) = min\{nível\_par(u) + nível\_par(v),$$

$$nível\_ímpar(u) + nível\_ímpar(v)\} + 1$$

Este valor representa o comprimento do menor "caminho alternado" entre vértices livres r e s e que contém a ponte (u, v). Este caminho não é necessariamente um caminho simples. Caso r e s sejam o mesmo vértice então a definição de tenacidade está relacionada com a ocorrência de uma florescência. Caso r e s sejam dois vértices livres distintos então o caminho alternado é também um caminho aumentante.

2.5.3 Descrição da sub-rotina busca_pontes

A sub-rotina busca_pontes é responsável pela construção das buscas em largura a partir de todos os vértices livres do grafo. Isto é feito atualizando as informações de *nível_par* e *nível_ímpar* dos vértices. Inicialmente todos os vértices possuem infinito em *nível_par* e *nível_ímpar*, com exceção dos vértices livres, que possuem *nível_par* inicializados com zero.

As buscas são sincronizadas por nível. Definimos *nível_busca* como sendo o nível corrente das buscas em largura. Quando *nível_busca* for par, a sub-rotina considera somente os vértices que possuem *nível_par* igual ao *nível_busca* e quando *nível_busca* for ímpar, ela considera somente os vértices que possuem *nível_ímpar* igual ao *nível_busca*. A primeira expansão das buscas é feita considerando somente os vértices que possuem $nível\_par = 0$, ou seja todos os vértices livres do grafo.

No caso de *nível_busca* ser par então os vértices v que possuem $nível\_par(v) = nível\_busca$ são os que estão na borda das árvores de busca. A expansão a partir destes vértices é feita analisando cada vértice u adjacente a v tal que a aresta (u, v) não esteja emparelhada e que u possua $nível\_ímpar(u) = \infty$, significando que ainda não foi encontrado um caminho alternado entre u e um vértice livre. Cada vértice u encontrado recebe as seguintes atualizações: $nível\_ímpar(u) = nível\_busca + 1$, o vértice u é adicionado ao conjunto de *sucessores* de v e o vértice v é adicionado ao conjunto de *predecessores* de u.

No caso de *nível_busca* ser ímpar então os vértices v que possuem o seu $nível\_ímpar(v) = nível\_busca$ são os que estão na borda da árvore de busca. A expansão é feita a partir destes vértices considerando a aresta emparelhada (v, u). O vértice u recebe a seguinte atualização: $nível\_par(u) = nível\_busca + 1$, indicando que existe um caminho alternado de comprimento $nível\_busca + 1$ entre u e um vértice livre. O inter-relacionamento entre u e v também é feito marcando-se $predecessor(u) = v$ e $sucessor(v) = u$.

A cada aresta (u, v) visitada pela sub-rotina busca_pontes, é feita uma verificação de ocorrência de uma ponte. Se $nível\_par(u)$ e $nível\_par(v)$ estiverem definidos ou se $nível\_ímpar(u)$ e $nível\_ímpar(v)$ estiverem definidos então a condição de ocorrência de uma ponte foi satisfeita. Deste modo, a aresta (u, v) é inserida no conjunto pontes do nível corrente da busca em largura.

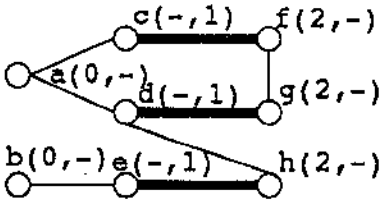

Figura 2.8: Buscas em largura do algoritmo de Micali e Vazirani.

Na exemplo da figura 2.8 observamos a expansão de duas buscas em largura a partir dos dois únicos vértices livres: a e b. Cada vértice da figura possui dois valores: o primeiro é o *nível_par* do vértice e o segundo é o *nível_ímpar*. Os vértices a e b possuem $nível\_par = 0$. No primeiro nível da busca, os vértices c e d são visitados a partir do vértice a e o vértice e a partir do b. No segundo nível a busca é expandida para os vértices f, g e h. No nível seguinte é observada a ocorrência da ponte (f, g), pois $nível\_par(f)$ e $nível\_par(g)$ estão definidos.

Além das marcações de sucessores e predecessores, o algoritmo também faz marcações de *anomalias*. Quando *nível_busca* for par e um vértice v estiver sendo considerado, podemos encontrar a seguinte situação: u é um vértice interno adjacente a v, $nível\_ímpar(u) < nível\_par(v)$ entretanto a aresta (u, v) não está emparelhada. Isto caracteriza uma aresta que faz uma ligação com um nível inferior da busca corrente. Neste caso o vértice v é adicionado ao conjunto de anomalias do vértice u. Na figura 2.8, observamos a ocorrência de uma

anomalia entre os vértices d e h. O vértice h é externo, o vértice d é interno, $nível\_ímpar(d) < nível\_par(h)$ e a aresta (d, h) não está emparelhada. Portanto, o vértice h é inserido no conjunto $anomalias(d)$.

2.5.4 Descrição da sub-rotina procura_caminho

As buscas em largura, feitas a partir de cada vértice livre, crescem sincronizadamente até que haja um encontro entre os seus ramos em arestas. O conjunto destas arestas representa o conjunto de pontes daquele nível de expansão. Cada ponte pode significar a presença de um caminho aumentante ou de uma florescência. Para descobrir qual dos dois casos ocorre numa determinada ponte, utiliza-se a sub-rotina procura_caminho. Ela é chamada para cada ponte e executa duas buscas em profundidade a partir de cada extremo da ponte. Se as buscas encontrarem dois vértices livres distintos então foi identificado um caminho aumentante, caso contrário foi identificada uma florescência. Para descrever o funcionamento da sub-rotina procura_caminho necessitamos do conceito de florescência, o qual passamos a descrever.

Seja (s, t) uma ponte e w um ancestral de s e t. Em outras palavras, é possível alcançar w seguindo as marcações de predecessores dos vértices s e t. O vértice w deve ser o único no seu nível, ou seja, seguindo as marcações de predecessores de s e t, obrigatoriamente passaremos por w naquele nível. Dentre os possíveis vértices w em nível diferentes, definimos b como sendo o de maior nível.

Formalmente, dada uma ponte (s, t) e o vértice b, uma florescência F (*bloom* em Inglês) é o conjunto dos vértices y que obedecem as seguintes restrições:

1. y não pertence a uma outra florescência quando F foi formada,
2. $y = s$ ou $y = t$ ou y é um ancestral de s ou de t , e
3. b é um ancestral próprio de y.

Resumindo, os vértices que pertencem à florescência F são os vértice que compõem a ponte (s, t) e todos os vértices que estão entre s e b e entre t e b, seguindo as marcações de predecessores.

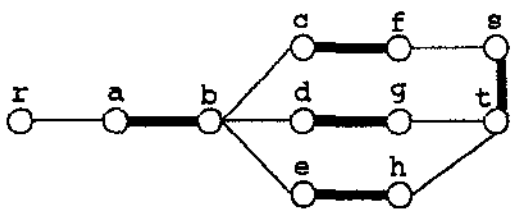

Figura 2.9: Uma florescência no Algoritmo de Micali e Vazirani.

No exemplo da figura 2.9 o vértice r é raiz de uma busca em largura. No quinto nível de expansão é identificada a ponte (s, t). Os ancestrais de s e t que

são únicos nos seus níveis são: a própria raiz da busca r, o vértice a no primeiro nível e b no segundo nível. O ancestral b é o que está no maior nível. Os vértices que compõem a florescência são: $\{c, d, e, f, g, h, s, t\}$.

A sub-rotina procura_caminho é responsável pela procura de dois vértices livres distintos para formar um caminho aumentante. Isto é feito através de duas buscas em profundidade, *direita* e *esquerda*, disparadas pelos extremos da ponte (s, t). Elas são executadas sincronizadamente por nível seguindo as marcações de predecessores deixadas pela sub-rotina busca_pontes.

Definimos ve como sendo o vértice onde se encontra a busca em profundidade esquerda e vd o da busca direita. No exemplo da figura 2.9, a busca esquerda está inicialmente sobre o vértice s, $ve = s$, e a direita sobre t, $vd = t$. A busca esquerda terá prioridade sobre a direita. Ela prosseguirá quando $nível(ve) \geq nível(vd)$. O vértice s é marcado como visitado pela busca esquerda e o t pela busca direita. Como $nível(s) = nível(t)$, a busca esquerda vai do vértice s para o vértice f, $ve = f$, a aresta (s, f) é marcada como *usada*, o vértice s é marcado como *pai* de f e o vértice f é marcado como visitado pela busca esquerda. Neste ponto, $nível(f) < nível(t)$, portanto a vez é da busca direita, ela passa do vértice t para o g. Este processo se repete alternadamente de acordo com o nível de ve e vd. A única restrição imposta é que os vértices visitados pela busca direita não poderão ser visitados pela busca esquerda e vice-versa.

Se as buscas, esquerda e direita, que partiram dos vértice s e t encontrarem dois vértices livres distintos qualquer, x e y, então foi identificado um caminho aumentante. Neste ponto, a sub-rotina identifica_arestas é chamada para descobrir o caminho alternado entre s e x e entre t e y. Estes dois caminhos são concatenados para formar o caminho aumentante. A cardinalidade do emparelhamento é aumentada e as suas arestas são marcadas como apagadas.

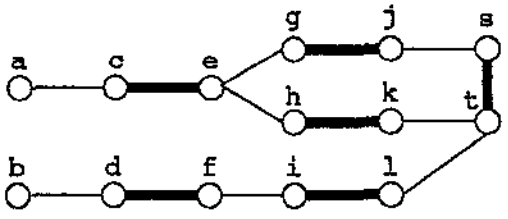

Figura 2.10: Identificação de um caminho aumentante.

No exemplo da figura 2.10 são disparadas duas busca em largura a partir dos vértices a e b. No quinto nível de expansão é encontrada a ponte (s, t). A sub-rotina procura_caminho é chamada para tratá-la. Isto é feito disparando duas buscas em profundidade. A busca esquerda parte do vértice s, $ve = s$, e a direita parte do vértice t, $vd = t$. As buscas prosseguem sincronizadamente. Quando ve estiver sobre o vértice g e vd sobre o h a busca esquerda visitará o vértice e, pois ela tem prioridade. A busca direita não pode descer para o vértice e pois

ele está visitado pela busca esquerda. Neste momento a busca direita retrocede para tentar encontrar um caminho que leve a um vértice mais profundo do que o vértice e. Quando ela retroceder até o vértice t, ela visitará o vértice l, seguindo até o vértice f, onde as duas buscas seguem sincronizadamente até os vértices livres a e b. Neste instante a sub-rotina identifica_arestas é chamada duas vezes: uma para identificar as arestas do caminho entre os vértices s e a e outra para o caminho entre t e b.

Um outro caso possível é quando as buscas, esquerda e direita, não encontram dois vértices livres distintos; neste caso uma florescência foi identificada. Isto ocorre quando elas partem dos extremos da ponte (s,t) e se encontram num ancestral b. De acordo com a definição de florescência, o vértice b é único naquele nível, portanto, somente uma das buscas poderá visitá-lo num determinado instante. Como a busca esquerda tem prioridade sobre a direita, ela coloca ve sobre b e marca b como visitado pela busca esquerda. Neste instante, a busca direita é forçada a retroceder para procurar um outro caminho que conduza a um nível mais profundo do que b. Como isto não é possível, a busca direita falha. Neste ponto, a busca direita toma o vértice b para si e força a busca esquerda a retroceder. Agora é a busca da esquerda que tenta procurar um outro caminho para um nível abaixo de b. A busca esquerda também não encontra, então uma florescência é criada. Os dois vértices mais elevados (s,t) são definidos como sendo os *picos* da florescência e a sua base é o vértice b. Os vértices que compõem a florescência são todos os vértices visitados pelas buscas em profundidade, ou seja, todos que possuem marcas direita ou esquerda. Estes vértices são marcados como pertencentes à florescência e o *outro_nível* de cada vértice v é calculado pela fórmula:

$$outro\_nível(v) = tenacidade(s,t) - nível(v)$$

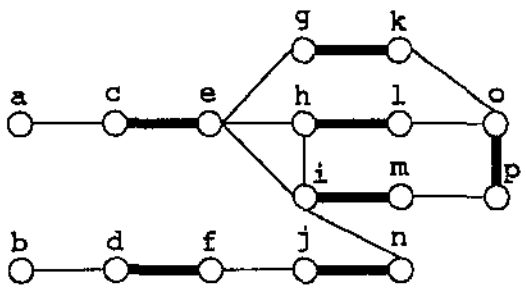

Figura 2.11: Identificação de uma florescência.

No exemplo figura 2.11, a busca em largura que parte do vértice a identifica a ponte (o,p) no quinto nível de expansão. Neste ponto são disparadas duas buscas em profundidade. A busca esquerda parte do vértice o, $ve = o$, e a direita parte do vértice p, $vd = p$. Quando ve estiver sobre o vértice h e vd sobre o vértice i, ve visitará o vértice e, pois a busca esquerda tem prioridade sobre a direita. Neste

instante a busca direita é forçada a retroceder. Entretanto ela não encontra um outro caminho para alcançar um vértice mais profundo do que o vértice e. Então ela é forçada a tomar o vértice e para si e forçar a busca esquerda a retroceder. Isto é feito atribuindo $vd = ve$ e $ve = pai(ve)$. A busca esquerda é forçada a retroceder. Ela encontra um outro caminho partindo do vértice o e seguindo pelo vértice k. Entretanto, este caminho também leva ao vértice e, então a busca esquerda também falha. Isto caracteriza a presença de uma florescência. Os vértices que pertencem a ela são todos os que foram visitados pelas buscas direita, $\{p, m, i\}$, e pela busca esquerda, $\{o, l, h, k, g\}$. A base da florescência é o ponto de encontro entre as buscas, ou seja, o vértice e.

Durante a construção da florescência, a sub-rotina verifica a ocorrência de novas pontes. As pontes com os dois extremos na florescência são desconsideradas, pois, como a condição de florescência também é satisfeita, elas não conduzirão a um caminho aumentante. Um exemplo deste tipo de ponte é encontrado na figura 2.11 onde a ponte (h, i) liga dois vértices da mesma florescência, portanto a ponte é desconsiderada. As pontes, (u, v), com somente um extremo na florescência são analisadas. Digamos que u pertença à florescência, então o vértice u é obrigatoriamente interno e v é uma anomalia de u. As pontes nestas situações podem gerar caminhos aumentantes, portanto são inseridas no conjunto de pontes do nível de expansão j (pontes(j)), onde j é calculado pela fórmula:

$$2j + 1 = tenacidade(u, v)$$

Quando as buscas chegarem no nível j, a ponte (u, v) será avaliada e poderá levar a algum caminho aumentante.

Na figura 2.11, a ponte (i, n) poderá gerar um caminho aumentante. O vértice n é uma anomalia do vértice i. O vértice i pertence a florescência gerada a partir da análise da ponte (o, p), cuja tenacidade é 11, pois tanto o vértice o quanto o p estão no nível 5 e $tenacidade(o, p) = 5 + 5 + 1 = 11$. O vértice i está no terceiro nível de expansão, portanto $nível\_ímpar(i) = 3$ e $outro\_nível(i) = tenacidade(o, p) - nível(i) = 11 - 3 = 8$, ou seja, $nível\_par(i) = 8$. Como o vértice n está no quarto nível de expansão, $nível\_par(n) = 4$. Deste modo $tenacidade(i, n) = 8 + 4 + 1 = 13$. Calculando j, obtemos $2j + 1 = 13$, ou seja, $j = 6$. A ponte (i, n) é inserida no sexto nível e, portanto, será analisada no próximo nível de expansão das buscas.

Durante a execução das buscas podem ocorrer a construção de florescências dentro de florescências. Deste modo é necessário impor uma ordem na sua formação. Um motivo para impor esta ordem é que quando uma busca em profundidade visita um vértice u pertencente a uma florescência F, ela automaticamente salta para $base^*(F)$. Esta é calculada da seguinte maneira: quando uma florescência A é formada, $base(A)$ representa o vértice b da definição de florescência. Se após a formação de uma outra florescência B, a base de A pertencer a B, então $base^*(A) = base(B)$. Se a base de B pertencer a outra florescência

C então $base^*(A) = base^*(B) = base(C)$. Esta situação caracteriza um aninhamento de florescências.

Na figura 2.11, quando a ponte (i, n) for analisada, a busca em profundidade que partir do vértice i verá que ele pertence a uma florescência F, portanto ela saltará para $base^*(F) = base(F) = e$.

A sub-rotina procura_caminho usa duas variáveis, $VCMP$ e *barreira*, para evitar trabalhos desnecessários e garantir que a complexidade numa fase seja $O(m)$. O $VCMP$ (vértice comum mais profundo) representa, dentre os vértices comuns às buscas direita e esquerda, o que está no menor nível. Inicialmente ele é indefinido.

A variável *barreira* está relacionada com a busca direita. Suponha que as buscas direita e esquerda se encontraram num vértice w e $VCMP$ recebeu o vértice w. A busca direita tentou encontrar um caminho para um vértice mais profundo do que w, mas não conseguiu. Entretanto, a busca esquerda conseguiu. Neste ponto as buscas direita e esquerda se encontraram novamente num outro vértice w' comum a elas. Nesse caso *barreira* recebe o vértice da variável $VCMP = w$ e $VCMP$ recebe o novo vértice comum mais profundo, w'. Agora a busca direita não deve retroceder acima do vértice w pois esta tarefa já foi feita anteriormente. Inicialmente a variável *barreira* recebe o vértice onde a busca direita foi iniciada e cada vez que ela falha a variável *barreira* recebe o $VCMP$ corrente.

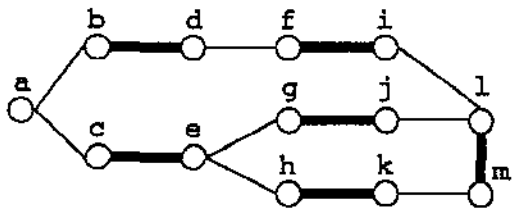

Figura 2.12: Aplicação das variáveis $VCMP$ e *barreira*.

No exemplo da figura 2.12 a ponte (l, m) é identificada no quinto nível de expansão. A busca em profundidade esquerda parte do vértice l e a direita do vértice m. No instante em que $ve = g$ e $vd = h$, a busca esquerda irá para o vértice e, pois ela tem prioridade. A busca direita não pode visitar o vértice e, pois ele está visitado pela busca esquerda. Este é o primeiro vértice em comum entre as buscas e, portanto, $VCMP = e$. A busca direita retrocede e falha, pois não existe um caminho que leve a um vértice mais profundo do que o vértice e. Ela então toma o vértice e para si e força a busca esquerda a retroceder. Esta retrocede até o vértice l e encontra um outro caminho, através do vértice i, que leva a um vértice mais profundo do que o vértice e. As duas buscas prosseguem até que haja o encontro entre elas no vértice a. Neste ponto, a variável *barreira*, que inicialmente recebeu o vértice onde foi disparada a busca direita, m, recebe o vértice da variável $VCMP$, ou seja, o vértice e. O novo vértice comum entre as buscas é o vértice a, $VCMP = a$. A busca direita deverá retroceder, entretanto,

não irá além do vértice e, pois a variável *barreira* indica que este trabalho já foi feito. A busca direita falha e ela, ao tomar o vértice a para si, força a busca esquerda a retroceder. A busca esquerda também falha e uma florescência é formada contendo todos os vértices da figura com exceção da base a.

2.5.5 Descrição da sub-rotina identifica_arestas

Quando a sub-rotina procura_caminho descobre dois vértices livres a e b a partir de dois vértices s e t respectivamente, então significa que ela cresceu uma árvore de busca a partir do vértice s e encontrou o vértice a. O mesmo aconteceu em relação aos vértices t e b. O objetivo da sub-rotina identifica_arestas é identificar nas duas árvores de busca as arestas que compõem o caminho entre s e a e entre t e b. Isto é necessário, pois a sub-rotina procura_caminho, quando encontra algum vértice v que pertence a uma florescência F, salta imediatamente para $base^*(F)$. Portanto, se existirem florescências no caminho, então existem pedaços dele que ainda não foram definidos.

A sub-rotina identifica_arestas recebe como parâmetro dois vértices, vértice *alto* e *baixo*, e uma florescência B. Necessariamente o $nível(alto) \geq nível(baixo)$. O objetivo da sub-rotina é partindo do vértice alto executar uma busca simples em profundidade e encontrar o vértice baixo. Isto é feito seguindo as marcações de predecessores e se limitando à árvore de busca definida pelo vértice alto, ou seja, se a partir do vértice s foi encontrado o vértice livre a através da busca esquerda, então a sub-rotina identifica_arestas que parte de s deverá visitar somente vértices marcados como visitados pela busca esquerda. Para evitar que sejam feitas buscas desnecessárias, um vértice v somente é visitado se o $nível(v) \geq nível(baixo)$.

Cada vez que a busca da sub-rotina identifica_arestas passa de um vértice u para um vértice v, é feito um inter-relacionamento entre u e v. O vértice u é marcado como *pai* de v e a aresta (u, v) é marcada como *visitada*. Em seguida é verificado se o vértice v pertencente a uma florescência F; se pertencer então o caminho através de F será encontrado numa segunda etapa. No momento, sabe-se que o caminho prossegue obrigatoriamente a partir de $base^*(F)$, portanto a busca salta automaticamente para a $base^*(F)$ e prossegue normalmente.

Numa segunda etapa, a sub-rotina identifica_arestas percorre o caminho inverso, do vértice baixo até o alto, através das marcações de vértices-pai feitas durante a descida. Se durante o retorno ela encontrar um vértice v que pertence a uma florescência F então significa que está faltando encontrar o caminho entre a $base(F)$ até o vértice v, atravessando a florescência F. Para isto, a sub-rotina identifica_arestas usa uma pequena sub-rotina chamada abre_florescência e passa como parâmetro o vértice v.

A sub-rotina abre_florescência é responsável pela construção do caminho através de uma florescência F e para isto ela utiliza recursivamente a sub-rotina identifica_arestas. Nesta situação a sub-rotina identifica_arestas é informada através

do parâmetro B, $B = F$, que está sendo construído um caminho através da florescência F.

O vértice v, passado como parâmetro para a sub-rotina abre_florescência, indica o início do caminho que deve ser encontrado através da florescência até a sua base. Se o vértice v for um vértice externo então abre_florescência chama recursivamente identifica_arestas para encontrar o caminho entre v e a $base(F)$, onde F é a florescência a qual o vértice v pertence. Se v é interno então o caminho passa pelos picos (s,t) pertencente à florescência F. Neste caso se ambos os vértices s e v estão marcados como visitados pela busca esquerda, então identifica_arestas é chamada duas vezes, uma para encontrar o caminho entre s e v e outra para encontrar o caminho entre t e $base(F)$; caso contrário ela é chamada duas vezes, uma com as entradas, t e v e outra com s e $base(F)$.

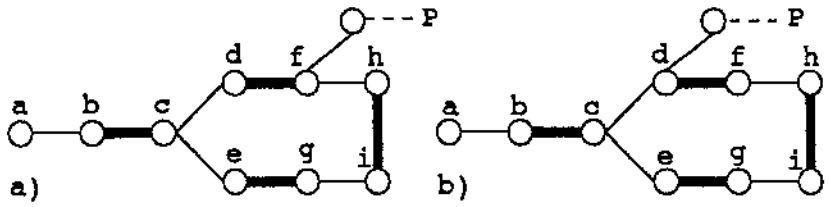

Figura 2.13: a) Caminho direto para a base. b) Caminho através dos picos.

No exemplo da figura 2.13a a sub-rotina identifica_arestas, seguindo as marcações de vértices-pai a partir do vértice a, observa que $pai(c) = f$ e que f que pertence a uma florescência. Como o vértice f é externo, a sub-rotina abre_florescência chama identifica_arestas recursivamente para encontrar o caminho entre o vértice f e c (base da florescência). Na figura 2.13b, a sub-rotina identifica_arestas, seguindo as marcações de vértices-pai a partir do vértice a, observa que $pai(c) = d$ e que o vértice d pertence a uma florescência. Como o vértice d é interno, o caminho passa pelos picos (h,i) da florescência. Neste caso a sub-rotina abre_florescência chama identifica_arestas recursivamente duas vezes. A primeira chamada é para encontrar o caminho entre h e d e a outra chamada é para encontrar o caminho entre i e c, contornando a florescência.

Após encontrado o caminho aumentante, os seus vértices deverão ser marcados como *apagados*. Isto para que sejam encontrados somente caminhos aumentantes disjuntos. As buscas controladas pela função identifica_arestas consideram somente vértices que não estejam apagados. Estas marcas deverão ser retiradas no início de cada fase.

Um vértice deve ser apagado se ele pertencer ao caminho aumentante recém descoberto ou se todos os seus predecessores estiverem apagados. Para isto, cada vértice z possui um contador contendo o número de predecessores não apagados. Cada vez que um predecessor de z for apagado o contador é subtraído de 1. Quando o contador atinge 0 o vértice z também é apagado.

No exemplo da figura 2.14, a busca em largura que parte do vértice b forma

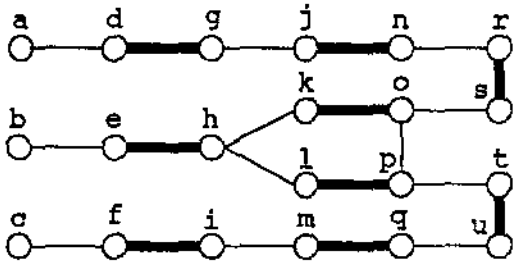

Figura 2.14: Identificação das arestas de um caminho aumentante.

a florescência a partir da ponte (o, p) com base o vértice h. Prosseguindo com a busca, são descobertas duas pontes: (r, s) a partir do encontro entre as buscas do vértice a e do vértice b, e a ponte (t, u) a partir do encontro entre as buscas do vértice b e c. Supondo que a ponte (r, s) gere o caminho aumentante entre os vértices a e b então as suas arestas são trocadas para aumentar a cardinalidade do emparelhamento e os seus vértices são marcados como apagados. O vértice l possuía um predecessor não apagado que era o vértice h. Como ele foi apagado, o vértice l também é apagado, pois o seu contador que antes possuía valor 1 agora possui 0. O mesmo ocorre com os vértices p e t. O resultado é que a ponte (t, u) não será explorada, pois o vértice t está apagado e não conduzirá a nenhum caminho aumentante.

Com isto finalizamos a descrição do algoritmo de Micali e Vazirani. Tomando como base esta descrição, desenvolvemos uma implementação deste algoritmo, a qual apresentou algumas dificuldades na resolução de algumas instâncias de grafos. Algumas destas instâncias também foram encontradas por R. Bruce Mattingly e Nathan P. Ritchey [MR93]. Estas instâncias estão descritas abaixo: primeiro as que também foram encontradas por Mattingly e Ritchey e em seguida apresentaremos uma instância nova não observada por Mattingly e Ritchey.

2.5.6 Casos especiais observados por Mattingly e Ritchey

Problema

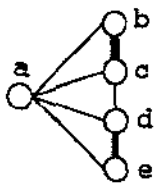

Figura 2.15: Problema na análise de uma ponte.

De acordo com a formulação original do algoritmo, uma ponte pode resultar ou na criação de uma florescência ou na identificação de um caminho aumentante. Entretanto, existem instâncias em que não ocorrem nenhum dos dois casos.

Na figura 2.15, observamos a criação de duas florescências: $F_1 = \{b, c\}$ e $F_2 = \{d, e\}$. O vértice a é a base das duas florescências e elas foram criadas no primeiro nível da busca em largura. No segundo nível de expansão a ponte (c, d) é identificada. Se ela for tratada de acordo com a descrição original do algoritmo, isto ocasionará a formação de uma nova florescência contendo os vértices c e d, os quais já pertencem a florescências. Entretanto, uma das restrições impostas pelo algoritmo é que um vértice pertença somente a uma florescência. Portanto, a florescência $F_3 = \{c, d\}$ não poderá ser criada e a ponte (c, d) não deverá produzir efeito algum.

Solução proposta por Mattingly e Ritchey

Quando uma ponte (s, t) for tratada, duas buscas em profundidade são disparadas, uma a partir de s e outra a partir de t. Se o vértice de partida s pertencer a uma florescência então a busca saltará para $base^*(s)$. O mesmo acontece com o vértice t. Se as duas buscas saltarem para o mesmo vértice então o algoritmo retorna sem executar trabalho algum.

Na figura 2.15, ao analisar a ponte (c, d) uma busca saltará para $base^*(c)$, pois o vértice c pertence a uma florescência, e a outra busca saltará para $base^*(d)$. Como $base^*(c) = base^*(d) = a$, as buscas em profundidade começam sobre o mesmo vértice. Entretanto, uma das condições para o crescimento destas buscas é que vértices distintos sejam visitados durante todo o processo. Como esta condição não é satisfeita neste primeiro passo as buscas retornam imediatamente sem identificar nenhuma florescência.

Problema

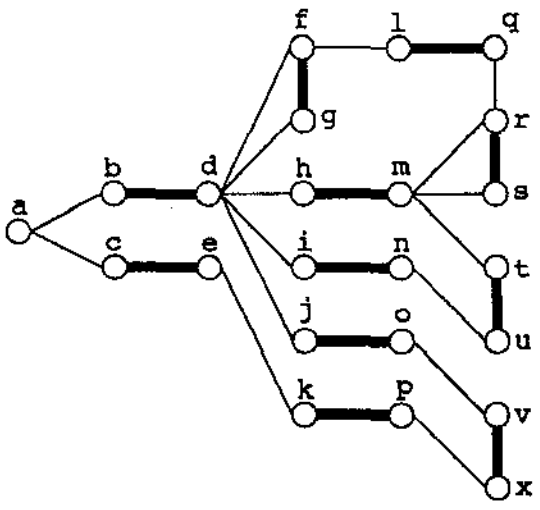

Figura 2.16: Problema na busca em profundidade.

Na figura 2.16 observamos um caso onde as duas buscas em profundidade da sub-rotina procura_caminho podem interferir uma na outra. A busca em largura parte do vértice a. No terceiro nível de expansão a ponte (f, g) gera a florescência contendo os vértices $\{f, g\}$ com base no vértice d. No quinto nível são formadas as seguintes florescências: $\{r, s\}$ com base m, $\{t, u, m, n, h, i\}$ com base d e $\{v, x, o, p, j, k, d, e, b, c\}$ com base a. No sexto nível a ponte (q, r) é identificada. A busca direita parte do vértice r, vai para o vértice m. Como o vértice m também pertence a uma florescência ela salta para o vértice d e depois para o a. A busca esquerda parte do vértice q, vai para o l e depois para o f. Como o vértice f pertence a uma florescência a busca salta para o vértice d e depois para o a. Durante as buscas, os vértices mantém um ponteiro para os vértices-pai. A busca direita marcou $pai(d) = m$. Entretanto, a busca esquerda escreveu sobre este valor marcando $pai(d) = f$. Quando a busca direita retornar ela seguirá o caminho deixado pela busca esquerda, ocasionando um erro.

Solução

A solução por nós adotada foi proporcionar marcações distintas para as duas buscas. Os vértices mantêm as informações de vértices pai direito e esquerdo. Na figura 2.16 o vértice d tem as seguintes marcações: $pai\_direito(d) = m$ e $pai\_esquerdo(d) = f$.

Esta solução de nada interfere no funcionamento do algoritmo, pois durante o crescimento das buscas em profundidade, todas as condições impostas pelo algoritmo não são modificadas e o retorno de uma busca sempre seguirá as suas próprias marcações de vértice-pai sem sofrer interferência alguma.

Problema

A função identifica_aresta, ao receber dois vértices como parâmetros, alto e baixo, executa um busca em profundidade a partir do alto em busca do baixo. Um vértice somente será visitado pela busca se possui a mesma marcação de direita ou esquerda do vértice alto. Entretanto, existem instâncias, como na figura 2.17, em que esta restrição ocasiona um erro.

Na figura 2.17 a ponte (w, x) é encontrada no sétimo nível. A busca direita parte do x e a esquerda do w. Elas se encontram no vértice l. A busca direita não encontra um outro caminho que leve a um vértice mais profundo do que l. Entretanto, a busca esquerda consegue. As buscas se encontram no vértice f formando a seguinte florescência $\{i, l, p, t, x\} \cup \{h, k, n, o, r, s, w\}$. Os vértices do primeiro conjunto estão marcados como pertencentes à busca direita e o segundo como pertencentes à busca esquerda. Quando a ponte (y, z) for analisada as duas buscas em profundidade encontrarão dois vértices livres distintos. A busca esquerda parte do vértice y e encontra o a. A busca direita parte do z, vai para o v e depois para o s. Como o vértice s pertence a uma florescência ela

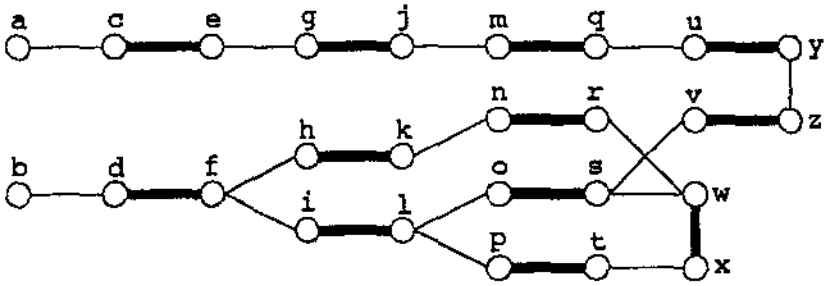

Figura 2.17: Problema na identificação das arestas de um caminho alternado.

salta para o vértice f, continuando para o vértice d e encontrando o vértice livre b. Na segunda etapa, a sub-rotina abre_florescência é chamada para achar o caminho entre os vértices s e f, pois o vértice s pertence a uma florescência. O único caminho possível é o (s, o, l, i, f). Na formulação original do algoritmo, este caminho não seria encontrado pois os vértices s e o estão marcados como pertencentes à busca esquerda e os vértices l e i como pertencentes à busca direita, ocasionando um erro.

Solução

A implementação permite que a busca possa visitar vértices com marcas diferentes do vértice alto quando for necessário somente uma chamada da função identifica_arestas, ou seja, quando o caminho não passa pelos picos da florescência.

Se uma florescência é formada então a busca que a gerou obrigatoriamente passou pela base e encontrou uma ponte responsável pela sua formação. Deste modo, por construção, podemos afirmar que as marcações de predecessores partem dos picos e vão até a base da florescência. Se um caminho passa através de uma florescência sem passar pelos seus picos, ou seja, vai diretamente até a base, então ele está representado pelas marcações de predecessores. Portanto, se for necessário somente uma chamada da função identifica_arestas não será necessário verificar as marcações de direita ou esquerda, basta seguir as marcações de predecessores.

2.5.7 Novo caso especial

Problema

Na formulação original do algoritmo, a busca esquerda tem prioridade sobre a da direita. Quando as duas buscas se encontram num vértice a busca da direita é a que retrocede antes. Caso o retrocesso falhe, o vértice comum mais profundo ($VCMP$) é definido e a busca esquerda é forçada a retroceder. Para estas ações acontecerem na ordem correta, o vértice comum às buscas deve ser visitado primeiro pela busca esquerda, impedindo o avanço da direita. Entretanto,

existem casos em que a busca da direita alcança este vértice antes da esquerda, ocasionando um erro.

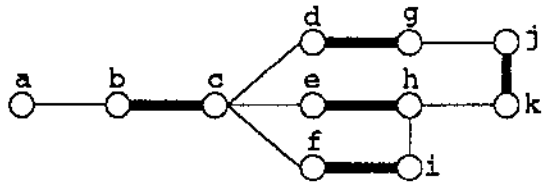

Figura 2.18: Problema na definição do VCMP.

Na figura 2.18, a ponte (h, i) é descoberta no quarto nível. A florescência criada contém os vértices $\{h, i, e, f\}$ com a base c. No quinto nível a ponte (j, k) é descoberta e a busca direita parte do vértice k e a esquerda do vértice j. A busca esquerda tem prioridade, portanto passa para o vértice g. A direita passa para o h. Como o vértice h pertence a uma florescência, a busca salta para o vértice c. A busca esquerda passa para o d e encontra o vértice c já visitado pela busca direita. Neste momento ela é obrigada a retroceder até o vértice j. De acordo com o algoritmo, a falha no retrocesso da busca esquerda ocasionaria a formação de uma florescência com a base no vértice $VCMP$. Entretanto, o $VCMP$ ainda não foi definido pela busca direita, pois ele somente é definido quando a direita falha no retrocesso. Como ela não retrocedeu, isto ocasiona um erro.

Solução

Para resolver este problema, a implementação possibilita que a busca esquerda se comporte como a direita e vice versa. Na figura 2.18, o retrocesso da busca esquerda não ocasionaria a formação de uma florescência, pois a busca direita ainda não tentou encontrar um outro caminho. O que acontece é uma troca dos papéis: a busca esquerda marca o $VCMP$ com sendo o vértice c e força a direita a encontrar um outro caminho. Como ela não encontra, a busca direita forma a florescência.

2.5.8 Corretude e complexidade computacional

A demonstração de corretude do algoritmo de Micali e Vazirani é complexa e aparece num artigo de Vazirani [Vaz94] publicado 14 anos após o artigo que apresentou o algoritmo [MV80]. Nesta seção nos limitamos a dar uma visão geral do tempo de execução.

Numa fase o algoritmo de Micali e Vazirani encontra o conjunto máximo de caminhos aumentantes disjuntos de menor comprimento. Para encontrar o emparelhamento máximo são necessárias $O(\sqrt{n})$ fases [HK73]. É necessário mostrar que uma fase possui complexidade de $O(m)$.

Durante uma fase, a sub-rotina busca_pontes visita um vértice no máximo duas vezes, uma num nível de busca par, outra num nível ímpar. Além do mais, ela visita uma aresta no máximo duas vezes, uma em cada direção. Portanto, a complexidade da sub-rotina busca_pontes é $O(m+n) = O(m)$.

A sub-rotina procura_caminho visita uma aresta no máximo uma vez durante uma fase. Uma vez que um vértice é marcado como visitado pela busca direita ou esquerda, a sub-rotina não mais visita as suas arestas predecessoras novamente, mesmo que a sub-rotina seja chamada uma segunda vez. Para o cálculo de $base^*$ pode-se utilizar o algoritmo de união incremental de conjuntos de Gabow e Tarjan [GT85]. Isto resulta numa complexidade da sub-rotina de $O(m+n)$.

A sub-rotina identifica_arestas marca cada aresta como visitada no máximo uma vez. O mesmo ocorre com a marcação dos vértices como visitados. O tempo computacional da sub-rotina identifica_arestas é $O(m+n) = O(m)$.

Durante uma fase, o contador de predecessores não apagados que cada vértice possui é decrementado uma vez para cada aresta que une o vértice ao seu predecessor, portanto a complexidade é $O(m)$. A complexidade computacional de uma fase resulta em $O(m)$ e portanto o algoritmo possui complexidade $O(\sqrt{n}\, m)$.

2.6 Algoritmo de Blum

Blum [Blu94] propôs um novo algoritmo para o problema do emparelhamento máximo em grafos genéricos. Este algoritmo evita a consideração explícita de caminhos aumentantes, flores e florescências. De acordo com o autor, a estrutura utilizada possibilita uma apresentação simplificada do algoritmo de Micali e Vazirani. Faremos uma descrição bastante breve do algoritmo apenas enunciando algumas das idéias principais.

Essencialmente, o problema de se encontrar caminhos aumentantes foi reduzido ao problema de *alcançabilidade em grafos bipartidos orientados*. Foi mostrado com resolver este problema utilizando uma busca em profundidade modificada. O algoritmo resultante não é essencialmente diferente dos algoritmos anteriores, pois, apesar de se estar observando o problema através de uma nova ótica, as operações executadas pelo algoritmo se assemelham muito às dos algoritmos anteriores, principalmente do algoritmo de Micali e Vazirani.

A redução apresentada no algoritmo de Blum é muito semelhante à redução feita para transformar o problema do emparelhamento máximo em grafos bipartidos num problema de alcançabilidade. Portanto, apresentaremos primeiramente a redução feita quando desejamos encontrar o emparelhamento máximo em grafos bipartidos, em seguida apresentaremos a redução feita quando se deseja obter o emparelhamento máximo em grafos genéricos.

2.6.1 Redução em grafos bipartidos

A partir de um grafo bipartido $G = (A, B, E)$ obtém-se um *emparelhamento inicial* $M \subseteq E$ verificando para cada vértice livre v do grafo, se existe algum outro vértice livre u que esteja adjacente a v. Se existir então a aresta (v, u) é emparelhada.

Utilizando o grafo $G = (A, B, E)$ com o emparelhamento inicial M constrói-se um grafo direcionado $G' = (A', B', E')$ seguindo passos descritos abaixo:

1. As arestas do emparelhamento M são direcionadas do conjunto de vértices A para o B, e as arestas não emparelhadas $E - M$ são direcionadas de B para A.

2. São adicionados dois novos vértices ao grafo, s e t.

3. Para cada vértice livre v do conjunto A é adicionada a aresta direcionada (v, t) a E', e para cada vértice livre w do conjunto B é adicionada a aresta direcionada (s, v) a E'.

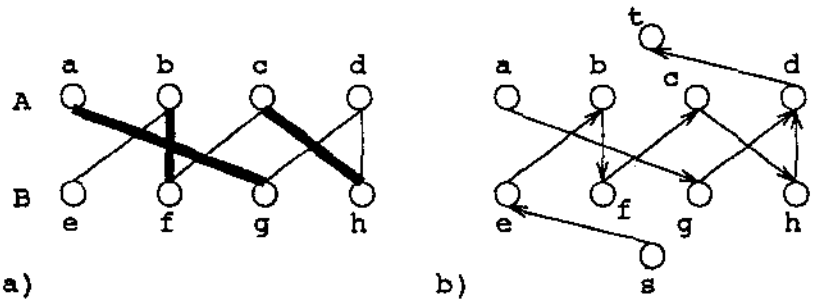

Figura 2.19: a) Grafo bipartido b) Redução para o problema de alcançabilidade.

Com esta nova configuração, pode-se provar que existe um caminho aumentante no grafo G se somente se existe um caminho simples entre os vértices s e t no grafo G'. No grafo bipartido da figura 2.19a obtemos um emparelhamento inicial contendo três arestas. Aplicando-se as três regras descritas acima obtemos o grafo direcionado bipartido da figura 2.19b, o qual apresenta um caminho simples entre s e t: (s, e, b, f, c, h, d, t). Isto necessariamente implica na existência de um caminho aumentante na figura 2.19a: (e, b, f, c, h, d).

Para se encontrar o emparelhamento máximo podemos utilizar um algoritmo descrito por Hopcroft e Karp [HK73]. Primeiramente, é feita uma redução do problema de encontrar caminhos aumentantes para o problema de alcançabilidade. Neste ponto, é construída uma busca em largura a partir do vértice s até que o vértice t seja alcançado. Deste modo, obtém-se um grafo direcionado em camadas, o qual corresponde exatamente aos caminhos aumentantes de menor comprimento no grafo G. Usando uma busca em profundidade a partir do vértice s encontra-se o conjunto máximo dos caminhos aumentantes disjuntos. Cada vez que um

caminho aumentante é encontrado, ele é apagado junto com todas as arestas incidentes sobre ele. A busca em profundidade continua logo em seguida. Tanto a busca em largura quanto a busca em profundidade podem ser implementadas em $O(m+n)$. Para se encontrar o emparelhamento máximo são necessárias $O(\sqrt{n})$ fases. Portanto a complexidade resultante para o caso bipartido é $O(\sqrt{n}\,m)$.

2.6.2 Redução em grafos genéricos

Para o caso de grafos genéricos $G = (V, E)$, a situação é mais complexa. Para obtermos $G' = (V', E')$ é necessário substituir cada vértice $v \in V$ por dois novos vértices: $[v, A]$ e $[v, B]$. As arestas do grafo G' são definidas da seguinte forma:

1. Se uma aresta (v, w) do grafo original G estava emparelhada então são adicionadas duas arestas orientadas a E', ambas de A para B: $([v, A], [w, B])$ e $([w, A], [v, B])$.

2. Se uma aresta (i, j) do grafo original G não estava emparelhada então são adicionadas duas arestas orientadas a E', ambas de B para A: $([i, B], [j, A])$ e $([j, B], [i, A])$.

3. Para cada vértice livre $u \in V$, adicionamos duas novas arestas a E': $(s, [u, B])$ e $([u, A], t)$, onde s e t são dois novos vértices inseridos em V'.

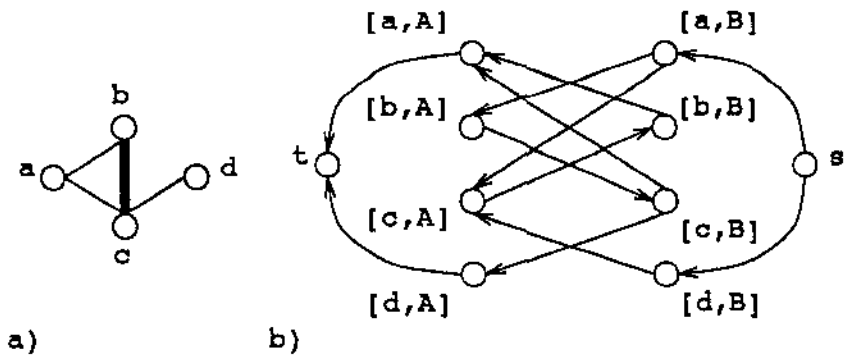

Figura 2.20: a) Grafo G b) Grafo G' gerado a partir do grafo G.

Ao aplicarmos as definições acima sobre o grafo G da figura 2.20a obtemos o grafo G' da figura 2.20b. O caminho aumentante (a, b, c, d) em G pode ser observado pelo caminho $(s, [a, B], [b, A], [c, B], [d, A\}, t)$ em G'.

A redução aplicada sobre grafos genéricos requer uma tratamento especial, pois a existência de um caminho simples P entre os vértices s e t do grafo G' não necessariamente implica na existência de um caminho aumentante em G. Na figura 2.20b, o caminho $(s, [a, B], [c, A], [b, B], [a, A], t)$ não corresponde a um caminho aumentante em G pois os vértices $[a, A]$ e $[a, B]$ são na realidade o mesmo vértice. Portanto, é necessário definir algumas restrições que devem ser

obedecidas pelo caminho P para que ele possa implicar na existência de um caminho aumentante no grafo G. Esta restrições são:

- P deve ser um caminho simples.
- $\forall [v, A] \in V' : [v, A] \in P \Rightarrow [v, B] \notin P$.

Se um caminho P obedecer a estas restrições então o chamaremos de *caminho fortemente simples*. Com esta definição, Blum provou o seguinte teorema:

Teorema 3 *Seja $G = (V, E)$ um grafo não orientado genérico, $M \subseteq E$ um emparelhamento inicial e $G' = (V', E')$ um grafo construído da forma descrita acima. Existe um caminho aumentante em G se e somente se existe um caminho fortemente simples em G'.*

A estratégia utilizada no algoritmo de Blum para encontrar o emparelhamento máximo em grafos genéricos segue a mesma linha do algoritmo descrito por Hopcroft e Karp para o caso bipartido. As únicas diferenças são que tanto as buscas em largura quanto as em profundidade devem encontrar somente caminhos fortemente simples. Primeiramente é disparada uma busca em largura partindo do vértice s até que o vértice t seja encontrado. A árvore de busca criada durante a construção da busca em largura corresponde ao conjunto máximo de caminhos fortemente simples de menor comprimento. Em seguida uma busca em profundidade é usada para encontrar o conjunto máximo dos caminho fortemente simples disjuntos de menor comprimento. Cada vez que um caminho fortemente simples é encontrado, o caminho com todas as arestas incidentes sobre ele são apagadas. Se com isto algum vértice ficar sem arestas incidentes sobre ele ou sem arestas que partem dele, então ele também é apagado. A busca em profundidade continua logo em seguida. Tanto as buscas em largura quanto as em profundidade podem ser implementadas em $O(m + n)$. Para se encontrar o emparelhamento máximo são necessárias $O(\sqrt{n})$ fases. Isto resulta numa complexidade final de $O(\sqrt{n}\, m)$.

Capítulo 3

Emparelhamento em Paralelo

Nesta seção descreveremos brevemente um algoritmo paralelo para o problema do emparelhamento devido a Mulmuley, Vazirani e Vazirani [MVV87]. Esse algoritmo foi projetado para o modelo de computação PRAM, e nosso objetivo neste capítulo é mostrar que o algoritmo não é prático.

O modelo PRAM [Jaj92] pode ser sumariamente descrito da seguinte forma. Existe um número P de processadores que é função do tamanho do problema e que se comunicam através de memória compartilhada. Cada processador é uma RAM, e num passo do algoritmo cada processador pode acessar qualquer posição de memória [jaja].

O algoritmo randomizado MVV se baseia numa generalização de uma caracterização algébrica de grafos. Nesta caracterização, o determinante de uma matriz, associada a um grafo G é diretamente relacionado com a existência ou não de um *emparelhamento perfeito* em G. Um emparelhamento somente é perfeito se ele emparelha todos os vértices do grafo.

A idéia de se identificar a existência de um emparelhamento perfeito através do determinante de uma matriz pode ser expandida para também fornecer as arestas que o compõem. Por sua vez, isto pode ser novamente expandido para fornecer as arestas pertencentes a um emparelhamento máximo.

A relação existente entre o determinante de matriz e emparelhamento em grafos foi descoberta por Tutte (conforme citado por MVV). Basicamente, o problema de se identificar a presença de um emparelhamento perfeito num grafo G foi reduzido ao problema de achar o determinante de uma matriz T.

Passaremos em seguida a descrever como é construída a matriz T . As provas dos lemas apresentados nesta seção não são essenciais para o entendimento do algoritmo e podem ser encontradas na literatura [MVV87], [Jaj92], [Lei90].

Dado um grafo $G = (V, E)$, definimos os elementos da *matriz de Tutte* T da

seguinte forma:

$$t_{ij} = \begin{cases} x_{ij} \text{ se } i < j \text{ e } (i,j) \in E \\ -x_{ij} \text{ se } i > j \text{ e } (i,j) \in E \\ 0 \text{ se } i = j \text{ ou } (i,j) \notin E \end{cases}$$

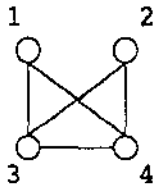

Figura 3.1: Grafo que admite um emparelhamento perfeito.

Um exemplo da construção da matriz de Tutte pode ser observado na matriz abaixo, a qual corresponde ao grafo da figura 3.1.

$$T = \begin{bmatrix} 0 & 0 & x_{13} & x_{14} \\ 0 & 0 & x_{23} & x_{24} \\ -x_{13} & -x_{23} & 0 & x_{34} \\ -x_{14} & -x_{24} & -x_{34} & 0 \end{bmatrix}$$

O determinante da matriz de Tutte T está diretamente relacionado com a existência de um emparelhamento perfeito no grafo do qual ela foi gerada. Para observar este fato, definimos S_n como sendo o conjunto de todas as permutações π de n elementos. Deste modo, o determinante da matriz de Tutte T pode ser expresso da seguinte forma:

$$det(T) = \sum_{\pi \in S_n} snl(\pi) t_{1\pi(1)} t_{2\pi(2)} \cdots t_{n\pi(n)}$$

O sinal $snl(\pi)$ de uma permutação π é positivo se a permutação for composta por um número par de *transposições* e negativo caso contrário. Uma transposição é uma troca de posição entre dois elementos formando um ciclo de comprimento 2, ou seja, o elemento i é transposto para a posição $\pi(i)$, e o elemento nesta posição é deslocado para a posição $\pi(\pi(i)) = i$.

Representamos $t_{1\pi(1)} t_{2\pi(2)} \cdots t_{n\pi(n)}$ como sendo $val(\pi)$. Se $val(\pi) \neq 0$ então $t_{i\pi(i)} \neq 0$ para $1 \leq i \leq n$, representando que nesta permutação $(i, \pi(i)) \in E$, para $1 \leq i \leq n$. Neste caso as arestas $(i, \pi(i))$ definem um subgrafo chamado de *trilha* da permutação π.

Uma permutação pode ser expressa como um produto de ciclos disjuntos. Um ciclo de comprimento 2 consiste em uma aresta transpassada duas vezes.

O determinante da matriz de Tutte T gerada a partir da figura 3.1 é dado por:

$$det(T) = x_{13}^2 x_{24}^2 + x_{14}^2 x_{23}^2 - 2x_{13}x_{24}x_{14}x_{23}$$

O primeiro termo corresponde à permutação $\pi(1) = 3$, $\pi(3) = 1$, $\pi(2) = 4$ e $\pi(4) = 2$. A trilha correspondente a este termos representa um possíveis emparelhamento perfeito: $\{(1,3),(2,4)\}$. O último termo corresponde à permutação $\pi(1) = 3$, $\pi(3) = 2$, $\pi(2) = 4$, $\pi(4) = 1$. A trilha desta permutação é o subgrafo $\{(1,3),(3,2),(2,4),(4,1)\}$, que corresponde a um ciclo de comprimento 4. Escolhendo qualquer subconjunto das arestas que compõem a trilha e que possuem vértices disjuntos, formamos um emparelhamento perfeito, por exemplo: $\{(3,2),(4,1)\}$.

De acordo com estas definições Tutte provou o seguinte teorema:

Teorema 4 *Seja $G = (V, E)$ um grafo e T a matriz de Tutte correspondente. Então, $det(T) \neq 0$ se e somente se G admite um emparelhamento perfeito.*

Este teorema somente afirma que é possível indicar a existência de um emparelhamento perfeito através do determinante da matriz de Tutte. Mas ele não fornece meios para se indicar as arestas que compõem um emparelhamento perfeito. Para que isto seja possível, é necessário expandir as idéias apresentadas até o momento.

O problema principal existente na identificação das arestas pertencentes a um emparelhamento perfeito é que um grafo G pode admitir vários emparelhamento perfeitos distintos. Isto prejudica o trabalho concorrente na convergência para um determinado emparelhamento. Este problema pode ser contornado atribuindo valores inteiros aleatórios às arestas do grafo. Este valores são escolhidos aleatoriamente e uniformemente entre $[1...2m]$. Com isto pode-se garantir a unicidade de um emparelhamento perfeito de peso mínimo com uma probabilidade de no mínimo $\frac{1}{2}$. Esta afirmação está baseada no seguinte lema, denominado *lema do isolamento* [MVV87]:

Lema 2 *Seja (S, F) um sistema de conjuntos, ou seja, S é um conjunto finito de elementos: $S = \{e_1, e_2, ..., e_n\}$ e F é uma coleção de subconjuntos de S: $F = \{S_1, S_2, ...S_t\}$, onde $S_j \subseteq S$ para $1 \leq j \leq t$. Cada elemento de S recebe um inteiro aleatório w_i escolhido uniformemente entre $[1...2n]$, onde n é o número de elementos em S. Cada subconjunto S_i é representado pela soma dos valores dos seus elementos. Deste modo a probabilidade de haver um subconjunto único de peso mínimo em F é no mínimo $\frac{1}{2}$.*

Para aplicar o lema 2 consideramos G como um sistema de conjuntos. Os elementos do sistema são as arestas de G e os conjuntos são os emparelhamentos perfeitos.

Para identificar as arestas que compõem o emparelhamento perfeito de peso mínimo, primeiramente deve-se calcular seu respectivo peso. Para isto substituímos as variáveis x_{ij} da matriz T de Tutte por $2^{w_{ij}}$, aonde w_{ij} é um inteiro aleatório do lema 2 atribuído à aresta $(i, j) \in E$. Denotamos a matriz resultante

por T'. Para calcular o peso mínimo de um emparelhamento perfeito usamos o seguinte lema:

Lema 3 *Seja $G = (V, E)$ um grafo com um único emparelhamento perfeito de peso mínimo, digamos peso W, e T' a matriz de pesos definida acima. Então $det(T') \neq 0$ e a maior potência de 2 que divide $det(T')$ é 2^{2W}.*

Sabendo-se o peso W de um emparelhamento perfeito, é possível computar as arestas que pertencem ao emparelhamento perfeito de peso mínimo.

Lema 4 *Seja $G = (V, E)$ um grafo com um único emparelhamento perfeito M de peso mínimo, digamos W. Então a aresta (i, j) pertence a M se e somente se $2^{w_{ij}} det(T'_{ij})/2^{2W}$ é um inteiro ímpar, aonde T'_{ij} é a submatriz de T obtida após a remoção da linha i e da coluna j.*

Com isto, podemos descrever os principais passos do algoritmo paralelo randomizado para se obter um emparelhamento perfeito. A entrada do algoritmo é um grafo $G = (V, E)$, tal que G admite um emparelhamento perfeito. A saída do algoritmo é um conjunto M de arestas tal que M é um emparelhamento com probabilidade de no mínimo $\frac{1}{2}$.

1. Atribuir pesos aleatórios w_{ij} às arestas (i, j) do grafo, tal que cada peso é escolhido uniformemente e independentemente entre $[1...2m]$, onde $m = |E|$.

2. Seja T' uma matriz derivada da matriz de Tutte do grafo G substituindo-se cada variável x_{ij} por $2^{w_{ij}}$.

3. Computar $det(T')$ e deduzir o valor W do peso do emparelhamento perfeito de peso mínimo usando o lema 3.

4. Computar a matriz adjunta $adj(T')$, onde $adj(T') = det(T') \times T'^{-1}$. É uma propriedade das matrizes adjuntas que o elemento ji de $adj(A)$ é $det(A_{ij})$ onde A_{ij} é o que resta da matriz A quando se remove a linha i e a coluna j.

5. Para cada aresta (i, j) faça em paralelo: se o resultado de $2^{w_{ij}} det(T'_{ij})/2^{2W}$, onde $det(T'_{ij})$ é dado pelo elemento ji da matriz $adj(T')$, for um número ímpar então inclua a aresta (i, j) no emparelhamento M.

O algoritmo acima apenas encontra o emparelhamento perfeito. É possível estender esse resultado para que o algoritmo encontre o emparelhamento máximo, caso G não tenha um emparelhamento perfeito. Maiores detalhes podem ser encontrados em [Jaj92].

3.1 Análise quanto a possibilidade de implementação

A complexidade do algoritmo MVV é determinada pelos passos 3 e 4, que exigem o cálculo de um determinante e de uma inversa de matriz. Conforme [Jaj92], isto pode ser feito em tempo $O(log^2 n)$ com $O(nM(n))$ operações. $M(n)$ é o número de operações necessárias para multiplicar 2 matrizes $n \times n$. Sabemos que $M(n) = \Omega(n^2)$, portanto o algoritmo acima exige pelo menos $\Omega(n^3)$ operações.

Considerando ainda os seguintes fatores:

- algoritmos paralelos para inversão de matrizes são "lentos, ineficientes em termos de número de processadores, horrivelmente complicados e numericamente instáveis" [Lei90]. Portanto, as constantes são altas.
- o número de operações do algoritmo de Micali e Vazirani é $O(n^{1.5})$ para grafos esparsos.
- manipulação de valores altos como $2^{w_{ij}}$.

Podemos concluir que MVV não é um algoritmo prático e que portanto para se desenvolver uma implementação paralela eficiente na prática, ela deveria se basear em algum algoritmo seqüencial existente para o problema do emparelhamento máximo em grafos genéricos.

Capítulo 4

Implementações Paralelas

Devido ao alto custo computacional de uma implementação do algoritmo paralelo randomizado MVV, optamos por desenvolver uma implementação paralela baseada em algum dos algoritmos seqüenciais existentes para o problema do emparelhamento máximo. O trabalho ocorreu em duas fases: primeiro, uma tentativa sem sucesso de paralelização do algoritmo de Micali e Vazirani; segundo, uma implementação com sucesso baseada em algumas características dos algoritmos de Edmonds, Gabow e de Micali e Vazirani.

A máquina destinada às implementações paralelas possui a sua memória compartilhada entre os seus processadores. Deste modo, o projeto das implementações foi, desde o seu início, dirigido para este tipo de arquitetura. Em especial, na implementação paralela baseada no algoritmo de Micali e Vazirani, os processadores compartilham todas as informações necessárias para o seu funcionamento.

A implementação paralela baseada no algoritmo de Micali e Vazirani foi precedida de uma implementação seqüencial, a qual através de um processo evolutivo foi sendo paralelizada aos poucos.

Primeiramente descreveremos a implementação seqüencial do algoritmo de Micali e Vazirani. Em seguida, descreveremos o processo de paralelização desta implementação e as dificuldades encontradas. Na seção 4.3 descrevemos a segunda implementação, com a qual foram obtidos os resultados do capítulo 5.

4.1 Implementação seqüencial de algoritmo de Micali e Vazirani

Para a obtenção de uma implementação paralela baseada no algoritmo de Micali e Vazirani adotamos um processo evolutivo. Ou seja, primeiro se desenvolveu uma implementação seqüencial e aos poucos foram introduzidos pontos de paralelização. As funções características do algoritmo seqüencial foram modificadas no sentido de facilitar ao máximo esta transição.

O grafo é armazenado numa lista de adjacência simples. Os conjuntos de predecessores, sucessores e anomalias foram implementados através de listas utilizando os elementos da própria lista de arestas de cada vértice. Cada florescência descoberta é armazenada numa lista de florescências. Nela são armazenados os picos da florescência e a sua base. Cada vértice possui a identificação da florescência à qual ele pertence.

As operações realizadas pelo algoritmo foram transformadas em *tarefas*. Uma tarefa é uma representação de uma ação que o algoritmo deve realizar e que é armazenada numa estrutura de dados como uma lista. Cada tarefa pode ser de três tipos: expandir uma busca em largura a partir de um vértice; analisar uma ponte ou tratar um caminho aumentante a partir de uma determinada ponte. No início de cada fase as únicas tarefas existentes são expansões de buscas em largura a partir de vértices livres. Estas tarefas geram novas tarefas. Por exemplo, se ao tratar uma tarefa de expansão de busca em largura for identificada uma ponte, então será criada uma nova tarefa para analisar esta ponte. A introdução de tarefas visa facilitar a introdução de pontos de paralelização, pois algumas tarefas podem ser executaras em paralelo.

As tarefas são armazenadas em níveis. Cada nível é uma lista de tarefas. Estes níveis representam os níveis de expansão das buscas em largura. Como as buscas são sincronizadas por nível, as tarefas de cada nível representam as ações que deverão ser feitas para que as buscas passem para o próximo nível. Uma tarefa pode gerar novas tarefas tanto para o seu próprio nível quanto para um outro nível. Por exemplo, ao se tratar uma tarefa de expansão de busca em largura podemos gerar uma tarefa para analisar uma ponte para o mesmo nível ou uma tarefa de expansão da busca para o próximo nível.

4.2 Implementação paralela do algoritmo de Micali e Vazirani

Para melhor controlar a paralelização das tarefas, os processadores foram numerados, $[1, 2, ...n]$, de modo que as tarefas seqüenciais fossem executadas somente pelo processador de número 1, enquanto que as demais fossem liberadas para todos o processadores. As primeiras tarefas a serem paralelizadas foram as responsáveis pela expansão das buscas em largura feita pela sub-rotina busca_pontes.

Antes da paralelização das busca em largura, estudamos a possibilidade da flexibilização deste processo. Isto significa verificar a possibilidade de realizar buscas não necessariamente em largura e não necessariamente sincronizadas. Se uma busca puder ser feita sem que seja necessário crescer em largura, isto resultaria numa maior liberdade entre os processadores, pois para prosseguir para o próximo nível não seria necessário que o nível anterior estivesse terminado. Se isto não for possível, então uma maior liberdade entre os processadores também

poderia ser alcançada se as buscas crescessem em largura, mas sem que fosse necessário sincronizá-las entre si, pois cada processador poderia ser responsável pelo crescimento de uma determinada busca e não necessitaria estar ciente sobre o nível das outras. O resultado desta maior liberdade na procura de pontes, tanto na quebra da ordem imposta pela busca em largura quanto pela quebra da sincronização entre elas, acarreta uma perda da noção de nível de busca, pois as buscas não estariam crescendo no mesmo nível. Entretanto, constatou-se que esta noção é realmente necessária, principalmente para o tratamento de anomalias, como veremos a seguir.

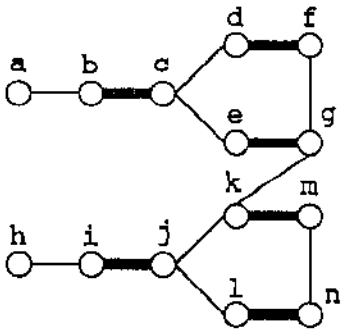

Figura 4.1: Necessidade da noção de nível de busca.

Para mostrar a necessidade da noção de nível de busca utilizaremos o exemplo da figura 4.1. Neste exemplo, se as duas buscas a partir dos dois únicos vértices livres, a e h, crescessem independentemente, a florescência $F_1 = \{d, e, f, g\}$ com base no vértice c, poderia ser formada antes da florescência $F_2 = \{k, l, m, n\}$ com base no vértice j. Quando a ponte (g, k) for analisada, duas buscas em profundidade seriam disparadas, uma em cada extremo da ponte. Entretanto, a busca que parte do vértice k, para encontrar o vértice h de forma correta, necessita da florescência F_2, pois ela saltaria para a sua $base(F_2) = j$ e seguiria até o vértice h. Posteriormente, ela constataria que o caminho deveria contornar a florescência F_2, como deveria ser o procedimento normal do algoritmo. Mas como vértice k ainda não pertence a uma florescência, a busca seguiria as marcações de predecessores, encontrando erroneamente o "caminho aumentante": $(a, b, c, e, g, k, j, i, h)$. A perda da noção de nível de busca, faz com que seja necessário tratar um número grande de exceções para evitar a identificação de caminhos errados. Diante disto, optou-se por liberar o paralelismo somente dentro de um nível de busca.

No caso das buscas em largura, o paralelismo estaria restrito somente ao nível corrente da busca. Deste modo, as tarefas foram dividida em níveis. Estes níveis correspondem aos níveis de expansão das buscas em largura. O primeiro nível contém somente as tarefas responsáveis pela expansão do primeiro nível da busca em largura. Os processadores têm livre acesso às tarefas do nível corrente da busca. O acesso ao próximo nível somente é liberado quando as tarefas do nível anterior foram todas executadas.

Uma tarefa pode ser de três tipos: expandir uma busca em largura a partir de um determinado vértice; analisar uma ponte ou tratar um caminho aumentante a partir de uma ponte. Cada processador adquire um pequeno conjunto de tarefas do nível corrente de busca para serem executadas em paralelo. Entretanto, o único tipo de tarefa que é permitido ser executada em paralelo é a responsável pela expansão da busca em largura. As outras somente são executadas pelo processador de número 1. As tarefas de busca em largura são executadas de duas formas distintas dependendo se o nível da busca em largura for par ou ímpar.

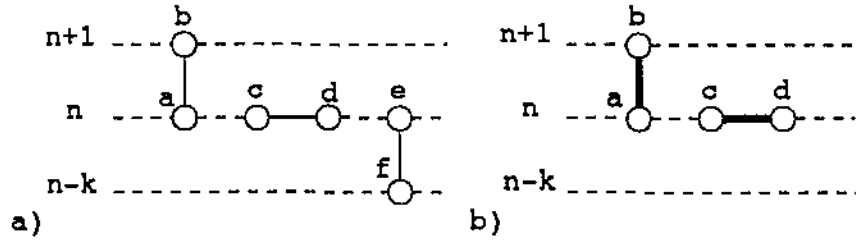

Figura 4.2: a) Nível par b) Nível ímpar.

Se o nível corrente da busca em largura for par então a expansão ocorrerá por arestas não emparelhadas. Estas arestas podem ser de três tipos: arestas que ligam um vértice do nível atual a um vértice do próximo nível ou do mesmo nível ou, ainda, a um nível inferior. Na figura 4.2a podemos observar os três tipos de expansões. No caso de se expandir a busca pelo vértice a então o *nível_ímpar* do vértice b recebe $n+1$ e é criada uma tarefa no nível $n+1$ para expandir a busca a partir do vértice b. Caso haja a concorrência de mais de um processador atuando sobre o vértice b, isto não acarretará problemas, pois o seu *nível_ímpar* terá obrigatoriamente o mesmo valor e a criação da tarefa para o próximo nível é feita localmente, e depois inserida no nível $n+1$ de tarefas através de uma área de exclusão mútua. Se a expansão for feita a partir do vértice c, então será observado que o vértice d pertence ao mesmo nível de c, deste modo, somente é criada uma tarefa de análise de ponte. Se a expansão for feita a partir do vértice e, então será observado que o vértice f pertence a um nível inferior. A única possibilidade disto acontecer é o vértice e ser uma anomalia do vértice f. Portanto, a única ação a ser feita é a criação de uma tarefa de análise da ponte (e, f).

Se o nível da busca em largura for ímpar então a expansão ocorrerá através de arestas emparelhadas. Elas tarefas podem ser de dois tipos: arestas que ligam um vértice do nível corrente ao próximo nível ou ao mesmo nível. Na figura 4.2b observam-se os dois tipos desta tarefa. Se a expansão for feita a partir do vértice a então o *nível_par* do vértice b recebe $n+1$ e é criada uma tarefa para expandir a busca a partir do vértice b. Se a busca for expandida a partir do vértice c então a única ação a ser feita é a criação de uma tarefa de análise da ponte (c, d).

As buscas prosseguem sincronizadas por nível até o final da fase. Quando isto

acontece, o grafo é inicializado em paralelo. Os vértices do grafo são distribuídos uniformemente entre os processadores e as suas informações são inicializadas em paralelo preparando o grafo para a próxima fase.

A próxima etapa do processo de paralelização é a realização das buscas em profundidade, executadas pela sub-rotina procura_caminho. O estudo foi direcionado para fazer com que cada processador fosse encarregado por uma determinada ponte e executasse as buscas em profundidade concorrentemente com os outros processadores. O resultado deste estudo mostrou que, apesar das buscas em profundidade serem executadas em locais diferentes do grafo, os casos em que há interseção entre elas são de difícil tratamento.

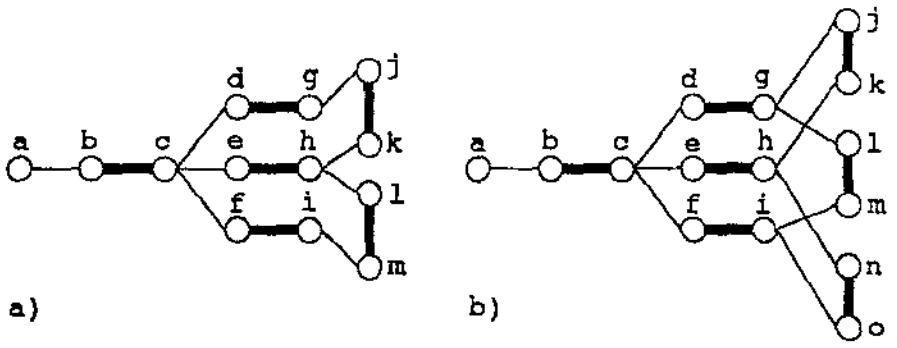

Figura 4.3: a) Utilização de bloqueio b) Situação de *Deadlock*.

Na figura 4.3a observamos a formação de duas florescências. A formação de uma interfere na formação da outra. Se a florescência $F_1 = \{d, e, g, h, j, k\}$ for formada primeiro, então a florescência F_2 conterá os vértices $\{f, i, l, m\}$, caso contrário teremos a florescência F_1 com os vértices $\{d, g, j, k\}$ e F_2 com $\{e, f, h, i, l, m\}$. Caso dois processadores tratem as duas pontes (j, k) e (l, m) ao mesmo tempo, será necessário que eles bloqueiem os vértices que forem sendo visitados. Deste modo, o processador que bloquear o vértice h primeiro, definirá a qual florescência pertencerão os vértices $\{e, h\}$. Esta técnica, no entanto, acarreta situações de *deadlock*. Na figura 4.3b, se três processadores tratarem as três pontes, (j, k), (l, m) e (n, o) ao mesmo tempo e bloquearem respectivamente os vértices g, i, h, eles ficarão bloqueados, pois como as duas buscas em profundidade de cada processador devem descer sincronizadamente, o bloqueio dos vértices pela segunda busca de cada um ficará comprometido. O tratamento deste tipo de situação não é trivial devido às inúmeras possibilidades e variações deste tipo de colisão.

Outro problema encontrado nesta tentativa de paralelização foi o aumento excessivo de tarefas criadas. Isto será mostrado através do exemplo da figura 4.4. Neste exemplo, a ordem na criação das tarefas, se o algoritmo fosse seqüencial, seria: identificar a ponte (g, h), trocar as arestas do caminho aumentante contendo esta ponte e apagar os seus vértices. Isto faz com que a ponte (h, i) não

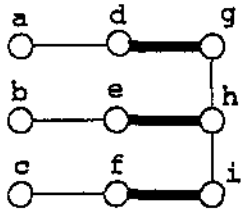

Figura 4.4: Aumento do número de tarefas.

seja tratada, pois o vértice h está apagado. No caso do algoritmo paralelo, observamos que no segundo nível de expansão da busca em largura existem três tarefas de expansão a partir dos vértices g, h e i. Se elas forem feitas em paralelo, isto ocasionará a identificação da ponte (g,h) duas vezes, uma pela tarefa do vértice g e outra pela tarefa do vértice h. O mesmo ocorre para a ponte (h,i). O tratamento de uma ponte inviabiliza o tratamento das outras, entretanto o trabalho de identificar pontes supérfluas já foi feito. Este aumento no número de tarefas de tratamento de pontes prejudica a performance da implementação.

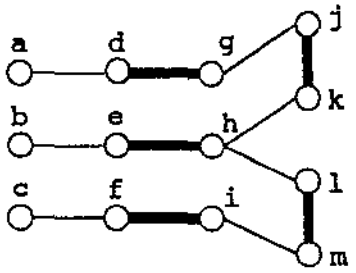

Figura 4.5: Incoerência aceita pela estrutura de dados.

Durante o estudo da paralelização das tarefas de busca em profundidade, observou-se que a estrutura de dados utilizada para armazenar o emparelhamento admite incoerências, como duas arestas emparelhadas possuírem um vértice em comum. Na figura 4.5 o desejável seria tratar as duas pontes (j,k) e (l,m) independentemente, sem que um processador tome conhecimento do que está ocorrendo na outra ponte. Isto poderia ser implementado possibilitando que cada processador armazenasse as próprias marcações de vértices-pai. Entretanto, se isto for feito, o vértice h ficaria emparelhado com os vértices k e l ao mesmo tempo. Isto acontece porque, na lista de adjacência, os nodos que representam as arestas indicam se elas estão, ou não, emparelhadas.

A impossibilidade de paralelização eficiente das buscas em profundidade comprometeu a paralelização das tarefas de tratamento de caminhos aumentantes, pois a segunda é decorrente da primeira. Com isto, não se obteve uma implementação paralela eficiente na prática devido à complexidade do algoritmo e do seu tratamento em paralelo.

4.3 Implementação paralela do algoritmo de Edmonds

A implementação paralela do algoritmo de Micali e Vazirani apresentou um conjunto de ineficiências que impossibilitaram a obtenção de uma implementação eficiente na prática. Após uma análise destas ineficiências, constatou-se que uma alternativa viável seria, não uma cooperação dos processadores para a execução de um algoritmo, mas sim que cada um executasse um algoritmo independente, com seus próprios dados. Deste modo, as buscas por caminhos aumentantes não mais precisarão ser sincronizadas por nível de expansão e a atualização do emparelhamento poderá ser feita a qualquer momento, ao invés de ser feita somente ao final de uma fase. Nesta seção descrevemos as idéias que resultaram numa nova implementação.

4.3.1 Visão geral

A nova proposta de paralelização consiste em permitir que cada processador procure caminhos aumentantes de forma independente e assíncrona. Embora a busca ocorra de forma paralela, o aumento do emparelhamento é feito por apenas um processador por vez, o que garante a corretude do algoritmo sem incorrer em atraso significativo no tempo de execução na prática.

A estrutura de dados escolhida para armazenar o emparelhamento é um vetor de vértices, o qual chamamos C. Se os vértices u e v estão emparelhados, então $C(u) = v$ e $C(v) = u$. Esta estrutura é a mesma utilizada pelo algoritmo de Gabow e foi escolhida porque ela não admite incoerências, tais como um vértice possuir duas arestas emparelhadas incidentes sobre ele.

Uma vez que nosso objetivo é prático, a implementação incorpora uma rotina inicial que visa obter, em paralelo, um emparelhamento inicial a um baixo custo computacional. Este emparelhamento é obtido dividindo os vértices uniformemente entre os processadores. Cada processador analisa seus vértices e, para cada vértice livre i identificado, é verificado se existe um outro vértice livre j adjacente a i. Se existir então a aresta (i, j) é emparelhada fazendo $C(i) = j$ e $C(j) = i$. Durante a execução deste procedimento, várias situações de concorrência entre os processadores poderão ocorrer. Um exemplo é quando um processador encontra dois vértices livres adjacente e antes de emparelhá-los, um outro processador modifica o emparelhamento tornando um dos dois vértices livres emparelhado. Quando o primeiro processador for atualizar o emparelhamento, um dos dois vértices não será mais livre. Entretanto, esta situação não acarreta problema algum, pois a estrutura de armazenamento do emparelhamento não admite incoerências. Deste modo, nenhum cuidado especial necessita ser tomado. Testes empíricos mostraram que esta rotina inicial já alcança pelo menos 80% do emparelhamento máximo (o que não é surpresa, visto que qualquer emparelhamento maximal alcança pelo menos 50% do máximo). Resta à segunda parte da imple-

mentação tentar emparelhar os restantes 20%, e esta é a parte que passaremos a descrever.

Após a aplicação do procedimento paralelo para a obtenção do emparelhamento inicial, é iniciada a execução do algoritmo paralelo propriamente dito. A primeira ação consiste em novamente dividir os vértices uniformemente entre os processadores. Cada processador p_i analisa seus vértices e constrói uma lista ligada l_i, onde cada elemento armazena um vértice livre identificado. As listas l_i de cada processador são unidas numa lista global L, assim que forem sendo terminadas. Ao final, todos os vértices livres do grafo que restaram após a obtenção do emparelhamento inicial estarão representados na lista L.

O trabalho de um processador p_i consiste em adquirir um vértice livre da lista L. Isto é feito numa região de exclusão mútua para garantir que somente um único processador esteja realizando uma busca a partir de um vértice livre. Por outro lado, mais de um processador pode estar realizando sua respectiva busca ao longo das mesmas arestas. Isso é possível pois as informações de cada busca são exclusivas de cada processador.

Quando um processador encontra um caminho aumentante, ele procura realizar a troca das arestas. Porém, somente um processador pode estar realizando a troca num dado instante. Esta exclusividade é necessária para garantir que o emparelhamento encontrado seja máximo.

Pelo fato da execução ser em paralelo, é possível que um processador encontre inconsistências no grafo.

Definição 5 *Uma* inconsistência *é uma situação onde um processador supõe que uma determinada aresta esteja emparelhada e ela não está, e vice-versa.*

Uma inconsistência pode ocorrer durante a busca por caminhos aumentantes ou durante a troca das suas arestas. Em ambos os casos a busca é refeita. Na seção 4.3.2 apresentamos uma demonstração de que com estas características o algoritmo está correto.

Em seguida apresentamos o algoritmo que propomos, na forma de pseudo-código:

INTERAÇÃO ENTRE OS PROCESSADORES

```
Cada processador em paralelo faça:
    C ← Emparelhamento Inicial
    Sincronismo
    L ← Lista de Vértices Livres
    Sincronismo
    Enquanto L ≠ ∅
        v ← Retira (L)
        Terminado ← Falso
        Enquanto não Terminado e v continua livre
            r1 ← Procura_Caminho (v)
            Se r1 = Insucesso
                Terminado ← Verdadeiro
                "Não foi encontrado caminho a partir de v"
            Senão Se r1 = Sucesso
                Entra em Área de Exclusão Mútua
                r2 ← Aumenta (v)
                Sai da Área de Exclusão Mútua
                Se r2 = Sucesso
                    Terminado ← Verdadeiro
                "Senão: uma inconsistência foi detectada
                no aumento do emparelhamento:
                é preciso refazer a busca"
            "Senão: uma inconsistência foi detectada
            na busca por caminho aumentante;
            é preciso refazer a busca"
Fim do Algoritmo
```

4.3.2 Demonstração de corretude

Nesta seção demonstramos que o algoritmo proposto sempre encontra um emparelhamento máximo. Considerando que o que cada processador faz é executar sua "cópia" do algoritmo de Edmonds, e que cada emparelhamento é alterado de forma exclusiva, o seguinte lema é suficiente para demonstrar a corretude.

Lema 5 *Se um processador não consegue encontrar um caminho aumentante a partir de um vértice v, e tampouco encontra inconsistências durante a busca, um*

tal caminho não existe, mesmo levando-se em conta que o emparelhamento pode estar sendo alterado por outros processadores durante a busca.

Demonstração: Dado um vértice livre v e a árvore de busca A a partir de v, chamamos de vértices vizinhos de fronteira de A todo vizinho de um vértice ímpar de A (após todas as contrações de flor que se fizerem necessárias).

É fácil ver que vértices vizinhos de fronteira não pertencem a A (isto é, não são visitados a partir de v).

Suponha agora que o processador i não encontrou um caminho aumentante a partir de v e nem inconsistências, mas um tal caminho existe. Considere a árvore de busca A a partir de v. Se o caminho existe, ele necessariamente chega em v através de algum vértice vizinho de fronteira de A, digamos f. Considere agora a situação vigente durante a formação de A. A aresta que une f a $u \in A$ era uma aresta não emparelhada quando u foi visitado. Portanto, a busca prosseguiu pela aresta emparelhada incidente em u, digamos (u, w). Considere agora a descoberta do caminho aumentante e sua conseqüente alteração por um outro processador j. A aresta (f, u) pertence ao caminho, por hipótese, e será alterada de não-emparelhada para emparelhada. A aresta seguinte do caminho deverá ser (u, w), e mudará de emparelhada para não-emparelhada. Vemos portanto que a alteração do caminho e a busca que forma A *seguem no mesmo sentido.* Agora duas situações podem ocorrer: ou as alterações ficam sempre "atrás" da busca, ou "ultrapassam" a busca. Se houver ultrapassagem, necessariamente a busca revelará uma inconsistência, e essa possibilidade está excluída por hipótese. Se as alterações permanecerem atrás, então a própria busca encontrará de novo o vértice v, o que vale dizer que v pertence a uma flor. Mas esta situação é impossível pelo fato de f ser um vértice vizinho de fronteira de A. Portanto tal caminho não existe. □

4.3.3 Detalhamento da implementação

Nesta seção detalhamos aspectos relativos ao processo de busca de caminhos aumentantes feito por um processador. Esta descrição supõe que a implementação está sendo executada seqüencialmente. O algoritmo consiste na aplicação do algoritmo de Edmonds, com algumas modificações obtidas a partir do algoritmo de Micali e Vazirani. Na seção seguinte descreveremos os problemas e as mudanças necessárias quando a execução ocorre em paralelo.

A implementação possui três sub-rotinas principais (além da rotina de emparelhamento inicial), que são: procura_caminho, constrói_flor e identifica_caminho. A sub-rotina procura_caminho é responsável pela expansão da busca no grafo. A expansão da busca é feita até que seja encontrado um caminho aumentante ou até que a busca não consiga visitar mais nenhum vértice. Durante a expansão, a sub-rotina poderá identificar flores (definidas na seção 2.3). Cada vez que uma flor é identificada, a sub-rotina constrói_flor é chamada para tratá-la adequadamente.

No final da expansão da busca, se um caminho aumentante for encontrado a sub-rotina identifica_caminho é chamada para identificar as arestas que compõem o caminho e para aumentar o emparelhamento.

A expansão da busca, a partir de um vértice livre r, é feita de maneira idêntica a do algoritmo de Edmonds. O tratamento dado às flores foi baseado, em parte, no algoritmo de Micali e Vazirani. Embora este algoritmo não utilize o conceito de flor, como é definido no algoritmo de Edmonds, os seus conceitos foram adaptados para a nova situação.

Para a sub-rotina procura_caminho controlar a expansão da busca, ela mantém uma lista R dos vértices que pertencem à fronteira da árvore de busca. Os vértices nesta lista são rotulados *par* ou *ímpar*, e um vértice v não poderá ser inserido na lista se ele já estiver presente nela. Inicialmente, o único vértice pertencente à lista R é o vértice livre r, o qual é marcado como *visitado*. Ele será a raiz da busca e possui rótulo par.

A sub-rotina procura_caminho inicia a busca retirando um vértice v da lista R. Se o vértice v possui rótulo par então, para cada vértice u adjacente a v tal que a aresta (u, v) não está emparelhada é feita a seguinte verificação. Se o vértice $u \neq r$ é um vértice livre então foi identificado um caminho aumentante. No entanto, se o vértice u estiver emparelhado, então a busca deve ser expandida para ele. Isto é feito marcando o vértice v como *predecessor* de u, rotulando u com ímpar e também como visitado e, finalmente, inserindo o vértice u na lista de vértices R. Agora, se o vértice v, retirado da lista, possui rótulo ímpar, então a aresta emparelhada (v, u) é analisada e a única ação a ser feita é expandir a busca. Isto é feito marcando o vértice v como predecessor de u, rotulando u com par e também como visitado e, finalmente, inserindo o vértice u na lista de vértice R. A busca prossegue desta maneira até encontrar algum caminho aumentante ou até acabarem os vértices da lista R. Um resumo em forma de pseudo-código da sub-rotina procura_caminho está descrito abaixo.

PROCURA_CAMINHO

```
Enquanto R ≠ ∅
    v ← Retira (R)
    Se v é Par
        Para cada w adjacente a v faça:
            Se (v, w) não está emparelhada
                Se w é um vértice livre e
                w não é raiz da árvore de busca
                    identifica_caminho (w,raiz)
                Senão Se w está Visitado
                    Se w é Par
                        constrói_flor (v,w)
                Senão, expandir a busca para w
    Senão "v é Ímpar"
        w ← C(v)
        Se w está Visitado
            Se w é Ímpar
                constrói_flor (v,w)
        Senão, expandir a busca para w
Fim da Sub-rotina
```

Como foi descrito anteriormente, durante a expansão da busca, é possível encontrar flores. Por isto, a cada expansão, a sub-rotina procura_caminho verifica se é possível atribuir um rótulo par a um vértice visitado com rótulo ímpar ou atribuir um rótulo ímpar a um vértice visitado com rótulo par. Se esta condição for satisfeita então a estrutura encontrada é a mesma descrita no algoritmo de Edmonds e possui todas as propriedades descritas na seção 2.3.2.

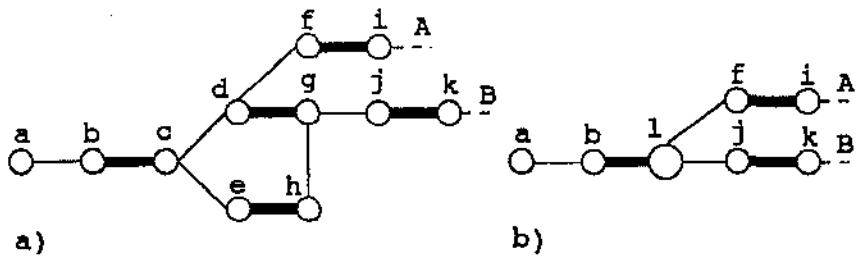

Figura 4.6: a) Identificação de uma flor b) Contração da flor.

O tratamento dado à flor pela sub-rotina constrói_flor, com já foi dito, é dife-

rente do tratamento dado pelo algoritmo de Edmonds, e foi baseado, em parte, no algoritmo de Micali e Vazirani. No algoritmo de Edmonds, quando uma flor é identificada, ela é contraída sobre a sua base, criando-se um pseudo-vértice. As arestas que antes eram incidentes sobre os vértices da flor agora são incidentes sobre esse pseudo-vértice. A expansão continua normalmente a partir do pseudo-vértice. Na figura 4.6a observamos a identificação da flor $F = \{d, e, g, h\}$ com base o vértice c. O algoritmo de Edmonds contrai esta estrutura no pseudo-vértice l da figura 4.6b. A busca prossegue normalmente através dos caminhos A e B. O tratamento dado pela sub-rotina constrói_flor se apóia nas seguintes observações: contrair uma flor equivale a rotulagem de seus vértices com rótulo par. Isto deve-se ao fato de que as arestas que incidem sobre uma flor são obrigatoriamente não emparelhadas e a expansão por arestas não emparelhadas se dá por vértices com rótulo par. O tratamento dado pela sub-rotina contrói_flor é: rotular os vértice ímpares com rótulo par e inseri-los novamente na lista de vértices R para serem novamente reavaliados, expandindo, assim, a busca através deles. Os vértices que tinham rótulos pares não são modificados, pois a busca prosseguirá a partir deles naturalmente. Na figura 4.6a, os vértices d e e são rotulados com rótulo par e re-inseridos na lista R, fazendo com que o caminho A seja avaliado. O caminho B será avaliado naturalmente, pois o vértice g possui rótulo par. Para a identificação dos vértices que compõem a flor e para a rotulagem par dos vértices com rótulo ímpar, é necessário que haja uma numeração dos vértices em ordem crescente a partir da raiz da árvore de busca. A cada expansão da busca, o vértice é numerado com o valor do predecessor acrescentado de 1. Com isto, na formação de uma flor, a base possui o menor valor entre os seus vértices, pois os vértices da flor foram visitados pela busca que partiu da base da flor. A explicação de como a sub-rotina constrói_flor identifica os vértices da flor (processo baseado no algoritmo de Micali e Vazirani) será feita utilizando o exemplo abaixo.

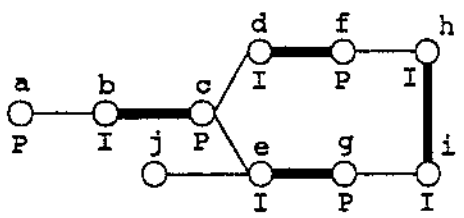

Figura 4.7: Tratamento dado pela sub-rotina constrói_flor.

Na figura 4.7, foi identificada uma flor a partir da análise do vértice i, pois é possível atribuir um rótulo par ao vértice h, o qual possui rótulo ímpar. Definimos a aresta (h, i) como sendo uma *ponte* e os vértices h e i como sendo os *picos* da flor. Eles pertencem ao quinto nível de expansão. Um pico é rotulado com *direito* e outro com *esquerdo*. A partir destes dois vértices são disparadas duas buscas em profundidade, uma direita e outra esquerda, seguindo as marcações

de predecessores. Cada vértice visitado por elas é rotulado como pertencente à busca direita ou esquerda. As buscas são sincronizadas pelos valores contidos nos vértices e a busca esquerda tem prioridade sobre a direita, caso elas estejam sobre vértices com o mesmo valor. O vértice onde ocorrer o encontro entre as buscas é a base da flor e os caminhos percorridos contêm os vértices pertencentes a ela. No caso da figura 4.7, elas encontrarão o vértice c, que pertence ao segundo nível de expansão. Se o pico h for o ponto de partida da busca esquerda então os vértices d e f serão marcados como esquerdos, o mesmo para os vértices e e g em relação ao pico i. Os vértices com rótulos ímpares, $\{d, e, h, i\}$, são marcados como pares e inseridos novamente na lista de vértices R. O vértice e, quando reavaliado, marcará o vértice j como ímpar e identificará a presença de um caminho aumentante. Um resumo da sub-rotina constrói_flor está descrito abaixo em forma de pseudo-código.

CONSTRÓI_FLOR (s, t)

Se s e t pertencem a mesma flor
    Retorne
Se s pertence a uma flor
    direita $\leftarrow$ $base^*(flor(s))$
Senão
    direita $\leftarrow$ s
Se t pertence a uma flor
    esquerda $\leftarrow$ $base^*(flor(t))$
Senão
    esquerda $\leftarrow$ t
Se direita = esquerda
    Retorne
Seja F uma nova flor com os picos s e t
Enquanto direita $\neq$ esquerda faça:
    Se $nível$(esquerda) $\geq$ $nível$(direita)
        adicione direita a F
        Se direita é Ímpar
            adicione direita a R
        direita $\leftarrow$ predecessor (direita)
        Se direita pertence a uma flor
            direita $\leftarrow$ $base^*(flor$(direita))
    Senão
        adicione esquerda a F
        Se esquerda é Ímpar
            adicione esquerda a R
        esquerda $\leftarrow$ predecessor (esquerda)
        Se esquerda pertence a uma flor
            esquerda $\leftarrow$ $base^*(flor$(esquerda))
$base(F)$ $\leftarrow$ direita
Fim da Sub-rotina

Quando um caminho aumentante é identificado a sub-rotina identifica_caminho é chamada com dois vértices como parâmetros. Um deles é o vértice livre recém descoberto, o qual chamamos de vértice alto, e o outro é a raiz da árvore de busca r, o qual chamamos de vértice baixo. A sub-rotina identifica_caminho é responsável pela construção do caminho entre estes dois vértices. Para auxiliar

neste processo, os vértices são rotulados como *externos* e *internos* durante a construção da busca. Um vértice é rotulado interno (externo) se o caminho entre ele e a raiz da busca, seguindo as marcas de predecessores, tem comprimento ímpar (par). O objetivo da sub-rotina é partir do vértice alto e, seguindo as marcações de predecessores, encontrar o vértice baixo. Se, durante a descida, ele encontrar um vértice v pertencente a uma flor F, a sub-rotina identifica_caminho chama uma pequena sub-rotina chamada abre_flor e passa como parâmetro o vértice v pertencente à flor. Um resumo em forma de pseudo-código da sub-rotina identifica_caminho está descrito abaixo.

IDENTIFICA_CAMINHO (alto, baixo)

```
Vértice ← alto
Enquanto Vértice ≠ baixo
        Vértice ← Predecessor (Vértice)
        Se Vértice pertence a uma flor
                Vértice ← abre_flor (Vértice)
Fim da Sub-rotina
```

A sub-rotina abre_flor é responsável pela identificação do caminho através de uma flor. Para isto, ela, ao receber o vértice v como parâmetro, verifica se ele é externo ou interno. Se for externo então o caminho parte de v e segue até a base da flor F. Neste caso a sub-rotina abre_flor chama a sub-rotina identifica_caminho recursivamente com os parâmetros v e $base(F)$ respectivamente. Se for interno, o caminho passa através dos picos da flor. Neste caso, se o vértice v possui rótulo direito (esquerdo) então a sub-rotina abre_flor chama a sub-rotina identifica_caminho duas vezes, uma para encontrar o caminho entre o pico direito (esquerdo) da flor F e o vértice v e outra para encontra o caminho entre o pico esquerdo (direito) e $base(F)$, contornando a flor F.

Na figura 4.7 a sub-rotina identifica_caminho é chamada para identificar as arestas que pertencem ao caminho aumentante entre os vértices j e a. A sub-rotina parte do vértice j e verifica que o vértice e pertence a uma flor F. Neste ponto ela chama a sub-rotina abre_flor com o vértice e como parâmetro. Por sua vez, a sub-rotina abre_flor verifica que o vértice e é externo, portanto o caminho passa através da ponte (h, i). Como o vértice e possui a mesma marcação de direita ou esquerda do vértice i, a sub-rotina abre_flor chama a sub-rotina identifica_caminho duas vezes, uma para encontrar o caminho entre o vértice i e o vértice e, a outra chamada é para o caminho entre o vértice h e $base(F) = c$. A busca prossegue a partir do vértice c até encontrar a raiz da árvore de busca a. Um resumo da sub-rotina abre_flor em forma de pseudo-código está apresentado abaixo.

ABRE_FLOR (v)

$F \leftarrow flor(v)$
Se v é Externo
    identifica_caminho ($base(F)$, v)
Senão
    Se v é marcado como Direita
        identifica_caminho (v, Pico_Direito (F))
        identifica_caminho ($base(F)$, Pico_Esquerdo (F))
    Senão
        identifica_caminho (v, Pico_Esquerdo (F))
        identifica_caminho ($base(F)$, Pico_Direito (F))
Fim da Sub-rotina

4.3.4 Execução em paralelo

Conforme mencionado, a atualização do emparelhamento corrente após a descoberta de um caminho aumentante só pode ser feita por um processador. A seguir descrevemos a razão dessa exclusividade, mostrando o que pode ocorrer se ela não existir.

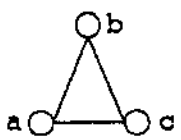

Figura 4.8: Caso de concorrência entre os processadores.

Na figura 4.8 observamos um caso em que o emparelhamento pode não crescer. Os três caminhos aumentantes são identificados ao mesmo tempo e as atualizações podem seguir a seguinte ordem: num primeiro passo, $C(a) = b$, $C(b) = c$ e $C(c) = a$; num segundo passo, $C(a) = c$, $C(b) = a$ e $C(c) = b$. Portanto, nenhuma aresta foi emparelhada.

Quando o algoritmo é executado em paralelo, é possível que um vértice que não era livre passe a sê-lo (dizemos que houve uma *troca*). Na figura 4.9, observamos que somente é possível tratar um caminho aumentante. Porém, é possível que dois processadores identifiquem dois caminhos distintos, um entre os vértices a e i e outro entre b e j, e que tentem tratá-los ao mesmo tempo. Quando ocorrer as alterações no emparelhamento temos $C(e) = g$ e $C(f) = g$. A atualização de

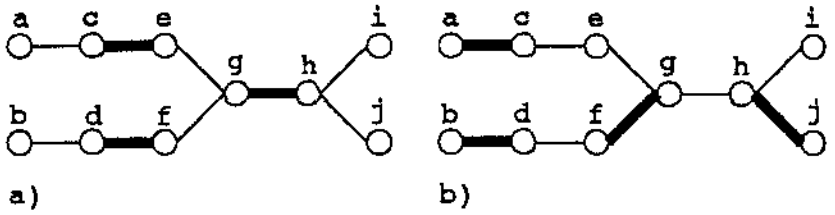

Figura 4.9: a) Identificação de dois caminhos. b) Troca de vértice livre.

$C(g)$ definirá qual aresta estará emparelhada. Isto é feito pelo último processador a atualizar o campo. Supondo que $C(g) = f$, observamos que o vértice e torna-se livre.

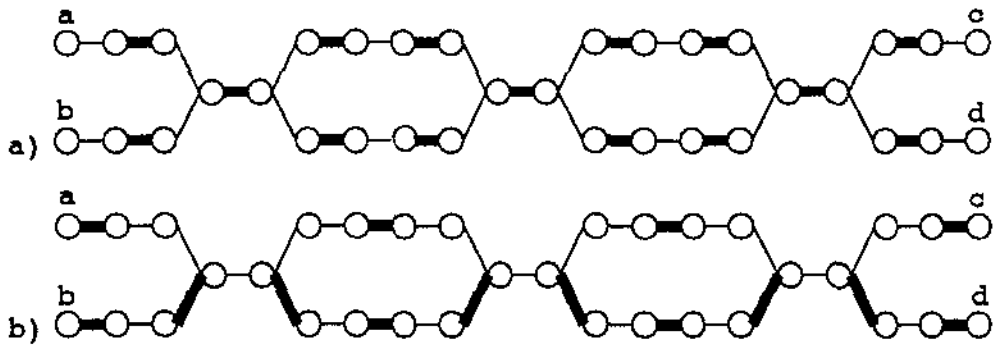

Figura 4.10: a) Identificação de dois caminhos. b) Após a atualização.

Além de "troca" de vértices livres, é possível haver a criação de novos vértices livres. Na figura 4.10, quando dois processadores atualizam os dois caminhos de a a c (passando pelos arcos superiores) e de b a d (passando pelos arcos inferiores) ao mesmo tempo, ocorre a criação de dois vértices livres. Antes tínhamos 15 arestas emparelhadas e depois da atualização obtemos somente 14, como mostra a figura 4.10b, ocorrendo, portanto, uma diminuição do emparelhamento. Além do mais é possível haver a criação de vértices livres que estejam adjacentes. Neste caso é possível que o emparelhamento gerado não seja sequer maximal.

Estes problemas são solucionados aplicando-se uma área de exclusão mútua quando um caminho aumentantes é identificado. Deste modo somente um processador pode atualizar o emparelhamento num determinado instante. Entretanto, os outros processadores continuam executando as buscas. Isto faz com que haja uma interferência da atualização do emparelhamento com as buscas dos outros processadores. Esta interferência faz com que seja necessário um tratamento especial por parte do algoritmo.

Devido às constantes mudanças do emparelhamento, devemos tomar cuidado nas expansões das buscas. Não é possível mais confiar nas informações das arestas emparelhadas. Anteriormente definimos que quando se expande uma busca a partir de um vértice v com rótulo par, deve-se analisar todos os vértices adjacentes

u, cujas arestas (v, u) não estejam emparelhadas. Num algoritmo seqüencial, a aresta emparelhada representa o caminho de onde veio a busca, mas em paralelo isto pode não ser verdade. Devemos, portanto, analisar todos os vértices u cujas arestas (v, u) não estejam emparelhadas e tal que o vértice u não esteja marcado como predecessor de v. Quando o vértice v tem rótulo ímpar deve-se expandir a busca para a aresta emparelhada (v, u) somente se o vértice u não está marcado como predecessor de v. Se a expansão se der através de um predecessor então foi encontrada uma situação de inconsistência.

Um processador pode detectar uma inconsistência em 2 momentos: durante uma busca, ou durante a efetivação de um caminho aumentante. Para lidar com as inconsistências adotamos a solução simples de obrigar o processador a refazer a busca a partir do vértice livre original. Com esta solução, o único problema que ainda poderia haver seria no caso de um processador não encontrar um caminho aumentante e nem inconsistências. Na seção 4.3.2 mostramos que esse caso não causa problemas, e portanto o algoritmo sempre acha um emparelhamento máximo.

A aplicação da área de exclusão mútua para aumento do emparelhamento e a necessidade de refazer certas buscas não produziram atrasos significativos em nossos testes. Segundo nossas medições, o trabalho necessário para tratar um caminho aumentante é muito menor do que o trabalho necessário para tratar flores e as expansões das buscas, e o número de buscas que precisam ser refeitas também é pequeno. Maiores detalhes estão no capítulo 5.

As informações necessárias num processador são: o predecessor de cada vértice, o valor atribuído durante a expansão da busca, as informações booleanas (par, ímpar, interno, externo, direita, esquerda, visitado e não visitado) e a identificação da flor a qual cada vértice pode pertencer. Cada flor é identificada por um elemento de uma lista ligada, onde são armazenados os picos e a base da flor. A memória necessária para cada processador é de $O(n)$. Portanto, a execução do algoritmo ocupa um espaço total de $O(pn)$, onde p é o número de processadores utilizados.

Capítulo 5

Resultados Computacionais

5.1 A máquina

A máquina utilizada é uma Sun Sparc Server 1000 com 8 processadores, com sistema operacional Solaris. A implementação paralela foi feita utilizando *threads* (novos pontos de execução sobre o mesmo código). Uma descrição do uso de *threads* está contido no Apêndice A. É possível criar tantos *threads* quantos forem necessários. A implementação procura fazer com que os *threads* cumpram o papel de processadores físicos, associando 1 *thread* a cada processador e mantendo-os durante todo o programa. Infelizmente o sistema operacional impede que uma vinculação física se realize, e portanto no decorrer da execução 1 *thread* pode ser re-escalonado para outro processador.

A função usada para medir os tempos de execução é `gethrtime`. Ela mede o tempo de relógio. Portanto, o tempo medido inclui o tempo gasto no escalonamento dos *threads*, além do tempo gasto em filas de acesso à áreas de exclusão mútua. Este tempo também pode ser influenciado pelos processos de outros usuários e pelos processos do sistema operacional. Estas interferências produzem um aumento do tempo de espera pelo escalonamento dos *threads* pelo sistema operacional. Para minimizar esta interferência, os teste foram feitos com a máquina totalmente dedicada às implementações, e com os *threads* executados em *real time*.

Resultados preliminares do desempenho da implementação paralela em comparação com uma implementação do algoritmo de Micali e Vazirani foram descritos em [BS95].

5.2 A metodologia

A implementação paralela foi comparada com duas implementações seqüenciais. A primeira é uma implementação do algoritmo de Micali e Vazirani (*MV*) feita em linguagem FORTRAN por Mattingly e Ritchey [MR93]. A segunda é uma

implementação do algoritmo de Gabow (G) feita em linguagem C por Edward Rothberg [MR93]. Ambas as implementações foram obtidas do repositório do centro DIMACS [Dim93].

Tanto a implementação do algoritmo de Gabow quanto a de Micali e Vazirani possuem uma sub-rotina inicial para a obtenção de um emparelhamento inicial. Esta sub-rotina ordena os vértices por ordem crescente de grau. Estes vértices ordenados são visitados e, para cada vértice livre encontrado, tenta-se emparelhá-lo com um outro vértice livre adjacente. A heurística para emparelhamento inicial usada pela implementação paralela não usa ordenação. Nela, faz-se uma varredura simples dos vértices, em paralelo, e para cada vértice procura-se emparelhá-lo com algum vizinho ainda livre (é uma heurística "gulosa"). Em geral a heurística com ordenação consegue emparelhamentos iniciais de maior cardinalidade do que a heurística "gulosa". Este é um fato importante a se considerar nas comparações que são apresentadas a seguir, pois significa que, em geral, a implementação dos algoritmos de Gabow e a de Micali e Vazirani partem de emparelhamentos iniciais de maior cardinalidade do que a implementação paralela. Por esse motivo presumivelmente as implementações de Gabow e a de Micali e Vazirani realizam menos trabalho do que a implementação paralela. Infelizmente não coletamos dados quantitativos para fundamentar esta observação.

A implementação paralela foi executada com 1, 2, 4, 6 e 7 *threads*. Os *threads* são executados com alta prioridade, pois são setados para serem executados com status de *real time*. Apesar do equipamento possuir 8 processadores, não podemos utilizar todos, pois, como os processos são executados com alta prioridade, um processador deve estar disponível para executar as chamadas ao sistema operacional.

As medições dos tempos dos códigos, tanto seqüencial quanto paralelo, foram feitas sobre cinco instâncias de cada um dos grafos gerados. Cada instância foi gerada com uma semente diferente para o gerador de grafos. Os tempos das implementações seqüenciais são médias das cinco instâncias mencionadas acima. Estas mesmas instâncias são usadas para medir os tempos da implementação paralela através do uso da mesma semente geradora.

No caso paralelo, para um determinado número de *threads*, cada instância foi resolvida 3 vezes, pois o tempo de execução apresenta uma certa variação entre uma execução e outra, mesmo com a mesma entrada. Tomou-se a média das 3 execuções, e o tempo paralelo para um grafo é a média das médias de cada uma das 5 instâncias.

5.3 Geradores de instâncias

Para avaliar a implementação paralela utilizamos quatro geradores de grafos randômicos, que denominamos de: Randômico (R), Grade (G), Anel (A) e Bipartido (B). Cada um deles constrói grafos com uma estrutura interna diferente. O

gerador Randômico constrói grafos onde as arestas e vértices não obedecem a uma ordem definida, pois as arestas são ligadas de uma maneira randômica. O gerador Grade constrói grafos em formato de grade bidimensional ou tridimensional. No caso bidimensional os vértices são dispostos em linhas e colunas, como numa matriz, e as arestas somente ligam vértices vizinhos nas linhas ou nas colunas. Deste modo, cada vértice poderá ter no máximo grau 4. No caso tridimensional o grau máximo de um vértice é 6. O gerador Anel constrói grafos que apresentam subgrupos de vértices na sua estrutura. Os vértices de cada subgrupo são ligados randomicamente sem uma ordem definida. Entretanto, a ligação entre os subgrupos obedece uma ordem específica. Eles são ligados entre si através de um número pequeno de arestas formando um anel de subgrupos. O gerador Bipartido constrói grafos bipartidos que apresentam subgrupos de vértices ligados entre si formando um anel. As arestas somente fazem a ligação entre os grupos e o número de grupos tem que ser par para manter válidas as características de grafo bipartido.

Cada grafo gerado é passado através de um embaralhador desenvolvido por nós. A numeração dos vértices é embaralhada assim como a ordem de aparição das aresta. Isto procura evitar um favorecimento de alguma implementação devido ao modo como o grafo foi gerado.

5.4 Tempos computacionais

Os tempos computacionais estão dispostos de acordo com o gerador de grafos utilizado e estão apresentados em segundos. Para cada tabela de tempos computacionais adicionamos uma tabela mostrando a variação dos tempos que compõem a média.

5.4.1 Randômico

O gerador de grafos randômicos foi escrito por C. McGeoch e foi usado num *workshop* de implementação de algoritmos do centro DIMACS [Dim93]. Com ele estipula-se o número de vértices e arestas do grafo e ele é montado randomicamente. Os grafos gerados foram divididos em dois grupos: pequenos (P) e grandes (G). Cada grupo foi dividido em três níveis de densidade (1, 2 e 3). Os grafos de densidade 2 têm 1/3 mais arestas do que os de densidade 1 e os de densidade 3 tem 1/4 mais arestas do que os de densidade 2, sendo que os grafos de densidade 1 são os mais esparsos. Os tempos obtidos estão na tabela 5.1. A coluna n e m apresentam respectivamente o número de vértices e de arestas dos grafos resolvidos. As coluna G e MV apresentam respectivamente os tempos obtidos com as implementações do algoritmo de Gabow e de Micali e Vazirani.

Observamos na tabela 5.1 que a implementação paralela conseguiu uma boa aceleração nos grafos P_1, inclusive obtendo um tempo menor do que as duas

R	n	m	MV	G	CPUs da Implementação Paralela				
					1	2	4	6	7
P_1	20.000	30.000	9,11	10,97	12.08	7,06	5,09	4,43	4,35
P_2	20.000	40.000	4,45	48,46	9.68	7,42	6,25	5,62	6,53
P_3	20.000	50.000	3,6	41,46	8,22	6,72	5,91	5,69	5,95
G_1	40.000	60.000	27,19	77,67	49,94	27,46	23,3	15,44	14,57
G_2	40.000	80.000	9,73	192,31	40	23,43	19,77	16,51	18,64
G_3	40.000	100.000	10,85	198	30,11	22,07	17,22	14,77	17,14

Tabela 5.1: Tempos em segundos dos grafos randômicos.

implementações seqüenciais. Observamos também que nos grafos P_2 o melhor tempo é alcançado com 6 *threads* e este tempo tende a aumentar com o aumento do número de *threads*. Isto deve-se ao *overhead* de criação, manipulação e de concorrência em áreas de exclusão mútua entre os *threads*.

Uma boa aceleração também foi conseguida pela implementação paralela nos grafos G_1. Além do mais, no grafo G_1 a aceleração obtida indica que o tempo poderia melhorar com a utilização de mais alguns *threads* e processadores.

Na tabela 5.2 observamos a variação dos tempos em porcentagem obtidos com o grafos randômicos pequenos. Esta variação (V) é calculada através da proporção entre a média e o desvio padrão dos tempos que a compõem, como na fórmula abaixo:

$$V = \frac{desvio \times 100}{média}$$

R	MV	G	CPUs da Implementação Paralela				
			1	2	4	6	7
P_1	10,47	14,1	1,43	6,15	13,27	12,69	9,05
P_2	17,79	72,34	5,77	7,62	13,82	23,35	19.29
P_3	38,3	53,5	4,89	8,44	18,88	22,06	17,53
G_1	10,76	16,49	4,26	2,01	29,02	16,5	11,38
G_2	14,62	30,63	3,13	8,13	12,83	16,5	22,09
G_3	20,1	40,15	3,5	9,38	10,48	13,3	20,41

Tabela 5.2: Variação dos tempos em porcentagem dos grafos randômicos.

Na tabela 5.2 notamos uma grande variação nos tempos obtidos pela implementação do algoritmo de Gabow nas instâncias P_2 e P_3. Entretanto estas variações, neste caso, não nos impede de afirmar que nestas instâncias a implementação do algoritmo de Gabow obteve performance pior do que a do algoritmo de Micali e Vazirani.

No algoritmo paralelo proposto, pode ocorrer o seguinte pior caso: os caminhos aumentantes são encontrados somente um após o outro, provocando por-

tanto uma execução puramente seqüencial. Procuramos investigar este tipo de problema medindo o número de vezes que os processadores detectaram inconsistências no processo de procura de caminhos aumentantes.

As inconsistências são apresentadas em forma de proporção entre o número de inconsistências detectadas durante a resolução dos grafos e o número de caminhos aumentantes encontrados pela implementação paralela, como a fórmula a seguir:

$$Inconsistências\% = \frac{média\_de\_inconsistências \times 100}{média\_de\_cam\_aument.}$$

Uma inconsistência é detectada quando encontra-se uma situação incoerente durante a expansão da busca ou quando o caminho aumentante identificado tem suas arestas modificadas e, portanto não é mais um caminho aumentante. Quando se utiliza somente 1 processador, não ocorrem inconsistências, portanto apresentamos valores partindo-se de 2 processadores. As inconsistências encontradas com o gerador Randômico estão apresentadas em porcentagem na tabela 5.3.

Randômico	CPUs			
	2	4	6	7
P_1	0,6	2,2	3,8	4,5
P_2	0,6	1,8	2,8	3,4
P_3	0,2	1,6	2,3	3
G_1	0,4	1,7	2,6	3,1
G_2	0,4	1,1	1,9	2,3
G_3	0,3	0,9	1,4	1,7

Tabela 5.3: Porcentagem de inconsistências nos grafos randômicos.

A tabela 5.3 mostra que relativamente poucas inconsistências foram encontradas para este tipo de grafo e que o número médio de inconsistências diminui quanto mais denso é o grafo, e que aumenta com o número de *threads* utilizados. Ambos esses resultados parecem concordar com nossa intuição.

5.4.2 Grade

Este gerador, por nós desenvolvido, constrói grafos em formato de grade bidimensional (B) ou tridimensional (T). No caso bidimensional informa-se o número de linha e de colunas e a probabilidade de existência das arestas. No caso tridimensional informa-se as três dimensões da grade e a probabilidade de existência das arestas. Os grafos bidimensionais possuem 200×200 vértices e estão divididos em três densidades geradas com uma probabilidade nas arestas de 50, 65 e 80%. Os grafos tridimensionais possuem $35 \times 35 \times 35$ vértices e estão divididos em três densidades geradas com uma probabilidade nas arestas de 50, 65 e 80%. Os grafos tridimensionais são mais densos do que os bidimensionais. Os tempos obtidos

nos grafos em grade estão na tabela 5.4 e a variação destes tempos está na tabela 5.5.

G	n	m	MV	G	CPUs da Implementação Paralela				
					1	2	4	6	7
B_1	43.000	41.000	8,1	16,47	55,19	28,36	14,89	10,88	9.62
B_2	43.000	52.000	23,04	10,74	37,21	19,28	10,27	7,65	6,63
B_3	43.000	65.000	54,38	16,71	37,7	23,99	16,29	13,9	12,93
T_1	43.000	62.000	44,31	14,09	39,1	24,34	14,62	12,43	12,3
T_2	43.000	78.000	19,51	14,66	33,32	21,04	13,99	11,81	11,45
T_3	43.000	100.000	16,9	10,27	20,32	12,76	8,88	6,8	6,76

Tabela 5.4: Tempos em segundos dos grafos em grade.

Nos grafos bidimensionais, a implementação paralela conseguiu tempos abaixo das implementações seqüenciais nos grafos B_2 e B_3. Entretanto, a aceleração apresentada nos grafos B_1 mostra que, se utilizássemos mais *threads* e mais processadores, talvez seria possível obtermos resultados melhores do que a implementação do algoritmo de Micali e Vazirani, o qual obteve a média de 8,1 segundos.

G	MV	G	CPUs da Implementação Paralela				
			1	2	4	6	7
B_1	15,91	1,5	1,76	1,11	1,34	1,78	0,92
B_2	12,18	1,13	2,49	1,28	1,18	1,32	1,77
B_3	9,69	21,31	13,03	21,67	29,1	28,16	6,58
T_1	5,06	14,9	9,33	15,7	14,6	16,98	15
T_2	20,21	26,5	15,6	19,61	20,63	14,96	15,84
T_3	20,12	7,62	7,58	6,59	15,22	10,18	12,95

Tabela 5.5: Variação em porcentagem dos tempos dos grafos em grade.

A medida das inconsistências detectadas nos grafos em grade estão na tabela 5.6.

5.4.3 Anel

O gerador Anel, por nós desenvolvido, constrói grafos que possuem subgrupos de vértices. Os vértice de cada subgrupo são ligados entre si randomicamente. Os subgrupos são ligados entre si formando um anel. Neste gerador, informa-se o número de subgrupos, o número de vértices de cada subgrupo, o número de arestas que ligarão dois subgrupos entre si, o número de arestas por vértice (estas arestas somente ligam vértices do mesmo grupo) e a probabilidade de existência destas arestas.

Grade	CPUs			
	2	4	6	7
B_1	0	0	0	0,1
B_2	0	0,1	0,1	0,1
B_3	0,2	0,7	1,2	1,6
T_1	0,3	0,9	1,8	2
T_2	0,3	0,7	1,4	1,5
T_3	0,3	0,8	1,2	1,4

Tabela 5.6: Porcentagem de inconsistências nos grafos em grade.

Os grafos gerados estão divididos em dois grupos: pequenos (P) e grandes (G). Cada grupo está dividido em três densidades. Os grafos pequenos possuem 15 grupos com 1.500 vértices cada. Estes grupos estão ligados em anel e a ligação entre dois grupos é feita por 150 arestas. Os grafos grandes possuem 20 grupos com 2.000 vértices cada. A ligação entre dois grupos é feita por 200 arestas. Os tempos obtidos nos grafos em anel estão na tabela 5.7.

A	n	m	MV	G	CPUs da Implementação Paralela				
					1	2	4	6	7
P_1	22.500	30.000	12,68	4,6	13,82	7,43	4,33	3,21	3,03
P_2	22.500	40.000	5,41	21,98	9,85	5,95	4,42	4,4	4,27
P_3	22.500	50.000	5,18	37,11	8,91	5,71	4,69	4,89	4,71
G_1	40.000	60.000	25,03	31,3	35,32	19,27	12,51	9,37	8,9
G_2	40.000	78.000	12,85	114,26	30,15	16,81	11,76	10,47	11,03
G_3	40.000	98.000	10,27	115,19	23,19	14,25	10,69	9,66	9,39

Tabela 5.7: Tempos dos grafos em anel.

Nos grafos em anel, a implementação paralela obteve tempos menores do que as médias das respectivas implementações seqüenciais em todas as instâncias. Observando o gráfico de variação do tempo de execução com o aumento do número de processadores, figura 5.1, notamos que a diminuição no tempo de execução de 6 para 7 processadores é muito pequena. Este fenômeno é característico de máquinas de memória compartilhada e é causado pela contenção pela memória entre os processadores (*threads*). Este particular gráfico mostra que com esta implementação não valeria a pena aumentar o número de processadores além de 10.

A variação dos tempos computacionais dos grafos em anel estão na tabela 5.8.

A medida das inconsistências detectadas nos grafos em anel estão na tabela 5.9.

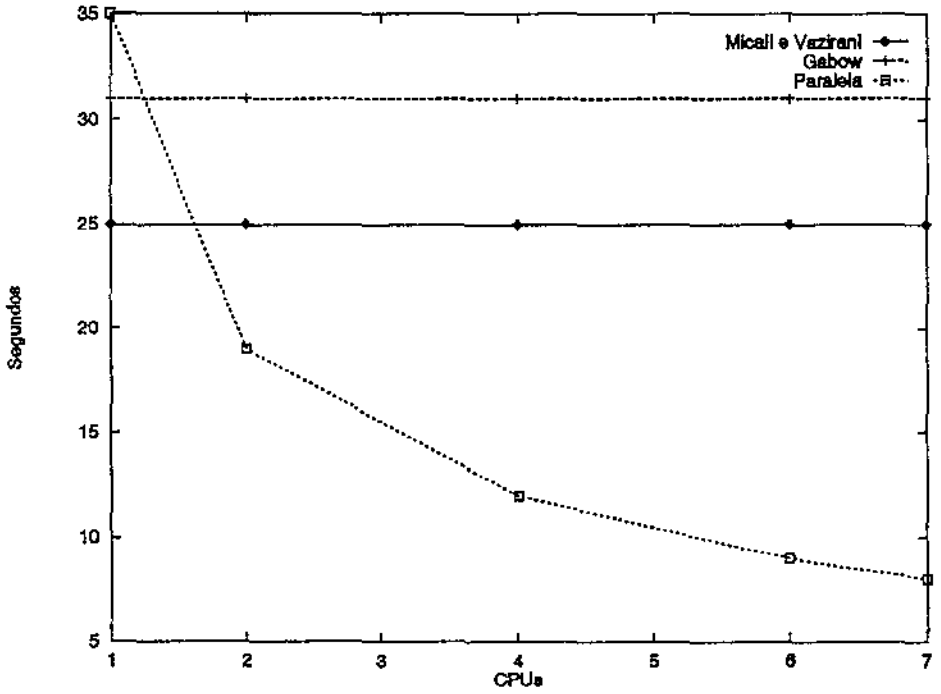

Figura 5.1: Aceleração obtida no grafo Anel G_1

A	MV	G	CPUs da Implementação Paralela				
			1	2	4	6	7
P_1	11,76	4,11	2,69	2,12	2,83	3,17	3,66
P_2	19	40,58	4,95	5,5	9,8	11,53	10,58
P_3	12,35	15,23	6,03	5,31	13,48	10,42	12,01
G_1	9,62	24,89	3,15	2,44	12,07	5,8	7,17
G_2	20,27	30,14	4,16	2,88	5,84	5,86	7,95
G_3	19,1	39,48	3,67	7,5	8,19	5,75	13,83

Tabela 5.8: Variação em porcentagem dos tempos dos grafos em anel.

Anel	CPUs			
	2	4	6	7
P_1	0	0,6	0,7	1,1
P_2	0,5	1,6	2,7	3,4
P_3	0,3	0,9	1.3	1,7
G_1	0,2	0,7	1.4	1,7
G_2	0,1	0,6	1	1,2
G_3	0,1	0,5	0,8	1

Tabela 5.9: Porcentagem de inconsistências nos grafos em anel.

5.4.4 Bipartido

O gerador Bipartido foi escrito por Setubal [Set93]. Ele cria grafos que possuem subgrupos de vértices. Estes subgrupos formam um anel e as arestas somente ligam vértices de um subgrupo ao outro. Estes grafos fazem com que as implementações não necessitem formar flores nem florescências. Neste gerador informa-se o número de grupos desejados, o número de vértices de cada grupo e o número esperado de arestas por vértice. Os grafos bipartidos foram divididos em três tamanhos. Todos os três possuem 256 vértices por grupo e aproximadamente 4 arestas por vértice. Os grafos pequenos possuem 16 grupos, os médios possuem 32 grupos e os grandes 64 grupos. Todos os grafos obtidos possuem aproximadamente a mesma densidade. A implementação paralela foi comparada com o algoritmo de Micali e Vazirani. Para este tipo de grafo não foram obtidos tempos com a implementação de Gabow. Os tempos obtidos estão na tabela 5.10.

B	n	m	MV	CPUs da Implementação Paralela				
				1	2	4	6	7
Pequeno	8.000	16.000	1,45	1,94	1,73	1,41	1,19	1,25
Médio	16.000	32.000	6,91	7,58	5,48	4,67	3,84	3,93
Grande	32.000	65.000	22,91	22,11	16,5	17,57	9,63	8,88

Tabela 5.10: Tempos dos grafos bipartidos.

Os tempos obtidos para os grafos grande bipartidos foram bem inferiores ao algoritmo de Micali e Vazirani e a aceleração obtida mostra ainda que tempos menores poderiam ser obtidos com mais *threads* e mais processadores.

A variação dos tempos obtidos com o gerador Bipartido estão na tabela 5.11.

B	MV	CPUs da Implementação Paralela				
		1	2	4	6	7
Pequeno	1,16	5,32	4,1	6,56	14,57	17,03
Médio	0,51	5,85	0,46	4,13	13,51	7,07
Grande	0,57	3,81	0,42	62,11	11,55	29,18

Tabela 5.11: Variação em porcentagem dos tempos dos grafos bipartidos.

A medida das inconsistências detectadas nos grafos bipartidos estão na tabela 5.12. Um fato curioso nos grafos bipartidos é que o número de inconsistências cresce nos grafos mais densos, fugindo à tendência apresentada pelos demais grafos.

Observamos que neste tipo de grafo o número de inconsistências detectadas foi bastante alto. Uma explicação para esse fato é que o tamanho dos grafos é bastante pequeno se comparado aos demais. Isto facilita a procura de caminhos

Bipartido	CPUs			
	2	4	6	7
Pequeno	3,7	16	20	20
Médio	0,1	9,9	20,5	20
Grande	0,7	4,3	9,8	9,3

Tabela 5.12: Porcentagem de inconsistências nos grafos bipartidos.

aumentantes, que são encontrados com maior freqüência pelos processadores num período de tempo menor, aumentando a probabilidade de ocorrer inconsistências. Um outro fator, que pode influenciar, é que quanto menor for o grafo, maior a chance dos processadores encontrarem caminhos aumentantes com interseção entre eles, ocasionando um maior número de inconsistências.

5.5 Análise dos resultados

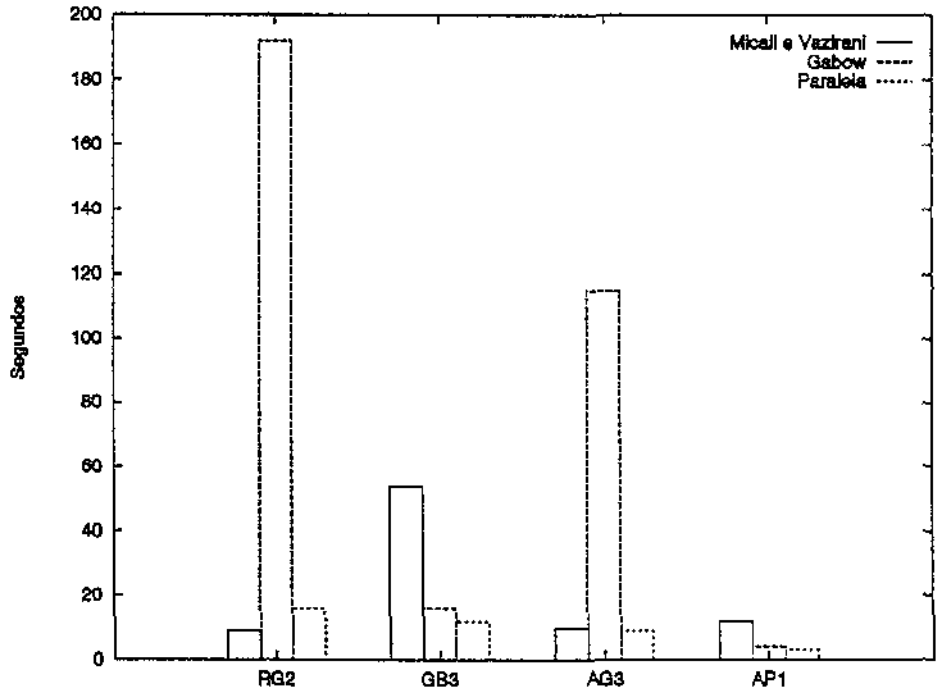

Figura 5.2: Robustez da implementação paralela

Na figura 5.2 observamos que a implementação do algoritmo de Gabow obteve um desempenho excessivamente lento em relação as outras implementações no grafo randômico grande de densidade 2 e no grafo em anel grande de densidade 3. Já a implementação do algoritmo de Micali e Vazirani obteve tempos maiores nos grafos em grade bidimensional de densidade 3 e nos grafos em anel pequeno

de densidade 1. No entanto, a implementação paralela apresentou um comportamento uniforme nestas instâncias e este comportamento se manteve durante todos os testes realizados. Isto demonstra uma robustez da implementação paralela, uma vantagem significativa em relação às outras duas implementações.

Um resumo dos melhores tempos computacionais pode ser encontrado na tabela 5.13. Nela apresentamos o melhor tempo obtido por uma implementação seqüencial e o melhor tempo obtido pela implementação paralela usando no máximo 7 processadores. Uma relação entre o tempo seqüencial (ts) e o paralelo (tp) também é mostrada.

Grafos	Melhor seqüencial	Melhor paralelo	ts/tp
Randômico P_1	9,11	4,35	**2,09**
Randômico P_2	4,45	5,62	0,79
Randômico P_3	3,6	5,69	0,63
Randômico G_1	27,19	14,57	**1,86**
Randômico G_2	9,73	16,51	0,58
Randômico G_3	10,85	14,77	0,73
Grade B_1	8,1	9,62	0,84
Grade B_2	10,74	6,63	**1,61**
Grade B_3	16,71	12,93	**1,29**
Grade T_1	14,09	12,3	**1,14**
Grade T_2	14,66	11,45	**1,28**
Grade T_3	10,27	6,76	**1,51**
Anel P_1	4,6	3,03	**1,51**
Anel P_2	5,41	4,27	**1,26**
Anel P_3	5,18	4,69	1,1
Anel G_1	25,03	8,9	**2,81**
Anel G_2	12,85	10,47	**1,22**
Anel G_3	10,27	9,39	**1,09**
Bipartido Pequeno	1,45	1,19	**1,21**
Bipartido Médio	6,91	3,84	**1,79**
Bipartido Grande	22,91	8,88	**2,57**

Tabela 5.13: Quadro sintético dos melhores tempos.

As principais observações feitas com base nos testes são:

- A implementação paralela obteve acelerações em 70% dos casos.
- Em 15% dos casos a aceleração foi superior a 2.
- Os tempos foram melhores em grafos esparsos.
- A implementação é robusta.
- O número de inconsistências em geral foi baixo.

- O único caso em que o número de inconsistências foi alto foi nos grafos bipartidos, ainda assim a implementação paralela foi melhor.

Estas observações demonstram a viabilidade da idéia básica do algoritmo.

Capítulo 6

Conclusões

Neste trabalho apresentamos um estudo dos principais algoritmos para o problema do Emparelhamento em Grafos e, com base neste estudo, desenvolvemos uma implementação paralela eficiente na prática. A solução paralela proposta por Mulmuley, Vazirani e Vazirani (MVV) apresenta ineficiências que dificultam uma implementação eficiente baseada neste algoritmo. Estas ineficiências fizeram com que as implementações paralelas fossem baseadas nos algoritmos seqüenciais. A primeira implementação paralela foi baseada no algoritmo de Micali e Vazirani e foi criada a partir de uma implementação seqüencial por nós desenvolvida. Esta implementação não é eficiente devido a uma série de dificuldades. Dentre elas destacamos:

- Excesso de sincronismo necessário.
- Aumento do número de tarefas de expansão de busca.
- Complexidade de paralelização das buscas em profundidade, devido aos inúmeros casos de *Deadlock*.

Devido a estas dificuldades concluímos que uma implementação paralela baseada no algoritmo de Micali e Vazirani não seria um caminho promissor. Portanto, desenvolvemos uma segunda implementação baseada em novas idéias. Esta implementação é um misto entre os algoritmos de Edmonds e de Micali e Vazirani. Esta implementação apresentou um conjunto de características que possibilitaram uma boa performance na prática. Entre elas destacamos:

- É uma implementação robusta. Não apresentou variações excessivas do tempo de execução nos vários tipos e tamanhos de grafos testados.
- Não necessita de pontos de sincronismo entre os processadores.

- Possui somente duas áreas de exclusão mútua: uma para adquirir um vértice livre e outra para aumentar caminhos aumentantes. Esta áreas de exclusão não produzem interferências significativas no tempo de execução.

Estas características possibilitaram que a implementação paralela obtivesse acelerações na maioria dos casos. Com isto concluímos que uma implementação paralela eficiente na prática foi obtida com sucesso.

6.1 Contribuições deste trabalho

As principais contribuições desse trabalho são:

- A identificação de um caso que requer um tratamento especial pelo algoritmo de Micali e Vazirani.
- Uma implementação paralela do algoritmo de Micali e Vazirani e uma relação das dificuldades de paralelização deste algoritmo.
- Uma implementação paralela do algoritmo de Edmonds, com idéias de Micali e Vazirani que consegue acelerações na prática sobre 2 boas implementações seqüenciais e que é robusta.

6.2 Possíveis linhas de continuidade

Algumas das possíveis extensões deste trabalho são:

- Implementação paralela de uma rotina de emparelhamento inicial mais efetiva.
- Aperfeiçoar operações mais custosas da implementação paralela.
- Testar a implementação em outros tipos de grafos.
- Obter uma outra implementação paralela usando novas idéias.

Apêndice A

Programação Paralela no SOLARIS

Durante o desenvolvimento das implementações paralelas criamos o texto abaixo, explicando as funções básicas para a criação, manipulação e sincronização de *threads* (uma maneira de se criar implementações paralelas) no sistema operacional SOLARIS. A linguagem de programação utilizada é a linguagem C e o compilador utilizado é o `gcc`. Primeiramente, é apresentada uma explicação de como criar *threads*; em seguida é apresentado como criar áreas de exclusão mútua e como sincronizar os *threads*; e ,finalmente, são apresentadas as diretivas necessárias para compilar um programa paralelo e ressaltadas as opções usadas nas implementações descritas por este trabalho.

A.1 Introdução

Existem duas maneiras de se criar programas paralelos no SOLARIS. A primeira é através da criação de processos como comando `fork` e da atribuição de uma memória comum a eles através da função `mmap` com o flag `MAP_SHARED`. Os processadores executarão os processos concorrentemente. A segunda maneira é através da criação de *threads* com o comando `thr_create`. Com isto, novos pontos de execução são criados sobre o mesmo código, a memória é automaticamente compartilhada e o sistema operacional executará os *threads* concorrentemente de acordo com o nível de concorrência estipulado pelo programador. Neste trabalho utilizamos *threads* para criar as implementações paralelas. Em seguida descreveremos como criar *threads*.

A.2 Criação de *threads*

A função `thr_create` necessita de seis parâmetros:
`thr_create (NULL, 0, funcao_ptr, parametro, tipo_thr, &thread_id)`.

O objetivo de cada um é:

- NULL e 0: estes dois primeiros parâmetros não são usados pela maioria dos programas, por serem de difícil mensuração (referentes à pilha do *thread*).
- funcao_ptr: é um ponteiro para uma função do usuário do tipo void *funcao (void *parametro). O *thread* iniciará sua execução a partir desta função.
- parametro: é uma variável do tipo void *. Este ponteiro é passado como parâmetro para o *thread*.
- tipo_thr: tipo do *thread*. Os mais comuns são:
  - THR_SUSPENDED: cria um *thread* que fica suspenso até que seja chamada a função thr_continue ().
  - THR_BOUND: cria um *thread* ligado a um LWP. Esta é a opção usada neste trabalho.
  - THR_NEW_LWP: cria um processo "leve" (*Light Weight Process*).
- &thread_id: é uma variável do tipo thread_t. A função thr_create preenche esta variável com o identificador do *thread* recém criado.

A função thr_create retorna um valor diferente de 0 se ocorreu algum erro. Cada *thread* possui sua própria pilha. As variáveis que forem declaradas dentro da função destinada para ser um *thread* são alocadas na pilha, portanto, são locais ao *thread*. As variáveis globais à função são também globais ao *thread*.

Após a criação dos *thread* é possível estipular o nível de concorrência desejado, ou seja, o número máximo de *threads* que podem ser executados ao mesmo tempo. Isto é feito pela função thr_setconcurrency com parâmetro o número de *threads* que podem estar paralelos. Cada vez que se cria um *thread*, o nível de concorrência é aumentado em 1 automaticamente. Portanto, thr_create cria *threads* paralelos e a função thr_setconcurrency é usada para diminuir o nível de concorrência.

O término de um processo implica no término de seus *threads*. Para garantir que o *thread* termine sua tarefa utiliza-se a função thr_join. Esta função interrompe a execução do processo até que o *thread* especificado termine. O primeiro parâmetro é o identificador do *thread* que a função deve esperar. Os dois últimos parâmetros são parâmetros de retorno, raramente são utilizados (NULL). Caso não seja especificado o *thread*, a função thr_join espera pelo término de qualquer *thread*. Portanto, se forem criados três *threads*. basta chamar três vezes thr_join para que o processo continue somente quando os três *threads* terminarem.

A.3 Esboço de um programa paralelo

```
#include <thread.h>
#include <synch.h>
#include <errno.h>

int variavel_global_aos_threads;

void *meu_thread (void *parametro) {

  int *meu_inteiro = (int *) parametro;
  int variavel_local_ao_thread;
          . . .
}

main () {

  thread_t thread_id;
  int  minha_variavel, i;
  void *parametro = &minha_variavel;

  for (i = 0; i < NUM_THREADS; i++)
    thr_create (NULL, 0, meu_thread, parametro,
    THR_BOUND, &thread_id);

  /* Espera pelo termino dos threads */
  for (i = 0; i < NUM_THREADS; i++)
    thr_join (NULL, NULL, NULL);

}
```

A.4 Exclusão mútua

Para garantir que somente um processo execute uma parte do código, geralmente utiliza-se variávies *mutex*, pois são flexíveis e consomem poucos recurso computacionais se comparadas com outras funções disponíveis. Todas as variáveis `mutex_t` devem ser inicializadas pela função `mutex_init`. Apenas os dois primeiros parâmetros são importantes para a função `mutex_init (&variavel_mutex, tipo, NULL)`:

- `&variavel_mutex`: é o endereço da variável *mutex*, a qual é do tipo `mutex_t`.

- `tipo`: variável que pode ser de dois tipos:
  - `USYNC_THREAD`, a variável *mutex* somente pode sincronizar *threads* do processo que a inicializou.
  - `USYNC_PROCESS`, a variável *mutex* pode sincronizar *threads* deste processo ou de outros processos. Somente um processo deve inicializá-la. Como neste trabalho utulizamos *threads*, esta é a opção usada nas implementações paralelas.

O *mutex* é adquirido pela função `mutex_lock (&variavel_mutex)` e é liberado pela função `mutex_unlock (&variavel_mutex)`. A função `mutex_trylock` tenta adquirir a variável *mutex*, se ela está ocupada a função retorna um erro e prossegue com a execução. Já a função `mutex_lock` é bloqueada até que a variável *mutex* seja liberada. A função `mutex_destroy` libera a área utilizada pela variável *mutex*.

A.5 Esboço da utilização de *mutex*

```
#include <synch.h>

mutex_t bloqueia_threads;

void *meu_thread (void *parametro) {
  mutex_lock (&bloqueia_threads);
  /* Area de exclusao mutua */
  mutex_unlock (&bloqueia_threads);
}

main () {
  mutex_init (&bloqueia_threads, USYNC_THREAD, NULL);
          /* Criam-se os threads... */
  mutex_destroy (&bloqueia_threads);
}
```

A.6 Sincronização de *threads*

Freqüentemente é necessário sincronizar os *threads* num determinado ponto. O SOLARIS não fornece explicitamente uma função que execute esta tarefa. Entretanto, ela pode ser implementada utilizando variáveis condicionais. As funções abaixo fornecidas implementam sincronização de *threads*. Deve-se inicializar a sincronização pela função `sync_init`, informando o número de *threads* que serão

sincronizados. A sincronização é feita pela função sync () chamada dentro do código do *thread*. Antes de terminar o processo chama-se sync_destroy ().

```
#include <thread.h>
#include <synch.h>
#include <errno.h>

/* Mutex exigido pela variavel condicional */
static mutex_t mutex_block;

/* Variavel condicional */
static cond_t cond_block;

/* Numero de threads bloqueados */
static int count_block;

/* Numero de threads que irao chamar sync () */
static int num_threads;

/* sync_init - Inicializa a sincronizacao
   entrada - o numero de threads
             para serem sincronizados. */

void sync_init (int num_thr) {
  count_block = num_threads = num_thr;
  mutex_init (&mutex_block, USYNC_THREAD, NULL);
  cond_init (&cond_block, USYNC_THREAD, NULL);
}

/* sync - sincronizacao */

void sync (void) {
  mutex_lock (&mutex_block);
  --count_block;
  if (count_block)
    cond_wait (&cond_block, &mutex_block);
  else {
    count_block = num_threads;
    cond_broadcast (&cond_block);
  }
  mutex_unlock (&mutex_block);
```

```
}

/* sync_destroy - destroi as variaveis usadas
                  na sincronizacao */

void sync_destroy (void) {
  cond_destroy (&cond_block);
  mutex_destroy (&mutex_block);
}
```

A.7 Compilando um programa paralelo

Para se compilar um programa paralelo deve-se acrescentar as diretivas de compilação `-lthread -REENTRANT` ao compilador cc. Estas diretivas fazem com que sejam usadas funções da biblioteca que suportam *threads*, ou seja, funções que podem ser chamadas simultaneamente por vários *threads* em que haja interferência alguma entre eles. Cada *manpage* sobre funções da linguagem C possui informação sobre o `MT_LEVEL` das funções. Se elas forem do tipo `MT_Safe` (*Multithreaded-Safe*) então elas podem ser chamadas por vários *threads* ao mesmo tempo, sem que seja necessário um tratamento especial via variável *mutex*.

Bibliografia

[AMO93] R. Ahuja, T. Magnanti e J. Orlin. *Network Flows.* Prentice-Hall: 475–494, 1993.

[BS95] C. F. Bella Cruz e J. C. Setubal. Uma adaptação do algoritmo de emparelhamento de Edmonds para execução em paralelo. II Oficina Nacional em Problemas Combinatórios, Relatório Técnico DCC-95-17: 39-48, 1995.

[Ber57] C. Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci.*, 43: 842–844, 1957.

[Blu94] N. Blum. A new approach to maximum matching in general graphs. Technical Report 94-06-23, Universität Bonn, Bonn, Germany, 1994.

[Dim93] David S. Johnson and Catherine C. McGeoch (editors). *First* DIMACS *Implementation Challenge — Network Flows and Matching.* American Mathematical Society, DIMACS Series in Mathematics and Theoretical Computer Science, volume 12, 1993.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17: 449–467, 1965.

[EK75] S. Even and O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. *Proc. 16th Annual IEEE Symposium on Foundations of Computer Science*, 100–112, 1975.

[Gab76] H. N. Gabow. An efficient implementation of Edmonds' maximum matching algorithm. *J. Assoc. Comput. Mach.*, 23: 221–234, 1976.

[GT85] H. N. Gabow e R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.* 30, 209–221, 1985.

[HK73] J. E. Hopcroft e R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2: 225–231, 1973.

[Jaj92] Joseph JáJá. *An introduction to parallel algorithms.* Addison-Wesley: 1992.

[Lei90] F. T. Leighton. *Introduction to parallel algorithms and architectures.* Morgan Kauffman, 1990.

[MR93] R. B. Mattingly and N. P. Ritchey. Implementing an $O(\sqrt{|V|}\,|E|)$ Cardinality Matching Algorithm. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching - 1st* DIMACS *Implementation Challenge*, 539–556. American Mathematical Society, 1993.

[MV80] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the IEEE 21st Annual Symposium on Foundations of Computer Science*, 17–27, 1980.

[MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7: 105–113, 1987.

[NR59] R. Z. Norman and M. O. Rabin. An algorithm for a minimum cover of a graph *Proc. Amer. Math. Soc.*, 10: 315–319, 1959.

[PL88] P. A. Peterson and Michael C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3: 511–533, 1988.

[Set93] J. C. Setubal. New experimental results for bipartite matching. In *Proceedings of* NETFLOW93, *tech. report TR-21/93, dipartamento de Informatica, Universitá di Pisa*, 211–216, 1993.

[Vaz94] V. V. Vazirani. A theory of alternating paths and blossoms for proving correcteness of $O(\sqrt{V}\,E)$ general graph maximum matching algorithm. *Combinatorica*, 14: 71–109, 1994.