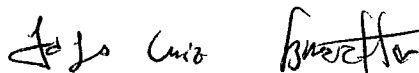


Algoritmos Paralelos para Problemas em Grafos

Edson Norberto Cáceres

Tese submetida ao corpo docente da Coordenação dos Programas de Pós-Graduação em Engenharia da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências em Engenharia de Sistemas e Computação.

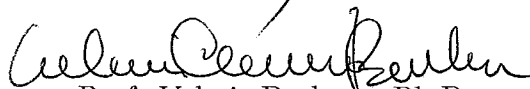
Aprovada por:



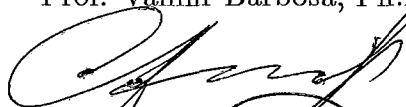
Prof. Jayme Luiz Szwarcfiter, Ph.D.
(Presidente)



Prof. Nelson Maculan Filho, D.Sc.



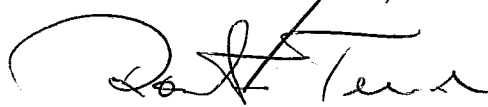
Prof. Valmir Barbosa, Ph.D.



Prof. Cláudio Amorim, Ph. D.



Prof. Celso Carneiro Ribeiro, Ph.D.



Prof. Routo Terada, Ph.D.

Rio de Janeiro, RJ - Brazil
Julho de 1992

Cáceres, Edson Norberto
Algoritmos Paralelos para Problemas em Grafos
Rio de Janeiro 1992
xii, 126p., 29,7 cm
(COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1992)
Tese - Universidade Federal do Rio de Janeiro, COPPE
1. Algoritmos paralelos. 2. Complexidade de algoritmos.
3. Teoria dos grafos.
I. COPPE/UFRJ II. Título (série).

Aos meus pais.

Agradecimentos

O trabalho de pesquisa para esta tese foi realizado na Universidade Federal do Rio de Janeiro e na University of Central Florida. Agradeço o carinho com que fui acolhido em ambas as instituições.

Agradeço ao Prof. Jayme pela competente e segura orientação.

Sou grato ao Prof. Narsingh Deo pelo apoio e orientação na U.C.F.

Aos Profs. Valmir e Celso pelo apoio nos Algoritmos Paralelos.

Aos colegas da COPPE, Célia, Celina, Carmen, Eliana, Márcia, Oscar e Sula.

Ao Nalvo, Arthur e Regina pelo grande suporte.

Aos colegas do DMT-UFMS que, de alguma forma, ajudaram na realização deste trabalho.

A Nahri pelo carinho e paciência.

Ao bom Deus.

Resumo da tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

Algoritmos Paralelos para Problemas em Grafos
Edson Norberto Cáceres
Julho de 1992

Orientador: Prof. Jayme Luiz Szwarcfiter

Programa: Engenharia de Sistemas e Computação

Apresentamos algoritmos paralelos para os problemas de computação de um *circuito de Euler*, árvores geradoras e componentes conexos de um grafo, busca em profundidade irrestrita, cliques maximais de um grafo *círculo* e determinação dos ciclos elementares de um digrafo utilizando uma CREW PRAM.

O algoritmo para *circuito de Euler* evita a computação de uma árvore geradora como passo intermediário e usa o conceito de *esteio* para identificar os vértices onde a operação de *costura* nos circuitos será efetuada. O algoritmo é ótimo desde que os vértices do *grafo bipartido auxiliar* estejam convenientemente rotulados.

Os algoritmos para computação de uma árvore geradora, computação dos componentes conexos de um grafo e computação dos componentes fracamente conexos de um digrafo usam *esteio* e são ótimos se o grafo de entrada estiver convenientemente rotulado. Essa rotulação é facilmente obtida para os grafos *st-planares* e *série-paralelos*.

Os algoritmos para busca irrestrita e caminhos simples maximais utilizam um *kl-caminho*. Utilizando o algoritmo para busca irrestrita em digrafos acíclicos computamos as cliques maximais de um *grafo círculo*.

Utilizando um *kl-caminho* apresentamos dois algoritmos para computar os ciclos elementares de um digrafo.

Abstract of thesis presented to COPPE/UFRJ as partial fulfillment for requirements for the degree of Doctor of Sciences (D.Sc.).

Parallel Algorithms for Problems in Graphs
July 1992

Thesis Supervisor: Prof. Jayme Luiz Szwarcfiter.

Department: Computation and Systems Engineering.

We present efficient parallel algorithms for computing an *Euler tour* of a graph, a spanning tree and the connected components, the maximal paths of a graph, the enumeration of the maximal cliques of a circle graph, and the enumeration of the elementary cycles of a directed graph.

The algorithm for computing an *Euler tour* avoids obtaining a spanning tree as an intermediate step. We use a *strut* to identify the vertices which have more than one circuit passing through them and perform a *stitch* on the circuits. If the vertices of the *auxiliary bipartite graph* are properly labeled, the algorithm is optimal.

We use *strut* for finding a spanning tree, the connected components of a graph and the weak connected components of a directed graph. The algorithms are optimal if the graph is properly labeled. The *st-planar* and *serie-parallel* graphs are easily labeled.

We use *kl-path* for finding the maximal paths of a graph. This algorithm is used for enumerating the maximal cliques of a *circle graph*

The *kl-path* is used to present two algorithms for the enumeration of the elementary cycles of a directed graph.

Índice

1	Introdução	1
1.1	Notação e Terminologia	2
1.2	Modelo Computacional	4
1.3	Resumo da Tese	4
2	Preliminares	8
2.1	O Modelo de Paralelismo	8
2.1.1	O Modelo PRAM	9
2.2	Algoritmos Paralelos Eficientes	11
2.3	Detalhamento do Modelo PRAM	14
2.4	Simulação entre Modelos	18
2.5	Algumas Técnicas Gerais para Algoritmos Paralelos	22
2.5.1	A Técnica da Árvore Binária Balanceada	22
2.5.2	A Técnica da Duplicação Recursiva	23
2.5.3	A Técnica da Conquista por Divisão	24
2.5.4	A Técnica de Compressão	26
2.5.5	A Técnica da Computação de Prefixos	26

3	Circuitos de Euler	28
3.1	Introdução	28
3.2	Notação e Terminologia	29
3.3	Algoritmo para Determinação de um Circuito de Euler	31
3.3.1	Descrição Preliminar do Algoritmo	32
3.3.2	O Algoritmo e um Exemplo	32
3.4	Correção e Análise	37
3.5	Um Exemplo Não Ótimo	39
3.6	Uma Simplificação do Algoritmo	41
4	Árvores Geradoras e Componentes Conexos	42
4.1	Introdução	42
4.2	Notação e Terminologia	43
4.3	Algoritmo para Determinação de uma Árvore Geradora	44
4.3.1	Descrição Preliminar do Algoritmo	44
4.3.2	O Algoritmo e um Exemplo	45
4.4	Grafos Não Orientados	48
4.5	Componentes Conexos	50
4.6	Uma Simplificação do Algoritmo	50
4.7	Correção e Análise	50
5	Busca em Profundidade Irrestrita	53
5.1	Introdução	53
5.2	Notação e Terminologia	54
5.3	Algoritmo para Busca Irrestrita em Digrafos Acíclicos	55

5.3.1	Descrição Preliminar do Algoritmo	56
5.3.2	O Algoritmo e um Exemplo	56
5.4	Algoritmo para Busca Irrestrita	62
5.4.1	Descrição Preliminar do Algoritmo	63
5.4.2	O Algoritmo e um Exemplo	63
5.5	Correção e Análise	69
6	Cliques Maximais em Grafos Círculo	72
6.1	Introdução	72
6.2	Notação e Terminologia	73
6.3	Grafos Círculo	75
6.4	Algoritmo Seqüencial	75
6.4.1	Descrição Preliminar do Algoritmo	76
6.4.2	O Algoritmo	76
6.5	Algoritmo para Determinação das Cliques Maximais em um Grafo Círculo	77
6.5.1	Descrição Preliminar do Algoritmo	78
6.5.2	O Algoritmo e um Exemplo	78
6.6	Algoritmo para Determinação do Número de Cliques Maximais de um Grafo Círculo	81
6.6.1	Descrição Preliminar do Algoritmo	82
6.6.2	O Algoritmo e um Exemplo	82
6.7	Correção e Análise	84
7	Algoritmos Seqüenciais para Determinação de Ciclos em um Digrafo	85

7.1	Introdução	85
7.2	Notação e Terminologia	86
7.3	Ciclos Elementares em Digrafos	86
7.4	Algoritmo de Tiernan	88
7.4.1	Descrição preliminar do algoritmo	88
7.4.2	O Algoritmo	89
7.5	Algoritmo de Szwarcfiter e Lauer	92
7.5.1	Descrição Preliminar do Algoritmo	92
7.5.2	O Algoritmo	92
7.6	Correção e Análise	95
7.6.1	Algoritmo de Tiernan	95
7.6.2	Algoritmo de Szwarcfiter e Lauer	96
8	Algoritmos Paralelos para Determinação de Ciclos em um Digrafo	98
8.1	Introdução	98
8.2	Notação e Terminologia	99
8.3	Algoritmo Paralelo de Tiernan	99
8.3.1	Descrição Preliminar do Algoritmo	101
8.3.2	O Algoritmo e um Exemplo	101
8.4	Algoritmo Paralelo de Szwarcfiter e Lauer	108
8.4.1	Descrição Preliminar do Algoritmo	109
8.4.2	O Algoritmo	110
8.5	Correção e Análise	115
9	Conclusões	117

Lista de Figuras

2.1	Modelo PRAM	10
2.2	Algoritmo Paralelo para Soma	13
2.3	Máximo de n Números	23
2.4	Duplicação Rucursiva	25
3.1	Grafo $G = (V, E)$ e uma partição de Euler de G	30
3.2	(a) Grafo Bipartido Auxiliar H , (b) um Esteio S em H e (c) um não Esteio em H	30
3.3	Uma costura em v_i	31
3.4	Grafo Bipartido Auxiliar $H(V_1, C', E_1)$	35
3.5	G_1 e uma partição de Euler.	39
3.6	(a) Grafo Bipartido Auxiliar H_1 e (b) Esteio S em H_1 com uma aresta adicionada a cada vértice não zero diferença.	40
3.7	Grafo Bipartido Auxiliar H_1 na segunda iteração.	41
4.1	Grafo $H = (V_1, V_2, E)$	45
4.2	Grafo $G = (V, E)$ e $H = (V, V', E')$	49
5.1	Grafo $G = (V, E)$	55
5.2	Grafo $G = (V, E)$	64

6.1	Grafo $G = (V, E)$	73
6.2	Digrafo $G = (V, E)$ localmente transitivo	74
6.3	(a) Grafo \vec{G} e (b) Grafo \vec{G}_R	76
6.4	Grafo \vec{H}	81
7.1	Grafo $G = (V, E)$	86
8.1	Grafo $G = (V, E)$	100

Capítulo 1

Introdução

A Computação paralela tem providenciado à Ciência da Computação uma série de problemas instigantes envolvendo projeto e análise de algoritmos. Dentre esses problemas, destaca-se a expectativa de melhorar os limites de tempo para problemas onde a computação seqüencial é quase impraticável.

Na área de algoritmos paralelos (bem como algoritmos de um modo geral) a preocupação com a complexidade computacional dos processos desenvolvidos é inerente, pois necessitamos de um instrumento que nos permita verificar se um dado algoritmo é *melhor* que um outro quando ambos resolvem o mesmo problema, além do fato de saber se um determinado problema possui um algoritmo *razoável* que o solucione. Essa preocupação, por sua vez, implica no objetivo de procurar elaborar algoritmos que sejam tão eficientes quanto possível. Conseqüentemente, torna-se imperioso o estabelecimento de critérios que possam avaliar essa eficiência.

Uma das formas de se projetar algoritmos paralelos é o de associar uma arquitetura particular ao problema, fazendo com que sua eficiência dependa diretamente da arquitetura envolvida e tornando difícil sua comparação com outros algoritmos que resolvam o mesmo problema mas que foram projetados para arquiteturas distintas. Para que possamos analisar a eficiência de um algoritmo independentemente de uma *máquina* particular, necessitamos adotar um processo que seja analítico. A tarefa de definir um critério analítico torna-se mais simples se a eficiência a ser avaliada for relativa a um *modelo* de computador.

Um objetivo da teoria da complexidade é o de encontrar o *modelo* mais apropriado de um computador. Modelos diferentes podem levar a diferentes computações e o número de passos executados por um mesmo algoritmo pode variar de acordo com o *modelo* considerado [78].

A utilização de um modelo abstrato de computador é motivada pelo fato de que vários algoritmos a nível de máquina são refinamentos de algoritmos que independem de uma máquina em particular, os quais foram projetados para um modelo genérico de computação paralela, além disso, até o momento não está claro quais modelos de computação paralela são razoáveis.

Isso faz com que nossa ênfase seja dada no projeto de algoritmos paralelos que utilizem o menor tempo e não dependam de uma linguagem de programação de alto nível nem dos detalhes da arquitetura da máquina. Essa abordagem está concentrada sobre a essência da teoria de algoritmos e capta coerentemente o que existe no momento dentro das atividades de pesquisa. Em função disso omitiremos as considerações de arquiteturas específicas tais como computadores *cube-connected*, *perfect-shuffle*, etc. Uma abordagem nessa área pode ser vista em [72,68].

Portanto, selecionaremos a corrente da área de computação paralela que concentra seu interesse no projeto de algoritmos, aproveitando a natureza inerentemente paralela do problema em estudo.

O desenvolvimento de algoritmos paralelos eficientes para problemas em grafos está diretamente condicionado à existência de algoritmos paralelos eficientes para os problemas de componentes conexos, árvores geradoras, ciclos de Euler e busca em um grafo. Neste trabalho apresentamos algoritmos paralelos para grafos utilizando *esteio* e *kl-caminho*. Essas estruturas decompõem um grafo e são simples de serem efetuadas em paralelo.

1.1 Notação e Terminologia

Utilizaremos a notação padrão de teoria dos grafos [10,30,78]. Um *grafo não orientado* $G = (V, E)$ tem um *conjunto de vértices* V e um *conjunto de arestas* E de pares ordenados de vértices distintos de V . Seja $n = |V|$ e $m = |E|$ o número de vértices e arestas em G , respectivamente. Cada aresta $e \in E$ será denotada pelo par de vértices $e = (u, v)$ que a forma. Neste caso os vértices u e v são os *extremos* da aresta e , sendo denominados *adjacentes*. Vértices adjacentes são chamados *vizinhos*. A aresta e é denominada *incidente* a ambos u e v . Um grafo $H = (V', E')$ é um subgrafo de G se $V' \subseteq V$ e $E' \subseteq E$. O subgrafo de G induzido por V' , denotado por $G(V')$, é o subgrafo maximal com relação a arestas de G que contém somente os vértices de V' .

Um *caminho* C é um grafo com vértices v_0, \dots, v_k e arestas (v_i, v_{i+1}) para $0 \leq i \leq k$. C é um caminho de v_0 a v_k , e dizemos que o *comprimento* deste caminho é k . Se todos os vértices do caminho C forem distintos, C é denominado um *caminho elementar* ou *caminho simples*. Um *ciclo* é um caminho onde $v_0 = v_k$

e $k \geq 3$. Se o caminho v_0, \dots, v_{k-1} for um caminho elementar, o ciclo v_0, \dots, v_k é denominado um *ciclo elementar*. Um grafo que não possui ciclos elementares é denominado *acíclico*. Dois ciclos são considerados *idênticos* se um deles puder ser obtido do outro, através de uma rotação de seus vértices. Um grafo G é *conexo* quando existe um caminho entre cada par de vértices de G . Seja S um conjunto e $S' \subseteq S$. Dizemos que S' é *maximal* em relação a uma certa propriedade P , quando S' satisfaz a propriedade P e não existe subconjunto $S'' \supset S'$, que também satisfaz P . Os *componentes conexos* de G são os subgrafos conexos maximais de G . Uma *árvore* $T = (V, E)$ é um grafo acíclico e conexo. Um conjunto de árvores é denominado *floresta*. Um *subgrafo gerador* do grafo $G = (V, E)$ é o subgrafo $H = (V', E')$ tal que $V = V'$. Se o subgrafo gerador H é uma árvore, o subgrafo H é denominado uma *árvore geradora*.

Um grafo $G = (V, E)$ é denominado *bipartido* se V pode ser particionado em dois conjuntos disjuntos $V_1 = \{v_1, v_2, \dots, v_{|V_1|}\}$ e $V_2 = \{u_1, u_2, \dots, u_{|V_2|}\}$ tal que toda aresta é incidente a exatamente um vértice em cada conjunto. Um grafo é *completo* quando existe uma aresta entre cada par de seus vértices. Uma *clique* é um subgrafo completo de G e uma *clique maximal* é aquela que não esteja propriamente contida em nenhuma outra.

Um *grafo orientado* (*digrafo*) $D = (V, E)$ é um conjunto finito não vazio V (os *vértices*), e um conjunto E (as *arestas*) de pares ordenados distintos. Portanto, num digrafo cada aresta (u, v) possui uma única direção de u para v . Definimos caminho, caminho elementar, ciclo e ciclo elementar em um digrafo de forma análoga a um grafo.

Seja $D = (V, E)$ um digrafo e um vértice $v \in V$. O *grau de entrada* de v é o número de arestas que chegam a v . O *grau de saída* de v é o número de arestas que saem de v . Uma *fonte* é um vértice com grau de entrada nulo, enquanto um *sumidouro* é um vértice com grau de saída nulo.

Um digrafo $D = (V, E)$ é *fortemente conexo* quando para todo par de vértices $u, v \in V$ existir um caminho em D de u para v e também de v para u . O digrafo D é chamado *fracamente conexo* ou *desconexo*, conforme seu grafo subjacente seja conexo ou desconexo, respectivamente. Se existir algum caminho em D de um vértice u para um vértice v , então diz-se que u *alcança* v , sendo este *alcançável* de u . Se v alcança todos os vértices de D então v é chamado *raiz* do digrafo.

Um digrafo D é acíclico quando não possui ciclos. Seja $D = (V, E)$ um digrafo acíclico. O *fechamento transitivo* de D é maior digrafo $D_f = (V, E_f)$ que preserva a alcançabilidade de D . Isto é, para todo $u, v \in V$, se u alcança v em D então $(u, v) \in E_f$. Analogamente, a *redução transitiva* de D é o menor digrafo $D_r = (V, E_r)$ que preserva a alcançabilidade de D . Isto é, se $(u, v) \in E_r$ então u não

alcança v em $D - (u, v)$.

1.2 Modelo Computacional

O modelo de computação paralela que utilizaremos é a máquina de acesso aleatório paralelo (*PRAM*) proposta por Fortune e Wyllie [33]. Neste modelo de memória compartilhada, em uma unidade de tempo cada processador pode ler uma unidade de informação, escrevê-la, ou executar uma operação aritmética elementar ou uma operação lógica sobre operandos localizados em endereços de memória quaisquer.

O modelo *PRAM* possui diversas variações as quais diferem na forma de resolver os conflitos de escrita e leitura simultâneas a uma mesma posição de memória por processadores diferentes.

Vamos apresentar o modelo *PRAM* com suas variações, e relacioná-lo com os principais modelos utilizados na análise da complexidade de algoritmos paralelos. Para definições formais sugerimos [33,89]. A literatura que trata desta abordagem está crescendo rapidamente de forma qualitativa [18,40,50,31].

Vamos também apresentar as classes *NC* (*Nick Pippenger's Class*) e *RNC* (*Random NC*). Os problemas pertencentes a essas classes possuem algoritmos determinísticos e randômicos respectivamente, que são executados em um tempo polilogarítmico usando um número polinomial de processadores. Os problemas da classe *NC* são altamente paralelizáveis, e consideraremos essa classe como sendo a dos algoritmos paralelos eficientes. A classe *NC*, como veremos posteriormente, apresenta algumas surpresas, pois alguns problemas que possuem algoritmos seqüenciais polinomiais eficientes (classe *P*) parecem não admitir paralelização eficiente (*P-completos*).

1.3 Resumo da Tese

O Capítulo 2 contém uma apresentação do Modelo *PRAM* e algumas técnicas para algoritmos paralelos.

No Capítulo 3 apresentamos um algoritmo para computação de um *circuito de Euler* utilizando *esteio*. A obtenção de algoritmos paralelos eficientes para a computação de *circuitos de Euler* proporcionará melhores algoritmos para diversos problemas de grafos tais como componentes conexos, árvores geradoras, emparelhamentos maximais e decomposição orelha.

Em um grafo com n vértices e m arestas, o melhor algoritmo paralelo existente para determinação de um *circuito de Euler*, requer tempo $O(\log n)$ utilizando m processadores em uma *CRCW PRAM* [6,8,50]. Estes algoritmos são executados em tempo paralelo $O(\log^2 n)$ com m processadores em uma *CREW PRAM*.

O algoritmo apresentado [15,16] é implementado em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores em uma *CREW PRAM*. O algoritmos pode ser implementado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores, desde que os vértices do *grafo bipartido auxiliar* estejam convenientemente rotulados.

No Capítulo 4 utilizamos a definição de *esteio* para computar uma árvore geradora e os componentes conexos de um grafo. A computação de uma árvore geradora e dos componentes conexos de um grafo são problemas básicos em teoria dos grafos e existe uma quantidade considerável de pesquisa no projeto de algoritmos para esses problemas. Os algoritmos paralelos existentes são implementados para uma *EREW PRAM* [61,53], *CREW PRAM* [47,20] e *CRCW PRAM* [7,21,76]. Os algoritmos mais eficientes admitem implantação em uma *CRCW PRAM* em tempo paralelo $O(\log n)$ com $\frac{O((m+n)\alpha(m,n))}{\log n}$ processadores, onde $\alpha(m,n)$ é função inversa de Ackermann (cf. [50]). Para uma *CREW PRAM* o algoritmo foi implementado [20] em tempo paralelo $(\log^2 n)$ com $\frac{n^2}{\log^2 n}$ processadores.

Apresentamos um algoritmo paralelo que utiliza um *esteio* para computar uma árvore geradora e os componentes conexos de um dado grafo $G = (V, E)$. A implementação utiliza uma *CREW PRAM* [50]. O algoritmo é executado em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores. Se o grafo de entrada estiver convenientemente rotulado o algoritmo é ótimo e pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores. Para a classe dos *st-grafos planares* e grafos *série-paralelo*, essa rotulação pode ser facilmente obtida em tempo $O(\log n)$ com $\frac{m}{\log n}$ processadores.

No Capítulo 5 descrevemos algoritmos paralelos para efetuar *busca irrestrita* em um grafo utilizando *kl-caminho*. O algoritmo seqüencial para busca irrestrita é baseado no algoritmo [82] de busca em profundidade (restrita). O problema de busca em profundidade paralela não parece ter uma solução simples como no caso seqüencial, e até o momento, só foram obtidas soluções *NC* para alguns casos particulares de grafos [39,90]. No caso geral, foi provado que a busca lexicográfica em profundidade é um problema *P-completo* [50].

A busca irrestrita em um grafo gera todos os caminhos maximais a partir do vértice r , raiz da busca. O problema do caminho maximal esta relacionado ao de busca em profundidade, pois qualquer *ramo* da árvore de busca em profundidade é um caminho maximal. O algoritmo para computação de um caminho maximal usando o método guloso é *P-completo* [4]. Um algoritmo *RNC* de tempo paralelo $O(\log^5 n)$ com $n^2 M$ processadores, onde M é o número de processadores

para contruir um emparelhamento máximo, foi apresentado por Anderson [5].

Apresentamos algoritmos paralelos para busca irrestrita para algumas classes de grafos. Para digrafos acíclicos, o algoritmo é executado em tempo paralelo $O(\alpha \log n)$ com m processadores em uma *CREW PRAM*, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz. O algoritmo é implementado para grafos gerais em tempo paralelo $O(\alpha\beta \log^2 n)$ com m processadores em uma *CREW PRAM*, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz e β é o número máximo de ciclos do grafo que pode ocorrer na computação de um caminho maximal.

No Capítulo 6 utilizamos o algoritmo paralelo de busca irrestrita para computar as cliques maximais de um *grafo círculo*. Os algoritmos de reconhecimento para *grafos círculos* [12,34,59] são seqüenciais (polinomiais). A existência de um algoritmo paralelo para reconhecimento de *grafos círculo* está em aberto.

Vamos descrever um algoritmo paralelo para geração de todas as cliques maximais de um *grafo círculo* em tempo paralelo de $O(\alpha \log^2 n)$ com n^3 processadores em uma *CREW PRAM*, onde n, m e α são o número de vértices, arestas e cliques maximais do grafo, respectivamente. O algoritmo apresentado é baseado na versão seqüencial apresentado por [79]. A versão seqüencial é uma aplicação de uma orientação especial de um grafo denominada *localmente transitiva*. Vamos também apresentar um algoritmo paralelo para computar o número de cliques maximais α_k de tamanho k , o número total de cliques maximais α e uma clique máxima de um *grafo círculo* em tempo paralelo $O(\log^2 n)$ com n^3 , $nM(n)$ e n^3 processadores, respectivamente, em uma *CREW PRAM*, onde $M(n)$ é o número de processadores necessários para efetuar uma multiplicação de duas matrizes $n \times n$.

As cliques maximais de um grafo qualquer podem ser computadas através do algoritmo [29] em tempo paralelo $O(\alpha \log^3 n)$ com $\alpha^6 n^2$ processadores em uma *CREW PRAM*.

O Capítulo 7 descreve os principais algoritmos seqüenciais para determinação dos ciclos elementares de um digrafo. Os algoritmos de Tiernan [84] e Szwarcfiter e Lauer [81] são descritos em detalhe.

No Capítulo 8 descrevemos algoritmos paralelos para determinação dos ciclos elementares de um digrafo. Para isso necessitamos simular algumas das estruturas que são usadas nos algoritmos seqüenciais.

Apresentamos dois algoritmos paralelos, esses algoritmos são baseados nos algoritmos de Tiernan [84] e Szwarcfiter e Lauer [81] e utilizam a abordagem proposta por [81] de reportar os ciclos assim que forem determinados, nos demais algoritmos os ciclos só são reportados a partir de um vértice inicial.

Os algoritmos determinam um ciclo elementar em tempo paralelo $O(\log^2 n)$ com m processadores em uma *CREW PRAM*. Para a determinação dos demais ciclos elementares, os algoritmos utilizam tempo paralelo exponencial na implementação da versão de Tiernan [84] e $O(\alpha n \log^2 n)$ com n^3 processadores na implementação da versão de Szwarcfiter e Lauer [81] em uma *CREW PRAM*, onde α é o número de ciclos elementares do digrafo.

Capítulo 2

Preliminares

2.1 O Modelo de Paralelismo

Em função da grande variedade de arquiteturas existentes, e de como o problema de comunicação entre os diversos processadores e memórias são abordados nessas arquiteturas, surge a necessidade de idealizar um modelo onde a análise do desempenho de um algoritmo paralelo não dependa das particularidades de cada máquina. Necessitamos de um modelo formal semelhante ao modelo *RAM* para algoritmos seqüenciais. Vários modelos formais de computação paralela aparecem na literatura, mas não existe um consenso de qual deles é o melhor.

Neste trabalho faremos uso de um computador paralelo idealizado, conhecido como *PRAM* (*Parallel Random-Access Machine*). Esse modelo essencialmente despreza restrições de *hardware* que uma especificação de arquitetura pode impor (existem todas as possíveis conexões entre os processadores e os endereços de memória, sendo que o número de processadores e o tamanho da memória global são arbitrários). O modelo dá a liberdade na apresentação dos algoritmos, pois não impõe restrições no paralelismo, o que não ocorre com um *hardware* específico.

O modelo *PRAM*, em função de sua simplicidade e universalidade, tem motivado sua utilização pelos cientistas da computação teórica, sendo um ponto inicial para uma metodologia de computação paralela. Contudo esse modelo ainda não é fisicamente realizável com a atual tecnologia.

A utilização do modelo *PRAM* possibilita a análise de algoritmos projetados para outros modelos, mesmo que esses modelos não utilizem memória compartilhada. Isso pode ser efetuado através da simulação entre modelos de computação distintos [56,87,49,70,45], além disso, o modelo *PRAM* pode ser simulado

em um computador mais *factível* em tempo polilogarítmico [23].

Isso faz com que o modelo *PRAM* seja extremamente utilizado para o estudo da estrutura lógica da computação paralela em um contexto que não aborda problemas de comunicação. Algoritmos desenvolvidos para outros modelos mais *realistas*, são muitas vezes baseados em algoritmos projetados para o modelo *PRAM*. Visto que o custo de simulação de uma *PRAM* em modelos mais factíveis é polilogarítmico, as classes *NC* e *RNC* não se alteram se passarmos de um computador idealizado para um mais realístico. Contudo o grau de dificuldade da especificação do algoritmo pode aumentar consideravelmente. Deste ponto de vista, o modelo *PRAM* se adapta perfeitamente às nossas necessidades para a concepção de algoritmos e para justificar a inclusão de problemas em *NC* ou *RNC*.

2.1.1 O Modelo PRAM

O modelo *PRAM* (*Parallel Random-Access Machine*) de computação paralela que abordaremos é o modelo proposto por [33,41,75]. Este modelo é análogo ao da *RAM* seqüencial [24] e é um modelo de memória compartilhada.

Uma *PRAM* consiste de uma coleção arbitrária de processadores seqüenciais independentes. Cada processador possui sua própria memória particular. Os processadores se comunicam entre si através de uma memória global de tamanho arbitrário. Os processadores trabalham sincronizadamente e em uma unidade de tempo cada processador pode acessar (executando uma leitura ou escrita) qualquer endereço de memória local ou global. Cada processador pode ser considerado como uma *RAM* de custo uniforme com as operações e instruções usuais. O custo de operações aritméticas, de atribuição ou de comparação é constante.

A entrada para um algoritmo PRAM é assumida como sendo previamente atribuída aos endereços de memória. Supomos que cada processador tem conhecimento do tamanho de uma dada entrada de uma instância (a qual pode ser representada por mais de um parâmetro; por exemplo os números de vértices e arestas de um grafo). O número de processadores usados pode variar com o tamanho da entrada, mas não varia ao longo da execução nem com o conteúdo da entrada. O programa é único, não variando com o tamanho da entrada.

Os processadores são arbitrariamente indexados por números naturais P_1, P_2, \dots, P_n os quais podem ser armazenados em um simples endereço de memória. Esses processadores sincronizadamente executam o mesmo programa (através de uma central principal de controle). Embora executando as mesmas instruções, os processadores podem estar trabalhando com dados distintos (dispostos em diferentes endereços de memória). O modelo PRAM é classificado como um

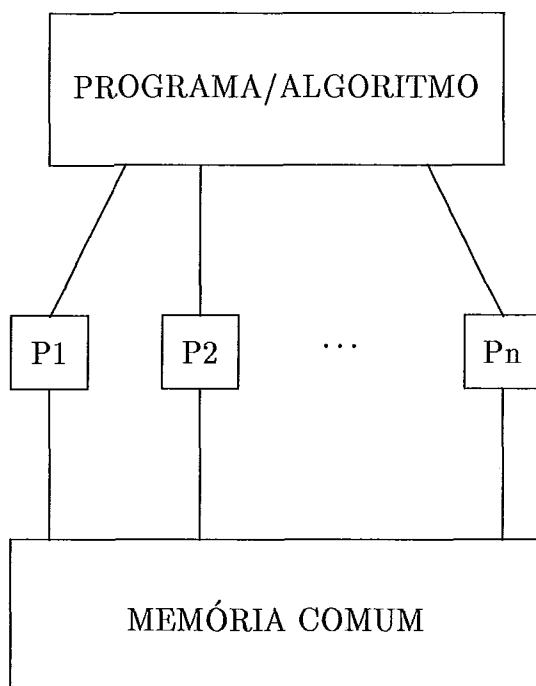


Figura 2.1: Modelo PRAM

modelo *single-instruction, multiple-data stream (SIMD)*. A Figura 2.1 descreve o modelo PRAM.

A motivação teórica para a utilização deste modelo é baseada em um outro modelo, o *modelo de circuitos*. Um algoritmo no modelo de circuitos consiste de uma família de circuitos booleanos, tais que cada circuito da família resolve instâncias de um mesmo tamanho (existe um circuito para cada tamanho da entrada). A saída do algoritmo corresponde ao resultado do circuito. Para corresponder com os requisitos do modelo PRAM de possuir um único programa para todos os tamanhos de entrada, é usualmente exigido que a família de circuitos satisfaça a *condição de uniformidade* [74].

Essa condição estabelece que deve existir um algoritmo satisfazendo certas condições, (por exemplo, utilizando tempo polinomial ou sendo executado em espaço logarítmico), e que possui como entrada um certo número m e produz como saída um circuito booleano que computa uma função cuja entrada possui tamanho m . A saída do algoritmo para uma dada instância m de um problema consiste do resultado do circuito de tamanho m . As duas medidas da complexidade do algoritmo neste modelo são o tamanho e a profundidade do circuito, ambas em função do tamanho da entrada.

Modelos de circuitos são bons para produzir limites inferiores na complexidade do problema, pois uma *porta* em um circuito booleano é mais simples que

um processador em um computador paralelo, e porque a maior parte dos demais modelos de paralelismo pode ser descrita através de circuitos. Por outro lado é extremamente difícil a construção de algoritmos para o modelo de circuitos exatamente em função de sua simplicidade.

Temos que todo circuito booleano uniforme pode ser simulado em um modelo PRAM, através da associação de cada *porta* do circuito com um processador do PRAM. O número de processadores usado é proporcional à profundidade do circuito. Reciprocamente todo algoritmo PRAM pode ser simulado por uma família de circuitos de tamanho proporcional ao número total de operações PRAM, de profundidade proporcional ao tempo paralelo. Nesse caso, porém, um fator logarítmico deve ser adicionado à complexidade do circuito, para exprimir o fato de que circuitos booleanos operam com bits, enquanto algoritmos PRAM manipulam inteiros.

Portanto se um problema pode ser solucionado por um algoritmo PRAM com um número polinomial de processadores no tamanho da entrada, e um tempo polinomial em logaritmo do tamanho da entrada, ele pode ser resolvido por um circuito de tamanho polinomial com profundidade polilogarítmica, e vice versa.

Na utilização de um modelo para a análise da complexidade para algoritmos paralelos, surgem várias diferenças com relação aos algoritmos seqüenciais. Nos algoritmos seqüenciais, nossa análise é direcionada na solução em tempo polinomial e completude para *NP*. Para algoritmos paralelos vários conceitos de complexidade tem sido estudados [52,64,63]. Vamos restringir nossa análise na solução de um problema *P* em tempo *polilog* (i.e., o tempo de computação paralela é limitado por uma potência fixa do logaritmo do tamanho da entrada), usando um número polinomial de processadores.

2.2 Algoritmos Paralelos Eficientes

Da mesma forma que no caso seqüencial, onde existe a preocupação de determinar a que tipo de classe pertence um determinado problema, *P*, *NP*, etc., no caso paralelo é importante identificar quais problemas podem ser eficientemente resolvidos em paralelo.

A eficiência de um algoritmo PRAM é medida por dois critérios, tomados como funções do tamanho da entrada: o número de processadores usados para executar a computação, e a quantidade de tempo paralelo usado pelos processadores na computação. Em ambos os casos, considera-se o pior caso tomados em relação ao tamanho de todas as entradas. Podemos ilustrar com o seguinte exemplo: O algoritmo *X* é executado em tempo paralelo de $O(\log n)$ com n^2 processadores.

Definimos um algoritmo paralelo *eficiente* como sendo o algoritmo que utiliza tempo paralelo polilogarítmico ($O(\log^k n)$ para alguma constante k , onde n é o tamanho da entrada), com um número polinomial de processadores. Os problemas que podem ser resolvidos dentro dessas restrições são considerados como tendo soluções paralelas eficientes e são definidos como pertencentes a classe NC . Logo a classe NC consiste dos problemas que podem ser resolvidos em tempo *polilog* com um número limitado (polinomialmente) de dispositivos de computação (processadores, ou *portas*) por algoritmos determinísticos.

Uma subclasse de problemas de particular interesse é aquela composta pelos problemas que admitem algoritmos paralelos *ótimos*. Um algoritmo paralelo *ótimo* é um algoritmo para o qual o produto do tempo paralelo t com o número de processadores utilizados p é linear no tamanho do problema, isto é, $pt = O(n)$. Otimalidade também pode significar que o produto pt é da mesma ordem do tempo de computação do algoritmo seqüencial mais eficiente conhecido para o problema.

A otimalidade em paralelismo depende do modelo específico que utilizaremos, pois veremos posteriormente que a simulação de um algoritmo projetado para um particular modelo, quando analisado em um outro modelo pode perder um fator polilogarítmico no produto pt . Como exemplo de um algoritmo ótimo, temos o algoritmo de ordenação paralela de Ajtai, Komlós e Szemerédi [2,3] que é executado em tempo paralelo $O(\log n)$ com n processadores.

Para apresentar os algoritmos paralelos utilizaremos uma linguagem tipo *ALGOL* adicionada de instruções inerentemente paralelas, e a essa linguagem denominaremos *PALGOL*.

Os comandos adicionais do *PALGOL* serão descritos como a seguinte instrução:

para todo $x \in X$ **em paralelo faça**
instruções (A)

A execução desse comando consiste de: 1) Um processador é alocado para cada $x \in X$; 2) Os elementos de X são inteiros ou podem ser codificados por inteiros. Então a cada $x \in X$ é associado um processador $P_{code(x)}$ onde $code(x)$ é um inteiro correspondente a x . Supomos que o processador i é endereçado para trabalhar para $x = code^{-1}(i)$ em tempo constante; 3) Todas as instruções (A) são executadas em paralelo pelos processadores alocados a cada $x \in X$.

Em certas ocasiões, considera-se que há necessidade de ativar os pro-

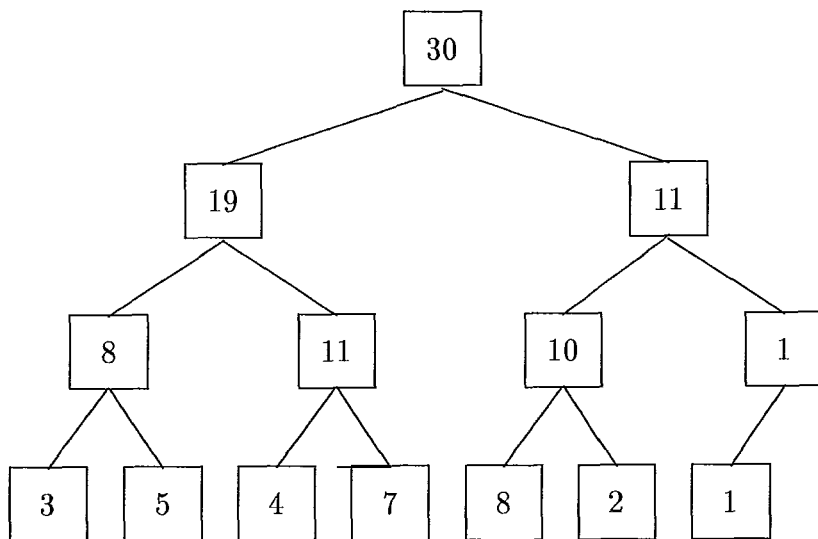


Figura 2.2: Algoritmo Paralelo para Soma

cessadores requisitados. Se esse for o caso, serão necessários $O(\log p)$ passos para se ativarem os p processadores.

procedimento Ativa

ativa-se um processador

enquanto existir um processador não ativado **faça**

 cada processador ativo procura um inativo e ativa-o

A execução termina quando todos os processadores envolvidos terminam a sua computação (individual). A Figura 2.2 ilustra o funcionamento de um algoritmo paralelo que efetua a soma de n números. A complexidade é de tempo paralelo $O(\log n)$ com n processadores.

Todos os logaritmos nesse trabalho serão assumidos como sendo de base 2 a não ser que outra base seja explicitamente estabelecida.

Para qualquer problema para o qual não exista uma solução seqüencial em tempo polinomial conhecida (por exemplo, qualquer problema *NP-completo*) não é de se esperar a existência de uma solução paralela eficiente, utilizando um número polinomial de processadores, pois em função da definição de algoritmo paralelo eficiente, também existiria um algoritmo seqüencial polinomial para o mesmo problema. Contudo, a nossa expectativa é a de encontrar uma solução paralela efi-

ciente para um problema que possua um algoritmo seqüencial em tempo polinomial (i.e., um problema da classe P).

Existem, contudo, vários problemas pertencentes à classe P , os quais parecem não admitir, num primeiro instante uma paralelização eficiente. Esses problemas formam a classe dos problemas P -completos. Usando a técnica de redutibilidade análoga à utilizada na teoria de NP -completude, é possível identificar quais problemas são P -completos. Tais problemas são solucionáveis seqüencialmente em tempo polinomial, mas não pertencem a classe NC .

Da mesma forma como ocorre com a classe NP , se uma solução paralela eficiente puder ser encontrada para qualquer problema P -completo então uma solução similar eficiente existirá para qualquer outro problema dessa classe. Não existe prova, mas uma grande quantidade de evidências circunstanciais indicam que $P \neq NC$.

Uma importante conexão entre computação seqüencial e paralela é ilustrada pela *tese da computação paralela* [11,19,33,41,65,67,89]. A tese da computação paralela estabelece que em alguns modelos de computação paralela, o tempo paralelo é polinomialmente equivalente ao espaço seqüencial. Podemos descrever isso da seguinte forma, seja F uma função qualquer de tamanho n do problema. A tese da computação paralela estabelece que a classe dos problemas que podem ser resolvidos com paralelismo ilimitado em tempo $F(n)^{O(1)}$ é igual a classe dos problemas que podem ser resolvidos por um algoritmo seqüencial em espaço $F(n)^{O(1)}$ sendo F um polinômio no tamanho do problema. O modelo PRAM está entre os modelos de computação paralela para os quais a tese da computação é válida.

No projeto de algoritmos paralelos utilizando o modelo PRAM, existe a possibilidade de se projetar algoritmos que possuam conflitos de leitura e escrita, na qual dois ou mais processadores tentem ler ou escrever concorrentemente em uma mesma posição de memória. As maneiras distintas de como esses problemas são abordados conduzem a diferentes variantes do modelo PRAM.

2.3 Detalhamento do Modelo PRAM

Vamos agora abordar a questão de como resolver os conflitos originados por acessos simultâneos dos processadores a um mesmo endereço de memória. Na solução desse problema adotam-se políticas diferentes, mas isso não significa que o algoritmo irá funcionar apenas para a política específica adotada, pois veremos posteriormente que existem simulações que fazem com que um algoritmo projetado para uma particular escolha possa ser executado em máquinas que utilizem uma forma diferente de resolver os conflitos de acesso simultâneo à memória.

As várias formas de solucionar esses conflitos, possibilitam uma classificação das variações do modelo PRAM [31]. As PRAM podem ser classificadas de acordo com as restrições ao acesso à memória global.

A primeira escolha é simplesmente não permitir tais acessos simultaneamente, e deixar indefinido o comportamento de um programa que tente efetuar tais acessos. Uma máquina com essas características é denominada *EREW PRAM* (*Exclusive-Read Exclusive-Write*), na qual o acesso simultâneo à qualquer posição de memória é proibido tanto para escrita como para leitura. Este modelo é o modelo mais fraco dentre os vários tipos de PRAM, e um algoritmo executável nesse modelo em tempo e número de processadores dados é preferível a outro algoritmo que funcione sobre as mesmas condições de tempo e número de processadores em um modelo mais forte.

O segundo tipo de PRAM é denominado *CREW PRAM* (*Concurrent-Read Exclusive-Write*), esse modelo permite que diversos processadores efetuem a leitura simultaneamente em um mesmo endereço de memória. Novamente, o comportamento do programa que viola essas restrições é indefinido. Por um longo tempo este parece ter sido o único tipo de PRAM usado [54]; esse modelo aparece na literatura desde 1976 [26,46,33].

A terceira e mais forte versão do modelo PRAM, é denominado como *CRCW PRAM* (*Concurrent-Read Concurrent-Write*), esse modelo permite que diversos processadores efetuem leituras e escritas simultâneas em um mesmo endereço de memória.

Necessitamos definir o que acontece quando diversos processadores escrevem simultaneamente em um mesmo endereço de memória valores diferentes. Diversas variedades de CRCW PRAM foram definidas; elas diferem no seu método de resolver os conflitos de escrita. Nesse caso temos que especificar como resolver esses conflitos de escrita.

CRCW FRACO: Neste modelo as escritas simultâneas só são permitidas se todos os processadores que estiverem executando uma escrita concorrente estiverem escrevendo o valor 1 (ou 0) no endereço correspondente. Além disso, por vezes há a restrição adicional de que os endereços que recebem a escrita simultânea só podem conter os valores 0 (ou 1) ao longo de todo processo. Contudo, pode-se provar que o submodelo decorrente dessa restrição adicional é equivalente ao outro. Esse modelo foi introduzido por [54], que mostrou que certos problemas podem ser resolvidos mais rapidamente neste modelo do que nos outros modelos *CRCW* que usualmente eram utilizados anteriormente, e que problemas que são resolvidos em um dado tempo em qualquer versão *CRCW* podem ser solucionados dentro de um fator constante do tempo nessa versão (possivelmente com um número maior de processadores).

CRCW COMUM: Neste modelo não existem restrições nos valores a serem escritos simultaneamente; contudo a escrita simultânea só é permitida quando todos os valores a serem escritos num mesmo endereço forem idênticos.

CRCW ARBITRÁRIO: Processadores podem escrever diferentes valores simultaneamente em um mesmo endereço; e um desses valores escritos será o valor que o endereço conterà no final da operação. Mas o processador que escreveu o valor final é arbitrário, dentre os processadores que desejam escrever os diferentes valores; em particular o resultado pode ser diferente se o mesmo passo for executado uma nova vez, e o algoritmo trabalhará corretamente sem levar em conta qual dos processadores obteve êxito. Esta versão de *CRCW* é uma das mais usadas pelos pesquisadores da área de algoritmos paralelos.

CRCW PRIORIDADE: Existe uma ordenação linear dos números de identificação dos processadores, e o valor escrito em uma escrita simultânea corresponde ao processador de maior identificação entre aqueles que desejavam escrever. Temos que esses identificadores podem ser escolhidos arbitrariamente, logo como no modelo *CRCW ARBITRÁRIO* não é claro a priori qual processador será aquele para ter seu valor como resultado da escrita. Contudo, diferentemente do modelo *CRCW ARBITRÁRIO* a repetição da mesma escrita concorrente dará o mesmo resultado.

CRCW FORTE: Este modelo é menos utilizado que os anteriores, mas ele tem aparecido na literatura [35,21]. Neste modelo o valor escrito em qualquer escrita simultânea é o maior (ou o menor) dentre os valores que os processadores desejavam escrever.

Embora exista uma variedade de modelos *CRCW*, eles não diferem muito em seu poder computacional, pois veremos posteriormente que podemos simular cada passo de um modelo *CRCW FORTE* em um modelo *CRCW FRACO*.

Notamos que cada um dos modelos é pelo menos tão forte quanto seu antecessor, dessa forma os modelos listados acima estão em ordem crescente de *força*. Logo, um algoritmo projetado para um determinado modelo, possivelmente utilizará mais tempo ou recursos computacionais, quando executado em modelo hierarquicamente menos forte. Existem outros modelos *CRCW*, [42] que são intermediários em força entre os listados acima, mas não obdecem uma hierarquia entre eles.

Existe um substancial campo da literatura que explora os limites inferiores de tempo requeridos pelas *EREW*, *CREW* e *CRCW PRAM* para executar simples tarefas computacionais. Com o propósito de provar tais limites inferiores é usual adotar um modelo chamado *PRAM IDEAL* [50]. Limites inferiores provados com um modelo poderoso de computação tem grande generalização porque eles não dependem de suposições sobre o conjunto de instruções ou estrutura interna dos processadores. Tais limites inferiores captam as limitações intrínsecas da memória

global com os recursos de comunicação entre os processadores e evidencia diferenças no poder entre os vários mecanismos de arbitragem das leituras concorrentes e escritas concorrentes.

Uma *PRAM IDEAL* consiste de processadores os quais se comunicam através de uma memória global dividida em posições de memória de capacidade de armazenagem ilimitada. Cada processador tem uma memória particular de tamanho ilimitado e a capacidade para computar em tempo unitário, qualquer função do conteúdo de sua memória particular. Assume-se que os dados de entrada serão armazenados nas posições M_1, \dots, M_n da memória global, e a computação é requerida para terminar sua saída na posição M_1 . A computação procede em passos, com cada passo consistindo de uma fase de leitura, uma fase de computação e uma fase de escrita. Na fase de leitura, cada processador lê para sua memória particular o conteúdo de uma posição na memória global. Na fase de computação, cada processador computa alguma função do conteúdo de sua memória particular. Na fase de escrita, cada processador armazena um valor no conteúdo de sua memória particular em alguma posição da memória global.

Uma *PRAM IDEAL* é denominada como EREW, CREW ou CRCW de acordo com as concorrências de leitura e escrita que forem permitidas, e uma *CRCW PRAM IDEAL* é classificada como COMUM, ARBITRÁRIO, etc., de acordo com o método de resolução do conflito de escrita.

A última distinção que deve ser feita entre os modelos *PRAM* é se eles são *determinísticos* ou *não determinísticos (randômicos)*. Se permitirmos randomização na computação paralela, ela será realizada de acordo com algumas regras básicas. Cada processador possui seu próprio gerador de números aleatórios, com o qual esse processador pode gerar em tempo constante um número extraído de uma distribuição uniforme sobre os números contidos em um endereço de memória. Esses números são próprios, se um processador desejar compartilhar seu número aleatório com outro processador ele deve escreve-lo em um endereço de memória. A distribuição dos números extraídos em um processador qualquer é usualmente tomada como sendo independente dos números extraídos pelos outros processadores, embora alguns algoritmos tenham requisitos de independência fracos.

Quando um algoritmo paralelo utiliza números aleatórios, continua-se a exigir que o número de processadores seja uma função definida no tamanho da entrada; isto é, somente o tempo de execução pode variar aleatoriamente. Os algoritmos que utilizam números aleatórios são divididos em duas classes, Monte Carlo e Las Vegas. O algoritmo Monte Carlo sempre chega a alguma resposta, e com grande probabilidade de ter obtido uma resposta correta. O algoritmo Las Vegas termina com uma resposta com grande probabilidade de estar correta ou com uma mensagem *falha*. Os algoritmos devem ser capazes de detectar quando devem encerrar sua própria execução. O tempo a ser medido num algoritmo aleatório é

tomado como o tempo do pior caso esperado de execução sobre todas as instâncias de um dado tamanho.

Números aleatórios são utilizados em um grande número de algoritmos paralelos para os quais existem soluções seqüenciais determinísticas eficientes. Uma importante utilização dos algoritmos paralelos randômicos, conhecida como *breaking symmetry*, é quando temos um grafo na qual vários nós têm vizinhanças locais semelhantes, mas após o término da execução do algoritmo, nós diferentes devem possuir valores diferentes. Um desses problemas é o da coloração de um grafo.

Existem diversas maneiras em que a randomização pode melhorar o tempo paralelo e os limites dos processadores em um algoritmo. Um algoritmo paralelo randômico pode resolver em tempo polilogarítimo um problema para o qual não existe um algoritmo NC conhecido. A classe dos problemas resolvidos por tais algoritmos é denominada *RNC (Random NC)*. Pertencem a essa classe os problemas de matching [51,36,58], construções de caminhos maximais [5] e árvores de busca em profundidade [1]. Também um algoritmo paralelo randômico pode executar uma tarefa que pode ser feita por um algoritmo determinístico, mas o algoritmo paralelo randômico pode ser mais eficiente que o melhor algoritmo paralelo determinístico conhecido, ou pode funcionar em algum modelo mais fraco de paralelismo, e que também pode ser de mais simples implementação.

2.4 Simulação entre Modelos

Um dos principais motivos no projeto de algoritmos paralelos é o de encontrar algoritmos ótimos e eficientes com tempo paralelo tão pequeno quanto possível. Claramente é mais fácil projetar algoritmos ótimos em uma PRAM que admita conflitos de leitura que em uma PRAM que não admita esses conflitos. Contudo as simulações entre os vários modelos PRAM induzem a noção de que um algoritmo eficiente é invariante com respeito a um modelo particular de PRAM utilizada. Desta forma esta definição é mais robusta.

Consideremos uma computação que possa ser feita em t passos paralelos com m_i operações primitivas no passo i . Se implementarmos esta computação diretamente em uma PRAM para executá-la em t passos paralelos, o número de processadores requeridos é $n = \max m_i$. Se dispormos de $p < n$ processadores, ainda podemos efetivamente simular esta computação observando que o i -ésimo passo pode ser simulado em tempo $\lceil m_i/p \rceil \leq (m_i/p) + 1$, e portanto o tempo paralelo total para simular a computação com p não é maior que $\lceil m/p \rceil + t$, onde $m = \sum_{i=1}^t m_i$. Podemos então simular um certo modelo (qualquer) nele mesmo, diminuindo o número

de processadores às custas de um aumento de tempo.

Teorema 2.4.1 (*Brent*) *Um certo algoritmo A requer t passos paralelos para ser executado. Suponha que A efetue um total de m operações. Então A pode ser executado utilizando p processadores e tempo $O(t + \frac{m}{p})$.*

Demonstração: Seja m_i , o número de operações no passo i , $1 \leq i \leq t$. O passo i pode ser executado com p processadores, se dividirmos essas operações em blocos de tamanho no máximo p , conforme indicado

$$i \equiv \boxed{\bullet \bullet \dots \bullet \bullet} p \boxed{\bullet \bullet \dots \bullet \bullet} p \dots \boxed{\bullet \bullet \dots \bullet \bullet} p \boxed{\bullet \dots \bullet} \leq p$$

no novo esquema são necessários $\lceil \frac{m_i}{p} \rceil$ passos para executar o passo i . Então o tempo total para do novo algoritmo é:

$$\sum_{i=1}^t \lceil \frac{m_i}{p} \rceil \leq \sum_{i=1}^t (\lfloor \frac{m_i}{p} \rfloor + 1) \leq \frac{m}{p} + t$$

Logo A pode ser executado em $O(t + \frac{m}{p})$. \square

Este Teorema, conhecido como princípio de *scheduling* de *Brent* [13], é muitas vezes usado no projeto de algoritmos paralelos eficientes. Pode ser notado que esta simulação assume que o processo de alocação não é um problema, i.e., que é possível a cada um dos p processadores determinar no momento de cada execução, os passos necessários para a simulação, contudo, isto algumas vezes não é uma tarefa trivial.

O próximo teorema simula o modelo CRCW FORTE no EREW. Esse algoritmo resolve o problema de simulação geral entre qualquer par de modelos PRAM, tendo em vista que qualquer algoritmo elaborado para um certo modelo da hierarquia funciona para qualquer outro modelo mais forte dentro da mesma complexidade [60,86].

Teorema 2.4.2 (*Nassimi e Sahni*) *Suponha que um algoritmo CRCW FORTE requiera tempo $O(t)$ e $O(p)$ processadores. Então esse algoritmo pode ser executado num tempo $O(t \log p)$ com $O(p)$ processadores, num modelo EREW.*

Demonstração: Cada passo no modelo CRCW FORTE será simulado em $O(\log p)$ passos do modelo EREW. Vamos efetuar a demonstração em duas partes: a) Simulação da Escrita e b) Simulação da Leitura.

a) - Simulação da Escrita: Suponha que os p processadores p_1, \dots, p_p queiram escrever nos endereços $\alpha_1, \alpha_2, \dots, \alpha_p$, respectivamente, os valores v_1, \dots, v_p .

1) - Inicialmente, os p processadores escrevem, simultaneamente, cada qual uma tripla de valores (p_i, α_i, v_i) .

2) - Em seguida ordena-se essas triplas por endereço-valor formando então uma lista L ordenada.

$$\boxed{p_1 | \alpha_1 | v_1 | p_2 | \alpha_2 | v_2 | \dots | p_p | \alpha_p | v_p}$$

3) - Em seguida, cada processador p_i lê a sua tripla em paralelo.

procedimento Le

se $i = p$ então escreva v_i em α_i

se $i < p$ e $\alpha_i \neq \alpha_{i+1}$ então escreva v_i em α_i

Complexidade da Simulação da escrita.

passo	tempo	# processadores
1	$O(1)$	$O(p)$
2	$O(\log p)$	$O(p)$
3	$O(1)$	$O(p)$

Logo a escrita pode ser simulada em tempo $O(\log p)$ com $O(p)$ processadores.

b) Simulação da Leitura: Supõe-se que p processadores $p_i, 1 \leq i \leq p$, desejam efetuar a leitura. O processador p_i deseja ler o conteúdo do endereço α_i . A idéia é criar uma célula v_i onde o endereço α_i será armazenado.

1) - Inicialmente os p processadores escrevem, simultaneamente, cada qual uma tripla de valores (p_i, α_i, v_i) sendo $v_i = 0$.

2) - Em seguida ordena-se essas triplas por endereço, formando uma lista ordenada.

$$\boxed{p_1 | \alpha_1 | 0 | p_2 | \alpha_2 | 0 | \dots | p_p | \alpha_p | 0}$$

3)- Em seguida cada processador p_i examina sua tripla em paralelo.

procedimento Examina

se $i = 1$ então p_1 lê α_1 e armazena esse valor em v_1

se $i > 1$ e $\alpha_i \neq \alpha_{i-1}$ então p_i lê α_i e armazena o valor lido em v_i

4) - Cada v_i armazenado pode ser propagado nas células v_{i+1}, \dots utilizando-se a idéia de propagação em árvore binária.

5) - Para efetuar a leitura propriamente, basta cada processador p_i , ler em paralelo o valor de v_i .

Complexidade da Leitura

passo	tempo	# processadores
1	$O(1)$	$O(p)$
2	$O(\log p)$	$O(p)$
3	$O(1)$	$O(p)$
4	$O(\log p)$	$O(p)$
5	$O(1)$	$O(p)$

Temos que o algoritmo é executado num tempo $O(t \log p)$ com $O(p)$ processadores. \square

Podemos também simular um modelo CRCW FORTE num CRCW FRACO, e manter o tempo paralelo de processamento, neste caso, devemos aumentar o número de processadores, de p para $O(p^2)$ [54].

Existem várias outras simulações, para uma análise mais completa, sugerimos [31].

Além do modelo PRAM, existem diversos outros modelos de computação paralela, os modelos *Circuitos Uniformes*, *Máquinas de Turing Alternadas* e *Máquinas Vetoriais* estão entre os mais utilizados. Todos esses modelos são equivalentes com relação a complexidade dos algoritmos $O(\log^k n)$, $k \in N$, com um número polinomial de recursos de hardware. Para uma PRAM a quantidade de recursos de hardware é medida pelo número de processadores, para os modelos de *Circuitos Uniformes* e *Máquinas de Turing Alternadas* esses recursos são medidos com a utilização do número de circuitos e do número de configurações possíveis, respectivamente [50].

2.5 Algumas Técnicas Gerais para Algoritmos Paralelos

Como observamos anteriormente, para projetar algoritmos paralelos, devemos tomar uma abordagem diferente da utilizada no desenvolvimento de algoritmos seqüenciais. Por outro lado, no desenvolvimento de algoritmos paralelos, encontramos várias *subrotinas* que são utilizadas freqüentemente nos mais diversos algoritmos paralelos. Vamos descrever aqui algumas técnicas e princípios gerais de uso comum para projetar algoritmos paralelos.

Uma das estruturas básicas utilizadas em algoritmos paralelos é a estrutura de árvore. Em computação paralela esta estrutura aparece de várias maneiras tais como, estrutura da computação, estrutura de dados ou na estrutura dos processadores.

2.5.1 A Técnica da Árvore Binária Balanceada

Este método faz uso de uma árvore binária balanceada. Cada nó interno corresponde a computação de um subproblema com a raiz correspondendo o problema global. Os problemas são resolvidos em uma estratégia *de baixo para cima*, de forma que os subproblemas que pertencerem à mesma profundidade na árvore serão computados em paralelo. Um exemplo típico é o de encontrar o máximo de n elementos. Uma tal computação é mostrada na Figura 2.3, onde n elementos são colocados nas folhas da árvore (nós externos) e a computação se processa de maneira óbvia. Supomos que nesse tipo de computação n é potência de 2. Se isto não ocorrer, então um número (mínimo) de elementos virtuais podem ser adicionados para satisfazer esse requisito. A profundidade da árvore será limitada por $\lceil \log n \rceil$ e a complexidade desse algoritmo será de tempo paralelo $O(\log n)$ com um máximo de $n/2$ processadores.

Para implementarmos esse algoritmo em PALGOL consideraremos $n = 2^m$ e A um vetor de dimensão $2n$. Os n valores os quais pretendemos determinar o máximo serão armazenados nas posições $A[n], A[n + 1], \dots, A[2n - 1]$. E no final da computação $A[1]$ armazenará o resultado.

procedimento MAX

para $k \leftarrow m - 1$ **ate** 0 **passo** -1 **faça**
para $j, 2^k \leq j < 2^{k+1}$ **em paralelo faça**
 $A[j] \leftarrow \max\{A[2j], A[2j + 1]\}$

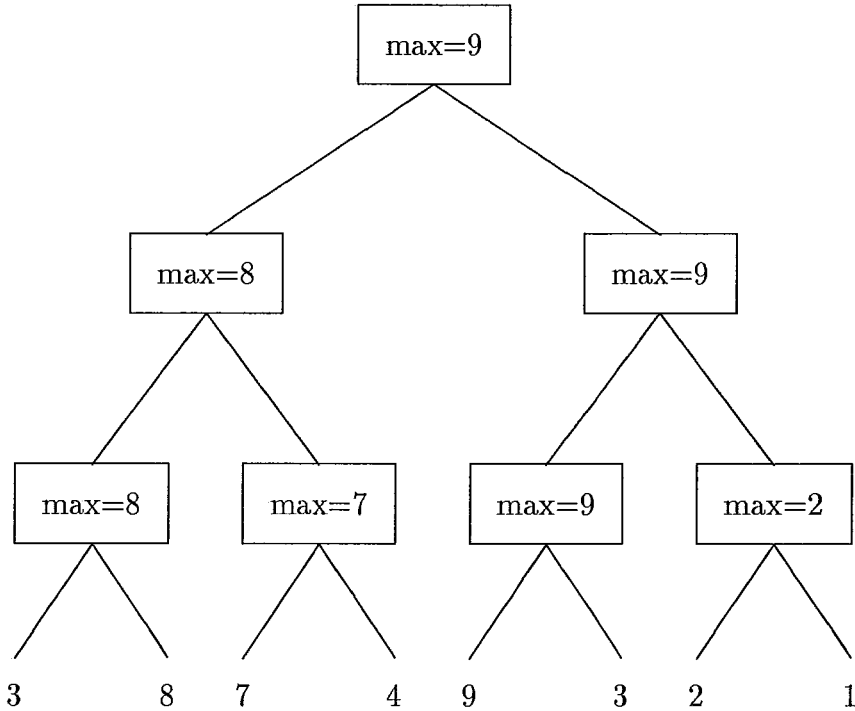


Figura 2.3: Máximo de n Números

O algoritmo é executado em uma EREW PRAM em tempo paralelo $O(\log n)$ com n processadores.

2.5.2 A Técnica da Duplicação Recursiva

Esta técnica é normalmente aplicada em um vetor ou uma lista de elementos. A computação processa-se por uma aplicação recursiva do cálculo de todos elementos a uma determinada distância (na estrutura de dados) de cada elemento individualmente. Essa distância duplica-se a cada passo. Desta forma após k passos a computação terá sido executada (para cada elemento) sobre todos elementos dentro de uma distância de 2^k .

Vamos utilizar essa técnica na solução do seguinte problema: Determinar em paralelo a posição de cada elemento em uma lista L de n elementos, isto é, associar um número $posto[k]$ a cada elemento $1 \leq k \leq n$ onde $posto[k]$ é o número de ordem do elemento k na lista. Vamos tomar este número como sendo a distância do elemento k ao final da lista. A lista é fornecida como um conjunto de elementos, cada um com um ponteiro para o próximo elemento da lista. O ponteiro do elemento k é $próximo[k]$. Se k é o elemento final na lista então $próximo[k] = k$. O algoritmo é então descrito como segue.

Algoritmo Posto

```

para todo  $k \in L$  em paralelo faça
   $P[k] \leftarrow \text{próximo}[k]$ 
  se  $P[k] \neq k$  então  $\text{distância}[k] \leftarrow 1$ 
  senão  $\text{distância}[k] \leftarrow 0$ 
repita  $\lceil \log n \rceil$  vezes
  para todo  $k \in L$  em paralelo faça
    se  $P[k] \neq P[P[k]]$  então
       $\text{distância}[k] \leftarrow \text{distância}[k] + \text{distância}[P[k]]$ 
       $P[k] \leftarrow P[P[k]]$ 
  para todo  $k \in L$  em paralelo faça
     $\text{posto}[k] \leftarrow \text{distância}[k]$ 

```

É fácil observar que o algoritmo computa a posição de cada elemento da lista em tempo paralelo $O(\log n)$ com n processadores. A complexidade decorre do número de reatribuições requeridas por cada um dos $P[k]$, antes de se tornarem o último elemento da lista. Visto que $P[k]$ é *duplicado* recursivamente pela atribuição $P[k] \leftarrow P[P[k]]$, temos que $P[k]$ (\neq do último elemento) está a uma distância 2^i de k após efetuadas i duplicações. Desta forma um número logaritmo de reatribuições é suficiente.

A Figura 2.4 ilustra o algoritmo para uma lista de sete elementos. Cada ponteiro $P[k]$ é denotado como um arco de um elemento para outro e o arco de um elemento k é rotulado com o valor corrente da $\text{distância}[k]$.

Podemos usar a técnica da duplicação recursiva para determinar vários tipos diferentes de cálculos sobre um conjunto de elementos armazenados em uma lista ou um vetor.

2.5.3 A Técnica da Conquista por Divisão

Dentro desta técnica, um dado problema é dividido em um número de subproblemas independentes os quais são recursivamente resolvidos. O tempo de computação paralela para o problema todo é proporcional à profundidade da recursão. A solução para um problema em um nível de recursão deve ser composto da solução de seus subproblemas. Se a cada nível de recursão os subproblemas tem um tamanho que não seja superior a uma proporção fixa do tamanho de seu problema pai, então a profundidade da recursão será logarítmica. A técnica de conquista por divisão é amplamente utilizada em computação seqüencial. Na computação paralela, a técnica

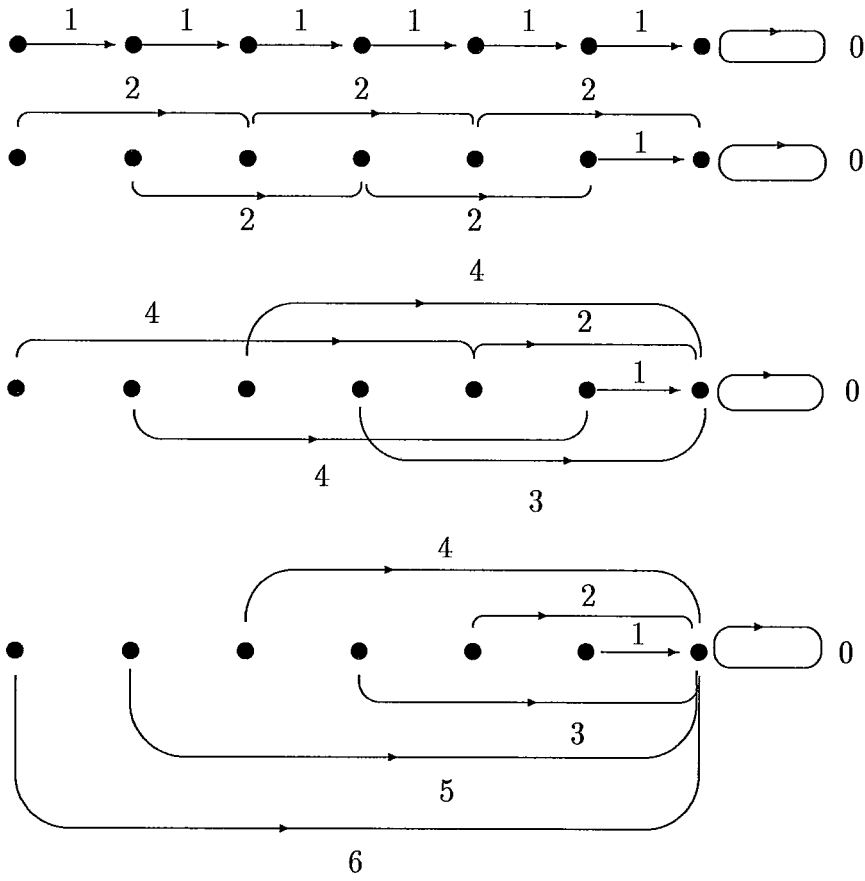


Figura 2.4: Duplicação Rucursiva

requer que os subproblemas em um mesmo nível de recursão devem ser computados independentemente.

A técnica da árvore binária pode ser visto como um caso especial de conquista por divisão, embora conquista por divisão seja uma estratégia *de cima para baixo*. Utilizaremos o mesmo exemplo que foi usado para ilustrar a técnica da árvore binária, o de computar o máximo de n números armazenados em um vetor. Este problema pode ser dividido em dois subproblemas, o primeiro é o de encontrar o máximo dos primeiros $n/2$ elementos e o segundo é o de encontrar o máximo dos últimos $n/2$ elementos. A solução para o problema original é simplesmente o máximo entre os resultados dos dois subproblemas.

Existem diversos outros exemplos onde a técnica de conquista por divisão é utilizada em algoritmos paralelos para problemas envolvendo grafos.

2.5.4 A Técnica de Compressão

Vamos ilustrar a *Técnica de Compressão* considerando novamente o problema de encontrar o máximo entre n elementos armazenados em um vetor X . Por conveniência, consideraremos n como sendo potência de 2 (se necessário podemos sempre adicionar, em tempo paralelo constante, um número mínimo de elementos virtuais para satisfazer esse caso). Recursivamente, para cada valor ímpar de i , em paralelo, comprimimos as duas entradas $X[i]$ e $X[i + 1]$ em uma única entrada com o valor máximo de $(X[i], X[i + 1])$. O comprimento de X é reduzido à metade de um nível de recursão para o outro. Desta forma X será reduzido até ser composto de um único elemento após um número logaritmo de passos. O seguinte exemplo ilustra essa técnica no caso de encontrar o máximo de um conjunto:

$$[3, 7, 8, 3, 9, 2, 3, 1] \rightarrow [7, 8, 9, 3] \rightarrow [8, 9] \rightarrow [9]$$

Notamos que nesse exemplo, a técnica de compressão é similar ao da árvore binária. As várias técnicas até aqui apresentadas não são necessariamente disjuntas. Contudo, a técnica de compressão é bastante aplicada. Por exemplo, em algoritmos paralelos para grafos podemos usar essa técnica para comprimir um grafo recursivamente em um único (super) vértice.

2.5.5 A Técnica da Computação de Prefixos

Seja \star uma operação em um domínio D . Dado um vetor $[x_1, \dots, x_n]$ de n elementos pertencentes a D , o problema da computação de prefixos é o de calcular as n somas

$S_i = x_1 \star \cdots \star x_i = \sum_{j=1}^i x_j$, $i = 1 \cdots n$. Este problema modela diversas aplicações. Por exemplo, considere o problema de compactar um vetor esparso: dado um vetor com n elementos, muitos dos quais iguais a zero em sua ordem original. Podemos computar a posição de cada elemento diferente de zero no novo vetor através da atribuição do valor 1 aos elementos diferentes de zero, e efetuar o cálculo da soma dos prefixos utilizando \star como adição.

Existe um algoritmo seqüencial bastante simples para resolver o problema da soma de prefixos usando $n - 1$ operações, onde o cálculo de S_i é efetuado incrementando-se S_{i-1} , para $i = 2, \dots, n$. Infelizmente esse algoritmo não pode ser paralelizado pois um de seus dois operandos na i -ésima computação depende do resultado da $i - 1$ operação.

O algoritmo paralelo que descreveremos para efetuar o cálculo da soma de prefixos em paralelo foi proposto por [55]. Sem perda de generalidade, supomos que n é potência de 2. Os n números serão armazenados em um vetor A nas posições $A[n], A[n + 1], \dots, A[2n - 1]$.

Algoritmo Soma de Prefixos

```

para  $k \leftarrow m - 1$  passo  $-1$  ate  $0$  faça
  para todo  $j, 2^k \leq j \leq 2^{k+1} - 1$  em paralelo faça
     $A[j] \leftarrow A[2j] + A[2j + 1]$ 
 $B[1] \leftarrow A[1]$ 
para  $k \leftarrow 1$  ate  $m$  faça
  para todo  $j, 2^k \leq j \leq 2^{k+1} - 1$  em paralelo faça
    se  $j = 2p + 1$  então  $B[j] \leftarrow B[(j - 1)/2]$ 
    senão  $B[j] \leftarrow B[j/2] - A[j + 1]$ 

```

Ao final da computação os valores de $A[n] + \cdots + A[n + j]$, para $0 \leq j \leq n - 1$ estarão armazenados nas posições $B[n + j]$. O algoritmo pode ser executado em uma EREW PRAM em tempo paralelo $O(\log n)$ com n processadores. O número de processadores pode ser reduzido para $n / \log n$ usando o mesmo tempo de computação, através de modificações convenientes na entrada do algoritmo [40]. A mesma estrutura desse algoritmo pode ser utilizada para qualquer outra operação associativa.

Capítulo 3

Circuitos de Euler

3.1 Introdução

A obtenção de algoritmos paralelos eficientes para a computação de *circuitos de Euler* proporcionará melhores algoritmos para diversos problemas de grafos tais como componentes conexos, árvores geradoras, emparelhamentos maximais e decomposição orelha.

Em um grafo com n vértices e m arestas, os melhores algoritmos paralelos existentes para determinação de um *circuito de Euler*, requerem tempo paralelo $O(\log n)$ com m processadores em uma CRCW PRAM [6,8,50]. Estes algoritmos requerem $O(\log^2 n)$ em uma CREW PRAM e todos são baseados na determinação de uma árvore geradora de um grafo bipartido derivado do grafo de entrada. Vamos descrever um algoritmo de simples implementação para computar um *circuito de Euler* em um grafo. O primeiro passo do algoritmo é a partição das arestas do grafo de entrada em um conjunto de circuitos disjuntos nas arestas. Ao invés de computar uma árvore geradora [6], a implementação proposta computa um *esteio* do grafo e diretamente identifica os vértices onde a operação denominada *costura* é executada. A operação *costura* transforma dois ou mais circuitos disjuntos passando através de um vértice em um único circuito. Visto que qualquer *grafo euleriano* pode ser decomposto em uma coleção de circuitos arestas disjuntos [30], costurando esses circuitos em determinados vértices obteremos um circuito de Euler.

A implementação [15,16] é efetuada em um modelo CREW PRAM, em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores. O algoritmo é baseado em operações padrões de uma lista tais como ordenação, compressão e partição [22,31,40,43,50]. Utilizamos ordenação inteira [43] e a técnica de *posto ótimo* [22]. Com isso podemos efetuar as operações nas listas em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$

processadores.

Os únicos algoritmos paralelos ótimos conhecidos para ordenação inteira são algoritmos randômicos ou não conservativos. Um algoritmo é denominado *conservativo* quando opera com uma palavra de tamanho maior que $O(\log n)$, onde n é o tamanho do problema. Ordenação inteira determinística pode ser efetuada em tempo paralelo $O(\log n)$ com $\frac{n}{\log n}$ processadores em uma EREW PRAM cuja palavra tem o comprimento de $O(n \log n)$ bits [43]. Reif e Rajasekaran apresentaram um algoritmo paralelo randômico ótimo de tempo $O(\log n)$ [69].

O algoritmo apresentado pode ser implementado em uma CREW PRAM em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores, desde que os vértices do *grafo bipartido auxiliar* estejam convenientemente rotulados.

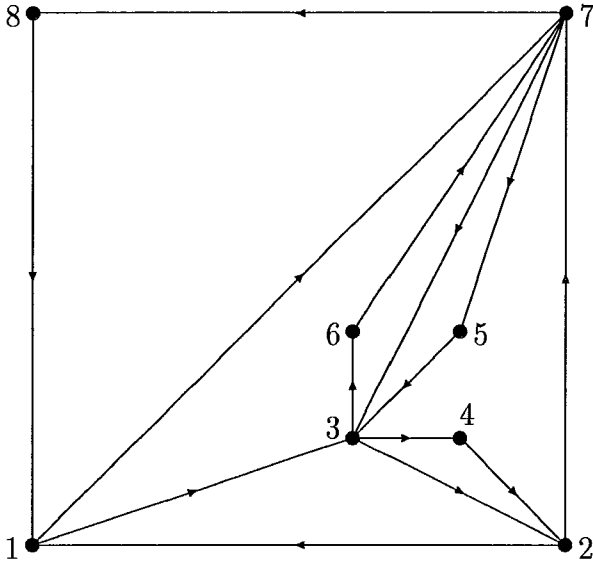
Na próxima seção apresentaremos as definições utilizadas. A descrição preliminar, detalhes da implementação e dois exemplos são descritos nas seções seguintes.

3.2 Notação e Terminologia

Ilustramos as definições e o algoritmo utilizando o grafo dirigido da Figura 3.1. Seja $G = (V, E)$ um grafo dirigido finito com um conjunto de vértices V e um conjunto de arestas E . Um *circuito de Euler* de G é um circuito que percorre cada aresta exatamente uma vez, embora o circuito possa passar por um vértice mais de uma vez. Um grafo dirigido fracamente conexo contendo um circuito de Euler é denominado um *digrafo euleriano*. Temos que todo digrafo euleriano pode ser decomposto em uma coleção de um ou mais circuitos arestas disjuntos, denominada uma *partição de Euler* de G . Nesta seção, um determinado vértice pode aparecer mais de uma vez em um circuito.

Seja $G = (V, E)$ um digrafo euleriano e $C = \{C_1, C_2, \dots, C_k\}$ a decomposição de G em um conjunto de circuitos disjuntos nas arestas, onde C_i representa o i -ésimo circuito. Para o nosso exemplo, C é mostrado na figura 3.1. Seja $V_1 \subseteq V$ o subconjunto de vértices de G pelos quais passam mais de um circuito de C . Seja C' um conjunto em uma correspondência um a um com C . Construímos um grafo bipartido $H = (V_1, C', E_1)$ denominado *grafo bipartido auxiliar* de G tal que as arestas em E_1 conectam um vértice $v_i \in V_1$ com os vértices de C' que correspondem aos circuitos que passam por v_i . Identificamos os circuitos com a menor aresta de cada circuito. Note que V_1 pode ser vazio. Neste caso $C = \{C_1\}$ e temos um circuito de Euler.

Se v é um vértice de um grafo H , então $d_H(v)$ denota o grau de v em



$$C_1 = \{(1,3), (3,2), (2,1)\}, C_2 = \{(1,7), (7,3), (3,6), (6,7), (7,8), (8,1)\} \text{ e}$$

$$C_3 = \{(2,7), (7,5), (5,3), (3,4), (4,2)\}.$$

Figura 3.1: Grafo $G = (V, E)$ e uma partição de Euler de G .

H .

Definimos um *esteio* S em V_1 como uma floresta geradora de H , tal que cada $c'_i \in C'$ é incidente em S com exatamente uma aresta de E_1 , e se (v_j, c'_i) é uma aresta de S implica que não existe uma aresta $(v_k, c'_i) \in H$, $v_k \in V_1$ e $d_H(v_k) > d_H(v_j)$.

Note que um esteio é essencialmente uma coleção de estrelas cujos centros são os vértices em V_1 . Um esteio admite vértices isolados em V_1 mas não em C' . Veja Figura 3.2.

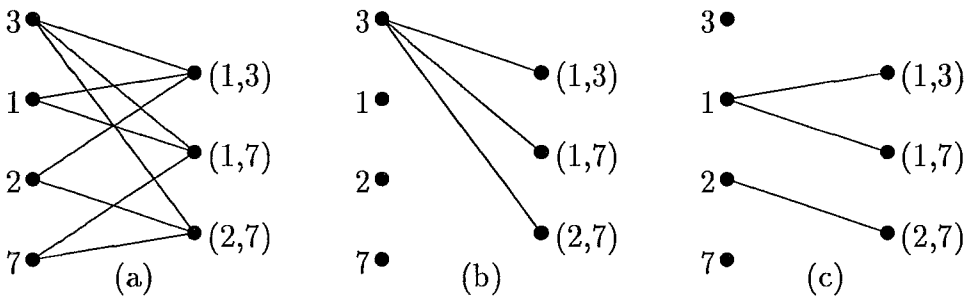


Figura 3.2: (a) Grafo Bipartido Auxiliar H , (b) um Esteio S em H e (c) um não Esteio em H .

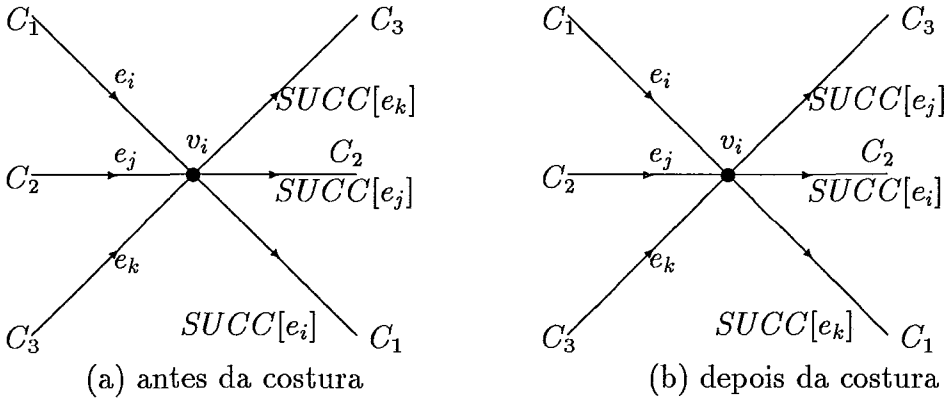


Figura 3.3: Uma costura em v_i

Para um esteio em C' , as regras para os conjuntos V_1 e C' , na definição acima são trocadas.

Um vértice $v \in V_1$ é denominado *zero diferença* em S se $d_H(v) - d_S(v) = 0$.

Sejam e_1, e_2, \dots, e_k , $k \geq 2$, as arestas de G , entrando em um mesmo vértice $v_i \in V$, cada e_j pertencendo a um circuito disjunto diferente C_j na decomposição de Euler C . Uma *costura* em v_i é uma operação que junta todos os circuitos que passam através de v_i em um único circuito. A Figura 3.3 mostra uma costura em v_i .

3.3 Algoritmo para Determinação de um Circuito de Euler

Nesta seção descrevemos um algoritmo para determinação de um circuito de Euler em digrafos eulerianos que é executado em tempo $O(\log^2 n)$ utilizando $\frac{m}{\log n}$ processadores em uma CREW PRAM. O algoritmo pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores, desde que os vértices do grafo bipartido auxiliar estejam rotulados obedecendo a seguinte ordem: (i) para todo $v_i \in V$, $1 < i \leq n$, v_i tem pelo menos um adjacente v_j tal que $j < i$; ou (ii) para todo $v_i \in V$, $1 \leq i < n$, v_i tem pelo menos um adjacente v_j tal que $j > i$. Utilizamos o exemplo da figura 3.1 para ilustrar os passos do algoritmo.

Caso a entrada seja um grafo euleriano não orientado, podemos efetuar uma orientação das arestas a fim de obter um digrafo euleriano em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM [6,40].

3.3.1 Descrição Preliminar do Algoritmo

Algoritmo: Circuito de Euler

1. Particionar as arestas do digrafo de entrada $G = (V, E)$ em um conjunto de circuitos disjuntos nas arestas $C = \{C_1, C_2, \dots, C_k\}$.
2. Computar os vértices V_1 em G pelos quais passam mais de um circuito de C . Computar o grafo bipartite auxiliar $H = (V_1, C', E_1)$ onde C' é o conjunto de vértices correspondendo a cada circuito de C .
3. Computar um esteio S em V_1 para H . Acrescentar a S uma aresta de $E_1 - S$ para cada um dos vértices não zero diferença de V_1 . Efetuar uma costura nos vértices $v_i \in V_1$ que possuam mais de uma aresta em S unido com as arestas adicionadas.
4. Aplicar os passos 2 e 3 até que um circuito de Euler seja encontrado.

3.3.2 O Algoritmo e um Exemplo

Nesta seção, apresentamos o algoritmo para computação de um circuito de Euler, em detalhe, com um exemplo. Os vetores $EDGE$, $EDGE'$, $SUCC$, $CYCREP$ e $STITCH$ contêm m elementos, cada elemento armazena uma *aresta* representada por $(vértice1, vértice2)$. O vetor POS contém m elementos, cada $POS[i]$ dando o índice de $SUCC[j]$ onde j é o índice em $EDGE$ da aresta armazenada em $SUCC[i]$. Quando o algoritmo termina, $SUCC[i]$ contém o sucessor de $EDGE[i]$ no circuito de Euler. Os vetores $BEDGE$ e $BEDGE'$ contêm m elementos onde cada elemento é um par, um *vértice* e uma *aresta*. Cada um desses pares corresponde a uma aresta do grafo bipartite auxiliar H . Os vetores D_G , D_H , $D_{H'}$, D_S e $D_{S'}$ são vetores de inteiros utilizados para armazenar valores temporários.

Agora descrevemos o algoritmo. A entrada é $G = (V, E)$, um digrafo euleriano. A lista das arestas E é armazenada em $EDGE$. Para nosso exemplo, o vetor $EDGE$ é

(2,1)(8,1)(7,8)(6,7)(3,2)(4,2)(1,3)(5,3)(7,3)(3,4)(7,5)(3,6)(1,7)(2,7)

Passo 1

Neste passo encontramos uma partição de Euler para $G = (V, E)$. Podemos efetuar essa partição executando uma ordenação em $EDGE$ na ordem

definida por: Dadas duas arestas (i, j) e (k, l) então $(i, j) < (k, l)$ se $(j < l)$ ou $((j = l) \text{ e } (i < k))$. O vetor $EDGE$ fica

$$\boxed{(2,1)(8,1)(3,2)(4,2)(1,3)(5,3)(7,3)(3,4)(7,5)(3,6)(1,7)(2,7)(6,7)(7,8)}$$

Efetuamos uma cópia do vetor $EDGE$ em $SUCC$ e reordenamos os elementos na ordem definida por $(i, j) < (k, l)$ se $(i < k)$ ou $((i = k) \text{ e } (j < l))$. Então o vetor $SUCC$ fica

$$\boxed{(1,3)(1,7)(2,1)(2,7)(3,2)(3,4)(3,6)(4,2)(5,3)(6,7)(7,3)(7,5)(7,8)(8,1)}$$

Com a definição do sucessor de cada $e_i \in EDGE$ como sendo a aresta $e'_i \in SUCC$, $1 \leq i \leq n$, os vetores $EDGE$ e $SUCC$ decompõe o digrafo G em uma coleção de circuitos aresta disjuntos. Os circuitos são computados com a utilização de duplicação recursiva.

$$C_1 = \{(2,1)(1,3)(3,2)\}, C_2 = \{(8,1)(1,7)(7,3)(3,6)(6,7)(7,8)\} \text{ e} \\ C_3 = \{(4,2)(2,7)(7,5)(5,3)(3,4)\}.$$

Para a computação dos representantes de cada um dos circuitos C_i , necessitamos conhecer a posição j em $EDGE$ na qual $SUCC[i]$ está armazenada. Para isso, durante a ordenação que foi efetuada cada aresta leva com ela um registro de sua posição original ($POS[i]$) no vetor ordenado $EDGE$. Para nosso exemplo, POS fica

$$(5,11,1,12,3,8,10,4,6,13,7,9,14,2).$$

Utilizando ordenação paralela inteira [43] e a técnica ótima de determinação do posto de uma lista em paralelo [22], este passo pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Passo 2

Neste passo, construímos um grafo bipartido auxiliar $H = (V_1, C', E_1)$ como foi definido na seção 2. O conjunto de vértices C' corresponde aos circuitos encontrados no final do passo 1. Por esta razão representamos cada circuito pela sua aresta lexicograficamente menor. O conjunto V_1 é o subconjunto de vértices em

G os quais têm mais de um circuito passando através deles. Ao final desse passo, $BEDGE$ conterá o conjunto de arestas E_1 do grafo H .

Primeiro determinamos os *representantes dos circuitos* para cada aresta e , que é a aresta que representa o circuito contendo e . O vetor $CYCREP[i]$ contém o representante do circuito da aresta $SUCC[i]$. Para o nosso exemplo, o vetor $CYCREP$ computado é:

$$\boxed{(1,3)(1,7)(1,3)(2,7)(1,3)(2,7)(1,7)(2,7)(2,7)(1,7)(1,7)(2,7)(1,7)(1,7)}$$

Usando a técnica ótima de determinação do posto de uma lista em paralelo [22] isto pode ser efetuado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Usando o representante de cada um dos circuitos, computamos o número de circuitos que passam por um dado vértice. Isso pode ser efetuado da seguinte maneira. Copiar $EDGE$ em $EDGE'$ e particionar $EDGE'$ em sublistas disjuntas, uma para cada vértice. Para nosso exemplo, após efetuar a partição do vetor $EDGE'$ temos

$$\boxed{(2,1)(8,1)|(3,2)(4,2)|(1,3)(5,3)(7,3)|(3,4)|(7,5)|(3,6)| (1,7)(2,7)(6,7)|(7,8)}$$

Contudo, em cada sublista podemos ter arestas que pertencem ao mesmo circuito. No nosso exemplo $(1,7)$ e $(6,7)$ pertencem ao mesmo circuito. Para a construção do grafo bipartite auxiliar, todas as sublistas devem ter uma única aresta para cada circuito. Retiramos as arestas duplicadas efetuando em paralelo uma compressão em cada sublista. A cardinalidade de cada uma dessas sublistas obtidas após a compressão dará o número de circuitos que passam por cada vértice. Computamos a cardinalidade da sublista j e armazenamos em $D_G[j]$. No nosso exemplo, após a compressão o vetor $EDGE'$ com $CYCREP$ e D_G são

$$\begin{array}{c} \boxed{(2,1)(8,1)|(3,2)(4,2)|(1,3)(5,3)(7,3)|(3,4)|(7,5)|(3,6)| (1,7)(2,7)|(7,8)} \\ \boxed{(1,3)(1,7)|(1,3)(2,7)|(1,3)(2,7)(1,7)|(2,7)|(2,7)|(1,7)| (1,7)(2,7)|(1,7)} \\ D_G = (2,2,3,1,1,1,2,1) \end{array}$$

Visto que somente estamos interessados em vértices tendo pelo menos dois circuitos passando por eles, retiramos as sublistas de $EDGE'$ que possuem um único elemento. Para o nosso exemplo $CYCREP$ fica

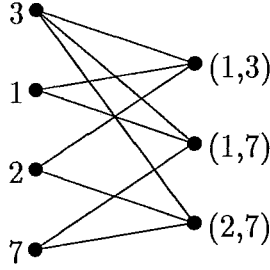


Figura 3.4: Grafo Bipartido Auxiliar $H(V_1, C', E_1)$.

$$\frac{(2,1)(8,1)|(3,2)(4,2)|(1,3)(5,3)(7,3)|(1,7)(2,7)}{(1,3)(1,7)|(1,3)(2,7)|(1,3)(2,7)(1,7)|(1,7)(2,7)}$$

O conjunto dos vértices V_1 e C' podem ser facilmente obtidos de $EDGE'$ e $CYCREP$. No nosso exemplo $V_1 = \{1,2,3,7\}$ e $C' = \{(1,3),(1,7),(2,7)\}$. Podemos computar o conjunto das arestas E_1 da seguinte forma

para todo $i \in V_1$ em paralelo faça

$$BEDGE[i] \leftarrow (EDGE'[i][v\acute{e}rtice2], CYCREP[i])$$

Computamos a cardinalidade de cada vértice $v_j \in V_1$ em $D_H[j]$. Para o nosso exemplo, $H = (V_1, C', E_1)$ é mostrado na Figura 3.4 e o vetor $BEDGE$ que representa E_1 e D_H são

$$\frac{(1,(1,3))(1,(1,7))(2,(1,3))(2,(2,7))(3,(1,3))(3,(2,7))(3,(1,7)) (7,(1,7))(7,(2,7))}{D_H=(2,2,3,2)}$$

Usando a técnica ótima de determinação do posto de uma lista em paralelo [22] e ordenação inteira paralela [43] o passo 2 pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Passo 3

Neste passo transformamos grupos de dois ou mais circuitos de C em um circuito para cada grupo. Para isso necessitamos determinar os vértices onde efetuaremos a costura nos circuitos. Ao final deste passo a lista $STITCH$ conterá as arestas cujos sucessores devem ser mudados para efetuar a operação costura e $EDGE$ e $SUCC$ conterão uma nova partição de Euler para $G = (V, E)$.

Agora computamos um esteio S em V_1 no grafo bipartite auxiliar $H = (V_1, C', E_1)$. Inicialmente efetuamos uma cópia do vetor $BEDGE$ em $BEDGE'$. Após isso, particionamos $BEDGE'$ em um conjunto de sublistas, uma para cada representante dos circuitos. Para isso efetuamos uma ordenação em $BEDGE'$ na ordem definida como segue: Dadas duas arestas (v_i, C_j) e (v_k, C_l) então $(v_i, C_j) < (v_k, C_l)$ se $v_j < v_l$ ou $((v_j = v_l) \text{ e } (v_i < v_k))$. Para nosso exemplo $BEDGE'$ fica

$$\boxed{(1,(1,3))(2,(1,3))(3,(1,3))(1,(1,7))(3,(1,7))(7,(1,7))(2,(2,7)) (3,(2,7))(7,(2,7))}$$

Para cada uma das sublistas escolhemos uma aresta (v_i, C_j) tal que o grau de v_i seja o maior na sublista. Armazenamos as arestas escolhidas em $BEDGE'$ e a cardinalidade de cada vértice v_i em $D_S[v_i]$. Para o nosso exemplo $BEDGE'$ e D_S ficam

$$\boxed{(3,(1,3))(3,(1,7))(3,(2,7))}$$

$$D_S = (0, 0, 3, 0)$$

Note que o vetor $BEDGE'$ acima representa um esteio S em V_1 para o grafo bipartite auxiliar $H = (V_1, C', E_1)$.

Agora para cada vértice não zero diferença (relembramos que um vértice $v_i \in V_1$ é denominado zero diferença se $D_H[v_i] - D_S[v_i] = 0$, em nosso exemplo os vértices v_1, v_2 e v_7 são não zero diferença), adicionamos a $BEDGE'$ uma aresta arbitrária que pertence a $BEDGE$ mas que não foi selecionada no esteio S . Armazenamos a cardinalidade de cada vértice v_i das arestas $(v_i, C_j) \in BEDGE'$ em $D_{S'}[v_j]$. Então os vetores $BEDGE'$ e $D_{S'}$ ficam

$$\boxed{(1,(1,3))(2,(1,3))(3,(1,3))(3,(1,7))(3,(2,7))(7,(1,7))}$$

$$D_{S'} = (1, 1, 3, 1)$$

Temos que $D_{S'}[v_i]$ informa o número de circuitos que serão unidos por meio de uma costura no vértice v_i para formar um único circuito. Pelo fato de termos um único vértice zero diferença, o vetor $BEDGE'$ acima representa uma árvore geradora para $H = (V_1, C', E_1)$.

O vetor $STITCH$ é formado pelas arestas incidentes ao vértice v_i e pertencem ao circuito C_j , tais que $(v_i, C_j) \in BEDGE'$ e $D_{S'}[v_i] > 1$. Isto pode ser efetuado com a utilização de um ponteiro para cada endereço $EDGE'[i]$, e transportando esse ponteiro na construção do vetor $BEDGE$. Para o nosso exemplo, o vetor $STITCH$ fica

$$\boxed{(1,3)(7,3)(5,3)}$$

Para cada aresta em $STITCH$, modificamos o sucessor em $SUCC$ como segue:

para todo $e \in STITCH$ em paralelo faça

$$SUCC[STITCH[i]] \leftarrow SUCC[STITCH[(i + 1) \bmod k]]$$

onde k é $D_{S'}[STITCH[i][v\acute{e}rtice2]]$

A nova partiao de Euler (circuito de Euler)  

EDGE

$$\boxed{(2,1)(8,1)(3,2)(4,2)(1,3)(5,3)(7,3)(3,4)(7,5)(3,6)(1,7)(2,7)(6,7)(7,8)}$$

SUCC

$$\boxed{(1,3)(1,7)(2,1)(2,7)(3,6)(3,2)(3,4)(4,2)(5,3)(6,7)(7,3)(7,5)(7,8)(8,1)}$$

Usando a t cnica  tima de determinaao do posto de uma lista em paralelo [22] e ordenaao inteira paralela [43] o passo 3 pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Passo 4

Usando os vetores *EDGE* e *SUCC* determinados no final do passo 3 como entrada, aplicamos os passos 2 e 3 at  que um circuito de Euler para G seja encontrado. O n mero m ximo de interaoes   da ordem de $O(\log n)$. A demonstraao   apresentada na pr xima subseao.

3.4 Correao e An lise

Lema 3.4.1 *Seja $H = (V_1, C', E_1)$ um grafo bipartite e S um esteio em V_1 . Seja H' o grafo obtido de H adicionando precisamente a S uma aresta de $E_1 - S$ incidente a cada v rtice n o zero diferena de V_1 . Ent o H'   ac clico. Al m disso, se V_1 cont m exatamente um v rtice zero diferena ent o H'   uma  rvore geradora de H .*

Demonstração: S é essencialmente uma coleção de estrelas cujos centros são vértices em V_1 . Adicionando precisamente uma aresta de $E_1 - S$ incidente a cada vértice não zero diferença de V_1 , cada estrela cujo centro é um vértice não zero diferença v_j será conectada com no máximo uma estrela distinta cujo centro é v_i . Então H' é acíclico.

Visto que, pela definição de esteio, o grau de cada c'_i é exatamente 1, existem exatamente $|C'|$ arestas em S . V_1 contém exatamente um vértice zero diferença, então o número de arestas adicionadas é $|V_1| - 1$. Portanto H' tem $|V_1| + |C'| - 1$ arestas, e H' é uma árvore geradora de H . \square

É fácil ver que se V_1 contém exatamente k vértices zero diferença e em $E_1 - H'$ existem precisamente $k - 1$ arestas que conectam circuitos distintos então H' juntamente com essas $k - 1$ arestas forma uma árvore geradora de H .

Teorema 3.4.1 *O número de circuitos arestas disjuntos que restam após o Passo 3, é pelo menos dividido por 2 em cada iteração.*

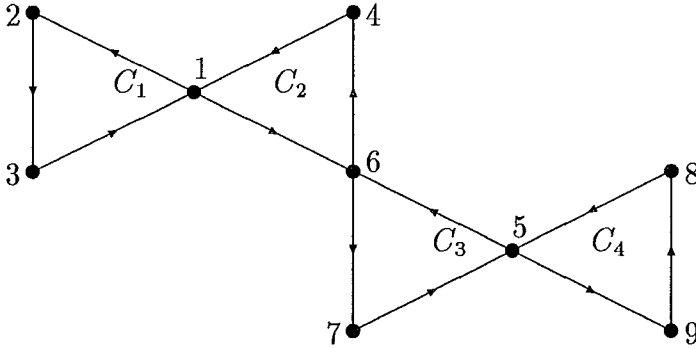
Demonstração: O grafo H' é uma floresta geradora com k árvores, onde k é o número de vértices zero diferença em V_1 . Seja v_i um vértice zero diferença. Visto que todos os vértices em V_1 tem mais que um circuito passando por eles então, $D_{H'}[v_i] > 1$. Após a operação de costura o número de circuitos da partição de Euler passando por v_i é um. O resultado segue. \square

Portanto, no Passo 4, o algoritmo itera no máximo $O(\log n)$ vezes no pior caso.

Teorema 3.4.2 *O algoritmo determina um circuito de Euler para um grafo em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Lema 3.4.2 *Seja $H = (V_1, C', E_1)$ um grafo bipartite tal que os vértices de G estão rotulados de acordo com uma das duas condições: (i) para todo vertice $v_i \in V_1$, $1 < i \leq n$, v_i está conectado a um vértice $v_j \in V_1$, $j < i$, através de um caminho passando por um único vértice em C' ; (ii) para todo vértice $v_i \in V_1$, $1 \leq i < n$, v_i está conectado a um vértice $v_j \in V_1$, $j > i$, através de um caminho passando por um único vértice de C' . Então H possui apenas um vértice zero diferença.*

Demonstração: Vamos considerar o caso (i), o (ii) pode ser demonstrado de forma análoga. Na computação de um esteio para o grafo bipartite auxiliar H , como vimos na demonstração do lema 3.4.1, não necessitamos escolher os vértices de maior



$C_1 = \{(1,2),(2,3),(3,1)\}$, $C_2 = \{(4,1),(1,6),(6,4)\}$, $C_3 = \{(6,7),(7,5),(5,6)\}$
e $C_4 = \{(8,5),(5,9),(9,8)\}$.

Figura 3.5: G_1 e uma partição de Euler.

grau. Para cada um dos vértices que representam os circuitos, podemos selecionar as arestas incidentes com os vértices de maior rótulo. Logo as arestas que saem do vértice de maior rótulo v_k são todas selecionadas. Em função de que para todo vértice $v_i \neq v_k$ tem pelo menos um caminho que o conecta a v_j , $j > i$, pelo menos uma aresta saindo de v_i que não será selecionada. Logo, apenas o vértice v_k será um vértice zero diferença. \square

Portando, nesse caso em uma única iteração o algoritmo computa um circuito de Euler.

Teorema 3.4.3 *Se os vértices do grafo bipartite auxiliar estiverem rotulados de acordo com o lema anterior, o algoritmo computa um circuito de Euler em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

3.5 Um Exemplo Não Ótimo

Para ilustrar as iterações no Passo 4 mais claramente, consideremos o exemplo dado pela Figura 3.5. Os vetores $EDGE$ e $SUCC$ são

(3,1)(4,1)(1,2)(2,3)(6,4)(7,5)(8,5)(1,6)(5,6)(6,7)(9,8)(5,9)
(1,2)(1,6)(2,3)(3,1)(4,1)(5,6)(5,9)(6,4)(6,7)(7,5)(8,5)(9,8)

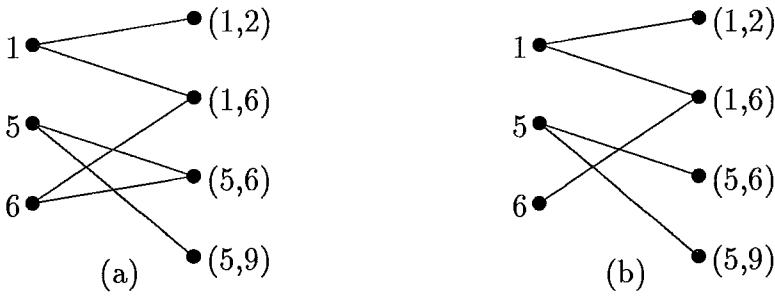


Figura 3.6: (a) Grafo Bipartido Auxiliar H_1 e (b) Esteio S em H_1 com uma aresta adicionada a cada vértice não zero diferença.

O grafo bipartite auxiliar e o esteio S em V_1 com as arestas adicionadas aos vértices não zero diferença são mostrados Figura 3.6. O esteio determinado no passo 3 da primeira iteração tem dois vértices zero diferença. Portanto a adição de mais uma aresta no Passo 3 não determina uma árvore geradora. Os vetores $BEDGE$, $BEDGE'$ e $STITCH$ tornam-se

$$\begin{array}{c} \boxed{(1,(1,2))(1,(1,6))(5,(5,6))(5,(5,9))(6,(1,6))(6,(5,6))} \\ \boxed{(1,(1,2))(1,(1,6))(5,(5,6))(5,(5,9))} \\ \boxed{(3,1)(4,1)(7,5)(8,5)} \end{array}$$

Ao final da primeira iteração, os vetores $EDGE$ e $SUCC$ são

$$\begin{array}{c} \boxed{(3,1)(4,1)(1,2)(2,3)(6,4)(7,5)(8,5)(1,6)(5,6)(6,7)(9,8)(5,9)} \\ \boxed{(1,6)(1,2)(2,3)(3,1)(4,1)(5,9)(5,6)(6,4)(6,7)(7,5)(8,5)(9,8)} \end{array}$$

Agora restam apenas dois circuitos arestas disjuntos,

$$C_1 = \{(1,2),(2,3),(3,1),(1,6),(6,4),(4,1)\} \text{ e } C_2 = \{(6,7),(7,5),(5,9),(9,8),(8,5),(5,6)\}.$$

Repetindo os passos 2 e 3, o grafo bipartido resultante é mostrado na Figura 3.7.

Após a segunda iteração, a nova partição de Euler (Circuito de Euler) para G_1 é

$EDGE$

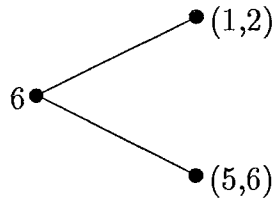


Figura 3.7: Grafo Bipartido Auxiliar H_1 na segunda iteração.

(3,1)(4,1)(1,2)(2,3)(6,4)(7,5)(8,5)(1,6)(5,6)(6,7)(9,8)(5,9)

SUCC

(1,6)(1,2)(2,3)(3,1)(4,1)(5,9)(5,6)(6,7)(6,4)(7,5)(8,5)(9,8)

3.6 Uma Simplificação do Algoritmo

O algoritmo para computação de um circuito de Euler para um digrafo euleriano $G = (V, E)$ pode ser bastante simplificado. Essa simplificação é no sentido de evitar as ordenações que são efetuadas nos passos do algoritmo.

Inicialmente, consideramos a entrada do algoritmo dada pelas listas de adjacências de entrada e saída de seus vértices. Computamos o posto das listas e emparelhamos as arestas de saída com as arestas de entrada que possuam o mesmo posto. Esse emparelhamento dá os sucessores para cada uma das arestas de G . Logo computamos uma partição de Euler para G sem efetuar ordenação.

Ao computarmos o esteio no grafo bipartite auxiliar $H = (V_1, C', E_1)$, não necessitamos ordenar os vértices. Podemos selecionar para cada uma das arestas saindo dos vértices $v \in C'$, a aresta que chegue ao vértice de maior grau de V_1 . Ao escolhermos os vértices de maior grau para selecionar as arestas, estamos apenas diminuindo o número de iterações necessárias para a computação de um circuito de Euler. Neste caso, podemos tomar uma aresta qualquer saindo de cada um dos vértices que representam os circuitos.

Capítulo 4

Árvores Geradoras e Componentes Conexos

4.1 Introdução

A computação de uma árvore geradora e dos componentes conexos de um grafo são problemas básicos em teoria dos grafos e existe uma quantidade considerável de pesquisa no projeto de algoritmos para esses problemas. Os algoritmos seqüenciais usam busca em profundidade ou em largura para resolver eficientemente esses problemas. Os algoritmos paralelos existentes para a computação de uma árvore geradora e dos componentes conexos de um grafo evitam esses procedimentos, visto que não são conhecidos algoritmos paralelos eficientes para busca em profundidade e em largura. Esses algoritmos utilizam a abordagem proposta por *Hirschberg et al.* [47], onde supervértices do grafo são continuamente combinados em supervértices maiores. Esta abordagem foi implementada para uma EREW PRAM [61,53], CREW PRAM [47,20] e CRCW PRAM [7,21,76]. O mais eficiente desses algoritmos admite uma implementação em uma CRCW PRAM em tempo paralelo $O(\log n)$ com $\frac{O((m+n)\alpha(m,n))}{\log n}$ processadores, onde $\alpha(m, n)$ é função inversa de Ackermann (cf. [50]).

Como vimos no capítulo anterior, um *esteio* [15,16] é uma floresta geradora especial para um grafo bipartido G . Apresentamos um algoritmo paralelo que utiliza um esteio para computar uma árvore geradora e os componentes conexos de um dado grafo $G = (V, E)$ [17,14]. Na computação de uma árvore geradora, o algoritmo tem como entrada um grafo bipartido conexo. No caso de termos um grafo geral, fazemos uma subdivisão de cada uma das arestas do grafo para que o grafo fique bipartido. Se o grafo bipartido obtido não for conexo, o algoritmo computará uma árvore geradora para cada componente. A implementação utiliza uma CREW PRAM [50]. O algoritmo é executado em tempo paralelo $O(\log^2 n)$ com

$\frac{m}{\log n}$ processadores, e é de simples implementação visto que é baseado em operações padrões em uma lista, tais como ordenação, compressão e partição, que são descritas em [31,40,43,50].

Se o grafo de entrada estiver rotulado de forma conveniente o algoritmo é ótimo e pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores. Para a classe dos *st-grafos planares* e grafos *série-paralelo*, essa rotulação pode ser facilmente obtida em tempo $O(\log n)$ com $\frac{m}{\log n}$ processadores.

Na próxima seção apresentaremos as definições utilizadas. A descrição preliminar, detalhes de implementação e exemplos são descritos nas seções seguintes.

4.2 Notação e Terminologia

Um *st-grafo planar* G é um grafo orientado acíclico com exatamente um vértice fonte s e exatamente um vértice sumidouro t , que admite uma representação plana tal que s e t estão na fronteira da face externa.

Seja $G = (V, E)$ um grafo. Dizemos que duas arestas $e_1 = (u, w)$ e $e_2 = (w, v)$ de G estão em *série* se o vértice comum às arestas w tem grau 2. Duas arestas e_1 e e_2 estão em *paralelo* se elas possuem o mesmo conjunto de vértices finais. Definimos uma *composição em série* de uma aresta $e = (u, v)$ como a substituição de e por duas arestas em série, i.e., por (u, w) e (w, v) , onde w é um novo vértice. Definimos uma *composição em paralelo* de uma aresta $e = (u, v)$ como a substituição de e por duas arestas e_1 e e_2 que possuam u e v como vértices finais. Uma *redução em série* de duas arestas em série $e_1 = (u, w)$ e $e_2 = (w, v)$ é a substituição de e_1 e e_2 por uma nova aresta $e = (u, v)$. Uma *redução em paralelo* de duas arestas paralelas $e_1 = (u, v)$ e e_2 é a substituição de e_1 e e_2 por uma nova aresta $e = (u, v)$.

Sejam s e t dois vértices de G . Se G pode ser obtido por uma seqüência de composições em série e em paralelo a partir de uma aresta (s, t) , G é denominado um grafo *série paralelo dois terminal* (SPDT) com respeito a s e t . Da mesma forma, G é um grafo SPDT com respeito a s e t se G pode ser reduzido a uma única aresta (s, t) por uma seqüência de reduções em série e em paralelo. Um grafo G é um grafo *série-paralelo* (SP) se existem dois vértices s e t tal que G é um grafo SPDT com respeito a s e t .

As arestas de um grafo SP podem ser orientadas, bastando para isso orientar inicialmente (s, t) , e quando nas composições, manter a orientação. O grafo orientado obtido é acíclico [44].

Relembramos algumas definições do capítulo anterior.

Seja $H = (V_1, V_2, E)$ um grafo bipartido. Se v é um vértice de um subgrafo H' de H , então $d_{H'}(v)$ denota o grau de v em H' . Consideremos os vértices de V_1 ordenados em ordem não crescente com relação a seus graus (em H) e rotulados $u_1, \dots, u_{|V_1|}$. Seja $\{v_1, \dots, v_{|V_2|}\}$ os vértices de V_2 .

Definimos um *esteio* S em V_1 como uma floresta geradora de H tal que cada $v_i \in V_2$ é incidente em S com exatamente uma aresta de E , e (u_j, v_i) é uma aresta de S implica que (u_k, v_i) não é uma aresta de H , para qualquer $u_k \in V_1$, $k < j$.

Para um *esteio* em V_2 , as regras para os conjuntos V_1 e V_2 , na definição acima são trocadas.

Um vértice $u \in V_1$ é denominado *zero diferença* em S se $d_H(u) - d_S(u) = 0$.

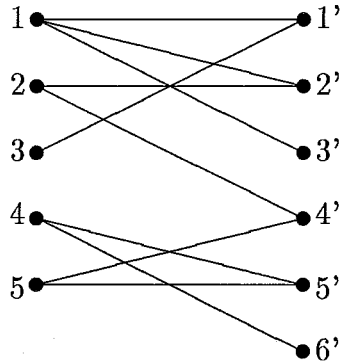
4.3 Algoritmo para Determinação de uma Árvore Geradora

Nesta seção descrevemos um algoritmo para computação de uma árvore geradora de um grafo bipartido que é implementado em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM. Caso o grafo esteja rotulado tal que todo vértice $v_i \in V$ tem um adjacente v_j tal que $j > i$ ($j < i$), o algoritmo é ótimo e pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM. Os grafos *st*-planares e série-paralelo podem ser facilmente rotulados dessa forma em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM. Usamos o exemplo da Figura 4.1 para ilustrar os passos do algoritmo.

4.3.1 Descrição Preliminar do Algoritmo

Algoritmo: Árvore Geradora

1. Computar uma floresta geradora para $H = (V_1, V_2, E)$, obtida com a determinação de um esteio S em H e com a adição de uma aresta arbitrária de $H - S$ a todos vértices não zero diferença de S .
2. Computar um novo grafo bipartido $H_1 = (V'_1, V'_2, E')$ para representar as conexões existentes entre as diferentes árvores da floresta geradora.



$$V_1 = \{1, 2, 3, 4, 5\}, V_2 = \{1', 2', 3', 4', 5', 6'\}$$

$$E = \{(1, 1')(1, 2')(1, 3')(2, 2')(2, 4')(3, 1')(4, 5')(4, 6')(5, 4')(5, 5')\}$$

Figura 4.1: Grafo $H = (V_1, V_2, E)$

3. Aplicar os passos 1 e 2 até que o número de vértices zero diferença seja igual a um.

4.3.2 O Algoritmo e um Exemplo

Nesta seção apresentamos o algoritmo para determinação de uma árvore geradora em detalhes juntamente com um exemplo. Os vetores $EDGE$, $EDGE'$, $BEDGE$ e $SPTREE$ contêm m elementos, cada elemento é uma aresta representada por $(vértice1, vértice2)$.

Vamos agora descrever o algoritmo. A entrada é o grafo bipartido $H = (V_1, V_2, E)$. A lista de arestas E é armazenada em $EDGE$. Para o nosso exemplo o vetor $EDGE$ é

$$(1, 1')(1, 2')(1, 3')(2, 2')(2, 4')(3, 1')(4, 5')(4, 6')(5, 4')(5, 5')$$

Passo 1

Neste passo utilizando um esteio S em V_1 , determinamos uma floresta geradora de H . Para a obtenção dos vértices zero diferença e não zero diferença do esteio S , determinamos os graus de cada um dos vértices em V_1 e armazenamos em D_H . Para o nosso exemplo D_H é

$$\boxed{(3, 2, 1, 2, 2)}$$

Agora determinamos o esteio S em V_1 . Inicialmente efetuamos uma cópia de $EDGE$ em $EDGE'$.

$$EDGE' \leftarrow EDGE$$

Efetuamos uma ordenação em $EDGE'$ da seguinte forma: Dadas duas arestas (i, j) e (k, l) então $(i, j) < (k, l)$ quando $j < l$ ou $((j = l) \text{ e } (i < k))$. O vetor $EDGE'$ fica

$$\boxed{(1, 1')(3, 1')(1, 2')(2, 2')(1, 3')(2, 4')(5, 4')(4, 5')(5, 5')(4, 6')}$$

Agora comprimimos $EDGE'$ a fim de que nenhum vértice de V_2 apareça mais de uma vez em $EDGE'$. O vetor $EDGE'$ representa S e para o nosso exemplo fica

$$\boxed{(1, 1')(1, 2')(1, 3')(2, 4')(4, 5')(4, 6')}$$

Cada uma das sublistas de $EDGE'$ formadas pelas arestas (u, v) , $u = u_i$ forma uma árvore. Rotulamos cada uma dessas árvores por $EDGE'_{u_i}$. Vamos agora determinar quais vértices de V_1 são zero diferença. Para isso, determinamos em paralelo o grau de cada um dos vértices de V_1 em $EDGE'$ e armazenamos em D_S . Para o nosso exemplo

$$\boxed{(3, 1, 0, 2, 0, 0)}$$

Logo, os vértices zero diferença são

$$\boxed{\{1, 4\}}$$

e os não zero diferença são

$$\boxed{\{2, 3, 5\}}$$

Agora adicionamos uma aresta arbitrária pertencente a $EDGE - S$ a cada um dos vértices não zero diferença ao vetor $EDGE'$. Para o nosso exemplo $EDGE'$ fica

$$\boxed{(1, 1')(1, 2')(1, 3')(2, 2')(2, 4')(3, 1')(4, 5')(4, 6')(5, 4')}$$

Essas arestas adicionadas, conectam as diferentes árvores $EDGE'_{u_i}$ da floresta geradora obtida em S em um nova floresta geradora. Para os vértices não zero diferença $u_j \in V_1$, as árvores $EDGE'_{u_j}$ estão agora conectadas a alguma árvore $EDGE'_{u_k}$ onde u_k é um vértice zero diferença.

O Passo 1 pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Passo 2

Neste passo construímos um novo grafo bipartido $H_1 = (V'_1, V'_2, E')$. O grafo H_1 representa as conexões existentes entre as diferentes árvores da floresta geradora representada em $EDGE'$. O grafo H_1 é formado pelas arestas $(u_j, v) \in EDGE - S$ tal que (u_j, v) conecte diferentes árvores $EDGE'_{u_i}$, onde u_i é um vértice zero diferença.

Vamos agora determinar a quais árvores $EDGE'_{u_i}$, u_i um vértice zero diferença, as árvores $EDGE'_{u_j}$, u_j um vértice não zero diferença, estão conectadas. Como demonstraremos posteriormente, a aresta (u_j, v) adicionada após o esteio pode conectar diretamente as árvores, ou fazer parte de um caminho que conecta as árvores.

Para o nosso exemplo, as arestas (u, v) tal que $u = 2, 3$ e 5 estão conectadas à árvore $EDGE'_1$.

Vamos agora determinar a que árvores $EDGE_{u_i}$, u_i um vértice zero diferença, as arestas $e \in EDGE - S$ pertencem. Para o nosso exemplo, as arestas

$$\boxed{(2, 2')(3, 1')(5, 4')}$$

estão conectadas a árvore $EDGE'_1$ e a aresta

$$\boxed{(5, 5')}$$

esta conectada a árvore $EDGE'_4$.

Vamos agora construir o grafo bipartido $H_1 = (V'_1, V'_2, E')$. H_1 é formado pelas arestas que conectam diferentes árvores em $EDGE'$. Logo E' é formado pelas arestas (u_j, v_i) e (u_j, v_k) , u_j um vértice não zero diferença, tal que essas arestas conectem diferentes árvores. Os conjuntos V'_1 e V'_2 são formados pelos vértice pertencentes as arestas descritas acima. A lista das arestas E' é armazenada no vetor $BEDGE$. Para o nosso exemplo $BEDGE$ é

$$\boxed{(5, 4')(5, 5')}$$

O passo 2 pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.

Passo 3

Neste passo aplicamos os passos 1 e 2 até que o número de vértices zero diferença ($numzerodif$) seja igual a 1. Isso pode ser efetuado da seguinte maneira.

repita

$$SPTREE \leftarrow SPTREE + EDGE'$$

$$EDGE \leftarrow BEDGE$$

ate $|numzerodif| = 1$

Após isso, retiramos as arestas duplicadas em $SPTREE$.

Como veremos, o número de vértices zero diferença ao final do Passo 2, é pelo menos dividido por dois a cada iteração. Portanto no Passo 3, o algoritmo executa no máximo $\lceil \log n \rceil$ iterações.

4.4 Grafos Não Orientados

Para grafos não orientados usamos o mesmo algoritmo, efetuando o seguinte passo adicional. Realizamos a subdivisão de cada aresta, e com isso colocamos um vértice adicional em cada aresta. Com isso, o grafo não orientado resultante é um grafo bipartido. Podemos considerar apenas uma das arestas (u, v) ou (v, u) , $v \in V'$, visto que com a subdivisão, todos os vértices adicionados serão considerados no conjunto de vértices V' .

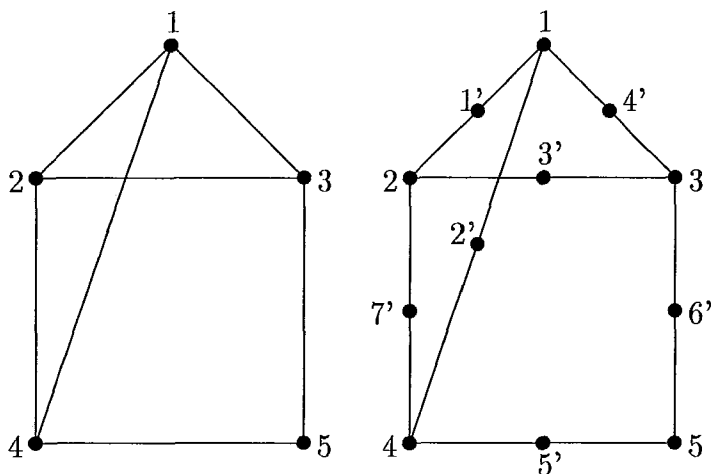


Figura 4.2: Grafo $G = (V, E)$ e $H = (V, V', E')$

Dessa forma, dado um grafo não orientado $G = (V, E)$, substituímos as arestas $(i, j) \in E$ por duas arestas (i, k) e (k, j) , $k \in V'$. O grafo obtido $H = (V, V', E')$ é um grafo bipartido. Logo podemos aplicar o algoritmo da seção 2 para obter uma árvore geradora para H .

Ilustramos os passos principais do algoritmo com relação ao grafo da Figura 4.2. O vetor $EDGE'$ é

$$\boxed{(1, 1')(1, 2')(1, 4')(2, 1')(2, 3')(2, 7')(3, 3')(3, 4')(3, 6')(4, 2')(4, 5')(4, 7')(5, 5')(5, 6')}$$

Aplicamos o algoritmo em H e obtemos uma árvore geradora. Ao final do algoritmo *SPTREE* armazena a árvore geradora de H . Vamos agora computar a árvore geradora de $G = (V, E)$. Isso pode ser feito através da computação do grau de cada um dos vértices $v_i \in V'$. Seja *SPTREE'* as arestas de *SPTREE* tal que $\text{grau}(v_i) = 2$. Temos que *SPTREE'* é uma árvore geradora de G . Para o nosso exemplo, temos que *SPTREE* é

$$\boxed{(1, 1')(1, 2')(1, 4')(2, 1')(2, 3')(2, 7')(3, 3')(3, 6')(4, 2')(4, 5')(5, 5')}$$

e *SPTREE'* é

$$\boxed{(1, 2)(1, 4)(2, 3)(4, 5)}$$

4.5 Componentes Conexos

Nesta seção descreveremos brevemente como o algoritmo para a computação de uma árvore geradora pode ser aplicado para determinar os componentes conexos de um grafo e os componentes fracamente conexos de um grafo orientado. Estes problemas podem ser resolvidos em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores e em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores quando os vértices do grafo estiverem convenientemente rotulados.

O algoritmo para determinação de uma árvore geradora conecta a cada iteração as árvores $EDGE'_{u_i}$. Ao final do algoritmo, representamos cada árvore com o menor vértice. Cada um dos diferentes vértices representa um componente conexo do grafo.

Os componentes fracamente conexos do grafo orientado podem ser obtidos com a desconsideração das orientações das arestas e com a remoção das arestas duplicadas.

4.6 Uma Simplificação do Algoritmo

Da mesma forma que no algoritmo de Circuito de Euler, o algoritmo para a computação de uma árvore geradora pode ser simplificado com o intuito de evitar a utilização de ordenação.

Uma outra abordagem para determinação de uma árvore geradora do grafo pode ser efetuada através da construção da floresta geradora formada por um elemento de cada uma das listas de adjacência de tal forma que em cada aresta (u, v) escolhida, $u < v$, ou de forma análoga $u > v$. Caso o grafo esteja numerado de acordo com o lema abaixo, a floresta será composta de apenas uma árvore. No caso geral, basta verificar as arestas que conectam as diferentes árvores da floresta geradora e aplicar o algoritmo para essas arestas. Em no máximo $\lceil \log n \rceil$ iterações a floresta geradora torna-se uma única árvore.

4.7 Correção e Análise

Lema 4.7.1 *Seja A o conjunto das arestas que foram adicionadas aos vértices não zero diferença u_j , $EDGE' = S \cup A$ e $EDGE'_{u_i}$ a sublista das arestas $(u, v) \in EDGE'$ tal que $u = u_i$. Então a aresta $e \in A$ unida a $EDGE'_{u_j}$ conecta a árvore*

$EDGE'_{u_j}$ com a árvore $EDGE_{u_i}$, u_i um vértice zero diferença ou existe um caminho $(u_j, v_k), \dots, (v_l, u_i)$ ligando $EDGE'_{u_j}$ a $EDGE_{u_i}$.

Demonstração: Utilização de duplicação recursiva. \square

Lema 4.7.2 *Seja $H = (V_1, V_2, E)$ um grafo bipartido conexo e S um esteio em V_1 . Seja $H' = (V_1, V_2, E_1)$ o grafo obtido de H adicionando precisamente a S uma aresta de $E - S$ incidente a cada vértice não zero diferença de V_1 . Então H' é acíclico. Além disso, se V_1 contém exatamente k vértices zero diferença então H' é uma floresta geradora de H com k árvores.*

Demonstração: S é essencialmente uma coleção de estrelas cujos centros são vértices em V_1 . Adicionando precisamente uma aresta de $E - S$ incidente a cada vértice não zero diferença de V_1 , cada estrela cujo centro é um vértice não zero diferença v_j será conectada com no máximo uma estrela distinta cujo centro é v_i . Então H' é acíclico.

Utilizando o lema acima, todas estrelas cujo centro é um vértice não zero diferença estarão conectadas por uma aresta ou por um caminho a uma estrela cujo centro é um vértice zero diferença. Logo para cada um dos k vértices zero diferença teremos uma árvore. \square

Teorema 4.7.1 *Seja $H_1 = (V'_1, V'_2, E')$ um grafo bipartido representando as conexões existentes entre as diferentes árvores da floresta geradora obtida ao final do passo 2. O número de vértices zero diferença de H_1 é no máximo $\lceil \frac{|V'_2|}{2} \rceil$.*

Demonstração: Temos que o esteio seleciona uma única aresta saindo de $v_i \in V'_2$, esta aresta conecta v_i a um vértice $u_j \in V'_1$, tal que o grau de u_j é o maior entre os vértices que estão conectados a v_i . Logo o grau de um vértice $u_j \in V'_1$ zero diferença é pelo menos dois, e o número de vértices zero diferença é no máximo $\lceil \frac{|V'_1|}{2} \rceil$. \square

Portanto, no Passo 3, o algoritmo itera no máximo $O(\log n)$ vezes no pior caso.

Teorema 4.7.2 *O algoritmo determina uma árvore geradora para um grafo em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Vamos agora demonstrar os resultados que são obtidos caso o grafo esteja convenientemente rotulado.

Lema 4.7.3 *Seja $G = (V, E)$ um grafo tal que os vértices de G estão rotulados de acordo com uma das duas condições: (i) para todo $1 < v_i \leq n$, v_i tem um adjacente v_j tal que $j < i$; (ii) para todo $1 \leq v_i < n$, v_i tem um adjacente v_j tal que $j > i$. Então o grafo bipartido H possui apenas um vértice zero diferença.*

Demonstração: Vamos considerar o caso (i), o (ii) pode ser demonstrado de forma análoga. Na computação de um esteio para o grafo bipartido obtido de G , para cada um dos vértices obtidos com a subdivisão, selecionamos as arestas incidentes com os vértices de maior rótulo. Logo as arestas que saem do vértice de maior rótulo v_k são selecionadas. Em função de que para todo vértice $v_i \neq v_k$ tem pelo menos um vértice adjacente v_j , $j > i$, pelo menos uma aresta saindo de v_i que não será selecionada. Logo, apenas o vértice v_k será um vértice zero diferença. \square

Portando, nesse caso em uma única iteração o algoritmo computa uma árvore geradora.

Teorema 4.7.3 *Se os vértices do grafo G estiverem rotulados de acordo com o lema anterior, o algoritmo computa uma árvore geradora em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Teorema 4.7.4 *Seja $G = (V, E)$ um grafo e uma orientação acíclica de G . Se o digrafo resultante da orientação contiver exatamente uma fonte ou exatamente um sumidouro então o algoritmo computa uma árvore geradora de G em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Demonstração: Vamos considerar o caso de obtermos uma orientação com apenas uma fonte. Podemos efetuar uma ordenação dos vértices de acordo com a orientação e rotular os vértices de acordo com a ordenação resultante. Logo para todo vértice $1 < v_i \leq n$, v_i tem um adjacente v_j tal que $j < i$. Isso faz com que o algoritmo computa uma árvore geradora de G em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$. \square

Teorema 4.7.5 *Seja $G = (V, E)$ um grafo st -planar, então o algoritmo computa uma árvore geradora para G em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Teorema 4.7.6 *Seja $G = (V, E)$ um grafo série paralelo, então o algoritmo computa uma árvore geradora para G em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM.*

Capítulo 5

Busca em Profundidade Irrestrita

5.1 Introdução

Os problemas de *busca em profundidade* e *busca em largura* em um grafo $G = (V, E)$ a partir de um vértice $v \in V$ determinam uma árvore que corresponde a execução da busca correspondente. Nesses casos, cada aresta é percorrida um número constante de vezes. Utilizamos o termo *busca restrita* para identificar uma busca com essa característica.

Definimos uma *busca irrestrita* [78] em um grafo $G = (V, E)$ a partir de um vértice $v \in V$ um processo sistemático de se percorrer G de tal modo que cada aresta seja visitada um número finito (qualquer) de vezes. Note que segundo a definição, a única restrição ao processo é que ele deve terminar.

O algoritmo seqüencial para busca irrestrita é baseado no algoritmo [82] de busca em profundidade (restrita). O problema de busca em profundidade paralela não parece ter uma solução simples como no caso seqüencial, e até o momento, só foram obtidas soluções *NC* para alguns casos particulares de grafos [39,90]. No caso geral, foi provado que a busca lexicográfica em profundidade é um problema *P-completo* [50].

Para desenvolver um algoritmo paralelo para busca irrestrita, temos que evitar os problemas que ocorrem quando o caminho encontra um ciclo no grafo. Além disso, precisamos saber em que parte do grafo o caminho se encontra a cada passo do algoritmo. Para efetuar isso, utilizamos uma decomposição do grafo denominada *kl-caminho*.

A busca irrestrita em um grafo gera todos os caminhos maximais a

partir do vértice r , raiz da busca. O problema do caminho maximal é: Dado um grafo $G = (V, E)$ e um vértice $r \in V$, determinar um caminho simples C iniciando em r tal que C não possa ser estendido sem encontrar um vértice que já pertence a C .

O problema do caminho maximal está relacionado ao de busca em profundidade, pois qualquer *ramo* da árvore de busca em profundidade é um caminho maximal. O algoritmo para computação de um caminho maximal usando o método guloso não possui uma paralelização eficiente [4]. Um algoritmo *RNC* de tempo paralelo $O(\log^5 n)$ com $n^2 M$ processadores, onde M é o número de processadores para contruir um emparelhamento máximo, foi apresentado por Anderson [5].

Vamos apresentar algoritmos para busca irrestrita para algumas classes de grafos. Dentre essas classes, a dos digrafos acíclicos possui uma versão simplificada do algoritmo, pois não precisamos nos preocupar com a extensão dos caminhos maximais entre os diferentes ciclos. Para digrafos acíclicos, o algoritmo é executado em tempo paralelo $O(\alpha \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz. O algoritmo é implementado para grafos gerais em tempo paralelo $O(\alpha \beta \log^2 n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz e β é o número máximo de ciclos do grafo que pode ocorrer na computação de um caminho maximal.

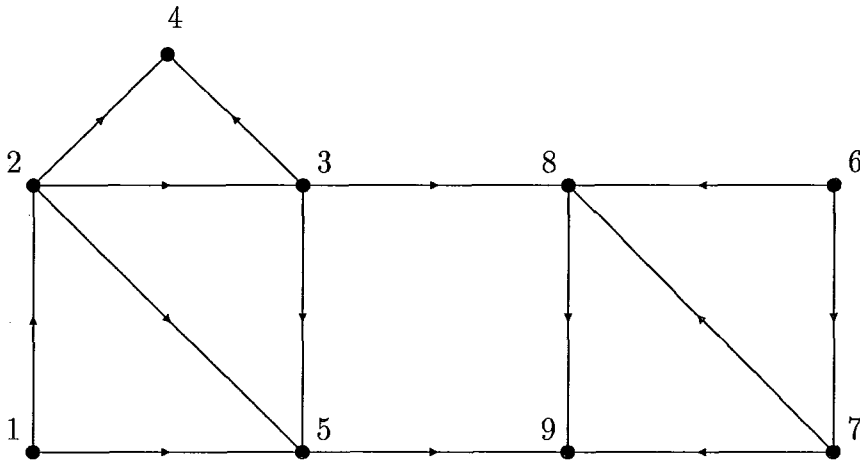
5.2 Notação e Terminologia

Dado um digrafo $G = (V, E)$, para cada vértice $v \in V$, definimos o *sucessor* de v ($suc[v]$) como sendo um elemento fixo da lista de adjacências de v . Seja $e = (u, v) \in E$, definimos o *sucessor* de e como sendo a aresta $(v, suc[v])$. Um *kl-caminho* é um caminho C em G com aresta inicial $e = (k, l)$. Uma aresta $e \in E$ pertence ao *kl-caminho* C se e é sucessor de alguma aresta do caminho C , e $e \notin C$. Uma aresta (u, v) pode pertencer a mais de um *kl-caminho*.

Para um grafo G não orientado, consideramos cada aresta (u, v) , como duas arestas orientadas distintas (u, v) e (v, u) .

Pelo fato do sucessor de cada vértice ser fixo, todas as arestas em um *kl-caminho* incidentes a um vértice v possuem um mesmo sucessor $(v, suc[v])$. Um *kl-caminho* pode ser um caminho simples, um ciclo, um caminho unido a um ciclo ou a união de dois ciclos.

No caso em que $G = (V, E)$ é um digrafo acíclico, os *kl-caminhos* são formados por caminhos simples, com um vértice inicial k e um vértice final t .

Figura 5.1: Grafo $G = (V, E)$

5.3 Algoritmo para Busca Irrestrita em Digrafos Acíclicos

Nesta seção apresentamos um algoritmo para busca em profundidade irrestrita para digrafos acíclicos. O algoritmo computa todos os caminhos simples maximais a partir de uma dada raiz r . O algoritmo é executado em tempo paralelo $O(\alpha \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos maximais existentes na busca. O exemplo da Figura 5.1 é utilizada para ilustrar as idéias.

O algoritmo inicialmente decompõe o digrafo $G = (V, E)$ em um conjunto de kl -caminhos. Essa decomposição é obtida através da definição dos sucessores para os vértices e arestas de G . O algoritmo inicializa o sucessor de cada um dos vértices $v \in V$ com o primeiro elemento da lista de adjacência de v . Após a decomposição o algoritmo explora as arestas do kl -caminho tal que $k = r$, onde r é a raiz da busca e $l = \text{suc}[r]$. As arestas visitadas formam um caminho simples maximal C . Uma vez determinado um caminho simples maximal $C = \{v_0, \dots, v_p\}$, para $r = v_0$ no kl -caminho, podemos obter (caso exista) um novo caminho simples maximal. Para isso, determinamos o último vértice $v_i \in C$, que possua algum vértice em sua lista de adjacência que ainda não tenha sido visitado. Os sucessores dos vértices $v_j \notin \{v_0, \dots, v_i\}$ são modificados para o primeiro elemento da lista de adjacência de cada v_j , e o sucessor de v_i é alterado para o elemento da lista de adjacências de v_i imediatamente posterior a $\text{suc}[v_i]$. Os sucessores dos vértices $\{v_0, \dots, v_{i-1}\}$ permanecem inalterados. Essa nova definição de sucessores determina uma nova decomposição do digrafo em um conjunto de kl -caminhos. Determinamos um caminho simples $C = \{v_i, \dots, v_t\}$ em um kl -caminho na nova decomposição, com $k = v_i$ e $l = \text{suc}[v_i]$. O caminho $C = \{v_0, \dots, v_i, \dots, v_t\}$ formado pela união dos caminhos $\{v_0, \dots, v_i\}$ e $\{v_i, \dots, v_t\}$ é um novo caminho simples maximal. Os demais

caminhos simples maximais (caso existam) são computados de maneira análoga.

Todos os vértices w_i que não são alcançáveis a partir do vértice raiz r não são incluídos na busca.

5.3.1 Descrição Preliminar do Algoritmo

Algoritmo: Busca Irrestrita

1. Definir uma raiz r , contruir as listas de adjacências de G e inicializar o caminho simples maximal CM .
2. Decompor o digrafo G em um conjunto de kl -caminhos.
3. Determinar um caminho simples C a partir de r no kl -caminho, com $k = r$ e $l = suc[r]$.
4. Computar o caminho simples maximal $CM = CM \cup C$. Verificar a existência de algum vértice $v_i \in CM$ que possua na sua lista de adjacências um vértice que ainda não tenha sido visitado.
5. Caso v_i exista, $CM = \{v_0, \dots, v_{i-1}\}$. Alterar os sucessores dos vértices $v_i \notin CM$, e desmarcar as arestas $e \in G$ que não pertencem ao caminho CM .
6. Aplicar os passos 2, 3, 4 e 5 ao conjunto de arestas desmarcadas de G para $r = v_i$ até que todos os possíveis caminhos simples maximais a partir de v_0 possam ser explorados.

5.3.2 O Algoritmo e um Exemplo

Nesta seção, discutimos o algoritmo paralelo para determinar uma busca irrestrita, em detalhe, juntamente com um exemplo. Os vetores $EDGE$, $BEDGE$ e SUC contêm m elementos, cada elemento sendo uma *aresta* representada por $e = (vértice1, vértice2)$. O vetor ADJ é um vetor de vetores, sendo que $ADJ[v_i][k]$ representa o k -ésimo elemento da lista de adjacências do vértice v_i , e a posição de cada aresta $SUC[e]$ em $EDGE$ é dada pelo vetor POS . O vetor $AEDGE$ contém m elementos, cada $AEDGE[e]$ é do tipo $(grau[v_i]-1, rank(v_i, v_j)-1)$ onde $rank(v_i, v_j)$ indica a posição do vértice v_j na lista de adjacências de v_i . O vetor $INFO$ contém m elementos, cada $INFO[e]$ é do tipo $(info1, info2)$, onde $info1$ informa em que passo a aresta e foi selecionada, e $info2$ informa a posição de e no caminho C . Os vetores $C1$ e CM , contêm n elementos, são utilizados para armazenar cada caminho

que está sendo computado. O vetor AC contém n elementos. Os vetores K e D são vetores de inteiros utilizados para armazenar valores temporários.

Agora descreveremos o algoritmo. A entrada é o digrafo acíclico $G = (V, E)$ da figura 5.1. Consideramos a lista de arestas E ordenada, tal que duas arestas (i, j) e (k, l) , $(i, j) < (k, l)$ se $(i < k)$ ou $((i = k) \text{ e } (j < l))$. A lista de arestas E é armazenada em $EDGE$. Para nosso exemplo, o vetor $EDGE$ é

(1,2)(1,5)(2,3)(2,4)(2,5)(3,4)(3,5)(3,8)(5,9)(6,7)(6,8)(7,8)(7,9)(8,9)

Passo 1

Neste passo, definimos uma raiz r , inicializamos o caminho simples maximal, contruimos as listas de adjacências de cada um dos vértices $v_i \in V$ e inicializamos uma estrutura auxiliar para computação dos caminhos C e do caminho simples maximal.

Primeiramente copiamos o vetor $EDGE$ em $BEDGE$, e particionamos $BEDGE$ em um conjunto de sublistas, uma para cada $v_i \in V$. Após a partição, computamos o posto ($rank(v_i, v_j)$) para cada uma dessas sublistas. Isso nos informa que o vértice v_j é o k -ésimo elemento da lista de adjacências de v_i , $k = rank(v_i, v_j)$. O número de elementos de cada uma das sublistas v_i é armazenado em $grau(v_i)$. Na construção de ADJ , que representa as listas de adjacências de cada v_i , consideramos o primeiro elemento de cada lista de adjacências como sendo de índice zero (0), e acrescentamos o vértice 0 após o último elemento em cada uma das listas para denotar o final da lista.

Além disso, computamos o vetor $AEDGE$ para auxiliar na obtenção do vértice no caminho simples maximal que é o mais distante da raiz da busca r e que possui pelo menos um vértice na lista de adjacências que ainda não foi explorado e inicializamos os vetores $INFO$, K , $C1$ e CM .

O grau de cada vértice, bem como a divisão em sublistas e a computação do posto podem ser efetuados utilizando duplicação recursiva na lista de arestas $EDGE$ em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

$r \leftarrow$ raiz da busca

$m \leftarrow 1$

$BEDGE \leftarrow EDGE$

para todo $e = (v_i, v_j) \in BEDGE$ em paralelo faça

$ADJ[v_i][rank(v_i, v_j) - 1] \leftarrow v_j$
 $ADJ[v_i][grau(v_i)] \leftarrow 0$
 $AEDGE[e] \leftarrow (grau[v_i] - 1, rank(v_i, v_j) - 1)$
 $INFO[e] \leftarrow (*, *)$
para todo $v_i \in V$ **em paralelo faça**
 $K[v_i] \leftarrow 0$
para todo $0 \leq t \leq n$ **em paralelo faça**
 $C1[t] \leftarrow 0$
 $CM[t] \leftarrow 0$

A inicialização do vetor *INFO* desmarca todas as arestas $e \in BEDGE$ e a do vetor *K*, que informa qual vértice está sendo utilizado em cada uma das listas de adjacência.

O passo 1 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 2

Neste passo determinamos uma decomposição do digrafo em um conjunto de kl -caminhos. Para isso computamos o vetor *SUC* que armazena os sucessores de cada $e = (v_i, v_j) \in BEDGE$. Utilizamos o vetor *POS* para armazenar a posição de $SUC[e]$ em *BEDGE* e que será utilizado na obtenção de um caminho C no kl -caminho, $k = r$.

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**
 $SUC[e] \leftarrow (v_i, ADJ[v_i][K[v_i]])$
 $POS[e] \leftarrow$ posição de $(v_i, ADJ[v_i][K[v_i]])$ em *BEDGE*

Para nosso exemplo, os vetores *BEDGE* e *SUC* são

(1,2)(1,5)(2,3)(2,4)(2,5)(3,4)(3,5)(3,8)(5,9)(6,7)(6,8)(7,8)(7,9)(8,9)
(2,3)(5,9)(3,4)(0,0)(5,9)(0,0)(5,9)(8,9)(0,0)(7,8)(8,9)(8,9)(0,0)(0,0)

Os vetores *BEDGE* e *SUC* juntos definem um conjunto de kl -caminhos no digrafo G . Em qualquer kl -caminho, a aresta seguinte à aresta $e = (v_i, v_j) \in BEDGE$ está em $SUC[e]$. Cabe lembrar que uma aresta pode pertencer a kl -caminhos distintos.

O passo 2 pode ser executado em tempo paralelo $O(1)$ como m processadores em uma CREW PRAM.

Passo 3

Neste passo, determinamos um caminho simples C no kl -caminho, onde $k = r$ e $l = \text{suc}[r]$. Isto pode ser efetuado utilizando duplicação recursiva no kl -caminho a partir da aresta $(r, \text{ADJ}[r][K[r]])$. Na iteração 0, selecionamos a aresta fixada na adjacência inicial a partir do vértice $v_i = r$.

$itera \leftarrow 0$
 $e \leftarrow (r, \text{ADJ}[r][K[r]])$
 $\text{INFO}[e] \leftarrow (itera, m)$

Após a seleção de uma aresta $e = (v_i, v_j)$ ($\text{INFO}[e] \neq (*, *)$), e é considerada *marcada*. Uma vez selecionada a primeira aresta, vamos computar um caminho C no kl -caminho, $k = r$. A cada iteração, com a utilização de duplicação recursiva, selecionamos as arestas que são sucessoras das arestas marcadas do kl -caminho na iteração anterior. Isso pode ser efetuado da seguinte forma

repita

$itera \leftarrow itera + 1$
para todo $e = (v_i, v_j) \in \text{BEDGE}$ **em paralelo faça**
 para $\text{INFO}[e] = (s, t)$ **faça**
 se $\text{INFO}[e] \neq (*, *)$ **e** $t \geq m$ **então**
 $\text{INFO}[\text{SUC}[e]] \leftarrow (itera, 2^{itera-1} + t)$
para todo $e = (v_i, v_j) \in \text{BEDGE}$ **em paralelo faça**
 $\text{SUC}[e] \leftarrow \text{SUC}[\text{BEDGE}[\text{POS}[e]]]$
 $\text{POS}[e] \leftarrow \text{POS}[\text{BEDGE}[\text{POS}[e]]]$

O caminho C pode ser computado em $\lceil \log n \rceil$ iterações, cada iteração é executada em tempo paralelo $O(1)$ com m processadores em uma CREW PRAM.

Para nosso exemplo, os vetores BEDGE e INFO são

$(1,2)(1,5)(2,3)(2,4)(2,5)(3,4)(3,5)(3,8)(5,9)(6,7)(6,8)(7,8)(7,9)(8,9)$
$(0,0)(*,*)(1,1)(*,*)(*,*)(2,2)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)$

Vamos agora armazenar os vértices do caminho simples que foram encontrados no caminho C . Utilizamos para isso o vetor $C1$. Armazenamos também as informações referentes as arestas que foram percorridas na determinação do caminho C no vetor AC .

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**
para $INFO[e] = (s, t)$ **faça**
 se $INFO[e] \neq (*, *)$ e $t \geq m$ **então**
 $C1[t] \leftarrow v_i$
 $AC[t] \leftarrow AEDGE[e]$

O caminho simples no 12-caminho é armazenado no vetor C , e é formado pelos vértices

1,2,3,4

O passo 3 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 4

Neste passo vamos computar um caminho simples maximal e determinar se esse caminho possui algum vértice com alguma adjacência que ainda não tenha sido explorada.

Para obter um caminho simples maximal, adicionamos o caminho $C1$ ao vetor CM . O índice m indica a posição a partir da qual o caminho $C1$ é adicionado em CM .

para todo $m \leq t \leq n$ **em paralelo faça**
 $CM[t] \leftarrow C1[t]$
 $ACM[t] \leftarrow AC[t]$

Vamos agora determinar se algum vértice do caminho simples maximal CM possui alguma adjacência que ainda não tenha sido explorada. Para isso, utilizamos as informações das arestas que pertencem ao caminho CM .

para todo $i, 1 \leq i \leq n$ **em paralelo faça**
para $ACM[i] = (s, t)$ **faça**
 $D[i] \leftarrow s - t$
 $max \leftarrow \max_{1 \leq i \leq n} \{i, D[i] \neq 0\}$

O máximo de um conjunto pode ser facilmente computado com a utilização de uma árvore binária, em tempo paralelo $O(\log n)$ com n processadores em uma CREW PRAM.

Caso $max \neq 0$, o caminho CM possui pelo menos um vértice $v_i = CM[max]$, tal que o caminho $\{v_0, \dots, v_i\}$ pode ser estendido a um novo caminho simples maximal a partir de v_i , e visitando um vértice da lista de adjacência de v_i que ainda não tenha sido explorado.

O passo 4 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 5

Se $max \neq 0$, temos que alterar os sucessores dos vértices $v \in V$, para determinar um novo caminho simples a partir do vértice $v_i = CM[max]$. Além disso temos que atualizar o vetor CM , que juntamente com o vetor $C1$, a ser determinado, formarão o novo caminho maximal.

$m \leftarrow max$
para $ACM[m] = (s, t)$ **faça**
 $K[CM[m]] \leftarrow t + 1$
 $B[CM[m]] \leftarrow 1$
para todo $v_j \notin \{CM[0], \dots, CM[m]\}$ **em paralelo faça**
 $K[v_j] \leftarrow 0$
para todo $i, m \leq i \leq n$ **em paralelo faça**
 $ACM[i] \leftarrow (*, *)$
 $CM[i] \leftarrow 0$

Vamos também desmarcar as arestas que não fazem parte do caminho simples maximal $CM = \{CM[1], \dots, CM[m]\}$ e reinicializar os vetores $C1$ e AC .

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**
para $INFO[e] = (s, t)$ **faça**

se $t \geq m$ **então**
 $INFO[e] \leftarrow (*, *)$
para todo $i, 1 \leq i \leq n$ **em paralelo faça**
 $C1[i] \leftarrow 0$
 $AC[i] \leftarrow (*, *)$

O passo 5 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 6

Usando as arestas desmarcadas de *BEDGE*, definimos uma nova raiz $r = v_i$ e aplicamos os passos 2, 3, 4 e 5 até que todos os possíveis caminhos maximais em G , com raiz $v = CM[1]$, sejam determinados.

Cada caminho simples maximal C pode ser computado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM. O número de caminhos simples maximais a partir de uma raiz r em um digrafo acíclico pode ser exponencial. A complexidade total do algoritmo é de tempo paralelo $O(\alpha \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos a partir da raiz até um vértice sumidouro no digrafo acíclico.

5.4 Algoritmo para Busca Irrestrita

Nesta seção, apresentamos um algoritmo para busca em profundidade irrestrita para digrafos. O algoritmo computa todos os caminhos simples maximais de um digrafo, e é executado em tempo paralelo $O(\alpha\beta \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos simples maximais, e β é o número máximo de ciclos do digrafo que podem ocorrer na computação de um caminho simples maximal.

O algoritmo utiliza a mesma idéia do algoritmo para digrafos acíclicos. O digrafo é decomposto em um conjunto de kl -caminhos e um caminho C é determinado no kl -caminho, onde $k = r$, e $l = suc[r]$. Em função dos ciclos do digrafo, necessitamos verificar quando ocorre um ciclo no caminho C . Caso ocorra um ciclo no caminho C , computamos um caminho simples $C' \subset C$. Uma vez computado o caminho simples C' no kl -caminho, verificamos se o caminho simples C' é maximal.

Caso o caminho C' não seja maximal, podemos estendê-lo até obter

um caminho simples maximal. Para isso, utilizamos a mesma abordagem da obtenção de novos caminhos maximais descrito no algoritmo para digrafos acíclicos, ou seja, alterando os sucessores dos vértices que não pertencem ao caminho simples C' . Uma vez obtido um caminho simples maximal, determinamos um novo caminho simples maximal (caso exista) de maneira análoga.

Da mesma forma que no digrafo acíclico, todos os vértices w_i que não são alcançáveis a partir do vértice raiz r não são incluídos na busca.

5.4.1 Descrição Preliminar do Algoritmo

Algoritmo: Busca Irrestrita

1. Definir uma raiz r , contruir as listas de adjacências de G e inicializar o caminho simples maximal CM .
2. Decompor o digrafo G em um conjunto de kl -caminhos.
3. Determinar um caminho simples C a partir de r no kl -caminho, com $k = r$ e $l = suc[r]$.
4. Computar o caminho simples maximal $CM = CM \cup C$. Verificar a existência de algum vértice $v_i \in CM$ que possua na sua lista de adjacências um vértice que ainda não tenha sido visitado.
5. Caso v_i exista, $CM = \{v_0, \dots, v_{i-1}\}$. Alterar os sucessores dos vértices $v_i \notin CM$, e desmarcar as arestas $e \in G$ que não pertencem ao caminho CM .
6. Aplicar os passos 2, 3, 4 e 5 ao conjunto de arestas desmarcadas de G para $r = v_i$ até que todos os possíveis caminhos simples maximais a partir de v_0 possam ser explorados.

5.4.2 O Algoritmo e um Exemplo

Nesta seção, discutimos o algoritmo para determinar uma busca paralela irrestrita em um digrafo, em detalhe, juntamente com um exemplo. Utilizaremos a mesma estrutura de dados que a usada na busca paralela irrestrita em digrafos acíclicos adicionada do vetor B , com n elementos. O vetor B armazenará o número de vezes que cada vértice v aparece no caminho.

Agora descreveremos o algoritmo. A entrada é o digrafo $G = (V, E)$ da Figura 5.2. Consideramos a lista de arestas E ordenada, tal que duas arestas

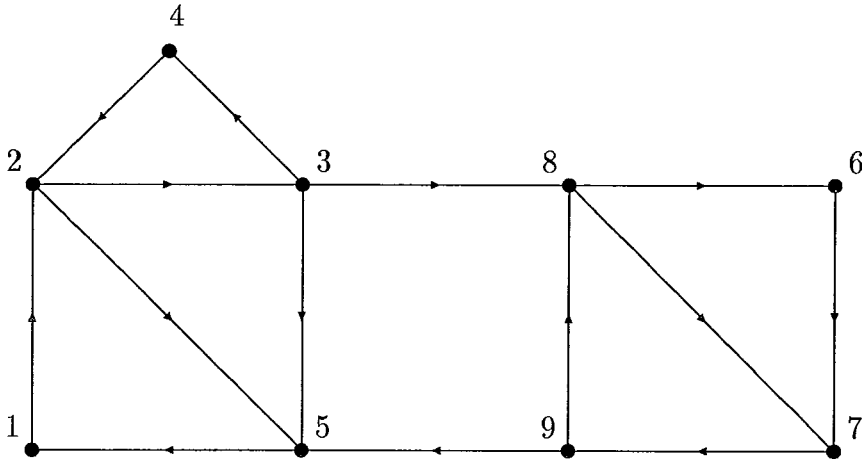


Figura 5.2: Grafo $G = (V, E)$

(i, j) e (k, l) , $(i, j) < (k, l)$ se $(i < k)$ ou $((i = k) \wedge (j < l))$. A lista de arestas E é armazenada em $EDGE$. Para nosso exemplo, o vetor $EDGE$ é

$(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)$

Passo 1

Nesse passo, efetuamos as mesmas operações que no passo 1 do algoritmo para digrafos acíclicos, adicionada da inicialização do vetor B . O vetor B contém n elementos, cada $B[v_i]$ indica o número de vezes que o vértice v_i aparece no caminho C .

para todo $v_i \in V$ em paralelo faça
 $B[v_i] \leftarrow 0$

O passo 1 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 2

O passo 2, tem as mesmas operações do passo 2 do algoritmo para digrafos acíclicos.

Para nosso exemplo, os vetores *BEDGE* e *SUC* são

(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)
(2,3)(3,4)(5,1)(4,2)(5,1)(8,6)(2,3)(1,2)(7,9)(9,5)(6,7)(7,9)(5,1)(8,6)

O passo 2 pode ser executado em tempo paralelo $O(1)$ com m processadores em uma CREW PRAM.

Passo 3

Neste passo, determinamos um caminho simples C no kl -caminho, onde $k = r$, e $l = \text{suc}[r]$. O caminho que obtemos aplicando duplicação recursiva no kl -caminho não é necessariamente simples, pois o digrafo pode ter ciclos. Portanto, necessitamos verificar quando um ciclo é encontrado no kl -caminho que está sendo explorado. Para efetuar isso utilizamos o vetor B .

```

itera ← 0
e ← (r, ADJ[r][K[r]])
INFO[e] ← (itera, 1)
B[r] ← B[r] + 1

```

No algoritmo para digrafos acíclicos, efetuamos a seleção das arestas em $\lceil \log n \rceil$ iterações. No caso de digrafos gerais, necessitamos verificar também se a seleção de arestas a cada iteração não acarretou nenhum ciclo. Isto pode ser feito com a marcação dos vértices que são visitados a cada iteração.

```

enquanto  $B[v_i] < 2, v_i \in V$  faça
  itera ← itera + 1
  para todo  $e = (v_i, v_j) \in \text{BEDGE}$  em paralelo faça
    para  $\text{INFO}[e] = (s, t)$  faça
      se  $\text{INFO}[e] \neq (*, *)$  e  $t \geq m$  então
        se  $\text{INFO}[\text{SUC}[e]] = (*, *)$  então
           $\text{INFO}[\text{SUC}[e]] \leftarrow (\text{itera}, 2^{\text{itera}-1} + t)$ 
        se  $s = \text{itera}$  então
           $B[v_i] \leftarrow B[v_i] + 1$ 
    para todo  $e = (v_i, v_j) \in \text{BEDGE}$  em paralelo faça
       $\text{SUC}[e] \leftarrow \text{SUC}[\text{BEDGE}[\text{POS}[e]]]$ 
       $\text{POS}[e] \leftarrow \text{POS}[\text{BEDGE}[\text{POS}[e]]]$ 

```

Na computação de cada um dos $B[v_i]$, pode ocorrer que o vértice v_i seja visitado por mais de uma aresta em uma mesma iteração. Como não estamos trabalhando com escrita concorrente, essa operação pode ser efetuada em tempo paralelo $O(\log n)$ com n processadores em uma CREW PRAM.

Temos que caso G tenha ciclos, um ciclo é determinado em G em pelo menos $\lceil \log n \rceil$ iterações. Portanto, o número de iterações é no máximo $\lceil \log n \rceil$. Em função de necessitarmos tempo paralelo $O(\log n)$ para computar o vetor B , a computação de um caminho no digrafo G é de tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM.

Para nosso exemplo, os vetores $BEDGE$, $INFO$ e B são

(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)
(0,1)(1,2)(*,*)(2,3)(*,*)(*,*)(2,4)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)
1,2,1,1,0,0,0,0,0

Caso ocorra um ciclo, (existe pelo menos um v_i tal que $B[v_i] \geq 2$), necessitamos retirar as arestas do caminho C no kl -caminho que ocorrem após a determinação do ciclo. Podemos fazer isso da seguinte maneira

```

para todo  $e = (v_i, v_j) \in BEDGE$  em paralelo faça
  para  $INFO[e] = (s, t)$  faça
    se  $s = itera$  e  $B[v_i] > 1$  então
       $s \leftarrow itera + 1$ 
 $min \leftarrow \min_{1 \leq t \leq m} \{INFO[e] = (s, t), s = itera + 1\}$ 
  para todo  $e \in BEDGE$  em paralelo faça
    para  $INFO[e] = (s, t)$  faça
      se  $t > min$  então
         $INFO[e] = (*, *)$ 

```

O mínimo de um conjunto pode ser computado com a utilização de uma árvore binária em tempo paralelo $O(\log n)$ com n processadores em uma CREW PRAM.

Vamos agora armazenar os vértices do caminho simples que foi encontrado no caminho C . Utilizamos para isso o vetor $C1$. Armazenamos também as informações referentes as arestas que foram percorridas na determinação do caminho C no vetor AC .

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**
para $INFO[e] = (s, t)$ **faça**
se $INFO[e] \neq (*, *)$ **e** $t \geq m$ **então**
 $C1[t] \leftarrow v_i$
 $AC[t] \leftarrow AEDGE[e]$

O passo 3 pode ser executado em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM.

Passo 4

Neste passo, vamos computar um caminho simples maximal. No caso dos digrafos acíclicos, todo caminho simples $C1$ adicionado a CM dava um novo caminho simples maximal. Pelo fato de podermos ter encontrado um ciclo, o caminho simples obtido na adição de $C1$ a CM pode não ser um caminho simples maximal. Uma vez determinado se $C1 \cup CM$ é um caminho maximal, a abordagem para a obtenção de um novo caminho simples maximal ou para estender o caminho simples existente em $CM \cup C1$ a um caminho simples maximal, é idêntica.

Vamos agora adicionar o caminho contido no vetor $C1$ ao caminho contido no vetor CM . O vetor CM armazenará o caminho simples a partir da raiz inicial da busca.

para todo $m \leq t \leq n$ **em paralelo faça**
 $CM[t] \leftarrow C1[t]$
 $ACM[t] \leftarrow AC[t]$

Vamos agora verificar a existência de alguma adjacência que ainda não tenha sido explorada no caminho simples obtido no passo anterior. Além disso, vamos verificar se o caminho contido em CM é maximal.

para todo $i, 1 \leq i \leq n$ **em paralelo faça**
para $ACM[i] = (s, t)$ **faça**
 $D[i] \leftarrow s - t$
 $max \leftarrow \max_{1 \leq i \leq n} \{i, D[i] \neq 0\}$

Caso $max \neq 0$, podemos obter um novo caminho, diferente do armazenado em CM . O novo caminho será formado da raiz inicial da busca até o vértice

$v_i = CM[max]$, unido ao novo caminho a ser obtido a partir da exploração da nova adjacência de v_i .

Vamos agora verificar se o caminho contido em CM é maximal. Isso pode ser facilmente verificado através da posição que o vértice v_i ocupa em CM . Caso v_i seja o último vértice de CM , temos que o caminho simples contido em CM não é maximal.

O passo 4 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 5

Se $max \neq 0$, temos que alterar os sucessores dos vértices $v \notin \{CM[1], CM[2] \dots, CM[max - 1]\}$ para determinar um novo caminho simples a partir do vértice $v_i = CM[max]$. Além disso, temos que atualizar o vetor CM , que juntamente com o vetor $C1$ a ser determinado formarão um novo caminho.

$m \leftarrow max$

para $ACM[m] = (s, t)$ **faça**

$K[CM[m]] \leftarrow t + 1$

para todo $v_j \notin \{CM[1], \dots, CM[m]\}$ **em paralelo faça**

$K[v_i] \leftarrow 0$

$B[v_i] \leftarrow 0$

para todo $i, m \leq i \leq n$ **em paralelo faça**

$ACM[i] \leftarrow (*, *)$

$CM[i] \leftarrow 0$

Vamos também desmarcar as arestas que não fazem parte do caminho simples $CM = \{CM[1], \dots, CM[m]\}$ e reinicializar os vetores $C1$ e AC .

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**

para $INFO[e] = (s, t)$ **faça**

se $t \geq m$ **então**

$INFO[e] \leftarrow (*, *)$

para todo $i, 1 \leq i \leq n$ **em paralelo faça**

$C1[i] \leftarrow 0$

$AC[i] \leftarrow (*, *)$

O passo 5 pode ser executado em tempo paralelo $O(1)$ com m processadores em uma CREW PRAM.

Passo 6

Usando as arestas desmarcadas de *BEDGE*, definimos uma nova raiz $r = v_k$ e aplicamos os passos 2, 3, 4 e 5 até que todos os possíveis caminhos maximais em G com raiz $v = CM[1]$ sejam determinados.

Cada caminho simples maximal C pode ser computado em tempo paralelo $O(\beta \log n)$ com m processadores em uma CREW PRAM, onde β é o número de ciclos que podem ocorrer na determinação de um caminho simples maximal. Como observamos anteriormente, o número de caminhos simples maximais em um digrafo a partir de uma raiz r pode ser exponencial. A complexidade total do algoritmo é de tempo paralelo $O(\alpha\beta \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos no digrafo acíclico a partir da raiz até um vértice sumidouro, e β é o número máximo de ciclos que pode ocorrer na determinação de um caminho maximal.

5.5 Correção e Análise

Lema 5.5.1 *Seja $G = (V, E)$ um digrafo fracamente conexo. Seja $v \in V$ e um kl -caminho em G tal que $k = v$ e $l = \text{suc}[v]$. Então o kl -caminho é: (i) um caminho simples; (ii) um ciclo; ou (iii) um caminho unido a um ciclo.*

Demonstração: Seja $C = \{v_1, \dots, v_i\}$ um caminho simples no kl -caminho, $k = v_1$ e $l = v_2$. Vamos estender o caminho C até o sucessor w de v_i . Caso $w \in C$, obtemos um ciclo e pelo fato de que os sucessores são fixos, $\text{suc}[w] \in C, w = \text{suc}[v_i]$. Se $w \neq v_1$, temos que o kl -caminho é um caminho simples unido de um ciclo. Se pudermos estender C e não obtivermos um ciclo, como o número de vértices é finito, existirá um sumidouro no kl -caminho. Nesse caso obtemos um caminho simples. \square

Lema 5.5.2 *Seja $G = (V, E)$ um digrafo acíclico fracamente conexo. Cada kl -caminho é um caminho simples maximal.*

Demonstração: Pelo fato de que o digrafo é acíclico, o caminho determinado é simples. Seja v_p o último vértice do caminho determinado no kl -caminho. Temos que v_k é um sumidouro, pois caso contrário existiria um elemento na lista de adjacência de v_k , e essa nova aresta faria parte do caminho. \square

Lema 5.5.3 *Seja $G = (V, E)$ um grafo não orientado conexo, aplicado ao algoritmo de busca irrestrita. Seja $v \in V$ incluído i vezes no vetor C durante o processo.*

Denote por C_i a configuração de C que antecede $C[j] = v$, após a inclusão de v em C pela i -ésima vez. Então necessariamente:

(i) A configuração de C , quando v for excluído pela i -ésima vez de C , é também igual a C_i .

(ii) $i \neq j \Rightarrow C_i \neq C_j$.

Demonstração: O vetor C simula uma pilha no algoritmo de busca irrestrita. Um vértice v só é incluído em C se estiver desmarcado. Logo a configuração de C quando v for excluído pela i -ésima vez é igual a C_i . Um vértice v só será reexplorado, caso exista um caminho de um vértice u que antecede v em C , e os vértices do caminho não pertencem a C . Logo, temos que se $i \neq j$ então $C_i \neq C_j$. \square

O lema acima garante que o algoritmo termina, pois se um vértice v for inserido em C mais de uma vez, então necessariamente a configuração em C , abaixo de v , é diferente em cada caso. Como existe um número finito de modos de se arranjar essas configurações, o algoritmo necessariamente chega ao final. Ressalte-se que esta propriedade depende fortemente do fato de que qualquer vértice não pode ser reexplorado, enquanto estiver *marcado*. A reexploração de um vértice marcado pode causar ciclicidade.

Teorema 5.5.1 *Seja $G = (V, E)$ um grafo conexo. Todo caminho C encontrado pelo algoritmo em G a partir do vértice raiz r em um kl -caminho com $k = r$ é maximal.*

Demonstração: A exploração de um kl -caminho com $k = r$ termina quando um ciclo é detectado. Os vértices do caminho simples encontrado no kl -caminho são armazenados em C . O algoritmo determina o vértice mais distante de r em C que possui alguma adjacência que possa ser explorada.

Caso o vértice com alguma adjacência que possa ser explorada for o último vértice de C , o algoritmo altera as adjacências dos vértices que não pertencem a C e o algoritmo estende o caminho C até que um novo ciclo seja detectado.

Caso o vértice com alguma adjacência que possa ser explorada seja diferente do último vértice em C , então C é um caminho maximal. \square

Teorema 5.5.2 *Seja $G = (V, E)$ um grafo conexo. Todo caminho maximal de G a partir do vértice raiz r será determinado pelo algoritmo pelo menos uma vez.*

Demonstração: Uma vez obtido um caminho maximal C , a partir de r , o algoritmo determina o vértice v_i em C mais distante de r . Todos os vértices após v_i em C são retirados de C e todas as adjacências dos vértices que não pertencem a C são reinicializadas. A adjacência do vértice v_i é aumentada em 1, e o grafo é decomposto em um novo conjunto de kl -caminhos, com $k = r$.

O caminho C obtido no novo kl -caminho é diferente do anterior, pois o vértice a ser incluído em C após v_i é diferente do vértice anteriormente em C .

Pelo fato de explorarmos todas as possíveis adjacências de cada caminho C , temos que todos os caminhos simples maximais são computados. \square

Teorema 5.5.3 *Seja $G = (V, E)$ um grafo conexo. Nenhum caminho maximal a partir do vértice raiz r será encontrado mais de uma vez pelo algoritmo.*

Demonstração: Pelo lema anterior, temos que se um vértice v é incluído em um caminho mais de uma vez, a configuração dos vértices que antecedem v são diferentes. Logo o algoritmo não computa um mesmo caminho mais de uma vez. \square

Capítulo 6

Cliques Maximais em Grafos Círculo

6.1 Introdução

O problema do reconhecimento de *grafos círculo* de maneira eficiente ficou aberto por vários anos e foi solucionado independente por [12,34,59]. Esses algoritmos basicamente fazem o uso de uma decomposição do grafo [28,27]. Os algoritmos apresentados são seqüenciais (polinomiais). A existência de um algoritmo paralelo para reconhecimento de *grafos círculo* está em aberto. Um algoritmo paralelo para a decomposição utilizada no algoritmo seqüencial é apresentada em [62]. Nesse trabalho é sugerido que as idéias da paralelização da decomposição sejam utilizadas na tentativa de se obter um possível algoritmo paralelo para reconhecimento de *grafos círculo*. Os problemas de clique máximo ponderado e conjunto independente máximo ponderado em *grafos círculo* podem ser resolvidos em tempo polinomial [38]. Contudo, os problemas de coloração mínima e cobertura por cliques em *grafos círculo* são *NP*-completos [37].

Vamos descrever um algoritmo para geração de todas as cliques maximais, de um *grafo círculo* em tempo paralelo de $O(\alpha \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde n, m e α são o número de vértices, arestas e cliques maximais do grafo, respectivamente. O algoritmo apresentado é baseado na versão seqüencial apresentado por Szwarcfiter e Barroso [79]. A versão seqüencial é uma aplicação de uma orientação especial de um grafo denominada *localmente transitiva*. Essa orientação é uma generalização de uma *orientação transitiva* de um grafo. Vamos também apresentar um algoritmo paralelo para computar o número de cliques maximais α_k de tamanho k , o número total de cliques maximais α e a clique máxima de um *grafo círculo* em tempo paralelo $O(\log^2 n)$ com n^3 , $nM(n)$ e n^3 processadores,

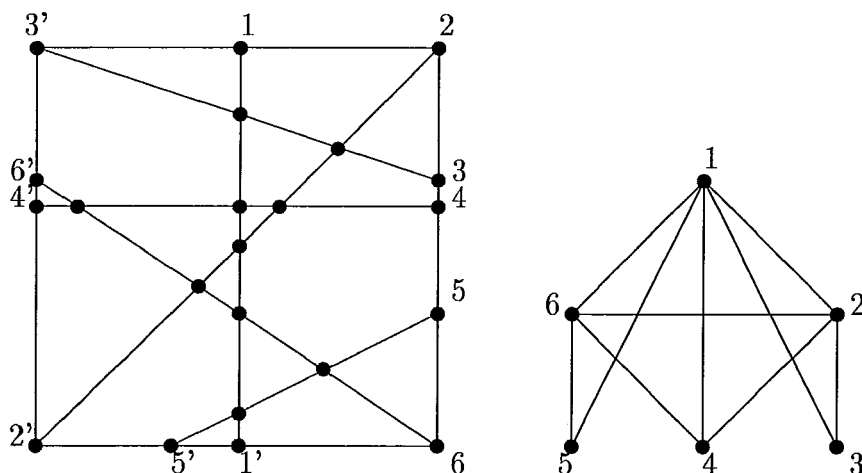


Figura 6.1: Grafo $G = (V, E)$

respectivamente, em uma CREW PRAM, onde $M(n)$ é o número de processadores necessários para efetuar uma multiplicação de duas matrizes $n \times n$.

O número de cliques maximais α de um *grafo círculo* pode ser exponencial [57]. Isso faz com que seja altamente ineficiente computar α através da geração de todas as cliques maximais e contá-las, sendo importante o cálculo do número de cliques maximais independente de computá-las. Cabe salientar que o problema de determinar o número de cliques maximais de um grafo qualquer é um problema $\#P$ -completo [85].

As cliques maximais de um grafo qualquer podem ser computadas através do algoritmo [29] em tempo paralelo $O(\alpha \log^3 n)$ com $\alpha^6 n^2$ processadores em uma CREW PRAM, onde α é o número de cliques.

6.2 Notação e Terminologia

Grafos Círculo são os grafos de interseção de uma família de cordas em um círculo C . A Figura 6.1 ilustra um exemplo. Ao invés de usarmos um círculo, podemos usar um retângulo. Assumimos que na nossa definição de grafo círculo duas cordas não possuem um ponto em comum em C . Uma *seqüência circular* S de G é a seqüência dos $2n$ pontos finais distintos das cordas em C que obtemos ao percorrer C em uma direção fixada, começando de um ponto escolhido em C . Denotamos por $S_1(v)$ e $S_2(v)$ respectivamente a primeira e a segunda instâncias em S da corda correspondente a $v \in V$ em C . Denotamos $S_i(v) < S_j(w)$ quando $S_i(v)$ precede $S_j(w)$ em S . Temos que $S_1(v) < S_2(v)$.

\vec{G} denota uma orientação acíclica de G . $A_v(\vec{G})$ e $A_v^{-1}(\vec{G})$ são respectivamente os subconjuntos dos vértices saindo e entrando em v . Para $v, w \in V$, v

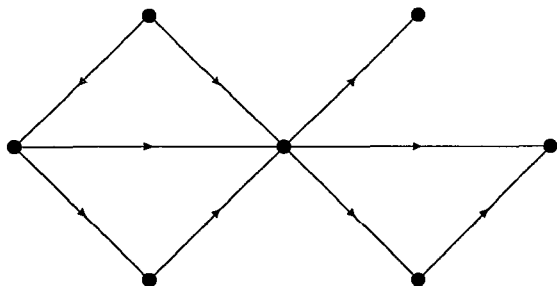


Figura 6.2: Digrafo $G = (V, E)$ localmente transitivo

é um *ancestral* de w em \vec{G} se o digrafo contiver um caminho $v - w$. Neste caso, w é um descendente de v . Denotamos por $D_v(\vec{G})$ o conjunto dos descendentes de v . Se $w \in D_v(\vec{G})$ e $v \neq w$, então v é um *ancestral próprio* de w e w um *descendente próprio* de v . \vec{G} é denominado *transitivo* com relação a arestas quando $(u, v), (w, z) \in E$ implica $(v, z) \in E$. A *redução transitiva* \vec{G}_R é o subgrafo de \vec{G} formado exatamente pelas arestas as quais não são motivadas pela transitividade.

Seja $G = (V, E)$ um grafo não orientado, $|V| > 1$ e \vec{G} uma orientação acíclica de G . Seja $v, w \in V$ e denotamos por $Z(v, w) \subset V$ o subconjunto de vértices os quais são simultaneamente descendentes de v e ancestrais de w em \vec{G} . Uma aresta $(v, w) \in E$ *induz uma transitividade local* quando $\vec{G}(Z(v, w))$ é um digrafo transitivo. Claramente, neste caso os vértices de qualquer caminho de v a w induzem uma clique em G . Além disso, (v, w) *induz uma transitividade local maximal* quando não existe $(v', w') \in E$ diferente de (v, w) tal que v' é simultaneamente um ancestral de v e w' um descendente de w em \vec{G} . A orientação \vec{G} é *localmente transitiva* quando cada uma de suas arestas induz transitividade local. A Figura 6.2 mostra um exemplo de uma orientação localmente transitiva.

A aplicação de orientações localmente transitivas para a enumeração de cliques maximais é baseada no seguinte teorema.

Teorema 6.2.1 *Seja $G = (V, E)$ um grafo não orientado, \vec{G} é uma orientação localmente transitiva de G e \vec{G}_R a redução transitiva de \vec{G} . Então existe uma correspondência um a um entre as cliques maximais de G e caminhos $v - w$ em \vec{G}_R , para todas arestas maximais $(v, w) \in E$.*

Demonstração: [79].

6.3 Grafos Círculo

O Teorema 1 sugere um método para a enumeração das cliques maximais de um grafo G , desde que uma orientação localmente transitiva seja dada. Descreveremos abaixo como Szwarcfiter e Barroso [79] mostram que isso pode ser obtido de maneira bastante simples para grafos círculo.

Lema 6.3.1 *Seja $G = (V, E)$ um grafo círculo, S a seqüência circular de G e $v_1, \dots, v_k \in V$ satisfazendo $S_1(v_i) < S_1(v_{i+1})$, $1 \leq i \leq k$. Então $\{v_1, \dots, v_k\}$ induz uma clique em G se e somente se $S_1(v_k) < S_2(v_1) < \dots < S_2(v_k)$.*

Demonstração: [79]

Seja $G = (V, E)$ um grafo círculo e S a seqüência circular de G . Uma S_1 -orientação \vec{G} de G é uma orientação na qual toda aresta dirigida $(v, w) \in E$ satisfaz $S_1(v) < S_1(w)$.

Lema 6.3.2 *Se (v_1, v_k) é uma aresta de uma S_1 -orientação \vec{G} então cada caminho v_1, \dots, v_k induz uma clique em G .*

Demonstração: [79]

Teorema 6.3.1 *S_1 -orientações são localmente transitivas.*

Demonstração: [79]

Logo, pelo fato de que as S_1 -orientações \vec{G} são localmente transitivas, podemos facilmente computar as clique maximais de G .

6.4 Algoritmo Seqüencial

Apresentamos o algoritmo seqüencial desenvolvido por Szwarcfiter e Barroso [79] para determinação de todas cliques maximais de um grafo círculo. Além disso, o algoritmo efetua a computação do número de cliques maximais.

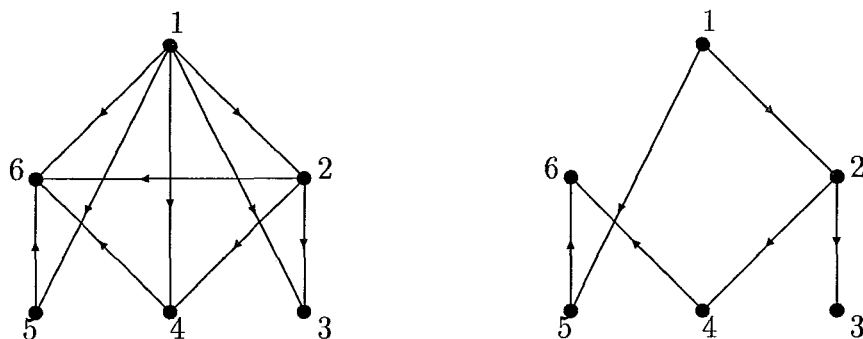


Figura 6.3: (a) Grafo \vec{G} e (b) Grafo \vec{G}_R

6.4.1 Descrição Preliminar do Algoritmo

Algoritmo: Cliques Maximais

1. Construir uma S_1 -orientação \vec{G} de $G = (V, E)$.
2. Construir a redução transitiva \vec{G}_R .
3. Determinar todas as arestas maximais de \vec{G} .
4. Para cada aresta maximal $(v, w) \in E$, determinar todos os caminhos $v - w$ em \vec{G}_R .

6.4.2 O Algoritmo

Vamos descrever de maneira sucinta como esses passos podem ser implementados sequencialmente.

A S_1 -orientação \vec{G} de um dado grafo círculo pode ser facilmente obtida através de sua seqüência circular. Em função disso, o algoritmo assume que seqüência é dada como entrada do algoritmo. O tempo utilizado para computar \vec{G}_R é menor que $O(mn)$. O passo 3 pode ser implementado observando que se \vec{G} é localmente transitivo, então $(v, w) \in E$ é uma aresta maximal se e somente se

$$A_v(\vec{G}) \cap A_w(\vec{G}) = A_v^{-1}(\vec{G}) \cap A_w^{-1}(\vec{G}) = \emptyset$$

A condição acima pode ser verificada facilmente e portanto obtemos todas as arestas maximais em tempo $O(nm)$. Para implementar o passo 4, para cada $v \in V$ definimos $W(v)$ como sendo o subconjunto de vértices w tal que (v, w) é uma aresta maximal. Seja \vec{H} o subgrafo de \vec{G}_R induzido pelos vértices simultaneamente descendentes de v e ancestrais de qualquer $w \in W(v)$. Temos que os caminhos $v - w$ a serem determinados em \vec{G}_R a partir de um determinado vértice v são exatamente

os caminhos fonte-sumidouro em \vec{H} . Cada um desses caminhos pode ser obtido em tempo $O(n)$, usando busca em profundidade irrestrita em \vec{H} , a partir de uma fonte v . Logo a complexidade total é $O(n(m + \alpha))$.

O algoritmo Cliques Maximais e o digrafo \vec{H} são utilizados para computar o número de Cliques maximais.

Algoritmo: Número de Cliques Maximais

1. Aplicar os passos 1,2, e 3 do algoritmo Cliques Maximais.
2. Para cada $v \in V$, construir \vec{H} como descrito acima e computar o número de caminhos fonte sumidouro de \vec{H} de acordo com a seguinte computação. Para cada vértice $u \in \vec{H}$ atribua um rótulo $L(u)$. Defina $L(u) = 1$ se u for um sumidouro de \vec{H} . Caso contrário,

$$L(u) = \sum L(t) \text{ para } t \in A_u(\vec{H})$$

$$L(v) \text{ é igual ao número de caminhos fonte sumidouro de } \vec{H}.$$
3. $\alpha = \sum L(v)$.

Como foi observado anteriormente, o passo 1 pode ser implementado em tempo $O(nm)$. Para cada digrafo \vec{H} , $L(v)$ é obtido em tempo $O(m)$, bastando para isso calcular $L(u)$ dos sumidouros para a fonte v . Logo a complexidade total é $O(nm)$.

A correção do algoritmo segue diretamente da correspondência um a um entre as cliques maximais e os caminhos fonte sumidouro em \vec{H} , para cada $v \in V$.

6.5 Algoritmo para Determinação das Cliques Maximais em um Grafo Círculo

Nesta seção, descrevemos um algoritmo paralelo para geração de todas as cliques maximais de um grafo círculo que é executado em tempo paralelo $O(\alpha \log^2 n)$ com n^3 processadores em uma CREW PRAM. Utilizamos o exemplo da figura 6.1 para ilustrar os passos do algoritmo.

O algoritmo paralelo para geração de todas as cliques maximais usa os mesmos passos básicos descritos no algoritmo seqüencial. O que difere do algoritmo seqüencial são as implementações dos passos.

6.5.1 Descrição Preliminar do Algoritmo

Algoritmo: Cliques Maximais

1. Construir a redução transitiva \vec{G}_R .
2. Determinar todas as arestas maximais de \vec{G} .
3. Para cada aresta maximal $(v, w) \in E$, determinar todos os caminhos $v - w$ em \vec{G}_R .

6.5.2 O Algoritmo e um Exemplo

Vamos agora descrever como os passos do algoritmo podem ser paralelizados. A entrada é a S_1 -orientação \vec{G} de um grafo $G = (V, E)$. Para o nosso exemplo da figura 6.1 as arestas são

$$\boxed{(1, 2)(1, 3)(1, 4)(1, 5)(1, 6)(2, 3)(2, 4)(2, 6)(4, 6)(5, 6)}$$

Passo 1

Neste passo, determinamos a redução transitiva \vec{G}_R de \vec{G} . Para isso, inicialmente determinamos as listas de adjacências $A_v^{-1}(\vec{G})$ para todos os vértices de \vec{G} .

Podemos facilmente obter as listas $A_v^{-1}(\vec{G})$ através da partição da lista de arestas em sublistas, uma para cada vértice v . Essa partição em sublistas pode ser efetuada em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Utilizamos o vetor de vetores ADJ para armazenar as listas de adjacências. $ADJ[v_i]$ contém a lista dos vértices v_j que chegam a v_i . Utilizando duplicação recursiva, retiramos de cada lista de adjacência $ADJ[v_i]$ todos os vértices constantes nas listas de adjacências $ADJ[v_j]$, para todo $v_j \in ADJ[v_i]$. Isso pode ser efetuado da seguinte maneira

para todo $v_i \in V$ em paralelo faça

$$\begin{aligned} ADJ[v_i] &\leftarrow A_{v_i}^{-1}(\vec{G}) \\ BADJ[v_i] &\leftarrow \bigcup_{v_j \in ADJ[v_i]} ADJ[v_j] \end{aligned}$$

repita $\lceil \log n \rceil$ vezes

para todo $v_i \in V$ em paralelo faça

$$BADJ[v_i] \leftarrow BADJ[v_i] \cup \{ \cup_{v_j \in BADJ[v_i]} BADJ[v_j] \}$$

para todo $v_i \in V$ em paralelo faça

$$ADJ[v_i] \leftarrow ADJ[v_i] - BADJ[v_i]$$

As listas de adjacências $A_{v_i}^{-1}(\vec{G}_R)$ são dadas por $ADJ[v_i]$.

Cada lista de adjacências pode ter no máximo n vértices. A cada passo fazemos a compressão de cada lista de adjacências para retirar os vértices repetidos. Logo as operações acima podem ser executadas em tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM.

Para o nosso exemplo \vec{G}_R é

$$(1, 2)(1, 5)(2, 3)(2, 4)(4, 6)(5, 6)$$

O passo 1 pode ser executado em tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM.

Passo 2

Nesse passo, determinamos as arestas maximais de \vec{G} . Para isso utilizamos o mesmo argumento usado no algoritmo seqüencial, de que se \vec{G} é localmente transitivo então $(v, w) \in E$ é uma aresta maximal se e somente se

$$A_v(\vec{G}) \cap A_w(\vec{G}) = A_v^{-1}(\vec{G}) \cap A_w^{-1}(\vec{G}) = \emptyset$$

Como no passo anterior, caso as listas de adjacências de entrada e saída de \vec{G} não façam parte da entrada, elas podem ser facilmente computadas através de \vec{G} . Observamos que, neste caso, temos que efetuar duas partições distintas na lista de arestas de \vec{G} .

$$\begin{array}{ll} A_1(\vec{G}) = \{2, 3, 4, 5, 6\} & A_1^{-1}(\vec{G}) = \emptyset \\ A_2(\vec{G}) = \{3, 4, 6\} & A_2^{-1}(\vec{G}) = \{1\} \\ A_3(\vec{G}) = \emptyset & A_3^{-1}(\vec{G}) = \{1, 2\} \\ A_4(\vec{G}) = \{6\} & A_4^{-1}(\vec{G}) = \{1, 2\} \\ A_5(\vec{G}) = \{6\} & A_5^{-1}(\vec{G}) = \{1\} \\ A_6(\vec{G}) = \emptyset & A_6^{-1}(\vec{G}) = \{1, 2, 4, 5\} \end{array}$$

Com as listas de adjacências de entrada e saída de cada um dos vértices calculamos as arestas maximais. As arestas maximais para o nosso exemplo são

$$\boxed{(1, 3)(1, 6)}$$

O passo 2 pode ser executado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 3

Neste passo, construímos para cada $v \in V$ os subconjuntos $W(v)$ formados pelos vértices w tal que (v, w) é uma aresta maximal. Isso pode ser efetuado com a ordenação da lista de arestas maximais (v, w) determinadas no passo anterior e pela divisão dessa lista em sublistas, uma para cada v . No nosso exemplo temos

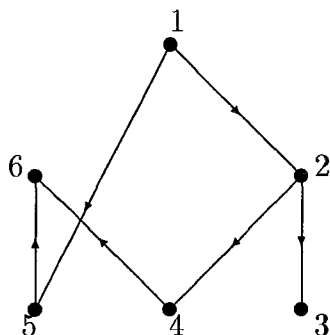
$$\begin{aligned} W(1) &= \{3, 6\} & W(2) &= \emptyset \\ W(3) &= \emptyset & W(4) &= \emptyset \\ W(5) &= \emptyset & W(6) &= \emptyset \end{aligned}$$

Construímos agora o subgrafo \vec{H} de \vec{G}_R induzido pelos vértices simultaneamente descendentes de v e ancestrais de qualquer $w \in W(v)$. Isso pode ser efetuado através da determinação do fecho transitivo [50] \vec{G}_R e a com a interseção dos vértices que saem de v com os que chegam em cada um dos $w \in W(v)$. Temos que os caminhos $v - w$ em \vec{G}_R tomados a partir de um determinado vértice v são exatamente os caminhos-fonte sumidouro em \vec{H} . Esses caminhos podem ser obtidos através do algoritmo paralelo para busca em profundidade irrestrita.

O grafo \vec{H} pode ser construído em tempo paralelo $O(\log^2 n)$ com $M(n)$ processadores em uma CREW PRAM, onde $M(n)$ é o número de processadores necessários para efetuar uma multiplicação de duas matrizes $n \times n$ em uma CREW PRAM. Em [25] $M(n)$ é $O(n^{2.376})$. Para o nosso exemplo, o subgrafo \vec{H} é formado por

$$\boxed{(1, 2)(1, 5)(2, 3)(2, 4)(4, 6)(5, 6)}$$

Vamos agora efetuar uma busca irrestrita no digrafo \vec{H} . Observamos que o digrafo \vec{H} é acíclico. Com a utilização do algoritmo paralelo para busca irrestrita apresentado no capítulo anterior, determinamos os caminhos maximais de \vec{H} . Para o nosso exemplo, os caminhos maximais em \vec{H} são

Figura 6.4: Grafo \vec{H}

$$C_1 = \{1, 2, 3\}$$

$$C_2 = \{1, 2, 4, 6\}$$

$$C_3 = \{1, 5, 6\}$$

O passo 3 pode ser executado em tempo $O(\log^2 n)$ com $M(n)$ processadores em uma CREW PRAM.

Os caminhos fonte-sumidouro de \vec{H} obtidos no passo anterior formam as cliques maximais de G .

A complexidade total do algoritmo de determinação das cliques maximais de um grafo círculo é de tempo paralelo $O(\alpha \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde α é o número de cliques maximais.

6.6 Algoritmo para Determinação do Número de Cliques Maximais de um Grafo Círculo

Nesta seção, descrevemos um algoritmo paralelo para determinação do número de cliques maximais α_k de tamanho k e o número total de cliques maximais α de um grafo círculo que é executado em tempo paralelo $O(\alpha \log^2 n)$ com n^3 e $nM(n)$ processadores em uma CREW PRAM, onde $M(n)$ é o número de processadores necessários para efetuar uma multiplicação de duas matrizes $n \times n$. Vamos também determinar uma clique máxima em tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM. Utilizamos o exemplo da figura 6.1 para ilustrar os passos do algoritmo.

Para determinar o número de cliques maximais em paralelo utilizamos uma idéia diferente da proposta para o algoritmo seqüencial. Isso é motivado pelo fato de no algoritmo seqüencial, inicialmente rotula-se os vértices sumidouro de

H com 1. A seguir, quando todos os vértices adjacentes a um vértice v_i estiverem rotulados, rotula-se v_i com a soma dos rótulos de seus vértices adjacentes. Esse procedimento é efetuado até que todas os vértices fontes estejam rotulados. Essa estratégia parece ser inerentemente seqüencial.

6.6.1 Descrição Preliminar do Algoritmo

Algoritmo: Número de Cliques Maximais

1. Aplicar os passos 1 e 2 do algoritmo Cliques Maximais.
2. Construir o subgrafo \vec{H} de \vec{G}_R induzido pelos vértices simultaneamente descendentes de v e ancestrais de qualquer $w \in W(v)$, onde $W(v)$ é o subconjunto de vértices w tal que (v, w) é uma aresta maximal.
3. Computar A^k , onde A é a matriz de adjacências do grafo H .

6.6.2 O Algoritmo e um Exemplo

Passo 1

Neste passo, executamos as mesmas operações do algoritmo anterior.

Passo 2

Neste passo, computamos apenas o grafo \vec{H}

Passo 3

Neste passo, vamos calcular o número de cliques maximais de tamanho k e o número de cliques maximais de G .

Motivados pelo fato de que a multiplicação de matrizes booleanas é um problema com solução em NC , e considerando que a matriz A como sendo a matriz de adjacência do digrafo \vec{H} , temos que A^2 corresponde aos caminhos de comprimento 2 existentes no digrafo. Logo basta examinarmos a linha da matriz referente aos vértices fontes com as colunas referentes aos vértices sumidouro para obter o número de cliques de tamanho 3 no grafo. De uma forma geral, temos que A^k fornece o número de cliques de tamanho $k + 1$. Para o nosso exemplo temos

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Logo, temos dois caminhos de comprimento 2 um entre os vértices 1 e 3 e outro entre os vértices 1 e 6.

$$A^3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A matriz A^3 indica a existência de um caminho de comprimento 3 entre os vértices 1 e 6.

A computação de cada uma das matrizes A^k pode ser efetuada em tempo paralelo $O(\log n)$ com $M(n)$ processadores em uma CREW PRAM [50]. Logo, para computar o número total de cliques maximais necessitamos efetuar cada uma das A^k em paralelo. Isso pode ser efetuado em tempo paralelo $O(\log^2 n)$ com $nM(n)$ processadores em uma CREW PRAM [50]. Para computar uma clique máxima, basta verificar o valor de k na obtenção de A^k .

A complexidade total do algoritmo de determinação do número de cliques maximais de tamanho k e do número de cliques maximais de um grafo círculo é de tempo paralelo $O(\log^2 n)$ utilizando n^3 e $nM(n)$ processadores em uma CREW PRAM respectivamente. A complexidade total para computar uma clique máxima de um grafo círculo é de tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM.

6.7 Correção e Análise

Lema 6.7.1 *O grafo \vec{G}_R obtido ao final do Passo 1 do algoritmo paralelo é a redução transitiva de \vec{G} .*

Demonstração: A cada iteração i , computamos para cada vértice v o conjunto L_v formado pela união das listas de adjacências dos vértices que chegam a v . O conjunto L_v contém os vértices para os quais existe pelo menos um caminho de comprimento $\geq 2^i$ a v . A cada iteração i , para cada vértice v , retiramos da lista de adjacências dos vértices que chegam a v , os vértices pertencentes a L_v . Como utilizamos duplicação recursiva, após $\lceil \log n \rceil$ iterações, todos os vértices que são ancestrais de v pertencem a L_v . Como retiramos de cada lista de adjacências dos vértices que chegam a v os vértices pertencentes a L_v , as listas de adjacências conterão apenas os vértices v_j , para os quais não existe um caminho ligando u_j a v diferente de (u_j, v) . \square

Teorema 6.7.1 *O algoritmo paralelo computa corretamente todas as cliques maximais de um grafo círculo em tempo paralelo $O(\alpha \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde α é o número de cliques maximais.*

Demonstração: Pelo lema anterior, a redução transitiva é computada corretamente. Com a utilização do algoritmo de busca em um digrafo acíclico, determinamos todos os caminhos maximais no digrafo \vec{H} . Pelo fato de que a S_1 -orientação \vec{G} é localmente transitiva, o resultado segue-se. \square

Teorema 6.7.2 *O algoritmo paralelo computa corretamente o número de cliques maximais de tamanho k e o número de cliques maximais de um grafo círculo em tempo paralelo $O(\log^2)$ com n^3 e $nM(n)$ processadores em uma CREW PRAM, respectivamente.*

Teorema 6.7.3 *O algoritmo paralelo computa corretamente a clique máxima de um grafo círculo em tempo paralelo $O(\log^2)$ com n^3 processadores em uma CREW PRAM.*

Capítulo 7

Algoritmos Seqüenciais para Determinação de Ciclos em um Digrafo

7.1 Introdução

Existe uma classe de problemas na qual o objetivo é a enumeração de um conjunto de objetos associados a um determinado grafo. Exemplos incluem a enumeração de ciclos elementares, enumeração de árvores geradoras e a enumeração das cliques de um grafo. Para cada destes três problemas, podemos construir grafos com $|V|$ vértices que possuem $2^{|V|}$ ou mais objetos para serem enumerados. Desta forma, em geral, algoritmos para resolver esses problemas possuem tempo de execução exponencial com relação ao tamanho do grafo. Para tais problemas, um dado algoritmo pode ser adicionalmente caracterizado pela introdução de um limite de tempo por objeto obtido, e dois algoritmos podem ser comparados de acordo com seus limites por objeto. Desta forma, é de grande utilidade o desenvolvimento de algoritmos para a enumeração cujo tempo de execução seja polinomial no número de objetos gerados. O problema da determinação de todos os ciclos elementares de um digrafo cai nesta categoria. Dentre o grande número de algoritmos para determinação de todos os ciclos descritos por [84,83,66,48,81,71], os algoritmos propostos por [48,71,81], apresentam o melhor limite de tempo, um limite linear em relação ao tamanho do grafo, por ciclo. Esses algoritmos foram obtidos através da imposição de restrições adicionais ao backtracking (com restrições) executado no algoritmo proposto por [83]. O algoritmo proposto por Szwarcfiter e Lauer [81] utiliza uma abordagem diferente do algoritmo de Johnson [48] e o algoritmo de Read e Tarjan [71] utiliza busca em profundidade no digrafo. Vamos agora descrever alguns dos principais algoritmos

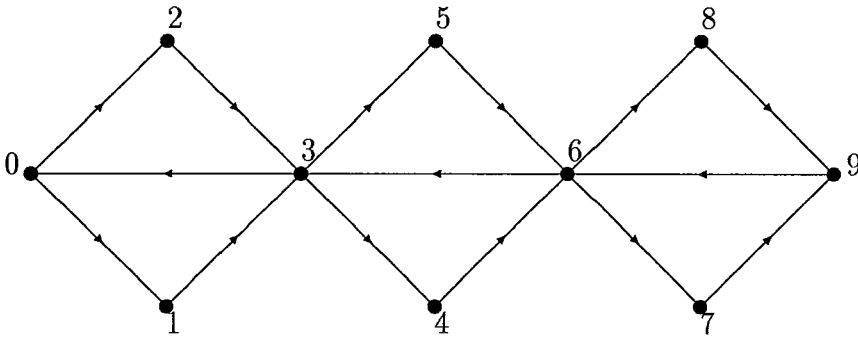


Figura 7.1: Grafo $G = (V, E)$

seqüenciais para enumerar todos os ciclos elementares de um dado digrafo.

7.2 Notação e Terminologia

Um digrafo G é *fortemente conexo* quando para todo par de vértices $u, v \in V$ existir um caminho em G de u para v e também de v para u .

7.3 Ciclos Elementares em Digrafos

Vamos descrever os principais algoritmos seqüenciais para a geração de todos os ciclos elementares de um digrafo. O algoritmo de Tiernan [84] essencialmente utiliza um procedimento de backtraking irrestrito o qual explora todos os caminhos elementares do digrafo e verifica se eles possuem ciclos. Se os vértices do digrafo são numerados de 1 a $|V|$, o algoritmo irá gerar todos caminhos elementares $p = \{v_1, \dots, v_k\}$ com $v_1 < v_i$ para $2 \leq i \leq k$, iniciado em algum vértice v_1 , escolhendo uma aresta para algum vértice v_2 , $v_2 > v_1$, e continuando dessa maneira. Quando nenhum novo vértice pode ser visitado, o procedimento efetua uma operação de retrocesso e escolhe um caminho diferente para percorrer. Se v_1 é adjacente a v_k , o algoritmo tem como saída um ciclo elementar (v_1, \dots, v_k, v_1) . O algoritmo enumera cada ciclo elementar exatamente uma vez, visto que cada ciclo contém o menor vértice v_1 uma única vez e desta forma corresponde a um único caminho elementar com vértice inicial v_1 . Contudo, o pior caso do algoritmo tem tempo exponencial no tamanho de sua saída; o algoritmo pode explorar muito mais caminhos elementares que o necessário. O exemplo da Figura 7.1 contém $3n + 1$ vértices, $5n$ arestas e $2n$ ciclos elementares. Entretanto, G contém 2^n caminhos elementares do vértice 1 ao vértice $3n + 1$, os quais são gerados pelo algoritmo de Tiernan.

A abordagem utilizada por Tiernan [84] foi utilizada anteriormente

por [9,32,73] que apresentaram versões não determinísticas para esse algoritmo. Weinblatt [88] também apresenta um algoritmo para determinação dos ciclos elementares de um digrafo, e melhora o tempo de execução do algoritmo proposto por [84], pois o algoritmo armazena os ciclos que foram encontrados e constrói os novos a partir destes. Tarjan [83] dá exemplos ilustrando o fato de que os algoritmos [84,88] podem levar tempo exponencial no número de ciclos enumerados.

O algoritmo proposto por Tarjan [83] é baseado no método de busca em profundidade de Tiernan [84]. O algoritmo utiliza um procedimento de *backtracking*, com a restrição de que somente caminhos viáveis serão explorados. O algoritmo assume que os vértices do digrafo são numerados de 1 a $|V|$, e que o digrafo é representado pelas suas listas de adjacências, uma para cada vértice. A lista de adjacência do vértice v , $A[v]$, contém todos os vértices w tal que (v, w) é uma aresta do digrafo. O algoritmo faz uso de duas pilhas, a pilha *point* e a pilha *mark*, além de um vetor booleano denominado vetor *mark*. A pilha *point* é usada para armazenar o caminho p que está sendo examinado; o caminho elementar tem vértice inicial s . A pilha *mark* é utilizada como um conjunto de ponteiros para o vetor *mark*.

Para cada vértice s , o algoritmo gera caminhos elementares que começam em s e não contêm nenhum vértice menor que s . Uma vez que um vértice tenha sido utilizado em um caminho, este vértice somente pode ser utilizado para um novo caminho quando tiver sido retirado da pilha e quando estiver desmarcado. Se nenhum ciclo foi determinado envolvendo um vértice, ele será retirado da pilha *point*, mas continuará marcado. Um vértice é desmarcado quando puder pertencer a um ciclo elementar que é a extensão do caminho elementar que está sendo explorado. Quando o último vértice de um caminho elementar é adjacente ao vértice inicial s , o caminho elementar corresponde a um ciclo elementar que é enumerado. Quando um novo ciclo é determinado, todos os vértices da pilha *point* serão desmarcados quando retirados da pilha.

O procedimento de marcação é a idéia básica de evitar buscas desnecessárias as quais podem ocorrer na utilização do algoritmo de Tiernan [84], pois um vértice só pode ser incluído na pilha *point* se estiver desmarcado. Contudo existem dois pontos onde o algoritmo proposto por Tarjan [83] continua a efetuar buscas desnecessárias. O primeiro é quando um vértice v vai ser desmarcado porque um ciclo envolvendo v foi encontrado, todos os vértices que estão acima de v na pilha *mark* serão desmarcados, mesmo se alguns deles não são relacionados com o ciclo. O segundo é o fato de que Tarjan segue a abordagem de Tiernan de só procurar ciclos elementares v_j, \dots, v_k com $v_j < v_i, 1 < i \leq k$, onde v_j é o vértice na base da pilha, denominado vértice inicial. O algoritmo proposto por [83] tem tempo de execução $O(nm(C + 1))$.

Read e Tarjan [71] apresentam um algoritmo de tempo de execução

$O(n + m(C + 1))$). Este algoritmo utiliza busca em profundidade no digrafo.

O algoritmo proposto por Johnson [48] também utiliza a abordagem de construir caminhos elementares a partir do vértice inicial da pilha. Para cada componente fortemente conexo, o menor vértice do componente torna-se o vértice inicial. Para cada vértice inicial s , um procedimento recursivo de backtracking é chamado e sua computação é semelhante ao do algoritmo de Tarjan com exceção do sistema de marcação, que é consideravelmente ampliado.

Um vértice v é marcado cada vez que é incluído na pilha. Ao sair da pilha, se um ciclo elementar foi determinado envolvendo v e o vértice inicial s , então v é desmarcado. Caso contrário, o vértice permanece marcado até que outro vértice u seja retirado da pilha e tal que um ciclo elementar envolvendo u e s foi encontrado e existe um caminho de v a u consistindo de vértices que estão marcados e não estão na pilha. Johnson implementa esta abordagem eficientemente usando um conjunto de listas B , uma lista $B[v]$ para cada vértice v . Em um dado momento, $B[v]$ contém os vértices u tais que $(u, v) \in E$ e u está marcado e não está na pilha. A desmarcação de um vértice é efetuada pelo procedimento *UNBLOCK*(u), se $u \in B[v]$. O tempo de execução do algoritmo de Johnson é $O(n + m(C + 1))$.

O algoritmo de Szwarcfiter e Lauer será descrito, em detalhe, na secção 7.5.

7.4 Algoritmo de Tiernan

Nesta secção, apresentamos o algoritmo seqüencial desenvolvido por Tiernan [84] para determinação de todos os ciclos elementares em um digrafo. O algoritmo ignora os ciclos encontrados durante uma iteração cujo vértice final seja diferente do vértice inicial do caminho que está sendo construído. Uma vez determinados todos os possíveis ciclos a partir do vértice inicial, o sucessor desse vértice em alguma ordem inicialmente fixada é definido como novo vértice inicial. O algoritmo não é eficiente, uma vez que não gera todos os ciclos elementares do digrafo em um tempo polinomial por ciclo.

7.4.1 Descrição preliminar do algoritmo

Algoritmo: Ciclos Elementares

1. Iniciar a construção do caminho P a partir do vértice inicial 1.

2. Estender o caminho P e verificar a existência de ciclo elementar.
3. Verificar se todos os ciclos contendo o vértice inicial foram obtidos.
4. Alterar o vértice inicial.
5. Aplicar os passos 2, 3 e 4 até que todos os vértices sejam percorridos e todos os ciclos elementares forem determinados.

7.4.2 O Algoritmo

Nesta seção, discutimos o algoritmo seqüencial para determinar todos os ciclos elementares de um digrafo proposto por Tiernan [84] em detalhe juntamente com um exemplo. O algoritmo necessita que os vértices de G sejam rotulados, em alguma ordem, por inteiros $1, \dots, n$. O algoritmo utiliza o vetor ADJ contendo as listas de adjacências de G , o vetor H que auxilia na verificação de que todos os caminhos a partir do vértice inicial foram explorados e o vetor P que armazena os vértices do caminho elementar que está sendo construído. O primeiro caminho inicia no vértice 1.

Agora descreveremos o algoritmo. A entrada é o digrafo $G = (V, E)$. A lista de adjacências rotuladas com inteiros de $1, \dots, n$ é armazenada em ADJ .

Passo 1

Neste passo, inicializamos os vetores ADJ , H e P , iniciando a determinação dos ciclos a partir do vértice 1. Primeiramente determinamos o grau de cada uma das listas de adjacências ADJ . O primeiro elemento de cada listas será considerado como sendo de índice zero (0), e o número de elementos de cada uma dessas listas será armazenado em $grau[v_i]$. Acrescentamos para cada uma das listas o vértice 0 que informa que toda a lista já foi percorrida.

```

 $k \leftarrow 1$ 
para  $i, 1 \leq i \leq n$  faça
  para  $j, 0 \leq j \leq grau[v_i] - 1$  faça
     $ADJ[i][j] \leftarrow j$  - ésimo elemento da sublista  $v_i$ ;
     $H[i][j] \leftarrow 0$ 
     $ADJ[i][grau(v_i)] \leftarrow 0$ 
     $H[i][grau(v_i)] \leftarrow 0$ 
para  $i, 1 \leq i \leq n$  faça
   $P[i] \leftarrow 0$ 
 $P[1] \leftarrow 1$ 

```

Passo 2

Neste passo, estendemos o caminho P , em uma aresta a cada iteração. A extensão deve obedecer a três critérios. O vértice a ser visitado não pode pertencer a P , além disso deve ser maior que o vértice inicial e não pode ser *fechado* com relação ao último vértice visitado.

A primeira condição assegura que um caminho elementar está sendo construído, a segunda garante que cada circuito irá ser considerado apenas uma vez e a terceira impede de que um caminho elementar seja considerado mais de uma vez.

Quando o caminho P não puder ser mais estendido, verificamos se existe um ciclo em P .

Primeiramente, descrevemos como podemos estender o caminho P .

```

para  $j, 0 \leq j \leq grau(P[k]) - 1$  faça
  se  $ADJ[P[k]][j] > P[1]$  então
     $visita \leftarrow$  verdadeiro
     $caminho \leftarrow$  verdadeiro
    para  $i, 1 \leq i \leq n$  faça
      se  $ADJ[P[k]][j] \neq P[i]$  então
         $visita \leftarrow visita \wedge$  verdadeiro
      para  $t, 0 \leq t \leq grau(P[k]) - 1$  faça
        se  $ADJ[P[k]][j] \neq H[P[k]][t]$  então
           $caminho \leftarrow caminho \wedge$  verdadeiro
    senão
       $visita \leftarrow$  falso
       $caminho \leftarrow$  falso
se  $visita$  e  $caminho$  então
   $k \leftarrow k + 1$ 
   $P[k] \leftarrow ADJ[P[k - 1]][j]$ 

```

Aplicamos o procedimento acima até que o caminho não possa ser mais estendido. Agora vamos verificar a existência de um ciclo em P . Podemos descrever isso da seguinte maneira

```

para  $j, 0 \leq j \leq grau(P[k] - 1)$  faça
  se  $P[1] \neq ADJ[P[k]][j]$  então
    não existe aresta ligando  $P[1]$  a  $j$ 

```

senão

$P[1]$ contem um circuito elementar

Passo 3

Neste passo, verificamos se todos os ciclos contendo o vértice inicial foram determinados.

se $k = 1$ então

todos os ciclos elementares contendo $P[1]$ foram determinados

senão

para $t, 0 \leq t \leq grau(P[k] - 1)$ faça

$H[P[k]][t] \leftarrow 0$

$m \leftarrow \min_{0 \leq t \leq grau(P[k]-1} \{t, H[P[k-1]][t] = 0\}$

$H[P[k-1]][m] \leftarrow P[k]$

$P[k] \leftarrow 0$

$k \leftarrow k - 1$

Caso todos os ciclos elementares não tenham sido determinados, aplicamos os passos 2 e 3 novamente.

Passo 4

Neste passo, alteramos o vértice inicial, e verificamos a existência de ciclos elementares a partir desse novo vértice inicial.

se $P[i] = n$ então

todos os ciclos elementares de G foram determinados

senão

$P[1] \leftarrow P[1] + 1$

$k \leftarrow 1$

$H \leftarrow 0$

Passo 5

Aplicamos os passos 2,3 e 4 até que todos os possíveis ciclos elementares de G sejam determinados.

7.5 Algoritmo de Szwarcfiter e Lauer

Nesta seção apresentamos o algoritmo seqüencial desenvolvido por Szwarcfiter e Lauer [81] para determinação de todos os ciclos elementares em um digrafo. O algoritmo de Szwarcfiter e Lauer também faz uso de um procedimento de backtracking, mas utiliza um sistema mais eficiente para determinar ciclos elementares. Essa determinação ocorre tão logo o ciclo elementar é gerado, qualquer que seja a parte do caminho elementar que está sendo explorado. Para que o algoritmo fique eficiente uma estratégia de marcação de vértices evita regiões do digrafo onde a busca não determinou novos ciclos. O algoritmo apresentado tem complexidade $O(n + m(C + 1))$, onde n , m e C são o número de vértices, arestas e ciclos elementares do digrafo respectivamente.

7.5.1 Descrição Preliminar do Algoritmo

Algoritmo: Ciclos Elementares

1. Determinar os componentes fortemente conexos do digrafo G . Para cada um dos componentes, determinar o vértice de maior grau de saída v_i . Iniciar a construção do caminho P a partir do vértice inicial v_i .
2. Estender o caminho P e verificar a existência de um novo ciclo elementar.
3. Efetuar um backtracking até o vértice mais distante do caminho P que possua alguma adjacência que ainda não tenha sido explorada.
4. Aplicar os passos 2 e 3 até que todos os ciclos elementares de cada componente conexo forem determinados.

7.5.2 O Algoritmo

O digrafo é representado por um conjunto de listas de adjacências com uma lista $A[v]$ para cada vértice v . Uma computação inicial é efetuada para determinar os componentes fortemente conexos do digrafo. Para cada componente fortemente conexo um vértice inicial é escolhido. Esse vértice é o que tem o maior grau de entrada do componente. O algoritmo assegura que quando o vértice inicial é retirado da pilha todos os ciclos elementares deste componente terão sido determinados. Logo apenas um vértice inicial por componente é utilizado. O algoritmo também pode ser executado para um vértice arbitrário ou independentemente da computação dos

componentes conexos, para isso, bastam que sejam efetuadas pequenas alterações no algoritmo. O algoritmo modificado possui a mesma complexidade.

A idéia básica do algoritmo é semelhante aos algoritmos anteriormente descritos, ou seja a cada passo tenta-se estender o caminho elementar que está sendo explorado. Consideremos o caso onde o conteúdo da pilha é $v_1 v_2 \cdots v_{k-1}$ e a aresta (v_{k-1}, v_k) é percorrida:

(i) Se v_k não está marcado então necessariamente v_k não está na pilha, o caminho elementar será estendido até v_k e uma aresta saindo de v_k será explorada.

(ii) Se v_k está marcado e não está na pilha então, necessariamente, não pode existir nenhum novo ciclo elementar a partir do caminho v_1, v_2, \cdots, v_k e portando v_k não será explorado neste passo. O vértice v_{k-1} é incluído na lista $B[v_k]$ e v_k é retirado de $A[v_{k-1}]$.

(iii) Se v_k está marcado e pertence à pilha então um ciclo elementar foi encontrado, e este ciclo é armazenado pelo menos uma vez.

Para considerarmos a determinação de um ciclo quando $v_k \neq v_1$, necessitamos determinar quando encontramos um ciclo anteriormente computado. Para isto, basta observar que um ciclo é um *novo* ciclo quando pelo menos um de seus vértices nunca foi retirado da pilha.

Nos casos em que v_k está marcado, o caminho elementar não é estendido, e uma nova aresta a partir de v_{k-1} será explorada. Se um determinado caminho elementar não pode mais ser estendido, o algoritmo retrocede para um vértice a partir do qual o caminho possa ser estendido. Quando o vértice inicial é retirado da pilha, um novo componente fortemente conexo é explorado. Isso é efetuado até que todos os componentes fortemente conexos sejam computados.

O caminho que está sendo explorado é mantido em uma pilha (pilha *point* no algoritmo de Tarjan [83]). Da mesma forma que no algoritmo de Tarjan [83], o vetor booleano é utilizado, mas não é utilizada a pilha *mark*. Ao invés dessa pilha, o algoritmo utiliza uma versão levemente modificada do sistema de marcação do algoritmo de Johnson [48], usando uma lista $B[v]$ para cada vértice v . Um vértice u é inserido na lista $B[v]$ se $(u, v) \in E$ e a exploração da aresta (u, v) não ocasionou um novo ciclo elementar. Em adição a estas estruturas, o algoritmo utiliza um vetor *position* e um vetor booleano *reach*. Se um vértice v é o j -ésimo vértice a partir do vértice da base da pilha, então $POSITION[v] = j$; quando v é retirado da pilha então $POSITION[v] = n + 1$. Se um vértice v ainda não foi retirado da pilha pela primeira vez, então $REACH[v] = \text{falso}$, caso contrário $REACH[v] = \text{verdadeiro}$. Um vértice v é marcado quando é inserido na pilha, e esse vértice é considerado

marcado pelo menos enquanto pertencer a pilha. Ao ser retirado da pilha, v é desmarcado somente se um novo ciclo elementar contendo v foi encontrado, mas que o vértice na base da pilha não pertence necessariamente a esse caminho. Se v é retirado da pilha e estiver marcado, v só será desmarcado quando um vértice z_1 é retirado da pilha de tal maneira que um novo ciclo elementar foi encontrado com z_1 , e existe um caminho z_k, z_{k-1}, \dots, z_1 , ($z_k = v$) tal que $z_{i+1} \in B[z_i]$, $k < i \leq 1$, naquele passo.

Algoritmo Determinação dos ciclos elementares de um digrafo

início

procedimento *CICLO*(inteiro v, q ; logical f);

início

procedimento *NOCICLO*(inteiro x, y);

início

 inserir x em $B[y]$;

 retirar y de $A[x]$

fim *NOCICLO*;

procedimento *DESMARQUE*(inteiro x);

início

$MARK[x] \leftarrow$ falso;

para $y \in B[x]$ **faça**

início

 inserir x em $A[y]$;

se $MARK[y]$ **então** *DESMARQUE*

fim;

 esvaziar $B[x]$

fim *DESMARQUE*;

logical g ;

$MARK[v] \leftarrow$ verdadeiro

$f \leftarrow$ falso;

 inserir v na pilha;

$t \leftarrow$ numero de vertices da pilha;

$POSITION[v] \leftarrow t$;

se $\neg REACH[v]$ **então** $q \leftarrow t$;

para $w \in A[v]$ **faça**;

se $\neg MARK[w]$ **então**

início

CICLO(w, q, g);

se g **então** $f \leftarrow$ verdadeiro

senão *NOCICLO*(v, w)

fim;

senão

se $POSITION[w] \leq q$ **então**

```

inicio
    escrever o ciclo de  $w$  a  $v$  da pilha;
     $f \leftarrow$  verdadeiro;
fim;
senão
     $NOCICLO(v, w)$ ;
retirar  $v$  da pilha;
se  $f$  então  $DESMARQUE[v]$ ;
 $REACH[v] \leftarrow$  verdadeiro;
 $POSITION[v] \leftarrow n + 1$ 
fim  $CICLO$ 
ler o digrafo  $D$ ;
 $A \leftarrow$  lista de adjacências dos componentes fortemente conexos de  $D$ ;
para  $1 \leq j \leq n$  faça
inicio
     $MARK[j] \leftarrow$  falso;
     $REACH[j] \leftarrow$  falso;
fim
para cada componente fortemente conexo não trivial faça
inicio
     $s \leftarrow$  vertice com o maior grau de entrada do componente;
     $CICLO(s, dummy, dummy)$ ;
fim;
fim;

```

7.6 Correção e Análise

Nesta seção, vamos apresentar as demonstrações referentes aos algoritmos apresentados acima.

7.6.1 Algoritmo de Tiernan

Teorema 7.6.1 *O algoritmo de Tiernan para determinação de todos os ciclos elementares de um digrafo termina.*

Teorema 7.6.2 *Todo ciclo elementar da forma $\{v_1, \dots, v_k\}$, $v_1 < v_2, v_3, \dots, v_k$, $k \leq n$ será determinado pelo algoritmo.*

Corolário 7.6.1 *Todo ciclo elementar será determinado pelo algoritmo.*

Teorema 7.6.3 *Nenhum ciclo elementar é determinado mais de uma vez pelo algoritmo.*

As demonstrações podem ser encontradas em [84].

7.6.2 Algoritmo de Szwarcfiter e Lauer

Seja $D = (V, E)$ o digrafo de entrada sem componentes triviais.

Lema 7.6.1 *Todo vértice entra na pilha pelo menos uma vez.*

Lema 7.6.2 *Se em um dado momento v_1, \dots, v_k estão na pilha e um novo ciclo elementar é encontrado com v_k então todos vértices v_1, \dots, v_k são desmarcados quando forem retirados da pilha.*

Lema 7.6.3 *Seja v_1, \dots, v_k, v_1 um caminho elementar tal que os vértices v_1, \dots, v_k ou uma permutação cíclica desses vértices já tenha aparecido nas k primeiras posições do topo da pilha em algum tempo anterior, e pelo menos um desses vértices tenha sido retirado da pilha anteriormente. Se v_1, \dots, v_k agora ocupam as k primeiras posições do topo da pilha, então todos vértices v_1, \dots, v_k foram anteriormente retirados da pilha.*

Lema 7.6.4 *Sejam z_1, \dots, z_k um caminho elementar e (z_k, v) uma aresta de D , onde v é um vértice da pilha que nunca foi retirado anteriormente da pilha. Então se os vértices z_1, \dots, z_k não pertencem a pilha, z_1 está desmarcado.*

Lema 7.6.5 *Seja v_1, \dots, v_k, v_1 uma permutação cíclica conveniente para um ciclo elementar, tal que v_1 foi o primeiro vértice entre os $v_j, 1 \leq j \leq k$ a ser incluído na pilha. Então existe uma configuração da pilha $v_1 v_2 \dots v_j, 1 \leq j \leq k$ que aparece nas j primeiras posições do topo da pilha antes de v_1 ser retirado da pilha pela primeira vez.*

Lema 7.6.6 *Se um vértice está na pilha, esse vértice está marcado.*

Lema 7.6.7 *Cada ciclo elementar de D é listado pelo menos uma vez.*

Lema 7.6.8 *Cada ciclo elementar de D é listado no máximo uma vez.*

Teorema 7.6.4 *O algoritmo proposto para determinação dos ciclos elementares de D está correto.*

Lema 7.6.9 *Se um vértice muda de marcado para desmarcado duas vezes, um novo ciclo é enumerado.*

Teorema 7.6.5 *O algoritmo requer tempo $O(n + m(C + 1))$ e espaço $O(n + m)$ para enumerar C ciclos elementares.*

Corolário 7.6.2 *Cada ciclo elementar é determinado em $O(m)$ exceto o primeiro ciclo que é determinado em $O(n + m)$.*

As demonstrações podem ser encontradas em [77,80].

Capítulo 8

Algoritmos Paralelos para Determinação de Ciclos em um Digrafo

8.1 Introdução

Como vimos no capítulo anterior, a enumeração dos ciclos elementares de um digrafo são efetuados basicamente com a utilização de *backtracking* e para que a enumeração seja eficiente, necessitamos evitar determinados vértices.

Uma das abordagens no desenvolvimento de algoritmos paralelos para determinação dos ciclos elementares em um digrafo, é a de simular algumas das estruturas que são usadas nos algoritmos seqüenciais. Nos algoritmos seqüenciais, o *backtracking* é efetuado com a utilização de uma pilha. A cada passo, os algoritmos utilizam o vértice que está no topo da pilha e procuram explorar as suas adjacências. No caso de algoritmos paralelos, trabalhamos com um conjunto de vértices ao mesmo tempo. Como numa pilha só podemos acessar o vértice do topo, substituímos a pilha por um vetor. Além disso utilizamos algumas estruturas de dados para auxiliar na simulação de passos dos algoritmos seqüenciais.

Vamos apresentar dois algoritmos paralelos, esses algoritmos são baseados nos algoritmos de Tiernan [84] e Szwarcfiter e Lauer [81] e utilizam a abordagem proposta por [81] de reportar os ciclos assim que forem determinados, nos demais algoritmos os ciclos só são reportados a partir de um vértice inicial.

Os algoritmos determinam um ciclo elementar em tempo paralelo $O(\log^2 n)$ com m processadores em uma *CREW PRAM*. Para a determinação dos

demais ciclos elementares, os algoritmos utilizam tempo paralelo exponencial na implementação da versão de Tiernan [84] e $O(\alpha n \log^2 n)$ com n^3 processadores na implementação da versão de Szwarcfiter e Lauer [81] em uma *CREW PRAM*, onde α é o número de ciclos elementares do digrafo.

8.2 Notação e Terminologia

Relembramos algumas definições do capítulo 5.

Dado um digrafo $G = (V, E)$, para cada vértice $v \in V$, definimos o *sucessor* de v ($suc[v]$) como sendo um elemento fixo da lista de adjacências de v . Seja $e = (u, v) \in E$, definimos o *sucessor* de e como sendo a aresta $(v, suc[v])$. Um *kl-caminho* é um caminho C em G com aresta inicial $e = (k, l)$. Uma aresta $e \in E$ pertence ao *kl-caminho* C se e é sucessor de alguma aresta do caminho C , e $e \notin C$. Uma aresta (u, v) pode pertencer a mais de um *kl-caminho*.

Como demonstraremos posteriormente, se G é um digrafo fortemente conexo, então todo *kl-caminho*, $l = suc[k]$ de G contém um ciclo.

8.3 Algoritmo Paralelo de Tiernan

Nesta seção, apresentamos um algoritmo paralelo para determinação de todos os ciclos elementares em um digrafo. O algoritmo apresentado nesta seção é baseado no algoritmo seqüencial proposto por Tiernan [84]. O algoritmo computa um ciclo elementar de um digrafo G em tempo paralelo $O(\log^2 n)$ com m processadores em uma *CREW PRAM*. Como no algoritmo de Tiernan [84], para determinar todos os ciclos elementares do digrafo o algoritmo pode computar um número muito grande de ciclos elementares que já foram enumerados previamente. O exemplo da Figura 8.1 é utilizado para ilustrar as idéias.

Para desenvolver um algoritmo paralelo para determinação dos ciclos elementares em um digrafo, utilizando a abordagem proposta por Tiernan [84], temos que alterar alguns passos. O passo referente a extensão do caminho verifica se o vértice a ser adicionado ao caminho está marcado. Além disso, o vértice só é visitado se o seu rótulo for maior que o vértice inicial do caminho. Seguindo essa idéia, teríamos $i, 1 \leq i \leq n$ vértices iniciais, e o algoritmo seria aplicado em paralelo para cada um desses vértices. Isso levaria a um número muito grande de processadores. Por outro lado, teríamos que verificar se um ciclo elementar ocorre com o vértice inicial, e pode ocorrer que o número de ciclos elementares encontrados e que não

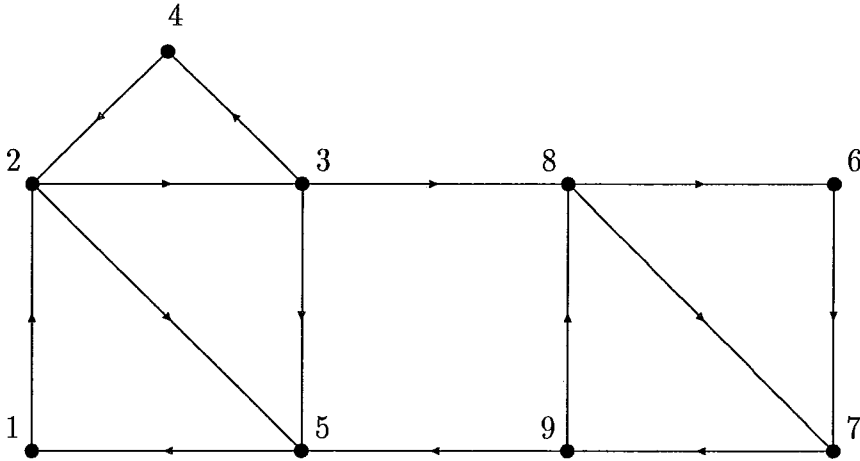


Figura 8.1: Grafo $G = (V, E)$

possuam o vértice inicial seja polinomial com relação a entrada do algoritmo. Em função disso, utilizaremos a abordagem proposta por Szwarcfiter e Lauer [81] para enumeração dos ciclos elementares assim que os mesmos forem computados.

Uma das características dos algoritmos paralelos é o da divisão do grafo em um conjunto de objetos. Cada um desses objetos é definido de tal forma que possam ser aplicadas técnicas paralelas para visitar os vértices ou arestas de cada um desses objetos. No nosso caso, utilizaremos o algoritmo paralelo de busca irrestrita. Como vimos anteriormente, inicialmente definimos os sucessores de cada um dos vértices e arestas do digrafo G . Essa definição divide o digrafo em um conjunto de kl -caminhos. Com a utilização de duplicação recursiva, visitamos as arestas de um kl -caminho a partir de uma raiz r .

Como demonstraremos, todo caminho C em um kl -caminho, $k = r$ e $l = SUC[r]$ termina em um ciclo elementar. Uma vez computado um ciclo elementar $C = \{v_1, \dots, v_k, v_{k+1}\}$, tal que v_{k+1} é igual a um dos $v_j \in C$, $1 \leq j \leq k-1$, utilizamos a mesma abordagem do algoritmo de paralelo de busca irrestrita, e determinamos qual é o vértice v_i mais distante da raiz de busca no caminho $C' = \{v_1, \dots, v_k\}$ que possui alguma adjacência que ainda não foi explorada. Alteramos os sucessores dos vértices que não pertencem ao caminho v_1, \dots, v_{i-1} . Com a alteração dos sucessores, determinamos um ciclo elementar C no subgrafo formado $\{v_1, \dots, v_{i-1}\}$ unido ao novo kl -caminho. Esse procedimento é efetuado até que todos os ciclos elementares sejam enumerados.

8.3.1 Descrição Preliminar do Algoritmo

Algoritmo: Ciclos Elementares em um Digrafo

1. Definir uma raiz $r = v_0$, contruir as listas de adjacências de G e inicializar o caminho simples CM .
2. Decompor o digrafo G em um conjunto de kl -caminhos.
3. Determinar um ciclo elementar C no subgrafo formado pela união de CM e o kl -caminho, com $k = r$ e $l = suc[r]$.
4. Verificar se o ciclo C é um novo ciclo.
5. Computar o maior caminho simples contido no subgrafo formado pela união de CM com o kl -caminho considerado. Armazenar esse caminho em CM . Verificar a existência de algum vértice $v_i \in CM$ que possua na sua lista de adjacências um vértice que ainda não tenha sido visitado.
6. Caso v_i exista, $CM = \{v_0, \dots, v_{i-1}\}$. Alterar os sucessores dos vértices $v_i \notin CM$, e desmarcar as arestas $e \in G$ que não pertencem ao caminho CM .
7. Aplicar os passos 2, 3, 4, 5 e 6 ao conjunto de arestas desmarcadas de G para $r = v_i$ até que todos os possíveis ciclos elementares a partir de v_0 possam ser explorados.

8.3.2 O Algoritmo e um Exemplo

Nesta seção, discutimos o algoritmo para determinar os ciclos elementares de um digrafo, em detalhe, juntamente com um exemplo. Os vetores $EDGE$, $BEDGE$ e SUC contêm m elementos, cada elemento sendo uma *aresta* representada por $e = (vértice1, vértice2)$. O vetor ADJ é um vetor de vetores, sendo que $ADJ[v_i][k]$ representa o k -ésimo elemento da lista de adjacências do vértice v_i , e a posição de cada aresta $SUC[e]$ em $EDGE$ é dada pelo vetor POS . O vetor $AEDGE$ contém m elementos, cada $AEDGE[e]$ é do tipo $(grau[v_i]-1, rank(v_i, v_j)-1)$ onde $rank(v_i, v_j)$ indica a posição do vértice v_j na lista de adjacências de v_i . O vetor $INFO$ contém m elementos, cada $INFO[e]$ é do tipo $(info1, info2)$, onde $info1$ informa em que passo a aresta e foi selecionada, e $info2$ informa a posição de e no caminho C . Os vetores $C1$ e CM , contêm n elementos, são utilizados para armazenar cada caminho que está sendo computado. Os vetores AC e B contêm n elementos. Os vetores K e D são vetores de inteiros utilizados para armazenar valores temporários.

Agora descreveremos o algoritmo. A entrada é o digrafo $G = (V, E)$ da figura 8.1. Consideramos a lista de arestas E ordenada, tal que duas arestas (i, j) e (k, l) , $(i, j) < (k, l)$ se $(i < k)$ ou $((i = k) \wedge (j < l))$. A lista de arestas E é armazenada em $EDGE$. Para nosso exemplo, o vetor $EDGE$ é

(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)

Passo 1

Neste passo, definimos uma raiz r , inicializamos o caminho simples CM , contruímos as listas de adjacências de cada um dos vértices $v_i \in V$ e uma estrutura auxiliar para computação dos ciclos elementares C e do caminho simples CM .

Primeiramente copiamos o vetor $EDGE$ em $BEDGE$, e particionamos $BEDGE$ em um conjunto sublistas, uma para cada $v_i \in V$. Após a partição, computamos o posto ($rank[v_i, v_j]$) em cada uma dessas sublistas. Isso nos informa que o vértice v_j é k -ésimo elemento da lista de adjacências de v_i , $k = rank[v_i, v_j]$. O número de elementos de cada uma das sublistas v_i é armazenado em $grau[v_i]$. Na construção de ADJ , que representa as listas de adjacências de cada v_i , consideramos o primeiro elemento de cada lista de adjacências como sendo de índice zero (0), e acrescentamos o vértice 0 após o último elemento em cada uma das listas para denotar o final da lista.

Para particionar o vetor $BEDGE$ em sublistas, efetuamos uma ordenação paralela de $BEDGE$, e com a utilização de duplicação recursiva computamos o posto das sublistas. Isso pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores.

Vamos computar o vetor $AEDGE$ para auxiliar na obtenção do vértice mais distante da raiz r no caminho simples CM e que possui pelo menos um vértice na lista de adjacências que ainda não foi explorado. Além disso inicializamos os vetores $INFO$, B , K , $C1$ e CM . O vetor B contém n elementos, cada $B[v_i]$ indica o número de vezes que o vértice v_i aparece no ciclo elementar C . Os vetores MV e $MV1$ serão utilizados para verificar se o ciclo elementar C é um *novo* ciclo.

$r \leftarrow$ raiz da busca

$m \leftarrow 1$

$BEDGE \leftarrow EDGE$

para todo $e = (v_i, v_j) \in BEDGE$ **paralelo faça**

$ADJ[v_i][rank(v_i, v_j) - 1] \leftarrow v_j$

$ADJ[v_i][grau(v_i)] \leftarrow 0$
 $AEDGE[e] \leftarrow (grau[v_i] - 1, rank(v_i, v_j) - 1)$
 $INFO[e] \leftarrow (*, *)$

para todo $v_i \in V$ **em paralelo faça**

$K[v_i] \leftarrow 0$
 $B[v_i] \leftarrow 0$
 $MV[v_i] \leftarrow 1$
 $MV1[v_i] \leftarrow 1$

para todo $0 \leq t \leq n$ **em paralelo faça**

$C1[t] \leftarrow 0$
 $CM[t] \leftarrow 0$

A inicialização do vetor *INFO* desmarca todas as arestas $e \in BEDGE$ e a do vetor *K*, que informa qual vértice está sendo utilizado em cada uma das listas de adjacência.

O Passo 1 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 2

Neste passo, determinamos uma decomposição do digrafo em um conjunto de kl -caminhos. Para isso computamos o vetor *SUC* que armazena os sucessores de cada $e = (v_i, v_j) \in BEDGE$. Utilizamos o vetor *POS* para armazenar a posição de *SUC*[e] em *BEDGE* e que será utilizado na obtenção de um caminho C no kl -caminho, $k = r$.

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**

$SUC[e] \leftarrow (v_i, ADJ[v_i][K[v_i]])$
 $POS[e] \leftarrow \text{posição de } (v_i, ADJ[v_i][K[v_i]]) \text{ em } BEDGE$

Para nosso exemplo, os vetores *BEDGE* e *SUC* são

(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)
(2,3)(3,4)(5,1)(4,2)(5,1)(8,6)(2,3)(1,2)(7,9)(9,5)(6,7)(7,9)(5,1)(8,6)

Os vetores *BEDGE* e *SUC* juntos definem um conjunto de kl -caminhos no digrafo G . Em qualquer kl -caminho, a aresta seguinte a aresta $e = (v_i, v_j) \in BEDGE$ está em *SUC*[e].

O Passo 2 pode ser executado em tempo paralelo $O(1)$ com m processadores em uma CREW PRAM.

Passo 3

Neste passo, utilizando duplicação recursiva, determinamos um ciclo C no subgrafo formado pelo caminho CM unido ao kl -caminho, onde $k = r$, e $l = \text{suc}[r]$. Para isso utilizamos o vetor B , para computar o número de vezes que um vértice $v_i \in V$ aparece no subgrafo.

Vamos agora selecionar a primeira aresta do caminho a ser determinado no kl -caminho, $k = r$ e $l = \text{suc}[r]$.

```

itera ← 0
e ← (r, ADJ[r][K[r]])
INFO[e] ← (itera, m)
B[r] ← B[r] + 1
B[ADJ[r][K[r]]] ← B[ADJ[r][K[r]]] + 1

```

Vamos agora determinar quando obtemos um ciclo elementar no kl -caminho que estamos explorando. Temos que se um vértice $v_i \in CM$, então $B[v_i] = 1$. Logo quando obtivermos $B[v_i] > 1$, determinamos um ciclo no subgrafo formado pela união de CM com o kl -caminho considerado. Pelo fato de estarmos aplicando o algoritmo para os componentes fortemente conexos do digrafo G , em no máximo $\lceil \log n \rceil$ iterações, determinamos pelo menos um ciclo.

```

enquanto  $\max_{v_i \in V} \{B[v_i] < 2\}$  faça
  itera ← itera + 1
  para todo  $e = (v_i, v_j) \in \text{BEDGE}$  em paralelo faça
    para  $\text{INFO}[e] = (s, t)$  faça
      se  $\text{INFO}[e] \neq (*, *)$  e  $t \geq m$  então
        se  $\text{INFO}[\text{SUC}[e]] = (*, *)$  então
           $\text{INFO}[\text{SUC}[e]] \leftarrow (\text{itera}, 2^{\text{itera}-1} + t)$ 
        se  $s = \text{itera}$  então
           $B[v_j] \leftarrow B[v_j] + 1$ 
    para todo  $e = (v_i, v_j) \in \text{BEDGE}$  em paralelo faça
       $\text{SUC}[e] \leftarrow \text{SUC}[\text{BEDGE}[\text{POS}[e]]]$ 
       $\text{POS}[e] \leftarrow \text{POS}[\text{BEDGE}[\text{POS}[e]]]$ 

```

Para computar o vetor B , necessitamos evitar escritas concorrentes e isso pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma

CREW PRAM. Pelo fato de termos no máximo $\lceil \log n \rceil$ iterações, determinamos um ciclo elementar em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM.

Vamos agora armazenar os vértices do caminho que foi encontrado. Utilizamos para isso o vetor $C1$. Armazenamos também as informações referentes às arestas que foram percorridas na determinação do caminho C no vetor AC .

para todo $e = (v_i, v_j) \in BEDGE$ **em paralelo faça**
para $INFO[e] = (s, t)$ **faça**
se $INFO[e] \neq (*, *)$ **e** $t \geq m$ **então**
 $C1[t] \leftarrow v_i$
 $AC[t] \leftarrow AEDGE[e]$

Para nosso exemplo, os vetores $BEDGE$ e $INFO$ são

(1,2)(2,3)(2,5)(3,4)(3,5)(3,8)(4,2)(5,1)(6,7)(7,9)(8,6)(8,7)(9,5)(9,8)
(0,0)(1,1)(*,*)(2,2)(*,*)(*,*)(2,3)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)(*,*)

O vetor B é

1,2,1,1,0,0,0,0

O Passo 3 pode ser efetuado em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM.

Passo 4

Neste passo, vamos computar o ciclo elementar C determinado no passo anterior e verificar se C é um novo ciclo.

Inicialmente adicionamos o caminho contido no vetor $C1$ ao caminho contido no vetor CM . O vetor CM armazenará o caminho a partir da raiz inicial da busca v_0 .

para todo $m \leq t \leq n$ **em paralelo faça**
 $CM[t] \leftarrow C1[t]$
 $ACM[t] \leftarrow AC[t]$

Utilizando CM , vamos computar o ciclo elementar C . Para isso, temos que determinar a posição do vértice v_i tal que $B[v_i] > 1$ em CM . Além disso, vamos calcular qual a última posição não nula de CM . Isso pode ser efetuado da seguinte maneira

para todo $1 \leq t \leq n$ **em paralelo faça**
 se $B[CM[t]] > 1$ **então** $vi \leftarrow t$
 $vf \leftarrow \max_{1 \leq t \leq n} \{t, CM[t] \neq 0\}$

Podemos efetuar a operação acima usando duplicação recursiva e árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

O ciclo elementar C é formado por $CM[vi], CM[vi+1], \dots, CM[vf]$. Uma vez determinado o ciclo elementar C , temos que verificar se C , ou alguma de suas permutações cíclicas não foi enumerada anteriormente. Para isso, utilizamos a abordagem proposta por Szwarcfiter e Lauer [81]. Um ciclo C é um *novo* ciclo se e somente se pelo menos um de seus vértices não foi retirado de CM . O vetor MV contém n elementos, cada $MV[v_i]$ indica se o vértice v_i foi retirado de CM .

para todo $t, vi \leq t \leq vf$ **em paralelo faça**
 $MV1[t] \leftarrow MV[CM[t]]$
 $nc \leftarrow \sum_{vi \leq t \leq vf} MV1[t]$

Isso pode ser efetuado usando uma árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Se $nc \neq 0$ temos que o ciclo elementar C não foi enumerado anteriormente.

O Passo 4 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 5

Neste passo, vamos computar o maior caminho simples contido em CM e verificar a existência de alguma adjacência que ainda não tenha sido explorada nos vértices pertencentes a esse caminho.

Para obtermos o maior caminho simples contido em CM , basta retirarmos o último vértice de CM .

Vamos agora determinar a existência de algum vértice $v_i \in CM$ que possui alguma adjacência que ainda não foi explorada.

para todo $i, 1 \leq i \leq n$ em paralelo faça

para $ACM[i] = (s, t)$ faça

$D[i] \leftarrow s - t$

$max \leftarrow \max_{0 \leq i \leq n} \{i, D[i] \neq 0\}$

Podemos efetuar as operações acima utilizando árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Caso $max \neq 0$, podemos estender o caminho $CM[1], \dots, CM[max]$ a partir de $CM[max]$ com a exploração da nova adjacência de $CM[max]$ e tentar obter um novo ciclo elementar.

O Passo 5 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 6

Neste passo, caso $max \neq 0$, atualizamos CM e alteramos os sucessores dos vértices que não pertencem a $\{v_0, \dots, v_{max-1}\}$. Além disso, atualizamos todas as estruturas de dados usadas.

Inicialmente vamos alterar os sucessores dos vértices que não pertencem ao caminho $\{v_0, \dots, v_{max-1}\}$. Além disso, vamos atualizar o vetor B, ACM e CM . Vamos também marcar os vértices que serão retirados de CM .

$m \leftarrow max$

para $ACM[m] = (s, t)$ faça

$K[CM[m]] \leftarrow t + 1$

para todo $v_j \notin \{CM[0], \dots, CM[m]\}$ em paralelo faça

$K[v_i] \leftarrow 0$

$B[v_i] \leftarrow 0$

para todo $i, m \leq i \leq n$ em paralelo faça

$ACM[i] \leftarrow (*, *)$

$CM[i] \leftarrow 0$

para todo $max < t \leq vf$ em paralelo faça

$MV[CM[t]] \leftarrow 0$

Vamos agora desmarcar as arestas que não fazem parte do caminho $CM = \{CM[1], \dots, CM[m]\}$ e reinicializar os vetores $C1$ e AC .

```

para todo  $e = (v_i, v_j) \in BEDGE$  em paralelo faça
  para  $INFO[e] = (s, t)$  faça
    se  $t \geq m$  então
       $INFO[e] \leftarrow (*, *)$ 
para todo  $i, 1 \leq i \leq n$  em paralelo faça
   $C1[i] \leftarrow 0$ 
   $AC[i] \leftarrow (*, *)$ 

```

Com a alteração dos sucessores e a desmarcação das arestas, obtemos uma nova decomposição das arestas desmarcadas do digrafo em um conjunto de kl -caminhos.

O Passo 6 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 7

Usando as arestas desmarcadas de $BEDGE$, definimos uma nova raiz $r = v_k$ e aplicamos os passos 2, 3, 4, 5 e 6 até que todos os possíveis ciclos elementares de G com raiz $v = CM[1]$ sejam determinados.

Um ciclo elementar é computado em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM. O fato de podermos computar um mesmo ciclo um número muito de grande vezes, faz com que o tempo paralelo para determinação de um novo ciclo pode ser exponencial.

8.4 Algoritmo Paralelo de Szwarcfiter e Lauer

Nesta seção, apresentamos um algoritmo paralelo para determinação de todos os ciclos elementares em um digrafo. O algoritmo apresentado nesta seção é baseado no algoritmo seqüencial proposto por Szwarcfiter e Lauer [81]. O algoritmo computa todos os ciclos elementares de um digrafo e é executado em tempo paralelo $O(\alpha n \log n)$ com m processadores em uma CREW PRAM, onde α é o número de ciclos elementares do digrafo G .

O algoritmo utiliza as mesmas idéias do algoritmo anterior. Para evitar que o algoritmo gerasse um mesmo ciclo um número muito grande de vezes,

utilizaremos o critério para marcação dos vértices apresentado por Szwarcfiter e Lauer [81]. Esse critério consiste da construção de um subgrafo de vértices marcados onde uma busca não determinou novos ciclos. Para implementar essa abordagem utilizamos uma lista $LB[v]$ para cada vértice v . Um vértice u é inserido na lista $LB[v]$ se $(u, v) \in E$ e a exploração da aresta (u, v) não determinou um novo ciclo elementar. Um vértice v é considerado marcado enquanto v pertencer ao caminho CM . Se o vértice v for retirado do caminho CM , v será desmarcado somente se um novo ciclo elementar foi computado contendo v . Se v é retirado de CM sem que um novo ciclo elementar contendo v tenha sido determinado, v permanecerá marcado até que um vértice z_1 seja retirado de CM tal que um novo ciclo elementar tenha sido determinado em z_1 , e existe um caminho z_k, z_{k-1}, \dots, z_1 , ($z_k = v$) tal que $z_{i+1} \in LB[z_i]$, $k < i \leq 1$.

8.4.1 Descrição Preliminar do Algoritmo

Algoritmo: Ciclos Elementares em um Digrafo

1. Definir uma raiz $r = v_0$, contruir as listas de adjacências de G e inicializar o caminho simples CM .
2. Decompor o digrafo G em um conjunto de kl -caminhos.
3. Determinar um ciclo elementar C no subgrafo formado pela união de CM e o kl -caminho, com $k = r$ e $l = suc[r]$.
4. Verificar se o ciclo C é um novo ciclo. Caso o ciclo já tenha sido enumerado, para cada aresta (u, v) do ciclo inserir u na lista $LB[v]$.
5. Computar o maior caminho simples contido no subgrafo formado pela união de CM com o kl -caminho considerado. Armazenar esse caminho em CM . Verificar a existência de algum vértice $v_i \in CM$ que possua na sua lista de adjacências um vértice que ainda não tenha sido visitado.
6. Caso v_i exista, $CM = \{v_0, \dots, v_{i-1}\}$. Alterar os sucessores dos vértices $v_i \notin CM$. Caso tenhamos computado um novo ciclo, desmarcar todos os vértices do ciclo e os caminhos formados pelos vértices estão marcados e chegam aos vértices do novo ciclo. Desmarcar as arestas $e \in G$ que não pertencem ao caminho CM .
7. Aplicar os passos 2, 3, 4, 5 e 6 ao conjunto de arestas desmarcadas de G para $r = v_i$ até que todos os possíveis ciclos elementares a partir de v_0 possam ser explorados.

8.4.2 O Algoritmo

Nesta seção, discutimos o algoritmo para determinar os ciclos elementares de um digrafo em detalhe juntamente com um exemplo. Utilizaremos a mesma estrutura de dados do algoritmo anterior, adicionada de um mecanismo para a marcação dos vértices (arestas). O vetor booleano VM de n elementos será utilizado para marcação dos vértices. Uma aresta (u, v) só será incluída em CM se $B[v] < 2$ e $VM[v] = \text{falso}$. As listas $LB[v], v \in V$ conterão os vértices que chegam a v e cuja exploração não determinou um novo ciclo. O vetor booleano NC contém m elementos. Para $e = (u, v)$, $NC[e]$ indica se a exploração da aresta e levou a um novo ciclo elementar. Além disso, incluiremos um campo no vetor $ADJ[v]$ para indicar se o vértice $v_j \in ADJ[v]$ está sendo utilizado.

Agora descreveremos o algoritmo. A entrada é um digrafo $G = (V, E)$. Consideramos a lista de arestas E ordenada, tal que duas arestas (i, j) e (k, l) , $(i, j) < (k, l)$ se $(i < k)$ ou $((i = k) \wedge (j < l))$. A lista de arestas E é armazenada em $EDGE$.

Passo 1

Neste passo, além das operações do algoritmo anterior, inicializamos o vetor VM e as listas $LB[v]$.

para todo $v \in V$ em paralelo faça

$VM[v] \leftarrow \text{falso}$

$LB[v] \leftarrow (0, 0)$

O Passo 1 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 2

Neste passo, utilizando o subgrafo de G , tal que todos os vértices do subgrafo estão desmarcados, efetuamos as mesmas operações do algoritmo anterior.

O Passo 2 pode ser efetuado em tempo paralelo $O(1)$ com m processadores em uma CREW PRAM.

Passo 3

Neste passo, como no algoritmo anterior, utilizando duplicação recursiva, determinamos um ciclo C no subgrafo formado pelo caminho CM unido ao kl -caminho, onde $k = r$, e $l = \text{suc}[r]$.

O Passo 3 pode ser efetuado em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM.

Passo 4

Neste passo, vamos computar um ciclo elementar e verificar se o ciclo elementar determinado já foi enumerado. Além disso, atualizar o vetor NC . Caso tenhamos computado um novo ciclo, temos que $NC[e]$ receberá **verdadeiro** para as arestas e no ciclo elementar. Para o caso em que o ciclo já tenha sido determinado, $NC[e]$ receberá **falso** para as arestas e pertencentes ao kl -caminho.

Temos que o caminho CM adicionado do caminho $C1$ forma um ciclo. Vamos agora adicionar o caminho contido no vetor $C1$ ao caminho contido no vetor CM . O vetor CM armazenará o caminho a partir da raiz inicial da busca.

para todo $m \leq t \leq n$ **em paralelo faça**

$CM[t] \leftarrow C1[t]$
 $ACM[t] \leftarrow AC[t]$

Utilizando CM , vamos computar o ciclo elementar C . Para isso, temos que determinar a posição do vértice v_i tal que $B[v_i] > 1$ em CM . Além disso, vamos calcular qual a última posição não nula de CM . Isso pode ser efetuado da seguinte maneira

para todo $1 \leq t \leq n$ **em paralelo faça**

se $B[CM[t]] > 1$ **então** $vi \leftarrow t$
 $vf \leftarrow \max_{1 \leq t \leq n} \{t, CM[t] \neq 0\}$

As operações acima podem ser feitas usando duplicação recursiva e árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

O ciclo elementar C é formado por $CM[vi], CM[vi+1], \dots, CM[vf], CM[vi]$. Vamos marcar os vértices que fazem parte do caminho CM no kl -caminho.

para todo $t, vi < t \leq vf$ **em paralelo faça**

$VM[CM[t]] \leftarrow \text{verdadeiro}$

Como no algoritmo anterior, uma vez determinado o ciclo elementar C , temos que verificar se C , ou alguma de suas permutações cíclicas não foi enumerada anteriormente. Temos que C é um novo ciclo se e somente se pelo menos um

de seus vértices não foi retirado de CM . O vetor MV contém n elementos, cada $MV[v_i]$ indica se o vértice v_i foi retirado de CM .

para todo $t, v_i \leq t \leq v_f$ **em paralelo faça**

$MV1[t] \leftarrow MV[CM[t]]$

$novociclo \leftarrow \sum_{v_i \leq t \leq v_f} MV[t]$

Isso pode ser efetuado usando árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Se $novociclo \neq 0$ temos que o ciclo elementar C não foi enumerado anteriormente. Caso tenhamos computado um novo ciclo elementar, temos que atualizar o vetor NC , pois ao retirarmos um vértice $CM[t]$ do caminho CM , temos que verificar se a exploração da aresta $(CM[t-1], CM[t])$ levou a um novo ciclo.

se $novociclo \neq 0$ **então**

para todo $t, v_i \leq t \leq v_f$ **em paralelo faça**

$NC[(CM[t], CM[t+1])] \leftarrow \text{verdadeiro}$

senão

para todo $t, m \leq t \leq v_f$ **em paralelo faça**

$NC[(CM[t], CM[t+1])] \leftarrow \text{falso}$

O Passo 4 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 5

Neste passo, vamos computar o maior caminho simples contido em CM e verificar a existência de alguma adjacência que ainda não tenha sido explorada nos vértices pertencentes a esse caminho.

Para obtermos o maior caminho simples contido em CM , basta retirarmos o último vértice de CM .

Vamos agora determinar a existência de algum vértice $v_i \in CM$ que possui alguma adjacência que ainda não foi explorada.

para todo $i, 1 \leq i \leq n$ **em paralelo faça**

para $ACM[i] = (s, t)$ **faça**

$$D[i] \leftarrow s - t$$

$$max \leftarrow \max_{0 \leq i \leq n} \{i, D[i] \neq 0\}$$

Isso pode ser efetuado usando árvore binária em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Caso $max \neq 0$, podemos estender o caminho $CM[1], \dots, CM[max]$ a partir da exploração da nova adjacência de $CM[max]$ e tentar obter um novo ciclo elementar.

Os vértices a partir de max serão retirados de CM . Antes é necessário marcar ou desmarcar os vértices $CM[t], t > max$ em função de termos encontrado um novo ciclo.

O Passo 5 pode ser efetuado em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Passo 6

Neste passo, vamos desmarcar ou marcar os vértices do ciclo determinado. No caso em que o ciclo computado seja um novo ciclo, vamos desmarcar os vértices do ciclo elementar e os vértices v tal que exista um caminho $z_k, z_{k-1}, \dots, z_1, (z_k = v)$ tal que $z_{i+1} \in LB[z_i], k < i \leq 1$, e z_1 pertença ao ciclo elementar determinado.

O procedimento $CICLO(u, v)$ marca o vértice v na lista $ADJ[u]$ e inclui o vértice u na lista $LB[v]$. Esse procedimento é utilizado sempre que a exploração da aresta (u, v) não levou a um novo ciclo.

procedimento $CICLO(u, v)$

início

inserir u em $LB[v]$

marcar v em $ADJ[u]$

fim

O procedimento $CICLO$ pode ser efetuado usando duplicação recursiva em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

O procedimento $DESMARCAR(v)$ desmarca o vértice v em VM e todos os vértices u tal que exista um caminho $u = z_k, z_{k-1}, \dots, z_1 = v$ tal que $z_{i+1} \in LB[z_i], k < i \leq 1$. Além disso, caso exista algum $v_i \in LB[v]$, o procedimento

desmarca v da lista $ADJ[v_i]$, e se v_i é um vértice marcado, o procedimento é aplicado a v_i . Após isso, o procedimento reinicializa a lista $LB[v]$. Esse procedimento é utilizado quando o vértice v vai ser retirado de CM e v pertence a um novo ciclo elementar que foi enumerado.

procedimento $DESMARCAR(v)$

inicio

$VM[v] \leftarrow \text{falso}$

para todo $v_i \in LB[v]$ **em paralelo faça**

desmarcar v em $ADJ[v_i]$

se $VM[v_i]$ **então** $DESMARCAR[v_i]$

reinicializar $LB[v]$

fim

O procedimento $DESMARCAR$ pode ser efetuado usando duplicação recursiva em tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM.

Vamos agora desmarcar ou marcar os vértices que serão retirados de CM .

$m \leftarrow \max$

para todo $t, m \leq t \leq vf$ **em paralelo faça**

se $NC[(CM[t], CM[t+1])]$ **então**

$CICLO(CM[t], CM[t+1])$

para todo $t, m < t \leq vf$ **em paralelo faça**

se $NC[(CM[t-1], CM[t])]$ **então**

$DESMARCAR(CM[t])$

As operações acima podem ser efetuadas em tempo paralelo $O(\log n)$ com m processadores em uma CREW PRAM.

Agora, como no algoritmo anterior, caso $\max \neq 0$, atualizamos CM e alteramos os sucessores dos vértices que não pertencem a $\{v_0, \dots, v_{\max-1}\}$. Os sucessores serão tomados entre os vértices de ADJ que não estão marcados. Além disso, atualizamos todas as estruturas de dados usadas.

Com a alteração dos sucessores e a desmarcação das arestas, obtemos uma nova decomposição das arestas desmarcadas do digrafo em um conjunto de kl -caminhos.

O Passo 6 pode ser efetuado em tempo paralelo $O(\log^2 n)$ com n^3 processadores em uma CREW PRAM.

Passo 7

Usando as arestas desmarcadas de $BEDGE$, definimos uma nova raiz $r = v_k$ e aplicamos os passos 2, 3, 4, 5 e 6 até que todos os possíveis ciclos elementares de G com raiz $v = CM[1]$ sejam determinados.

O primeiro ciclo elementar é computado em tempo paralelo $O(\log^2 n)$ com n processadores. Em função da desmarcação das arestas, a computação dos demais ciclos pode utilizar um número maior de processadores. Isso faz com que os demais ciclos sejam computado em tempo paralelo $O(\log^2 n)$ com n^3 processadores.

Temos que o algoritmo só marca os vértices de um ciclo C quando C for computado novamente. Pode ocorrer, em situações muito particulares, de termos que marcar um número $O(n)$ de vértices antes de obtermos um novo ciclo e esses vértices estarem distribuídos por um número polinomial de ciclos.

Logo a complexidade total do algoritmo paralelo de determinação de todos os ciclos elementares de um digrafo, em alguns casos particulares, é de tempo paralelo $O(\alpha n \log^2 n)$ com n^3 processadores em uma CREW PRAM.

8.5 Correção e Análise

Lema 8.5.1 *Seja $G = (V, E)$ um digrafo fortemente conexo, então todo kl -caminho de G contém em um ciclo.*

Demonstração: Consideremos o caminho simples maximal $C = \{v_1, \dots, v_k\}$ em um kl -caminho de G . Pelo fato de que G é fortemente conexo existe $suc[v_k]$. Como C é um caminho maximal no kl -caminho, $suc[v_k] \in C$. \square

Lema 8.5.2 *Seja G um digrafo fortemente conexo. Dado um ciclo elementar C em G , existe um kl -caminho de G contendo C .*

Demonstração: Seja $C = \{v_i, \dots, v_k, v_i\}$ um ciclo elementar de G . Temos que existe um kl -caminho contendo C , pois basta definir os sucessores dos vértices v_j pertencentes ao kl -caminho da mesma forma como aparecem no ciclo. \square

Teorema 8.5.1 *Todo ciclo elementar do digrafo G será encontrado pelos algoritmos paralelos para determinação dos ciclos elementares.*

Demonstração: Temos que o algoritmo gera todos os possíveis kl -caminhos a partir de um vértice raiz r . Como todo kl -caminho contém um ciclo elementar, todos os ciclos elementares são determinados. \square

Teorema 8.5.2 *Nenhum ciclo elementar será enumerado mais de uma vez pelos algoritmos paralelos para determinação de ciclos elementares.*

Demonstração: Pelo fato de que um ciclo só é enumerado se pelo menos um de seus elementos nunca foi retirado do caminho CM , temos que nenhuma permutação de um ciclo elementar já computado será novamente enumerada. \square

Teorema 8.5.3 *O algoritmo paralelo computa corretamente todos os ciclos elementares de um digrafo em tempo paralelo $O(\alpha n \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde α é o número de ciclos elementares do digrafo.*

Capítulo 9

Conclusões

Apresentamos uma implementação simples do algoritmo paralelo para computação de um circuito de Euler de um grafo em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores em uma CREW PRAM. O algoritmo pode ser implementado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores, desde que os vértices do *grafo bipartido auxiliar* estejam convenientemente rotulados.

Utilizando esteio, apresentamos um algoritmo paralelo de simples implementação para a computação de uma árvore geradora e dos componentes conexos de um grafo. A implementação utiliza uma CREW PRAM. O algoritmo é executado em tempo paralelo $O(\log^2 n)$ com $\frac{m}{\log n}$ processadores. Se o grafo de entrada estiver rotulado convenientemente algoritmo é ótimo e pode ser executado em tempo paralelo $O(\log n)$ com $\frac{m}{\log n}$ processadores. Para a classe dos *st-grafos planares* e grafos *série-paralelo*, essa rotulação pode ser facilmente obtida em tempo $O(\log n)$ com $\frac{m}{\log n}$ processadores.

Acreditamos que outras classes de grafos possuem algoritmos paralelos ótimos. Para isso, basta que os vértices do grafo possam ser convenientemente rotulados, ou que possuam uma orientação com uma única fonte ou com um único sumidouro, e que a rotulação e a orientação possam ser efetuadas em tempo paralelo menor que $O(\log^2 n)$ em uma CREW PRAM.

Podemos também utilizar esteio para obter em tempo paralelo ótimo a coloração das arestas e vértices de uma árvore. Esse algoritmo pode ser utilizado na tentativa de obter a coloração das arestas e vértices de outras classes de grafos.

Uma variação de esteio pode ser utilizado na tentativa de obter um algoritmo paralelo para matching maximal em grafos bipartidos que não possuam o subgrafo $K_{3,3}$.

Descrevemos algoritmos paralelos para efetuar *busca irrestrita* em um grafo utilizando kl -caminho. A busca irrestrita em um grafo gera todos os caminhos maximais a partir do vértice r , raiz da busca. O problema do caminho maximal esta relacionado ao de busca em profundidade, pois qualquer *ramo* da árvore de busca em profundidade é um caminho maximal.

Para digrafos acíclicos, o algoritmo é executado em tempo paralelo $O(\alpha \log n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz. O algoritmo é implementado para grafos gerais em tempo paralelo $O(\alpha\beta \log^2 n)$ com m processadores em uma CREW PRAM, onde α é o número de caminhos maximais do grafo a partir de um vértice raiz, e β é o número máximo de ciclos do grafo que pode ocorrer na computação de um caminho maximal.

Continua em aberto a existência de um algoritmo NC para determinação de um caminho maximal de um grafo em geral. A utilização de kl -caminho, de tal forma que uma aresta não possa pertencer a mais de um kl -caminho, pode levar a obtenção de algoritmos NC para o problema do caminho maximal para outras classes de grafos.

Apresentamos um algoritmo para geração de todas as cliques maximais de um grafo círculo em tempo paralelo de $O(\alpha \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde n, m e α são o número de vértices, arestas e cliques maximais do grafo respectivamente. Apresentamos também um algoritmo paralelo para computar o número de cliques maximais α_k de tamanho k , o número total de cliques maximais α e uma clique máxima de um grafo círculo em tempo paralelo $O(\log^2 n)$ com n^3 , $nM(n)$ e n^3 processadores, respectivamente, em uma CREW PRAM, onde $M(n)$ é o número de processadores necessários para efetuar uma multiplicação de duas matrizes $n \times n$.

Esses algoritmos podem ser utilizados para outras classes de grafos, desde que o grafo possua uma orientação localmente transitiva.

Apresentamos dois algoritmos paralelos para determinação dos ciclos elementares de um digrafo, esses algoritmos são baseados nos algoritmos de Tiernan [84] e Szwarcfiter e Lauer [81] e utilizam a abordagem proposta por [81] de reportar os ciclos assim que forem determinados. Nos demais algoritmos os ciclos só são reportados a partir de um vértice inicial.

Os algoritmos determinam um ciclo elementar em tempo paralelo $O(\log^2 n)$ com m processadores em uma CREW PRAM. Para a determinação dos demais ciclos elementares, a implementação da versão de Szwarcfiter e Lauer [81] utiliza tempo paralelo $O(\alpha n \log^2 n)$ com n^3 processadores em uma CREW PRAM, onde α é o número de ciclos elementares do digrafo.

Na implementação dos algoritmos, apresentamos algumas técnicas para simular algumas das estruturas que são usadas nos algoritmos seqüenciais. A utilização de kl -caminhos de tal forma que uma aresta não possa pertencer a mais de um kl -caminho, auxiliada por uma outra estratégia de marcação dos vértices pode levar a um algoritmo para computação de todos os ciclos elementares de um digrafo em tempo paralelo polilogarítmico por ciclo.

Referências Bibliográficas

- [1] A. Aggarwal e R. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8:1–12, 1988.
- [2] M. Ajtai, J. Komlós, e E. Szemerédi. An $O(n \log n)$ sorting network. *Proceedings of the 18th ACM Annual Symposium on Theory of Computing*, pgs 1–9, 1983.
- [3] M. Ajtai, J. Komlós, e E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.
- [4] R. Anderson e Mayr. Parallelism and greedy algorithms. Technical Report STAN-CS-84-1003, Computer Science Department, Stanford University Bonn, 1984.
- [5] R.J. Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 4:315–326, 1987.
- [6] M. Atallah e U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and Systems Science*, 29:330–337, 1984.
- [7] B. Awerbuch e Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computer*, C-36:1258–1263, 1987.
- [8] B.Awerbuch, A.Israeli, e Y. Shiloach. Finding Euler circuits in logarithmic parallel time. *ACM Annual Symposium on Theory of Computing*, pgs 249–257, 1984.
- [9] A.T. Berztiss. *Data Structure: Theory and Practice*. Academic Press, 1971.
- [10] J.A. Bondy e U.S.R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.
- [11] A. Borondin. On relating time and space to size and depth. *SIAM J. Computing*, 6:733–744, 1977.
- [12] A. Bouchet. Reducing prime graphs and recognizing circle graphs. *Combinatorica*, 7:243–254, 1987.

- [13] R.P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21:201–206, 1974.
- [14] E.N. Cáceres. Árvore Geradoras e Componentes Conexos. *II ERMAC-RJ/ES*, Vitória, ES, 1992.
- [15] E.N. Cáceres, N. Deo, S. Sastry, e J.L. Szwarcfiter. Parallel algorithm for Euler tour. Technical report, University of Central Florida, 1990.
- [16] E.N. Cáceres, N. Deo, S. Sastry, e J.L. Szwarcfiter. On finding Euler tours. *Parallel Proc. Letters*, a ser publicado, 1992.
- [17] E.N. Cáceres e S. Sastry. Parallel algorithm for computing a spanning tree. Manuscript, University of Central Florida, 1990.
- [18] V. Carneiro e J.L. Szwarcfiter. *Algoritmos Paralelos*. Notas de Aula - COPPE, 1989.
- [19] A.K. Chandra, D.C. Kozen, e L.J. Stockmeyer. Alternation. *JACM*, 28:114–133, 1981.
- [20] F.Y. Chin, J. Lam, e I. Chen. Efficient parallel algorithms for some graph problems. *Comm. ACM*, 25:659–665, 1982.
- [21] R. Cole e U. Vishkin. Aproximate and exact parallel scheduling with applications to list, tree and graph problems. *Proceedings of the 27th Annual IEEE Symp. on Foundations of Comp. Sci.*, pgs 478–491, 1986.
- [22] R. Cole e U. Vishkin. Deterministic coin tossing with applications to optimal list ranking. *Information and Control*, 70:32–53, 1986.
- [23] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.
- [24] S. Cook e R. Reckhow. Time-bounded random access machines. *Journal of Computer Systems Science*, 7:354–375, 1973.
- [25] D. Coppersmith e S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9:251–280, 1990.
- [26] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Computing*, 5:618–623, 1976.
- [27] W.H. Cunningham. Decomposition of directed graphs. *SIAM J. Alg. and Disc. Methods*, 3:214–228, 1982.
- [28] W.H. Cunningham e J. Edmonds. A combinatorial decomposition theory. *Canad. J. Math.*, 32:734–765, 1980.

- [29] E. Dahlhaus e M. Karpinski. A fast parallel algorithm for computing all maximal cliques in a graph and the related problems. Technical Report 8516-CS, Institut fur Informatik der Universitat Bonn, Bonn, 1987.
- [30] N. Deo. *Graph Theory with applications to Engineering and Computer Science*. Prentice Hall, 1974.
- [31] D. Eppstein e Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annals Review Computer Science*, 3:233–238, 1988.
- [32] R.W. Floyd. Nondeterministic algorithms. *JACM*, 14:636–644, 1967.
- [33] S. Fortune e J. Wyllie. Parallelism in random access machines. *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pgs 114–118, 1978.
- [34] C.P. Gabor, W.L. Hsu, e K.J. Supowit. Recognizing circle graphs in polynomial time. *JACM*, 36:435–474, 1989.
- [35] Z. Galil. Sequential and parallel algorithms for finding maximum matching in graphs. *Annals Review Computer Science*, 1:197–224, 1986.
- [36] Z. Galil e V. Pan. Improved processor bounds for algebraic and combinatorial problems in *RNC*. *Proceedings of the 26th Annual IEEE Symp. on Foundations of Comp. Sci.*, pgs 490–495, 1985.
- [37] M.R. Garey, D.S. Johnson, G.L. Miller, e C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Alg. and Disc. Methods*, 1:216–228, 1981.
- [38] F. Gavril. Algorithms for maximum clique and independent set of a circle graph. *Networks*, 3:261–273, 1973.
- [39] R.K. Ghosh e G.P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:90–90, 1984.
- [40] A. Gibbons e W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
- [41] L.M. Goldschlager. A unified approach to models of synchronous parallel machines. *JACM*, 29:1073–1086, 1982.
- [42] V. Grolmusz e P. Radge. Incomparability in parallel computation. *Proceedings of the 28th Annual IEEE Symp. on Foundations of Comp. Sci.*, pgs 89–98, 1987.
- [43] T. Hagerup e H. Shen. Improved nonconservative sequential and parallel integer sorting. *Information Processing Letters*, 36:57–63, 1990.

- [44] X. He. Efficient parallel algorithms for series parallel graphs. *J. Algorithms*, 12:409–430, 1991.
- [45] W.D. Hillis e G.L. Steele. Data parallel algorithms. *Comm. ACM*, 29:1170–1183, 1986.
- [46] D.S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. *Proceedings of the 8th ACM Annual Symposium on Theory of Computing*, pgs 55–57, 1976.
- [47] D.S. Hirschberg, A.K. Chandra, e D.V. Sarwate. Computing connected components on parallel computers. *Comm. ACM*, 22:461–464, 1979.
- [48] D.B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Computing*, 4:77–84, 1975.
- [49] A. Karlin e E. Upfal. Parallel hashing - an efficient implementation of shared memory. *Proceedings of the 18th ACM Annual Symposium on Theory of Computing*, pgs 160–168, 1986.
- [50] R.M. Karp e V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science - Vol. A*, capítulo 17, pgs 869–941. The MIT Press/Elsevier, 1990.
- [51] R.M. Karp, E. Upfal, e A. Wigderson. Constructing a perfect matching is random NC . *Combinatorica*, 6:35–48, 1986.
- [52] G.A.P. Kindervater e J.K. Lenstra. An introduction to parallelism in combinatorial optimization. Technical Report OS-R8501, Centre for Mathematics and Computer Science, Amsterdam, 1985.
- [53] C.P. Kruskal, L. Rudolph, e M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.
- [54] L. Kucera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14:93–96, 1982.
- [55] R.E. Ladner e M.J. Fischer. Parallel prefix computation. *JACM*, 27:831–838, 1980.
- [56] K. Mehlhorn e U. Vishkin. Randomized and deterministic simulation on PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, 21:339–374, 1984.
- [57] J.W. Moon e L. Moser. On cliques in graphs. *Israel J. Math.*, 3:23–28, 1965.

- [58] K. Mulmuley, U.V. Vazirani, e V.V. Vazirani. Matching is as easy as matrix inversion. *Proceedings of the 19th ACM Annual Symposium on Theory of Computing*, pgs 345–354, 1987.
- [59] W. Naji. Graphes des cordes, caractérisation et reconnaissance. *Disc. Math.*, 54:329–337, 1985.
- [60] D. Nassimi e S. Sahni. Data broadcasting in SIMD computers. *IEEE Trans. Comput.*, 30:101–107, 1981.
- [61] D. Nath e N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14:7–11, 1982.
- [62] M.B. Novick. *Parallel algorithms for intersection graphs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1990.
- [63] C.H. Papadimitriou e J.D. Ullman. A communication-time tradeoffs. *SIAM J. Computing*, 16:639–646, 1987.
- [64] C.H. Papadimitriou e M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *Proceedings of the 20th ACM Annual Symposium on Theory of Computing*, pgs 510–513, 1988.
- [65] I. Parberry. *Parallel Complexity Theory*. Pitman Publishing, 1987.
- [66] M. Prabhaker e N. Deo. On algorithms for enumerating all circuits of a graph. Technical report, University of Illinois, 1973.
- [67] V.R. Pratt e L.J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and Systems Science*, 12:198–221, 1976.
- [68] M.J. Quinn. *Designing Efficient Algorithms*. McGraw-Hill, 1987.
- [69] S. Rajasekaran e J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18:594–607, 1989.
- [70] A. Ranade. How to emulate shared memory. *Proceedings of the 28th Annual IEEE Symp. on Foundations of Comp. Sci.*, pgs 185–194, 1987.
- [71] R.C. Read e R.E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.
- [72] C.C. Ribeiro. Parallel computer models and combinatorial algorithms. *Annals of Discrete Mathematics*, 31:325–364, 1987.
- [73] S.M. Roberts e B. Flores. Systematic generation of hamiltonian circuits. *Comm. ACM*, 9:690–694, 1966.

- [74] W.L. Ruzzo. On uniform circuit complexity. *Journal of Computer Systems Science*, 22:365–383, 1981.
- [75] W.J. Savitch e M.J. Stimson. Time bounded random access machines with parallel processing. *JACM*, 26:103–118, 1979.
- [76] Y. Shiloach e U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–63, 1982.
- [77] J.L. Szwarcfiter. *On optimal and near-optimal algorithms for some computational graph problems*. PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, England, 1975.
- [78] J.L. Szwarcfiter. *Grafos e Algoritmos Computacionais*. Editora Campus, 2a edição, 1986.
- [79] J.L. Szwarcfiter e M. Barroso. Enumerating the maximal cliques of circle graph. In F.R.K. Chung, R.L. Graham, and D.F. Hsu, editors, *Graph Theory, Combinatorics, Algorithms and Applications*, pgs 511–517. SIAM Publications, 1991.
- [80] J.L. Szwarcfiter e P.E. Lauer. A new backtracking search strategy for the enumeration of the elementary cycles of a directed graph. Technical Report Series 69, University of Newcastle upon Tyne, 1975.
- [81] J.L. Szwarcfiter e P.E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT*, 16:192–204, 1976.
- [82] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.
- [83] R.E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Computing*, 2:211–216, 1973.
- [84] J.C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Comm. ACM*, 13:722–726, 1970.
- [85] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Computing*, 8:410–421, 1979.
- [86] U. Vishkin. Implementation of simulations memory access machines in models that forbid it. *J. Algorithms*, 4:45–50, 1983.
- [87] U. Vishkin. A parallel-design distributed implementation (PDDI) general-purpose computer. *Theor. Comput. Sci.*, 32:157–172, 1984.
- [88] H. Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *JACM*, 19:43–56, 1972.

- [89] J.C. Wyllie. *The complexity of parallel computations*. PhD thesis, Computer Science Dept., Cornell Univ., Ithaca, NY, 1981.
- [90] Y. Zhang. *Parallel algorithms for problems involving directed graphs*. PhD thesis, Department of Computer Science, Drexel University, 1990.