O Problema da k-Cobertura por Vértices: uma Implementação FPT no Modelo BSP/CGM

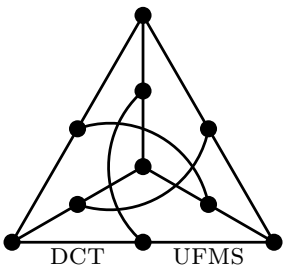
Erik Joey Hanashiro

Dissertação de Mestrado

Orientação:Prof. Dr. Henrique Mongelli

Área de Concentração: Ciência da Computação

Durante parte da elaboração desse trabalho o autor recebeu apoio financeiro da CAPES.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Março/2004

O Problema da k-Cobertura por Vértices: uma Implementação FPT no Modelo BSP/CGM

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Erik Joey Hanashiro
e aprovada pela comissão julgadora.

Campo Grande/MS, 15 de março de 2004.

Banca Examinadora:

- Prof. Dr. Henrique Mongelli (orientador) (DCT-UFMS)
- Prof. Dr. Siang Wun Song (IME-USP)
- Prof. Dr. Edson Norberto Cáceres (DCT-UFMS)

à minha esposa Priscilla
e aos meus pais

Agradecimentos

Ao meu orientador, professor Henrique Mongelli, pela dedicação, generosidade e paciência. Obrigado por ter acreditado em mim. Agradeço pelas palavras de ânimo nos momentos de preocupação e pelos sábios conselhos que me guiaram durante este período.

À minha esposa Priscilla, pelo apoio incondicional. Seu amor, confiança e compreensão foram fundamentais para a conclusão de mais esta etapa.

Aos meus pais, Eduardo e Paulina, eternos incentivadores dos meus estudos. Obrigado pelo carinho e pelo apoio.

Ao amigo Nalvo Franco de Almeida Junior, sempre disposto a ajudar. Obrigado pelo incentivo e pela confiança.

Ao professor Edson Norberto Cáceres que nunca mediu esforços para ajudar. Muito obrigado por intermediar os contatos com o professor Frank Dehne e o IC-Unicamp.

Ao professor Frank Dehne e a Peter J. Taillon, ambos da Carleton University, que gentilmente nos cederam os grafos de conflito para nossos experimentos.

Ao Instituto de Computação da Unicamp que permitiu a utilização do *cluster* Beowulf referente ao projeto da FAPESP No. 1997/10982-0.

Aos professores Marcelo Henriques de Carvalho, Ronaldo Alves Ferreira, Nahri Balesdent Moreano, Marcelo Ferreira Siqueira, Paulo Aristarco Pagliosa, Kátia Mara França da Silva e Renata Spolon, pelas oportunidades que me foram dadas no DCT durante este período.

Aos amigos Gonda, Amaury, Graziela, Luciana, Sérgio, Peralta, Guilherme e Tatiana, pela ajuda, trocas de idéias e palavras de estímulo.

A todos que trabalham no DCT/UFMS e àqueles que contribuíram, direta ou indiretamente, no desenvolvimento deste trabalho.

Resumo

Em muitas situações práticas precisamos resolver problemas NP-completos com exatidão. A Complexidade Parametrizada é um método promissor para lidarmos com a intratabilidade de alguns problemas, principalmente aqueles cuja entrada pode ser dividida em uma parte principal e um parâmetro. A parte principal da entrada contribui polinomialmente na complexidade total do problema, enquanto a aparentemente inevitável explosão combinatorial fica confinada ao parâmetro. Neste trabalho estudamos a teoria sobre Complexidade Parametrizada e o algoritmo FPT paralelo de Cheetham *et al.* no modelo BSP/CGM para o problema da k-Cobertura por Vértices, e nossa contribuição é apresentar uma implementação refinada e melhorada de tal algoritmo. A escolha de boas estruturas de dados e o uso da técnica de *backtracking* em nossa implementação foram fundamentais para que obtivessemos tempos paralelos melhores em nossos experimentos. Utilizamos cinco grafos de conflitos referentes a aminoácidos, os mesmos usados por Cheetham *et al.* em seus experimentos. Mesmo utilizando um ambiente computacional inferior ao usado por Cheetham *et al.*, para dois desses grafos obtivemos tempos aproximadamente 115 vezes melhores, para um deles 16 vezes melhor e, para os grafos restantes obtivemos tempos um pouco melhores. Além disso, para três desses grafos obtivemos coberturas por vértice de tamanho menor.

Conteúdo

Capítulo 1

Introdução

A abordagem da intratabilidade dos problemas é um grande desafio teórico e prático da Ciência da Computação. Como muitos problemas NP-completos têm grande aplicação prática, precisamos buscar formas de contornar a falta de um algoritmo de tempo eficiente que apresente soluções exatas para todas as instâncias do problema. Em situações práticas, o fato de um pesquisador descobrir que o problema analisado é NP-completo não elimina a necessidade existente por resultados. Alguns métodos, como a aproximação [22], análise de caso médio [23], randomização [27] e o uso de heurísticas [25] foram desenvolvidos com o objetivo de tentar superar a barreira da intratabilidade. A aproximação, por exemplo, sacrifica a exatidão da solução pela garantia de uma solução aproximada que seja computável eficientemente.

Neste trabalho apresentamos outro método que tenta lidar com a intratabilidade dos problemas, a Complexidade Parametrizada. Estamos interessados em problemas computacionais cuja entrada pode ser dividida em duas partes: a parte principal e o parâmetro [11]. Para simplificar, problemas cuja entrada pode ser dividida assim são ditos parametrizáveis. Muitos problemas computacionais são naturalmente especificados dessa forma. Por exemplo, na versão parametrizada do problema da Cobertura por Vértices, também conhecido como problema da k-Cobertura por Vértices, dado um grafo $G = (V, E)$ e um número inteiro k como entrada, queremos determinar se existe um subconjunto de no máximo k vértices em V, tal que qualquer aresta do grafo incide em pelo menos um vértice desse subconjunto. Nesse problema, a parte principal da entrada é um grafo e o parâmetro é um número inteiro.

Seja n o tamanho da parte principal da entrada e k o parâmetro, um problema parametrizável é tratável por parâmetro fixo ou FPT (do inglês *fixed-parameter tractable*) se existe um algoritmo que o resolve em tempo $O(f(k)n^{\alpha})$, onde α é uma constante e f é uma função arbitrária [14]. A parte principal da entrada contribui polinomialmente para a complexidade total do problema, enquanto a aparentemente inevitável explosão combinatorial fica confinada ao parâmetro. Essa definição só é aceitável se a constante α for pequena, tal como acontece na Complexidade Computacional, e o parâmetro k estiver em um pequeno, porém útil, intervalo. O problema da k-Cobertura por Vértices foi um dos primeiros problemas que provou-se ser tratável por parâmetro fixo e um dos algorit-

mos FPT mais conhecidos para resolvê-lo é o algoritmo de Balasubramanian *et al.* [1], de complexidade de tempo $O(kn + 1.324718^k k^2)$, onde n é o tamanho do grafo e k é o tamanho máximo desejado para a cobertura. O problema da k-Cobertura por Vértices é muito importante do ponto de vista prático. Por exemplo, em Biologia Computacional podemos aplicá-lo na análise de alinhamentos múltiplos de seqüências [6].

Existem duas técnicas comumente aplicadas no desenvolvimento de algoritmos FPT, a redução ao núcleo do problema e a árvore limitada de busca, as quais podem ser combinadas para resolver problemas tratáveis por parâmetro fixo [14]. Inicialmente, na fase de redução ao núcleo do problema a instância original é transformada em uma instância menor, porém equivalente, em tempo polinomial. O tamanho da instância resultante deve ser limitada em função do parâmetro. Depois, na fase de árvore limitada de busca analisamos exaustivamente uma árvore computada a partir da instância resultante da fase anterior, em busca de uma solução para o problema. O tamanho de tal árvore também deve ser limitado em função do parâmetro. Um problema é tratável por parâmetro fixo se e somente se é redutível ao núcleo do problema [17], sendo que o uso desta técnica resulta numa conexão aditiva entre as contribuições $f(k)$ e n^α, ou seja, $O(f(k) + n^\alpha)$, como acontece no algoritmo de Balasubramanian *et al.* [1].

Algoritmos FPT têm sido implementados e obtido sucesso na solução de instâncias de problemas NP-completos que antes eram consideradas muito grandes para serem resolvidas com os métodos já existentes. Entretanto, a complexidade exponencial do parâmetro ainda pode resultar em custos proibitivos. Neste trabalho mostramos como resolver instâncias ainda maiores do problema da k-Cobertura por Vértices usando o modelo paralelo BSP/CGM.

Uma máquina CGM (*Coarse-Grained Multicomputer*) [10] consiste de p processadores conectados por algum meio de interconexão, cada um com memória local de tamanho $O(N/p)$, onde N é o tamanho total do problema. Um algoritmo CGM é composto por rodadas bem definidas de computação local e comunicação, no qual o limite de memória local tem que ser respeitado e cada processador pode enviar e/ou receber no máximo $O(N/p)$ dados por rodada de comunicação.

Estudamos o algoritmo FPT paralelo de Cheetham *et al.* [6] no modelo BSP/CGM para o problema da k-Cobertura por Vértices, o qual paraleliza ambas as fases do algoritmo FPT seqüencial, redução ao núcleo do problema e árvore limitada de busca. Nossa contribuição é apresentar uma implementação refinada e melhorada deste algoritmo. Em nossos experimentos utilizamos cinco grafos de conflitos referentes a aminoácidos, os mesmos usados por Cheetham *et al.* [6] em seus experimentos. Mesmo utilizando um ambiente computacional inferior, nossos resultados foram melhores. Para dois desses grafos obtivemos tempos paralelos 115 vezes melhores, para um deles 16 vezes melhor e, para os dois grafos restantes obtivemos tempos paralelos um pouco melhores. Além disso, para três desses grafos encontramos coberturas por vértices de tamanho menor do que aqueles relatados por Cheetham *et al.* [6].

No Capítulo 2 apresentamos alguns conceitos fundamentais no desenvolvimento do nosso trabalho. Na seção sobre Complexidade Computacional mostramos as definições de problemas tratáveis e intratáveis, das classes de problemas P, NP e NP-completo,

além da descrição do problema da Cobertura por Vértices. Na seção sobre Complexidade Parametrizada temos o conceito de problemas tratáveis por parâmetro fixo e a descrição do problema da k-Cobertura por Vértices. Para finalizar, temos uma seção que descreve o modelo CGM mais detalhadamente.

No Capítulo 3 descrevemos algoritmos FPT que resolvem o problema da k-Cobertura por Vértices e que são utilizados em nossa implementação. Apresentamos o algoritmo FPT seqüencial de Buss [2] e os dois algoritmos FPT seqüenciais de Balasubramanian *et al.* [1]. Além disso, discutimos o algoritmo FPT paralelo de Cheetham *et al.* [6] no modelo BSP/CGM.

No Capítulo 4 discutimos a nossa implementação do algoritmo de Cheetham *et al.* [6] na linguagem C/C++ em conjunto com a biblioteca de comunicações MPI.

No Capítulo 5 temos os resultados experimentais obtidos pela nossa implementação e a comparação de tais resultados com os relatados por Cheetham *et al.* [6].

No Capítulo 6 apresentamos as conclusões e sugestões para trabalhos futuros.

Capítulo 2

Fundamentos

2.1 Introdução

Neste capítulo introduzimos alguns conceitos fundamentais para o desenvolvimento de algoritmos tratáveis por parâmetro fixo ou FPT no modelo paralelo BSP/CGM para resolver o problema NP-completo da k-Cobertura por Vértices. Na Seção 2.2 apresentamos os conceitos referentes à Complexidade Computacional, bem como a descrição do problema da Cobertura por Vértices. Na Seção 2.3 abordamos uma forma eficiente para lidar com a intratabilidade de alguns problemas, a Complexidade Parametrizada. Nesta seção ainda apresentamos a definição de problemas FPT e a descrição do problema da k-Cobertura por Vértices. Finalmente, na Seção 2.4 apresentamos o modelo realístico de computação paralela CGM, utilizado para desenvolver versões paralelas de algoritmos FPT.

2.2 Complexidade Computacional

A Complexidade Computacional é uma área da Ciência da Computação que estuda o grau de dificuldade inerente aos problemas que podem ser resolvidos computacionalmente, sendo que um de seus objetivos é analisar a eficiência dos algoritmos que resolvem esses problemas. Em geral, a métrica aplicada na avaliação de eficiência do algoritmo é o tempo de execução, medido pela quantidade de passos executados, em função do tamanho da entrada. Quanto menor o tempo gasto pelo algoritmo na solução do problema, maior é sua eficiência.

Um **algoritmo eficiente** é aquele que tem tempo de execução $O(p(n))$, onde $p(n)$ é um polinômio no tamanho da entrada [24]. Em outras palavras, o número de passos executados pelo algoritmo na solução do problema aumenta numa taxa polinomial em relação ao tamanho da entrada, razão pela qual os algoritmos eficientes também são chamados de **algoritmos de tempo polinomial**. No entanto, essa definição de eficiência também abrange algoritmos notoriamente não eficientes, por exemplo de complexidade $O(n^{100})$ ou

$O(10^{99}n)$. Porém, aceitamos a definição anterior porque, na prática, a complexidade da grande maioria dos algoritmos de tempo polinomial é limitada por polinômios de grau 2 ou 3, sem o envolvimento de coeficientes extremamente grandes [19]. Entre a minoria de algoritmos com tempo de execução $O(p(n))$ que não respeita o limite do grau quadrático ou cúbico, estão algoritmos de complexidade $O(n^k)$, do tipo "teste todos os subconjuntos de tamanho k", os quais não são considerados eficientes [14]. Os problemas que podem ser resolvidos por algoritmos eficientes são ditos **tratáveis** [8]. A noção de tratabilidade vem do fato que algoritmos de tempo polinomial podem solucionar seus problemas em tempo computacionalmente factível.

Infelizmente não existem algoritmos de tempo polinomial para resolver todos os problemas computacionais. Estes problemas são considerados **intratáveis**, pois, de tão difíceis, não conhecemos algoritmos eficientes para resolvê-los [19]. Além disso, grandes instâncias de problemas intratáveis podem gastar quantias de tempo computacionalmente inaceitáveis para serem resolvidas. Algoritmos cujo tempo de execução não pode ser limitado por polinômios no tamanho da entrada são chamados de **algoritmos de tempo exponencial** (apesar de tal definição incluir funções que normalmente não são consideradas exponenciais, como $n^{\log n}$).

Na Tabela 2.1, podemos observar a razão da grande aceitação dos algoritmos de tempo polinomial como modelos de eficiência. Quando aumentamos o tamanho da instância a ser resolvida, o tempo gasto por algoritmos de tempo polinomial sofre um crescimento de ordem polinomial. Como podemos verificar, esses tempos de processamento são computacionalmente aceitáveis. Por outro lado, o tempo de execução de algoritmos de tempo exponencial apresenta uma enorme taxa de crescimento quando aumentamos o tamanho das instâncias a serem resolvidas. Seriam necessários anos, décadas ou até mesmo séculos de processamento para que a solução fosse encontrada, o que torna esses algoritmos computacionalmente inviáveis. Chamamos esse fenômeno de **explosão combinatorial**, o qual aparenta ser inerente a problemas intratáveis.

Ademais, a Tabela 2.2 apresenta uma comparação entre o que ocorreria entre algoritmos de tempo polinomial e de tempo exponencial se acontecerem melhorias tecnológicas nas máquinas, considerando a maior instância que pode ser resolvida em 1 hora de processamento. Podemos observar que os algoritmos de tempo polinomial, sobretudo aqueles limitados por polinômios de grau baixo, aproveitam muito mais os efeitos de eventuais aumentos na velocidade de processamento dos computadores, visto que resolvem instâncias muito maiores. Já os algoritmos de tempo exponencial, na mesma 1 hora de processamento, resolvem instâncias pouco maiores, mesmo em computadores 100 ou 1000 vezes mais rápidos.

Entretanto, também existem exceções para a tal eficiência dos algoritmos de tempo polinomial em comparação com algoritmos de tempo exponencial. Para instâncias de tamanho pequeno, por exemplo, algoritmos de tempo exponencial podem ser mais rápidos do que algoritmos de tempo polinomial. Na comparação entre os tempos de execução das funções de complexidade n^5 e 2^n da Tabela 2.1, podemos notar que o algoritmo de tempo exponencial (2^n) é mais rápido que o algoritmo de tempo polinomial (n^5) para instâncias de tamanho $n \leq 20$.

	Tamanho da entrada (n)					
Função	10	20	30	40	50	60
n	.00001 segundo	.00002 segundo	.00003 segundo	.00004 segundo	.00005 segundo	.00006 segundo
n^2	.0001 segundo	.0004 segundo	.0009 segundo	.0016 segundo	.0025 segundo	.0036 segundo
n^3	.001 segundo	.008 segundo	.027 segundo	.064 segundo	.125 segundo	.216 segundo
n^5	.1 segundo	3.2 segundos	24.3 segundos	1.7 minutos	5.2 minutos	13.0 minutos
2^n	.001 segundo	1 segundo	17.9 minutos	12.7 dias	35.7 anos	366 séculos
3^n	.059 segundo	58 minutos	6.5 anos	3855 séculos	2×10^8 séculos	1.3×10^{13} séculos

Tabela 2.1: Comparação do tempo de execução entre algoritmos de tempo polinomial e de tempo exponencial para instâncias de tamanhos variados, supondo 1μs para uma operação. [19]

Além disso, existem casos raros de algoritmos de tempo exponencial que se mostram extremamente rápidos na prática, como o que resolve o método Simplex (Programação Linear), por exemplo. Situações como essa ocorrem porque a complexidade de tempo é uma medida de pior caso, ou seja, o fato da complexidade do algoritmo ser $O(2^n)$ apenas significa que pelo menos uma das possíveis instâncias de tamanho n precisa dessa quantia de tempo para ser resolvida. Portanto, pode ocorrer de muitas outras instâncias precisarem de bem menos tempo para serem resolvidas [19].

No estudo de Complexidade Computacional, agrupamos os problemas em classes, de acordo com sua complexidade. Por uma questão de simplicidade, consideramos apenas **problemas de decisão**, aqueles que podem ser respondidos com "sim" ou "não" [31]. É importante salientar que qualquer problema de outro tipo pode ser reescrito como um problema de decisão. Por exemplo, qualquer problema de otimização, aquele em que queremos saber qual é a melhor solução, possui um problema de decisão correspondente. Os problemas tratáveis formam a classe P, como definimos a seguir.

Definição 2.1 (Classe P [31]) *A classe de problemas P abrange os problemas de decisão que podem ser resolvidos por um algoritmo de tempo polinomial.*

Apesar de todos os esforços da comunidade científica, existem muitos problemas para os quais não conhecemos nenhum algoritmo de tempo polinomial. É claro que tais algoritmos ainda podem ser descobertos, mas há uma forte suspeita de que eles realmente não existam, dados os inúmeros insucessos em todos esses anos de estudo, mesmo com o empenho dos mais qualificados pesquisadores. Da mesma forma, até hoje ninguém foi capaz de provar que tais algoritmos não existem. Estes problemas formam a classe NP.

	Tamanho de instâncias resolvidas em 1 hora de processamento num computador		
Função	com velocidade atual	100 vezes mais rápido	1000 vezes mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Tabela 2.2: Comparação do tamanho de instâncias resolvidas em 1 hora de processamento caso aconteçam melhorias tecnológicas, para algoritmos de tempo polinomial e de tempo exponencial. [19]

Definição 2.2 (Classe NP) *A classe de problemas NP abrange problemas de decisão para os quais toda instância com resposta "sim" pode ser verificada em tempo polinomial.*

Para que um problema seja considerado da classe NP, não é necessário que todas as instâncias sejam resolvidas em tempo polinomial. Porém, qualquer instância desse problema com resposta "sim" deve possuir um algoritmo de verificação que se certifique da corretude da solução em tempo polinomial [31].

Existem duas relações entre as classes de problemas P e NP que merecem ser observadas:

- Note que P $\subseteq$ NP. Seja D um problema de decisão. Se $D \in$ P, então existe um algoritmo de tempo polinomial que soluciona D, e que pode ser utilizado para verificar a solução de uma instância com resposta "sim" do problema.
- Mas será que P = NP? Se quisermos provar que esta proposição é verdadeira, temos que mostrar que para todo problema pertencente à classe NP existe um algoritmo de tempo polinomial para resolvê-lo. Por outro lado, se quisermos provar que P $\neq$ NP, temos que mostrar um problema NP que não está em P. A maioria dos pesquisadores considera que P $\neq$ NP, ou seja, existem problemas em NP que não pertencem a P, mas até hoje ninguém conseguiu provar isso [24]. Esta questão é conhecida como problema P = NP e é um problema fundamental da Teoria da Computação.

Existe uma subclasse de problemas da classe NP que abrange os problemas computacionais mais difíceis de se resolver, chamada de classe de problemas NP-completo [8]. Para estes problemas devem haver provas formais da dificuldade intrínseca em resolvê-los. Em 1971, o pesquisador americano Stephen Cook foi o primeiro a provar formalmente que um problema pertence à classe NP-completo [19]. Esta demonstração, conhecida como Teorema de Cook, afirma que o problema da Satisfatibilidade é NP-completo. Através

de uma operação conhecida como redução polinomial, definida a seguir, podemos provar formalmente que outros problemas também são NP-completos.

Definição 2.3 (Redução Polinomial) *Dados dois problemas de decisão X e Y, X é polinomialmente redutível (ou transformado) para Y, denotado por $X \leq_P Y$, se para cada entrada I_X de X podemos construir em tempo polinomial uma entrada I_Y de Y, tal que I_X é uma instância de X com resposta "sim" se, e somente se, I_Y é uma instância de Y com resposta "sim".*

Se um problema X é polinomialmente redutível para um problema Y, significa que X não é mais difícil de resolver do que Y. Em outras palavras, Y é pelo menos tão difícil de resolver quanto X. Também sabemos que a relação de redução polinomial é transitiva [31]. Para provarmos formalmente que um problema faz parte da classe que abrange os problemas computacionais mais difíceis de se resolver (classe NP-completo), devemos mostrar que o problema satisfaz a seguinte definição.

Definição 2.4 (Problema NP-completo [31]) *Um problema D é dito NP-completo se:*

1. *$D \in NP$, e*
2. *Todos os outros problemas em NP são polinomialmente redutíveis (ou transformados) para D.*

Na prática, ao invés de mostrarmos que todos os problemas da classe NP são polinomialmente redutíveis para D (parte 2 da Definição 2.4), geralmente provamos que um reconhecido problema NP-completo é polinomialmente redutível para D [31]. Garey e Johnson [19] compilaram uma lista com vários problemas NP-completos, entre os quais podemos citar o problema do Circuito Hamiltoniano, Caixeiro Viajante, Mochila, Clique e Cobertura por Vértices.

A classe de problemas NP-completo é a maior responsável pelo fato dos pesquisadores considerarem que P $\neq$ NP. Uma importante propriedade desta classe define que se existir algoritmo de tempo polinomial para resolver qualquer um dos problemas NP-completo, então existem algoritmos de tempo polinomial para resolver todos os problemas da classe NP [8]. Portanto, se um algoritmo de tempo polinomial for encontrado para um problema NP-completo, a conseqüência imediata será P = NP. Grande parte da comunidade científica acredita que não existe algoritmo eficiente para resolver qualquer problema NP-completo, ou seja, as classes P e NP-completo são disjuntas, apesar de ninguém ter sido capaz de provar isso. A provável relação entre as classes P, NP e NP-completo está ilustrada na Figura 2.1.

Na próxima subseção, apresentamos o problema da Cobertura por Vértices, que faz parte de um grupo composto por seis problemas NP-completos considerados básicos por Garey e Johnson [19].

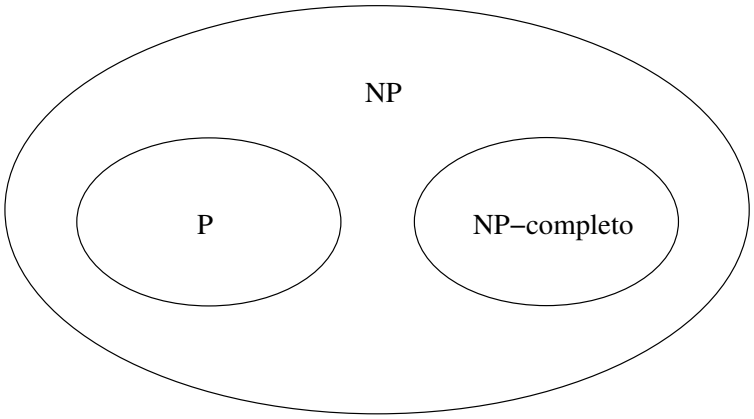


Figura 2.1: Provável relação entre as classes P, NP e NP-completo. NP-completo $\subseteq$ NP, P $\subseteq$ NP e P $\bigcap$ NP-completo $= \varnothing$. [8]

2.2.1 O Problema da Cobertura por Vértices

Uma cobertura por vértices para um grafo é um conjunto de vértices tal que, cada aresta do grafo incide em pelo menos um dos vértices desse conjunto, como ilustrado na Figura 2.2. A definição do problema da Cobertura por Vértices é apresentada a seguir.

Definição 2.5 (Problema da Cobertura por Vértices [19])
***Instância**: Grafo $G = (V, E)$, inteiro positivo $k \leq |V|$.*
***Questão**: Existe uma cobertura por vértices para G de tamanho menor ou igual a k, ou seja, um subconjunto $V' \subseteq V$, $|V'| \leq k$, tal que para cada aresta $(u, v) \in E$, pelo menos um dos vértices u ou v pertence a V'.*

Caso exista uma cobertura por vértices para o grafo G de tamanho menor ou igual a k, é importante salientar que esta não é, necessariamente, única, ou seja, podem existir outros subconjuntos de V com k ou menos vértices que satisfazem a condição de cobertura para o grafo.

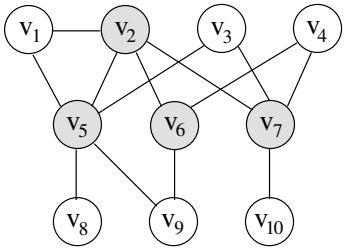


Figura 2.2: O conjunto $V' = \{v_2, v_5, v_6, v_7\}$ é uma das possíveis coberturas por vértice para o grafo, pois cada aresta incide em pelo menos um vértice da cobertura.

Uma aplicação prática para o problema da Cobertura por Vértices pode ser encontrada na área de Biologia Computacional, mais precisamente no alinhamento múltiplo

de seqüências [6]. Uma solução para resolver os conflitos entre as seqüências é excluir algumas destas seqüências da amostra. Existe um conflito quando o alinhamento de duas seqüências possui uma contagem abaixo de um limiar. Assim, definimos o que chamamos de grafo de conflitos, onde cada seqüência é um vértice e cada aresta é um conflito entre duas seqüências. Nosso objetivo é remover o menor número de seqüências que irá excluir todos os conflitos, ou seja, queremos encontrar uma cobertura por vértices mínima para o grafo de conflitos.

O melhor algoritmo conhecido para resolver o problema da Cobertura por Vértices é o algoritmo trivial da força bruta, o qual verifica todos os subconjuntos de vértices de tamanho menor ou igual a k em busca da solução [14], onde k é a quantidade máxima de vértices desejados na cobertura, n é a quantidade de vértices do grafo e $k \leq n$. Existem $C_{n,k}$ subconjuntos de vértices com k elementos e o algoritmo que encontra todos os subconjuntos de tamanho no máximo k gasta tempo $O(n^k)$. Para cada um dos subconjuntos encontrados, executamos um algoritmo de tempo polinomial que verifica se o mesmo forma ou não uma cobertura por vértices para o grafo. Este algoritmo de verificação simplesmente testa se todas as arestas do grafo incidem em vértices da suposta cobertura. Como temos no máximo $n.n$ arestas, o algoritmo de verificação gasta $O(n^2)$. Portanto, o tempo de execução total do algoritmo é $O(n^k n^2)$. Como vimos anteriormente, algoritmos de complexidade $O(n^k)$ não podem ser considerados eficientes.

Para os pesquisadores que têm problemas práticos para resolver, apenas saber que o problema é NP-completo não é suficiente, pois as soluções das instâncias continuam sendo necessárias. No entanto, como sabemos que é improvável a existência de um algoritmo eficiente que mostre soluções exatas para todas as instâncias do problema, redirecionamos nossos esforços na busca de formas de lidar com essa intratabilidade. Por exemplo, podemos procurar por algoritmos que resolvam eficientemente apenas algumas instâncias do problema, ou então, algoritmos que apresentam respostas próximas às soluções exatas do problema. Uma vez que muitos problemas NP-completos têm grande importância prática, um dos maiores desafios da Ciência da Computação é encontrar formas eficientes de superar a barreira da intratabilidade. Vários métodos foram desenvolvidos com esse intuito, dentre os quais podemos citar a aproximação [22], análise de caso médio [23], randomização [27] e o uso de heurísticas [25], cada um com suas vantagens e desvantagens. Todos, sem exceção, são obrigados a sacrificar algo para que esse objetivo seja atingido. Os algoritmos de aproximação, por exemplo, sacrificam a exatidão da solução, ou seja, a solução ótima, pela garantia de uma solução aproximada que seja computável eficientemente [8]. Uma outra tentativa de superar a barreira da intratabilidade é a Complexidade Parametrizada, que é o foco deste trabalho e será abordada na seção seguinte.

2.3 Complexidade Parametrizada

A Complexidade Parametrizada [11, 12, 13, 14, 15, 16, 17, 30] é mais uma tentativa de lidarmos com a intratabilidade dos problemas em alguns casos, a qual vem obtendo vários sucessos na solução de instâncias que eram muito grandes para serem resolvidas com os métodos já existentes [17]. Ao contrário da Complexidade Computacional clássica,

onde os problemas são especificados por dois itens (a entrada do problema e a questão a ser respondida), na Complexidade Parametrizada estamos interessados em problemas especificados por três itens:

1. A parte principal da entrada do problema (ou instância).
2. Alguns aspectos da entrada, que formam o parâmetro.
3. A questão a ser respondida.

Como podemos observar, a entrada desses problemas computacionais é dividida em duas partes: a parte principal e o parâmetro. Existem vários problemas computacionais que são naturalmente especificados dessa maneira [11]. Para exemplificar problemas desse tipo, apresentamos a versão parametrizada do problema NP-Completo da Cobertura por Vértices, também conhecido como problema da k-Cobertura por Vértices, onde a parte principal da entrada é um grafo e o parâmetro é um número inteiro. Muitos outros problemas da Teoria dos Grafos podem ser parametrizados de forma semelhante ao problema da k-Cobertura por Vértices.

Definição 2.6 (Problema da Cobertura por Vértices - Parametrizado [13])
Instância: *Grafo $G = (V, E)$.*
Parâmetro: *Inteiro positivo k.*
Questão: *Existe um conjunto de vértices $V' \subseteq V$, de tamanho máximo k, tal que para cada aresta $(u, v) \in E$, ou $u \in V'$ ou $v \in V'$?*

Pela Complexidade Computacional clássica (Seção 2.2) sabemos que o fenômeno da explosão combinatorial atinge toda a entrada de um problema intratável, tal como ilustrado na Figura 2.3(a). Na Complexidade Parametrizada, procuramos entender como as diferentes partes da entrada contribuem para a complexidade total do problema, sendo que gostaríamos de identificar as partes da entrada que causam a aparentemente inevitável explosão combinatorial. Falando em termos de complexidade, nosso objetivo é combinar o que chamamos de "bom" e "mau" comportamento. Por "bom" comportamento, entendemos que a parte principal da entrada contribui de forma polinomial para a complexidade total do problema. Por outro lado, o parâmetro provavelmente contribui de forma exponencial para a complexidade total do problema, provocando o "mau" comportamento. Assim, se conseguirmos fazer isso, problemas NP-completos podem ter algoritmos que gastam tempo exponencial em relação ao parâmetro, mas tempo polinomial em relação à parte principal da entrada. No entanto, precisamos confinar o parâmetro em um pequeno, porém útil, intervalo. Downey e Fellows [15] chamam essa combinação de "acordo com o demônio da intratabilidade", a qual é ilustrada na Figura 2.3(b). Em muitas aplicações, o parâmetro pode ser considerado "bem pequeno" quando comparado ao tamanho da parte principal da entrada.

As principais áreas de atuação da Complexidade Parametrizada são aquelas nas quais parâmetros pequenos são naturalmente úteis em aplicações práticas. Assim, se houver problemas NP-completos, podemos aplicar as técnicas de Complexidade Parametrizada

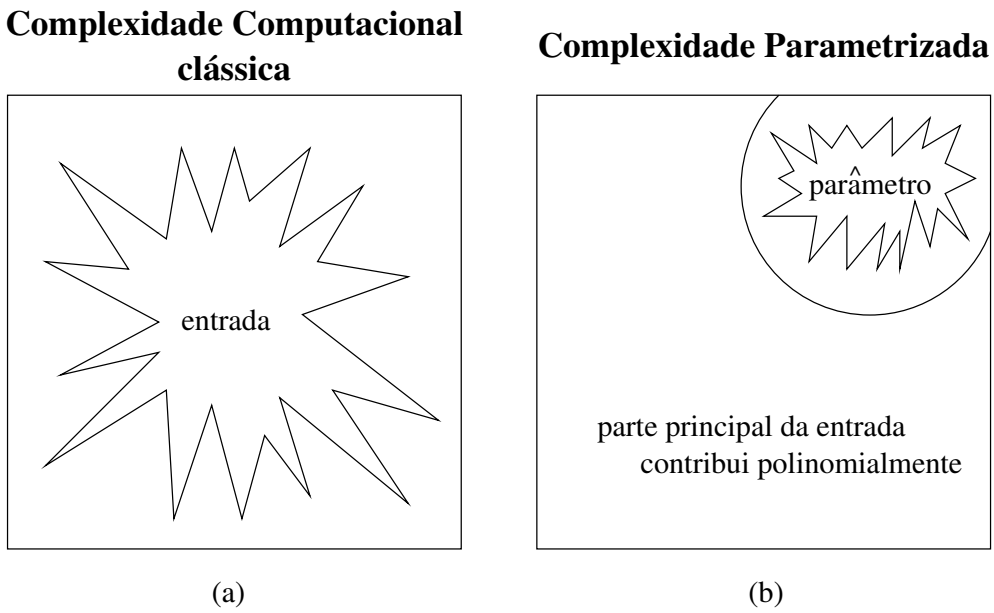


Figura 2.3: (a) A explosão combinatorial atinge toda a entrada do problema. (b) A explosão combinatorial fica confinada ao parâmetro, enquanto a parte principal da entrada contribui polinomialmente para a complexidade total do problema. [14]

para tentar superar a barreira provocada pela intratabilidade. Dentre essas áreas, podemos citar: Biologia Computacional, Projetos de Sistemas VLSI, Robótica, Inteligência Artificial e Planejamento de Redes [15].

A seguir, definimos alguns conceitos importantes da Complexidade Parametrizada. Por uma questão de simplicidade, tal como fizemos na Complexidade Computacional clássica, restringimos nosso interesse a problemas de decisão.

Definição 2.7 (Problema Parametrizável [11]) *Um problema parametrizável é um conjunto $L \subseteq \Sigma^* \times \Sigma^*$, onde Σ é um alfabeto fixo. Se o par $(x, y) \in L$, chamamos x de parte principal da entrada (ou instância) e y de parâmetro.*

Por conveniência, mas sem perder a generalidade da definição anterior, dizemos que um problema parametrizável é um conjunto $L \subseteq \Sigma^* \times \mathbb{N}^*$ [16], tal como acontece no problema da k-Cobertura por Vértices, onde a parte principal da entrada é um grafo e o parâmetro, um número inteiro que representa o tamanho máximo desejado para a cobertura.

A noção de problema tratável por parâmetro fixo, a qual apresentamos a seguir, é tão fundamental para a Complexidade Parametrizada quanto a noção de tempo polinomial é fundamental para a Complexidade Computacional clássica. A definição é baseada em Downey e Fellows [14].

Definição 2.8 (Problema Tratável por Parâmetro Fixo) *Um problema parametrizável $L \subseteq \Sigma^* \times \mathbb{N}^*$ é tratável por parâmetro fixo se existe um algoritmo que, dada uma*

entrada $(x, y) \in L$, resolve o problema em tempo $O(f(k)n^{\alpha})$, onde n é o tamanho da parte principal da entrada x, ou seja, $|x| = n$; k é o tamanho do parâmetro y, ou seja, $|y| = k$; α é uma constante (independente de k); e f é uma função arbitrária.

A função arbitrária $f(k)$ citada na definição anterior é a contribuição do parâmetro y para a complexidade total do problema. Provavelmente, esta contribuição é exponencial. Entretanto, a parte principal da entrada (x) contribui polinomialmente para a complexidade total do problema, ou seja, n^{α}. A assunção fundamental é que $k \ll n$ [30]. Da mesma forma que na Complexidade Computacional clássica, essa contribuição polinomial só é aceitável se a constante α for pequena. No entanto, a propriedade mais interessante e fundamental é que a definição de problema tratável por parâmetro fixo permanece inalterada se trocamos a conexão multiplicativa entre as duas contribuições, ou seja, $f(k)n^{\alpha}$, por uma conexão aditiva, isto é, $f(k) + n^{\alpha}$ [14].

Os problemas tratáveis por parâmetro fixo formam uma classe de problemas denominada FPT (*Fixed-Parameter Tractability*). O problema da k-Cobertura por Vértices foi um dos primeiros problemas que provou-se pertencer a essa classe. Um dos resultados mais conhecidos é o algoritmo de Balasubramanian *et al.* [1], de complexidade de tempo $O(kn + 1.324718^k k^2)$. Na Tabela 2.3, apresentamos uma relação com autores, complexidade de tempo e ano de criação de algoritmos FPT para o problema da k-Cobertura por Vértices, onde n é o tamanho do grafo e k o tamanho máximo desejado para a cobertura. Alguns deles são apresentados com maiores detalhes no Capítulo 3.

Autor(es)	**Complexidade de tempo**	**Ano**
Buss [2]	$O(kn + 2^k k^{2k+2})$	1993
Downey e Fellows [12]	$O(kn + 2^k k^2)$	1995
Balasubramanian *et al.* [1] (Alg. B1)	$O(kn + (\sqrt{3})^k k^2)$	1998
Balasubramanian *et al.* [1] (Alg. B2)	$O(kn + 1.324718^k k^2)$	1998
Downey, Fellows e Stege [17]	$O(kn + 1.31951^k k^2)$	1999
Niedermeider e Rossmanith [29]	$O(kn + 1.29175^k k^2)$	1999
Fellows e Stege [18]	$O(kn + max(1.25542^k k^2, 1.2906^k 2.5^k))$	1999
Chen, Jia e Kanj [7]	$O(kn + 1.271^k k^2)$	2001

Tabela 2.3: Relação dos algoritmos FPT para o problema da k-Cobertura por Vértices

Os algoritmos FPT têm tido sucesso na solução de problemas NP-completos para certas instâncias de enorme importância prática. Antes desses algoritmos, essas instâncias eram muito grandes para serem resolvidas pelos métodos existentes [6].

Um importante item para a comparação do desempenho de algoritmos FPT é o tamanho máximo que o parâmetro k pode atingir, sem afetar a desejada eficiência do algoritmo. Esse valor é denominado ***klam*** e definido como sendo o maior valor de k tal que $f(k) \leq U$, onde U é algum limite absoluto do número de passos computacionais. Downey e Fellows [14] sugerem $U = 10^{20}$. Um desafio existente nos problemas tratáveis por parâmetro fixo é o desenvolvimento de algoritmos FPT com valores *klam* cada vez

maiores. Na Tabela 2.4, podemos notar a diferença de desempenho entre os dois algoritmos apresentados em Balasubramanian *et al.* [1]. O Algoritmo B2 melhora muito o processo, quase dobrando o valor *klam* em relação ao Algoritmo B1.

Algoritmo	**Complexidade de tempo**	**Valor do *klam***
B1	$O(kn + (\sqrt{3})^k k^2)$	68
B2	$O(kn + 1.324718^k k^2)$	129

Tabela 2.4: Valor *klam* do Algoritmo B1 e do Algoritmo B2 de Balasubramanian *et al.* [1].

Agora que já discutimos o que é um problema tratável por parâmetro fixo e aprendemos como comparar algoritmos FPT, apresentamos dois métodos considerados elementares no desenvolvimento de algoritmos para problemas tratáveis por parâmetro fixo: redução ao núcleo do problema e árvore limitada de busca. A aplicação desses métodos, nessa ordem, como um algoritmo de duas fases, é a base de vários algoritmos FPT. Apesar de serem frutos de estratégias algorítmicas simples, essas técnicas nem sempre são lembradas de imediato, pois envolvem custos exponenciais em relação ao parâmetro [15]. As explicações sobre essas técnicas são baseadas em Downey e Fellows [14].

- **Redução ao núcleo do problema** (*reduction to a problem kernel*): O objetivo é reduzir, em tempo polinomial, uma instância I do problema parametrizável em outra instância equivalente I', cujo tamanho deve ser limitado por uma função do parâmetro k. Se uma solução de I' for encontrada, provavelmente após uma análise exaustiva da instância gerada, esta solução pode ser transformada em uma solução de I. O uso dessa técnica sempre resulta em uma conexão aditiva entre as contribuições n^α e $f(k)$ da complexidade total. Downey, Fellows e Stege [17] mostraram que um problema é tratável por parâmetro fixo se e somente se é redutível ao seu núcleo. O uso do método de redução ao núcleo do problema é exemplificado pelo algoritmo FPT de Buss [2] para a k-Cobertura por Vértices, descrito na Subseção 3.2.

- **Árvore limitada de busca** (*bounded search tree*): Essa técnica computa uma árvore de tamanho exponencial, talvez de forma ineficiente, cujo tamanho deve ser limitado em função do parâmetro k. O nó raiz da árvore limitada geralmente armazena a instância resultante da fase de redução ao núcleo do problema. Para construir a árvore limitada, esta técnica executa em cada nó da árvore algum algoritmo relativamente eficiente na instância armazenada e ramifica tal nó. Analisamos exaustivamente a árvore limitada em busca de uma solução para o problema. Comumente fazemos uma busca em profundidade na árvore limitada. No pior caso temos que percorrer toda a árvore limitada de busca para determinar se existe ou não uma solução para o problema. No caso de muitos problemas parametrizáveis, como o tamanho da árvore de busca é limitado em função do parâmetro k, para um k fixo o espaço de busca torna-se constante, e o algoritmo passa a ser eficiente para esse k. Os algoritmos FPT de Balasubramanian *et al.* [1] para a k-Cobertura por Vértices, descritos na Seção 3.3, aplicam o método de árvore limitada de busca.

Em Downey e Fellows [14], podemos encontrar uma lista com outros problemas FPT, além do problema da k-Cobertura por Vértices. Apesar de quase metade dos problemas catalogados como NP-completos em Garey e Johnson [19] serem tratáveis por parâmetro fixo, existem muitos outros problemas parametrizáveis que provavelmente não pertencem à classe FPT. Para ilustrar esse fato, considere o seguinte problema parametrizável:

Definição 2.9 (Problema do Conjunto Dominante - Parametrizado [13])
Instância: *Grafo $G = (V, E)$.*
Parâmetro: *Inteiro positivo k.*
Questão: *Existe um conjunto de vértices $V' \subseteq V$, de tamanho máximo k, tal que para cada vértice $u \in V$, existe uma aresta $(u, v) \in E$ para algum vértice $v \in V'$?*

No caso do problema NP-completo do k-Conjunto Dominante [19], por exemplo, o melhor algoritmo para resolvê-lo ainda é o da "força bruta", aquele que testa todos os subconjuntos de tamanho k. Segundo Downey e Fellows [14], aparentemente não há como limitar a explosão combinatorial apenas em termos do parâmetro k. Para abranger os problemas aparentemente intratáveis por parâmetro fixo, existe uma hierarquia de classes parametrizadas definidas como

$$\text{FPT} \subseteq W[1] \subseteq W[2] ... \subseteq W[P]$$

baseadas no grau de dificuldade em resolver esses problemas segundo a teoria de Complexidade Parametrizada. Downey, Fellows e Stege [17] afirmam que a classe $W[1]$ é análoga à classe de problemas NP da Complexidade Computacional clássica, e que a dificuldade característica em $W[1]$ é a evidência básica de que o problema provavelmente não é FPT. Em tempo, o problema do k-Conjunto Dominante é $W[2]$-*completo* [14], portanto o problema aparentemente não é tratável por parâmetro fixo.

2.4 Modelo CGM

No início dos anos 90, Valiant [33] introduziu o conceito de que a computação paralela poderia ser modelada como uma série de **superpassos** em vez de passos individuais de troca de mensagens ou acessos à memória compartilhada. Um superpasso é uma combinação de passos de computação local e passos de comunicação global, no qual os passos de computação utilizam dados disponibilizados localmente no início do superpasso e os passos de comunicação são feitos através de instruções de envio e recebimento de mensagens. Nessa seção, descrevemos dois modelos que seguem esse conceito, o **modelo BSP** (*Bulk Synchronous Parallel*) e o **modelo CGM** (*Coarse Grained Multicomputer*), os quais fornecem uma previsão razoável do desempenho de seus algoritmos quando implementados nas máquinas paralelas existentes. Por essa razão, tais modelos são denominados **modelos realísticos** [3].

O modelo BSP, proposto por Valiant [33] em 1990, tornou a computação paralela

de **granularidade grossa**[1] e levou a uma redução substancial no *overhead* de sincronização [9]. Além disso, foi um dos primeiros modelos a considerar os custos de comunicação como característica de um modelo computacional de computação paralela. Uma máquina BSP consiste de um conjunto de p processadores com memória local, cuja comunicação é feita através de algum meio de interconexão, gerenciados por um roteador que possibilite a troca de mensagens ponto-a-ponto entre pares de processadores, e com facilidades de sincronização global. Um algoritmo BSP consiste de uma seqüência de superpassos, os quais são separados por barreiras de sincronização, como podemos observar na Figura 2.4. Neste modelo, uma h**-relação** em um processador corresponde ao envio e/ou recebimento de no máximo h mensagens em cada processador [26]. As mensagens enviadas só estarão disponíveis para o recebedor no próximo superpasso.

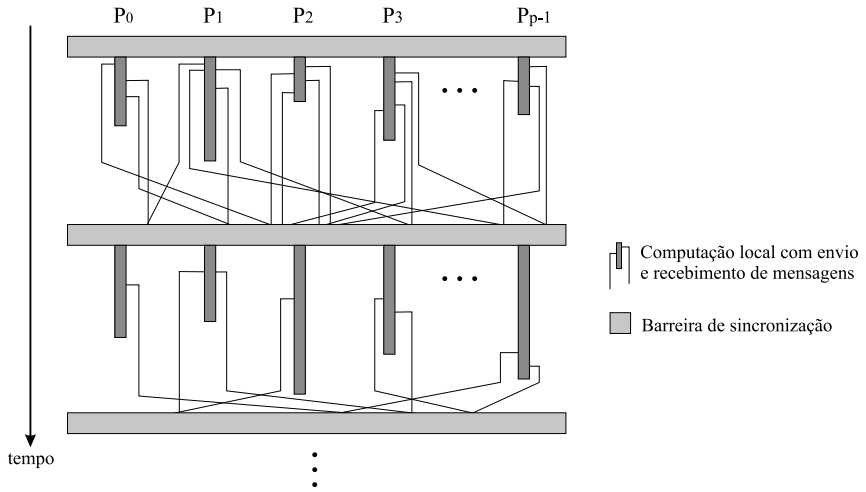


Figura 2.4: Algoritmo BSP [20]

Em 1993, Dehne *et al.* [10] propuseram o modelo CGM, o qual é definido apenas com dois parâmetros: n (tamanho do problema) e p (número de processadores). O termo *coarse grained* (granularidade grossa) aplicado ao nome do modelo vem do fato que o tamanho do problema é consideravelmente maior que o número de processadores, ou seja, $n/p \gg p$ [3]. O modelo CGM é caracterizado pela adição da restrição de comunicação de granularidade grossa ao modelo BSP, portanto um algoritmo CGM é um caso especial de um algoritmo BSP, onde todas as operações de comunicação de um superpasso são feitas em uma h**-relação** com $h \leq n/p$.

Uma máquina CGM consiste de p processadores com memória local de tamanho $O(n/p)$, conectados por algum meio de interconexão. O tamanho da memória local de cada processador é tipicamente maior que $O(1)$, pois $n/p \geq p$ [10]. Podemos ver uma representação do modelo CGM na Figura 2.5.

Um algoritmo CGM consiste de rodadas (*rounds*) alternadas de computação local dos processadores e de comunicação global, as quais são separadas por barreiras de sincronização, como podemos observar na Figura 2.6. Nas rodadas de computação local,

[1] A granularidade (nível de paralelismo) está relacionada com o tamanho dos processos (subtarefas) executados pelos processadores, e é classificada grossa quando cada um desses processos contém um grande número de instruções seqüenciais e gasta um tempo considerável para executá-las [34].

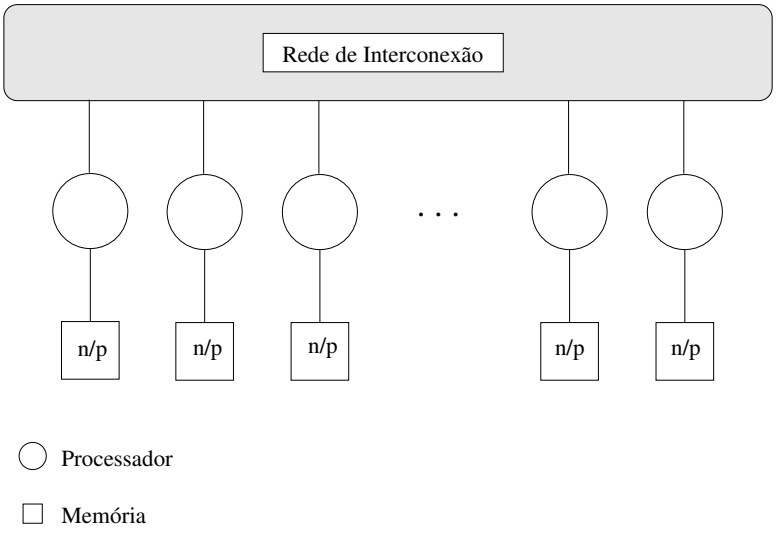


Figura 2.5: Representação do modelo CGM para p processadores, considerando entrada de tamanho n. [26]

normalmente aplicamos o melhor algoritmo seqüencial para o processamento dos dados armazenados localmente. Nas rodadas de comunicação global, cada processador pode enviar e/ou receber, no máximo, $O(n/p)$ dados. O tempo de um algoritmo CGM é a soma dos tempos de suas rodadas de computação local e comunicação global.

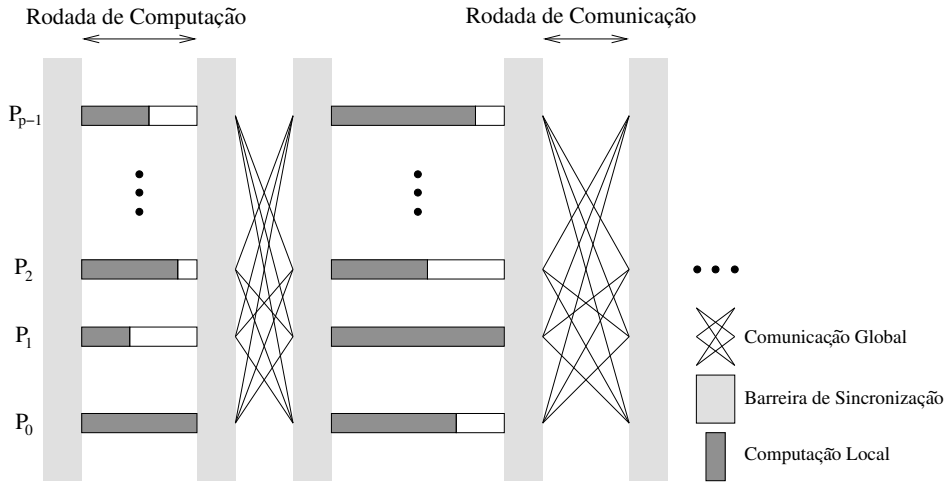


Figura 2.6: Algoritmo CGM [4]

Os algoritmos CGM são bastante adequados para as máquinas paralelas atuais, as quais possuem velocidade global de computação bem superior à velocidade global de comunicação, sendo que o desafio no desenvolvimento desses algoritmos é minimizar o número de rodadas de comunicação [28]. Uma das vantagens do modelo CGM é que seus algoritmos fornecem um prognóstico de desempenho bastante realista quando implementados em multiprocessadores atualmente disponíveis [9].

2.5 Notas

Neste capítulo, apresentamos alguns conceitos da Complexidade Computacional, como problemas tratáveis e intratáveis, e as classes de problemas P, NP e NP-completo. Definimos também o problema da Cobertura por Vértices e mostramos uma forma para lidar com sua intratabilidade: a Complexidade Parametrizada. Vimos que o problema da k-Cobertura por Vértices é tratável por parâmetro fixo ou FPT. Para finalizar, introduzimos o modelo CGM, utilizado no desenvolvimento das versões paralelas de algoritmos FPT que resolvem o problema da k-Cobertura por Vértices.

Capítulo 3

Algoritmos FPT para a k-Cobertura por Vértices

3.1 Introdução

Neste capítulo apresentamos algoritmos FPT que resolvem o problema da k-Cobertura por Vértices e que são utilizados em nossas implementações. Estes algoritmos recebem como entrada um grafo G com n vértices e um inteiro positivo k.

Na Seção 3.2 apresentamos o algoritmo FPT seqüencial de Buss [2], o qual inspira a fase de redução ao núcleo do problema em nossa implementação paralela. Na Seção 3.3 descrevemos os algoritmos FPT seqüenciais de Balasubramanian *et al.* [1], que apresentam duas opções diferentes para a fase de árvore limitada de busca. Por fim, mostramos na Seção 3.4 o algoritmo FPT paralelo de Cheetham *et al.* [6] no modelo BSP/CGM, o qual implementamos neste trabalho.

3.2 Algoritmo de Buss

O algoritmo de Buss [2] apresenta como principal contribuição a descrição de um método de redução ao núcleo do problema que remove os vértices de grau maior que k do grafo G, bem como as arestas neles incidentes e quaisquer vértices isolados.

ALGORITMO: Buss
Entrada: o grafo $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho no máximo k, se existir; ou uma mensagem, se tal cobertura por vértices não existir.

1. Seja H o conjunto de vértices em V de grau maior que k.
2. **se** $|H| > k$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
3. Seja $G' = (V', E')$ o grafo resultante da remoção dos vértices de G que estão em

H, bem como das arestas neles incidentes e de qualquer vértice isolado.
4. $k' \leftarrow k - |H|$
5. **se** $|E'| > k.k'$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
6. Encontre todos os subconjuntos de vértices de G' de tamanho menor ou igual a k'.
7. **se** algum desses subconjuntos forma uma cobertura por vértices para G'
   devolva os vértices desse subconjunto mais os vértices de H.
   senão
   devolva "Não existe a cobertura por vértices desejada"

fim algoritmo

Nos primeiros cinco passos do algoritmo temos a definição de um método de redução ao núcleo do problema, ou seja, reduzimos o grafo G em um grafo G' equivalente em tempo polinomial, sendo que o tamanho de G' é limitado por uma função do parâmetro k. O algoritmo baseia-se na idéia de que todos os vértices do grafo G de grau maior que k (conjunto H) pertencem a qualquer cobertura por vértices para G de tamanho menor ou igual a k. Portanto, se houver mais do que k vértices nessa situação, uma cobertura por vértices para G de tamanho no máximo k não é possível, conforme o Passo 2.

No Passo 3 geramos o grafo G' a partir da remoção dos vértices de G que pertencem ao conjunto H, bem como das arestas neles incidentes e de qualquer vértice isolado. Note que as arestas incidentes nos vértices do conjunto H são removidas porque já estão ligadas a pelo menos um vértice da cobertura, e os vértices isolados são removidos porque não têm arestas a cobrir.

Como k' é a quantidade de vértices que falta para completar o tamanho máximo da cobertura para o grafo G, o objetivo a partir de agora é encontrar uma cobertura por vértices para o grafo G' de tamanho menor ou igual a k'. Se tal cobertura existir, esses vértices mais aqueles do conjunto H formam uma cobertura por vértices para o grafo G de tamanho menor ou igual a k. No Passo 5 verificamos se é possível encontrar uma cobertura para G', pois k' vértices podem cobrir no máximo $k.k'$ arestas.

Se não temos mais do que $k.k'$ arestas em G' e não existem vértices isolados, podemos concluir que existem no máximo $2k.k'$ vértices em G'. Como k' é no máximo k, o tamanho do grafo G' é $O(k^2)$. Assim o tamanho do grafo gerado pelo método de redução ao núcleo do problema realmente é limitado por uma função do parâmetro k. Assumindo que o grafo G é dado por uma lista de adjacências, esses cinco primeiros passos gastam tempo $O(kn)$. Além disso, esses passos constituem o método de redução ao núcleo do problema que será aplicado nas fases correspondentes dos algoritmos de Balasubramanian *et al.* [1], descritos na Seção 3.3.

Do Passo 6 em diante aplicamos um algoritmo de força bruta para determinar se existe ou não uma cobertura por vértices para G' de tamanho menor ou igual a k', ou seja, testamos todos os subconjuntos de V' de tamanho menor ou igual a k'. Este algoritmo gasta tempo $O(|V'|^{k'})$ para encontrar todos os subconjuntos de vértices de tamanho menor ou igual a k'. Já vimos que o número de vértices em G' é $2k^2$ e que k' é no máximo k, então gasta-se tempo $O((2k^2)^k)$. Para cada um desses subconjuntos de vértices, o algoritmo verifica se cada aresta de G' incide em pelo menos um vértice da suposta cobertura. Como

são no máximo $k.k'$ arestas, gasta-se tempo $O(k^2)$. Portanto, o algoritmo de força bruta gasta tempo $O((2k^2)^k k^2)$. Devemos lembrar que se houver uma cobertura por vértices para G' de tamanho menor ou igual a k', os vértices dessa cobertura mais os vértices do conjunto H formam uma cobertura por vértices para o grafo G de tamanho menor ou igual a k.

O tempo total do algoritmo é o tempo gasto no método de redução ao núcleo do problema mais o tempo gasto no algoritmo de força bruta, ou seja, $O(kn + (2k^2)^k k^2)$.

Exemplo de Execução do Algoritmo

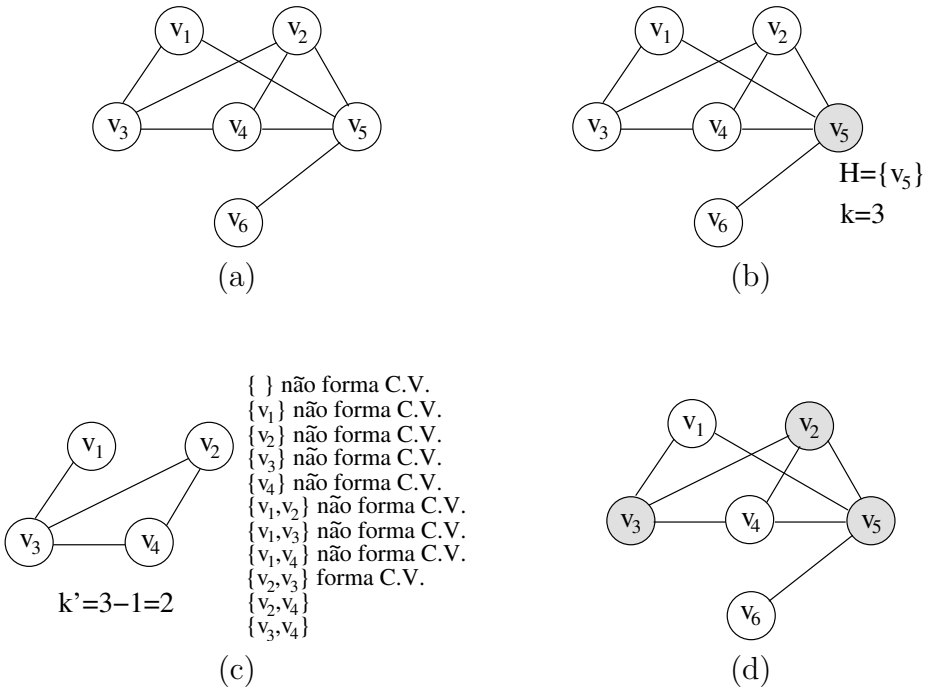


Figura 3.1: Execução do algoritmo de Buss. (a) Grafo G. (b) Seleção dos vértices de grau maior que k. (c) Grafo G'. (d) Cobertura por vértices para G de tamanho menor ou igual a k.

Na Figura 3.1 mostramos um exemplo de execução do algoritmo de Buss [2]. Seja o grafo G apresentado na Figura 3.1(a), queremos determinar se existe uma cobertura por vértices para o grafo G de tamanho menor ou igual a 3, ou seja, $k = 3$. Pela Figura 3.1(b), temos $H = \{v_5\}$ pois este é o único vértice de grau maior que 3 em G. Como só temos um elemento em H e a cobertura por vértices que procuramos é de tamanho menor ou igual a 3, continuamos a execução do algoritmo.

O grafo G' apresentado na Figura 3.1(c) resulta da remoção do vértice de G que está em H, bem como das arestas nele incidentes e de qualquer vértice isolado. Note que o vértice v_6 foi removido porque estava isolado. Daqui em diante, o objetivo é encontrar uma cobertura por vértices para G' de tamanho 2, pois $k' = k - |H| = 3 - 1 = 2$. Como temos menos do que 6 arestas ($k.k'$) em G', podemos tentar encontrar uma cobertura por

vértices para tal grafo de tamanho menor ou igual a k'. Então, testamos cada subconjunto de V' de tamanho menor ou igual a 2 em busca de uma cobertura por vértices para G', até encontrar o conjunto $\{v_2, v_3\}$. Estes vértices mais o vértice v_5 (conjunto H) formam uma cobertura por vértices para o grafo G de tamanho menor ou igual a 3, conforme podemos ver na Figura 3.1(d). Note que outro subconjunto de V' ($\{v_3, v_4\}$) satisfaz as condições de cobertura para G', mostrando que a cobertura por vértices para um grafo não é única.

3.3 Algoritmos de Balasubramanian *et al.*

Como vimos na Seção 2.3, um algoritmo FPT pode executar a fase de árvore limitada de busca logo após a fase de redução ao núcleo do problema. Em ambos os algoritmos de Balasubramanian *et al.* [1], inicialmente executamos uma fase de redução ao núcleo do problema inspirada no algoritmo de Buss [2], já descrito na Seção 3.2. Assim, o nó raiz da árvore armazena a situação imediatamente após a fase de redução ao núcleo do problema, ou seja, a instância $\langle G', k' \rangle$ e H como cobertura parcial.

Cada nó da árvore armazena uma cobertura por vértices parcial (CVP) e uma instância reduzida do problema $\langle G'' = (V'', E''), k'' \rangle$. O grafo G'' é resultante da remoção dos vértices de G que estão em CVP, bem como das arestas neles incidentes e quaisquer vértices isolados, e o número inteiro k'' corresponde ao tamanho máximo desejado para a cobertura por vértices de G''.

As arestas da árvore representam as várias possibilidades de adição de vértices à cobertura parcial existente no nó pai. Note que um nó da árvore tem um CVP com mais elementos e um grafo com menos vértices e arestas do que seu pai, pois sempre que um vértice é adicionado à cobertura parcial, fazemos sua remoção do grafo, bem como das arestas nele incidentes e quaisquer vértices isolados.

A árvore deve ser analisada exaustivamente, geralmente em uma busca em profundidade, com o objetivo de encontrar uma solução para o problema. Os algoritmos de Balasubramanian *et al.* [1] apresentam duas opções distintas para a geração da árvore de busca. É importante destacar que não geramos todos os nós da árvore antes de fazer a busca em profundidade. Um nó da árvore de busca só é computado no momento em que este é visitado, de acordo com a busca em profundidade.

O crescimento da árvore de busca (ramificação dos nós) é interrompido quando o nó em questão tem uma cobertura parcial de tamanho k. Note que não adianta continuar ramificando nós que já tenham $|CVP| = k$, pois seus nós filhos terão coberturas parciais de tamanho maior que k. Dessa forma, limitamos o tamanho da árvore de busca em função do parâmetro k.

A árvore limitada de busca tem a seguinte propriedade: para cada cobertura por vértices de tamanho menor ou igual a k existente para o grafo G, existe um nó correspondente na árvore limitada de busca com um grafo resultante vazio e uma cobertura por vértices de tamanho válido (não necessariamente a mesma). Porém, se não existe cobertura por vértices de tamanho menor ou igual a k para o grafo G, então nenhum nó

da árvore limitada de busca possui um grafo resultante vazio. Assim, no pior caso temos que percorrer toda a árvore limitada de busca para determinar se existe ou não cobertura por vértices de tamanho menor ou igual a k para o grafo G.

Em cada nó da árvore limitada de busca, temos que escolher os vértices a adicionar na cobertura parcial e removê-los do grafo, bem como as arestas neles incidentes e quaisquer vértices isolados. Assim, dada a lista de adjacências do grafo, gastamos tempo $O(m)$ em cada nó da árvore, onde m é o número de vértices do grafo da instância reduzida. O maior grafo da árvore limitada de busca é armazenado no nó raiz, e é resultante da fase de redução ao núcleo do problema. O tamanho deste grafo é limitado por $O(k^2)$, como vimos na Seção 3.2. Portanto, se $C(k)$ é a quantidade de nós da árvore limitada de busca, então o tempo gasto para percorrer toda a árvore é de $O(C(k)k^2)$.

Os dois algoritmos de Balasubramanian *et al.* [1], doravante denominados **Algoritmo B1** e **Algoritmo B2**, diferem nas regras de escolha dos vértices a adicionar na cobertura parcial em cada nó da árvore de busca e, conseqüentemente na ramificação dos nós da árvore. O Algoritmo B1 e o Algoritmo B2 são descritos nas subseções a seguir.

3.3.1 Algoritmo B1

Em cada nó da árvore limitada de busca, a escolha dos vértices a adicionar na cobertura parcial depende de uma busca em profundidade feita a partir de um vértice qualquer do grafo, que passe por no máximo três arestas. A seguir apresentamos uma descrição deste algoritmo.

ALGORITMO B1
Entrada: o grafo $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho no máximo k, se existir; ou uma mensagem, se tal cobertura por vértices não existir.

1. Seja H o conjunto de vértices em V de grau maior que k.
2. **se** $|H| > k$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
3. Seja $G' = (V', E')$ o grafo resultante da remoção dos vértices de G que estão em H, bem como das arestas neles incidentes e de qualquer vértice isolado.
4. $k' \leftarrow k - |H|$
5. **se** $|E'| > k.k'$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
6. Seja T a árvore de busca cujo nó raiz armazena $\langle G', k' \rangle$ e H. Fazemos busca em profundidade na árvore T.
7. **para** cada nó da árvore T **faça**
8. Seja r o nó atual da árvore T. Sejam $\langle G'' = (V'', E''), k'' \rangle$ a instância reduzida do problema e CVP a cobertura por vértices parcial armazenados em r.
9. **se** $k'' >= 0$
10. **se** G'' for vazio
    devolva CVP; **termina**

11. Escolha aleatoriamente um vértice v de V''. A partir de v, faça uma busca em profundidade no grafo que passe por no máximo três arestas. (Nos próximos quatro passos, E''' resulta da remoção das arestas de E'' que incidem nos vértices removidos de V''. Além disso, qualquer vértice isolado em V''' também é removido.)
12. **se** o Passo 11 resulta em um caminho simples de tamanho três que consiste na seqüência de vértices v, v_1, v_2, v_3; o nó r gera três nós filhos assim:
    a) $CVP = CVP \bigcup \{v, v_2\}$; $\langle G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$
    b) $CVP = CVP \bigcup \{v_1, v_2\}$; $\langle G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$
    c) $CVP = CVP \bigcup \{v_1, v_3\}$; $\langle G''' = (V'' - \{v_1, v_3\}, E'''), k''' = k'' - 2\rangle$
    Vá para o próximo nó da busca em profundidade na árvore T.
13. **se** o Passo 11 resulta em um ciclo de tamanho três que consiste na seqüência de vértices v, v_1, v_2, v; o nó r gera três nós filhos assim:
    a) $CVP = CVP \bigcup \{v, v_1\}$; $\langle G''' = (V'' - \{v, v_1\}, E'''), k''' = k'' - 2\rangle$
    b) $CVP = CVP \bigcup \{v_1, v_2\}$; $\langle G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$
    c) $CVP = CVP \bigcup \{v, v_2\}$; $\langle G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$
    Vá para o próximo nó da busca em profundidade na árvore T.
14. **se** o Passo 11 resulta em um caminho simples de tamanho dois que consiste na seqüência de vértices v, v_1, v_2; executa-se no próprio nó r a seguinte operação:
    a) $CVP = CVP \bigcup \{v_1\}$; $\langle G''' = (V'' - \{v_1\}, E'''), k''' = k'' - 1\rangle$
15. **se** o Passo 11 resulta em um caminho simples de tamanho um que consiste na seqüência de vértices v, v_1; executa-se no próprio nó r a seguinte operação:
    a) $CVP = CVP \bigcup \{v\}$; $\langle G''' = (V'' - \{v\}, E'''), k''' = k'' - 1\rangle$

    senão
    Vá para o próximo nó da busca em profundidade na árvore T.
16. **devolva** "Não existe a cobertura por vértices desejada".

Os primeiros cinco passos do algoritmo são responsáveis pela fase de redução ao núcleo do problema e gastam tempo $O(kn)$. Esses passos são os mesmos cinco passos iniciais do algoritmo de Buss [2], já discutido na Seção 3.2.

Do Passo 6 em diante temos a fase da árvore limitada de busca. O nó raiz da árvore de busca armazena a instância $\langle G', k'\rangle$ e o conjunto H como cobertura parcial, resultantes da fase de redução ao núcleo do problema. Fazemos uma busca em profundidade na árvore a ser computada.

Dos Passos 7 até 16 temos um laço para percorrer todos os nós da árvore. Como vimos, cada nó da árvore de busca armazena uma instância reduzida do problema ($\langle G'' = (V'', E''), k''\rangle$) e uma cobertura parcial (CVP). Nesse laço, o tamanho da árvore de busca é limitada por k'' (Passo 9), pois enquanto esse valor for positivo significa que o tamanho da cobertura por vértices para G ainda é menor ou igual a k. No Passo 10, temos uma condição de saída para o algoritmo, pois se o grafo G'' é vazio (não tem vértices nem arestas), significa que uma cobertura por vértices de tamanho menor ou igual a k para o grafo G (CVP) foi encontrada.

Em cada nó da árvore de busca, escolhemos aleatoriamente um vértice $v \in V''$ e

fazemos uma busca em profundidade que passe por no máximo três arestas de G'' (Passo 11). Baseado nos possíveis caminhos que podem ser obtidos nessa busca em profundidade, um dos quatro casos a seguir podem acontecer (Passos 12, 13, 14 e 15):

- **Caso 1.** A busca em profundidade resulta em um caminho simples de tamanho três que consiste na seqüência de vértices v, v_1, v_2, v_3. Existem três possibilidades de adição de vértices à cobertura parcial: $\{v, v_2\}$ ou $\{v_1, v_2\}$ ou $\{v_1, v_3\}$, portanto três nós filhos são gerados a partir do nó atual. Note que além de cobrir todas as arestas do caminho simples, estes pares de vértices podem acabar cobrindo outras arestas do grafo, já que não sabemos o grau de nenhum desses vértices, como podemos ver na Figura 3.2.

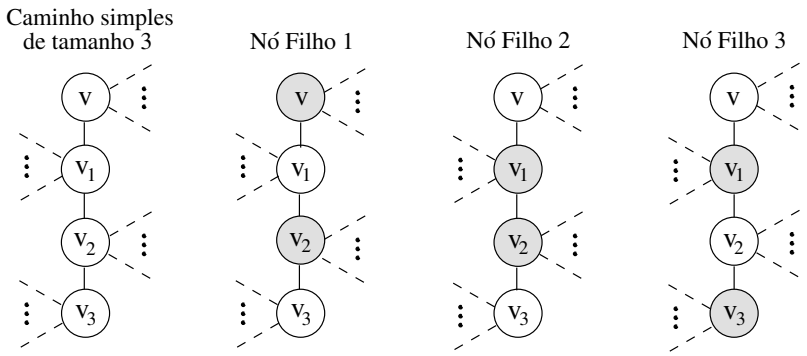


Figura 3.2: Caso 1 do Algoritmo B1. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

- **Caso 2.** A busca em profundidade resulta em um ciclo de tamanho três que consiste na seqüência de vértices v, v_1, v_2, v. Existem três possibilidades de adição de vértices à cobertura parcial: $\{v, v_1\}$ ou $\{v_1, v_2\}$ ou $\{v, v_2\}$, portanto três nós filhos são gerados a partir do nó atual. Note que além de cobrir todas as arestas do caminho simples, estes pares de vértices podem acabar cobrindo outras arestas do grafo, já que não sabemos o grau de nenhum desses vértices, como podemos ver na Figura 3.3.

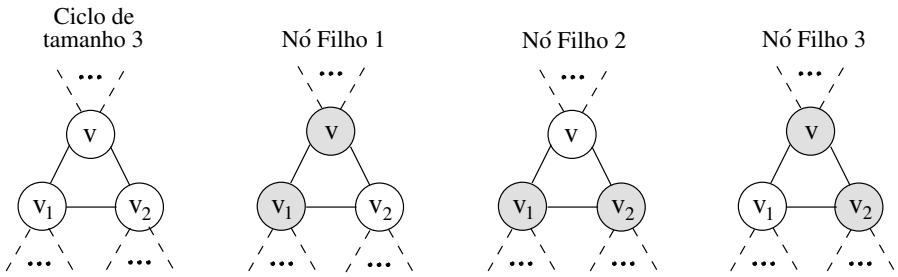


Figura 3.3: Caso 2 do Algoritmo B1. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

Nos Casos 1 e 2, cada nó filho do nó atual da árvore recebe em sua cobertura parcial um dos pares de vértices sugeridos. Além disso, o grafo G''' resulta da remoção dos vértices de G'' adicionados em CVP, bem como das arestas neles incidentes e quaisquer vértices isolados. O valor de k''' (tamanho máximo desejado da cobertura por vértices para G''') é subtraído de 2, que foi a quantidade de vértices acrescentados à cobertura parcial. O próximo nó da busca em profundidade na árvore é avaliado.

- **Caso 3.** A busca em profundidade resulta em um caminho simples de tamanho dois que consiste na seqüência de vértices v, v_1, v_2. Sabemos que o vértice v_2 tem grau 1, pois a busca em profundidade acabou nele. Apenas v_1 pode cobrir todas as arestas do caminho simples, portanto este vértice é acrescentado na cobertura parcial do próprio nó, como podemos ver na Figura 3.4.

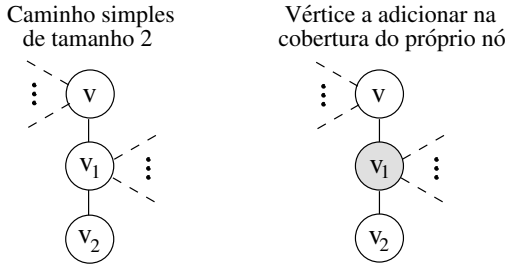


Figura 3.4: Caso 3 do Algoritmo B1. O vértice a adicionar em CVP está hachurado. A linha contínua indica aresta. A linha tracejada indica possível aresta.

- **Caso 4.** A busca em profundidade resulta em um caminho simples de tamanho um que consiste na seqüência de vértices v, v_1. Sabemos que o vértice v_1 tem grau 1, pois a busca em profundidade acabou nele. Portanto é melhor adicionar v na cobertura parcial do próprio nó, já que além de cobrir a única aresta do caminho, ainda pode cobrir outras possíveis arestas incidentes em v, como podemos ver na Figura 3.5.

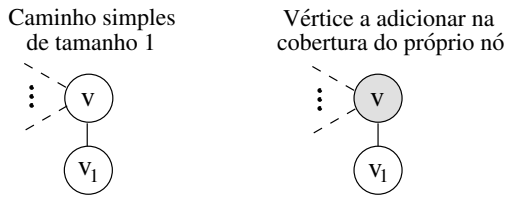


Figura 3.5: Caso 4 do Algoritmo B1. O vértice a adicionar em CVP está hachurado. A linha contínua indica aresta. A linha tracejada indica possível aresta.

Nos Casos 3 e 4, como existe apenas uma única possibilidade de adição de vértices à cobertura parcial, adicionamos tal vértice na cobertura do próprio nó da árvore e o removemos do grafo atual, bem como as arestas nele incidentes e quaisquer vértices isolados. Além disso, o tamanho máximo desejado para a cobertura por vértices desse grafo é

subtraído de 1, que foi a quantidade de vértices acrescentados à cobertura parcial. Então, executamos o Algoritmo B1 novamente sobre o mesmo nó da árvore de busca. Note que ainda não houve nenhuma ramificação a partir deste nó da árvore.

Os Casos 1, 2, 3 e 4 garantem a geração de uma árvore limitada de busca ternária pelo Algoritmo B1. A cada nível da árvore adicionamos no mínimo dois vértices na cobertura parcial. Portanto, a quantidade de nós na árvore limitada de busca é $O(3^{k/2})$. No pior caso temos que percorrer toda a árvore limitada de busca para determinar se existe ou não cobertura por vértices de tamanho menor ou igual a k para o grafo G. Como gastamos tempo $O(k^2)$ em cada nó da árvore, a fase de árvore limitada de busca desse algoritmo gasta tempo $(3^{k/2}k^2)$. O tempo total gasto pelo Algoritmo B1 é a soma do tempo gasto na redução ao núcleo do problema e na árvore limitada de busca, ou seja, $O(kn + \sqrt{3}^{k}k^2)$.

Exemplo de Execução do Algoritmo

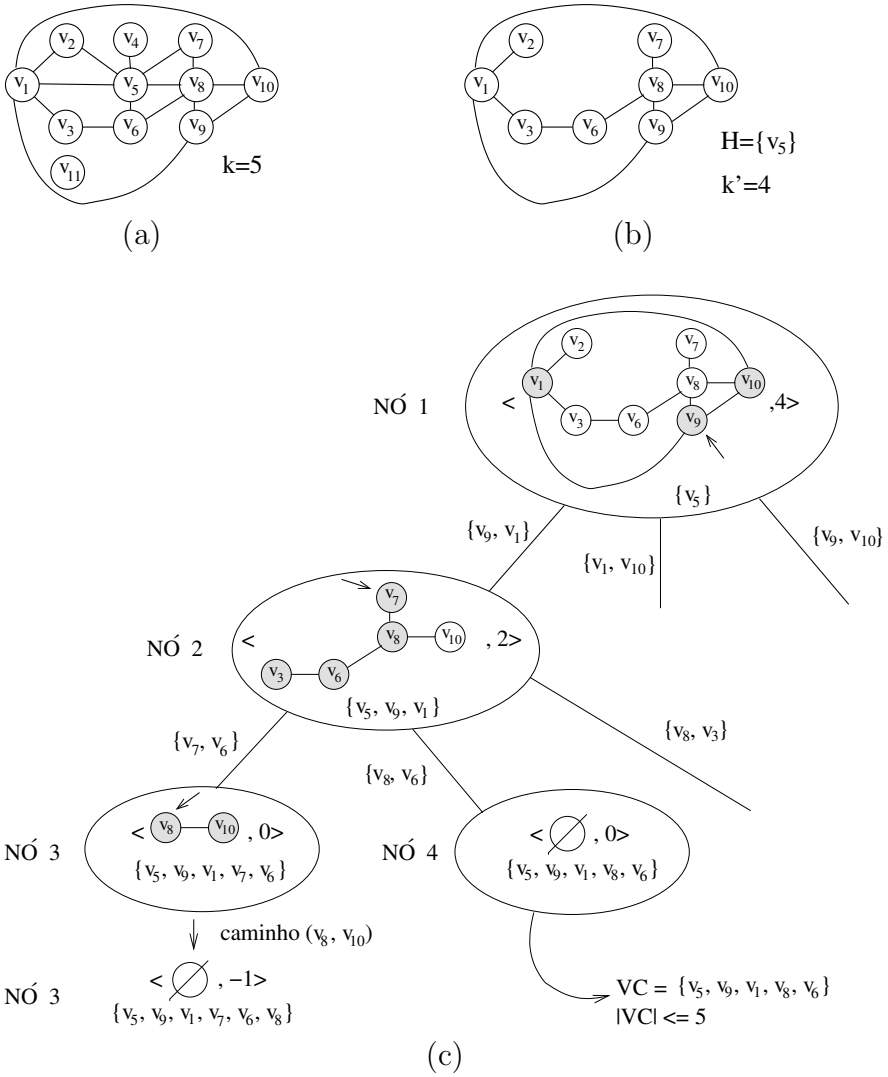


Figura 3.6: Execução do Algoritmo B1. (a) Grafo G. (b) Grafo G'. (c) Árvore limitada de busca do Algoritmo B1. Os vértices hachurados fazem parte da busca em profundidade que passa por no máximo três arestas do grafo, cujo vértice inicial é apontado pela seta.

Na Figura 3.6 apresentamos um exemplo de execução do Algoritmo B1 proposto por Balasubramanian *et al.* [1]. Seja o grafo G apresentado na Figura 3.6(a), queremos determinar se existe uma cobertura por vértices para o grafo G de tamanho menor ou igual a 5, ou seja, $k = 5$.

O grafo G' apresentado na Figura 3.6(b) resulta da fase de redução ao núcleo do problema. Removemos de G os vértices de grau maior que 5 (v_5), bem como as arestas neles incidentes e quaisquer vértices isolados (v_4). O vértice v_5 é adicionado ao conjunto H porque sabemos que ele pertence a qualquer cobertura por vértices de tamanho menor ou igual a 5 para o grafo G. Dessa forma, podemos adicionar no máximo mais 4 vértices na cobertura e ainda obter uma cobertura de tamanho válido, por isso $k' = 4$.

A árvore limitada de busca gerada pelo Algoritmo B1 é exibida na Figura 3.6(c). Note que a árvore gerada é ternária, pois sempre que houver ramificação em um nó, o algoritmo garante a geração de três nós filhos. O nó raiz da árvore de busca armazena a instância resultante da redução ao núcleo do problema ($\langle G', 4\rangle$) e uma cobertura parcial composta por $\{v_5\}$. O primeiro nó da árvore que visitamos é exatamente o nó raiz (NÓ 1). Suponha que o vértice v_9 seja sorteado como vértice inicial da busca em profundidade que passa por no máximo três arestas do grafo, e que tal busca resulta em um ciclo de tamanho três (Caso 2), formado por (v_9, v_1, v_{10}, v_9). Nesse caso temos três possibilidades de adição de vértices à cobertura parcial, portanto três nós filhos são gerados. Escolhemos o primeiro filho (NÓ 2) para visitar, pois estamos fazendo busca em profundidade na árvore.

O grafo do NÓ 2 resulta da remoção dos vértices v_9 e v_1 do grafo do NÓ 1, bem como das arestas neles incidentes e quaisquer vértices isolados (v_2). Como v_9 e v_1 são adicionados à cobertura parcial, podemos acrescentar no máximo mais dois vértices na cobertura e ainda obter uma cobertura de tamanho válido. Suponha que o vértice v_7 seja sorteado como vértice inicial da busca em profundidade que passa por no máximo três arestas do grafo, e que tal busca resulta em um caminho simples de tamanho três (Caso 1), formado por (v_7, v_8, v_6, v_3). Nesse caso temos três possibilidades de adição de vértices à cobertura parcial, portanto três nós filhos são gerados. O primeiro filho (NÓ 3) é visitado.

O grafo do NÓ 3 resulta da remoção dos vértices v_7 e v_6 do grafo do NÓ 2, bem como das arestas neles incidentes e quaisquer vértices isolados (v_3). Como v_7 e v_6 são adicionados à cobertura parcial, não podemos mais acrescentar vértices na cobertura e ainda obter uma de tamanho válido. Suponha que o vértice v_8 seja sorteado como vértice inicial da busca em profundidade que passa por no máximo três arestas do grafo, e que tal busca resulta em um caminho simples de tamanho um (Caso 4), formado por (v_8, v_{10}). Assim, o vértice v_8 é adicionado à cobertura parcial e removido do grafo do próprio NÓ 3. No entanto, como a cobertura parcial fica com mais de 5 elementos, interrompemos o crescimento da árvore nesse nó e vamos para o próximo nó válido da busca em profundidade na árvore.

O próximo nó válido da busca em profundidade é o NÓ 4, cujo grafo resulta da remoção dos vértices v_8 e v_6 do grafo do NÓ 2, bem como das arestas neles incidentes e quaisquer vértices isolados (v_3, v_7 e v_{10}). Como v_8 e v_6 são adicionados à cobertura parcial, não podemos mais adicionar vértices na cobertura e ainda obter uma de tamanho válido. No entanto, como o grafo resultante nesse nó é vazio, significa que sua cobertura parcial

formada por $\{v_5, v_9, v_1, v_8, v_6\}$ é uma cobertura por vértices de tamanho menor ou igual a 5 para o grafo G.

3.3.2 Algoritmo B2

Em cada nó da árvore limitada de busca, a escolha dos vértices a adicionar na cobertura parcial depende de cinco casos que consideram o grau dos vértices do grafo resultante. A seguir apresentamos uma descrição deste algoritmo.

ALGORITMO B2
Entrada: o grafo $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho no máximo k, se existir; ou uma mensagem, se tal cobertura por vértices não existir.

1. Seja H o conjunto de vértices em V de grau maior que k.
2. **se** $|H| > k$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
3. Seja $G' = (V', E')$ o grafo resultante da remoção dos vértices de G que estão em H, bem como das arestas neles incidentes e de qualquer vértice isolado.
4. $k' \leftarrow k - |H|$
5. **se** $|E'| > k.k'$
   devolva "Não existe a cobertura por vértices desejada"; **termina**
6. Seja T a árvore de busca cujo nó raiz armazena $\langle G', k' \rangle$ e H. Fazemos busca em profundidade na árvore T.
7. **para** cada nó da árvore T **faça**
8. Seja r o nó atual da árvore T. Sejam $\langle G'' = (V'', E''), k'' \rangle$ a instância reduzida do problema e CVP a cobertura por vértices parcial armazenados em r. Seja $N(v)$ o conjunto dos vértices vizinhos ao vértice v e $N(S) = \bigcup_{v \in S} N(v)$.
9. **se** $k'' >= 0$
10. **se** G'' for vazio
    devolva CVP; **termina**
11. Escolha um vértice $v \in V''$ priorizando vértices de grau 1, depois de grau 2, de grau maior ou igual a 5, de grau 3 e de grau 4. (Nos próximos quinze passos, E''' resulta da remoção das arestas de E'' que incidem nos vértices removidos de V''. Além disso, qualquer vértice isolado em V''' também deve ser removido.)
12. **se** o vértice v escolhido no Passo 11 tem grau 1, seja o vértice x seu único vizinho. O nó r gera um nó filho assim:
    a) $CVP = CVP \bigcup \{x\}$; $\langle G''' = (V'' - \{x\}, E'''), k''' = k'' - 1 \rangle$
13. **se** o vértice v escolhido no Passo 11 tem grau 2, sejam os vértices x e y seus vizinhos
14 **se** existe uma aresta entre x e y, o nó r gera um nó filho assim:
    a) $CVP = CVP \bigcup \{x, y\}$; $\langle G''' = (V'' - \{x, y\}, E'''), k''' = k'' - 2 \rangle$
15. **senão se**, juntos, x e y têm pelo menos dois vizinhos diferentes de v, o nó r gera dois nós filhos assim:

a) $CVP = CVP \bigcup \{x, y\}$; $\langle G''' = (V'' - \{x, y\}, E'''), k''' = k'' - 2\rangle$
b) $CVP = CVP \bigcup N(\{x, y\})$; $\langle G''' = (V'' - N(\{x, y\}), E'''),$
$k''' = k'' - |N(\{x, y\})|\rangle$

16. **senão se** x e y compartilham um único vizinho em comum além de v, digamos o vértice a, o nó r gera um nó filho assim:
a) $CVP = CVP \bigcup \{v, a\}$; $\langle G''' = (V'' - \{v, a\}, E'''), k''' = k'' - 2\rangle$
17. **se** o vértice v escolhido no Passo 11 tem grau maior ou igual a 5, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{v\}$; $\langle G''' = (V'' - \{v\}, E'''), k''' = k'' - 1\rangle$
b) $CVP = CVP \bigcup N(v)$; $\langle G''' = (V'' - N(v), E'''), k''' = k'' - |N(v)|\rangle$
18. **se** o vértice v escolhido no Passo 11 tem grau 3, sejam os vértices x, y e z seus vizinhos
19. **se** existe uma aresta entre x e y, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}$; $\langle G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3\rangle$
b) $CVP = CVP \bigcup N(z)$; $\langle G''' = (V'' - N(z), E'''), k''' = k'' - |N(z)|\rangle$
20. **senão se** x e y têm um vizinho em comum diferente de v, digamos o vértice a, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}$; $\langle G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3\rangle$
b) $CVP = CVP \bigcup \{v, a\}$; $\langle G''' = (V'' - \{v, a\}, E'''), k''' = k'' - 2\rangle$
21. **senão se** x tem três vizinhos diferentes de v, o nó r gera três nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}$; $\langle G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3\rangle$
b) $CVP = CVP \bigcup N(x)$; $\langle G''' = (V'' - N(x), E'''), k''' = k'' - 4\rangle$
c) $CVP = CVP \bigcup \{x\} \bigcup N(\{y, z\})$; $\langle G''' = (V'' - \{x\} \bigcup N(\{y, z\}), E'''),$
$k''' = k'' - 1 - |N(\{y, z\})|\rangle$
22. **senão se** x, y e z têm dois vizinhos privados diferentes de v cada um, sendo que os vértices a e b são os vizinhos de x, o nó r gera três nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}$; $\langle G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3\rangle$
b) $CVP = CVP \bigcup N(x)$; $\langle G''' = (V'' - N(x), E'''), k''' = k'' - 3\rangle$
c) $CVP = CVP \bigcup N(\{y, z, a, b\})$; $\langle G''' = (V'' - N(\{y, z, a, b\}), E'''),$
$k''' = k'' - |N(\{y, z, a, b\})|\rangle$
23. **se** o vértice v escolhido no Passo 11 tem grau 4, sejam os vértices x, y, z e w seus vizinhos
24. **se** existe uma aresta entre x e y, o nó r gera três nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z, w\}$; $\langle G''' = (V'' - \{x, y, z, w\}, E'''),$
$k''' = k'' - 4\rangle$
b) $CVP = CVP \bigcup N(z)$; $\langle G''' = (V'' - N(z), E'''), k''' = k'' - 4\rangle$
c) $CVP = CVP \bigcup \{z\} \bigcup N(w)$; $\langle G''' = (V'' - \{z\} \bigcup N(w), E'''),$
$k''' = k'' - |\{z\} \bigcup N(w)|\rangle$
25. **senão se** x, y e z têm um vizinho em comum diferente de v, digamos o vértice a, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z, w\}$; $\langle G''' = (V'' - \{x, y, z, w\}, E'''),$
$k''' = k'' - 4\rangle$
b) $CVP = CVP \bigcup \{v, a\}$; $\langle G''' = (V'' - \{v, a\}, E'''), k''' = k'' - 2\rangle$
26. **senão se** x, y, z e w têm pelo menos três vizinhos diferentes de v, o nó r gera quatro nós filhos assim:

a) $CVP = CVP \bigcup \{x, y, z, w\}$; $\langle G''' = (V'' - \{x, y, z, w\}, E'''),$
$k''' = k'' - 4\rangle$
b) $CVP = CVP \bigcup N(y)$; $\langle G''' = (V'' - N(y), E'''), k''' = k'' - 4\rangle$
c) $CVP = CVP \bigcup \{y\} \bigcup N(w)$; $\langle G''' = (V'' - \{y\} \bigcup N(w), E'''),$
$k''' = k'' - 5\rangle$
d) $CVP = CVP \bigcup \{y, w\} \bigcup N(\{x, z\})$;
$\langle G''' = (V'' - \{y, w\} \bigcup N(\{x, z\}), E'''), k''' = k'' - 2 - |N(\{x, z\})|\rangle$

senão
Vá para o próximo nó válido da busca em profundidade na árvore T.
27. **devolva** "Não existe a cobertura por vértices desejada".

Os primeiros cinco passos do algoritmo representam a fase de redução ao núcleo do problema, que gasta tempo $O(kn)$. Esses passos são os mesmos cinco passos iniciais do algoritmo de Buss [2], já discutido na Seção 3.2.

A fase de árvore limitada de busca acontece a partir do Passo 6. Como vimos, cada nó da árvore armazena uma instância reduzida do problema ($\langle G'', k''\rangle$) e uma cobertura por vértices parcial (CVP). O nó raiz da árvore armazena $\langle G', k'\rangle$ e o conjunto H, resultantes da redução ao núcleo do problema.

Faremos uma busca em profundidade na árvore com o objetivo de encontrar soluções para o problema. Por isso, temos um laço dos Passos 7 a 26 para garantir que todos os nós da arvore de busca sejam percorridos. Nesse laço, o tamanho da árvore de busca é limitada por k'' (Passo 9), pois enquanto esse valor for positivo significa que o tamanho da cobertura por vértices para G ainda é menor ou igual a k. Se o grafo G'' é vazio (não tem vértices nem arestas) em um dos nós da árvore limitada de busca, significa que encontramos uma cobertura por vértices de tamanho menor ou igual a k para o grafo G (CVP), justificando o Passo 10.

Denotamos por $N(v)$ o conjunto dos vértices vizinhos a v e $N(S) = \bigcup_{v \in S} N(v)$. É importante lembrar que se um vértice com vizinhos não está na cobertura, então seus vizinhos têm que estar, para cobrir as arestas que os ligam. Além disso, sempre que adicionamos um vértice na cobertura parcial temos que removê-lo do grafo resultante, bem como as arestas nele incidentes e quaisquer vértices isolados.

Ao contrário do Algoritmo B1, este algoritmo não garante a geração de uma árvore ternária, nem que no mínimo dois vértices são adicionados à cobertura parcial a cada ramificação do nó. No Algoritmo B2, os nós da árvore de busca podem ter 1, 2, 3 ou 4 filhos, dependendo do caso selecionado. A quantidade de vértices adicionados à cobertura parcial também varia de acordo com o caso selecionado. Em um dos casos, pelo menos oito vértices são adicionados à cobertura parcial do nó filho.

Cada caso do Algoritmo B2 resulta em uma ramificação do nó atual, a qual nos leva a uma recorrência. Usamos as recorrências desses casos para calcular a quantidade de nós da árvore limitada. Tal quantidade é necessária no cálculo da complexidade de tempo do algoritmo. As recorrências relacionam a quantidade de nós da subárvore do nó pai com a quantidade de nós das subárvores dos nós filhos. Por exemplo, a recorrência:

$$C(k) \leq 1 + C(k-2) + C(k-3)$$

relaciona a quantidade de nós da subárvore do nó pai ($C(k)$) com a quantidade de nós das subárvores de seus dois nós filhos ($C(k-2)$ e $C(k-3)$). O número 1 que aparece na inequação representa o próprio nó pai. $C(k-2)$ indica que dois vértices foram adicionados à cobertura de um filho. $C(k-3)$ indica que três vértices foram adicionados à cobertura do outro filho. Após cada caso do Algoritmo B2 apresentamos a recorrência resultante.

No Passo 11 escolhemos um vértice v do grafo G'', priorizando vértices de grau 1, depois de grau 2, de grau maior ou igual a 5, de grau 3 e, por fim, de grau 4. Em cada caso deste algoritmo, a seleção dos vértices a adicionar na cobertura parcial deve sempre cobrir todas as arestas entre v e seus vizinhos.

- **Caso 1.** Existe no grafo G'' vértice v de grau 1. Seja x o vértice vizinho de v. Como v cobre apenas uma aresta, é mais vantajoso adicionar x na cobertura parcial porque além de cobrir tal aresta ainda pode cobrir outras possíveis arestas, como podemos ver na Figura 3.7. O Caso 1 resulta na seguinte recorrência:

  $$C(k) \leq 1 + C(k-1) \tag{3.1}$$

  Pois o nó atual gera apenas um nó filho que recebe um vértice ($\{x\}$) em sua cobertura parcial.

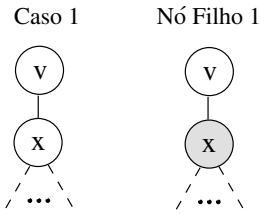


Figura 3.7: Caso 1 do Algoritmo B2. O vértice a adicionar em *CVP* está hachurado. A linha contínua indica aresta. A linha tracejada indica possível aresta.

- **Caso 2.** Existe no grafo G'' vértice v de grau 2. Sejam x e y os vértices vizinhos de v. Em qualquer cobertura por vértices, se apenas um dos vizinhos de v estiver na cobertura, x por exemplo, então v tem que estar para cobrir a aresta vy. Entretanto, ao invés de adicionar v na cobertura, é melhor adicionar y porque além de cobrir tal aresta ainda pode cobrir outras possíveis arestas. Então, sem perda de generalidade, podemos afirmar que ou x e y estão juntos na cobertura ou nenhum deles está, como podemos ver na Figura 3.8.

  O Caso 2 (Geral) apresenta duas opções para cobrirmos todas as arestas entre v e seus dois vizinhos. No entanto, existem três situações peculiares no Caso 2 que determinam quais dessas opções devemos usar, descritas a seguir.

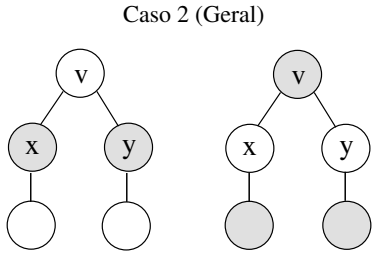


Figura 3.8: Caso 2 (Geral) do Algoritmo B2. Ou x e y estão juntos na cobertura ou nenhum deles está.

– **Subcaso 2.1.** Existe uma aresta entre os vértices x e y. Neste caso é melhor adicionar x e y na cobertura parcial porque além de cobrirem as três arestas entre v, x e y, podem cobrir outras possíveis arestas, como podemos ver na Figura 3.9. O Subcaso 2.1 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-2) \tag{3.2}$$

Pois o nó atual gera apenas um nó filho que recebe dois vértices ($\{x, y\}$) em sua cobertura parcial.

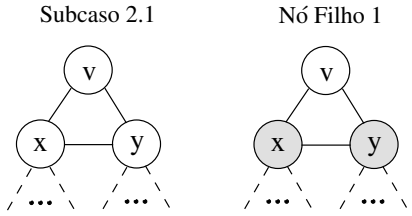


Figura 3.9: Subcaso 2.1 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

– **Subcaso 2.2.** Não existe aresta entre os vizinhos de v e, juntos, x e y têm pelo menos dois vizinhos diferentes de v, digamos os vértices a e b. Existem duas possibilidades de adição de vértices à cobertura parcial: $\{x, y\}$ ou $N(\{x, y\})$, como podemos ver na Figura 3.10. O Subcaso 2.2 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-2) + C(k-3) \tag{3.3}$$

Pois o nó atual gera dois nós filhos. O primeiro filho recebe dois vértices ($\{x, y\}$) em sua cobertura parcial. O segundo filho recebe pelo menos três vértices ($\{v, a, b\}$) em sua cobertura parcial.

– **Subcaso 2.3.** Não existe aresta entre os vizinhos de v e os vértices x e y compartilham um único vizinho em comum além do próprio v, digamos o vértice a. Ao invés de adicionar à cobertura x e y, que têm grau 2 e cobrem quatro

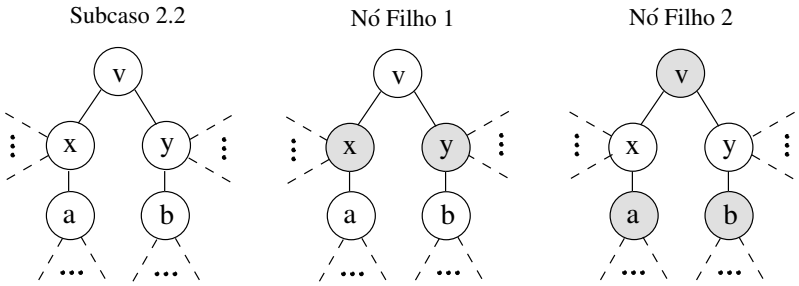


Figura 3.10: Subcaso 2.2 do Algoritmo B2. Os vértices a adicionar em CVP estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

arestas, é melhor acrescentar v e a porque além de cobrirem as mesmas quatro arestas ainda podem cobrir outras possíveis arestas, como podemos ver na Figura 3.11. O Subcaso 2.3 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-2) \tag{3.4}$$

Pois o nó atual gera apenas um filho que recebe dois vértices ($\{v, a\}$) em sua cobertura parcial.

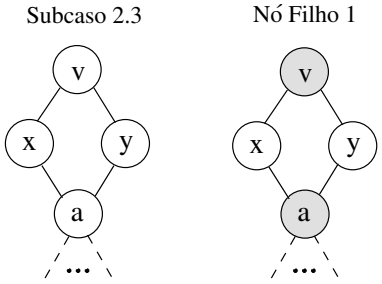


Figura 3.11: Subcaso 2.3 do Algoritmo B2. Os vértices a adicionar em CVP estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

- **Caso 3**. Existe no grafo G'' vértice v de grau maior ou igual a 5. Existem duas possibilidades de adição de vértices à cobertura parcial: $\{v\}$ e $N(v)$, como podemos ver na Figura 3.12. O Caso 3 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-1) + C(k-5) \tag{3.5}$$

Pois o nó atual gera dois nós filhos. O primeiro filho recebe apenas um vértice ($\{v\}$) em sua cobertura parcial. O segundo filho recebe pelo menos cinco vértices (vizinhos de v, que tem no mínimo grau 5).

- **Caso 4.** Existe no grafo G'' vértice v de grau 3. Sejam x, y e z os vértices vizinhos de v. Em qualquer cobertura por vértices, se tivermos um par entre os vizinhos de v na cobertura, digamos x e y, então v tem que estar para cobrir a aresta vz.

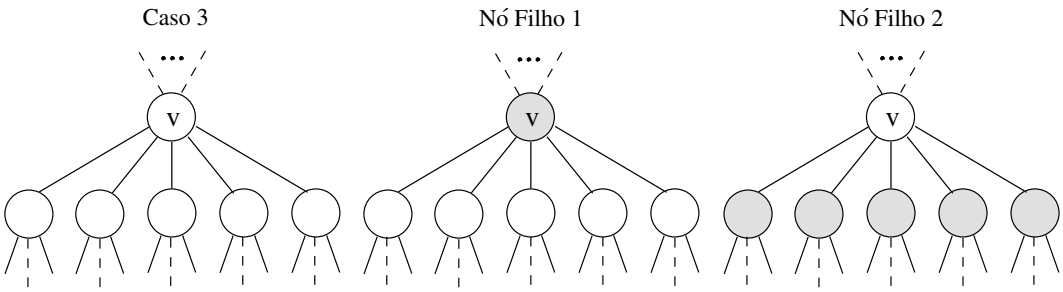


Figura 3.12: Caso 3 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

Entretanto, ao invés de adicionar v na cobertura, é melhor adicionar z porque além de cobrir tal aresta ainda pode cobrir outras possíveis arestas. Então, sem perda de generalidade, podemos afirmar que ou x, y e z estão juntos na cobertura, ou apenas um deles está ou nenhum deles está, como podemos ver na Figura 3.13.

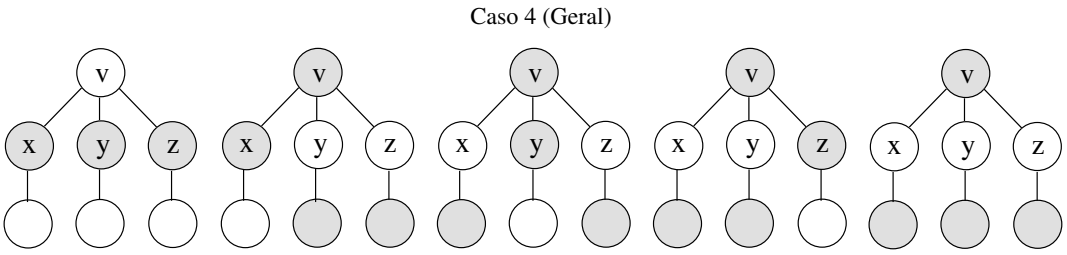


Figura 3.13: Caso 4 (Geral) do Algoritmo B2. Ou x, y e z estão juntos na cobertura ou apenas um deles está ou nenhum deles está.

O Caso 4 (Geral) apresenta cinco opções para cobrirmos todas as arestas entre v e seus três vizinhos. No entanto, existem quatro situações peculiares no Caso 4 que determinam quais dessas opções devemos usar, descritas a seguir.

- **Subcaso 4.1.** Existe uma aresta entre dois vizinhos de v, digamos x e y. Existem duas possibilidades de adição de vértices à cobertura parcial: $\{x, y, z\}$ ou $N(z)$, como podemos ver na Figura 3.14. O Subcaso 4.1 resulta na seguinte recorrência:

$$C(k) \leq 1 + 2C(k-3) \tag{3.6}$$

  Pois o nó atual gera dois nós filhos. O primeiro filho recebe três vértices ($\{x, y, z\}$) em sua cobertura parcial. O segundo filho recebe pelo menos três vértices (vizinhos de z, que tem grau 3 ou 4).

- **Subcaso 4.2.** Não existe aresta entre os vizinhos de v, mas dois deles (x e y, por exemplo) compartilham um vizinho em comum além de v, digamos o vértice a. Existem duas possibilidades de adição de vértices à cobertura parcial:

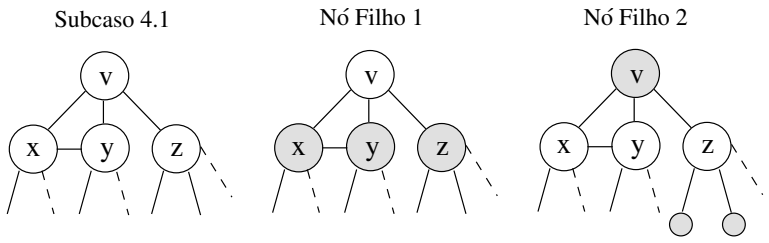


Figura 3.14: Subcaso 4.1 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

$\{x, y, z\}$ ou $\{v, a\}$, como podemos ver na Figura 3.15. O Subcaso 4.2 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-3) + C(k-2) \tag{3.7}$$

Pois o nó atual gera dois nós filhos. O primeiro filho recebe três vértices ($\{x, y, z\}$) em sua cobertura parcial. O segundo filho recebe dois vértices ($\{v, a\}$) em sua cobertura parcial.

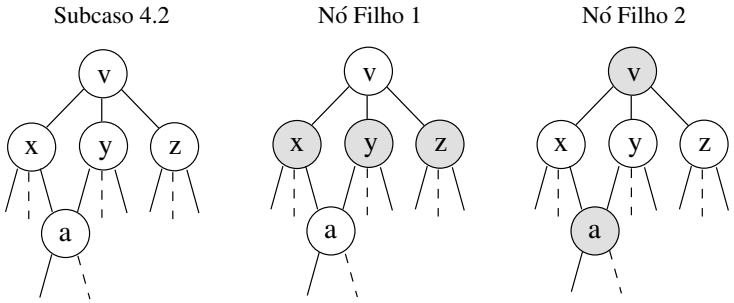


Figura 3.15: Subcaso 4.2 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

– **Subcaso 4.3.** Não existe aresta entre os vizinhos de v e eles não compartilham vizinhos em comum além do próprio v. Além disso, um deles, digamos x, tem três vizinhos diferentes de v. Existem três possibilidades de adição de vértices à cobertura parcial: $\{x, y, z\}$ ou $N(x)$ ou $\{x\} \bigcup N(\{y, z\})$, como podemos ver na Figura 3.16. O Subcaso 4.3 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-3) + C(k-4) + C(k-6) \tag{3.8}$$

Pois o nó atual gera três nós filhos. O primeiro filho recebe três vértices ($\{x, y, z\}$) em sua cobertura parcial. O segundo filho recebe quatro vértices (vizinhos de x, que tem grau 4) em sua cobertura parcial. O terceiro filho recebe pelo menos seis vértices (o vértice x mais os vizinhos de y e z). Os vértices y e z têm pelo menos grau 3 e compartilham um vizinho em comum (v), portanto são existem pelo menos cinco vértices vizinhos de y e z.

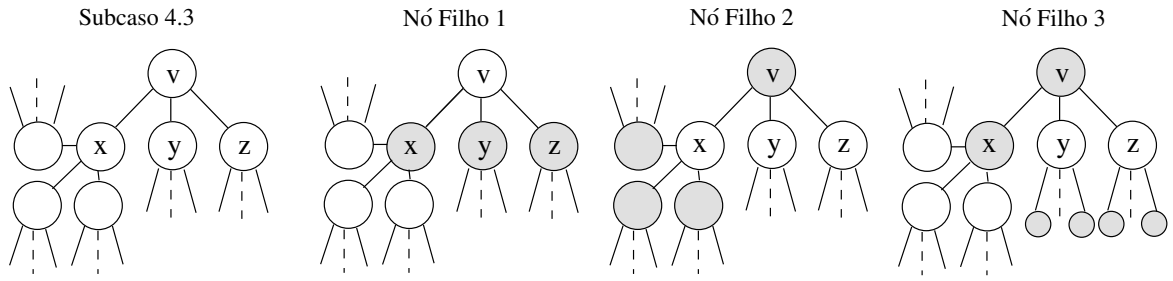


Figura 3.16: Subcaso 4.3 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta. A linha tracejada indica possível aresta.

– **Subcaso 4.4.** Não existe aresta entre os vizinhos de v e cada um deles tem exatamente dois vizinhos privados, desconsiderando v. Sejam os vértices a, b, 1, 2, 3 e 4 os vizinhos privados de x, y e z, respectivamente. Seja o vértice 5 um vizinho de a ou b. Existem três possibilidades de adição de vértices à cobertura parcial: $\{x, y, z\}$ ou $\{v, a, b\}$ ou $N(\{y, z, a, b\})$, como podemos ver na Figura 3.17. O Subcaso 4.4 resulta na seguinte recorrência:

$$C(k) \leq 1 + 2C(k-3) + C(k-7) \tag{3.9}$$

Pois o nó atual gera três nós filhos. O primeiro filho recebe três vértices ($\{x, y, z\}$) em sua cobertura parcial. O segundo filho também recebe três vértices ($\{v, a, b\}$) em sua cobertura parcial. O terceiro filho recebe pelo menos sete vértices (vizinhos de y, z, a e b). Note que y e z têm grau 3 e ambos são vizinhos de v, portanto juntos têm cinco vizinhos, mais pelo menos o vértice x e o vértice 5 (vizinhos de a e b).

- **Caso 5.** Existe no grafo G'' vértice v de grau 4. Sejam x, y, z e w os vértices vizinhos de v. Em qualquer cobertura por vértices, se tivermos três dos vizinhos de v na cobertura, digamos x, y e z, então v tem que estar para cobrir a aresta vw. Entretanto, ao invés de adicionar v na cobertura, é melhor adicionar w porque além de cobrir tal aresta ainda pode-se cobrir outras possíveis arestas. Então, sem perda de generalidade, podemos afirmar que ou x, y, z e w estão juntos na cobertura, ou no máximo um par deles está ou nenhum deles está, como podemos ver na Figura 3.18.

  O Caso 5 (Geral) apresenta doze opções para cobrirmos todas as arestas entre v e seus quatro vizinhos. No entanto, existem três situações peculiares no Caso 4 que determinam quais dessas opções devemos usar, descritas a seguir.

  – **Subcaso 5.1.** Existe uma aresta entre dois vizinhos de v, digamos x e y. Existem três possibilidades de adição de vértices à cobertura parcial: $\{x, y, z, w\}$ ou $N(z)$ ou $\{z\} \bigcup N(w)$, como podemos ver na Figura 3.19. O Subcaso 5.1 resulta na seguinte recorrência:

$$C(k) \leq 1 + 3C(k-4) \tag{3.10}$$

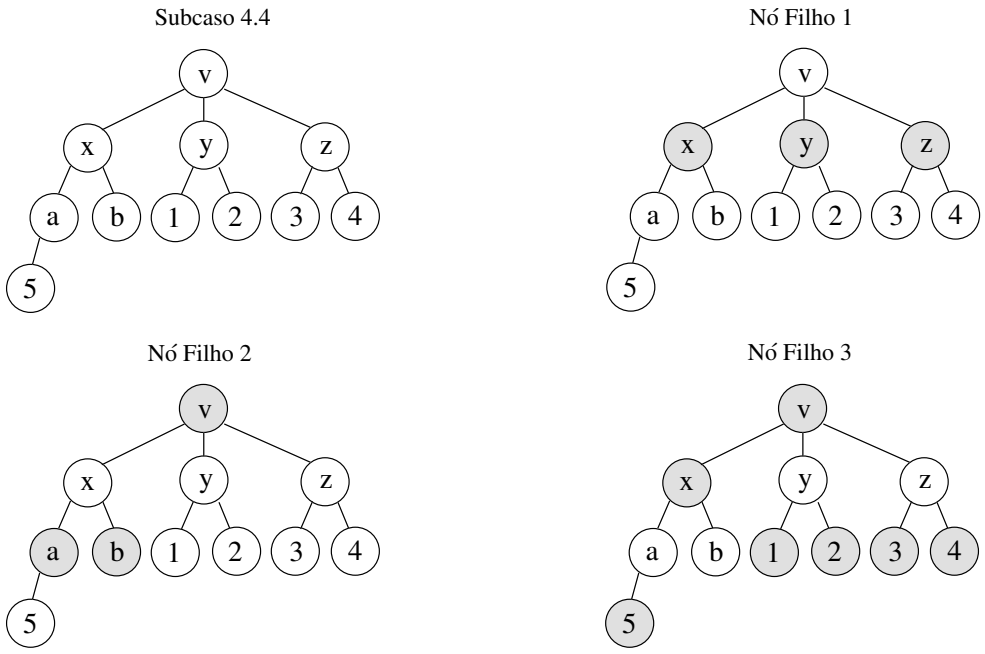


Figura 3.17: Subcaso 4.4 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta.

Pois o nó atual gera três nós filhos. O primeiro filho recebe quatro vértices ($\{x, y, z, w\}$) em sua cobertura parcial. O segundo filho também recebe quatro vértices (vizinhos de z, que tem grau 4) em sua cobertura parcial. O terceiro filho recebe pelo menos quatro vértices (vértice z mais os vizinhos de w). Note que w tem grau 4, mas pode ser adjacente a z.

– **Subcaso 5.2.** Não existe aresta entre os vizinhos de v e três deles (x, y e z, por exemplo) compartilham um vizinho em comum além do próprio v, digamos o vértice a. Existem duas possibilidades de adição de vértices à cobertura parcial: $\{x, y, z, w\}$ e $\{v, a\}$, como podemos ver na Figura 3.20. O Subcaso 5.2 resulta na seguinte recorrência:

$$C(k) \leq 1 + C(k-4) + C(k-2) \tag{3.11}$$

Pois o nó atual gera dois nós filhos. O primeiro filho recebe quatro vértices ($\{x, y, z, w\}$) em sua cobertura parcial. O segundo filho recebe dois vértices ($\{v, a\}$) em sua cobertura parcial.

– **Subcaso 5.3.** Não existe aresta entre os vizinhos de v e nenhum grupo de três deles compartilha um vizinho em comum além do próprio v. Além disso, todos eles têm pelo menos três vizinhos diferentes de v. Existem quatro possibilidades de adição de vértices à cobertura parcial: $\{x, y, z, w\}$ ou $N(y)$ ou $\{y\} \bigcup N(w)\}$ ou $\{y, w\} \bigcup N(\{x, z\})$, como podemos ver na Figura 3.21. O Subcaso 5.3 resulta na seguinte recorrência:

$$C(k) \leq 1 + 2C(k-4) + C(k-5) + C(k-8) \tag{3.12}$$

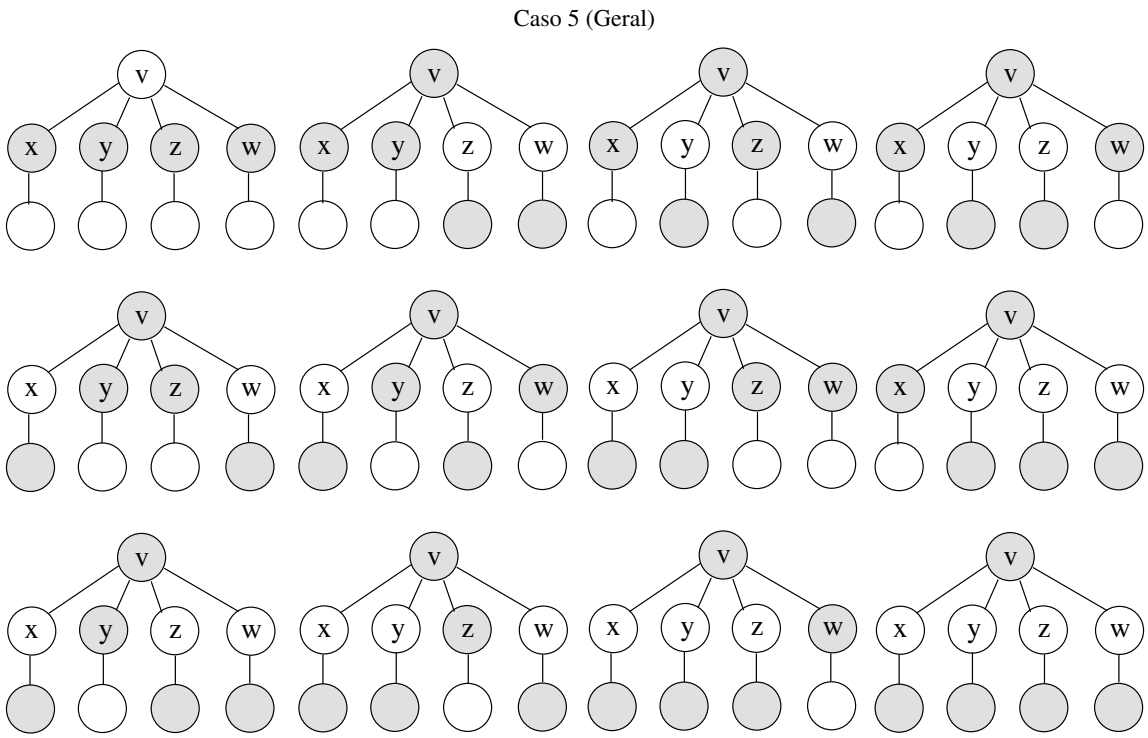


Figura 3.18: Caso 5 (Geral) do Algoritmo B2. Ou x, y, z e w estão juntos na cobertura ou no máximo um par deles está ou nenhum deles está.

> Pois o nó atual gera quatro nós filhos. O primeiro filho recebe quatro vértices ($\{x, y, z, w\}$) em sua cobertura parcial. O segundo filho também recebe quatro vértices (vizinhos de y, que tem grau 4) em sua cobertura parcial. O terceiro filho recebe cinco vértices (y mais os vizinhos de w, que tem grau 4) em sua cobertura parcial. O quarto filho recebe pelo menos oito vértices (vértices y e w mais os vizinhos de x e z). Note que x e z têm grau 4 e ambos são vizinhos de v, mas esse par de vértices pode compartilhar um vizinho em comum além de v.

Usando indução em k, podemos provar que $C(k) = d((1.324718)^k - 1)$ para alguma constante d que satisfaz todas as inequações, de 3.1 a 3.12. As Inequações 3.3 e 3.7 são as

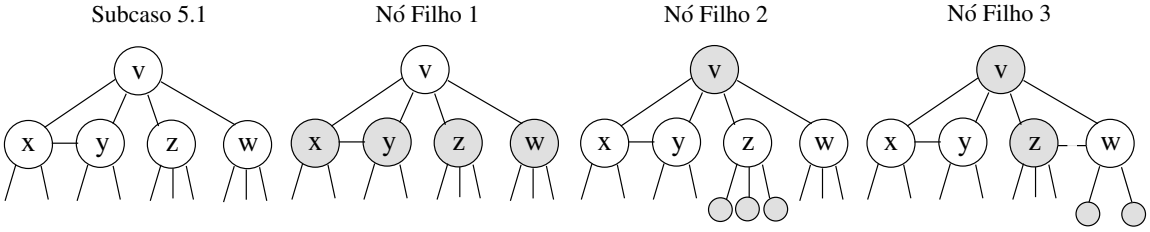


Figura 3.19: Subcaso 5.1 do Algoritmo B2. Os vértices a adicionar em CVP estão hachurados. A linha contínua indica aresta.

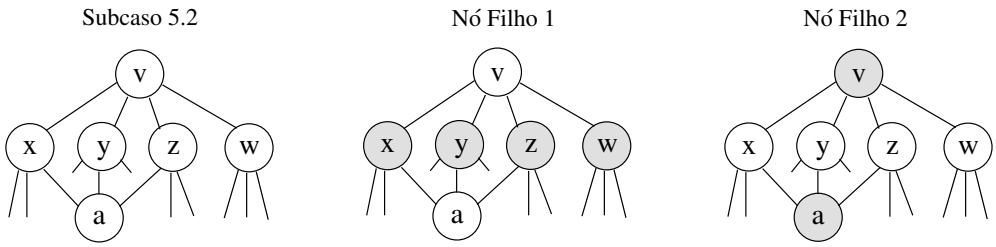


Figura 3.20: Subcaso 5.2 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta.

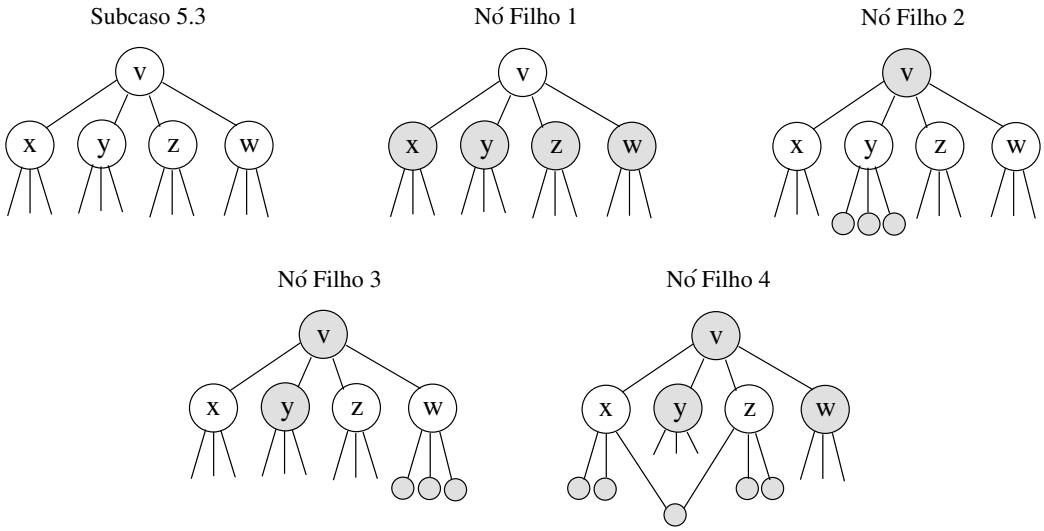


Figura 3.21: Subcaso 5.3 do Algoritmo B2. Os vértices a adicionar em *CVP* estão hachurados. A linha contínua indica aresta.

mesmas e ambas são críticas. É importante destacar que as recorrências dessas inequações já apresentam os piores casos para nossa árvore. A raiz real positiva para a equação $c^3 - c - 1 = 0$ é 1.324718 (aproximado para 6 casas decimais), que vem da recorrência da Inequação 3.3. Isso é suficiente apenas para verificar a solução nas Inequações 3.3, 3.5, 3.8, 3.9, 3.10 e 3.12. Da prova da Inequação 3.5 segue que $d((1.324718)^k - 1)$ satisfaz a Inequação 3.1. Da prova da Inequação 3.3 segue que $d((1.324718)^k - 1)$ também satisfaz as Inequações 3.2, 3.4, 3.6, 3.11. É fácil construir uma árvore de busca usando o Algoritmo B2 que tenha $C(k) = d((1.324718)^k - 1)$ nós, basta que o algoritmo sempre utilize o Subcaso 2.2. Então, a complexidade de tempo do Algoritmo B2 é $O(kn + 1.324718^k k^2)$.

Exemplo de Execução do Algoritmo

Na Figura 3.22 apresentamos um exemplo de execução do Algoritmo B2 proposto por Balasubramanian *et al.* [1]. Seja o grafo G apresentado na Figura 3.22(a), queremos determinar se existe uma cobertura por vértices para o grafo G de tamanho menor ou igual a 8, ou seja, $k = 8$.

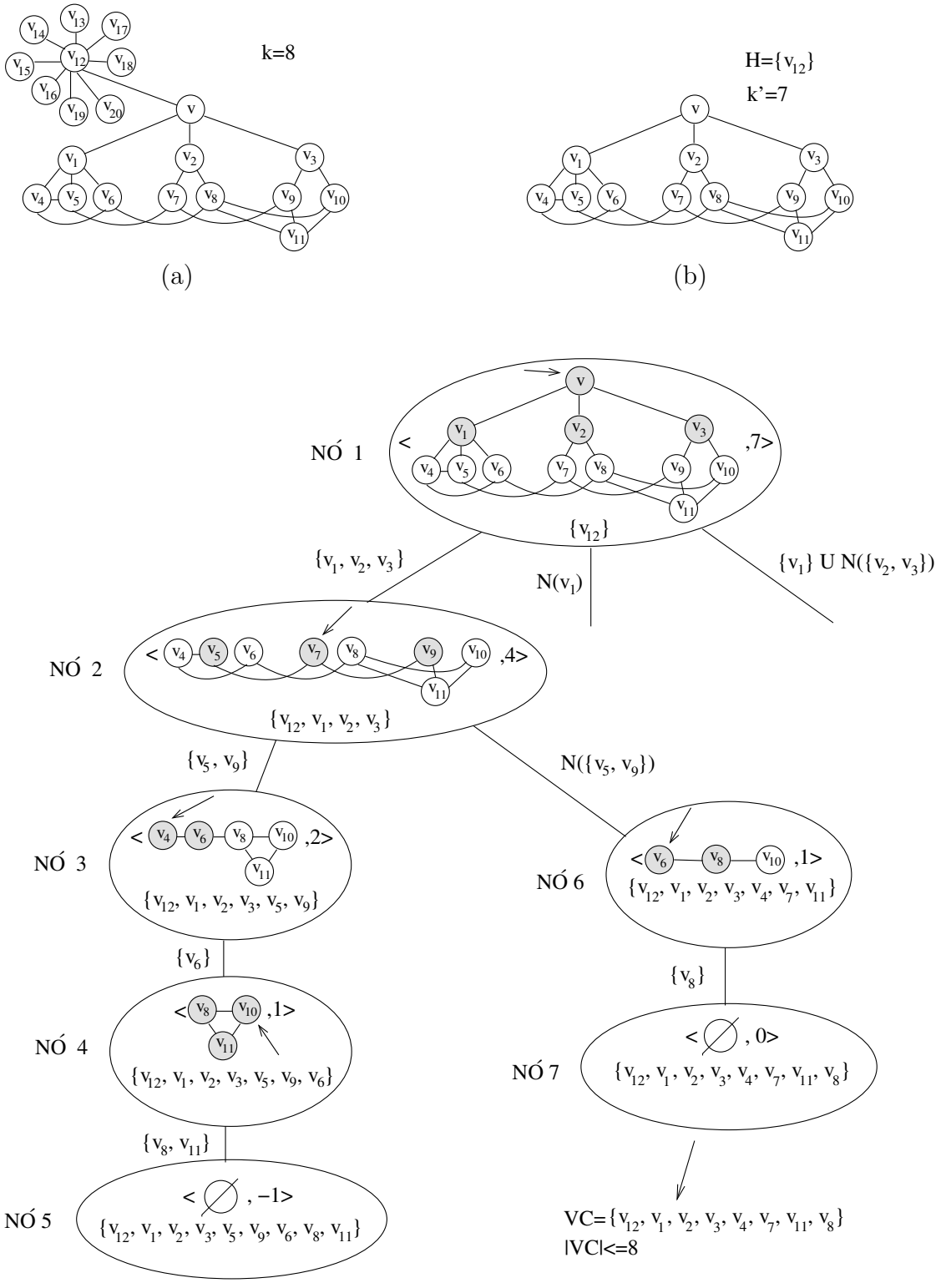


Figura 3.22: Execução do Algoritmo B2. (a) Grafo G. (b) Grafo G'. (c) Árvore limitada de busca do Algoritmo B2. O vértice selecionado pelo seu grau (apontado por uma seta) e seus vizinhos estão hachurados.

A fase de redução ao núcleo do problema resulta no grafo G', apresentado na Figura 3.22(b), no qual removemos os vértices de G de grau maior que 8 (v_{12}), bem como as arestas neles incidentes e quaisquer vértices isolados ($v_{13}, v_{14}, v_{15}, v_{16}, v_{17}, v_{18}, v_{19}, v_{20}$). O vértice v_{12} é adicionado ao conjunto H porque sabemos que ele pertence a qualquer cobertura por vértices de tamanho menor ou igual a 8 para o grafo G. Com isso, o valor de k' é 7.

Mostramos a árvore limitada de busca gerada pelo Algoritmo B2 na Figura 3.22(c). Como visto, este algoritmo não garante uma árvore ternária, pois os casos analisados por esse algoritmo podem gerar diferentes números de filhos para cada nó da árvore. O nó raiz da árvore de busca armazena a instância resultante da redução ao núcleo do problema ($\langle G', 7\rangle$) e uma cobertura parcial composta por $\{v_{12}\}$. O primeiro nó da árvore que visitamos é exatamente o nó raiz (NÓ 1). Como o grafo contido nesse nó não tem vértices de grau 1, grau 2 e grau maior ou igual a 5, então selecionamos um vértice de grau 3, v por exemplo. Temos o Subcaso 4.3 que resulta em três possibilidades de adição de vértices à cobertura parcial, portanto três nós filhos são gerados. O primeiro filho (NÓ 2) é visitado, pois sempre estamos fazendo busca em profundidade na árvore.

O grafo contido no NÓ 2 resulta da remoção dos vértices v_1, v_2 e v_3 do grafo contido no NÓ 1, bem como das arestas neles incidentes e quaisquer vértices isolados (v). Como v_1, v_2 e v_3 são adicionados à cobertura parcial, podemos acrescentar no máximo mais quatro vértices na cobertura e ainda obter uma de tamanho válido. Como o grafo contido nesse nó não tem vértices de grau 1, selecionamos um vértice de grau 2, v_7 por exemplo. Temos o Subcaso 2.2 que resulta em duas possibilidades de adição de vértices à cobertura parcial, portanto dois nós filhos são gerados. O primeiro filho (NÓ 3) é visitado.

O grafo contido no NÓ 3 resulta da remoção dos vértices v_5 e v_9 do grafo contido no NÓ 2, bem como das arestas neles incidentes e quaisquer vértices isolados (v_7). Como v_5 e v_9 são adicionados à cobertura parcial, podemos adicionar no máximo mais dois vértices na cobertura e ainda obter uma de tamanho válido. Como o grafo contido nesse nó tem um vértice de grau 1 (v_4), vamos selecioná-lo. Temos o Caso 1 que resulta em uma possibilidade de adição de vértices à cobertura parcial, portanto um nó filho é gerado. O NÓ 4 é visitado.

O grafo contido no NÓ 4 resulta da remoção do vértice v_6 do grafo contido no NÓ 3, bem como das arestas nele incidente e quaisquer vértices isolados (v_4). Como v_6 é adicionado à cobertura parcial, podemos adicionar no máximo mais um vértice na cobertura e ainda obter uma de tamanho válido. Como o grafo contido nesse nó não tem vértices de grau 1, selecionamos um vértice de grau 2, v_{10} por exemplo. Temos o Subcaso 2.1 que resulta em uma possibilidade de adição de vértices à cobertura parcial, portanto um nó filho é gerado. O NÓ 5 é visitado.

O grafo contido no NÓ 5 resulta da remoção dos vértices v_8 e v_{11} do grafo contido no NÓ 4, bem como das arestas neles incidentes e quaisquer vértices isolados (v_{10}). Os vértices v_8 e v_{11} são adicionados à cobertura parcial, mas a cobertura fica com mais de 8 elementos. Então, interrompemos o crescimento da árvore nesse nó e vamos para o próximo nó da busca em profundidade na árvore, o NÓ 6.

O grafo contido no NÓ 6 resulta da remoção dos vértices vizinhos de v_5 e v_9, ou

seja, v_4, v_7 e v_{11}, do grafo contido no NÓ 2, bem como das arestas neles incidentes e quaisquer vértices isolados ($v5$). Como v_4, v_7 e v_{11} são adicionados à cobertura parcial, podemos adicionar no máximo mais um vértice na cobertura e ainda obter uma cobertura de tamanho válido. Como o grafo contido nesse nó tem vértices de grau 1, podemos selecionar v_6, por exemplo. Temos o Caso 1 que resulta em uma possibilidade de adição de vértices à cobertura parcial, portanto um nó filho é gerado. O NÓ 7 é visitado.

O grafo contido no NÓ 7 resulta da remoção do vértice v_8 do grafo contido no NÓ 6, bem como das arestas nele incidentes e quaisquer vértices isolados (v_6 e v_{10}). Como v_8 é adicionado à cobertura parcial, não podemos mais adicionar vértices na cobertura e ainda obter uma cobertura de tamanho válido. No entanto, como o grafo contido nesse nó é vazio, significa que sua cobertura parcial formada por $\{v_{12}, v_1, v_2, v_3, v_4, v_7, v_{11}, v_8\}$ é uma cobertura por vértices de tamanho menor ou igual a 8 para o grafo G.

3.4 Algoritmo de Cheetham *et al.*

O algoritmo BSP/CGM proposto por Cheetham *et al.* [6] paraleliza ambas as fases do algoritmo FPT seqüencial, redução ao núcleo do problema e árvore limitada de busca. Trabalhos anteriores desenvolvidos no modelo PRAM paralelizavam apenas o método de redução ao núcleo do problema [6]. A paralelização da fase de árvore limitada de busca é uma contribuição de grande importância desse algoritmo BSP/CGM, pois as implementações geralmente gastam minutos na fase de redução ao núcleo do problema, e horas, ou até mesmo dias, na fase de árvore limitada de busca.

ALGORITMO: Paralelização da redução ao núcleo do problema (Cheetham)
Entrada: o grafo $G = (V, E)$ e um inteiro positivo k.
Saída: uma instância $\langle G' = (V', E'), k' \rangle$ ou uma cobertura por vértices de tamanho no máximo k ou uma mensagem se tal cobertura não existir.

1. Simule a fase de redução ao núcleo do problema descrito pelo algoritmo de Buss [2] em $O(1)$ ordenações paralelas de inteiros, usando ordenação determinística por amostragem [5].

2. Devolva $\langle G', k' \rangle$ ou uma cobertura por vértices de tamanho menor ou igual a k para G ou escreva "Não existe a cobertura por vértices desejada".

A idéia básica da fase de redução ao núcleo do problema proposta no algoritmo de Buss [2], já discutida na Seção 3.2, é que os vértices de grau maior que k pertencem a qualquer cobertura por vértices de tamanho menor ou igual a k para o grafo G. Tais vértices são adicionados em uma cobertura parcial e removidos do grafo G, bem como as arestas neles incidentes e quaisquer vértices isolados, resultando no grafo G'.

Seja n a quantidade de vértices e m a quantidade de arestas do grafo G. Na fase de redução paralela ao núcleo do problema, cada processador P_i, $0 \leq i \leq p-1$ é responsável por calcular o grau de n/p vértices. Como os vértices do grafo são rotulados de 0 a $n-1$,

o processador P_i é responsável por computar o grau dos vértices rotulados de $i * (n/p)$ a $((i+1) * (n/p)) - 1$.

Cada processador P_i recebe m/p arestas da lista de arestas do grafo G e ordena-as pelo primeiro vértice no qual incide. Assim é possível identificar o grau de tais vértices nesse processador. É possível que um processador calcule o grau de vértices que não são sua responsabilidade, portanto envia tais informações para os devidos processadores. Depois dessa comunicação, os processadores são capazes de computar precisamente o grau dos vértices sob sua responsabilidade.

No passo seguinte, cada processador P_i informa aos demais processadores quais os vértices locais de grau maior que k. Assim, todos os processadores são capazes de remover as arestas que incidem em tais vértices. Por fim, as arestas restantes em cada processador são enviadas aos demais, de forma que cada processador passa a armazenar o mesmo grafo resultante da redução ao núcleo do problema.

ALGORITMO: Paralelização da árvore limitada de busca (Cheetham)
Entrada: o grafo $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho no máximo k, se existir; ou uma mensagem, se tal cobertura por vértices não existir.

1. Seja a árvore ternária completa T gerada pelo Algoritmo B1, sendo que tal algoritmo é executado de forma determinística e usando busca em largura até que existam p folhas $\gamma_0 ... \gamma_{p-1}$. O nó raiz de T armazena $\langle G', k' \rangle$ e uma cobertura parcial (vértices com grau maior que k). Cada processador P_i, $0 \leq i \leq p-1$, percorre o caminho único que leva da raiz de T até a folha γ_i. Sejam $\langle G''_i, k''_i \rangle$, as instâncias associadas a cada folha γ_i da árvore T.

2. Cada processador P_i executa localmente o Algoritmo B2 sequencial a partir da folha γ_i, gerando uma subárvore cujo nó raiz é a folha γ_i, da seguinte forma:

   Sempre que houver ramificação em um nó de sua subárvore, o processador P_i escolhe aleatoriamente um dos nós filhos gerados e executa o algoritmo recursivamente até que uma solução seja encontrada. Se atingir uma folha de sua subárvore limitada de busca, o processador P_i volta na subárvore e escolhe outro nó filho ainda não explorado. Esse processo é repetido até encontrarmos uma solução (nesse caso temos que notificar os outros processadores para terminarem) ou até que a subárvore de raiz γ_i seja completamente percorrida.

No início da fase de árvore limitada de busca paralela, todos os processadores armazenam a mesma cobertura por vértices parcial (conjunto de vértices com grau maior que k) e a mesma instância reduzida ($\langle G', k' \rangle$), resultantes da redução paralela ao núcleo do problem. Essa situação representa o nó raiz da árvore limitada de busca.

É importante destacar que a árvore ternária e completa T, com p folhas (γ_0 a γ_{p-1}) e de altura $\log_3 p$ do Passo 1 não é explicitamente criada pelos processadores. Na verdade, cada processador P_i, $i \leq 0 \leq p-1$, computa apenas os nós do caminho único entre a raiz de T até a folha γ_i. A árvore T é fruto da sobreposição imaginária dos caminhos únicos computados por todos os processadores.

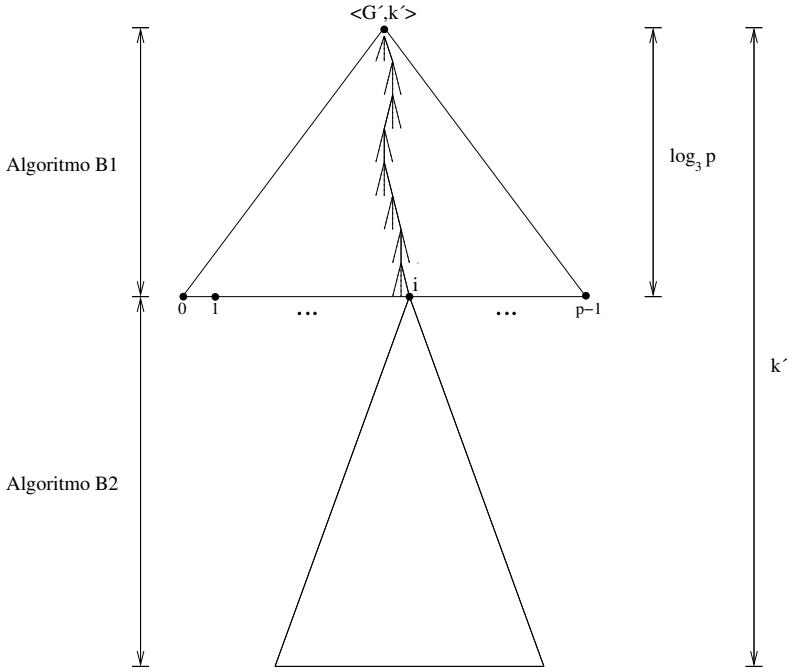


Figura 3.23: O processador P_i processa o caminho único até a folha γ_i usando o Algoritmo B1. Depois, P_i processa toda a subárvore de raiz γ_i usando o Algoritmo B2.

Para que todos os processadores computem a mesma árvore T, a escolha do vértice inicial da busca em profundidade que passa por no máximo três arestas do grafo seja determinística. Ou seja, todos os processadores que computam um dado nó da árvore T devem escolher o mesmo vértice inicial da busca em profundidade no grafo.

Em um nó da árvore de busca, cada processador P_i sabe qual nó filho deve visitar para percorrer o caminho único entre a raiz de T e a folha γ_i. Seja h o nível do nó atual na árvore e f a parte inteira resultante de $i/(p/3^{h+1})$. Se o resto da divisão de f por 3 for zero, o processador P_i visita o primeiro filho. Se for 1, visita o segundo filho. Se for 2, visita o terceiro filho. O Algoritmo B1 paralelo é interrompido quando o processador P_i atinge um nó de nível $\log_3 p$ na árvore.

Depois dessas operações, cada processador P_i está trabalhando com a instância $\langle G_i'', k_i'' \rangle$, referente à folha γ_i da árvore T. Então, todos os processadores executam seqüencialmente o Algoritmo B2 até encontrar uma solução ou percorrer toda subárvore de raiz γ_i. Note que se um processador encontrar uma solução, ele deve notificar os outros processadores para que terminem. Se todos os processadores percorrerem completamente suas subárvores limitadas de busca, significa que não existe cobertura por vértices de tamanho menor ou igual a k para o grafo G. Na Figura 3.23 mostramos a participação do processador P_i na fase de árvore limitada de busca paralela.

Com vários pontos iniciais de busca na árvore limitada de busca T (folhas $\gamma_i ... \gamma_{p-1}$), é provável que o algoritmo paralelo visite nós que nunca seriam visitados pelo algoritmo seqüencial. Com o algoritmo de Cheetham *et al.* [6] consegue-se resolver instâncias do problema da k-Cobertura por Vértices ainda maiores do que aquelas resolvidas pelos

algoritmos FPT seqüenciais. O problema da k-Cobertura por Vértices é considerado bem resolvido para $k \leq 200$ no método FPT seqüencial [17]. A implementação do algoritmo FPT paralelo feita por Cheetham *et al.* [6] resolve instâncias de $k \geq 400$ em menos de 75 minutos. Esse aumento no valor de k é significante, pois o tempo seqüencial cresce exponencialmente em relação à k.

3.5 Notas

Neste capítulo apresentamos algoritmos FPT para o problema da k-Cobertura por Vértices que são utilizados em nossa implementação. Começamos mostrando o algoritmo seqüencial de Buss [2] que descreve a fase de redução ao núcleo do problema. A seguir, mostramos os algoritmos seqüenciais de Balasubramanian *et al.* [1], que apresentam duas opções distintas de árvore limitada de busca. Para finalizar, apresentamos o algoritmo BSP/CGM proposto por Cheetham *et al.* [6] e que utilizamos em nossa implementação. Este algoritmo tem como principal contribuição a paralelização de ambas as fases do algoritmo FPT seqüencial: redução ao núcleo do problema e árvore limitada de busca.

Capítulo 4

Implementação

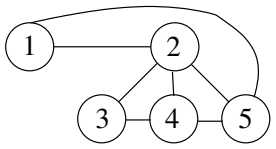
4.1 Introdução

Neste capítulo mostramos as estruturas de dados utilizadas em nosso programa que resolve o problema da k-Cobertura por Vértices, bem como os detalhes de sua implementação. Implementamos o algoritmo FPT paralelo de Cheetham *et al.* [6] no modelo BSP/CGM, usando a linguagem C/C++ em conjunto com a biblioteca de comunicações MPI. Na Seção 4.2 descrevemos a entrada do programa. Depois, na Seção 4.3 discutimos a implementação da fase de redução ao núcleo do problema e na Seção 4.4 discutimos a implementação da fase de árvore limitada de busca.

4.2 Entrada do Programa

O programa recebe como entrada um arquivo texto e um número inteiro. O arquivo texto descreve o grafo G por sua lista de adjacências e o número inteiro (k) determina o tamanho máximo desejado para uma cobertura por vértices para o grafo G. Seja n o número de vértices do grafo G, m o número de arestas e p o número de processadores em que o programa é executado. Para facilitar a identificação dos processadores, cada processador recebe um rótulo P_i, $0 \leq i \leq p-1$. Apenas o processador P_0 lê o arquivo de entrada.

Na Figura 4.1 apresentamos um grafo e um exemplo de arquivo de entrada que o descreve. Note que em cada linha temos uma seqüência de números inteiros que representa um vértice e sua lista de vértices adjacentes. O primeiro número é o identificador de um vértice. Do segundo ao penúltimo número temos os identificadores dos vértices adjacentes ao primeiro. O número -1 marca o final da lista de adjacências deste vértice. Por exemplo, os vértices 1, 3, 4 e 5 são adjacentes ao vértice 2.




```
1 2 5 -1
2 1 3 4 5 -1
3 2 4 -1
4 2 3 5 -1
5 1 2 4 -1
```

Figura 4.1: Um grafo e o arquivo de entrada que o descreve.

4.3 Redução ao Núcleo do Problema

Na implementação da fase de redução ao núcleo do problema, o grafo lido como lista de adjacências pelo processador P_0 é transformada em uma lista de arestas. Seja n o número de vértices do grafo e m o número de arestas. A lista de arestas que representa o grafo é armazenada na memória em um vetor de tamanho n do tipo `KERNEL_EDGES`. A estrutura é definida da seguinte forma:

```
struct KERNEL_EDGES
{
    int vbeg_fake;
    int vbeg;
    int vend;
}
```

Cada elemento do tipo `KERNEL_EDGES` representa uma aresta do grafo G e armazena o identificador dos vértices no qual incide (`vbeg` e `vend`). O identificador auxiliar `vbeg_fake` rotula os vértices iniciais de cada aresta de 0 a $n-1$. A necessidade deste campo é mostrada mais adiante.

O vetor de arestas é distribuído em blocos entre os processadores, de forma que cada um deles recebe um subvetor de tamanho n/p.

Cada processador Pi, $0 \leq i \leq p-1$, é responsável por calcular o grau de n/p vértices, rotulados de $i * (n/p)$ a $((i+1) * (n/p)) - 1$ (`vbeg_fake`). Os processadores armazenam esses vértices em vetores de tamanho n/p do tipo `KERNEL_VERTDEG`. A estrutura é definida da seguinte forma:

```
struct KERNEL_VERTDEG
{
    int vid;
    int degree;
}
```

Cada elemento do tipo `KERNEL_VERTDEG` representa um vértice do grafo G e armazena seu identificador (`vid`) e grau (`degree`).

Como vimos na Seção 3.2, a idéia do método de redução ao núcleo do problema proposto por Buss [2] é remover do grafo G os vértices de grau maior que k, bem como

as arestas neles incidentes e quaisquer vértices isolados. Isso porque os vértices de G de grau maior que k pertencem a qualquer cobertura por vértices para o grafo G de tamanho menor ou igual a k. Em nossa implementação, cada processador deve ordenar suas arestas locais pelo vértice inicial (`vbeg`) no qual incidem. O objetivo desta ordenação é facilitar a determinação do grau desses vértices. Porém, em nossa implementação as arestas já estão agrupadas por `vbeg` devido à conversão da lista de adjacências em lista de arestas.

Note que é possível que um processador calcule o grau de vértices que não são sua responsabilidade, portanto cada processador P_i envia uma mensagem para cada um dos demais processadores. Essa mensagem é um vetor de inteiros de tamanho n/p que armazena o grau dos vértices calculados por P_i e que são responsabilidade do processador destinatário.

Depois de calcular efetivamente o grau dos vértices que são de sua responsabilidade no grafo G, cada processador gera um vetor de inteiros de tamanho no máximo k com os identificadores dos vértices locais de grau maior que k. Então, envia tal informação para todos os demais processadores.

Assim, todos os processadores conhecem os vértices do grafo G de grau maior que k e são capazes de remover as arestas locais que incidem em tais vértices. Depois dessas remoções, as arestas restantes em cada processador são enviadas aos demais processadores. Portanto, no final da fase de redução ao núcleo do problema, cada processador P_i armazena um mesmo vetor de arestas do tipo `KERNEL_EDGES` (grafo G') e um mesmo vetor de inteiros de tamanho no máximo k com a identificação dos vértices que pertencem à cobertura parcial (conjunto H). O vetor de arestas armazenado em cada processador corresponde ao vetor de arestas original de G sem aquelas que incidem em vértices de grau maior que k.

4.4 Árvore Limitada de Busca

No início da fase de árvore limitada de busca, a lista de arestas que representa o grafo G' é transformada em uma lista de adjacências. Na memória, temos uma lista duplamente encadeada de elementos do tipo `VERTEX` que representa a lista de vértices do grafo. A estrutura é definida da seguinte forma:

```
struct VERTEX
{
    int identifier;
    int degree;
    EDGE* incident_edges;
    VERTEXDEGREE* link_degree;
    VERTEX* prev;
    VERTEX* next;
}
```

Cada elemento do tipo `VERTEX` representa um vértice do grafo G e armazena o seu identificador (`identifier`) e grau (`degree`). O campo `incident_edges` aponta para uma lista duplamente encadeada de elementos que representam as arestas incidentes neste vértice, a qual, por questões de simplicidade, chamamos de lista de arestas do vértice. O campo `link_degree` aponta para seu representante na lista de graus, cuja importância descreveremos mais adiante. Além disso, um elemento do tipo `VERTEX` também armazena os ponteiros para os elementos anterior e posterior na lista (`prev e next`).

A lista de arestas do vértice é uma lista duplamente encadeada de elementos do tipo `EDGE`. Essa lista sempre é apontada por um vértice. A estrutura é definida da seguinte forma:

```
struct EDGE
{
    VERTEX* end_vertex;
    EDGE* inverse;
    EDGE* prev;
    EDGE* next;
}
```

Cada elemento do tipo `EDGE` representa uma aresta incidente no vértice. Apesar de trabalharmos com um grafo não-dirigido, cada aresta do grafo é representada duas vezes em nossa estrutura de dados. Assim, a aresta (a, b) é representada na lista de arestas do vértice a (e o campo `end_vertex` aponta para o vértice b), enquanto a aresta (b, a) é representada na lista de arestas do vértice b (e o campo `end_vertex` aponta para o vértice a). O campo `inverse` é usado para ligar essas representações da mesma aresta. Além disso, um elemento do tipo `EDGE` também armazena os ponteiros para os elementos anterior e posterior na lista (`prev e next`). Na Figura 4.2 apresentamos um grafo e a estrutura de dados usada para armazená-lo na memória.

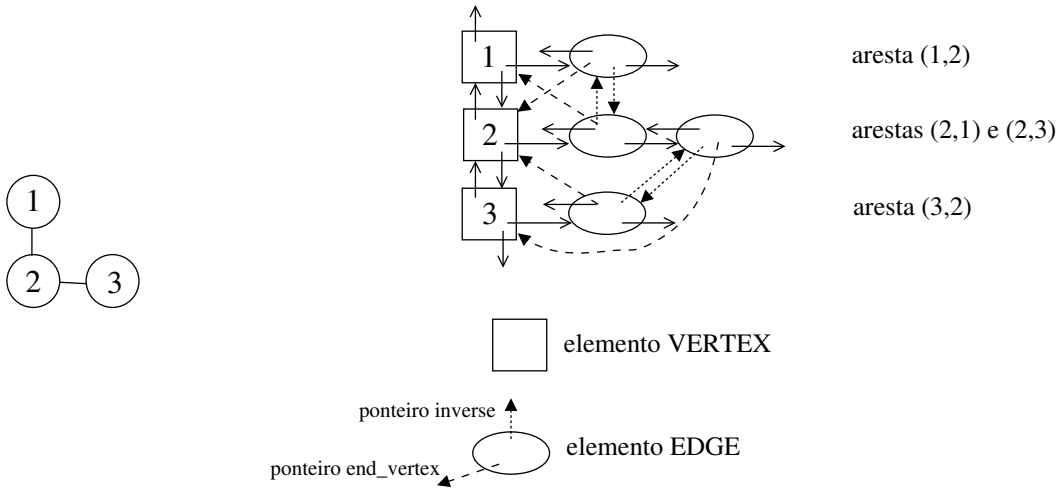
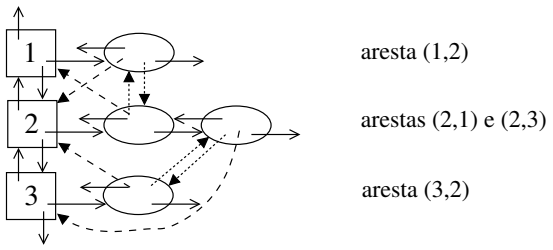


Figura 4.2: Um grafo e a estrutura de dados que o armazena na memória.

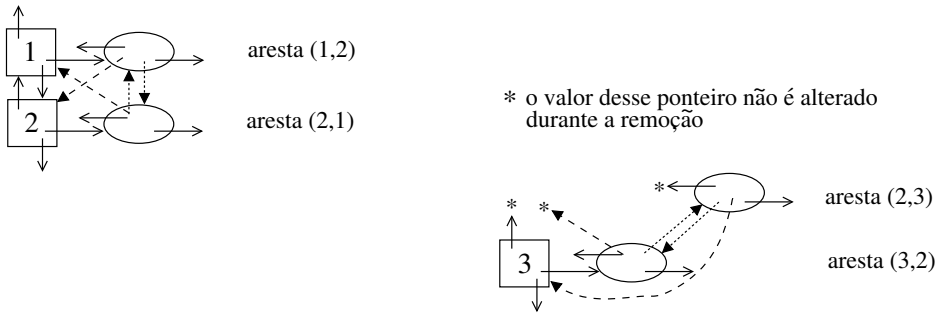
A inserção de um novo elemento na lista de vértices gasta tempo $O(n)$, pois precisamos verifica se tal elemento já não foi inserido. A inserção de um novo elemento na lista de arestas de um vértice, caso ele ainda não exista, resulta na inserção de elementos nas listas de arestas dos vértices de suas duas extremidades e também gasta tempo $O(n)$.

A remoção de vértices e arestas de suas respectivas listas é simplesmente um rearranjo dos ponteiros dos elementos anterior e posterior a eles. Note que eles não são desalocados da memória, são apenas "retirados" da lista. Quando um vértice é removido do grafo, todas as arestas nele incidentes também são. No entanto, ainda temos que remover de nossa estrutura de dados a outra representação dessas arestas. Como cada aresta incidente no vértice tem um ponteiro para sua outra representação, gasta-se tempo $O(1)$ para removê-la da lista de arestas do outro vértice. Se esse outro vértice fica isolado, adicionamo-o em uma lista de vértices isolados para removê-lo posteriormente. Portanto, gastamos tempo $O(k)$ para remover um vértice da lista, uma vez que os graus dos vértices do grafo são limitados a k.

Na Figura 4.3(a) apresentamos a estrutura de dados que armazena um grafo na memória antes da remoção do vértice 3. Na Figura 4.3(b) mostramos a estrutura de dados após a remoção do vértice 3. A lista de arestas do vértice 3 é removida junto com o vértice. A aresta $(2, 3)$ também é removida. Tais elementos não são desalocados da memória. Houve um rearranjo no ponteiro `next` do vértice 2 e no ponteiro `next` da aresta $(2, 1)$. Os ponteiros dos elementos removidos não sofrem alterações durante o processo.



(a) Estrutura de dados antes da remoção do vértice 3.



(b) Estrutura de dados após a remoção do vértice 3. Note que o vértice 3 e sua lista de arestas, bem como a aresta $(2, 3)$ não são desalocados da memória.

Figura 4.3: Estrutura de dados antes e depois da remoção do vértice 3.

Em nossa implementação, armazenamos na memória apenas os dados referentes ao nó atual da árvore de busca, ou seja, apenas um nó da árvore é processado de cada vez. Para computar um nó filho do nó atual, alguns vértices e arestas determinados são removidos do grafo. No entanto, como fazemos busca em profundidade na árvore, em algum ponto da execução do programa pode ser necessário voltar na árvore e recuperar um estado do grafo, anterior a essas remoções. Para isso, aplicamos a técnica de *backtracking*. Temos uma pilha com elementos do tipo `BACKTRACKING` para controlar a recuperação de estados anteriores do grafo. A estrutura é definida da seguinte forma:

```
struct BACKTRACKING
{
    VERTEX* vertex;
    int degree_before;
    EDGE* edge;
    VERTEX* begin_vertex;
    char group;
    BACKTRACKING* link;
}
```

Cada elemento do tipo `BACKTRACKING` armazena ou um ponteiro para um vértice removido (`vertex`) ou um ponteiro para uma aresta removida (`edge`). No caso da remoção de um vértice, ainda temos que armazenar seu grau no momento da remoção (`degree_before`). No caso da remoção de uma aresta, armazenamos um ponteiro para o vértice que contém tal aresta em sua lista de arestas (`begin_vertex`). O campo `group` é utilizado para agrupar os vértices e arestas que devem retornar ao grafo em um mesmo passo de *backtracking*, ou seja, vértices e arestas removidos do grafo contido no nó atual da árvore têm um `group` diferente dos vértices e arestas removidos do grafo contido no nó pai. O campo `link` é o ponteiro para ligar os elementos da pilha. Para adicionar ou remover um elemento na pilha gasta-se tempo $O(1)$.

É importante destacar que a "recuperação" de um vértice ou aresta removido é novamente uma questão de rearranjo dos ponteiros dos elementos anterior e posterior a eles na lista. Cada elemento removido contém informações sobre sua posição original na lista (ponteiros para o elemento anterior e posterior). Como cada elemento na pilha de *backtracking* tem um ponteiro para o elemento removido, gasta-se tempo $O(1)$ em cada recuperação.

A cobertura por vértices parcial também é uma pilha, mas com elementos do tipo `VERTEXCOVER`. A estrutura é definida da seguinte forma:

```
struct VERTEXCOVER
{
    VERTEX* vertex;
    BACKTRACKING* link;
}
```

Cada elemento do tipo `VERTEXCOVER` armazena um ponteiro para o vértice que faz parte da cobertura parcial (`vertex`) e um ponteiro para ligar os elementos da pilha (`link`). Adicionar ou remover um elemento na pilha da cobertura parcial gasta tempo $O(1)$. No início da fase de árvore limitada de busca, a pilha do tipo `VERTEXCOVER` recebe os vértices pertencentes à cobertura parcial obtida na fase de redução ao núcleo do problema (conjunto H).

Como vimos, em nossa implementação armazenamos na memória apenas os dados referentes ao nó atual da árvore de busca, ou seja, as estruturas de dados que representam o grafo, a cobertura por vértices parcial e a pilha de *backtracking*. Implementamos duas funções, uma para o Algoritmo B1 e outra para o Algoritmo B2. A execução dessas funções sugere ramificações para o nó atual. Para processar um nó filho do nó atual, adicionamos os vértices escolhidos na cobertura parcial e os removemos do grafo, bem como as arestas neles incidentes e quaisquer vértices isolados. Então, executamos a função recursivamente no grafo resultante. Quando a execução do programa atinge uma folha da árvore limitada, podemos voltar nos nós da árvore de busca utilizando a pilha de *backtracking* até atingir um nó que ainda tenha filhos não visitados. Em ambos os algoritmos, se o nó da árvore tem um grafo vazio e o parâmetro é positivo, então a cobertura parcial é uma cobertura por vértices de tamanho menor ou igual a k para o grafo G.

O Algoritmo B1 é executado em todos os processadores após o final da fase de redução ao núcleo do problema. Lembremos que todos os processadores contêm o mesmo grafo G' e a mesma cobertura por vértices parcial. No Algoritmo B1, a escolha dos vértices a adicionar na cobertura parcial é realizada de acordo com uma busca em profundidade feita a partir de um vértice qualquer do grafo que passe por no máximo três arestas. Entretanto, para que todos os processadores gerem uma mesma árvore T, ternária e completa, com p folhas rotuladas de γ_0 a γ_{p-1}, é necessário que a escolha desse vértice inicial seja determinística, portanto usamos sempre o primeiro vértice da lista de vértices do grafo. Cada processador P_i, $0 \leq i \leq p-1$, percorre o caminho único entre a raiz de T até a folha γ_i. Este caminho único é garantido conforme a descrição a seguir. Seja h o nível do nó atual na árvore e f a parte inteira resultante de $i/(p/3^{h+1})$. Nesse nó, se o resto da divisão de f por 3 for zero, o processador escolhe o primeiro filho para percorrer. Se for igual a 1, escolhe o segundo filho. E, se for igual a 3, escolhe o terceiro filho. A condição de parada para a execução do Algoritmo B1 é atingir um nó de nível $\log_3 p$ na árvore limitada de busca.

Após executar o Algoritmo B1, todos os processadores executam seqüencialmente o Algoritmo B2. Cada processador P_i executa a função do algoritmo B2 na folha γ_i da árvore T. Lembre-se que efetuamos uma busca em profundidade na subárvore cuja raiz é γ_i. A escolha dos vértices a adicionar na cobertura parcial é feita de acordo com os 5 casos que analisam o grau dos vértices do grafo, como vimos na seção 3.3. Por isso, no Algoritmo B2 são utilizados seis listas duplamente encadeadas auxiliares, do tipo `VERTEXDEGREE`, as quais organizam os vértices do grafo pelos seus graus (grau 0, grau 1, grau 2, grau 3, grau 4 e grau 5 ou mais). A estrutura é definida da seguinte forma:

```
struct VERTEXDEGREE
{
    VERTEX* link_vertex;
    VERTEXDEGREE* prev;
    VERTEXDEGREE* next;
}
```

Cada elemento do tipo `VERTEXDEGREE` armazena um ponteiro para o vértice na lista de vértices do grafo (`link_vertex`) e os ponteiros para os elementos anterior e posterior da lista do respectivo grau (`prev` e `next`). Como vimos, na cada elemento `VERTEX` tem um ponteiro `link_degree` que aposta para o seu representante na lista de graus. Na Figura 4.4 mostramos a ligação entre as listas de graus e a estrutura de dados que representa o grafo.

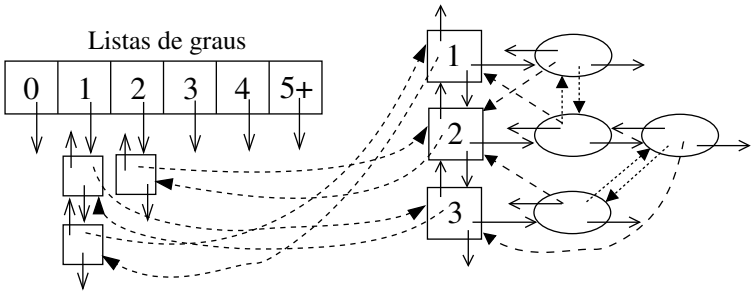


Figura 4.4: Ligação entre as listas de graus e a estrutura de dados que representa o grafo.

Essas seis listas auxiliares são importantes porque podemos selecionar um vértice do grau desejado em tempo $O(1)$. Além disso, como os vértices do grafo têm o ponteiro `link_degree`), em caso de mudança do grau do vértice gasta-se tempo $O(1)$ para mudar um vértice de lista de grau.

É importante lembrar que no algoritmo de Cheetham *et al.* [6] a escolha do nó filho a visitar é feita aleatoriamente. Somente após fazer uma busca em profundidade em toda a subárvore desse nó filho, escolhe-se outro filho inexplorado para visitar. Seja x a quantidade de nós filhos gerados no nó pai. Em um dado nó da árvore, para escolher o nó filho a visitar, nossa implementação rotula os nós filhos ainda não visitados de 0 a $x - 1$ e sorteia aleatoriamente um número nesse intervalo.

4.5 Notas

Neste capítulo mostramos os detalhes da implementação e as estruturas de dados usados em nosso programa paralelo, o qual foi inspirado no algoritmo FPT de Cheetham *et al.* [6], no modelo BSP/CGM. Nosso programa para resolver o problema da k-Cobertura por Vértices foi implementado na linguagem C/C++ em conjunto com a bibioteca de comunicações MPI. Foram descritos a entrada do problema, a implementação da fase de redução ao núcleo do problema e da fase de árvore limitada de busca.

Capítulo 5

Resultados Experimentais

Neste capítulo apresentamos os resultados experimentais obtidos por nossa implementação BSP/CGM do algoritmo FPT de Cheetham *et al.* [6] que usa as estruturas de dados discutidas no Capítulo 4. Na Seção 5.1 apresentamos o ambiente computacional e a metodologia utilizadas para computar os resultados obtidos nos experimentos. Na Seção 5.2 mostramos os grafos de entrada utilizados nos experimentos. Na Seção 5.3 discutimos os resultados obtidos por nossas implementações seqüencial e paralela, bem como comparamos tais resultados com os obtidos por Cheetham *et al.* [6].

5.1 Ambiente Computacional e Metodologia

O algoritmo paralelo FPT no modelo BSP/CGM de Cheetham *et al.* [6], apresentado na Seção 3.4, foi implementado na linguagem C/C++ em conjunto com a biblioteca de comunicações MPI e a chamamos de **Par-Impl**. Implementamos também um programa sequencial correspondente ao Algoritmo B2 de Balasubramanian *et al.* [1], descrito na Seção 3.3, na linguagem C/C++ e a chamamos de **Seq-Impl**.

O ambiente computacional utilizado em nossos experimentos foi um *cluster* Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM, pertencente à UNICAMP (Universidade Estadual de Campinas). Todos os nós estavam interconectados por um *switch* Fast Ethernet. Cada nó executava o sistema operacional Linux Red Hat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Os tempos obtidos por Seq-Impl foram medidos em segundos[1], incluindo o tempo para leitura dos dados de entrada, desalocação das estruturas utilizadas e impressão da saída. Os tempos decorridos por Par-Impl também foram medidos em segundos entre o início do primeiro processo até o final do último processo (incluindo as operações de entrada e saída e desalocação das estruturas utilizadas), o qual chamamos de tempos paralelos.

[1] *wall clock times*

5.2 Grafos de Entrada

Os grafos utilizados em nossos experimentos foram gentilmente cedidos pelo Professor Frank Dehne (Carleton University). Para gerar os grafos referentes aos aminoácidos Somatostatin, WW, Kinase, SH2 (*src-homology domain 2*) e PHD (*pleckstrin homology domain*), seqüências desses aminoácidos foram coletados do banco de dados do NCBI (http://www.ncbi.nlm.nih.gov).

No alinhamento múltiplo de sequências, uma idéia para resolver os conflitos entre as seqüências é remover da amostra algumas das seqüências conflitantes. Definimos que ocorre um conflito entre duas seqüências quando a pontuação de alinhamento entre ambas está abaixo de um valor pré-definido. Por exemplo, o programa ClustalW [32] gera o alinhamento dois a dois entre as seqüências e as pontua. Dessa forma, podemos gerar um grafo onde cada vértice é uma seqüência e toda aresta é um conflito entre duas seqüências. Esse grafo é chamado de **grafo de conflitos**. Portanto, o objetivo é remover o menor número possível de seqüências da amostra que eliminem todos os conflitos entre elas, ou seja, encontrar a cobertura por vértices mínima para o grafo de conflitos.

Na Tabela 5.1 apresentamos um resumo das características dos grafos utilizados em nossos experimentos, que incluem o nome do aminoácido, número de vértices ($|V|$), número de arestas ($|E|$), tamanho da cobertura por vértices desejada (k) e tamanho da cobertura a procurar após a redução ao núcleo do problema (k').

Grafo	$\|V\|$	$\|E\|$	k	k'
Kinase	647	113122	495	391
PHD	670	147054	601	600
SH2	730	95463	461	397
Somatostatin	559	33652	272	254
WW	425	40182	322	318

Tabela 5.1: Resumo das características dos grafos usados nos experimentos.

5.3 Comparação de Resultados

Nesta seção discutimos os resultados obtidos em nossos experimentos com Seq-Impl e Par-Impl. Além disso, vamos comparar nossos resultados com aqueles obtidos por Cheetham *et al.* [6]. Os valores que originaram nossos gráficos são apresentados em tabelas no Apêndice A.

Na Figura 5.1 comparamos os tempos obtidos por Seq-Impl e Par-Impl em um único processador para os grafos PHD, Somatostatin e WW. Escolhemos esses grafos porque Seq-Impl resolve-os em tempo razoável. Cada valor no gráfico referente ao algoritmo seqüencial equivale ao tempo médio de 10 experimentos. Usamos o modo de simulação do MPI/LAM para executar Par-Impl em um único processador. Nesse modo, o MPI/LAM simula processadores virtuais a partir de processos individuais em um mesmo processador físico. O tempo seqüencial de Par-Impl equivale a soma dos tempos dos processos

individuais mais o *overhead* criado pela comunicação deles. Em nossa comparação executamos Par-Impl em três processadores virtuais e cada valor no gráfico referente a essa implementação equivale ao tempo médio de 30 experimentos.

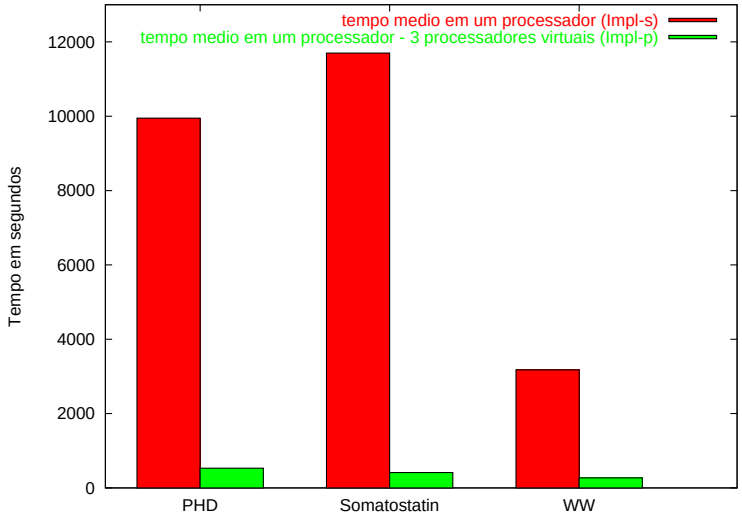


Figura 5.1: Comparação dos tempos seqüenciais e paralelos em um processador.

Observamos na Figura 5.1 que os tempos médios obtidos pela execução de Par-Impl no modo de simulação (3 processadores virtuais) do MPI/LAM foram bem menores do que os tempos médios obtidos por Seq-Impl, provavelmente porque Seq-Impl explora a árvore de busca a partir de um único ponto inicial, enquanto Par-Impl a explora a partir de p pontos iniciais. Dessa forma, é provável que Par-Impl visite nós que Seq-Impl nunca visitaria. Podemos observar também que os tempos de Seq-Impl não aumentam com k ou k', o que nos leva à conclusão de que a estrutura do grafo é um fator importante para o desempenho da implementação seqüencial.

Na Figura 5.2 comparamos os tempos obtidos pela execução de Seq-Impl e Par-Impl (3 processadores virtuais) em um único processador e Par-Impl em 27 processadores. Os experimentos foram feitos para os grafos PHD, Somatostatin e WW porque seus tempos seqüenciais são razoáveis. Cada valor no gráfico referente à Seq-Impl equivale ao tempo médio de 10 experimentos. Cada valor no gráfico referente à Par-Impl equivale ao tempo médio de 30 experimentos. Como existe uma grande variação de valores no gráfico, usamos uma escala $\log_2$.

Na Figura 5.3 mostramos os tempos obtidos pela execução de Par-Impl em 27 processadores para os grafos Kinase, PHD, SH2, Somatostatin e WW. Cada valor no gráfico equivale ao tempo médio de 30 experimentos. Nossa implementação FPT paralela consegue resolver instâncias do problema da k-Cobertura por Vértices com $k \geq 400$ em menos de 3 minutos. Destacamos o grafo PHD com $k = 601$, o qual é resolvido, em média, em menos de 1 minuto. O problema da k-Cobertura por Vértices é considerado bem resolvido

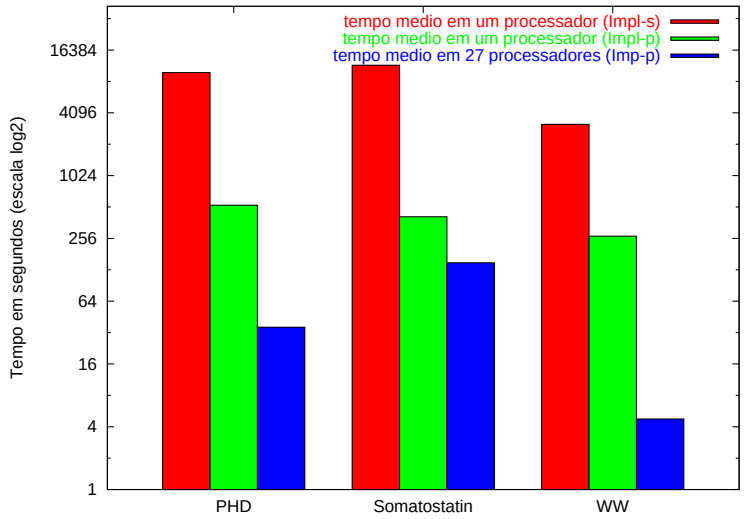


Figura 5.2: Comparação dos tempos seqüenciais e paralelos em um processador, e paralelos em 27 processadores.

para $k \leq 200$ no método FPT seqüencial. Esse aumento no valor de k é significante, pois o tempo seqüencial cresce exponencialmente em relação a k. Como acontece em Seq-Impl, os tempos de Par-Impl não aumentam com k ou k', o que nos leva à conclusão de que a estrutura do grafo é também um fator importante para o desempenho da implementação paralela.

Par-Impl também foi executado em 1, 3, 9 e 27 processadores e calculamos o *speedup* relativo para os grafos PHD (Figura 5.4), Somatostatin (Figura 5.5)e WW (Figura 5.6). Cada valor no gráfico equivale ao tempo médio de 30 experimentos. O tempo seqüencial usado para o cálculo do *speedup* relativo foi o menor tempo médio obtido entre Seq-Impl e Par-Impl em um único processador. Para estes três grafos, usamos o tempo médio da execução de Par-Impl no modo de simulação (três processadores virtuais), como vimos na Figura 5.1.

Apesar da quantidade de processadores influenciar diretamente na quantidade de pontos iniciais de busca na árvore, um maior número de processadores envolvidos na execução de Par-Impl não garante a determinação de uma solução muito mais rapidamente. Além da estrutura do grafo, parece que a quantidade de nós que contêm soluções na árvore de busca também influencia no tempo de execução. Como fazemos busca em profundidade na árvore, uma má escolha de um nó filho a visitar obriga o programa a percorrer toda a subárvore desse nó filho antes de poder escolher outro nó filho.

Outro resultado interessante de nossos experimentos é que pudemos confirmar a obtenção da cobertura mínima para os grafos PHD, SH2, Somatostatin e WW em menos de 75 minutos, executando Par-Impl em 27 processadores. Para comprovar que a co-

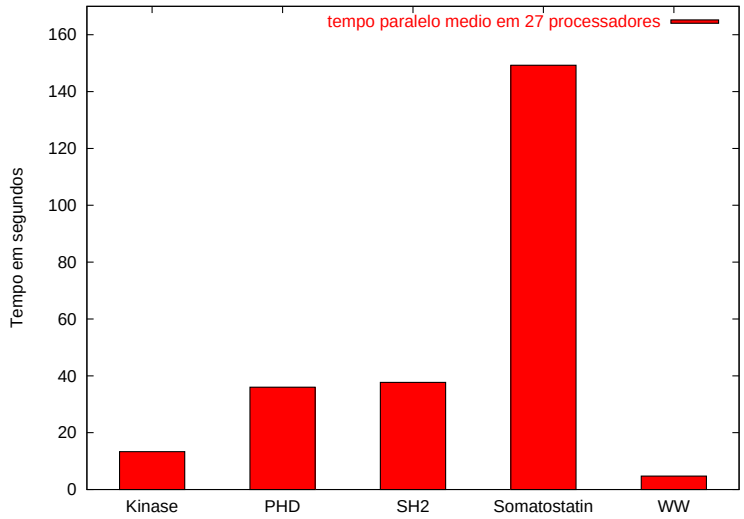


Figura 5.3: Comparação dos tempos paralelos em 27 processadores.

bertura era realmente mínima, executamos o programa para o valor de k uma unidade menor. Toda a árvore de busca foi percorrida sem encontrar uma solução, garantindo a minimalidade da cobertura encontrada. Na Tabela 5.2 apresentamos o tempo gasto pela nossa implementação para garantir a minimalidade da cobertura para os grafos PHD, SH2, Somatostatin e WW. Apenas para o grafo Kinase isso não foi possível em tempo razoável.

Grafo	Tempo
PHD	1.162,11 seg
SH2	4.374,14 seg
Somatostatin	272,102 seg
WW	664,25 seg

Tabela 5.2: Tempos gastos pela nossa implementação para garantir a minimalidade da cobertura.

Uma importante contribuição de nossa implementação foi encontrar coberturas por vértices de tamanho menor do que aqueles relatados por Cheetham *et al.* [6]. Encontramos coberturas por vértices de tamanho menor para os seguintes grafos: Kinase (de 497 para 495), PHD (de 603 para 601) e Somatostatin (de 273 para 272).

Em seus experimentos, Cheetham *et al.* [6] utilizaram o seguinte ambiente computacional, um *cluster* Beowulf de 32 processadores Xeon de 1.8 GHz, cada um com 512 MB de memória RAM, interligados por um *switch* Gigabit Ethernet. Cada nó executava o sistema operacional Linux Red Hat 7.2 com g++ 2.95.3 e MPI/LAM 6.5.6. Mesmo utilizando um ambiente computacional inferior, em nossos experimentos obtivemos tempos paralelos melhores. Consideramos que a escolha das estruturas de dados e o uso da

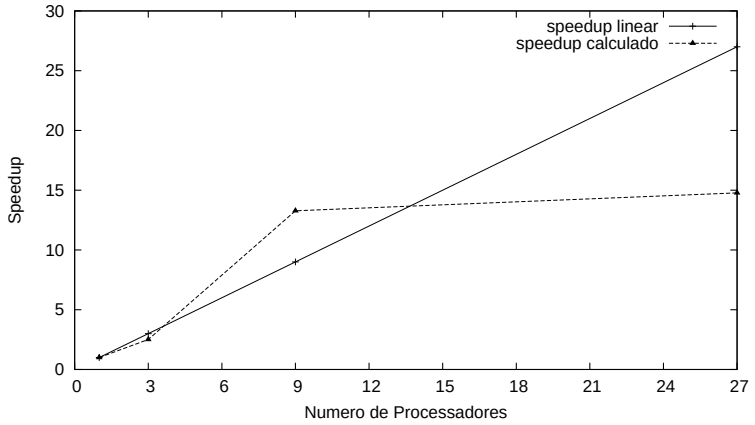


Figura 5.4: *Speedup* relativo para o grafo PHD.

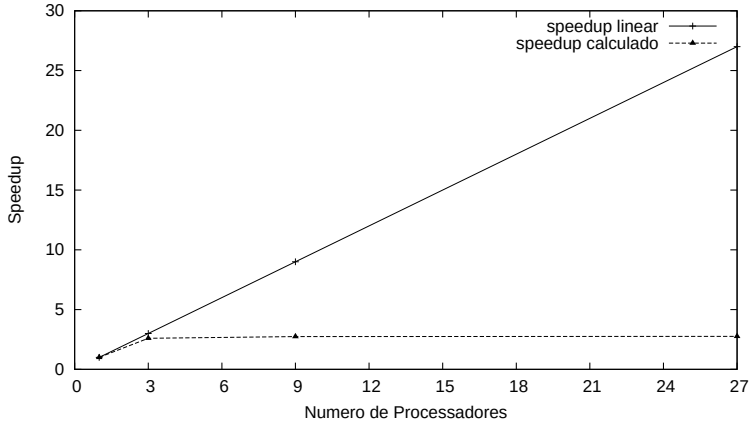


Figura 5.5: *Speedup* relativo para o grafo Somatostatin.

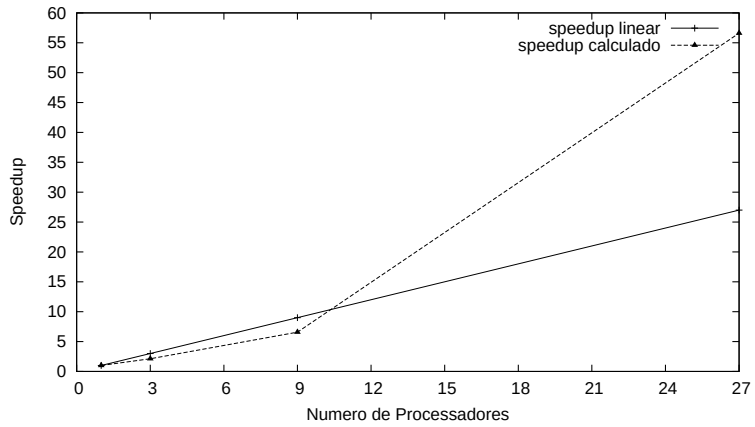


Figura 5.6: *Speedup* relativo para o grafo WW.

técnica de *backtracking* foram fundamentais para a obtenção de nossos bons resultados.

Na Tabela 5.3 apresentamos os tempos médios obtidos pela implementação paralela de Cheetham *et al.* [6] (valores aproximados) e a nossa implementação paralela em 27 processadores, para os grafos Kinase, PHD, SH2, Somatostatin e WW.

Grafo	Cheetham*	Nossa Implementação	Taxa de Melhoria
Kinase	1550 seg	13,3273 seg	119,23
PHD	600 seg	36,0201 seg	16,65
SH2	4400 seg	37,7083 seg	116,68
Somatostatin	150 seg	149,2843 seg	1
WW	10 seg	4,7455 seg	2,1

* valores aproximados

Tabela 5.3: Tempos médios obtidos pela implementação de Cheetham *et al.* [6] e a nossa para 27 processadores.

Como podemos ver na Tabela 5.3, para os grafos Kinase e SH2 obtivemos resultados muito melhores, com tempos aproximadamente 115 vezes melhores. Para o grafo PHD obtivemos tempo aproximadamente 16 vezes melhor, e para os grafos Somatostatin e WW os tempos foram um pouco melhores.

Entretanto, no caso dos grafos Kinase, PHD e Somatostatin, ainda temos que considerar a diminuição no tamanho da cobertura por vértices em relação àquela apresentada por Cheetham *et al.* [6], já que isso implica na redução do universo de soluções existentes na árvore limitada de busca e provoca um aumento no tempo necessário para encontrar uma solução.

Capítulo 6

Conclusão

As técnicas de tratabilidade por parâmetro fixo (*fixed-parameter tractability* ou FPT) têm sido utilizadas com sucesso na solução de instâncias de problemas NP-completos de grande importância prática, como o problema da k-Cobertura por Vértices, por exemplo. Tais problemas computacionais têm a entrada dividida em duas partes: uma parte principal e um parâmetro. O objetivo é fazer com que a parte principal da entrada contribua polinomialmente para a complexidade total do problema, enquanto que a aparentemente inevitável explosão combinatorial fique confinada ao parâmetro. Seja n o tamanho da entrada e k o parâmetro, um problema parametrizável é FPT se existe um algoritmo que o resolve em tempo $O(f(k)n^{\alpha})$, onde α é uma constante e f é uma função arbitrária.

Como vimos, o problema da k-Cobertura por Vértices tem grande importância prática. Em Biologia Computacional, por exemplo, pode ser usado na análise de alinhamentos múltiplos de seqüências. O problema da k-Cobertura por Vértices foi um dos primeiros problemas que provou-se ser FPT. Na Tabela 6.1 apresentamos alguns algoritmos FPT conhecidos que o resolvem.

Autor(es)	**Complexidade de tempo**
Buss [2]	$O(kn + 2^k k^{2k+2})$
Downey e Fellows [12]	$O(kn + 2^k k^2)$
Balasubramanian *et al.* [1] (Alg. B1)	$O(kn + (\sqrt{3})^k k^2)$
Balasubramanian *et al.* [1] (Alg. B2)	$O(kn + 1.324718^k k^2)$
Downey, Fellows e Stege [17]	$O(kn + 1.31951^k k^2)$
Niedermeider e Rossmanith [29]	$O(kn + 1.29175^k k^2)$
Fellows e Stege [18]	$O(kn + max(1.25542^k k^2, 1.2906^k 2.5^k))$
Chen, Jia e Kanj [7]	$O(kn + 1.271^k k^2)$

Tabela 6.1: Algoritmos FPT para o problema da k-Cobertura por Vértices

Os algoritmo FPT têm sido implementados e obtido sucesso na solução de instâncias de problemas NP-completos que antes eram consideradas muito grandes para serem resolvidas pelos métodos já existentes. No entanto, a complexidade exponencial do parâmetro

ainda pode resultar em custos proibitivos. O problema da k-Cobertura por Vértices é considerado bem resolvido para $k \leq 200$ pelo método FPT seqüencial. Adicionando paralelismo às técnicas de FPT, Cheetham *et al.* [6] descreveram um algoritmo paralelo no modelo BSP/CGM que resolve o problema da k-Cobertura por Vértices em menos de 75 minutos para $k \geq 400$. Esse aumento no valor de k é significante, já que o tempo seqüencial cresce exponencialmente em relação a k. O ganho do algoritmo paralelo em relação ao algoritmo seqüencial deve-se ao fato do algoritmo paralelo explorar a árvore limitada de busca em mais pontos iniciais do que o algoritmo seqüencial.

Em nossa implementação do algoritmo de Cheetham *et al.* [6] obtivemos tempos paralelos melhores do que os relatados pelos próprios autores, mesmo utilizando um ambiente computacional inferior. Para alguns grafos conseguimos melhorar o tempo paralelo em aproximadamente 115 vezes. Consideramos que a escolha das estruturas de dados e o uso da técnica de *backtracking* em nossa implementação foram fundamentais para a obtenção de nossos bons resultados. Na Tabela 6.2 temos um resumo dos ganhos obtidos por nossa implementação.

Grafo	Cheetham*	Nossa Implementação	Taxa de Melhoria
Kinase	1550 seg	13,3273 seg	119,23
PHD	600 seg	36,0201 seg	16,65
SH2	4400 seg	37,7083 seg	116,68
Somatostatin	150 seg	149,2843 seg	1
WW	10 seg	4,7455 seg	2,1

* valores aproximados

Tabela 6.2: Tempos médios obtidos pela implementação de Cheetham *et al.* [6] e a nossa para 27 processadores.

Outra contribuição deste trabalho foi encontrar coberturas por vértices de tamanho menor para três grafos (Kinase, PHD e Somatostatin) do que aqueles informados por Cheetham *et al.* [6].

Este trabalho também resultou no artigo *Efficient Implementation of the BSP/CGM Parallel Vertex Cover FPT Algorithm*, de Hanashiro, Mongelli e Song [21], aceito no *III Workshop on Efficient and Experimental Algorithms* - WEA/2004 e a ser publicado na série *Lecture Notes in Computer Science*.

Em relação aos resultados de Cheetham *et al.* [6], apenas para o grafo Thrombin não conseguimos obter tempos paralelos melhores, em média. Apenas 1 de cada 10 experimentos eram resolvidos em tempo aceitável por Par-Impl. Com o intuito de melhorar esse desempenho, ainda implementamos duas outras versões para o programa. Em uma versão, tornamos aleatória a escolha do vértice inicial da busca em profundidade que passa por no máximo três arestas do grafo no Algoritmo B1. Na outra versão, também tornamos aleatória a escolha dos vértices de um determinado grau do grafo no Algoritmo B2. Apesar de melhorar um pouco a freqüencia de soluções do grafo Thrombin em tempo aceitável, 4 em 10, tais implementações pioravam consideravelmente o tempo obtido nos outros grafos, o que justifica sua não utilização.

Trabalhos futuros incluem a substituição do algoritmo executado localmente por cada processador na segunda fase da árvore limitada de busca em nosso algoritmo paralelo por um algoritmo mais eficiente. Por exemplo, podemos substituir o Algoritmo 2 de Balasubramanian *et al.* [1], o qual gasta tempo $O(1.324818^k k^2)$ na fase da árvore limitada de busca, pelo algoritmo de Niedermeier e Rossmanith [29], que gasta tempo $O(1.29175^k k^2)$ e também é baseado em casos que levam em consideração o grau dos vértices restantes no grafo.

Apêndice A

Tabelas

As tabelas listadas a seguir apresentam os tempos obtidos em nossos experimentos, executado em um *cluster* Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados por um *switch* Fast Ethernet. Cada nó executava o sistema operacional Linux Red Hat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Grafo	Seq-Impl	Par-Impl
PHD	9.946,7	532,214
Somatostatin	11.699,3	412,413
WW	3.174,7	268,611

Tabela A.1: Tempos em segundos obtidos por Seq-Impl e Par-Impl (3 processadores virtuais) em um único processador. Os tempos de Seq-Impl equivalem ao tempo médio de 10 experimentos. Os tempos de Par-Impl equivalem ao tempo médio de 30 experimentos.

Grafo	Seq-Impl		Par-Impl	
	Maior	Menor	Maior	Menor
PHD	27.302,0	12,0	3.218,56	70,9218
Somatostatin	27.341,0	198,0	719,567	17,922
WW	10.943,0	30,0	1.345,1	13,119

Tabela A.2: Tempos em segundos obtidos por Seq-Impl e Par-Impl (3 processadores virtuais) em um único processador. Os tempos de Seq-Impl equivalem ao maior e menor tempo de 10 experimentos. Os tempos de Par-Impl equivalem ao maior e menor tempo de 30 experimentos.

Grafo	Seq-Impl	Par-Impl (1 processador)	Par-Impl (27 processadores)
PHD	9.946,7	532,214	36,0201
Somatostatin	11.699,3	412,413	149,284
WW	3.174,7	268,611	4,7455

Tabela A.3: Tempos em segundos obtidos por Seq-Impl e Par-Impl (3 processadores virtuais) em um único processador, e Par-Impl em 27 processadores. Os tempos de Seq-Impl equivalem ao tempo médio de 10 experimentos. Os tempos de Par-Impl equivalem ao tempo médio de 30 experimentos.

Grafo	Par-Impl (27 processadores)
Kinase	13,3273
PHD	36,0201
SH2	37.7083
Somatostatin	149,284
WW	4,7455

Tabela A.4: Tempos em segundos obtidos por Par-Impl em 27 processadores. Esses tempos equivalem ao tempo médio de 30 experimentos.

Grafo	Par-Impl (27 proc)	
	Maior	Menor
Kinase	13.9714	13.0917
PHD	38.6285	33.9702
SH2	398.931	12.3588
Somatostatin	363.475	4.14061
WW	7.55741	4.01567

Tabela A.5: Tempos em segundos obtidos por Par-Impl em 27 processadores. Esses tempos equivalem ao maior e menor tempo de 30 experimentos.

Processadores	Tempo Médio	*Speedup*
1	532.214	1,0
3	212.248	2.5075
9	40.0797	13.2789
27	36.0201	14.7755

Tabela A.6: *Speedup* relativo para o grafo PHD. Os tempos equivalem ao tempo médio de 30 experimentos.

Processadores	Tempo Médio	*Speedup*
1	412,413	1,0
3	158,817	2,5967
9	150,552	2,7393
27	149,284	2,7626

Tabela A.7: *Speedup* relativo para o grafo Somatostatin. Os tempos equivalem ao tempo médio de 30 experimentos.

Processadores	Tempo Médio	*Speedup*
1	268,611	1,0
3	126,21	2,1282
9	40,8968	6,5680
27	4,7455	56,6033

Tabela A.8: *Speedup* relativo para o grafo WW. Os tempos equivalem ao tempo médio de 30 experimentos.

Referências Bibliográficas

[1] R. Balasubramanian, M. R. Fellows, e V. Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65:163–168, 1998.

[2] J. F. Buss e J. Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[3] E. N. Cáceres, H. Mongelli, e S. W. Song. *Algoritmos paralelos usando CGM/PVM/MPI: uma introdução*, páginas 219–278. XXI Congresso da Sociedade Brasileira de Computação, Jornada de Atualização de Informática, 2001.

[4] E. N. Cáceres, H. Mongelli, e S. W. Song. *Topics in parallel algorithms using CGM/MPI*. SBAC-PAD 2002 (Tutorial), 2002.

[5] A. Chan e F. Dehne. A note on course grained parallel integer sorting. In *13th Annual Int. Symposium on High Performance Computers*, páginas 261–267. 1999.

[6] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, e P. J. Taillon. Solving large FPT problems on coarse grained parallel machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003.

[7] J. Chen, W. Jia, e I. A. Kanj. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[8] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[9] F. Dehne. Guest editor's introduction - coarse grained parallel algorithms. *Algorithmica*, 24(3/4):173–176, 1999.

[10] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of the ACM 9th Annual Computational Geometry*, páginas 298–307. 1993.

[11] R. G. Downey e M. R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM Journal on Computing*, 24:873–921, 1995.

[12] R. G. Downey e M. R. Fellows. Fixed-parameter tractability and completeness II: completeness for W[1]. *Theoretical Computer Science*, 141:109–131, 1995.

[13] R. G. Downey e M. R. Fellows. Parameterized computational feasibility. In *Feasible Mathematics II*, páginas 219–244. Birkhauser Boston, 1995.

[14] R. G. Downey e M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1998.

[15] R. G. Downey e M. R. Fellows. Parameterized complexity after (almost) 10 years: review and open questions. In *Combinatorics, Computation & Logic, DMTCS'99 and CATS'99*, volume 21, número 3, páginas 1–33. Australian Computer Science Communications, Springer-Verlag, 1999.

[16] R. G. Downey, M. R. Fellows, e U. Stege. Computational tractability: the view from Mars. *Bulletin of the European Association for Theoretical Computer Science*, 69:73–97, 1999.

[17] R. G. Downey, M. R. Fellows, e U. Stege. Parameterized complexity: a framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49 de *AMS-DIMACS Proceedings Series*, páginas 49–99. 1999.

[18] M. R. Fellows e U. Stege. An improved fixed-parameter-tractable algorithm for vertex cover. Relatório técnico, Department of Computer Science, ETH Zurich, 1999.

[19] M. R. Garey e D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[20] S. M. Götz. *Communication-efficient parallel algorithms for minimum spanning tree computation*. Tese de Doutoramento, University of Paderborn, 1998.

[21] E. J. Hanashiro, H. Mongelli, e S. W. Song. Efficient implementation of the bsp/cgm parallel vertex covert fpt algorithm. Aceito no III Workshop on Efficient and Experimental Algorithms (WEA'2004). A publicar como Lecture Notes in Computer Science.

[22] D. S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1997.

[23] M. Hofri. *Analysis of algorithms: computational methods and mathematical tools*. Oxford University Press, 1995.

[24] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company, 1989.

[25] Z. Michalewicz e D. B. Fogel. *How to solve it: Modern Heuristics*. Springer-Verlag, 2000.

[26] H. Mongelli. *Algoritmos CGM para busca uni e bidimensional de padrões com e sem escala*. Tese de Doutoramento, Universidade de São Paulo, 2000.

[27] R. Motwani e P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[28] C. Y. Nasu. *Algoritmo BSP/CGM para computação de circuitos de Euler em grafos*. Tese de Mestrado, Universidade Federal de Mato Grosso do Sul, 2002.

[29] R. Niedermeider e P. Rossmanith. Upper bounds for vertex cover further improved. In *Proceedings of the 16th Symposium on Theoretical Aspects in Computer Science (STACS'99)*, páginas 561–570. 1999.

[30] R. Niedermeier. Some prospects for efficient fixed parameter algorithms. In *Conference on Current Trends in Theory and Practice of Informatics*, páginas 168–185. 1998.

[31] C. H. Papadimitriou e K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc, 1982.

[32] J. D. Thompson, D. G. Higgins, e T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[33] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[34] B. Wilkinson e M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, 1999.