Implementações alternativas FPT BSP/CGM para o problema k-Cobertura por Vértices

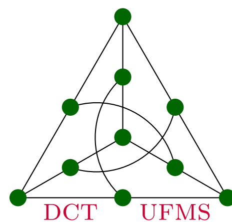
Deiviston da Silva Aguena

Dissertação de Mestrado

Orientação: Prof. Dr. Henrique Mongelli

Área de Concentração: Ciência da Computação

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, Curso de Pós-Graduação em Ciência da Computação, Departamento de Computação e Estatística da Fundação Universidade Federal de Mato Grosso do Sul.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Junho/2009

Implementações alternativas FPT BSP/CGM para o problema k-Cobertura por Vértices

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Deiviston da Silva Aguena e aprovada pela comissão julgadora.

Campo Grande, 22 de Junho de 2009.

Banca Examinadora:

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

aos meus pais Silvio e Nair

Agradecimentos

Em primeiro lugar a Deus, que além da oportunidade da vida, permitiu que eu tivesse saúde, paz, felicidade, disposição e perseverança nesta caminhada.

Ao meu orientador, professor Dr. Henrique Mongelli, por ter sempre acreditado em mim, por ter me ajudado e apoiado em cada desafio.

Aos professores Dr. Siang Song, e Dr. Edson Cáceres, por todo apoio oferecido, além do auxílio e parceria, juntamente com o professor Dr. Henrique Mongelli, para publicação de parte deste trabalho no congresso da Ásia em 2009.

A todos os professores do Mestrado pelos ensinamentos recebidos, em especial, aos professores Dr. Marcelo Henriques e Dr. José Craveiro pelas cartas de recomendação concedidas, que me levaram a ser aceito no programa de Mestrado da UFMS.

A todos os funcionários da Universidade Federal de MS que contribuíram direta ou indiretamente.

Aos meus companheiros de Mestrado, Rodrigo Sakamoto, principalmente pelas ajudas nas tomadas de tempos e Marcos Mariano, pelos grupos de estudos antes das provas.

Ao meu amigo Charles Viegas, por antes de tudo, ter me incentivado e apoiado a iniciar o Mestrado. Aos meus amigos de hoje e sempre, César Favorete, Clebiano Nogueira e Silvio Martins pelo apoio e torcida.

À toda minha família pelo carinho e apoio recebido durante toda vida, ao meu pai Silvio e a memória de minha mãe Nair.

À minha noiva Ana Paula pela dedicação, amor e compreensão.

A Deus.

"O maior bem que podemos fazer aos outros não é oferecer-lhes nossa riqueza, mas levá-los a descobrir a deles" Louis Lavelle

Resumo

Muitas das aplicações do mundo real requerem soluções para problemas *NP-Completos*. A inexistência de algoritmos polinomiais conhecidos para resolvê-los resulta na grande variedade de propostas de soluções. Estas soluções utilizam principalmente heurísticas e algoritmos de aproximação. Uma abordagem alternativa é a utilização de algoritmos *FPT*(*Fixed Parameter Tractability*). Enquanto as técnicas baseadas em heurísticas e em algoritmos de aproximação relaxam a busca por soluções ótimas ou exatas, mas usualmente insistem em algoritmos de tempo polinomial, as técnicas que utilizam algoritmos *FPT* sempre encontram resultados exatos, mas podem apresentar soluções eficientes na teoria, embora inviáveis na prática. Para controlar o tempo de processamento, os algoritmos *FPT* possuem um parâmetro k associado à instância do problema que resolvem. Neste sentido, pequenos valores configurados no parâmetro k produzem soluções polinomiais. Mas, como nem sempre, pequenos valores no parâmetro são suficientes para suprir a real necessidade de um problema, estratégias como utilizar o paralelismo tem sido pesquisadas com objetivo de melhorar tanto o tempo de resposta quanto o tamanho da instância do problema que pode ser resolvida. Neste trabalho, estaremos interessados na pesquisa de algoritmos *FPT* e na implementação eficiente destes algoritmos utilizando o poder do paralelismo no modelo BSP/CGM para diferentes abordagens do problema *NP-Completo* k-Cobertura por Vértices (k-*Vertex Cover*).

Palavras-chave: FPT, Parallel FPT, k-Vertex Cover.

Conteúdo

1 Introdução **12**

1.1 Organização do Texto 14

2 Revisão da Literatura **16**

2.1 Considerações iniciais 16

2.2 Complexidade Parametrizada e FPT 16

2.2.1 Complexidade Parametrizada e FPT 16

2.2.2 Redução ao Núcleo 18

2.2.3 Árvore Limitada de Busca 18

2.3 Modelos Paralelos 19

2.3.1 Modelos Realísticos BSP e CGM 19

2.4 Considerações finais 21

3 O problema da k-Cobertura por Vértices **22**

3.1 Considerações Iniciais 22

3.2 Algoritmo de Buss [3] 22

3.3 Algoritmo B1 de Balasubramanian *et al.* [1] 23

3.4 Algoritmo B2 de Balasubramanian *et al.* [1] 27

3.5 O Algoritmo de Cheetham *et al.* [6] 30

3.6 Algoritmo de Downey et al [14] 32

3.7 Algoritmo de Niedermeier e Rossmanith [32] 35

3.7.1 Vértices com grau 2 37

3.7.2 Vértices com grau 3 38

3.7.3 Vértices com grau 4 39

4 Estruturas de Dados e Implementação **42**

4.1 Considerações Iniciais 42

4.2 Implementação Cheetham-Downey 43

4.3 Implementação Cheetham-Niedermeier 45

4.4 Considerações Finais 46

5 Resultados Experimentais **48**

5.1 Considerações Iniciais 48

5.2 Ambiente Computacional e Metodologia 48

5.3 Grafos de Entrada 49

5.4 Comparação dos Resultados 49

5.5 Considerações Finais 62

6 Conclusão **63**

Referências Bibliográficas **66**

Lista de Figuras

1.1 Exemplo de uma cobertura por vértices. 14

2.1 Execução no Modelo BSP. 20

2.2 Execução no Modelo CGM 21

3.1 Exemplo de execução do Algoritmo de Buss [3]. 24

3.2 Exemplo de execução do Algoritmo B1 [1]. 26

3.3 Exemplo de execução do Algoritmo B2 [1]. 31

3.4 Exemplo de execução do Algoritmo de Cheetham [6]. 32

3.5 Exemplo de execução do Algoritmo de Downey [14]. 34

3.6 Exemplo de execução da Fase 2 do Algoritmo de Downey [14]. 36

3.7 Exemplo de execução do Algoritmo de Niedermeier e Rossmanith [32] . . . 41

4.1 Implementação de Cheetham [6] e novas versões. 43

4.2 Estrutura de dados *degree controller*. 44

4.3 Estrutura de dados *partial degree controller*. 45

4.4 Estrutura de dados Base: Lista de vértices e suas arestas. 46

5.1 Comparação *Cluster* x Grade Somatostatin - 3 Processadores 50

5.2 Comparação *Cluster* x Grade Somatostatin - 9 Processadores. 51

5.3 Comparação *Cluster* x Grade WW - 3 Processadores. 51

5.4 Comparação *Cluster* x Grade WW - 9 Processadores. 52

5.5 Comparação *Cluster* x Grade Kinase - 3 Processadores. 52

5.6 Comparação *Cluster* x Grade Kinase - 9 Processadores. 53

5.7 Comparação *Cluster* x Grade SH2 - 3 Processadores. 53

5.8 Comparação *Cluster* x Grade SH2 - 9 Processadores. 54
5.9 Comparação *Cluster* x Grade PHD - 3 Processadores. 54
5.10 Comparação *Cluster* x Grade PHD - 9 Processadores. 55
5.11 Comparação dos Grafos no *Cluster* - 3 Processadores. 56
5.12 Comparação dos Grafos no *Cluster* - 9 Processadores. 56
5.13 Comparação dos Grafos na Grade - 3 Processadores. 57
5.14 Comparação dos Grafos no Grade - 9 Processadores. 57
5.15 Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - Somatostatin. 58
5.16 Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - WW. 59
5.17 Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - Kinase. 59
5.18 Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - SH2. 60
5.19 Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - PHD. 60

Lista de Tabelas

2.1 Problemas e melhores resultados utilizando FPT. 18

5.1 Grafos utilizados nos experimentos e suas características 49

5.2 Speedup - Grade com 3 Processadores 61

5.3 Speedup - Grade com 9 Processadores 61

5.4 Speedup - *Cluster* com 3 Processadores 62

5.5 Speedup - *Cluster* com 9 Processadores 62

Lista de Abreviaturas e Siglas

ACO	Ant Colony Optimization Algorithm
BSP	Bulk Synchronous Parallel
CGM	Coarse Grained Multicomputer
CRCW	Concurrent Read Exclusive Write
CREW	Concurrent Read Exclusive Write
EREW	Exclusive Read Exclusive Write
FPT	Fixed Parameter Tratable
P	Polinomial Time
MIMD	Multiple Instruction stream - Multiple Data stream
MISD	Multiple Instruction stream - Single Data stream
MPI	Message Passing Interface
NP	Nondeterministic Polynomial Time
PRAM	Parallel Randon Access Machine
RAM	Randon Access Memory
VC	Vertex Cover
SAT	Satisability
SIMD	Single Instruction stream - Multiple Data stream
SISD	Single Instruction stream - Single Data stream

Capítulo 1

Introdução

Soluções para problemas computacionais utilizam recursos como tempo de processamento e memória. O quanto será necessário de cada um destes recursos varia conforme o grau de dificuldade para se resolver um determinado problema e o quão eficiente será a solução proposta. A teoria da complexidade concentra-se em aspectos quantitativos da computação e tem como tema central a medição da dificuldade das funções computacionais [26]. Problemas computacionais são classificados em classes de complexidade. Duas importantes classes de complexidade são: a classe *P*, formada pelo conjunto de problemas que podem ser resolvidos em tempo polinomial, e a classe *NP*, formada pelo conjunto de problemas para os quais podemos verificar, em tempo polinomial, se uma possível solução é correta. A classe *NP-completo* é um subconjunto da classe *NP* formada pelos seus problemas mais difíceis [8]. Problemas da classe *NP-completo* são tão difíceis que não existem algoritmos conhecidos, em tempo polinomial, para resolvê-los. Problemas que não possuem solução em tempo polinomial são ditos intratáveis.

Lidar com a intratabilidade tem sido um dos grandes desafios da Ciência da Computação. Muitos problemas importantes na área da Computação são intratáveis. Com o objetivo de tentar resolver estes problemas e trabalhar com a intratabilidade, métodos que utilizam principalmente aproximações e heurísticas têm sido propostos. Porém, ambos apresentam as desvantagens de não oferecer garantia quanto ao desempenho ou exatidão do resultado [33].

Neste trabalho, apresentamos a *parametrização* como uma alternativa a estes métodos e foi proposta para se trabalhar com a intratabilidade em alguns problemas. Alguns trabalhos propõem o uso da parametrização combinada com heurísticas e a utilização do paralelismo [5, 17, 18, 25]. A complexidade parametrizada foi principalmente desenvolvida por *Downey et al* [11, 12, 14, 15]. A diferença entre a complexidade clássica e a complexidade parametrizada é que a complexidade parametrizada admite um terceiro argumento além da entrada do problema e da questão a ser resolvida. Este argumento é chamado de *parâmetro* [15]. A principal idéia da complexidade parametrizada é tentar olhar mais profundamente a estrutura da entrada do problema, ou seja, através do parâmetro, isolar da entrada o componente que causa o tempo exponencial [6].

A seguir, apresentamos definições importantes da complexidade parametrizada, ne-

cessárias para que possamos formalmente apresentar uma definição de problema de acordo com a complexidade parametrizada.

Definição 1.1 (Linguagem parametrizada L) [16]
Uma linguagem parametrizada L é um subconjunto $L \subseteq \Sigma^* \times \Sigma^*$. Se L é uma linguagem parametrizada e $(x, y) \in L$ então nos referimos a x como parte principal e y como parâmetro.

Isto não cria nenhuma diferença na teoria e é ocasionalmente mais conveniente considerar y como um inteiro ou, de maneira equivalente, definir uma linguagem parametrizada sendo subconjunto de $\Sigma^* \times \mathbb{N}$.

Definição 1.2 (Linguagem parametrizada tratável por parâmetro fixo) [16]
Uma linguagem parametrizada L é tratável por parâmetro fixo, se puder ser determinado, se $(x, y) \in L$, em tempo $f(k)n^\alpha$, na qual f é uma função arbitrária, $|x| = n$ e α é uma constante independente de ambos n e k.

A familia das linguagens parametrizadas tratáveis por parâmetro fixo são denotadas FPT (do inglês *Fixed-Parameter Tractability*).

O problema da cobertura por vértices foi um dos primeiros problemas que provou-se ser tratável por parâmetro fixo [12]. Neste trabalho, focaremos como objeto de estudo, em sua forma parametrizada, o problema da Cobertura por Vértices. Este problema, segundo *Niedermeier e Rossmanith*, é um problema de central importância na Ciência da Computação e fazem as seguintes colocações [32]:

- ele está entre os primeiros problemas NP-Completos;
- existem numerosos esforços para se desenvolver algoritmos de aproximação, porém ele também é conhecido por ser um problema de difícil aproximação;
- é um problema de central importância da Complexidade Parametrizada, e possui um dos mais eficientes algoritmos FPT;
- é importante para diversas aplicações, como por exemplo, na Biologia Computacional, em que a Cobertura por Vértices é utilizada para resolver conflitos entre seqüencias.

Uma cobertura por vértice sobre um Grafo $G(V, E)$ é um conjunto de vértices V' de G tal que, para toda aresta (u, v) pertencente a E, pelo menos um entre u e v pertence a V'. O problema da cobertura mínima de vértices consiste em determinar para um Grafo $G(V, E)$ o menor subconjunto de vértices V' que formam uma cobertura em G. O problema da cobertura por vértices na versão parametrizada é conhecido por k-Cobertura por Vértices.

Definição 1.3 (Problema k-Cobertura por Vértices)
O problema da cobertura por vértices parametrizado conhecido como k-VERTEX COVER apresenta a seguinte forma:
Instância: Um Grafo *G(V,E)* e um inteiro positivo k.

Parâmetro: k.
Questão: O grafo G possui uma cobertura por vértices de tamanho menor ou igual a k?

A Figura 1.1 apresenta uma cobertura por vértices $V' = \{v_2, v_5, v_6, v_7\}$ de tamanho $k = 4$. A solução para este problema não é única. No exemplo, $V' = \{v_1, v_5, v_6, v_7\}$ também é uma cobertura, e possui tamanho $k = 4$.

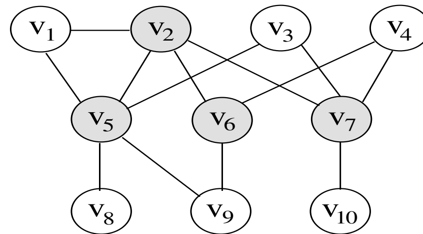


Figura 1.1: Exemplo de uma cobertura por vértices.

No mundo real, problemas de ordem prática podem ser difíceis de se resolver. De fato, grande parte destes problemas é intratável. A dificuldade em se encontrar soluções eficientes, ou seja, soluções em tempo polinomial, para problemas intratáveis como os da classe *NP-Completo* resulta em várias propostas de métodos. A utilização de algoritmos *FPT* é uma destas propostas. Algoritmos *FPT* têm sido utilizados com sucesso para resolver instâncias práticas de problemas *NP-Completos.* O uso do paralelismo aplicado aos algoritmos de *FPT* tem potencializado a pesquisa nesta área. O problema da cobertura de vértices é um dos 21 problemas originais *NP-Completos* que Karp reduziu do problema *SAT - satisability* [27]. O problema da cobertura por vértices é um problema importante por que tem aplicação em problemas de diversas áreas como: Banco de Dados, Redes de Computadores e Biologia Computacional.

Este trabalho tem como objetivos principais: (*i*) a aplicação do uso do paralelismo aliado a algoritmos *FPT* em implementações práticas; (*ii*) a investigação do comportamento de diferentes abordagens de parametrização e redução ao núcleo, para o problema da k-Cobertura por Vértices; (*iii*) o projeto e implementação de programas paralelos no modelo *BSP/CGM*; e (*iv*) a apresentação de implementações eficientes para o problema o problema da k-Cobertura por Vértices, assim como, a apresentação de resultados experimentais, para estas implementações, utilizando-se como ambiente *Cluster* e Grade Computacional.

Nas implementações utilizaremos a linguagem C e a biblioteca de comunicação MPI. Utilizaremos como direcionamento inicial a proposta apresentada em [1, 3, 7, 13, 32].

1.1 Organização do Texto

Este trabalho está organizado do seguinte modo: no Capítulo 2 são apresentados os principais conceitos de Complexidade, Complexidade Parametrizada, e do problema da

k-Cobertura por Vértices. No Capítulo 3, apresentamos alguns algoritmos FPT para o problema da Cobertura por Vértices. No Capítulo 4, são apresentados a proposta, a metodologia e o cronograma de atividades a serem desenvolvidas. Por fim, são apresentadas as referências bibliográficas que fundamentam este trabalho.

Capítulo 2

Revisão da Literatura

2.1 Considerações iniciais

Este capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados os principais conceitos da Teoria da Complexidade Parametrizada e a classe de problemas FPT. Conceitos básicos e Modelos computacionais são apresentados na Seção 2.3. Por fim na Seção 2.4 são apresentadas as considerações finais deste Capítulo.

2.2 Complexidade Parametrizada e FPT

2.2.1 Complexidade Parametrizada e FPT

Downey e *McCartim* [10] definem a Complexidade parametrizada como uma ferramenta que foi desenvolvida para se tentar mapear instâncias práticas de problemas na complexidade de algoritmos, ou seja, a idéia básica é que a explosão combinatorial que ocorre em algoritmos exatos, para vários problemas intratáveis, pode ser sistematicamente mapeada em função de parâmetros que consigam contê-la . A lógica é que pequenos e úteis intervalos destes parâmetros possam ser revolvidos eficientemente.

O que a Complexidade Parametrizada faz, é tentar olhar mais profundamente a estrutura da entrada do problema, isolar alguns aspectos ou partes da entrada como um *parâmetro* e confinar a aparentemente inevitável explosão combinatorial da dificuldade computacional em uma função aditiva sobre o *parâmetro* e outros custos de origem polinomial [13].

Como mencionado antes, a principal idéia da Tratabilidade por Parâmetros Fixos (FPT) é que, se um problema é parametrizável, então existe um um parâmetro k, cuja a solução do problema possa ser encontrada em tempo $O(f(k)n^{\alpha})$. Dado que, a natureza exponencial do problema esteja em k e não em n e, dependendo do problema, talvez k seja pequeno com relação a n, então, talvez, a natureza exponencial do problema pode ser tratada.

A Complexidade Parametrizada vem sendo aplicada em áreas como Banco de dados [22, 23], Inteligência Artificial [20] e Biologia Computacional [2, 21, 35].

Definições de problema parametrizável e parametrizável tratável por parâmetro fixo, de acordo *Downey* e *Fellows* são apresentados a seguir.

Definição 2.1 (Problema Parametrizável) [12] Um problema parametrizável é um conjunto $L \subseteq \Sigma^* \times \Sigma^*$ onde Σ é um alfabeto fixo.

Para um problema parametrizável L e $y \in \Sigma^*$ nós escrevemos L_y para denotar a associação do problema parametrizável (y é o parâmetro) $L_y = \{x|(x, y) \in L\}$. Nós nos referimos a L_y como a y-ésima fatia de L.

Definição 2.2 (Problema Parametrizável Tratável por Parâmetro Fixo)[12] Um problema parametrizável L é tratável por parâmetro fixo, se existe uma constante α e um algoritmo A que determina se $(x, y) \in L$ em tempo $f(|y|).|x|^{\alpha}$, na qual $f : N \rightarrow N$ é uma função arbitrária, x é a parte principal e y é o parâmetro. A classe de problemas tratáveis por parâmetro fixo é denominada FPT.

Problemas parametrizáveis que não são tratáveis por parâmetro fixo são chamados de intratáveis por parâmetro fixo. Para estudar e comparar a complexidade de problemas parametrizados, *Downey* e *Fellows* propõem a seguinte noção de redução de Problemas FPT de parâmetro fixo:

Definição 2.3 (Redução de Problemas FPT)[12] Uma redução uniforme de um problema parametrizável L para o problema parametrizável L', é um Oráculo (Algoritmo) A que, dada a entrada (x, y), determina se $x \in L_y$ e satisfaz:

- Existe uma função arbitrária $f : N \rightarrow N$ e um polinômio q, cujo tempo de A seja $f(|y|)q(|x|)$.
- Para cada $y \in \Sigma^*$ existe um conjunto finito $J_y \subseteq \Sigma^*$, na qual A consulta o Oráculo apenas para problemas FPT L'_w onde $w \in J_y$.

Se, para dois problemas parametrizáveis, L_1 e L_2, L_1 pode ser reduzido para L_2 e o oposto também é verdade, então podemos dizer que L_1 e L_2 são fixados por parâmetro mutuamente [29]. *Downey* e *Fellows* definem a hierarquia das classes de complexidade com a hierarquia definida como W [14]:

$$\text{FPT} \subseteq \text{W}[1] \subseteq \text{W}[2] \subseteq \text{W}[3] \subseteq ...$$

A classe W[t] pode ser descrita em termos de problemas que são completos para eles, ou seja, um problema D é completo para uma classe de complexidade C se $D \subseteq C$ e todo problema nesta classe pode ser reduzido para D [29].

A Tabela 2.1, apresentada a seguir, foi retirada do folheto informativo da Comunidade de Complexidade Parametrizada de maio de 2008 [34] e apresenta os melhores resultados obtidos para alguns problemas FPT.

Problem	$f(k)$	kernel
Vertex Cover	1.2738^k	$2k$
Feedback Vertex Set	5^k	k^3
Planar DS	$2^{1513\sqrt{k}}$	$67k$
1-Sided Crossing Min	1.4656^k	
Max Leaf	6.75^k	$4k$
Set Splitting	2^k	$2k$
Nonblocker	2.5154^k	$5k/3$
3-D Matching	2.77^{3k}	
Edge Dominating Set	2.4181^k	$8k^2$
k-Path	4^k	
Convex Recolouring	4^k	$O(k^2)$
VC-max degree 3	1.1899^k	

Tabela 2.1: Problemas e melhores resultados utilizando FPT.

2.2.2 Redução ao Núcleo

A idéia da redução ao núcleo (*Kernalization*) é, rapidamente, resolver algumas partes da instância do problema que são relativamente fáceis de se lidar. Em outras palavras, redução de dados em tempo polinomial. Em alguns cenários específicos, é possível que a redução de dados torne uma instância, de um problema difícil de se resolver, em uma instância trivial. De fato, o pré-processamento através de regras de redução de dados é apontado como a ferramenta central de desenvolvimento de algoritmos *FPT* [31]. No contexto *FPT*, o uso de redução de dados não é apenas visto como uma heurística, mas como uma forma de garantia de desempenho. A instância reduzida fica restrita a um limite superior que depende apenas do parâmetro.

Definição 2.3 (Redução de um problema ao núcleo) [13, 31] Seja I uma instância de um problema parametrizável e um dado parâmetro k. Uma redução do problema ao núcleo ou *kernalization* é um algoritmo, que em tempo polinomial, transforma I em uma nova instância I' e k em um novo parâmetro $k' \leq k$. Independente do tamanho de I, o tamanho de I' é determinado de acordo com a função em k. Além disso, a nova instância I' tem uma solução com respeito a k' se, e apenas se, a instância I tem uma solução com respeito ao parâmetro original k.

A redução do problema ao núcleo ajuda a entender efeitos práticos de regras de redução em instâncias práticas de um problema. Aliada a um conhecimento mais aprofundado de uma estrutura, regras de redução do problema ao núcleo tornam-se ferramentas poderosas para algoritmos *FPT*.

2.2.3 Árvore Limitada de Busca

Muitos problemas parametrizáveis podem ser resolvidos pela construção de um espaço de busca, usualmente uma árvore de busca, cujo tamanho depende apenas do crescimento

do parâmetro. Dado este espaço de busca, necessitamos apenas encontrar algoritmos eficientes para processar cada ponto deste espaço [30]. Uma das formas para se explorar o grande espaço de busca de soluções ótimas de problemas *NP* é realizando explorações sistemáticas e exaustivas sobre o espaço. Neste sentido, a principal contribuição da teoria *FPT* nesta área está em como a busca em árvores é considerada com relação à profundidade de busca, que é limitada sobre o parâmetro. Combinado com algum conhecimento prévio, mas não óbvio, de mecanismo mais eficiente de busca, o espaço para exploração das buscas torna-se muito menor do que utilizando-se um mecanismo ingênuo da força bruta.

2.3 Modelos Paralelos

2.3.1 Modelos Realísticos BSP e CGM

Leslie Valiant [36], em 1990, introduziu o Modelo BSP (*Bulk Synchronous Parallel*), que foi um dos primeiros modelos a considerar os custos de comunicação como característica de um modelo de computação paralela . As principais características do modelo proposto por Valiant é a consideração dos custos de comunicação e a abstração das características de uma máquina paralela através de um pequeno número de parâmetros.
Os parâmetros considerados no modelo BSP são [4]:

- n: tamanho do problema;
- p: número de processadores com memória local;
- L: tempo máximo de um super-passo(periodicidade);
- g: descreve a taxa de eficiência de computação e comunicação, que corresponde a razão entre:
  (a). o número total de operações de computação local de todos os processadores (em unidades básicas de computação) em uma unidade de tempo e,
  (b). o número total de mensagens enviadas/recebidas (em palavras) em unidade de tempo.

O modelo BSP pode ser definido como um conjunto de p processadores e uma seqüencia de super-passos, em que cada processador, em um super-passo, executa uma combinação de trocas de mensagens e computações locais e as seqüencias de super-passos são separadas por barreiras de sincronização. A Figura 2.1 apresenta um exemplo de processamento utilizando o modelo BSP.

O modelo CGM (Coarse Grained Multicomputer) ou Multicomputador de “granulidade grossa”foi apresentado por Dehne [9]. Este termo “granulidade grossa”foi introduzido devido ao fato que o tamanho do problema, a ser resolvido, é consideravelmente maior que o número de processadores, ou seja, $n/p >> p$ [4]. O modelo CGM é similar ao modelo BSP, contudo ele é definido com apenas dois parâmetros:

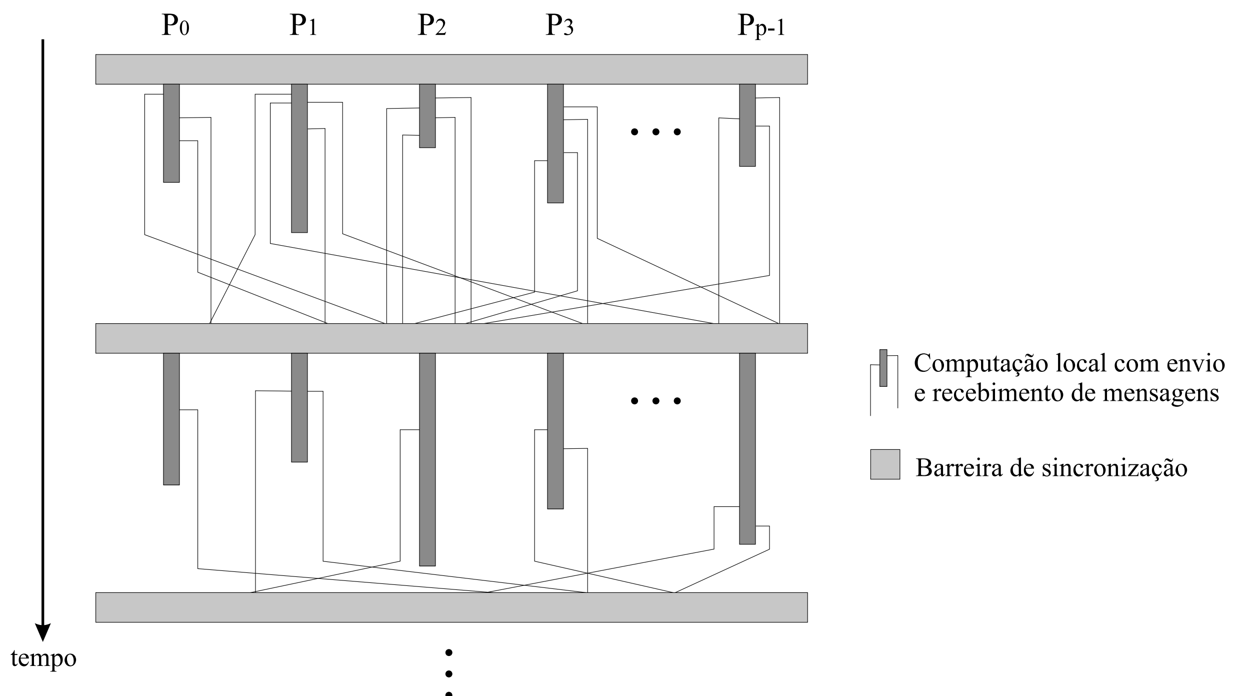


Figura 2.1: Execução no Modelo BSP.

- n: tamanho do problema;
- p: número de processadores com memória local do tamanho igual a n/p;

Apesar de, à primeira vista, notar-se a ausência das variáveis que antes eram utilizadas para contabilizar o custo de comunicação no modelo BSP e levar a conclusão que o modelo CGM não considera este custo, de fato o modelo CGM considera o custo de comunicação através do número de rodadas de comunicação. Em uma rodada de comunicação cada processador envia e recebe $O(n/p)$ dados. Por tratar o custo de comunicação de forma mais simples, o modelo CGM apresenta a vantagem de maior simplicidade no projeto de algoritmos.

A Figura 2.2 apresenta um exemplo de execução utilizando o modelo CGM.

No modelo CGM, o esforço para reduzir a comunicação está centralizado na redução do número de rodadas de comunicação.

O modelo *Coarsed Grained Multicomputer* (CGM) consiste em um conjunto de p processadores, cada qual com memória local de tamanho $O(n/p)$, conectados por uma rede de interconexão, em que n é o tamanho do problema e $n/p \geq p$. Este modelo é uma simplificação do modelo *Bulk Synchronous Parallel* (BSP) proposto por Valiant [36], que também é um modelo realístico, pois define parâmetros para mapear as principais características de máquinas paralelas reais, levando em consideração, dentre outras coisas, o tempo de comunicação entre os processadores.

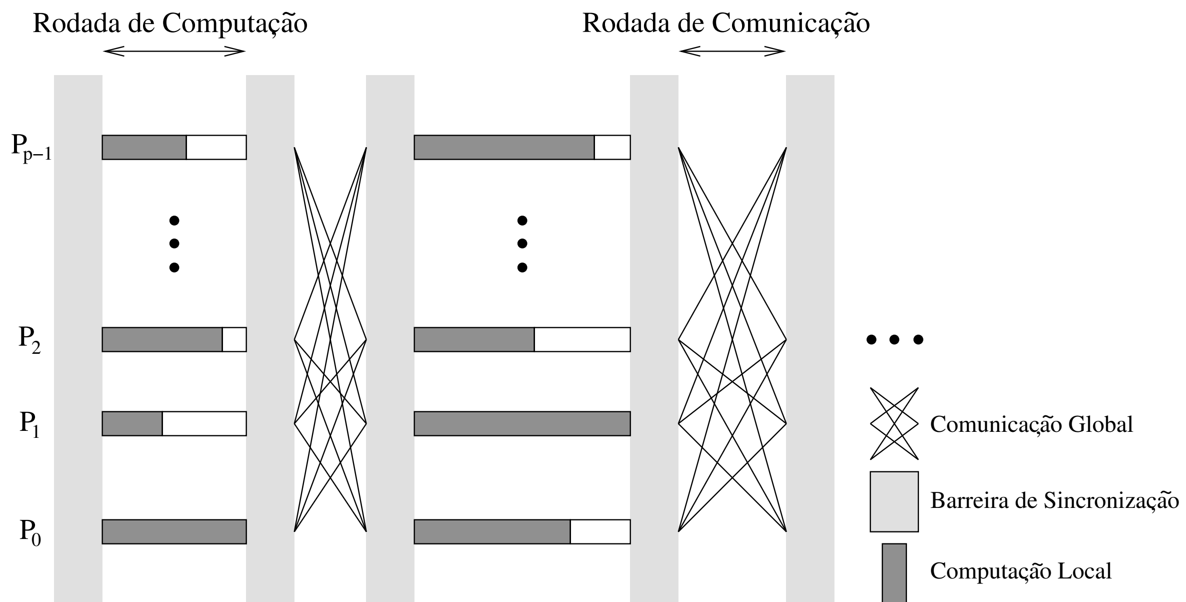


Figura 2.2: Execução no Modelo CGM

Um algoritmo BSP/CGM possui uma combinação destes dois modelos. Em um algoritmo BSP/CGM são realizadas rodadas de computação local alternadas com rodadas de comunicação entre os processadores. Nas rodadas de comunicação, cada processador envia e recebe no máximo $O(n/p)$ dados. Barreiras de sincronização separam rodadas de computação local de rodadas de comunicação entre os processadores.

A soma dos tempos gastos tanto com computação local quanto com comunicações entre os processadores define o tempo de execução do algoritmo BSP/CGM. Geralmente, o melhor algoritmo seqüencial é utilizado nas rodadas de computação local e o número de rodadas de comunicação é reduzido ao máximo.

2.4 Considerações finais

Neste capítulo, apresentamos alguns conceitos da Complexidade Parametrizada e a teoria de Tratabilidade por Parâmetros Fixos. Apresentamos as definições de problemas parametrizáveis e problemas parametrizáveis por parâmetro fixo de maneira mais formal. Também abordamos as técnicas de Redução ao Núcleo do Problema e a Árvore Limitada de busca. Apresentamos os principais conceitos da Computação Paralela e descrevemos modelos paralelos realísticos BSP e CGM.

Capítulo 3

O problema da k-Cobertura por Vértices

3.1 Considerações Iniciais

Neste capítulo, apresentamos algoritmos FPT (tratáveis por parâmetro fixo) para o Problema da k-Cobertura por Vértices [1]. Estes algoritmos recebem como entrada um grafo G com n vértices e um inteiro k.

Este capítulo está organizado nas seguintes seções: na Seção 3.2 é apresentado o algoritmo de Buss [3]; na Seção 3.3 é apresentado o algoritmo B1 de Balasubramanian *et al.* [1]; na Seção 3.4 é apresentado o algoritmo B2 de Balasubramanian *et al.* [1]; na Seção 3.5 é apresentado o algoritmo Cheetham *et al.* [6]; na Seção 3.6 é apresentado o algoritmo de Downey *et al*[14]; Por fim, na Seção 3.7 é apresentado o algoritmo de Niedermeier e Rossmanith [32].

3.2 Algoritmo de Buss [3]

Este algoritmo descreve um método de redução ao núcleo do problema que reduz, em tempo polinomial, o grafo G em G', sendo que o tamanho de G' é limitado por uma função do parâmetro k.

O algoritmo remove todos os vértices do grafo G de grau maior que k e adiciona-os à cobertura parcial H.

Caso haja mais que k vértices de grau maior ou igual a k, não haverá uma cobertura por vértices para G de tamanho no máximo k. A partir deste momento, se o número de arestas for menor que $k.k'$, 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', para

[1]Foram utilizados nos algoritmos de Buss, B1, B2 e Cheetham as figuras e exemplos apresentados no trabalho de Hanashiro[25].

a instância (G', k') e cobertura parcial por vértices H.

ALGORITMO BUSS
Entrada: $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho máximo k, se existir; ou uma mensagem, se tal cobertura não existe.

1. Seja H um conjunto de vértices V com 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 nele 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** alguns desses subconjuntos forma uma cobertura por vértices para G'
   devolva os vértices desse subconjunto mais os vértices de H.
8. **senão devolva**
   "Não existe a cobertura por vértices desejada."

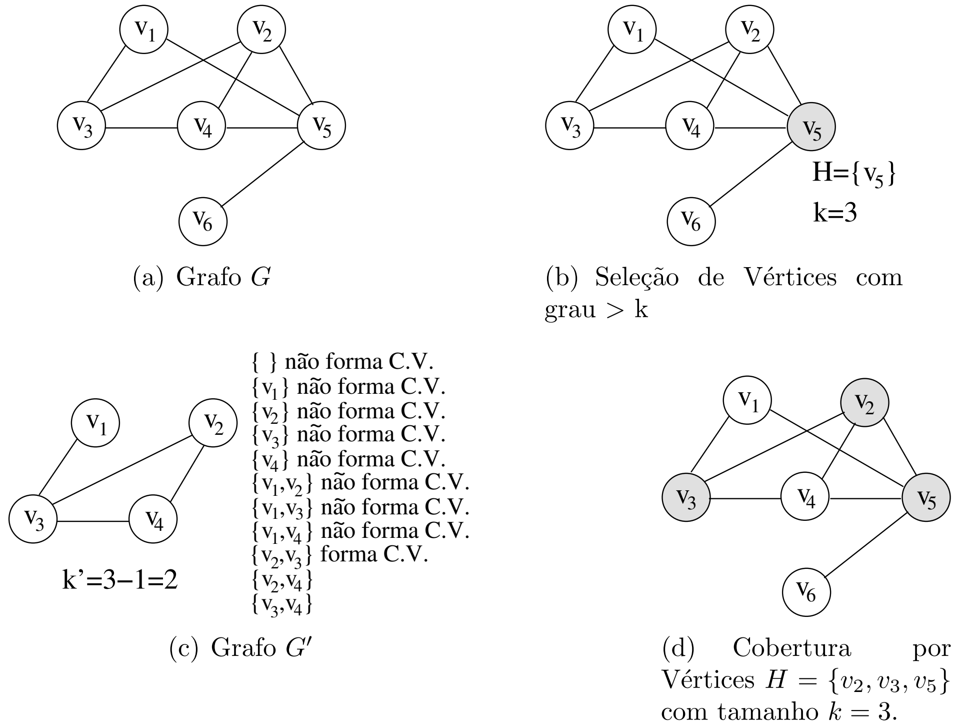
   fim algoritmo

A Figura 3.1 apresenta um exemplo de execução do Algoritmo de Buss. A entrada do algoritmo é o grafo G, apresentado na Figura 3.1(a), e um valor inteiro k igual a 3. Na Figura 3.1(b) podemos observar a redução ao núcleo com a retirada do vértice 5, que possui grau maior que k. A Figura 3.1(c) apresenta um algoritmo de força bruta que tenta encontrar uma Cobertura por Vértices de tamanho maior ou igual a k e, por fim, a Figura 3.1(d) apresenta uma Cobertura por Vértices de tamanho k igual a 3.

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

3.3 Algoritmo B1 de Balasubramanian *et al.* [1]

Denominamos o primeiro algoritmo de Balasubramanian *et al.* [1] de Algoritmo B1. Neste algoritmo, inicialmente, aplicamos o método de redução ao núcleo do problema, gerando a instância (G', k') e uma cobertura parcial H.



(a) Grafo G

(b) Seleção de Vértices com grau > k

(c) Grafo G'

(d) Cobertura por Vértices $H = \{v_2, v_3, v_5\}$ com tamanho $k = 3$.

Figura 3.1: Exemplo de execução do Algoritmo de Buss [3].

No passo seguinte, geramos uma árvore limitada de busca, que é ternária, com o nó raiz contendo a instância (G', k') e H, gerados no passo anterior. Devemos destacar que apenas geramos os nós à medida que a busca em profundidade é realizada.

Cada nó da árvore armazena uma cobertura por vértices parcial H e uma instância reduzida do problema $(G'' = (V'', E''), k'')$. O grafo G'' é resultante da remoção das arestas incidentes em H e quaisquer vértices isolados, e o número k'' é o tamanho máximo da cobertura por vértices de G''. Observe que um nó da árvore tem uma cobertura por vértices parcial H com mais vértices e um grafo com menos vértices e arestas do que seu pai, pois a cada nova ramificação um possível vértice é adicionado à cobertura parcial, e é removido do grafo, bem como as arestas nele incidentes e quaisquer vértices isolados.

Durante a busca em profundidade, escolhemos um vértice $v \in V''$ que passe por, no máximo, três arestas, obtendo os seguintes possíveis caminhos: caminhos simples de comprimento três; ciclos de comprimento três; caminhos de comprimento dois; e caminhos de comprimento um. Os dois primeiros caminhos originam, cada um, três possibilidades de cobertura e os dois últimos reduzem o grafo dentro do próprio nó.

A árvore tem seu crescimento interrompido quando o nó que está sendo processado tem uma instância com uma cobertura parcial de tamanho menor ou igual a k e com um grafo resultante vazio, ou quando percorremos toda a árvore, garantindo a limitação em função do parâmetro k.

ALGORITMO B1
Entrada: $G = (V, E)$ e um inteiro positivo k.

Saída: uma cobertura por vértices de tamanho máximo k, se existir; ou uma mensagem, se tal cobertura não existe.

1. Algoritmo de redução ao núcleo, Buss [3], resultando (G', k') e H.

2. Seja T a árvore de busca cujo nó raiz armazena (G', k') e H. Fazemos uma busca em profundidade em T, à medida que novos vértices são criados.

3. **para** cada nó da árvore T **faça**

   Seja r o nó atual da árvore T. Sejam $(G'' = (V'', E''), k'')$ a instância reduzida do problema e CVP a cobertura por vértices parcial armazenados em r.

   3.1 **se** $k'' >= 0$

   3.1.1 **se** G'' for vazio

   devolva CVP; **termina**

   3.1.2 Escolha aleatóriamente um vértice v em 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 em E'' que incidem nos vértices removidos de V''. Além disso, qualquer vértice isolado em V'' também é removido.)

   3.1.3 **se** o Passo 3.1.2 resulta em um caminho simples de tamanho três que consiste na seqüencia de vértices v, v_1, v_2, v_3; o nó r gera três nós:

   (a) $CVP = CVP \bigcup \{v, v_2\}; (G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2)$

   (b) $CVP = CVP \bigcup \{v_1, v_2\}; (G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2)$

   (c) $CVP = CVP \bigcup \{v_1, v_3\}; (G''' = (V'' - \{v_1, v_3\}, E'''), k''' = k'' - 2)$

   3.1.4 **se** o Passo 3.1.2 resulta em um ciclo de tamanho três que consiste na seqüencia de vértices v, v_1, v_2, v; o nó r gera três nós:

   (a) $CVP = CVP \bigcup \{v, v_1\}; (G''' = (V'' - \{v, v_1\}, E'''), k''' = k'' - 2)$

   (b) $CVP = CVP \bigcup \{v_1, v_2\}; (G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2)$

   (c) $CVP = CVP \bigcup \{v, v_2\}; (G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2)$

   3.1.5 **se** o Passo 3.1.2 resulta em um caminho simples de tamanho dois que consiste na seqüencia de vértices $,v, v_1, v_2$; executa-se, no próprio nó r, a operação:

   (a) $CVP = CVP \bigcup \{v_1\}; (G''' = (V'' - \{v_1\}, E'''), k''' = k'' - 1)$

   3.1.6 **se** o Passo 3.1.2 resulta em um caminho simples de tamanho um que consiste na seqüencia de vértices $,v, v_1$; executa-se, no próprio nó r, a operação:

   (a) $CVP = CVP \bigcup \{v\}; (G''' = (V'' - \{v\}, E'''), k''' = k'' - 1)$

   3.1.7 **senão**

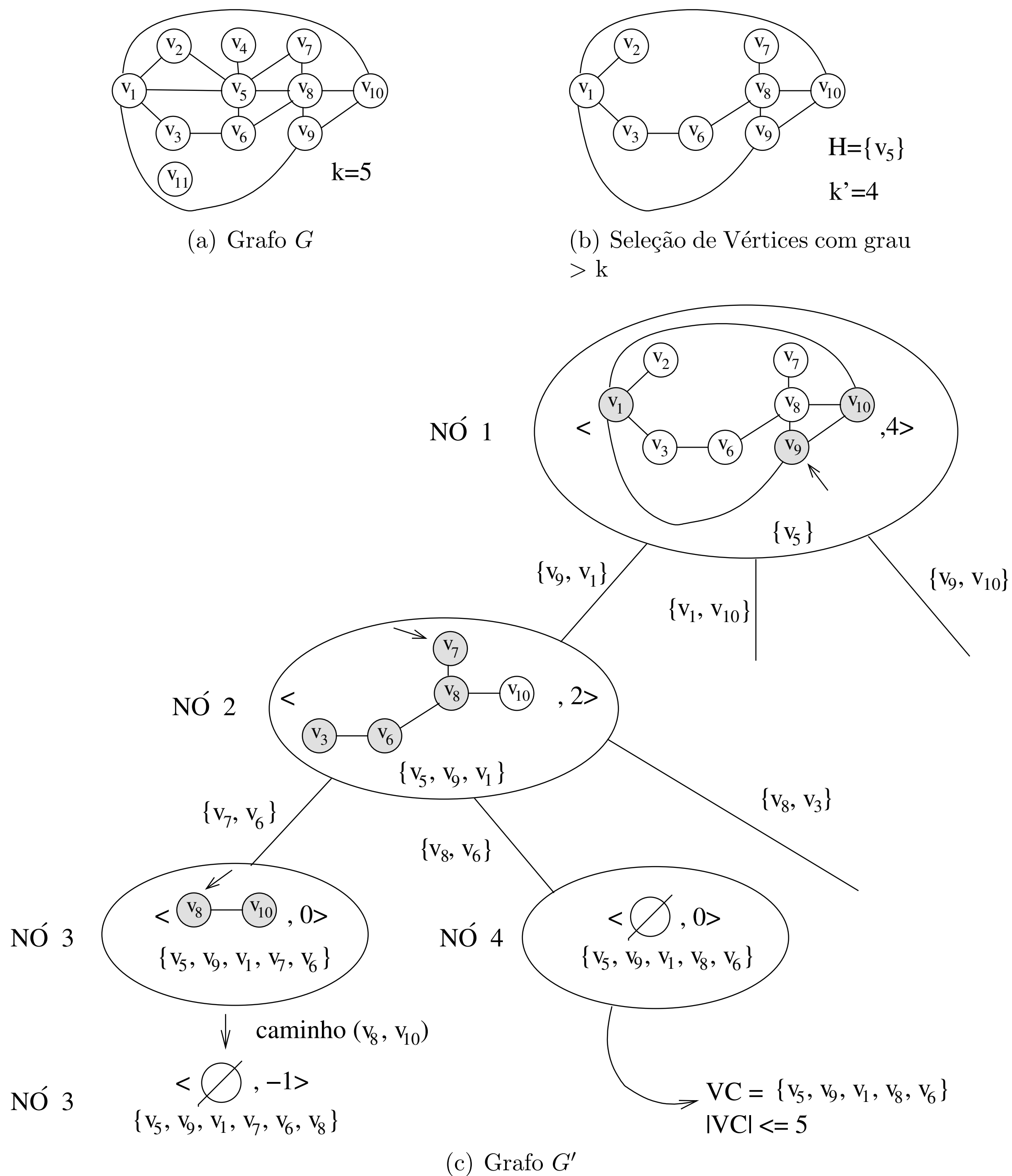
   Vá para o próximo nó da busca em profundidade na árvore T.

4. **devolva**

   "Não existe a cobertura por vértices desejada".

fim algoritmo

A Figura 3.2 apresenta um exemplo de execução do Algoritmo B1. A entrada do algoritmo é o grafo G apresentado na Figura 3.2(a) e um valor inteiro k igual a 5. Na Figura 3.2(b), podemos observar o grafo G, após a redução ao núcleo, com a retirada dos vértices que possuem grau maior do que 5, as arestas incidentes e os vértices isolados. A Figura 3.2(c) apresenta um algoritmo de busca em profundidade na árvore em que os nós são gerados à medida que a busca acontece. Podemos perceber que cada nó na árvore apresenta exatamente 3 filhos. Na busca em profundidade foram gerados os nós NÓ 1, NÓ 2 e NÓ 3, quando o crescimento é interrompido. Após isto, o NÓ 4 é gerado em outra opção de ramificação e a cobertura é encontrada.



(a) Grafo G

(b) Seleção de Vértices com grau > k

(c) Grafo G'

Figura 3.2: Exemplo de execução do Algoritmo B1 [1].

A complexidade de tempo do Algoritmo B1 é $O(kn + \sqrt{3^k}k^2)$, formado pela complexidade de tempo do método de redução ao núcleo do problema e a árvore ternária limitada de busca.

3.4 Algoritmo B2 de Balasubramanian *et al.* [1]

O segundo algoritmo de Balasubramanian *et al.*[1], que denominamos Algoritmo B2, recebe um grafo $G = (V, E)$ e um inteiro k. Inicialmente, aplicamos o método de redução ao núcleo do problema semelhante ao método aplicado no algoritmo de Buss[3], gerando a instância (G', k') e uma cobertura parcial H. No passo seguinte, geramos uma árvore limitada de busca T, não necessariamente ternária. A árvore T terá o nó raiz contendo a instância (G', k') e H, gerados no passo anterior.

Devemos destacar que apenas geramos os nós, à medida que a busca em profundidade é realizada. Cada nó da árvore armazena uma cobertura por vértices parcial H e uma instância reduzida do problema $(G'' = (V'', E''), k'')$. O grafo G'' é resultante da remoção das arestas incidentes em H e quaisquer vértices isolados, e o número k'' é o tamanho máximo da cobertura por vértices de G''. Observe que um nó da árvore tem uma cobertura por vértices parcial H com mais vértices e um grafo com menos vértices e arestas do que seu pai, pois a cada nova ramificação um possível vértice é adicionado à cobertura parcial, fazemos sua remoção do grafo, bem como das arestas nele incidentes e quaisquer vértices isolados.

Na busca em profundidade, escolhemos um vértice v do grafo G'' e priorizamos de acordo com a seguinte ordem de precedência: caminhos com vértice de grau um, caminhos com vértice de grau dois, caminhos com vértice de grau maior ou igual a cinco, caminhos com vértice de grau três e caminhos com vértice de grau quatro. Conforme o grau e a escolha do vértice, temos várias possibilidades de cobertura, que dá origem a um ramo da árvore limitada de busca. Cada caso de ramificação, associado a uma respectiva escolha de vértice pelo seu grau, pode ser vista em detalhes em [3].

A árvore tem seu crescimento interrompido quando o nó em questão tem uma instância com uma cobertura parcial de tamanho menor ou igual a k, e um grafo resultante vazio, ou quando percorremos toda a árvore, tendo a profundidade limitada em função do parâmetro k.

ALGORITMO B2
Entrada: $G = (V, E)$ e um inteiro positivo k.
Saída: uma cobertura por vértices de tamanho máximo k, se existir; ou uma mensagem, se tal cobertura não existe.

1. Algoritmo de redução ao núcleo, Buss [3], resultando (G', k') e H.

2. Seja T a árvore de busca cujo nó raiz armazena (G', k') e H. Fazemos busca em profundidade na árvore T.

3. **para** cada nó da árvore T **faça**

Seja r o nó atual da árvore T. Sejam $(G'' = (V'', E''), k'')$ a instância reduzida do problema e a cobertura por vértices parcial (CVP) armazenados em r. Seja $N(v)$ o conjunto dos vértices vizinhos ao vértice v e $N(S) = \bigcup_{v \in S} N(v)$.

3.1 **se** $k'' >= 0$

3.1.1 **se** G'' for vazio

devolva CVP; **termina**

3.1.2 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 passos 3.1.3 a 3.1.7 E''' resulta da remoção das arestas de E'' que incidem nos vértices removidos de V''. Além disso, qualquer vértice isolado em V''' também deverá ser removido.)

3.1.3 **se** o vértice v escolhido no Passo 3.1.2 tem grau 1, seja o vértice x seu único vizinho. O nó r gera um nó filho assim:
a) $CVP = CVP \bigcup \{x\}; (G''' = (V'' - \{x\}, E'''), k''' = k'' - 1)$

3.1.4 **se** o vértice v escolhido no Passo 3.1.2 tem grau 2, sejam os vértices x e y seus vizinhos

3.1.4.1 **se** existe uma aresta entre x e y, o nó r gera um nó filho assim:
a) $CVP = CVP \bigcup \{x, y\}; (G''' = (V'' - \{x, y\}, E'''), k''' = k'' - 2)$

3.1.4.2 **senão se**, juntos, x e y têm pelo menos dois vizinhos diferentes de v, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{x, y\}; (G''' = (V'' - \{x, y\}, E'''), k''' = k'' - 2)$
b) $CVP = CVP \bigcup N(\{x, y\}); (G''' = (V'' - N(\{x, y\}), E'''), k''' = k'' - |N(\{x, y\})|)$

3.1.4.3 **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) $CVP = CVP \bigcup \{v, a\}; (G''' = (V'' - \{v, a\}, E'''), k''' = k'' - 2)$

3.1.5 **se** o vértice v escolhido no Passo 3.1.2 tem grau maior ou igual a 5, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{v\}; (G''' = (V'' - \{v\}, E'''), k''' = k'' - 1)$
b) $CVP = CVP \bigcup N(v); (G''' = (V'' - N(v), E'''), k''' = k'' - |N(v)|)$

3.1.6 **se** o vértice v escolhido no Passo 3.1.2 tem grau 3, sejam os vértices x, y e z seus vizinhos

3.1.6.1 **se** existe uma aresta entre x e y, o nó r gera dois nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}; (G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3)$
b) $CVP = CVP \bigcup N(z); (G''' = (V'' - N(z), E'''), k''' = k'' - |N(z)|)$

3.1.6.2 **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) $CVP = CVP \bigcup \{x, y, z\}; (G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3)$
b) $CVP = CVP \bigcup \{v, a\}; (G''' = (V'' - \{v, a\}, E'''), k''' = k'' - 2)$

3.1.6.3 **senão se** x tem três vizinhos diferentes de v, o nó r gera três nós filhos assim:
a) $CVP = CVP \bigcup \{x, y, z\}; (G''' = (V'' - \{x, y, z\}, E'''), k''' = k'' - 3)$
b) $CVP = CVP \bigcup N(x); (G''' = (V'' - N(x), E'''), k''' = k'' - 4)$

c) $CVP = CVP \bigcup \{x\} \bigcup N(\{y,z\}); (G''' = (V'' - \{x\} \bigcup N(\{y,z\}), E'''),$ $k''' = k'' - 1 - |N(\{y,z\})|)$

3.1.6.4 **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) $CVP = CVP \bigcup \{x,y,z\}; (G''' = (V'' - \{x,y,z\}, E'''), k''' = k'' - 3)$

b) $CVP = CVP \bigcup N(x); (G''' = (V'' - N(x), E'''), k''' = k'' - 3)$

c) $CVP = CVP \bigcup N(\{y,z,a,b\}); (G''' = (V'' - N(\{y,z,a,b\}), E'''),$ $k''' = k'' - |N(\{y,z,a,b\})|)$

3.1.7 **se** o vértice v escolhido no Passo 3.1.2 tem grau 4, sejam os vértices x, y, z e w seus vizinhos

3.1.7.1 **se** existe uma aresta entre x e y, o nó r gera três nós filhos assim:

a) $CVP = CVP \bigcup \{x,y,z,w\}; (G''' = (V'' - \{x,y,z,w\}, E'''), k''' = k'' - 4)$

b) $CVP = CVP \bigcup N(z); (G''' = (V'' - N(z), E'''), k''' = k'' - 4)$

c) $CVP = CVP \bigcup z \bigcup N(w); (G''' = (V'' - z \bigcup N(w), E'''), k''' = k'' - |z \bigcup N(w)|)$

3.1.7.2 **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) $CVP = CVP \bigcup \{x,y,z,w\}; (G''' = (V'' - \{x,y,z,w\}, E'''), k''' = k'' - 4)$

b) $CVP = CVP \bigcup \{v,a\}; (G''' = (V'' - \{v,a\}, E'''), k''' = k'' - 2)$

3.1.7.3 **senão se** x, y, z e w têm pelo menos três vizinhos diferentes de v, o nó r gera quatro nós filhos assim:

a) $CVP = CVP \bigcup \{x,y,z,w\}; (G''' = (V'' - \{x,y,z,w\}, E'''), k''' = k'' - 4)$

b) $CVP = CVP \bigcup N(y); (G''' = (V'' - N(y), E'''), k''' = k'' - 4)$

c) $CVP = CVP \bigcup \{y\} \bigcup N(w); (G''' = (V'' - y \bigcup N(w), E'''), k''' = k'' - 5)$

d) $CVP = CVP \bigcup \{y,w\} \bigcup N(\{x,z\});$ $(G''' = (V'' - \{y,w\} \bigcup N(\{x,z\}), E'''),\ k''' = k'' - 2 - |N(\{x,z\})|)$

3.1.8 **senão**

Vá para o próximo nó da busca em profundidade na árvore T.

4. **devolva**

“Não existe a cobertura por vértices desejada”.

fim algoritmo

A Figura 3.3 apresenta um exemplo de execução do Algoritmo B2. A entrada do algoritmo é o grafo G apresentado na Figura 3.3(a) e um valor inteiro k com valor igual a 8. Na Figura 3.3(b), podemos observar o grafo G após a redução ao núcleo, com a retirada dos vértices que possuem grau maior que 8, das arestas neles incidentes, além de quaisquer vértices isolados. A Figura 3.3(c) apresenta um algoritmo de busca em profundidade na árvore em que os nós são gerados à medida que a busca é realizada.

Como podemos perceber, o algoritmo não garante uma árvore de busca ternária, pois os casos destes algoritmo podem gerar diferentes números de escolhas de ramificação. Neste exemplo de execução, a cobertura foi encontrada através das escolhas de ramificação que resultaram na busca em profundidade pelos nós NÓ 1, NÓ 2, NÓ 6 e NÓ 7.

A complexidade de tempo do Algoritmo B2 é $O(kn + 1.324718^k k^2)$, formado pela complexidade de tempo do método de redução ao núcleo do problema e a árvore limitada de busca.

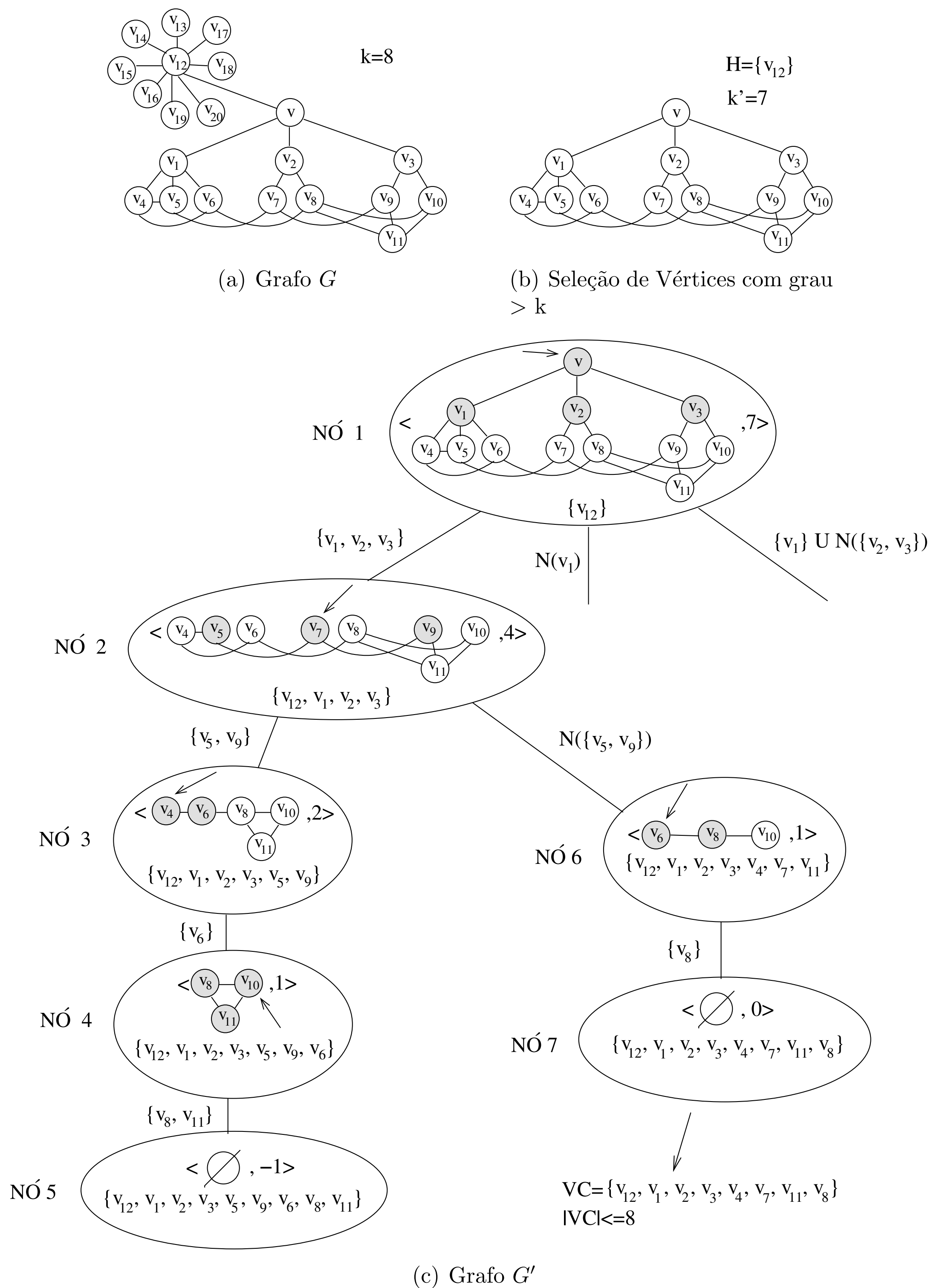
3.5 O Algoritmo de Cheetham *et al.* [6]

O algoritmo de Cheetham *et al.* [6] paraleliza ambas as fases de redução ao núcleo do problema e da árvore limitada de busca. A fase de redução ao núcleo do problema é baseada no algoritmo de Buss [3]. Na versão paralela, cada processador $P_i, 0 \leq i \leq p-1$, é responsável por computar o grau de n/p vértices, os quais são rotulados de $i * (n/p)$ a $((i+1) * (n/p)) - 1$. Cada processador recebe m/p arestas da lista de arestas do grafo G e as ordena de acordo com os vértices em que incide. Desta forma, é possível calcular o grau dos vértices locais e dos vértices que não são de seu domínio e enviar tais informações aos vértices destino. Após isto cada processador é capaz de calcular precisamente o valor dos graus dos vértices que estão sob o seu domínio. No passo seguinte, cada processador P_i informa aos demais processadores, quais os vértices locais que possuem grau maior que k. Desta maneira, todos os processadores são capazes de remover as arestas que neles incidem. 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.

Na fase da árvore limitada de busca, geramos a árvore ternária completa T com altura $h = \log_3 p$ e com p folhas (y_0 a y_{p-1}), pelo Algoritmo B1 [1], de forma determinística, com o nó raiz armazenando a cobertura por vértices parcial e a instância (G', k'). Observe que no início desta fase, todos os processadores têm a mesma instância obtida ao final da fase anterior. A árvore T não é explicitamente criada pelos processadores. Cada processador $P_i, 0 \leq i \leq p-1$ percorre o único caminho entre a raiz e a folha y_i calculado pela fórmula $i/(p/3^{h+1})$. O Algoritmo B1 é interrompido quando atinge um nó de nível $\log_3 p$ (folha y_i).

Com isto, finalizamos a primeira parte desta fase. Na segunda parte cada processador P_i executa localmente, a partir da instância (G''_i, k''_i), referente à folha y_i da árvore T, usando o Algoritmo B2, como apresentado na Figura 3.4. Esta fase será encerrada quando encontrarmos uma cobertura por vértices para G de tamanho menor ou igual a k, ou se todos os processadores percorrerem toda a árvore, significando que não foi possível encontrar uma cobertura válida.

A Figura 3.4 apresenta um exemplo de execução do algoritmo de Cheetham. O processador P_i processa o caminho único até a folha y_i usando o Algoritmo B1. Depois, P_i processa toda a subárvore de raiz y_i usando o Algoritmo B2.



(c) Grafo G'

Figura 3.3: Exemplo de execução do Algoritmo B2 [1].

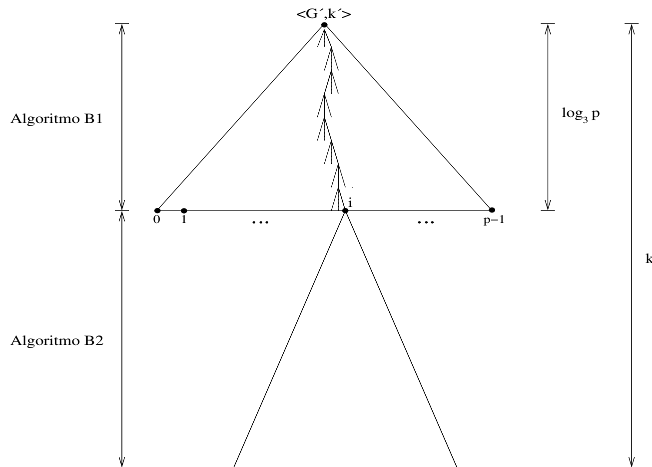


Figura 3.4: Exemplo de execução do Algoritmo de Cheetham [6].

3.6 Algoritmo de Downey et al [14]

Nesta seção, apresentamos o algoritmo de Downey *et al* para o problema da k-Cobertura por Vértices [14] e o chamaremos de Dk-VC. Segundo Downey *et al* [14], diferentes técnicas podem sistematicamente ser adaptadas em uma correspondente variedade de heurísticas. O Dk-VC é baseado nas técnicas de redução ao núcleo e árvore limitada de busca. Este algoritmo é composto por duas fases. Na Fase 1 é determinado um *núcleo* sobre uma instância (G, k), ou é produzida a resposta "não". A resposta "não"significa que não existe uma cobertura de tamanho k para a instância (G, k). O *núcleo* é uma instância (G', k') com $|G'| < k^2$ e $k' \leq k$, tal que G' possui uma cobertura por vértices de tamanho k', se, e somente se, G possuir uma cobertura de tamanho k. A redução de (G, k) para (G', k') é realizada em tempo $O(kn)$, onde k é o número de vértices em G. Na Fase 2, é construída uma árvore de busca com altura no máximo k e o nó raiz é rotulado como (G', k'), de acordo com a saída da Fase 1.

Fase 1 (Redução ao núcleo do Problema): Iniciamos por (G, k) e, enquanto for possível, aplicamos as seguintes regras:

1. Se G possui algum vértice v com grau maior que k então substitua (G, k) por $(G - v, k - 1)$.

2. Se G possui dois vértices u, v não adjacentes, tal que $|N(u) \cup N(v)| > k$, então substitua (G, k) por $(G + uv, k)$.

3. Se G possui dois vértices u, v adjacentes, tal que $N(v) \subseteq N(v)$, então substitua (G, k) por $(G - u, k - 1)$.

4. Se G possui uma aresta uv, ligando dois vértices u, v, com u tendo grau 1, então substitua (G, k) por $(G - uv, k - 1)$.

5. Se G possui um vértice x com grau 2, com vizinhos a e b, em que nenhum dos casos anteriores se aplica (desta forma a e b são não adjacentes), então substitua (G, k)

por (G', k), na qual G' é obtido através de G da seguinte maneira:

- excluindo o vértice x;
- adicionando a aresta ab; e
- adicionando todos os possíveis vértices entre a, b e $N(a) \cup N(b)$.

6. Se G possui um vértice x com grau 3, com vizinhos a, b, c, e nenhum dos casos anteriores se aplica, então substitua (G, k) por (G', k), com G' obtido de G de acordo com os seguintes casos, dependendo do número de arestas entre a, b e c:

   Não existem arestas entre a, b e c. Neste caso, G' é obtido através de G da seguinte maneira:

   - excluindo x de G;
   - adicionando arestas de c para todos os vértices $N(a)$;
   - adicionando arestas de a para todos os vértices $N(b)$;
   - adicionando arestas de b para todos os vértices $N(c)$; e
   - adicionando as arestas ab e bc.

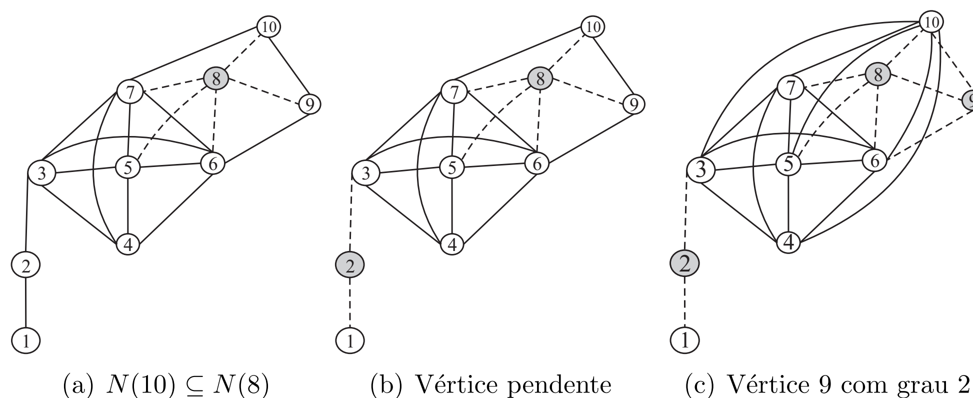
   Existe exatamente uma aresta em G entre a, b, c. Sem perda de generalidade, ab, neste caso, G' é obtido através de G da seguinte maneira:

   - excluindo x de G;
   - adicionando arestas de c para todos os vértices $N(b) \cup N(a)$;
   - adicionando arestas de a para todos os vértices $N(c)$;
   - adicionando arestas de b para todos os vértices $N(c)$;
   - adicionando as arestas bc; e
   - adicionando as arestas ac.

A Figura 3.5 apresenta um exemplo de execução do algoritmo de Downey [14]. A entrada do algoritmo é o grafo G e um valor inteiro k com valor igual a 7. Na Figura 3.5(a), os vértices 8 e 10 são adjacentes e $N(10) \subseteq N(8)$ (caso 3). O vértice 8 é adicionado na cobertura resultando em $k' = 6$ e $G' = G - \{8\}$. Na Figura 3.5(b), é verificado que o vértice 1 é um vértice pendente, ou vértice com grau 1 (caso 4), então o vértice 2 é adicionado na cobertura resultando em $k'' = 5$ e $G'' = G - \{1, 2\}$. Na Figura 3.5(c), o grau do vértice 9 é 2 (caso 5). Substituímos (G'', k'') por (G''', k'') onde G''' que é obtido adicionando-se os seguintes vértices em $G'' - \{9\}$: $(10, 3), (10, 4), (10, 5), (10, 6)$.

É possível perceber, ao fim da Fase 1, que a redução do grafo (G, k) para (G', k') permitiu que o grafo G' tivesse vértices com grau no mínimo 4, isso se uma resposta negativa já não foi dada a questão. De acordo com as regras de redução (2) e (3), essa resposta seria "não"se o número de vértices de G' fosse maior do que k^2.

Fase 2 (Construção da árvore de busca). O nó raiz da árvore de busca é o grafo (G', k') resultante da Fase 1 do algoritmo. O procedimento de ramificação é realizado enquanto houver algum vértice com grau pelo menos 6. Em cada ramificação, são aplicadas,



(a) $N(10) \subseteq N(8)$ (b) Vértice pendente (c) Vértice 9 com grau 2

Figura 3.5: Exemplo de execução do Algoritmo de Downey [14].

novamente, as regras de redução da Fase 1. Com isso, podemos assumir que cada folha resultante da árvore de busca é um grafo resultante com vértices de grau 4 ou 5.

- **Vértices com grau 4**. Se existe um vértice x com grau 4, então suponha que os vizinhos de x são a, b, c, d. Vários casos serão considerados, de acordo com o número de arestas presentes nos vértices a, b, c e d. Note que se nem todos a, b, c, d estão na cobertura, então no mínimo 2 deles estão.

  Caso 1. O subgrafo formado por a, b, c, d possuem uma aresta, digamos ab. Então c e d juntos não podem estar dentro da cobertura, a menos que todos os vértices a, b, c e d estejam. Podemos concluir que, necessariamente, um dos subconjuntos seguintes fazem parte da cobertura C, e ramificamos de acordo com:

  1. $a, b, c, d \subseteq C$
  2. $N(c) \subseteq C$
  3. $c \cup N(d) \subseteq C$.

  Caso 2. O subgrafo formado por a, b, c, d é vazio. Consideramos três subcasos.

  Subcaso 2.1. Três dos vértices (digamos a, b, c) têm um vizinho em comum y. O vértice d, por sua vez, tem x. Então, quando nenhum dos vértices a, b, c, d estão na cobertura, x e y estão. Com isso podemos concluir que a ramificação deve ser feita de acordo com os seguinte subconjuntos:

  1. $a, b, c, d \subseteq C$
  2. $x, y \subseteq C$.

  Subcaso 2.2. Se o Subcaso 2.1 não se aplica, então talvez exista um par de vértice (suponha a e b), não adjacentes a x e que possuam um total de seis vizinhos. Se nenhum destes vértices a, b, c, d está na cobertura C, então $c \notin C$ ou $c \in C$ e $d \notin C$, ou ambos $c \in C$ e $d \in C$ (e neste caso $a \notin C$ e $b \notin C$). Podemos concluir que a ramificação deve ser feita de acordo com os seguinte subconjuntos:

  1. $a, b, c, d \subseteq C$

2. $N(c) \subseteq C$
3. $c \cup N(d) \subseteq C$
4. $c, d \cup N(a, b) \subseteq C$.

Subcaso 2.3. Se os Subcasos 2.1 e 2.2 não se aplicam, então o grafo deve ter a seguinte estrutura na vizinhança com relação ao vértice x:

1. x tem quatro vizinhos a, b, c, d e cada um deles tem grau 4.
2. Existe um conjunto E de seis vértices, tal que cada vértice em E é adjacente para exatamente dois vértices em a, b, c, d, e o subgrafo formado por $E \cup \{a, b, c, d\}$ é subdividido com cada aresta subdividida uma vez.

   Neste caso, ramificamos de acordo com:
   (a) $a, b, c, d \subseteq C$
   (b) $(E \cup x) \subseteq C$.

- **Vértices com grau 5**. Se o grafo é regular de grau 5 e nenhuma da regras de redução da Fase 1 podem ser aplicadas, então escolhemos um vértice x de grau 5 e ramificamos de (G, k) para $(G - x, k - 1)$ e $(G - N[x], k - 5)$. Então, escolhemos um vértice u de grau 4 em $G - x$, e ramificamos de acordo com a regra para vértices com grau 4.

  O resultado desses dois passos combinados, a partir de (G, k), cria uma subárvore em que um dos seguintes casos se aplica:

  1. Existem quatro filhos com parâmetro $k - 5$ do Caso 1.
  2. Existem três filhos com parâmetro $k_1 = k - 5, k_2 = k - 5$ e $k_3 = k - 3$, do Subcaso 2.1.
  3. Existem cinco filhos com parâmetro $k_1 = k-5, k_2 = k-5$, $k_3 = k-3$, $k_4 = k-6$ e $k_5 = k - 9$ do Subcaso 2.2.

  Note que a regra de redução (2) da Fase 1 não pode ser aplicada para $G - x$, então no mínimo um dos vizinhos de u tem grau 5, e também o Subcaso 2.3 é impossível.

A Figura 3.6 apresenta um exemplo de execução da Fase 2 do algoritmo de Downey *et al.*. A entrada é (G', k'), resultante da Fase 1 com $k' = 5$ e G' é um grafo 5-regular. Ramificamos de acordo com $(G-x, k-1)$ e $(G-N[x], k-5)$ gerando, respectivamente, os Nós 1 e 2. Podemos perceber que com na ramificação do Nó 1 já encontramos a resposta, mas se o Nó 2 fosse processado resultaria em 4 nós, de acordo com o *subcaso 2.2* da regra para vértices de grau 4 da Fase 2.

3.7 Algoritmo de Niedermeier e Rossmanith [32]

Nesta seção apresentamos o algoritmo de Niedermeier e Rossmanith [32] para o problema k-Cobertura por Vértices que denominamos NRk-VC. O NRk-VC foi desenvolvido em

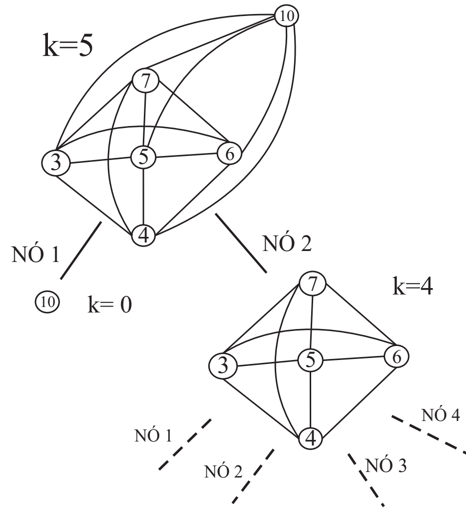


Figura 3.6: Exemplo de execução da Fase 2 do Algoritmo de Downey [14].

essência com base em todos os algoritmos previamente existentes para o problema k-VC. A parte principal é construir uma árvore limitada de busca. Neste sentido, o esforço consiste em construir a árvore de busca cada vez menor.

O NRk-VC é um algoritmo recursivo e o número de recursões é determinada de acordo com o número de nós da árvore. Este número é guiado por recorrências com coeficientes constantes. A solução assintótica é determinada por raízes de característica polinomial. Utilizamos a mesma notação proposta por Kullmann [28]. Se o algoritmo resolve um problema de tamanho n e chama a si mesmo iterativamente, para problemas de tamanho $n - d_1, ..., n - d_k$, então $(d_1, ...d_k)$ é chamado de *vetor de ramificação* da recursão. Isto corresponde à recorrência:

$$t_n = t_{n-d_1} + ... + t_{n-d_k}. \tag{3.1}$$

A característica polinomial desta recorrência é:

$$z^d = z^{d-d_1} + ... + z^{d-d_k}, \tag{3.2}$$

onde $d = max(d_1, ..., d_k)$. Se α é a raiz de (3.2) com valor máximo absoluto, então t_n é α_n limitado por um fator polinomial.

Chamamos $|\alpha|$ de *número de ramificação* que corresponde ao *vetor de ramificação* $(d_1, ..., d_k)$. Além disso, se α é uma raiz única, então $t_n = O(\alpha^n)$ e todos os *números de ramificações* que ocorrem neste algoritmo também são raízes únicas. No algoritmo NRk-VC, o tamanho da árvore de busca é $O(\alpha^r)$, na qual r é o parâmetro e α é o maior valor do *número de ramificação* que pode ocorrer. Isto ocorre na sessão 5 e subcaso 5.2.1 sendo o vetor de ramificação (3, 5, 8, 8) que resulta em r =1.291742754.

O algoritmo NRk-VC trabalha procurando, recursivamente, uma cobertura ótima. Dado um grafo G, escolhemos vários subgrafos $G_1, ..., G_k$ e localizamos uma cobertura ótima para cada um deles. De acordo com as coberturas ótimas deles, podemos construir a cobertura ótima para G. Por exemplo, seja x um vértice qualquer de G e seja G_1 um subgrafo resultante de G, excluindo-se x e todos suas arestas incidentes. Uma cobertura

de G_1, adicionada de x, é uma cobertura de G. Além disso, se uma cobertura ótima para G contém x, então podemos construir uma cobertura ótima para G, através de uma cobertura ótima de G_1. De outra maneira, se uma cobertura ótima em G não contém x, então ela deve conter todos os seus vizinhos, e denotamos por $N(x)$ o conjunto formado por todos os vizinhos de x. Então, seja G_2 um grafo resultante do grafo G, excluindo-se $N(x)$ e todas as suas arestas incidentes. Novamente, podemos construir uma cobertura ótima para G, através de uma cobertura ótima de G_2 e a adição de $N(x)$. Logo, se podemos iniciar uma cobertura ótima para G através de coberturas ótimas de G_1 e G_2, então uma das duas coberturas resultantes deve ser ótima, desde que contenha x ou $N(x)$. Chamamos de *ramificação de acordo com x e $N(x)$* a construção das coberturas ótimas que, respectivamente, serão construídas através da ramificação de x, ou seja, x será parte da cobertura; e através da ramificação de $N(x)$ na qual quem será parte da cobertura ótima será $N(x)$. O tamanho da cobertura cresce a cada passo, desde que não ultrapasse o tamanho k, quando o algoritmo termina.

Em princípio, é desta maneira que o algoritmo trabalha, mas escolher os subgrafos $G_1, ..., G_k$ é uma tarefa complicada. Abaixo seguem as regras que guiam as escolhas dos conjuntos de ramificações:

- Se o grafo é não conexo, então o algoritmo escolhe alguma componente G' e testa, recursivamente, se G' possui uma cobertura de tamanho k ou menor, e se possui, descobre uma cobertura ótima k' de G'. Então, isto procede-se testando se $G - G'$ e outros componentes possuem uma cobertura de tamanho $k - k'$. Desta forma, o algoritmo descobre se o grafo inteiro possui uma cobertura de tamanho k.
- Se o grafo é conexo, então o conjunto de ramificações segue as seguintes regras:
  1. Se existe um vértice x com grau 1, então ramifique de acordo com $N(x)$. Se existe uma cobertura ótima que contém x, então ela não contém $N(x)$ e vice-versa.
  2. Se existe um vértice x com grau 6 ou maior, então ramifique de acordo com x e $N(x)$.
  3. Se não existe vértice com grau 1 ou pelo menos 6, mas existe vértice com grau 2, então proceda conforme o descrito na Sessão 3.7.1.
  4. Se as regras 1-3 não se aplicam, e se o grafo é regular, então escolha algum vértice x e ramifique de acordo com x e $N(x)$.
  5. Se as regras 1-4 não se aplicam, e se existe um vértice com grau 3, então proceda conforme a Sessão 3.7.2.
  6. Caso contrário, deve existir um vértice com grau 4 e todos os outros com grau 4 ou 5, proceda conforme a Sessão 3.7.3.

3.7.1 Vértices com grau 2

Antes de iniciarmos a descrição da regras pertencentes a esta seção é necessário incluir o conceito de *ponte*. Três vértices a, b, y são *ponte* de x se x, a, b, y formam um ciclo.

Se o grafo é 2-regular, todos os vértices constituem um ciclo e é fácil construir uma cobertura ótima em tempo linear. Se o grafo não é 2-regular, então seja x um vértice com grau 2 e a, b seus vizinhos, onde a tem grau maior ou igual a 3. Nesta situação, quatro casos se aplicam:

- **Caso 1**: Existe uma aresta entre a e b ou x tem uma ponte com a qual o vetor ponte tem grau 2. Então, inclua a, b na cobertura o qual é ótimo e nenhuma ramificação é necessária.
- **Caso 2**: Seja $|N(a) \cup N(b)| \geq 4$. Então ramifique de acordo com a, b e $N(a) \cup N(b)$, onde o vetor de ramificação é pelo menos (2,4).
- **Caso 3**: Assuma que x tem exatamente um ponte. Então, o grau de a deve ser 3, e o grau de b deve ser 2. De outra forma, $|N(a) \cup N(b)| \geq 4$. Então, existe uma cobertura ótima que não contém ambos a e y. Se a e y são parte da cobertura ótima então, podemos assumir que x também é, e desta forma, b não. Substituindo a por $N(a)$ produzimos outra cobertura que não possui tamanho maior do esta. Portanto, podemos ramificar de acordo com $N(y)$ e $N(a)$. O vetor de ramificação não é maior que (3,3).
- **Caso 4**: Seja x com duas pontes, então o grau de ambos a e b deve ser 3, de outra forma, $|N(a) \cup N(b)| >= 4$. Seja y e z dois vértices ponte. Podemos ramificar de acordo com y e $N(y)$. Se y está em uma cobertura ótima, junto com y e z, então não existirá uma cobertura ótima com a ou b, visto que, mais adiante, estes dois vértices serão necessários (em qualquer caminho) para cobrir as arestas incidentes de a e b. Assim, podemos ramificar de acordo com $N(y)$ e x, y, z com vetor de ramificação pelo menos (3,3).

3.7.2 Vértices com grau 3

Para os casos 1,2,3 e 4, sejam x um vértice com grau 3 e a, b e c seus vizinhos. Os primeiros quatro casos distinguem-se pela estrutura do subgrafo em torno de x, em particular sobre o grau de seus vizinhos e se x possui triângulos ou pontes. O caso 5 é diferente, preferimos assumir que não existe vértice algum no Grafo, a qual um dos primeiros quatro casos se aplique.

- **Caso 1**: x é parte de um triângulo, exemplo: $\{x, a, b\}$, e não existem outros triângulos. Então ramifique de acordo com $N(x)$ e $N(c)$. Se x não é parte da cobertura, $N(x)$ é. Se x é parte da cobertura, então a ou b são. Se c também é parte da cobertura, então dois vizinhos de x são e podemos trocar x por $N(x)$. O vetor de ramificação é pelo menos $(3, 3)$.
- **Caso 2**: Consideremos que x tem pelo menos duas pontes. Seja y e z os vértices do meio da ponte. Podemos ramificar de acordo com $N(x)$ e $\{x, y, z\}$. O vetor de ramificação é pelo menos (3,3). Se x está na cobertura, então y e z também estão. Assuma que y não está na cobertura então todos os vizinhos de z devem estar.

- **Caso 3**: Consideremos que x tem exatamente uma ponte, e seja ela entre a e b. E seja y o vértice do centro da ponte. Em seguida, podemos assumir que a ou b tem grau 3, seja tal vértice a. Então, podemos ramificar de acordo com $N(x)$ e $N(a)$. O vetor de ramificação é pelo menos (3,3).

- **Caso 4**: Novamente, assuma que x tem exatamente uma ponte, mas assuma ambos a e b com grau pelo menos 4. Então, podemos ramificar de acordo com $N(x)$, $N(a)$ e $\{a, x, N(b), N(c)\}$. Uma vez que podemos assumir que x não é parte do triângulo, e existe exatamente uma ponte, temos um vetor (3,4,7).

- **Caso 5**: Finalmente, podemos assumir que não há vértices com grau 3 que formam uma ponte ou um triângulo.

  Caso 5.1: Consideremos que exista um vértice x com grau 3 e seus vizinhos a, b e c, dois com grau pelo menos 4, digamos a e b. Podemos ramificar de acordo com $N(x)$ ou $\{x, N(a), N(b)\}$ ou $\{x, a, N(b), N(c)\}$ ou $\{x, b, N(b), N(c)\}$ nesta ordem, de forma a encontrar uma cobertura ótima. O vetor de ramificação é pelo menos (3,7,7,7).

  Caso 5.2: De outra forma, podemos considerar que cada vértice com grau 3 tem pelo menos um vizinho com grau maior ou igual a 4. Uma vez que o grafo é conexo, deve ter pelo menos um vértice com grau 3 que possui exatamente um vizinho com grau 4 ou 5.

  Caso 5.2.1 Vamos assumir que não existe um ciclo de tamanho 5 com as seguintes propriedades: (i). Cada vértice no ciclo tem grau 3 e (ii). Existe um vértice no ciclo que tem um vizinho com grau ≥ 4. Escolhemos algum vértice com grau 3 que possui um vizinho com grau 4 ou 5, chame este vértice de vértice a_3 e o vizinho de b_3 . Os outros dois vizinhos de a_3 devem ter grau 3. Para cada vértice com grau 3 podemos recursivamente seguir o caminho que consiste somente de vértices com grau 3: Escolha um vértice vizinho com grau 3, mas não um que venha de a_3. Inicie, tal pelo caminho de a_3 e chame os vértices a_2, a_1, a_0. Inicie por outro caminho e chame os vértices de a_4, a_5, a_6. Cada a_i tem pelo menos dois vizinhos com grau 3. O terceiro vizinho é chamado de b_i, e talvez tenha grau 3, 4 ou 5. Então ramifique de acordo com: $\{a_2, b_3, a_4\}, \{a_1, b_2, a_3, b_4, a_5\}, \{a_1, b_2, N(b_3), N(b_4), b_5, a_6\}$ e $\{a_0, b_1, N(b_2), N(b_3), b_4, a_5\}$. O vetor de ramificação não é maior que (3,5,8,8).

  Caso 5.2.2 Por fim, vamos assumir um ciclo de tamanho 5 que consiste de vértices de grau 3 e no mínimo um deles tem um vizinho com grau maior ou igual 4. O ciclo consiste de $a_0, ..., a_4$ com vizinhos $b_0, ..., b_4$ de fora do ciclo, onde b_2 é o vizinho com grau pelo menos 4. Podemos ramificar como: $\{a_1, b_2, a_3\}$ ou $\{a_0, b_1, a_2, b_3, a_4\}$ ou $\{a_0, b_1, N(b_2), N(b_3), b_4\}$ ou $\{a_0, N(b_1), N(b_2), b_3, a_4\}$. Teremos um vetor de ramificação (3,5,8,8) ou (3,5,7).

3.7.3 Vértices com grau 4

- **Caso 1**: Assuma que exista um vértice x com grau 4 e possui um vizinho y com grau 5 e faz parte de um triângulo, ou seja, dois de seus vizinhos possuem uma

aresta em comum. Sejam a e b os vizinhos de x onde $\{a, b, x\}$ forma um triângulo (a ou b, podem, mas não necessariamente, coincidir com y). Primeiro assumimos que $a, b \neq y$. Seja $c \notin \{a, b, y\}$ outro vizinho de x. Podemos ramificar de acordo com $N(x), N(y)$ e $\{x, y, N(c)\}$. O resultado é um vetor de ramificação pelo menos (4,5,4). Agora assumimos que $a = y$. Logo, podemos assumir que o remanejamento dos dois vizinhos c e d de x, por exemplo: (não sendo a ou b), não estão conectados por uma aresta. Ramificamos de acordo com $N(x), N(c)$, e $\{x, c, N(d)\}$. O vetor de ramificação é novamente pelo menos (4,5,4), porque c não pertence a $N(d)$.

- **Caso 2**: Assumimos neste caso que não existe vértice x, simultaneamente, com as seguintes propriedades: (i). x tem um vizinho y m grau 5 e (ii). x tem pelo menos duas pontes a quem y não é parte. Podemos assumir que não há triângulos que contém um vértice com grau 4 que possui um vizinho com grau 5, por exemplo, que o caso 1 não se aplica. Escolha algum vértice x com grau 4 que possui um vizinho a com grau 5 (tal vértice existe, pois, de outra forma, o grafo seria regular ou não conectado). Seja b, c, d outros vizinhos de x (os quais podem ter graus 4 e 5). Podemos ramificar de com: $N(a), N(x)$ e $\{x, a, N(b), N(d)\}$ e $\{x, a, d, N(b), N(c)\}$ e $\{x, a, b, N(c), N(d)\}$ Temos um vetor de ramificação (5,4,8,9,8).

- **Caso 3**: Agora, assumimos que existe um vértice x exatamente com as propriedades que proibimos no caso 2: ele tem grau 4 e duas pontes. Existe a um vizinho de y com grau 5, que não é parte nenhuma das duas pontes. Existem duas possibilidades mas tanto em quanto em outra x tem duas pontes separadas ou uma ponte dupla. O algoritmo pode ramificar de acordo com $N(y), N(x), \{x, y\}$. Na última ramificação, se $\{x, y\}$ é parte da cobertura, então podemos assumir que os vértices sobre a ponte também são membros da cobertura. Caso contrário seus vizinhos seriam parte da cobertura e isto implicaria que pelo menos três dos vizinhos de x estão na cobertura ótima. Então, de qualquer forma, podemos escolher $N(x)$ ao invés de x. Isto implica em um vetor de ramificação (5,4,4).

A Figura 3.7 apresenta um exemplo de execução do algoritmo de Niedermeier e Rossmanith [32]. A entrada é o Grafo G e um inteiro $k = 7$. Na Figura (I) podemos observar o vértice 1 que possui grau 1, e a regra 1 é aplicada sobre G. Neste caso, nenhuma ramificação é necessária, e o vértice 2 (vizinho do vértice 1) é adicionado na Cobertura resultando em $k = 5$ e $G - \{2\}$. Na Figura (II) o vértice $x = 7$ possui grau maior ou igual a 6, deste modo ramificamos de acordo com $N(x)$ e x, resultando, respectivamente, nos Nós 1 e 2. O Nó 1 possui a Cobertura desejada e não é apresentado, os demais nós são apresentados apenas para fins didáticos. Na Figura (III) o vértice $x = 10$, do Nó 2, possui grau igual 2, e seus vizinhos 8 e 9 possuem uma aresta de ligação, neste caso, é aplicado o *Caso 1* da regra para vértices de grau 2. Esta escolha também é ótima, e nenhuma ramificação é necessária. Os vértices 8 e 9 são adicionados na Cobertura e o resultado é $k = 3$ e $G = G - \{8, 9, 10\}$. Na Figura (IV) o grafo é regular e qualquer vértice x pode ser escolhido, seja tal vértice o vértice 5. Ramificamos de acordo com $\{(G - N(x), k - |N(x)|)\}$ e $(G - x, k - 1)$, respectivamente, os Nós 4 e 5. Podemos observar que o Nó 4 possui uma resposta válida e o Nó 5 apresenta um ciclo com vértices de grau 2 onde será aplicado o *Caso 1* para vértices de grau 2 e a resposta também será encontrada através desta ramificação.

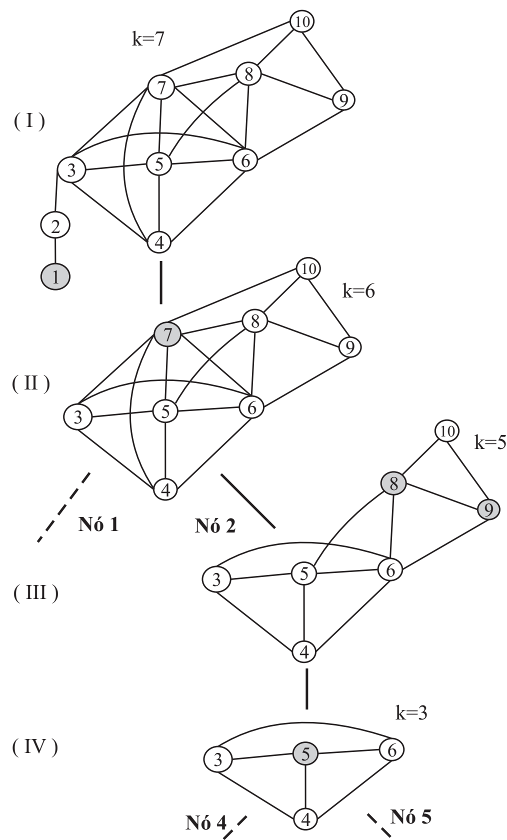


Figura 3.7: Exemplo de execução do Algoritmo de Niedermeier e Rossmanith [32]

Capítulo 4

Estruturas de Dados e Implementação

4.1 Considerações Iniciais

Neste capítulo, apresentamos detalhes de implementação e as estruturas de dados utilizadas nas implementações para o Problema da k-Cobertura por Vértices. Apresentamos duas implementações FPT paralelas no modelo BSP/CGM para o problema da k-Cobertura por Vértices. A entrada de ambas é um inteiro positivo k, que representa o tamanho da cobertura desejada, e o caminho para um arquivo texto. Este arquivo texto contém a representação do grafo em listas de adjacências.

Nossas implementações, assim como Cheetham *et al.*, utilizam os algoritmos de Buss e B1 nas primeiras duas etapas do algoritmo, na última etapa ao invés de utilizarmos o algoritmo B2 como feito por Cheetham *et al.* utilizamos Downey *et al.* [14] em uma implementação e Niedermeier et al. [32] em outra. Chamamos estas implementações, respectivamente, de Cheetham-Downey e Cheetham-Niedermeier. A Figura 4.1(a) apresenta o algoritmo de Cheetham *et al.* e os algoritmos Cheetham-Downey 4.1(b) e Cheetham-Niedermeier 4.1(c).

Cabe ressaltar que cada um destes algoritmos (Buss, B1, B2, Downey e Niedermeier) resolvem o problema da k-Cobertura por Vértices isoladamente, ou seja, sem necessitar, um da ajuda do outro. O objetivo de combiná-los é tentar encontrar uma solução robusta que localiza a resposta para o problema da maneira mais eficiente possível. Cada algoritmo segue uma idéia e uma linha de execução, neste sentido, um dos maiores desafios ao combinar estes algoritmos foi encontrar uma estrutura de dados que se adapta-se e atende-se eficientemente o algoritmo resultante do conjunto formado pela integração dos algoritmos. Durante a etapa de desenvolvimento dos nossos algoritmos realizamos várias provas de conceito, utilizando várias estruturas de dados e diferentes mecanismos para realizar operações sobre elas. O resultado apresentado é fruto de várias experiências e implementações realizadas.

Anteriormente, Hanashiro [25] havia apresentado uma implementação refinada e me-

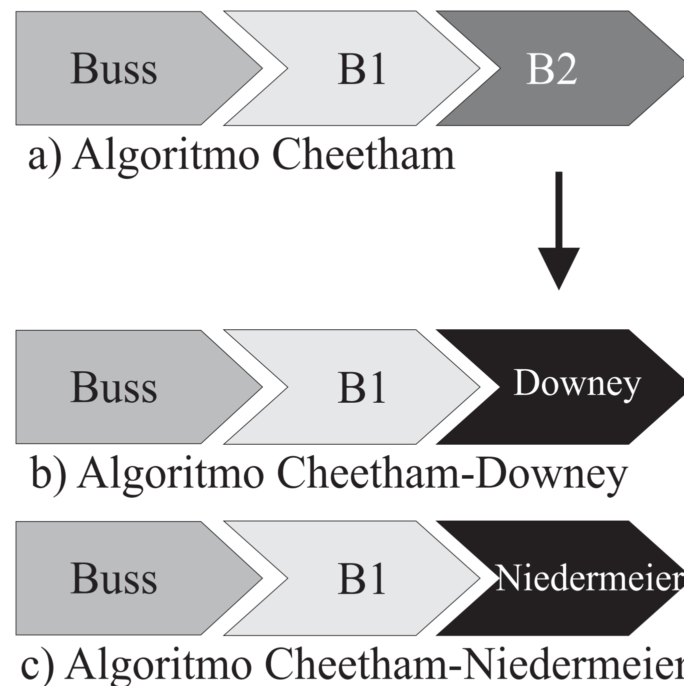


Figura 4.1: Implementação de Cheetham [6] e novas versões.

lhorada com base em Cheetham *et al.* 4.1(a). Em seu trabalho, Hanashiro realizou testes experimentais e obteve resultados superiores aos que Cheetham *et al.*. Também realizamos testes experimentais para medir o desempenho das nossas implementações. Utilizamos um executável de Hanashiro com as mesmas entradas e ambiente computacional e comparamos os resultados obtidos com os nossos. Os resultados experimentais são apresentados no capítulo 5.

Em nossas implementações, utilizamos a linguagem C++ junto com bibliotecas que implementam a especificação MPI.

Este capítulo está organizado nas seguinte seções: na Seção 4.2 apresentamos os detalhes de implementação da implementação Cheetham-Downey; na Seção 4.3 apresentamos os detalhes de implementação da implementação Cheetham-Niedermeier. Por fim, na Seção 4.4 apresentamos as considerações finais.

4.2 Implementação Cheetham-Downey

Nesta implementação, fazemos uma adaptação das idéias apresentadas por Cheetham *et al.* [6] descritas no capítulo anterior. Como mencionado antes, implementamos os algoritmos paralelos de Buss e B1, respectivamente, para a redução do problema ao núcleo e construção da árvore limitada de busca. Para processar cada nó resultante da árvore limitada de busca utilizamos o algoritmo seqüencial de Downey *et al.* [14] ao invés do algoritmo seqüencial B2, como feito por Cheetham *et al.* [6]. Na Figura 4.1(a), podemos verificar o algoritmo original de Cheetham e o algoritmo Cheetham-Downey 4.1(b).

Durante a implementação, surgiram várias versões de implementação, utilizando estratégias e estruturas de dados variadas. Dentre os vários desafios que enfrentamos para construir a implementação final, destacamos a preocupação com o balanceamento de carga entre os processadores. Em especial, com relação às operações que o algoritmo executa, destacamos a operação de validação da regra 2, da Fase de redução ao núcleo, do algoritmo de Downey *et al.*. Destacamos-na não apenas pelo alto custo de processamento, mas também pela sua natureza recorrente em todo o tempo de execução do algoritmo. A operação de validação que esta regra desempenha, é tentar localizar um par de vértices não adjacentes (u, v), tal que $|N(u) \cup N(v)| > k$.

Uma vez que, por vários motivos, optamos por utilizar lista de adjacências para representar o grafo, é fácil perceber que a tarefa de se procurar, eficientemente, dois vértices não adjacentes com uma dada propriedade, não é uma tarefa óbvia. Um algoritmo ingênuo poderia por exemplo, iniciar a busca em um vértice v qualquer, percorrer sua lista de adjacência, detectando cada vértice não adjacentes a ele, por exemplo u, e aplicar sobre o par (u, v) o teste se a propriedade requerida (por exemplo: $|N(u) \cup N(v)| > k$) é verdadeira. Desta forma, no pior caso, *todos* os pares de vértices não adjacentes seriam testados, mesmos aqueles sem a menor chance de possuir tal propriedade.

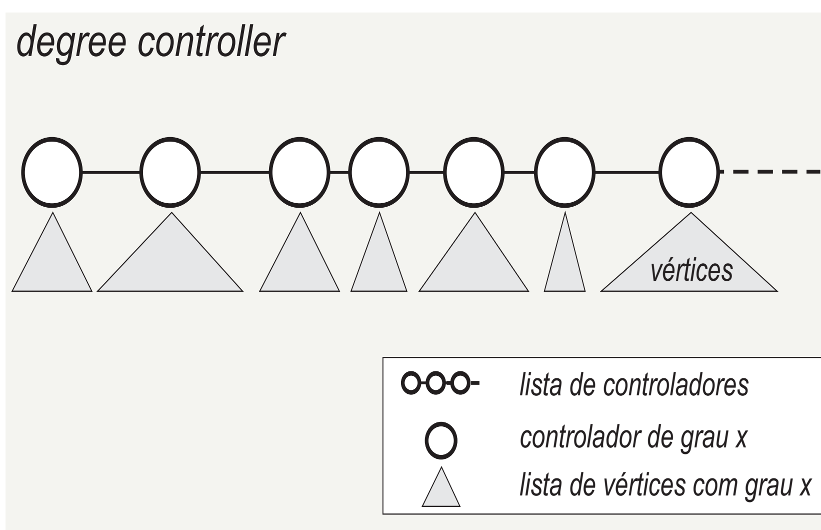


Figura 4.2: Estrutura de dados *degree controller*.

Precisávamos de um algoritmo que, rapidamente, localizasse e testasse *apenas* os pares de vértices *não adjacentes* que possuíssem alguma chance de possuir tal propriedade. Para suprir tal necessidade, construímos uma estrutura de dados que chamamos de *degree controller*. Como podemos observar na Figura 4.2, esta estrutura é formada por uma lista de controladores de grau, onde cada controlador representa um valor de grau presente no grafo. Cada controlador de grau controla uma lista de vértices, onde cada vértice desta lista possui o mesmo valor de grau correspondente ao valor do controlador. Além disso, esta lista de controladores é mantida ordenada. Com o auxílio desta estrutura é fácil construir um algoritmo que percorre a lista de controladores de grau, do maior para o menor valor, até que possa parar, testando a condição de não adjacência dos vértices e a propriedade $|N(u) \cup N(v)| > k$. Cabe ressaltar que esta estrutura também possibilitou

estratégias de ganho de processamento em outros pontos da implementação do algoritmo.

4.3 Implementação Cheetham-Niedermeier

Nesta implementação, assim como em Cheetham-Downey, fazemos uma adaptação das idéias apresentadas por Cheetham *et al.* [6]. Utilizamos os algoritmos de Buss e B1 para a redução do problema ao núcleo e a construção da árvore limitada de busca, respectivamente e para processar cada nó resultante da árvore limitada de busca, utilizamos o algoritmo seqüencial de Niedermeier *et al.* [32], ao invés do algoritmo B2, como feito por Cheetham *et al.* [6]. Na Figura 4.1(a), podemos verificar o algoritmo original de Cheetham e o algoritmo Cheetham-Niedermeier 4.1(c).

Como este algoritmo foi implementado após o Cheetham-Downey, muitos desafios já haviam sido superados e estratégias pensadas. Desta forma, algumas destas estratégias puderam ser adaptadas e utilizadas nesta implementação. Uma diferença na característica destas duas implementações é que, enquanto o algoritmo de Downey cria um número pequeno de ramificações, mas de oneroso custo computacional, o algoritmo de Niedermeier, por sua vez, cria um número ligeiramente maior de ramificações mas, com operações que puderam ser implementadas de forma a não consumir grande custo computacional. Isto leva a questão: o que é mais rápido de se resolver, um conjunto grande de ramificações, mas rápido de se resolver ou um conjunto pequeno, porém demorado? Na prática, de acordo com o resultado de nossos testes experimentais, isto depende da entrada. Mas, a implementação Cheetham-Downey obteve resultados melhores para a maioria das entradas submetidas.

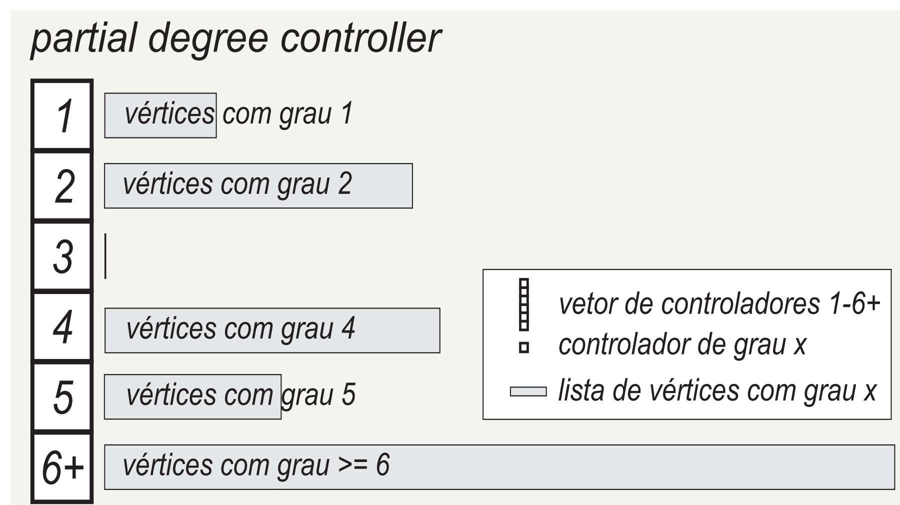


Figura 4.3: Estrutura de dados *partial degree controller*.

A Figura 4.3 apresenta a estrutura de dados *partial degree controller* utilizada para manutenção e controle dos graus dos vértices desta implementação. Como podemos perceber, existe um vetor fixo de controladores. Cada controlador possui um valor entre 1

e 6 que representa quais vértices estarão sendo controlados por ele. Desta forma, o controlador com valor 1 possui uma lista com todos os vértices que possuem grau com valor igual a 1 (ou apenas um vizinho), o controlador com valor 2 possui uma lista com todos os vértices que possuem valor de grau igual a 2 (ou dois vizinhos), e assim, sucessivamente, até o controlador com valor 5. O controlador com valor 6 possui uma lista com todos os vértices que possuem valores de grau maiores ou iguais a 6. Este controle simplificado de graus possibilitou atender todos os casos e regras descritos no algoritmo, assim como, as estratégias de implementação utilizadas.

A Figura 4.4 apresenta a estrutura de dados base para a representação dos vértices a arestas do grafo. Esta estrutura é semelhante para ambas implementações Cheetham-Downey e Cheetham-Niedermeier diferindo apenas nos ponteiros de ligação com as estruturas de manutenção de graus *degree controller* e *partial degree controller*, respectivamente. Esta estrutura é formada por uma lista vértices, contendo todos os vértices do grafo G. Cada vértice V possui um ponteiro para sua lista de arestas e cada nó que representa a aresta de um vértice possui um ponteiro para o vértice que o controla e para a aresta inversa (por exemplo, aresta (v_1, v_2) e aresta (v_2, v_1)).

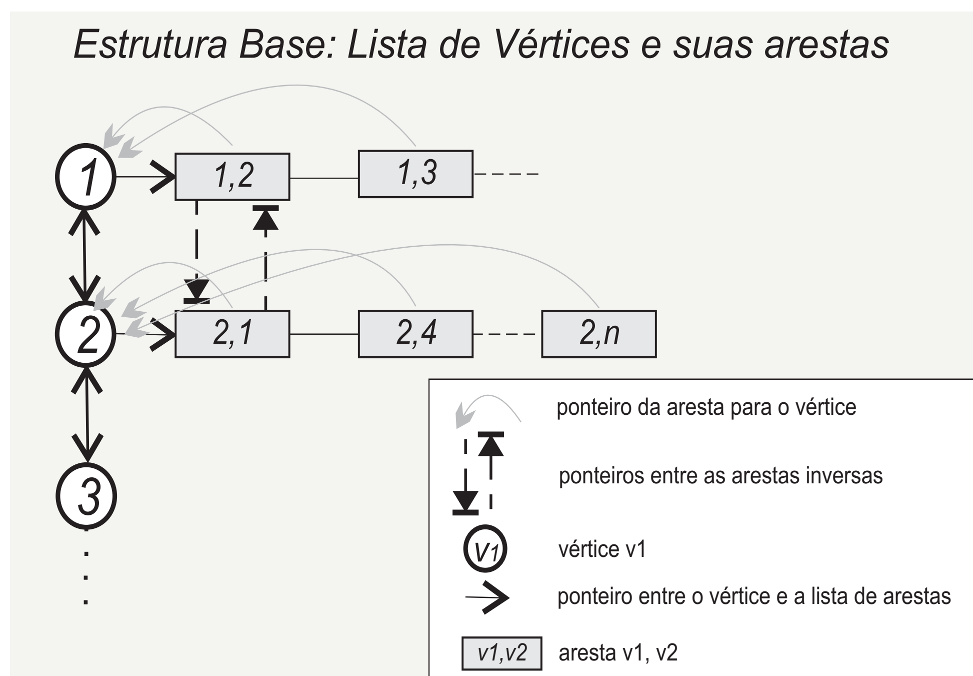


Figura 4.4: Estrutura de dados Base: Lista de vértices e suas arestas.

4.4 Considerações Finais

Neste capítulo, apresentamos as implementação Cheetham-Downey e Cheetham-Niedermeier e as estruturas de dados mais importantes utilizadas por elas. Consideramos que boas estratégias de implementação e estruturas de dados adequadas contribuíram para o bom desempenho dos algoritmos. Para comprovar o bom desempenho das nossas implementações

realizamos testes experimentais e comparamos o tempo de execução delas entre si, e com relação a Hanashiro. Obtivemos resultados promissores. Estes resultados são apresentados no próximo capítulo.

Capítulo 5

Resultados Experimentais

5.1 Considerações Iniciais

Neste capítulo, apresentamos os resultados experimentais obtidos com as implementações BSP/CGM dos algoritmos Cheetham-Downey e Cheetham-Niedermeier para o Problema da k-Cobertura por Vértices que utilizam as estruturas de dados apresentadas no Capítulo 4. Na Seção 5.2, apresentamos o ambiente computacional e a metodologia utilizada para computar os resultados obtidos nos experimentos. Na Seção 5.3, mostramos os grafos de entrada utilizados nos experimentos. Na Seção 5.4, discutimos os resultados obtidos por nossas implementações comparadas às implementações de Cheetham *et al.* [6] e Hanashiro [25].

5.2 Ambiente Computacional e Metodologia

Os algoritmos paralelos FPT Cheetham-Downey e Cheetham-Niedermeier, apresentados no capítulo anterior, foram implementados utilizando a linguagem C/C++ e junto com as bibliotecas que implementam a especificação MPI.

Executamos nossos algoritmos em dois ambientes computacionais: em um *Cluster* e em uma Grade Computacional. O *Cluster* era composto por 12 nós: uma máquina AMD Athlon(tm) 1800+, com 1GB de memória RAM; uma máquina Intel(R) Pentium(R) 4 CPU 1.70GHz, com 1GB de memória de RAM; três máquinas Pentium IV 2.66GHz, com 512MB de memória RAM; uma máquina Pentium IV 2.8GHz, com 512MB de memória RAM; uma máquina Pentium IV 1.8GHz, com 480MB de memória RAM; quatro máquinas AMD Athlon(tm) 1.66GHz, com 480MB de memória RAM; uma máquina AMD Sempron(tm) 2600+, com 480MB de memória RAM. Os nós estavam conectados por um *switch fast-Ethernet* de 1Gb. Cada nó executava o sistema operacional Linux Fedora 6 com g++ 4.0 e MPI/LAM 7.1.2. No caso da Grade Computacional, utilizamos o mesmo ambiente computacional descrito acima com a Grade InteGrade versão 0.4 [19].

Todos os tempos apresentados são em segundos e incluem o tempo de leitura dos

dados de entrada, desalocação das estruturas de dados utilizadas e impressão da saída. Os resultados apresentados são a média de 30 execuções realizadas.

5.3 Grafos de Entrada

Em nossos experimentos utilizamos grafos de entrada referentes aos aminoácidos[1]: Somatostatin, WW, Kinase, SH2 (*src-homology domain 2*) e PHD (*pleckstrin homology domain)*, seqüencias destes aminoácidos podem ser coletadas no banco de dados do NCBI (http://www.ncbi.nlm.nih.gov). Na Tabela 5.1, apresentamos um resumo das características dos grafos de entrada utilizados nas implementações, que incluem o nome do aminoácido, o número de vértices ($|V|$), o número de arestas ($|E|$), e o tamanho da cobertura desejada (k).

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

Tabela 5.1: Grafos utilizados nos experimentos e suas características

5.4 Comparação dos Resultados

Nesta seção, iremos discutir os resultados de nossos experimentos com as implementações Cheetham-Downey e Cheetham-Niedermeier. Além disso, iremos comparar os resultados com os resultados da implementação, baseada em Cheetham *et al.* [6], de Hanashiro [25]. Comparamos também, o desempenho das implementações, para cada Grafo executando em ambos ambientes: *Cluster* e Grade Computacional (neste caso, o InteGrade versão 0.4). Nas figuras e tabelas a seguir, iremos nos referir a implementação de Hanashiro como Cheetham*.

Observamos na Figura 5.1 que, para o grafo Somatostatin, os tempos médios, obtidos pela execução com 3 processadores da implementação Cheetham-Downey, foram menores que as outras implementações em ambos ambientes. Como também podemos comprovar pela Figura 5.2, mais uma vez o desempenho da implementação Cheetham-Downey foi superior às demais implementações em ambos ambientes.

O grafo Somatostatin pode ser considerado um grafo pequeno, visto que em todas as execuções o tempo consumido com 9 processadores foi maior que o tempo consumido utilizando-se 3 processadores, evidenciando o custo predominante com as comunicações.

[1]Estes grafos foram gentilmente cedidos pelo professor Frank Dehne(Calerton University).

Além disso, dentro os grafos utilizados como entrada, o Somatostatin é o que apresenta o menor número de arestas por vértices, cerca de 60, contra, por exemplo, 219 do grafo PHD.

De fato, a implementação Cheetham-Downey destacou-se na execução para o grafo Somatostatin, e acreditamos que o motivo seja, como mencionado anteriormente, a característica do algoritmo de abrir poucas ramificações aliado ao pequeno tamanho da entrada.

Também podemos verificar, nestas figuras, que o desempenho em ambos números de processadores, manteve-se superior no *Cluster* para as implementações Cheetham-Downey e Cheetham-Niedermeier, e inferior para a implementação Cheetham*.

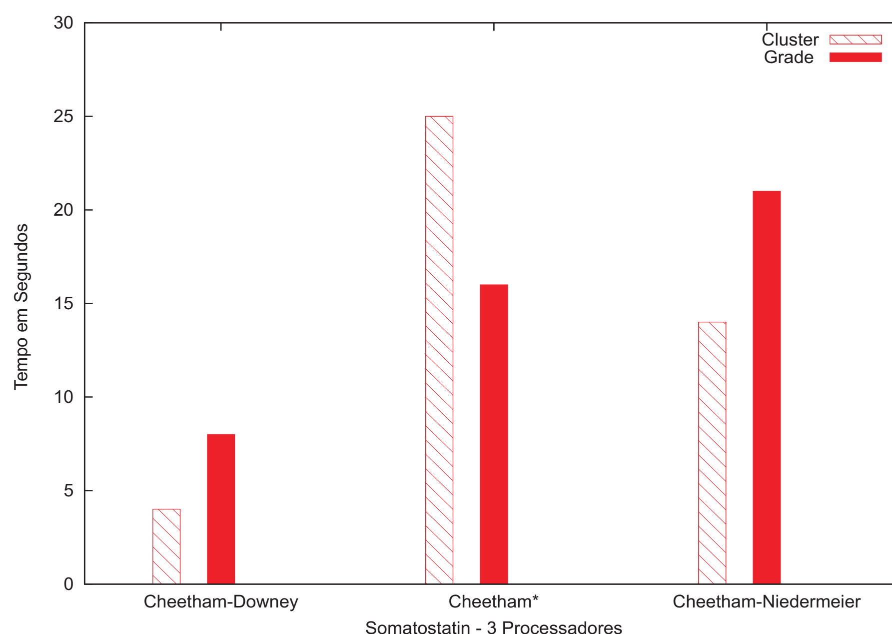


Figura 5.1: Comparação *Cluster* x Grade Somatostatin - 3 Processadores .

Nas Figuras 5.3 e 5.4 apresentamos o desempenho das implementações, em ambos ambientes: *Cluster* e Grade, com 3 e 9 processadores, para o grafo WW. Observando estas figuras, podemos observar o bom desempenho da implementação Cheetham-Niedermeier, cuja execução foi 6,31 vezes mais rápida que a pior execução apresentada pelas outras implementações, na Grade com 3 processadores, e 22,88 vezes mais rápida que a pior execução apresentada pelas outras implementações no *Cluster* com 3 processadores.

Dentre as entradas submetidas a implementação Cheetham-Niedermeier, o grafo de entrada WW, foi a que mais destacou a eficiência desta implementação, obtendo resultados superiores às outras implementações em ambos ambientes e número de processadores. Não é fácil apontar um motivo exato pelo qual a implementação Cheetham-Niedermeier se sobressaiu perante as outras. Podemos apenas, apontar como determinante, que são as características intrínsecas em cada Grafo de entrada que determinam o desempenho de execução de uma implementação. Porém, saber quais são estas características e como elas afetam o desempenho exige um estudo complexo e detalhado, de fato, este é dos temas de nossos trabalhos futuros.

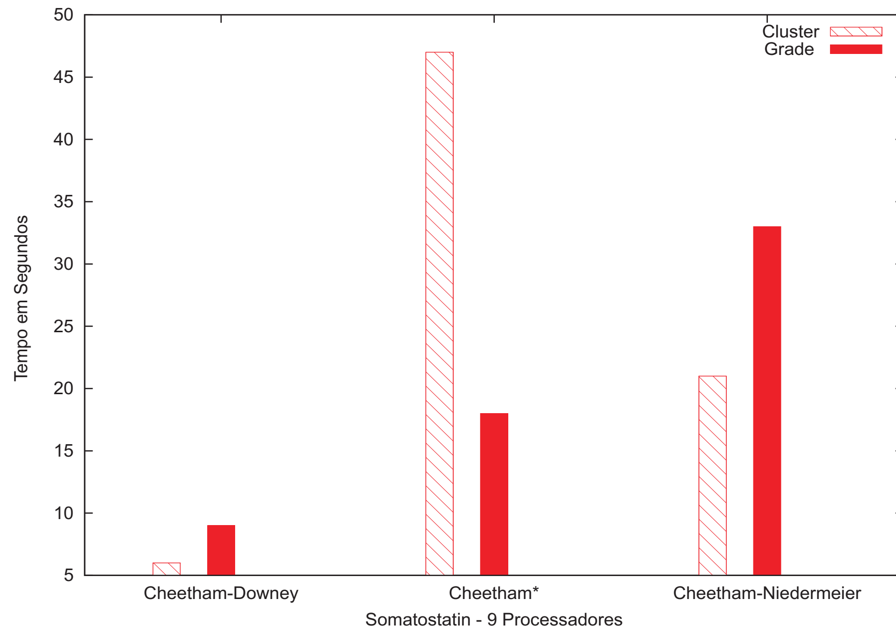


Figura 5.2: Comparação *Cluster* x Grade Somatostatin - 9 Processadores.

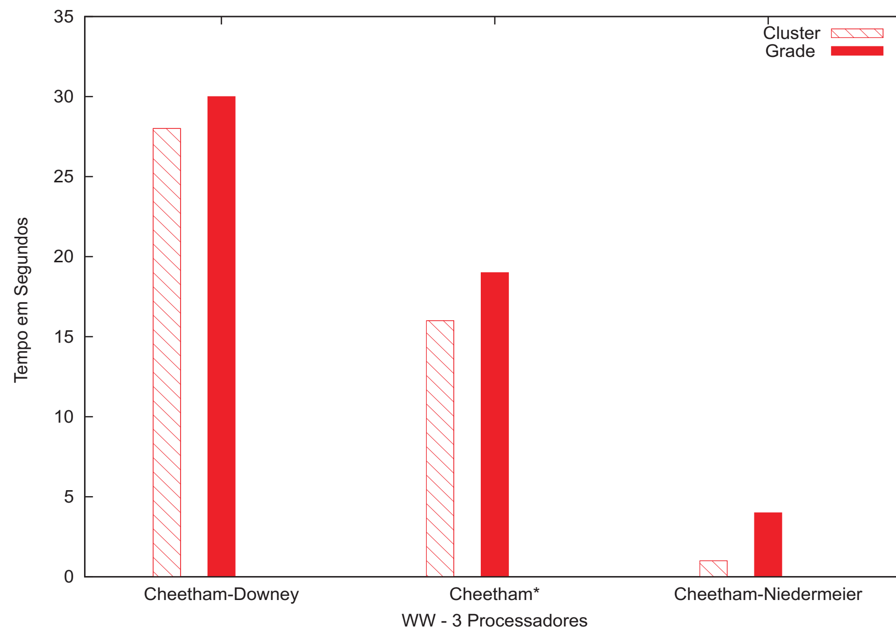


Figura 5.3: Comparação *Cluster* x Grade WW - 3 Processadores.

Apresentamos, para o grafo Kinase, na Figura 5.5, os resultados obtidos com as implementações em *Cluster* e Grade com 3 processadores. Na Figura 5.6 apresentamos os resultados obtidos com as implementações utilizando 9 processadores. Nestas figuras, podemos observar que as implementações Cheetham-Downey e Cheetham* tiveram desempenho bastante semelhante, e bem superior ao desempenho apresentado pela implementação Cheetham-Niedermeier. O melhor tempo obtido foi de 8,73 segundos da implementação Cheetham* em *Cluster* com 9 processadores. Nesta mesma configuração, a

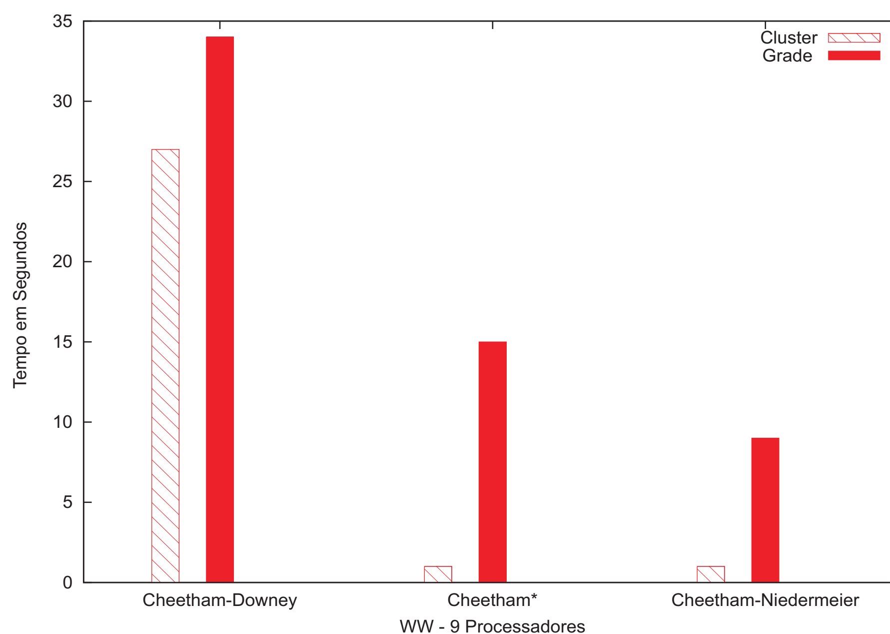


Figura 5.4: Comparação *Cluster* x Grade WW - 9 Processadores.

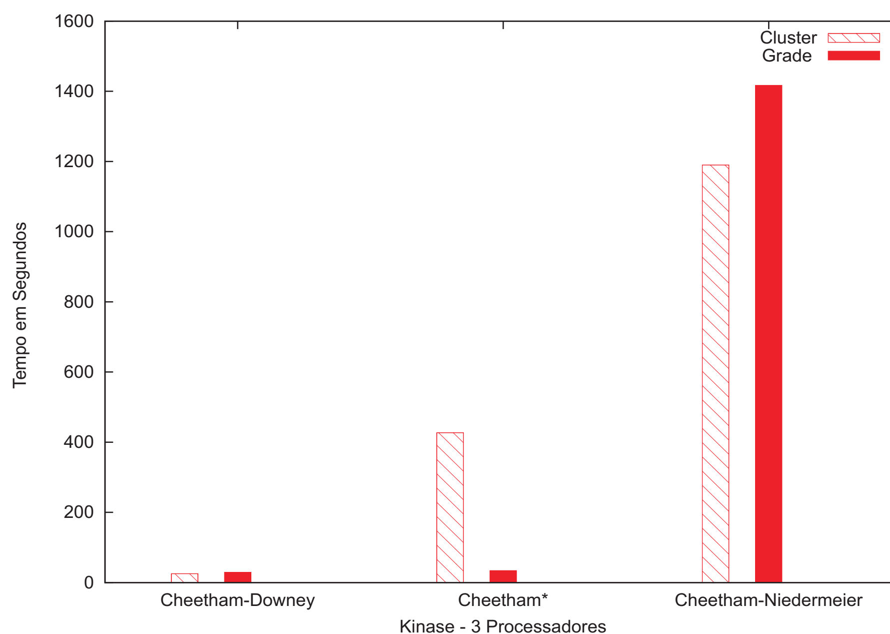


Figura 5.5: Comparação *Cluster* x Grade Kinase - 3 Processadores.

implementação Cheetham-Downey gastou 24,52 segundos, e a implementação Cheetham-Niedermeier necessitou de 1215,56 segundos para encontrar a resposta.

A Figura 5.7 apresenta os resultados obtidos com as implementações, tanto em *Cluster* como na Grade, para o grafo SH2 utilizando-se 3 processadores. A Figura 5.7 apresenta os resultados obtidos com as implementações, tanto em *Cluster* como na Grade, para o grafo SH2 utilizando-se 9 processadores. Como podemos observar, o grafo SH2 foi o que produziu a maior variação de desempenho entre as implementações, nas diferentes

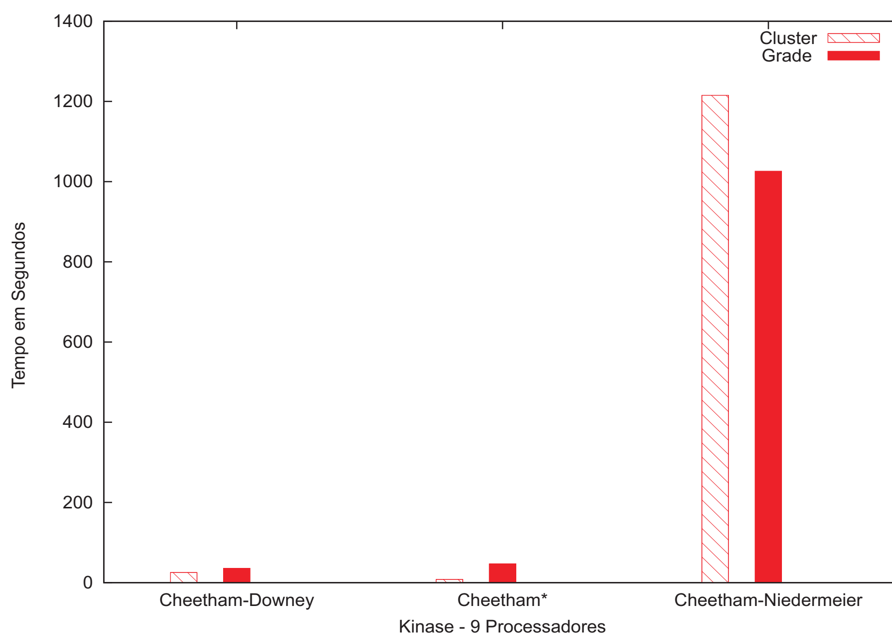


Figura 5.6: Comparação *Cluster* x Grade Kinase - 9 Processadores.

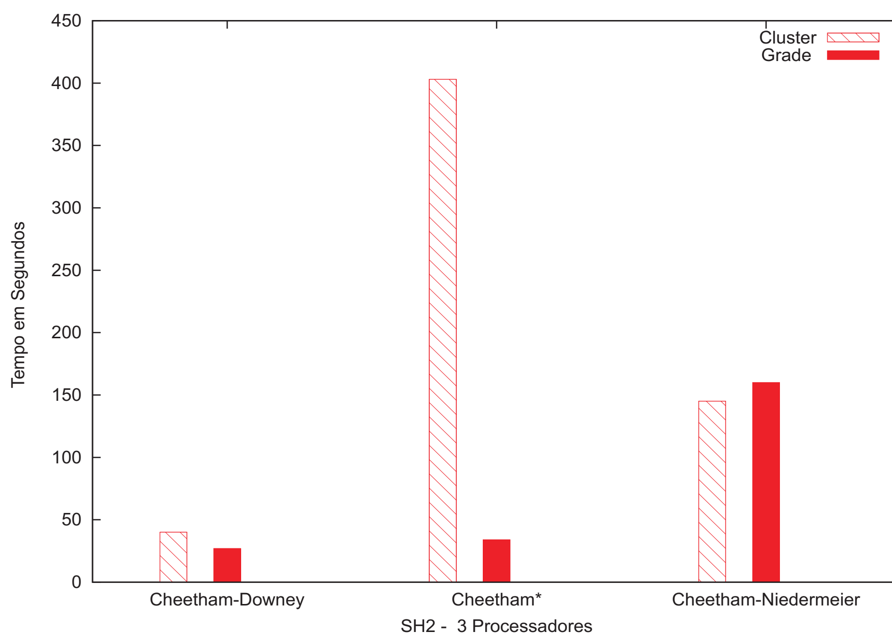


Figura 5.7: Comparação *Cluster* x Grade SH2 - 3 Processadores.

configurações de execução.

Para a implementação Cheetham-Downey, na execução com 3 processadores, o desempenho do *Cluster* foi pouco superior à grade. Mas, na execução com 9 processadores o resultado se inverte: o desempenho do *Cluster* foi inferior à grade. A implementação Cheetham*, no *Cluster*, com 3 processadores, apresenta o pior resultado gastando cerca de 403 segundos. Quase 4 vezes mais tempo do que a segunda pior execução, também em *Cluster*, com 3 processadores, que gastou cerca de 160 segundos. Já na execução com 9

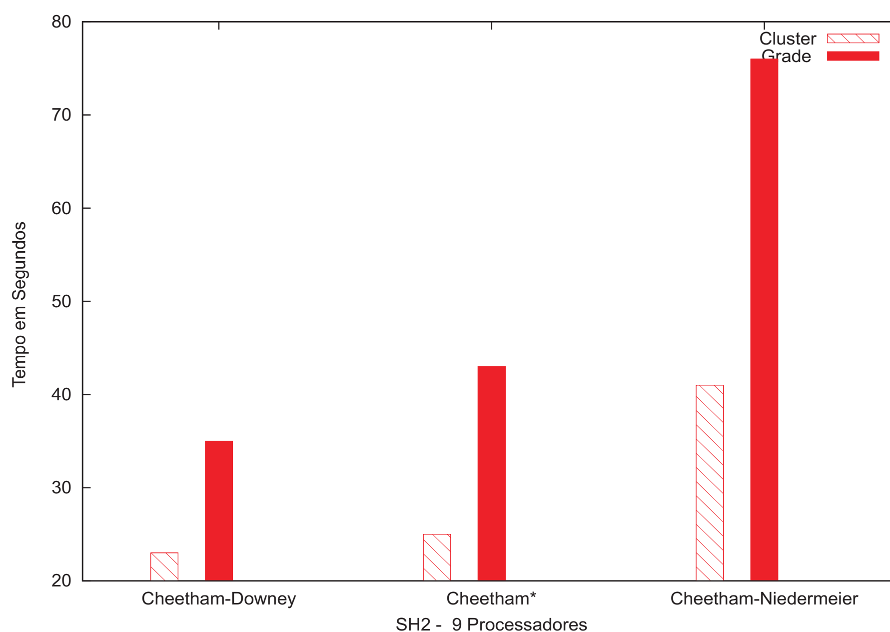


Figura 5.8: Comparação *Cluster* x Grade SH2 - 9 Processadores.

processadores, no *Cluster*, a implementação Cheetham* necessitou de apenas 25 segundos para encontrar a resposta. Acreditamos que a natureza aleatória de escolhas de vértices (que ocorre apenas quando, na escolha do próximo vértice, o estado momentâneo do grafo determina que nenhum vértice tem precedência sobre outro) tenham sido determinantes para estes resultados.

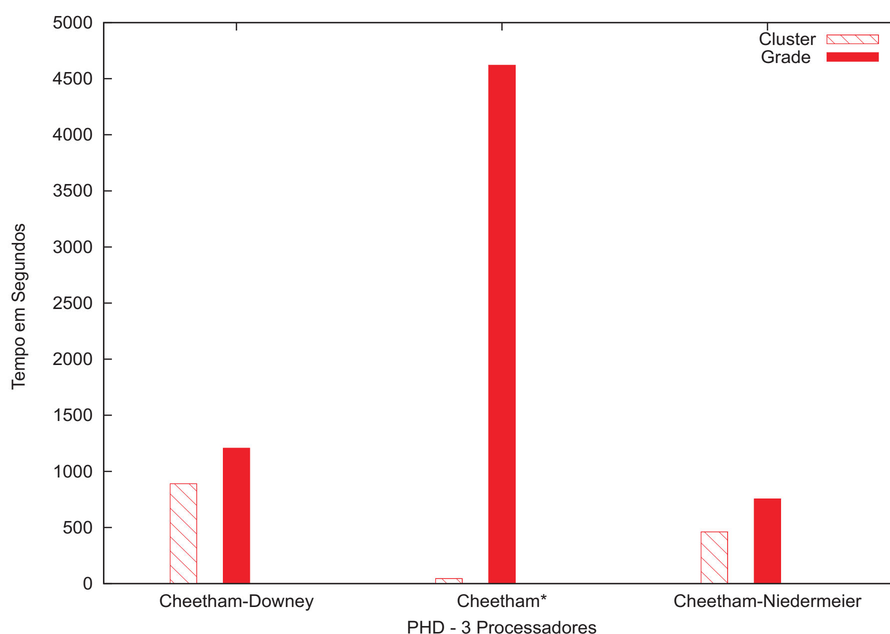


Figura 5.9: Comparação *Cluster* x Grade PHD - 3 Processadores.

Nas Figuras 5.9 e 5.10, apresentamos, para o grafo PHD, os resultados obtidos com as implementações utilizando *Cluster* e Grade, com 3 e 9 processadores. Para a imple-

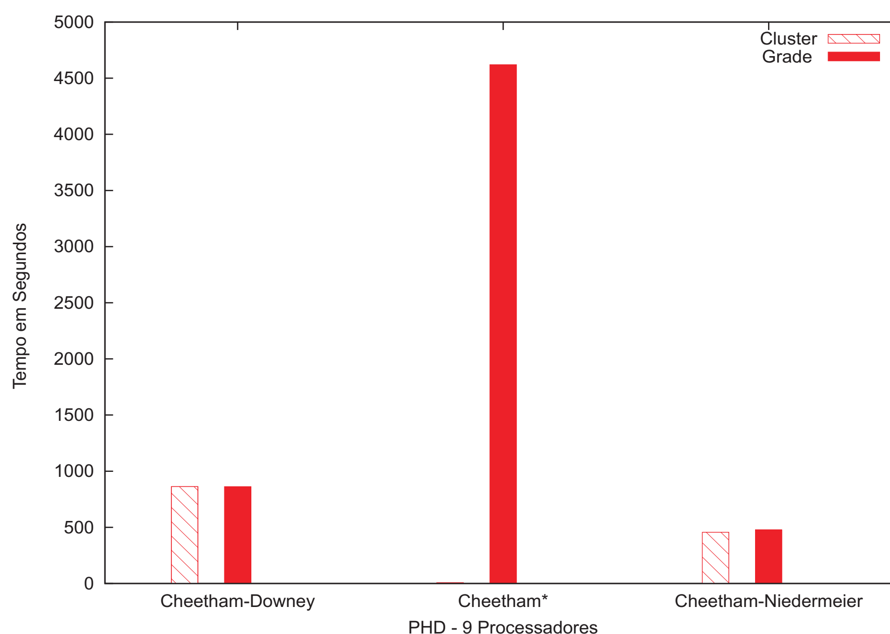


Figura 5.10: Comparação *Cluster* x Grade PHD - 9 Processadores.

mentação Cheetham*, na Grade, para 3 e 9 processadores, não conseguimos obter uma resposta em menos de 4600 segundos. Porém, utilizando *Cluster* esta implementação conseguiu os melhores resultados obtendo uma resposta em apenas 44,14 segundos com 3 processadores e 7,59 segundos com 9 processadores. As outras implementações necessitavam de pelo menos 400 segundos para conseguir a resposta.

As Figuras 5.1 a 5.10, apresentaram uma visão comparativa entre as implementações, fixando o grafo de entrada e o número de processadores, em dois ambientes: *Cluster* e Grades computacionais.

A seguir, apresentaremos figuras que mostram uma visão comparativa entre as implementações, simultaneamente para todos os grafos de entrada, fixando o ambiente e o número de processadores. Em cada figura, é identificado o ambiente utilizado e o número de processadores, e então são apresentados os resultados obtidos com as implementações Cheetham-Downey, Cheetham* e Cheetham-Niedermeier, para cada um dos grafos de entrada: Somatostatin, WW, Kinase, SH2, e PHD. Estas figuras mostram, simultaneamente, para cada grafo de entrada, seu comportamento diante de cada implementação.

Nas Figuras 5.11 e 5.12, podemos observar o desempenho dos grafos, comparativamente entre cada implementação, de acordo com os resultados obtidos na execução em *Cluster* utilizando-se 3 e 9 processadores, respectivamente. Na Figura 5.11, por exemplo, evidenciamos a trajetória da linha que representa o tempo de execução do grafo PHD, cujos menores valores encontram-se nas extremidades (referente as implementações Cheetham-Downey e Cheetham-Niedermeier) e maior valor encontra-se no centro (referente a implementação Cheetham*). Ao oposto, temos a trajetória da linha que representa o tempo de execução do grafo Somatostatin, cujos maiores valores encontram-se nas extremidades (referente as implementações Cheetham-Downey e Cheetham-Niedermeier) e menor valor encontra-se no centro (referente a implementação Cheetham*). Esta visão

deixa claro que uma determinada implementação pode se sobressair para uma determinada entrada, mas não para outra.

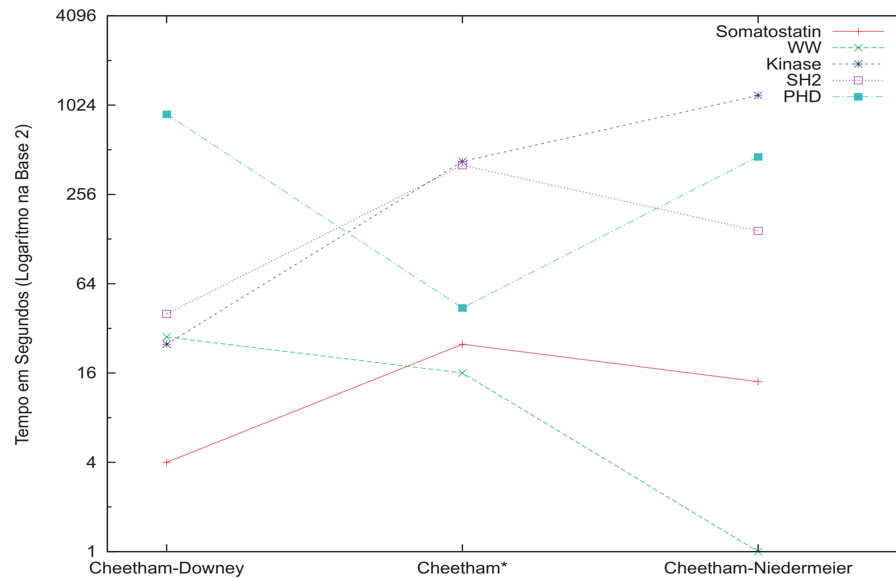


Figura 5.11: Comparação dos Grafos no *Cluster* - 3 Processadores.

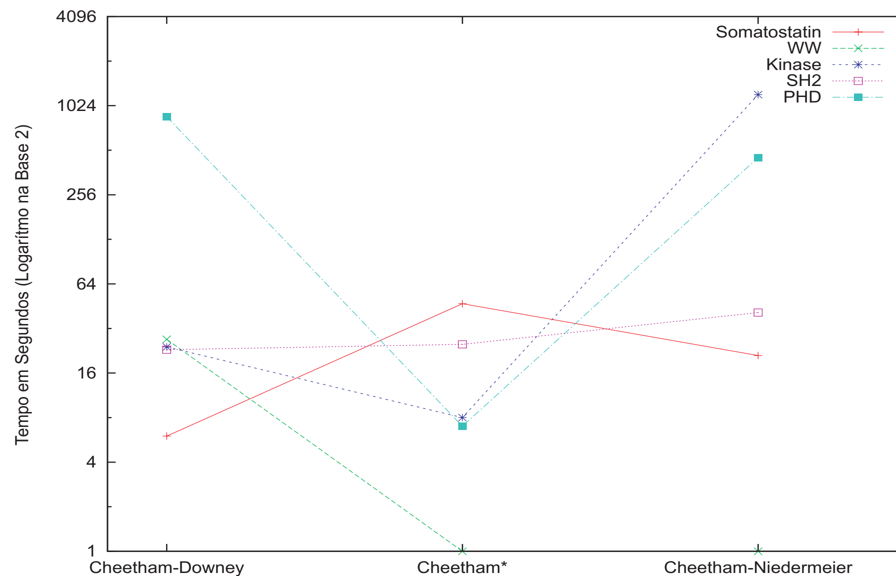


Figura 5.12: Comparação dos Grafos no *Cluster* - 9 Processadores.

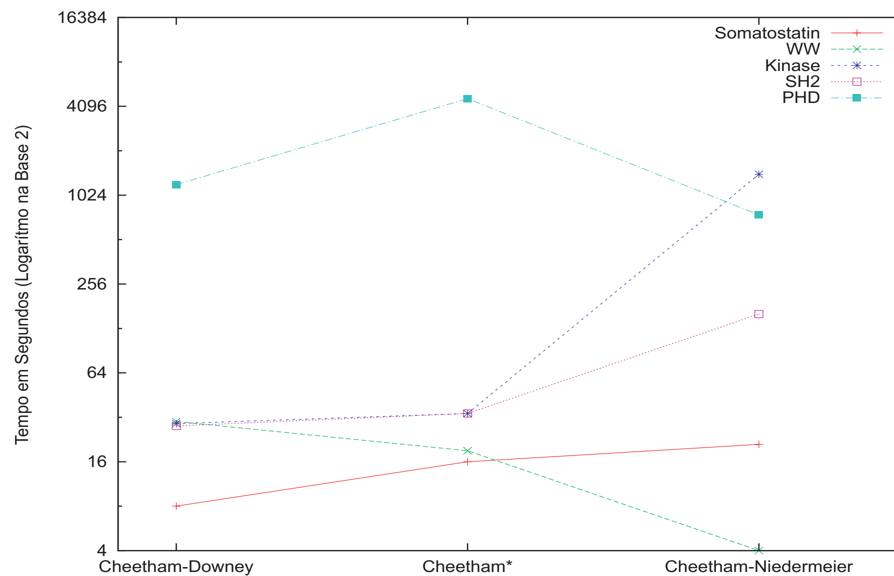


Figura 5.13: Comparação dos Grafos na Grade - 3 Processadores.

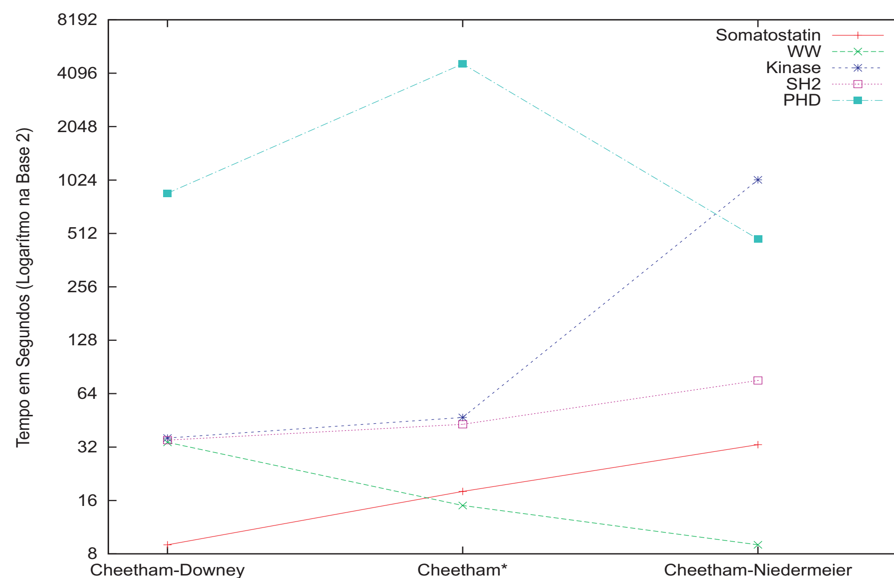


Figura 5.14: Comparação dos Grafos no Grade - 9 Processadores.

Nas Figuras 5.13 e 5.14 podemos observar o desempenho dos grafos em relação às implementações, utilizando-se a grade com 3 e 9 processadores, respectivamente. Como podemos perceber, o comportamento geral na grade, com a alteração do número de processadores, demonstrou-se bastante estável, diferente do comportamento observado no *Cluster*. No *Cluster*, de maneira geral, a alteração no número de processadores, afetou apenas o comportamento da implementação Cheetham*, embora difícil de visualizar nas Figuras 5.11 e 5.12, o comportamento das implementação Cheetham-Downey e Cheetham-Niedermeier permaneceram bastante estáveis. Conforme dito anteriormente, acreditamos que o fato do comportamento da implementação Cheetham* ter se alterado, se justifique pela natureza aleatória de algumas escolhas que o algoritmo desta implementação executa.

As Figuras 5.11 a 5.14 apresentaram o desempenho comparativo dos grafos de entrada, dentro de um mesmo ambiente e número de processadores. As próximas figuras apresentarão, para cada grafo de entrada, uma visão comparativa de desempenho entre as implementações, simultaneamente para os dois ambientes (*Cluster* e Grade) com diferentes número de processadores.

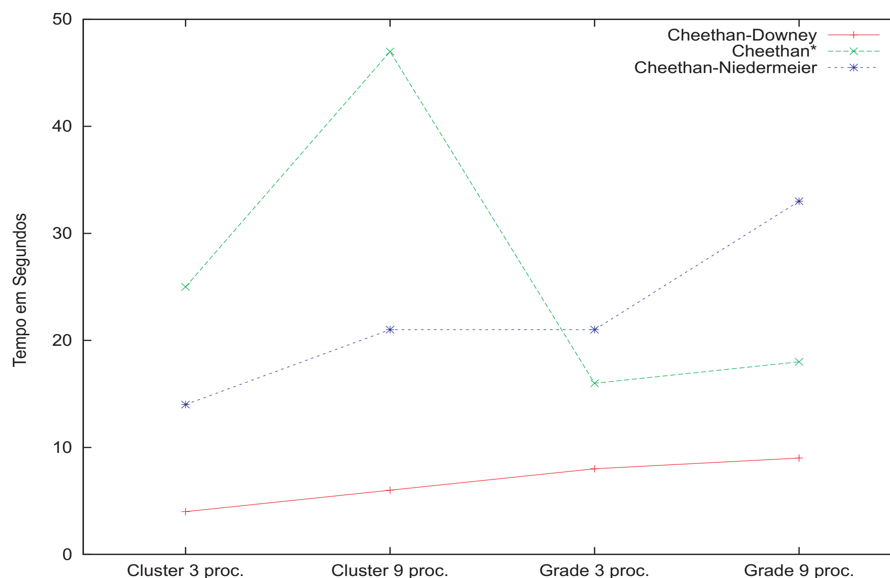


Figura 5.15: Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - Somatostatin.

Observando a Figura 5.15, podemos perceber que à medida que o número de processadores aumenta, o desempenho é piorado em todas as implementações, isto se justifica pelo fato de o custo de comunicação se sobressair à melhoria de desempenho obtida com o aumento do número de processadores. Outra informação que podemos extrair é que, por uma grande diferença, a implementação Cheetham-Downey é superior as demais, em todas as configurações (ambiente e número de processadores).

Na Figura 5.16, observamos que a implementação Cheetham-Niedermeier apresenta melhores resultados que as demais em todos os casos. Exceto pelo resultado utilizando-se *Cluster* e 9 processadores, cuja a implementação Cheetham* se aproxima, os demais resultados obtidos pela implementação Cheetham-Niedermeier são muito melhores que as demais implementações.

Observamos os resultados obtidos com o grafo Kinase na Figura 5.17. Nesta figura podemos ver que a implementação Cheetham-Downey tem melhor desempenho que a demais, para a maioria dos casos. Com desempenho próximo está a implementação Cheetham* que consegue obter melhor resultado utilizando *Cluster* e 9 processadores.

Na Figura 5.18 que apresenta os resultados obtidos para o grafo SH2, observamos mais uma vez que a implementação Cheetham-Downey obteve melhores resultados que as demais em todos os casos. A implementação Cheetham*, exceto pelo resultado obtido utilizando *Cluster* e 3 processadores, possui resultados próximos a implementação Cheetham-Downey.

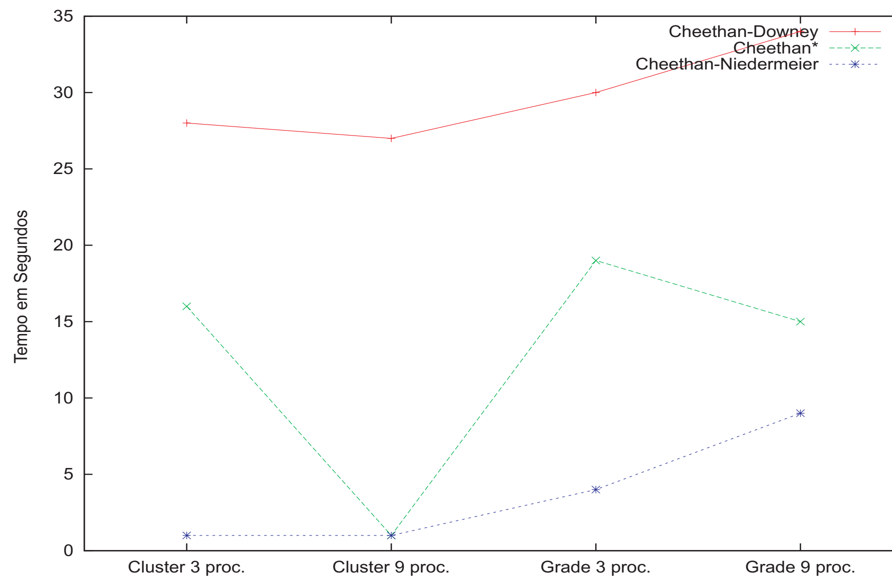


Figura 5.16: Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - WW.

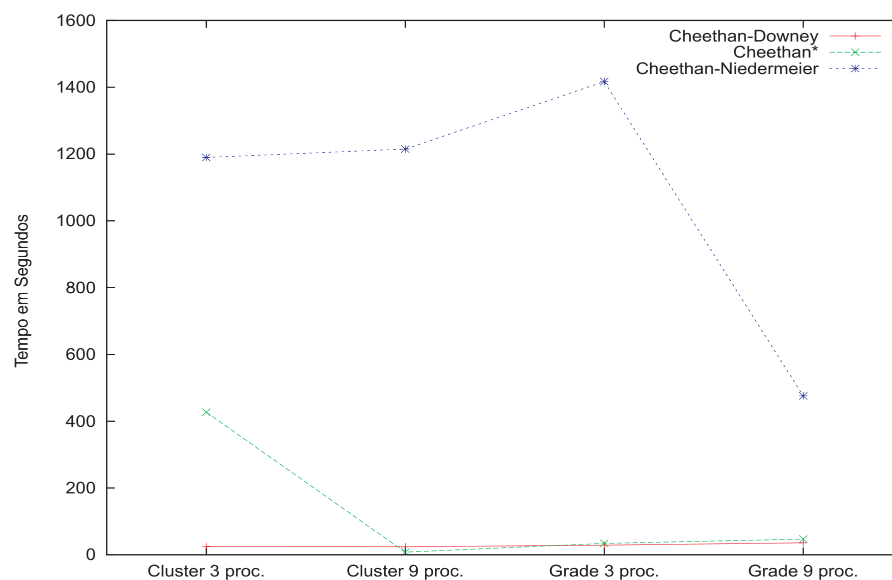


Figura 5.17: Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - Kinase.

Na Figura 5.19, observamos os resultados obtidos com o grafo PHD. Para este grafo, a implementação Cheetham* executada no *Cluster* obteve os melhores resultados, em contrapartida, na grade, não conseguimos obter resultados com menos 4600 segundos. As implementações Cheetham-Downey e Cheetham-Niedermeier tiveram comportamentos estáveis em ambos ambientes (*Cluster* e grade). A implementação Cheetham-Niedermeier obteve os melhores resultados na grade.

As Figuras 5.15 a 5.19 apresentaram uma visão de desempenho geral da implementações sobre os grafos de entrada. Para cada grafo, dada uma implementação, era apresentado o seu desempenho conforme o número de processadores era alterado e, além disso, as implementações eram comparadas entre si.

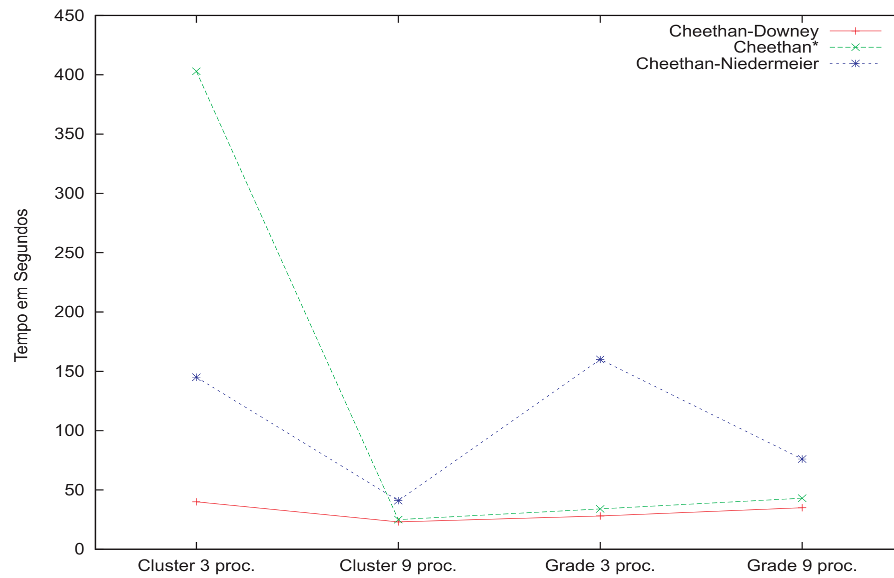


Figura 5.18: Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - SH2.

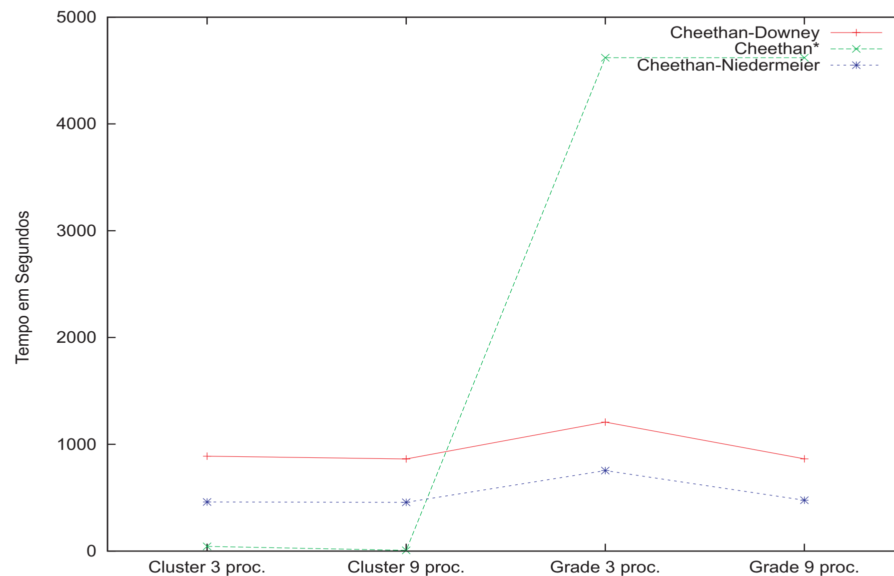


Figura 5.19: Comparação dos Grafos no *Cluster* e Grade com 3 e 9 Processadores - PHD.

Diante do exposto, podemos comprovar que o desempenho das implementações no *Cluster* manteve-se superior à Grade na maioria das execuções. Considerando o resultado das médias de execuções, temos que, em 86% dos casos, o desempenho do *Cluster* foi superior, e, em apenas 14% dos casos, o desempenho Grade foi superior. Considerando, por implementação, temos que em 93% dos casos, o *Cluster* apresentou melhor desempenho utilizando a implementações Cheetham-Downey, em 90% dos casos, o *Cluster* apresentou melhor desempenho com a implementação Cheetham-Niedermeier, e em, apenas 40% dos casos *Cluster* apresentou melhor desempenho na implementação Cheetham*.

A variação de eficiência das implementações conforme a entrada e o ambiente onde são executadas nos dão indícios de que é difícil construir um algoritmo único que seja capaz

de se sobressair em todas as situações. Embora, seja óbvio que algumas características dos grafos da entrada sejam determinantes para o bom desempenho de um algoritmo que o executa, ainda é bastante complexo entender que características são estas e como elas afetam o desempenho.

Com relação ao desempenho das implementações Cheetham-Downey e Cheetham-Niedermeier, podemos comprovar que, a implementação Cheetham-Downey tem desempenho superior, para a maioria das entradas. A exceção ocorre para a entrada WW (Figuras 5.3, 5.4) e PHD (Figuras 5.9, 5.10). Afim de se obter um bom resultado para todas as entradas seria possível desenvolver um algoritmo que intercalasse ambas implementações e devolvesse sempre o melhor resultado dentre elas.

A seguir, apresentamos os resultados obtidos pela implementação Cheetham* (de Hanashiro) e de nossas implementações (Cheetham-Downey e Cheetham-Niedermeier). As Tabelas 5.2 a 5.5 apresentam, respectivamente, os resultados obtidos utilizando Grade Computacional e 3 processadores, Grade Computacional e 9 processadores, *Cluster* e 3 processadores e *Cluster* e 9 processadores. O *Speedup*** refere-se a taxa de melhoria com relação à implementação Cheetham* e o melhor resultado obtido por nossas implementações (Cheetham-Downey e Cheetham-Niedermeier). Os tempos apresentados são em segundos e se referem ao resultado da média de 30 execuções.

Grafo	Cheetham*	Cheetham-Downey	Cheetham-Niedermeier	*Speedup***
Somatostatin	16,37	8,49	21,02	1,93
WW	19,30	30,56	4,84	3,99
Kinase	34,62	29,78	1417,95	1,16
SH2	34,87	28,78	160,67	1,21
PHD	4620	1207,06	754,66	6,12

Tabela 5.2: Speedup - Grade com 3 Processadores

Grafo	Cheetham*	Cheetham-Downey	Cheetham-Niedermeier	*Speedup***
Somatostatin	18,66	9,84	33,45	1,90
WW	15,12	34,51	9,30	1,63
Kinase	47,30	36,81	1026,44	1,28
SH2	43,80	35,33	76,64	1,24
PHD	4620	863,03	476,75	9,69

Tabela 5.3: Speedup - Grade com 9 Processadores

Como podemos perceber, analisando as tabelas de *speedup*, para quase todos os resultados obtivemos uma melhora de desempenho. Apenas não obtivemos resultados superiores para o Grafo PHD (exceto na Grade computacional para 3 processadores) e para o Grafo SH2 no *Cluster* com 9 processadores.

Grafo	Cheetham*	Cheetham-Downey	Cheetham-Niedermeier	*Speedup***
Somatostatin	25,01	4,15	14,14	6,03
WW	16,08	28,60	1,25	12,86
Kinase	427,99	25,62	1190,18	16,71
SH2	403,14	40,27	145,69	10.01
PHD	44,14	889	460,88	0,10

Tabela 5.4: Speedup - *Cluster* com 3 Processadores

Grafo	Cheetham*	Cheetham-Downey	Cheetham-Niedermeier	*Speedup***
Somatostatin	47,48	6,99	21,62	6,79
WW	1,81	27,84	1,75	1,03
Kinase	8,73	24,52	1215,56	0,36
SH2	25,34	23,27	41,37	1,09
PHD	7,59	862,64	456,74	0,02

Tabela 5.5: Speedup - *Cluster* com 9 Processadores

5.5 Considerações Finais

Nesta seção, apresentamos os resultados dos experimentos realizados com nossas implementações Cheetham-Downey e Cheetham-Niedermeier. Antes disso, apresentamos o ambiente computacional e os grafos de entrada. Como pudemos observar, conseguimos obter resultados promissores para quase todas as situações em que as implementações foram submetidas. Também observamos que o comportamento de nossas implementações manteve-se bastante estável em ambos modelos Grade e *Cluster*. A implementação Cheetham* apresentou maior variação de comportamento cujo motivo acreditamos que seja em virtude de escolhas aleatórias que o algoritmo executa.

Capítulo 6

Conclusão

Muitos problemas de grande importância no mundo real são NP-Completos, diante da inexistência de algoritmos eficientes para resolvê-los, uma grande variedade de métodos tem sido propostos. Com a vantagem de oferecer garantia quanto à exatidão e ao desempenho, a Complexidade Parametrizada vem se destacando como uma opção para resolver instâncias destes problemas, cujo tamanho encontra-se em um pequeno, mas útil, intervalo. Com o auxílio de outras técnicas, como a programação paralela, é possível aumentar este intervalo, resolvendo instâncias que eram, antes, impraticáveis seqüencialmente. Infelizmente, nem todos problemas são tratáveis por parâmetro fixo (FPT). Um problema FPT tem a entrada divida em duas partes: a parte principal e o parâmetro. Muitos problemas são naturalmente formulados nesta forma, como é o caso do problema da k-Cobertura por Vértices.

O problema da k-Cobertura por Vértices é um problema importante na Ciência da Computação. Ele possui aplicação prática em vários campos da computação como, por exemplo, a Biologia Computacional, em que a cobertura por vértices é utilizada para resolver conflitos entre seqüencias. Este problema foi um dos primeiros que se provou FPT.

Nosso objetivo foi o desenvolvimento de algoritmos paralelos, no modelo BSP/CGM, para problemas FPT. O modelo BSP/CGM é um modelo realístico de computação paralela em que são considerados, além da complexidade de tempo de processamento, o custo de comunicação entre os processadores. Implementações dos algoritmos projetados nesse modelo, têm obtido tempos bastante próximos aos previstos no modelo. Algoritmos FPT têm sido implementados e constituem uma abordagem promissora na solução de problemas NP-completos que necessitam de soluções exatas e para os quais podemos fixar, na prática, o parâmetro responsável pela explosão combinatorial. A combinação do paralelismo e de algoritmos FPT tem se mostrado bastante promissora na obtenção de soluções para problemas práticos.

Algoritmos FPT, muitas vezes, não são fáceis de ser entendidos e, com certeza, implementados. Além disso, nem todos algoritmos que são propostos, são viáveis do ponto de vista da implementação.

Vários algoritmos foram propostos para o problema FPT da k-Cobertura por Vértices. Neste trabalho, apresentamos os Algoritmos de Buss [3], de Balasubramanian *et al.* [1] que denominamos de B1 e B2, Cheetham *et al.* [6], Downey *et al*[14], Niedermeier e Rossmanith [32].

Como Cheetham *et al.* [6], empregamos a abordagem de utilizar algoritmos existentes, combinando-os entre si, a fim de se obter um novo algoritmo, mais robusto e eficiente. O desafio era identificar quais algoritmos poderiam colaborar entre si, para uma solução conjunta que fosse mais eficiente do que os algoritmos isoladamente. Era preciso extrair o melhor de cada um deles e colocá-los para executar em um ponto preciso do problema (o ponto em quem eles conseguem resolver melhor). De fato, nem todos algoritmos, por mais que resolvam o mesmo problema, são viáveis de serem unidos para colaborar entre si, ou por problemas de integração (visto que cada um segue um estilo, o estilo de quem o criou), ou por problemas de eficiência (qualquer adaptação que tenha de ser feita não pode impactar no tempo total do algoritmo).

Como nossa proposta era prover uma solução robusta, capaz de suportar instâncias maiores do problema, além dos desafios já mencionados, ainda tínhamos de realizar as adaptações necessárias para a paralelização no modelo BSP/CGM. Cabe lembrar, que nem todos algoritmos que são eficientes seqüencialmente, podem ser re-escritos para uma forma paralela eficiente. Estas restrições também foram observadas em nossa pesquisa.

Depois de pesquisar várias propostas na literatura, definimos quais algoritmos iríamos utilizar. Assim como Cheetham, utilizamos, nas primeiras etapas, os algoritmos de Buss [3] e B1 [1], porém ao invés de utilizar o algoritmo B2 [1] na última etapa, optamos por criar duas versões: uma utilizando o algoritmo proposto por Downey *et al*[14] na última etapa, e outra utilizando o algoritmo proposto por Niedermeier e Rossmanith [32] na última etapa.

Chamamos nossa implementação sobre o algoritmo que utiliza Buss, B1 e Downey de Cheetham-Downey e a implementação que utiliza Buss, B1 e Niedermeier de Cheetham-Niedermeier. Nossas implementações FPT paralelas Cheetham-Downey e Cheetham-Niedermeier foram construídas utilizando o modelo BSP/CGM.

Em nossos experimentos, utilizamos dois modelos de rede: *Clusters* e Grades Computacionais. Conseguimos resultados promissores, em ambos modelos de rede, com uma pequena vantagem utilizando *Clusters*. Destacamos que o uso adequado de estruturas de dados, aliado a coerentes estratégias de manipulação, tenha contribuído para o bom desempenho dos algoritmos de nossas implementações.

Anteriormente, Hanashiro [25] havia apresentado uma implementação refinada e melhorada em relação a implementação apresentada por Cheetham [6] (ambos utilizando os algoritmos: Buss, B1, B2). Hanashiro, em seus experimentos, apresentou resultados superiores aos apresentados por Cheetham, mesmo utilizando um ambiente computacional inferior.

Para comprovar o desempenho de nossas implementações, comparamos nossos resultados com os resultados da implementação de Hanashiro. Utilizamos o mesmo ambiente computacional, e obtivemos resultados superiores para grande parte das entradas subme-

tidas. O *speedup* calculado variou de 1,03 à 16,71 no melhor caso.

A contribuição deste trabalho, é apresentar duas propostas de algoritmos paralelos, no modelo BSP/CGM, para o problema FPT da k-Cobertura por Vértices e suas respectivas implementações.

Parte deste trabalho resultou no artigo: *An Alternative Implementation for the FPT k-Vertex Cover Parallel Algorithm* [24].

Trabalhos futuros incluem: (i) experimentos com utilização de escolhas aleatórias, para os algoritmos Cheetham-Downey e Cheetham-Niedermeier (apenas em momentos, cuja escolha do próximo vértice não segue qualquer regra de precedência); (ii) a substituição dos Algoritmos Buss e B1 por algoritmos mais eficientes, (iii) pesquisas e análises da viabilidade de paralelização dos algoritmos de Downey *et al.* [14] e Niedermeier *et al.* [32]; (iv) pesquisar quais e como as características dos grafos de entrada afetam o desempenho dos algoritmos Cheetham-Downey e Cheetham-Niedermeier, e com base nesta pesquisa, refinar ambos algoritmos; e (v) a análise de utilização das estratégias, estruturas de dados e lições aprendidas neste trabalho em outros problemas FPT como Conjunto Dominante, 3-Hitting Set, entre outros.

Referências Bibliográficas

[1] R. Balasubramanian, Michael R. Fellows, e Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Inf. Process. Lett.*, 65(3):163–168, 1998. ISSN 0020-0190.

[2] Vincent Berry e Francois Nicolas. Improved parameterized complexity of the maximum agreement subtree and maximum compatible tree problems. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(3):289–302, 2006. ISSN 1545-5963.

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

[4] Edson Norberto Cáceres, Henrique Mongelli, e Siang Wun Song. Algoritmos paralelos usando CGM/PVM/MPI: uma introdução. In *XXI Congresso da Sociedade Brasileira de Computação, Jornada de Atualização de Informática*, páginas 219–278. 2001.

[5] Marco Cesati e Miriam Di Ianni. Parameterized parallel complexity. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, páginas 892–896. Springer-Verlag, London, UK, 1998. ISBN 3-540-64952-2.

[6] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, e Peter J. Taillon. A parallel fpt application for clusters. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, página 70. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1919-9.

[7] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, e Peter J. Taillon. Solving large fpt problems on coarse-grained parallel machines. *J. Comput. Syst. Sci.*, 67(4):691–706, 2003. ISSN 0022-0000.

[8] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, páginas 151–158. ACM Press, New York, NY, USA, 1971.

[9] 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.

[10] Downey e McCartin. Some new directions and questions in parameterized complexity. In *International Conference on Developments in Language Theory (DLT), LNCS*, volume 8. 2004.

[11] R. G. Downey e M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[12] Rod G. Downey e Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theor. Comput. Sci.*, 141(1-2):109–131, 1995. ISSN 0304-3975.

[13] Rod G. Downey, Michael R. Fellows, Alexander Vardy, e Geoff Whittle. Parameterized complexity: A framework for. systematically confronting computational. intractability. In *AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, páginas 49–99. Proceedings of the DIMACS-DIMATIA Workshop, 1999.

[14] Rod G. Downey, Michael R. Fellows, Alexander Vardy, e Geoff Whittle. The parametrized complexity of some fundamental problems in coding theory. *SIAM J. Comput.*, 29(2):545–570, 1999. ISSN 0097-5397.

[15] Rodney G. Downey e Michael R. Fellows. Parameterized complexity after (almost) 10 years: Review and open questions, 1999.

[16] M. Fellows. Parameterized complexity: the main ideas and connections to practical computing. *Electronic Notes in Theoretical Computer Science*, 61, 2002.

[17] Stephen Gilmour e Mark Dras. A two-pronged attack on the dragon of intractability. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, páginas 183–192. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2005. ISBN 1-920-68220-1.

[18] Stephen Gilmour e Mark Dras. Kernelization as heuristic structure for the vertex cover problem. In *ANTS '06: Fifth International Workshop on Ant Colony Optimization and Swarm Intelligence*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006.

[19] Andrei Goldchleger, Fabio Kon, Alfredo Goldman vel Lejbman, e Marcelo Finger. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. In *Proceedings of the ACM/IFIP/USENIX Middleware'2003 1st International Workshop on Middleware for Grid Computing*, páginas 232–234. Rio de Janeiro, junho 2003.

[20] Georg Gottlob, Francesco Scarcello, e Martha Sideri. Fixed-parameter complexity in ai and nonmonotonic reasoning. *Artif. Intell.*, 138(1-2):55–86, 2002. ISSN 0004-3702.

[21] Jens Gramm e Rolf Niedermeier. A fixed-parameter algorithm for minimum quartet inconsistency. *J. Comput. Syst. Sci.*, 67(4):723–741, 2003. ISSN 0022-0000.

[22] Martin Grohe. The parameterized complexity of database queries. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, páginas 82–92. ACM Press, New York, NY, USA, 2001. ISBN 1-58113-361-8.

[23] Martin Grohe. Parameterized complexity for the database theorist. *SIGMOD Rec.*, 31(4):86–96, 2002. ISSN 0163-5808.

[24] E. N. Cáceres S. W. Song H. Mongelli, D. S. Aguena. An alternative implementation for the fpt k-vertex cover parallel algorithm. In *The 10th International Conference on High Performance Computing, Grid and e-Science in Asia Pacific Region (HPC Asia 2009)*, páginas 148–155. Kaohsiung, Taiwan, March 2009.

[25] Erik Joey Hanashiro. *O problema da k-cobertura por vértices: uma implementação FPT no modelo BSP/CGM*. mestrado, UFMS, Campo Grande, MS, Março 2004.

[26] J. Hartmanis e J. E. Hopcroft. An overview of the theory of computational complexity. *J. ACM*, 18(3):444–475, 1971. ISSN 0004-5411.

[27] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller e J. W. Thatcher, editores, *Complexity of Computer Computations*, páginas 85–103. Plenum Press, 1972.

[28] O. Kullmann. Deciding propositional tautologies: Algorithms and their complexity, agosto 09 1997.

[29] Zbigniew Lonc e Miroslaw Truszczyski. Fixed-parameter complexity of semantics for logic programs. *ACM Trans. Comput. Logic*, 4(1):91–119, 2003. ISSN 1529-3785.

[30] Catherine Mccartin. Contributions to parameterized, fevereiro 20 2004.

[31] Niedermeier. Ubiquitous parameterization – invitation to fixed-parameter algorithms. In *MFCS: Symposium on Mathematical Foundations of Computer Science*. 2004.

[32] Niedermeier e Rossmanith. Upper bounds for vertex cover further improved. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*. 1999.

[33] Rolf Niedermeier. Some prospects for efficient fixed parameter algorithms. In *SOFSEM '98: Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics*, páginas 168–185. Springer-Verlag, London, UK, 1998. ISBN 3-540-65260-4.

[34] Frances Rosamond. The newsletter of the parameterized complexity community. volume 3, página 3. 2008.

[35] Yinglei Song, Chunmei Liu, Xiuzhen Huang, Russell L. Malmberg, Ying Xu, e Liming Cai. Efficient parameterized algorithms for biopolymer structure-sequence alignment. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):423–432, 2006. ISSN 1545-5963.

[36] Leslie G. Valiant. A bridging model for parallel computation. *CACM: Communications of the ACM*, 33, 1990.