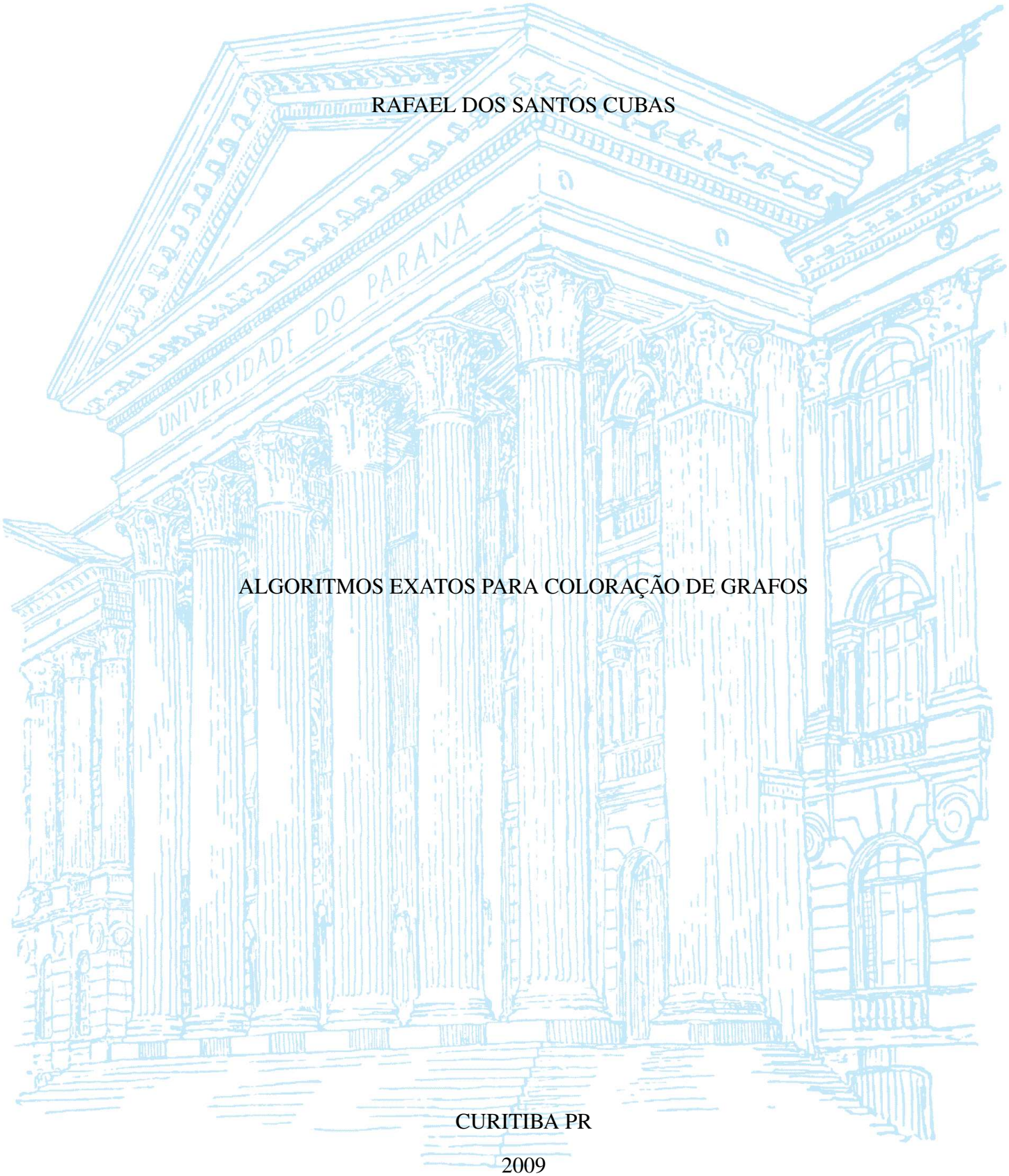
UNIVERSIDADE FEDERAL DO PARANÁ

RAFAEL DOS SANTOS CUBAS



ALGORITMOS EXATOS PARA COLORAÇÃO DE GRAFOS

CURITIBA PR

2009

RAFAEL DOS SANTOS CUBAS

ALGORITMOS EXATOS PARA COLORAÇÃO DE GRAFOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. André Guedes.

CURITIBA PR

2009

Catalogação na publicação
Selma Regina Ramalho Conte - CRB-9/888
Biblioteca de Ciência e Tecnologia - UFPR

Cubas, Rafael dos Santos
Algoritmos exatos para coloração de grafos / Rafael dos Santos Cubas. – Curitiba, 2009.
67 f. : il., grafs.; tabs

Dissertação (mestrado) – Universidade Federal do Paraná. Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: André Guedes

1. Grafos – Algoritmos. I. Guedes, André. II. Título.

CDD 511.56

Universidade Federal do Paraná
Setor de Ciências Exatas
Programa de Pós-graduação em Informática

UFPR

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Rafael dos Santos Cubas, avaliamos o trabalho intitulado, "ALGORITMOS EXATOS PARA COLORAÇÃO DE GRAFOS", cuja defesa foi realizada no dia 17 de julho de 2009, às 14:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 17 de julho de 2009.

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR – Orientador

Prof. Dr. Luérbio Faria
UERJ- Membro Externo

Prof. Dr. Fabiano Silva
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Coordenação de Pós-Graduação em Informática - Setor de Ciências Exatas

Ao futuro.

AGRADECIMENTOS

Agradeço em primeiro lugar ao meu Deus que viabilizou todas as coisas boas em minha vida. Agradeço a minha Tali e aos meus ex-patrões Giovanni Sugamosto e Luiz Ferraz por me apoiarem nos meus estudos.

RESUMO

Neste documento apresentamos três recentes trabalhos no campo de estudos da coloração de grafos. Os algoritmos dos planos de corte, do Branch and Cut e o algoritmo de Lucet. Todos são algoritmos exatos para a coloração e apresentaram excelentes resultados nos testes práticos realizados. No nosso estudo iremos mais afundo nas características de cada trabalho, principalmente no algoritmo de Lucet. Através de uma criteriosa comparação entre os três algoritmos será traçado um panorama geral entre os prós e contras de cada algoritmo, com destaque para o algoritmo de Lucet. Além desta comparação, o trabalho também reporta os resultados dos experimentos práticos realizados com alguns algoritmos de coloração.

Palavras-chave: Coloração de Grafos. Lucet. Graph. Teoria de Grafos

ABSTRACT

In this document, it will be discussed three recent researches on the graph coloring problem: the Cutting Planes, Branch and Cut and Lucet algorithms. All of them provided exact answers and delivered excellent results in the actual tests. In our research, we have gone deeper into the characteristics of each algorithm, in particular Lucet´s algorithm. As a result of a thorough comparison between them, an overview of their pros and cons will be presented, especially of the Lucet´s algorithm. In addition to the aforementioned comparison, this research also reports the actual results of some graph coloring algorithms.

Keywords: Graph coloring. Lucet. Graph. Graph Theory.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
PCG	Problema da coloração de grafos
DSATUR algorithm	Degree of saturation algorithm (Algoritmo de grau de saturação)
DIMACS	Discrete mathematics and theoretical computer science
LDC	Linear decomposition coloring (coloração de decomposição linear)
LB	Lower bound (Limite inferior)
LI	Limite inferior
UB	Upper bound (Limite superior)
LS	Limite superior

SUMÁRIO

1 INTRODUÇÃO

1.1 TEORIA DOS GRAFOS

O estudo da teoria dos grafos começou dentro da matemática. Entretanto, o aspecto computacional dos grafos e seus algoritmos também tem sido objetos de pesquisas no campo da ciência da computação. Apesar da teoria dos grafos ter sido pesquisada por vários cientistas, existem ainda várias perguntas que permanecem em aberto ou parcialmente respondidas por vários anos.

Por exemplo, a conjectura de Berge e o teste de perfeição de grafos que permaneceu como um problema em aberto por muitos anos e foi solucionada apenas em 2002 [1]. Entenda como parcialmente respondido um problema que não tem um algoritmo exato eficiente para grandes instâncias ou então, que apenas tem soluções algorítmicas que basicamente determinam limites superiores ou inferiores, principalmente através de heurísticas [2, 3, 4, 5].

Naturalmente existem problemas, como por exemplo a coloração de grafos, que devido as suas características não tem uma solução eficiente (polinomial) para quaisquer instâncias [5]. Tal solução, se existir, e for encontrada com certeza será um notável avanço no estudo da teoria dos grafos.

1.2 DEFINIÇÕES

Neste trabalho utilizaremos a seguinte terminologia para representar os grafos e suas características, esta terminologia de um modo geral é a mesma utilizada na maior parte da literatura disponível. Caso o(a) leitor(a) já esteja familiarizado com o tópico, ele(a) é fortemente encorajado(a) a pular para a próxima seção. Seja G = (V,E) um grafo não orientado, onde V(G) representa o conjunto de vértices deste grafo e E(G) o seu conjunto de arestas.

Uma aresta é definida como e={u,v} sendo u,v $\in$ V(G) e u $\neq$ v. Dois vértices u e v são adjacentes quando existe uma aresta {u,v} $\in$ E(G). O grau de um vértice é conhecido através do número de vértices adjacentes a ele e, é representado por *d(v)*.

Para um grafo como um todo estão definidos: $\bar{d}$(G) como o grau médio, δ(G) = *min* { *d(v)* | v $\in$ V(G) } como o grau mínimo e, Δ(G) = *max*{ *d(v)* | v $\in$ V(G) } como sendo o grau máximo. O conjunto de vértices adjacentes ao vértice u $\in$ V(G), é chamado de a vizinhança de u e é denotado por N(u) = { v $\in$ V(G) | $\exists$ {u,v} $\in$ E(G)}. Um grafo é chamado de completo quando existe uma aresta ligando um vértice a qualquer outro vértice do grafo.

Um grafo H = (V', E') é considerado um subgrafo de um grafo G = (V,E) se V'(H) $\subseteq$ V(G) e E'(H) $\subseteq$ E(G). Um subgrafo é chamado de subgrafo próprio quando H $\neq$ G. Se o subgrafo H = (V', E') de G contém todas as arestas que unem dois vértices em V'(G) $\subseteq$ V(G) então é dito que o grafo G é induzido por V'(G). Um subgrafo completo de G=(V,E) é chamado de clique. Uma clique máxima é uma clique que tem o maior número de vértices dentre todas as cliques de G=(V,E). Naturalmente, a clique máxima pode não ser única em um grafo.

É chamado de caminho uma sequência v_0, v_1 ... v_k $\in$ V(G) de vértices distintos ligados um a um por arestas. Um grafo G é conexo se existe um caminho entre cada par de vértices de V(G). É chamado de ciclo uma seqüência v_0, v_1 ... v_k $\in$ V(G), de três ou mais vértices ligados por arestas. Em um ciclo, o primeiro e do último vértice são iguais e todos os demais são distintos. Se o grafo contiver nada além de um ciclo de n vértices ele é chamado de C_n ou de grafo ciclo.

Seja um ciclo C com n ≥ 4 vértices em G. Se o subgrafo de G induzido pelos vértices de C for um C_n, então C é um buraco de tamanho n em G.

Um subgrafo maximal conexo é chamado de componente conexa. Para todo k ≥ 2, um grafo conexo G é k-conexo se ele tem pelo menos dois vértices e nenhum conjunto de no máximo k-1 arestas que possam separá-lo. Uma aresta é chamada de ponte quando a sua remoção automaticamente aumenta o número de componentes conexas de um grafo. Um subgrafo B de G é chamado de bloco de G se ele é uma ponte (junto com os vértices incidentes na ponte) ou se ele é um subgrafo maximal 2-conexo.

Seja v ∈ V(G), a função cor(v) retorna a cor utilizada para colorir o vértice v. Uma coloração de um grafo é um conjunto de cores utilizadas para colorir todos os vértices de um grafo. O problema da coloração de grafos (PCG) [6] tem como o objetivo de colorir, com o menor número de cores possíveis, os vértices de um grafo G tal que ∀ u,v ∈ V(G), v ≠ u e {u,v} ∈ E(G) então cor(u) ≠ cor(v). Ou seja, dados dois vértices quaisquer, eles não poderão ter a mesma cor se existir uma aresta entre eles.

Entretanto, são possíveis colorações nas quais dois vértices não adjacentes tem cores diferentes. Uma coloração que vértices adjacentes tem cores diferentes é conhecida como uma coloração **própria** [6]. O menor número de cores utilizadas para colorir um grafo é conhecido como **número cromático** e, é representado por χ(G). **Limite inferior** de χ(G) é um valor menor ou igual a χ(G) que é utilizado como uma estimativa do número cromático. Por conseqüência o **limite superior** é maior ou igual a χ(G) e tem o mesmo propósito.

Um algoritmo para o problema da coloração de grafos é dito **exato**, quando o algoritmo colore o grafo utilizando um número de cores igual a χ(G). Naturalmente, para um mesmo grafo pode existir mais de uma coloração ótima, entretanto o número de cores de cada uma delas é igual a χ(G).

Um grafo é k-colorível se admite uma coloração com no máximo k cores. Um grafo é k-cromático se ele é k-colorível mas não é (k-1)-colorível. Um grafo é crítico se χ(H) < χ(G) para todo subgrafo próprio H ⊂ G. Um grafo crítico é chamado de k-crítico quando este tem χ(G) = k. A densidade de um grafo G de ordem n é definida como sendo $\frac{|E(G)|}{\binom{n}{2}}$.

1.3 OBJETIVOS

O Problema da coloração de grafos(PCG) é um objeto de estudo valioso pois tem aplicação real em vários problemas práticos como por exemplo, atribuição de rádio-freqüências [7], Alocação de registros [3], o jogo sudoku [8], armazenamento de produtos químicos [6], alocação de máquinas ou pessoal para trabalho [9], alocação de salas de aula e estudantes [10, 11]. O PCG é conhecido por ser um problema NP-Dificil [12, 3] para um grafo qualquer.

Um dos fatores que influenciam negativamente na performance dos algoritmos de coloração é o fato de que existem muitas soluções que são simétricas, ou seja, são colorações que tem o mesmo número de cores e estas cores se alternam, hora colorindo um conjunto de vértices hora colorindo outro. Entretanto, para algumas famílias de grafos, como por exemplo os grafos perfeitos, o problema torna-se solúvel em tempo polinomial [13].

O objetivo desta pesquisa foi fazer um estudo aprofundado dos recentes artigos publicados por notáveis pesquisadores do assunto, como Mendez Diaz e Zabala [2, 13] e Lucet, Mendes e Moukrim [14], que desenvolveram bons trabalhos no campo dos algoritmos exatos para coloração de grafos. Com base nesses estudos, o texto faz uma profunda comparação entre os algoritmos e explica o algoritmo de Lucet com um nível maior de detalhes. A comparação dos algoritmos é feita identificando os pontos a serem comparados e utilizando tabelas e gráficos para tirar

conclusões a respeito dos algoritmos em cada quesito comparado e, finalmente montando uma conclusão global sobre os três principais algoritmos estudados neste trabalho.

O texto também contribui com conclusões a respeito dos resultados das implementações de alguns dos algoritmos apresentados neste trabalho.

O trabalho a seguir está divido da seguinte maneira: capítulo 2 apresentação dos trabalhos [2, 13, 14]. Capítulo 3, comparação entre os algoritmos apresentados, capítulo 4 discute as características de alguns dos algoritmos citados no texto sobre a perspectiva das suas implementações. E finalmente a conclusão no capítulo 5 e referências bibliográficas.

Para facilitar a compreensão do conteúdo aqui apresentado, é sugerido ao leitor(a) que ao imprimir este documento, que o faça em uma impressora **colorida**. Desta maneira a compreensão dos detalhes das figuras coloridas não será prejudicada.

2 REVISÃO BIBLIOGRÁFICA

2.1 CONTEÚDO

Esta seção está organizada da seguinte maneira. Primeiro serão apresentados os algoritmos Branch and Cut[2] e Planos de Corte[13], que além de terem os mesmos autores, são complementares e, em seguida apresentarei o algoritmo proposto por Lucet, Moukrim e Mendes em [14]. É importante frizar que o autor, de maneira alguma, reclama a autoria do conteúdo e das idéias que seguem dentro do capítulo 2. Eventualmente algumas opiniões pessoais ou comentários complementares podem estar inseridos no texto de maneira que não comprometam o conteúdo apresentado.

2.1.1 Formulações da Programação Inteira

Esta seção irá abordar superficialmente alguns tópicos relevantes de programação inteira que são usados neste documento. As inequações bem como as suas aplicações foram retiradas dos artigos do Branch and Cut[2] e Planos de Corte[13].

2.1.1.1 Propriedades básicas

O problema da coloração de grafos é modelado na programação inteira (PI) utilizando variáveis binárias, com valor 0 ou 1. Considerando i ∈ V(G) e 1 ≤ j ≤ | V(G) | = n, se $x_{ij} = 1$ então a cor j colore o vértice i, caso contrário $x_{ij} = 0$. Se variável binária $w_j = 1$, isto indica que a cor j está sendo utilizada na coloração. Abaixo está formulada a representação clássica do PCG(Problema da coloração de grafos):

$$\min \sum_{j=1}^{n} w_j \tag{2.1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \; \forall i \in V \tag{2.2}$$

$$\begin{aligned} x_{ij} + x_{kj} \leq w_j \; \forall \{i,k\} \in \mathrm{E}, 1 \leq j \leq n \\ x_{ij} \in \{0,1\} \; \forall \mathrm{i} \in \mathrm{V}, 1 \leq j \leq n \; w_j \in \{0,1\} \; 1 \leq j \leq n \end{aligned} \tag{2.3}$$

A equação (2.1) define o objetivo do PCG, ou seja, minimizar o número de cores utilizadas em uma coloração. A inequação (2.2) define que cada vértice recebe somente uma cor, a inequação (2.3) define que cada par de vértices adjacentes tem cores diferentes e que $w_j = 1$ somente quando algum vértice tem a cor j.

Entretanto, o conjunto de equações apresentadas ainda não tratam do problema da indistinção das variáveis e por conseqüência da simetria das soluções. Para resolver este problema,

Mendez Diaz e Zabala[13, 2] proporam três modelos. Contudo, será apresentado aqui somente aquele se mostrou como computacionalmente mais viável.

$$w_j \leq \sum_{i \in V} x_{ij} \; \forall 1 \leq j \leq n \tag{2.4}$$

$$w_j \geq w_{j+1} \; \forall \; 1 \leq j \leq n-1 \tag{2.5}$$

Para eliminar alguma simetria nas soluções os autores impuseram que a cor j pode ser usada para colorir um vértice somente se a cor j - 1 já colore algum outro vértice (inequações 2.4 e 2.5). Dessa maneira toda k-coloração simétrica usando cores com valor maior do que k são desconsideradas. É um avanço modesto, entretanto tem a sua importância principalmente tratando-se de um problema NP-Difícil.

2.1.1.2 Inequações

As inequações definidas pelos autores tem um papel importante para o algoritmo proposto. Elas entram em cena sempre que uma variável inteira assume um valor fracionário. Seu objetivo é "cortar"o politopo da coloração, gerando uma nova solução que pode tornar a variável inteira novamente. Abaixo temos as inequações mais importantes referentes a este trabalho.

$$\sum_{v \in K} x_{vj_0} - w_{j_0} \leq 0 \tag{2.6}$$

Na inequação 2.6, **inequação da clique**, temos K como sendo a clique maximal. Sendo v ∈ K e K uma clique, a inequação define que não é possível que dois ou mais vértices de K compartilhem uma mesma cor j_0. Esta equação é uma consequência da inequação (2.3).

A **inequação da vizinhança** (2.7), usada mais adiante, é obtida da seguinte maneira: seja v ∈ V(G), *d(v)* = | N(v) | e a combinação de $x_{vj} + x_{kj} \leq w_j$ (inequação 2.3) para todo k ∈ N(v).

$$\mathrm{d}(v)x_{vj} + \sum_{k \in N(v)} x_{kj} \leq \mathrm{d}(v)w_j \tag{2.7}$$

Contudo a inequação da vizinhança ainda pode ser objeto de melhoria, se um valor r for tomado como o tamanho do maior conjunto independente em N(v), não mais do que r vértices poderão ser coloridos com a mesma cor, portanto a inequação (2.7) pode ser fortalecida tornando a desigualdade mais forte que a anterior. Desta maneira os autores encontraram a inequação (2.8) que é de grande valia durante o relaxamento do politopo da coloração.

$$rx_{vj} + \sum_{k \in N(v)} x_{kj} \leq rw_j \tag{2.8}$$

Seja $C_k = \{v_1, ..., v_k\}$ um buraco de tamanho k, k > 3. A **inequação de buraco** é (2.9) :

$$\sum_{i=1}^{k} x_{v_i j_0} + \sum_{j=n-\lfloor k/2 \rfloor+1}^{n} \sum_{v \in V} x_{vj} \leq \lfloor k/2 \rfloor w_{j0} + w_{n-\lfloor k/2 \rfloor+1} \tag{2.9}$$

Com $j_0 \leq n - \lfloor k/2 \rfloor$ 2.9 é uma inequação definidora de face se:

- ∀ v ∈ V \ C_k, existe um conjunto independente do tamanho ⌊ k / 2 ⌋ + 1 em G [C_k ∪ {v}]

- Uma χ(G)-coloração existe sobre a face

Seja $i_0 \in V(G)$ e $1 \leq j_0 \leq n$, a inequação **cor de bloco** (2.10) é uma inequação válida. Se $\chi(G) + 1 \leq j_0 \leq n - 2$, então a inequação torna-se uma definidora de face e, portanto pode ser usada para eliminar soluções fracionárias.

$$\sum_{j=j_0}^{n} x_{i_0 j} \leq w_{j_0} \tag{2.10}$$

Sejam $K_1, K_2, ..., K_r$ cliques tal que $K_i \cap K_j = 0$ se $j \neq i - 1, i + 1$ e $| K_i \cap K_{i+1} | \leq 1$. Considere as cores $c_1, ..., c_r, c_{j0}$ com $c_j \leq c_{j0} \leq n - 1 \; \forall \; j = 1, ..., r$. Combinando inequações clique e cor de bloco, os autores obtiveram em 2.11 a inequação **caminho multi cor**:

$$x_{v_1 c_1} + \sum_{i=2}^{k-1} (x_{v_i c_{i-1}} + x_{v_i c_i}) + x_{v_k c_{k-1}} + \sum_{j=c_{j0}}^{n} \sum_{i=1}^{k} x_{v_i j} \leq \sum_{i=1}^{k-1} w_{c_i} + w_{c_{j0}} \tag{2.11}$$

A inequação 2.11 é definidora de face se as seguintes condições foram satisfeitas:

- $V \setminus \{v_1,...,v_k\}$ não é uma clique
- Se $v \in K_i$, existe uma $(c_{j0} - 1)$-coloração tal que c_i não é a cor que colore v
- Uma χ(G)-coloração existe sobre a face

A última e não menos importante inequação que será apresentada é a inequação **clique multi cor** (2.13). Seja $K = \{ v_1, ..., v_p \}$ uma clique de tamanho p, k tal que $p \leq k \leq n - 1$ e $Col = \{j_1,....,j_{p-1}\} \subset \{1,...,k - 1\}$. Considerando a seguinte inequação:

$$\sum_{i=1}^{p} x_{v_i j} \leq w_j \; \forall j \in Col \tag{2.12}$$

Somando-as e considerando que qualquer coloração precisa de p cores para colorir K, os autores obtiveram a inequação válida (2.13). Se $V \setminus K$ não é uma clique e $\chi(G) + 1 \leq k \leq n - 2$, a inequação (2.13) é uma definidora de face.

$$\sum_{i=1}^{p} \sum_{j=k}^{n} x_{v_i j} + \sum_{i=1}^{p} \sum_{j \in Col} x_{v_i j} \leq w_k + \sum_{j \in Col} w_j \tag{2.13}$$

2.2 ALGORITMO PLANOS DE CORTE PARA COLORAÇÃO DE GRAFOS

A abordagem proposta por Mendez Dias e Zabala [13] utiliza a programação inteira (PI) para reduzir a simetria das soluções e assim ter um modelo computacional mais tratável. Como muitos problemas de otimização em grafos, o problema da coloração de grafos pode ser formulado como um problema de programação linear.

Os algoritmos de planos de corte são excelentes ferramentas para lidar com problemas de lineares de programação inteira. Na figura (2.1) podemos ver um exemplo do funcionamento deste algoritmo. Os cortes no politopo eliminam soluções inexatas ou simétricas e o aproximam de uma solução exata.

A idéia é considerar o relaxamento linear e tentar fortalecê-lo adicionando inequações válidas que foram violadas. O algoritmo utiliza planos de cortes (cuts) gerais que não tiram

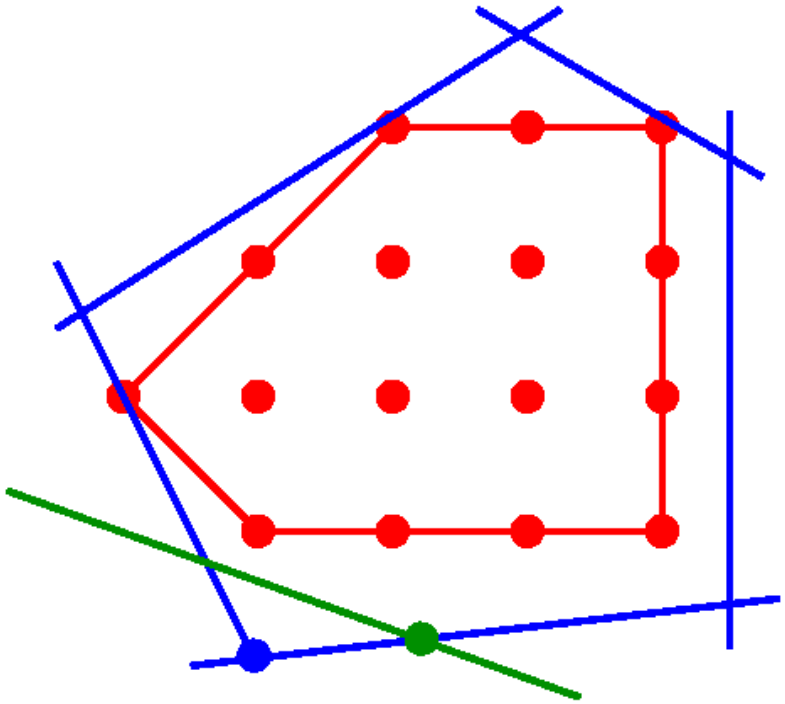


Figura 2.1: Exemplo do funcionamento do algoritmo Planos de Corte

vantagem da estrutura, ou então cortes especialmente desenvolvidos para explorar propriedades do problema. Como por exemplo, a simetria das soluções.

Este algoritmo fortalece o poliedro associado com os modelos propostos de programação inteira com fortes inequações válidas, as quais foram provadas pelos autores como sendo definidoras de face(facet-defining) em várias instâncias. O algoritmo foi testado em grafos aleatórios[15] e também em um conjunto de problemas de testes estudados na literatura.

2.2.1 Relaxamento LP

O primeiro passo no desenvolvimento do algoritmo planos de corte é a definição de um relaxamento-LP (LP-relaxation) inicial. Os autores acreditam que o relaxamento linear (linear relaxation) do problema tem muitas restrições de adjacência (inequação 2.3), o que torna a resolução muito lenta, principalmente para grafos de média e alta densidade. Três soluções foram consideradas para este problema: heurísticas sobre cliques, a remoção das restrições de adjacência e, a substituição das mesmas pela inequação da vizinhança (2.8).

Buscando um bom equilíbrio entre tempo de CPU, requisitos de memória e crescimento do limite superior para χ(G), os autores descobriram que o relaxamento inicial, que demonstrou o melhor comportamento, foi aquele resultante da substituição das restrições de adjacência por uma versão das inequações de vizinhança(2.8).

O coeficiente r desta equação é substituído pelo tamanho de uma clique de N(v) encontrada por uma heurística. Essa substituição relaxa o politopo mas também permite uma melhor abordagem para grafos grandes.

2.2.2 Limites Inferiores e Superiores

Os limites superiores assim como limites inferiores são muito importantes para manter o programa linear com um tamanho razoável. O **limite inferior** é obtido encontrando uma clique maximal C_1 com uma heurística. Todas as variáveis relacionadas com os vértices de

C_1 são ajustadas no modelo considerando que as primeiras | $V(C_1)$ | cores são usadas pelos vértices da clique. O **limite superior** é obtido através da solução encontrada por uma heurística DSATUR[5, 16]. DSATUR é um método exato para grafos bipartites[5]. Ambos os limites permitem que variáveis sejam eliminadas no modelo.

Ao curso da definição do algoritmo dos planos de corte, do Branch-and-Cut e do algoritmo exato de Lucet, fica nítido que limites tanto superiores, com inferiores pré-definidos são vitais para limitar a expansão da solução do problema e, por consequência diminuir o tempo gasto em sua busca.

2.2.3 Algoritmos de Separação

Dada uma solução fracionária do politopo, o algoritmo procura por um conjunto de restrições para cortar o politopo. Após adicionar estas inequações o relaxamento LP é resolvido. Com este resolvido tem início a fase central do algoritmo dos planos de corte, a fase de separação. Algoritmos de separação são responsáveis pela identificação das inequações violadas.

Algoritmos eficientes são cruciais para o sucesso da abordagem proposta pelos autores. Eis algumas inequações descritas naquele trabalho: Clique e clique multi-cor, Cor do bloco, Caminho multi cor e desigualdade de buraco. Na sequência serão descritas cada uma delas, bem como os procedimentos para o seu reconhecimento.

2.2.3.1 Clique e Clique Multi-Cor

O processo de separação das cliques em um grafo é conhecido por ser NP-Difícil[13, 17]. Na literatura são encontradas muitas estratégias para detectá-las[17]. Entretanto, apenas a escolhida pelos autores será mencionada aqui. Para cada cor j_0, a busca pelas inequações violadas de clique é feita considerando a lista de variáveis fracionárias $\{x^*_{ij0} : i \in V(G) \text{ e } x^*_{ij0} < 1 \}$ em ordem decrescente de valor de x^*_{ij0}, onde x* denota o valor da solução fracionária atual.

Se x_{ij0} é uma variável fracionária, nós inicializamos a clique com o vértice i. A clique é construída tentando-se adicionar outros vértices seguindo a ordem de x*. As tentativas foram feitas usando um parâmetro. Na tentativa k, a variável fracionária $x_{i'j0}$ tal que o vértice i' é o k-ésimo vértice adjacente a i na lista. Este vértice é adicionado à clique e então o resto da lista é analisada em ordem.

Para evitar qualquer esforço computational adicional, a clique encontrada é também usada para verificar se a equação **clique multi-cor** (2.13) foi violada. O método descrito acima também é usado no algoritmo Branch-and-Cut descrito mais adiante na seção 2.3.

2.2.3.2 Inequação de Cor de Bloco

Estas inequações são trabalhadas pela "força bruta". Para haverem chances de encontrar uma inequação cor de bloco (2.10) violada, é necessário que w_{j0} seja fracionária. Então, para todo j_0 tal que $0 < w_{j0} < 1$, todas as inequações são numeradas e encontram-se aquelas violadas para solução fracionária atual.

2.2.3.3 *Inequação Caminho Multi Cor*

Para cada variável fracionária w^*_k , o custo c_{uv} é associado a cada aresta (u,v) ∈ E(G). É calculado para cada vértice v ∈ V(G), através de uma heurística, o caminho mais custoso em G.

$$c_{uv} = \max_{j=1,\ldots,k-1}\{x^*_{uj} + x^*_{vj} - w^*_j\} + \sum_{j=k}^{n}(x^*_{vj} + x^*_{vj}) \tag{2.14}$$

Para cada v ∈ V(G), é inicializado o candidato a caminho P com v e são feitas t_v tentativas. Na tentativa j o caminho é extendido adicionando o vértice w caracterizado por ter o j-ésimo maior c_{vw} dentro todos os vértices adjacentes a v. Então, iterativamente, vértices são adicionados ao caminho escolhendo um vértice adjacente ao último vértice adicionado ao caminho, desde que aquele já não pertença a P e tenha o maior custo.

Os experimentos computacionais revelam que não é produtivo ter um vértice pertencendo a vários caminhos. Além disso, caminhos muito longos tem poucas chances de violar a inequação. Para evitar o excesso de caminhos, o algoritmo não considera um vértice, que pertence a |V(G)|/10 caminhos violados, nos futuros caminhos. Complementarmente, foi definido um parâmetro de entrada para limitar o tamanho dos caminhos, o parâmetro foi definido com valor 6. Todo caminho com custo total maior do que w^*_k viola a inequação do caminho multi cor.

2.2.3.4 *Inequação de Buraco*

Dado um grafo G' = (V', E'), a inequação de buraco é aplicada da seguinte maneira. Seja B = ($V_1 \cup V_2, E_B$) um grafo bipartite onde para cada v ∈ V' são incluídos dois vértices $v_1 \in V_1$ e $v_2 \in V_2$. Se (u, v) ∈ E', então (u_1, v_2) e $(v_1, u_2) \in E_B$. É fácil perceber que um caminho P_{v1} em B começando com v_1 e terminando com v_2, considerando-o como um conjunto de vértices, é um ciclo C_v em G'.

Para encontrar inequações de buraco (2.9) violadas, é considerado j_0 tal que $w^*_{j0} > 0$ e V' ⊂ V onde v ∈ V' se x^*_{vj0} é fracionário. É construído o grafo bipartite B e os autores consideraram o custo $c_{u1,v2} = c_{v1,u2} = \max(0, w^*_{j0} - x^*_{uj0} - x^*_{vj0})$ para cada aresta (u, v) de B. O custo de P_{v1} é:

$$\sum_{(u'z'')\in P_{v1}} \max(0, w^*_{j0} - x^*_{uj0} - x^*_{zj0}) \tag{2.15}$$

Por fim, se for encontrado o menor caminho entre v_1 e v_2, o ciclo C_v será um bom candidato para violar uma inequação de buraco. O caminho mais curto é calculado pelo algoritmo de Dijkstra [18].

2.2.4 Esquema de Gerenciamento de Corte

Um algoritmo de planos de corte constrói e resolve a sequência do relaxamento LP[13]. Na fase de separação, um conjunto de restrições violadas é adicionada a uma lista a cada iteração do algoritmo. É importante cuidar para que o programa linear não torne-se muito grande, o que tomaria ainda mais tempo para resolvê-lo. Este problema pode ser evitado removendo alguns cortes que tornaram-se irrelevantes para a solução em curso.

Estes cortes adicionados à lista, ou não foram incluídos no relaxamento ou foram eliminados porque aparentaram estar inativos. Esta lista de cortes é muito útil para o gerenciamento de memória e também pode ser considerado como um mecanismo auxiliar para executar a separação, uma vez que a lista de cortes pode ser checada rapidamente.

Uma característica muito comum deste trabalho [13] é o uso de heurísticas, inclusive na detecção das inequações violadas. Por esse motivo o algoritmo pode não ser capaz de detectar algumas inequações que foram detectadas em uma execução anterior. Quando o programa linear(LP) for re-otimizado, primeiro são checados todos os cortes na lista de cortes e, o programa linear é re-otimizado se mais de 200 cortes violados foram encontrados. Caso contrário as rotinas de separação são invocadas para encontrar novas inequações violadas.

2.2.5 Experimentos Computacionais

O objetivo de Mendez Diaz e Zabala neste artigo é avaliar o melhoramento do limite inferior de χ(G) obtido no relaxamento LP quando eles o fortaleceram adicionando inequações válidas que já tinham sido caracterizadas pelo politopo. Os autores introduziram uma série de inequações válidas e definiram as condições para que estas sejam definidoras de face. Embora o passo de separação seja bem sucedido, os cortes podem não ajudar no melhoramento do limite inferior.

2.2.5.1 Planos de corte

Um meio indireto de avaliar a qualidade dos planos de corte (cutting planes) é observar o aumento produzido no limite inferior quando é utilizado o relaxamento LP. Grandes aumentos no limite inferior significam melhores inequações, por que definem cortes mais profundos no politopo de relaxamento. Todavia, cortes densos no politopo de relaxamento aumentam o consumo de memória e podem diminuir a velocidade de solução. Além disso, se a fase de separação para uma classe de inequações é muito custosa em relação ao crescimento do limite inferior, pode não compensar incluí-las no algoritmo.

Aliás a função desta última sentença fica clara mais adiante, quando uma combinação menos restritiva de inequações tem um desempenho muito superior do que um conjunto mais restritivo. Mendez Diaz e Zabala conduziram vários experimentos para tentar encontrar uma boa combinação de classes de inequações. Na tabela 2.1 existe a lista das classes de inequações que pertencem a cada combinação.

Combinação	Classes de inequações
C1	Clique
C2	Clique + Cor do Bloco + Caminho multi cor
C3	Clique + Cor do Bloco + Caminho multi cor + Clique multi-cores
C4	Clique + Clique multi-cores + Buraco
C5	Clique + Cor do Bloco + Caminho multi cor + Clique multi-cores + Buraco

Tabela 2.1: Combinações de classes de inequações

Para comparar as combinações os autores utilizaram o algoritmo em grafos aleatórios [15] em 50 rodadas de testes. Os autores [13] utilizaram grafos com 125 vértices e densidades descritas na tabela (2.2).

Os autores observaram a evolução do limite inferior de χ(G) e o tempo de CPU para obte-lo. Os experimentos mostraram que qualquer combinação obtem o mesmo limite inferior ao final das 50 rodadas, com exceção das combinações que **excluem** as inequações com cliques. O desempenho dos planos de corte é principalmente devido a inclusão destas inequações.

As diferentes combinações de inequações foram testadas em oito instâncias de cada classe de densidades (baixa, média e alta). O melhoramento do limite inferior em relação a

Nome	Intervalo
Baixa	Menor do que 30%
Média	Entre 40% e 60%
Alta	Maior do que 70%

Tabela 2.2: Classes de densidades testadas no plano de cortes

heurística inicial, que utilizava a clique maximal como limite inferior inicial, é medido para cada densidade. Na figura (2.2) temos um gráfico detalhando a diferença média entre o valor do limite inferior antes e depois da execução do algoritmo. A diferença foi medida para cada uma das combinações de equações definidas na tabela (2.2) dentre as densidades disponíveis.

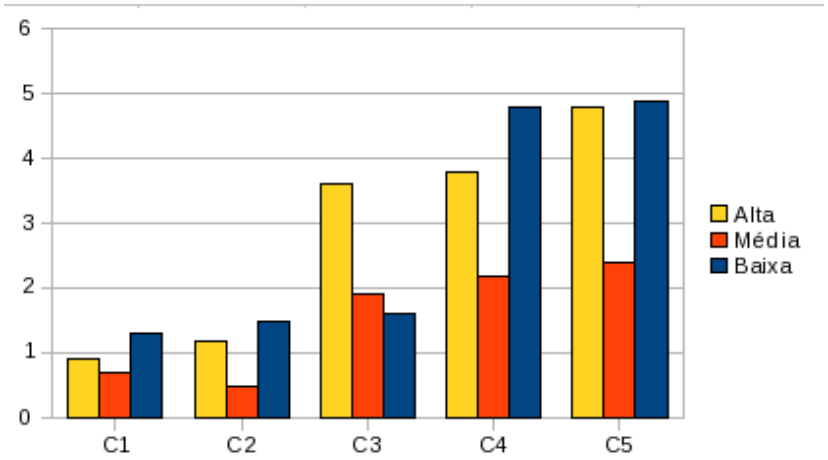


Figura 2.2: Diferença média do limite inferior inicial e final entre as combinações de inequações e as densidades.

A conclusão dos autores foi que não existe uma combinação perfeita entre as citadas acima considerando diferentes densidades. Entretanto, a combinação que se saiu melhor considerando todas densidades foi a C2 (Clique + Cor do bloco + Caminho multi cor).

2.2.5.2 Eficiência da Separação

Experimentos práticos provaram que os algoritmos de separação não tem uma participação significante no tempo total de processamento do algoritmo. Embora a sua participação no tempo de processamento aumente a medida que a densidade do grafo diminua. A eficiência dos algoritmos de separação é avaliada pela relação entre o número de cortes violados sobre o número de cortes analisados sobre cada rotina de separação.

O processo de separação da inequação do caminho multi cor(2.11) tem uma boa taxa de eficiência, **independente** da densidade do grafo. A separação das inequações cor de bloco (2.10) tem um aproveitamento é muito baixo. A sua inclusão somente é justificável pela sua contribuição no desempenho do algoritmo e ao seu baixo tempo de processamento. A inequação da clique multi cor (2.13) é a mais ineficiente, ao que tudo indica esta inequação não é frequentemente violada. A eficiência da separação da inequação de buraco (2.9) aumenta quando a densidade diminui, enquanto que a separação da clique mantém sua eficiência estável independente da densidade.

A presença das inequações de clique é essencial, junto com a cor de bloco e caminho multi cor combinados resultam em bons planos de corte. Clique multi cor e inequações do buraco

não apresentam nenhum recurso que justifique a sua inclusão exceto quando nenhum outro corte for encontrado.

2.2.6 Considerações Finais

Os resultados dos testes computacionais do artigo de Mendez Diaz e Zabala [13] foram obtidos sobre as instâncias DIMACS [19]. Os resultados dos testes apontaram para uma significante melhora no cálculo do limite inferior do número cromático, especialmente quando o número cromático χ(G) é muito maior que a clique máxima. Ainda houveram instâncias as quais o algoritmo foi capaz de resolver de maneira ótima, uma vez que a diferença inicial entre o limite inferior e superior foi fechada.

Os experimentos confirmam que é uma boa estratégia obter os limites inferiores se comparado com a abordagem clássica da clique máxima, com exceção das instâncias criadas especialmente para serem difíceis de colorir.

2.3 ALGORITMO BRANCH AND CUT PARA COLORAÇÃO DE GRAFOS

2.3.1 Introdução

Como a maioria dos problemas de otimização em grafos, o PCG (Problema da coloração de grafos) pode ser formulado como um problema de programação linear (LP) inteira. Os algoritmos Branch-and-Cut baseados em programação linear são uma das ferramentas mais importantes para tratar esses modelos computacionalmente [2]. O desempenho do algoritmo Branch-and-Cut depende da combinação de vários fatores. Pré-processamento, pesquisa e estratégias de ramificação, limites inferiores, limites superiores, relaxamento LP e os Planos de Corte são os principais componentes considerados em uma implementação [2]. Esses são métodos generalistas, entretanto aqueles que demonstram eficiência perante casos particulares tem provado serem os mais bem sucedidos.

O Branch-and-Cut, assim como todo algoritmo para o PCG, também enfrenta o problema da simetria das soluções. Para contornar este problema os autores propuseram uma abordagem de programação inteira que reduz a simetria, o número de soluções possíveis e deriva famílias de inequações que definem faces (facet-defining).

Também fazem parte do artigo [2] métodos para implementar a separação para algumas das inequações e introduzir estratégias para rejeitar simetria na fase de geração da árvore de busca.

2.3.2 Algoritmo Branch And Cut

A idéia do algoritmo proposto por Mendez Diaz e Zabala[2] é particionar recursivamente o grafo em subconjuntos e resolver o problema para cada um deles. Este procedimento gera uma árvore de enumeração e os filhos de um vértices correspondem a uma partição do conjunto associado com o vértices pai. Para cada vértice da árvore, um relaxamento linear é executado removendo os requerimentos de integridade e adicionando inequações válidas que cortam (o politopo) descartando a solução fracionária.

Para reduzir o número de vértices na árvore, é importante ter bons limites inferiores e superiores, boas regras para particionar o conjunto de vértices, boas estratégias para pesquisa na árvore e relaxamentos lineares fortes.

2.3.3 Pré-Processamento

O objetivo da fase de pré-processamento é preparar o grafo sobre o qual será calculado o número cromático para o algoritmo que efetivamente o faz. Neste trabalho [2] os autores sugerem que uma heurística seja usada para encontrar uma clique maximal, com tamanho *m*. Este número é usado como limite inferior do número cromático. Então, são eliminados os vértices que não tem um vértice adjacente a clique mas é adjacente a qualquer vértice da sua vizinhança (da clique).

Em seqüência os vértices que tem grau menor do que *m - 1* são deletados. Qualquer coloração ótima do novo grafo segue uma coloração ótima do grafo original. Além disso, é gerada uma coloração inicial válida [6] aplicando uma heurística de enumeração parcial baseada no DSATUR [16]. Esta solução calcula um **limite superior** do número cromático e permite a eliminação de variáveis do modelo.

Apesar da redução do tamanho do grafo sugerir um ganho de desempenho, o que de fato acontece para as instâncias DIMACS [19], ela não obteve resultados significativos com uma das famílias de grafos escolhida pelos autores para executar os testes computacionais, os grafos aleatórios [15]. A participação do pré-processamento no tempo total de execução do algoritmo é praticamente insignificante.

2.3.4 Regras De Ramificação

A regra clássica de ramificação em uma variável fracionária, que é atribuí-la o valor 1 em um subproblema e 0 em outro, é muito assimétrica. A árvore gerada é desbalanceada por que atribuir 1 a variável significa colorir o vértice, ao passo que atribuí-la 0 significa que a cor não é considerada para colorir o vértice. Esta abordagem mostrou-se infrutífera.

A regra de ramificação utilizada pelos autores é a seguinte: primeiro um vértice é escolhido, para cada cor possível para o vértice, dentre aquelas usadas para colorir o grafo, um novo subproblema é criado. Além disso, um subproblema é criado com o vértice recebendo a próxima cor àquela que foi escolhida. Na figura 2.3, temos um grafo que tem uma solução parcial na coloração de um dos vértices.

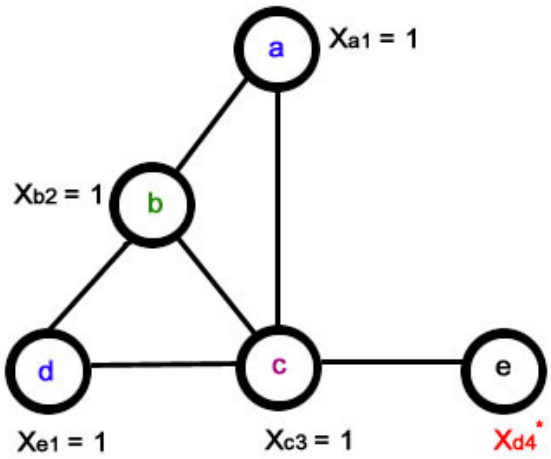
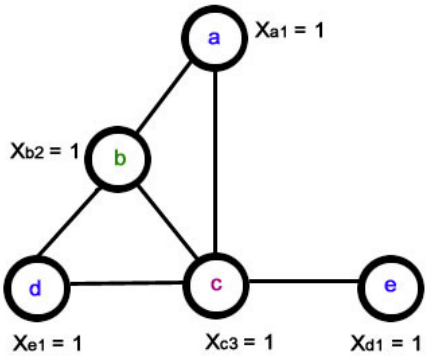


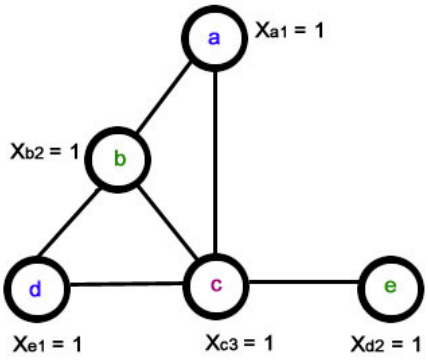
Figura 2.3: Ramificação da árvore do Branch and Cut - Solução fracionária

Para tornar inteira a variável fracionária, três novos subproblemas são criados de acordo com os critérios descritos anteriormente. Na figura 2.4 podemos ver os novos subproblemas. Para efeitos de simplificação foi montado um grafo pequeno e, os novos subproblemas na verdade

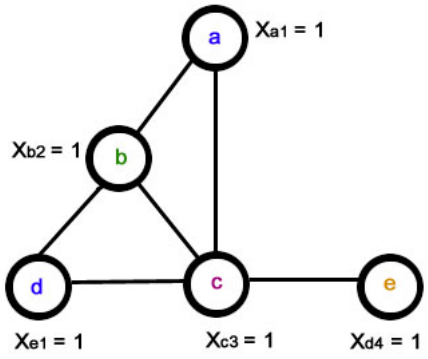
resolvem o problema da coloração. Portanto, podemos considerar o grafo apresentado na figura 2.3 e na figura 2.4 como apenas um subgrafo de um grafo bem maior.



(a)



(b)



(c)

Figura 2.4: Vértices criados na ramificação para resolver a solução fracionária original

Os autores escolheram um vértice, com valor fracionário, adjacente ao maior número de vértices com cores diferentes. Em caso de empate, entre um ou mais vértices, as regras utilizadas estão detalhadas na tabela 2.3.

Regra	Descrição
VB1	o vértice com o maior grau no subgrafo não colorido
VB2	o vértice que produz o maior decréscimo no número de cores disponíveis para os vértices não coloridos restantes

Tabela 2.3: Regras de ramificação desempate

As regras de ramificação na tabela 2.3 especificam como dividir o conjunto de soluções possíveis para o subproblema em questão. Para determinar a ordem que os subproblemas, ou subgrafos, são examinados foi utilizada uma busca em profundidade para decidir qual vértice da árvore seria avaliado primeiro. Foram consideradas 4 maneiras diferentes de adicionar novos vértices à lista dos subproblemas ativos, conforme detalhes na tabela 2.4.

Opção	Descrição
O1	pela ordem crescente das cores
O2	primeiro a nova cor e então pela ordem crescente das cores
O3	ordem crescente do número de vértices que foram coloridos com cada cor
O4	pela ordem decrescente do número de vértices que já foram coloridos com cada cor

Tabela 2.4: Opções para adicionar vértices à lista dos subproblemas ativos

De acordo com os testes práticos dos autores, a combinação que obteve o melhor desempenho computacional foi a VB2 + O2. Para grafos menores do que um certo número, que é definido por parâmetro, é mais eficiente a enumeração das colorações possíveis do que o algoritmo Branch-and-cut. Os autores fixaram o parâmetro em 60, para grafos maiores do que 60 a enumeração completa começa no nível 2 da árvore do branch and cut.

A presença deste parâmetro, de certa forma, autentica a fragilidade do algoritmo proposto. O algoritmo não deixa de ser bastante inteligente e de atingir os seus objetivos. Entretanto, quanto mais restritivo ou seletivo for o algoritmo menos instâncias do PCG (Problema da coloração de grafos) ele poderá resolver eficientemente. De acordo com os experimentos dos autores a performance das regras de ramificação não foi alterada com o uso dos planos de corte durante o algoritmo.

Um traço marcante dos trabalhos de Mendez Diaz e Zabala é o uso de vários parâmetros tanto na definição teórica das soluções como na sua implementação. De fato, os parâmetros são muito eficientes e tem o seu uso facilmente justificado. Entretanto, eles podem deformar a natureza determinística que se busca em um algoritmo exato. Um parâmetro mal configurado pode levar o algoritmo a devolver resultados completamente diferentes. Um exemplo disto é um parâmetro que determina o número máximo de iterações de um algoritmo, que mal definido pode levar ao término prematuro do algoritmo.

2.3.5 Geração De Planos de Corte

Em um algoritmo Branch and cut, algumas decisões precisam ser tomadas: quando gerar os planos de corte, por quantas iterações executar o algoritmo de planos de corte e quantos

cortes gerar em cada iteração. É fundamental encontrar o equilíbrio entre ramificação (branch) e corte (cut), uma árvore pequena nem sempre corresponde a baixos tempos de computação.

O problema de identificar inequações violadas é chamado pelos autores de **separação**. A separação é um processo que busca dentro do grafo estruturas, como por exemplo, bloco, cliques etc. e verifica se ela viola a inequação específica para aquela estrutura. Inequações são criadas com base em características particulares das estruturas em questão, o seu objetivo, naturalmente, é eliminar ainda mais a simetria entre as colorações possíveis.

2.3.5.1 Inequações de Clique

Para encontrar as cliques que violam as inequações, os autores desenvolveram uma heurística. Esta heurística foi definida anteriormente na seção 2.2.3.1 do algoritmo dos planos de corte e, não está sendo reproduzida aqui para evitar redundâncias.

2.3.5.2 Inequações Cor Bloco

A inequação **cor de bloco** (2.10) é aplicada pelo método da "força bruta". Os autores enumeram todas as n^2 inequações e encontram aquelas que são violadas pela solução fracionária atual.

2.3.5.3 Inequação Caminho Multi Cor

Para cada variável fracionária w_k, é associado a cada aresta (u,v) $\in$ E, o custo c_{uv} conforme descrito abaixo:

$$c_{uv} = \max_{j=1,\ldots,k-1}\{x_{uj} + x_{vj} - w_j\} + \sum_{j=k}^{n}(x_{vj} + x_{vj}) \tag{2.16}$$

Por meio de uma heurística, é computado para cada vértice v $\in$ E, o caminho mais custoso em G. Um caminho com o custo maior do que w_k corresponde a uma inequação **caminho multi-cor** (2.11) violada. A heurística utiliza-se de um parâmetro para limitar o número de vezes que um mesmo vértice é utilizado para formar caminhos distintos.

2.3.6 Experimentos Computacionais

Nos experimentos os autores utilizaram instâncias COLOR02 [19] e grafos aleatórios [15]. Na seção 2.3.3 foi descrita a redução no tamanho do problema. A remoção provou-se muito eficiente em instâncias DIMACS, o percentual de redução varia de 1% a 93%. De fato, o desempenho da redução é muito bom em grafos com baixa densidade embora existam instâncias de diferentes densidades. A redução é **ineficiente** em grafos aleatórios.

Na seção 2.3.4 foram descritas as regras para a ramificação da árvore de subproblemas. As regras de ramificação e desempate combinadas resultam em um total de 8 possíveis estratégias de ramificação e pesquisa na árvore. Os resultados computacionais, obtidos sobre grafos aleatórios de diferentes densidades, revelaram que as regras de ramificação não são modificadas se os planos de corte forem adicionados durante a execução do algoritmo. Dentre as oito possibilidades a mais produtiva é a VB2 + O2 (ambos descritas nas tabelas 2.3 e 2.4).

2.3.6.1 Planos de Corte

Segundo os autores uma maneira indireta de medir a qualidade dos planos de corte é observar a aproximação do valor do limite inferior em relação a χ(G) quando o algoritmo é adicionado ao relaxamento-LP. Existe um fator de equilíbrio a ser considerado nos planos de corte, os cortes no politopo não devem ser muito profundos, por que desta maneira eles aumentam o consumo de memória atrasando o tempo de solução. As pesquisas dos autores apontam que a boa performance do algoritmo é devido principalmente a presença das inequações de clique.

No algoritmo dos planos de corte, a combinação utilizando as inequações da **clique** (2.6), **cor de bloco** (2.10)e **caminho multi-cor**(2.11) é a melhor para grafos de média densidade e, seu desempenho é razoável para as demais densidades. Entretanto, o que pesou na escolha dessas inequações foi o fato de que os grafos de média densidade são os mais difíceis de colorir. A densidade dos grafos não aparenta ser um fator crucial, entretando o desempenho dos cortes aumenta quando diferença entre o tamanho da clique máxima e o número cromático aumenta.

Como conclusão final dos experimentos com o Branch-and-Cut, os autores chegaram a conclusão de que este é capaz de resolver as instâncias de teste mais rápido e produzinho menos subproblemas do que o Branch-and-Bound [20].

2.3.7 Considerações Finais

O algoritmo proposto pelos autores conseguiu solucionar instâncias que o DSATUR [5] não foi capaz. Em muitos casos o DSATUR encontra a solução ótima cedo, mas o processo de enumeração leva muito tempo para concluir que não há solução melhor. Branch and cut foi capaz de identificar a solução ótima mais rápido que o DSATUR [5].

O melhoramento do limite inferior inicial permitiu aos autores provar que a solução obtida pela heurística inicial era ótima. A grande vantagem do método proposto é que ele provê bons limites inferiores. Então, se o limite superior é bom o suficiente, o algoritmo tem boas chances de encontrar uma solução ótima.

2.4 UM ALGORITMO EXATO PARA COLORAÇÃO DE GRAFOS

2.4.1 Introdução

A abordagem proposta por Lucet, Mendes e Moukrim[14] é um método baseado em sucessivas decomposições do grafo representativo provendo subgrafos sucessivos resolvidos e os seus correspondentes conjuntos separadores chamados **conjuntos limítrofes**. Conjunto limítrofe é um conjunto de vértices que separam os vértices coloridos dos não coloridos.

Os vértices que fazem parte do conjunto limítrofe são vértices que ainda não foram coloridos. Não existe conectividade entre os vértices coloridos e os não coloridos, a não ser através do conjunto limítrofe. A cada passo as colorações para os subgrafos são resolvidas e são representadas por diferentes estados do conjunto limítrofe. O conceito de conjunto limítrofe será discutido mais adiante em 2.4.3.1.

O número de estados cresce exponencialmente com o tamanho do conjunto limítrofe. A principal vantagem do método é que o fator exponencial de sua complexidade não depende do tamanho do grafo, mas somente de uma medida que será detalhada em 2.4.2, o comprimento linear [21]. Em uma ordenação ótima dos vértices, o tamanho máximo do conjunto limítrofe corresponde ao comprimento linear. Este algoritmo, como a grande maioria dos seus pares, trabalha sobre grafos não direcionados e conexos.

2.4.2 Comprimento Linear

O comprimento linear (*Linearwidth*) de um grafo G é definida por Bodlaender e Thilikos em [21] como o menor inteiro k tal que as arestas de G podem ser arranjadas em uma ordenação linear (e_1,...,e_r) de tal maneira que para todo i= 1,...,r – 1, existem no máximo k vértices incidentes a pelo menos uma aresta que pertence a {e_1,...,e_i} e que são incidentes a pelo menos uma aresta que pertence a {e_{i+1},...,e_r}, conforme podemos observar na figura (2.5). O cálculo do comprimento linear é um problema NP-Difícil [14, 21].

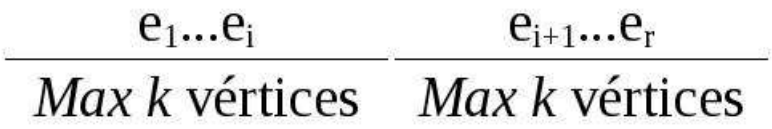


Figura 2.5: Comprimento linear

Segundo Bodlaender e Thilikos [21], está provado que muitas variantes de problemas que aparecem na busca em grafos podem ser reduzidos ao problema de computação do comprimento linear.

2.4.3 O Algoritmo

Considere um grafo G = (V,E). Seja N = |V| e M = |E|. Os vértices de G são numerados de acordo com a ordenação linear f: V $\Rightarrow$ {1,..,N}. Seja V_i o subconjunto de V, com vértices numerados de 1 até i. Seja H_i = (V_i, E_i) o subgrafo de G induzido por V_i. F_i é o conjunto limítrofe de H_i, u $\in$ F_i se e somente se $\exists$ (u,v) $\in$ E(G) e u $\leq$ i < v, vide figura (2.7). Seja H'_i = (V'_i, E'_i) o subgrafo de G induzido por V'_i = (V \ V_i) $\cup$ F_i. Qualquer tipo de relação entre os vértices de H_i e aqueles de H'_i depende dos vértices de F_i.

É importante notar que a definição de V'_i inclue F_i, portanto o conjunto limítrofe também faz parte da porção não colorida do grafo G.

2.4.3.1 Príncipio da Decomposição Linear

A decomposição linear é um método dinâmico. Durante a coloração o algoritmo desmembra o grafo G em N subgrafos H_1 ... H_n e os N conjuntos limítrofes correspondentes F_1,...,F_N. Começando a partir de uma ordenação linear o algoritmo constroe na primeira iteração um subgrafo H_1 com apenas 1 vértice. A partir deste primeiro passo a cada iteração o próximo vértice e as suas arestas correspondentes são adicionadas, formando assim soluções parciais construídas a partir do passo anterior até H_n.

Conforme podemos observar na figura 2.6 no passo (a) o conjunto limítrofe (cinza) tem apenas um elemento. A cada iteração o conjunto limítrofe vai adicionando novos vértices e, os vértices ao saírem do conjunto limítrofe são coloridos. Na execução passo a passo é possível notar que não existem arestas entre os vértices coloridos e os vértices não coloridos. A conectividade é feita através do conjunto limítrofe.

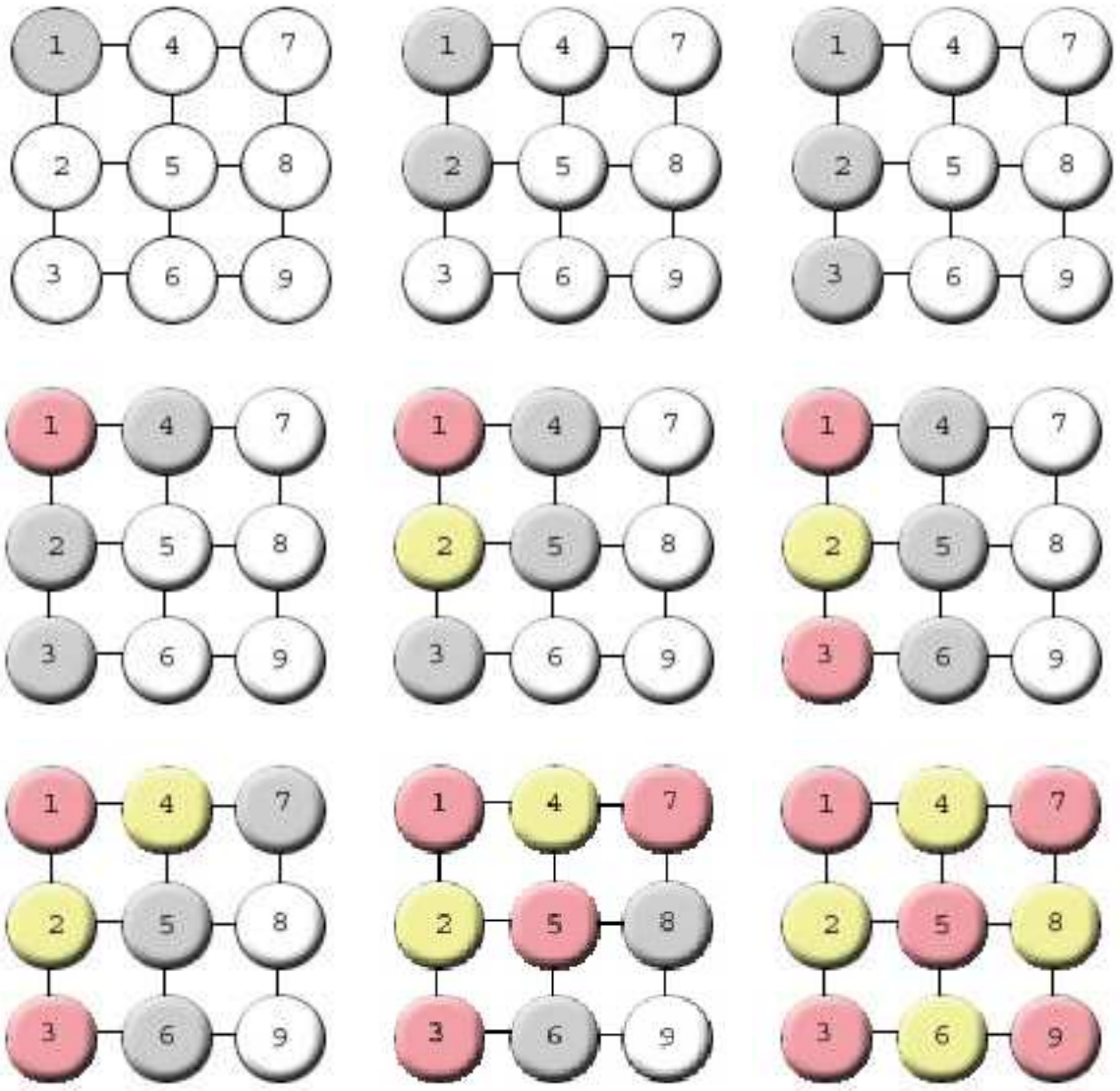


Figura 2.6: Coloração executada passo a passo

Para todo 1 ≤ i ≤ N, em cada subgrafo H_i existe um conjunto limítrofe F_i contendo vértices de H_i os quais tem pelo menos 1 vizinho em H_i' (restante do grafo). Na figura 2.7 podemos ver o algoritmo colorindo um grafo. Na figura 2.8 podemos ver uma representação de um subgrafo H_i daquele contendo apenas os vértices coloridos e o conjunto limítrofe e, na figura 2.9 temos representado o grafo H_i'.

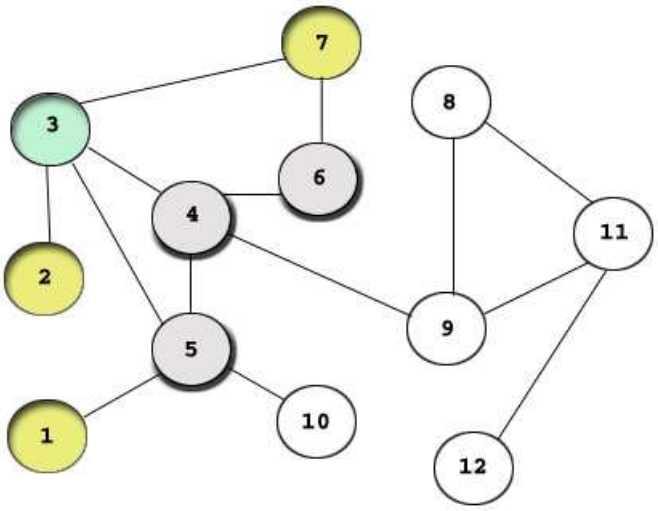


Figura 2.7: Coloração parcial de G. Conjuntos limítrofes destacados em cinza.

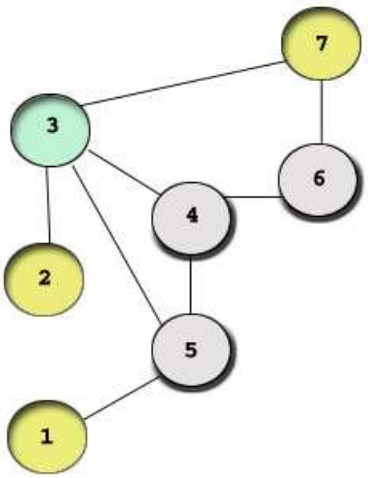


Figura 2.8: O subgrafo H_i. F*i* (Conjunto limítrofe) em cinza.

O conjunto limítrofe F_i é construído a partir de F_{i-1} adicionando o vértice i e removendo os vértices que não são vizinhos de vértices com um número de ordenação maior do que i. Entenda como número de ordenação a posição(número) do vértice após a enumeração inicial dos vértices, adiante será possível ver mais detalhes a este respeito.

Diversas colorações de H_i podem corresponder a mesma coloração de F_i. As cores utilizadas pelos vértices $V_i \setminus F_i$ não interferem com a coloração dos vértices que tem um número de ordenação maior do que i (porção não colorida do grafo), uma vez que não existem arestas entre eles. Portanto, somente soluções parciais de F_i tem que ser armazenadas em memória. Desta maneira diversas soluções parciais em H_i podem ser resumidas por uma solução única parcial em F_i, chamada **configuração** de F_i.

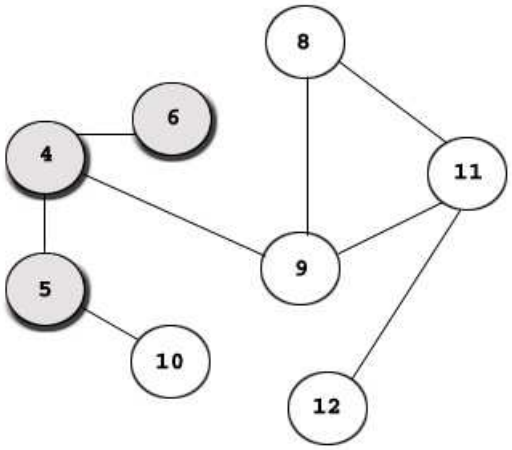


Figura 2.9: O subgrafo H_i'. Vértices não coloridos e F*i* (conjunto limítrofe).

O problema da coloração do grafo é resolvido a cada passo pelas configurações dos conjuntos limítrofes F_i. Na iteração i, o subgrafo H_{i-1} é resolvido. Isso significa que a cada configuração de F_{i-1} corresponde as cores necessárias para colorir H_{i-1}. Portanto, as soluções parciais são construídas usando soluções dos passos anteriores, conforme podemos verificar na figura (2.6).

2.4.3.2 Configurações do Conjunto Limítrofe

A configuração do conjunto limítrofe F_i é uma dada coloração dos vértices de F_i. Ele pode ser representado por uma partição de F_i, denotada B_1,...,B_j, tal que dois vértices u,v de F_i estão no mesmo bloco B_c se, e somente se eles tem a mesma cor. O número de configurações de F_i depende obviamente no número de arestas entre os vértices de F_i.

O número mínimo de configurações é 1. O número máximo de configurações de F_i é igual ao número de partições de um conjunto com $| F_i |$ elementos, ou seja, para uma clique o número de configurações equivale ao tamanho da clique $| F_i |$. Quando não existem arestas entre os vértices que compõem o conjunto limítrofe, todas as partições devem ser consideradas.

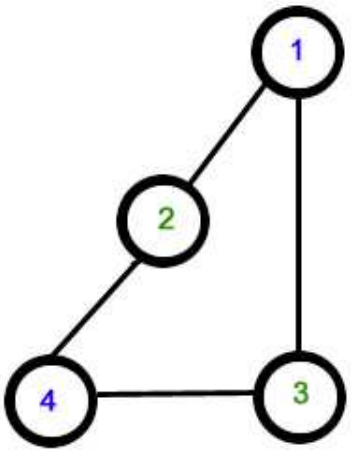


Figura 2.10: Grafo de exemplo para os conjuntos de partições

	J=1	J=2	J=3	J=4
I=1	[1]			
I=2		[1][2]		
I=3		[1][23]	[1][2][3]	
I=4		[14][23]	[1][23][4] [14][2][3]	[1][2][3][4]

Figura 2.11: Classificação dos conjuntos de partições contendo até 4 elementos

Na figura (2.10) temos o grafo G de 4 vértices e 4 arestas, este será usado para exemplificar a construção das configurações dos conjutos limítrofes. Na figura (2.11) temos a construção passo a passo das configurações de G. Na primeira rodada existe apenas uma configuração, com o vértice 1.

A cada iteração do grafo (representado por I), um novo vértice é adicionado às configurações criadas no passo anterior e dando origem a novas configurações. Na figura (2.11) é possível ver o novo vértice, destacado em vermelho, entrando nas configurações do passo anterior e gerando novas configurações. Ao final, a configuração com o menor número de blocos é utilizada para calcular χ(G).

É possível ver claramente na figura (2.11) que as configurações criadas no passo 2, por exemplo, são usadas para criar as configurações do passo 3, mas não as do passo 4. Na figura (2.11) a variável J representando o número de blocos (conjuntos independentes) de cada configuração.

Seja C(H_i, x) a x-ésima configuração de F_i para o subgrafo H_i. O seu valor, denotado por val(C(H_i,x)) é igual ao número mínimo de cores necessárias para colorir H_i para esta configuração. No caso do grafo da figura (2.10), um grafo de 4 vértices, val(C(H_4,1)) é igual a 2. Ou seja, o grafo pode ser colorido com apenas 2 cores. Veja na figura (2.11) linha (I) 4, coluna (J) 2 a coloração em questão.

Mais um exemplo do cálculo de val(C(H_i,x)), está representado na figura (2.12). Para o mesmo conjunto limítrofe F_5 = [4,5], duas diferentes colorações de H_5 correspondem a mesma configuração de F_5. Somente 2 cores são necessárias para colorir H_5 com a configuração de vértices 2 e 3 com a mesma cor, enquanto 3 cores são necessárias quando vértices 2 e 3 tem cores diferentes. O valor da configuração C(H_5, 1), representada pela partição [45], é 2. Portanto, somente o melhor resultado é considerado.

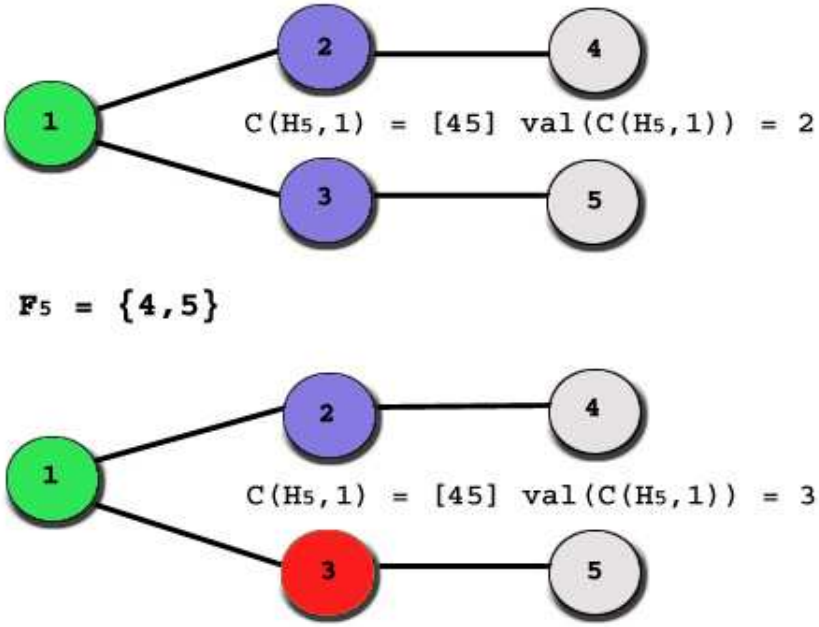


Figura 2.12: Diferentes colorações de H_5 mas a mesma configuração de F_5.

2.4.3.3 Algoritmo de Coloração

Conforme podemos observar no pseudo código do algoritmo na tabela 2.5, notem que H_1 = ({1}, 0) e F_1 = {1}. Portanto, no início existe apenas uma configuração de F_1, C(H_1, 1) = [1]. A inserção do vértice 2 de G em C(H_1, 1) pode prover uma ou duas configurações de F_2(somente uma configuração se os vértices 1 e 2 são vizinhos, caso contrário 2 configurações).

No passo i, não são examinados todas as possíveis colorações dos passos anteriores, mas somente aquelas que foram criadas no passo i - 1. Para cada configuração de F_{i-1}, obtida no passo anterior, é incluído o vértice i em cada bloco sucessivamente. Cada vez a inclusão é possível sem quebrar as regras de coloração, a configuração correspondente de F_i é gerada. Também são geradas configurações adicionando a cada configuração de F_{i-1} um novo bloco contendo o vértice i.

Ao final do algoritmo, no passo N, a configuração C(H_n, 1) é gerada das configurações do passo N - 1. Ela representa todas as colorações ótimas e o seu valor é igual a χ(G). Na tabela (2.5) temos o pseudo código do algoritmo.

Algoritmo de Coloração

```
    função ColorirGrafo( Grafo G ) retorna χ(G)
1       H_1 = ({1}, 0)
2       F_1 = {1}
3       C(H_1, 1) = [1]
4       para i = 2 até N faça
5           Construa H_i e F_i
6           para cada configuração de C(H_{i-1}, x) de F_{i-1} faça
7               para j = 1 até o número de blocos de C(H_{i-1}, x) faça
8                   se i não tem vizinhos no bloco j então
9                       config = C(H_{i-1}, x)
10                      insira i no bloco j de config
11                      gerar configuração C(H_i, y) correpondente a config
12                      se existe C(H_i, y) então
13                          val(C(H_i, y)) = min(val(C(H_i, y)), val(C(H_{i-1}, x)) )
14                      senão
15                          val(C(H_i, y)) = val(C(H_{i-1}, x))
16                      fim se
17                  fim se
18              fim para
19              config = C(H_{i-1}, x)
20              adicione a config um novo bloco contendo i
21              val(config) = max(val(C(H_{i-1}, x)), número de blocos de config)
22              gerar a configuração C(H_i, y) correspondente a config
23              se existe C(H_i, y) então
24                  val(C(H_i, y)) = min(val(C(H_i, y)), val(config)
25              senão
26                  val(C(H_i, y)) = val(config)
27              fim se
28          fim para
29      fim para
30      χ = val(C(H_N,1))
```

Tabela 2.5: Pseudo-código do Algoritmo de coloração

2.4.4 Exemplo de Cálculo da Configuração

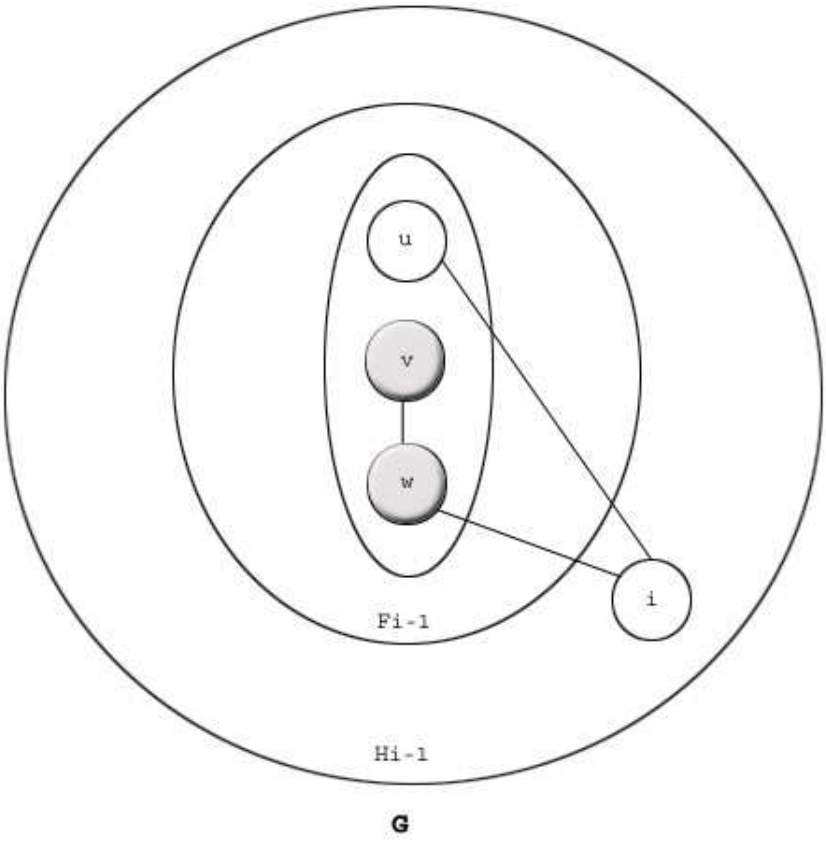


Figura 2.13: Construção de $H_i = (V_{i-1} \bigcup \{i\}, E_{i-1} \bigcup \{(u,i), (w,i)\})$

Considere que o grafo G (figura 2.13) está sendo colorido e que estamos no passo i com $F_i = \{u, v, w, i\}$. Suponha que no passo i-1, havia $F_{i-1} = \{u, v, w\}$ e que as configurações de F_{i-1} eram:

- $C(H_{i-1}, 1)$ = [uw][v] com número cromático α;
- $C(H_{i-1}, 2)$ = [uv][w] com número cromático β;
- $C(H_{i-1}, 3)$ = [u][v][w] com número cromático γ;

Os valores de α e β são pelo menos 2, uma vez que as correspondentes configurações tem pelo menos 2 blocos. O valor pode ser maior do que 2 dependendo das configurações dos passos anteriores. Pelo mesmo motivo γ tem valor pelo menos 3. A configuração de F_i será gerada com base em F_{i-1}. A configuração de F_i é gerada a partir das configurações de F_{i-1}:

- $C(H_i, 4)$ = [uw][vi] com número cromático α;
- $C(H_i, 5)$ = [uw][v][i] com número cromático $\max(\alpha, 3)$;
- $C(H_i, 6)$ = [uv][w][i] com número cromático $\max(\beta, 3)$;
- $C(H_i, 7)$ = [u][vi][w] com número cromático γ;

- C(H_i, 8) = [u][v][w][i] com número cromático max(γ,4);

Com a configuração de F_i o processo continua até que o grafo esteja completamente colorido.

2.4.5 Abordagem Alternativa

O número de configurações de um conjunto limítrofe F_i é exponencial de acordo com $|F_i|$. Além disso, uma coloração representada por k blocos precisa de pelo menos k cores. No algoritmo descrito na tabela (2.6) verifica se um grafo é k-colorível, evitando examinar configurações com mais do que k blocos, o que provou ser muito interessante quando F_{max}(tamanho máximo do conjunto limítrofe) é maior do que χ(G).

Algoritmo que testa se um grafo G é k-colorível

```
função KColorivel( Grafo G, inteiro k ) retorna booleano
    H_1 = ({1}, 0)
    F_1 = {1}
    C(H_1, 1) = [1]
    i = 2
    Resultado = verdadeiro
    enquanto i ≤ N e Resultado faça
        Resultado = falso
        Construa H_i e F_i
        para cada configuração de C(H_{i-1}, x) de F_{i-1} faça
            para j = 1 até número de blocos de C(H_{i-1}, x) faça
                se i não tem vizinhos no bloco j então
                    Resultado = verdadeiro
                    config = C(H_{i-1}, x)
                    inserir i no bloco j de config
                    gerar a configuração C(H_i, y) correspondente a config
                    se existe C(H_i, y) então
                        val(C(H_i, y)) = min (val (C(H_i, y)), val(C(H_{i-1}, x)))
                    senão
                        val(C(H_i, y)) = val(C(H_{i-1}, x))
                    fim se
                fim se
            fim para
            se número de blocos de C(H_{i-1},x) < k então
                Resultado = verdadeiro
                config = C(Hi − 1, x)
                adicionar config ao novo bloco contendo i
                val(config) = max(val(C(H_{i-1}, x)), número de blocos de config)
                gerar a configuração C(H_i, y) correspondente a config
                se existe C(H_i, y) então
                    val(C(H_i, y)) = min (val (C(H_i, y)), config)
                senão
                    val(C(H_i, y)) = val(config)
                fim se
            fim se
        fim para
        i = i + 1
    fim enquanto
```

Tabela 2.6: Pseudo-código da função que testa a k-coloração

2.4.5.1 *Decomposição Linear*

O algoritmo de decomposição linear melhorado (LDC), descrito na tabela (2.7), baseia-se em heurísticas para encontrar valores para o limite inferior (LI), através da triangulação do grafo e, o limite superior (LS), aplicando o algoritmo DSATUR [5, 16], do número cromático de G. De posse dos limites, o algoritmo começa a busca pelo valor de χ(G). Tal busca é feita através de sucessivas k-colorações, com k variando entre o limite inferior e o limite superior de χ(G).

```
Algoritmo de coloração decomposição linear (LDC)
função LDC ( Grafo G ) retorna χ(G)
        LI = Limite Inferior de χ(G)
        LS = Limite Superior de χ(G)
        enquanto LI <> LS faça
                k = valor entre LI e LS (*)
                resultado = KColorivel(G)
                se resultado = falso então
                        LI = k + 1
                senão
                        LS = k
                fim se
        fim enquanto
(*) Kdic : k = (LI+LS)/2
kSeq : k = (LI + 1)
```

Tabela 2.7: Pseudo-código do algoritmo de coloração decomposição linear (LDC)

Conforme podemos observar na tabela (2.7) são consideradas duas versões do LDC, elas variam de acordo com a maneira que o valor de k é definido: por dicotomia, denotado por Kdic-LDC ou aumentando seqüêncialmente a partir do limite inferior (LI) até o limite superior (LS), Kseq-LDC. Embora Kdic-LDC tenha uma complexidade menor que Kseq, Kseq é capaz de prover bons limites inferiores de instâncias que Kdic não é capaz de resolver.

O algoritmo LDC não produz uma coloração diretamente, entretando ela poderia ser obtida facilmente. Para tal, todas as configurações geradas durante as k-colorações deveriam ser armazenadas em memória. Quando uma k-coloração é encontrada, a configuração de F_{N-1} correspondente a F_N é escolhida, a cor 1 colore o vértice N, uma outra cor colore o vértice N - 1 e, assim por diante até o vértice 1 seja colorido.

Colorir vértices requer muito espaço em memória, entretanto não aumenta a complexidade do algoritmo.

2.4.6 Ordenação Linear

O tamanho máximo dos conjuntos limítrofes F_{max} depende da ordenação escolhida para os vértices, que por sua vez corresponde ao comprimento linear da ordenação usada para construir diferentes subgrafos H_i.

A complexidade da decomposição linear é exponencial com respeito a F_{max}, portanto é necessária fazer uma boa escolha quando ordenar os vértices de um grafo. Infelizmente, encontrar uma ordenação ótima dos vértices para obter o menor comprimento linear é um problema NP-Difícil [21].

Os autores testaram alguns métodos para realizar a ordenação dos vértices, o objetivo de ordenar os vértices é diminuir o máximo possível o número de vértices dos conjuntos limítrofes

de tamanho F_{max}. O método que retornou os melhores resultados para mais instâncias do que os demais, é o que ordena os vértices a partir de uma clique e segue numerando, em ordem decrescente, os vértices com o menor número de arestas ligando-os a vértices já numerados.

Apesar disso, o número de configurações cresce algumas vezes quando um vértice é introduzido com poucos vizinhos no conjunto limítrofe. Para evitar este efeito adverso foi adicionada uma redução dos vértices antes de cada k-coloração.

2.4.7 Redução de Vértices

Antes de cada k-coloração no LDC, alguns vértices são deletados do grafo usando a seguinte propriedade: para cada vértice u, se o grau de u é estritamente menor do que k, u e os seus adjacentes podem ser deletados do grafo.

De fato, assuma que u tem k - 1 vértices adjacentes. No pior caso, o seus vizinhos tem cores diferentes. Portanto, o vértice u teria a k-ésima cor. Ele não interfere na coloração dos demais vértices por todos os seus vizinhos já foram coloridos. Portanto, desde o início podemos considerar que ele irá ser colorido por uma cor não usada por seus vizinhos e removê-lo antes do grafo começar a coloração. Este conceito é aplicado recursivamente até o grafo ser totalmente reduzido ou não ser possível remover qualquer outro vértice.

2.4.8 Experimentos Computacionais

2.4.8.1 Grafos Aleatórios

Os algoritmos Kdic-LDC e Kseq-LDC foram testados com grafos aleatórios das densidades descritas na tabela (2.8):

Nome	Densidade
Baixa	10 %
Média	50 %
Alta	90 %

Tabela 2.8: Densidades de grafos aleatórios testados com o algoritmo de Lucet

Os algoritmos não foram muito eficientes em grafos aleatórios se comparado com o resultado de outros métodos exatos [4]. De fato, é possível colorir grafos com densidade média e com até 100 vértices. Isto se deve principalmente ao fato das arestas serem homogêneamente repartidas induzindo a um comprimento linear grande. O algoritmo proposto pelos autores foi capaz de resolver grafos com alta densidade e 60 vértices. Tais grafos tem um grande comprimento linear mas também são muito restritos[1], a cada passo poucas configurações eram geradas apesar do tamanho grande dos conjuntos limítrofes.

A decomposição linear não é eficiente com grafos aleatórios de média densidade. O tamanho do comprimento linear induz a possibilidade de gerar um número exponencial de configurações e a densidade média não limita o número de configurações válidas de maneira suficiente.

A decomposição funciona melhor com grafos de baixa densidade. O tamanho máximo do conjunto limítrofe F_{max} pode ser reduzido pela enumeração descrita em (2.4.6). Para este tipo de grafo o algoritmo foi capaz de resolver instâncias com até 100 vértices.

[1] Os autores não definem o que é um grafo restrito

2.4.8.2 Instâncias de Teste

Além dos grafos aleatórios, testes também foram feitos em instâncias do simpósio COLOR02, dentre elas as conhecidas instâncias DIMACS [19]. Para cada instância são calculados os limites inferiores e superiores. Assim como o Branch and Cut e os Planos de Corte [13, 2] a coloração não é calculada quando o limite inferior e superior, calculado pelas heurísticas iniciais, se igualam.

Os algoritmos LDC quando comparados com o Branch-and-Cut [2], de Mendez Diaz e Zabala, demonstram ser 3 vezes mais rápidos resolvendo as mesmas instâncias. Esta comparação não pode ser tomada como absoluta, pois há diferença entre ferramentas e computadores utilizados.

Os resultados de Kseq-LDC são equivalentes ou melhores do que os de Kdic-LDC nas instâncias testadas, por esta razão somente os resultados do Kseq-LDC são considerados. Os algoritmos também foram capazes de resolver instâncias que tem um grande comprimento linear. Isto se deve ao fato de nem todas as partições dos conjuntos limítrofes tiveram que ser geradas.

2.4.9 Conclusões

O método sugerido por Lucet, Mendes e Moukrim em [14] utiliza uma abordagem bastante interessante e criativa do problema. Entretanto, para grafos aleatórios a performance dos algoritmos foi inferior do que outros métodos exatos para o PCG, devido ao valor alto do comprimento linear [21] de algumas instâncias. O método proposto tem a vantagem de resolver facilmente grandes instâncias que tem um pequeno comprimento linear [21].

3 COMPARAÇÃO ENTRE OS ALGORITMOS

3.1 INTRODUÇÃO

Anteriormente foi explicado, de maneira bastante sucinta, as idéias e técnicas utilizadas para implementar os algoritmos Branch and cut [2], Planos de Corte [13] e Lucet [14]. Neste capítulo iremos um pouco mais a fundo nas particularidades de cada método. Em busca de pontos em que os métodos divergem e também dos pontos que eles convergem, independente das técnicas usadas por eles.

O resultado deste capítulo será uma visão objetiva dos algoritmos e, dentre eles qual é o que apresenta as melhores características para uma eventual otimização. A mencionada otimização pode ser o produto de uma combinação dos métodos apresentados anteriormente ou até mesmo surgir da introdução de um novo elemento.

3.2 MÉTODO DE COMPARAÇÃO

Ao decorrer deste capítulo iremos comparar os três algoritmos apresentados no capítulo 2. A comparação será feita categorizando os pontos a serem comparados e abordando-os separadamente. Os pontos de comparação foram categorizados de acordo com a sua natureza e também com a sua utilização na resolução do problema. As categorias são: Limites Inferior e Superior, Processamento e Testes Computacionais.

3.2.1 Limite Inferior e Superior

Os limites inferiores e superiores estão sendo analisados separadamente devido ao seu papel na limitação do tamanho do problema. Devido a complexidade computacional do PCG[1] é impossível garantir a entrega de resultados em tempos aceitáveis para grafos quaisquer. Portanto, criar estratégias para limitar ao máximo o tamanho do problema é vital para o sucesso de algoritmos exatos. E é exatamente este o papel dos limites inferiores e superiores nos métodos exatos descritos anteriormente.

Limite	Planos de Corte	Branch-and-Cut	Lucet
Inferior	Clique Maximal	Clique Maximal	Triangular Grafo
Superior	DSATUR	DSATUR	DSATUR

Tabela 3.1: Métodos de cálculo dos limites inferiores e superior em cada algoritmo

Conforme podemos observar na tabela (3.1) todos os métodos tem abordagens bastante parecidas para os limites inferiores e superiores. De fato, a heurística para o limite superior é a mesma para todos eles. Entretanto, o que chama a atenção é o método usado por Lucet [14] para calcular o limite inferior. Conforme o artigo de Lucet [14] é fácil calcular o número cromático de grafo triangulado.

Nos artigos de Mendez Diaz, Zabala[13, 2] e Lucet [14], os autores aplicaram os seus algoritmos tanto sobre grafos aleatórios, como sobre as instâncias do simpósio computacional COLOR02[19]. Nesta seção serão analizados os desempenhos dos algoritmos sobre 66 instâncias

[1]Problema de coloração de grafos

COLOR02. Analisar os resultados dos algoritmos sobre os mesmos problemas, ajuda na comparação e nas conclusões sobre os algoritmos em questão.

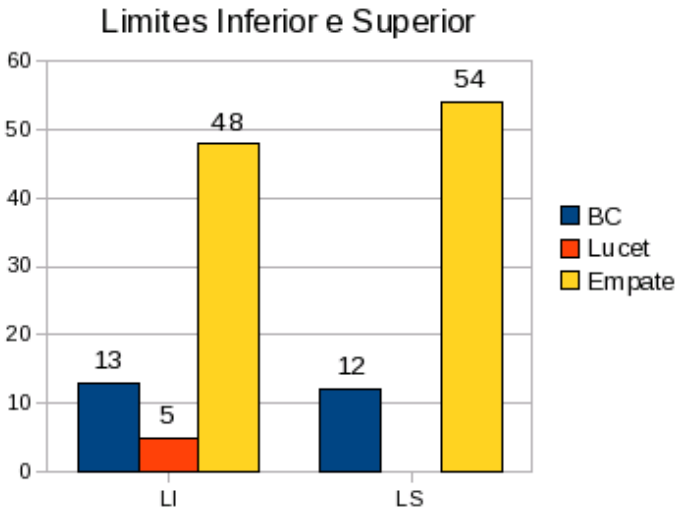


Figura 3.1: Algoritmos para Limite Inferior (LI) e Superior (LS) aplicados por Mendez Diaz, Zabala e Lucet.

Na figura (3.1) é possível ver os resultados da comparação da eficiência dos métodos de limite inferior (LI) e limite superior (LS). O algoritmo dos Planos de Corte [13] não foi incluído no gráfico por reportar poucas instâncias (14 em um total de 66 possíveis), testadas em comum com os algoritmos Branch and Cut e Lucet. Entretanto, como o algoritmo Branch and Cut utiliza os mesmos métodos, para calcular os limites inferior e superior, que o algoritmo dos Planos de Corte os resultados do Branch and Cut são usados para representar os do algoritmo dos Planos de Corte.

Conforme é possível observar na figura (3.1) existem três colunas sobre o Limite Inferior. Uma representando, as instâncias que o Branch and Cut (BC) foi melhor que Lucet, uma representando as instâncias que Lucet foi melhor que o Branch and Cut e uma terceira reportando as instâncias que ambos algoritmos calcularam os mesmos valores para o limite inferior (Empate).

Para o limite inferior podemos ver que o Branch and Cut consegue prover melhores valores para 14 instâncias (19,6%), enquanto a triangulação usada por Lucet superou a clique máxima do Branch and Cut em apenas 5 instâncias (7,5%). Para as demais 48 instâncias (72,9%) os algoritmos empatam obtendo os mesmos valores para o limite inferior.

Baseado nesses números é possível concluir que utilizar o tamanho da clique máxima como um limite inferior, na maioria dos casos, é uma abordagem mais bem sucedida do que a triangulação do grafo. Apesar do cálculo da clique máxima também ser um problema NP-Difícil[17], a heurística usada por Mendez Diaz e Zabala compensa esse fato tanto no quesito tempo de processamento, como no quesito resultado.

Para o limite superior existem apenas duas colunas na figura (3.1), uma representando as instâncias que o Branch and Cut(BC) proveu melhores limites superiores (18%) e outra representando as instâncias que ambos calcularam o mesmo valor para o limite superior (Empate). Apesar de tanto Lucet como o Branch and Cut utilizarem o DSATUR[5] para o calcular o limite superior, em nenhuma instância o limite superior calculado por Lucet foi capaz de superar o valor calculado no Branch and Cut. Uma vantagem tão grande do Branch and Cut não era esperada. Uma explicação para tal pode estar nas particularidades da heurística baseada no DSATUR utilizada no Branch and Cut.

3.2.2 Processamento

Nesta seção serão abordadas as características referentes muito mais a implementação dos algoritmos, do que a proposta teórica de suas soluções. Todos as características na tabela (3.2) aqui tem impacto considerável no resultado final das colorações e, por esta razão tem papel importante na avaliação dos algoritmos.

Característica	Planos de Corte	Branch-and-Cut	Lucet
Pré-processamento	Relaxamento LP inequações da vizinhança. Agoritmos de separação	Remove adjacentes a clique máxima	Ordenação e redução de vértices
Tempo máximo de processamento	2h	2h	30 min
Parâmetros de entrada	Max path size	Skip factor, rodadas por vértice, número máximo de cortes	k (no LDC)
Utiliza heurísticas	Sim	Sim	Sim
Programação Inteira	Sim	Sim	Não
Tamanho Mínimo do Problema	-	60	-
Softwares	C++, ABACUS framework e CPLEX LP solver	C++, ABACUS framework e CPLEX LP solver	C
Plataforma	SUN Ultra	SUN Ultra	PC AMD ATHLON XP2000+
Reduz o problema	-	Sim	Sim
Ordenação de vértices	-	Sim	Sim

Tabela 3.2: Características da implementações que impactam no resultado final dos métodos.

3.2.2.1 Pré-processamento

A característica pré-processamento descreve os procedimentos executados antes do início da coloração do grafo. Eles variam muito de algoritmo para algoritmo, entretanto, dois tipos de pré-processamento se destacam entre os demais: a ordenação dos vértices e a redução de vértices.

A ordenação dos vértices é usada para organizar a ordem em que os vértices são analisados pelos algoritmos. Este processamento está intrínsicamente ligado ao comprimento linear, no caso do algoritmo de Lucet [14]. Entretanto, o pré-processamento mais importante é a redução de vértices. Pois, tratando-se de um problema NP-Difícil, diminuir o seu tamanho ao máximo é fundamental para atingir melhores resultados.

Nesta característica não existe um claro vencedor entre os algoritmos. Na verdade, embora não fique muito claro no artigo do algoritmo Branch and Cut, a sua estratégia para redução de vértices é muito parecida com a utilizada por Lucet. Somando se a isso o fato de ambos fazerem referência a Glover, Parker e Ryan [22], o método de redução de vértices pode ser o mesmo para ambos os algoritmos. O método utilizado por Lucet foi definido por Glover, Parker e Ryan em [22].

3.2.2.2 Tempo Máximo de Processamento

Em um problema de natureza NP-Difícil é fundamental ter um limite para impedir que os algoritmos sejam executados indefinidamente, sem que resultados sejam obtidos. Entretanto, definir um limite máximo de tempo ou de número de iterações não é uma tarefa fácil, por que de maneira geral, ele é um atestado de que se o algoritmo não resolveu o problema até aquele momento ele não será mais capaz de fazê-lo, ou então, de que qualquer resultado obtido além daquele ponto não justifica o esforço computacional.

Os autores dos algoritmos Planos de Corte, Branch and Cut e Lucet escolheram o tempo máximo de execução como um limitador para os seus algoritmos de coloração. A razão pela qual os autores destes algoritmos não limitaram o número de iterações ao invés do tempo não fica clara nos textos. Entretanto, acredito que essa escolha foi feita baseada na versatilidade que os algoritmos se propoem a entregar. É difícil estimar quantas iterações um algoritmo pode ter e, aplicar esta estimativa a grafos das mais diferentes densidades e tamanhos.

Podemos considerar os valores adotados em cada algoritmo para limitar o tempo máximo de processamento como um bom parâmetro para medir a solidez do método. Os algoritmos de Mendez Diaz e Zabala [2, 13] tem um tempo limite muito alto, 2 horas. Enquanto que no algoritmo de Lucet [14] o limite é de 30 minutos, **25% do tempo daquele**.

Esta diferença pode ser lida como um indício de que o algoritmo de Lucet é mais consistente no que se refere a capacidade, que os autores atribuíram, do algoritmo resolver problemas dentro de uma janela de tempo. Além do mais, uma diferença tão acentuada entre os limites define Lucet como um algoritmo potencialmente mais rápido do que os seus pares.

3.2.2.3 Parâmetros de Entrada

São considerados parâmetros de entrada valores que são definidos antes da execução do algoritmo e, de alguma maneira influem no comportamento do algoritmo e, por fim na solução obtida. O seu uso é facilmente justificado pois eles dão flexibilidade ao algoritmo e permitem regular o seu funcionamento.

Para o algoritmo dos planos de corte, existe o parâmetro "Max Path size"que determina o tamanho máximo que um caminho pode ter na busca por violações na inequação do caminho multi cor (2.11).

Para o algoritmo Branch and Cut aplicam-se os seguintes parâmetros:

- **Skip factor** Número de vértices enumerados, na árvore enumeração, antes que o algoritmo de planos de corte sejam aplicados.
- **Rodadas por vértice** Número de iterações do algoritmo de planos de corte
- **Número máximo de cortes** Número máximo de cortes adicionados por iteração

Por último, o algoritmo de Lucet aceita apenas o parâmetro k no algoritmo LDC. O algoritmo LDC, definido na seção 2.4.5.1 determina se o grafo de entrada pode ser colorido com k cores. Contudo, o algoritmo na sua forma original não requer parâmetros de entrada, além do grafo a ser colorido.

Parâmetros de entrada também tem os seus pontos desfavoráveis. Valores mal-definidos podem levar o algoritmo a um desempenho pobre e, diferentes famílias de grafos podem ter diferentes conjuntos de valores ótimos para os parâmetros. Os algoritmos de Mendez Diaz e Zabala fazem extensivo uso de parâmetros e portanto ficam mais sujeitos as variações descritas anteriormente.

Neste quesito o Algoritmo de Lucet tem uma leve vantagem, pois utiliza muito pouco o recurso dos parâmetros e, portanto obtem resultados mais influenciados pelas particularidades do grafo de entrada, do que a fatores que não tem uma relação intrínsica com o problema a ser resolvido.

3.2.2.4 Utiliza Heurísticas

A utilização de heurísticas é uma característica presente em todos os algoritmos mencionados neste documento. E, é uma alternativa totalmente válida pois provê uma solução razoável para problemas menores, quando comparado com o PCG (Problema da coloração de grafos), como o cálculo do limite inferior e superior, mas que tem um papel importante na computação da solução do PCG. De fato, não é compensador investir mais tempo resolvendo um problema menor do que resolvendo o PCG em si.

As heurísticas podem ser um objeto de melhoria nos algoritmos apresentados no capítulo 2. Um exemplo são as heurísticas utilizadas para calcular o limite inferior nos algoritmos Branch and Cut e Planos de corte[2, 13], veja tabela (3.2). Esses algoritmos utilizam uma heurística baseada na clique máxima.

Entretanto, para os grafos que o valor de χ(G) é bem superior ao tamanho da clique máxima, as heurísticas de clique máxima tornam se menos eficientes pois retornam valores distantes de χ(G). De acordo com Ramsey[6] se G não tem cliques grandes, é de se esperar que G tenha um conjunto independente grande. Segundo Gross[11], as heurísticas normalmente proveêm resultados rápidos para limites inferiores e superiores, entretanto não se pode esperar que os resultados sejam de alta qualidade.

Esses pontos fracos do uso das heurísticas de certa forma expõe muito o algoritmo as decorrências do uso daquelas. Nos algoritmos de Mendez Diaz e Zabala[2, 13] as heurísticas são extensivamente usadas, principalmente na fase de separação. Apesar disso, o processo de separação tem uma parcela de participação mínima no tempo dos algoritmos citados. Contudo, o seu uso extensivo ainda sim deixa o algoritmo vulnerável aos resultados de várias heurísticas.

3.2.2.5 Programação Inteira

A característica Programação Inteira indica quando o algoritmo utiliza técnicas de programação inteira para resolver o problema. Conforme podemos observar na tabela (3.2), Planos de Corte e Branch-and-Cut fazem uso desta técnica. Tal fato justifica por que estes são tão diferentes do algoritmo de Lucet. Apesar disso, não existem indícios de que os algoritmos não possam ser combinados de maneira nenhuma.

3.2.2.6 Tamanho Mínimo do Problema

A característica Tamanho Mínimo do Problema representa o tamanho mínimo que um grafo precisa ter para que o algoritmo seja aplicado. Para o Branch-and-Cut [2] o grafo precisa ter no mínimo 60 vértices para o algoritmo seja utilizado na árvore de pesquisa. Um grafo de 60 vértices é um número considerável de vértices, o que invariavelmente projeta uma sombra sobre a versatilidade do método.

3.2.2.7 Reduz o Problema e Ordenação de vértices

Essas duas características já foram abordadas no tópico Pré-processamento (3.2.2.1).

3.2.3 Testes Computacionais

Nesta seção iremos abordar as estratégias utilizadas para por a prova o desempenho computacional dos algoritmos apresentados no capítulo 2. Várias instâncias de teste foram utilizadas por todos os autores, incluindo grafos aleatórios e instâncias do simpósio computacional COLOR02. Na tabela (3.3) alguns detalhes das instâncias de grafos aleatórios testadas são apresentados.

Característica	Planos de Corte	Branch and Cut	Lucet
Grafos Aleatórios	Sim	Sim	Sim
Densidade	Baixa < 30% Média ≥ 40% ≤ 60% Alta > 70%	Baixa < 30% Média ≥ 40% ≤ 60% Alta > 70%	Baixa = 10% Média = 50% Alta = 90%
Vértices nas instâncias aleatórias	125	125	Baixa = 30 a 120 Média = 30 a 60 Alta = 30 a 70
Número de instâncias aleatórias	-	24	**95**
Instâncias	COLOR02, Aleatórios	COLOR02 , Aleatórios	COLOR02, Aleatórios

Tabela 3.3: Características dos testes computacionais aplicados sobre os 3 algoritmos

3.2.3.1 Instâncias aleatórias

A característica **Número de instâncias aleatórias** que está descrita na tabela (3.3) refere-se justamente ao número de instâncias testadas. Tanto para o algoritmo de Lucet como para o Branch and Cut, os seus respectivos autores decidiram fazer rodadas de testes para cada densidade.

No Branch and Cut, nos testes para a escolha dos esquemas de corte, foram utilizadas 8 instâncias para cada densidade (baixa, média e alta). Para o algoritmo de Lucet, 5 instâncias foram testadas em todas as densidades com tamanhos de grafos que variam de 10 em 10. Por exemplo, a densidade média foi testada com grafos que tem o número de vértices variando entre 30 e 60. Portanto, foram testadas 5 instâncias para grafos com 30, 40, 50 e 60 vértices.

O artigo dos Planos de Corte [13] não detalha quantas instâncias foram testadas e sim, quantas vezes cada instância foi testada. No caso 50 vezes.

Nas estratégias de testes computacionais vemos abordagens bem distintas para o problema. Mendez Diaz e Zabala implementaram testes exaustivos para cada densidade, entretanto não fizeram diferenciações no número de vértices dos grafos testados para cada densidade. Todos tinham 125 vértices. Por sua vez o algoritmo de Lucet utilizou uma estratégia muito mais refinada, que de certa maneira privilegiou um dos pontos fortes do seu algoritmo, a sua performance em grafos de baixa densidade. De acordo com Glover, Parker e Ryan em [22] nas aplicações práticas, os grafos geralmente não são excessivamente densos.

Embora a estratégia de aumentar de 10 em 10 os tamanhos dos grafos testados seja bastante interessante, o teste massivo do mesmo número de instâncias para diferentes densidades demonstra um potencial maior de simulação do uso do algoritmo para resolver problemas práticos.

3.2.3.2 Instâncias COLOR02

Um outro tipo de instâncias testadas foram as instâncias do simpósio computacional COLOR02[19]. Os artigos do Branch and Cut e de Lucet reportam testes sobre várias instâncias do simpósio. Das instâncias do COLOR02, 66 foram testadas tanto por Branch and Cut e Lucet. O algoritmo de planos de corte também reporta testes sobre instâncias do COLOR02. Entretanto, o número de instâncias em comum com aqueles algoritmos não é suficiente para fazer uma boa comparação. Na tabela (3.4) temos a listagem das instâncias COLOR02 utilizadas na comparação entre os algoritmos, como também nos testes práticos reportados mais adiante no capítulo 4.

Instância	Vértices	Arestas	Instância	Vértices	Arestas
1-FullIns3	30	100	Inithx.i.3	621	13969
1-FullIns4	93	593	Jean	80	508
1-FullIns5	282	3247	Le450-15a	450	8168
1-Inser_4	67	232	Le450-15b	450	8169
1-Inser_5	202	1227	Le450-15c	450	16680
1-Inser_6	607	6337	Le450-15d	450	16750
2-FullIns3	52	201	Le450-25a	450	8260
2-FullIns4	212	1621	Le450-25b	450	8260
2-FullIns5	852	12201	Le450-25c	450	17343
2-Inser_3	37	72	Le450-25d	450	17425
2-Inser_4	149	541	Le450-5a	450	5714
2-Inser_5	597	3936	Le450-5b	450	5734
3-FullIns3	80	346	Le450-5c	450	9803
3-FullIns4	405	3524	Le450-5d	450	9757
3-Inser_3	56	110	Miles1000	128	6432
3-Inser_4	281	1046	Miles1500	128	10396
3-Inser_5	1406	9695	Miles250	128	387
4-FullIns3	114	541	Miles500	128	2340
4-FullIns4	690	6650	Miles750	128	4226
4-Inser_3	79	156	Mug100-1	100	166
4-Inser_4	475	1795	Mug100-25	100	166
5-FullIns3	154	792	Mug88-1	88	146
5-FullIns4	1085	11395	Mug88-25	88	146
Anna	138	493	Mulsol.i.1	197	3925
David	87	812	Mulsol.i.2	188	3885
Fpsol2.i.1	496	11654	Mulsol.i.3	184	3916
Fpsol2.i.2	451	8691	Mulsol.i.4	185	3946
Fpsol2.i.3	425	8688	Mulsol.i.5	186	3973
Games120	120	638	School1	385	19095
Homer	561	1629	School1_nsh	352	14612
Huck	74	602	Zeroin.i.1	211	4100
Inithx.i.1	864	18707	Zeroin.i.2	211	3541
Inithx.i.2	645	13979	Zeroin.i.3	206	3540

Tabela 3.4: Lista das 66 instâncias COLOR02 que foram utilizadas para a comparação dos algoritmos e também nos testes práticos.

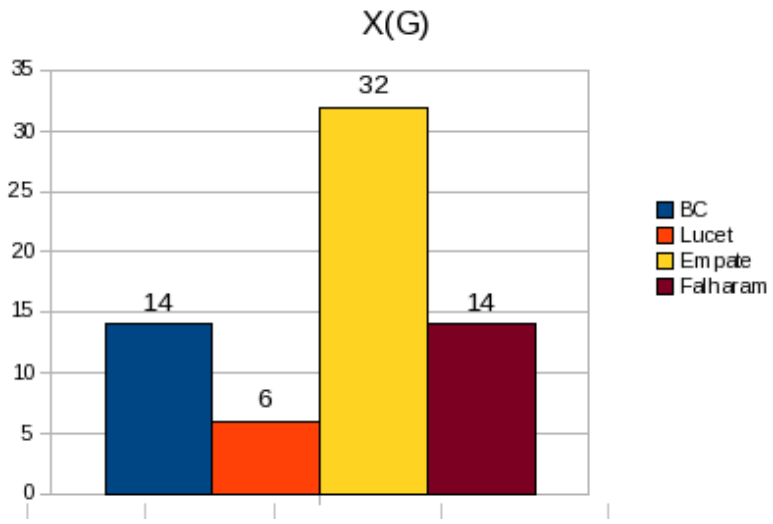


Figura 3.2: χ(G) obtido por Branch and Cut e Lucet nos testes em 66 instâncias COLOR02

Na figura (3.2) nós temos quatro colunas. De um total de 66 instâncias, listadas na tabela (3.4), a coluna BC representa as instâncias que o Branch and Cut conseguiu colorir e o algoritmo de Lucet não conseguiu. A coluna "Lucet"representa as 6 instâncias que o algoritmo de Lucet conseguiu colorir e Branch and Cut não. A coluna "Empate"representa as instâncias que foram coloridas pelos dois algoritmos. E a coluna "Falharam"representa as instâncias que ambos algoritmos não conseguiram colorir.

Analisando superficialmente os números apresentados na figura (3.2) é possível chegar a conclusão de que o algoritmo Branch and Cut[2] apresenta um desempenho superior do que o algoritmo de Lucet[14]. Branch and Cut (BC) resolveu 14 de 66 instâncias, enquanto Lucet resolveu 6 de 66. Uma vantagem de 8 (12,12%) instâncias. Entretanto, o algoritmo BC tem um tempo de máximo de execução de **2 horas**, quatro vezes mais do que Lucet, que tem apenas 30 minutos.

Considerando estes fatos e também o fato de Lucet ter resolvido uma instância COLOR02 (**4-Inser_3**), que segundo o artigo de Lucet[14] até então não tinha sido resolvida por um método exato, é possível dizer que ambos algoritmos tiveram um desempenho muito bom na amostra testada e, que a diferença no número de instâncias resolvidos é atenuada pelo tempo máximo de processamento que favorece o algoritmo Branch and Cut.

3.3 COMPARAÇÃO ENTRE BRANCH AND CUT E LUCET PUBLICADA

No artigo no qual apresenta os resultados de seu algoritmo, Lucet[14] também faz uma comparação dos valores de χ(G) obtidos pelo seu algoritmo com os valores de χ(G) obtidos pelo algoritmo do Branch and Cut[2]. Esta comparação é bem diferente da que foi feita neste documento, tanto no formato quanto em conteúdo.

Checando os resultados relatados em cada um dos artigos, foi possível identificar pequenas diferenças entre os resultados testes do Branch and Cut sobre as instâncias COLOR02. Tais diferenças estão relatadas na tabela (3.5).

De fato, o algoritmo Branch and Cut[2] está entre as referências do artigo de Lucet[14]. Entretanto, a diferença principal entre a referência feita por Lucet e a feita neste trabalho, é o ano de publicação do artigo sobre o Branch and Cut. Lucet referencia a um artigo publicado em 2002. Enquanto que este trabalho referencia o mesmo artigo, porém publicado em 2006.

Branch and Cut		
Instância	χ(G) - **2002**	χ(G) - **2006**
2-Inser_4	4-5	4
Miles250	*[2]	8
Le450_5a	5-9	5
Le450_5b	5-9	5
Le450_5c	*	5
Le450_5d	5-10	5
Le450_15b	15-17	15
Le450_15c	15-24	15
Le450_15d	15-23	15
Le450_25a	*	25
Le450_25c	25-28	25
Le450_25d	25-28	25

Tabela 3.5: Diferença entre resultados do Branch and Cut reportados em **anos** distintos

Conforme está descrito na tabela (3.5) é possível deduzir que tal diferença seja devida à evolução do algoritmo do Branch and Cut neste período de tempo, que melhorou em muito os seus resultados e, por consequência sobreescreveu resultados anteriores.

3.4 CONCLUSÕES

Após analisar profundamente os três artigos, várias referências na literatura e, comparar as principais características dos três algoritmos é possível desenhar um panorama mais claro a respeito dos mesmos. Embora muito diferentes, as suas soluções se interceptam em muitos momentos, além do objetivo inicial de resolver o problema da coloração. Como por exemplo, no uso de heurísticas, na redução do número de vértices no pré-processamento, utilização de grafos aleatórios e instâncias COLOR02 para testes e assim por adiante.

Apesar dos métodos propostos por Mendez Diaz e Zabala[2, 13] apresentarem excelentes resultados e terem uma abordagem bastante interessante do problema, o algoritmo de Lucet, Mendes e Moukrim [14] apresenta nas características mencionadas acima uma série de pequenas vantagens, que quando combinadas, sugerem uma maior eficiência na resolução do problema.

O desempenho do algoritmo de Lucet pode sofrer variações nos momentos que antecedem a execução do mesmo. Na ordenação inicial dos vértices e nos valores dos limites superiores e inferiores, ainda na fase de pré-processamento. Entretanto, durante a execução do miolo do algoritmo não existe a utilização de heurísticas, portanto o algoritmo assume uma postura mais determinística. Menos suscetível a parâmetros de entrada e também às particularidades das implementações de heurísticas.

Considerando todos os fatos mencionados anteriormente, o algoritmo de Lucet [14] será utilizado como ponto de partida para uma possível otimização e combinação do conhecimento gerado em todos os artigos.

4 IMPLEMENTAÇÕES DA DISSERTAÇÃO

4.1 INTRODUÇÃO

Neste capítulo serão descritas as experiências práticas obtidas na implementação dos algoritmos de coloração grafos. Foram implementados o seguintes algoritmos: Coloração sequencial[23], o qual foi provido por uma das bibliotecas utilizadas, DSATUR[5] e Lucet[14].

Para a implementação dos algoritmos descritos anteriormente, foi utilizado um computador equipado com processador Intel Core 2 Duo 2GHz, 120GB de disco rígido e 2GB de memória RAM. O sistema operacional foi Mandriva Linux 2008, kernel 2.6.24, linguagem C++, compilador g++ (GCC) versão 4.2.3. A biblioteca BGL[24] (Boost Graph Library) versão 1.38 foi utilizada por oferecer uma variedade de algoritmos e estruturas de dados para a implementação de algoritmos de grafos. A biblioteca é quase 100% implementada utilizando templates da linguagem C++, este fato também influenciou a implementação que será descrita mais adiante.

Todos os algoritmos foram testados com as mesmas 66 instâncias COLOR02 que foram testadas em comum por Branch and Cut e Lucet, a listagem completa das instâncias pode ser encontrada na tabela (3.4).

4.2 COLORAÇÃO SEQUÊNCIAL

O algoritmo de coloração sequêncial foi definido por Coleman e More em [23] e está implementado dentro da biblioteca boost. Trata-se de um algoritmo simples e muito rápido para a coloração de grafos. Ele foi incluído na fase de implementação por tratar se de um algoritmo pronto na biblioteca e pelo conhecimento da biblioteca que a sua utilização agregou.

Em termos de tempo de processamento este algoritmo é realmente excepcional. Ele foi testado contra as mesmas 66 instâncias que o Branch and Cut e o algoritmo de Lucet foram testados, veja a tabela (3.4), conforme descrito na seção (3.2.3). O algoritmo sequêncial coloriu todas as instâncias e, para nenhuma delas ele requereu mais do que **2 segundos** para calcular uma estimativa do número cromático e também a coloração.

Entretanto, para uma performance tão espetacular ser possível, em termos de tempo total de processamento, é inevitável que o algoritmo tenha um ponto fraco. Neste caso, o ponto fraco do algoritmo é a precisão da estimativa do número cromático χ(G) calculado. Foi feita uma comparação entre os valores de χ(G) obtidos ou pelo Branch and Cut ou pelo algoritmo de Lucet e, as estimativas obtidas pelo algoritmo sequêncial. Os resultados estão compilados na figura (4.1).

Na figura (4.1) existem 5 colunas distintas, elas representam as seguintes informações:

- **Falharam**: representa as 14 instâncias que Branch and Cut e Lucet não conseguiram colorir. O algoritmo sequêncial coloriu todas elas e, para 7 dessas instâncias a estimativa do valor de χ(G) igualou o melhor limite superior calculado por Branch and Cut e Lucet.

- **Empate**: representa as 24 instâncias que a estimativa de χ(G) calculado pelo algoritmo sequêncial igualou o χ(G) calculado por Branch and Cut ou o algoritmo de Lucet.

- **Próximo (até 5)**: representa as 12 instâncias que o valor de χ(G) estimado pelo algoritmo sequêncial foi no máximo 5 unidades maior do que o valor de χ(G) calculado ou pelo Branch and Cut e pelo algoritmo de Lucet.

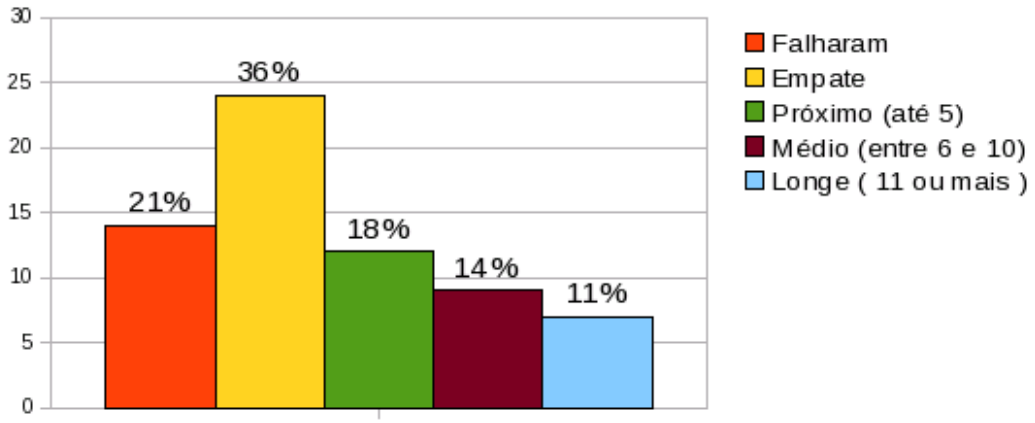


Figura 4.1: Comparação dos resultados do algoritmo sequêncial com os melhores resultados obtidos por Branch and Cut e Lucet.

- **Médio (entre 6 e 10)**: representa as 9 instâncias que o valor de χ(G) estimado pelo algoritmo sequêncial foi entre 6 e 10 unidades superior, que o valor de χ(G) calculado ou pelo Branch and Cut e pelo algoritmo de Lucet.

- **Longe (11 ou mais)**: representa as 7 instâncias que o valor de χ(G) estimado pelo algoritmo sequêncial foi 11, ou mais, unidades superior que o valor de χ(G) calculado ou pelo Branch and Cut e pelo algoritmo de Lucet.

Como podemos ver na figura (4.1) o algoritmo sequêncial fica muito atrás dos algoritmos de Lucet e Branch and Cut em termos de precisão dos valores de χ(G). O algoritmo sequêncial retorna boas aproximações de χ(G) para mais de 50% das instâncias testadas. Entretanto, em nenhum momento ele supera os limites superiores calculados pelos outros dois algoritmos, apesar dele o igualar em muitas vezes.

Portanto, o algoritmo sequêncial pode ser a melhor opção para colorir um grafo somente quando estão presentes severas restrições de tempo. Como por exemplo, a compilação de um programa ou então uma aplicação de tempo real.

4.3 DSATUR

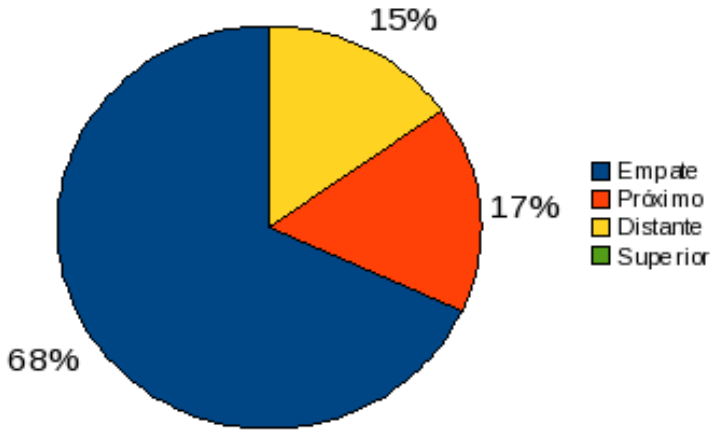
O passo seguinte da implementação foi explorar mais os recursos da biblioteca Boost e, da própria linguagem implementando um algoritmo de coloração mais simples. O escolhido foi o algoritmo DSATUR[5] por se encaixar naquela descrição e, também por ser usado para calcular o limite superior nos algoritmos do Branch and Cut[2] e Lucet[14].

O algoritmo DSATUR foi testado contra as mesmas 66 instâncias COLOR02[19] que os algoritmos de Lucet e Branch and Cut foram, conforme dados reportados anteriormente. O algoritmo foi implementado exatamente conforme descrito por Brélaz em [5]. A lista completa das 66 instâncias testadas está na tabela (3.4).

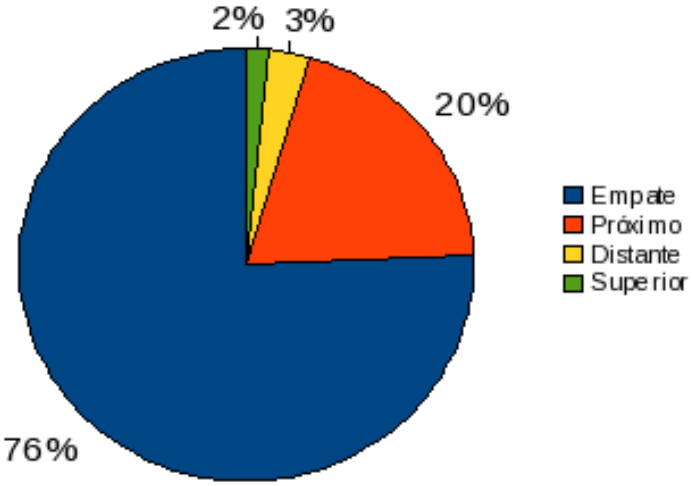
Nas figuras (4.2) temos as seguintes categorias representadas:

- **Empate**: A implementação do DSATUR e o limite superior calculado para cada algoritmo retornaram o mesmo valor para χ(G).

- **Próximo**: O limite superior calculado por cada algoritmo para χ(G) foi apenas 1 unidade inferior do que a implementação do DSATUR. DSATUR = Limite superior de Lucet/Branch and Cut + 1.



(a) DSATUR e Limite Superior calculado para Branch and Cut



(b) DSATUR e Limite Superior calculado para Lucet

Figura 4.2: Comparação do algoritmo DSATUR com o limite superior calculados nos artigos do Branch and Cut e Lucet.

- **Superior**: A implementação do DSATUR retornou uma estimativa para χ(G) melhor do que o limite superior calculado por cada algoritmo.
- **Distante**: A implementação do DSATUR estimou um valor para χ(G) muito acima do que o limite superior calculado por cada algoritmo.

Após a implementação do algoritmo foi possível tirar algumas conclusões a respeito dos métodos para calcular o limite superior usado por cada algoritmo. Conforme podemos observar na figura (4.2) a nossa implementação do DSATUR e o limite superior calculado pelo algoritmo de Lucet[14], que também é uma implementação do DSATUR, tem resultados muito parecidos. Para 96% das instâncias eles são ou iguais ou o DSATUR[5] de Lucet é uma unidade inferior. Para uma instância a nossa implementação do DSATUR foi 1 unidade mais próxima de χ(G) do que o DSATUR de Lucet.

A implementação que foi feita do DSATUR foi exatamente aquela proposta por Brelaz em [5], portanto a implementação feita por Lucet também deve ser muito parecida com a original.

Na comparação com os resultados reportados no algoritmo do Branch and Cut[2], os valores reportados por Mendez Diaz e Zabala em [2] são muito melhores do que os da

implementação original do DSATUR. Apesar que, de acordo com o artigo original [2] o limite superior é calculado por uma heurísitica **baseada** no DSATUR e, portanto não é exatamente o DSATUR. Como é possível ver na figura (4.2) para 85% ambas implementações reportaram valores muito aproximados, entretanto para os 15% restantes das instâncias o limite superior do Branch and Cut foi muito mais apurado.

4.4 ALGORITMO DE LUCET

O algoritmo de Lucet[14] teve a sua implementação iniciada, porém ela não foi totalmente concluída. Apesar disso a tentativa de implementar este algoritmo foi muito valiosa, no sentido de que ajudou a entender melhor como o algoritmo funciona e como as suas partes interagem. No artigo original de Lucet [14] não são mencionados os detalhes práticos da implementação do mesmo.

Um exemplo é o fato do algoritmo retornar sempre a primeira configuração (val(C(H_N,1))), vide tabela (2.5), como o valor de χ(G). O algoritmo retorna a primeira configuração por que ela é gerada com o menor número de blocos que o algoritmo pode calcular. Quanto menor o número de blocos, menor o número cromático.

A implementação foi modularizada utilizando classes, existem classes para os blocos, as configurações, o conjunto limítrofe, sem contar a classe principal da coloração que controla todas as demais. Para agilizar a codificação e também aprofundar o conhecimento do algoritmo, a implementação começou no alto nível do algoritmo. Primeiro foram declarados os principais métodos e, a partir de então o esqueleto do algoritmo foi escrito e concluído. A estratégia é, a partir daí implementar os métodos um a um.

Após as declarações iniciais os métodos começaram a ser implementados. A partir deste código o algoritmo pode ser concluído, entretanto a estrutura que temos na tabela (2.5), foi modularizada e finalizada. É possível futuramente concluir a implementação do algoritmo utilizando o código já escrito.

Durante discussões a respeito da implementação do algoritmo de Lucet uma idéia surgiu para otimizar um dos passos do algoritmo. Na tabela (4.1) temos um trecho do pseudo código do algoritmo de Lucet. O trecho foi extraído da tabela (2.5). Na linha 8, o algoritmo testa se o vértice i tem vizinhos no bloco j.

Em um primeiro momento essa verificação poderia ser implementada checando a vizinhança de **todos** os vértices do bloco j. Entretando, considerando que o algoritmo mantém os conjuntos limítrofes e, como tais eles atuam como um divisor entre um lado do grafo e o outro. Ou seja, os vértices que não estão mais no conjunto limítrofe não podem ter arestas com aqueles que ainda vão entrar no conjunto limítrofe. Como é o caso do vértice i, no nosso exemplo.

8	**se** i não tem vizinhos no bloco j **então**
	...
17	**fim se**

Tabela 4.1: Trecho do pseudo código do algoritmo de Lucet que testa vizinhança de vértices no bloco

Portanto faz mais sentido, verificar se i é vizinho dos vértices que estão em F_i e no bloco j ao mesmo tempo. A linha de código poderia ser reescrita de acordo com a tabela (4.2). Naturalmente o uso da interseção na linha 8 da tabela (4.2) foi meramente ilustrativo. É possível obter os mesmos resultados apenas testando se um vértice u $\in$ bloco j é membro de F_i antes de verificar se i é vizinho de u.

8	**se** i não tem vizinhos em (bloco j $\cap$ F_i) **então**
	...
17	**fim se**

Tabela 4.2: Trecho modificado do pseudo código do algoritmo de Lucet que testa vizinhança de vértices no bloco

4.5 CONCLUSÃO

A fase da implementação permitiu o aprofundamento do estudo dos algoritmos e seus comportamentos. O estudo da biblioteca Boost[24] foi muito importante pois agregou uma série de classes e templates que facilitaram a implementação de todos os algoritmos. Além disso trouxe consigo um novo algoritmo para a coloração de grafos, o algoritmo sequêncial[23]. Apesar dos resultados práticos estarem bem aquém daqueles obtidos por Mendez Diaz e Zabala[2, 13] e também por Lucet[14], aquele algoritmo revelou-se uma boa surpresa em termos de tempo de processamento de um problema de alta complexidade.

O algoritmo DSATUR foi o primeiro algoritmo que, neste trabalho, foi implementado completamente. Os resultados foram muito bons e a implementação feita foi muito próxima daquela feita por Lucet[14] para calcular o limite superior de χ(G). Em contrapartida, o limite superior de χ(G), calculado por Mendez Diaz e Zabala em [2], foi bem melhor do que a nossa implementação do DSATUR. Apesar de o limite superior ter sido calculado baseado no DSATUR, com certeza existe uma otimização muito eficiente do algoritmo original para que tais resultados sejam possíveis.

Analisando o algoritmo DSATUR originalmente publicado em [5] existe um ponto em que uma otimização pode ser feita. Este ponto é a escolha do vértice a ser colorido. A cada iteração DSATUR escolhe o próximo vértice a ser colorido, baseado no grau de saturação (número de vizinhos coloridos) dos vértices não coloridos. Em caso de empate Brelaz[5] sugere que o desempate seja favorável ao vértice com o maior grau. Mesmo assim, ainda existe espaço para novos empates.

É possível que alguma otimização tenha sido aplicada por Mendez Diaz e Zabala nas regras de desempate mencionadas. Os autores foram contatados para esclarecer este e outros detalhes, mas infelizmente não possível obter uma resposta para confirmar tais suspeitas.

Concluindo, a implementação do algoritmo de Lucet trouxe luz sobre algumas dúvidas a respeito do comportamento do algoritmo e, também sobre como os resultados são produzidos. Um outro resultado positivo desta fase foi o conhecimento adquirido da linguagem C++, da biblioteca Boost e sobre tudo na misteriosa implementação dos templates pela linguagem de programação. Definitivamente, as implementações feitas atuaram como uma fonte esclarecedora de pequenas dúvidas remanecentes, como uma junção das peças de um quebra cabeça.

5 CONCLUSÃO

Após o extensivo estudo da teoria dos grafos e suas aplicações no mundo real, fiquei convencido que o problema da coloração de grafos é um dos mais desafiadores problemas ainda em aberto. Quando digo em aberto, não pretendo desmerecer o brilhante trabalho desenvolvido por dezenas de pesquisadores ao longo de tantos anos.

O sentido de considerar o problema em aberto é explicitar que ainda não encontramos uma solução exata, que seja exeqüível em tempo adequado, para qualquer instância. Naturalmente, devido as características do problema uma solução assim pode não ser encontrada nunca. Entretanto, caso tal solução seja encontrada, é fácil perceber todos os grandes benefícios que ela traria tanto para a teoria dos grafos, como para as suas aplicações práticas.

Neste trabalho não foi possível buscar uma solução nesse sentindo. Entretanto, um valioso conhecimento pode ser extraído do texto apresentado anteriormente. Os algoritmos do Branch and Cut[2] e Planos de Corte[13] foram apresentados. O algoritmo de Lucet[14] foi detalhadamente explicado e também exemplificado utilizando várias figuras, muitas delas criadas para este trabalho. Além do texto produzido no artigo original, ele foi complementado utilizando novos exemplos e também uma explicação mais detalhada em alguns pontos.

Na comparação entre os algoritmos, no capítulo 3, mais uma contribuição foi dada. Os algoritmos Branch and Cut[2], Planos de Corte[13] e Lucet[14] foram comparados utilizando vários quesitos incluindo os métodos para calcularem o limite inferior e superior, pré-processamento, tempo máximo de execução e etc. Os algoritmos também foram comparados utilizando os resultados dos seus testes práticos.

A comparação dos resultados foi uma das principais contribuições deste texto. Apesar de haver no artigo de Lucet[14], uma comparação entre este e o Branch and Cut[2], esta comparação foi muito superficial sem apresentar muitos detalhes. Além disso, como foi dito anteriormente Lucet pode não ter utilizado a versão mais recente do artigo do Branch and Cut[2] para fazer a tal comparação. Como resultado principal da comparação resultados, podemos reconhecer Lucet como um algoritmo mais eficiente que o Branch and Cut. Infelizmente, no artigo dos Planos de Corte[13] não foram reportados resultados suficientes para permitir uma comparação com os demais algoritmos.

A fase da implementação dos algoritmos também proporcionou bons resultados. A utilização da biblioteca Boost[24] trouxe consigo um novo algoritmo para coloração de grafos, o algoritmo sequêncial[23]. Este algoritmo está muito aquém em termos de resultados se comparado com os demais algoritmos apresentados neste trabalho. Entretanto, a velocidade em que coloriu a amostra de 66 instâncias COLOR02[19] foi supreendente. No máximo 2 segundos por instância.

Um outro resultado da fase de implementação foi a comparação da nossa implementação do DSATUR[5] com os resultados reportados nos algoritmos do Branch and Cut[2] e Lucet[14]. Nessa comparação pudemos ver que Lucet implementou algo muito parecido com o algoritmo original, enquanto que Mendez Diaz e Zabala realmente implementaram uma heurística baseada em DSATUR muito eficiente. O início da implementação do algoritmo de Lucet realmente foi decisivo para elucidar as dúvidas restantes a respeito do comportamento algoritmo.

Como sugestões para trabalhos futuros ficam a implementação dos três principais algoritmos aqui apresentados: Planos de Corte[13], Branch and Cut[2] e especialmente o algoritmo de Lucet[14]. Além das implementações é fortemente recomendado que os algoritmos sejam testados contra as 66 instâncias COLOR02 que estão listadas na tabela (3.4) e, naturalmente

os seus resultados comparados. Um outro possível trabalho futuro é, dentro do algoritmo de lucet, a implementação das melhorias sugeridas no capítulo 4.

Concluindo, o tema desta pesquisa realmente foi muito desafiador. Apesar de o problema da coloração de grafos ter sido estudado por vários cientistas[14, 2, 13, 25, 3, 1, 16, 22], ainda existe muito espaço para novas descobertas e pesquisas, como demonstrou Chudnovsky em [25]. O trabalho aqui apresentado, foi mais uma pequena contribuição a tudo o que já foi feito e descoberto. A pesquisa foi muito preciosa para documentar o conhencimento sobre o problema da coloração e, também para documentar o entendimento do problema e alguma das soluções sugeridas para ele. Além disso, ele também trouxe os resultados apresentados nos parágrafos anteriores.

REFERÊNCIAS

[1] Murilo Da Silva. Dissertação de mestrado, Algoritmos para teste de perfeição de grafos. *Universidade Federal do Paraná*, 2004.

[2] I. Mendez Díaz and P. Zabala. A Branch-and-Cut algorithm for Graph Coloring. *Discrete Applied Mathematics*, 154:826–847, 2006.

[3] Wu Qunyan. Register Allocation via Hierarchical Graph Coloring. *Discrete Applied Mathematics*, 1:3–5, 1996.

[4] C. Desrosiers, P. Galinier, and A. Hertz. Efficient algorithms for finding critical subgraphs. *Discrete Applied Mathematics*, 156:244–266, 2008.

[5] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22.4:251–256, 1979.

[6] J. A Bondy and U. S. R Murty. *Graph Theory with Applications*. North-Holland, Nova Iorque, Estados Unidos, 1976.

[7] Timothy Clark and George Smith. US Patent 6023459 Frequency assignment in wireless networks. Disponível em `https://www.patentstorm.us/patents/6023459/fulltext.html`. Acessado em 26/04/2009.

[8] Wikipedia. Mathematics of sudoku. Disponível em `http://en.wikipedia.org/wiki/Mathematics_of_Sudoku`. Acessado em 26/04/2009.

[9] Martin Charles Golumbic and Irith Ben-Arroyo Hartman. *Graph Theory, Combinatorics and Algorithms*. Springer, Nova Iorque, Estados Unidos, 2005.

[10] E.K.Burke, D.G.Elliman, and R. Weare. A University Timetabling System based on Graph Colouring and Constraint Manipulation, 1993. Disponível em `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.5959`. Acessado em 26/04/2009.

[11] Jonathan Gross and Jay Yellen. *Handbook of Graph Theory*. CRC Press, Nova Jersei, Estados Unidos, 2004.

[12] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 156:85–104, 1972.

[13] I. Mendez Díaz and P. Zabala. A cutting plane algorithm for graph coloring. *Discrete Applied Mathematics*, 156:159–179, 2008.

[14] C. Lucet, F. Mendes, and A. Moukrim. An exact method for graph coloring. *Computers & Operations Research*, 33:2189–2207, 2006.

[15] Béla Bollobás. *Modern Graph Theory*. Springer, Nova Iorque, Estados Unidos, 1998.

[16] Walter Klotz. Graph coloring algorithms. *Tech. Rep. Mathematik-Bericht 5, Clausthal University of Technology*, 2002.

[17] Etsuji Tomita and Toshikatsu Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. of Global Optimization*, 37(1):95–111, 2007.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Estados Unidos, 2001.

[19] COLOR02. Color02 computational symposium - dimacs graph coloring instances. Disponível em `http://mat.gsia.cmu.edu/COLOR02/`. Acessado em 26/04/2009.

[20] Wikipedia. Branch and bound. Disponível em `http://en.wikipedia.org/wiki/Branch_and_bound`. Acessado em 26/04/2009.

[21] H. Bodlaender and D. Thilikos. Computing small search numbers in linear time, parameterized and exact computation . 1998. Disponível em `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid= 64B341B2F0BA37514C48E72E6E81F861?doi=10.1.1.3.3507&rep=rep1&type=pdf`. Acessado em 26/04/2009.

[22] Fred Glover, Mark Parker, and Jennifer Ryan. Coloring by tabu branch and bound. *David S. Johnson, Michael A. Trick. Cliques, Coloring, and Satisfiability: Proceedings of the second DIMACS Implementation Challenge.*, pages 285–308, 1996.

[23] Thomas F. Coleman and Jorge J. More. Estimation of sparse Jacobian matrices and graph coloring problems. *Journal of Numerical Analasis*, 20:187–209, 1983.

[24] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library. User Guide and Reference Manual*. Addison-Wesley, Estados Unidos, 2001.

[25] Maria Chudnovsky, Neil Robertson, Paul Seymor, and Robin Thomas. The strong perfect graph theorem. 2002. Disponível em `http://www.math.gatech.edu/~thomas/PAP/spgc.pdf`. Acessado em 31/05/2009.