UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

WLADIMIR ARAÚJO TAVARES

Algoritmos Exatos para Problema da Clique Máxima Ponderada

FORTALEZA
2016

WLADIMIR ARAÚJO TAVARES

Algoritmos Exatos para Problema da Clique Máxima Ponderada

Tese ou Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Departamento de Computação da Universidade Federal do Ceará, como parte dos requisitos necessários para a obtenção do título de Doutor em Computação. Área de concentração: Algoritmos e Otimização.

Orientador: Prof. Dr. Manoel Bezerra Campêlo Neto
Coorientador : Prof. Dr. Philippe Michelon.

FORTALEZA
2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

T233a Tavares, Wladimir Araújo.
Algoritmos Exatos para Problema da Clique Máxima Ponderada / Wladimir Araújo Tavares. – 2016.
182 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2016.
Orientação: Prof. Dr. Manoel Bezerra Campêlo Neto.
Coorientação: Prof. Dr. Philippe Michelon.

1. Algoritmos exatos. 2. Coloração de grafos. 3. Paralelismo de bits. 4. Bonecas Russas. 5. Busca por Resolução. I. Título.

CDD 005

WLADIMIR ARAÚJO TAVARES

ALGORITMOS EXATOS PARA O PROBLEMA DA CLIQUE MÁXIMA PONDERADA.

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito parcial à obtenção do título de Doutor em Ciência da Computação. Área de concentração: Ciência da Computação.

Aprovada em: 06/04/2016.

BANCA EXAMINADORA

_____
Prof. Dr. Manoel Bezerra Campelo Neto (Orientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Philippe Michelon (Orientador - Cotutela)
Université d'Avignon (Univ-Avignon - França)

_____
Prof. Dr. Carlos Diego Rodrigues (Coorientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Thierry Mautor
Université de Versailles Saint-Quentin-en-Yvelines (UVSQ-França)

_____
Prof. Dr. Haroldo Gambini Santos
Universidade Federal de Ouro Preto (UFOP)

_____
Prof. Dr. Ricardo Cordeiro Correa
Universidade Federal Rural do Rio de Janeiro (UFRRJ)

_____
Prof. Dr. Rudini Menezes Sampaio
Universidade Federal do Ceará (UFC)

Dedico este trabalho a todas as pessoas que me apoiaram, em especial minha mãe Joyceleide Pinheiro.

AGRADECIMENTOS

À Deus pela força, perseverança e capacidade dada a mim para a realização deste trabalho.

À minha mãe, Joyceleide Pinheiro, por acreditar em mim e sempre me "empurrar" para que eu chegue mais longe.

Às minhas irmãs, Wadlia e Wanessa, por estarem comigo em todos os momentos.

À minha esposa, Marilena, por trilhar comigo essa jornada.

Ao meu orientador, Prof. Manoel Bezerra, pelo exemplo de pessoa, professor e pesquisador que um algum dia irei alcançar.

Ao meu coorientador, Prof. Philippe Michelon, por toda a sua paciência comigo e pelo exemplo de pessoa resoluta.

Tudo posso naquele que me fortalece.
Filipenses 4:13

RESUMO

Neste trabalho, apresentamos três algoritmos enumerativos para o problema da clique máxima ponderada. Todos eles dependem de uma ordenação inicial dos vértices do grafos. Duas ordens são consideradas, em função dos pesos dos vértices ou dos pesos de seus vizinhos no grafo, levando a duas versões de cada algoritmo. O primeiro algoritmo, denominado **BITCLIQUE**, é um *Branch & Bound* combinatório. Ele combina de forma efetiva adaptações de várias ideias já empregadas com sucesso para resolver o problema, como a ação de uma heurística de coloração ponderada inteira, para definir limites superiores assim como regras de poda e ramificação, e o uso de vetores de bits, como estrutura de dados para simplificar operações sobre o grafo.O algoritmo proposto supera os algoritmos de *Branch & Bound* do estado da arte na maior parte das instâncias analisadas tanto na quantidade de subproblemas enumerados quanto no tempo de computação demandado.

O segundo é um algoritmo de Bonecas Russas, denominado **BITRDS**, que incorpora ao método uma estratégia de poda e ramificação baseada em coloração ponderada. Testes computacionais demonstram que **BITRDS** reduz a quantidade o número de subproblemas quando o tempo de execução quando comparado com o melhor algoritmo de Bonecas Russas para o problema em instâncias aleatórias com densidade superior 50%. Essa diferença cresce à medida que a densidade do grado aumenta. Além disso, **BITRDS** mostra-se competitivo com **BITCLIQUE**, obtendo melhor desempenho em instâncias aleatórias com densidade entre 50% e 80%.

Por último, apresentamos uma cooperação entre o método de Bonecas Russas e o método de Busca por Resolução. O algoritmo proposto, denominado **BITBR**, utiliza tanto a coloração ponderada quanto o limite superior dado pelas bonecas para encontrar um *nogood*. O algoritmo híbrido reduz o número de chamadas à heurística de coloração, chegando até 1 ordem de magnitude, quando comparamos com **BITRDS**. Porém, essa redução diminui o tempo de execução apenas em poucas instâncias.

Diversos experimentos computacionais são realizados com os algoritmos propostos e os principais algoritmos do estado da arte. Resultados computacionais são apresentados para cada algoritmo utilizando as principais instâncias disponíveis na literatura. Finalmente, futuras direções de pesquisas são discutidas.

Palavras-chave: Algoritmos exatos. Coloração de grafos. Paralelismo em nível de bits. Bonecas Russas. Busca por Resolução.

ABSTRACT

In this work, we present three new exact algorithms for the maximum weight clique problem. The three algorithms depend on an initial ordering of the vertices. Two ordering are considered, as a function of the weights of the vertices or the weights of the neighborhoods of the vertices. This leads to two versions of each algorithm. The first onde, called **BITCLIQUE**, is a combinatorial Branch & Bound algorithm. It effectively combines adaptations of several ideas already successfully employed to solve the problem, such as the use of a weighted integer coloring heuristic for pruning and branching, and the use of bitmap for simplifying the operations on the graph. The proposed algorithm outperforms state-of-the-art Branch & Bound algorithms in most instances of the considered in terms of the number of enumerated subproblems as well in terms of computational time

The second one is a Russian Dolls, called **BITRDS**, which incorporates the pruning and branching strategies based on weighted coloring. Computational tests show that **BITRDS** reduces both the number of enumerated subproblems and execution time when compared to the previous state-of-art Russian Dolls algorithm for the problem in random graph instances with density above 50%. As graph density increases, this difference increases. Besides, **BITRDS** is competitive with **BITCLIQUE** with better performance in random graph instances with density between 50% and 80%.

Finally, we present a cooperation between the Russian Dolls method and the Resolution Search method. The proposed algorithm, called **BITBR**, uses both the weighted coloring and upper bounds given by the dolls to find a nogood. The hybrid algorithm reduces the number of coloring heuristic calls, reaching up to 1 order of magnitude when compared with **BITRDS**. However, this reduction decreases the execution time only in a few instances.

Several computational experiments are carried out with the proposed and state-of-the-art algorithms. Computational results are reported for each algorithm using the main instances available in the literature. Finally, future directions of research are discussed.

Keywords: Exact Algorithm. Graph coloring. Bit-Level Parallelism . Russian Dolls. Resolution Search.

RÉSUMÉ

Dans ce travail, nous présentons trois nouveaux algorithmes pour le problème de la clique de poids maximum. Les trois algorithmes dépendent d'un ordre initial des sommets. Deux ordres sont considérés, l'un en fonction de la pondération des sommets et l'autre en fonction de la taille voisinage des sommets. Le premier algorithme, que nous avons appelé BITCLIQUE, est une algorithme de séparation et évaluation. Il réunit efficacement plusieurs idées déjà utilisées avec succès pour résoudre le problème, comme l'utilisation d'une heuristique de coloration pondérée en nombres entiers pour l'évaluation ; et l'utilisation de vecteurs de bits pour simplifier les opérations sur le graphe. L'algorithme proposé surpasse les algorithmes par séparation et évaluation de l'état de l'art sur la plupart des instances considérées en terme de nombre de sous-problèmes énumérés ainsi que en terme de temps d'exécution.

La seconde version est un algorithme des poupées russes, BITRDS, qui intègre une stratégie d'évaluation et de ramification de noeuds basée sur la coloration pondérée. Les simulations montrent que BITRDS réduit à la fois le nombre de sous-problèmes traités et le temps d'exécution par rapport à l'algorithme de l'état de l'art basée sur les poupées russes sur les graphes aléatoires avec une densité supérieure à 50

Enfin, nous présentons une coopération entre la méthode poupées russes et la méthode de "Resolution Search". L'algorithme proposé, appelé BITBR, utilise au même temps la coloration pondérée et les limites supérieures donnés par les poupées pour trouver un "nogood". L'algorithme hybride réduit le nombre d'appels aux heuristiques de coloration pondérée, atteignant jusqu'à 1 ordre de grandeur par rapport à BITRDS.

Plusieurs simulations sont réalisées avec la algorithmes proposés et les algorithmes de l'état de l'art. Les résultats des simulations sont rapportés pour chaque algorithme en utilisant les principaux instances disponibles dans la littérature. Enfin, les orientations futures de la recherche sont discutées.

Keywords: Algorithme Exact. Coloration de graphes. Parallélisme au niveau du bit. Poupées russes. Recherche par résolution.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O problema da clique ponderada máxima (CLIQUE PONDERADA) pode ser apresentado da seguinte maneira: dado um grafo G com peso $w(v)$ associado a cada vértice $v \in V(G)$, encontre uma clique C (ou seja, um subconjunto de vértices adjacentes entre si) tal que a soma dos pesos dos vértices em C seja a maior possível. Este problema é $\mathcal{NP}$-difícil, mesmo quando todos os pesos são iguais Karp (1972). Quando isso acontece, ele é referenciado simplesmente como o problema da clique máxima (CLIQUE).

O problema CLIQUE é um dos mais estudados em otimização combinatória. Além de ser equivalente a outros problemas centrais em teoria dos grafos, como *conjunto independente máximo* (Ver Capítulo 2), ele aparece frequentemente como subestrutura em vários problemas importantes de otimização combinatória.

Em termos práticos, CLIQUE tem um extenso número de aplicações em diversas áreas. A restrição de adjacência, que aparece em CLIQUE, pode representar, por exemplo, a escolha de elementos não-conflitantes de um conjunto (faixas de frequência, horários escolares) ou ainda de elementos que compartilhem um recurso (atividades de um mesmo operador, atores de uma certa comissão). Dentre as aplicações citadas na literatura, encontramos:

- Seleção de projetos Christofides (1975).
- Economia Boginski, Butenko, and Pardalos (2006).
- Teoria de Códigos Brouwer *et al.* (1990); Sloane (1989).
- Bioinformática e Computação biológica Tomita and Seki (2003); Butenko and Wilhelm (2005).
- Visão ComputacionalHotta, Tomita, and Takahashi (2003).
- RobóticaSegundo *et al.* (2010).
- Determinação de vencedores em leilões combinatórios Wu and Hao (2015).
- Redes Sociais McClosky and Hicks (2012); Trukhanov *et al.* (2013); Gschwind *et al.* (2015).

Devido a sua grande importância, existem três levantamentos bibliográficos sobre o problema Pardalos and Xue (1994); Bomze *et al.* (1999); Wu and Hao (2014).

Mais ainda, a versão ponderada de CLIQUE ocorre naturalmente em um grande número de problemas reais. Nesta tese, estamos especificamente interessados em resolver o caso ponderado.

Do ponto de vista algorítmico, vários procedimentos, para resolver CLIQUE PONDERADA de forma exata, já foram desenvolvidos usando técnicas de programação matemática Nemhauser and Trotter (1975); Nemhauser and Sigismondi (1992); Rossi and Smriglio (2001); Rebennack *et al.* (2011); Warrier *et al.* (2005) ou métodos combinatórios Carraghan and Pardalos (1990b); Balas and Xue (1991); Babel (1994); Balas and Xue (1996); Ostergard (1999); Warren and Hicks (2006); Yamaguchi and Masuda (2008);

Kumlander (2008); Shimizu *et al.* (2012); Held, Cook, and Sewell (2012).

Em programação matemática, o estudo do polítopo do conjunto independente (clique em $\overline{G}$) foi iniciado em Padberg (1973). Diversas desigualdades válidas e facetas são conhecidas para este polítopo. Em Rebennack *et al.* (2011), encontramos uma lista de desigualdades válidas e algoritmos especializados para a separação delas. Essas desigualdades válidas e suas rotinas de separação levam ao desenvolvimento de vários algoritmos de *Branch & Cut*. Todavia, o tempo exigido para realizar a rotina de separação e a resolução das relaxações lineares nem sempre justifica a melhoria dos limites superiores proporcionada pela inclusão dos cortes (exceto para grafos com densidade mais alta). Dessa maneira, a maior parte dos algoritmos de *Branch & Cut* são superados por algoritmos de *Branch & Bound* combinatórios.

O algoritmo *Branch & Bound* combinatório mais utilizado para CLIQUE PONDERADA é **CLIQUER**Ostergard (2002). Ele utiliza um método de busca que, mais tarde, ficou conhecido como método das Bonecas RussasVerfaillie, Lemae, and Schiex (1996). Um uso interessante dele acontece na resolução de subproblemas de geração de colunas para um modelo de coloração de grafos Gualandi and Malucelli (2012). Neste caso, ele pode ser usado tanto como heurística para encontrar uma clique ponderada com peso acima de um certo valor ou como método exato para encontrar uma clique ponderada máxima. Porém, sua utilização fica comprometida devido a seu desempenho abaixo do esperado em grafos mais densos (com pelo menos 50% das arestas de um grafo completo).

O algoritmo proposto por Yamaguchi e Masuda, denominado aqui por **YM**Yamaguchi and Masuda (2008), veio para sanar esse problema. Ele utiliza um procedimento de limite superior baseado no caminho mais pesado, no grafo direcionado obtido a partir de uma ordem ρ dos vértices. Testes computacionais, realizados em grafos aleatórios, comprovaram que ele é mais eficiente que **CLIQUER** para grafos com densidade superior a 50%. Sua superioridade cresce à medida que a densidade do grafo aumenta. Além disso, ele também é mais eficiente que o algoritmo proposto em Kumlander (2008), denotado por **DK**.

O algoritmo **DK** é um algoritmo de Boneca Russas que incorpora um limite superior baseado no que nós chamamos de coloração ponderada particionada (Ver Capítulo 3). Yamaguchi e Masuda mostraram que seu procedimento de limite superior (caminho mais pesado) é melhor teoricamente que o utilizado por Kumlander (coloração ponderada particionada).

Em 2012, Held, Cook e Sewell Held, Cook, and Sewell (2012) desenvolveram um algoritmo de Branch & Bound, que identificamos por HCS, adotando as ideias presentes em Balas and Xue (1991); Babel (1994); Warren and Hicks (2006); Sewell (1998). O algoritmo foi utilizado para resolver o subproblema de *pricing* do problema de coloração de grafos. Nesse trabalho, os autores geraram um conjunto de instâncias do *problema do conjunto independente ponderado máximo*, oriundas do subproblema de pricing (algumas

delas ocorreram depois de centenas de iterações do algoritmo de geração de colunas). Elas são denominadas instâncias *EXACTCOLOR*.

O algoritmo **HCS** foi comparado com o algoritmo **CLIQUER** e outros dois algoritmos de *Branch & Cut* dos resolvedores GUROBI 3.0.0 e CPLEX 12.2. A performance de **HCS** foi observada no conjunto de 25 instâncias difíceis de *EXACTCOLOR*. **CLIQUER** foi o mais eficiente para as instâncias com densidade inferior a 50%. Para densidade maior ou igual 97%, os algoritmos de Branch & Cut foram os mais eficientes. Porém, os dois falham nas situações contrárias, ou seja, **CLIQUER** falha em densidades maior ou igual 97% enquanto os algoritmos de Branch & Cut tem um desempenho pobre em grafos esparsos. Diferentemente deles, HCS mostrou-se competitivo em instâncias de todas densidades, obtendo melhor desempenho que os concorrentes nas faixas intermediárias de densidade.

Por outro lado, o algoritmo **HCS** não foi comparado nem teórica ou computacionalmente com outros algoritmos de Branch & Bound da literatura, como **YM**. Mesmo com vários algoritmos disponíveis na literatura, com comprovada eficiência, ainda há espaço para a proposição de outros procedimentos que possa apresentar desempenho ainda melhor. Essa demanda justifica-se pela importância do problema e sua recorrência, assim como a necessidade freqüente de resolução consecutiva de várias instâncias do mesmo, como ocorre no processo de geração de colunas para coloração de grafosMehrotra and Trick (1996); Gualandi and Malucelli (2012); Held, Cook, and Sewell (2012) ou em procedimentos gerais de geração de cortes para problemas de otimização combinatória variadosCorrêa *et al.* (2015). Essa é uma das motivações principais do nosso trabalho.

Apresentamos, nesta tese, novos algoritmos para CLIQUE PONDERADA e avaliamos seus desempenhos frente aos algoritmos do estado da arte para o problema

No Capítulo 4, apresentamos um algoritmo de *Branch & Bound* mais eficiente que **HCS** e **YM**, que combina também as ideias presentes Balas and Xue (1991, 1996); Babel (1994); Warren and Hicks (2006); Sewell (1998) e ainda idéias mais recentes apresentadas em Tomita *et al.* (2010); Segundo *et al.* (2010); Segundo, Rodriguez-Losada, and Jimenez (2011); Corrêa *et al.* (2014), denominado **BITCLIQUE**. Embora a maior parte dos ingredientes usados em nosso algoritmo já esteja disponível na literatura, a forma como combinamos foi capaz de produzir melhor desempenho computacional.

Antes da apresentação de **BITCLIQUE**, que usa coloração ponderada inteira como limite superior empregado na poda durante o processo enumerativo, procuramos justificar, no Capítulo 3, nossa escolha por esse limite em lugar de outros já propostos na literatura. Assim mostramos teoricamente que o limite superior baseado em uma heurística de coloração ponderada inteira é, em potencial, equivalente àquele produzido pelo caminho de maior peso, como utilizado em **YM**. Através de testes computacionais, mostramos que nosso procedimento de limite superior, denominado **BITCOLOR**, apresenta melhor compromisso entre a qualidade do limite superior e o tempo gasto para

calculá-lo do que seus competidores diretos (**YM** e **HCS**). Um dos pontos-chave para a boa eficiência computacional da heurística **BITCOLOR** encontra-se na forma como são efetuadas as operações sobre o grafo, sobretudo cálculo da vizinhança dos vértices. Em nosso algoritmo, implementamos o grafo e as estruturas relacionadas a ele através de vetores de bits (*bitmaps*). Essa estrutura de dados possibilita realizar operações como união e intersecção de conjuntos com menos instruções Segundo *et al.* (2010); Segundo, Rodriguez-Losada, and Jimenez (2011); Segundo *et al.* (2013). Além disso, adotamos uma ordem fixa ρ dos vértices, que será utilizada para organizar os bits nos vetores de bits e usada para selecionar os vértices durante a heurística de coloração. Por isso, nosso algoritmo **BITCLIQUE** é apresentado em duas versões, que variam de acordo com a ordem inicial dos vértices escolhida. Para avaliá-las, realizamos extensos experimentos computacionais. Ainda no Capítulo 4, apresentamos uma comparação dos resultados computacionais entre os algoritmos de Branch & Bound (aqui proposto e da literatura) tendo como base os principais conjuntos de instâncias disponíveis na literatura.

No Capítulo 5, apresentamos a estrutura geral do Algoritmo de Bonecas Russas para CLIQUE PONDERADA, proposto por Östergård em Ostergard (2002). Depois, mostramos nossas intervenções na estrutura geral para incorporar eficientemente a heurística de coloração ponderada como procedimento de limite superior e regra de ramificação. Aqui, utilizamos uma versão modificada da heurística BITCOLOR. Geramos um algoritmo, chamado **BITRDS**, também apresentado em duas versões. Uma comparação computacional é efetuada entre **BITCLIQUE**, **BITRDS** e **CLIQUER**.

No Capítulo 6, apresentamos o método de Busca por Resolução proposto por Chvátal em Chvátal (1997) para problemas gerais em programação inteira 0-1. Em seguida, apresentamos as nossas adaptações ao método para aplicação à CLIQUE PONDERADA. Uma hibridização entre nossos algoritmos de **BITRDS** e Busca por Resolução é, então, proposta. Testes computacionais são executados e uma comparação entre todos os algoritmos propostos é feita nesta tese.

No Capítulo 7, apresentamos uma conclusão revendo as principais contribuições presentes nesta tese. Apontamos também perspectivas para aprimoramentos e extensões do trabalho.

No Apêndice A, discutimos o problema da enumeração de cliques com o peso acima de um limiar. Apresentamos duas variações do algoritmo clássico de **Bron-Kerbosch** para esse problema. Além disso, adaptamos esses algoritmos para resolver o problema da CLIQUE PONDERADA e comparamos tais algoritmos de enumeração "exaustiva" de cliques maximais com o algoritmo **BITRDS1**, utilizando grafos de conflitos gerados a partir de problemas de programação inteira e os grafos gerados aleatoriamente.

No que diz respeito aos resultados desta tese, uma apresentação inicial do algoritmo BITCLIQUE foi publicado nos anais do Simpósio Brasileiro de Pesquisa Operacional (SBPO) em 2015Tavares *et al.* (2015). Uma versão estendida do algoritmo BIT-

CLIQUE foi publicada no ano seguinte nos anais do SBPOTavares *et al.* (2016). Os capítulos 4 e 5 formam um único trabalho, intitulado "Exact algorithms for Maximum Weight Clique Problem" que será submetido a um periódico. Ressaltamos que parte deste trabalho foi realizado durante o período em cotutela internacional com a Université D'Avignon sob a orientação do Prof. Philippe Michelon, pesquisador francês vinculado ao projeto Solving Combinatorial Optimization Problems with Stable Sets Constraints (`http://lia.ufc.br/sticamsud/`) do Programa STIC-AmSud-Capes. Durante um ano e meio, eu estive afastado das minhas atividades docentes da Universidade Federal do Ceará no Campus de Quixadá para a realização desta tese.

2 DEFINIÇÕES BÁSICAS

Neste capítulo, revisamos os principais conceitos de grafos e apresentamos a notação utilizada ao longo do texto. Além disso, exibimos as principais formulações para o problema da clique máxima ponderada e alguns tipos de colorações ponderadas.

2.1 Grafos Simples

Um **grafo** G é par ordenado (V, E) composto por um conjunto finito V, cujos elementos são denominados **vértices**, e por um conjunto $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$, cujos elementos são denominados **arestas**. Para todo grafo G, denotamos $V(G)$ e $E(G)$, respectivamente, os conjuntos de vértices e arestas de G. Designamos por $n = |V(G)|$ e $m = |E(G)|$ a quantidade de vértices e arestas, nessa ordem.

Para cada aresta $e = \{u, v\}$, dizemos que u e v são suas **extremidades**, e que u e v são **vizinhos** ou **adjacentes**. A **vizinhança** de um vértice v, denotado por $N(v)$($N_G(v)$ quando o grafo de referência G precisa ser explícito), é o conjunto de todos os vértices vizinhos de v. O **grau** de um vértice v, denotado por $d(v)$, é a cardinalidade de sua vizinhança.

Um grafo H é dito **subgrafo** de G, representado por $H \subseteq G$, se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Um grafo é dito **subgrafo induzido** por $S \subseteq V(G)$, caracterizado por $G[S]$, se seu conjunto de vértices é S e o seu conjunto de arestas é formado pela arestas $\{i, j\}$ tais que $i \in S$ e $j \in S$. Logo, $G[S] = (S, E \cap (S \times S))$.

O **complemento** de um grafo $G = (V, E)$, denotado por $\overline{G}$, é o par $(V, \overline{E})$, composto pelo conjunto de vértices V e pelo conjunto de arestas

$$\overline{E} = \{\{i, j\} | i, j \in V, i \neq j \text{ e } \{i, j\} \notin E\}$$

Um conjunto de vértices $C \subseteq V$ é uma **clique** se todo par de vértices em C é adjacente entre si. Uma clique C de G é dita **maximal** se não existe uma clique C' de G tal que $C \subset C'$. Uma clique C é dita **máxima** se não existe uma clique C' de G tal que $|C'| > |C|$. O problema CLIQUE consiste em determinar a clique máxima de G. O tamanho da clique máxima G é denotado por $\omega(G)$.

Do ponto de vista teórico, é interessante notar que o problema decisão associado é $\mathcal{NP}$-completo Karp (1972):

ENTRADA: Um grafo $G = (V, E)$ e um inteiro positivo k.

PERGUNTA: Existe um clique C em G tal que $|C| \geq k$?

Do ponto de vista prático, o problema CLIQUE tem um grande número de aplicações práticas em diversas áreas: Seleção de Projetos Christofides (1975), Economia Boginski, Butenko, and Pardalos (2006), Teoria de Códigos Brouwer *et al.* (1990); Sloane (1989), Bioinformática e Computação Biológica Tomita and Seki (2003); Butenko and Wilhelm (2005), Visão ComputacionalHotta, Tomita, and Takahashi (2003), RobóticaSegundo *et al.* (2010), etc.

Devido a sua grande importância, existem três levantamentos bibliográficos sobre o problema Pardalos and Xue (1994); Bomze *et al.* (1999); Wu and Hao (2014).

O problema da clique máxima é estritamente equivalente a outros dois problemas conhecidos de otimização combinatória: problema do conjunto independente máximo e o problema da cobertura de vértices mínima, que serão apresentados a seguir.

Um conjunto de vértices $S \subseteq V$ é um **conjunto independente** de G se todo par de vértices em S não é adjacente entre si em G. Um conjunto independente S de G é dito **maximal** se não existe um conjunto independente S' de G tal que $S \subset S'$. Um conjunto independente S é dito **máximo** se não existe um conjunto independente S' tal que $|S'| > |S|$. O problema INDEPENDENTE consiste em determinar o conjunto independente máximo de G. O tamanho do conjunto independente máximo de G é denotado por $\alpha(G)$.

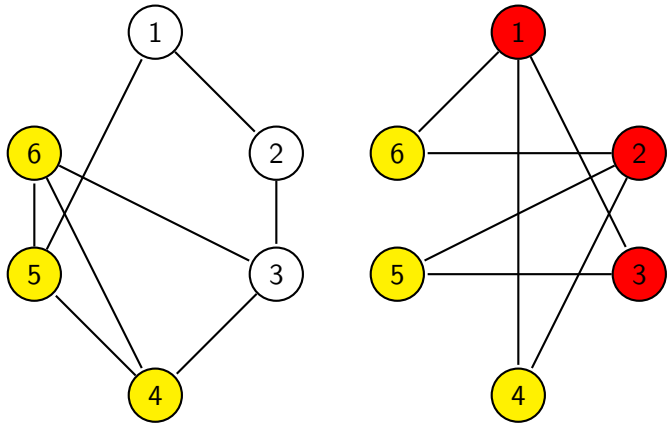
Um conjunto de vértices $K \subseteq V$ é uma **cobertura de vértices** de G se toda aresta $\{i, j\} \in E(G)$ tem, pelo menos, uma extremidade no conjunto K. Dado um grafo G, o problema COBERTURA consiste em determinar uma cobertura de vértices de G com a menor cardinalidade.

A seguinte relação existe entre os problemas apresentados:

- C é uma clique máxima de G se somente se C é um conjunto independente máximo de $\overline{G}$
- C é um conjunto independente máximo de $\overline{G}$ se somente se $V \setminus C$ é uma cobertura mínima de $\overline{G}$

A Figura 1 ilustra a relação entre os três problemas citados.

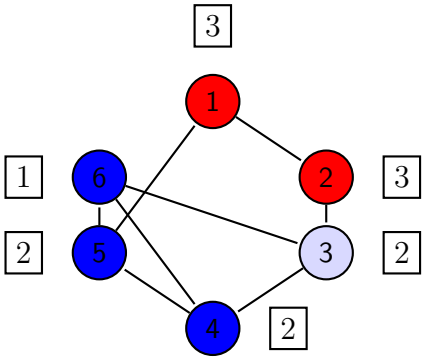
Figura 1 – A relação entre a clique máxima, conjunto independente máximo e cobertura mínima de vértices. O grafo da direita, denotado por $\overline{G}$, é o complemento do grafo da esquerda denotado por G. O conjunto de vértices {4,5,6} define a clique máxima de G e o conjunto independente máximo de $\overline{G}$. O conjunto de vértices {1,2,3} representa a cobertura mínima de $\overline{G}$.



Fonte: Elaborada pelo autor.

Um limite superior para $\omega(G)$ pode ser obtido através de uma partição do conjunto de vértices em conjuntos independentes. Uma **partição** de um conjunto A é

Figura 2 – Relação entre o problema CLIQUE e CLIQUE PONDERADA: o conjunto de vértices {4,5,6} é uma clique máxima de G , mas não é a clique máxima de ponderada de G, que é definida por {1,2}.



Fonte: Elaborada pelo autor.

qualquer coleção $A_1, A_2, \ldots, A_n$ de subconjuntos não vazios de A tal que $\cup_{i=1}^{n} A_i = A$ e $A_i \cap A_j = \emptyset$ para todo $1 \leq i < j \leq n$.

Uma **k-coloração** de um grafo G é uma partição do conjunto de vértices V em conjunto independentes $S_1, \ldots, S_k$. O menor valor k para o qual existe uma k-coloração é chamado de número cromático de G denotado por $\chi(G)$. O número cromático de G é um conhecido limite superior para $\omega(G)$, ou seja,

$$\omega(G) \leq \chi(G)$$

Determinar $\chi(G)$ é um problema $\mathcal{NP}$-difícil, porém encontrar uma coloração, que fornece um limite superior para $\chi(G)$ e, conseqüentemente para $\omega(G)$, pode ser feita por diversos procedimentos polinomiais, comumente chamados heurísticas de coloração.

Um grafo **ponderado**, denotado por (V, E, w) ou (G, w), é composto por um grafo G e uma função de ponderação dos vértices $w : V \to \mathbb{R}^+$. O peso de um vértice v, denotado por $w(v)$, é o peso associado ao vértice v pela função w. O **peso** de um conjunto $S \subseteq V$, denotado $w(S)$, representa a soma dos pesos de todos os vértices de S. O peso da clique com peso máximo de G é denotado por $\omega(G, w)$. O seguinte problema será investigado nesta tese:

Problema 1 *Dado um grafo ponderado $G = (V, E, w)$ com $w : V \to \mathbb{Z}^+$, o problema CLIQUE PONDERADA consiste em determinar uma clique de G com peso máximo.*

A Figura 2 ilustra a diferença entre o problema CLIQUE e CLIQUE PONDERADA.

Uma **coloração ponderada** de um grafo ponderado $G = (V, E, w)$ é um par $(\mathbb{S}, y)$ composto por uma coleção de conjuntos independentes $\mathbb{S} = \{S_1, \ldots, S_k\}$ e uma função de ponderação dos conjuntos independentes $y : \mathbb{S} \to \mathbb{R}^+$ satisfazendo a seguinte propriedade

$$\sum_{S \in \mathbb{S}: v \in S} y(S) \geq w(v) \quad \forall v \in V(G). \tag{1}$$

Em particular, se a imagem da função y é $\mathbb{Z}$ ($\mathbb{Q}$) chamamos a coloração ponderada de inteira (respectivamente, fracionária).

O peso $y(\mathbb{S})$ da coloração ponderada $(\mathbb{S}, y)$ é dado pelo somatório dos pesos dos conjuntos independentes, ou seja,

$$y(\mathbb{S}) = \sum_{S \in \mathbb{S}} y(S) \tag{2}$$

O número cromático ponderado inteiro (fracionário), denotado por $\chi(G, w)$ ($\chi_f(G, w)$), é o peso de uma coloração ponderada inteira (fracionário) mínima. O número cromático ponderado inteiro (fracionário) é um conhecido limite superior para $\omega(G, w)$. Mais precisamente, temos

$$\omega(G, w) \leq \chi_f(G, w) \leq \chi(G, w) \tag{3}$$

2.2 Grafos Direcionados

Um **grafo direcionado** $\overrightarrow{G}$ é um par ordenado (V, A), composto por um conjunto finito V, cujos elementos são denominados vértices, e por um conjunto $A \subseteq \{(u, v) : u, v \in V, u \neq v\}$, cujos elementos são denominados **arcos**. Diferentes das arestas, os arcos são pares ordenados.

Para cada arco (u, v), dizemos que u é a **extremidade inicial** e que v é a **extremidade final**; dizemos ainda que v é um vizinho positivo de u e que u é um vizinho negativo de v.

Para cada vértice v, os conjuntos $N^+(v)$,$N^-(v)$ são, respectivamente, a **vizinhança positiva** de v, definida como $\{u : (v, u) \in A\}$, e a **vizinhança negativa**, definida como $\{u : (u, v) \in A\}$.

Um **passeio orientado** em um grafo direcionado é uma sequência finita de vértices $(v_1, v_2, \ldots, v_k)$ tal que existe um arco (v_i, v_{i+1}) para todo $i = 1, \ldots, k-1$. Se não houver repetição de vértices , o passeio é um **caminho orientado**. Se apenas o primeiro e o último vértices são iguais, o passeio é um **ciclo orientado**. Um grafo direcionado que não possui ciclos orientados é um **grafo direcionado acíclico**.

Um **grafo direcionado ponderado**, denotado por (V, A, w), é composto por um grafo direcionado $\overrightarrow{G}$ e uma função de ponderação $w : V \to \mathbb{R}^+$.

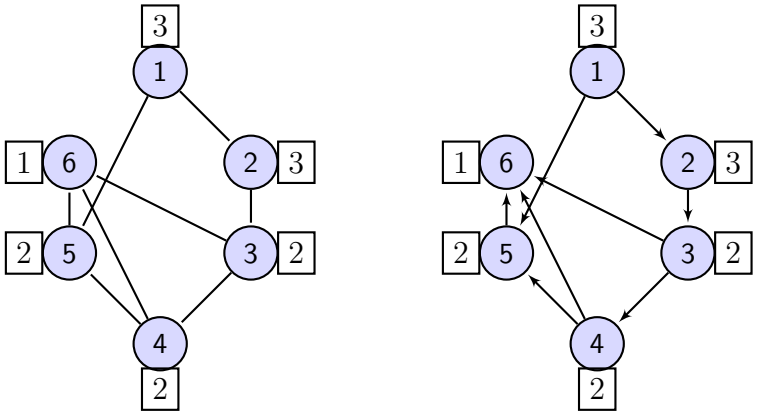
O **peso de um caminho direcionado** P em $\overrightarrow{G}$ é dado pela soma dos pesos de todos os vértices de P.

Dados um grafo $G = (V, E)$ e uma sequência dos vértices $\rho = [\rho_1, \ldots, \rho_{|V|}]$, o **grafo direcionado acíclico induzido por** ρ, denotado $\overrightarrow{G}_\rho$, é obtido trocando cada aresta $\{\rho_i, \rho_j\} \in E (i < j)$ em G por um arco de ρ_i para ρ_j. Logo,

$$A = \{(\rho_i, \rho_j) : \{\rho_i, \rho_j\} \in E, i < j\}$$

A Figura 3 ilustra a obtenção do grafo direcionado acíclico induzido por uma sequência de vértices ρ.

Figura 3 – O grafo da esquerda é um grafo ponderado utilizado para obter o grafo direcionado acíclico induzido pela sequência (1,2,3,4,5,6) (grafo da direita).



Fonte: Elaborada pelo autor.

Dado um grafo direcionado acíclico $\overrightarrow{G}$ e uma vértice $v \in V$, o **peso do caminho com peso máximo terminado em** v é denotado por $\ell(\overrightarrow{G}, v)$. O **peso do caminho com peso máximo** de $\overrightarrow{G}$,denotado $\ell(\overrightarrow{G})$, e portanto $\max_{v \in V} \ell(\overrightarrow{G}, v)$.

Em Yamaguchi and Masuda (2008), o seguinte teorema é apresentado, relacionando o caminho com peso máximo em $\overrightarrow{G}_\rho$, dada uma sequência ρ dos vértices, e o peso da clique ponderada máxima em G .

Teorema 1 *Yamaguchi and Masuda (2008) Para um grafo ponderado* $G = (V, A, w)$ *e uma seqüencia* ρ, $\omega(G, w) \leq \ell(\overrightarrow{G}_\rho)$

Prova Seja K uma clique máxima em G. Para qualquer sequência ρ, $\overrightarrow{G}_\rho$ tem um caminho que contém todos os vértices em K. Logo, $\omega(G) = w(K) \leq \ell(\overrightarrow{G}_\rho)$.

Dado um grafo ponderado e uma sequência ρ dos vértices, o Algoritmo 2.1 calcula $\ell(v, \overrightarrow{G}_\rho)$ para todo vértice v.

Algorithm 2.1 O algoritmo CaminhoMaisPesado recebe um grafo G e uma sequência ρ, e devolve o caminho mais pesado no grafo $\overrightarrow{G}_\rho$ terminando em cada vértice

```
1: function CaminhoMaisPesado(G, w, ρ, ℓ)
2:     for v ∈ V do
3:         ℓ(v) ← w(v)
4:     for i ← 1 até n do
5:         for j ← i + 1 até n do
6:             if (ρ_i, ρ_j) ∈ E(G⃗_ρ) then
7:                 ℓ(ρ_j) ← max{ℓ(ρ_j), ℓ(ρ_i) + w(ρ_j)}
```

2.3 Formulações

A formulação mais simples para o problema da clique máxima ponderada é a formulação por aresta. Nesta formulação, uma variável $x_i \in \{0,1\}$ é associada a cada vértice i. Se $x_i = 1$ então o vértice i está na clique; caso contrário, o vértice i não está.

$$\text{maximize} \sum_{i=1}^{n} w_i x_i \tag{4}$$

$$\text{s.a. } x_i + x_j \leq 1, \quad \forall \{i,j\} \in \overline{E} \tag{5}$$

$$x_i \in \{0,1\} \qquad i \in V \tag{6}$$

A restrição (5) garante que, no máximo, um vértice pode ser escolhido em cada não aresta. A restrição (6) garante que todas as variáveis são binárias.

Um estudo desta formulação foi conduzido por Nemhauser e Trotter Nemhauser and Trotter (1975). Eles mostraram que, se $x = (x_1, x_2, \ldots, x_n)$ é um ponto extremo da região viável da relaxação linear de (4), (5) e (6), então cada $x_i \in \{0, 1, \frac{1}{2}\}$. Além disso, se uma variável $x_i = 1$ na solução ótima da relaxação linear da formulação, então $x_i = 1$ em pelo menos uma solução ótima da formulação inteira.

Note que esta propriedade pode ser aproveitada em um algoritmo enumerativo. Contudo, na maior parte dos casos, poucas variáveis assumem um valor 1 na solução ótima da relaxação linear. A diferença entre o valor ótimo do problema inteiro e de sua relaxação linear é bastante grande. Isso representa uma séria restrição da utilização dessa formulação em um algoritmo de Branch & Bound.

Uma formulação alternativa considera uma restrição para cada conjunto independente maximal de G. A variável x_i é definida como anteriormente.

$$\text{maximize} \sum_{i=1}^{n} w_i x_i \tag{7}$$

$$\text{subject to} \sum_{\{S : i \in S\}} x_i \leq 1, \quad \forall S \in \mathcal{S} \tag{8}$$

$$x_i \in \{0,1\} \qquad i \in V \tag{9}$$

onde $\mathcal{S}$ é o conjunto de todos os conjuntos independentes maximais de G.

A restrição (8) garante que podemos escolher no máximo um vértice em cada conjunto independente de G.

Embora a diferença esperada entre a solução ótima dessa formulação e sua relaxação linear seja menor que da formulação por arestas, resolver o problema relaxado é NP-difícil em grafos arbitrários. Porém, qualquer solução viável para seu dual é um limite superior para $\omega(G, w)$.

O dual da formulação por conjuntos independentes maximais é definido da

seguinte maneira:

$$\min \sum_{S \in \mathcal{S}} y_S \tag{10}$$

$$\text{s. a} \sum_{\{S: i \in S\}} y_S \geq w(i), \quad \forall i = 1, \ldots, n \tag{11}$$

$$y_S \geq 0 \qquad\qquad S \in \mathcal{S} \tag{12}$$

onde y_S é o peso do conjunto independente S.

Note que as desigualdades do problema dual são exatamente aquelas que definem uma coloração ponderada. Logo, este problema determina uma coloração ponderada mínima, que fornece um limite superior para $\omega(G, w)$. Na verdade, a coloração ponderada do grafo G também gera limites superiores para as cliques ponderadas de cada subgrafo, como segue:

Proposição 1 *Seja* $(\mathbb{S}, y)$ *uma coloração ponderada para um grafo ponderado* (G, w). *Dado* $V' \subseteq V$. *Seja* $\mathbb{S}[V'] = \{S \in \mathbb{S} : S \cap V' \neq \emptyset\}$. *Então,*

$$y(\mathbb{S}[V']) \geq \omega(G[V'], w|V')$$

onde $w|V'$ *é a função* w *restrita ao conjunto* V'.

Esse limite superior pode ser obtido através de heurísticas de coloração ponderada. Na próxima seção, apresentaremos alguns tipos de colorações ponderada.

2.4 Colorações Ponderadas

Uma coloração ponderada pode ser entendida como uma solução viável para o dual da formulação por conjunto independentes. Uma solução viável pode ser obtida de diversas maneiras. Para facilitar a referência atribuímos uma denominação diferente à cada coloração ponderada, de acordo com a forma usada para obter a solução viável correspondente. A seguir, apresentamos os seguintes tipos de coloração ponderada utilizadas para gerar limites superiores para CLIQUE PONDERADA:

- Coloração Ponderada Trivial.
- Coloração Ponderada Particionada.
- Coloração Ponderada Inteira.

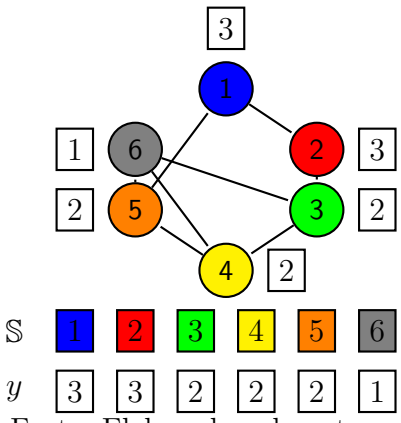
2.4.1 Coloração Ponderada Trivial

Uma **coloração ponderada trivial** de G é uma coleção de conjuntos independentes $\mathbb{S} = \{S_1, \ldots, S_k\}$ com uma função de peso $y : \mathbb{S} \to \mathbb{Z}^+$ tal que a cardinalidade de cada conjunto independente é 1 e o seu peso é igual ao peso do vértice que o define.

A Figura 4 ilustra uma coloração ponderada trivial.

Uma coloração ponderada trivial pode ser obtida através do seguinte algoritmo:

Figura 4 – Coloração ponderada trivial: o peso da coloração ponderada trivial de G é 13. A clique máxima ponderada de G é 6.



Fonte: Elaborada pelo autor.

Algorithm 2.2 Heurística de Coloração Ponderada Trivial

function ColoraçãoTrivial(G, w, ρ)
  $i \leftarrow 1$
  $U \leftarrow V$
  while $U \neq \emptyset$ **do**
    Seja v o primeiro vértice em U seguindo a ordem ρ
    $S_i \leftarrow \{v\}$
    $y(S_i) \leftarrow w(v)$
    $U \leftarrow U \setminus \{v\}$
    $i \leftarrow i + 1$
  return $\sum_{k=1}^{i-1} y(S_i)$

Observe que o peso da coloração ponderada trivial obtida pela heurística não depende da ordem em que os vértices são escolhidos durante a sua execução.

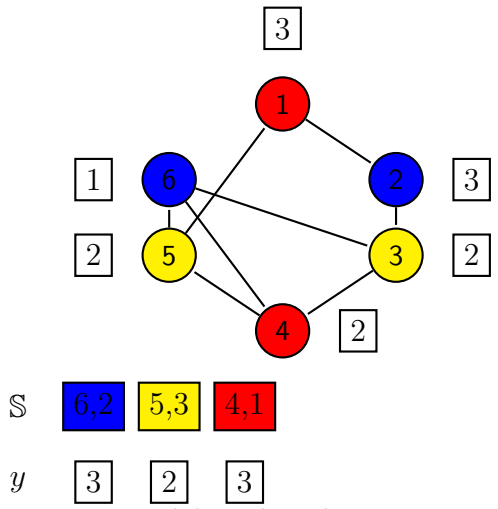
O custo computacional para obter essa coloração pode ser considerado insignificante, porém a qualidade do limite superior é baixa. Apesar disso, essa coloração foi utilizada nos algoritmos *Branch & Bound* para CLIQUE PONDERADA, proposto em Carraghan and Pardalos (1990a); Ostergard (1999), obtendo bons resultados em grafos esparsos.

2.4.2 Coloração Ponderada Particionada

Uma **coloração ponderada particionada** de G é composta por uma partição $\mathbb{S} = \{S_1, \ldots, S_k\}$ dos vértices V e uma função de peso $y : \mathbb{S} \to \mathbb{Z}^+$. O peso de cada conjunto independente é dado pelo maior peso dos vértices que ele contém.

A Figura 5 ilustra uma coloração ponderada particionada obtida seguindo a ordem (6,5,4,3,2,1).

Figura 5 – A coloração ponderada particionada de G mostrada na figura tem peso 8 e a clique máxima ponderada de G é 6.



Fonte: Elaborada pelo autor.

Uma **coloração ponderada particionada** pode ser obtida através do seguinte algoritmo:

Algorithm 2.3 Heurística de Coloração Ponderada Particionada

function ColoraçãoParticionada(G, w, ρ)
  $i \leftarrow 1$
  $U \leftarrow V$
  while $U \neq \emptyset$ **do**
    Encontre um conjunto independente maximal S_i em $G[U]$ seguindo a ordem ρ.
    $y(S_i) \leftarrow max_{v \in S_i} w(v)$
    $U \leftarrow U \setminus \{S_i\}$
    $i \leftarrow i + 1$

Avenali (2007) utiliza uma versão fracionária da coloração particionada para obter limites superiores em seu algoritmo de *Resolution Branch & Bound*.

Kumlander (2008) usa uma coloração ponderada particionada em seu algoritmo de *Branch & Bound* para CLIQUE PONDERADA. O autor usa a ordem decrescente de pesos dos vértices para definir a sequência dos vértices ρ. Essa coloração ponderada inteira é realizada, uma única vez, no nó raiz do *Branch & Bound* e utilizada para determinar os limites superiores, em cada nó, da árvore de Branch & Bound.

Em Fang *et al.* (2014), uma coloração ponderada particionada é utilizada para codificar o problema da clique máxima ponderada em um problema de satisfabilidade máxima (MaxSat) ponderada. Técnicas de resolvedores de MaxSat são utilizadas para encontrar um limite superior mais apertado.

Em geral, os algoritmos que utilizam esse tipo de coloração ponderada conseguem obter bons resultados para grafos mais densos.

2.4.3 Coloração Ponderada Inteira

Uma coloração ponderada inteira de G é um conjunto de conjuntos independentes $\mathbb{S} = \{S_1, \ldots, S_k\}$ juntamente com seus pesos inteiros positivos tais que $\sum_{S \in \mathbb{S}: v \in S} y_S = w(v)$, para todo $v \in V(G)$. O peso da coloração ponderada inteira é dado pela soma dos pesos dos conjuntos independentes.

Uma coloração ponderada inteira pode ser obtida pelo seguinte processo iterativo: Para cada vértice v, uma variável $res(v)$ representa o peso residual de v, ou seja, a diferença entre $w(v)$ e soma dos conjuntos independentes que contém v escolhido até agora (Veja equação (13)). Assim, O peso residual de um vértice v é inicializado com o valor de $w(v)$.

$$res(v) = w(v) - \sum_{S: v \in S} y(S) \tag{13}$$

Em cada iteração, encontra-se uma conjunto independente maximal S usando vértices ainda não totalmente coloridos, ou seja, com o peso residual maior que zero. O peso do conjunto independente S será dado pelo menor peso residual de seus vértices. O peso residual de S é, então, decrementado do peso de S. O processo de coloração continua enquanto existirem vértices com peso residual maior que zero. Os passos anteriores podem ser vistos no Algoritmo 2.4:

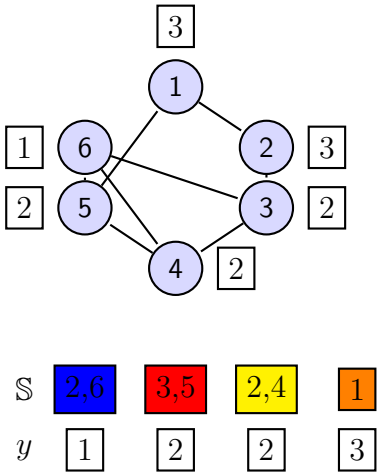
Algorithm 2.4 Heurística de Coloração Ponderada Inteira

```
function ColoraçãoInteira(G, w, ρ)
    for v ∈ V do
        res(v) ← w(v)
    i ← 0
    U ← V
    while U ≠ ∅ do
        i ← i + 1
        Seja u o primeiro vértice de U seguindo ρ
        Encontre um conjunto independente maximal S_i em G[U] contendo o vértice u
seguindo a ordem π
        y(S_i) ← min{res(v)|v ∈ S_i}
        for v ∈ S_i do
            res(v) ← res(v) − y(S_i)
            if res(v) = 0 then
                U ← U \ {v}
```

A heurística de coloração ponderada inteira é utilizada como limite superior nos seguintes algoritmos Balas and Xue (1991),Babel (1994),Balas and Xue (1996),Warren and Hicks (2006),Held, Cook, and Sewell (2012) e Tavares *et al.* (2015).

A Figura 6 ilustra uma coloração ponderada inteira.

Figura 6 – O peso da coloração ponderada inteira de G é 8. A clique máxima ponderada para G é 6.



Fonte: Elaborada pelo autor.

2.5 Importância da ordem inicial

Neste momento, espera-se que o leitor entenda que o peso da coloração ponderada inteira é bastante influenciado pela ordem dos vértices ρ utilizada. Esse critério determina os conjuntos independentes são gerados. Consequentemente, a escolha da ordem inicial ρ afeta o desempenho do algoritmo que usa a coloração ponderada para definir o procedimento de poda. Mais recentemente, em Tomita *et al.* (2010), os autores reportaram resultados melhores que os anteriores com a adoção de uma ordem fixa dos vértices para a heurística de coloração para CLIQUE. Em Segundo *et al.* (2010), esse resultado é ainda mais potencializado pela utilização dos vetores de bits para CLIQUE.

Uma segunda ordem que merece destaque é aquela em que os vértices são completamente coloridos, que chamamos de ordem de coloração, denotada por π. Se $S_1, \ldots, S_k$ é a sequência de conjuntos independentes gerados, a ordem π é tal que $\pi(u) < \pi(v)$ sempre que $\max\{k|u \in S_k\} < \max\{k|v \in S_k\}$; no caso de empate, u vem antes de v na ordem ρ. Note que a ordem π depende de ρ e da política de atribuição de pesos aos conjuntos independentes.

Esta ordenação terá influência no cálculo dos limites superiores para a clique máxima ponderada no subgrafos. Por exemplo, o limite superior para $G[\pi_1, \ldots, \pi_j]$, conforme a Proposição 1 é

$$\sum_{i=1}^{\max\{k|\pi_j \in S_k\}} S_i$$

No próximo capítulo, mostraremos que, potencialmente, o peso da coloração ponderada inteira que escolhe os vértices, seguindo a ordem ρ, é igual ao peso do caminho mais pesado no grafo ponderado direcionado $\overrightarrow{G}_\rho$. Apresentaremos uma heurística de coloração ponderada inteira que encontra resultados melhores que as principais heurísticas encontradas na literatura. A nossa heurística especifica uma ordem fixa ρ para a escolha

dos vértices durante o processo e utiliza uma estrutura de dados chamada vetores de bits para acelerar o processo de determinação dos conjuntos independentes. Essas duas características combinadas definem uma heurística de coloração ponderada que fornecem um limite superior de qualidade com custo computacional baixo.

3 LIMITE SUPERIOR

Durante um algoritmo de enumeração implícita para um problema de maximização, a maneira mais comum de podar um subproblema é calcular um limite superior e verificar que ele é menor que a melhor solução conhecida. De maneira geral, um limite superior é computado a partir de uma relaxação do problema. Para o problema CLIQUE PONDERADA, os dois procedimentos principais de limite superior para a clique máxima ponderada são:

- **heurística de coloração ponderada**
- **heurística para encontrar o caminho com peso máximo**

Os procedimentos de limites superiores podem variar quanto à qualidade do limite superior e ao tempo gasto para obtê-lo. Uma boa estratégia é procurar o melhor compromisso entre a qualidade do limite superior e o tempo gasto para calculá-lo em cada subproblema.

Na seção 3.1, fazemos uma comparação teórica entre o limite superior dado por uma coloração ponderada inteira e o limite superior dado pelo caminho com peso máximo no grafo ponderado direcionado. Mostramos que os dois limites são potencialmente equivalentes.

Na seção 3.2, apresentamos um procedimento de limite superior baseado no caminho direcionado com o maior peso Yamaguchi and Masuda (2008). Esse limite superior é utilizado em um algoritmo de *Branch & Bound*, obtendo bons resultados computacionais em grafos mais densos

Na seção 3.3, apresentamos a heurística proposta em Held, Cook, and Sewell (2012). Nessa heurística, a ordem em que os vértices são escolhidos durante a coloração ponderada é definida dinamicamente usando os pesos residuais.

Na seção 3.4, apresentamos uma heurística de coloração proposta em Tavares *et al.* (2015). Essa heurística utiliza uma ordem inicial dos vértices fixa para gerar os conjuntos independentes durante a heurística de coloração. Nós propomos dois esquemas de ordenação, que serão testados computacionalmente.

3.1 Comparação entre limites superiores

Primeiramente, mostramos que dada uma sequência dos vértices ρ utilizada para induzir o grafo direcionado $\overrightarrow{G}_\rho$, podemos encontrar uma coloração ponderada inteira $(\mathbb{S}, y)$, utilizando o algoritmo 2.4, tal que $y(\mathbb{S}) \leq \ell(\overrightarrow{G}_\rho)$. O seguinte lema estabelece esse resultado:

Lema 1 *Seja (G, w) um grafo ponderado. Para qualquer seqüencia dos vértices ρ, existe uma coloração ponderada inteira $(\mathbb{S}, y)$ de (G, w) tal que $y(\mathbb{S}) \leq \ell(\overrightarrow{G}_\rho)$.*

Prova Considere a coloração ponderada inteira $(\mathbb{S}, y)$, $\mathbb{S} = \{S_1, \ldots, S_k\}$, construída pelo o Algoritmo 2.4, usando como entrada o grafo ponderado (G, w) e a seqüencia ρ . Vamos

mostrar que existe um caminho direcionado P em $\overrightarrow{G}_\rho$ que contém pelo menos um vértice de cada S_i. O caminho será construído iterativamente considerando os conjuntos independentes na ordem reversa, ou seja, $S_k, S_{k-1}, \ldots, S_1$. Em S_k, escolhemos arbitrariamente um vértice v, para ser o último vértice de P. Então para cada $i = k-1, \ldots, 1$, seja $P = vP'$ o caminho direcionado formado após considerar os conjuntos independentes $S_k, \ldots, S_{i+1}$, temos duas possibilidades:

1. **Caso** $P \cap S_i \neq \emptyset$, não escolhemos nenhum vértice de S_i
2. **Caso** $P \cap S_i = \emptyset$, estendemos P para uvP', escolhendo um vértice $u \in S_i$ tal que $(u, v) \in E(\overrightarrow{G}_\rho)$ tal vértice existe porque S_i é maximal e foi formado seguindo a sequência ρ

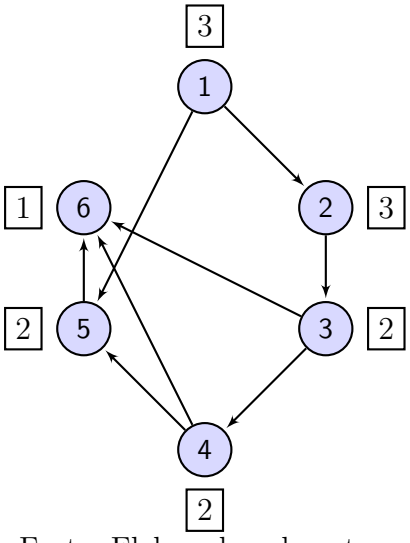
No final, temos que

$$y(\mathbb{S}) = \sum_{i}^{k} y(S_i) \leq \sum_{v \in P} \sum_{S \in mathbbS : v \in S} y(S) = \sum_{v \in P} w(v) \leq \ell(\overrightarrow{G}_\rho). \tag{14}$$

Os exemplos 1 e 2 ilustram o lema 1.

Exemplo 1 *Considere o seguinte DAG obtido pela seqüência* $\rho = (1, 2, 3, 4, 5, 6)$*:*

Figura 7 – Grafo ponderado direcionado $\overrightarrow{G}_\rho$, onde $\rho = (1, 2, 3, 4, 5, 6)$



Fonte: Elaborada pelo autor.

O Algoritmo 2.4 devolve a seguinte coloração ponderada inteira $(\mathbb{S}, y)$*, onde* $y(\mathbb{S}) = 7$*:*

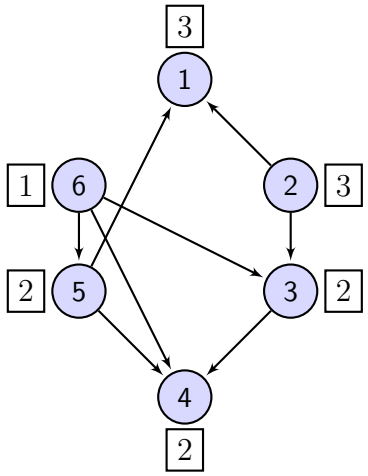
- $S_1 = \{1, 3\}$,$y(S_1) = 2$
- $S_2 = \{1, 4\}$,$y(S_2) = 1$
- $S_3 = \{2, 4\}$,$y(S_3) = 1$
- $S_4 = \{2, 5\}$,$y(S_4) = 2$
- $S_5 = \{6\}$,$y(S_5) = 1$

Podemos construir o seguinte caminho direcionado P. Na classe de cor S_5, escolhemos o vértice 6. Na classe de cor S_4, o vértice escolhido será o vértice 5 porque ele é adjacente ao vértice 6. Na classe de cor S_3, o vértice escolhido será o vértice 4 porque ele é adjacente ao vértice 5. Na classe de cor S_2, nenhum vértice será escolhido porque o vértice 4 já foi escolhido. Na classe de cor S_1, o vértice escolhido será o vértice 3 porque ele é adjacente ao vértice 4. O caminho direcionado P será (3,4,5,6). O peso do caminho P também é 7. Lembrando que P não é o maior caminho direcionado em $\overrightarrow{G}_\rho$. O caminho direcionado de peso máximo é (1,2,3,4,5,6) e seu peso é 13. Logo,

$$7 = y(\mathbb{S}) \leq \sum_{v \in P} w(v) = 7 \leq \ell(\overrightarrow{G}_\rho) = 13$$

Exemplo 2 *Considere o seguinte DAG obtido pela sequência $\rho = (6, 2, 5, 3, 4, 1)$:*

Figura 8 – Grafo ponderado direcionado $\overrightarrow{G}_\rho$, onde $\rho = (6, 2, 5, 3, 4, 1)$



Fonte: Elaborada pelo autor.

O algoritmo 2.4 devolve a seguinte coloração ponderada inteira $(\mathbb{S}, y)$, onde $y(\mathbb{S}) = 7$:

- $S_1 = \{6, 2\}$,$y(S_1) = 1$
- $S_2 = \{2, 5\}$,$y(S_2) = 2$
- $S_3 = \{3, 1\}$,$y(S_3) = 2$
- $S_4 = \{4, 1\}$,$y(S_4) = 1$
- $S_5 = \{4\}$,$y(S_5) = 1$

Podemos construir o seguinte caminho direcionado P: Na classe de cor S_5, escolhemos o vértice 4. Na classe de cor S_4, nenhum vértice escolhido será escolhido porque o vértice 4 já foi escolhido. Na classe de cor S_3, o vértice escolhido será o vértice 3 porque ele é adjacente ao vértice 4. Na classe de cor S_2, o vértice escolhido será o vértice 2 porque ele é adjacente ao vértice 3. Na classe de cor S_1, nenhum vértice será escolhido porque o vértice 2 já foi escolhido. O caminho direcionado P será (2,3,4). O comprimento do caminho P também é 7. Neste exemplo, P é o maior caminho direcionado em $\overrightarrow{G}_\rho$.

Logo,

$$7 = y(\mathbb{S}) \leq \sum_{v \in P} w(v) = 7 = \ell(\overrightarrow{G}_\rho)$$

No lema seguinte, mostramos que dada uma coloração ponderada inteira $(\mathbb{S}, y)$, podemos encontrar uma ordem dos vértices ρ tal que o peso máximo de um caminho direcionado em $\overrightarrow{G}_\rho$ seja menor que o peso da coloração ponderada inteira $(\mathbb{S}, y)$.

Lema 2 *Seja (G, w) um grafo ponderado. Dada uma coloração ponderada inteira $(\mathbb{S}, y)$ de (G, w), existe uma sequência dos vértices ρ tal que*

$$\ell(\overrightarrow{G}_\rho) \leq y(\mathbb{S}) \tag{15}$$

Prova Como $\bigcup_i S_i = V$, podemos construir uma partição do conjunto dos vértices $\{S'_1, S'_2, \ldots, S'_k\}$ tal que

$$S'_1 = S_1 S'_{i+1} = S_{i+1} \setminus \cup_{l=1}^{i} S'_l, \text{ para } i \geq 1 \tag{16}$$

Defina ρ seguindo $S'_1, S'_2, \ldots, S'_k$, ou seja, os vértices em S'_i vem antes daqueles em S'_{i+1} (dentro do mesma parte, a ordem é arbitrária).

Seja P um caminho de peso máximo em $\overrightarrow{G}_\rho$. Note que P não contém dois vértices de um mesmo S_i. Do contrário, se u e v são dois vértices distintos de P em S_i, todo o subcaminho de P entre u e v também estaria em S_i, um absurdo pois S_i é um conjunto independente. Logo,

$$\sum_{v \in P} \sum_{i: v \in S_i} y(S_i) \leq \sum_i y(S_i)$$

Por outro lado, na coloração ponderada inteira temos que

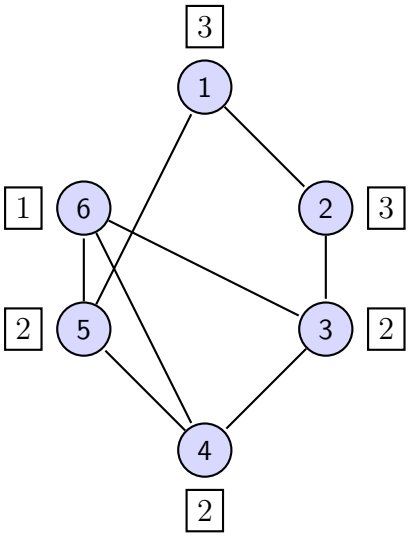
$$\sum_{i: v \in S_i} y(S_i) = w(v)$$

Logo,

$$\ell(\overrightarrow{G}_\rho) = \sum_{v \in P} w(v) \leq \sum_i y(S_i)$$

Os exemplos 3 e 4 ilustram o lema 2.

Exemplo 3 *Considere o seguinte grafo ponderado:*



Considere a seguinte coloração ponderada inteira $(\mathbb{S}, y)$*, onde* $y(\mathbb{S}) = 7$*:*

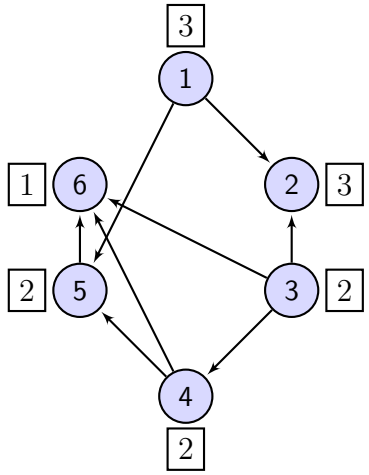
- $S_1 = \{1, 3\}$,$y(S_1) = 2$
- $S_2 = \{1, 4\}$,$y(S_2) = 1$
- $S_3 = \{2, 4\}$,$y(S_3) = 1$
- $S_4 = \{2, 5\}$,$y(S_4) = 2$
- $S_5 = \{6\}$,$y(S_5) = 1$

Defina a seguinte partição de V:

- $S'_1 = \{1, 3\}$
- $S'_2 = \{4\}$
- $S'_3 = \{2\}$
- $S'_4 = \{5\}$
- $S'_5 = \{6\}$

A partir da partição, podemos definir a seguinte seqüência $\rho = (1, 3, 4, 2, 5, 6)$*. O grafo direcionado induzido por* ρ *é:*

Figura 9 – Grafo ponderado direcionado $\overrightarrow{G}_\rho$, onde $\rho = (1, 3, 4, 2, 5, 6)$



Fonte: Elaborada pelo autor.

O caminho mais pesado no grafo direcionado induzido por ρ é (3,4,5,6), cujo peso é 7.

Exemplo 4 *Considere o grafo ponderado do exemplo 3. Considere a seguinte coloração ponderada inteira $(\mathbb{S}, y)$:*

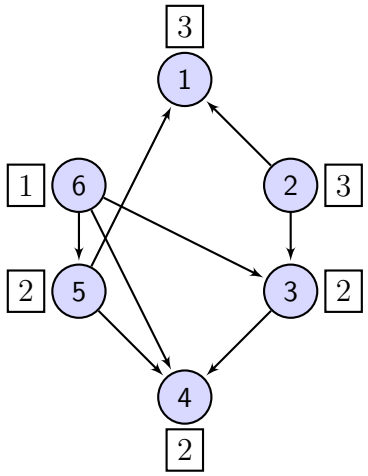
- $S_1 = \{6, 2\}$,$y(S_1) = 1$
- $S_2 = \{2, 5\}$,$y(S_2) = 2$
- $S_3 = \{3, 1\}$,$y(S_3) = 2$
- $S_4 = \{4, 1\}$,$y(S_4) = 1$
- $S_5 = \{4\}$,$y(S_5) = 1$

Defina a seguinte partição de V:

- $S'_1 = \{2, 6\}$
- $S'_2 = \{5\}$
- $S'_3 = \{1, 3\}$
- $S'_4 = \{4\}$
- $S'_5 = \{\}$

A partir da partição, podemos definir a seguinte sequencia $\rho = (2, 6, 5, 1, 3, 4)$. O grafo direcionado induzido por ρ é:

Figura 10 – Grafo ponderado direcionado $\overrightarrow{G}_\rho$, onde $\rho = (2, 6, 5, 1, 3, 4)$



Fonte: Elaborada pelo autor.

O caminho mais pesado no grafo direcionado induzido por ρ é (2,3,4), cujo peso é 7.

3.2 Heurística de Yamaguchi e Masuda

Em Yamaguchi and Masuda (2008), é apresentado um algoritmo polinomial que encontra uma seqüência dos vértices π utilizada para definir o grafo direcionado $\overrightarrow{G}_\pi$. O algoritmo apresentado é uma variação do algoritmo de caminho mínimo proposto por Djikstra (Ver Algoritmo 3.1).

Durante a execução do algoritmo, as seguintes variáveis são atualizadas: o conjuntos dos vértices que não estão na seqüência, representado por S; a sequência de vértices π que induz o grafo direcionado; e $\ell(v)$, representando uma estimativa do peso do caminho até o vértice v no grafo direcionado. Inicialmente, π começa vazia, S começa com todos os vértices e $\ell(v)$ recebe o peso do vértice v.

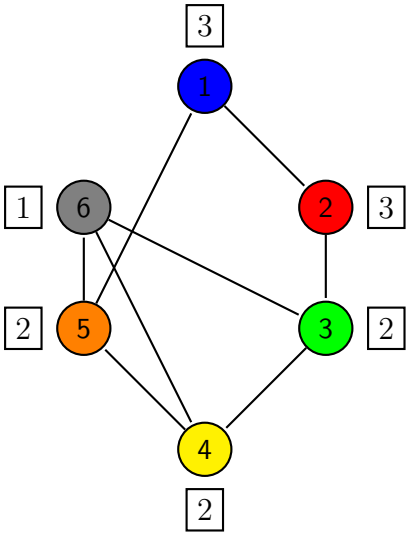
Em cada iteração, o vértice com o menor estimativa do maior caminho é escolhido. Então, ele é adicionado à sequência π, e o peso do caminho dos vértices adjacentes ao vértice v são atualizados. O algoritmo devolve uma ordem dos vértices $\pi_1, \pi_2, \ldots, \pi_n$ tal que $\ell(\pi_1) \leq \ell(\pi_2) \leq \ldots \leq \ell(\pi_n)$, onde $\ell(\pi_i)$ é o peso do caminho mais pesado no subgrafo $\overrightarrow{G}_\pi[\pi_1, \ldots, \pi_{i-1}]$. A complexidade do algoritmo utilizado é $O(|V|^2)$. Observe que neste algoritmo não é dada a priori nenhuma ordem inicial dos vértices, sendo construída iterativamente a sequência π que induz a orientação do grafo.

Algorithm 3.1 Heurística proposta por Yamaguchi e Masuda

```
1: function CaminhoMaisPesado(G = (V, E), w, π, ℓ)
2:     for v ∈ V do
3:         ℓ(v) ← w(v)
4:     S ← V
5:     i ← 1
6:     while S ≠ ∅ do
7:         Encontre o vértice v de S com o menor valor de ℓ().
8:         S ← S \ {v}
9:         π(i) ← v
10:        for u ∈ N(v) ∩ S do
11:            if ℓ(v) + w(u) > ℓ(u) then
12:                ℓ(u) ← ℓ(v) + w(u)
13:        i ← i + 1
```

Exemplo 5 *Considere o seguinte grafo. Os seguintes passos são realizados pelo Algoritmo 3.1.*

Figura 11 – Grafo Ponderado



Fonte: Elaborada pelo autor.

Tabela 1 – Tabela mostrando os vértices selecionados pela heurística de Yamaguchi

vetor ℓ	**Vértice Selecionado**	**Atualizações**
(3,3,2,2,2,1)	vértice 6	$\ell(3) = 3$ $\ell(4) = 3$ $\ell(5) = 3$
(3,3,3,3,3,1)	vértice 5	$\ell(1) = 6$ $\ell(4) = 5$
(6,3,3,5,3,1)	vértice 2	$\ell(3) = 5$
(6,3,5,5,3,1)	vértice 3	$\ell(4) = 7$
(6,3,5,7,3,1)	vértice 1	
(6,3,5,7,3,1)	vértice 4	

Fonte: Elaborada pelo autor.

A ordem devolvida pelo algoritmo é [6,5,2,3,1,4] e o maior caminho no grafo $\overrightarrow{G}_{\Pi}$ *é 7.*

Similarmente a Proposição **?**, podemos obter limites superiores para a clique máxima ponderada em subgrafos de G.

Proposição 2 *Yamaguchi and Masuda (2008) Sejam* π *e* ℓ *devolvidos pelo Algoritmo 3.1. Para todo* $i = 1, \ldots, n,$.

$$\omega(G[\pi_1, \ldots, \pi_i], w) \leq \ell(\pi_i)$$

3.3 Heurística de Held, Cook e Sewell

Em Held, Cook, and Sewell (2012), uma heurística iterativa é usada para encontrar uma coloração ponderada inteira de G (Ver Algoritmo 3.2). Durante a execução do algoritmo,

as seguintes variáveis são atualizadas: o conjunto de vértices ainda não coloridos, representado por U; o vetor π, representando a ordem em que os vértices são coloridos; e o vetor $color[\pi_i]$, representando o peso de uma coloração ponderada do subgrafo induzido por todos os vértices coloridos até o vértice π_i, ou seja, $G[\pi_1, \ldots, \pi_i]$; e $res(v)$ representando o peso residual do vértice v. Inicialmente, o conjunto U recebe o conjunto dos vértices do grafo, π começa vazio.

Em cada iteração da heurística, um conjunto independente maximal S_i é construído. Os vértices são adicionados ao conjunto S_i, seguindo a ordem crescente dos pesos residuais. O peso de S_i será dado pelo menor peso residual dos seus vértices. O peso residual de cada vértice em S_i é decrementado de $res(v)$. Quando o peso residual de um vértice v torna-se zero, o vértice v é adicionado à seqüencia π, $color[v]$ recebe a soma dos pesos de todos os conjuntos independentes gerados e o vértice v é removido do conjunto U. Note que $color[\pi_j]$ representa o peso de uma coloração ponderada do subgrafo $G[\pi_1, \ldots, \pi_j]$, conforme a Proposição 1.

Na heurística de coloração ponderada proposta por Held, Cook e Sewell, a ordem de escolha dos vértices para gerar uma conjunto independente é determinada durante o processo, seguindo a ordem crescente dos pesos residuais. Com essa alteração, a heurística de coloração ponderada tenta maximizar a quantidade de vértices que são coloridos inicialmente pelo processo.

Algorithm 3.2 Heurística proposta por Held, Cook e Sewell

```
1: function HeurísticaHCS(G, π, color, tamanho)
2:     for v ∈ V do
3:         res(v) ← w(v)
4:     i ← 1
5:     tamanho ← 1
6:     U ← V
7:     UB ← 0
8:     while U ≠ ∅ do
9:         Encontre um conjunto independente maximal S_i em U seguindo a ordem crescente dos pesos residuais.
10:        y(S_i) ← min{res(v) : v ∈ S_i}
11:        UB ← UB + y(S_i)
12:        for u ∈ S_i do
13:            res(v) ← res(v) − y(S_i)
14:            if res(v) = 0 then
15:                U ← U \ {v}
16:                π[tamanho] = v.
17:                color[v] ← UB
18:                tamanho ← tamanho + 1
19:        i ← i + 1
```

Exemplo 6 *A execução da coloração ponderada inteira proposta em Held, Cook, and*

Sewell (2012) aplicada ao grafo da Figura 11 está descrita na Tabela 2. A ordem de coloração devolvida pela heurística é [6,5,3,4,2,1]. O peso da coloração ponderada é 8. Note que, diferentemente das heurísticas apresentadas no Capítulo 2, onde uma ordem inicial é dada como entrada, na Heurística de Held, Cook e Sewell, a ordem usada para selecionar os vértices é determinada durante o processo, escolhendo o vértice com o menor peso residual.

Tabela 2 – Execução do Algoritmo 3.2.

Peso Residual	**Ordem dos vértices seguindo o peso residual**	**Lista dos Conjuntos Independentes**
(3,3,2,2,2,1)	6,5,4,2,3,1	$S_1 = \{6, 2\}, y(S_1) = 1$
(3,2,2,2,2,0)	5,4,3,2,1	$S_1 = \{6, 2\}, y(S_1) = 1$ $S_2 = \{5, 3\}, y(S_2) = 2$
(3,2,0,2,0,0)	4,2,1	$S_1 = \{6, 2\}, y(S_1) = 1$ $S_2 = \{5, 3\}, y(S_2) = 2$ $S_3 = \{4, 2\}, y(S_3) = 2$
(3,0,0,0,0,0)	1	$S_1 = \{6, 2\}, y(S_1) = 1$ $S_2 = \{5, 3\}, y(S_2) = 2$ $S_3 = \{4, 2\}, y(S_3) = 2$ $S_4 = \{1\}, y(S_4) = 3$

Fonte: Elaborada pelo autor.

A partir da Proposição 1, temos que:

Proposição 3 *Held, Cook, and Sewell (2012) Sejam π e color devolvidos pelo Algoritmo 3.2. Para todo $i = 1, \ldots, n$, .*

$$\omega(G[\pi_1, \ldots, \pi_i], w) \leq color(\pi_i)$$

3.4 Heurística BITCOLOR

Uma heurística de coloração ponderada inteira com paralelismo de bits foi apresentada em Tavares *et al.* (2015). Em cada iteração, um conjunto independente maximal é obtido seguindo uma ordem inicial pré-estabelecida dos vértices ρ. A ordem ρ é utilizada para organizar os bits nos vetores de bits utilizados e é passada implicitamente juntamente com os vetores de bits. O peso do conjunto independente S é igual ao menor peso residual dos vértices de S. O peso residual dos vértices de S é reduzido pelo peso do conjunto independente S. Quando o peso residual de um vértice v torna-se zero, o vértice é adicionado à sequência π, $color[v]$ recebe a soma dos pesos dos conjuntos independentes gerados até o momento e o vértice v é removido do processo de coloração. Note que $color[\pi[j]]$ representa o peso da coloração ponderada do subgrafo $G[\pi[1], \ldots, \pi[j]]$. O processo continua enquanto existir vértice com peso residual maior do que zero.

A heurística de coloração ponderada inteira proposta em Tavares *et al.* (2015) está definida no Algoritmo 3.3:

Um passo importante neste Algoritmo é a sequência que é seguida para determinar a escolha dos vértices a entrarem no conjunto independente S_i, realizada na Linha

Algorithm 3.3 Heurística Coloração Ponderada Inteira Tavares *et al.* (2015)

```
1: function BITCOLOR(V , ρ, π, color, tamanho)
2:     for v ∈ V do
3:         res(v) ← w(v)
4:     i ← 1
5:     tamanho ← 1
6:     U ← V
7:     UB ← 0
8:     while U ≠ ∅ do
9:         Encontre um conjunto independente maximal S_i em G[U] selecionando vértices
       seguindo a ordem ρ.
10:        min_res ← min{res(v) : v ∈ S_i}
11:        y(S_i) ← min_res
12:        UB ← UB + min_res
13:        for u ∈ S_i do
14:            res(v) ← res(v) − y(S_i)
15:            if res(v) = 0 then
16:                U ← U \ {v}
17:                Π[tamanho] = v
18:                color[v] ← UB
19:                tamanho ← tamanho + 1
20:        i ← i + 1
```

9. Para o caso não ponderado, as duas estratégias de ordenação iniciais mais utilizadas em algoritmos de coloração seqüencial são:

- Maior grau primeiro (*largest-first ordering*) Welsh and Powell (1967). Uma ordem $(v_1, \ldots, v_n)$ dos vértices de G é dita **maior grau primeiro** se $d(v_1) \geq d(v_2) \geq \ldots \geq d(v_1)$.
- Menor grau último (*smallest-last ordering*) Matula and Beck (1983). Uma ordem $(v_1, \ldots, v_n)$ dos vértices de G é dita **menor grau último** se v_i é o vértice com o menor grau no grafo $G[V \setminus \{v_{i+1}, \ldots, v_n\}]$, para todo $i \in \{1, \ldots, n\}$.

No caso ponderado, consideramos duas possíveis estratégias de ordenação inicial dos vértices:

- A ordem de **menor peso primeiro** dos vértices de G é uma ordem $(v_1, \ldots, v_n)$ tal que v_i é o vértice com o menor peso em $G[V \setminus \{v_1, \ldots, v_{i-1}\}]$. Em caso de empate, o vértice v_i é o vértice com a maior soma dos pesos dos adjacentes em $G[V \setminus \{v_1, \ldots, v_{i-1}\}]$. A ordenação inicial dos vértices baseada na ordem crescente dos pesos é utilizada em Carraghan and Pardalos (1990b) e com um critério de desempate em Ostergard (1999). Em Tavares *et al.* (2015), um algoritmo de Branch & Bound foi apresentado utilizando esta ordenação inicial dos vértices com uma heurística de coloração ponderada. Os resultados computacionais mostraram que esse algoritmo superou os algoritmos de Ostergaard Ostergard (2002) e o algoritmo

de YamaguchiYamaguchi and Masuda (2008) para grafos com densidade maior que 50%.

- A ordem de **maior grau ponderado primeiro** dos vértices de G é uma ordem $(v_1, \ldots, v_n)$ tal que v_i é o vértice com o menor soma dos pesos dos seus adjacentes em $G[V \setminus \{v_{i+1}, \ldots, v_n\}]$. Em caso de empate, o vértice v_i é o vértice com a maior peso em $G[V \setminus \{v_{i+1}, \ldots, v_n\}]$. Essa estratégia de ordenação foi utilizada para ordenar os vértices do conjunto de ramificação em Warren and Hicks (2006)(Ver Seção 3.6). Essa ordem pode ser entendida como uma extensão para o caso ponderado da ordem proposta em Matula and Beck (1983).

Os resultados computacionais das diferentes estratégias de ordenação inicial dos vértices serão apresentados na seção seguinte.

3.5 Testes Computacionais

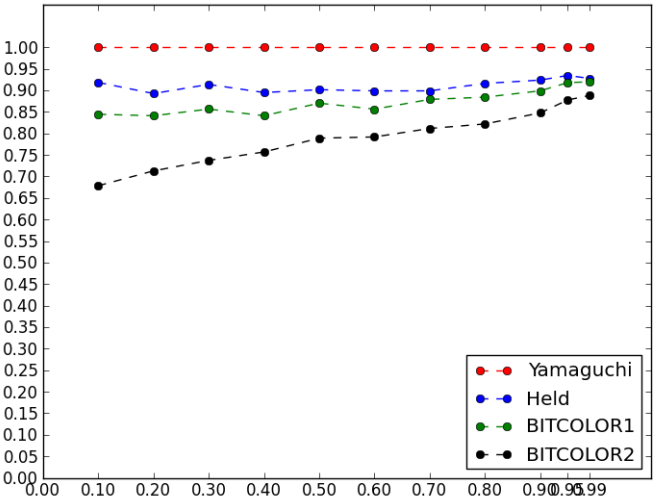
Comparamos o desempenho computacional da Heurística de Yamaguchi & Masuda (denotada por **YM**), Heurística de Held *et al* (denotada por **HCS**) e a Heurística de Coloração Ponderada com paralelismo de bits com as ordens iniciais **menor peso primeiro** (denotada por **BITCOLOR1**)e **maior grau ponderado primeiro** (denotada por **BITCOLOR1**). O código da Heurística **YM** foi disponibilizado gentilmente pelos autores. Os demais códigos são implementações próprias. Vale lembrar que tanto a **YM** quanto **HCS** não utilizam uma ordenação inicial dos vértices.

Os testes computacionais foram conduzidos num computador com processador Intel Core i7-2600K 3.40 Ghz, com 16Gb de memória, utilizando o sistema operacional Linux.

Avaliamos o desempenho das heurísticas utilizando instâncias aleatórias. Para cada combinação n (número de vértices) e p (probabilidade de existência de aresta), 10 grafos aleatórios foram gerados. Como é usual na literatura, os pesos dos vértices são distribuídos uniformemente entre 1 e 10. No total, consideramos 22 combinações (n, p), levando a 220 instâncias no total. Para cada (n, p), calculamos a média do tempo consumido e a média do limite superior obtidos pelas 10 instâncias para cada heurística.

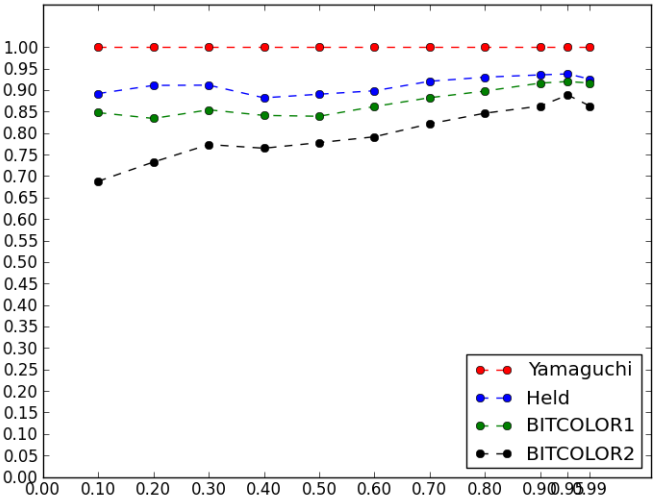
As Figuras 12 e 13 apresentam uma comparação entre as médias dos limites superiores das quatros heurísticas para instâncias com 200 e 300 vértices, respectivamente. A comparação tem por base a Heurística YM. Assim, para cada grupo de 10 instâncias, calculamos a razão entre a média de cada heurística e a média de YM. Os resultados revelam que **BITCOLOR1** e **BITCOLOR2** obtém os melhores limites superiores em média em todas as instâncias geradas. Os valores dos limites superiores calculados podem ser consultados no Apêndice A.

Figura 12 – Razão entre a média dos limites superiores de cada heurística e a média do limite superior de YM, por densidade, para instâncias com 200 vértices



Fonte: Elaborada pelo autor.

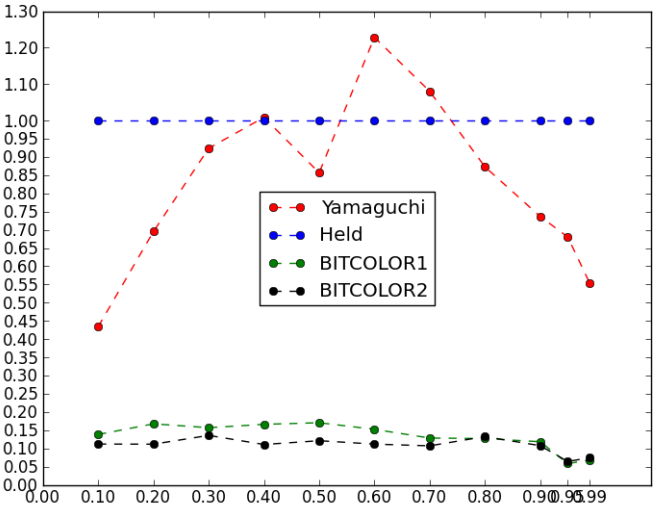
Figura 13 – Razão entre a média dos limites superiores de cada heurística e a média do limite superior de YM, por densidade, para instâncias com 300 vértices



Fonte: Elaborada pelo autor.

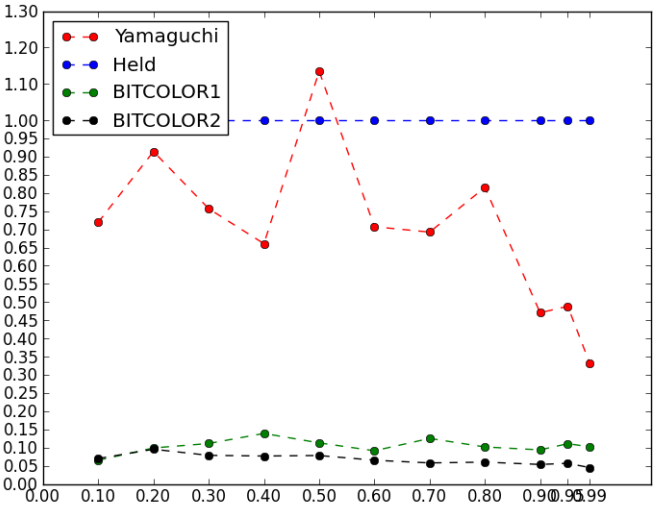
As Figuras 14 e 15 apresentam uma comparação similar , agora com relação ao tempo de execução, tomando HCS como base. Percebemos que **BITCOLOR1** e **BITCOLOR2** são os mais eficientes das 4 heurísticas e ambos consomem uma quantidade de tempo similar. Além disso, **BITCOLOR1** e **BITCOLOR2** consomem apenas de 15% do tempo gasto por **HCS**. Os valores dos tempo de execução absolutos podem ser consultados no Apêndice A.

Figura 14 – Razão entre a média dos tempos de execução de cada heurística e a média do tempo de execução de HCS, por densidade, para instâncias com 200 vértices



Fonte: Elaborada pelo autor.

Figura 15 – Razão entre a média dos tempos de execução de cada heurística e a média do tempo de execução de HCS, por densidade, para instâncias com 300 vértices



Fonte: Elaborada pelo autor.

3.6 Esquema de Ramificação

A maioria dos algoritmos de Branch & Bound para o CLIQUE PONDERADA, utiliza o esquema de ramificação proposto por Balas-YuBalas and Yu (1986). Neste esquema, o conjunto de vértices V é particionado em dois subconjuntos U e V' tais que

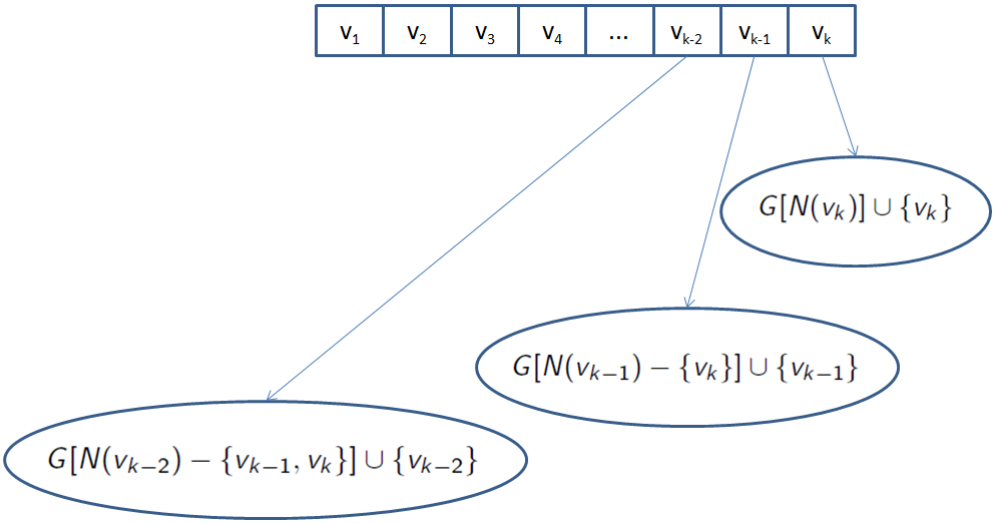
$$\omega(G[U], w) \leq LB \text{ e } V' = V \setminus U \tag{17}$$

,onde LB é o maior peso de uma clique conhecida. O conjunto V' é chamado de conjunto de ramificação. Podemos garantir que toda clique máxima ponderada de $G[V]$, deve possuir pelo menos um vértice de V'. Desta forma, se $(v_1, v_2, \ldots, v_n)$ é uma ordem dos vértices em V', então toda clique máxima ponderada de $G[V]$ está contida em pelo menos um dos seguintes subgrafos:

$$G[\{v_n\} \cup N(v_n)]$$
$$G[\{v_{n-1}\} \cup (N(v_{n-1}) \setminus \{v_n\}]$$
$$G[\{v_{n-2}\} \cup (N(v_{n-2}) \setminus \{v_{n-1}, v_n\})]$$
$$\vdots$$
$$G[\{v_1\} \cup (N(v_1) \setminus \{v_2, \ldots, v_{n-1}, v_n\})]$$

A Figura 16 ilustra o esquema de ramificação de Balas e Yu. Dessa maneira, a estratégia de ramificação fica responsável por definir uma ordem dos vértices em V', que será utilizada para gerar os subgrafos a serem analisados.

Figura 16 – Esquema de ramificação de Balas-Yu



Fonte: Elaborada pelo autor.

A corretude do esquema de ramificação está garantida pelo seguinte teorema:

Teorema 2 *Balas and Xue (1991) Seja K_0 uma clique máxima de $G[U]$ ou um limite superior para ela e seja $(v_1, \ldots, v_n)$ uma ordem arbitrária dos vértices de $V \setminus U$. Se G tem uma clique K_1 tal que $w(K_1) > w(K_0)m$ então K_1 está contida em uns dos seguintes m conjuntos $V_i = \{v_i\} \cup (N(v_i) \setminus \{v_i, \ldots, v_n\}), \quad i = m, \ldots, 1$*

Facilmente, podemos modificar o procedimento de limite superior para encontrar um subgrafo $G[U]$ tal que $\omega(G[U], w) \leq LB$, onde LB é um limite inferior conhecido para o problema.

Em Yamaguchi and Masuda (2008), o conjunto de ramificação é definido pelos vértices v tal que o maior caminho no grafo direcionado até v seja maior que um limite

inferior conhecido LB, ou seja:

$$V' = \{v : \ell(v) > LB\} \tag{18}$$

Em Held, Cook, and Sewell (2012) e Tavares *et al.* (2015), o conjunto de ramificação é definido pelos vértices π_i tal que o peso da coloração do grafo $G[v_1, \ldots, \pi_i]$ seja maior que um limite inferior conhecido LB, ou seja, se $color[\pi_i] > LB$

$$V' = \{\pi_i : color(\pi_i) > LB\} \tag{19}$$

Analisamos agora o tamanho do conjunto de ramificação para cada uma das heurísticas. Nos testes computacionais realizados, um limite inferior é obtido heuristicamente em todas as instâncias por um procedimento guloso. Esse limite inferior é utilizado para encontrar o conjunto V' em cada instância. O valor reportado é a média das 10 instâncias para cada combinação (n, p). As Tabelas 3 e 4 mostra a média da cardinalidade de V'.

Tabela 3 – Tamanho médio dos conjuntos de ramificação gerados por cada heurística, por densidade, para instâncias com 200 vértices.

Instancia	**Cardinalidade de V'**			
(n, p)	**Yamaguchi & Masuda**	**Held, Cook e Sewell**	**BITCOLOR1**	**BITCOLOR2**
(200,0.10)	67.70	**62.40**	68.70	93.60
(200,0.20)	84.10	**78.70**	86.50	119.50
(200,0.30)	91.70	**85.80**	92.00	131.20
(200,0.40)	93.60	**87.90**	94.40	134.80
(200,0.50)	96.50	**91.10**	96.10	138.70
(200,0.60)	94.40	**88.30**	95.20	138.40
(200,0.70)	91.30	**85.10**	91.40	134.90
(200,0.80)	82.10	**75.70**	80.80	122.30
(200,0.90)	65.30	**59.60**	64.90	101.10
(200,0.95)	45.10	**38.10**	44.50	66.70
(200,0.99)	13.80	**7.10**	9.40	12.40

Fonte: Elaborada pelo autor.

Tabela 4 – Tamanho médio dos conjuntos de ramificação gerados por cada heurística, por densidade, para instâncias com 300 vértices.

Instancia	**Cardinalidade de V'**			
(n, p)	**Yamaguchi & Masuda**	**Held, Cook e Sewell**	**BITCOLOR1**	**BITCOLOR2**
(300,0.10)	116.80	**110.70**	121.90	169.10
(300,0.20)	153.20	**143.70**	154.00	216.70
(300,0.30)	159.40	**151.00**	161.40	224.50
(300,0.40)	165.90	**156.90**	167.30	230.30
(300,0.50)	164.40	**155.40**	164.40	230.10
(300,0.60)	158.80	**150.10**	157.40	226.40
(300,0.70)	155.10	**146.60**	154.70	224.50
(300,0.80)	146.90	**137.80**	146.90	214.70
(300,0.90)	123.00	**114.60**	122.80	183.70
(300,0.95)	90.60	**81.50**	90.20	138.10
(300,0.99)	29.10	**17.20**	23.20	27.10

Fonte: Elaborada pelo autor.

Os testes computacionais revelam que a heurística proposta em Held, Cook, and Sewell (2012) é aquela que gera o menor conjunto de ramificação. Nessa heurística, os vértices são selecionados em ordem crescente de peso residual para cada conjunto independente S_i. Dessa maneira, o algoritmo tem a tendência de colorir mais vértices por cada conjunto independente construído.

Considerando o custo computacional, a heurística **BITCOLOR1** parece apresentar o melhor compromisso entre o tamanho do conjunto de ramificação e o esforço para obtê-lo.

Se por um lado **BITCOLOR1** gera menos subproblemas que **BITCOLOR2**. Por outro lado, a ordem **BITCOLOR2** obtém limites superiores melhores que **BITCOLOR1**.

De maneira geral, o tamanho do conjunto de ramificação dá uma idéia do número de subproblemas que serão resolvidos em um *Branch & Bound*. Entretanto, não traz informação sobre a dificuldade dos subproblemas, que está relacionada, por exemplo, à capacidade de poda.

No próximo capítulo, apresentaremos duas versões do algoritmo Branch & Bound utilizando as heurística **BITCOLOR1** e **BITCOLOR2** bem como os compararemos aos algoritmos de *Branch & Bound* do estado da arte.

4 ALGORITMOS DE BRANCH-AND-BOUND

Nesta seção, apresentamos os principais algoritmos exatos de *Branch & Bound* para CLIQUE PONDERADA. Embora a maioria deles utilizem a mesma estrutura, tais algoritmos diferem entre si nos seguintes pontos:

- Procedimentos de Limite inferior
- Procedimentos de Limite Superior
- Estratégias de Ramificação.

Na Seção 4.1, apresentamos a estrutura geral de um algoritmo de *Branch & Bound* para CLIQUE PONDERADA e seus pontos chaves.

Na Seção 4.2, apresentamos os algoritmos de *Branch & Bound* do estado da arte para CLIQUE PONDERADA: Algoritmo de Yamaguchi & Masuda (YM) e o Algoritmo de Held, Cook e Sewell (HCS).

Na Seção 4.3, apresentamos o nosso algoritmo de *Branch & Bound* para a CLIQUE PONDERADA, chamado de BITCLIQUE.

Na Seção 4.4, os testes computacionais são apresentados comparando os algoritmos do estado da arte com o algoritmo BITCLIQUE, utilizando as principais instâncias disponíveis na literatura.

4.1 Estrutura geral para CLIQUE PONDERADA

Apesar de sua simplicidade, o algoritmo apresentado em Carraghan and Pardalos (1990a,b) mostra a estrutura geral utilizada pela maioria dos algoritmos de *Branch & Bound* para CLIQUE PONDERADA propostos na literatura. A partir dele, podemos destacar os elementos chaves que impactam no desempenho de um procedimento utilizando essa estrutura, que pode ser vista no Algoritmo 4.1.

Basicamente, ele enumera as cliques de G seguindo a estratégia descrita na Seção 3.6. Nesse algoritmo há uma chamada recursiva à função CLIQUE, responsável por gerar todos os subproblemas da árvore de *Branch & Bound*.

Cada subproblema é definido pela tripla (C, P, LB), onde LB é um limite inferior para o problema, dado pela clique com o maior peso encontrada até o momento, $C = \{v_1, \ldots, v_d\}$ (onde d é a profundidade do subproblema) é a clique atual e P é o conjunto de vértices que podem entrar na clique atual , ou seja, $P \subseteq \cap_{i=1}^{d} N(v_i)$, chamado de conjunto candidato.

Seja $UB(P)$ um limite superior para a clique ponderada mais pesada de $G[P]$. Para um subproblema (C, P, LB), se $w(C)+UB(P) > LB$, então toda clique $C' \subseteq (C \cup P)$ tal que $w(C') \geq LB$ deve possuir pelo menos um vértice de P. Usando algum critério, um vértice v de P é selecionado, para ser adicionado à clique atual. Quando o vértice v é adicionado, dizemos que o vértice v foi ramificado. Depois que todas as cliques contendo

$C \cup \{v\}$ são enumeradas implicitamente pela chamada recursiva $CLIQUE(C \cup \{v\}, P \cap N(v), LB)$, o vértice v é removido do conjunto P.

Em cada subproblema, pode-se determinar um conjunto $U \subseteq P$ tal que $UB(U) \leq LB - w(C)$, de modo que os vértices em U não precisam ser ramificados. Em outras palavras, apenas os vértices de $P \setminus U$ são ramificados em uma ordem definida pela estratégia de ramificação. Esse esquema de ramificação foi apresentado primeiramente em Balas and Yu (1986).

No algoritmo *Branch & Bound* podemos destacar quatro pontos chaves, que estão destacados nas de caixas:

1. Procedimento de limite inferior.
2. Procedimento de limite superior.
3. Estratégia de ramificação.
4. Estrutura de dados para representação do grafo.

Algorithm 4.1 Algoritmo CP Carraghan and Pardalos (1990b)

```
1:  function MAIN
2:      [LB ← 0]
3:      CLIQUE(∅, V, LB)
4:  function CLIQUE(C, P, LB)
5:      if w(C) > LB then
6:          LB ← w(C)
7:      while P ≠ ∅ do
8:          if [w(C) + UB(P) ≤ LB] then
9:              return
10:         [v ← Selecione um vértice de P]
11:         CLIQUE(C ∪ {v}, [P ∩ N(v)], LB)
12:         P ← P \ {v}
```

A seguir, apresentamos sucintamente como esses quatro pontos são tratados nos principais algoritmos de *Branch & Bound* da literatura. Ressaltamos que a efetividade desses procedimentos no processo global pode não ser a mesma quando observamos cada ponto separadamente. Por exemplo, o procedimento que gera o melhor limite superior pode não ser o mais efetivo para a estratégia de ramificação escolhida.

O primeiro ponto chave, destacado no Algoritmo 4.1, está relacionado com a obtenção de um limite inferior de qualidade. Aqui há que se procurar um equilíbrio entre a qualidade do limite, que impacta no tamanho da árvore de busca, com o tempo para calculá-lo. Várias opções têm sido propostas na literatura.

Há também que se decidir se o limite inferior será calculado no nó raiz (Linha 2) e atualizado quando uma solução for encontrada (Linha 6), ou se vamos tentar melhorar o limite inferior corrente a cada nó da árvore.

Para o problema da clique máxima, onde todos pesos são iguais, uma heurística

de busca local eficiente foi utilizada em Maslov, Batsyn, and Pardalos (2014) para a obtenção de uma solução de alta qualidade apenas no nó raiz. Sua utilização mostrou-se bastante efetiva em instâncias difíceis da DIMACS.

Para o problema com pesos, destacamos os limites inferiores apresentados por Balas and Xue (1996),Babel (1994), Warren and Hicks (2006) e Held, Cook, and Sewell (2012).

Em Balas and Xue (1996), o limite inferior é calculado no nó raiz, a partir de um subgrafo cordal maximal. Nos outros nós, uma heurística gulosa é utilizada.

Em Babel (1994), uma heurística de coloração ponderada, inspirada no DSATURBrélaz (1979), é responsável por gerar um limite superior, e também um limite inferior em cada subproblema.

Em Warren and Hicks (2006), um limite inferior é obtido a partir de uma heurística gulosa do tipo *best-in*.

Em Held, Cook, and Sewell (2012), o limite inferior é calculado apenas no nó raiz, através de três algoritmos gulosos. O resultado é então melhorado por um busca local proposta em Andrade, Resende, and Werneck (2008).

Tabela 5 – Procedimentos de limite inferior

Algoritmo	**Referência**	**Limite Inferior**
BX,BX2	Balas and Xue (1991),Balas and Xue (1996)	Um limite inferior inicial é obtido a partir da clique máxima ponderada de um subgrafo cordal
Ba	Babel (1994)	Um limite inferior é obtido a partir de uma heurística de coloração ponderada inspirada no DSATUR
WH	Warren and Hicks (2006)	Um limite inferior é obtido a partir de uma heurística gulosa *best-in*
HCS	Held, Cook, and Sewell (2012)	Um limite inferior inicial é obtido a partir de heurísticas gulosas e melhorado por um procedimento de busca local

Fonte: Elaborada pelo autor.

O segundo ponto chave $\boxed{w(C) + UB(P) \leq LB}$ está relacionado com a poda de nós, ou seja, a enumeração implícita de cliques com o peso menor que o limite inferior corrente. Esta poda depende fortemente do procedimento de limite superior utilizado. A maior parte dos algoritmos utiliza heurísticas de coloração ponderada para obter limites superiores para a clique ponderada máxima em cada subproblema.

Em Balas and Xue (1991), utiliza-se uma heurística de coloração ponderada inteira, onde uma família de conjuntos independentes é construída iterativamente seguindo a ordem decrescente dos pesos.

Em Babel (1994), o limite superior é dado por uma heurística de coloração ponderada inteira, baseada no DSATUR, que fornece também um limite inferior.

Em Balas and Xue (1996), uma heurística de coloração fracionária, que potencialmente gera um limite superior melhor, é incorporada ao algoritmo de *Branch & Bound*.

Em Warren and Hicks (2006), uma versão otimizada da heurística de coloração ponderada proposta por Babel Babel (1994) é utilizada nos níveis iniciais da árvore de

busca. Nos outros níveis, usa-se uma variação da heurística de coloração ponderada inteira de Balas and Xue (1991).

Em Kumlander (2008), uma heurística de coloração ponderada particionada é aplicada no nó raiz. A coloração obtida é reaproveitada em todos os nós da árvore para tentar reduzir o custo computacional do processo.

Diferente das referências anteriores, que usam coloração ponderada, em Yamaguchi and Masuda (2008), o limite superior é determinado a partir de uma orientação acíclica dos vértices de G. Os autores apresentam um algoritmo polinomial para encontrar um sequência dos vértices ,que é utilizada para definir o grafo direcionado acíclico $\overrightarrow{G}_\pi$. O limite superior é dado pelo caminho com o maior peso em $\overrightarrow{G}_\pi$.

Em Held, Cook, and Sewell (2012) e Tavares *et al.* (2015), as heurísticas de coloração ponderada apresentadas no Capítulo 2 são utilizadas como procedimento de limite superior.

Tabela 6 – Procedimentos de limite superior

Algoritmo	**Referência**	**Limite Superior**
CP	Carraghan and Pardalos (1990b)	Heurística de coloração ponderada trivial
BX	Balas and Xue (1991)	Heurística de coloração ponderada inteira
Ba	Babel (1994)	Heurística de coloração ponderada inteira baseada no DSATUR
BX2	Balas and Xue (1996)	Heurística de coloração ponderada fracionária
WH	Warren and Hicks (2006)	Método híbrido combinando as heurística de Balas & Xue e de Babel
DK	Kumlander (2008)	Uma coloração ponderada particionada realizada uma única vez
YM	Yamaguchi and Masuda (2008)	Tamanho do maior caminho no grafo direcionado pela sequencia Π
HCS	Held, Cook, and Sewell (2012)	Heurística de coloração ponderada inteira
TCRM	Tavares *et al.* (2015)	Heurística de coloração ponderada inteira seguindo uma ordem inicial ρ

Fonte: Elaborada pelo autor.

O terceiro ponto chave $\boxed{v \leftarrow \text{Selecione um vértice de } P}$ está relacionado com a estratégia de ramificação. A ordem em que os vértices são ramificados determina os subproblemas a serem avaliados (em outros termos, os subgrafos pesquisados) e o seu potencial para gerarem limites favoráveis à poda. Em alguns algoritmos, uma única ordem de ramificação é usada em todos os subproblemas desde a raiz. Em outros, a cada nó da árvore, uma nova ordem de ramificação dos vértices candidatos é determinada. Tais ordens podem seguir um critério estático ou dinâmico. Observe que a ordem de ramificação pode ser definida antes de sabermos exatamente que são os vértices que devem ser ramificados. Nesse caso, o conjunto de ramificação é encontrado à medida que os vértices são ramificados e o conjunto candidato resultante ainda consiga superar a melhor clique conhecida. Em outros algoritmos, um procedimento é utilizado para encontrar o conjunto de ramificação previamente.

Em Carraghan and Pardalos (1990b), os vértices de P em cada subproblema, são ramificados em ordem decrescente de pesos. De maneira geral, essa estratégia de ramificação reduz a quantidade de vértices que precisam ser ramificados. Nesse algoritmo, o conjunto de ramificação é definido durante o processo de ramificação.

Em Babel (1994), uma ordem total dos vértices de P é definida $(\pi_1, \pi_2, \ldots, \pi_p)$ tal que $color[\pi_1] \leq color[\pi_2] \leq color[\pi_3] \leq \ldots \leq color[\pi_p]$, onde $color[\pi_i]$ representa o peso da coloração ponderada de $G[\pi_1, \pi_2, \ldots, \pi_{i-1}, \pi_i]$. Os vértices são ramificados em ordem decrescente do valor de color enquanto ainda for possível superar a melhor clique conhecida. Também neste caso o conjunto de ramificação é determinado ao longo do processo.

Em Balas and Xue (1991, 1996), um procedimento é utilizada para definir o conjunto de ramificação F. Os vértices de F são ramificados em uma ordem arbitrária.

Vale salientar aqui que enquanto em Carraghan and Pardalos (1990b), o foco é reduzir o conjunto de ramificação, em Babel (1994); Balas and Xue (1991, 1996), o foco é reduzir limite superior, ou seja, na capacidade de poda. A maior parte dos algoritmos falham em não conciliar esses dois objetivos.

Em Yamaguchi and Masuda (2008), uma ordem total para os vértices de P é definida $[v_1, v_2, \ldots, v_p]$, tal que $\ell(v_1) \leq \ell(v_2) \leq \ell(v_3) \leq \ldots \leq \ell(v_p)$, onde $\ell(v_i)$ representa o tamanho do maior caminho no subgrafo $\overrightarrow{G}_\pi[v_1, v_2, \ldots, v_{i-1}, v_i]$. Os vértices são ramificados em ordem decrescente do tamanho do maior caminho no grafo $\overrightarrow{G}_\pi$ enquanto for possível superar a melhor clique conhecida.

Em Held, Cook, and Sewell (2012), três regras são utilizados para definir três conjuntos de ramificações. O algoritmo efetivamente utiliza aquela de menor cardinalidade com um critério de desempate. Os vértices do menor conjunto de ramificação são ramificados em ordem crescente de grau.

Em Tavares *et al.* (2015), uma ordem total dos vértices $[\pi_1, \pi_2, \ldots, \pi_p]$ de P é definida tal que $color[\pi_1] \leq color[\pi_2] \leq color[\pi_3] \leq \ldots \leq color[\pi_p]$, onde $color[\pi_i]$ representa o peso da coloração ponderada de $G[\pi_1, \pi_2, \ldots, \pi_{i-1}, \pi_i]$. Os vértices são ramificados em ordem decrescente do peso da coloração ponderada. Embora a estratégia seja similar a de Babel, a coloração ponderada usada aqui é diferente.

Na maior parte dos algoritmos analisados, o conjunto de ramificação não é definido previamente. A principal vantagem dessa abordagem é que a melhoria do limite inferior pode contribuir para a redução do conjunto de ramificação. No caso dos algoritmo de Balas and Xue (1991), Balas and Xue (1996) e Held, Cook, and Sewell (2012), o conjunto de ramificação é definido previamente. No caso do algoritmo proposto em Held, Cook, and Sewell (2012), a situação é atenuada pelo fato de que diferentes regras para gerar o conjunto de ramificação são utilizadas.

O quarto ponto chave $\boxed{P \cap N(v)}$ está relacionado com a estrutura de dados que armazena o grafo e com a eficiência em que operações são realizadas sobre G durante o algoritmo. Mais recentemente, a utilização de estrutura de dados que armazena bits individualmente (vetores de bits) começou a ser explorada Segundo, Rodriguez-Losada, and Jimenez (2011); Segundo *et al.* (2013); Corrêa *et al.* (2014) para CLIQUE. A utilização dos vetores de bits possibilita a redução dos requisitos de memória e uma melhor apro-

veitamento do paralelismo das operações bit-a-bit realizadas pelo hardware. A utilização dessa estrutura de dados para CLIQUE PONDERADA é proposta nesta tese.

4.2 Algoritmos do Estado da Arte

Dentre os métodos do estado da arte para o problema CLIQUE PONDERADA, podemos destacar o algoritmo CLIQUER baseado no método das bonecas russas que será apresentado no próximo capítulo, e os seguintes 2 algoritmos de B & B:

- Algoritmo YM: Yamaguchi e Masuda apresentaram um algoritmo de *Branch & Bound* em Yamaguchi and Masuda (2008). Neste algoritmo, uma heurística é utilizada para encontrar uma orientação acíclica do grafo objetivando minimizar o peso do caminho mais pesado no grafo orientado. O algoritmo YM supera o algoritmo CLIQUER para grafos aleatórios com densidade superior 50%.
- Algoritmo HCS: Held, Cook e Sewell apresentaram um algoritmo de *Branch & Bound* melhorado para o problema do conjunto independente ponderado máximo em Held, Cook, and Sewell (2012). O algoritmo utiliza duas estratégias de poda e três estratégias de ramificação. No trabalho, o algoritmo é comparado com o algoritmo CLIQUER e com dois algoritmos de Branch & Cut que usam o CPLEX 12.2 e Gurobi 3.0.0, respectivamente. O algoritmo CLIQUER mostrou-se mais eficiente para grafos com densidade inferior 50%. Por outro lado, os algoritmos de Branch & Cut mostraram-se mais eficientes para grafos com densidade maior que 97%. O algoritmo HCS mostrou-se mais eficientes para as faixas intermediárias de densidade entre 50% e 97%.

4.3 Algoritmo de Yamaguchi e Masuda

O algoritmo **YM** segue a estrutura geral do Algoritmo 4.1. Um subproblema é definido pela tripla (C, P, LB). Em cada nó, aplica-se a heurística que devolve uma ordem dos vértices $\pi_1, \ldots, \pi_n$ tal que $\ell(\pi_1) \leq \ell(\pi_2) \leq \ldots < \ell(\pi_n)$, onde $\ell(\pi_i)$ representa o tamanho do maior caminho direcionado no subgrafo $\overrightarrow{G}_\pi[\pi_1, \ldots, \pi_i]$. O primeiro vértice escolhido para ser ramificado é π_n, se $a(\pi_n) > LB - w(C)$. Então, todas as cliques contendo o vértice π_n são enumeradas implicitamente. O vértice π_n é removido de P e o processo continua enquanto $\ell(\pi_k) > LB - w(C)$ para $k = n-1, \ldots, 1$. Esses passos podem ser vistos no Algoritmo 4.2.

Algorithm 4.2 Algoritmo proposto por Yamaguchi e Masuda

```
function MAIN
    LB ← 0
    YM(∅, V, LB)
function YM(C, P, LB)
    if w(C) > LB then
        LB ← w(C)
    CaminhoMaisPesado(G[P], w, π, ℓ)
    for i ← |P| até 1 do
        v ← π(i)
        if w(C) + ℓ(v) ≤ LB then
            retorne
        YM(C ∪ {v}, P ∩ N(v), LB)
        P ← P \ {v}
```

4.4 Algoritmo de Held, Cook e Sewell

O algoritmo de *Branch & Bound* apresentado por Held, Cook e Sewell, que denotaremos por HCS, resolve o problema do conjunto independente máximo ponderado. Esse algoritmo adota várias idéias dos algoritmos apresentados em Balas and Xue (1996); Babel (1994); Warren and Hicks (2006); Sewell (1998); Bron and Kerbosch (1973). Como o problema da clique máxima ponderada de G pode ser resolvido encontrando o conjunto independente máximo no grafo $\overline{G}$, vamos descrever o algoritmo no contexto de clique máxima ponderada.

Um subproblema na árvore de *Branch & Bound* gerado por HCS é definido pela quádrupla (C, P, X, LB), onde LB é um limite inferior, dado pela clique com o maior peso até o momento, $C = \{v_1, \ldots, v_d\}$ (onde d é a profundidade do subproblema) é a clique ponderada atual, P é o conjunto de vértices que podem entrar na clique atual chamado de conjunto candidato e X é o conjunto de vértices que já explorados em algum subproblema antecessor, ou seja, o peso de qualquer clique ponderada contendo um vértice de X é no máximo LB.

O algoritmo usa duas regras de poda e três estratégias de ramificação.

A **primeira regra de poda** utiliza o conjunto X e pode ser estabelecida da seguinte maneira: Se existe um vértice $x \in X$ tal que

$$w(x) \geq w((C \cup P) \cap \overline{N}(x))$$

então o subproblema atual não deriva uma clique ponderada que supere LB e pode ser podado.

Lema 3 *Dado um subproblema (C, P, X, LB), se existe um $x \in X$ tal que $w(x) \geq w((C \cup P) \cap \overline{N}(x))$ então toda clique $C' \subseteq (C \cup P)$ tem $w(C') \leq LB$.*

Prova Considere uma clique $C' \subseteq (C \cup P)$. Então, $C'' = \{x\} \cup (C' \setminus \overline{N}(x))$ é uma clique contendo $x \in X$, de modo que $w(C'') \leq LB$. Logo, as seguintes relações são válidas:

$$\begin{aligned} w(C'') &= w(\{x\} \cup C' \setminus \overline{N}(x)) \\ &= w(x) + w(C' \setminus \overline{N}(x)) \\ &= w(x) + w(C') - w(C' \cap \overline{N}(x)), && \text{como } C' \subseteq C \cup P \text{ e } \forall v\ w(v) \geq 0 \\ &\geq w(x) + w(C') - w((C \cup P) \cap \overline{N}(x)), && \text{como } w(x) - w((C \cup P) \cap \overline{N}(x)) \geq 0 \\ &\geq w(C') \end{aligned}$$

Logo, $w(C') \leq w(C'') \leq LB$.

A **segunda regra de poda** usa uma *coloração ponderada inteira*. Dado um subproblema (C, P, X, LB), se o peso da coloração ponderada de P for inferior ou igual a $LB - w(C)$, então este subproblema não pode gerar uma clique ponderada com peso superior a LB. Logo, todas as suas cliques estão implicitamente enumeradas.

A **primeira regra de ramificação** é uma extensão, para o caso ponderado, da regra de ramificação proposta por Balas e Yu Balas and Yu (1986). Dado um subproblema (C, P, X, LB), suponha que exista um $U \subseteq P$ tal que

$$\omega(G[U], w) \leq LB - w(C)$$

Então toda clique $C' \subseteq C \cup P$ com $w(C') > LB$ deve conter pelo menos um vértice de $P \setminus U$. O conjunto de ramificação é reduzido para $F = P \setminus U$.

A determinação do conjunto F assim como a verificação da segunda regra de poda são feitas conjuntamente por uma modificação da HeurísticaHCS (Ver Algoritmo 3.2). A versão modificada, que denotamos por HeurísticaHCSRestrita, termina quando todos os vértices estiverem totalmente coloridos (como antes) ou quando o peso da coloração atingir $LB - w(C)$, o que ocorrer primeiro. No primeiro caso, a segunda regra de poda é satisfeita. No segundo caso, os vértices não coloridos ou parcialmente coloridos definem F.

A **segunda regra de ramificação** utiliza o conjunto de vértices X. Podemos notar que toda clique ponderada máxima C' em $G[C \cup F]$ com $w(C') > LB$ deve conter pelo menos um vértice da não vizinhos de $x \in X$, um contradição, pois toda clique contendo x tem peso no máximo LB. Neste caso, podemos tomar o conjunto de ramificação como $F = P \cap \overline{N}(x)$.

A **terceira regra de ramificação** pode ser definida se existe um $v \in P$ tal que

$$w(v) \geq w(P \cap \overline{N}(v))$$

Podemos mostrar que existe uma clique ponderada máximo C' de $G[P]$ que inclui o vértice v.

Lema 4 *Dado um subproblema (C, P, X, LB). Se existe um $v \in P$ tal que $w(v) \geq w(P \cap \overline{N}(v))$ então existe uma clique ponderada máxima C' de $G[P]$ que inclui o vértice v.*

Prova Suponha que existe uma clique ponderada máxima C' de $G[P]$ que não inclui o vértice v. Seja $C'' = \{v\} \cup (C' \setminus \overline{N}(v))$ uma clique ponderada de $G[P]$ que inclui o vértice v. Vamos mostrar que $w(C'') \geq w(C')$. De fato, temos as seguintes relações são válidas:

$$
\begin{aligned}
w(C'') &= w(\{v\} \cup (C' \setminus \overline{N}(v)) \\
&= w(v) + w(C' \setminus \overline{N}(v)) \\
&= w(v) + w(C') - w(C' \cap \overline{N}(v)), \quad \text{como } C' \subseteq P \\
&\geq w(v) + w(C') - w(P \cap \overline{N}(v)), \quad \text{como } w(v) - w(F \cap \overline{N}(v)) \geq 0 \\
&\geq w(C')
\end{aligned}
$$

Portanto, existe uma clique ponderada máxima que deve incluir o vértice v em $G[P]$, o mesmo acontece para qualquer clique ponderada de $G[S \cup P]$

Neste caso, o conjunto de ramificação será $F = \{v\}$.

A cada subproblema, o algoritmo usa a regra que gera o menor conjunto de ramificação. Em caso de empate, a primeira regra tem maior prioridade, depois a terceira regra. O conjunto de ramificação é ordenado em ordem decrescente de seus graus em $G[P]$. Seja $v_1, v_2, \ldots, v_p$ a ordem do conjunto de ramificação F''. O problema é ramificado da seguinte maneira. Sendo

$$P_i = (P \cap N(v_i)) \setminus \{v_{i+1}, \ldots, v_p\} \ \forall v_i \in F'' \tag{20}$$

Resolve-se recursivamente o problema em cada $G[P_i]$ para $i = p, p-1, \ldots, 1$.

O algoritmo usa três heurísticas gulosas para encontrar uma solução inicial e depois essa solução inicial é melhorada através de uma busca local. Cada heurística gulosa constrói uma clique iniciando de um conjunto vazio e adicionando um vértice por vez usando um critério guloso diferente. O peso da clique obtida é utilizada para atualizar o limite inferior inicial do problema.

O algoritmo HCS completo encontra-se descrito pelo Algoritmo 4.3.

Algorithm 4.3 Algoritmo proposto por Held, Cook e Sewell

1: **function** CLIQUE_PONDERADA(G, w)
2: $LB \leftarrow 0$
3: **for** $i \leftarrow 1$ até 3 **do**
4: $C \leftarrow$ HeurísticaGulosa(G, i)
5: $C' \leftarrow$ BuscaLocal(G, S)
6: **if** $w(C') > LB$ **then**
7: $LB \leftarrow w(C')$
8: $HCS(\emptyset, V, \emptyset, LB)$
9: **function** HCS(C, P, X, LB)
10: **if** $w(C) > LB$ **then**
11: $LB \leftarrow w(S)$
12: **if** $\exists x \in X$ tal que $w(x) \geq w((C \cup P) \cap \overline{N}(x))$ **then**
13: **retorne**
14: HeurísticaHCSRestrita($G[P], \pi, color, tamanho$)
15: ▷ Todos os vértices são coloridos pela heurística de coloração ponderada
16: **if** $tamanho = |P|$ **then**
17: **returne**
18: Escolha o menor conjunto de ramificação $F = \{f_1, f_2, \ldots, f_p\} \subset P$ usando as três regras de ramificação
19: Ordene F em ordem decrescente de seus graus em $G[P]$
20: **for** $i \leftarrow p$ até 1 **do**
21: $(P_i \leftarrow (P \cap N(f_i)) \setminus \{f_{i+1}, \ldots, f_p\}$
22: HCS($C \cup \{f_i\}, P_i, X$)
23: $X \leftarrow X \cup \{f_i\}$

4.5 Algoritmo BITCLIQUE

Nesta seção, apresentamos o nosso algoritmo de *Branch & Bound*, que chamaremos de **BITCLIQUE**. Ele combina de forma efetiva vários elementos que já foram utilizados em outros algoritmos de *Branch & Bound*. Mais precisamente, usamos:

- Uma metaheurística para a obtenção de um limite inferior inicial.
- Uma ordenação inicial fixa dos vértices. Essa ordenação inicial dos vértices cumpre dois papeis importantes: evitar a reordenação dos vértices toda vez que a heurística de coloração ponderada for executada e prover um critério simples para a escolha dos vértices durante a coloração.
- Uma heurística de coloração ponderada inteira, utilizando vetores de bits. Os vetores de bits são utilizados aqui para acelerar as operações realizadas pela heurística.
- Uma Estratégia de ramificação inspirada em Babel (1994); Tomita *et al.* (2010) com base na coloração ponderada obtida.

Nas próximas seções, detalhamos melhor nossas escolhas e implementações para cada um dos quatro pontos chaves apontados acima.

4.5.1 Procedimento de Limite Inferior

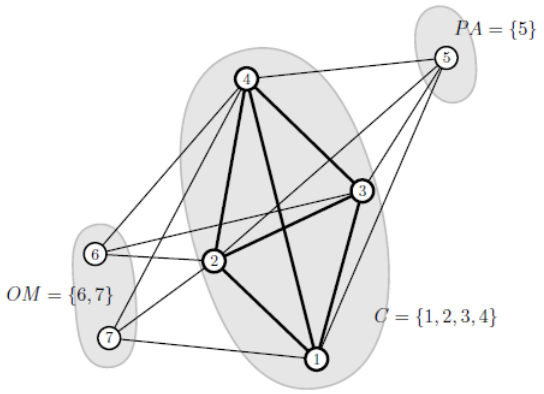
Diversas heurísticas disponíveis são capazes de encontrar boas soluções viáveis para o problema da CLIQUE PONDERADA:

- Algoritmo aumentante Mannino and Stefanutti (1999)
- Busca Local Escalonada (Phased Local Search)Pullan (2008)
- Busca Tabu com Multivizinhaça Wu, Hao, and Glover (2012)
- Busca Local de Fuga (Breakout Local Search)Benlic and Hao (2013)

A heurística escolhida para obter uma boa solução inicial foi a Busca Tabu com Multivizinhança Wu, Hao, and Glover (2012) (Ver Algoritmo 4.4). Em cada iteração, nessa metaheurística, é explorada a união de 3 vizinhanças, e a melhor solução admissível da vizinhança (não proibida ou melhor global) é escolhida. Inicialmente, uma solução inicial é construída de forma iterativa, a partir de um conjunto vazio e adicionando um vértice por vez, sem um critério específico, até encontrar uma clique maximal.

Os operadores de movimento, que definem respectivamente as 3 vizinhanças, são ADD, $SWAP$ e $DROP$. Esses operadores são definidos em função de dois subconjuntos de vértices PA e OM, relacionados com uma clique C. O subconjunto de vértices PA é formado pelos vértices que não estão em C, mas são adjacentes a todos os vértices de C. O subconjunto de vértices OM é formado pelos vértices que não estão em C, mas têm $|C| - 1$ vizinhos em C. A relação entre a clique C e os conjuntos de vértices PA e OM estão ilustradas na Figura 17, extraída do artigo Wu, Hao, and Glover (2012).

Figura 17: Relação entre uma clique C e os subconjuntos de vértices PA e OM.



Fonte: Wu, Hao, and Glover (2012)

O operador $ADD(v)$ adiciona um vértice v de PA na clique atual C. Esse movimento só pode ser aplicado quando $PA \neq \emptyset$. Depois do movimento $ADD(v)$, o peso da clique ponderada atual aumenta em $w(v)$. O operador $DROP(u)$ remove um vértice da clique atual C. Esse movimento só pode ser aplicado quando $C \neq \emptyset$. Depois do movimento $DROP(u)$, o peso da clique ponderada atual diminui em $w(u)$. O operador $SWAP(v, u)$ troca um vértice $v \in OM$ pelo vértice $u \in C$ que não é vizinho de v. Depois do movimento $SWAP(v, u)$, o peso da clique ponderada atual varia em $w(v) - w(u)$.

Além das restrições impostas pela busca tabu, a metaheurística implementa uma estratégia de reinicialização aleatória. Enquanto as restrições da busca tabu estabelecem uma forma de diversificação local, a estratégia de reinicialização aleatória assegura uma forma de diversificação global, forçando a busca a deixar regiões visitadas. O processo de reinicialização é acionado quando a busca atual estiver presa em um ótimo local profundo. Isso acontece quando o número de iterações consecutivas sem melhoras no peso da clique (representado por NI) excede o valor de L (número máximo permitido de iterações sem melhorias). Em outras palavras, L representa a profundidade da busca tabu. Um outro parâmetro da metaheurística é o número máximo de iterações, representado por $Iter_{max}$.

No nosso algoritmo, o número máximo de iterações, $Iter_{max}$, depende da densidade do grafo. Quando a densidade do grafo for maior 80%, o número máximo de iterações será 1.000.000. Caso contrário, será 100.000. A profundidade da busca tabu será 4000, valor sugerido em Wu, Hao, and Glover (2012) para grafos ponderados.

Algorithm 4.4 Busca Tabu com Multivizinhança para clique máxima ponderada

```
function BuscaTabuMultivinhanca (G, L,Iter_max)
Require: Um grafo ponderado G = (V, E, w), um inteiro L (profundidade da busca) e
  Iter_max número máximo de iterações.
    Iter ← 0
    C* ← ∅
    while Iter < Iter_max do
        C ← CliqueMaximalAleatória()
        Inicialize lista tabu
        NI ← 0
        C_local = C
        while NI < L do
            Construa a vizinhança N_1, N_2 e N_3 de C
            Escolha o melhor vizinho permitido C' ∈ N_1 ∪ N_2 ∪ N_3
            C ← C'
            NI ← NI + 1
            Iter ← Iter + 1
            Atualize a lista tabu
            if W(C) > W(C_local) then
                NI ← 0
                C_local ← C
        if W(C_local) > W(C*) then
            C* ← C_local
     return C*
```

4.5.2 Procedimento de Limite Superior

No capítulo anterior, mostramos que as heurísticas $BITCOLOR1$ e $BITCOLOR2$ conseguem obter o melhor compromisso entre a qualidade do limite superior e o tempo gasto

para calculá-lo. A redução do custo computacional é alcançada por dois fatores:

- Utilização de vetores de bits.
- Critério simples de escolha do vértice durante a coloração ponderada.

No Algoritmo 4.5, propomos uma implementação, utilizando vetores de bits, para a heurística de coloração ponderada inteira. O procedimento tem como parâmetros: um vetor de bits U, representando um conjunto de vértices; um vetor ρ, representando a ordem inicial dos vértices; um vetor π, representando a ordem obtida a partir da coloração, um vetor *color*, que armazena o limite superior dado pela coloração ponderada de subgrafos; e um inteiro tamanho que armazena o número de vértices coloridos. A ordenação inicial dos vértices ρ é colocada como parâmetro de entrada apenas para dar maior clareza ao texto. Os bits de todos os vetores de bits são organizados seguindo a ordem ρ. Logo, a instrução selecione o primeiro vértice do conjunto Q seguindo a ordem ρ pode ser substituído por selecione o índice do primeiro bit setado em 1 do vetor de bits Q.

A cada iteração, um conjunto independente ponderado S_i será construído. O vetor de bits Q representa os vértices que podem ser adicionados ao conjunto independente corrente S_i. Os vértices adicionados em S_i são selecionados do vetor de bits Q seguindo a ordem inicial ρ. Cada vez que um vértice v é selecionado, o conjunto independente S_i é atualizado $S_i \leftarrow S_i \cup \{v\}$. Os vértices adjacentes a v e o próprio v são removidos do bitmap Q, ou seja, $Q \leftarrow Q \setminus N(v); Q \leftarrow Q \setminus \{v\}$. O processo continua enquanto ainda existir um vértice $v \in Q$. Quando Q torna-se vazio, um conjunto independente maximal S_i é obtido. O peso do conjunto independente S_i é dado pelo menor peso residual dos vértices de S_i. O peso residual de cada vértice v de S_i é atualizado, decrementando-o do peso da classe de cor S_i.Os vértices com peso residual iguais a zero podem ser removidos de U. A heurística de coloração ponderada termina quando U torna-se vazio. As instruções em caixas são aquelas substituídas por instruções específicas que manipulam diretamente o vetores de bits.

4.5.3 Vetores de bits

Vetores de bits são uma estrutura de dados discreta, apropriada para modelar uma sequência de bits que pode ser manipuladas eficientemente por computadores, utilizando poucas instrução. Particularmente, vetores de bits podem ser utilizados para armazenar a matriz de adjacência de um grafo ou um subconjunto de um conjunto ordenado. Por exemplo, em um grafo com 6 vértices $\{v_1, \ldots, v_6\}$, uma clique formada pelos vértices v_1, v_3, v_5, v_6 pode ser representada pelo vetor de bits $\{101011\}$, onde cada bit está associado a um vértice do grafo e cada bit igual a 1 indica um vértice que está na clique.

O paralelismo de bits é uma forma de paralelismo alcançada quando representamos os dados em vetores de bits de tamanho w, onde w é o tamanho da palavra do computador (por exemplo, 32 ou 64 bits). Operações específicas podem ser utilizadas

Algorithm 4.5 Heurística de Coloração Ponderada

function BITCOLOR($U, \rho, \pi, color, tamanho$)
  $\forall v \in V, res(v) \leftarrow w(v)$
  $i \leftarrow 1$
  $tamanho \leftarrow 1$
  $UB \leftarrow 0$
  while $\boxed{U \neq \emptyset}$ **do**
    $S_i \leftarrow \emptyset$; $\boxed{Q \leftarrow U}$
    while $\boxed{Q \neq \emptyset}$ **do**
      $\boxed{\text{v} \leftarrow \text{Selecione o primeiro elemento de } Q \text{ seguindo a ordem } \rho.}$
      $\boxed{Q \leftarrow Q \setminus N(v)}$; $\boxed{Q \leftarrow Q \setminus \{v\}}$; $S_i \leftarrow S_i \cup \{v\}$
    $min\_res \leftarrow min\{res(v) | v \in S_i\}$
    $y(S_i) \leftarrow min\_res$
    $UB \leftarrow UB + y(S_i)$
    for $v \in S_i$ **do**
      $res(v) \leftarrow res(v) - y(S_i)$
      if $res(v) = 0$ **then**
        $\boxed{U \leftarrow U \setminus \{v\}}$
        $\pi[tamanho] \leftarrow v$
        $color[v] \leftarrow UB$
        $tamanho \leftarrow tamanho + 1$
    $i \leftarrow i + 1$

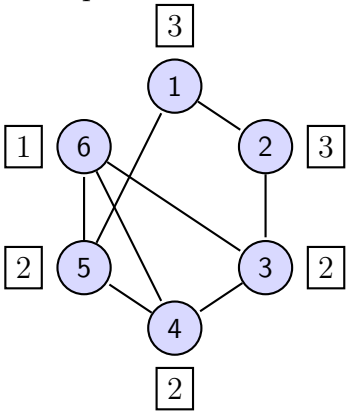
para processar um vetor de bits de tamanho w em uma única instrução do processador. Paralelismo de bits tem sido utilizado com bastante sucesso em muitos algoritmos, particularmente para casamento de cadeia de caracteresBaeza-Yates and Gonnet (1992). Recentemente, o paralelismo de bits tem sido explorado para resolver problemas de otimização combinatória como *SAT*Segundo *et al.* (2008), *Clique Máxima*Segundo *et al.* (2010); Segundo, Rodriguez-Losada, and Jimenez (2011); Segundo *et al.* (2013); Segundo and Tapia (2014); Segundo, Nikolaev, and Batsyn (2015) e *Coloração de Vértices*Komosko *et al.* (2015).

Neste trabalho, o paralelismo de bits foi usado na heurística de coloração ponderada sequencial e na atualização do conjunto de vértices candidatos. Além disso, o uso de vetores de bits diminui os requisitos de memória para a representação de matriz da adjacência e dos subconjuntos de vértices manipulados durante o algoritmo.

4.5.4 Estratégia de Ramificação

Dado um subproblema (C, P, LB), a heurística de coloração ponderada devolve uma ordem total dos vértices $\pi_1, \pi_2, \ldots, \pi_n$ tal que $color[\pi_1] \leq color[\pi_2] \leq \ldots \leq color[\pi_n]$, onde $color[\pi_i]$ é um limite superior para $\omega(G[\pi_1, \ldots, \pi_i], w)$. O vértice π_i é escolhido para ser ramificado enquanto $w(C) + color[\pi_i] > LB$, para todo $i = n, \ldots, 1$.

Considere o seguinte exemplo:



Seja (6, 5, 4, 3, 2, 1) a ordem inicial dos vértices. A coloração ponderada obtida pela heurística de coloração será:

1. $S_1 = \{6, 2\}, y(S_1) = 1$
2. $S_2 = \{5, 3\}, y(S_2) = 2$
3. $S_3 = \{4, 2\}, y(S_3) = 2$
4. $S_4 = \{1\}, y(S_4) = 3$

Na próxima tabela, veremos a ordem π devolvida pela heurística de coloração e o valor de *color*:

π	6	5	3	4	2	1
color	1	3	3	5	5	8

Considerando o limite inferior inicial igual a zero, o primeiro vértice a ser

ramificado será o vértice 1. Quando o vértice 1 é ramificado, a clique $\{1, 2\}$ é encontrada com peso 6. O próximo vértice a ser ramificado seria o vértice 2, começando com $C = \emptyset$, mas $w(C) + color[2] < LB$, então o processo de ramificação é interrompido.

4.5.5 Algoritmo de *Branch & Bound*

Nesta seção, apresentamos o algoritmo de *Branch & Bound* para o problema CLIQUE PONDERADA, que chamaremos de BITCLIQUE. Ele combina de forma efetiva vários elementos que já foram utilizados em outros algoritmos de *Branch & Bound*. Mais precisamente, usamos:

- Uma ordenação inicial fixa dos vértices, que estabelece uma organização dos vetores de bits utilizados. Essa ordenação inicial dos vértices também define a ordem de seleção de vértices na heurística de coloração ponderada inteira proposta.
- Limite superior obtido por uma implementação baseada em vetores de bits de uma uma heurística de coloração ponderada inteira inspirado em Balas and Xue (1996). Os resultados computacionais apresentados no Capítulo 3 mostram que o limite superior obtido por essa heurística apresenta um bom compromisso entre a qualidade da solução e o tempo computacional, frente às demais propostas do estados da arte.
- Estratégia de ramificação baseada na coloração ponderada inteira, porém inspirada em Babel (1994)
- Utilização de instruções com paralelismo em nível de bits em várias operações requeridas pelo algoritmo.

A estrutura de BITCLIQUE está descrita no Algoritmo 4.6. Note que ela segue o formato geral apresentado na Seção 3.1. O procedimento $BITCOLOR$, que realiza a coloração ponderada, é aquele definido pelo Algoritmo 4.5. Relembre que ele considera os vértices para a coloração seguindo a ordem ρ e devolve a ordem *ordem* em que eles são efetivamente coloridos; *color* armazena os limite superiores para os subgrafos definidos por *ordem*; *cont* guarda a quantidade de vértices que está no conjunto P.

Algorithm 4.6 Algoritmo BITCLIQUE

```
function MAIN
    ρ ← ordenação inicial dos vértices.
    LB ← heurística tabu com multivizinhaça.
    BITCLIQUE(∅, V, LB)
function BITCLIQUE(C, P, LB)
    if w(C) > LB then
        LB ← w(C)
    BITCOLOR(P, ρ, ordem, color, cont)
    for i ← cont até 1 do
        v ← ordem[i]
        if w(C) + color[v] ≤ LB then
            retorne
        newP ← P ∩ N(v)
        P ← P \ {v};
        BITCLIQUE(C ∪ {v}, newP, LB)
```

4.6 Resultados Computacionais

Nós comparamos o desempenho computacional do nosso algoritmo BITCLIQUE com o algoritmo *Branch & Bound* de Yamaguchi e Masuda (YM), disponibilizado gentilmente pelos os autores, e com algoritmo de *Branch & Bound* proposto em Held, Cook, and Sewell (2012), disponibilizado em `https://code.google.com/p/exactcolors/`. Consideramos duas versões do nosso algoritmo: BITCLIQUE1, que usa a ordenação inicial dada pelo **menor peso primeiro**; BITCLIQUE2 onde a ordem inicial considera o critério do **maior grau ponderado primeiro**.

Todos os testes foram realizados num computador com processador *Intel Core i7-2600K 3.40Ghz*, 8 Mb de cache, com 6Gb de memória, utilizando o sistema operacional Linux.

Nós avaliaremos o desempenho do nosso algoritmo utilizando as principais instâncias disponíveis na literatura:

- Grafos Aleatórios
- DIMACS-W
- ExactColor

4.6.1 Grafos Aleatórios

Em nosso experimento, nós geramos 10 grafos aleatórios para cada combinação n (número de vértices) e p (probabilidade de existência de cada aresta) considerada. Nós estamos usando o modelo de grafos aleatórios $G(n, p)$. Neste modelo, o grafo é construído adicionando arestas aleatoriamente. Cada aresta é incluída no grafo com probabilidade p, independentemente de todas as outras arestas. Logo, todos os grafos com n vértices e m

arestas têm a mesma probabilidade de serem gerados definida por

$$p^m(1-p)^{\frac{n(n-1)}{2}-m} \tag{21}$$

Os pesos atribuídos aos vértices estão uniformemente distribuído no intervalo entre 1 e 10. Nós estamos usando dois geradores de números pseudo-aleatório: *drand*, que devolve um número em ponto flutuante de precisão dupla uniformemente distribuído no intervalo [0,1) e *urand*, que retorna um inteiro sem sinal uniformemente distribuído no intervalo $0, .., 2^{24} - 1$.

Algorithm 4.7 Algoritmo para a geração de grafos aleatórios no modelo $G(n, p)$

```
function GrafoAleatório(n, p, seed)
    V ← {1, ..., n}
    E ← ∅
    for i ← 1 até n do
        w(i) ← (urand(seed) mod 10) + 1
        for j ← i + 1 até n do
            if drand(seed) ≤ p then
                E ← E ∪ {i, j}
```

Como é usual na literatura, o número de vértices das instâncias diminui à medida que a probabilidade de existência de aresta aumenta. Consideramos 20 combinações, levando a 200 instâncias no total. Para cada par (n, p), calculamos a média do número de subproblemas na árvore de B & B e a média do tempo consumido pelas 10 instâncias para cada algoritmo. O tempo de limite de resolução para cada instância é 1h (3600s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada na tabela como $*^x$, onde $x \in [1, 10]$ significa o número de instâncias não resolvidas no tempo limite. Identificamos em negrito o melhor desempenho médio em cada caso.

A Tabela 7 mostra que, para essas instâncias, ambos **BITCLIQUE1** e **BITCLIQUE2** dominam amplamente **YM** e **HCS** em número de subproblemas gerados. Podemos observar que o número médio de subproblemas cai em pelo menos uma ordem de magnitude. Além disso, notamos também que os novos algoritmos resolvem bem mais instâncias, dentro do tempo limite, que os algoritmos do estado da arte.

Já a Tabela 8, mostra que a redução do número de subproblemas se traduz igualmente em uma redução do tempo de execução. Por exemplo, o Algoritmo **BITCLIQUE2** chega a ser 3860 vezes mais rápido em relação YM para o grupo (200,0.90)

Uma comparação direta entre **BITCLIQUE1** e **BITCLIQUE2** revela que o primeiro é superior ao segundo em todos os grupos de instâncias até $p = 0.8$, chegando a obter um tempo de processamento até 3 vezes menor. Essa relação, porém, inverte-se para os grupos com densidade maior ou igual a 0.90, quando **BITCLIQUE2** passa a apresentar um desempenho superior, tanto em número de nós, quanto em tempo de

execução. Conjeturamos que, à medida que a densidade aumenta, a importância dos pesos diminui e o problema fica mais próximo do problema não ponderado.

Tabela 7 – Média dos número de subproblemas para 10 instâncias de grafos aleatórios para cada par (n, p).

Instância	**Números de Subproblemas**			
(n,p)	**YM**	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**
(2500,0.10)	2.34e+04	2.18e+04	**7.22e+02**	1.20e+03
(5000,0.10)	2.02e+05	2.12e+05	**2.52e+03**	3.72e+03
(2500,0.20)	2.83e+05	2.32e+05	1.54e+04	**1.07e+04**
(5000,0.20)	5.41e+06	5.01e+06	**2.12e+05**	4.25e+05
(2500,0.30)	4.77e+06	4.15e+06	**2.11e+05**	3.16e+05
(5000,0.30)	1.58e+08	$*^{10}$	**7.48e+06**	1.15e+07
(1200,0.40)	1.67e+06	1.29e+06	**1.03e+05**	1.15e+05
(2500,0.40)	1.00e+08	$*^{8}$	**5.13e+06**	1.19e+07
(600,0.50)	4.72e+05	3.22e+05	**3.95e+04**	4.06e+04
(1200,0.50)	2.98e+07	2.58e+07	**1.95e+06**	3.36e+06
(600,0.60)	6.06e+06	4.74e+06	**4.84e+05**	6.07e+05
(1200,0.60)	$*^{10}$	$*^{10}$	**6.91e+07**	$*^{10}$
(400,0.70)	6.70e+06	4.22e+06	5.67e+05	**5.33e+05**
(500,0.70)	4.54e+07	3.29e+07	**3.62e+06**	4.42e+06
(600,0.70)	1.82e+08	$*^{10}$	**1.31e+07**	1.82e+07
(300,0.80)	2.11e+07	1.17e+07	1.81e+06	**1.22e+06**
(400,0.80)	$*^{8}$	$*^{8}$	2.73e+07	**2.28e+07**
(200,0.90)	1.28e+07	3.01e+06	1.03e+06	**8.84e+04**
(300,0.90)	$*^{10}$	$*^{10}$	2.40e+08	**3.47e+07**
(200,0.95)	1.34e+08	2.44e+06	5.30e+06	**3.74e+04**

Fonte: Elaborada pelo autor.

Tabela 8 – Média dos tempo de execução para 10 instâncias de grafos aleatórios para cada par (n, p).

Instância	**Tempo de execução(segundos)**			
(n,p)	**YM**	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**
(2500,0.10)	**0.26**	5.98	0.80	0.88
(5000,0.10)	**3.19**	99.04	3.32	5.13
(2500,0.20)	3.74	33.88	**1.99**	3.60
(5000,0.20)	88.30	920.76	**41.76**	83.11
(2500,0.30)	63.45	373.57	**22.63**	57.79
(5000,0.30)	2988.57	> 3600	**1230.47**	3358.41
(1200,0.40)	19.00	74.25	**4.79**	10.64
(2500,0.40)	1586.25	> 3600	**443.90**	1216.67
(600,0.50)	3.84	10.61	**0.91**	1.46
(1200,0.50)	385.89	1228.10	**76.17**	194.71
(600,0.60)	60.93	127.47	**9.40**	18.65
(1200,0.60)	> 3600	> 3600	**2608.70**	> 3600
(400,0.70)	58.32	78.17	**6.77**	11.10
(500,0.70)	446.44	686.61	**54.01**	101.71
(600,0.70)	2195.10	> 3600	**248.48**	553.79
(300,0.80)	175.92	168.28	**15.39**	17.34
(400,0.80)	> 3600	> 3600	**311.87**	441.60
(200,0.90)	111.56	36.29	5.60	**0.98**
(300,0.90)	> 3600	> 3600	2279.47	**591.86**
(200,0.95)	1582.71	34.44	24.01	**0.41**

Fonte: Elaborada pelo autor.

4.6.2 DIMACS-W

O conjunto de instâncias da DIMACS foi criada para o Segundo Desafio em Cliques, Satisfabilidade e Coloração de Grafos. Ele é formado por 80 grafos obtidos a partir de vários problemas:

- *brock* : instâncias do gerador de Mark Brockington e Joe Culberson que tenta esconder uma clique com um tamanho maior do que o esperado pelo modelo clássico de grafos aleatórios.
- *p_hat*: instâncias construídas por um gerador que usa 3 parâmetros: n, número de nós, e a e b, dois parâmetros de probabilidade de existência de arestas tal que $0 \leq a \leq b \leq 1$, gerando grafos com cliques maiores do que o esperado pelo modelo clássico de grafos aleatórios.
- *hamming*: instâncias de problemas de Teoria de Códigos.
- *san* e *sanr*: instâncias obtidas a partir do problema de cobertura de vértices mínima.
- *MANN*: instâncias obtidas da formulação do problema da cobertura tripla de Steiner

O conjunto de instâncias DIMACS-W são instâncias obtidas a partir das instâncias DIMACS para o problema da clique máxima. Várias heurísticas utilizam as instâncias da DIMACS-W como instâncias de referência. Há diversas maneiras de definir uma função de ponderação dos vértices. Vamos adotar a função de ponderação descrita

em Pullan (2008): para cada vértice i, w_i é igual a $(i \bmod 200) + 1$.

Os resultados dos experimentos computacionais com as instâncias desse grupo podem ser vistos na Tabela 9. Agora, o tempo limite de resolução para cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Identificamos em negrito o melhor desempenho em cada caso.

Podemos observar novamente que **BITCLIQUE1** gera sempre menos subproblemas que **YM** e **HCS**. O mesmo ocorre com **BITCLIQUE2**, com exceção das instâncias *hamming10-2*, que esse algoritmo não consegue resolver, e *san400_0.7_1*, onde o número de subproblemas é superior ao de **HCS**.

Por outro lado, não há prevalência clara de **BITCLIQUE1** sobre **BITCLIQUE2** ou vice-versa, a não ser para instâncias com densidade a partir de 0.90, quando o segundo gera menos subproblemas que o primeiro.

Em relação a tempo, o algoritmo **BITCLIQUE1** é mais rápido que o algoritmo **BITCLIQUE2** para instâncias com densidade menor que 0.90 (com exceção dos grafos da família *p_hat*). Além disso, o algoritmo **BITCLIQUE1** resolve todas as instâncias resolvidas pelos algoritmos anteriores enquanto **BITCLIQUE2** não consegue resolver a instância *hamming10-2*, que é resolvida por **HCS**.

Note ainda que os novos algoritmos são quase sempre mais rápidos que os da literatura. Há reduções bastante significativas, como no caso das instâncias *phat_1000-2* e *phat_1500-2*. Notamos também que os novos algoritmos conseguiram resolver mais instâncias dentro do tempo limite estabelecido.

4.6.3 Exactcolor

Essas instâncias representam subproblemas de conjunto independente máximo ponderado, obtidos durante o algoritmo de *Branch & Price* para o problema de coloração de vértices Held, Cook, and Sewell (2012).

Seja $\mathcal{S}$ o conjunto de todos os conjuntos independentes em G. O número cromático de G, denotado por $\chi(G)$, pode ser obtido da seguinte maneira:

$$\chi(G) = \min \sum_{S \in \mathcal{S}} x_S \tag{22}$$

$$\text{s.a.} \sum_{\{S \in \mathcal{S}: v \in S\}} x_S \leq 1 \quad \forall v \in V \tag{23}$$

$$x_S \in \{0, 1\} \qquad \forall S \in \mathcal{S} \tag{24}$$

Em Mehrotra and Trick (1996), este problema é resolvido por um algoritmo de Branch & Price. O problema relaxado resolvido em cada nó da árvore é:

Tabela 9 – Resultados computacionais com as instâncias DIMACS-W

Instância		**Números de Subproblemas**				**Tempo de execução(segundos)**			
Nome	**(n,p)**	**YM**	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**	**YM**	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**
sanr400_0.5	(400,0.50)	4.05e+04	3.51e+04	**3.74e+03**	4.05e+03	0.30	1.09	**0.15**	0.18
san1000	(1000,0.50)	1.15e+05	2.98e+04	3.06e+03	**2.35e+03**	4.50	3.43	**1.99**	2.29
brock400_1	(400,0.75)	2.34e+07	1.56e+07	1.96e+06	**1.76e+06**	256.57	339.63	**29.98**	37.80
brock400_2	(400,0.75)	2.54e+07	1.87e+07	**2.28e+06**	2.51e+06	267.38	418.33	**34.75**	51.32
brock400_3	(400,0.75)	2.26e+07	1.74e+07	**1.77e+06**	1.85e+06	229.46	388.23	**28.66**	38.58
brock400_4	(400,0.75)	9.37e+06	7.55e+06	1.04e+06	**7.56e+05**	123.42	200.16	**17.83**	20.11
brock800_1	(800,0.65)	1.33e+08	1.28e+08	**8.77e+06**	1.64e+07	2267.59	5130.53	**391.50**	853.90
brock800_2	(800,0.65)	2.11e+08	1.93e+08	**1.62e+07**	3.07e+07	3499.57	7104.80	**598.11**	1323.48
brock800_3	(800,0.65)	1.62e+08	1.63e+08	**1.15e+07**	2.12e+07	2326.46	6104.75	**457.88**	986.15
brock800_4	(800,0.65)	2.23e+08	*	**1.78e+07**	3.38e+07	2938.27	*	**610.96**	1412.36
p_hat500-2	(500,0.50)	4.65e+05	1.94e+05	1.83e+04	**5.32e+02**	8.68	8.08	0.51	**0.09**
p_hat700-2	(700,0.50)	1.83e+07	3.98e+06	7.59e+05	**2.07e+03**	410.32	190.80	18.28	**0.27**
p_hat1000-2	(1000,0.49)	*	*	1.67e+07	**8.48e+04**	*	*	716.71	**9.11**
p_hat1500-2	(1500,0.51)	*	*	*	**3.20e+06**	*	*	*	**566.07**
sanr400_0.7	(400,0.70)	4.15e+06	2.55e+06	3.46e+05	**3.46e+05**	36.97	62.83	**5.65**	7.95
san400_0.7_1	(400,0.70)	1.15e+07	6.07e+04	**4.30e+04**	9.19e+04	134.27	2.86	**2.13**	4.56
san400_0.7_2	(400,0.70)	1.44e+06	1.94e+05	**6.45e+04**	7.20e+04	20.34	6.74	**2.95**	5.12
san400_0.7_3	(400,0.70)	9.69e+05	4.36e+05	4.66e+04	**7.85e+03**	11.42	12.23	2.12	**0.76**
p_hat300-3	(300,0.74)	6.47e+05	2.67e+05	4.66e+04	**2.64e+03**	9.06	6.70	0.57	**0.10**
p_hat500-3	(500,0.75)	*	6.51e+07	1.59e+07	**1.05e+05**	*	2412.55	301.27	**4.53**
gen200_p0.9_44	(200,0.90)	7.49e+06	1.27e+06	4.23e+05	**1.73e+04**	63.40	18.76	2.91	**0.48**
gen200_p0.9_55	(200,0.90)	3.78e+06	7.88e+05	1.12e+05	**4.22e+03**	33.55	11.15	1.11	**0.35**
gen400_p0.9_55	(400,0.90)	*	*	*	**1.32e+08**	*	*	*	**3272.15**
gen400_p0.9_65	(400,0.90)	*	*	*	*	*	*	*	*
gen400_p0.9_75	(400,0.90)	*	*	3.48e+08	**9.03e+06**	*	*	6738.43	**325.59**
C250.9	(250,0.90)	2.96e+07	1.24e+07	2.30e+06	**2.15e+05**	393.56	225.40	18.33	**3.55**
san200_0.9_1	(200,0.90)	3.78e+05	1.72e+04	6.26e+03	**2.60e+01**	4.64	**0.39**	0.54	0.46
san200_0.9_2	(200,0.90)	2.97e+06	3.51e+05	2.29e+04	**6.54e+03**	25.85	5.27	0.50	**0.39**
san200_0.9_3	(200,0.90)	1.28e+07	2.51e+06	1.14e+06	**9.52e+04**	107.34	37.19	7.41	**1.51**
san400_0.9_1	(400,0.90)	*	2.96e+07	6.59e+06	**2.33e+06**	*	1236.82	174.92	**83.22**
sanr200_0.9	(200,0.90)	6.11e+06	1.56e+06	7.24e+05	**5.56e+04**	56.74	24.29	4.22	**0.92**
hamming10-2	(1024,0.99)	*	5.97e+02	**1.00e+01**	*	*	**0.10**	0.62	*
MANN_a27	(378,0.99)	*	*	*	**7.68e+03**	*	*	*	**1.17**
		Números de instâncias resolvidas				24	26	29	31

Fonte: Elaborada pelo autor.

$$\chi_f(G, \mathcal{S}') = \min \sum_{S \in \mathcal{S}'} x_S \tag{25}$$

$$\text{s.a.} \sum_{\{S \in \mathcal{S}' : v \in S\}} x_S \leq 1 \quad \forall v \in V \tag{26}$$

$$x_S \geq 0 \qquad \forall S \in \mathcal{S}' \tag{27}$$

onde $\mathcal{S}' \subseteq \mathcal{S}$. $\chi_f(G) = \chi_f(G, \mathcal{S})$ é o número cromático fracionário de G.

Seja (x, π) uma solução primal-dual ótima para (4.6)-(4.8), onde o vetor dual $\pi = (\pi_v)_{v \in V} \in [0, 1]^{|V|}$. Logo, temos duas opções:

1. O par (x, π) é ótimo.
2. π é dual inviável para o problema de coloração fracionária.

O último caso acontece quando existe um conjunto independente $S \in \mathcal{S} \setminus \mathcal{S}'$ tal que

$$\sum_{v \in S} \pi_v > 1 \tag{28}$$

Um conjunto independente satisfaz essa condição se somente se o conjunto independente máximo ponderado por π de G é maior que 1. O valor deste conjunto é dado por:

$$\begin{aligned} \alpha(G, \pi) = \max & \sum_{v \in V} \pi_v y_v \\ \text{s.a. } & y_v + y_u \leq 1 \quad \forall \{u, v\} \in E \\ & y_v \in \{0, 1\} \qquad \forall v \in V \end{aligned}$$

Na literatura, este problema é atacado de maneira exata e/ou heurística utilizando alguns algoritmos de *Branch & Bound* eficientes para o problema da clique máxima ponderada ou, equivalentemente, conjunto independente máximo ponderado. Em Held, Cook, and Sewell (2012), um conjunto de instâncias do problema $\alpha(G, \pi)$ é gerado durante o método de geração de colunas. O conjunto completo de instâncias pode ser obtido em:

`http://code.google.com/p/exactcolors/wiki/MWISInstances`

Em Held, Cook, and Sewell (2012), um conjunto de 25 instâncias do problema $\alpha(G, \pi)$ é utilizado como instâncias desafio para o problema do conjunto independente máximo ponderado (Ver Tabela 10). Essas instâncias podem ser convertidas para instâncias do problema da clique máxima ponderada no grafo complementar $\overline{G}$. A densidade reportada na tabela é do grafo $\overline{G}$. Para o algoritmo ser numericamente seguro,

os pesos duais são escalonados por um valor K. Na prática, isso significa que queremos encontrar conjuntos independentes com peso maior que K. Nos testes realizados em Held, Cook, and Sewell (2012), para as instâncias grandes como C2000.5.1029, DSJC1000.1.3915 e DSJC500.1.117, a computação é finalizada quando um conjunto independente com peso maior que K é encontrado. O algoritmo apresentado pelos autores não conseguiu encontrar tal conjunto independente para essas instâncias grandes em tempo limite de 10h.

Tabela 10 – Instâncias do problema de clique máxima ponderada obtidas a partir das instâncias de conjunto independente máximo ponderado

Instância	**(n,p)**	**K**	**Instância**	**(n,p)**	**K**
1-Insertions_6	(527,0.96)	3537864	2-Insertions_4	(149,0.95)	14412641
2-Insertions_5	(369,0.97)	3597125	3-Insertions_4	(208,0.97)	7642290
3-Insertions_5	(460,0.98)	1527371	4-Insertions_4	(295,0.98)	4521018
C2000.5.1029	(2000,0.50)	1073741	DSJC250.1	(241,0.89)	8589934
DSJC250.5	(250,0.50)	8589934	DSJC250.9	(250,0.10)	8589934
DSJC500.1.117	(494,0.90)	4294967	DSJC500.5	(500,0.50)	4294967
DSJC500.9	(500,0.10)	4294967	DSJC1000.1.3915	(998,0.90)	2147483
DSJC1000.5	(1000,0.50)	2147483	DSJC1000.9	(1000,0.10)	2147483
DSJR500.1c	(189,0.02)	4294967	flat300_28_0	(300,0.52)	7158278
flat1000_50_0	(967,0.51)	2147483	flat1000_60_0	(999,0.51)	2147483
flat1000_76_0	(1000,0.51)	2147483	latin_square_10	(90,0.00)	2386092
r1000.1c	(511,0.03)	2147483	r1000.5	(234,0.00)	2147483
school1	(336,0.71)	5577879			

Fonte: Elaborada pelo autor.

Em nosso experimentos, o tempo limite de resolução para cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Além do tempo de execução, o melhor limite inferior será também reportado. Identificamos em negrito o melhor tempo de execução e o melhor limite inferior.

Dentro do tempo limite, os algoritmos **BITCLIQUE1** e **BITCLIQUE2** encontram os melhores limites inferiores (Ver Tabela 11), à exceção de 2 instâncias para **BITCLIQUE2**. Em particular, para as instâncias *C2000.5.1029*, *DSJC500.1.117* e *DSJC1000.1.3915*, os algoritmos **BITCLIQUE1** e **BITCLIQUE2** encontram uma clique ponderada com peso superior ao valor K. Isso significa que o algoritmo consegue encontrar uma coluna para ser adicionada pelo algoritmo de geração de colunas. O algoritmo HCS não encontra tal coluna no tempo limite de 10h.

Em geral, os algoritmos **BITCLIQUE1** e **BITCLIQUE2** mostram-se competitivos em todas as densidades (Ver Tabela 12). Contudo, o algoritmo **BITCLIQUE1** conseguiu resolver mais instâncias que **BITCLIQUE2**. Quando um dos novos algoritmos não consegue resolver a instância com o melhor tempo, o seu tempo de execução é bastante próximo do melhor tempo de execução. Principalmente quando os tempos dos algoritmos da literatura são maiores, **BITCLIQUE1** e **BITCLIQUE2** conseguem reduzi-los significativamente.

Tabela 11 – Melhor limite inferior encontrado para cada instância Exactcolor

Instancia	**(n,p)**	cutoff	**Melhor Limite Inferior Encontrado**			
			YM	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**
1-Insertions_6	(527,0.96)	3537864	**3532481**	**3532481**	**3532481**	**3532481**
2-Insertions_4	(149,0.95)	14412641	**14412616**	**14412616**	**14412616**	**14412616**
2-Insertions_5	(369,0.97)	3597125	**3054331**	**3054331**	**3054331**	**3054331**
3-Insertions_4	(208,0.97)	7642290	**7018037**	**7018037**	**7018037**	**7018037**
3-Insertions_5	(460,0.98)	1527371	1379326	1393155	**1427097**	1377656
4-Insertions_4	(295,0.98)	4521018	**4286105**	**4286105**	**4286105**	**4286105**
C2000.5.1029	(2000,0.50)	1073741	1105401	1070695	**1128306**	1108370
DSJC250.1	(241,0.89)	8589934	**8226390**	**8226390**	**8226390**	**8226390**
DSJC250.5	(250,0.50)	8589934	**8589932**	**8589932**	**8589932**	**8589932**
DSJC250.9	(250,0.10)	8589934	**8589933**	**8589933**	**8589933**	**8589933**
DSJC500.1.117	(494,0.90)	4294967	4043060	3885643	**4344477**	4308933
DSJC500.5	(500,0.50)	4294967	**4294964**	**4294964**	**4294964**	**4294964**
DSJC500.9	(500,0.10)	4294967	**4294966**	**4294966**	**4294966**	**4294966**
DSJC1000.1.3915	(998,0.90)	2147483	2043939	1935058	**2291982**	2268749
DSJC1000.5	(1000,0.50)	2147483	**2147479**	**2147479**	**2147479**	**2147479**
DSJC1000.9	(1000,0.10)	2147483	**2147482**	**2147482**	**2147482**	**2147482**
DSJR500.1c	(189,0.02)	4294967	**4294967**	**4294967**	**4294967**	**4294967**
flat300_28_0	(300,0.52)	7158278	**7158275**	**7158275**	**7158275**	**7158275**
flat1000_50_0	(967,0.51)	2147483	**2147477**	**2147477**	**2147477**	**2147477**
flat1000_60_0	(999,0.51)	2147483	**2147478**	**2147478**	**2147478**	**2147478**
flat1000_76_0	(1000,0.51)	2147483	**2147479**	**2147479**	**2147479**	**2147479**
latin_square_10	(90,0.00)	2386092	**2386091**	**2386091**	**2386091**	**2386091**
r1000.1c	(511,0.03)	2147483	**2147482**	**2147482**	**2147482**	**2147482**
r1000.5	(234,0.00)	2147483	**2147483**	**2147483**	**2147483**	**2147483**
school1	(336,0.71)	5577879	**5577873**	**5577873**	**5577873**	**5577873**

Fonte: Elaborada pelo autor.

Tabela 12 – Tempo de Execução para cada instância Exactcolor

Instância	**(n,p)**	**Tempo de Execução**			
		YM	**HCS**	**BITCLIQUE1**	**BITCLIQUE2**
1-Insertions_6	(527,0.96)	2242.33236	44.24440	**1.60668**	21.09391
2-Insertions_4	(149,0.95)	0.38999	**0.06504**	0.19627	0.26794
2-Insertions_5	(369,0.97)	6089.65302	85.43386	5.97728	**5.65855**
3-Insertions_4	(208,0.97)	94.30376	0.60412	0.96879	**0.24253**
3-Insertions_5	(460,0.98)	*	*	**1404.59394**	*
4-Insertions_4	(295,0.98)	4083.37884	11.38059	**3.63031**	64.45659
C2000.5.1029	(2000,0.50)	*	*	*	*
DSJC250.1	(241,0.89)	5831.52009	5261.10567	646.72284	**508.14150**
DSJC250.5	(250,0.50)	**0.09572**	0.53295	0.10656	0.10380
DSJC250.9	(250,0.10)	**0.00043**	0.00526	0.07170	0.07037
DSJC500.1.117	(494,0.90)	*	*	*	*
DSJC500.5	(500,0.50)	5.89928	28.16698	**3.21632**	3.22747
DSJC500.9	(500,0.10)	**0.00499**	0.04924	0.13717	0.14236
DSJC1000.1.3915	(998,0.90)	*	*	*	*
DSJC1000.5	(1000,0.50)	693.66427	3091.07580	418.77270	**411.11693**
DSJC1000.9	(1000,0.10)	**0.01523**	0.46503	0.28024	0.28836
DSJR500.1c	(189,0.02)	**0.00028**	0.00280	0.15113	0.15162
flat300_28_0	(300,0.52)	0.44294	1.80078	**0.26329**	0.26556
flat1000_50_0	(967,0.51)	159.55484	673.98914	**53.00945**	93.38235
flat1000_60_0	(999,0.51)	409.66139	1674.89864	**178.62988**	227.20397
flat1000_76_0	(1000,0.51)	863.26174	3940.79916	539.63199	**524.88818**
latin_square_10	(90,0.00)	**0.00006**	0.00119	0.08183	0.08926
r1000.1c	(511,0.03)	**0.00194**	0.02384	0.16896	0.15756
r1000.5	(234,0.00)	**0.00028**	0.00080	0.18398	0.18849
school1	(336,0.71)	39.23673	12.60694	2.12217	**0.59738**

Fonte: Elaborada pelo autor.

4.7 Conclusão

Neste capítulo, apresentamos um algoritmo de Branch & Bound combinatório com um desempenho melhor que os outros algoritmos do estado da arte na mesma categoria, atendendo a uma das principais motivações do nosso trabalho. Os resultados computacionais mostraram que **BITCLIQUE** reduz tanto a quantidade de subproblemas quanto o tempo de execução na maior parte das instâncias testadas.

O principal ativo de **BITCLIQUE** é a heurística de coloração ponderada inteira **BITCOLOR**. Essa heurística consegue o melhor compromisso entre a qualidade do limite superior e o tempo gasto para computá-lo. A qualidade do limite deve-se, em boa parte, a ordem ρ dos vértices adotada. Observe que ela é utilizada a todo momento sem a necessidade de reordenação dos vértices. Já a eficiência computacional é garantida pela utilização de uma estrutura de dados, chamada de vetores de bits (*bitmaps*), usada para representar o grafo e as estruturas relacionadas a ele. Ela possibilita a realização de operações recorrentes com menos instruções. Além disso, ela fornece uma outra ordenação dos vértices π usada pela estratégia de ramificação.

BITCLIQUE1 é mais rápido que os algoritmos anteriores em quase todas as densidades. Essa vantagem cresce à medida que a densidade do grafo aumenta. Ela só é

superada por **BITCLIQUE2**, quando a densidade do grafo ultrapassa 90%.

No próximo capítulo, mostraremos como incorporar a heurística **BITCOLOR** em um algoritmo de Bonecas Russas. Essa adaptação está sendo chamada de **BITRDS**.

5 ALGORITMOS DE BONECA RUSSA

O método de Busca de Bonecas Russas (*Russian Doll Search*) foi proposto originalmente em 1996 para problemas de satisfação de restrições em Verfaillie, Lemae, and Schiex (1996). De maneira independente, Östergård apresentou uma variação do método de Bonecas Russas para um problema de otimização discreta em Ostergard (2002). Recentemente, o método de Bonecas Russas foi aplicado em outros problemas de otimização discreta:

- Problema da clique máxima Ostergard (2002); Corrêa *et al.* (2014).
- Problema da clique máxima ponderada Ostergard (1999); Kumlander (2008); Shimizu *et al.* (2012).
- Problema da cobertura tripla de Steiner Ostergard and Vaskelainen (2011).
- Problema do subtorneio transitivo máximo Kiviluoto, terg, and Vaskelainen (2014).
- Problema do melhor barbeque combinatórioOstergard, Vaskelainen, and Mosig (2007).
- Problema s-plex máxima McClosky and Hicks (2012); Trukhanov *et al.* (2013); Gschwind *et al.* (2015).
- Problema da clique s-deficiente máxima Trukhanov *et al.* (2013); Gschwind *et al.* (2015).
- Problema s-feixo máximo Gschwind *et al.* (2015).
- Problema da clique máxima probabilística Miao, Balasundaram, and Pasiliao (2014).
- Problema da clique máxima ponderada com restrição de agrupamentos Malladi (2014).

De maneira geral, o método consiste em resolver iterativamente subproblemas cada vez maiores (também chamados de **bonecas**) até chegar ao problema completo. As bonecas são geradas de maneira que uma esteja contida na próxima, o que sugere o nome do método. Por exemplo, uma boneca pode ser definida por um subproblema contendo menos variáveis (ou menos restrições) que aquele relativo à próxima.

A eficiência do método de Bonecas Russas depende também da utilização do conhecimento acumulado durante a resolução de subproblemas menores (bonecas menores), de modo a reduzir o tamanho da árvore de busca empregada na resolução dos subproblemas maiores (bonecas maiores). Com a ajuda das bonecas, podemos definir uma relação de dominação entre subproblemas que permite podar os subproblemas dominados. Dizer que um subproblema S_1 domina um outro subproblema S_2 significa que, para qualquer solução de S_2, existe uma solução melhor ou igual em S_1. A partir dessa relação, podemos definir regras de poda que utilizem eficientemente o conhecimento previamente obtido.

A estrutura geral do método de Busca das Bonecas Russas pode ser separada em duas partes:

- Estratégia de enumeração das bonecas.
- Estratégia de solução das bonecas.

Para o problema da clique (ponderada) máxima Ostergard (1999, 2002) , Östergård propõe a seguinte implementação dessas estratégias. A enumeração das bonecas baseia-se em uma ordem inicial dos vértices. Uma boneca corresponde ao problema da clique ponderada máxima no subgrafo que inclui um vértice a mais seguindo essa ordem. Cada boneca é resolvida por um algoritmo de *Branch & Bound*, fortalecido por uma estratégia de poda obtida pela relação entre os subproblemas.

Em Trukhanov *et al.* (2013), mostra-se que o método de Busca de Bonecas Russas pode ser estendido para qualquer propriedade Π que seja hereditária nos subgrafos induzidos, ou seja, para qualquer $S \subseteq V$ tal que $G[S]$ satisfaz a propriedade Π, a remoção de qualquer subconjunto de vértices em $G[S]$ não produz um grafo que viola a propriedade Π.

Várias propriedades em grafos são hereditárias, tais como: clique, conjunto independente, bipartido, acíclico, perfeito, planar entre outros. Trukhanov apresenta uma extensão do algoritmo das Bonecas Russas para o problema s-plex máxima e o problema da clique s-deficiente máxima. De maneira geral, esses dois problemas são relaxações do problema da clique máxima e/ou conjunto independente máximo.

Em Corrêa *et al.* (2014), apresenta-se um algoritmo de Busca das Bonecas Russas para o problema da clique máxima que apresenta duas modificações importantes em relação ao algoritmo de Östergård:

- O processo de enumeração de bonecas é alterado para que um número significativo de bonecas deixem de ser resolvidas devido à adoção de uma regra de eliminação.
- Cada boneca é resolvida por um algoritmo de enumeração recursivo, baseado nos princípios do método das Bonecas Russas. Mais precisamente, os subproblemas associados às bonecas também são resolvidos por chamadas ao método de Bonecas Russas.

5.1 Estrutura Geral para CLIQUE PONDERADA

Nesta seção, apresentamos a estrutura geral do método das Bonecas Russas através do algoritmo proposto em Ostergard (1999) e Ostergard (2002) (Ver Algoritmo 5.1). Inicialmente, uma ordem dos vértices ρ é definida. Vamos usar a seguinte notação $\rho = (\rho_1, \rho_2, \ldots, \rho_n)$ para falar sobre o conjunto de vértices ordenados. Essa ordem é utilizada para o processo de geração das bonecas. Cada boneca está associado ao subgrafo $G_i = (V_i, E_i)$, $i \in \{1, \ldots, n\}$, onde $V_1 = \{\rho_1\}$ e $V_{i+1} = V_i \cup \{\rho_{i+1}\}$, $E_i = E[V_i]$.

O Algoritmo 5.1 resolve cada uma das n bonecas, ou seja, encontra a clique máxima ponderada em cada subgrafo G_i, para $i \in \{1, \ldots, n\}$; seu peso é armazenado em $c[\rho_i]$. O problema da i-ésima boneca divide-se em verificar se existe ou não uma clique máxima ponderada em G_i com o vértice ρ_i. Sabemos que o peso da clique máxima ponderada sem o vértice ρ_i em G_i é $c[\rho_{i-1}]$. Assim, o problema da clique ponderada

máxima em G_i se resume a encontrar a clique máxima ponderada em $G[V_{i-1} \cap N(\rho_i)]$ ou mostrar que seu peso é no máximo $c[\rho_{i-1}] - w(\rho_i)$. No Algoritmo 5.1, este problema é resolvido pela chamada $RDS(\{\rho_i\}, U \cap N(\rho_i), W)$, onde U mantém os iterados V_i e W representa um limite inferior para a próxima boneca, que passa a receber o peso da clique máxima em G_i, caso este seja maior.

De maneira geral, a ordem utilizada em Ostergard (2002) para CLIQUE e a utilizada em Ostergard (1999) para CLIQUE PONDERADA tentam limitar o crescimento de $c[\rho_i]$ durante o processo de enumeração.

Algorithm 5.1 Algoritmo de enumeração das bonecas

```
function INICIAL
    ρ ← (ρ_1, ..., ρ_n)
    c[ρ_1] ← w(ρ_1)
    W ← c(ρ_1)
    U ← {ρ_1}
    for i ← 2 até n do
        RDS({ρ_i}, U ∩ N(ρ_i), W)
        c[ρ_i] ← W
        U ← U ∪ {ρ_i}
```

Segundo Östergård Ostergard (1999), é um problema em aberto estudar como diferentes ordens dos vértices afetam o desempenho do Algoritmo 5.1 e determinar quais são as características de uma boa ordem.

Em Ostergard (2002), a ordem inicial dos vértices baseada em uma heurística de coloração mostrou-se experimentalmente eficiente para a clique máxima. Nessa coloração, as classes de cores são construídas uma por vez, adicionando sempre o vértice com o maior grau no grafo não colorido e numerando estes vértices como $(\rho_1, \ldots, \rho_n)$.

Já para a clique máxima ponderadaOstergard (1999), a ordem dos vértices baseada em seus pesos(com um critério de desempate) obteve os melhores resultados experimentais. Nessa ordem, ρ_n é o vértice com o maior peso. Em caso de empate, será escolhido o vértice com a maior soma dos pesos de seus vizinhos. Iterativamente, para $i = n-1, \ldots, 1$, o vértice ρ_i é selecionado aplicado a mesma regra em G_i.

Em Ostergard (2002), cada boneca é resolvida por um algoritmo de *Branch & Bound* implementado pela função RDS do Algoritmo 5.1. Um subproblema na árvore de *Branch & Bound* é definido pela tripla (C, P, LB), onde LB é um limite inferior, dado pelo peso da clique mais pesada encontrada até o momento, C é a clique atual e P é o conjunto de vértices que podem entrar na clique atual chamado de conjunto candidato. Particularmente, o subproblema da raiz da árvore de *B & B* associada à boneca $RDS(\{\rho_i\}, U \cap N(\rho_i), W)$ é dado por $C = \{\rho_i\}$, $P = U \cap N(\rho_i)$ e $LB = W$.

Esse algoritmo *Branch & Bound* emprega duas regras de poda e uma estratégia de ramificação, baseada na ordem ρ (Ver Algoritmo 5.2).

A primeira regra de poda utiliza o peso do conjunto P. O subproblema (C, P, LB) pode ser podado, se $w(C) + w(P) \leq LB$.

A segunda regra de poda utiliza o vetor de soluções $c[.]$, gerado ao longo do processo, como segue.

Lema 5 *Para um subproblema (C, P, LB), se existe um j tal que $P \subseteq V_j$ e $c(\rho_j) + w(C \setminus V_j) \leq LB$ então toda clique C' tal que $C' \subseteq (C \cup P)$ tem $w(C') \leq LB$.*

Prova Seja $C' \subseteq C \cup P$. Então $C' \subseteq C \cup V_j$ e, conseqüentemente, $C' = (C' \cap V_j) \cup (C' \cap (C \setminus V_j))$. Como essa partição é disjunta, temos que $w(C') = w(C' \cap V_j) + w(C' \cap (C \setminus V_j)) \leq c(\rho_j) + w(C \setminus V_j) \leq LB$.

Particularmente, no subproblema $(\{\rho_i\}, V_{i-1} \cap N(\rho_i), c(\rho_{i-1}))$, temos que $P \subseteq V_{i-1}$. Logo, sempre existe $j \leq i - 1$ tal que $P \subseteq V_j$. O índice j que minimiza $c(\rho_j)$ é dado pelo último vértice de P seguindo a ordem ρ. Neste caso, $C \setminus V_j = C$. Então, se $w(C) + c(\rho_j) \leq LB$, o subproblema (C, P, LB) pode ser podado.

A efetividade da primeira regra de poda depende de uma redução rápida do conjunto candidato durante o Branch & Bound. A efetividade da segunda regra depende de que $c[\rho_i]$ seja o menor possível para cada ρ_i. Esses dois objetivos são influenciadas diretamente pela ordem inicial dos vértices.

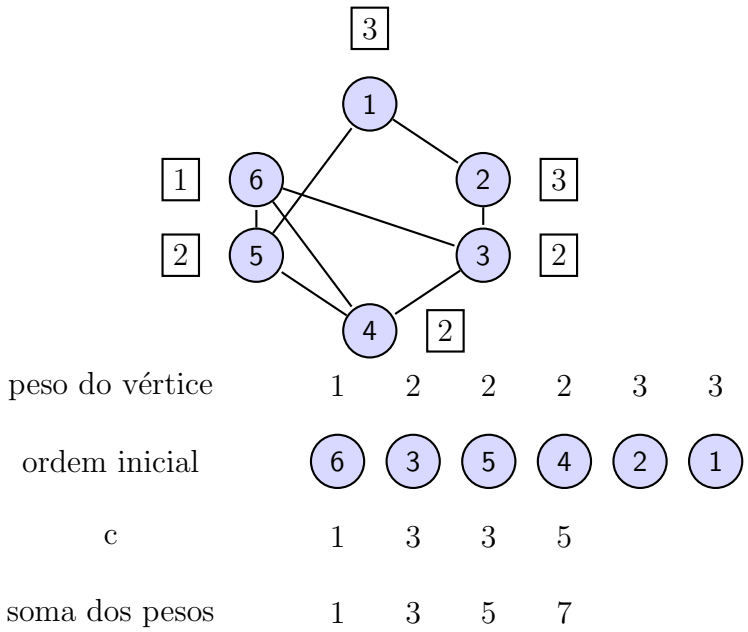
A ramificação é definida pela escolha de um vértice v em P para ser adicionado à clique atual C. A estratégia sugerida por Ostergard (2002) é selecionar o último vértice de P seguindo a ordem inicial ρ. Depois que todas as cliques $C' \subseteq ((C \cup \{v\}) \cup (P \cap N(v)))$ forem enumeradas implicitamente pela chamada recursiva $RDS(C \cup \{v\}, P \cap N(v), LB)$ no Algoritmo 5.2, o vértice v pode ser removido do conjunto P. Observe que a ordem ρ não é passada explicitamente no procedimento RDS, mas ela é importante para a estratégia de ramificação em RDS.

Algorithm 5.2 Algoritmo de resolução das bonecas.

```
function RDS(C, P, LB)
    if w(C) > LB then
        LB ← w(C)
    while P ≠ ∅ do
        if w(C) + w(P) ≤ LB then
            return
        j ← max{i|ρ_i ∈ P}
        if w(C) + c[ρ_j] ≤ LB then
            return
        RDS(C ∪ {ρ_j},P ∩ N(ρ_j), LB)
        P ← P \ {ρ_j}
```

A Figura 18 ilustra os limites superiores para a clique máxima ponderada em cada grafo G_i, dados pela soma dos pesos dos vértices em G_i e pelo valor $c[\rho_i]$ relativos aos quatro primeiros vértices da ordem ρ utilizada.

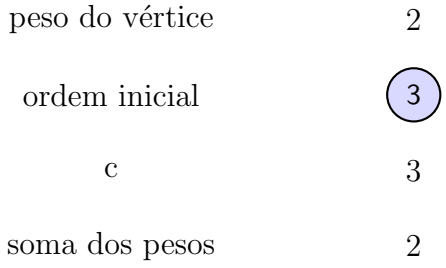
Figura 18 – Limites superiores dados pela soma dos pesos dos vértices e pelos valores de $c[i]$ para cada G_i, para $i = 1, \ldots, 4$.



Fonte: Elaborada pelo autor.

Seguimos agora a resolução das duas últimas bonecas pelo Algoritmo 5.2. A penúltima boneca é resolvida pela chamada $RDS(\{2\}, \{6,3,5,4\} \cap N(2), 5)$ no Algoritmo 5.1. Precisamos encontrar uma clique máxima ponderada em $G[\{6,3,5,4\} \cap N(2)]$ com um limite inferior igual a $c(4) - w(2)$, ou seja, 2 (lembrando que o limite inferior inicial da clique ponderada máxima de $G[\{6,3,5,4,2\}]$ é dado por $c[4] = 5$). O vértice 2 é vizinho apenas do vértice 3. Logo, $G[\{6,3,5,4\} \cap N(2)] = G[\{3\}]$. Os limites superiores disponíveis para este subproblema são:

Figura 19 – Limites superiores dados pela soma dos pesos dos vértices e pelos valores de $c[]$ para a resolução da penúltima boneca.



Fonte: Elaborada pelo autor.

Neste subproblema, $w(C) + w(P) = w(2) + w(3) = 2 + 3 \leq LB = 5$, então o subproblema pode ser podado, e o valor limite inferior não é alterado.

A última boneca é resolvida pela chamada $RDS(\{1\}, \{6,3,5,4,2\} \cap N(1), 5)$ no Algoritmo 5.1. Precisamos encontrar uma clique máxima ponderada em $G[\{6,3,5,4,2\} \cap N(1)]$ com um limite inferior igual a $c[2] - w(1)$, ou seja, 2 (lembrando que o limite inferior inicial da clique ponderada máxima de $G[\{6,3,5,4,2,1\}]$ é dado por $c[2] = 5$). O vértice

1 é vizinho apenas dos vértices 5, 2. Logo, $G[\{6,3,5,4,2\} \cap N(1)] = G[\{5,2\}]$. Os limites superiores disponíveis para este subproblema são:

Figura 20 – Limites superiores dados pela soma dos pesos dos vértices e pelos valores de $c[]$ para a resolução da última boneca.

peso do vértice	2	3
ordem inicial	5	2
c	3	5
soma dos pesos	2	5

Fonte: Elaborada pelo autor.

Neste subproblema, $w(C)+w(P) = 3+5 > LB$ e $w(C)+c[2] = 3+5 > LB$, logo o subproblema não é podado por nenhum critério de poda. O vértice 2 é escolhido pela regra de ramificação, resultando na chamada recursiva $RDS(\{1,2\}, \{5,2\} \cap N(2), LB)$. O limite inferior é atualizado com o peso da clique $\{1,2\}$, que é igual a 6, e o subproblema é podado, pois $P = \{5,2\} \cap N(2) = \emptyset$. O vértice 2 é então removido de P, retornando ao início do laço, agora os dois critérios de poda são satisfeitos; pois $LB = 6$ e $P = \{5\}$, de modo que os limites superiores são os seguintes:

peso do vértice	2
ordem inicial	5
c	3
soma dos pesos	2

5.2 Algoritmo BITRDS

Nesta seção, propomos um algoritmo de Bonecas Russas para o problema da CLIQUE PONDERADA. Nosso algoritmo, o processo de enumeração das bonecas segue o Algoritmo 5.1, porém o processo de resolução do problema associado a cada boneca é feito de forma diferente do Algoritmo 5.2. Embora também empreguemos um algoritmo de Branch & Bound, agora com uma heurística de coloração ponderada inteira será utilizada tanto para estabelecer o critério de poda como para definir o conjunto de ramificação. A ordem de ramificação continua sendo aquela definida pela ordem inicial ρ. Detalhamos a seguir nosso algoritmo.

Como no Algoritmo 5.1, uma ordem inicial dos vértices ρ é definida. Em nosso caso, vamos utilizar as duas ordens já apresentadas no Capítulo 4. Esta ordem inicial será utilizada para definir os vértices associados a cada boneca. Precisamente, cada boneca

está associada ao subgrafo $G_i = (V_i, E_i)$, $i \in \{1, \ldots, n\}$, onde $V_1 = \{\rho_1\}$,$V_{i+1} = V_i \cup \{\rho_{i+1}\}$ e $E_i = E[V_i]$. A criação das bonecas segue pois o Algoritmo 5.1, partindo entretanto de outras ordens.

Como antecipado, cada boneca é resolvida por um *B & B* que utiliza a coloração ponderada inteira para definir uma regra de poda e/ou o conjunto de ramificação.

A heurística de coloração apresentada no Algoritmo 5.3 usa dois parâmetros: R é um vetor de bits e K é um valor inteiro. O parâmetro R devolve o conjunto de ramificação e é inicializado com P (o conjunto de candidatos). O parâmetro K mantém o peso ainda disponível para a coloração, sendo inicializado com $LB - w(C)$ (o limite inicial máximo).

O Algoritmo 5.3 apresenta um procedimento da heurística de coloração ponderada inteira que contempla essas duas utilizações da coloração ponderada.

A cada iteração, um conjunto independente ponderado S_i será construído, até se atingir o limite de peso da coloração ou se colorir P completamente. O vetor de bits Q representa os vértices que podem ser adicionados ao conjunto independente ponderado S_i atual. Os vértices a serem adicionados a S_i são selecionados do vetor de bits Q, seguindo a ordenação inicial dos vértices ρ. Cada vez que um vértice v é selecionado, o conjunto independente S_i é atualizado $S_i \leftarrow S_i \cup \{v\}$. Os vértices adjacentes a v e o próprio v são removidos do vetor de bits Q, ou seja, $Q \leftarrow Q \setminus N(v); Q \leftarrow Q \setminus \{v\}$. A construção de S_i continua até Q tornar-se vazio, quando S_i será um conjunto independente maximal. O peso de S_i é dado pelo menor peso residual de seus vértices ou valor corrente de K, o que for menor. O valor de K e o peso residual de cada vértice v de S_i é atualizado, decrementando-os do peso da classe de cor S_i. Os vértices com peso residual igual a zero podem ser removidos de R. A heurística de coloração ponderada termina quando K torna-se zero ou R torna-se vazio.

Os bits de todos os vetores de bits são organizados seguindo a ordem ρ. Logo, por exemplo, a instrução ŽŽselecione o primeiro vértice na ordem inicial que está em Q" poderia ser substituída por ŽŽselecione o primeiro bit setado em 1 do vetor de bits Q". As instruções em caixas são substituídas por instruções específicas que manipulam diretamente os vetores de bits.

Se, ao final do Algoritmo 5.3, o conjunto R for igual a $\emptyset$, então o peso da coloração ponderada inteira de P é menor igual a $LB - w(C)$. Logo, o subproblema (C, P, LB) pode ser podado.

O uso da heurística de coloração leva à modificação da função RDS (Algoritmo 5.2), que passamos a chamar de **BITRDS**, conforme descrição no Algoritmo 5.4. Incluímos uma segunda regra de poda, alteramos a definição do conjunto de ramificação, porém mantemos a estratégia de ramificar o último vértice do conjunto seguindo a ordem ρ. Observe que essa combinação consegue aproveitar tanto a regra de poda definida pela vetor c quanto pela heurística de coloração ponderada.

Algorithm 5.3 Heurística de Coloração Ponderada Inteira Modificada

function ColoraçãoRestrita(R,K)
  for $v \in V$ **do**
    $res(v) \leftarrow w(v)$
  $i \leftarrow 1$
  while $K > 0$ e $R \neq \emptyset$ **do**
    $S_i \leftarrow \emptyset$
    $Q \leftarrow R$
    while $Q \neq \emptyset$ **do**
      $v \leftarrow$ Selecione o primeiro elemento de Q.
      $Q \leftarrow Q \setminus N(v)$
      $Q \leftarrow Q \setminus \{v\}$
      $S_i \leftarrow S_i \cup \{v\}$
    $min\_res \leftarrow \min\{res(v) | v \in S_i\}$
    $y(S_i) \leftarrow \min(min\_res, K)$
    $K \leftarrow K - y(S_i)$
    for $v \in S_i$ **do**
      $res(v) \leftarrow res(v) - y(S_i)$
      if $res(v) = 0$ **then**
        $R \leftarrow R \setminus \{v\}$
    $i \leftarrow i + 1$

Algorithm 5.4 Algoritmo de Bonecas Russas **BITRDS**.

function BITRDS(C, P, LB)
  if $w(C) > LB$ **then**
    $LB \leftarrow w(C)$
  $R \leftarrow P$
  ColoraçãoRestrita($R, LB - w(C)$)
  while $R \neq \emptyset$ **do**
    $v \leftarrow$ Selecione o último vértice de R seguindo a ordem ρ
    $R \leftarrow R \setminus \{v\}$
    $u \leftarrow$ Selecione o último vértice de P seguindo a ordem ρ
    if $w(C \setminus P) + c[u] \leq LB$ **then**
      retorne
    $P \leftarrow P \setminus \{v\}$ ▷ remova o bit ρ_v de P
    $newP \leftarrow P \cap N(v)$
    $BITRDS(C \cup \{v\}, newP, LB)$

5.3 Resultados Computacionais

Neste seção, compararamos o desempenho computacional do nosso algoritmo de Bonecas Russas, identificado como BITRDS, com o algoritmo CLIQUER, proposto por Östergård, disponibilizado em `http://tcs.legacy.ics.tkk.fi/~pat/wclique.html`, e com o algoritmo BITCLIQUE, apresentado no capítulo anterior.

Todos os testes foram realizados num computador equipado com processador *Intel Core i7-2600K 3.40Ghz*, *8Mb de cache*, *6Gb de memória*, utilizando sistema operacional *Linux*.

Vamos comparar as nossas duas versões do algoritmo de Bonecas Russas, dada pelas duas ordens iniciais dos vértices diferentes. O algoritmo de Bonecas Russas que utiliza a ordem **menor peso primeiro** será denotado por **BITRDS1**, enquanto o que usa a ordem inicial **maior grau ponderado primeiro** será denotado por **BITRDS2**.

Novamente, avaliamos o desempenho do nosso algoritmo de Bonecas Russas utilizando as principais instâncias disponíveis na literatura:

- Grafos Aleatórios
- DIMACS-W
- ExactColor

5.3.1 Grafos Aleatórios

Para o nosso experimento, geramos 10 grafos aleatórios para cada combinação n (número de vértices) e p (probabilidade de existência de cada aresta). Os pesos atribuídos aos vértices variam entre 1 e 10. Como é usual na literatura, o número de vértices das instâncias diminui à medida que a probabilidade de existência de aresta aumenta. Consi-

deramos 20 combinações, levando a 200 instâncias no total. Para cada par (n, p), calculamos a média do tempo consumido pelas 10 instâncias para cada algoritmo.

O tempo de limite de resolução para cada instância é 1h (3600s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada na tabela como $*^x$, onde $x \in [1, 10]$ significa o número de instâncias não resolvidas no tempo limite. Identificamos em negrito o melhor desempenho médio em cada caso.

5.3.2 Número de subproblemas

Uma comparação direta do algoritmo **BITRDS1** com **CLIQUER**(Ver Tabela 13) revela que o primeiro reduz o número médio de subproblemas em pelo menos uma ordem de magnitude (exceto para densidade 0.10), chegando até 3 ordens de magnitude com relação **CLIQUER**(densidade 0.90). Além disso, **BITRDS1** resolve mais instâncias que **CLIQUER**. Essa redução torna-se mais significativa à medida que a densidade do grafo aumenta. Porém, os algoritmos **CLIQUER** e **BITRDS1** são superados por **BITCLIQUE1** com relação ao número de subproblemas.

Uma comparação direta entre algoritmo **BITRDS2** e **CLIQUER**(Ver Tabela 13) evidencia que **RDS2** também reduz o número de subproblemas gerados. Contudo, a redução obtida por **BITRDS2** é inferior à redução alcançada por **BITRDS1** para as instâncias até 80%. A partir de 80%, a redução alcançada por **RDS2** supera aquela alcançada por **BITRDS1**. Novamente, os algoritmos de Bonecas Russas (**CLIQUER** e **RDS2**) são ultrapassados pelo algoritmo de Branch & Bound **BITCLIQUE2**.

Tabela 13 – Média dos números de subproblemas para grafos aleatórios gerados por par (n, p)

Instancia	**Número médio de subproblemas**				
(n,p)	**BITCLIQUE1**	**BITCLIQUE2**	**CLIQUER**	**BITRDS1**	**BITRDS2**
(2500,0.10)	**7.22e+02**	1.20e+03	6.41e+04	4.81e+04	2.88e+04
(5000,0.10)	**2.52e+03**	3.72e+03	3.68e+05	2.89e+05	3.91e+05
(2500,0.20)	1.54e+04	**1.07e+04**	1.09e+06	2.89e+05	3.43e+05
(5000,0.20)	**2.12e+05**	4.25e+05	1.25e+07	5.66e+06	6.98e+06
(2500,0.30)	**2.11e+05**	3.16e+05	1.77e+07	3.57e+06	9.45e+06
(5000,0.30)	**7.48e+06**	1.15e+07	4.78e+08	1.04e+08	2.49e+08
(1200,0.40)	**1.03e+05**	1.15e+05	7.19e+06	1.26e+06	2.50e+06
(2500,0.40)	**5.13e+06**	1.19e+07	4.41e+08	6.38e+07	1.75e+08
(600,0.50)	**3.95e+04**	4.06e+04	2.76e+06	3.88e+05	5.40e+05
(1200,0.50)	**1.95e+06**	3.36e+06	1.51e+08	1.60e+07	4.78e+07
(600,0.60)	**4.84e+05**	6.07e+05	4.66e+07	3.55e+06	6.52e+06
(1200,0.60)	**6.91e+07**	$*^{10}$	$*^{8}$	4.01e+08	$*^{10}$
(400,0.70)	5.67e+05	**5.33e+05**	7.52e+07	3.23e+06	4.91e+06
(500,0.70)	**3.62e+06**	4.42e+06	5.77e+08	1.93e+07	3.97e+07
(600,0.70)	**1.31e+07**	1.82e+07	2.44e+09	7.37e+07	1.82e+08
(300,0.80)	1.81e+06	**1.22e+06**	6.63e+08	9.64e+06	8.69e+06
(400,0.80)	2.73e+07	**2.28e+07**	$*^{4}$	1.15e+08	1.88e+08
(200,0.90)	1.03e+06	**8.84e+04**	2.79e+09	5.66e+06	5.78e+05
(300,0.90)	2.40e+08	**3.47e+07**	$*^{10}$	$*^{1}$	2.70e+08
(200,0.95)	5.30e+06	**3.74e+04**	$*^{10}$	2.99e+07	2.15e+05

Fonte: Elaborada pelo autor.

5.3.3 Tempo de Execução

Os tempos de execução (em segundos) para cada algoritmo estão apresentados na Tabela 14. Para as instâncias até 40%, **CLIQUER** é mais rápido que os demais. Nessa faixa de densidade, a vantagem de **CLIQUER** aumenta à medida que a quantidade de vértices aumenta. Além disso, **CLIQUER** chega a ser mais de 2 vezes mais rápido que o segundo melhor algoritmo (**BITRDS1**).

Entre 50% e 80%, o algoritmo **BITRDS1** assume a dianteira. Nessa faixa, **BITRDS1** chega a ser no máximo 2 vezes mais rápido que o segundo melhor **BITCLIQUE1**. Para a densidade maior ou igual 90%, o algoritmo **BITCLIQUE2** vence, sendo o segundo melhor **BITRDS2**.

Tabela 14 – Média dos tempos de execução para os grafos aleatórios gerados por par (n, p)

Instancia	**Tempo de Execução**				
(n,p)	**BITCLIQUE1**	**BITCLIQUER2**	**CLIQUER**	**BITRDS1**	**RDS2**
(2500,0.10)	0.80	0.88	**0.10**	0.21	0.18
(5000,0.10)	3.32	5.13	**0.59**	2.03	3.19
(2500,0.20)	1.99	3.60	**0.61**	1.14	2.32
(5000,0.20)	41.76	83.11	**9.86**	36.68	66.89
(2500,0.30)	22.63	57.79	**9.21**	15.96	51.56
(5000,0.30)	1230.47	3358.41	**336.84**	771.58	2847.88
(1200,0.40)	4.79	10.64	3.68	**3.48**	9.58
(2500,0.40)	443.90	1216.67	**256.77**	313.20	1087.55
(600,0.50)	0.91	1.46	1.12	**0.73**	1.37
(1200,0.50)	76.17	194.71	81.16	**50.39**	187.93
(600,0.60)	9.40	18.65	19.49	**7.40**	17.63
(1200,0.60)	2608.70	> 3600	> 3600	**1469.30**	> 3600
(400,0.70)	6.77	11.10	25.11	**5.66**	10.96
(500,0.70)	54.01	101.71	204.14	**39.95**	101.11
(600,0.70)	248.48	553.79	942.94	**173.64**	546.67
(300,0.80)	15.39	17.34	177.40	**15.32**	18.02
(400,0.80)	311.87	441.60	> 3600	**237.26**	470.21
(200,0.90)	5.60	**0.98**	509.71	8.08	1.16
(300,0.90)	2279.47	**591.86**	> 3600	> 3600	707.35
(200,0.95)	24.01	**0.41**	> 3600	49.60	0.52

Fonte: Elaborada pelo autor.

5.3.4 DIMACS-W

As instâncias DIMACS-W são gerados a partir de grafos do desafio DIMACS, com pesos dos vértices definidos do intervalo [1..200]. Os resultados dos experimentos computacionais com as instâncias desse grupo podem ser vistos nas Tabelas 15 e 16 . Agora, o tempo limite de resolução para cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Identificamos em negrito o melhor desempenho em cada caso.

5.3.5 Número de Subproblemas

Comparando apenas os algoritmos de Bonecas Russas com relação ao número de subproblemas(Ver Tabela 15), temos os seguintes casos:

- Para as instâncias com densidade entre 65% e 90%(com exceção da família *p-hat*), **BITRDS1** vence **BITRDS2**, seguido por **CLIQUER**.
- Para as instâncias com densidade superior 90% e da família *p-hat*, **RDS2** supera **BITRDS1**, seguido por **CLIQUER**.

A diferença no número de subproblemas é bem mais expressiva quando comparamos cada um dos 2 algoritmos com **CLIQUER** do que em uma comparação entre os dois. O algoritmo **BITRDS1** resolve 7 instâncias que **CLIQUER** não foi capaz de resolver em 2h (*san1000, san400_0.7_1, san400_0.7_2, san400_0.7_3, p_hat500-3, san200_0.9_1 e san400_0.9_1*). Para as instâncias que ambos conseguem resolver dentro do tempo li-

mite, a redução no número de subproblemas chega a ser de 4 ordens de magnitude na instância *hamming10-2*, levando a uma redução no tempo de processamento de 716.315s para 1.81s (Tabela 16)

Já **RDS2** resolve 11 instâncias não resolvidas por **CLIQUER** dentro do tempo limite considerado (*san1000, san400_0.7_1, san400_0.7_2, san400_ 0.7_ 3, p_hat1500-2, p_hat500-3, gen400_ p0.9_ 55, gen400_ p0.9_ 75, san200_0.9_1,san400_0.9_1 e MANN_a27*). Para as instâncias em ambos conseguem resolver dentro do tempo limite, a redução no número de subproblemas chega a ser 5 ordens de magnitude na instância *gen200_p0.9_55*, levando a uma redução no tempo de processamento de 885.36s para 0.26s (Tabela 16).

Comparando todos os algoritmos entre si, vemos que os algoritmos de *Branch & Bound* **BITCLIQUE1** e **BITCLIQUE2** ainda geram menos subproblemas os demais algoritmos de Bonecas Russas.

Tabela 15 – Números de subproblemas gerados para as instâncias DIMACS-W

Instância		**Números de Subproblemas**				
Nome	**(n,p)**	**BITCLIQUER1**	**BITCLIQUER2**	**CLIQUER**	**BITRDS1**	**RDS2**
sanr400_0.5	(400,0.50)	**3.74e+03**	4.05e+03	2.88e+05	5.22e+04	4.13e+04
san1000	(1000,0.50)	3.06e+03	**2.35e+03**	*	1.03e+05	4.05e+04
brock400_1	(400,0.75)	1.96e+06	**1.76e+06**	5.14e+08	1.14e+07	1.94e+07
brock400_2	(400,0.75)	**2.28e+06**	2.51e+06	7.05e+08	1.69e+07	2.58e+07
brock400_3	(400,0.75)	**1.77e+06**	1.85e+06	5.44e+08	1.11e+07	1.55e+07
brock400_4	(400,0.75)	1.04e+06	**7.56e+05**	5.54e+08	1.36e+07	3.24e+07
brock800_1	(800,0.65)	**8.77e+06**	1.64e+07	2.23e+09	9.85e+07	2.44e+08
brock800_2	(800,0.65)	**1.62e+07**	3.07e+07	2.34e+09	1.01e+08	3.41e+08
brock800_3	(800,0.65)	**1.15e+07**	2.12e+07	2.44e+09	1.04e+08	2.56e+08
brock800_4	(800,0.65)	**1.78e+07**	3.38e+07	2.86e+09	1.19e+08	3.84e+08
sanr400_0.7	(400,0.70)	3.46e+05	**3.46e+05**	6.06e+07	2.83e+06	3.88e+06
san400_0.7_1	(400,0.70)	**4.30e+04**	9.19e+04	*	3.85e+05	6.77e+05
san400_0.7_2	(400,0.70)	**6.45e+04**	7.20e+04	*	1.04e+06	2.29e+06
san400_0.7_3	(400,0.70)	4.66e+04	**7.85e+03**	*	1.19e+06	1.37e+05
p_hat500-2	(500,0.50)	1.83e+04	**5.32e+02**	1.43e+07	4.01e+05	6.76e+03
p_hat700-2	(700,0.50)	7.59e+05	**2.07e+03**	4.29e+08	6.00e+06	2.24e+04
p_hat1000-2	(1000,0.49)	1.67e+07	**8.48e+04**	1.07e+10	9.76e+07	9.12e+05
p_hat1500-2	(1500,0.51)	*	**3.20e+06**	*	*	3.74e+07
p_hat300-3	(300,0.74)	4.66e+04	**2.64e+03**	4.70e+07	6.05e+05	2.26e+04
p_hat500-3	(500,0.75)	1.59e+07	**1.05e+05**	*	1.00e+08	1.10e+06
gen200_p0.9_44	(200,0.90)	4.23e+05	**1.73e+04**	1.89e+09	2.39e+06	1.38e+05
gen200_p0.9_55	(200,0.90)	1.12e+05	**4.22e+03**	4.68e+09	5.03e+06	9.73e+04
gen400_p0.9_55	(400,0.90)	*	**1.32e+08**	*	*	1.42e+09
gen400_p0.9_65	(400,0.90)	*	*	*	*	*
gen400_p0.9_75	(400,0.90)	3.48e+08	**9.03e+06**	*	*	7.48e+08
C250.9	(250,0.90)	2.30e+06	**2.15e+05**	1.18e+10	2.37e+07	2.49e+06
san200_0.9_1	(200,0.90)	6.26e+03	**2.60e+01**	*	4.65e+05	5.48e+02
san200_0.9_2	(200,0.90)	2.29e+04	**6.54e+03**	8.43e+08	5.64e+05	3.66e+06
san200_0.9_3	(200,0.90)	1.14e+06	**9.52e+04**	7.14e+09	6.31e+06	7.19e+05
san400_0.9_1	(400,0.90)	6.59e+06	**2.33e+06**	*	1.06e+08	5.38e+08
sanr200_0.9	(200,0.90)	7.24e+05	**5.56e+04**	3.11e+09	7.11e+06	3.59e+05
hamming10-2	(1024,0.99)	**1.00e+01**	*	1.44e+09	3.41e+05	*
MANN_a27	(378,0.99)	*	**7.68e+03**	*	*	2.58e+04

Fonte: Elaborada pelo autor.

5.3.6 Tempo de Execução

As instâncias com densidade superior a 90% (com exceção de *hamming10-2*) e aquelas da família *p-hat* são mais rapidamente resolvidas quando usamos a segunda ordem (**maior**

grau ponderado primeiro). Embora, nem sempre melhor que **BITRDS2**, o desempenho médio de **BITCLIQUE2** é superior. Vale salientar aqui que a ordem **maior grau ponderado primeiro** compromete o poder de poda do vetor c, uma vez que o valor das soluções armazenadas cresce rapidamente.

Nas outras instâncias, temos que os algoritmos que usam a primeira ordem, **BITCLIQUE1** e **BITRDS1**, são mais rápidos que os demais. Da mesma maneira, nem sempre **BITCLIQUE1** é superior a **BITRDS1**, contudo seu melhor desempenho médio prevalece.

Percebemos também que **BITRDS1** resolve 7 instâncias, que não são resolvidas por **CLIQUER** em 2h, em menos de 600s, enquanto, **BITRDS2** resolve 11 instâncias não resolvidas por CLIQUER.

Em geral, na maioria das instâncias, qualquer um dos quatro algoritmos demanda menos tempo de computação que **CLIQUER**.

Tabela 16 – Tempos de execução para as instâncias DIMACS-W

Instância		**Tempo de Execução**				
Nome	**(n,p)**	**BITCLIQUER1**	**BITCLIQUER2**	**CLIQUER**	**BITRDS1**	**RDS2**
sanr400_0.5	(400,0.50)	0.15	0.18	**0.11**	0.12	0.12
san1000	(1000,0.50)	**1.99**	2.29	*	5.61	2.22
brock400_1	(400,0.75)	29.98	37.80	152.09	**29.21**	51.94
brock400_2	(400,0.75)	**34.75**	51.32	213.81	44.37	65.67
brock400_3	(400,0.75)	28.66	38.58	170.55	**27.94**	43.63
brock400_4	(400,0.75)	**17.83**	20.11	168.36	32.79	78.49
brock800_1	(800,0.65)	**391.50**	853.90	972.69	413.09	1096.71
brock800_2	(800,0.65)	598.11	1323.48	1005.68	**437.06**	1531.12
brock800_3	(800,0.65)	**457.88**	986.15	1062.58	462.08	1144.26
brock800_4	(800,0.65)	610.96	1412.36	1248.55	**511.67**	1681.99
sanr400_0.7	(400,0.70)	**5.65**	7.95	18.08	6.90	11.05
san400_0.7_1	(400,0.70)	**2.13**	4.56	*	2.75	3.94
san400_0.7_2	(400,0.70)	**2.95**	5.12	*	6.69	15.43
san400_0.7_3	(400,0.70)	2.12	**0.76**	*	6.94	1.27
p_hat500-2	(500,0.50)	0.51	0.09	3.22	1.64	**0.03**
p_hat700-2	(700,0.50)	18.28	0.27	93.14	37.05	**0.13**
p_hat1000-2	(1000,0.49)	716.71	9.11	2955.63	864.28	**5.31**
p_hat1500-2	(1500,0.51)	*	566.07	*	*	**325.62**
p_hat300-3	(300,0.74)	0.57	0.10	9.33	1.47	**0.08**
p_hat500-3	(500,0.75)	301.27	**4.53**	*	419.27	5.31
gen200_p0.9_44	(200,0.90)	2.91	0.48	345.45	5.13	**0.37**
gen200_p0.9_55	(200,0.90)	1.11	0.35	885.36	9.68	**0.26**
gen400_p0.9_55	(400,0.90)	*	**3272.15**	*	*	6950.08
gen400_p0.9_65	(400,0.90)	*	*	*	*	*
gen400_p0.9_75	(400,0.90)	6738.43	**325.59**	*	*	3200.62
C250.9	(250,0.90)	18.33	**3.55**	2437.97	60.19	7.94
san200_0.9_1	(200,0.90)	0.54	0.46	*	1.01	**0.00**
san200_0.9_2	(200,0.90)	0.50	**0.39**	118.20	1.07	7.49
san200_0.9_3	(200,0.90)	7.41	**1.51**	1287.49	12.57	2.20
san400_0.9_1	(400,0.90)	174.92	**83.22**	*	492.12	2269.02
sanr200_0.9	(200,0.90)	4.22	0.92	577.69	13.13	**0.91**
hamming10-2	(1024,0.99)	**0.62**	*	716.31	1.81	*
MANN_a27	(378,0.99)	*	1.17	*	*	**0.20**

Fonte: Elaborada pelo autor.

5.4 EXACTCOLOR

As instâncias `EXACTCOLOR` são instâncias oriundas da geração de colunas para resolução do *problema do número cromático fracionário*. Os pesos atribuídos aos vértices são dados pelos valores das variáveis duais do problema escalonados por um valor K, para o algoritmo ser numericamente seguro. O tempo limite para a resolução de cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Identificamos em negrito o melhor desempenho em cada caso. A Tabela 17 apresenta a quantidade de subproblemas enumerados por cada algoritmo. Já a Tabela 18 mostra o tempo de execução consumido por cada um deles.

5.4.1 Número de Subproblemas

BITRDS1 e **BITRDS2** reduzem o número de subproblemas enumerados em até 4 ordens de magnitude com relação ao algoritmo **CLIQUER** (Instância *2-Insertions_4*). Novamente, essa redução aumenta à medida que a densidade do grafo aumenta. O algoritmo **BITRDS1** resolve 5 instâncias que **CLIQUER** não resolve dentro do tempo limite, enquanto, **BITRDS2** resolve 6 instâncias não resolvidas por ele. Destacamos que **BITRDS2** resolve 1 instância não resolvida por **BITCLIQUE2**. Porém, na maioria das instâncias, **BITRDS1** e **BITRDS2** são superados pelo algoritmo de Branch & Bound usando a mesma ordem.

Tabela 17 – Números de subproblemas para cada instância EXACTCOLOR

Instancia	**(n,p)**	**Número de Subproblemas**				
		BITCLIQUER1	**BITCLIQUER2**	**CLIQUER**	**BITRDS1**	**RDS2**
latin_square_10	(90,0.00)	**1.00e+00**	**1.00e+00**	9.00e+01	9.00e+01	9.00e+01
r1000.5	(234,0.00)	**1.00e+00**	**1.00e+00**	2.35e+02	2.34e+02	2.34e+02
DSJR500.1c	(189,0.02)	7.00e+00	**4.00e+00**	3.34e+02	2.25e+02	1.96e+02
r1000.1c	(511,0.03)	1.60e+02	**1.26e+02**	1.68e+03	9.56e+02	6.22e+02
DSJC250.9	(250,0.10)	**9.50e+01**	1.08e+02	9.75e+02	5.25e+02	4.49e+02
DSJC500.9	(500,0.10)	**2.78e+02**	3.47e+02	4.39e+03	2.10e+03	2.00e+03
DSJC1000.9	(1000,0.10)	**1.41e+03**	1.50e+03	2.55e+04	1.31e+04	1.44e+04
DSJC250.5	(250,0.50)	1.30e+04	**1.27e+04**	3.54e+05	5.65e+04	5.29e+04
DSJC500.5	(500,0.50)	**4.50e+05**	4.55e+05	2.00e+07	2.24e+06	2.28e+06
DSJC1000.5	(1000,0.50)	**3.08e+07**	3.16e+07	2.19e+09	1.78e+08	1.83e+08
C2000.5.1029	(2000,0.50)	*	*	*	*	*
flat300_28_0	(300,0.52)	**4.58e+04**	4.67e+04	1.49e+06	1.98e+05	2.05e+05
flat1000_50_0	(967,0.51)	**1.16e+06**	2.04e+06	7.38e+07	7.86e+06	2.27e+07
flat1000_60_0	(999,0.51)	**5.97e+06**	7.80e+06	4.16e+08	3.47e+07	7.07e+07
flat1000_76_0	(1000,0.51)	**3.93e+07**	3.99e+07	2.79e+09	2.25e+08	2.30e+08
school1	(336,0.71)	1.31e+05	**4.42e+04**	1.49e+08	3.97e+05	1.27e+05
DSJC250.1	(241,0.89)	1.12e+08	**7.60e+07**	*	5.76e+08	3.85e+08
DSJC500.1.117	(494,0.90)	*	*	*	*	*
DSJC1000.1.3915	(998,0.90)	*	*	*	*	*
2-Insertions_4	(149,0.95)	**1.02e+04**	4.34e+04	2.29e+08	2.16e+04	6.87e+04
1-Insertions_6	(527,0.96)	**5.66e+04**	8.50e+05	*	1.95e+07	4.43e+06
2-Insertions_5	(369,0.97)	6.10e+05	**6.01e+05**	*	1.40e+07	6.65e+05
3-Insertions_4	(208,0.97)	2.20e+05	**4.50e+03**	*	2.31e+06	1.03e+04
4-Insertions_4	(295,0.98)	**4.56e+05**	9.57e+06	*	1.05e+07	5.16e+06
3-Insertions_5	(460,0.98)	1.19e+08	*	*	*	**3.92e+07**

Fonte: Elaborada pelo autor.

5.4.2 Tempo de Execução

Analisando os tempos de computação dos 4 novos algoritmos, percebemos que para grafos com a densidade até 70%, os algoritmos de Bonecas Russas apresentam melhor desempenho que seus correspondentes em *Branch & Bound*(com raras exceções). Para a densidade superior 70%, não há uma tendência clara.

Para instâncias até 50%, **CLIQUER** mostra-se competitivo com os demais algoritmos de Bonecas Russas. A partir de 50%, ou **CLIQUER** supera o tempo limite, ou obtém um tempo de execução bastante superior aos outros algoritmo de Bonecas Russas. Para essa mesma faixa, **BITRDS1** e **BITRDS2** conseguem resolver seis instâncias, que não foram resolvidas por **CLIQUER**, em menos de 10 minutos . Por outro lado, nenhum dos algoritmos conseguiu encontrar a solução ótima das instâncias *C2000.5.1029*, *DSJC500.1.117* e *DSJC1000.1.3915* dentro do tempo limite.

Tabela 18 – Tempos de execução para cada instância EXACTCOLOR

Instância	**(n,p)**	**Tempo de Execução**				
		BITCLIQUER1	**BITCLIQUER2**	**CLIQUER**	**BITRDS1**	**BITRDS2**
latin_square_10	(90,0.00)	0.08	0.09	**0.00**	**0.00**	**0.00**
r1000.5	(234,0.00)	0.18	0.19	**0.00**	**0.00**	**0.00**
DSJR500.1c	(189,0.02)	0.15	0.15	**0.00**	**0.00**	**0.00**
r1000.1c	(511,0.03)	0.17	0.16	**0.00**	**0.00**	**0.01**
DSJC250.9	(250,0.10)	0.07	0.07	**0.00**	**0.00**	**0.00**
DSJC500.9	(500,0.10)	0.14	0.14	**0.01**	**0.01**	**0.01**
DSJC1000.9	(1000,0.10)	0.28	0.29	**0.02**	0.03	0.03
DSJC250.5	(250,0.50)	0.11	0.10	0.09	**0.05**	**0.05**
DSJC500.5	(500,0.50)	3.22	3.23	5.51	3.22	**3.14**
DSJC1000.5	(1000,0.50)	418.77	411.12	733.11	416.52	**396.96**
C2000.5.1029	(2000,0.50)	*	*	*	*	*
flat300_28_0	(300,0.52)	0.26	0.27	0.36	0.30	**0.21**
flat1000_50_0	(967,0.51)	53.01	93.38	**36.58**	38.20	97.06
flat1000_60_0	(999,0.51)	178.63	227.20	187.11	**131.87**	232.54
flat1000_76_0	(1000,0.51)	539.63	524.89	935.53	531.48	**504.08**
school1	(336,0.71)	2.12	0.60	48.65	3.22	**0.57**
DSJC250.1	(241,0.89)	646.72	**508.14**	*	1007.56	737.97
DSJC500.1.117	(494,0.90)	*	*	*	*	*
DSJC1000.1.3915	(998,0.90)	*	*	*	*	*
2-Insertions_4	(149,0.95)	0.20	0.27	29.38	**0.03**	0.11
1-Insertions_6	(527,0.96)	**1.61**	21.09	*	132.64	46.32
2-Insertions_5	(369,0.97)	5.98	5.66	*	72.38	**4.19**
3-Insertions_4	(208,0.97)	0.97	0.24	*	5.06	**0.04**
4-Insertions_4	(295,0.98)	**3.63**	64.46	*	40.48	25.54
3-Insertions_5	(460,0.98)	1404.59	*	*	*	**381.25**

Fonte: Elaborada pelo autor.

5.5 Conclusão

Neste capítulo, apresentamos uma maneira eficiente de incorporar a heurística **BITCOLOR** ao algoritmo de Bonecas Russas, dando origem ao algoritmo **BITRDS**. Em relação ao melhor algoritmo de Bonecas Russas da literatura, em grafos gerados aleatoriamente, **BITRDS1** reduz tanto o número de subproblemas quanto o tempo de execução. Essa diferença fica mais evidente à medida que a densidade do grafo aumenta. Essa redução torna-se bastante expressiva quando comparamos **CLIQUER** com **BITRDS2** para grafos com a densidade superior ou igual a 90%. Além disso, vale notar que **BITRDS1** mostra-se mais eficiente que os algoritmos de *Branch & Bound* para instâncias com a densidade entre 50% e 80%.

O nosso método de Bonecas Russas para CLIQUE PONDERADA adiciona algumas características interessantes ao método original:

- Integração com um procedimento de limite superior diferente, baseado em uma heurística de coloração.
- Definição de uma estratégia de ramificação diferente. Ela utiliza a coloração ponderada para definir o conjunto de ramificação e a ordem de ramificação segue a ordem inicial ρ.

Acreditamos que essas modificações, junto com o uso de vetores de bits, foram responsáveis pelo bom desempenho de **BITRDS** nos grafos com a densidade média.

A principal vantagem do método de Bonecas Russas é a estratégia de poda definida a partir do vetor de soluções armazenadas c. Contudo, o tempo requerido para calcular completamente o vetor c pode superar a redução de tempo proporcionada por ele. No próximo capítulo, apresentamos uma cooperação entre o algoritmo de Bonecas Russas e o Algoritmo de Busca por Resolução. Nela, o vetor c não é calculado completamente, mas ele continua sendo utilizado para enumerar implicitamente regiões do espaço de busca.

6 ALGORITMO DE BUSCA POR RESOLUÇÃO

Este capítulo é dedicado à apresentação do método de *Busca por Resolução* e da nossa proposta de aplicação ao problema CLIQUE PONDERADA. Este método foi apresentado por Chvátal em Chvátal (1997), como uma alternativa ao método de enumeração implícita que fosse menos dependente da estratégia de ramificação.

Durante o método de enumeração implícita usando a estratégia de busca em profundidade, o processo pode gastar bastante tempo de exploração em uma região infértil do espaço de busca. Esse fenômeno, conhecido como *thrashing*, ocorre quando diferentes regiões do espaço de busca falham pelo mesmo motivo ou por motivos similares. Por exemplo, isso ocorre quando uma inviabilidade causada pela fixação de uma variável é encontrada somente quando sua sub-árvore é totalmente explorada.

Essa característica torna a enumeração implícita bastante dependente de uma estratégia de ramificação eficiente. Com o objetivo de reduzir o *thrashing*, a Busca por Resolução apresenta uma exploração mais inteligente do espaço de busca, baseada nos algoritmos de *backtracking inteligente*. O método da Busca por Resolução já foi utilizado nos seguintes problemas com um relativo sucesso:

- Em Demassey, Artigues, and Michelon (2004), o método foi aplicado ao problema de escalonamento de projetos com restrição de recursos. Neste trabalho, uma versão básica da Busca por Resolução é comparada com uma versão clássica de enumeração implícita, ambas usando a mesma formulação e mesma regra de ramificação. Os resultados computacionais indicaram uma vantagem da Busca por Resolução sobre o método de enumeração. Porém, os autores relatam que a integração de métodos mais elaborados de enumeração implícita à Busca por Resolução representa um grande desafio.
- Em Palpant, Artigues, and Oliva (2007), uma hibridização da Busca por Resolução com técnicas de programação por restrições é apresentada. Neste trabalho, o algoritmo é adaptado para resolver o problema das n^2-rainhas. Novamente, o método baseado na Busca por Resolução mostrou-se superior ao algoritmo clássico de enumeração implícita usando a mesma estratégia de ramificação. Contudo, apesar de interessante, o resultado computacional obtido não foi competitivo com o algoritmo do estado da arte para o problema.
- Em Boussier *et al.* (2010), um algoritmo híbrido combinando Busca por Resolução, enumeração implícita e enumeração explícita foi apresentado para resolver o problema da mochila multidimensional. Com essa abordagem, instâncias previamente não resolvidas do problema da mochila inteira 0-1 multidimensional foram solucionadas de maneira exata. Os resultados indicam a vantagem de deixar a Busca por Resolução conduzir o processo de Busca, confirmando os resultados obtidos em Demassey, Artigues, and Michelon (2004) e Palpant, Artigues, and Oliva (2007).

- Em Posta, Ferland, and Michelon (2011), a Busca por Resolução é generalizada para resolver um problema qualquer de programação inteira.

6.1 O Método de Busca

Em 1997, Chvátal Chvátal (1997) propôs um método exato diferente para problemas de otimização com variáveis binárias, chamado *Busca por Resolução*. O objetivo principal era definir um método alternativo ao de enumeração implícita menos dependente da estratégia de ramificação. A dependência extrema da estratégia de ramificação ocasiona o fenômeno conhecido como *thrashing* nos métodos de enumeração implícita.

Na busca por resolução, a árvore de busca é substituída por uma família de condições lógicas que, representam o espaço de busca explorado. Por exemplo, a cláusula $x_1 \wedge \overline{x_3} \wedge x_6$ representa o conjunto de soluções com $x_1 = x_6 = 1$ e $x_3 = 0$ e qualquer valor para as demais variáveis binárias. Uma cláusula é declarada um *nogood* [1] quando ela está associada a uma região já explorada do espaço de busca. Um mecanismo de inferência é aplicado à família de condições lógicas para a obtenção de uma região do espaço de busca não alcançada pelos *nogoods* memorizados. Uma restrição sobre estrutura das condições lógicas armazenadas garante que o mecanismo de inferência execute em tempo polinomial e que o número de *nogoods* memorizados seja linear.

Em geral, o esforço computacional da etapa de atualização da família está diretamente relacionado com o grau de liberdade de escolha da região do espaço de busca ainda não alcançado. Quanto maior a liberdade de escolha da região não alcançada, maior será o esforço computacional relacionado com a atualização da família.

6.2 Definições preliminares

Na *Busca por Resolução*, um *nogood* pode ser entendido como uma combinação de valores para algumas variáveis que não podem fazer parte de uma solução desejada (viável ou melhor que uma solução viável conhecida). Logo, qualquer combinação de valores das variáveis que seja derivada de um *nogood* deve ser evitada durante o processo de busca. Essas informações são utilizadas para selecionar novas regiões do espaço de busca ainda não explorado. A menos que dito o contrário, consideramos um problema de maximização, definido sobre n variáveis binárias: $\max\{c^T x | Ax \geq b, x \in \{0,1\}\}$.

A seguir, apresentaremos as notações e definições que serão utilizadas ao longo desse capítulo.

[1] O termo *nogoods* é um termo emprestado da terminologia de programação por restrições.

6.2.1 Solução Parcial

Definição 1 *Uma solução parcial é um vetor* $(u_1, u_2, \ldots, u_n) \in \{0, 1, *\}^n$. *Se* $u_i = *$ *dizemos que* u_i *está livre; caso contrário,* u_i *está fixa. Uma solução parcial define, pois, uma fixação de valor para algumas das variáveis do problema.*

Uma solução parcial é denominada solução completa se não tem variáveis livres.

Definição 2 *Se* $(u_1, u_2, \ldots, u_n) \in \{0, 1, *\}^n$ *e* $(v_1, v_2, \ldots, v_n) \in \{0, 1, *\}^n$ *tais que*

$$u_j = v_j, \text{ se } u_j \neq *$$

então dizemos que $(v_1, v_2, \ldots, v_n)$ *é uma extensão de* $(u_1, u_2, \ldots, u_n)$ *e denotamos por*

$$(u_1, u_2, \ldots, u_n) \sqsubseteq (v_1, v_2, \ldots, v_n)$$

Por exemplo, se $U = (1, 1, *, *)$ e $V = (1, 1, *, 1)$, então V é uma extensão de U ($U \sqsubseteq V$).

Definição 3 *Para uma solução parcial* U, *o conjunto* $X(U)$, *chamado cobertura ou alcance de* U, *será definido como*

$$X(U) = \{V \ — U \sqsubseteq V\}$$

Uma solução parcial V *é coberta por* U *se somente se* $V \in X(U)$.

Definição 4 *Duas soluções parciais* $U = (u_1, \ldots, u_n)$ *e* $V = (v_1, \ldots, v_n)$ *são* ***conflitantes*** *ou* ***conflitam*** *se existe* $i \in \{1, \ldots, n\}$ *tal que* $u_i \in \{0, 1\}$ *e* $v_i \in \{0, 1\}$ *e* $u_i \neq v_i$. *Caso contrário, são ditos não-conflitantes.*

Definição 5 *A união de duas soluções parciais não-conflitantes* $U = (u_1, \ldots, u_n)$ *e* $V = (v_1, \ldots, v_n)$ *é a solução parcial* $W = U \cup V$ *tal que*

$$w_i = \begin{cases} u_i & , \text{se} u_i \in \{0, 1\} \\ v_i & , \text{se} v_i \in \{0, 1\} \\ * & u_i = v_i = * \end{cases}$$

Proposição 4 *Para duas soluções parciais* U *e* V *valem as seguintes propriedades:*

- *Se*$U \sqsubseteq V$ *então* $X(V) \subseteq X(U)$.
- *Se* U *e* V *não* ***conflitam*** *então* $X(U \cup V) = X(U) \cap X(V)$
- *Se* U *e* V *não* ***conflitam*** *então* $X(U) \cap X(V) \neq \emptyset$

Toda solução parcial $U = (u_1, \ldots, u_n) \in \{0, 1, *\}^n$ pode ser descrita, alternativamente, por uma cláusula não conflitante C, definida pela a conjunção dos literais x_i, quando $u_i = 1$, e $\overline{x_i}$, quando $u_i = 0$. Em outras palavras, os literais das cláusulas indicam os valores das variáveis fixadas na solução parcial. Por exemplo,

$$U = (1, *, 0, 1, *) \Rightarrow C = x_1 \wedge \overline{x_3} \wedge x_4 \tag{29}$$

Dessa forma, vamos indistintamente usar a representação vetorial ou em cláusula, conforme seja mais conveniente, assim como usar os termos solução parcial e cláusula de forma indistinta. Além disso, por simplicidade, vamos alternativamente denotar uma cláusula conjuntiva sem os "$\wedge$" ou pelo conjunto de literais que nela aparecem. Assim, no exemplo acima,

$$U = (1, *, 0, 1, *) \Rightarrow C = x_1 \wedge \overline{x_3} \wedge x_4 = x_1\overline{x_3}x_4 = \{x_1, \overline{x_3}, x_4\} \quad (30)$$

Em particular, $\emptyset$ denota uma solução parcial com todas as variáveis livres, ou seja, (*,*,...,*).

6.2.2 *Nogood*

Definição 6 *Para um problema de maximização, uma função f é chamada oráculo se somente se $f(U) \geq f(V)$, quando $U \sqsubseteq V$.*

Observe que um procedimento de limite superior para um problema de maximização pode ser utilizado como uma função `oráculo`. No algoritmo proposto por Chvátal, a função `oráculo` é dada pela relaxação linear do problema de maximização, aplicada a uma solução parcial U.

Algorithm 6.1 `oráculo` ($U = (u_1, u_2, \ldots, u_n)$)

1: Seja $X_U = \{Ax \geq b, 0 \leq x \leq 1, x_j = u_j$ para todo $u_j \neq *\}$
2: **if** $\{c^T x — x \in X_U\}$ é inviável **then**
3: **return** $-\infty$
4: **else if** $\max\{c^T x — x \in X_U\}$ é ilimitado **then**
5: **return** $+\infty$
6: **else**
7: **return** $\max\{c^T x — x \in X_U\}$

Finalmente, podemos definir uma combinação de valores das variáveis que não podem fazer parte de nenhuma solução desejável. Essa combinação de valores está associada com uma solução parcial considerada não promissora pela função `oráculo`.

Definição 7 *Seja record um limite inferior para um problema de maximização. Uma solução parcial U é chamada um nogood se `oráculo`$(U) \leq$ record.*

Nogoods podem ser obtidos de diversas formas. Em particular, dada uma solução parcial U, tal que $X(U)$ represente uma região do espaço de busca ainda não explorado, deseja-se encontrar um *nogood* que cubra pelo menos uma parte dessa região. Chvátal sugere um procedimento, chamado `obstáculo`, para encontrar uma solução parcial S com tais propriedades, ou seja,

1. S é um *nogood*.
2. $X(S) \cap X(U) \neq \emptyset$

Uma implementação simples do procedimento `obstáculo` seria iterativamente fixar variáveis livres em U até descobrir uma solução parcial S tal que `oráculo`$(S) = -\infty$ ou S é viável (neste caso, *record* pode ser atualizado por $\max(record,$ `oráculo`$(S))$. Nos dois casos, temos que S é um *nogood*, pois `oráculo`$(S) \leq record$. Adicionalmente, $X(U) \cap X(S) \neq \emptyset$, pois $S \sqsubseteq U$.

Por ora é suficiente saber que encontrar uma solução com as propriedades (i) e (ii) é possível. Mais detalhes sobre a implementação do obstáculo serão apresentados na Subseção 6.2.5.

Definição 8 *O alcance $X(\mathcal{F})$ de uma família $\mathcal{F} = [U_1, \ldots, U_m]$ de soluções parciais é o conjunto de todas as soluções parciais cobertas por pelo menos um elemento da família $\mathcal{F}$, ou seja,*

$$X(\mathcal{F}) = \bigcup_{i=1}^{m} X(U_i)$$

6.2.3 Estrutura Geral

O algoritmo genérico de *Busca por Resolução*, que pode ser descrito como no Algoritmo 6.2, começa inicializando a família $\mathcal{F}$ de *nogoods*. Enquanto a família de *nogoods* $\mathcal{F}$ não alcançar todo o espaço de busca, uma solução parcial ainda não alcançada pela família $\mathcal{F}$, denotado por $U_{\mathcal{F}}$, será selecionada. Na linha 5, o procedimento **obstáculo** é responsável por encontrar um *nogood* S tal que $X(U_{\mathcal{F}}) \cap X(S) \neq \emptyset$. Depois, o *nogood* S é "adicionado" a família $\mathcal{F}$. Essa adição pode ser simplesmente inserir a solução parcial S na família ou modificar a família $\mathcal{F}$ para possibilitar a sua inserção.

A eficácia do Algoritmo 6.2 depende de pelo menos 2 pontos que estão inter-relacionados. O primeiro ponto fundamental é a implementação do procedimento `obstáculo`, cujo propósito descrito, já antecipado, é realizar uma busca em $X(U_{\mathcal{F}})$, procurando por um *nogood* S. Deve-se ter em mente um compromisso entre o tempo gasto nessa busca e o tamanho da região coberta por S.

Algorithm 6.2 Busca por Resolução

1: *record* ← valor de uma solução inicial.
2: Inicialize a família $\mathcal{F}$ de *nogoods*.
3: **while** $X(\mathcal{F}) \neq \{0,1\}^n$ **do**
4: Selecione uma solução parcial $U_{\mathcal{F}}$ tal que $U_{\mathcal{F}} \notin X(\mathcal{F})$
5: `obstáculo`$(U_{\mathcal{F}}, record, S)$.
6: Atualize a família de *nogoods* com S.

O segundo ponto fundamental é encontrar a solução parcial $U_{\mathcal{F}}$ não alcançada pela família $\mathcal{F}$. Isto equivale a um problema de satisfabilidade de uma fórmula obtida pelo complemento dos *nogoods* da família $\mathcal{F}$. No exemplo seguinte, ilustramos como obter essa fórmula.

Exemplo 7 *Considere a seguinte família $\mathcal{F} = [x_1x_2, \overline{x_1}x_2x_3, \overline{x_3}x_4x_5]$. Associada à família $\mathcal{F}$, temos uma fórmula $\alpha_{\mathcal{F}}$, formada pela conjunção do complemento das cláusulas associadas a cada* nogood *de $\mathcal{F}$:*

$$\alpha_{\mathcal{F}} = (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_5})$$

Essa fórmula representa a região do espaço de busca ainda não explorado pela família $\mathcal{F}$. Ela pode ser satisfeita de diversas maneiras. Por exemplo, $\overline{x_1}\overline{x_2}x_3$ e $\overline{x_2}x_3$ satisfazem a

fórmula obtida pela família $\mathcal{F}$. Cada solução parcial que satisfaz a fórmula $\alpha_{\mathcal{F}}$ representa uma solução parcial não alcançada por $\mathcal{F}$, utilizada na Linha 4 do Algoritmo 6.2.

Claramente, encontrar $U_{\mathcal{F}}$ torna-se mais complicado à medida que o tamanho da família cresce. Em princípio, a quantidade de *nogoods* gerado no processo pode ser exponencial. Este poderia ser um grave problema para a *Busca por Resolução*.

Por essas razões, faz-se necessário impor certas restrições à família $\mathcal{F}$ para que ela mantenha uma estrutura que permita gerar de maneira mais fácil $U_{\mathcal{F}}$. Além disso, essa restrição mantém o tamanho da família polinomial. A estrutura será apresentada na subseção seguinte.

6.2.4 Família *Path-Like*

Uma família de cláusulas $\mathcal{F} = [C_1, C_2, \ldots, C_m]$ e um conjunto de literais $W = [w_1, w_2, \ldots, w_m]$ têm estrutura especial chamada de *path-like* se as seguintes propriedades são válidas:

$$w_i \in C_j \text{ se somente se } j = i \tag{31}$$

$$\text{Se } \overline{w_i} \in C_j \text{ então } j > i \tag{32}$$

$$\text{Se } w \in C_i \wedge \overline{w} \in C_j \text{ então } (w_i = w) \vee (w_j = \overline{w}) \tag{33}$$

Dizemos que w_i é o representante de C_i, $i = 1, \ldots, m$.

A propriedade (31) estabelece que o representante de C_i só aparece em C_i. A propriedade (32) diz que a negação do representante de C_i só pode figurar em uma cláusula C_j, quando $j > i$. A propriedade (33) assegura que apenas representantes podem aparecer também negados em uma outra cláusula da família.

Exemplo 8 *A família $\mathcal{F} = [C_1 = x_1x_2, C_2 = \overline{x_1}x_2x_3, C_3 = \overline{x_3}x_4x_5]$ e o conjunto de literais $W = [x_1, x_3, x_4]$ têm a estrutura* path-like. *Note que x_1,x_3 e x_4 só aparecem, respectivamente, em C_1,C_2 e C_3. A negação de cada w_i ou não aparece ou aparece apenas C_j tal que $j > i$. Todo literal que aparece em uma cláusula assim como sua negação pertence a W.*

Exemplo 9 *A família $\mathcal{F} = [C_1 = x_1x_2, C_2 = x_1x_2x_3, C_3 = x_4x_5]$ e o conjunto de literais $W = [w_1 = x_1, w_2 = x_2, w_3 = x_4]$ não têm a estrutura* path-like*, pois o representante de C_2(a saber, w_2) aparece em C_1 e C_2.*

Dada uma família $\mathcal{F}$ e um conjunto de literais W com a estrutura *path-like*, uma solução parcial não alcançada por $\mathcal{F}$, denotada $U_{\mathcal{F}}$, pode ser determinada da seguinte maneira:

Proposição 5 *Seja $\mathcal{F} = [C_1, C_2, \ldots, C_m]$ uma família e $W = [w_1, w_2, \ldots, w_m]$ um conjunto de literais com a estrutura de* path-like. *Uma solução parcial válida não alcançada por $\mathcal{F}$ é:*

$$U_{\mathcal{F}} = \bigcup_{i=1}((C_i - \{w_i\}) \cup \{\overline{w_i}\})$$

Prova Vamos mostrar que $U_{\mathcal{F}}$ é uma solução parcial válida, ou seja, sua cláusula associ-

ada não é uma contradição. A propriedade (31) garante que não existe contradição entre $(C_i - w_i) \wedge \overline{w_j}$. A propriedade (32) implica que um literal e a sua negação não podem ser simultaneamente representante , ou seja, não existe contradição em $\bigcup_{i=1}^{m}\{\overline{w_i}\}$. Já (33) implica que qualquer literal de $C_i - w_i$ não ocorre negado em $C_j - w_j$. Logo, as três propriedades garantem que $U_{\mathcal{F}}$ é uma solução parcial válida.

Mais ainda, a inclusão de $\overline{w_i}$ em $U_{\mathcal{F}}$ torna verdadeira a fórmula $\alpha_{\mathcal{F}} = \bigwedge_{i=1}^{m} \overline{C_i}$, levando a $U_{\mathcal{F}} \notin X(\mathcal{F})$.

A família *path-like* $\mathcal{F}$ pode ser estendida seguinte maneira:

Proposição 6 *Uma família* $\mathcal{F}' = [C_1, C_2, \ldots, C_m, C_{m+1}]$ *e um conjunto de literais* $W' = [w_1, w_2, \ldots, w_m, w_{m+1}]$ *têm uma estrutura* path-like *se a família* $\mathcal{F} = [C_1, C_2, \ldots, C_m]$ *e um conjunto de literais* $W = [w_1, w_2, \ldots, w_m]$ *têm a estrutura* path-like *e:*

1. $w_{m+1} \in C_{m+1}$
2. $w_{m+1} \notin U_{\mathcal{F}}$
3. $\overline{w_{m+1}} \notin U_{\mathcal{F}}$.
4. *Se* $w \in U_{\mathcal{F}}$ *então* $\overline{w} \notin C_{m+1}$

Prova Para mostrar (31) é suficiente mostrar que $w_{m+1} \notin C_i$, para $i = 1, \ldots, m$. A propriedade (ii) garante que $w_{m+1} \notin C_i - w_i$. A propriedade (iii) assegura que $\overline{w_{m+1}} \notin \{\overline{w_i}\}$, ou seja, $w_{m+1} \notin \{w_i\}$, para $i = 1, \ldots, m$. Logo, $w_{m+1} \notin C_i$, para $i = 1, \ldots, m$

Para (32), vamos demonstrar que $\overline{w_{m+1}} \notin C_i$, para $i = 1, \ldots, m$. A propriedade (iii) garante que $\overline{w_{m+1}} \notin C_i - w_i$. A propriedade (ii) assegura que $w_{m+1} \notin \{\overline{w_i}\}$, ou seja, $\overline{w_{m+1}} \notin \{w_i\}$, para $i = 1, \ldots, m$. Logo, $\overline{w_{m+1}} \notin C_i$ para todo $i = 1, \ldots, m$

Para (33), devemos mostrar que se $w \in C_i - w_i$, para $i = 1, \ldots, m$ (ou seja, w está em C_i, mas não é representante), então $\overline{w} \notin C_{m+1} - \{w_{m+1}\}$, ou seja, $\overline{w}$ não está em C_{m+1} ou é seu complemento. Isto é garantindo pela propriedade (iv), pois $C_i - w_i \subseteq U_{\mathcal{F}}$.

Por definição, $U_{\mathcal{F}}$ não está na cobertura de nenhuma solução parcial de $\mathcal{F}$, isto é, $U_{\mathcal{F}} \notin X(\mathcal{F})$. A solução parcial $U_{\mathcal{F}}$ é o ponto inicial utilizado pela função **obstáculo** para encontrar um *nogood* S tal que $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$. O *nogood* S é usado para atualizar a família $\mathcal{F}$. Tal atualização de $\mathcal{F}$ será separada em dois casos:

1. $S \not\subseteq U_{\mathcal{F}}$ (S contém um literal que pode ser representante)
2. $S \subseteq U_{\mathcal{F}}$ (S não contém um literal que pode ser representante)

No caso 1, o *nogood* S contém um literal w que satisfaz a condição *path-like*, permitindo que S seja adicionado diretamente à família. No caso 2, o *nogood* S não pode ser adicionado diretamente na família $\mathcal{F}$. A seguir, mostramos esses dois casos:

O Algoritmo 6.3 apresenta a atualização da família *path-like* $\mathcal{F}$, de tamanho m, a partir de um *nogood* S tal que $S \not\subseteq U_{\mathcal{F}}$ e $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$. Sua corretude é dada pelo Corolário 1.

Algorithm 6.3 AtualizaçãoFamília1($\mathcal{F}, W, S$)

1: Escolha um literal w tal que $w \in S \setminus U_{\mathcal{F}}$
2: $C_{m+1} = S$
3: $w_{m+1} = w$
4: $\mathcal{F} = \mathcal{F} \cup C_{m+1}$
5: $m \leftarrow m + 1$

Corolário 1 *Seja uma família* $\mathcal{F} = [C_1, \ldots, C_m]$ *e um conjunto de literais* $W = [w_1, \ldots, w_m]$ *com a estrutura* path-like *e um* nogood S *tal que* $S \not\subseteq U_{\mathcal{F}}$ *e* $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$ *então* $\mathcal{F}' = [C_1, \ldots, C_m, S]$ *e* $W' = [w_1, \ldots, w_m, w]$ *tal que* $w \in S \setminus U_{\mathcal{F}}$ *(gerado pelo Algoritmo 6.3) mantêm a estrutura* path-like.

Prova Por hipótese, $S \not\subseteq U_{\mathcal{F}}$. Então, podemos escolher um literal $w \in S \setminus U_{\mathcal{F}}$. Também, por hipótese, $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$. Logo, $w \notin U_{\mathcal{F}}$. Logo, $u \in S$ implica que $\overline{u} \notin U_{\mathcal{F}}$, $u \in U_{\mathcal{F}}$ implica que $\overline{u} \notin S$. Então $w \in S, w \notin U_{\mathcal{F}}$, conseqüentemente $\overline{w} \notin U_{\mathcal{F}}$ e assim todas as condições da Proposição 6 são satisfeitas.

Exemplo 10 *Considere o seguinte exemplo:*

$\mathcal{F}_0 = \emptyset, W = \emptyset, U_{\mathcal{F}_0} = \emptyset$. *Suponha* $S = x_2x_3$, *e portanto,* $C_1 = S$

Podemos escolher x_2 *ou* x_3 *para ser* w_1.

$\mathcal{F}_1 = [x_2x_3], W = [x_2], U_{\mathcal{F}_1} = \overline{x_2}x_3$. *Suponha* $S = x_3x_6$, *e portanto,* $C_2 = S$

Neste caso, podemos escolher x_6 *para ser* w_2.

$\mathcal{F}_2 = [x_2x_3, x_3x_6], W = [x_2, x_6], U_{\mathcal{F}_2} = \overline{x_2}x_3\overline{x_6}$. *Suponha* $S = \overline{x_2}x_3\overline{x_6}$

Neste caso, não existe nenhum literal que possa ser escolhido para ser w_3.

Quando $S \subseteq U_{\mathcal{F}}$ não existe um representante de modo a estender a família diretamente. Neste caso, a atualização da família passa por duas etapas. Primeiro, são resolvidos os conflitos entre S e a família $\mathcal{F}$, gerando um *nogood* reduzido R, através da regra da resolução. Em seguida, R é usado para atualizar a família, pela regra da absorção.

Definição 9 *Sejam* A *e* B *dois* nogoods *que conflitam em um único literal* w, *tal que* $w \in A$ *e* $\overline{w} \in B$. *O* ***resolvente*** *de* A *e* B *é definida pela seguinte solução parcial:*

$$A \nabla B = (A - w) \cup (B - \overline{w})$$

Proposição 7 *Sejam* A *e* B *dois* nogoods *que conflitam em um único literal* w, *tal que* $w \in A$ *e* $\overline{w} \in B$, *então* $A \nabla B$ *é um* nogood.

Prova É suficiente mostrar que $X(A \nabla B) \subseteq X(A) \cup X(B)$. De fato, se U é uma extensão de $A \nabla B$, então U é uma extensão de A, ou B, ou de ambos, dependendo se a variável associada ao literal conflitante w é 0,1 ou * em U. Logo, $X(A \nabla B) \subseteq X(A) \cup X(B)$.

A seguinte proposição estabelece que um nogood S obtido pelo **obstáculo** e os nogoods da família $\mathcal{F}$ podem conflitar apenas nos literais representantes, ou seja, $\overline{w_i} \in S$, para $i = 1, \ldots, m$.

Proposição 8 *Seja* S *um* nogood *tal que* $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$, *onde* $F = [C_1, \ldots, C_m]$ *e* $W = [w_1, \ldots, w_m]$ *é uma família* path-like. *Para* $i = 1, \ldots, m$, S ***conflita*** *com* C_i *se*

somente se $\overline{w_i} \in S$.

Prova Como $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$ e $C_i - w_i \subseteq U_{\mathcal{F}}$, temos que $w \in C_i - w_i$, implica que $\overline{w} \notin S$. Logo, se $w \in C_i$ e $\overline{w} \in S$ então $w = w_i$

Pela Proposição 8, todos os conflitos entre o *nogood* S e algum *nogood* C_i da família *path-like* $\mathcal{F}$ são resolvidos na fase de `ResoluçãoNogood` formalmente definida pelo Algoritmo 6.4.

Algorithm 6.4 `ResoluçãoNogood`$(\mathcal{F}, W, S, R)$

1: $R \leftarrow S$
2: **for** $i \leftarrow m$ até 1 **do**
3: **if** $\overline{w}_i \in R$ **then**
4: $R \leftarrow R \nabla C_i$

Dado que $S \subseteq U_{\mathcal{F}}$, depois de cada execução i do laço do algoritmo 6.4,temos a seguinte propriedade:

$$R \subseteq (U_{\mathcal{F}} \setminus \{\overline{w_i}, \ldots, \overline{w_m}\} \tag{34}$$

Ao final do laço, temos:

$$R \subseteq (\bigcup_{j=1}^{m}(C_j - w_j)) \tag{35}$$

Uma vez obtida a propriedade (35) acima com a fase `ResoluçãoNogood`, podemos proceder com a atualização da família, realizando os seguintes passos:

Passo 1: Encontre o menor k tal que $R \subseteq (\bigcup_{j=1}^{k}(C_j - w_j))$.

Passo 2: Escolha um literal $w \in R - (C_1 \cup \ldots C_{k-1})$

Passo 3: Faça $C_k \leftarrow R$ e $w_k \leftarrow w$

Passo 4: Remova todas as cláusulas $C_{k+1}, \ldots, C_m$ que contenham w.

Os passos (1) a (3) são realizados pelo procedimento `AbsorçãoNogood`(Algoritmo 6.5), cuja corretude segue da Proposição 9.

Algorithm 6.5 `AbsorçãoNogood`$(\mathcal{F}, W, R, k)$

1: **for** $k \leftarrow 1$ até m **do**
2: **if** $R \subseteq (\bigcup_{j=1}^{k}(C_j - w_j))$ **then**
3: **pare**
4: Escolha um $w \in R - (C_1 \cup \ldots C_{k-1})$
5: $C_k \leftarrow R$
6: $w_k \leftarrow w$

Proposição 9 *Seja* $\mathcal{F} = [C_1, \ldots, C_{k-1}]$ *e* $W = [w_1, \ldots, w_{k-1}]$ *uma família* path-like. *Seja um nogood* $R \subseteq \bigcup_{i=1}^{m} C_i - w_i$, *obtido por ResoluçãonNogood. Seja* $k \leq m$ *o menor índice tal que* $R \subseteq \bigcup_{i=1}^{k} C_i - w_i$ *e* $w \in R - (C_1 \cup C_2 \cup \ldots \cup C_{k-1})$. *Então* $\mathcal{F}' = [C_1, \ldots, C_{k-1}, R]$ *e* $W' = [w_1, \ldots, w_{k-1}, w]$ *é uma família* path-like.

Prova Como k é o menor índice tal que $R \subseteq \bigcup_{i=1}^{k} C_i - w_i$, temos que $R \not\subseteq \bigcup_{i=1}^{k-1} C_i - w_i$. Como $\bigcup_{i=1}^{k-1}\{\overline{w_i}\} \cap R \neq \emptyset$, devido a ResoluçãoNogood, podemos concluir que $R \not\subseteq \bigcup_{i=1}^{k-1}(C_i - w_i) \cup \{\overline{w_i}\}$, ou seja, $R \not\subseteq U_{\mathcal{F}_{k-1}}$, onde $\mathcal{F}_{k-1} = [C_1, \ldots, C_{k-1}]$. Além disso, como R não tem conflitos com $U_{\mathcal{F}_{k-1}}$ após a ResoluçãoNogood, temos que $X(R) \cap X(U_{\mathcal{F}_{k-1}}) \neq \emptyset$. Pelo Corolário 1, $\mathcal{F}' = [C_1, \ldots, C_{k-1}, R]$ e $W' = [w_1, \ldots, w_{k-1}, w]$ é uma família *path-like*.

Os *nogoods* $C_{k+1}, C_{k+2}, \ldots, C_m$ podem ser aproveitados dependendo se o representante w do novo *nogood* R pertence ou não a eles, como mostra a seguinte proposição.

Proposição 10 *Sejam k, R e w como na Proposição 9. Sejam $\{i_1, i_2, \ldots, i_p\} \subseteq \{k+1, \ldots, m\}$ tais que $i_1 < i_2 < \ldots < i_p$. Então $\mathcal{F}' = [C_1, \ldots, C_{k-1}, R, C_{i_1}, C_{i_2}, \ldots, C_{i_p}]$ e $W' = [w_1, \ldots, w_{k-1}, w, w_{i_1}, w_{i_2}, \ldots, w_{i_p}]$ é* path-like *se somente se $w \notin C_{i_j}$, para $j = 1, \ldots, p$*

Prova Se $\mathcal{F}'$ e W' é *path-like*, pela propriedade (31), o literal w só pode aparecer em R. Logo, $w \notin C_{i_j}, j = 1, \ldots, p$. Suponha agora que $w \notin C_{i_j}, j = 1, \ldots, p$. Como $\mathcal{F}$ e W é *path-like* e $R \subseteq \bigcup_{i=1}^{k} C_i$ (o que implica, por exemplo, $w_{i_j} \notin R$), temos a propriedade (31). Para (32), falta garantir que $\overline{w_{i_j}} \notin R, j = 1, \ldots, p$, o que é verdade, pois R e C_{i_j} não conflitam. Finalmente, (33) também é assegurada pelo fato de que R e $C_{i_j}, j = 1, \ldots, p$ são não conflitantes e, portanto, não pode ocorrer $w \in R$ e $\overline{w} \in C_{i_j}$, para qualquer w.

Formalmente podemos definir essa sequência de operações da seguinte maneira:

Algorithm 6.6 `ReciclandoNogood`$(\mathcal{F}, W, k)$

```
1: M ← m
2: m ← k
3:                                   ▷ w_k é o representante do novo nogood C_k
4: for i ← k + 1 até M do
5:     if w_k ∉ C_i then
6:         m ← m + 1
7:         C_m ← C_i
8:         w_m ← w_i
```

A atualização da família quando S não tem um literal representante pode ser vista no Algoritmo 6.7.

Algorithm 6.7 AtualizaçãoFamília2$(\mathcal{F}, W, S, k)$

```
1: ResoluçãoNogood(F, W, S, R)
2: AbsorçãoNogood(F, W, R, k)
3: ReciclandoNogood(F, W, k)
```

Corolário 2 *Seja $\mathcal{F}$ uma família* path-like *e um* nogood $S \subseteq U_{\mathcal{F}}$ *tal que $X(S) \cap X(U_{\mathcal{F}}) \neq \emptyset$. Então Algoritmo 6.8 que gera uma nova família que mantém a propriedade* path-like.

Prova Pela Proposição 9, podemos concluir que $\mathcal{F} = [C_1, \ldots, C_{k-1}, R]$ e $W = [w_1, \ldots, w_{k-1}, w]$ é *path-like*. Pela Proposição 10, as cláusulas dentre $C_{k+1}, \ldots, C_m$ restauradas por ReciclandoNogood(Algoritmo 6.6) mantêm a estrutura *path-like*.

O processo completo de atualização da família pode ser apresentado pelo Algoritmo 6.8. Note que $w_k \notin S$, o nogood S pode ser usado novamente para atualizar a família corrente (Linha 6). O Exemplo 11 ilustra uma atualização da família em que o nogood S é usado uma única vez. Já no Exemplo 12, o mesmo nogood é utilizado para atualizar a família duas vezes.

Algorithm 6.8 AtualizaçãoFamília($\mathcal{F}, W, S$)

1: **if** $S \not\subseteq U_{\mathcal{F}}$ **then**
2: AtualizaçãoFamília1($\mathcal{F}, W, S$)
3: **else**
4: AtualizaçãoFamília2($\mathcal{F}, W, S, k$)
5: **if** $w_k \notin S$ **then**
6: AtualizaçãoFamília($\mathcal{F}, W, S$)

Exemplo 11 *Considere a família* path-like *e o* nogood S *abaixo:*

$$\begin{aligned}
C_1 &= x_2x_3 && (w_1 = x_2)\\
C_2 &= x_3x_6 && (w_2 = x_6)\\
C_3 &= \overline{x_1x_5x_9} && (w_3 = \overline{x_1})\\
C_4 &= \overline{x_2x_5x_8} && (w_4 = \overline{x_8})\\
C_5 &= x_3\overline{x_5}x_7\overline{x_9} && (w_5 = x_7)\\
C_6 &= \overline{x_2x_4x_6} && (w_6 = \overline{x_4})\\
S &= \overline{x_5x_6x_7}
\end{aligned}$$

Aplicando o procedimento `ResoluçãoNogood`*:*

$R = S\nabla C_5 = x_3\overline{x_5x_6}\ \overline{x_9}$

$R = R\nabla C_2 = x_3\overline{x_5}\ \overline{x_9}$

Aplicando o procedimento `AbsorçãoNogood`*:*

O menor k *tal que* $R \subseteq (\bigcup_{j=1}^{k}(C_j - w_j))$ *é 3.*

$C_3 = R$

Escolha $w = \overline{x_5}$

Aplicando o procedimento `ReciclandoNogood`*:*

Remova C_4 *e* C_5 *de* $\mathcal{F}$ *e mantenha* C_6

A família atualizada $\mathcal{F}$ *é:*

$$\begin{aligned}
C_1 &= x_2x_3 && (w_1 = x_2)\\
C_2 &= x_3x_6 && (w_2 = x_6)\\
C_3 &= x_3\overline{x_5}\ \overline{x_9} && (w_3 = \overline{x_5})\\
C_4 &= \overline{x_2}\ \overline{x_4}\ \overline{x_6} && (w_4 = \overline{x_4})
\end{aligned}$$

Observe que $\overline{x_5} \in S$.

Exemplo 12 *Considere a seguinte família* path-like *e o* nogood S*:*

$C_1 = x_3 \quad (w_1 = x_3)$
$C_2 = \overline{x_2}\ \overline{x_6} \quad (w_2 = \overline{x_2})$
$C_3 = x_1 x_5 \overline{x_6} \quad (w_3 = x_1)$
$C_4 = \overline{x_3}\ \overline{x_6}\ \overline{x_9} \quad (w_4 = \overline{x_9})$
$C_5 = x_5 x_7 x_9 \quad (w_5 = x_7)$
$C_6 = x_5 \overline{x_7} x_8 \quad (w_6 = x_8)$
$S = \overline{x_6}\ \overline{x_7}\ \overline{x_8}$

Aplicando o procedimento `ResoluçãoNogood`*:*

$R = S \nabla C_6 = \overline{x_6} x_7 x_8 \nabla x_5 \overline{x_7} x_8 = x_5 \overline{x_6} x_7$
$R = R \nabla C_5 = x_5 \overline{x_6} x_7 \nabla x_5 x_7 x_9 = x_5 \overline{x_6} x_9$
$R = R \nabla C_4 = x_5 \overline{x_6} x_9 \nabla \overline{x_3} x_6 x_9 = \overline{x_3} x_5 \overline{x_6}$
$R = R \nabla C_1 = \overline{x_3} x_5 \overline{x_6} \nabla x_3 = x_5 \overline{x_6}$

Aplicando o procedimento `AbsorçãoNogood`*:*

O menor k *tal que* $R \subseteq (\bigcup_{j=1}^{k}(C_j - w_j))$ *é 3.* $C_3 = R$

Escolha $w = x_5$

Aplicando o procedimento `ReciclandoNogood`*:*

Remova C_5 *e* C_6 *de* $\mathcal{F}$ *e mantenha* C_4

A família atualizada $\mathcal{F}$ *é:*

$C_1 = x_3 \quad (w_1 = x_3)$
$C_2 = \overline{x_2}\ \overline{x_6} \quad (w_2 = \overline{x_2})$
$C_3 = x_5 \overline{x_6} \quad (w_3 = x_5)$
$C_4 = \overline{x_3}\ \overline{x_6}\ \overline{x_9} \quad (w_4 = \overline{x_9})$

Observe que $x_5 \notin S$*. Logo,* S *pode ser adicionada na família* $\mathcal{F}$ *construída:*

$C_5 = \overline{x_6}\ \overline{x_7}\ \overline{x_8} \quad (w_1 = \overline{x_7})$

6.2.5 Procedimento obstáculo

Em cada iteração da *Busca por Resolução*, uma região do espaço de busca ainda não explorada é parcialmente explorada por um procedimento chamado `obstáculo`$(U, record, S)$ resultando em um *nogood* S. O procedimento `obstáculo` recebe dois parâmetros:

- U: Uma solução parcial não coberta pela família corrente, que, portanto representa uma região do espaço de busca ainda não explorada; e
- *record*: um limite inferior para um problema de maximização.

  e devolve um terceiro
- S: Um *nogood* encontrado a partir da solução parcial U tal que $X(U) \cap X(S) \neq \emptyset$, representando a parte do espaço de busca implicitamente enumerado pelo procedimento `obstáculo`.

Essa exploração pode ser realizada de diversas maneiras, desde que $X(U) \cap X(S) \neq \emptyset$. Assim, $w \in U$ implica que $\overline{w} \notin S$ e bem como $w \in S$ implica que $\overline{w} \notin U$. Como vimos, essa propriedade garante que seja possível atualizar a família $\mathcal{F}$ usando S e mantendo a propriedade *path-like*.

Em Chvátal (1997), é apresentada uma heurística para encontrar um *nogood* S que não **conflita** com U. O procedimento `obstáculo` apresentado por Chvátal pode ser separado em duas fases específicas: a fase de crescimento e a fase de decrescimento.

- Na fase de crescimento, uma solução parcial U^+ é construída a partir de U tal que $U \sqsubseteq U^+$ e U^+ seja um *nogood*.
- Na fase de decrescimento, uma solução parcial S é construída a partir da solução parcial U^+ tal que $S \sqsubseteq U^+$ e S seja um *nogood*.

Na fase de crescimento, o procedimento `obstáculo` substitui uma variável livre por 0 ou 1 até que

$$\texttt{oráculo}(U^+) \leq \ record \text{ ou } U^+ \in \{0,1\}^n. \tag{36}$$

Então temos dois casos:

- $\texttt{oráculo}(U^+) \leq record$ e, conseqüentemente U^+ é um *nogood*.
- $\texttt{oráculo}(U^+) > record$ e U^+ é uma solução completa. Neste caso, o valor de *record* é atualizado para $\texttt{oráculo}(U^+)$ e U^+ torna-se um *nogood*.

Claramente, $X(U) \cap X(U^+) \neq \emptyset$ e U^+ é um *nogood*. Logo, $S = U^+$ pode ser uma opção de retorno do procedimento obstáculo. Uma versão rápida para o procedimento obstáculo pode ser obtida implementando apenas a fase de crescimento, como no Algoritmo 6.9.

Algorithm 6.9 `obstáculo_rápido`$(U, record, S)$

1: $bound \leftarrow \texttt{oráculo}(U)$
2: **while** $U \notin \{0,1\}^n$ e $bound > record$ **do**
3: Escolha um subscrito j com $u_j = *$ e um valor c $\in$ {0,1}
4: $u_j \leftarrow c$
5: $bound \leftarrow \texttt{oráculo}(U)$
6: **if** $bound > record$ **then** ▷ U é uma solução completa
7: $record \leftarrow bound$
8: $S \leftarrow U$

Na fase de decrescimento, escolhemos uma variável já fixada para ser liberada de maneira que a solução parcial continue representando um `nogood`, ou seja,

- Procure um índice j tal que $u_j^+ \in \{0,1\}$, faça $u_j^+ = *$ e verifique se $\texttt{oráculo}(U^+)$ ¡ record.

A fase de decrescimento é responsável por encontrar um *nogood* minimal S tal que $S \sqsubseteq U^+$. Um *nogood* S é dito minimal, se somente se não existe um *nogood* S' tal que $S' \sqsubseteq S$.

A seguir, apresentamos a heurística proposta por Chvátal para encontrar um *nogood* minimal (Algoritmo 6.10). Inicialmente, as variáveis fixadas em U são empilhadas. Depois, as variáveis fixadas na fase de crescimento são empilhadas. Após o final da fase de crescimento, S recebe uma solução parcial U^+ representando um *nogood*. Enquanto a

pilha não estiver vazia, uma variável x_j é desempilhada e liberada em S. Se `oráculo`(S) ¿ record, então a variável x_j é necessária para definir um *nogood* minimal e seu valor anterior é recuperado. Caso contrário, a variável x_j permanece liberada.

Algorithm 6.10 obstáculo(U, record, S)

1: Inicialize a pilha vazia
2: **for** i = 1 até n **do**
3: **if** $u_j \neq *$ **then**
4: empilhe j na pilha
5: ▷ **Início da Fase de crescimento**
6: $bound \leftarrow$ `oráculo`(U)
7: **while** $U \notin \{0,1\}^n$ e $bound > record$ **do**
8: Escolha um subscrito j com $u_j = *$ e um valor c ∈ {0,1}
9: $u_j \leftarrow c$
10: $bound \leftarrow$ `oráculo`(U)
11: **if** $bound > record$ **then**
12: Empilhe j na pilha
13: **if** $bound > record$ **then**
14: $record \leftarrow bound$
15: ▷ **Início Fase de decrescimento**
16: $S \leftarrow U$
17: **while** a pilha não estiver vazia **do**
18: Desempilhe j da pilha
19: $S_j \leftarrow *$ ▷ Liberando a variável j
20: **if** `oráculo`(S) ¿ record **then**
21: $S_j \leftarrow U_j$ ▷ Essa variável contribui para `oráculo`(S) ≤ record, logo ela é necessária para o *nogood*

Observe que, se a fase de crescimento não gera uma solução completa, a última variável fixada em U^+ não é empilhada e não é analisada na fase de decrescimento.

6.2.6 Políticas de Ramificação

O algoritmo de Busca por Resolução usa duas políticas de ramificação durante o processo de busca, que diferenciamos como: (1) Uma política de mergulho e (2) Uma política de recomeço.

A política de mergulho é a estratégia de ramificação utilizada no procedimento obstáculo na fase de crescimento. Ela consiste em escolher uma variável livre e um valor para ser fixado, correspondendo a seguinte instrução:

`Escolha um subscrito j com` u_j `= * e um valor c ∈ {0,1}`

Essa política corresponde a estratégia de ramificação clássica utilizada nos algoritmos de enumeração implícita.

A política de recomeço é a estratégia de ramificação responsável por escolher o representante do novo *nogood* que, em última análise, é essencial para a definição do ponto

de recomeço do mergulho, ou seja, $U_{\mathcal{F}}$. Tal política é implementada em duas situações:

- No passo 1 do Algoritmo 6.3: `Escolha um literal` w `tal que` $w \in S \setminus U_{\mathcal{F}}$
- No passo 4 do Algoritmo 6.5: `Escolha um` $w \in R - (C_1 \cup \ldots C_{k-1})$

Em Chvátal (1997), a política de mergulho utilizada é

`Escolha a primeira variável na ordem lexicográfica tal que` $u_j = *$ `e fixe seu valor em 0.`

Já a política de recomeço utilizada é

`Escolha a última variável na ordem lexicográfica`

Em Boussier (2008), temos uma implementação da *Busca por Resolução* sem a fase de decrescimento, utilizando diferentes políticas de mergulho e recomeço. As duas políticas de mergulho testadas são:

M_1 Escolha a variável livre com o menor custo reduzido e fixe-a no valor inteiro mais próximo de seu valor ótimo na relaxação linear do problema.

M_2 Escolha a primeira variável livre na ordem lexicográfica e fixe-a no valor inteiro mais próximo de seu valor ótimo na relaxação linear do problema.

Também são avaliados duas políticas de recomeço:

R_1 Escolha o literal associado à variável mais recentemente instanciada

R_2 Escolha o literal associado à variável de menor custo reduzido.

Os resultados computacionais obtidos pelos autores mostram que o número médio de chamadas ao `oráculo` no algoritmo de Busca por Resolução que usa $R_1 - M_1$ é praticamente o mesmo daquele que usa $R_1 - M_2$, sendo também similar ao número de resoluções da relaxação linear em um processo de enumeração implícita que usa as mesmas políticas de mergulho.

Por outro lado, a implementação da Busca por Resolução que utiliza $R_2 - M_1$ realiza menos chamadas ao `oráculo` que a versão que usa $R_1 - M_1$, na maior parte das instâncias testadas. Esse resultado sugere que a política de recomeço introduz um nível de flexibilidade ao processo de busca que não consegue ser alcançado por um método de enumeração implícita.

6.2.7 Convergência

Para mostrar a convergência do método de Busca por Resolução, precisamos mostrar que a cada iteração o alcance da família cresce. Dado um problema com n variáveis, podemos definir a força de uma família $\mathcal{F}$, denotada por $\tau(\mathcal{F})$, como $2^{-n}|X_C(\mathcal{F})|$, onde $X_C(\mathcal{F})$ são as soluções completas em $X(\mathcal{F})$. Este valor, entre 0 e 1, representa a razão entre número de soluções exploradas pela família $\mathcal{F}$ e o número total de soluções. Infelizmente, o valor de $\tau(\mathcal{F})$ não necessariamente aumenta a cada iteração do algoritmo de Busca por Resolução. Contudo, podemos identificar um limite inferior para a força da família que cresce a cada iteração.

Dizemos que uma família $\mathcal{F}$ cobre uma variável x_i se pelo menos um dos

dois literais associados a x_i aparece em uma cláusula de $\mathcal{F}$. Dada uma família $\mathcal{F} = [C_1, \ldots, C_m]$, seja n_i o número de variáveis cobertas pela família $[C_1, \ldots, C_i]$.

Proposição 11 *Se* $\mathcal{F} = [C_1, \ldots, C_m]$ *então* $\tau(\mathcal{F}) \geq \sigma(\mathcal{F})$, *onde* $\sigma(\mathcal{F}) = \sum_{i=1}^{m} 2^{-n_i}$

Prova Por indução, Se $\mathcal{F} = [C_1]$, as soluções completas exploradas por $\mathcal{F}$ são aquelas onde as n_1 variáveis cobertas por C_1 estão fixas e as outras $n - n_1$ variáveis podem assumir qualquer valor em {0,1}. Logo, $\tau(\mathcal{F}) = 2^{-n}(2^{n-n_1}) = 2^{-n_1} = \sigma(\mathcal{F})$.

Suponha o resultado verdadeiro para $\mathcal{F}' = [C_1, \ldots, C_{m-1}]$, ou seja, $\tau(\mathcal{F}') \geq \sum_{i=1}^{m-1} 2^{-n_i}$. Considere $\mathcal{F} = [C_1, \ldots, C_{m-1}, C_m]$. As soluções completas cobertas por $\mathcal{F}$ são as $2^n\tau(\mathcal{F}')$ cobertas por $\mathcal{F}'$ junto com aquelas não cobertas por $\mathcal{F}'$ mas cobertas por C_m. Estas últimas são aquelas cobertas por $\overline{C_1} \wedge \overline{C_2} \wedge \overline{C_2} \wedge \ldots \wedge \overline{C_{m-1}} \wedge C_m$. Entre elas estão todas as 2^{n-n_m} soluções completas, onde as $n - n_m$ variáveis não cobertas por $\mathcal{F}$ podem assumir qualquer valor em {0,1}. Logo,

$$\tau(\mathcal{F}) = 2^{-n}(2^n\tau(\mathcal{F}') + 2^{n-n_m}) = \tau(\mathcal{F}') + 2^{-n_m} \geq \sum_{i=1}^{m-1} 2^{-n_i} + 2^{-n_m}\sigma(\mathcal{F})$$

Teorema 3 *Se* $\mathcal{F}$ *e* $\mathcal{F}'$ *são duas famílias geradas pela* Busca por Resolução *com* $\mathcal{F}'$ *obtida após* $\mathcal{F}$, *então* $\sigma(\mathcal{F}) < \sigma(\mathcal{F}')$.

Prova Vamos mostrar que $\sigma(\mathcal{F})$ aumenta estritamente a cada execução da atualização da família. Sendo $\mathcal{F} = [C_1, \ldots, C_m]$ a família corrente, a nova família $\mathcal{F}'$ pode ser obtida de duas maneiras:

1. $S \not\subseteq U_{\mathcal{F}}$: a família é aumentada para $\mathcal{F}' = [C_1, \ldots, C_m, C_{m+1}]$, onde $C_{m+1} = S$. Claramente,

$$\sigma(\mathcal{F}') = \sum_{i=1}^{m} 2^{-n_i} + 2^{-n_{m+1}} > \sum_{i}^{m} 2^{n_i} = \sigma(\mathcal{F}'). \tag{37}$$

2. $S \subseteq U_{\mathcal{F}}$: $\mathcal{F} = [C_1, \ldots, C_m]$ é substituída por $\mathcal{F}' = [C_1, \ldots, C_{k-1}, R]$ e então possivelmente estendida por alguns dos nogoods $C_{k+1}, \ldots, C_m$. Primeiramente, calculamos $\sigma(\mathcal{F})$ e então mostramos que $\sigma(\mathcal{F}') > \sigma(\mathcal{F})$. Como $n_{i+1} > n_i,\ i = 1, \ldots, m-1$,

$$\begin{aligned}
\sigma(\mathcal{F}) &= \sum_{i=1}^{k-1} 2^{-n_i} + \sum_{i=k}^{m} 2^{-n_i} \\
&= \sum_{i=1}^{k-1} 2^{-n_i} + 2^{-n_k}(1 + 2^{n_k - n_{k+1}} + \ldots + 2^{n_k - n_m}) \\
&< \sum_{i=1}^{k-1} 2^{-n_i} + 2^{-n_k} * 2
\end{aligned}$$

Por outro lado, como $R \subseteq \bigcup_{i=1}^{k}(C_i - \{w_i\})$, toda variável coberta por $[C_1, \ldots, C_{k-1}, R]$ é coberta por $[C_1, \ldots, C_{k-1}, C_k]$. Além disso, a variável associada a w_k não é coberta por $[C_1, \ldots, C_{k-1}]$, pois w_k é o representante de C_k, de modo que então nem ele e nem sua negação aparecem na família $[C_1 \ldots C_{k-1}]$. Sabemos também que nem w_k

nem $\overline{w_k}$ aparecem em R. Logo, o número de literais cobertos por $\mathcal{F}'$, denotado por n', é tal que $n' \leq n_k - 1$. Portanto,

$$\sigma(\mathcal{F}') = \sum_{i=1}^{k-1} 2^{-n_i} + 2^{-n'} \geq \sum_{i=1}^{k-1} 2^{-n_i} + 2^{-(n_k-1)} = \sum_{i=1}^{k-1} 2^{-n_i} + 2 * 2^{-n_k} > \sigma(\mathcal{F}) \quad (38)$$

A possível inclusão em $\mathcal{F}'$ de alguns dos *nogoods* $C_{k+1}, \ldots, C_m$ só faria aumentar ainda mais $\sigma(\mathcal{F}')$.

Em qualquer dos casos, o limite inferior da força da família cresce a cada iteração.

Corolário 3 *O método da* Busca por Resolução *converge.*

6.2.8 Otimalidade

Nesta seção, provamos a corretude do método de Busca por Resolução, ou melhor, que o mesmo converge para a solução ótima.

Definição 10 *(Boussier (2008)) Seja $r \in \mathbb{R}$. Um nogood C é r-nogood sse, para todo $U \in \{0,1\}^n$ tal que$C \sqsubseteq U$, `oráculo`$(U) \leq r$.*

Lema 6 *Se C_1 e C_2 são r-nogoods conflitantes com $w \in C_1$ e $\overline{w} \in C_2$, então $C_3 = C_1 \nabla C_2$ é também r-nogood.*

Prova Suponha por absurdo que C_1 e C_2 são dois *r-nogoods* e $C_3 = (C_1 - w) \cup (C_2 - \overline{w})$, não é um *r-nogood*. Seja $U \in \{0,1\}^n$ tal que $C_3 \sqsubseteq U$ e `oráculo`$(U) > r$. Então $C_1 - w \sqsubseteq C_3 \sqsubseteq U$ e $C_2 - \overline{w} \sqsubseteq C_3 \sqsubseteq U$. Por outro lado, $w \in U$ ou $\overline{w} \in U$.

- Se $w \in U$ então $C_1 \sqsubseteq U$, pois $C_1 - w \sqsubseteq U$. Logo, `oráculo`$(U) \leq r$, porque C_1 é um *r-nogood*.
- Se $\overline{w} \in U$ então $C_2 \sqsubseteq U$, pois $C_2 - \overline{w} \sqsubseteq U$. Logo, `oráculo`$(U) \leq r$, porque C_2 é um *r-nogood*.

Teorema 4 *Seja r^* o último valor de record obtido pelo Algoritmo de Busca por Resolução. Então `oráculo`$(U) \leq r^*$ $\forall U \in \{0,1\}^n$*

Prova Considere a lista $R_1, R_2, \ldots, R_N$ de todos os *nogoods* gerados ao longo do algoritmo, ordenados segundo o momento em que foram gerados. Para todo $k = 1, \ldots, N$, R_k foi gerado tendo `oráculo`$(R_k) \leq r_k \leq r^*$ ou $R_k = R_i \nabla R_j$ para $i, j < k$, onde r_k é o valor corrente de *record* quando R_k foi gerado.

Vamos mostrar por indução que R_k é *r^*-nogood*, para todo $k = 1, \ldots, N$.

Trivialmente, R_1 é *r^*-nogood*, pois R_1 foi gerado satisfazendo `oráculo`$(R_1) \leq r_1 \leq r^*$. Suponha que R_k seja um *r^*-nogood* para $k = 1, \ldots, n$. Se `oráculo`$(R_{n+1}) \leq r_{n+1} \leq r^*$, segue que R_{n+1} é *r^*-nogood*. Do contrário, $R_{n+1} = R_i \nabla R_j$, para $i, j \in \{1, \ldots, n\}$ e R_{n+1} é *r^*-nogood* pelo Lema 6.

Considere agora $\mathcal{F} = [C_1, \ldots, C_m]$ a última família gerada. Tome $U \in \{0,1\}^n$. Como $X(\mathcal{F}) = \{0,1\}^n$, então $C_i \sqsubseteq U$ para algum $i = 1, \ldots, m$. Como C_i é um *r^*-nogood*,

concluímos que $\texttt{oráculo}(U) \leq r^*$

Corolário 4 *O valor de record retornado pelo Algoritmo de* Busca por Resolução *é o valor ótimo do problema.*

6.3 Algoritmo de Busca por Resolução para CLIQUE PONDERADA

Nesta seção, apresentamos o nosso algoritmo de Busca por Resolução para CLIQUE PONDERADA. Como visto nas seções anteriores, para a descrição do algoritmo, precisamos definir os procedimentos `oráculo` e obstáculo, assim como as políticas de mergulho e recomeço.

6.3.1 Oráculo

Na implementação clássica do algoritmo de Busca por Resolução, o procedimento `oráculo` é responsável apenas por prover um limite superior para uma solução parcial U. Ele é utilizado como uma caixa preta. No nosso `oráculo`, utilizamos a heurística de coloração ponderada para prover um limite superior, assim como a ordem de mergulho.

No problema CLIQUE PONDERADA, uma solução parcial é um vetor indexado pelos vértices do grafo $G = (V, E)$. Dado uma solução parcial U, definimos os conjuntos C, que contém uma clique maximal em $\{j \in V | u_j = 1\}$, e P, formado pelos vértices que podem estender a clique C. Esses conjuntos são usados na descrição do `oráculo` (Ver Algoritmo 6.11): C é construído iterativamente no Linha 7; P começa com V e é atualizado com a remoção de todo vértice j tal que $u_j = 0$ ou $j \notin \bigcap_{v \in C} N(v)$ (Linhas 6 e 11).

Trivialmente, se existem vértices não vizinhos i e j tais que $u_i = 1$ e $u_j = 1$, então $X(U)$ não contém nenhuma solução viável, de modo que $\texttt{oráculo}(U) = -\infty$. Caso contrário, um limite superior para U é obtido pelo peso de C mais um limite superior para a maior clique ponderada em P, que será dado por uma heurística de coloração ponderada.

A heurística de coloração ponderada recebe um vetor de bits com os vértice de P e uma ordem ρ (ordem inicial dos vértices), representando a ordem em que os bits estão organizados no vetor de bits. Ela é passada implicitamente juntamente com o vetor de bits P. Ao longo deste trabalho, temos utilizado duas ordens iniciais ρ: a ordem **menor peso primeiro** e a ordem **maior grau ponderado primeiro**. A ordem ρ influencia a sequência em que os vértices são analisados durante a coloração ponderada. Por outro lado, a heurística de coloração ponderada devolve uma outra ordem π dos vértices de P e o vetor *color* com o valor das colorações ponderadas dos subgrafos, ou seja, $color[\pi_i]$ é o valor da coloração ponderada do subgrafo $G[\pi_1, \ldots, \pi_i]$.

Por simplicidade e para manter a uniformidade com as seções anteriores, man-

temos a chamada ao `oráculo` como `oráculo`(U), quando, na verdade, em nosso caso, deveria ser `oráculo`$(U, C, P, \pi, color, cont)$. Os demais parâmetros são retornados pelo `oráculo` e podem ser usados em outros procedimentos

Algorithm 6.11 `oráculo`(U)

1: $\boxed{C \leftarrow \emptyset}$
2: $\boxed{P \leftarrow V}$
3: **for** $j = 1$ até n **do**
4: **if** $u_j = 1$ **then**
5: **if** $j \in P$ **then**
6: $\boxed{P \leftarrow P \cap N(j)}$;
7: $\boxed{C \leftarrow C \cup \{j\}}$;
8: **else**
9: **return** $-\infty$ ▷ Inviabilidade Encontrada
10: **else if** $u_j = 0$ **then**
11: $\boxed{P \leftarrow P \setminus \{j\}}$;
12: `BITCOLOR`$(P, \rho, \pi, color, cont)$
13: **return** $w(C) + color[pi_c ont]$

6.3.2 Obstáculo e política de mergulho

A partir da função `oráculo`, apresentada no Algoritmo 6.11, definimos um versão para o procedimento obstáculo rápido (Ver Algoritmo 6.12). O `oráculo` apresentado aqui não é totalmente caixa preta, uma vez que podemos acessar a ordem devolvida pela heurística de coloração ponderada. Tal ordem será usada para definir a política de mergulho: Enquanto o limite superior for superior a record e U ainda tem variável livre, o último vértice colorido pela heurística de coloração é escolhido e a variável associada a ele é fixada em 1.

Algorithm 6.12 obstáculo(U, record, S)

1: $bound \leftarrow$ `oráculo`(U)
2: **while** $U \notin \{0,1\}^n$ e $bound > record$ **do**
3: $j \leftarrow \pi[cont]$
4: $u_j \leftarrow 1$
5: $bound \leftarrow$ `oráculo`(U) ▷ `oráculo`$(U, C, P, \pi, color, cont)$
6: **if** bound ¿ record **then**
7: $record \leftarrow bound$
8: $S \leftarrow U$

6.3.3 Política de recomeço

Para cada ordem inicial ρ, vamos propor duas estratégias de recomeço:

R_1 : Escolha a primeira variável que pode ser representante na ordem inicial.

R_2 : Escolha a última variável que pode ser representante na ordem inicial.

Para cada ordem inicial, o critério de recomeço tem um significado diferente. Por exemplo, na ordem de **menor peso primeiro**, a estratégia R_1 prioriza inverter a fixação da variável associada ao vértice com o menor peso, enquanto a estratégia R_2 prioriza fixar no valor oposto a variável do vértice com o maior peso.

Em Chvátal (1997), não temos uma indicação sólida para um bom critério para a escolha da política de recomeço. Em nosso caso, para cada critério de ordenação inicial dos vértices, vamos escolher uma política de recomeço mais adequada com base em testes computacionais preliminares.

A Tabela 19 apresenta o número de chamadas ao `oráculo` e o tempo de execução do algoritmo de Busca por Resolução utilizando as duas estratégias de recomeço com a ordem de **menor peso primeiro**. Apesar da estratégia R_1 obter o menor número de chamadas ao `oráculo` e o menor tempo de execução na maioria das vezes, as diferenças são muito pouco expressivas, de maneira que podemos considerar as duas estratégias comparáveis.

Tabela 19 – Resultados das estratégias de recomeço para a ordem de **menor peso primeiro** para o Algoritmo de Busca por Resolução usando o Algoritmo 6.12 como obstáculo.

Instância		**Chamadas do `oráculo`**		**Tempo de execução**	
Nome	**(n,p)**	R_1	R_2	R_1	R_2
brock200_1	(200,0.75)	6.03e+04	**5.70e+04**	0.25	**0.24**
brock200_2	(200,0.50)	**3.20e+03**	3.23e+03	**0.06**	0.06
brock200_3	(200,0.61)	**5.68e+03**	5.77e+03	**0.07**	0.07
brock200_4	(200,0.66)	1.50e+04	**1.48e+04**	**0.09**	0.09
brock400_1	(400,0.75)	**1.74e+07**	1.76e+07	**108.91**	108.96
brock400_2	(400,0.75)	**2.06e+07**	2.08e+07	**125.22**	127.85
brock400_3	(400,0.75)	**1.57e+07**	1.58e+07	**96.11**	98.78
brock400_4	(400,0.75)	**1.04e+07**	1.08e+07	**64.50**	67.30
p_hat300-1	(300,0.24)	**5.95e+02**	6.01e+02	**0.08**	0.08
p_hat300-2	(300,0.49)	**1.07e+04**	1.14e+04	**0.10**	0.11
p_hat300-3	(300,0.74)	**3.02e+05**	3.03e+05	**1.69**	1.72
p_hat500-1	(500,0.25)	**3.79e+03**	3.88e+03	0.16	**0.16**
p_hat500-2	(500,0.50)	**1.35e+05**	1.37e+05	**1.35**	1.38
p_hat500-3	(500,0.75)	**1.05e+08**	1.10e+08	**829.97**	898.94
p_hat700-1	(700,0.25)	**1.17e+04**	1.19e+04	0.59	**0.40**
p_hat700-2	(700,0.50)	4.79e+06	**4.67e+06**	**51.67**	51.68
p_hat1000-1	(1000,0.24)	**5.57e+04**	5.67e+04	**1.46**	1.50
p_hat1000-2	(1000,0.49)	**1.33e+08**	1.34e+08	**1857.04**	1898.04
p_hat1500-1	(1500,0.25)	**3.60e+05**	3.71e+05	**11.18**	11.54

Fonte: Elaborada pelo autor.

Já a Tabela 20 apresenta uma comparação entre as estratégias de recomeço para a ordem de **maior grau ponderado primeiro**. A estratégia R_2 obtém resultados melhores em mais instâncias. Em alguns casos, o número de chamadas ao `oráculo` é reduzido em duas ordens de magnitude e acompanhado de uma grande redução no tempo de execução. A partir dessas instâncias, podemos notar o impacto da estratégia de recomeço no algoritmo de *Busca por Resolução*. Para a ordem de **maior grau ponderado primeiro**, consideramos a estratégia R_2 como uma boa escolha para a política

de questionamento.

Tabela 20 – Resultados das estratégias de recomeço para a ordem de **maior grau ponderado primeiro** para o Algoritmo de Busca por Resolução usando o Algoritmo 6.12 como obstáculo

Instância		**Chamadas do `oráculo`**		**Tempo de execução**	
Nome	**(n,p)**	R_1	R_2	R_1	R_2
brock200_1	(200,0.75)	6.03e+04	**3.59e+04**	0.25	**0.19**
brock200_2	(200,0.50)	**3.20e+03**	3.20e+03	**0.06**	0.07
brock200_3	(200,0.61)	5.68e+03	**4.06e+03**	0.07	**0.06**
brock200_4	(200,0.66)	1.50e+04	**1.35e+04**	**0.09**	0.10
brock400_1	(400,0.75)	**1.74e+07**	2.21e+07	**106.30**	155.66
brock400_2	(400,0.75)	**2.06e+07**	3.09e+07	**125.56**	211.37
brock400_3	(400,0.75)	**1.57e+07**	2.35e+07	**95.86**	165.28
brock400_4	(400,0.75)	**1.04e+07**	1.15e+07	**64.90**	82.53
p_hat300-1	(300,0.24)	5.95e+02	**4.03e+02**	0.09	**0.08**
p_hat300-2	(300,0.49)	1.07e+04	**2.09e+03**	0.11	**0.06**
p_hat300-3	(300,0.74)	3.02e+05	**2.93e+04**	1.68	**0.25**
p_hat500-1	(500,0.25)	3.79e+03	**3.59e+03**	0.17	**0.16**
p_hat500-2	(500,0.50)	1.35e+05	**7.97e+03**	1.35	**0.18**
p_hat500-3	(500,0.75)	1.04e+08	**1.44e+06**	825.42	**15.73**
p_hat700-1	(700,0.25)	1.17e+04	**1.16e+04**	0.37	**0.37**
p_hat700-2	(700,0.50)	4.79e+06	**2.95e+04**	51.44	**0.74**
p_hat1000-1	(1000,0.24)	5.57e+04	**4.90e+04**	1.46	**1.42**
p_hat1000-2	(1000,0.49)	1.33e+08	**1.22e+06**	1852.11	**30.75**
p_hat1500-1	(1500,0.25)	3.60e+05	**3.15e+05**	11.12	**10.70**

Fonte: Elaborada pelo autor.

Na próxima subseção, apresentamos um procedimento de obstáculo que realiza menos chamadas ao `oráculo` para encontrar um nogood. Este procedimento será chamado de *obstáculo modificado*.

6.3.4 Obstáculo Modificado

A heurística de coloração ponderada, além de prover um limite superior para o problema, devolve uma ordenação dos vértices e limites superiores parciais que podem ser aproveitados para a identificação de um *nogood* de maneira mais rápida. Durante a fase de crescimento, se todas variáveis associadas aos vértices v tal que $w(C) + color[v] > record$ forem fixados em zero em U, então U será um nogood, uma vez que passamos a ter $oraculo(U) \leq record$. A implementação dessa estratégia é facilitada pelo fato de os valores em *color* estarem ordenados em ordem não-decrescente quando seguimos a seqüencia π.

No Algoritmo 6.13, apresentamos um obstáculo modificado que realiza apenas uma única chamada ao `oráculo` para identificar um *nogood*.

6.3.5 Política de recomeço para obstáculo modificado

Novamente, testamos a influência das duas políticas de recomeço com o obstáculo modificado. A Tabela 21 apresenta o número de chamadas e o tempo de execução para cada política de recomeço para a ordem de **menor peso primeiro**. A política de recomeço

Algorithm 6.13 obstáculo modificado$(U, record, S)$

```
1: bound ← oraculo(U)
2: if w(C) > record then
3:     record ← w(C)
4: if bound > record then
5:     for i ← cont até 1 do
6:         j ← π[i]
7:         if w(C) + color[j] ≤ record then
8:             pare
9:         u_j ← 0
10: S ← U
```

Tabela 21 – Resultados das estratégias de recomeço para a ordem de **menor peso primeiro** para o obstáculo modificado

Instância		**Chamadas do `oráculo`**		**Tempo de execução**	
Nome	**(n,p)**	R_1	R_2	R_1	R_2
brock200_1	(200,0.75)	**3.00e+04**	3.74e+04	**0.13**	0.15
brock200_2	(200,0.50)	**1.60e+03**	2.08e+03	**0.06**	0.06
brock200_3	(200,0.61)	**2.85e+03**	3.63e+03	**0.05**	0.06
brock200_4	(200,0.66)	**7.53e+03**	1.05e+04	**0.06**	0.07
brock400_1	(400,0.75)	**8.76e+06**	1.89e+07	**43.69**	92.79
brock400_2	(400,0.75)	**1.04e+07**	2.08e+07	**51.15**	103.32
brock400_3	(400,0.75)	**7.90e+06**	1.87e+07	**39.70**	92.03
brock400_4	(400,0.75)	9.59e+06	**9.07e+06**	**47.46**	48.81
p_hat300-1	(300,0.24)	**3.09e+02**	3.92e+02	0.08	**0.08**
p_hat300-2	(300,0.49)	**5.17e+03**	7.64e+03	**0.07**	0.08
p_hat300-3	(300,0.74)	**1.49e+05**	2.48e+05	**0.75**	1.25
p_hat500-1	(500,0.25)	**1.91e+03**	2.80e+03	**0.13**	0.13
p_hat500-2	(500,0.50)	**6.79e+04**	1.31e+05	**0.61**	1.05
p_hat500-3	(500,0.75)	**5.46e+07**	9.64e+07	**396.17**	710.15
p_hat700-1	(700,0.25)	**5.89e+03**	8.84e+03	**0.19**	0.21
p_hat700-2	(700,0.50)	**2.32e+06**	4.70e+06	**21.24**	43.35
p_hat1000-1	(1000,0.24)	**2.84e+04**	4.63e+04	**0.47**	0.62
p_hat1000-2	(1000,0.49)	**6.55e+07**	1.39e+08	**821.84**	1722.87
p_hat1500-1	(1500,0.25)	**1.83e+05**	3.76e+05	**2.65**	4.71

Fonte: Elaborada pelo autor.

R_1 mostrou uma supremacia em relação à política R_2. Em alguns casos, a redução no número de chamadas chega a ser de uma ordem de magnitude e o algoritmo chega a ser duas vezes mais rápido. Portanto, considerando a **ordem de menor peso primeiro**, a estratégia de recomeço R_1 obtém resultados mais expressivos.

A Tabela 22 apresenta a comparação entre as estratégias de recomeço com a ordem **maior grau ponderado primeiro**. Com essa ordem, a política de recomeço R_1 obteve os melhores resultados no maior número de instâncias. Das 19 instâncias testadas, essa política obteve o melhor tempo em 17 delas.

Julgamos que estas evidências credenciam a estratégia R_1 para ser utilizada nos testes mais intensivos com o obstáculo modificado, em ambas as ordens.

Tabela 22 – Resultados das estratégias de recomeço para a ordem de **maior grau ponderado primeiro**.

Instância		**Chamadas do** `oráculo`		**Tempo de execução**	
Nome	**(n,p)**	R_1	R_2	R_1	R_2
brock200_1	(200,0.75)	**2.08e+04**	3.08e+04	**0.105**	0.140
brock200_2	(200,0.50)	**1.96e+03**	2.10e+03	**0.056**	0.057
brock200_3	(200,0.61)	**2.35e+03**	2.70e+03	**0.052**	0.054
brock200_4	(200,0.66)	**7.69e+03**	1.07e+04	**0.063**	0.072
brock400_1	(400,0.75)	**1.44e+07**	2.01e+07	**69.192**	106.008
brock400_2	(400,0.75)	**1.99e+07**	2.30e+07	**93.778**	120.083
brock400_3	(400,0.75)	**1.51e+07**	1.82e+07	**72.683**	96.580
brock400_4	(400,0.75)	2.12e+07	**9.24e+06**	99.242	**53.336**
p_hat300-1	(300,0.24)	**2.13e+02**	3.66e+02	0.077	**0.077**
p_hat300-2	(300,0.49)	**1.22e+03**	1.88e+03	**0.048**	0.050
p_hat300-3	(300,0.74)	**1.82e+04**	4.02e+04	**0.130**	0.270
p_hat500-1	(500,0.25)	**2.14e+03**	2.94e+03	**0.129**	0.132
p_hat500-2	(500,0.50)	**4.76e+03**	1.25e+04	**0.097**	0.171
p_hat500-3	(500,0.75)	**8.78e+05**	3.99e+06	**6.794**	35.890
p_hat700-1	(700,0.25)	**7.53e+03**	8.40e+03	**0.199**	0.207
p_hat700-2	(700,0.50)	**1.72e+04**	6.18e+04	**0.300**	0.909
p_hat1000-1	(1000,0.24)	**3.47e+04**	4.25e+04	**0.516**	0.584
p_hat1000-2	(1000,0.49)	**7.77e+05**	3.77e+06	**11.759**	60.423
p_hat1500-1	(1500,0.25)	**1.99e+05**	3.47e+05	**2.865**	4.291

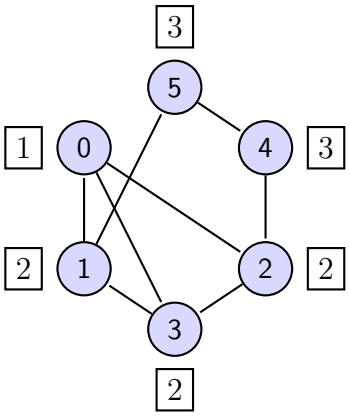
Fonte: Elaborada pelo autor.

6.3.6 Exemplo de execução do Algoritmo

Para ilustrar o funcionamento do Algoritmo de Busca por Resolução, apresentamos o resultado de sua aplicação a um exemplo.

Exemplo 13 *Considere o seguinte grafo ponderado:*

Figura 21 – Grafo ponderado com os vértices ordenados seguindo a ordem do **menor peso primeiro**.



Fonte: Elaborada pelo autor.

A Tabela 23 exemplifica uma execução do algoritmo Busca por Resolução com a ordem inicial de menor peso primeiro, obstáculo modificado e a política de recomeço R_1. Apresentamos as seguintes informações para o acompanhamento da execução do algoritmo : a família path-like *corrente, representada pelo conjunto de nogoods $\mathcal{F}$ e seus representantes W, o nogood S obtido pela exploração da família* path-like *e o melhor limite inferior até momento, representado por* record*. Destacamos a iteração 3. O nogood $x_0x_1\overline{x_2x_3x_4x_5}$*

foi obtido pela exploração da família path-like $\mathcal{F} = [\overline{x_1}x_2x_3x_4x_5, \overline{x_0}x_1\overline{x_2}x_3x_4x_5]$ *e* $W = [\overline{x_3}, \overline{x_1}]$*. O mesmo será utilizado para atualização da família. Primeiramente, aplicamos a regra de resolução com o nogood* $x_0x_1\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$ *e os nogoods armazenados pela família, obtendo o nogood* $\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$*. Na fase de absorção, o nogood* $\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$ *assume a posição número 1 na família e* $\overline{x_2}$ *será escolhido como representante. Depois o nogood da posição 2 não é mantido pelo procedimento* `ReciclandoNogood`*.*

Tabela 23 – Execução do Algoritmo de Busca de Resolução utilizando o obstáculo modificado.

Iteração	**Família** ($\mathcal{F}$)	**Representante** (W)	**S**	record
0	$\emptyset$	$\emptyset$	$\overline{x_0}\overline{x_1}\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$	0
1	$[\overline{x_0}x_1x_2x_3x_4x_5]$	$[\overline{x_0}]$	$x_0\overline{x_1}\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$	1
2	$[\overline{x_1}x_2x_3x_4x_5]$	$[\overline{x_1}]$	$x_1\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$	2
3	$[\overline{x_1}x_2x_3x_4x_5, \overline{x_0}x_1\overline{x_2}x_3x_4x_5]$	$[\overline{x_1}, \overline{x_0}]$	$x_0x_1\overline{x_2}\overline{x_3}\overline{x_4}\overline{x_5}$	3
4	$[\overline{x_2}x_3x_4x_5]$	$[\overline{x_2},]$	$x_2\overline{x_3}\overline{x_4}\overline{x_5}$	3
5	$[\overline{x_3}x_4x_5]$	$[\overline{x_3},]$	$\overline{x_1}\overline{x_2}x_3\overline{x_4}\overline{x_5}$	3
6	$[\overline{x_3}x_4x_5, \overline{x_1}\overline{x_2}x_3\overline{x_4}x_5]$	$[\overline{x_3}, \overline{x_1}]$	$\overline{x_0}x_1\overline{x_2}x_3\overline{x_4}\overline{x_5}$	4
7	$[\overline{x_3}x_4x_5, \overline{x_1}\overline{x_2}x_3\overline{x_4}x_5, \overline{x_0}x_1\overline{x_2}x_3\overline{x_4}x_5]$	$[\overline{x_3}, \overline{x_1}, \overline{x_0}]$	$x_0x_1\overline{x_2}x_3\overline{x_4}\overline{x_5}$	5
8	$[\overline{x_3}x_4x_5, \overline{x_2}x_4x_5]$	$[\overline{x_3}, \overline{x_2}]$	$x_2x_3\overline{x_4}\overline{x_5}$	5
9	$[\overline{x_4}\overline{x_5}]$	$[\overline{x_4}]$	$x_4\overline{x_5}$	5
10	$[\overline{x_5}]$	$[\overline{x_5}]$	$\overline{x_4}x_5$	5
11	$[\overline{x_5}, \overline{x_4}x_5]$	$[\overline{x_5}, \overline{x_4}]$	x_4x_5	6
12	$\emptyset$	$\emptyset$	$\emptyset$	6

Fonte: Elaborada pelo autor.

6.3.7 Obstáculo Modificado com fase de decrescimento

Para o nosso obstáculo modificado, podemos ainda definir uma fase de decrescimento simplificada. Para liberar uma fixação $s_j = 0$ tal que $j \in P$, precisamos mostrar que um limite superior para a clique ponderada contendo $C \cup \{j\}$, que é dado pela soma de $w(C \cup \{j\})$ com um limite superior para a clique ponderada de $P \cap N(j)$, é inferior ou igual a *record*. A segunda parcela desse limite superior pode ser obtida encontrando o último vértice vizinho de j em P na ordem π dada pela heurística de coloração ponderada. Em outras palavras, o valor que devemos comparar a *record* é

$$w(C) + w(v) + color[\pi[u]], \text{ onde } u = \max\{k \text{ — } \pi[k] \in P \cap N(v)\} \tag{39}$$

O Algoritmo 6.14 apresenta o procedimento `obstáculo_modificado` com a fase de decrescimento aqui proposto.

Algorithm 6.14 obstáculo modificado com decrescimento$(U, record, S)$

```
1: Inicialize a pilha vazia
2: bound ← oraculo(U)
3: if w(C) > record then
4:     record ← w(C)
5: R ← P
6: if bound > record then
7:     for i ← cont até 1 do                          ▷ ordem reversa
8:         j ← π[i]
9:         if w(C) + color[j] ≤ record then
10:            pare
11:        u_j ← 0
12:        R ← R \ {j};
13:        Empilhe j na pilha
14: S ← U
15: while a pilha não estiver vazia do
16:     Desempilhe v da pilha
17:     i ← max{k|π[k] ∈ R ∩ N(v)}
18:     u ← π[i]
19:     UB ← w(C) + w(v) + color[u]
20:     if UB ≤ record then
21:         S_j ← ∗
22:         R ← R ∪ {v}
```

A Tabela 24 apresenta uma comparação entre os algoritmos de Busca por Resolução com a regra de recomeço R_1 e com procedimentos obstáculos apresentados (obstáculo padrão, obstáculo modificado e obstáculo modificado com decrescimento), para a ordem de **menor peso primeiro**. Podemos ver que a utilização do obstáculo modificado consegue uma boa redução do números de chamadas total ao `oráculo`, como também uma boa redução do tempo de execução com relação ao obstáculo padrão. Já a redução obtida pela adoção da fase de decrescimento acontece em uma escala menor, mas consideramos que ela é ainda evidente.

Tabela 24 – Comparação entre os procedimentos obstáculo padrão, modificado e modificado com fase de decrescimento considerando a ordem de menor peso primeiro.

Instância		Chamadas do `oráculo`			Tempo de Execução		
Nome	(n,p)	Obstáculo Padrão	Obstáculo Modificado	Obstáculo Modificado com decrescimento	Obstáculo Padrão	Obstáculo Modificado	Obstáculo Modificado com decrescimento
brock200_1	(200,0.75)	6.03e+04	3.00e+04	**2.96e+04**	0.246	**0.129**	**0.128**
brock200_2	(200,0.50)	3.20e+03	1.60e+03	**1.58e+03**	**0.063**	**0.055**	**0.055**
brock200_3	(200,0.61)	5.68e+03	2.85e+03	**2.77e+03**	0.066	**0.053**	**0.053**
brock200_4	(200,0.66)	1.50e+04	7.53e+03	**7.37e+03**	0.092	**0.063**	**0.065**
brock400_1	(400,0.75)	1.74e+07	8.76e+06	**8.59e+06**	108.911	43.692	**43.371**
brock400_2	(400,0.75)	2.06e+07	1.04e+07	**1.02e+07**	125.219	51.151	**50.979**
brock400_3	(400,0.75)	1.57e+07	7.90e+06	**7.76e+06**	96.113	39.702	**39.058**
brock400_4	(400,0.75)	1.04e+07	9.59e+06	**9.42e+06**	64.497	47.456	**46.678**
p_hat300-1	(300,0.24)	5.95e+02	3.09e+02	**3.01e+02**	**0.079**	**0.078**	**0.076**
p_hat300-2	(300,0.49)	1.07e+04	5.17e+03	**5.07e+03**	0.103	**0.066**	**0.063**
p_hat300-3	(300,0.74)	3.02e+05	1.49e+05	**1.46e+05**	1.691	**0.751**	**0.742**
p_hat500-1	(500,0.25)	3.79e+03	1.91e+03	**1.87e+03**	0.164	**0.126**	**0.121**
p_hat500-2	(500,0.50)	1.35e+05	6.79e+04	**6.68e+04**	1.348	0.610	**0.578**
p_hat500-3	(500,0.75)	1.05e+08	5.46e+07	**5.37e+07**	829.970	396.175	**386.770**
p_hat700-1	(700,0.25)	1.17e+04	5.89e+03	**5.76e+03**	0.589	**0.191**	**0.186**
p_hat700-2	(700,0.50)	4.79e+06	2.32e+06	**2.30e+06**	51.671	**21.243**	21.292
p_hat1000-1	(1000,0.24)	5.57e+04	2.84e+04	**2.78e+04**	1.456	**0.470**	**0.465**
p_hat1000-2	(1000,0.49)	1.33e+08	6.55e+07	**6.44e+07**	1857.036	821.838	**809.260**
p_hat1500-1	(1500,0.25)	3.60e+05	1.83e+05	**1.79e+05**	11.179	2.652	**2.632**

Fonte: Elaborada pelo autor.

A Tabela 25 apresenta uma comparação similar, agora para a ordem de **maior grau ponderado primeiro**. Novamente, conseguimos uma redução significativa com a utilização do procedimento de `obstáculo modificado` e uma redução menos expressiva com a fase de decrescimento, tanto no número de chamadas como no tempo de execução total.

Tabela 25 – Comparação entre os procedimentos obstáculo padrão, modificado e modificado com fase de decrescimento considerando a ordem de menor peso primeiro.

Instância		Chamadas do `oráculo`			Tempo de Execução		
Nome	(n,p)	Obstáculo Padrão	Obstáculo Modificado	Obstáculo Modificado com decrescimento	Obstáculo Padrão	Obstáculo Modificado	Obstáculo Modificado com decrescimento
brock200_1	(200,0.75)	3.59e+04	2.08e+04	**1.86e+04**	0.189	**0.105**	**0.097**
brock200_2	(200,0.50)	3.20e+03	1.96e+03	**1.82e+03**	**0.065**	**0.056**	**0.056**
brock200_3	(200,0.61)	4.06e+03	2.35e+03	**2.12e+03**	0.064	**0.052**	**0.054**
brock200_4	(200,0.66)	1.35e+04	7.69e+03	**7.03e+03**	0.095	**0.063**	**0.062**
brock400_1	(400,0.75)	2.21e+07	1.44e+07	**1.28e+07**	155.657	69.192	**63.060**
brock400_2	(400,0.75)	3.09e+07	1.99e+07	**1.77e+07**	211.372	93.778	**85.161**
brock400_3	(400,0.75)	2.35e+07	1.51e+07	**1.34e+07**	165.284	72.683	**65.182**
brock400_4	(400,0.75)	**1.15e+07**	2.12e+07	1.88e+07	**82.529**	99.242	89.161
p_hat300-1	(300,0.24)	4.03e+02	2.13e+02	**2.03e+02**	**0.082**	**0.077**	**0.075**
p_hat300-2	(300,0.49)	2.09e+03	1.22e+03	**1.14e+03**	0.060	**0.048**	**0.048**
p_hat300-3	(300,0.74)	2.93e+04	1.82e+04	**1.62e+04**	0.247	**0.130**	**0.122**
p_hat500-1	(500,0.25)	3.59e+03	2.14e+03	**2.01e+03**	0.160	**0.129**	**0.121**
p_hat500-2	(500,0.50)	7.97e+03	4.76e+03	**4.38e+03**	0.177	**0.097**	**0.095**
p_hat500-3	(500,0.75)	1.44e+06	8.78e+05	**7.94e+05**	15.730	6.794	**6.310**
p_hat700-1	(700,0.25)	1.16e+04	7.53e+03	**7.08e+03**	0.367	**0.199**	**0.190**
p_hat700-2	(700,0.50)	2.95e+04	1.72e+04	**1.62e+04**	0.740	0.300	**0.287**
p_hat1000-1	(1000,0.24)	4.90e+04	3.47e+04	**3.28e+04**	1.425	0.516	**0.496**
p_hat1000-2	(1000,0.49)	1.22e+06	7.77e+05	**7.19e+05**	30.749	11.759	**10.942**
p_hat1500-1	(1500,0.25)	3.15e+05	1.99e+05	**1.87e+05**	10.698	2.865	**2.728**

Fonte: Elaborada pelo autor.

6.3.8 Comparação com o algoritmo BITCLIQUE

Realizamos um experimento computacional breve para comparar o desempenho do algoritmo de Busca por Resolução com Branch & Bound apresentado no Capítulo 3. Considerando a ordem de **menor peso primeiro** (Ver Tabela 26), o algoritmo de Busca por Resolução realizou menos chamadas ao `oráculo` do que o Algoritmo de Branch & Bound. Já com a ordem de **maior grau ponderado primeiro**, o algoritmo de Branch & Bound foi com maior freqüência aquele que gerou o menor número de chamadas ao `oráculo`. De maneira geral, porém, para a mesma ordem inicial dos vértices, os algoritmos possuem um comportamento bastante similar com respeito ao número de chamadas ao `oráculo`.

Mesmo quando o algoritmo de Busca por Resolução realiza menos chamadas ao `oráculo` do que o Algoritmo de Branch & Bound (**ordem de menor peso primeiro**), o tempo de execução daquele fica superior ao deste (Ver Tabela 27).

Na próxima seção, propomos um algoritmo que realiza uma cooperação entre o algoritmo de Bonecas Russas e o Algoritmo de Busca por Resolução. A ideia é realizar uma exploração parcial do espaço de busca usando o algoritmo de Bonecas Russas e, depois, o restante da exploração passe a ser realizada pelo Algoritmo de Busca por Resolução, fazendo o uso da informação coletada durante a resolução das bonecas.

Tabela 26 – Comparação entre o número de chamadas ao `oráculo` pelo Algoritmo de Busca por Resolução utilizando obstáculo modificado com decrescimento e pelo Algoritmo de Branch & Bound, nas duas ordem testadas. ★ indica que o algoritmo consegue o menor número de chamadas ao `oráculo` entre os quatros algoritmos considerados.

Instância		**Ordem de menor peso primeiro**		**Ordem de maior grau ponderado primeiro**	
Instância		**Chamadas do `oráculo`**		**Chamadas do `oráculo`**	
Nome	**(n,p)**	**Obstáculo Modificado com decrescimento**	**Branch & Bound**	**Obstáculo Modificado com decrescimento**	**Branch & Bound**
brock200_1	(200,0.75)	**2.96e+04**	2.97e+04	1.86e+04	**1.80e+04★**
brock200_2	(200,0.50)	**1.58e+03★**	1.62e+03	1.82e+03	**1.79e+03**
brock200_3	(200,0.61)	**2.77e+03**	2.85e+03	2.12e+03	**2.11e+03★**
brock200_4	(200,0.66)	**7.37e+03**	7.47e+03	7.03e+03	**6.81e+03★**
brock400_1	(400,0.75)	**8.59e+06★**	8.65e+06	1.28e+07	**1.10e+07**
brock400_2	(400,0.75)	**1.02e+07★**	**1.02e+07★**	1.77e+07	**1.55e+07**
brock400_3	(400,0.75)	**7.76e+06★**	7.81e+06	1.34e+07	**1.15e+07**
brock400_4	(400,0.75)	9.42e+06	**4.80e+06★**	1.88e+07	**5.20e+06**
p_hat300-1	(300,0.24)	**3.01e+02**	3.09e+02	**2.03e+02★**	2.18e+02
p_hat300-2	(300,0.49)	**5.07e+03**	5.36e+03	1.14e+03	**1.08e+03★**
p_hat300-3	(300,0.74)	**1.46e+05**	1.47e+05	1.62e+04	**1.50e+04★**
p_hat500-1	(500,0.25)	**1.87e+03★**	1.96e+03	2.01e+03	**2.01e+03**
p_hat500-2	(500,0.50)	**6.68e+04**	6.70e+04	4.38e+03	**4.12e+03★**
p_hat500-3	(500,0.75)	5.37e+07	**5.10e+07**	7.94e+05	**6.97e+05★**
p_hat700-1	(700,0.25)	**5.76e+03★**	6.07e+03	7.08e+03	**6.62e+03**
p_hat700-2	(700,0.50)	**2.30e+06**	2.36e+06	1.62e+04	**1.57e+04★**
p_hat1000-1	(1000,0.24)	**2.78e+04★**	2.91e+04	3.28e+04	**2.93e+04**
p_hat1000-2	(1000,0.49)	**6.44e+07**	6.60e+07	7.19e+05	**6.48e+05★**
p_hat1500-1	(1500,0.25)	**1.79e+05**	1.89e+05	1.87e+05	**1.77e+05★**

Fonte: Elaborada pelo autor.

Tabela 27 – Comparação entre o tempo de execução do Algoritmo de Busca por Resolução utilizando obstáculo modificado com decrescimento com o Algoritmo de Branch & Bound, nas duas ordem testadas. ★ indica que o algoritmo consegue o menor tempo de execução entre os quatros algoritmos considerados.

Instância		**Ordem de menor peso primeiro**		**Ordem de maior grau ponderado primeiro**	
		Chamadas ao `oráculo`		**Chamadas ao** `oráculo`	
Nome	**(n,p)**	**Obstáculo Modificado com decrescimento**	**Branch & Bound**	**Obstáculo Modificado com decrescimento**	**Branch & Bound**
brock200_1	(200,0.75)	0.13	**0.10**	0.10	**0.08★**
brock200_2	(200,0.50)	0.05	**0.05**	0.06	**0.05★**
brock200_3	(200,0.61)	0.05	**0.05**	0.05	**0.05★**
brock200_4	(200,0.66)	0.06	**0.06**	0.06	**0.05★**
brock400_1	(400,0.75)	43.37	**29.94★**	63.06	**38.91**
brock400_2	(400,0.75)	50.98	**34.67★**	85.16	**51.84**
brock400_3	(400,0.75)	39.06	**26.93★**	65.18	**38.91**
brock400_4	(400,0.75)	46.68	**17.66★**	89.16	**20.41**
p_hat300-1	(300,0.24)	0.08	**0.07★**	0.08	**0.07**
p_hat300-2	(300,0.49)	0.06	**0.06**	0.05	**0.04★**
p_hat300-3	(300,0.74)	0.74	**0.57**	0.12	**0.10★**
p_hat500-1	(500,0.25)	0.12	**0.12**	0.12	**0.12★**
p_hat500-2	(500,0.50)	0.58	**0.50**	0.10	**0.09★**
p_hat500-3	(500,0.75)	386.77	**296.29**	6.31	**4.57★**
p_hat700-1	(700,0.25)	0.19	**0.18**	0.19	**0.18★**
p_hat700-2	(700,0.50)	21.29	**17.69**	0.29	**0.27★**
p_hat1000-1	(1000,0.24)	0.46	**0.41**	0.50	**0.38★**
p_hat1000-2	(1000,0.49)	809.26	**686.72**	10.94	**9.32★**
p_hat1500-1	(1500,0.25)	2.63	**2.15**	2.73	**1.78★**

Fonte: Elaborada pelo autor.

6.4 BITBR - Cooperação entre Busca por Resolução e Boneca Russas para CLIQUE PONDERADA

Nesta seção, apresentamos um algoritmo exato para o problema da CLIQUE PONDERADA, combinando o algoritmo de Busca por Resolução e o Algoritmo de Bonecas Russas. No método das Bonecas Russas, o problema é decomposto em uma sequência de subproblemas aninhados. O método sugere que os subproblemas devam ser resolvidos seqüencialmente, e a sua solução ótima deve ser memorizada para ser utilizada como critério de poda para os futuros subproblemas. Note que uma subsequência inicial de subproblemas resolvidos pelo método das Bonecas Russas representa uma exploração parcial do espaço de busca, que podemos representar como um *nogood*.

No problema da CLIQUE PONDERADA, dada uma ordem inicial dos vértices $\rho = (\rho_1, \ldots, \rho_n)$, definimos uma sequência de n subproblemas (as bonecas), onde i-ésimo subproblema está associado ao subgrafo $G[\rho_1, \ldots, \rho_i], i \in \{1, \ldots, n\}$. Depois da solução ótima do i-ésimo subproblema, podemos representar a região do espaço de busca explorada pelo seguinte nogood:

$$\overline{x_{\rho_{i+1}}}\ \overline{x_{\rho_{i+2}}}\ \ldots \overline{x_{\rho_n}} \tag{40}$$

Esse nogood diz que, para obter uma solução de valor maior que $c(\rho_i)$, é preciso fixar em 1 pelo menos uma variável entre $i+1$ e n, ou seja, é preciso escolher pelo menos um

vértice entre $\rho_{i+1}, \ldots, \rho_n$

Podemos usar esta ideia em conjunto com uma solução parcial U para gerar um *nogood* fixando, possivelmente, menos variáveis. Sejam C e P encontrados por `oráculo`(U). Analogamente, ao caso da coloração, obtemos um *nogood* se fixamos em 0 todo o vértice v que $w(C) + c(v) > record$. Novamente o teste é facilitado porque os valores de c estão ordenados em ordem não-decrescente, quando seguimos a sequência ρ.

Se interrompemos o processo de Bonecas Russas na iteração l, não temos disponível $c(v)$, para $v \in \{\rho_{l+1}, \ldots, \rho_n\}$. Mesmo assim, podemos aplicar a estratégia de obtenção dos *nogoods* descrita acima, se atribuirmos a $c(v)$ um limite superior para a clique ponderada máxima em $G[\rho_1, \ldots, \rho_i = v]$ para $v \in \{\rho_{l+1}, \ldots, \rho_n\}$.

Usando essa ideia, apresentamos o Algoritmo 6.15 que propõe uma cooperação entre Bonecas Russas e Busca por Resolução. Inicialmente, aplicamos o algoritmo de Bonecas Russas, seguindo a ordem ρ, até um vértice v_l ($l = limite$). Com isso calculamos exatamente $c(v_1) \leq \ldots \leq c(v_l)$. Os demais valores $c(v_{l+1}) \leq \ldots \leq c(v_n)$ são calculados de forma aproximada (limite superior). Nos testes realizados, utilizamos *limite* como $0.90{*}n$. Acreditamos que, a partir de $0.90 * n$, o custo para calcular $c(v)$ não é proporcional ao seu poder de poda no algoritmo de Bonecas Russas.

Também com o algoritmo de Bonecas Russas, geramos o *nogood* inicial $S = \overline{x_{\rho_{l+1}} x_{\rho_{l+2}}} \ldots \overline{x_{\rho_n}}$. Para potencializar o processo de poda, calculamos também um *novo_record*, usando a heurística tabu do nosso algoritmo de Branch & Bound.

Após esses passos, aplicamos o Algoritmo de Busca por Resolução, que faz uso de um obstáculo híbrido a ser descrito na Subseção 6.4.2. Ele aproveita tanto a informação gerada pelas bonecas quanto da coloração dos vértices candidatos para gerar os *nogoods*.

Algorithm 6.15 Cooperação entre o algoritmo de Bonecas Russas e o Algoritmo de Busca por Resolução

function MAIN

▷ Algoritmo de Bonecas Russas

$\rho = (\rho_1, \ldots, \rho_n)$
$record \leftarrow c(v_1)$
$c(\rho_1) \leftarrow record$
$U \leftarrow \{\rho_1\}$
for $i \leftarrow 2$ até $limite$ **do**
    $RDS(\{\rho_i\}, U \cap N(\rho_i), record)$
    $c[\rho_i] \leftarrow record$
    $U \leftarrow U \cup \{\rho_i\}$
for $i \leftarrow limite + 1$ até n **do**
    $j \leftarrow \max\{k \mid v_k \in N(v_i)\}$
    $c[v_i] \leftarrow \max\{c[v_{i-1}, c[v_j] + w(v_i)]\}$

▷ Algoritmo de Busca por Resolução

$novo\_record \leftarrow$ heurística tabu com multivizinhaça.
$record \leftarrow max(record, novo\_record)$
for $i \leftarrow 1$ até $limite$ **do**
    $S_i \leftarrow *$
for $i \leftarrow limite + 1$ até n **do**
    $S_i \leftarrow 0$
$\mathcal{F} \leftarrow \emptyset$
$W \leftarrow \emptyset$
AtualizaçãoFamília$(S, \mathcal{F}, W)$.
while $X(\mathcal{F}) \neq \{0,1\}^n$ **do**
    Encontre $U_{\mathcal{F}}$
    `obstáculo_híbrido`$(U_{\mathcal{F}}, record, S)$.
    AtualizaçãoFamília$(S, \mathcal{F}, W)$.

6.4.1 Oráculo

No oráculo proposto na Subseção 6.3.1 (Algoritmo 6.11), usamos a coloração ponderada para fornecer um limite superior. Agora, porém, temos um procedimento de limite superior com um custo computacional bem menor, dado pelo vetor de soluções ótimas das bonecas. Definimos então uma nova função `oráculo`, tirando proveito desse limite (Ver Algoritmo 6.16). A única mudança em relação ao anterior é que o limite superior para a clique ponderada máxima de P será dada pela tabela construída pelo Algoritmo de Bonecas Russas .

Algorithm 6.16 `oráculo`(U)

1: $\boxed{C \leftarrow \emptyset}$
2: $\boxed{P \leftarrow V}$
3: **for** $j = 1$ até n **do**
4: **if** $u_j = 1$ **then**
5: **if** $j \in P$ **then**
6: $\boxed{P \leftarrow P \cap N(j)}$;
7: $\boxed{C \leftarrow C \cup \{j\}}$;
8: **else**
9: **return** $-\infty$ ▷ Inviabilidade Encontrada
10: **else if** $u_j = 0$ **then**
11: $\boxed{P \leftarrow P \setminus \{j\}}$;
12: $j \leftarrow \max\{k \text{ — } \rho_k \in P\}$
13: **return** $w(C) + c(\rho_j)$

6.4.2 Obstáculo Híbrido

No obstáculo híbrido, vamos gerar dois tipos de *nogoods*. O primeiro tipo de *nogood* é obtido utilizando como limite superior o vetor gerado pelo Algoritmo de Bonecas Russas. Analogamente ao `obstáculo_modificado`, o segundo tipo de *nogood* é obtido através da heurística de coloração.

O obstáculo híbrido começa chamando o novo oráculo, que constrói os conjuntos C e P e calcula um limite superior, que será armazenado em *bound*. Se *bound* for maior que *record*, geramos o nogood U_1, do primeiro tipo. O conjunto R mantém os vértices de P ainda não analisados. Iterativamente, determinamos o limite superior para a clique máxima ponderada de $G[C \cup R]$. Esse limite é dado pela soma de $w(C)$ com $c[\max\{\rho_j \mid \rho_j \in R\}]$. Caso seja superior a bound, fixamos a variável associada ao último vértice de R ($\max\{\rho_j \mid \rho_j \in R\}$) em zero em U_1 e removemos ele de R. No final do laço, U_1 é um nogood com respeito aos limites dados pelo vetor c.

Na segunda parte do obstáculo híbrido, procedemos de maneira semelhante, utilizando agora a heurística de coloração ponderada. Aplicamos a heurística de coloração ponderada, que devolve uma ordem de coloração π, juntamente com um vetor de limites superiores *color*. Agora o conjunto T faz o papel do conjunto R do primeiro tipo de *nogood*. Enquanto o limite superior para $\omega(G[T], w)$ for menor que $record - w(C)$ e $T \neq \emptyset$, escolhemos o último vértice $j \in T$ colorido pela heurística de coloração, fixamos a variável associada ao vértice j em zero em U_2 e removemos o vértice j de T. Novamente, ao final do laço, U_2 é um nogood, agora com respeito à coloração ponderada.

O nogood retornado pelo obstáculo híbrido será aquele com o menor número de variáveis fixadas. Heuristicamente, o nogood com o menor número de fixações explora uma região maior do espaço de busca ainda não explorado.

Algorithm 6.17 `obstáculo_hibrido`$(U, record, S)$

```
1:  bound ← oraculo(U)
2:  if w(C) > record then
3:      record ← w(C)
4:  if bound > record then
5:                                                  ▷ Nogood do tipo 1
6:      R ← P
7:      U_1 ← U
8:      while R ≠ ∅ e bound > record do
9:          j ← max{k — v_k ∈ R}
10:         bound ← w(C) + c(v_j)
11:         if bound > record then
12:             (u_1)_j ← 0
13:             R ← R \ {j}
14:     bound ← BITCOLOR(P, ρ, π, color, cont)
15:     T ← P
16:     U_2 ← U
17:     for i ← cont até 1 do
18:         j ← π[i]
19:         if w(C) + color[j] ≤ record then
20:             pare
21:         (u_2)_j ← 0
22:     if |U_1| < |U_2| then
23:         S ← U_1
24:     else
25:         S ← U_2
26: else
27:     S ← U
```

6.5 Resultados Computacionais

Nesta subseção, analisamos o desempenho computacional do nosso algoritmo híbrido de Busca por Resolução com Bonecas Russas. Novamente, temos duas versões, definidas pelas ordens iniciais dos vértices. O algoritmo que usa a ordem **menor peso primeiro** será denotado por **BR1**, enquanto o que usa a ordem inicial **maior grau ponderado primeiro** será denotado por **BR2**.

Todos os testes foram realizados num computador equipado com processador *Intel Core i7-2600K 3.40Ghz*, *8Mb de cache*, *6Gb de memória*, utilizando sistema operacional *Linux*.

Similarmente aos experimentos anteriores, avaliamos o desempenho do algoritmo híbrido utilizando as principais instâncias disponíveis na literatura.

6.5.1 Grafos Aleatórios

No nosso experimento, nós geramos 10 grafos aleatórios para cada combinação n (número de vértices) e p (probabilidade de existência de cada aresta). Os pesos atribuídos aos vértices variam entre 1 e 10. Como é usual na literatura, o número de vértices das instâncias diminui à medida que a probabilidade de existência de aresta aumenta. Consideramos 20 combinações, levando a 200 instâncias no total. Para cada par (n, p), calculamos a média do tempo consumido pelas 10 instâncias para cada algoritmo.

O tempo de limite de resolução para cada instância é 1h (3600s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada na tabela como $*^x$, onde $x \in [1, 10]$ significa o número de instâncias não resolvidas no tempo limite . Identificamos em negrito o melhor desempenho médio em cada caso.

Inicialmente, analisamos o comportamento dos 6 algoritmos propostos até agora, com relação ao número médio de colorações realizadas por cada um. A coloração ponderada é o procedimento com o maior custo computacional realizado por todos os algoritmos. No caso do algoritmo híbrido, contamos o número de colorações ponderadas realizadas na fase de Bonecas Russas e da Busca por Resolução. A Tabela 28 mostra que, em praticamente todos os grupos, **BR1** e **BR2** reduzem o número de colorações ponderadas realizadas com relação à **RDS1** e **RDS2**, respectivamente. Além disso, **BR1** consegue executar o menor número de colorações que todos os outros algoritmos em muitas instâncias com densidade entre 30% e 80%. A redução em **BR1** chega a ser 1 ordem de magnitude em relação **BITCLIQUER1** para o grupo (400,0.8).

Já a Tabela 29 apresenta o tempo de execução médio de cada algoritmo. Apesar de **BR1** conseguir uma redução do número de colorações ponderadas com relação aos demais algoritmos em várias instâncias, essa redução não foi acompanhada por uma melhoria no tempo de execução. Mesmo assim, **BR1** mostrou-se bastante competitivo com os demais algoritmos usando a mesma ordem inicial dos vértices. Além disso, dentro

do tempo limite, **BR1** resolveu uma instância do grupo $(300, 0.90)$ que não foi resolvida por **RDS1**. O mesmo não aconteceu com **BR2**, que não conseguiu um tempo próximo ao melhor algoritmo que usa a mesma ordem. Na maioria dos casos, o tempo de execução de **BR2** chega a ser perto do dobro de **BITCLIQUE2**

Tabela 28 – Número médio de colorações ponderadas realizadas para cada grupo (n, p).

Instância	**Colorações Ponderada**					
(n,p)	**BITCLIQUE1**	**BITCLIQUE2**	**RDS1**	**RDS2**	**BR1**	**BR2**
(2500,0.10)	**1.82e+04**	1.97e+04	4.75e+04	2.88e+04	4.20e+04	2.81e+04
(5000,0.10)	**1.79e+05**	3.16e+05	2.67e+05	3.91e+05	2.46e+05	3.86e+05
(2500,0.20)	**1.95e+05**	2.91e+05	2.83e+05	3.42e+05	2.43e+05	3.48e+05
(5000,0.20)	**3.82e+06**	5.96e+06	5.63e+06	6.98e+06	4.75e+06	7.63e+06
(2500,0.30)	3.27e+06	6.99e+06	3.46e+06	9.45e+06	**2.87e+06**	8.99e+06
(1200,0.40)	**1.00e+06**	1.90e+06	1.24e+06	2.50e+06	1.02e+06	2.39e+06
(2500,0.40)	6.01e+07	1.34e+08	6.32e+07	1.75e+08	**4.88e+07**	1.83e+08
(600,0.50)	**2.65e+05**	4.07e+05	3.83e+05	5.40e+05	3.03e+05	5.19e+05
(1200,0.50)	1.61e+07	3.49e+07	1.58e+07	4.78e+07	**1.24e+07**	4.46e+07
(600,0.60)	2.90e+06	4.83e+06	3.51e+06	6.52e+06	**2.58e+06**	6.17e+06
(1200,0.60)	5.01e+08	$*^{10}$	3.97e+08	$*^{10}$	**3.01e+08**	$*^{10}$
(400,0.70)	2.63e+06	3.35e+06	3.20e+06	4.91e+06	**2.23e+06**	4.53e+06
(500,0.70)	1.79e+07	2.84e+07	1.91e+07	3.97e+07	**1.34e+07**	3.69e+07
(600,0.70)	7.00e+07	1.28e+08	7.31e+07	1.82e+08	**5.25e+07**	1.66e+08
(300,0.80)	6.79e+06	**5.95e+06**	9.59e+06	8.69e+06	5.96e+06	8.25e+06
(400,0.80)	1.09e+08	1.22e+08	1.14e+08	1.88e+08	**7.07e+07**	1.73e+08
(200,0.90)	2.72e+06	**3.40e+05**	5.64e+06	5.78e+05	3.17e+06	5.36e+05
(300,0.90)	7.26e+08	**1.53e+08**	$*^{1}$	2.70e+08	5.03e+08	2.52e+08
(200,0.95)	1.08e+07	**1.16e+05**	2.98e+07	2.15e+05	1.27e+07	1.80e+05

Fonte: Elaborada pelo autor.

Tabela 29 – Tempo de execução médio para cada grupo (n, p).

Instância	**Tempo de Execução**					
(n,p)	**BITCLIQUE1**	**BITCLIQUE2**	**RDS1**	**RDS2**	**BR1**	**BR2**
(2500,0.10)	0.77	0.81	0.21	**0.18**	0.85	0.84
(5000,0.10)	3.25	4.80	**1.89**	3.18	3.35	5.52
(2500,0.20)	2.00	3.07	**1.12**	2.30	1.89	3.63
(5000,0.20)	42.11	77.35	**36.30**	67.07	46.44	123.85
(2500,0.30)	23.46	50.48	**15.57**	51.77	16.88	73.10
(1200,0.40)	4.94	9.22	**3.39**	9.56	3.80	12.87
(2500,0.40)	464.34	1094.99	**311.57**	1090.54	336.76	1926.47
(600,0.50)	0.91	1.39	**0.71**	1.37	0.86	1.91
(1200,0.50)	79.12	171.66	**49.72**	187.34	54.19	258.82
(600,0.60)	9.54	16.46	**7.35**	17.66	7.82	24.72
(1200,0.60)	2730.49	> 3600	**1456.73**	> 3600	1645.06	> 3600
(400,0.70)	6.75	9.52	**5.58**	10.91	5.77	15.04
(500,0.70)	53.97	89.38	**39.74**	101.17	41.94	142.79
(600,0.70)	254.26	484.17	**173.01**	549.77	175.52	792.54
(300,0.80)	15.42	14.93	15.17	17.95	**14.40**	27.26
(400,0.80)	313.46	380.15	**234.03**	467.94	237.37	703.49
(200,0.90)	5.46	**1.12**	7.97	1.16	6.75	1.77
(300,0.90)	1924.74	**480.46**	> 3600	704.62	1629.50	1173.66
(200,0.95)	23.75	0.61	49.46	**0.51**	35.75	0.79

Fonte: Elaborada pelo autor.

6.5.2 DIMACS-W

Os resultados dos experimentos computacionais com as instâncias **DIMACS-W** podem ser vistos na Tabelas 30 e 31 . Agora, o tempo limite de resolução para cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Identificamos em negrito o melhor desempenho médio em cada caso.

A Tabela 30 revela que **BR1** conseguiu reduzir o número de colorações ponderadas realizadas em algumas instâncias da família *brock*. Nas outras instâncias com densidade até 80%, o algoritmo **BR1** conseguiu um número de colorações bastante próximo. Porém, **BR2** não conseguiu superar os demais algoritmos em nenhuma classe de instâncias.

Já a Tabela 31 confirma que a redução do número de colorações promovida por **BR1** implicou em uma redução do tempo de execução. Mostrou também que o algoritmo **BR1** é bastante competitivo com os demais algoritmos que usam a mesma ordem inicial dos vértices. Já **BR2** teve um comportamento muito parecido com **RDS2**, que por sua vez perdem para **BITCLIQUE2**.

Tabela 30 – Número médio de colorações ponderadas realizadas pelo algoritmo nas instâncias DIMACS-W.

Instância		**Colorações Ponderada**					
Nome	**(n,p)**	**BITCLIQUER1**	**BITCLIQUER2**	**RDS1**	**RDS2**	**BR1**	**BR2**
sanr400_0.5	(400,0.50)	**2.39e+04**	3.08e+04	5.13e+04	4.14e+04	4.06e+04	4.14e+04
san1000	(1000,0.50)	2.64e+04	**2.21e+04**	1.04e+05	4.53e+04	8.83e+04	5.75e+04
brock400_1	(400,0.75)	8.65e+06	1.10e+07	1.14e+07	1.99e+07	**7.45e+06**	1.64e+07
brock400_2	(400,0.75)	1.02e+07	1.55e+07	1.68e+07	2.73e+07	**8.64e+06**	2.27e+07
brock400_3	(400,0.75)	7.81e+06	1.15e+07	1.11e+07	1.55e+07	**7.35e+06**	1.57e+07
brock400_4	(400,0.75)	**3.53e+06**	5.20e+06	1.35e+07	3.27e+07	9.54e+06	2.63e+07
brock800_1	(800,0.65)	5.79e+07	1.47e+08	9.78e+07	2.65e+08	**5.10e+07**	2.23e+08
brock800_2	(800,0.65)	9.89e+07	2.51e+08	9.99e+07	3.50e+08	**7.12e+07**	3.30e+08
brock800_3	(800,0.65)	7.31e+07	1.79e+08	1.04e+08	2.73e+08	**6.12e+07**	2.36e+08
brock800_4	(800,0.65)	1.07e+08	2.75e+08	1.18e+08	3.94e+08	**8.33e+07**	3.63e+08
sanr400_0.7	(400,0.70)	**1.68e+06**	2.39e+06	2.81e+06	4.00e+06	2.06e+06	3.52e+06
san400_0.7_1	(400,0.70)	**2.32e+05**	5.71e+05	3.85e+05	7.07e+05	2.94e+05	1.05e+06
san400_0.7_2	(400,0.70)	**3.61e+05**	3.99e+05	1.04e+06	2.30e+06	9.01e+05	2.11e+06
san400_0.7_3	(400,0.70)	2.72e+05	**6.25e+04**	1.19e+06	1.41e+05	7.56e+05	1.32e+05
p_hat500-2	(500,0.50)	6.70e+04	**4.12e+03**	4.40e+05	7.59e+03	3.15e+05	7.57e+03
p_hat700-2	(700,0.50)	2.36e+06	**1.57e+04**	7.07e+06	2.36e+04	2.92e+06	2.36e+04
p_hat1000-2	(1000,0.49)	6.60e+07	**6.48e+05**	1.17e+08	9.19e+05	4.12e+07	9.17e+05
p_hat1500-2	(1500,0.51)	*	**2.59e+07**	*	3.75e+07	*	3.74e+07
p_hat300-3	(300,0.74)	1.47e+05	**1.50e+04**	6.04e+05	2.33e+04	4.13e+05	2.33e+04
p_hat500-3	(500,0.75)	5.12e+07	**6.97e+05**	1.03e+08	1.11e+06	5.76e+07	1.10e+06
gen200_p0.9_44	(200,0.90)	1.09e+06	**6.20e+04**	2.40e+06	1.41e+05	1.54e+06	1.35e+05
gen200_p0.9_55	(200,0.90)	3.13e+05	**1.71e+04**	5.04e+06	1.01e+05	1.62e+06	6.94e+04
gen400_p0.9_55	(400,0.90)	*	**5.73e+08**	*	*	*	*
gen400_p0.9_65	(400,0.90)	*	*	*	*	*	*
gen400_p0.9_75	(400,0.90)	*	**4.96e+07**	*	7.89e+08	*	7.09e+08
C250.9	(250,0.90)	6.19e+06	**9.62e+05**	2.37e+07	2.55e+06	8.51e+06	2.14e+06
san200_0.9_1	(200,0.90)	1.45e+04	**1.20e+02**	4.67e+05	7.29e+03	2.90e+05	6.92e+03
san200_0.9_2	(200,0.90)	6.76e+04	**2.85e+04**	5.65e+05	3.87e+06	5.31e+05	1.83e+06
san200_0.9_3	(200,0.90)	3.14e+06	**4.15e+05**	6.31e+06	7.23e+05	4.69e+06	7.90e+05
san400_0.9_1	(400,0.90)	**9.79e+06**	*	1.05e+08	5.79e+08	1.03e+08	*
sanr200_0.9	(200,0.90)	1.83e+06	**2.27e+05**	7.15e+06	3.63e+05	3.01e+06	3.50e+05
hamming10-2	(1024,0.99)	**1.80e+01**	*	3.11e+05	*	8.77e+05	*
MANN_a27	(378,0.99)	*	**1.82e+04**	*	7.17e+04	*	6.64e+04

Fonte: Elaborada pelo autor.

Tabela 31 – Tempo de execução dos algoritmos nas instâncias DIMACS-W.

Instância		Tempo de Execução					
Nome	**(n,p)**	**BITCLIQUER1**	**BITCLIQUER2**	**RDS1**	**RDS2**	**BR1**	**BR2**
sanr400_0.5	(400,0.50)	0.17	0.18	**0.12**	0.13	0.19	0.22
san1000	(1000,0.50)	**2.21**	2.57	5.43	2.50	5.18	4.08
brock400_1	(400,0.75)	29.72	38.73	29.45	57.00	**26.09**	64.43
brock400_2	(400,0.75)	39.61	58.49	48.49	79.25	**35.20**	87.13
brock400_3	(400,0.75)	30.53	43.56	30.97	49.89	**25.70**	77.78
brock400_4	(400,0.75)	**16.29**	23.02	36.38	91.16	33.07	91.82
brock800_1	(800,0.65)	400.76	946.47	428.67	1268.82	**253.16**	1209.56
brock800_2	(800,0.65)	640.83	1458.54	434.19	1656.72	**410.82**	2237.91
brock800_3	(800,0.65)	470.91	1064.17	445.39	1275.23	**320.50**	1436.51
brock800_4	(800,0.65)	644.40	1539.57	509.33	1759.07	**461.69**	2319.97
sanr400_0.7	(400,0.70)	6.26	8.90	6.56	11.68	**5.98**	12.46
san400_0.7_1	(400,0.70)	2.36	5.19	2.54	4.09	**2.13**	8.23
san400_0.7_2	(400,0.70)	**3.35**	4.73	6.65	16.07	6.22	19.11
san400_0.7_3	(400,0.70)	2.34	**0.86**	6.66	1.18	4.24	1.30
p_hat500-2	(500,0.50)	0.59	0.10	1.78	**0.03**	1.52	0.08
p_hat700-2	(700,0.50)	19.67	0.29	45.71	**0.13**	19.62	0.20
p_hat1000-2	(1000,0.49)	750.75	10.09	1055.14	5.70	361.84	**5.51**
p_hat1500-2	(1500,0.51)	> 3600	624.42	> 3600	**329.62**	> 3600	335.99
p_hat300-3	(300,0.74)	0.66	0.11	1.45	**0.08**	1.12	0.12
p_hat500-3	(500,0.75)	328.54	**5.17**	427.52	5.58	241.42	5.91
gen200_p0.9_44	(200,0.90)	3.24	0.52	5.05	**0.38**	4.87	0.50
gen200_p0.9_55	(200,0.90)	1.26	0.37	9.38	0.28	3.50	**0.22**
gen400_p0.9_55	(400,0.90)	> 3600	**3595.54**	> 3600	> 3600	> 3600	> 3600
gen400_p0.9_65	(400,0.90)	> 3600	> 3600	> 3600	> 3600	> 3600	> 3600
gen400_p0.9_75	(400,0.90)	> 3600	**363.60**	> 3600	3384.94	> 3600	3061.18
C250.9	(250,0.90)	21.52	**4.20**	59.04	8.01	28.22	8.30
san200_0.9_1	(200,0.90)	0.58	0.50	0.98	**0.01**	0.73	0.06
san200_0.9_2	(200,0.90)	0.55	**0.49**	1.05	8.09	1.20	3.88
san200_0.9_3	(200,0.90)	8.25	**1.64**	11.93	1.93	12.89	3.76
san400_0.9_1	(400,0.90)	**78.08**	> 3600	459.00	2534.88	485.89	> 3600
sanr200_0.9	(200,0.90)	4.69	1.08	13.04	**0.91**	8.27	1.35
hamming10-2	(1024,0.99)	**0.66**	> 3600	1.67	> 3600	60.89	> 3600
MANN_a27	(378,0.99)	> 3600	1.28	> 3600	**0.30**	> 3600	0.53

Fonte: Elaborada pelo autor.

6.5.3 EXACTCOLOR

Para as instâncias EXACTCOLOR, o tempo limite para a resolução de cada instância é 2h (7200s). Quando o tempo limite é ultrapassado, tal ocorrência é assinalada como * na tabela. Identificamos em negrito o melhor desempenho em cada caso.

Com relação ao número de colorações realizadas, a Tabela 32 mostra que **BR1** é bastante competitivo com os demais algoritmos para instâncias até 80%. Além disso, **BR1** reduz o número de colorações realizadas por **RDS1** na maioria das instâncias. Em algumas instâncias, **BR1** obtém o menor número de colorações ponderada dentre todas. Já **BR2** não superou nenhum algoritmo com respeito ao número de colorações em nenhuma instância.

A Tabela 33 apresenta o tempo de execução de cada algoritmo proposto. Mesmo com o número de colorações comparáveis, **BR1** e **BR2** tiveram um tempo de execução bastante distante dos algoritmos **RDS1** e **RDS2**, indicando que o algoritmo de Busca por Resolução demandou um esforço computacional considerável para estas instâncias, que talvez possa ser reduzido pela combinação de outras estratégias de mer-

gulho e recomeço.

Tabela 32 – Número de colorações ponderadas para cada instância EXACTCOLOR

Instância	**(n,p)**	**Colorações Ponderadas**					
		BITCLIQUER1	**BITCLIQUER2**	**RDS1**	**RDS2**	**BR1**	**BR2**
latin_square_10	(90,0.00)	1.00e+00	1.00e+00	**0.00e+00**	**0.00e+00**	1.00e+00	1.00e+00
r1000.5	(234,0.00)	1.00e+00	1.00e+00	**0.00e+00**	**0.00e+00**	1.00e+00	1.00e+00
DSJR500.1c	(189,0.02)	4.60e+01	**4.10e+01**	1.31e+02	1.77e+02	1.32e+02	1.78e+02
r1000.1c	(511,0.03)	3.76e+02	**3.56e+02**	7.33e+02	6.29e+02	7.30e+02	6.30e+02
DSJC250.9	(250,0.10)	4.07e+02	**4.04e+02**	4.87e+02	4.49e+02	4.88e+02	4.50e+02
DSJC500.9	(500,0.10)	1.96e+03	**1.95e+03**	2.06e+03	2.00e+03	2.06e+03	2.00e+03
DSJC1000.9	(1000,0.10)	**1.28e+04**	1.42e+04	1.31e+04	1.44e+04	1.31e+04	1.44e+04
DSJC250.5	(250,0.50)	**5.18e+04**	5.23e+04	5.65e+04	5.29e+04	5.60e+04	5.29e+04
DSJC500.5	(500,0.50)	**2.22e+06**	2.26e+06	2.24e+06	2.28e+06	2.23e+06	2.28e+06
DSJC1000.5	(1000,0.50)	1.80e+08	1.83e+08	1.78e+08	1.83e+08	**1.77e+08**	1.83e+08
C2000.5.1029	(2000,0.50)	*	*	*	*	*	*
flat300_28_0	(300,0.52)	**1.91e+05**	2.01e+05	1.98e+05	2.05e+05	1.97e+05	2.05e+05
flat1000_50_0	(967,0.51)	9.68e+06	2.00e+07	7.72e+06	2.28e+07	**7.31e+06**	2.25e+07
flat1000_60_0	(999,0.51)	4.50e+07	6.55e+07	3.45e+07	7.08e+07	**3.25e+07**	7.07e+07
flat1000_76_0	(1000,0.51)	2.30e+08	2.30e+08	2.25e+08	2.30e+08	**2.24e+08**	2.30e+08
school1	(336,0.71)	3.05e+05	**1.14e+05**	4.03e+05	1.28e+05	3.73e+05	1.68e+05
DSJC250.1	(241,0.89)	3.34e+08	**2.75e+08**	5.77e+08	3.85e+08	4.60e+08	3.98e+08
DSJC500.1.117	(494,0.90)	*	*	*	*	*	*
DSJC1000.1.3915	(998,0.90)	*	*	*	*	*	*
2-Insertions_4	(149,0.95)	**1.53e+04**	6.86e+04	2.14e+04	7.16e+04	2.10e+04	7.15e+04
1-Insertions_6	(527,0.96)	**1.17e+05**	2.06e+06	2.23e+07	4.60e+06	1.95e+07	4.15e+06
2-Insertions_5	(369,0.97)	9.38e+05	9.78e+05	1.46e+07	**7.12e+05**	1.02e+07	7.18e+05
3-Insertions_4	(208,0.97)	2.99e+05	**7.68e+03**	2.34e+06	1.85e+04	1.54e+06	1.62e+04
4-Insertions_4	(295,0.98)	**6.83e+05**	1.38e+07	1.10e+07	5.18e+06	8.16e+06	5.20e+06
3-Insertions_5	(460,0.98)	2.18e+08	*	*	**3.93e+07**	*	3.97e+07

Fonte: Elaborada pelo autor.

Tabela 33 – Tempo de execução dos algoritmos para cada instância EXACTCOLOR.

Instancia	(n,p)	**Tempo de Execução**					
		BITCLIQUER1	**BITCLIQUER2**	**RDS1**	**RDS2**	**BR1**	**BR2**
latin_square_10	(90,0.00)	0.08	0.09	**0.00**	**0.00**	0.09	0.08
r1000.5	(234,0.00)	0.18	0.18	**0.00**	**0.00**	0.19	0.19
DSJR500.1c	(189,0.02)	0.16	0.15	**0.00**	**0.00**	0.15	0.16
r1000.1c	(511,0.03)	0.16	0.16	**0.00**	**0.00**	0.16	0.25
DSJC250.9	(250,0.10)	0.07	0.07	**0.00**	**0.00**	0.07	0.07
DSJC500.9	(500,0.10)	0.22	0.15	**0.01**	**0.01**	0.20	0.14
DSJC1000.9	(1000,0.10)	0.29	0.28	**0.03**	**0.03**	0.30	0.29
DSJC250.5	(250,0.50)	0.10	0.10	**0.05**	**0.05**	0.14	0.14
DSJC500.5	(500,0.50)	**3.26**	3.30	3.33	3.27	6.02	5.64
DSJC1000.5	(1000,0.50)	449.52	458.15	423.38	**422.89**	873.09	853.38
C2000.5.1029	(2000,0.50)	*	*	*	*	*	*
flat300_28_0	(300,0.52)	0.30	0.30	**0.21**	**0.21**	0.46	0.54
flat1000_50_0	(967,0.51)	58.27	103.87	**36.44**	103.99	55.08	153.28
flat1000_60_0	(999,0.51)	194.60	252.23	**130.11**	245.43	219.94	410.02
flat1000_76_0	(1000,0.51)	586.53	572.81	534.42	**530.53**	1132.91	1095.70
school1	(336,0.71)	2.45	0.67	3.21	**0.62**	3.86	1.42
DSJC250.1	(241,0.89)	717.33	**592.85**	1059.60	753.20	1522.98	1750.34
DSJC500.1.117	(494,0.90)	*	*	*	*	*	*
DSJC1000.1.3915	(998,0.90)	*	*	*	*	*	*
2-Insertions_4	(149,0.95)	0.20	0.30	**0.03**	0.11	0.10	0.38
1-Insertions_6	(527,0.96)	**1.78**	24.36	156.03	50.89	136.31	71.81
2-Insertions_5	(369,0.97)	7.03	6.53	74.19	**4.32**	66.45	8.10
3-Insertions_4	(208,0.97)	1.09	0.26	4.85	**0.04**	7.01	0.09
4-Insertions_4	(295,0.98)	**4.16**	67.19	42.54	25.57	41.38	56.44
3-Insertions_5	(460,0.98)	1887.79	*	*	**404.22**	*	693.26

Fonte: Elaborada pelo autor.

6.6 Conclusão

Neste capítulo, apresentamos o método de Busca por Resolução proposto em Chvátal (1997). Depois, mostramos como usar a heurística **BITCOLOR** no procedimento `obstáculo` para obter um *nogood*. Em seguida, realizamos alguns testes computacionais para justificar a nossa escolha para a política de recomeço. O algoritmo obtido realizou menos chamadas ao `oráculo` que o Algoritmo **BITCLIQUE**, que utiliza a mesma ordem; Todavia essa redução não resultou em uma diminuição no tempo de execução, sugerindo que a "gerência" do método demandou grande esforço computacional. Com a perspectiva de melhorarmos o desempenho, pensamos em uma cooperação entre o algoritmo de Bonecas Russas e Busca por Resolução. Essa cooperação mostrou-se bem sucedida em reduzir o número de chamadas à heurística de coloração ponderada em comparação ao algoritmo de Bonecas Russas. Porém, o tempo de execução não acompanhou essa redução, indicando que o algoritmo de Busca por Resolução demandou um esforço computacional considerável para estas instâncias, que talvez possa ser reduzido pela combinação de outras estratégias de mergulho e recomeço.

7 CONCLUSÃO E DIREÇÕES FUTURAS

A proposição de algoritmos cada vez mais eficientes para CLIQUE PONDERADA é um imperativo dada a sua importância e recorrência. Um exemplo disso acontece no processo de geração de colunas para a coloração de grafos. Neste caso, métodos exatos e heurísticas são utilizados para a resolução consecutiva de várias instâncias de CLIQUE PONDERADA Mehrotra and Trick (1996); Gualandi and Malucelli (2012); Held, Cook, and Sewell (2012). Note que pequenos ganhos em desempenho são potencializados dada a recorrência do problema.

A partir dessa motivação, apresentamos três algoritmos exatos para CLIQUE PONDERADA usando abordagens diferentes. O primeiro algoritmo, denominado **BITCLIQUE**, mostrou-se mais eficiente que os demais algoritmos do estado da arte. Seu melhor desempenho ocorre em grafos com a densidade superior ou igual 90%. Nesta faixa, a variante **BITCLIQUE2** obteve os melhores resultados.

O segundo algoritmo, denominado **BITRDS**, pode ser entendido como uma extensão do algoritmo **CLIQUER**, incorporando uma estratégia de poda e ramificação baseada na coloração ponderada. Nos testes computacionais realizados, **CLIQUER** confirmou a sua eficiência para grafos esparsos, com a densidade entre 10% e 40%. Por outro lado, para grafos com a densidade entre 50% e 80%, a variação **BITRDS1** apresentou o melhor desempenho computacional nas instâncias analisadas.

Por fim, apresentamos um algoritmo híbrido, denominado **BITBR**, que combina o método de Bonecas Russas e o método de Busca por Resolução. Essa integração é apresentada e testada pela primeira vez neste trabalho. O novo algoritmo consegue reduzir o número de chamadas à heurística de coloração ponderada, que é o procedimento mais dispendioso do processo. Porém, o tempo de execução não é reduzido. Acreditamos que o esforço computacional do método de Busca por Resolução possa ser decrescido com o desenvolvimento de novas estratégias de mergulho e recomeço mais apropriadas.

Os algoritmos apresentados nesta tese abre várias direções de pesquisas promissoras. Uma delas é a utilização dos algoritmos em problemas que exijam a resolução de várias instâncias de CLIQUE PONDERADA, como ocorre no processo de geração de colunas ou na geração de cortes de cliques para problemas de programação inteira.

Outro rumo é o desenvolvimento de algoritmos de Bonecas Russas para diversas variações do problema da clique McClosky and Hicks (2012); Trukhanov *et al.* (2013); Gschwind *et al.* (2015); Miao, Balasundaram, and Pasiliao (2014); Malladi (2014). Uma vez que vimos que o algoritmo de Bonecas Russas pode incorporar com sucesso novas estratégias de poda e ramificação.

Outro trabalho que deve ser feito é uma comparação de **BITRDS** com o algoritmo apresentado em Corrêa *et al.* (2014). Ambos integram a heurística de coloração ao método original, porém essa integração é realizada de maneira diferente.

Com relação ao algoritmo de Busca por Resolução, temos vários obstáculos pela frente, como:

- Desenvolvimento de políticas de mergulho e recomeço eficientes.
- Desenvolvimento de regras mais efetivas para a fase de decrescimento.
- Adaptação do método para novos problemas de programação inteira.

A –PROBLEMA DE ENUMERAÇÃO DE CLIQUES MAXIMAIS

Neste apêndice, discutimos o problema da enumeração de cliques com peso acima de um limiar. Em programação inteira, a enumeração de cliques maximais com certo peso está relacionada, por exemplo, com o problema de encontrar todas as desigualdades de clique violadas. O impacto destas desigualdades na resolução de vários problemas de programação inteira vem sendo explorado em diversos trabalhos Avella and VasilEv (2005); Burke *et al.* (2012); Santos *et al.* (2014); Brito, Santos, and Poggi (2015); Corrêa *et al.* (2015). Além disso, em muitas aplicações, é importante não apenas encontrar a clique com o peso máximo ou com a cardinalidade máxima, mas todas as cliques maximais ou ainda um subconjuntos delas.

Muitos algoritmos são conhecidos para o problema de enumeração de todas as cliques maximais Akkoyunlu (1973); Bron and Kerbosch (1973); Cazals and Karande (2008); Chiba and Nishizeki (1985); Gerhards and Lindenberg (1979); Harary and Ross (1957); Johnston (1976); Makino and Uno (2004); Tomita, Tanaka, and Takahashi (2006). Um dos mais bem sucedidos na prática é o algoritmo de *Bron-Kerbosch*Bron and Kerbosch (1973), que identificamos por **BK**.

O algoritmo **BK** foi adaptado para o problema da enumeração de cliques maximais acima de um limiar em Brito and Santos (2011); Boas, Santos, and Brito (2015) e aplicado com sucesso em problemas de programação inteira em Santos *et al.* (2014); Brito, Santos, and Poggi (2015).

Na Subseção A.1, apresentamos o algoritmo BK com uma regra de pivoteamento. Na Subseção A.2, apresentamos algumas adaptações do algoritmo BK para o problema de enumeração de cliques maximais com peso acima de um limiar. Na Subseção C, usamos esses algoritmos adaptados para resolver o problema CLIQUE PONDERADA e os comparamos computacionalmente com os outros algoritmos propostos ao longo deste trabalho.

A.1 Algoritmo de Bron-Kerbosch

BK é um algoritmo simples de *backtracking* que resolve recursivamente subproblemas definidos pela tripla (C, P, X), onde C é uma clique atual, P é o conjunto dos vértices que ainda podem entrar na clique atual, chamado de conjunto candidato e X é o conjunto dos vértices que já foram explorados em algum subproblema anterior e que poderiam entrar na clique não-maximal C. Em outras palavras, $P \cup X$ é uma partição da vizinhança comum dos vértices em C.

Bron e Kerbosch também introduziram uma variante do algoritmo que envolve a escolha de um vértice pivô u. Eles observaram que qualquer clique maximal deve incluir o vértice u ou um dos vértices não adjacentes a ele, pois, caso contrário, a clique não seria maximal. Nessa variação de **BK**, temos uma regra de pivoteamento responsável por

selecionar o vértice pivô.

Em Tomita, Tanaka, and Takahashi (2006), os autores sugerem escolher o pivô $u \in P \cup X$ a fim de maximizar $|P \cap N(u)|$, ou seja, o pivô é o vértice da vizinhança comum de C com o maior número de vizinhos no conjunto de candidatos. Com isso, garantem que o algoritmo de Bron-Kerbosch executa com uma complexidade de tempo no pior caso $O(3^{\frac{n}{3}})$. Apesar de sua complexidade exponencial, o algoritmo tem se mostrado extremamente rápido na prática.

O Algoritmo A.1 apresenta o algoritmo de Bron-Kerbosck com a regra de pivoteamento proposta em Tomita, Tanaka, and Takahashi (2006). O processo de enumeração de cliques maximais de um grafo G começa com a seguinte chamada $BK(\emptyset, V(G), \emptyset)$. Note que a regra de pivoteamento proposta em Tomita, Tanaka, and Takahashi (2006) acelera o processo de descoberta de cliques maximais com uma grande cardinalidade.

Algorithm A.1 Algoritmo BK com a regra de pivoteamento

1: **function** BK(C, P, X)
2: **if** $P = \emptyset$ e $X = \emptyset$ **then**
3: Adicione a clique C no conjunto solução
4: Escolha um vértice pivô $u \in P \cup X$ ▷ Em Tomita, Tanaka, and Takahashi (2006), os autores sugerem escolher o vértice $u \in (P \cup X)$ que maximiza $|P \cap N(u)|$
5: **for** $v \in P \setminus N(u)$ **do**
6: BK($C \cup \{v\}, P \cap N(v), X \cap N(v)$)
7: $P \leftarrow P \setminus \{v\}$
8: $X \leftarrow X \cup \{v\}$

A.2 Enumeração de cliques maximais com peso acima de um limiar

Em Brito and Santos (2011), uma versão modificada de **BK**, para a enumeração de cliques com peso acima de um limiar, é apresentada (Ver Algoritmo A.2). A regra de pivoteamento utilizada consiste em escolher um vértice $u \in P$ com o maior grau modificado (soma do grau do vértice u e dos graus dos seus vizinhos) e com o maior peso. Segundo os autores esta regra de pivoteamento acelera o processo de descoberta de cliques maximais pesadas. Além disso, o algoritmo utiliza uma regra de poda que usa como estimativa para a clique ponderada máxima de P a soma dos pesos dos vértices de P. Resultados computacionais mostraram que essas adaptações levam a um bom desempenho prático.

Algorithm A.2 Algoritmo BKA Brito and Santos (2011)

```
1: function BKA(C, P, X, limiar)
2:     if P = ∅ e X = ∅ then
3:         if w(C) ≥ limiar then
4:             Adicione a clique C ao conjunto solução
5:     if w(C) + w(P) ≥ limiar then
6:         Escolha um vértice pivô u ∈ P com o maior grau modificado.
7:         for v ∈ P \ N(u) do
8:             BKA(C ∪ {v}, P ∩ N(v), X ∩ N(v))
9:             P ← P \ {v}
10:            X ← X ∪ {v}
```

Em Boas, Santos, and Brito (2015), uma versão melhorada do algoritmo BKA é apresentada. Além disso, testes computacionais foram conduzidos utilizando 7292 instâncias oriundas de quatro conjuntos de instâncias de problemas de programação inteira. O primeiro conjunto, denominado MIPLIB, advém da biblioteca de instâncias MIPLIB2010Koch *et al.* (2011), contendo 87 instâncias. O segundo conjunto de instâncias, denominado INRC, foi obtido pela formulação usada em Santos *et al.* (2014), e contém 60 instâncias. O terceiro conjunto de instâncias, denominado TelebusBorndorfer *et al.* (1999), origina-se do problema de planejamento de rotas para o Telebus. O quarto conjunto de instâncias, denominado Uchoa, deriva de uma instância do problema de p-dispersão Franco and Uchoa (2015). A partir dos problemas de programação inteira são gerados grafos de conflito que definem as instâncias para o problema de enumeração de cliques.

Em geral, dado um problema de programação inteira, um grafo de conflito pode ser construído a partir da análise de suas restrições, usando técnicas de *probing*. O grafo de conflito tem um vértice i para cada variável x_i. Dois vértices i e j definem uma aresta se, por exemplo, as variáveis correspondentes não podem assumir valor não nulo simultaneamente. Logo, uma clique K no grafo de conflito define uma desigualdade válida

$$\sum_{i \in K} x_i \leq 1$$

Assim, as desigualdades de clique violadas por uma solução fracionária $\bar{x}$ são definidas pelas cliques com peso maior que 1 no grafo de conflito onde cada vértice i é ponderado com $\bar{x_i}$. De outra forma, interessa-nos enumerar as cliques com peso maior que *limiar*, no grafo de conflito onde cada vértice i tem peso $limiar * \bar{x_i}$, sendo *limiar* escolhido para tornar os pesos inteiros.

Na verdade, grafos de conflitos podem ser construídos de forma mais geral, seja a partir de restrições originais do problema ou gerados pelo método de plano de cortes.

Em Brito, Santos, and Poggi (2015), um procedimento de construção do grafo de conflitos é apresentado. Com esse procedimento, os autores geraram um total de 7292 instâncias, sendo 399 instâncias a partir do conjunto MIPLIB, 3804 de INRC, 3085 de

Telebus e 4 de Uchoa . A Tabela 34 mostra a densidade mínima, máxima e média dos grafos de conflitos. .

Tabela 34 – Densidade mínima, máxima e média das instâncias do problema de detecção de cliques

	MIPLIB	**INRC**	**Telebus**	**Uchoa**
Densidada Mínima	0.01	0.01	0.01	0.11
Densidade Máxima	0.84	0.07	0.37	0.21
Densidade Média	0.14	0.01	0.04	0.16

Fonte: Elaborada pelo autor.

A Tabela 35 mostra o tempo reportado em Boas, Santos, and Brito (2015) para enumeração de todas as cliques violadas para todas as instâncias de cada grupo.

Tabela 35 – Tempo para enumerar todas as cliques violadas para todas as instâncias de cada grupo Boas, Santos, and Brito (2015)

	MIPLIB	**INRC**	**Telebus**	**Uchoa**
Tempo de Execução em segundos	65.25	54.17	39.17	18.75

Fonte: Elaborada pelo autor.

Propomos aqui uma variante do Algoritmo de BK, denominada **BITBKC**, que utiliza a heurística de coloração ponderada como procedimento de limite superior e como estratégia de pivoteamento. Além disso, o conjunto P e X são representado por vetores de bits(*bitmap*). A ordem inicial ρ é a ordem não decrescente dos grau modificados. No algoritmo original BK, em cada chamada recursiva, a ordem em que os vértices de P são escolhidos para entrar na clique atual C não é especificada. Nessa versão, seguimos a ordem inicial dos vértices ρ. A estratégia de pivoteamento utilizada será escolher o último vértice colorido pela heurística de coloração como pivô (Ver Algoritmo A.3). As instruções em caixas são aquelas simplificadas pela utilização de vetores de bits.

Algorithm A.3 Algoritmo BITBKC

```
1: function MAIN
2:     ρ ← ordem ordem crescente dos graus modificados
3:     BITBKC(∅, V(G), ∅, limiar)
4: function BITBKC(C, P, X, limiar)
5:     if P = ∅ e X = ∅ then
6:         if w(C) ≥ limiar then
7:             Adicione a clique C ao conjunto solução
8:     BITCOLOR(P, ρ, π, color, cont)
9:     if w(C) + color[π[cont]] ≥ limiar then
10:        u ← π[cont]
11:        for v ∈ P \ N(u) do          ▷ ordem não decrescente de grau modificados
12:            BITBKC(C ∪ {v}, [P ∩ N(v)], [X ∩ N(v)] )
13:            [P ← P \ {v}]
14:            [X ← X ∪ {v}]
```

A Tabela 36 mostra uma comparação entre os tempos de execução do algoritmo **BKA** apresentado em Boas, Santos, and Brito (2015), gentilmente cedido pelos autores, e de **BITBKC**. O algoritmo **BITBKC** conseguiu uma boa redução no tempo para enumerar todas as cliques maximais violadas. Essa redução reveste-se de maior significado quando relembramos que esse algoritmo é utilizado com uma grande frequência na rotina de separação de cortes.

Tabela 36 – Tempo para enumerar todas as cliques violadas para todas as instâncias de cada grupo

	MIPLIB	**INRC**	**Telebus**	**Uchoa**
BKA	78.75	71.44	67.19	25.04
BITBKC	**7.89**	**60.64**	**34.63**	**3.82**

Fonte: Elaborada pelo autor.

Consideramos também uma segunda variante do algoritmo BK, denominada **BITBKA** (Ver Algoritmo A.4), que utiliza o mesmo critério de poda e a mesma estratégia de escolha do pivô que **BKA**, ou seja, a estimativa da clique ponderada máxima de P é dada por $w(P)$ e o pivô é o vértice com o maior grau modificado e maior peso. A diferença aqui é que utilizamos vetores de bits (bitmap) para representar os conjuntos P e X. O grau modificado é calculado, uma única vez, antes do procedimento **BITBKA**. O pivô é dado pelo último vértice do conjunto P. A ordem de expansão dos vértices P segue a ordem inicial. Novamente, as instruções em caixas são aquelas simplificadas pela utilização de vetores de bits.

Algorithm A.4 Algoritmo BITBKA

```
1: function MAIN
2:     ρ ← ordem não decrescente de grau modificados.
3:     BITBKA(∅, V(G), ∅, limiar)
4: function BITBKA(C, P, X, limiar)
5:     if P = ∅ e X = ∅ then
6:         if w(C) ≥ limiar then
7:             Adicione a clique C no conjunto solução
8:     if w(C) + w(P) ≥ limiar then
9:         Escolha o último vértice u de P
10:        for v ∈ P \ N(u) do
11:            BITBKA(C ∪ {v}, P ∩ N(v), X ∩ N(v))
12:            P ← P \ {v}
13:            X ← X ∪ {v}
```

A Tabela 37 mostra que tanto **BITBKC** quanto **BITBKA** reduziram o tempo para enumerar todas as cliques em relação ao algoritmo **BKA**. Além disso, **BITBKA** foi mais efetivo que **BITBKC** no grupo de instâncias INRC, Telebus e Uchoa, tendo acontecido a situação inversa nas instâncias **MIPLIB**. Isso indica que a estratégia de pivoteamento de **BITBKC** foi mais efetiva que a de **BITBKA** para instâncias MIPLIB.

Tabela 37 – Tempo para enumerar todas as cliques violadas para todas as instâncias de cada grupo

	MIPLIB	**INRC**	**Telebus**	**Uchoa**
BKA	78.75	71.44	67.19	25.04
BKC	7.89	60.64	34.63	3.82
BITBKA	17.56	**31.67**	**23.97**	**2.81**

Fonte: Elaborada pelo autor.

A.3 Adaptação para o problema da clique máxima ponderada

Apesar de o problema de enumeração de todas as cliques com peso a partir de um certo limiar e o problema da clique máxima ponderada serem problemas diferentes, podemos adaptar facilmente um algoritmo que resolve uma instância do primeiro para resolver uma do segundo. Basta modificar o valor do limiar toda vez que uma nova clique ponderada for encontrada (Linha 4 Algoritmo A.5). Outra modificação possível é enumerar apenas as cliques com limite superior estritamente acima do limiar estabelecido (Linha 5 Algoritmo A.5).

Algorithm A.5 Algoritmo BKA_{MAX}

```
1: function BKA_MAX (C, P, X, limiar)
2:     if P = ∅ e X = ∅ then
3:         if w(C) > limiar then
4:             limiar ← w(C)
5:     if w(C) + w(P) > limiar then
6:         Escolha um vértice pivô u de P
7:         for v ∈ P \ N(u) do
8:             BKA_MAX(C ∪ {v}, P ∩ N(v), X ∩ N(v))
9:             P ← P \ {v}
10:            X ← X ∪ {v}
```

Aqui, nós comparamos o desempenho computacional do nosso algoritmo **BITRDS1** com a modificação dos seguintes algoritmos para resolver CLIQUE PONDERADA :

- Algoritmo BKA apresentado em Boas, Santos, and Brito (2015), agora denominado $\mathbf{BKA}_{MAX}$.
- Algoritmo BITBKA(Ver Algoritmo A.4), agora denominado $\mathbf{BITBKA}_{MAX}$.

Novamente, utilizamos todas as 7292 instâncias para a realização dos testes computacionis. Agora, realizamos uma análise mais acurada do tempo de execução dos algoritmos. Separamos as instâncias em três faixas de densidades: $densidade < 0.1$, $0.1 \leq densidade \leq 0.5$ e $densidade > 0.5$.

A Tabela 38 apresenta os resultados das instâncias MIPLIB. $\mathbf{BITBKA}_{MAX}$ teve o melhor desempenho nas instâncias com baixa densidade e para as demais densidades, **BITCLIQUE1** foi superior ou teve um desempenho comparável com $\mathbf{BITKBA}_{MAX}$.

Note que todas as instâncias INRC estão na primeira faixa de densidade (Ver Tabela 39). Novamente, $\mathbf{BITBKA}_{MAX}$ superou os demais algoritmos.

Já as instâncias Telebus concentram-se apenas nas duas primeiras faixas(Ver Tabela 40). $\mathbf{BITBKA}_{MAX}$ encontrou a clique máxima mais rapidamente na primeira faixa de densidade. Na segunda faixa, podemos considerar que os três algoritmos tem um comportamento comparável, embora exista uma ligeira vantagem de $\mathbf{BKA}_{MAX}$ e $\mathbf{BITBKA}_{MAX}$.

No último conjunto, todas as instâncias estão na segunda faixa de densidade(Ver Tabela 41). O algoritmo **BITCLIQUE1** teve o melhor desempenho em três das quatro instâncias do conjunto.

Na próxima seção, comparamos o desempenho computacional do algoritmo **BITRDS1** e $\mathbf{BITBKA}_{MAX}$ em grafos gerados aleatoriamente.

Tabela 38 – Resultados computacionais para instâncias MIPLIB

	Número de Instâncias	**Média de número de vértices**	BKA_{MAX}	$BITBKA_{MAX}$	**BITRDS1**
$densidade < 0.1$	275	807.37	7.52	**2.30**	5.17
$densidade \leq 0.5$	79	2810.46	2.38	1.34	**0.56**
$densidade > 0.5$	45	75.64	0.19	**0.084**	0.13
Total			10,09	**3.72**	5.86

Fonte: Elaborada pelo autor.

Tabela 39 – Resultados computacionais para instâncias INRC

	Número de Instâncias	**Média de número de vértices**	BKA_{MAX}	$BITBKA_{MAX}$	**BITRDS1**
$densidade < 0.1$	3804	820.07	61.27	**27.66**	56.47

Fonte: Elaborada pelo autor.

Tabela 40 – Resultados computacionais para instâncias Telebus

	Número de Instâncias	**Média de número de vértices**	BKA_{MAX}	$BITBKA_{MAX}$	**BITRDS1**
$densidade < 0.1$	2821	686.93	64.14	**21.71**	31.82
$densidade \leq 0.5$	254	7656.38	**0.11**	**0.11**	0.19
Total			64.25	21.82	32.01

Fonte: Elaborada pelo autor.

Tabela 41 – Resultados computacionais para instâncias UCHOA

Instância	**Número de vértices**	**densidade**	BKA_{MAX}	$BITBKA_{MAX}$	**BITRDS1**
pdistuchoa_cut_1	500	0.211	0.73	0.08	**0.01**
pdistuchoa_cut_2	310	0.116	0.009	**0.002**	0.003
pdistuchoa_cut_3	371	0.145	0.030	0.015	**0.007**
pdistuchoa_cut_4	428	0.169	0.064	0.025	**0.010**
Total			0.83	0.12	**0.03**

Fonte: Elaborada pelo autor.

A.4 Testes com Grafos Aleatórios

Em nosso experimento, geramos 10 grafos aleatórios para cada combinação n (número de vértices) e p (probabilidade de existência de cada aresta). Os pesos são distribuídos uniformemente entre 1 e 10. Consideramos 8 combinações, levando a 80 instâncias no total. Para cada par (n, p), calculamos a média do tempo consumido pelas 10 instâncias para cada algoritmo. O número de vértices varia entre 600 até 5000, enquanto a densidade varia entre 10% e 60%. A comparação computacional entre **BITRDS1** e **BITBKA**$_{MAX}$

nas instâncias aleatórias é apresentada na Tabela 42, mostrando que, para essa faixa de densidade e dimensão do problema, a segunda abordagem (de enumeração "exaustiva" das cliques maximais) não é competitiva.

Tabela 42 – Tempo de execução do algoritmo **BITRDS1** e **BITBKA**$_{MAX}$

Instancia	Tempo de Execução	
(n,p)	**BITRDS1**	**BITBKA**$_{MAX}$
(2500,0.10)	**0.21**	0.48
(5000,0.10)	**1.89**	7.57
(2500,0.20)	**1.12**	9.55
(5000,0.20)	**36.30**	363.19
(2500,0.30)	**15.57**	265.72
(1200,0.40)	**3.39**	81.30
(600,0.50)	**0.71**	21.00
(600,0.60)	**7.35**	718.18

Fonte: Elaborada pelo autor.

REFERÊNCIAS

Akkoyunlu, EA. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, v. 2, n. 1, p. 1–6, 1973.

Andrade, Diogo Vieira; Resende, Mauricio G. C.; Werneck, Renato Fonseca F. Fast Local Search for the Maximum Independent Set Problem. *Proceedings of 7th International Workshop Experimental Algorithms*. 2008, p. 220–234.

Avella, Pasquale; VasilEv, Igor. A computational study of a cutting plane algorithm for university course timetabling. *Journal of Scheduling*, v. 8, n. 6, p. 497–514, 2005.

Avenali, Alessandro. Resolution Branch and Bound and an Application: The Maximum Weighted Stable Set Problem. *Operations Research*, v. 55, n. 5, p. 932–948, 2007.

Babel, Luitpold. A fast algorithm for the maximum weight clique problem. *Computing*, v. 52, n. 1, p. 31–38, 1994.

Baeza-Yates, Ricardo; Gonnet, Gaston H. A new approach to text searching. *Communications of the ACM*, v. 35, n. 10, p. 74–82, 1992.

Balas, Egon; Xue, Jue. Minimum Weighted Coloring of Triangulated Graphs, with Application to Maximum Weight Vertex Packing and Clique Finding in Arbitrary Graphs. *SIAM J. Comput.*, v. 20, n. 2, p. 209–221, 1991.

Balas, Egon; Xue, Jue. Weighted and Unweighted Maximum Clique Algorithms with Upper Bounds from Fractional Coloring. *Algorithmica*, v. 15, n. 5, p. 397–412, 1996.

Balas, Egon; Yu, Chang Sung. Finding a Maximum Clique in an Arbitrary Graph. *SIAM J. Comput.*, v. 15, n. 4, p. 1054–1068, 1986.

Benlic, Una; Hao, Jin-Kao. Breakout Local Search for maximum clique problems. *Computers & Operations Research*, v. 40, n. 1, p. 192 – 206, 2013.

Boas, Matheus G. V.; Santos, Haroldo G.; Brito, Samuel S. Algoritmos Exatos e Heurísticos para o Problema da Detecção de Cliques com Peso acima de um Limiar. *Anais do XLVII Simpósio Brasileiro de Pesquisa Operacional(SBPO)*, 2015.

Boginski, Vladimir; Butenko, Sergiy; Pardalos, Panos M. Mining Market Data: A Network Approach. *Comput. Oper. Res.*, v. 33, n. 11, p. 3171–3184, 2006.

Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M. The maximum clique problem. *Handbook of Combinatorial Optimization (Supplement Volume A)*, v. 4, p. 1–74, 1999.

Borndorfer, Ralf; Grotschel, Martin; Klostermeier, Fridolin; Kuttner, Christian. Telebus Berlin: Vehicle Scheduling in a Dial-a-Ride System. *Proceedings of the 7th International Workshop on Computer Aided Transit Scheduling*, v. 1, p. 391–422, 1999.

Boussier, Sylvain. *Étude de Resolution pour la Programmation Linéaire en Variables binaires*. L'Université Montpellier II, 2008.

Boussier, Sylvain; Vasquez, Michel; Vimont, Yannick; Hanafi, Saïd; Michelon, Philippe. A multi-level search strategy for the 0–1 Multidimensional Knapsack Problem. *Discrete Applied Mathematics*, v. 158, n. 2, p. 97–109, 2010.

Brélaz, Daniel. New Methods to Color Vertices of a Graph. v. 22, n. 4, p. 251–256, 1979.

Brito, Samuel Souza; Santos, Haroldo Gambini. Pivoteamento no Algoritmo Bron-Kerbosch para a Detecção de Cliques com Peso Máximo. *Anais do XLIII Simpósio Brasileiro de Pesquisa Operacional*, 2011.

Brito, Samuel Souza; Santos, Haroldo Gambini; Poggi, Marcus. A Computational Study of Conflict Graphs and Aggressive Cut Separation in Integer Programming. *Electronic Notes in Discrete Mathematics*, v. 50, p. 355–360, 2015.

Bron, Coen; Kerbosch, Joep. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM*, v. 16, n. 9, p. 575–577, 1973.

Brouwer, A. E.; Shearer, Lames B.; Sloane, N. I. A.; Warren; Smith, D. A new table of constant weight codes. *IEEE Trans Inform Theory*, 1990.

Burke, Edmund K; Mareček, Jakub; Parkes, Andrew J; Rudová, Hana. A branch-and-cut procedure for the Udine course timetabling problem. *Annals of Operations Research*, v. 194, n. 1, p. 71–87, 2012.

Butenko, S.; Wilhelm, W. E. Clique-detection Models in Computational Biochemistry and Genomics. *European Journal of Operational Research*, v. 173, p. 1–17, 2005.

Carraghan, R.; Pardalos, P. M. An exact algorithm for the maximum clique problem. *Operations Research Letters*, v. 9, n. 6, p. 375 – 382, 1990a.

Carraghan, R.; Pardalos, P. M. A parallel algorithm for the maximum weight clique problem. *Relatório Técnico Dept. of Computer Science, Pennsylvania State University*, 1990b.

Cazals, Frédéric; Karande, Chinmay. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, v. 407, n. 1, p. 564–568, 2008.

Chiba, Norishige; Nishizeki, Takao. Arboricity and subgraph listing algorithms. *SIAM*

Journal on Computing, v. 14, n. 1, p. 210–223, 1985.

Christofides, Nicos. *Graph Theory: An algorithmic approach*. New York: Academic Press Inc, 1975.

Chvátal, Vasek. Resolution Search. *Discrete Applied Mathematics*, v. 73, n. 1, p. 81–99, 1997.

Corrêa, Ricardo C; Marenco, Javier; Delle Donne, Diego; Koch, Ivo. A Strengthened General Cut-Generating Procedure for the Stable Set Polytope. *Electronic Notes in Discrete Mathematics*, v. 50, p. 261–266, 2015.

Corrêa, Ricardo C.; Michelon, Philippe; Cun, Bertrand Le; Mautor, Thierry; Donne, Diego Delle. A Bit-Parallel Russian Dolls Search for a Maximum Cardinality Clique in a Graph. *CoRR*, v. abs/1407.1209, 2014.

Demassey, S.; Artigues, C.; Michelon, P. An application of resolution search to the RCPSP. *European Conference on Combinatorial Optimization (ECCO XVII)*. Beytouth, 2004.

Fang, Zhiwen; Li, Chu-Min; Qiao, Kan; Feng, Xu; Xu, Ke. Solving Maximum Weight Clique Using Maximum Satisfiability Reasoning. Torsten Schaub; Gerhard Friedrich; Barry O'Sullivan (Ed.), *ECAI 2014 - 21st European Conference on Artificial Intelligence*. IOS Press, 2014, v. 263 of *Frontiers in Artificial Intelligence and Applications*, p. 303–308.

Franco, Liana; Uchoa, Eduardo. SOLVING THE P-DISPERSION PROBLEM AS A SEQUENCE OF SET PACKING FEASIBILITY PROBLEMS. *XLVII SBPO*, 2015.

Gerhards, L; Lindenberg, W. Clique detection for nondirected graphs: Two new algorithms. *Computing*, v. 21, n. 4, p. 295–322, 1979.

Gschwind, Timo; Irnich, Stefan; Podlinski, Isabel; *et al.* Maximum weight relaxed cliques and Russian doll search revisited. 2015.

Gualandi, Stefano; Malucelli, Federico. Exact Solution of Graph Coloring Problems via Constraint Programming and Column Generation. *INFORMS Journal on Computing*, v. 24, n. 1, p. 81–100, 2012.

Harary, Frank; Ross, Ian C. A procedure for clique detection using the group matrix. *Sociometry*, v. 20, n. 3, p. 205–215, 1957.

Held, Stephan; Cook, William; Sewell, Edward C. Maximum-weight stable sets and safe lower bounds for graph coloring. *Math. Program. Comput.*, v. 4, n. 4, p. 363–381, 2012.

Hotta, KAZUHIRO; Tomita, ETSUJI; Takahashi, HARUHISA. A view-invariant human face detection method based on maximum cliques. *Trans. IPSJ*, v. 44, n. SIG14 (TOM9, p. 57–70, 2003.

Johnston, HC. Cliques of a graph-variations on the Bron-Kerbosch algorithm. *International Journal of Computer & Information Sciences*, v. 5, n. 3, p. 209–238, 1976.

Karp, Richard M. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*. 1972, p. 85–103.

Kiviluoto, Lasse; terg, PatricR.J.; Vaskelainen, VesaP. Algorithms for finding maximum transitive subtournaments. *Journal of Combinatorial Optimization*, p. 1–13, 2014.

Koch, Thorsten; Achterberg, Tobias; Andersen, Erling; Bastert, Oliver; Berthold, Timo; Bixby, Robert E; Danna, Emilie; Gamrath, Gerald; Gleixner, Ambros M; Heinz, Stefan; *et al.* MIPLIB 2010. *Mathematical Programming Computation*, v. 3, n. 2, p. 103–163, 2011.

Komosko, Larisa; Batsyn, Mikhail; San Segundo, Pablo; Pardalos, Panos M. A fast greedy sequential heuristic for the vertex colouring problem based on bitwise operations. *Journal of Combinatorial Optimization*, p. 1–13, 2015.

Kumlander, Deniss. On Importance of a Special Sorting in the Maximum-Weight Clique Algorithm Based on Colour Classes. v. 14, p. 165–174, 2008.

Makino, Kazuhisa; Uno, Takeaki. New algorithms for enumerating all maximal cliques. *Algorithm Theory-SWAT 2004*, Springer, p. 260–272. 2004.

Malladi, Krishna Teja. *Cluster Restricted Maximum Weight Clique Problem and Linkages with Satellite Image Acquisition Scheduling*. SIMON FRASER UNIVERSITY, Canad014.

Mannino, Carlo; Stefanutti, Egidio. An Augmentation Algorithm for the Maximum Weighted Stable Set Problem. *Computational Optimization and Applications*, v. 14, n. 3, p. 367–381, 1999.

Maslov, Evgeny; Batsyn, Mikhail; Pardalos, PanosM. Speeding up branch and bound algorithms for solving the maximum clique problem. *Journal of Global Optimization*, v. 59, n. 1, p. 1–21, 2014.

Matula, David W.; Beck, Leland L. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM*, v. 30, n. 3, p. 417–427, 1983.

McClosky, Benjamin; Hicks, IllyaV. Combinatorial algorithms for the maximum k-plex problem. *Journal of Combinatorial Optimization*, v. 23, n. 1, p. 29–49, 2012.

Mehrotra, Anuj; Trick, Michael A. A Column Generation Approach for Graph Coloring. *INFORMS Journal on Computing*, v. 8, n. 4, p. 344–354, 1996.

Miao, Zhuqi; Balasundaram, Balabhaskar; Pasiliao, EduardoL. An exact algorithm for the maximum probabilistic clique problem. *Journal of Combinatorial Optimization*, v. 28, n. 1, p. 105–120, 2014.

Nemhauser, G. L.; Sigismondi, G. A Strong Cutting Plane/Branch-and-Bound Algorithm for Node Packing. *The Journal of the Operational Research Society*, v. 43, n. 5, p. 443–457, 1992. URL `http://dx.doi.org/10.2307/2583564`.

Nemhauser, G. L.; Trotter, L. E. Vertex packings: Structural properties and algorithms. *Math. Programming*, v. 8, 1975.

Ostergard, Patric R.J. A New Algorithm for the Maximum-Weight Clique Problem. *Electronic Notes in Discrete Mathematics*, v. 3, n. 0, p. 153 – 156, 1999. 6th Twente Workshop on Graphs and Combinatorial Optimization.

Ostergard, Patric R.J. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, v. 120, p. 197–207, 2002. Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization.

Ostergard, Patric RJ; Vaskelainen, Vesa; Mosig, Axel. Algorithms for the combinatorial best barbeque problem. *MATCH Communications in Mathematical and in Computer Chemistry*, v. 58, p. 309–321, 2007.

Ostergard, Patric R.J.; Vaskelainen, VesaP. Russian doll search for the Steiner triple covering problem. *Optimization Letters*, v. 5, n. 4, p. 631–638, 2011.

Padberg, Manfred W. On the facial structure of set packing polyhedra. *Mathematical Programming*, v. 5, n. 1, p. 199–215, 1973.

Palpant, M.; Artigues, C.; Oliva, C. MARS: une méthode hybride pour la résolution de problèmes de satisfaction de contraintes. *Conférence conjointe FRANCORO V/ROADEF 2007*. 2007, p. 355–356.

Pardalos, Panos M.; Xue, Jue. The maximum clique problem. *Journal of Global Optimization*, v. 4, n. 3, p. 301–328, 1994.

Posta, Marius; Ferland, Jacques A.; Michelon, Philippe. Generalized resolution search. *Discrete Optimization*, v. 8, n. 2, p. 215–228, 2011.

Pullan, Wayne. Approximating the maximum vertex/edge weighted clique using local search. *Journal of Heuristics*, v. 14, n. 2, p. 117–134, 2008.

Rebennack, Steffen; Oswald, Marcus; Theis, DirkOliver; Seitz, Hanna; Reinelt, Gerhard; Pardalos, PanosM. A Branch and Cut solver for the maximum stable set problem. *Journal of Combinatorial Optimization*, v. 21, n. 4, p. 434–457, 2011.

Rossi, F; Smriglio, S. A branch-and-cut algorithm for the maximum cardinality stable set problem. *Operations Research Letters*, v. 28, n. 2, p. 63 – 74, 2001.

Santos, Haroldo G.; Toffolo, Túlio; Gomes, Rafael; Ribas, Sabir. Integer programming techniques for the nurse rostering problem. *Annals of Operations Research*, p. 1–27, 2014.

Segundo, Pablo San; Matia, Fernando; Rodriguez-Losada, Diego; Hernando, Miguel. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, v. 7, n. 3, p. 467–479, 2013.

Segundo, Pablo San; Nikolaev, Alexey; Batsyn, Mikhail. Infra-chromatic bound for exact maximum clique search. *Computers & Operations Research*, v. 64, p. 293 – 303, 2015.

Segundo, Pablo San; Rodriguez-Losada, Diego; Jimenez, Agustin. An exact bit-parallel algorithm for the maximum clique problem. *Computers OR*, v. 38, n. 2, p. 571–581, 2011.

Segundo, Pablo San; Rodriguez-Losada, Diego; Matia, Fernando; Galan, Ramon. Fast exact feature based data correspondence search with an efficient bit-parallel MCP solver. *Appl. Intell.*, v. 32, n. 3, p. 311–329, 2010.

Segundo, Pablo San; Tapia, Cristóbal. Relaxed approximate coloring in exact maximum clique search. *Computers & OR*, v. 44, p. 185–192, 2014.

Segundo, Pablo San; Tapia, Cristobal; Puente, Julio; Rodriguez-Losada, Diego. A new exact bit-parallel algorithm for sat. *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*. IEEE, 2008, v. 2, p. 59–65.

Sewell, E. C. A Branch and Bound Algorithm for the Stability Number of a Sparse Graph. *INFORMS J. on Computing*, v. 10, n. 4, p. 438–447, 1998.

Shimizu, Satoshi; Yamaguchi, Kazuaki; Saitoh, Toshiki; Masuda, Sumio. Some Improvements on Kumlander-s Maximum Weight Clique Extraction Algorithm. v. 6, n. 12, p. 1770 – 1774, 2012.

Sloane, N. J. A. Unsolved Problems in Graph Theory Arising from the Study of Codes. *in Graph Theory Notes of New York 18*. 1989, p. 11–20.

Tavares, Wladimir Araujo; Neto, Manoel Bezerra Campelo; Rodrigues, Carlos Diego; Michelon, Philippe. Um Algoritmo de Branch and Bound para o Problema da Clique Maxima Ponderada. *Proceedings of XLVII SBPO*, v. 1, 2015.

Tavares, Wladimir Araújo; Neto, Manoel Bezerra Campêlo; Rodrigues, Carlos Diego; Michelon, Philippe. BITCLIQUE: um algoritmo de Branch-and-Bound para o problema da clique mma ponderada. *Proceedings of XLVIII SBPO*, v. 1, 2016.

Tomita, Etsuji; Seki, Tomokazu. An Efficient Branch-and-bound Algorithm for Finding a Maximum Clique. *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science*. Springer-Verlag, 2003, DMTCS'03, p. 278–289.

Tomita, Etsuji; Sutani, Yoichi; Higashi, Takanori; Takahashi, Shinya; Wakatsuki, Mitsuo. A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique. *WALCOM*. 2010, p. 191–203.

Tomita, Etsuji; Tanaka, Akira; Takahashi, Haruhisa. The Worst-case Time Complexity for Generating All Maximal Cliques and Computational Experiments. *Theor. Comput. Sci.*, v. 363, n. 1, p. 28–42, 2006.

Trukhanov, Svyatoslav; Balasubramaniam, Chitra; Balasundaram, Balabhaskar; Butenko, Sergiy. Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. *Computational Optimization and Applications*, v. 56, n. 1, p. 113–130, 2013.

Verfaillie, Gerard; Lemae, Michel; Schiex, Thomas. Russian Doll Search for Solving Constraint Optimization Problems. *National Conference on Artificial Intelligence*. 1996, p. 181–187.

Warren, Jeffrey S; Hicks, Illya V. Combinatorial branch-and-bound for the maximum weight independent set problem. *Relat tico, Texas A&M University*, 2006.

Warrier, Deepak; Wilhelm, Wilbert E; Warren, Jeffrey S; Hicks, Illya V. A branch-and-price approach for the maximum weight independent set problem. *Networks*, v. 46, n. 4, p. 198–209, 2005.

Welsh, Dominic JA; Powell, Martin B. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, v. 10, n. 1, p. 85–86, 1967.

Wu, Qinghua; Hao, Jin-Kao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, , n. 0, p. –, 2014.

Wu, Qinghua; Hao, Jin-Kao. Solving the winner determination problem via a weighted maximum clique heuristic. *Expert Systems with Applications*, v. 42, n. 1, p. 355–365, 2015.

Wu, Qinghua; Hao, Jin-Kao; Glover, Fred. Multi-neighborhood tabu search for the maximum weight clique problem. *Annals OR*, v. 196, n. 1, p. 611–634, 2012.

Yamaguchi, K.; Masuda, S. A new exact algorithm for the maximum weight clique problem. Proc. of the 23rd International Technical Conference on Circuits/Systems, Computers and Communications, 2008, p. 317–320.