UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE ESTATÍSTICA E MATEMÁTICA APLICADA
CURSO DE GRADUAÇÃO EM MATEMÁTICA INDUSTRIAL

ALYSSON MONTEIRO BARBOSA VASCONCELOS

A INSERÇÃO DE MÉTODOS NO RESOLVEDOR CPLEX: UMA APLICAÇÃO AO PROBLEMA DE EMPARELHAMENTO DE PESO MÁXIMO COM RESTRIÇÕES DE CONFLITO

FORTALEZA
2018

ALYSSON MONTEIRO BARBOSA VASCONCELOS

A INSERÇÃO DE MÉTODOS NO RESOLVEDOR CPLEX: UMA APLICAÇÃO AO PROBLEMA DE EMPARELHAMENTO DE PESO MÁXIMO COM RESTRIÇÕES DE CONFLITO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Matemática Industrial do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Matemática Industrial.

Orientador: Prof. Dr. Manoel Bezerra Campêlo Neto

Coorientador: Prof. Dr. Carlos Diego Rodrigues

FORTALEZA

2018

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)

V45i Vasconcelos, Alysson Monteiro Barbosa.
A inserção de métodos no resolvedor CPLEX: Uma aplicação ao problema de Emparelhamento de Peso Máximo com Restrições de Conflito / Alysson Monteiro Barbosa Vasconcelos. – 2018.
110 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Matemática Industrial, Fortaleza, 2018.
Orientação: Prof. Dr. Manoel Bezerra Campêlo Neto.
Coorientação: Prof. Dr. Carlos Diego Rodrigues.

1. CPLEX. 2. Programação Inteira. 3. Branch-and-cut. 4. Relaxação Lagrangiana. 5. Emparelhamento de Peso Máximo com Restrições de Conflito. I. Título.

CDD 510

ALYSSON MONTEIRO BARBOSA VASCONCELOS

A INSERÇÃO DE MÉTODOS NO RESOLVEDOR CPLEX: UMA APLICAÇÃO AO PROBLEMA DE EMPARELHAMENTO DE PESO MÁXIMO COM RESTRIÇÕES DE CONFLITO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Matemática Industrial do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Matemática Industrial.

Aprovada em: 04 de Dezembro de 2018

BANCA EXAMINADORA

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

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

Prof. Dr. Ronan Pardo Soares
Universidade Federal do Ceará (UFC)

Prof. Dr. Jesus Ossian da Cunha Silva
Universidade Federal do Ceará (UFC)

Aos meus pais, por acreditarem e investirem em mim. Mãe, seu cuidado e dedicação foi que deram a esperança e força para seguir. Pai, sua presença e história de vida significaram a certeza de que estou no caminho certo.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus por ter me amparado e concedido à força necessária para a superação dos inúmeros obstáculos, pois sem Ele nada seria possível.

Agradeço aos meus pais, Carlos Alberto e Maria Zildilene, por todo exemplo, amor, apoio, comprometimento e investimentos depositados em mim durante toda minha vida escolar e acadêmica. A vocês, dedico o meu maior agradecimento.

Agradeço à Universidade Federal do Ceará e ao corpo docente do Departamento de Estatística e Matemática Aplicada por me oferecer um ambiente para o meu desenvolvimento intelectual e social durante minha formação.

Agradeço aos meus orientadores e professores Manoel Bezerra Campêlo Neto e Carlos Diego Rodrigues, por todo o suporte, paciência e interesse pelo meu aprendizado desde o começo do curso, transmitindo conhecimentos necessários para que esta monografia fosse possível. Agradeço por serem meus orientadores e pela confiança em meu potencial. Agradeço também ao professor Rafael Castro de Andrade pelos conhecimentos e experiências de vida compartilhadas em disciplinas que foram a base para a construção deste trabalho.

Agradeço à Lady Daiany, que se manteve sempre presente e confiante ao meu lado em toda a graduação, além do seu carinho, amor e positividade em acreditar no meu êxito. Seu apoio foi crucial para que este trabalho fosse realizado.

E por fim, agradeço aos meus amigos e colegas de turma que adquiri durante a graduação e que contribuíram de alguma forma, direta ou indiretamente, para este trabalho. Em especial, agradeço aos meus amigos e companheiros de curso, Anderson Henrique, Yan Saraiva e Rômulo Marques, que me proporcionaram inúmeras contribuições para a evolução do meu conhecimento e aprovações em várias disciplinas fundamentais ao longo do curso.

“Faça o teu melhor, na condição que você tem, enquanto você não tem condições melhores, para fazer melhor ainda!”

(Mario Sergio Cortella)

RESUMO

Métodos de otimização procuram encontrar boas soluções ou, se possível, a melhor solução dentro de um conjunto viável de possibilidades em um tempo razoável. Em geral, eles demandam o ajuste de parâmetros e uso de diferentes estratégias para encontrar soluções cada vez mais eficientes e com um menor esforço computacional. Há disponíveis alguns resolvedores, como o CPLEX, capazes de tratar diversos problemas de otimização que se enquadram nas áreas de Programação Linear, Programação Inteira, Programação Mista e, até certa extensão, Programação Quadrática. Além disso, eles nos fornecem a possibilidade de incorporar nossos próprios métodos às suas ferramentas, com o intuito de fortalecer ainda mais o resolvedor no tratamento do problema em questão. Diante dessa circunstância, nosso trabalho exemplifica, através do problema de Emparelhamento de Peso Máximo com Restrições de Conflito, como implementar técnicas estudadas durante a graduação para resolução de problemas de programação linear inteira, tais como planos de corte, relaxação lagrangiana, branch-and-cut, heurísticas lagrangianas. Mostramos como usar, customizar ou incorporar esses métodos no ambiente do resolvedor CPLEX, bem como avaliamos os efeitos no desempenho do resolvedor devido às diferentes formas de uso, parametrizações ou alterações aplicadas.

Palavras-chave: CPLEX. Programação Inteira. Branch-and-cut. Relaxação Lagrangiana. Emparelhamento de Peso Máximo com Restrições de Conflito.

ABSTRACT

Optimization methods seek to find good solutions or, if possible, the best one within a feasible set of possibilities in a reasonable time. In general, they require parameters adjustment and the use of different strategies to find increasingly efficient solutions while taking lesser computational effort. Some solvers, such as CPLEX, are available to handle various optimization problems that fall into the areas of Linear Programming, Integer Programming, Mixed Integer Programming and, to some extent, Quadratic Programming. In addition, they give us the possibility of incorporating our own methods into their tools, in order to further strengthen the solver to tackle the problem in focus.. Given this circumstance, our work exemplifies, through the Maximum Weight Matching Problem with Conflict Constraints, how to implement techniques studied during our graduation course (in order to solve problems of integer linear programming), such as cutting planes, lagrangian relaxation, branch-and-cut, lagrangian heuristics. We show how to use, customize or incorporate these methods in the CPLEX solver environment, as well as we evaluate the effects on the solver performance due to different ways of use, parameterizations or applied changes..

Keywords: CPLEX. Integer Programming. Branch-and-cut. Lagrangian Relaxation. Maximum Weight Matching Problem with Conflict Constraints.

LISTA DE FIGURAS

Figura 1 – Grafo $G = (V,E)$ 18
Figura 2 – Grafo Simples 19
Figura 3 – Emparelhamento Máximo 21
Figura 4 – Emparelhamento de Peso Máximo 21
Figura 5 – $V(G) = S \cup \overline{S}$ 24
Figura 6 – Grafo G^* 26
Figura 7 – Corte ímpar mínimo. $|S^*|$ é ímpar. 26
Figura 8 – Grafo $G_c = (E,E_c)$ 43
Figura 9 – Emparelhamento de Peso Máximo com Restrições de Conflito 43
Figura 10 – Conjunto Independente Máximo 45
Figura 11 – Conjunto Independente de Peso Máximo 45
Figura 12 – Grafo de conflitos estendido G_{c*}: G_c com inclusão de arestas entre arestas adjacentes de G 46
Figura 13 – Os primeiros grafos de *Mycielski* ilustrados 76
Figura 14 – *Branch* dicotômico em variável 83
Figura 15 – *Branch* por fixação de variável 83
Figura 16 – GUB 83
Figura 17 – Ramificação 88

LISTA DE TABELAS

Tabela 1 – Resultados 40
Tabela 2 – Problema de Emparelhamento de Peso Máximo com Conflitos - Resultados Computacionais Obtidos Pelas Versões 4.1, 4.2.1, 4.2.2, 4.2.3, 4.2.4 e 4.2.5 . 79
Tabela 3 – Problema de Emparelhamento de Peso Máximo com Conflitos - Qualidade dos Limites Inferiores e Superiores Obtidos Pelas Versões 4.1, 4.2.1, 4.2.2, 4.2.3, 4.2.4 e 4.2.5 80
Tabela 4 – Resultados utilizando *goals* 104

LISTA DE ALGORITMOS

Algoritmo 1 – Corte ímpar mínimo 30

Algoritmo 2 – Geração de Cortes 53

Algoritmo 3 – Subgradiente 56

Algoritmo 4 – branch_and_bound 85

Algoritmo 5 – avaliaNo 86

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Versão 2.1 31
Código-fonte 2 – Versão 2.2.1 34
Código-fonte 3 – Versão 2.2.2 37
Código-fonte 4 – Versão 4.1 58
Código-fonte 5 – Versão 4.2.1 61
Código-fonte 6 – Versão 4.2.2 67
Código-fonte 7 – Versão 4.2.3 70
Código-fonte 8 – Versão 4.2.4 72
Código-fonte 9 – Versão 4.2.5 74
Código-fonte 10 – Versão Final 90

SUMÁRIO

1	**INTRODUÇÃO**	15
2	**EMPARELHAMENTO DE PESO MÁXIMO**	18
2.1	**Conceitos preliminares**	18
2.2	**Definição do problema**	20
2.3	**Complexidade**	21
2.4	**Modelo matemático**	22
2.5	**Restrições de blossom**	23
2.6	**Problema do T-corte mínimo**	27
2.7	**Implementações**	29
2.7.1	*Resolução direta do modelo*	31
2.7.2	*Resolução iterativa do modelo com restrições de blossom*	33
2.8	**Experimentos computacionais**	39
3	**EMPARELHAMENTO DE PESO MÁXIMO COM RESTRIÇÕES DE CONFLITO**	41
3.1	**Restrições disjuntivas**	41
3.2	**Definição do problema**	42
3.3	**Modelo matemático**	44
3.4	**Equivalência com conjunto independente máximo**	44
4	**RELAXAÇÃO LAGRANGIANA**	47
4.1	**Definição do dual lagrangiano**	47
4.2	**Resolução do dual lagrangiano**	50
4.2.1	*Método de geração de cortes*	50
4.2.2	*Método de subgradientes*	53
4.3	**Heurística lagrangiana**	55
4.4	**Implementações**	57
4.4.1	*Resolução da relaxação linear do modelo*	58
4.4.2	*Resolução da relaxação lagrangiana do modelo*	61
4.5	**Experimentos computacionais**	75
5	**MÉTODO EXATO**	81
5.1	**Método branch-and-bound**	81

5.1.1 *Ramificação* 82
5.1.2 *Avaliações* 84
5.1.3 *Poda* 84
5.1.4 *Branch-and-bound com relaxação linear e divisão dicotômica* 84
5.2 **Árvore de subgradientes** 85
5.3 **Ramificação** 87
5.4 **Implementações** 88
5.5 **Experimentos computacionais** 102
6 **CONCLUSÕES** 105
REFERÊNCIAS 108

1 INTRODUÇÃO

Os problemas em pesquisa operacional usualmente demandam boas soluções em um tempo aceitável e isso, em geral, envolve a necessidade de calibração de vários parâmetros em métodos de solução de propósito geral ou a necessidade de métodos de solução específicos e mais eficientes. Quase sempre, existem várias abordagens possíveis para um mesmo problema, levando a métodos de resolução diferentes; entretanto o tempo de execução desses métodos muitas vezes é impraticável, o que torna necessário o auxílio de técnicas mais avançadas para a redução do esforço computacional.

Muitos desses problemas são de otimização, nos quais se procura encontrar a melhor solução, segundo um certo critério, dentro de um conjunto de possibilidades. Um primeiro passo para tratá-los consiste em modelá-los matematicamente, da forma mais fiel possível. Conforme as características dos modelos, os problemas podem ser classificados como, por exemplo, problemas de PL (Programação Linear), PLI (Programação Linear Inteira), OC (Otimização Combinatória) etc. Em cada uma dessas classes existem problemas fundamentais, que se revestem de grande importância por sua aplicabilidade, por figurarem recorrentemente como subproblemas ou ainda por suas propriedades teóricas.

Nos últimos anos, vários problemas clássicos de Otimização Combinatória têm sido revisitados, com a inclusão, ao conjunto de restrições, de um novo critério a ser obedecido. Nesses problemas, uma solução é dada pela escolha de um subconjunto de elementos de um certo universo (por exemplo, vértices ou arestas de um grafo), e o critério adicional a ser satisfeito diz respeito a restrições de conflito entre tais elementos, que nada mais são do que restrições definidas sobre pares de elementos, proibindo que ambos possam figurar em uma solução.

Neste trabalho, foi realizado um estudo sobre o problema clássico de Emparelhamento de Peso Máximo, porém agora acrescido de restrições de conflitos. Nesse caso, certos pares de arestas (não apenas arestas adjacentes) são proibidas de participarem do emparelhamento desejado. No Capítulo 2, revisitamos o problema original, apresentando sua definição, complexidade, modelos matemáticos, com destaque para as restrições de blossom e sua separação. Dedicada à versão com conflitos, o Capítulo 3 a define formalmente, apresenta sua complexidade computacional e a modela via programação inteira 0-1. Todo esse conteúdo é uma compilação de resultados da literatura.

Mais do que estudar esse problema, o objetivo desta monografia é utilizá-la para ilustrar diferentes aspectos que devem ser considerados no momento de se resolver efetivamente

um problema de otimização, especialmente a opção por uma ou outra abordagem, dentre diversas alternativas possíveis, e ainda a escolha da forma segundo a qual se implementar o método escolhido, usando um resolvedor de otimização matemática.

Nesse sentido, já no Capítulo 2, implementamos duas estratégias de resolução do problema de Emparelhamento de Peso Máximo. Embora seja um problema polinomial, que pode ser resolvido por algoritmo combinatório específico, ele também pode ser abordado via programação inteira ou mesmo programação linear. Tal opção passa a ser interessante particularmente quando se deseja modificar, mesmo que ligeiramente, a definição do problema (suas restrições ou função objetivo). Com relação à versão com restrições de conflito, que se torna NP-Difícil, apresentamos duas abordagens, uma visando encontrar um bom limite superior (Capítulo 4) e outra procurando resolver o problema exatamente (Capítulo 5).

Para aplicação das abordagens e realização dos testes computacionais, foi utilizado o *software* de otimização `IBM ILOG CPLEX Optimization Studio v12.7.1.0`(CPLEX), que já implementa e disponibiliza um conjunto de métodos eficientes para resolver problemas de otimização ou para fornecer boas soluções, dependendo do tamanho das instâncias consideradas. A escolha do resolvedor se deu principalmente pelo fato de ele disponibilizar uma versão acadêmica e pelo seu alto reconhecimento em eficiência na resolução de problemas de otimização.

Mesmo existindo *softwares* como esse, a dificuldade de solução de um problema NP-Difícil está sempre presente e, pensando nisso, os desenvolvedores nos deram a possibilidade de configurar os métodos de resolução que o resolvedor executa, através dos parâmetros disponibilizados, ou de incrementar a estrutura de código, acrescentando funções do usuário ao processo de otimização. A alteração ou ajuste de pequenos critérios de escolha já mostram um diferencial no tempo de resolução do problema, podendo-se gerar uma possível melhora ou piora.

Neste trabalho, aplicamos alguns métodos clássicos de otimização, como planos de corte, relaxação lagrangiana, *branch-and-bound*, assim como adaptações e combinações destes. Utilizando o resolvedor, objetivamos mostrar como uma implementação em C_{++} pode ser realizada com a API do CPLEX, tanto para desenvolvimento quanto para verificação de eficiência de métodos de solução. Experimentos computacionais foram realizados, a fim de apresentar uma comparação entre os tempos de resolução obtidos pelos métodos padrões do resolvedor com aqueles dos métodos implementados pelo usuário. Com isso, analisamos mudanças em tempo de resolução que se podem obter, caso alterações na escolha/implementação de um método sejam realizadas. Os testes foram efetuados com instâncias, geradas aleatoriamente, do problema de

emparelhamento de peso máximo, em sua versão clássica bem com na variante com restrições de conflito.

2 EMPARELHAMENTO DE PESO MÁXIMO

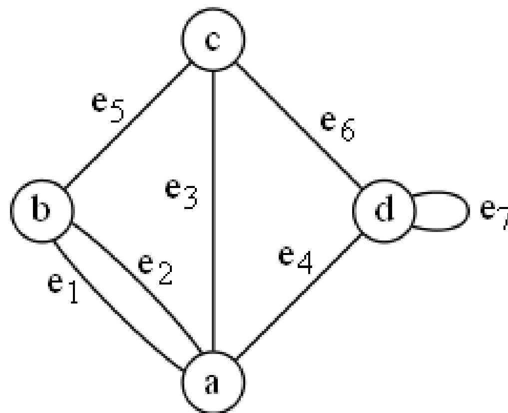
Neste Capítulo, iremos definir o problema de emparelhamento de peso máximo, enunciar sua complexidade computacional e apresentar modelos de programação matemática. Começamos, porém, com alguns conceitos básicos em teoria do grafos que serão necessários durante o trabalho.

2.1 Conceitos preliminares

Um **grafo** $G = (V,E)$ é um par ordenado, definido por *V*, o conjunto de vértices, e *E*, o multiconjunto de arestas, que são pares não ordenados de vértices. A notação *V(G)* e *E(G)* será utilizada para representar o conjunto de vértices e de arestas de *G*, respectivamente. A cardinalidade do conjunto de vértices de *G* é denotada por |V|, assim como a cardinalidade do conjunto de arestas por |E|. Dada uma aresta $e = (u,v)$, tal que $e \in E$, com *u* e $v \in V$, então *u* e *v* são as **extremidades** de *e*, que é a **aresta incidente** a *u* e *v*. Além disso, quando dois vértices são conectados por uma aresta, eles são chamados de **vértices adjacentes**. De modo similar, duas arestas que possuem uma extremidade comum são ditas **arestas adjacentes**.

Um grafo pode ser representado de diversas formas. A mais comum é a representação geométrica, na qual se utiliza de pontos para representar os vértices e segmentos de retas para as arestas. A Figura 1 mostra uma representação geométrica de um grafo *G* com $V(G) = \{a,b,c,d\}$ e $E(G) = \{e_1 = (a,b), e_2 = (a,b), e_3 = (a,c), e_4 = (a,d), e_5 = (b,c), e_6 = (c,d), e_7 = (d,d)\}$.

Figura 1 – Grafo $G = (V,E)$

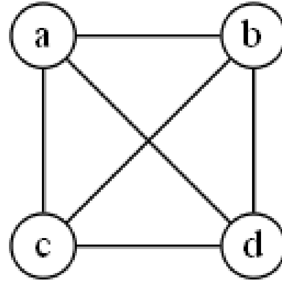


Fonte: Elaborada pelo autor.

Um grafo $G = (V,E)$ é dito **grafo simples** se ele não contém laços e nem arestas múltiplas. Um **laço** é uma aresta que apresenta extremidades iguais, e **arestas múltiplas**

constituem-se em arestas que apresentam os mesmos vértices como extremidades. Observando o exemplo da Figura 1, podemos notar que a aresta $e_7 = (d,d)$ é um laço e as arestas $e_1 = (a,b)$ e $e_2 = (a,b)$ são múltiplas. A Figura 2 ilustra um grafo simples.

Figura 2 – Grafo Simples



Fonte: Elaborada pelo autor.

Um grafo simples $G = (V,E)$ é **bipartido** se o conjunto de vértices pode ser particionado em dois subconjuntos disjuntos X e Y ($V = X \cup Y$, $X \cap Y = \emptyset$), tais que toda aresta tem uma extremidade em X e outra em Y, quer dizer, $E \subset X \times Y$. Quando $E = X \times Y$, o grafo é dito **bipartido completo**. Por exemplo, removendo-se as arestas (a,c) e (b,d) do grafo da Figura 2, ele torna-se bipartido completo.

Um **ciclo** é uma sequência (ou conjunto ordenado) de arestas $\{e_1, e_2, \dots, e_p\}$, definida por uma sequência de vértices $\{v_1, v_2, \dots, v_p\}$ distintos, tais que $e_i = (v_i, v_{i+1})$, $i = 1, \dots, p-1$, e $e_p = (v_p, v_1)$. Por exemplo, o grafo da Figura 2 contém os ciclos $\{(a,b),(b,d),(d,a)\}$ e $\{(a,b),(b,d),(d,c),(c,a)\}$. Um ciclo é dito **ciclo ímpar** se a cardinalidade do seu conjunto de vértices (e de arestas) é ímpar; caso contrário é um **ciclo par**. Pode-se demonstrar que um grafo é bipartido se, e somente se, não possui ciclo ímpar (BONDY *et al.*, 1976).

A **vizinhança** de um vértice v, denotado por $N(v)$, no grafo é o conjunto de todos os vértices adjacentes a v, ou seja, w pertence a $N(v)$ se o par (v,w) é uma aresta do grafo. Associado à vizinhança de v, também definimos o conjunto $\delta(v)$, compreendido pelas arestas que possuem v como extremidade. Em outros termos $w \in N(v)$ se, e somente se, $(v,w) \in \delta(v)$. Na Figura 2, temos $N(a) = \{b,c,d\}$ e $\delta(a) = \{(a,b),(a,c),(a,d)\}$.

Para um subconjunto $S \subseteq V(G)$ de vértices, definimos $E(S) = \{(u,v) \in E(G) : u \in S, v \in S\}$ e $\delta(S) = \{(u,v) \in E(G) : u \in S, v \in V(G) \setminus S\}$, respectivamente, como o subconjunto de arestas com ambas as extremidades ou com exatamente uma extremidade em S. Note que $\delta(S) = \delta(V(G) \setminus S)$. Na Figura 2, temos que $E(\{a,b,c\}) = \{(a,b),(a,c),(b,c)\}$ e $\delta(\{a,b,c\}) = \delta(d) = \{(a,d),(b,d),(c,d)\}$. Sendo não vazio, $\delta(S)$ é dito o **corte de arestas** induzido por S.

Se $s \in S$ e $t \in V(G) \setminus S$, então o corte $\delta(S)$ é chamado (s,t)**-corte**. Quando pesos são atribuídos às arestas, o peso de um corte é a soma dos pesos das arestas que o compõem.

Um **emparelhamento** é um subconjunto de arestas duas-a-duas não adjacentes, isto é, que não compartilhem um vértice. As extremidades de qualquer aresta de um emparelhamento são ditas cobertas por ele. O emparelhamento é dito **perfeito** se cobre todos os vértices do grafo; será dito **emparelhamento máximo**, se é um emparelhamento de máxima cardinalidade. Na Figura 2, $\{(a,b),(c,d)\}$ é um emparelhamento que cobre todos os vértices, sendo portanto máximo (e perfeito).

Em um grafo onde os vértices representam atletas e as arestas indicam possíveis duplas para disputar um campeonato, um emparelhamento consiste em um subconjunto de duplas, onde cada jogador participa de no máximo uma delas. Um emparelhamento perfeito seria uma dessas escolhas onde todos participam de (exatamente) uma dupla.

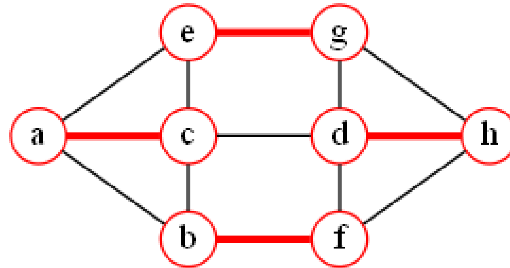
2.2 Definição do problema

Considere um grafo simples *G=(V, E)*, com um conjunto de vértices V e um conjunto de arestas E. Seja também uma função $c : E \to \mathbb{R}_+$, que atribui pesos não negativos às arestas de G. Por simplicidade, denotamos por c_e o custo $c(e)$ da aresta e.

Como vimos, um emparelhamento M em *G=(V, E)* é um conjunto de arestas, onde todo vértice de G incide em no máximo um elemento de M, ou seja, as arestas presentes em M não compartilham extremidades. O problema do **Emparelhamento de Peso Máximo (EPM)** consiste em encontrar um emparelhamento cuja soma dos pesos de suas arestas seja o máximo possível. Quando os pesos das arestas são iguais entre si, um emparelhamento de peso máximo é exatamente um emparelhamento máximo.

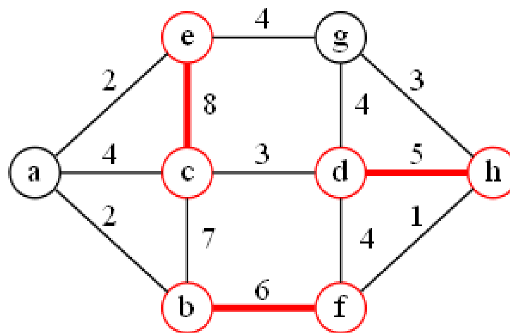
Podemos perceber que a Figura 3 e a Figura 4 representam o mesmo grafo (com o mesmo conjunto de vértices e arestas), sendo que na segunda as arestas estão ponderadas. As figuras destacam, respectivamente, um emparelhamento máximo e outro de peso máximo. É fácil ver como a função de pesos interfere na solução do problema: na Figura 3, o emparelhamento máximo é $\{(a,c),(b,f),(e,g),(d,h)\}$, tendo cardinalidade 4; na Figura 4, o emparelhamento de peso máximo é $\{(e,c),(b,f),(d,h)\}$, com peso ótimo de 19 unidades.

Figura 3 – Emparelhamento Máximo



Fonte: Elaborada pelo autor.

Figura 4 – Emparelhamento de Peso Máximo



Fonte: Elaborada pelo autor.

2.3 Complexidade

O problema de emparalhamento com peso máximo pode ser resolvido em tempo polinomial. Na literatura, podemos encontrar vários algoritmos para esse fim. Possivelmente, o mais conhecido deles deve-se a Jack Edmonds (EDMONDS, 1965b). Embora proposto para o problema sem pesos, tal algoritmo pode ser adaptado para o caso ponderado, mantendo sua complexidade (LOVÁSZ; PLUMMER, 1986). Para um grafo de entrada arbitrário, com $n = |V(G)|$ vértices e $m = |E(G)|$ arestas, o algoritmo de Edmonds tem complexidade $O(n^2m)$. Um aprimoramento desse algoritmo, devido a Micali e Vazirani (MICALI; VAZIRANI, 1980), leva-o a executar em $O(\sqrt{n}m)$. Uma recente implementação em C_{++} desse algoritmo pode ser encontrada em (KOLMOGOROV, 2009).

O algoritmo de Edmonds é uma generalização do algoritmo de Ford-Fulkerson (FORD L. R.; FULKERSON, 1956), que resolve o problema de emparelhamento máximo em grafos bipartidos. Na verdade, o algoritmo de Ford-Fulkerson é comumente chamado de método, dado que ele pode ser implementado de diferentes formas. A rigor, ele destina-se à determinação do fluxo máximo em uma rede. Note que o problema de emparelhamento máximo em um grafo bipartido $G = (X \cup Y, E)$ é equivalente ao de fluxo máximo em uma rede obtida de G,

direcionando as arestas de X para Y e adicionando um nó fonte s que se liga a todos os vértices em X bem como um nó sumidouro t que recebe arco de todos os vértices em Y. Atribuindo capacidade unitária a todos os arcos e determinando o fluxo máximo de s para t, obtemos um emparelhamento máximo, constituindo pelas arestas de G que transportarem fluxo.

Uma implementação do método de Ford-Fulkerson, proposta por Edmonds e Karp (EDMONDS; KARP, 1972), leva a um algoritmo de complexidade $O(nm^2)$. Um melhoria desse algoritmo foi desenvolvida por Hopcroft e Karp (HOPCROFT; KARP, 1973), reduzindo a complexidade para $O(\sqrt{n}m)$.

Vale destacar ainda o clássico algoritmo Húngaro, que resolve o problema em um grafo bipartido ponderado, neste caso comumente chamado problema de alocação. Trata-se de um dos primeiros algoritmos de Otimização Combinatória, proposto em 1955, por Harold Kuhn. Ele usa como subrotina um procedimento de caminho mínimo modificado. A depender da implementação desse procedimento, a complexidade do algoritmo Húngaro pode chegar a $O(n^3)$.

2.4 Modelo matemático

Considere um grafo $G = (V,E)$ com custo c_e associado a cada aresta $e \in E$. Usando uma variável binária x_e para indicar se a aresta e é escolhida $(x_e = 1)$ ou não $(x_e = 0)$, o problema de emparelhamento de peso máximo em G pode ser modelado matematicamente como o seguinte problema de programação inteira $0-1$:

$$(EPM)\ \text{Maximizar} \sum_{e \in E} c_e x_e \tag{2.1}$$

Sujeito a:

$$\sum_{e \in \delta(v)} x_e \leq\ 1,\ \forall\, v \in V, \tag{2.2}$$

$$x_e \in \{0,1\},\ \forall\, e \in E. \tag{2.3}$$

A restrição (2.2) expressa a definição de emparelhamento, onde garantimos que, para todo vértice $v \in V(G)$, deve existir no máximo uma aresta incidente a ele. A restrição (2.3) estabelece o domínio das variáveis de decisão.

Vamos denotar por $\mathscr{P}(G)$ o politopo de emparelhamento de um grafo G, dado pela

envoltória convexa dos pontos viáveis para EPM, ou seja,

$$\mathscr{P}(G) = \text{conv}\{x \in \{0,1\}^{|E|} : x \text{ satisfaz (2.2)}\}.$$

2.5 Restrições de blossom

Os problemas de Programação Inteira são geralmente de difícil resolução, já que na maioria dos casos o poliedro definido pela envoltória convexa dos pontos viáveis é desconhecido. Isso demanda métodos de solução mais complexos. Por outro lado, quando o poliedro descrito pelas restrições lineares do modelo definem a envoltória convexa das soluções inteiras válidas, o problema inteiro pode ser resolvido como um problema de Programação Linear, ou seja, as restrições de integralidade podem ser ignoradas e a solução ótima fornecida para esse problema relaxado ainda assim será uma solução viável inteira. Neste caso, o problema tem complexidade polinomial.

O modelo EPM (2.1)-(2.3) apresenta variáveis inteiras e sua relaxação linear pode não possuir solução inteira. Considere, por exemplo, o grafo que é um ciclo com três vértices. Claramente, todo emparelhamento tem uma única aresta. Por outro lado, a solução que atribui valor $1/2$ a cada aresta satisfaz (2.2) e leva a um peso total $3/2$ (considerando pesos unitários), mostrando que a solução da relaxação linear não é ótima para o problema 0-1. Assim, em princípio, resolver o modelo EPM demanda métodos mais complexos.

A partir desse exemplo, podemos perceber que as restrições (2.2) não capturam a seguinte propriedade: em qualquer emparelhamento, a quantidade de arestas com ambas as extremidades em um conjunto ímpar de vértices é no máximo (o piso da) metade do tamanho desse conjunto. De fato, cada aresta de um emparelhamento cobre dois vértices. Tal constatação leva às seguintes desigualdades válidas, conhecidas como restrições de *blossom*:

$$\sum_{e \in E(S)} x_e \leq \frac{|S|-1}{2}, \; \forall\, S \subseteq V(G), \; |S| \geq 3 \text{ e ímpar}, \tag{2.4}$$

onde $E(S) = \{(u,v) \in E : u \in S, v \in S\}$ é subconjunto de arestas com ambas as extremidades em S

Na verdade, segundo um importante resultado de Edmonds, obtido em 1965, a inclusão desse conjunto de restrições ao modelo EPM faz com que a relaxação linear tenha solução inteira e, consequentemente, forneça a ótima do problema.

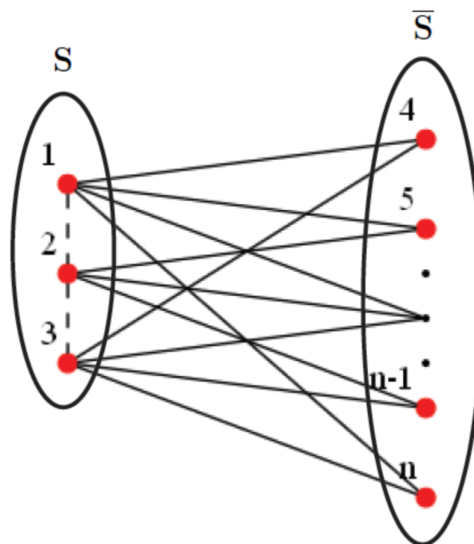
Teorema 2.5.1 ((EDMONDS, 1965a)) *Para qualquer grafo G=(V,E), o politopo* $\mathscr{P}(G)$ *é descrito pelo sistema:*

1. $x_e \geq 0, \forall\, e \in E(G)$
2. $\sum_{e\in\delta(v)} x_e \leq\ 1,\ \forall\, v \in V(G)$
3. $\sum_{e\in E(S)} x_e \leq\ \frac{|S|-1}{2},\ \forall\, S \subseteq V(G),\ |S| \geq 3$ *e ímpar.*

Embora o acréscimo de todas as restrições (2.4) ao modelo EPM faça com que o mesmo possa ser resolvido como um problema de PL, a quantidade de restrições a ser acrescentada é exponencial, tornando impraticável essa abordagem. Porém, para se obter o ótimo, não se precisa necessariamente de todas as restrições definidas pelos subconjuntos ímpares de vértices, e sim aquelas que são suficientes para que a solução do problema linear relaxado seja a ótima para o problema inteiro. Assim, a ideia é acrescentar uma restrição de blossom ao modelo apenas quando necessária. Mais precisamente, adicionamos iterativamente as restrições que violem a solução ótima da relaxação corrente.

Para implementar tal estratégia, vamos reescrever as restrições (2.4) como segue. Primeiro, considere um subconjunto $S \subseteq V(G)$ de vértices. Podemos particionar $V(G)$ em S e $\overline{S} = V(G) \setminus S$ (Ver Figura 5 para uma ilustração). Seja $\delta(S) = \{(u,v) \in E : u \in S, v \in \overline{S}\}$ o subconjunto de arestas com uma extremidade em S e outra em $\overline{S}$.

Figura 5 – $V(G) = S \cup \overline{S}$



Fonte: Elaborada pelo autor.

Queremos mostrar que

$$\sum_{v\in S}\sum_{e\in\delta(v)} x_e \;=\; 2\sum_{e\in E(S)} x_e + \sum_{e\in\delta(S)} x_e. \tag{2.5}$$

Note que, para todo $v \in S$, cada aresta em $\delta(v)$ ou pertence a $E(S)$ ou a $\delta(S)$. Em outros termos,

$$\sum_{e\in\delta(v)} x_e = \sum_{e\in\delta(v)\cap E(S)} x_e + \sum_{e\in\delta(v)\cap\delta(S)} x_e \quad \forall v \in S. \tag{2.6}$$

Somando as equações acima, podemos observar que:

- cada aresta $e = (u,v) \in \delta(S)$, com $v \in S$ e $u \in \overline{S}$, aparece uma vez, na equação referente a v (por exemplo, x_{14} aparece na equação (2.6) relativa ao vértice $1 \in S$);
- cada aresta $e = (u,v) \in E(S)$, com $v \in S$ e $u \in S$, aparece duas vezes: na equação referente a v e naquela referente a u (por exemplo, x_{12} aparece na equação (2.6) relativa a $1 \in S$ assim como naquela correspondente a $2 \in S$).

Dessa forma, a soma das equações (2.6) resulta em (2.5).

Usando (2.5), podemos reescrever (2.4) como:

$$\sum_{v\in S}\sum_{e\in\delta(v)} x_e - \sum_{e\in\delta(S)} x_e \le |S| - 1,\ \forall\, S \subseteq V(G),\ |S| \ge 3 \text{ } e \text{ ímpar}, \tag{2.7}$$

Acrescentando uma variável de folga f_v à restrição (2.2) relacionada a $v \in V$, obtemos:

$$\sum_{e\in\delta(v)} x_e + f_v =\; 1,\ \forall\, v \in V(G). \tag{2.8}$$

Então (2.7) torna-se:

$$\sum_{v\in S}(1 - f_v) - \sum_{e\in\delta(S)} x_e \le |S| - 1,\ \forall\, S \subseteq V(G),\ |S| \ge 3 \text{ } e \text{ ímpar}$$

ou ainda

$$\sum_{v\in S} f_v + \sum_{e\in\delta(S)} x_e \ge 1,\ \forall\, S \subseteq V(G),\ |S| \ge 3 \text{ } e \text{ ímpar}. \tag{2.9}$$

Portanto, (2.9) é uma forma equivalente das restrições de blossom.

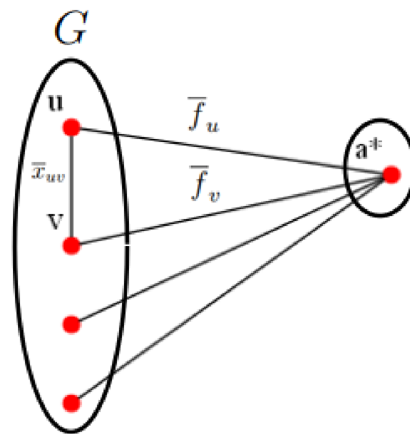
Dada uma solução relaxada $\overline{x} \in \mathbb{R}^{|E|}$ e sendo $\overline{f} \in \mathbb{R}^{|V|}$ o vetor de folgas correspondentes, obtidas segundo (2.8), a questão torna-se verificar (e determinar) se tem algum subconjunto ímpar $S \subseteq V(G)$ tal que $val_{\overline{x}}(S) < 1$, onde

$$val_{\overline{x}}(S) = \sum_{e\in\delta(S)} \overline{x}_e + \sum_{v\in S} \overline{f}_v. \tag{2.10}$$

Isto equivale a encontrar um subconjunto S^* que minimize a expressão (2.10). Se o valor mínimo for ≥ 1, então podemos garantir que não há necessidade de acrescentar mais nenhuma restrição ao modelo, pois todas as restrições de blossom estão satisfeitas. Caso o valor seja < 1, então deve-se acrescentar a restrição de blossom relativa a S^*.

Considere o grafo G^*, ponderado em arestas, obtido de G com o acréscimo de um vértice a^* adjacente a todo vértice de G, ou seja, $V(G^*) = V(G) \cup \{a^*\}$ e $E(G^*) = E(G) \cup \{(a^*, v) : v \in V(G)\}$, e atribuindo-se peso $\overline{x}_e$ a cada aresta $e \in E(G)$ e peso $\overline{f}_v$ a aresta (a^*, v), para cada $v \in V(G)$. Ver Figura 6.

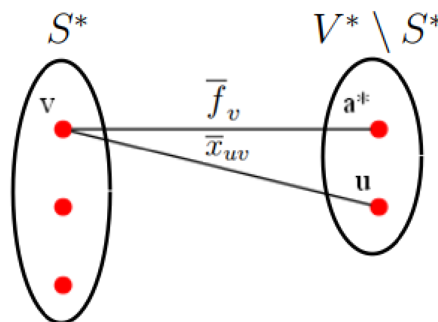
Figura 6 – Grafo G^*



Fonte: Elaborada pelo autor.

Note que $val_{\overline{x}}(S)$ é o peso do corte de arestas em G^* induzido por S. Sendo assim, determinar S^* corresponde a encontrar um corte de arestas de peso mínimo em G^* (induzido por S^*), onde a partição dos vértices que não contém a^* tenha cardinalidade ímpar. Ver Figura 7.

Figura 7 – Corte ímpar mínimo. $|S^*|$ é ímpar.



Fonte: Elaborada pelo autor.

2.6 Problema do T-corte mínimo

O problema do T-Corte Mínimo (ou T-Ímpar Corte Mínimo) tem como entrada um grafo $G^* = (V^*, E^*)$ não orientado, ponderado em arestas, com conjunto de vértices V^*, conjunto de arestas E^* e função de custos $w : E^* \to \mathbb{R}_+$, e um subconjunto de vértices $T \subseteq V^*$ de cardinalidade par. Seu objetivo é encontrar um subconjunto S de vértices, tal que $|S \cap T|$ seja ímpar e o peso do corte $\delta(S)$ seja o mínimo possível (PADBERG; RAO, 1982; RIZZI, 2003). Relembre que $\delta(S)$ é o subconjunto de arestas com uma extremidade em S e a outra em $\overline{S} = V^* \setminus S$. Sendo $|S \cap T|$ ímpar, $\delta(S)$ é chamado um T-corte. Quando $T = V^*$, o problema costuma ser referenciado simplesmente como Corte Ímpar Mínimo.

Mostraremos a seguir que o problema do T-Corte Mímimo modela o problema de separação das restrições de blossom, definido no final da seção anterior. De fato, dado o grafo de entrada $G = (V, E)$ para o problema de emparelhamento de peso máximo, defina o grafo $G^* = (V^*, E^*)$ acrescentando um vértice artificial a^*, adjacente a todos os vértices de G, ou seja, $V^* = V \cup \{a^*\}$ e $E^* = E \cup \{(a^*, v) : v \in V\}$. Caso $|V|$ seja par, defina $T = V$; caso contrário, seja $T = V^*$. Em todo caso, $|T|$ é par. Considere a função $w : E^* \to \mathbb{R}_+$ tal que

$$w_e = \begin{cases} \bar{x}_e, & e \in E, \\ \bar{f}_v = 1 - \sum_{e \in \delta(v)} \bar{x}_e, & e = \{a^*, v\} : v \in V. \end{cases} \tag{2.11}$$

Seja $S \subset V^*$ uma solução do problema de T-corte mínimo definido sobre (G^*, T, w), de modo que $|S \cap T|$ é ímpar. Como T é par, ocorre que $|(V^* \setminus S) \cap T|$ também é ímpar. Vamos mostrar que uma solução S^* para o problema de separação é dada por S (se $a^* \notin S$) ou $V \setminus S = V^* \setminus S$ (se $a^* \in S$). Como $\delta(S) = \delta(V^* \setminus S)$, levando a $\delta(S^*) = \delta(S)$ em qualquer caso, resta argumentar que $|S^*|$ é ímpar. Consideramos os seguintes casos:

- $a^* \notin S$: temos que $S \cap T = S$ e $S^* = S$ e, portanto, $|S^*| = |S \cap T|$ é ímpar;
- $a^* \in S$: sendo $\overline{S} = V \setminus S = V^* \setminus S$, temos que $\overline{S} \cap T = \overline{S}$, $S^* = \overline{S}$ e, logo, $|S^*| = |(V^* \setminus S) \cap T|$ é ímpar.

A seguir, apresentaremos duas abordagens para a resolução do problema de T-Corte Mínimo. A primeira delas consiste em resolver uma formulação de programação inteira. Para sua descrição, por simplicidade, vamos nos restringir ao caso de interesse, onde G^*, T e w são dados como acima. Usamos as seguintes variáveis:

$$y_u = \begin{cases} 1, & \text{se o vértice } u \in S, \\ 0, & \text{caso contrário.} \end{cases}$$

$$z_e = \begin{cases} 1, & \text{se a aresta } e \in \delta(S), \\ 0, & \text{caso contrário.} \end{cases}$$

Assim, obtemos o modelo:

$$(TCIM)\,\text{Minimizar} \sum_{e \in E^*} w_e z_e \tag{2.12}$$

Sujeito a:

$$z_e \geq \; y_u - y_v, \; \forall \; e = \{u, v\} \in E^*, \tag{2.13}$$

$$z_e \geq \; y_v - y_u, \; \forall \; e = \{u, v\} \in E^*, \tag{2.14}$$

$$\sum_{u \in S} y_u = \; 2k + 1, \tag{2.15}$$

$$y_{a^*} = 0, \tag{2.16}$$

$$z_e \leq 1, \; \forall \; e \in E^*, \tag{2.17}$$

$$y_u \in \{0, 1\}, \; \forall \; u \in V^*, \tag{2.18}$$

$$k \in \mathbb{Z}_+. \tag{2.19}$$

As restrições (2.13) e (2.14) estão associadas à determinação das arestas que pertencem e das que não pertencem ao corte da seguinte forma:

1. Se $y_u = 1$ *e* $y_v = 0$, iremos obter em (2.13) que $z_e \geq 1$ e em (2.14) que $z_e \geq -1$, logo $z_e \geq 1$ e, por (2.17), $z_e = 1$, obedecendo à definição do corte;
2. Se $y_u = 0$ *e* $y_v = 1$, iremos obter em (2.13) que $z_e \geq -1$ e em (2.14) que $z_e \geq 1$, logo $z_e \geq 1$; similarmente, por (2.17), $z_e = 1$, obedecendo à definição do corte;
3. Se $y_u = 0$ *e* $y_v = 0$, iremos obter em (2.13) e (2.14) que $z_e \geq 0$ e, como estamos em um processo de minimização, na solução ótima teremos $z_e = 0$, obedecendo à definição do corte;
4. Se $y_u = 1$ *e* $y_v = 1$, iremos obter em (2.13) e (2.14) que $z_e \geq 0$; analogamente obteremos $z_e = 0$ na solução ótima, em acordo com a definição do corte.

Percebemos então que a correção na escolha das arestas para a solução está sendo assegurada por essas restrições.

As restrições (2.15) e (2.19) garantem que o subconjunto S terá pelo menos um vértice e que sua cardinalidade será ímpar. Com a restrição (2.16), garantimos que o conjunto

$\overline{S}$ conterá pelo menos o vértice artificial a^*, assim dando possibilidade para que as arestas que apresentam como custo o valor das folgas da solução do problema (EPM) possam ser escolhidas, levando ao cômputo da expressão (2.10) e, consequentemente, à avaliação da satisfatibilidade das inequações (2.9).

A segunda abordagem ao problema de T-corte mínimo é através do algoritmo proposto em (RIZZI, 2003), que transcrevemos abaixo. Para tal, dada uma entrada (G,T,w) para o problema e um subconjunto S dos vértices de G, denotamos por T_S o subconjunto $T \setminus S$, por G_S o subgrafo obtido de G contraindo os vértices em S e por w_S a função w restrita às arestas em G_S. Precisamente, o grafo G_S é obtido substituindo-se o conjunto S por um único vértice s, que será adjacente a cada vértice em $V(G) \setminus S$ adjacente a algum vértice em S, ou seja, $V(G_S) = (V(G) \setminus S) \cup \{s\}$ e $E(G_S) = E(V(G) \setminus S) \cup \{\{s,v\} : v \in \bigcup_{u \in S} N_G(u)\}$. Já a função w_S é tal que

$$w_S(e) = \begin{cases} w(e), & e \in E(G), \\ w(\delta(v) \cap S), & e = \{s,v\} \in E(G_S) \setminus E(G), \end{cases}$$

onde $w(E') = \sum_{e \in E'} w(e)$, para todo $E' \subseteq E$.

O algoritmo calcula recursivamente (u,v)-cortes mínimos em grafos obtidos de G por contrações, veja o Algoritmo 1.

A corretude desse procedimento é formalmente demonstrada em (RIZZI, 2003). Além disso, o autor mostra que o número de chamadas recursivas é pelo menos $|T|/2$ e no máximo $|T|-1$. Em outras palavras, o algoritmo resolve no máximo $|T|-1$ problemas de fluxo máximo/corte mínimo, em grafos que vão sendo reduzidos em tamanho.

2.7 Implementações

Conciliar a base teórica em otimização matemática com a prática foi o ponto mais forte abordado neste trabalho. Especialmente para alunos em formação, esse pode constituir um importante desafio, que precisa ser vivenciado e vencido. Mesmo que se tenha disponível um resolvedor, questões de diversos níveis de complexidade precisam ser enfrentadas, entre elas: a escolha do modelo matemático a ser implementado, a escolha do método (ou combinação de métodos) a ser empregado na resolução e a calibração de seus parâmetros e, não menos importante, a identificação das formas que o resolvedor disponibiliza para a implementação do método escolhido e a opção por uma ou alguma delas.

Algoritmo 1: Corte ímpar mínimo

Entrada: (G, w, T)

$\triangleright$ G - Grafo, w - função de pesos nas arestas e T - subconjunto par de vértices

Saída: S tal que $\delta(S)$ é T-corte mínimo

```
1  início
2  |  se T = ∅ então
3  |  |  retorne ∅
4  |  senão
5  |  |  Sejam s,t ∈ T, s ≠ t
6  |  |  Determine S' ⊆ V(G) tal que δ(S') seja (s,t)-Corte Mínimo
7  |  |  se |S' ∩ T| é ímpar então
8  |  |  |  S ← {s,t}
9  |  |  |  S'' ← Corte ímpar mínimo(G_S, w_S, T_S)
10 |  |  senão
11 |  |  |  S ← S'
12 |  |  |  S' ← Corte ímpar mínimo(G_S, w_S, T_S)
13 |  |  |  S'' ← Corte ímpar mínimo(G_S̄, w_S̄, T_S̄)
14 |  |  fim
15 |  |  se w(δ(S'')) < w(δ(S')) então
16 |  |  |  retorne S''
17 |  |  senão
18 |  |  |  retorne S'
19 |  |  fim
20 |  fim
21 fim
```

Para nossas implementações, optamos pelo uso do software de otimização CPLEX, pois é amplamente utilizado em trabalhos científicos e apresenta uma versão acadêmica gratuita.

O desenvolvimento e implementação realizados neste trabalho, bem como a análise dos resultados, serviram para aprendizagem do uso dessa ferramenta de otimização e uma compreensão do quanto alterações, mesmo que pequenas, podem alterar drasticamente o tempo de solução de um problema. Compartilhamos aqui essa experiência, que esperamos possa ajudar outros iniciantes na área.

Nesta seção, relatamos algumas experiências computacionais com o problema de emparelhamento de peso máximo. A partir do estudo teórico sobre o mesmo, descrito nas seções

anteriores, realizamos os testes práticos, visando observar mudanças na resolução do problema e o aperfeiçoamento na programação com a API do CPLEX. As implementações foram realizadas segundo três versões (que serão detalhadas à frente):

1. Resolução direta do modelo EPM;
2. Resolução iterativa do modelo EPM com a inclusão de restrições de blossom:
   2.1. via modelo inteiro - com variáveis inteiras;
   2.2. via modelo relaxado - com variáveis contínuas.

2.7.1 Resolução direta do modelo

O CPLEX é um *software* computacional que implementa e disponibiliza inúmeros métodos e técnicas para a resolução de problemas lineares e quadráticos, contínuos e inteiros. Desenvolvido e aprimorado no decorrer de anos, o *software* oferece periodicamente atualizações de seus métodos, diversificando ainda mais o seu conteúdo e eficiência. Contudo o CPLEX não deixa especificado exatamente a forma através da qual é tratado um modelo, seja ele inteiro ou contínuo, dando certa (muitas vezes bastante) liberdade ao usuário em como utilizá-lo. Por outro lado, dispõe de uma documentação acessível, para que os usuários sejam capazes de entender e usar suas potencialidades, por meio de uma linguagem de programação para a qual ele forneça suporte. No caso deste trabalho, a linguagem C_{++} foi a utilizada.

O modelo EPM foi montado e resolvido através das funções contidas na API do CPLEX. As partes principais do código em C_{++}, que permitem sua reprodução quase direta no computador, podem ser vistas abaixo.

Código-fonte 1 – Versão 2.1

```
1  #include<iostream>
2  #include<ilcplex/ilocplex.h>
3  ILOSTLBEGIN
4  using namespace std;
5  int main(){
6      \\Leitura da instancia
7      \\Criacao da Matriz de Adjacencias (Ma)
8      \\Criacao da Matriz de Custos (Cst)
9  
10     IloEnv env;\\Definindo o ambiente de programacao
11 
```

```
12    \\ DECLARACAO DAS ESTRUTURAS DE PROGRAMACAO
13    IloModel modelo(env); \\Modelo de programacao matematica
14    IloRangeArray rest(env); \\Restricoes do modelo
15    IloObjective obj(env); \\Funcao objetivo
16    IloExpr expr_obj(env), expr(env); \\Expressoes lineares uteis na
        definicao de restricoes e da funcao objetivo
17
18    \\ DECLARACAO DAS VARIAVEIS DE DECISAO BINARIAS
19    IloArray<IloNumVarArray> x(env, n);
20    for (int i = 0; i < n; i++)
21        x[i] = IloNumVarArray(env, n, 0, 1, ILOINT);\\Variaveis inteiras
            0 ou 1
22
23    \\ CRIACAO DO MODEL0 EPM
24    for (int i = 0; i < n; i++) {
25        for (int j = i + 1; j < n; j++) {
26            if (Ma[i][j] == 1) {
27                expr_obj += Cst[i][j] * x[i][j];
28                modelo.add(x[i][j]);
29            }
30        }
31    }
32    obj = IloMaximize(env, expr_obj); \\Sentido de otimizacao
33    modelo.add(obj);
34    for (int i = 0; i < n; i++) {
35        for (int j = i + 1; j < n; j++) {
36            if (Ma[i][j] == 1)
37                expr += x[i][j];
38        }
39        for (int j = i - 1; j >= 0; j--) {
40            if (Ma[j][i] == 1)
41                expr += x[j][i];
42        }
43        rest.add(expr <= 1);
44        expr.clear();
45    }
46    modelo.add(rest);
47
48    \\ SOLUCAO DO PROBLEMA DE EMPARELHAMENTO DE PESO MAXIMO
```

```
49      IloCplex  cplex(modelo);
50      cplex.solve();
51
52      \\Inquisicao da solucao obtida.
53
54      env.end();
55  }
56  return 0;
```

O código acima ilustra como foi realizada a construção do modelo EPM com a API do CPLEX. O primeiro passo a se destacar (linha 10) é declaração do ambiente, que armazenará as estruturas básicas para a construção e resolução do modelo (linhas 13-21). Em particular, nas linhas 19-21, encontra-se a declaração de uma variável binária para cada possível aresta, porém apenas aquelas relacionadas a arestas efetivamente presentes no grafo são adicionadas ao modelo (linha 28). A opção por declarar todas essas variáveis deve-se à facilidade de indexação. Observe na linha 21 os limites inferior (0) e superior (1) de cada variável x, declarada como inteira (*ILOINT*), ou seja, cada variável assumirá apenas um dos valores 0 ou 1.

Entre as linhas 24-31, construímos a expressão da função objetivo e, posteriormente, na linha 32, indicamos o sentido de otimização, acrescido da expressão. Definimos assim completamente a função objetivo (2.1), adicionada a seguir ao modelo na linha 33.

Em seguida, nas linhas 34-45, há a formação das restrições (2.2), que definem um emparelhamento, depois acrescentadas ao modelo (linha 46).

Por último, na linha 50, encontra-se a chamada ao método *solve* para a solução do modelo 0-1. As linhas seguintes são destinadas à investigação do resultado obtido: se a solução obtida é de fato ótima (*cplex.getStatus()*), qual o valor da função objetivo (*cplex.getObjValue()*) das variáveis (*cplex.getValues()*).

2.7.2 Resolução iterativa do modelo com restrições de blossom

Nesta segunda versão, a estrutura do modelo inicial é a mesma da versão anterior, sendo construída de forma semelhante. Todavia, mais informações serão fornecidas ao CPLEX, com o intuito de interferir no processo de solução. Em nosso caso, essas informações adicionais referem-se às restrições de blossom.

Pelo Teorema 2.5.1, se tais restrições fossem todas adicionadas ao modelo, podería-

mos descartar a integralidade das variáveis e obter a solução ótima do problema apenas com a resolução da relaxação linear. No entanto, como o número dessas restrições é exponencial (no número de vértices), essa estratégia torna-se impraticável. Uma alternativa é acrescentar iterativamente as restrições de blossom ao modelo, quando violadas pela solução da relaxação corrente. Como vimos, a restrição mais violada, caso exista uma, pode ser identificada pela solução do modelo (2.12)-(2.19) ou ainda pelo Algoritmo 1.

Em nossa implementação, optamos por usar o modelo (2.12)-(2.19) para a separação das desigualdades de blossom. Essa alternativa nos permitirá ilustrar o uso simultâneo de dois modelos no resolvedor, trabalhando conjuntamente para resolver um problema. Tal cenário é bastante comum em métodos de programação inteira, como planos-de-corte, geração de colunas, *branch-and-cut-and-price*, relaxação lagrangiana, decomposição de Benders.

Com a inclusão das restrições de blossom, podemos manter ou liberar a integralidade das variáveis, levando às duas variações enunciadas para essa segunda versão. No primeiro caso, aplicaremos o método *branch-and-cut*. No segundo, usaremos apenas o método de planos-de-corte (sem a realização de particionamento), incluindo restrições de blossom até que a solução relaxada se torne inteira.

A implementação da primeira variação (*brach-and-cut*) foi desenvolvida usando *callbacks*, que são ferramentas disponibilizadas pelo resolvedor para possibilitar o monitoramento e orientação de seu funcionamento. Em particular, os *callbaks* permitem que o código do usuário possa ser executado regularmente durante uma otimização ou durante uma sessão de ajuste. A utilização de um *callback* demanda que o modelo apresente variáveis inteiras, para que os métodos do usuário sejam incorporados ao processo de *Branch and Bound*(B&B) do resolvedor.

Especificamente, utilizamos a macro *ILOUSERCUTCALLBACKn*, que proporciona e facilita a inclusão de restrições enquanto o modelo está sendo resolvido. Observe que, se o particionamento (*braching*) não for realizado enquanto houver restrição de blossom violada, o problema será resolvido no nó raiz, em consequência do Teorema 2.5.1.

Abaixo encontra-se o código para a implementação do algoritmo de solução usando a macro *ILOUSERCUTCALLBACK*:

Código-fonte 2 – Versão 2.2.1

```
1 #include<iostream>
2 #include<ilcplex/ilocplex.h>
3 ILOSTLBEGIN
```

```
4   using namespace std;
5   #define eps2 0.9999;
6
7     ILOUSERCUTCALLBACK2(Meu_Corte, IloArray<IloNumVarArray>&,x, int**,Ma){
8     \\(x) armazena a solucao da relaxacao corrente; (Ma) contem a matriz de
          adjacencia de (G)
9
10    \\Criacao da nova matriz de adjacencia(Ma2), englobando o vertice
         artificial(a*)
11    \\Criacao da matriz de custos (Cst2) com os valores de (w_e) dados por
         (2.11)
12
13    IloEnv env2;
14
15    \\ DECLARACAO DAS ESTRUTURAS DE PROGRAMACAO
16    IloModel modelo2(env2);
17    IloRangeArray Rest(env2);
18    IloObjective obj2(env2);
19
20    \\ DECLARACAO DAS VARIAVEIS DE DECISAO
21    IloIntVar k(env2);
22    IloBoolVarArray y(env2, n+1);
23    IloArray<IloNumVarArray> z(env2, n+1);
24    for(int i = 0; i < n+1; i++)
25        z[i] = IloNumVarArray(env2, n+1, 0, 1);
26
27    \\ CRIACAO DO MODELO CIM
28    CORTE_IMPAR_MIN(Ma2, Cst2, env2, modelo2, Rest, obj2, y, z, k);
29
30    \\ SOLUCAO DO PROBLEMA DE CORTE IMPAR MINIMO
31    IloCplex cplex2(modelo2);
32    cplex2.solve();
33    IloArray<IloNumArray> val_z(env2, n+1);
34    for(int i = 0; i < n+1; i++){
35        val_z[i] = IloNumArray(env2, n);
36        cplex2.getValues(val_z[i], z[i]);
37    }
38
39    \\Calculo do peso do corte da vizinhanca de (S) representado por (z),
```

```
        ou seja, o valor da expressao (2.10), que sera armazenado na
        variavel (soma)
40
41    if (soma < eps2){
42         IloNumArray val_y(env2, n+1);
43         cplex2.getValues(val_y, y);
44
45         IloExpr expr_corte(getEnv());
46        \\(expr_corte) armazena a expressao formada pelo somatorio das
              variaveis (x) indexadas por (E(S)), ou seja, de todas as
              variaveis x[i][j] tais que y[i] == 1 e y[j] == 1
47
48         int cardinalidade = 0;
49         \\A variavel (cardinalidade) recebe a cardinalidade do conjunto (S)
              , ou seja, o somatorio dos valores otimos de (y)
50
51         add(IloRange(getEnv(), expr_corte , (cardinalidade - 1) / 2));
52         expr_corte.clear();
53    }
54    env2.end();
55 }
56
57 int main(){
58      \\Construcao do modelo (EPM) - linhas 6-46 da Versao 2.1
59
60      \\ SOLUCAO DO PROBLEMA DE EMPARELHAMENTO DE PESO MAXIMO
61      IloCplex cplex(env);
62      cplex.use(Meu_Corte(env, x, Ma));
63      cplex.extract(modelo);
64      cplex.solve();
65
66      \\Inquisicao da solucao obtida.
67
68      env.end();
69 }
70 return 0;
```

O código acima segue a mesma organização daquele referente à Versão 2.1, mas

vale destacar a forma como foi estruturada a macro *ILOUSERCUTCALLBACKn*. Na linha 7, notamos que os parâmetros usados na definição da macro seguem uma regra um pouco diferente: o primeiro parâmetro é sempre o nome do método, que é usado para a chamada do *callback* durante o processo de otimização, como mostra a linha 62; os outros parâmetros vem aos pares, com os dois elementos separados por vírgula - o primeiro é um tipo e o segundo, o nome do objeto/variável daquele tipo, podendo tais tipos serem definidos a partir de classes do CPLEX ou não. Como podemos notar, apenas 2 (pares de) parâmetros foram passados para a macro, por isso ela ficou definida como *ILOUSERCUTCALLBACK2*. Em geral, caso haja n parâmetros e seja um valor de no máximo 7, o valor de n é acrescentado ao final do termo *ILOUSERCUTCALLBACK*.

Como vimos anteriormente na Versão 2.1, definimos estruturas de dados para a formação do modelo CIM (linhas 16-25), incluindo as variáveis de decisão (k, y e z) e seus respectivos domínios (inteiro, binário e real, respectivamente). Após a criação (através da chamada ao procedimento CORTE-IMPAR-MIN na linha 28) e resolução (linha 33) do modelo TCIM, capturamos o valor das variáveis da solução (linhas 34-37) e calculamos o valor da expressão (2.10), conforme indicado na linha 39. Caso este seja menor que 1 (na verdade usamos 0.9999, por questões de precisão numérica), a restrição de blossom correspondente ao corte é construída, a partir do valor das variáveis y (linhas 34-49) e adicionda ao modelo (linha 51) como um plano-de-corte.

Observamos a diferença, em relação à Versão 2.1, na definição do objeto *IloCplex* e a atribuição do modelo a ele. Agora, tais tarefas são realizadas em dois comandos separados (linha 62 e linha 64), entre os quais encontra-se a instrução para usar a *callback* anteriormente definida (linha 63).

A implementação da segunda variação da Versão 2.2 (planos-de-corte) pode ser vista abaixo:

Código-fonte 3 – Versão 2.2.2

```
1  #include<iostream>
2  #include<ilcplex/ilocplex.h>
3  ILOSTLBEGIN
4  using namespace std;
5
6  int main(){
7      \\Construcao do modelo de (EPM) - linhas 6-46 da Versao 2.1,
           porem declarando as variaveis (x) como continuas
```

```
8           \\Construcao do modelo de (CIM) - linhas 11-29 da Versao 2.2.1
9
10          bool ok;
11          do{
12              ok = true;
13              cplex.solve();
14              IloArray<IloNumArray> val_x(env, n);
15              for(int i = 0; i < n; i++){
16                  val_x[i] = IloNumArray(env, n);
17                  cplex.getValues(val_x[i], x[i]);
18              }
19              for(int i = 0; i < n-1; i++)
20                  for(int j = i+1; j < n; j++)
21                      if (val_x[i][j] < 0.9999 && val_x[i][j] > 0.0001) {
22                          ok = false; \\solucao fracionaria
23                          break;
24                      }
25
26              if(!ok)
27                  \\Funcao FAZER_CORTE forma a nova restricao e a adiciona
                        ao modelo de (EPM) - linhas 32-54 da Versao 2.2.1
28                  FAZER_CORTE(env, env2, rest, modelo, cplex, cplex2,
                        modelo2, obj2, x, z, y, Ma, Ma2, Cst2);
29          }while(!ok);
30
31         \\Inquisicao da solucao obtida.
32
33          env.end();
34          env2.end();
35      }
36      return 0;
```

Nessa segunda variação, as variáveis do modelo EPM foram declaradas como contínuas. Criou-se uma função FAZER_CORTE para gerar os cortes (restrições de blossom), enquanto a solução ótima não seja inteira. A ideia é observar diferenças que podem ocorrer no tempo de processamento com uma implementação menos “dependente” dos recursos do CPLEX, já que nesta segunda variação não temos a utilização do *B&B*.

2.8 Experimentos computacionais

Para os testes computacionais com as três implementações descritas acima, foram geradas 18 instâncias aleatoriamente, subdivididas em:

- 09 instâncias com $n = 50$ vértices;
- 09 instâncias com $n = 100$ vértices.

A quantidade de arestas foi definida pela fórmula $\frac{n*(n-1)}{2} * p$, onde p indica a probabilidade de existência de aresta no grafo, dada pelos seguintes valores:

- 0.25 ou 25%;
- 0.50 ou 50%;
- 0.75 ou 75%.

Para cada possível par (n, p), foram geradas 3 instâncias. Os pesos para as arestas foram gerados aleatoriamente com valores reais entre 0 e 20. Cada uma delas é nomeada como match-*n*-*p*(*i*), de modo a identificar que se trata da *i*-ésima $(i = 1, 2, 3)$ instância relativa ao par (n, p).

Para todas as implementações e testes realizados neste trabalho, utilizamos a linguagem C_{++}, com compilador Visual C_{++}, com o auxílio do resolvedor `IBM ILOG CPLEX Optimization Studio 12.7.1.0`. A máquina utilizada possui o sistema operacional Windows® 10 Home Sigle Language 64 Bits com 4GB de memória RAM, processador Intel® Core™ i5-3317U CPU @ 1.70GHz. Nos capítulos seguintes serão vistos e analisados mais experimentos executados no mesmo ambiente.

Um resumo dos resultados dos experimentos com as três versões pode ser visto na Tabela 1. Além do nome da instância e o valor de sua solução ótima, apresentamos o tempo (em segundos) demandado por cada uma das versões para a resolução do problema.

Observamos pela Tabela 1 que os três métodos encontram as soluções ótimas rapidamente, e os tempos médios de resolução ficaram próximos, com leve vantagem para a Versão 2.1. Por outro lado, analisando cada instância de forma independente, percebemos que a Versão 2.2.2 se mostrou mais efetiva na maioria dos testes. Relembre que esta versão baseia-se na garantia, dada pelo Teorema 2.5.1, de que o problema pode ser resolvido através da relaxação linear do modelo com as restrições de blossom. Esse resultado ilustra como o conhecimento de propriedades específicas do problema podem ser usadas para fortalecer uma formulação e, por conseguinte, fornecer ao resolvedor desigualdades válidas particulares que podem ser mais efetivas que os cortes gerais por ele derivados, dando a possibilidade de uma maior eficiência no processo de solução.

Tabela 1 – Resultados

Instâncias	Solução Ótima	Versão 2.1 Tempo	Versão 2.2.1 Tempo	Versão 2.2.2 Tempo
match-50-025(1)	329,16	**0,128**	0,223	0,781
match-50-025(2)	293,651	**0,162**	0,224	0,340
match-50-025(3)	323,305	**0,147**	0,215	0,894
match-50-050(1)	343,917	0,087	0,101	**0,012**
match-50-050(2)	371,653	0,091	0,086	**0,017**
match-50-050(3)	356,404	0,096	0,111	**0,013**
match-50-075(1)	384,062	0,123	0,106	**0,014**
match-50-075(2)	390,819	0,126	0,104	**0,014**
match-50-075(3)	405,101	0,147	0,108	**0,014**
match-100-025(1)	722,544	0,146	0,128	**0,057**
match-100-025(2)	752,026	0,123	0,133	**0,057**
match-100-025(3)	720,036	**0,272**	0,383	0,534
match-100-050(1)	809,205	0,198	0,192	**0,064**
match-100-050(2)	803,427	0,413	0,544	**0,300**
match-100-050(3)	821,03	0,233	0,177	**0,080**
match-100-075(1)	848,595	0,250	0,255	**0,083**
match-100-075(2)	832,517	0,289	0,254	**0,070**
match-100-075(3)	859,986	0,265	0,242	**0,075**
Média	-	0,183	0,199	0,190

Versão 2.1: Resolução direta do modelo EPM;
Versão 2.2.1: Resolução iterativa do modelo EPM com a inclusão de restrições de blossom: via modelo inteiro - com variáveis inteiras;
Versão 2.2.2: Resolução iterativa do modelo EPM com a inclusão de restrições de blossom: via modelo relaxado - com variáveis contínuas.

3 EMPARELHAMENTO DE PESO MÁXIMO COM RESTRIÇÕES DE CONFLITO

Nos últimos anos, alguns problemas clássicos de otimização em grafos têm sido revisitados com a inclusão de requisitos adicionais que levam a variações na estrutura do modelo original. Essas variações são construídas ao redor de restrições disjuntivas binárias em certos pares de elementos do grafo (vértices ou arestas). As restrições disjuntivas são classificadas como:

- Restrições disjuntivas positivas (restrições de imposição): impõem que pelo menos um elemento de cada par deverá compor uma solução viável.
- Restrições disjuntivas negativas (restrições de conflito): expressam uma incompatibilidade ou conflito entre os elementos de cada par, exigindo que no máximo um deles poderá compor uma solução viável.

Neste capítulo, iremos definir o problema de emparelhamento de peso máximo acrescido de restrições de conflito, apresentar sua complexidade computacional, um modelo de programação matemática para o problema e sua equivalência a um problema de conjunto independente. Iniciamos comentando sobre a inclusão e os efeitos gerados pelas restrições de conflitos ou restrições de imposição a problemas conhecidos de otimização combinatória.

3.1 Restrições disjuntivas

A introdução de restrições disjuntivas, positivas ou negativas, em problemas clássicos de otimização em grafos tem sido considerada em vários trabalhos recentes. A maior parte dos resultados apresentados diz respeito à complexidade computacional dos problemas originados.

O problema da árvore geradora com conflitos foi introduzido em 2009 (DARMANN *et al.*, 2009). Nesse trabalho, mostra-se que o problema se mantém polinomial quando os pares conflitantes não têm interseção, mas que se torna NP-Difícil, mesmo quando, em qualquer trio de pares conflitantes, pelo menos um deles não tem interseção com os outros dois.

Mais resultados de complexidade podem ser vistos em (ZHANG *et al.*, 2011), que também apresenta algumas heurísticas para o problema e experimentos computacionais preliminares. Métodos de solução mais elaborados, através de uma abordagem via *branch-and-cut*, foram propostos em (SAMER; URRUTIA, 2015). Esses resultados computacionais foram melhorados recentemente, a partir de uma formulação de programação inteira baseada nas restrições de Miller-Tucker-Zemlin (MILLER *et al.*, 1960).

Restrições de conflito também foram consideradas em problemas sobre grafos direcionados como, por exemplo, fluxo máximo. Em (PFERSCHY; SCHAUER, 2011; PFERSCHY; SCHAUER, 2013), mostra-se que o problema passa a ser NP-Difícil, mesmo quando os pares conflitantes são disjuntos e a rede de entrada (através da qual se envia o fluxo) é formada por caminhos (direcionados) disjuntos, de tamanho 3, entre a origem e o destino.

O estudo de problemas com restrições de imposição aparece com menor frequência na literatura. Para o caso de árvore geradora, os resultados de complexidade são similares àqueles obtidos com as restrições de conflitos: o problema continua polinomial, se os pares conflitantes não possuem interseção, e passa a ser NP-Difícil, mesmo quando, em cada três pares conflitantes, pelo menos um não tem interseção com os demais (DARMANN *et al.*, 2011). Já o problema de fluxo máximo com restrições de imposição é polinomial, caso não seja considerada a integralidade sobre o fluxo, e fortemente NP-Difícil, caso contrário (PFERSCHY; SCHAUER, 2011; PFERSCHY; SCHAUER, 2013).

3.2 Definição do problema

As restrições de conflito são usualmente descritas por um grafo, chamado grafo de conflitos, cujos vértices são os elementos envolvidos nas restrições e cujas arestas são definidas pelos pares conflitantes. Assim, para o problema de emparelhamento com conflitos em um grafo G, o grafo de conflitos G_c é tal que $V(G_c) = E(G)$ e $E(G_c)$ é composto pelos pares de arestas conflitantes, ou seja, que não podem figurar simultaneamente em um emparelhamento.

Precisamente, o problema do Emparelhamento de Peso Máximo com Restrições de Conflito (EPMRC) é definido a partir de um grafo $G = (V,E)$, com conjunto de vértices V, conjunto de arestas E e uma função de pesos $c : E \to \mathbb{R}_+$, juntamente com um grafo de conflitos $G_c = (E,E_c)$. Deseja-se encontrar um emparelhamento de peso máximo em G, dentre aqueles que não contêm quaisquer duas arestas de G que definem uma aresta em E_c.

Tomemos como exemplo o grafo $G = (V,E)$ da Figura 4, tal que

$$V(G) = \{a,b,c,d,e,f,g,h\}$$

e

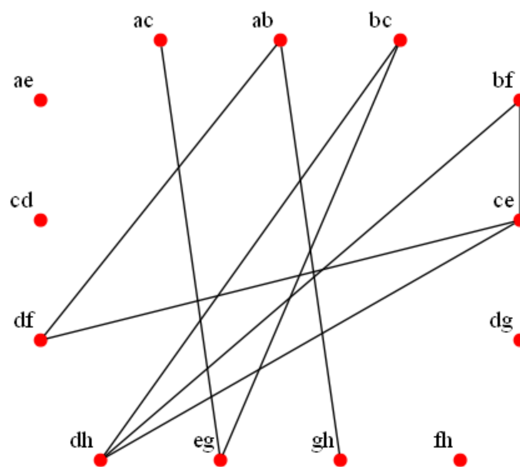
$$E(G) = \{(a,b),(a,c),(a,e),(b,c),(b,f),(c,d),(c,e),(d,f),(d,g),(d,h),(e,g),(f,h),(g,h)\}.$$

Suponha que os pares de arestas conflitantes formam o conjunto

$$E_c = \{[(a,b),(d,f)],[(a,c),(e,g)],[(b,c),(e,g)],[(d,h),(c,e)],[(b,f),(c,e)],[(b,f),(d,h)],$$
$$[(a,b),(g,h)],[(b,c),(d,h)],[(d,f),(c,e)]\},$$

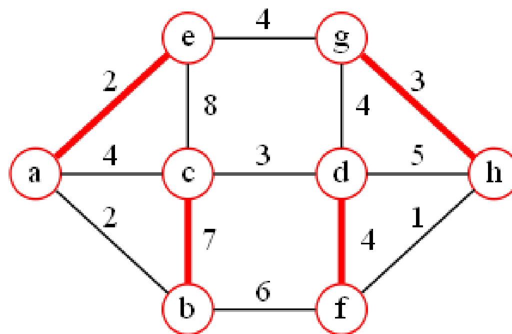
de modo que o grafo de conflitos $G_c = (E, E_c)$ é aquele representado na Figura 8. Note que o emparelhamento máximo apresentado na Figura 4 não respeita os conflitos da Figura 8, deixando de ser viável. Agora, uma solução para EPMRC, apresentada na Figura 9, é $\{(a,e),(b,c),(d,f),(g,h)\}$, com peso ótimo de 16 unidades.

Figura 8 – Grafo $G_c = (E, E_c)$



Fonte: Elaborada pelo autor.

Figura 9 – Emparelhamento de Peso Máximo com Restrições de Conflito



Fonte: Elaborada pelo autor.

A inclusão de restrições de conflito no problema de emparelhamento de peso máximo foi considerada em (DARMANN *et al.*, 2011), onde se prova que o mesmo torna-se fortemente NP-Difícil, ainda que cada componente conexa do grafo de conflitos seja uma simples aresta. O resultado vale mesmo que os pesos sejam todos unitários.

3.3 Modelo matemático

Considere o grafo $G = (V,E)$, com custo $c_e \in \mathbb{R}_+$ associado a cada aresta $e \in E$, e o grafo de conflitos $G_c = (E,E_c)$. A seguir apresentamos um modelo matemático de programação inteira $0-1$, semelhante ao modelo EPM, usando a variável binária x_e para indicar a escolha da aresta na solução ($x_e = 1$) ou não ($x_e = 0$).

$$(EPMRC)\ \text{Maximizar} \sum_{e \in E} c_e x_e \tag{3.1}$$

Sujeito a:

$$\sum_{e \in \delta(v)} x_e \leq\ 1,\ \forall\ v \in V, \tag{3.2}$$

$$x_e + x_{e'} \leq\ 1,\ \forall\ \{e,\ e'\} \in E_c \tag{3.3}$$

$$x_e \in \{0,1\},\ \forall\ e \in E. \tag{3.4}$$

As restrições (3.2) representa a definição de emparelhamento, onde garantimos que, para todo vértice $v \in V(G)$, deve existir no máximo uma aresta incidente a ele. As restrições (3.3) asseguram à obediência aos conflitos via desigualdades que garantem que, para cada par de arestas em E_c, a escolha deve ser de no máximo uma para compor a solução. As restrições 3.4 definem o domínio das variáveis de decisão.

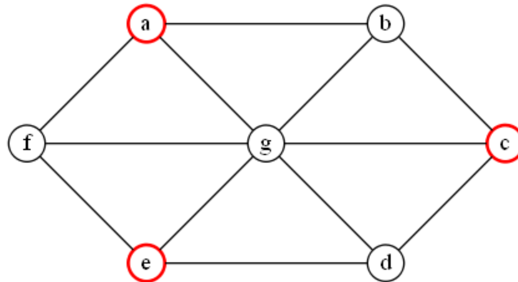
Podemos observar que as restrições de blossom, vistas no Capítulo 2, continuam como desigualdades válidas para o EPMRC, entretanto estas não são mais suficientes para definir o politopo do problema. Em outras palavras, não dá para utilizar somente a relaxação linear como método de resolução.

3.4 Equivalência com conjunto independente máximo

Seja um grafo $G = (V,E)$ e $w : V \rightarrow \mathbb{R}_+$ uma função de pesos associada aos vértices de G. Um conjunto independente em G é um subconjunto $V_I \subseteq V(G)$ tal que $E(V_I) = \emptyset$, ou seja, não existe nenhuma aresta $e \in E(G)$ entre qualquer par de vértices de V_I. O peso de V_I é $w(V_I) = \sum_{v \in V_I} w(v)$. Um conjunto independente V_I é dito máximo se não existe um outro conjunto independente $V_I' \subseteq V(G)$ tal que $|V_I'| > |V_I|$. E é dito um conjunto independente de peso máximo se não existe um outro conjunto independente $V_I' \subseteq V(G)$ tal que $w(V_I') > w(V_I)$. Na Figura 10, observamos um grafo e, em destaque, um de seus conjuntos independentes máximos,

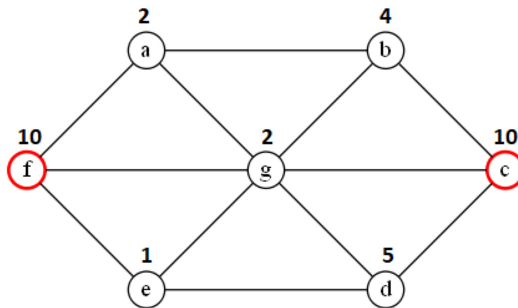
dado por $V_{I_{max}} = \{a, c, e\}$. Na Figura 11, temos um grafo ponderado em vértices, destacando um conjunto independente de peso máximo, definido pelos vértices $\{c, f\}$.

Figura 10 – Conjunto Independente Máximo



Fonte: Elaborada pelo autor.

Figura 11 – Conjunto Independente de Peso Máximo

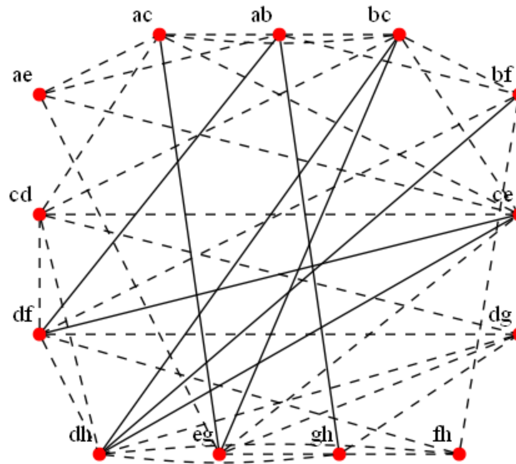


Fonte: Elaborada pelo autor.

Podemos transformar o problema EPMRC em um problema de conjunto independente de peso máximo. Para ilustrar essa transformação, considere o grafo $G = (V, E)$ da Figura 3 e o grafo de conflitos G_c da Figura 8, que definem uma entrada para o problema EPMRC. Vamos gerar um grafo de conflitos estendido, a ser denotado por G_{c*}, que inclui não apenas os conflitos iniciais, como também os "conflitos" entre arestas adjacentes em G (pela definição de emparelhamento, elas não podem ser ambas escolhidas). Podemos ver na Figura 12 o grafo G_{c*}, que apresenta o mesmo conjunto de vértices de G_c e seu conjunto de arestas é definido pelas arestas de G_c (linhas retas) juntamente com uma aresta $\{e, e'\}$, para cada par de arestas $e, e' \in E(G)$ que sejam adjacentes em G (linhas tracejadas).

Como cada vértice do grafo G_{c*} representa uma aresta em G e cada aresta de G_{c*} representa um conflito em G_c ou uma adjacência de arestas em G, podemos observar que um conjunto independente em G_{c*} é um emparelhamento em G que respeita os conflitos em G_c, e

Figura 12 – Grafo de conflitos estendido G_{c*}: G_c com inclusão de arestas entre arestas adjacentes de G



Fonte: Elaborada pelo autor.

vice-versa. Logo, ponderando os vértices de G_{c*} com os pesos das arestas de G, o problema EPMRC equivale a encontrar um conjunto independente de peso máximo em G_{c*}. Em outras palavras, um problema pode ser convertido.

Tal equivalência pode, por exemplo, ser usada como estratégia para fortalecer a formulação (3.1)-(3.4). De fato, toda desigualdade válida para o estudado politopo de conjunto independente pode ser transformada em desigualdade válida para o problema EPMRC. Por exemplo, para a instância da Figura 12, as restrições de conflito

$$x_{df} + x_{ce} \leq 1 \quad \text{e} \quad x_{dh} + x_{ce} \leq 1$$

podem ser substituídas pela restrição

$$x_{df} + x_{dh} + x_{ce} \leq 1,$$

fortalecendo a formulação.

4 RELAXAÇÃO LAGRANGIANA

Em matemática, por definição, um limitante superior (respectivamente, inferior) para uma família de valores é um valor k^* que é maior (respectivamente, menor) ou igual a qualquer valor da família. Em problemas onde se deseja encontrar um ponto x^* em um conjunto X que maximize (ou minimize) uma função f, a existência de bons limitantes superiores e inferiores para $\{f(x) : x \in X'\}$, com $X' \subseteq X$, permite fornecer uma aproximação para $f(x^*)$ e/ou descartar subconjuntos de X onde x^* não se encontra.

A relaxação lagrangiana é uma técnica muito conhecida e utilizada no desenvolvimento de métodos para a busca de soluções ótimas ou fortemente desejáveis para problemas da área Otimização Combinatória. Em particular, ela permite a obtenção de limitantes para esses problemas.

Neste capítulo, iremos aplicar a relaxação lagrangiana ao problema de EPMRC, utilizando os métodos de geração de cortes ou subgradientes na resolução do dual lagrangiano. Assim, obtemos limitantes superiores para o problema de EPMRC, além de uma heurística lagrangiana, utilizada para determinar uma solução viável, cujo valor fornece, pois, um limitante inferior para o problema. Realizamos experimentos computacionais com algumas combinações dos métodos e heurística, comparando o limitantes superiores e inferiores gerados e o tempo computacional dispendido, com o objetivo de determinar a melhor versão de combinação para ser utilizada junto ao processo de *B&B* no capítulo seguinte.

4.1 Definição do dual lagrangiano

A relaxação lagrangiana é uma técnica bastante empregada em problemas de otimização onde as restrições podem ser divididas em grupos que poderiam ser melhor tratados separadamente. A ideia consiste em ampliar o conjunto de soluções do problema original, retirando-se um subconjunto de restrições e acrescentando-as à função objetivo junto a um vetor de multiplicadores. Tal estratégia é comumente utilizada para fornecer limitantes, assim como a relaxação linear, e pode ser incorporada ao método de *B&B*.

Para descrever a aplicação dessa ideia em programação inteira 0-1, considere o

seguinte modelo genérico, onde as restrições estão divididas em dois grupos:

$$(P)\,\text{Maximizar } c^t x \tag{4.1}$$

Sujeito a:

$$Ax \leq b, \tag{4.2}$$

$$Dx \leq e, \tag{4.3}$$

$$x \in \{0,1\}^n. \tag{4.4}$$

Por simplicidade, vamos admitir que (P) é um problema viável e, por conseguinte, tem solução ótima.

Transferindo as restrições (4.2) para a função objetivo (4.1), penalizadas por multiplicadores lagrangianos λ não negativos, obtemos o seguinte modelo:

$$P(\lambda)\,\text{Maximizar } c^t x + \lambda^t(b - Ax) \tag{4.5}$$

Sujeito a:

$$Dx \leq e, \tag{4.6}$$

$$x \in \{0,1\}^n. \tag{4.7}$$

Dizemos então que $P(\lambda)$ é um problema lagrangiano associado a (P).

Com a relaxação das restrições (4.2) do modelo (P), podemos observar que o modelo $P(\lambda)$ abrange um conjunto maior de soluções, possivelmente muitas delas inviáveis para (P). Além disso, nos pontos viáveis para ambos os modelos, o valor da função objetivo de $P(\lambda)$ é no mínimo aquele em (P). Isto nos leva a obter que

$$val[P] \leq val[P(\lambda)] \quad \forall \lambda \geq 0, \tag{4.8}$$

onde $val[\,]$ denota o valor ótimo do problema. Isso é fácil mostrar. Assuma que x^* é ótimo para (P). Como x^* também é solução viável para $P(\lambda)$, temos que $val[P(\lambda)] \geq c^t x^* + \lambda^t(b - Ax^*)$, $\lambda \geq 0$ e $(b - Ax^*) \geq 0$. Então obtemos que

$$val[P(\lambda)] \geq c^t x^* + \lambda^t(b - Ax^*) \geq c^t x^* = val[P],$$

ou melhor, o valor ótimo de $P(\lambda)$ é um limite superior para (P).

Por outro lado, considere uma solução $\bar{x}$, ótima para $P(\lambda)$. Se $A\bar{x} \leq b$, então $\bar{x}$ é viável para (P). Consequentemente,

$$A\bar{x} \leq b \Rightarrow val[P] \geq c^t \bar{x},$$

Nesse caso, obtemos também um limite inferior para (P) a partir da solução de $P(\lambda)$. E se $\bar{x}$ satisfaz $\lambda^t(b - A\bar{x}) = 0$, temos, por (4.8), que $val[P] \leq c^t\bar{x} + \lambda^t(b - A\bar{x}) = c^t\bar{x}$. Em outras palavras,

$$A\bar{x} \leq b, \lambda^t(b - A\bar{x}) = 0 \Rightarrow val[P] = val[P(\lambda)] = c^t\bar{x}. \tag{4.9}$$

A expressão (4.8) nos diz que o valor ótimo de $P(\lambda)$, para qualquer $\lambda \geq 0$, é automaticamente um limite superior para (P). Visando encontrar o menor limite superior, formulamos o problema dual (ou dual lagrangiano) de (P), que consiste em encontrar o conjunto de componentes não negativas do vetor λ que determine a menor cota superior para (P). Precisamente, tal problema é

$$(D)\,\text{Minimizar}\;\; val[P(\lambda)]\;\;\text{Sujeito a}\;\; \lambda \geq 0, \tag{4.10}$$

que será apresentado com mais detalhes na próxima seção.

Aplicando agora a descrição de relaxação lagrangiana ao modelo de EPMRC (3.1)-(3.4), escolhemos dualizar as restrições de emparelhamento (3.2), em lugar as restrições de conflito (3.3), na tentativa de obter um limite superior melhor que aquele fornecido pela relaxação linear. Penalizando, então, as restrições de emparelhamento, chegamos à seguinte função objetivo:

$$\text{Maximizar} \sum_{e \in E} c_e x_e + \sum_{v \in V} \Big[\lambda_v \Big(1 - \sum_{e \in \delta(v)} x_e\Big)\Big]. \tag{4.11}$$

Desmembrando o segundo somatório em (4.11), obtemos:

$$\sum_{e \in E} c_e x_e + \sum_{v \in V} \lambda_v - \sum_{v \in V} \Big(\lambda_v \sum_{e \in \delta(v)} x_e\Big) \tag{4.12}$$

Podemos notar na expressão (4.12) que, para cada aresta $e = (u, v)$ ($e \in E$, com u e $v \in V$), existem duas componentes do vetor λ que multiplicam x_e, a saber λ_u e λ_v. Em outros termos, temos $(\lambda_u + \lambda_v)x_e$ como parcela no último somatório de (4.12). A partir dessa constatação podemos reescrever a expressão (4.12) como:

$$\sum_{(u,v) \in E} c_{uv} x_{uv} + \sum_{v \in V} \lambda_v - \sum_{(u,v) \in E} \big[\big(\lambda_u + \lambda_v\big) x_{uv}\big]. \tag{4.13}$$

Reduzindo ainda mais a expressão (4.13), temos:

$$\sum_{(u,v) \in E} \big[\big(c_{uv} - \lambda_u - \lambda_v\big) x_{uv}\big] + \sum_{v \in V} \lambda_v. \tag{4.14}$$

Sendo assim, o modelo matemático da relaxação lagrangiana para o EPMRC é dado por:

$$RL(\lambda)\,\text{Maximizar} \sum_{(u,v)\in E} \left[(c_{uv} - \lambda_u - \lambda_v)x_{uv}\right] + \sum_{v\in V} \lambda_v \tag{4.15}$$

Sujeito a:

$$x_e + x_{e'} \leq 1,\ \forall\ \{e,\ e'\} \in E_c \tag{4.16}$$

$$x_e \in \{0,1\},\ \forall\ e \in E. \tag{4.17}$$

Note que cada aresta $e = (u,v)$ está associada agora a um novo custo $\bar{c}_e = c_e - \lambda_u - \lambda_v$, que é modificado a cada λ fornecido, a ser chamado de *custo lagrangiano*.

4.2 Resolução do dual lagrangiano

A seguir, iremos descrever teoricamente dois métodos que resolvem o dual lagrangiano, para obtenção do vetor λ que fornece a melhor cota superior.

4.2.1 Método de geração de cortes

O método de geração de cortes se aplicará a partir da reescrita do dual lagrangiano (4.10) como um problema de programação linear com muitas restrições. Este será resolvido gerando iterativamente as restrições, a partir das soluções dadas pela relaxação lagrangiana. Demostraremos as seguir como se dará o processo, utilizando primeiro o modelo genérico (P) e a relaxação lagrangiana $P(\lambda)$.

Definindo $\Omega = \{x \in \{0,1\}^n : Dx \leq e\} \neq \emptyset$, temos inicialmente:

$$val[P(\lambda)] = \max\{c^t x + \lambda^t(b - Ax) : x \in \Omega\} \tag{4.18}$$

e

$(D)\,\text{Minimizar}\,Z$

Sujeiro a:

$$Z \geq \max\{c^t x + \lambda^t(b - Ax) : x \in \Omega\},$$

$$\lambda \geq 0.$$

Sendo Ω o conjunto de pontos 0-1 em um poliedro não vazio, podemos admitir

$\Omega = \{x^1, x^2, ..., x^{\mathrm{p}}\}$ com p finito. Portanto, podemos reestruturar o dual lagrangiano como:

$$(D)\,\text{Minimizar}\; Z \tag{4.19}$$

Sujeito a:

$$Z \geq c^t x^i + \lambda^t (b - Ax^i),\ \forall\ i = 1, ..., p, \tag{4.20}$$

$$\lambda \geq 0. \tag{4.21}$$

Podemos resolver o modelo acima, gerando suas restrições iterativamente da seguinte forma. Admita que $(\overline{\lambda}, \overline{Z})$ seja uma solução ótima de (D) com apenas um subconjunto de restrições (4.20) (possivelmente nenhuma). Considere também $\bar{x}$ uma solução ótima para $P(\overline{\lambda})$ (tal que $val[P(\overline{\lambda})] = c^t\overline{x} + \lambda^t(b - A\overline{x})$. Analisando essas soluções, apenas os seguintes casos podem ocorrer:

1. Se $val[P(\overline{\lambda})] \leq \overline{Z}$, então todas as restrições (4.20) estão satisfeitas por $(\overline{\lambda}, \overline{Z})$, mesmo que não estejam explicitamente no modelo, de modo que $\overline{\lambda}$ é o vetor de multiplicadores que determina $\overline{Z}$ como o menor limite superior (o valor do dual lagrangiano);
2. Se $val[P(\overline{\lambda})] > \overline{Z}$, então uma restrição em (D), relacionada a $\overline{x} \in \Omega$, está sendo violada; logo vamos adicionar a restrição $Z \geq c^t\overline{x} + \lambda^t(b - A\overline{x})$ ao modelo, reotimizá-lo e analisar as novas soluções obtidas.

A forma equivalente (4.19)-(4.21) do dual lagrangiano também permite estimar o seu valor ótimo. De fato, considere o dual de (D), que é dado por:

$$(DD)\,\text{Maximizar} \sum_{i=1}^{p} (c^t x^i) u_i \tag{4.22}$$

Sujeito a:

$$\sum_{i=1}^{p} (Ax^i - b) u_i \leq 0, \tag{4.23}$$

$$\sum_{i=1}^{p} u_i = 1, \tag{4.24}$$

$$u_i \geq 0\ \forall i = 1, ..., p. \tag{4.25}$$

Definindo

$$x = \sum_{i=1}^{p} x^i u_i,$$

as expressões em (4.22) e (4.23) tornam-se $c^t x$ e $\sum_{i=1}^{p} Ax \leq b \sum_{i=1}^{p} u_i = b$. Assim, podemos

reescrever (DD) como:

(DD) Maximizar $c^t x$

Sujeito a:

$$Ax \leq b,$$
$$x = \sum_{i=1}^{p} x^i u_i,$$
$$\sum_{i=1}^{p} u_i = 1,$$
$$u_i \geq 0 \ \forall i = 1, ..., p,$$

ou ainda

(DD) Maximizar $c^t x$

Sujeito a:

$$Ax \leq b,$$
$$x \in \text{conv}(\Omega).$$

O modelo acima mostra que o valor do dual lagrangiano é no máximo aquele da relaxação linear, pois $\text{conv}(\Omega) \subseteq \{x \in \mathbb{R}^n : Dx \leq e, 0 \leq x \leq 1\}$. Isso justifica a utilização do primeiro em substituição ao último como limite superior.

Aplicando agora a metodologia acima ao problema EPMRC e usando a relaxação lagrangiana (4.15)-(4.17), temos o modelo dual lagrangiano da seguinte forma:

$$(DL) \text{ Minimizar } Z \tag{4.26}$$

Sujeito a:

$$Z \geq \sum_{(u,v) \in E} \left[(c_{uv} - \lambda_u - \lambda_v) \bar{x}_{uv} \right] + \sum_{v \in V} \lambda_v, \ \forall \bar{x} \in X \tag{4.27}$$

$$\lambda_v \geq 0, \ \forall \ v \in V, \tag{4.28}$$

onde $X = \{x \in \{0,1\}^{|E|} : x_e + x_{e'} \leq 1 \, \forall \{e, e'\} \in E_c\}$. Dado $X' \subseteq X$, vamos denotar por $DL(X')$ o modelo acima apenas com as restrições (4.27) relativas a X'.

O Algoritmo 2 realiza o procedimento descrito acima, obtendo um limite superior para o EPMRC. Além disso, incorporamos a esse procedimento uma heurística lagrangiana, a ser comentada mais adiante, que retorna um limite inferior.

Algoritmo 2: Geração de Cortes

Entrada: (Modelo RL, Modelo DL)

▷ Modelo RL - (4.15)-(4.17), Modelo DL - (4.26)-(4.28)

Saída: UB - Limite Superior, LB - Limite Inferior

```
1  início
2      LB ← −∞, UB ← ∞
3      Escolha X′ ⊆ X (possivelmente vazio)
4      ok ← 0
5      faça
6          (Z̄, λ̄) ← Resolva DL(X′)
7          (x̄, ub) ← Resolva RL(λ̄)                     ▷ ub = val[RL(λ̄)]
8          se ub < UB então
9              UB ← ub
10         fim
11         (lb, x_viavel) ← Heurística Lagrangiana(x̄)    ▷ Ver Seção 4.3
12         se lb > LB então
13             LB ← lb
14             x_solução ← x_viável
15         fim
16         se Z̄ < ub então
17             X′ ← X′ ∪ {x̄}
18         senão
19             ok ← 1
20         fim
21     enquanto ok = 0;
22     retorne UB,LB
23 fim
```

4.2.2 *Método de subgradientes*

O método de subgradientes consiste em um processo iterativo muito utilizado para resolver problemas de otimização convexos não diferenciáveis, como é o caso do dual lagrangiano. Por definição, um vetor γ é um subgradiente de uma função convexa $f : \mathbb{R}^n \rightarrow \mathbb{R}$ no ponto $\bar{x}$, se $f(x) \geq f(\bar{x}) + \gamma^t(x - \bar{x})$.

Considere então o problema

$$\text{Minimizar } f(x) \text{ Sujeito a } x \in C,$$

onde $f : \mathbb{R}^n \to \mathbb{R}$ e $C \subseteq \mathbb{R}^n$ são convexos. O método de subgradientes para esse problema gera, a partir de um ponto inicial x^0, uma sequência $x^0, x^1, x^2, \ldots$ tal que $x^{k+1} = \text{Proj}_C(x^k - \alpha_k s^k)$, onde Proj_C representa o operador de projeção sobre C, s^k é um subgradiente de f em x^k e α_k é um tamanho de passo escolhido para garantir a convergência da sequência para uma solução do problema. Observe que, sendo $C = \mathbb{R}^n_+$, o operador de projeção corresponde simplesmente a anular as componentes negativas de $x^k - \alpha_k s^k$ para obter x^{k+1}.

Por simplicidade de notação, para aplicação desse método ao dual lagrangiano (4.10), vamos definir $\mathscr{L}(\lambda) = val(RL(\lambda))$. Primeiro, mostramos que $\mathscr{L}$ é convexa. De fato, dado $\varepsilon \in [0,1]$, temos que

$$\begin{aligned}
&\mathscr{L}(\varepsilon\lambda^1 + (1-\varepsilon)\lambda^2) \\
&\quad = \max\{c^t x + [\varepsilon\lambda^1 + (1-\varepsilon)\lambda^2]^t(b - Ax) : x \in \Omega\} \\
&\quad = \max\{\varepsilon[c^t x + (\lambda^1)^t(b - Ax)] + (1-\varepsilon)[c^t x + (\lambda^2)^t(b - Ax)] : x \in \Omega\} \\
&\quad \leq \max\{\varepsilon[c^t x + (\lambda^1)^t(b - Ax)] : x \in \Omega\} + \max\{(1-\varepsilon)[c^t x + (\lambda^2)^t(b - Ax)] : x \in \Omega\} \\
&\quad = \varepsilon\mathscr{L}(\lambda^1) + (1-\varepsilon)\mathscr{L}(\lambda^2)
\end{aligned}$$

Agora mostramos que um subgradiente de $\mathscr{L}$ em $\overline{\lambda}$ pode ser determinado a partir do ponto $\bar{x} \in \Omega$ que fornece o valor de $\mathscr{L}(\overline{\lambda})$, ou seja, de $\bar{x} \in \Omega$ tal que $\mathscr{L}(\overline{\lambda}) = c^t\bar{x} + \overline{\lambda}(b - A\bar{x})$. De fato,

$$\begin{aligned}
\mathscr{L}(\lambda) &= \max\{c^t x + \lambda^t(b - Ax) : x \in \Omega\} \\
&\geq c^t\bar{x} + \lambda^t(b - A\bar{x}) \\
&= c^t\bar{x} + \overline{\lambda}^t(b - A\bar{x}) + (\lambda - \overline{\lambda})^t(b - A\bar{x}) \\
&= \mathscr{L}(\overline{\lambda}) + (b - A\bar{x})^t(\lambda - \overline{\lambda}),
\end{aligned}$$

mostrando que $(b - A\bar{x})$ é um subgradiente de $\mathscr{L}$ em $\overline{\lambda}$.

No caso de EPMRC temos, a partir de (4.11), que um sugradiente $s \in \mathbb{R}^{|V|}$ tem componentes

$$s_v = 1 - \sum_{e \in \delta(v)} x_e \;\; \forall v \in V,$$

onde $x \in \{0,1\}^{|E|}$ é uma solução ótima de (4.15)-(4.17). Com isso, a partir de um vetor inicial λ^0, a sequência de multiplicadores é dado por

$$\lambda_v^{k+1} = \max\left\{0, \lambda_v^k - \alpha_k\left(1 - \sum_{e\in\delta(v)} x_e^k\right)\right\} \; \forall v \in V,$$

onde x^k é uma solução ótima de $RL(\lambda^k)$.

O Algoritmo 3 apresenta então o método de subgradientes aplicado ao EPMRC. Escolhemos a atualização do tamanho do passo conforme recomendado na literatura (ver por exemplo (WOLSEY, 1998)). Adotamos também critérios de parada usuais (número máximo de iterações, norma do subgradiente ou tamanho do passo abaixo de um limiar) . Observe que, mesmo parando antes da convergência, o melhor valor de $RL(\lambda)$ obtido é um limite superior. Assim, como no Algoritmo 2, embutimos um procedimento para obtenção de um limite inferior, a ser descrito na próxima seção.

Os parâmetros adotados para a definição dos critérios de parada foram:

1. Número de iterações: 500
2. $\alpha_k < 0.001$
3. $\pi_k < 0.001$
4. $||s^k||^2 < 0.02$

4.3 Heurística lagrangiana

Como vimos anteriormente, a relaxação lagrangiana nos fornece uma maneira de encontrar um limitante superior (considerando o problema de maximização), possivelmente mais apertado em comparação àquele da relaxação linear, o que poderia, em princípio, melhorar o procedimento de poda de nós, se utilizado no método de B&B e, consequentemente, poderia levar a um menor custo computacional do processo de busca pela solução ótima.

Sabemos que a solução fornecida por qualquer relaxação é, em geral, inviável, já que alguns critérios ou restrições foram ignoradas no processo. Na relaxação lagrangiana, fazemos “vista grossa” com um subconjunto de restrições, para obtermos uma solução em tempo computacional menor. Entretanto, sendo tal solução inviável, podemos procurar torná-la viável, realizando modificações para a solução passar a obedecer o conjunto de restrições antes ignorado. Com isso, podemos obter também um limitante inferior.

Para tornar viável a solução, podemos fazer uso de heurísticas e, já que estas vão ser aplicadas a soluções dadas pela relaxação lagrangiana, podemos chamá-las de *heurísticas*

Algoritmo 3: Subgradiente

Entrada: (Modelo RL) ▷ Modelo RL - (4.15)-(4.17)

Saída: UB - Limite Superior, LB - Limite Inferior

1 **início**
2 $LB \leftarrow -\infty$, $UB \leftarrow \infty$
3 Escolha λ^0, $k \leftarrow 0$
4 **faça**
5 $(\bar{x}, ub) \leftarrow$ Resolva $RL(\lambda^k)$
6 **se** $ub < UB$ **então**
7 $UB \leftarrow ub$
8 **fim**
9 $(lb, x_{\text{viável}}) \leftarrow$ *Heurística Lagrangiana*$(\bar{x})$
10 **se** $lb > LB$ **então**
11 $LB \leftarrow lb$
12 $x_{\text{solução}} \leftarrow x_{\text{viável}}$
13 **fim**
14 $s^k \leftarrow \{1 - \sum_{e \in \delta(v)} \bar{x}_e\}$
15 $\alpha_k \leftarrow \frac{\pi_k(ub-LB)}{||s^k||^2}$, $\pi_k \in (0,2]$ ▷ $\pi_0 = 2$ e $\pi_k/2$ a cada 20 iterações sem melhorar ub
16 $\lambda^{k+1} \leftarrow \max\{0, \lambda^k - \alpha_k s^k\}$
17 $k \leftarrow k+1$
18 **enquanto** *Critério de parada não for atendido*;
19 **retorne** UB,LB
20 **fim**

lagrangianas.

Heurísticas para problemas de otimização são procedimentos que exploram o conjunto de possíveis soluções, na tentativa de resolver efetivamente o problema ou fornecer um ponto de partida viável para um método exato. Sua praticidade está em encontrar uma solução viável em tempo computacional aceitável, sem necessariamente focar em determinar uma solução ótima.

Neste trabalho, propomos heurísticas para o problema de EPMRC baseadas na relaxação lagrangiana (4.15)-(4.17). Precisamente, usamos duas informações fornecidas por esse modelo: (i) o ponto solução, que representa um conjunto de arestas sem conflito, mas que podem

desrespeitar as restrições de emparelhamento e (ii) os custos lagrangianos, cuja evolução ao longo do método de resolução do dual lagrangiano tende a atribuir pesos às arestas que procuram fazer cumprir também as restrições de emparelhamento.

Usando essas informações, optamos por duas estratégias heurísticas para obtenção de soluções viáveis (ou limites inferiores) para o nosso problema:

1. Heurística Lagrangiana EPM - Resolve um problema de emparelhamento de peso máximo
2. Heurística Lagrangiana Gulosa - Encontra emparelhamento de forma gulosa

A primeira estratégia consiste em obter a melhor solução viável possível, considerando apenas as arestas indicadas pela solução da relaxação lagrangiana. Isso corresponde a resolver um problema de emparelhamento de peso máximo (considerando os pesos originais) no subgrafo induzido por essas arestas, visto que elas já são livres de conflitos. Embora polinomial, a resolução repetida desse problema, ao longo do processo de resolução do dual lagrangiano, pode ser bastante dispendioso computacionalmente. Assim, avaliamos também outra alternativa.

A segunda estratégia emprega uma forma mais simples e de menor esforço computacional que a primeira para obtenção das soluções viáveis. A partir de uma ordenação das arestas indicadas pela solução da relaxação lagrangiana, segundo ordem decrescente dos custos (originais ou lagrangianos), determinamos um emparelhamento: nessa ordem, uma a uma cada aresta é analisada, escolhida para compor a solução, caso não seja adjacente a outra já selecionada, ou descartada, caso contrário. Note que temos duas versões dessa heurística: uma considerando os pesos originais das arestas e outra que usa os pesos modificados pelos multiplicadores lagrangianos (custos lagrangianos).

4.4 Implementações

Assim como no Capítulo 2, utilizamos o CPLEX para implementação e avaliação dos métodos descritos para o problema de EPMRC. Avaliamos a qualidade dos limites inferiores e superiores obtidos, bem como o esforço computacional dispendido para isso. Primeiramente, implementamos a formulação do problema de EPMRC (3.1)-(3.4) e resolvemos sua relaxação linear. Em seguida, implementamos a relaxação lagrangiana (4.15)-(4.17), resolvendo o dual lagrangiano via geração de cortes (Algoritmo 2) ou método de subgradientes (Algoritmo 3), incorporando uma das duas heurísticas lagrangianas propostas na Seção 4.3.

Note que, para implementação do algoritmo lagrangiano, temos muitas possibilidades, a depender do método usado para resolver o dual lagrangiano, da heurística a ele incorporada

e de que momentos ela é executada. Além disso, há diferentes variações de cada uma das duas heurísticas. Aquela que resolve um problema de emparelhamento de peso máximo depende, por exemplo, de como esse problema é abordado, se faz ou não uso das restrições de blossom. Já a heurística gulosa é influenciada pela escolha dos custos adotados na ordenação das arestas (custos originais ou lagrangianos).

Para efeitos comparativos, apresentamos a implementação e relatamos algumas experiências computacionais com as seguintes alternativas para obtenção dos limitantes:

1. Relaxação linear do modelo (3.1)-(3.4)
2. Relaxação lagrangiana (das restrições de emparelhamento) via:
   - 2.1. Método de Geração de Cortes com Heurística Lagrangiana EPM;
   - 2.2. Método de Subgradientes com Heurística Lagrangiana EPM;
   - 2.3. Método de Subgradientes com Heurística Lagrangiana EPM por Restrições de Blossom;
   - 2.4. Método de Subgradientes com Heurística Lagrangiana Gulosa;
   - 2.5. Método de Subgradientes com Heurística Lagrangiana Gulosa por Custos Lagrangianos.

4.4.1 Resolução da relaxação linear do modelo

Montamos e resolvemos a relaxação linear do modelo de EPMRC através da API CPLEX. Abaixo exibimos as partes principais do código em C_{++}.

Código-fonte 4 – Versão 4.1

```
1  #include<iostream>
2  #include<ilcplex/ilocplex.h>
3  ILOSTLBEGIN
4  using namespace std;
5  int main(){
6      \\Leitura da instancia;
7      \\Criacao da Matriz de Custos e Adjacencia (C_Ma) -
         C_Ma[i][j]<0 se nao existe aresta (i,j);
8      \\Criacao da Matriz de Conflitos (Gc);
9
10     IloEnv env;\\Definindo o ambiente de programacao
11
```

```
12  \\ DECLARACAO DAS ESTRUTURAS DE PROGRAMACAO
13  IloModel modelo(env); \\Modelo de programacao matematica
14  IloRangeArray rest(env); \\Restricoes do modelo
15  IloObjective obj(env); \\Funcao objetivo
16  IloExpr expr_obj(env), expr(env); \\Expressoes lineares uteis na
        definicao de restricoes e da funcao objetivo
17
18  \\ DECLARACAO DAS VARIAVEIS CONTINUAS
19  IloArray<IloNumVarArray> x(env, n);
20  for (int i = 0; i < n; i++)
21      x[i] = IloNumVarArray(env, n, 0, 1);
22
23  \\ CRIACAO DO MODELO EPMRC
24  for (int i = 0; i < n; i++) {
25      for (int j = i + 1; j < n; j++) {
26          if (C_Ma[i][j] >= 0) {
27              expr_obj += C_Ma[i][j] * x[i][j];
28              modelo.add(x[i][j]);
29          }
30      }
31  }
32  obj = IloMaximize(env, expr_obj); \\Sentido de otimizacao
33  modelo.add(obj);
34  for (int i = 0; i < n; i++) {
35      for (int j = i + 1; j < n; j++) {
36          if (C_Ma[i][j] >= 0)
37              expr += x[i][j];
38      }
39      for (int j = i - 1; j >= 0; j--) {
40          if (C_Ma[j][i] >= 0)
41              expr += x[j][i];
42      }
43      rest.add(expr <= 1); \\Restricao de emparelhamento
44      expr.clear();
45  }
46  for(int i = 0; i < mc; i++){
47      rest.add(x[Gc[i][0]][Gc[i][1]] + x[Gc[i][2]][Gc[i][3]] <= 1);
            \\Restricao de conflito
48  }
```

```
49      modelo.add(rest);
50
51      \\ SOLUCAO DA RELAXACAO LINEAR DO PROBLEMA DE EPMRC
52      IloCplex cplex(modelo);
53      cplex.solve();
54
55      \\Inquisicao da solucao obtida.
56
57      env.end();
58  }
59  return 0;
```

O código acima ilustra como foi realizada a construção da relaxação linear do modelo EPMRC, ignorando a integralidade de suas variáveis, com a API do CPLEX. Destacamos nas linhas 6-8 um processo básico de leitura e definição de estruturas C_{Ma}, como a matriz de custos e adjacência, e Gc, como uma matriz de dimensão $|E_c|$x4, tal que cada linha contém os índices que determinam o par de arestas em conflito, para guardar os dados da instância do problema. Na linha 10, temos a declaração do ambiente, que armazenará as estruturas para a construção e resolução do modelo. Nas linhas 19-21, temos a declaração das variáveis relaxadas, assumindo valores entre 0 e 1 como viáveis, para cada aresta candidata à solução, ou seja, sem o uso da integralidade (ILOINT). Entre as linhas 24-31, construímos a expressão da função objetivo e definimos seu sentido de otimização (linha 32), adicionando-a posteriormente ao modelo na linha 33. Embora seja definida uma variável para cada possível aresta, apenas aquelas relacionadas a arestas efetivamente presentes no grafo são adicionadas ao modelo (linha 28). Novamente, como visto no Capítulo 2, a declaração de todas essas variáveis deve-se à facilidade de indexação. Seguindo, nas linhas 34-45, há a formação das restrições que definem um emparelhamento e depois as que estabelecem os pares de arestas conflitantes (linhas 46-48), acrescentadas posteriormente ao modelo (linha 49). Por último, na linha 53, encontra-se a chamada ao método *solve* para a solução do modelo. As linhas seguintes são destinadas à investigação do resultado obtido, em particular se a solução obtida é de fato ótima (*cplex.getStatus()*) e qual o valor da função objetivo (*cplex.getObjValue()*) e das variáveis (*cplex.getValues()*). Lembrando que a solução gerada pelo resolvedor será apenas um limite superior para o problema em questão, devido a relaxação da integralidade das variáveis.

4.4.2 Resolução da relaxação lagrangiana do modelo

Apresentamos abaixo as implementações das cinco versões referentes à relaxação lagrangiana, exibindo as partes principais dos códigos em C_{++} e a descrição do processo, evitando redundâncias nas explicações, tendo em vista o que já foi apresentado.

Começamos com a primeira versão, que implementa a relaxação lagrangiana através do método de geração de cortes e usa como heurística lagrangiana a resolução de um emparelhamento de peso máximo no subgrafo induzido pelas arestas sem conflito, indicadas pelo subproblema lagrangiano. Chamamos a atenção para que, nesse caso, trabalhamos simultaneamente com três modelos de programação matemática, relacionados respectivamente ao dual lagrangiano, ao subproblema lagrangiano e à heurística.

Código-fonte 5 – Versão 4.2.1

```
1  #include<iostream>
2  #include<ilcplex/ilocplex.h>
3  ILOSTLBEGIN
4  using namespace std;
5  void XRL_INICIAL(IloEnv& env_RL, IloModel& modelo_RL, double** C_Ma, int
       n, IloArray<IloNumVarArray>& x_RL, IloObjective& obj_RL, IloCplex&
       cplex_RL, double* val_lambda){
6       IloExpr expr_RL(env_RL);
7       for(int i = 0; i < n; i++){
8           expr_RL += val_lambda[i];
9           for(int j = i+1; j < n; j++){
10              if(C_Ma[i][j] >= 0)
11                  expr_RL += (C_Ma[i][j] - val_lambda[i] - val_lambda[j])*
                        x_RL[i][j];
12          }
13      }
14      obj_RL.setExpr(expr_RL);
15      cplex_RL.extract(modelo_RL);
16      expr_RL.end();
17      cplex_RL.solve();
18 }
19
20 double HEURIST_EPM(int n, IloEnv& env_HL, IloModel& modelo_HL,
       IloRangeArray& Rest_HL, IloObjective& obj_HL, IloCplex& cplex_HL,
```

```
    double** C_Ma, IloArray<IloNumVarArray>& x_HL, IloArray<IloNumArray>
    val_x_RL);
21      IloExpr expr_obj(env_HL);
22      for (int i = 0; i < n; i++) {
23          for (int j = i + 1; j < n; j++) {
24              if (val_x_RL[i][j] > 0.9999)
25                  expr_obj += C_Ma[i][j] * x_HL[i][j];
26          }
27      }
28      obj_HL.setExpr(expr_obj);
29      cplex_HL.extract(modelo_HL);
30      cplex_HL.solve();
31      return (double)cplex_HL.getObjValue();
32  }
33
34  int main(){
35      \\Leitura da instancia e criacao das matrizes (C_Ma) e (Gc) como na
        versao 4.1 (linhas 6-8)
36      \\DECLARACAO DAS ESTRUTURAS DE PROGRAMCAO DOS MODELOS RL, DL E HL
37      IloEnv env_RL, env_DL, env_HL;
38      IloModel modelo_RL = IloModel(env_RL), modelo_DL = IloModel(env_DL),
        modelo_HL = IloModel(env_HL);
39      IloRangeArray rest_RL = IloRangeArray(env_RL), rest_DL = IloRangeArray(
        env_DL), rest_HL = IloRangeArray(env_HL);
40      IloNumVar Z = IloNumVar(env_DL, 0, IloInfinity);
41      IloObjective obj_RL = IloMaximize(env_RL), obj_DL = IloMinimize(env_DL,
        Z), obj_HL = IloMaximize(env_HL);
42      IloExpr objexpr_RL(env_RL);
43      IloCplex cplex_RL = IloCplex(env_RL), cplex_DL = IloCplex(env_DL),
        cplex_HL = IloCplex(env_HL);
44      IloNumVarArray lambda = IloNumVarArray(env_DL, n, 0, IloInfinity);
45      IloNumArray val_lambda = new IloNumArray(env_DL, n);
46      IloArray<IloNumVarArray> x_RL = IloArray<IloNumVarArray>(env_RL, n);
47      IloArray<IloNumVarArray> x_HL = IloArray<IloNumVarArray>(env_HL, n);
48      IloArray<IloNumArray> val_x_RL = IloArray<IloNumArray>(env_RL, n);
49      for (int i = 0; i < n; i++) {
50          x_HL[i] = IloNumVarArray(env_HL, n, 0, 1, ILOINT);
51          x_RL[i] = IloNumVarArray(env_RL, n, 0, 1, ILOINT);
52          val_x_RL[i] = IloNumArray(env_RL, n);
```

```
53  }
54  \\MONTAGEM DOS MODELOS RL, DL E HL
55  \\Modelo (4.15)-(4.17)
56  MONTAGEM_MODELO_RL(n, mc, C_Ma, Gc, env_RL, modelo_RL, Rest_RL, x_RL,
        obj_RL, cplex_RL);
57
58  \\O vetor (val_lambda) recebe inicialmente os valores duais associados
        as restricoes relaxadas
59
60  \\Determinando a solucao inicial do problema RL. Utiliza-se a solucao
        para a formacao de uma restricao na montagem do modelo DL
61  XRL_INICIAL(env_RL, modelo_RL, C_Ma, n, x_RL, obj_RL, cplex_RL,
        val_lambda);
62
63  \\Modelo (4.26)-(4.28)
64  MONTAGEM_MODELO_DL(n, C_Ma, env_DL, cplex_RL, x_RL, modelo_DL, Rest_DL,
         Z, lambda, obj_DL, cplex_DL);
65
66  \\Heuristica lagrangiana: resolucao do modelo de emparelhamento de peso
         maximo (2.1)-(2.3)
67  MONTAGEM_MODELO_HL(n, C_Ma, env_HL, modelo_HL, rest_HL, obj_HL,
        cplex_HL, x_HL)
68
69  double UB, LB, val_RL, val_HL, val_Z; bool ok = false;
70  UB = (double)cplex_RL.getObjValue(); \\Primeiro limite superior(
        associado ao XRL inicial)
71  \\(val_x_RL) captura os valores de (x_RL)
72  LB = HEURIST_EPM(n, env_HL, modelo_HL, rest_HL, obj_HL, cplex_HL, C_Ma,
         x_HL, val_x_RL);
73  do{
74       \\Resolucao do modelo dual lagrangeano
75       RESOLVE_DL(cplex_DL);
76       val_z = (double)cplex_DL. getObjValue();
77       \\(val_lambda) captura os valores de (lambda)
78       \\Atualizacao da funcao objetivo do modelo RL
79       for(int i = 0; i < n; i++){
80           objexpr_RL += val_lambda[i];
81           for(int j = i+1; j < n; j++){
82                if(C_Ma[i][j] >= 0)
```

```
83                  objexpr_RL += (C_Ma[i][j] - val_lambda[i] - val_lambda[j
                        ])*x_RL[i][j];
84              }
85          }
86          obj_RL.setExpr(objexpr_RL);
87          cplex_RL.extract(modelo_RL);
88
89          \\Resolucao do modelo da relaxacao lagrangeana
90          RESOLVE_RL(env_RL, modelo_RL, obj_RL, cplex_RL, x_RL, C_Ma);
91          val_RL = (double)cplex_RL.getObjValue(); \\ UPPER BOUND
92          \\(val_x_RL) captura os valores de x_RL
93
94          \\ Se valor da Funcao Objetivo do modelo (RL) < valor de UB, entao
                salva-se a solucao correspondente
95          if (val_RL - UB < -0.001){
96              UB = val_RL;
97              \\Salvar val_x_RL como melhor solucao para o limite superior
                    minimo encontrado
98          }
99
100         val_HL = HEURIST_EPM(n, env_HL, modelo_HL, rest_HL, obj_HL,
                cplex_HL, C_Ma, x_HL, val_x_RL);\\ LOWER BOUND
101         \\(val_x_HL) captura os valores de (x_HL)
102
103         \\Se valor da Funcao Objetivo do modelo HL > valor de (LB), entao
                salva-se a solucao correspondente
104         if (val_HL - LB > 0.001){
105           LB = val_HL;
106           \\Salvar o (x_HL) como melhor solucao para o limite inferior
                  maximo encontrado
107         }
108
109         \\Se (val_RL) > (val_Z), entao adicionamos uma restricao em DL
                correspondente ao (x) solucao de RL.
110         if (val_RL - val_Z > 0.001) {
111             IloExpr expr_DL(env_DL);
112             for (int i = 0; i < n; i++) {
113                 expr_DL += lambda[i];
114                 for (int j = i + 1; j < n; j++) {
```

```
115                     if (val_x_RL[i][j] > 0.9999)
116                         expr_DL += C_Ma[i][j] - lambda[i] - lambda[j];
117                 }
118             }
119             rest_DL.add(Z >= expr_DL); \\Adicao da restricao violada ao DL
120             modelo_DL.add(rest_DL);
121             cplex_DL.extract(modelo_DL);
122             expr_DL.end();
123         }else{
124             ok = true;
125         }
126     }while (!ok);
127 
128     \\Inquisicao dos limites
129 
130     env_RL.end();
131     env_DL.end();
132     env.HL.end()
133     return 0;
134 }
```

Essa primeira versão está descrita no Algoritmo 2. Após a leitura das instâncias e alocação dos dados nas matrizes de custo (C_{Ma}) e de conflitos (Gc), como visto na Versão 4.1, temos as declarações das estruturas de programação (linhas 37-53) que compõem os seguintes modelos ativos: (i) modelo RL (Relaxação Lagrangiana), (ii) modelo DL (Dual Lagrangiano) e (iii) modelo HL (Heurística Lagrangiana). A linha 56 chama a função *MOTAGEM_MODELO_RL*, responsável pela construção do modelo RL (4.15)-(4.17), recebendo como parâmetros todas as estruturas necessárias para definir suas restrições e função objetivo; similarmente, as funcões *MOTAGEM_MODELO_DL* e *MOTAGEM_MODELO_HL*, chamadas nas linhas 64 e 67, respectivamente, formam os modelos DL (4.26)-(4.28) e HL, este último construído exatamente como o modelo de EPM (2.1)-(2.3), visto no Capítulo 2.

Lembramos que o modelo RL é definido a partir de um conjunto de multiplicadores (dados pelo vetor *val_lambda*). Usamos como multiplicadores iniciais os valores das variáveis duais, associadas às restrições dualizadas, no ótimo da relaxação linear. Por sua vez, as restrições do modelo DL são dadas pelas soluções do modelo RL. Para definir a primeira restrição, usamos a solução de RL associada aos multiplicados lagrangianos iniciais, obtida pela função

XRL_INICIAL. Já o modelo HL irá conter inicialmente as variáveis relativas a todas as arestas. Antes de sua resolução, porém, a função objetiva será atualizada e composta por apenas as variáveis indicadas pela solução corrente do modelo RL.

Após a formação dos modelos, salvamos os limites iniciais: superior (linha 70), fornecido pelo ponto calculado pela função *XRL_INICIAL*, e inferior (linha 72), obtido através de uma heurística (função *HEURIST_EPM*), detalhada mais adiante. Esses limites serão atualizados ao longo do laço que se inicia na linha 73.

Resolvemos o modelo DL, através da função *RESOLVE_DL* (linha 75), e com sua resolução, utilizamos o valor do vetor λ obtido para modificar a função objetivo do modelo RL (linhas 77-87), que é depois resolvido com a chamada da função *RESOLVE_RL* (linha 90). Os valores de função objetivo das soluções encontradas são passados (através dos comandos *cplex_DL.getObjValue()* e *cplex_RL.getObjValue()*) para as variáveis *val_z* (linha 76), correspondendo ao modelo DL, e *val_RL* (linha 91), correspondendo ao modelo RL, para uma comparação que determinará a necessidade ou não de o processo continuar em busca do limite superior ótimo (linha 110).

Antes, porém, entre as linhas 95-98, realizamos um teste de melhora de limite superior; caso o limite da iteração atual seja menor que o melhor limite obtido, realizamos uma atualização do melhor limite superior e salvamos os valores das variáveis que contêm a solução a ele correspondente.

A resolução do modelo RL fornece um subconjunto de arestas sem conflitos. Encontramos o melhor emparelhamento no subgrafo induzido por essas arestas, chamando a função *HEURIST_EPM*. Passamos para a variável *val_HL* o valor da função objetivo (linha 100), determinando assim o limite inferior dessa iteração. Como feito para o limite superior, realizamos uma comparação para o limite inferior nas linhas 104-107: se o limite inferior atual é maior que o melhor limite inferior, uma atualização é realizada, além de salvar a solução correspondente.

Na linha 110, realizamos o teste entre os valores obtidos pela Relaxação Lagrangiana (*val_RL*) e pelo Dual Lagrangiano (*val_z*). Caso a condição de parada não seja verificada, realizamos o processo descrito nas linhas 110-122, que corresponde à construção e adição de uma nova restrição ao modelo DL, para assim reiniciar o processo a partir da linha 75. Ainda sobre a adição da nova restrição (linhas 119-121), vale destacar que, após adicionar a expressão ao objeto *rest_DL* para se unir às demais restrições, e depois ao *modelo_DL*, foi necessário utilizarmos o comando *cplex_DL.extract(modelo_DL)*, para atualizar o objeto *cplex_DL* do novo

modelo a ser otimizado.

Após diversas iterações realizadas, resolvendo e compartilhando informações entre os modelos RL, DL e HL, quando a condição de parada é satisfeita, podemos concluir, inquirindo os limites obtidos, armazenados pelas variáveis *UB*(limite superior) e *LB*(limite inferior), além das variáveis que mantêm as soluções correspondentes. Por fim, liberamos a memória dos 3 modelos, através dos comandos *env_RL.end(), env_DL.end(), env_HL.end()*.

Código-fonte 6 – Versão 4.2.2

```
1  #include<ilcplex/ilocplex.h>
2  #include<iostream>
3  ILOSTLBEGIN
4  using namespace std;
5  int main(){
6    \\Leitura da instancia e criacao das matrizes (C_Ma) e (Gc) como na
       Versao 4.1(linhas:6-8)
7    \\Definicao das estruturas de programacao como na Versao 4.2.1 com
       excecao do modelo DL(linhas:37-53)
8    \\Montagem dos modelos RL(linha 56) e HL(linha 67) semelhantes a Versao
       4.2.1
9    double *r = new double[n];
10   double *Sg = new double[n];
11   double UB, LB = 0, UB_k, val_HL, pi_k = 2;
12   UB = IloInfinity;
13   int k = 0, cont = 0;
14   bool ok = false;
15   do{
16       \\Modifica funcao objetivo como nas linhas 78-87 da Versao 4.2.1
17       \\Resolucao do modelo da relaxacao lagrangeana
18       RESOLVE_RL(env_RL, modelo_RL, obj_RL, cplex_RL, x_RL, C_Ma);
19       UB_k = (double)cplex_RL.getObjValue(); \\UPPER BOUND
20
21       \\(val_x_RL) captura os valores de (x_RL)
22
23       if (UB_k - UB < -0.001) {
24          cont = 0;
25          UB = UB_k;
26          \\Salvar (val_x_RL) como melhor solucao para o limite superior
               minimo encontrado
```

```
27        }else{
28            cont++;
29            if (cont == 20) {
30                cont = 0;
31                pi_k = pi_k / 2;
32            }
33        }
34
35        \\Uso da heuristica lagrangeana descrita nas linhas 20-32 como na
              Versao 4.2.1
36        val_HL = HEURIST_EPM(n, env_HL, modelo_HL, rest_HL, obj_HL,
              cplex_HL, C_Ma, x_HL, val_x_RL); \\LOWER BOUND
37        \\(val_x_HL) captura os valores de (x_HL)
38
39        if (val_HL - LB > 0.001) {
40            LB = val_HL;
41            \\Salvar o (val_x_HL) como melhor solucao para o limite inferior
                  maximo encontrado
42        }
43
44        \\Metodo do Subgradiente
45        double norma2_Sg = 0;
46        bool ok = false;
47        for (int i = 0; i < n; i++) {
48            Sg[i] = 1;
49            for (int j = i + 1; j < n; j++) {
50                if (C_Ma[i][j] >= 0)
51                    Sg[i] -= val_x_RL[i][j];
52            }
53            for (int j = i - 1; j >= 0; j--) {
54                if (C_Ma[j][i] >= 0)
55                    Sg[i] -= val_x_RL[j][i];
56            }
57            norma2_Sg += Sg[i] * Sg[i];
58        }
59
60        double tam_passo = (pi_k*(UB_k - LB)) / (norma2_Sg+0.001);
61
62        if ((norma2_Sg < 0.002) || (pi_k < 0.001) || (tam_passo < 0.001))
```

```
63            ok = true;
64        else
65            for (int i = 0; i < n; i++) {
66                val_lambda[i] -= tam_passo * Sg[i];
67                if (val_lambda[i] < 0)
68                    val_lambda[i] = 0;
69            }
70
71        \\Fim de uma iteracao do subgradiente
72        k++;
73
74    }while (k < 500 && !ok);
75
76    \\Inquisicao dos limites
77    env_RL.end();
78    env.HL.end();
79    return 0;
80 }
```

Acima apresentamos a versão da relaxação lagrangiana utilizando agora o método de subgradientes para obter o melhor limite superior, ou pelo menos aproximado, com menos esforço computacional que a Versão 4.2.1. A heurística lagrangiana nesta versão é a mesma da versão anterior.

Podemos observar que as linhas 6-8 mostram uma semelhança direta na leitura, definição de estruturas e modelos utilizados e explicados na versão anterior. Na linha 18, temos a chamada para a resolução do modelo RL através da função *RESOLVE_RL*, atribuindo o valor da função objetivo à variável UB_k.

Após esse procedimento, realizamos nas linhas 23-27 as etapas descritas na versão 4.2.1 (linhas 95-98) de atualização do limite superior e variáveis. Caso o limite superior calculado na iteração atual seja inferior ao melhor limite superior, incrementamos um contador, que indica o número de iterações sem melhorar o limite superior. Quando esse contador chega a um certo limite (usamos 20), o parâmetro π_k, que multiplica o tamanho do passo, será reduzido pela metade, para forçar o processo a "caminhar a passos mais curtos" em direção ao limite ótimo (ver linhas 27-32).

Na linha 36, há a chamada à mesma função usada na Versão 4.2.1, para encontrar o

emparelhamento máximo no subgrafo induzido pelas arestas (sem conflito) indicadas na solução do modelo RL. Como nas linhas 104-107 da Versão 4.2.1, repetimos a etapa de obtenção e atualização do limite inferior, nesta versão encontrada nas linhas 39-42.

Na linha 45, começamos o processo de atualização do vetor λ, realizando o cálculo de suas componentes, utilizando o vetor *Sg* para guardar o subgradiente e o escalar *norma2_Sg* para armazenar o quadrado de sua norma euclideana (linhas 47-58). Na linha 60, calculamos o tamanho do passo, como descrito pelo Algoritmo 3.

Na linha 62, testamos se algum dos critérios de parada é satisfeito, para verificar se o processo chegou ao fim e não há necessidade de atualização do limite superior. Caso nenhum critério seja atendido, entre as linhas 65-69, realizamos o processo de atualização dos multiplicadores λ e reiniciamos o processo a partir da linha 16. Mas isso apenas no caso de a quantidade de iterações realizadas ainda ser inferior a 500.

Após algum critério de parada ser atendido, podemos inquerir os limites obtidos através das variáveis *UB*(limite superior) e *LB*(limite inferior), liberando a memória dos modelos RL e HL posteriormente, semelhante à versão anterior.

Código-fonte 7 – Versão 4.2.3

```
1  #include<ilcplex/ilocplex.h>
2  #include<iostream>
3  ILOSTLBEGIN
4  using namespace std;
5  HEURIST_EPM_BLOSSOM(IloEnv &env_HL, IloModel& modelo_HL, IloRangeArray
       rest_HL, IloObjective &obj_HL, IloCplex &cplex_HL, double **C_Ma,
       IloArray<IloNumVarArray> &x_HL, IloArray<IloNumArray> &val_x_RL){
6      \\Atualizacao de (obj_HL) com as variaveis (x_HL) semelhante a
          solucao das variaveis (x_RL) dado por (val_x_RL)
7      \\Uso do codigo visto no Capitulo 2, Versao 2.2.2, com respeito a
          construcao e resolucao do emparelhamento de peso maximo com
          restricoes de blossom
8      return (double)cplex_HL.getObjValue();
9  }
10
11 int main(){
12     \\Leitura da instancia e criacao das matrizes (C_Ma) e (Gc) como na
          Versao 4.1(linhas 6-8)
13     \\Definicao das estruturas de programacao como na Versao 2.1 com
```

```
        excecao do modelo DL(linhas 37-53)
14     \\Montagem dos modelos RL(linha 56) e HL(linha 67) semelhantes a
        Versao 4.2.1
15     \\Definicao das variaveis presentes na Versao 4.2.2(linhas 9-14)
16
17     do{
18         \\Resolucao do modelo da relaxacao lagrangeana
19         RESOLVE_RL(val_lambda, env_RL, modelo_RL, obj_RL, cplex_RL, x_RL
              , C_Ma);
20         UB_k = (double)cplex_RL.getObjValue(); \\UPPER BOUND
21         \\(val_x_RL) captura os valores de (x_RL)
22
23         \\Realiza os processos descritos de atualizacao de (UB) e o
              parametro (pi_k) nas linhas 23-33 Versao 4.2.2
24         val_HL = HEURIST_EPM_BLOSSOM(env_HL, modelo_HL, rest_HL, obj_HL,
               cplex_HL, C_Ma, x_HL, val_x_RL);\\ LOWER BOUND
25         \\(val_x_HL) captura os valores de (x_HL)
26
27         \\Realiza os processos descritos de atualizacao de (LB) nas
              linhas 39-42 Versao 4.2.2
28
29         \\Atualizacao do vetor (lambda) pelo metodo do subgradiente
              linhas 45-69
30     }while(k < 500 && !ok);
31
32     \\Inquisicao dos limites
33
34     env_RL.end();
35     env.HL.end();
36     return 0;
37 }
```

O código acima ilustra uma variação semelhante à anterior, com o uso do método de subgradientes, para o obter limite superior, e a resolução do modelo de EPM, para o limite inferior. A diferença entre esta versão e a Versão 4.2.2 está na forma de como resolvemos o modelo de EPM na heurística lagrangiana. Aplicamos os conceitos vistos no Capítulo 2 sobre as restrições de blossom, com o objetivo de verificar uma possível melhora no tempo de solução da

heurística, que passa a resolver um modelo linear (com a separação das restrições de blossom) em lugar de um modelo inteiro.

Inicialmente definimos as estruturas necessárias, após a leitura da instância, como descrito nas versões anteriores. Repetimos o mesmo processo descrito na Versão 4.2.2, com uma exceção e destaque para a chamada da função *HEURIST_EPM_BLOSSOM* na linha 24, que substitui a função *HEURIST_EPM* da versão anterior. As linhas 5-9 descrevem o processo utilizado dentro da nova função, de acordo com a Versão 2.2.2 apresentada na Seção 2.7.

Código-fonte 8 – Versão 4.2.4

```
1  #include<ilcplex/ilocplex.h>
2  #include<iostream>
3  ILOSTLBEGIN
4  using namespace std;
5  HEURIST_GULOSA(double** C_Ma, int n, IloArray<IloNumArray>& val_x_RL,
       double** val_x_HL){
6      int p;
7      \\A variavel (p) recebe a quantidade de arestas presentes na solucao
           da relaxacao lagrangiana, ou seja, de variaveis (x_RL) com valor 1
8
9      int** Aresta = new int*[p];
10     double* Custo = new double*[p];
11     for (int i = 0; i < p; i++)
12         Aresta[i] = new double[2];
13
14     int k = 0;
15     for (int i = 0; i < n; i++) {
16         for (int j = i + 1; j < n; j++) {
17             if (C_Ma[i][j] >= 0 && val_x_RL[i][j] > 0.9999) {
18                 Aresta[k][0] = i;
19                 Aresta[k][1] = j;
20                 Custo[k] = C_Ma[i][j];
21                 k++;
22             }
23         }
24     }
25
26     ORD_MERGESORT(Aresta, Custo, 0, p-1, 3);
```

```
27
28      bool *vertice = new bool[n];
29      for (int i = 0; i < n; i++)
30          vertice[i] = false;
31
32      double ValObj = 0;
33      for (int i = 0; i < p; i++) {
34          if (!vertice[Aresta[i][0]] && !vertice[Aresta[i][1]]) {
35              vertice[Aresta[i][0]] = true;
36              vertice[Aresta[i][1]] = true;
37              val_x_HL[ Aresta[i][0] ] [ Aresta[i][1] ] = 1;
38              ValObj += Custo[i];
39          }
40      }
41
42      return ValObj;
43  }
44
45  int main(){
46
47      \\Repetir os passos da Versao 4.2.3, substituindo a heuristica
            HEURIST_EPM_BLOSSOM(linha 24) por HEURIST_GULOSA
48      val_HL = HEURIST_GULOSA(C_Ma, n, val_x_RL, val_x_HL); \\ LOWER BOUND
49
50      \\Inquisicao dos limites
51
52      env_RL.end();
53      env.HL.end();
54      return 0;
55  }
```

Na Versão 4.2.4, apresentada acima, mantemos o uso do método de subgradientes, porém substituímos a heurística, que agora busca encontrar um emparelhamento máximo (aproximado) de forma gulosa, para assim obter soluções viáveis com menor esforço computacional.

A nova heurística é implementada pela função *HEURIST_GULOSA*, descrita com em detalhes nas linhas 5-43. O procedimento primeiramente ordena, de forma não crescente, os custos originais das arestas presentes na solução dada pela relaxação lagrangiana, armazenada no

vetor *val_x_RL*. Essas arestas e seus pesos originais são armazenados na matriz bidimensional *Aresta* e o vetor *Custo* (linhas 9-24), e a função *ORD_MERGESORT* faz a ordenação desejada (linha 26). A escolha de uma matriz e um vetor como estruturas de dados deve-se à praticidade de visualização e facilidade no processo de ordenação, que utiliza o método do *Merge Sort*.

Em seguida, determinamos um emparelhamento dentro desse conjunto de arestas, escolhendo e analisando uma aresta por vez, segundo a ordenação, para compor a solução viável. Um vetor booleano *vertice* de tamanho $n = |V|$ é usado para indicar se cada vértice $i \in V$ já está coberto pelo emparelhamento (*vertice[i]=true*) ou não (*vertice[i]=false*). Nas linhas 33-40, mostramos este processo, percorrendo toda a matriz *Aresta* e verificando se ambos os vértices da aresta em análise estão como *false*. Nesse caso, a aresta definida pelo par de vértices ainda não coberto é acrescentada à solução viável (mantida pela matriz *val_x_HL*) e seu custo adicionado ao total (armazenado na variável *ValObj*). Ao final da função, retornamos o valor do limite inferior obtido na iteração.

Código-fonte 9 – Versão 4.2.5

```
1  #include<ilcplex/ilocplex.h>
2  #include<iostream>
3  ILOSTLBEGIN
4  using namespace std;
5  HEURIST_GULOSA_CL(double** C_Ma, int n, IloArray<IloNumArray>& val_x_RL,
       double** val_x_HL, double *val_lambda){
6      int p;
7      \\A variavel (p) recebe a quantidade de arestas presentes na solucao
           da relaxacao lagrangeana, ou seja, as variaveis (x_RL) com valor 1
8      \\Definicao da matriz (Aresta) e o vetor (Custo), como descrito na
           Versao 4.2.4 linhas 6-12
9
10     int k = 0;
11     for (int i = 0; i < n; i++) {
12         for (int j = i + 1; j < n; j++) {
13             if (C_Ma[i][j] >= 0 && val_x_RL[i][j] > 0.9999) {
14                 Aresta[k][0] = i;
15                 Aresta[k][1] = j;
16                 Custo[k] = C_Ma[i][j] - val_lambda[i] - val_lambda[j];
17                 k++;
18             }
```

```
19            }
20        }
21        \\Realizacao do procedimentos descritos na Versao 4.2.4 linhas 26-40
22
23        return ValObj;
24    }
25
26    int main(){
27
28        \\Repetir os passos da Versao 4.2.3, substituindo apenas a heuristica
              HEURIST_EPM_BLOSSOM(linha 24) por HEURIST_GULOSA_CL
29        val_HL = HEURIST_GULOSA_CL(C_Ma, n, val_x_RL, val_x_HL, val_lambda);
             \\LOWER BOUND
30
31        \\Inquisicao dos limites
32
33        env_RL.end();
34        env.HL.end();
35        return 0;
36    }
```

Nesta última versão, realizamos os mesmos procedimentos descritos pela Versão 4.2.4, com uma pequena variação: em lugar dos custos originais, os custos lagrangianos são usados na ordenação. Essa mudança se reflete na linha 16, onde cada entrada do vetor *Custo* recebe o *custo lagrangiano* em vez do custo original. Logo, uma ordenação diferente das arestas será dada nesta versão em comparação a anterior e, por consequência, uma solução viável possivelmente diferente.

4.5 Experimentos computacionais

Para os testes, foram gerados aleatoriamente 24 grafos base, com 50 ou 100 vértices e quantidade de arestas definida pela fórmula $\frac{|V(G)|*(|V(G)|-1)}{2} * p$, onde p, que indica a probabilidade de existência de aresta, foi dado pelos seguintes valores:

- 0.25 ou 25%;
- 0.50 ou 50%;
- 0.75 ou 75%.

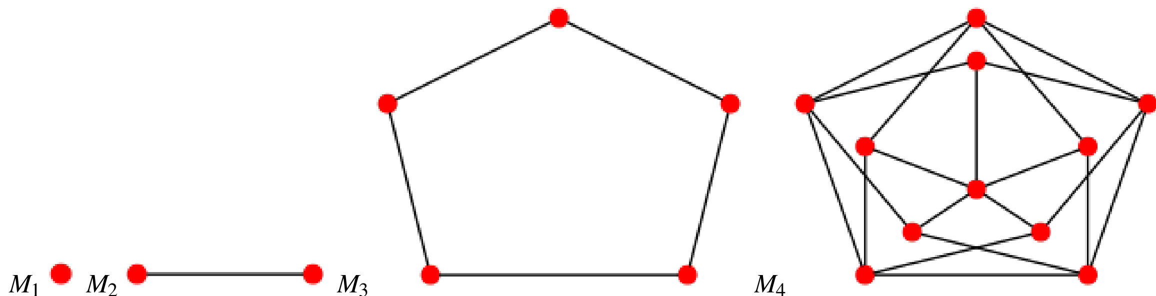
Para a geração de conflitos, usamos duas estratégias. Na primeira, definimos uma probabilidade q de existência de conflito entre duas arestas que não compartilham vértice, e adotamos os seguintes valores:

- 0.005 ou 0.5%;
- 0.010 ou 1%;
- 0.015 ou 1.5%.

Dessa maneira, para cada par de arestas não adjacentes G, geramos, com probabilidade q, um conflito. As instâncias geradas dessa forma serão identificadas como match-$< n >$-$< p >$-$< q >$.

A segunda estratégia consiste em gerar grafos de conflito que sejam de *Mycielski* (MYCIELSKI, 1955). Tais grafos podem ser definidos de forma recursiva. O primeiro e segundo grafos de *Mycielski* (M_1 e M_2) são exatamente um vértice e uma aresta, respectivamente. Para $k \geq 3$, o k-ésimo grafo da família é construído da seguinte forma: toma-se uma cópia idêntica de M_{k-1}, uma segunda cópia de M_{k-1} sem qualquer aresta e um vértice adicional. Esse vértice é ligado a todos da cópia sem arestas. Além disso, cada vértice dessa cópia se liga também aos vértices da cópia idêntica que são vizinhos ao seu correspondente nela. A Figura 13 ilustra essa construção.

Figura 13 – Os primeiros grafos de *Mycielski* ilustrados



Fonte: Elaborada pelo autor.

Para $k \geq 2$, o número de vértices e arestas de M_k é

$$|V(M_k)| = 3 \cdot 2^{k-2} - 1 \quad \text{e} \quad |E(M_k)| = \frac{1}{2}(7 \cdot 3^{k-2} - 6 \cdot 2^{k-2} + 1).$$

Note que $k = \log_2 \frac{|V(M_k)|+1}{3} + 2$.

Esses grafos não possuem triângulos (as maiores cliques são arestas). Dessa maneira, as restrições de conflito não podem ser fortalecidas como restrições de clique no grafo de conflitos.

Para utilizarmos esta ideia na formação de conflitos para nossas instâncias, dado o número $m = |E(G)|$ de arestas do grafo sobre o qual vamos determinar o emparelhamento, determinamos a ordem k do grafo de *Mycielski* pela fórmula $k = \lfloor \log_2 \frac{m+1}{3} \rfloor + 2$. Sabendo a ordem do grafo de *Mycielski* ($G_M = (E, E_M)$), assumimos que um vértice de G_M irá representar uma aresta de G e uma aresta de E_M irá representar um conflito entre as arestas correspondentes de G. As instâncias geradas com essa segunda estratégia serão denominadas match-<n>-<p>(i)_myci, de modo a identificar que se trata da i-ésima ($i = 1, 2, 3, ...$) instância relativa ao par (n, p). O mapeamento dos vértices de G_M em arestas de G ocorre segundo a ordem de leitura dos grafos. Por exemplo, caso $k = 4$, G_M seria representado pelo grafo M_4 da Figura 13. Então, cada um de seus 11 vértices, selecionado segundo a ordem de leitura do arquivo que contém M_4, seria mapeado em exatamente uma aresta diferente de G, seguindo a ordem de leitura dessas arestas no arquivo de entrada que armazena G. Cada uma das 20 arestas de M_4 definiria um conflito entre o par de arestas correspondentes em G.

Para as implementações, utilizamos o mesmo ambiente descrito na Seção 2.8 com um limite máximo de tempo de uma hora (3600 segundos) estabelecido para cada instância. Um resumo dos resultados dos experimentos realizados com todas as versões pode ser visto nas tabelas 2 e 3. Além do nome da instância e do valor de sua solução ótima, apresentamos o tempo (em segundos) demandado por cada uma das versões para a obtenção dos limites. Apresentamos também o cálculo da distância relativa entre cada limite (superior e inferior) encontrado e o valor ótimo da instância, dados pelas fórmulas $100\frac{UB-OPT}{OPT}$ e $100\frac{OPT-LB}{OPT}$.

Observamos na Tabela 2 os resultados obtidos pelos experimentos em busca de bons limites superiores e inferiores. Primeiro, podemos notar que os limites inferiores gerados pelas diferentes heurísticas têm qualidade similar, especialmente aqueles derivados a partir do uso de subgradientes. Analisando os desvios, os *gaps* percentuais médios ficaram em torno de 20% e podem ser considerados altos. Com respeito aos limites superiores, todas as versões produziram *gaps* percentuais médios bastante baixos, em torno de 0,6%. Destacamos ainda que o esforço computacional dispendido pelas versões que utilizam a relaxação lagrangiana é bem superior àquele da relaxação linear (Versão 4.1), não justificando a melhoria na qualidade do limite superior gerado. O melhor *gap* médio dessas versões foi 0.548%, que é comparável ao da relaxação linear (0.63%).

Tais resultados sugerem que, pelo menos para as instâncias testadas, o uso da relaxação linear seria mais adequado como forma de calcular um limite superior dentro de

um método exato. Entretanto, ainda podemos considerar a possibilidade de que a pequena superioridade da relaxação lagrangiana venha a se tornar mais significativa ao longo da árvore de B&B, o que poderia levar a um processo de poda mais efetivo e reduzir o número de ramificações. Além disso, considerando o aspecto didático do nosso trabalho, vale a pena analisar como se daria a implementação da relaxação lagrangiana incorporada a um B&B. No Capítulo 5, esse processo será melhor detalhado.

Para esse fim, vamos escolher uma das versões 4.2 consideradas neste capítulo. Analisando a Tabela 3, podemos afirmar que a Versão 4.2.2 apresentou desempenho ligeiramente superior, pois o desvio percentual de seu limite inferior (GapLB) foi o melhor dentre os demais e seu GapUB foi praticamente igual ao melhor (da Versão 4.2.5). Além disso, seu tempo de computação médio foi o menor, como mostra a Tabela 2. Lembramos que a Versão 4.2.2 implementa a relaxação lagrangiana utilizando o método de subgradientes, com o uso do modelo EPM (2.1)-(2.3) como heurística lagrangiana.

Tabela 2 – Problema de Emparelhamento de Peso Máximo com Conflitos - Resultados Computacionais Obtidos Pelas Versões 4.1, 4.2.1, 4.2.2, 4.2.3, 4.2.4 e 4.2.5

		Versão 4.1		Versão 4.2.1			Versão 4.2.2			Versão 4.2.3			Versão 4.2.4			Versão 4.2.5		
Instâncias	**Ótimo**	**UB**	**Tempo**	**UB**	**LB**	**Tempo**	**UB**	**LB**	**Tempo**	**UB**	**LB**	**Tempo**	**UB**	**LB**	**Tempo**	**UB**	**LB**	**Tempo**
match-50-0.5-0.005	363,960	**364,575**	**0,027**	**364,575**	**361,641**	1629,910	**364,575**	358,470	34,815	**364,575**	358,470	46,618	**364,575**	358,470	19,233	**364,575**	353,663	10,241
match-50-0.5-0.01	350,514	**353,970**	**0,029**	**353,970**	335,882	2067,350	**353,970**	**342,643**	29,975	**353,970**	**342,643**	21,862	**353,970**	**342,643**	21,953	**353,970**	342,183	14,037
match-50-0.5-0.015	363,272	**364,945**	**0,028**	**364,945**	356,303	3100,680	**364,945**	360,492	34,024	**364,945**	360,492	36,031	**364,945**	**360,517**	23,950	**364,945**	360,492	17,194
match-50-0.75-0.005	410,238	**410,238**	**0,041**	410,373	401,290	3600	**410,238**	**410,238**	33,594	**410,238**	**410,238**	39,374	**410,238**	**410,238**	24,190	**410,238**	**410,238**	21,557
match-50-0.75-0.01	391,476	**392,113**	**0,061**	**392,113**	379,209	3600	**392,113**	**380,093**	83,560	**392,113**	**380,093**	167,415	**392,113**	379,209	40,784	**392,113**	377,536	33,416
match-50-0.75-0.015	388,682	**389,510**	**0,142**	**389,510**	189,686	3600	**389,510**	379,451	47,839	**389,510**	379,451	81,506	**389,510**	**379,689**	29,022	**389,510**	379,451	35,865
match-100-0.25-0.005	724,917	**725,516**	**0,142**	**725,516**	675,738	3600	**725,516**	**722,364**	47,376	**725,516**	**722,364**	112,569	**725,516**	**722,364**	31,984	**725,516**	720,729	29,341
match-100-0.25-0.01	705,998	**713,305**	**0,323**	**713,305**	351,136	3600	**713,305**	**683,718**	52,369	**713,305**	**683,718**	200,171	**713,305**	667,046	43,632	**713,305**	682,325	55,328
match-100-0.25-0.015	701,471	711,565	**0,546**	711,565	380,292	3600	711,208	641,262	83,340	711,208	641,262	163,331	711,305	**655,756**	675,164	**711,126**	640,008	1073,810
match-100-0.5-0.005	785,313	**787,090**	**6,058**	**787,090**	298,709	3600	**787,090**	**770,949**	103,530	**787,090**	**770,949**	391,106	**787,090**	769,403	154,285	**787,090**	769,649	745,077
match-100-0.5-0.01	796,814	**802,417**	**18,085**	**802,417**	420,970	3600	**802,417**	**511,355**	3600	**802,417**	**511,355**	3600	**802,417**	459,094	3600	**802,417**	459,094	3600
match-100-0.5-0.015	772,845	**781,517**	**19,919**	**781,517**	**404,657**	3600	**781,517**	**404,657**	3600	**781,517**	**404,657**	3600	**781,517**	**404,657**	3600	**781,517**	402,529	3600
match-100-0.75-0.005	837,255	**837,633**	**30,032**	**837,633**	**291,671**	3600	**837,633**	**291,671**	3600	**837,633**	**291,671**	3600	**837,633**	**291,671**	3600	**837,633**	**291,671**	3600
match-100-0.75-0.01	816,471	**822,139**	**73,855**	**822,139**	**423,672**	3600	**822,139**	**423,672**	3600	**822,139**	**423,672**	3600	**822,139**	**423,672**	3600	**822,139**	**423,672**	3600
match-100-0.75-0.015	825,451	834,291	**72,970**	**832,234**	**346,142**	3600	**832,234**	**346,142**	3600	**832,234**	**346,142**	3600	**832,234**	**346,142**	3600	**832,234**	**346,142**	3600
match-100-0.5(1)_myci	791,341	796,660	**6,717**	796,660	723,699	3600	796,246	**775,225**	191,265	796,246	**775,225**	271,737	**796,199**	774,550	221,449	796,336	773,624	199,417
match-100-0.5(2)_myci	780,511	796,225	**7,056**	796,225	**746,000**	3600	794,428	693,361	227,021	794,428	693,361	279,218	**794,351**	680,768	213,785	794,418	681,764	239,656
match-100-0.5(3)_myci	788,323	795,527	**7,052**	795,527	713,801	3600	795,190	**715,196**	202,385	795,190	**715,196**	2080,720	795,204	679,135	201,986	**795,095**	652,023	198,305
match-100-0.5(4)_myci	780,410	791,755	**5,826**	791,755	**727,328**	3600	791,530	698,583	194,698	791,530	698,583	329,213	791,546	685,538	197,449	**791,527**	693,504	201,436
match-100-0.75(1)_myci	817,351	822,647	**21,203**	822,626	**780,846**	3600	822,141	731,189	413,353	**822,141**	731,189	969,005	**822,140**	668,963	448,670	822,176	686,114	505,959
match-100-0.75(2)_myci	835,483	851,934	**19,933**	851,934	**749,757**	3600	850,627	736,045	483,253	850,627	736,045	611,674	850,646	733,145	509,957	**850,554**	732,890	507,705
match-100-0.75(3)_my	815,285	829,469	**18,758**	829,469	718,613	3600	825,926	**756,045**	455,067	825,926	**756,045**	635,931	**825,913**	691,107	511,597	825,920	736,200	498,481
match-100-0.75(4)_myci	823,122	837,635	**16,919**	837,635	**725,803**	3600	**834,210**	708,177	527,953	**834,210**	708,177	3600	834,217	694,555	521,013	834,232	718,710	519,396
match-100-0.75(5)_myci	819,544	824,969	**23,157**	824,969	**725,556**	3600	**823,636**	724,185	459,527	**823,636**	724,185	623,463	823,644	701,698	414,395	823,644	714,319	513,601
Tempo Médio			14,537			3433,248			904,373			1194,206			929,354			975,826

Versão 4.1: Relaxação Linear do modelo EPMRC;
Versão 4.2.1: Relaxação Lagrangiana: Método de Geração de Cortes com Heurística Lagrangiana EPM;
Versão 4.2.2: Relaxação Lagrangiana: Método de Subgradientes com Heurística Lagrangiana EPM;
Versão 4.2.3: Relaxação Lagrangiana: Método de Subgradientes com Heurística Lagrangiana EPM por restrições de blossom;
Versão 4.2.4: Relaxação Lagrangiana: Método de Subgradientes com Heurística Lagrangiana Gulosa;
Versão 4.2.5: Relaxação Lagrangiana: Heurística Lagrangiana Gulosa por Custos Lagrangianos.

Tabela 3 – Problema de Emparelhamento de Peso Máximo com Conflitos - Qualidade dos Limites Inferiores e Superiores Obtidos Pelas Versões 4.1, 4.2.1, 4.2.2, 4.2.3, 4.2.4 e 4.2.5

	Versão 4.1	**Versão 4.2.1**		**Versão 4.2.2**		**Versão 4.2.3**		**Versão 4.2.4**		**Versão 4.2.5**	
Instâncias	**GapUB**	**GapUB**	**GapLB**	**GapUB**	**GapLB**	**GapUB**	**GapLB**	**GapUB**	**GapLB**	**GapUB**	**GapLB**
match-50-0.5-0.005	**0,169**	**0,169**	**0,637**	**0,169**	1,508	**0,169**	1,508	**0,169**	1,508	**0,169**	2,829
match-50-0.5-0.01	**0,986**	**0,986**	4,174	**0,986**	**2,246**	**0,986**	**2,246**	**0,986**	**2,246**	**0,986**	2,377
match-50-0.5-0.015	**0,461**	**0,461**	1,918	**0,461**	0,765	**0,461**	0,765	**0,461**	**0,758**	**0,461**	0,765
match-50-0.75-0.005	**0,000**	0,033	2,181	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**
match-50-0.75-0.01	**0,163**	**0,163**	3,134	**0,163**	**2,908**	**0,163**	**2,908**	**0,163**	3,134	**0,163**	3,561
match-50-0.75-0.015	**0,213**	**0,213**	51,198	**0,213**	2,375	**0,213**	2,375	**0,213**	**2,314**	**0,213**	2,375
match-100-0.25-0.005	**0,083**	**0,083**	6,784	**0,083**	**0,352**	**0,083**	**0,352**	**0,083**	**0,352**	**0,083**	0,578
match-100-0.25-0.01	**1,035**	**1,035**	50,264	**1,035**	**3,156**	**1,035**	**3,156**	**1,035**	5,517	**1,035**	3,353
match-100-0.25-0.015	1,439	1,439	45,786	1,388	8,583	1,388	8,583	1,402	**6,517**	**1,376**	8,762
match-100-0.5-0.005	**0,226**	**0,226**	61,963	**0,226**	**1,829**	**0,226**	**1,829**	**0,226**	2,026	**0,226**	1,995
match-100-0.5-0.01	**0,703**	**0,703**	47,168	**0,703**	**35,825**	**0,703**	**35,825**	**0,703**	42,384	**0,703**	42,384
match-100-0.5-0.015	**1,122**	**1,122**	**47,641**	**1,122**	**47,641**	**1,122**	**47,641**	**1,122**	**47,641**	**1,122**	47,916
match-100-0.75-0.005	**0,045**	**0,045**	**65,163**	**0,045**	**65,163**	**0,045**	**65,163**	**0,045**	**65,163**	**0,045**	**65,163**
match-100-0.75-0.01	**0,694**	**0,694**	**48,109**	**0,694**	**48,109**	**0,694**	**48,109**	**0,694**	**48,109**	**0,694**	**48,109**
match-100-0.75-0.015	1,071	**0,822**	**58,066**	**0,822**	**58,066**	**0,822**	**58,066**	**0,822**	**58,066**	**0,822**	**58,066**
match-100-0.5(1)_myci	0,672	0,672	8,548	0,620	**2,037**	0,620	**2,037**	**0,614**	2,122	0,631	2,239
match-100-0.5(2)_myci	2,013	2,013	**4,422**	1,783	11,166	1,783	11,166	**1,773**	12,779	1,782	12,652
match-100-0.5(3)_myci	0,914	0,914	9,453	0,871	**9,276**	0,871	**9,276**	0,873	13,851	**0,859**	17,290
match-100-0.5(4)_myci	1,454	1,454	**6,802**	**1,425**	10,485	**1,425**	10,485	1,427	12,157	**1,425**	11,136
match-100-0.75(1)_myci	0,648	0,645	**4,466**	**0,586**	10,542	**0,586**	10,542	**0,586**	18,155	0,590	16,056
match-100-0.75(2)_myci	1,969	1,969	**10,261**	1,813	11,902	1,813	11,902	1,815	12,249	**1,804**	12,279
match-100-0.75(3)_myci	1,740	1,740	11,857	1,305	**7,266**	1,305	**7,266**	**1,304**	15,231	**1,304**	9,700
match-100-0.75(4)_myci	1,763	1,763	**11,823**	**1,347**	13,965	**1,347**	13,965	1,348	15,619	1,350	12,685
match-100-0.75(5)_myci	0,662	0,662	**11,468**	**0,499**	11,636	**0,499**	11,636	0,500	14,379	0,500	12,839
Média	0,844	0,834	23,887	0,765	15,283	0,765	15,283	0,765	16,762	0,764	16,463
Desvio	0,630	0,626	23,465	0,550	19,726	0,550	19,726	0,550	19,810	0,548	19,849

5 MÉTODO EXATO

O CPLEX contém um conjunto vasto de métodos desenvolvidos para a resolução de problemas de programação linear, inteira, mista e, até certa extensão, programação quadrática. Em particular, esse resolvedor disponibiliza uma implementação do método Branch-and-Bound(B&B) que utiliza particionamento dicotômico (como forma de subdivisão do problema) e relaxação linear (para obtenção do limite dual). Em alguns casos, o CPLEX não é capaz de resolver eficientemente um problema. Nesses casos, duas saídas são possíveis:

- Alterar as configurações do método, através dos parâmetros disponibilizados.
- Interferir diretamente no método de resolução, criando funções do usuário a serem executadas ao longo da busca arborescente, possibilitando assim diversas formas de alterar a resolução padrão do CPLEX.

Neste capítulo, iremos definir brevemente o método de B&B e mostrar uma variação do método de resolução do CPLEX, alterando o modo de ramificação utilizado, assim como incorporando ao B&B a versão do método de relaxação lagrangiana escolhido na Seção 4.5.

5.1 Método branch-and-bound

Um problema de Programação Inteira é um problema que pode ser formulado por um modelo de otimização linear no qual algumas ou todas as variáveis pertencem ao conjunto dos números inteiros. Dada a semelhança com Programação Linear, um Problema Linear Inteiro (PLI) aparenta ter resolução fácil, no entanto trata-se de um problema NP-difícil em geral. Estratégias eficientes para busca de uma solução ótima são imprescindíveis para um menor esforço computacional e, por consequência, um tempo menor de resposta.

O método de B&B se baseia na técnica de divisão e conquista onde, dado o problema $\{Z = \max\ cx \mid x \in X\}$, realiza-se um particionamento do conjunto viável X em X_1,X_2,...,X_k, subconjuntos disjuntos de X, tal que $\bigcup_{i=1}^{k} X_i = X$ e $X_i \cap X_j = \emptyset$, $\forall i \neq j$. Tomando a solução ótima para cada subproblema $\{Z_i = \max\ cx \mid x \in X_i\}$, $\forall i = 1, 2, ..., k$, temos que a solução ótima do problema original é dada por $Z = \max\{Z_1, ..., Z_k\}$. Além disso, conhecidos um limite inferior LI para o problema e um limite superior LS_i para um subproblema i tais que $LI \geq LS_i$, ou seja, sabendo-se que $Z \geq LI \geq LS_i \geq Z_i$, então temos que $Z = \max\{Z_1, ..., Z_{i-1}, LI, Z_{i+1}, ..., Z_k\}$. Em outras palavras, o subproblema i não precisa ser resolvido, podendo ser descartado. Seguindo essa estratégia, o B&B realiza uma enumeração inteligente das soluções candidatas à solução ótima

do problema, efetuando recursivamente partições no espaço das soluções viáveis e calculando limites ao longo da enumeração, com o objetivo de evitar a busca em certas regiões onde o ótimo não se encontre. Dessa maneira, consegue assegurar uma prova de otimalidade, mesmo sem pesquisar todo o conjunto X. O descarte de um subproblema é chamado poda.

O limite inferior *LI* pode ser dado pelo valor de qualquer solução viável, isto é, podemos tomar $LS = c\hat{x}$, para qualquer $\hat{x} \in X$. Por outro lado, no caso de Programação Inteira, um limite superior para cada subproblema i pode ser fornecido pelo valor de sua relaxação linear. Além disso, caso a solução relaxada $\bar{x}$ seja inteira, ela é, na verdade, uma solução do subroblema. Do contrário, sendo $\bar{x}_j \notin \mathbb{Z}$ uma componente não inteira, o subproblema pode ser subdividido em dois: um obtido com a acréscimo da restrição $x_j \leq \lfloor \bar{x}_j \rfloor$ e outro, com a restrição $x_j \geq \lceil \bar{x}_j \rceil$. Essas estratégias de particionamento e cálculo de limites estão implementadas no CPLEX e são de uso direto.

Neste trabalho, ilustramos o uso de outras formas de particionamente e cálculo de limites, baseadas nos resultados do Capítulo 4, onde apresentamos a relaxação lagrangiana. Em geral, para descrição de um algoritmo de B&B, precisamos definir três operações:

1. Ramificação(*branch*);
2. Avaliação(*bound*);
3. Poda(*prune*).

É o que detalhamos nas próximas subseções.

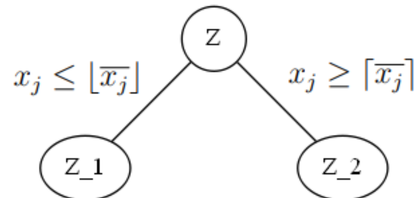
5.1.1 Ramificação

A operação de ramificação consiste em dividir o conjunto viável X nos subconjuntos $X_1, \ldots, X_k$. Em geral, isso é realizado com a acréscimo de restrições ao conjunto X para obter cada subconjunto X_i. As três principais formas implementadas no CPLEX são:

- Dicotomia em variável: Como descrito no início da seção, sendo $\bar{x}$ uma solução ótima não inteira da relaxação, a estratégia consiste em dividir o problema em dois subproblemas, conforme ilustra a Figura 14.
- Fixação de valores: Conhecido o domínio (finito) de uma variável $x_j \in \{v_1, v_2, ..., v_q\}$, isto é, o conjunto de valores que ela admite, podemos subdividir o problema como mostra a Figura 15.

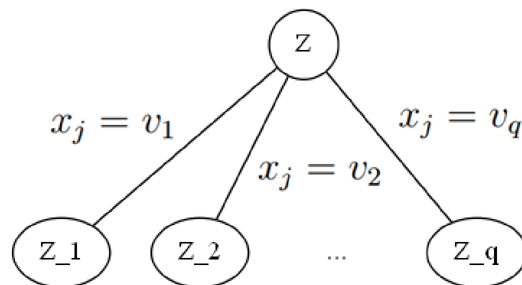
  Essa estratégia é bastante comum para variáveis binárias, coincidindo com a divisão dicotômica em variável.

Figura 14 – *Branch* dicotômico em variável



Fonte: Elaborada pelo autor.

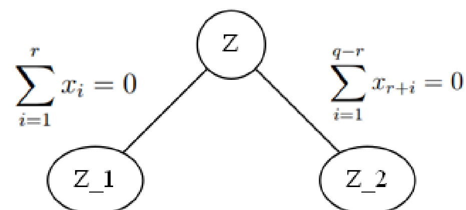
Figura 15 – *Branch* por fixação de variável



Fonte: Elaborada pelo autor.

- GUB(UB generalizado): Particionamento adequado para problemas com variáveis binárias, na presença de restrições do tipo $\sum_{i=1}^{q} x_i = 1$. Suponha que $\bar{x} \notin \mathbb{Z}^n$ seja a solução não inteira da relaxação linear. Logo, existe $r < q$ tal que $0 < \sum_{i=1}^{r} \bar{x}_i < 1$. Com isso, podemos particionar o problema como na Figura 16. Note que diferentes ordenações das variáveis e diferentes valores de r levam a particionamentos distintos. Usualmente, dada uma ordenação dos índices das variáveis, escolhe-se r para equilibrar os somatórios $\sum_{i=1}^{r} \bar{x}_i$ e $\sum_{i=1}^{q-r} \bar{x}_{r+i}$.

Figura 16 – GUB



Fonte: Elaborada pelo autor.

5.1.2 Avaliações

Após a ramificação do problema, cada novo subproblema obtido será avaliado. Para o caso de maximização, um limite superior será calculado. Esse cálculo pode ser feito imediatamente em seguida à ramificação ou quando o subproblema for efetivamente considerado para uma possível poda ou nova ramificação. O limite superior pode ser dado por algum tipo de relaxação ou problema dual. Adicionalmente, um limite inferior também poderá ser calculado, na tentativa de melhorar o limite inferior do problema original. Isso pode ser feito, por exemplo, com a aplicação de heurísticas.

O critério de escolha do próximo nó (subproblema) a ser avaliado pode influenciar fortemente no descarte de subregiões, devido à qualidade dos limitantes obtidos para o subproblema em foco. Alguns critérios bastante usuais são descritos abaixo:

- Melhor limite (*Best bound*): escolhe-se o nó com o menor limite superior (para o caso de maximização) a ser o próximo avaliado (note que esse nó não pode ser podado a partir dos limites dos outros nós); esse critério tende a enumerar menos nós.
- Busca em profundidade: escolhe-se como próximo nó a ser avaliado um daqueles originados a partir do nó corrente; esse critério tende a encontrar soluções viáveis de forma rápida.

5.1.3 Poda

Operação responsável por descartar regiões improdutivas, sendo por um dos seguintes motivos:

- Otimalidade: Quando o nó avaliado apresenta solução relaxada inteira e, portanto, foi resolvido exatamente;
- Limite: Quando o melhor valor relaxado obtido no nó (limite superior) é menor que a melhor solução inteira encontrada (limite inferior);
- Inviabilidade: Quando o nó avaliado apresenta inconsistência entre as restrições, descrevendo uma região sem pontos inteiros viáveis.

5.1.4 Branch-and-bound com relaxação linear e divisão dicotômica

O Algoritmo 4 apresenta o funcionamento básico de uma versão do método de B&B que usa relaxação linear para obtenção do limite superior e divisão dicotômica em variável para

particionar o problema. Esse é o processo padrão implementado pelo CPLEX, que pode ser organizado de diferentes formas, entre as quais as que podem ser vistas nos algoritmos 4 e 5 .

Algoritmo 4: branch_and_bound

Entrada: (*no_raiz*, *ListaNo*, *ListaLS*)

$\triangleright$ *no_raiz* - Nó que contém o problema original, *ListaNo* - Lista para guardar os subproblemas ativos, *ListaLS* - Lista para guardar os limites gerados pela relaxação linear dos subproblemas ativos

Saída: *LI* - Solução ótima

```
1  início
2     LI ← −∞
3     avaliaNo(no_raiz, LI, ListaNo, ListaLS)
4     enquanto ListaNo ≠ ∅ faça
5        no_{k*} ← Escolha de algum no da ListaNo
6        Seja x̄ ∉ ℤ^n solução relaxada de no_{k*}. Escolha variável x̄_j ∉ ℤ
7        No1 ← Criação de um nó adicionando a restrição x_j ≤ ⌊x̄_j⌋
8        avaliaNo (No1, LI, ListaNo, ListaLS)
9        No2 ← Criação de um nó adicionando a restrição x_j ≥ ⌈x̄_j⌉
10       avaliaNo (No2, LI, ListaNo, ListaLS)
11    fim
12    retorne LI
13 fim
```

O CPLEX oferece ferramentas para interferir no processo descrito pelo Algorimo 4. Usando o problema de Emparelhamento de Peso Máximo com Restrições de Conflitos (EPMRC) como exemplo, vamos substituir a relaxação linear no cálculo do limite superior (linha 2 do Algoritmo 5), assim como propor uma forma mais complexa de particionamento (linhas 7 e 9). Essas mudanças são detalhadas nas duas próximas seções.

5.2 Árvore de subgradientes

Observando os resultados apresentados nas tabelas 2 e 3 do capítulo anterior para o problema de EPMRC, percebemos que a Versão 4.1, que emprega Relaxação Linear, apresentou um tempo de resolução bastante inferior e limite superior comparável àqueles das melhores versões que usam Relaxação Lagrangiana (versões 4.2.2 e 4.2.4). Embora em algumas instâncias

Algoritmo 5: avaliaNo

Entrada: (no, $ListaNo$, $ListaLS$)

▷ no - subproblema a ser avaliado, LI - Melhor solução inteira encontrada, $ListaNo$ - Lista com nos ativos, $ListaLS$ - Lista para guardar os limites gerados pela relaxação linear dos nos ativos.

Saída:

```
1  início
2  |  LSno ← Relaxacao_linear(no)          ▷ LSno = −∞ se relaxação inviável
3  |  se LSno > LI então
4  |  |  Seja x̄ solução da relaxação linear
5  |  |  se x̄ não é solução inteira então
6  |  |  |  Adiciona no na ListaNo
7  |  |  |  Adiciona LSno na ListaLS
8  |  |  senão
9  |  |  |  LI ← LSno
10 |  |  |  para todo no_i ∈ ListaNo faça
11 |  |  |  |  se LS_i ≤ LI então
12 |  |  |  |  |  Remova o no_i da ListaNo
13 |  |  |  |  |  Remova o LS_i da ListaLS
14 |  |  |  |  fim
15 |  |  |  fim
16 |  |  fim
17 |  fim
18 fim
```

o valor da Relaxação Linear seja um pouco pior, seu tempo de computação bem menor parece sugerir que o uso da Relaxação Lagrangiana não será compensador. Mesmo assim, vamos apresentar e testar um algoritmo de B&B para EPMRC, baseado em Relaxação Lagrangiana.

Como observamos nas tabelas 2 e 3, o Método de Subgradientes se mostrou mais eficiente que o Método de Geração de Cortes para resolver o problema Lagrangiano. Por isso, ele será usado para obter o limite superior em cada nó da árvore de busca. Sabemos que esse método necessita de um vetor inicial de multiplicadores lagrangianos λ_0 para iniciar o processo iterativo. Inspirados no trabalho de (ANDRADE; FREITAS, 2008), vamos usar o que os autores chamam de Árvore de Subgradientes. No nó raiz, o vetor λ_0 será o valor ótimo, na relaxação linear, das

variáveis duais associadas às restrições dualizadas, ou seja, as restrições de emparelhamento. Após a convergência, o vetor de multiplicadores finais $\bar{\lambda}$ é passado para cada nó filho, que os tomará como multiplicadores iniciais. Recursivamente, cada nó gerado vai ser ramificado (a menos que seja podado) e usa os multiplicadores finais do seu pai como ponto de partida para a resolução do seu próprio Dual Lagrangiano.

5.3 Ramificação

Resolvido o dual Lagrangiano de um certo nó, chega-se a um conjunto de multiplicadores finais $\bar{\lambda}$ e à solução $\bar{x}$ de $RL(\bar{\lambda})$. Ver modelo (4.15)-(4.17). Conforme expressão (4.11), o valor de $RL(\bar{\lambda})$ é

$$V[RL(\bar{\lambda})] = \sum_{e \in E} c_e \bar{x}_e + \sum_{v \in V} \left[\bar{\lambda}_v \left(1 - \sum_{e \in \delta(v)} \bar{x}_e \right) \right].$$

A condição (4.9) estabelece que o valor dessa relaxação coincide com o do problema original se:

$$\sum_{e \in \delta(v)} \bar{x}_e \leq 1, \quad \forall v \in V \tag{5.1}$$

e

$$\sum_{v \in V} \left[\bar{\lambda}_v \left(1 - \sum_{e \in \delta(v)} \bar{x}_e \right) \right] = 0. \tag{5.2}$$

Nesse caso o subproblema pode ser podado. Caso contrário, é preciso ser ramificado. Note, porém, que $\bar{x}$ é um vetor inteiro, de modo que não podemos efetuar a subdivisão dicotômica tradicional.

Procurando potencializar o limite superior dos filhos, vamos usar a seguinte estratégia de ramificação. Primeiro, encontramos o vértice v^* tal que:

$$\left| \bar{\lambda}_{v^*} \left(1 - \sum_{e \in \delta(v^*)} \bar{x}_e \right) \right| = \max_{v \in V} \left\{ \left| \bar{\lambda}_v \left(1 - \sum_{e \in \delta(v)} \bar{x}_e \right) \right| \right\} \tag{5.3}$$

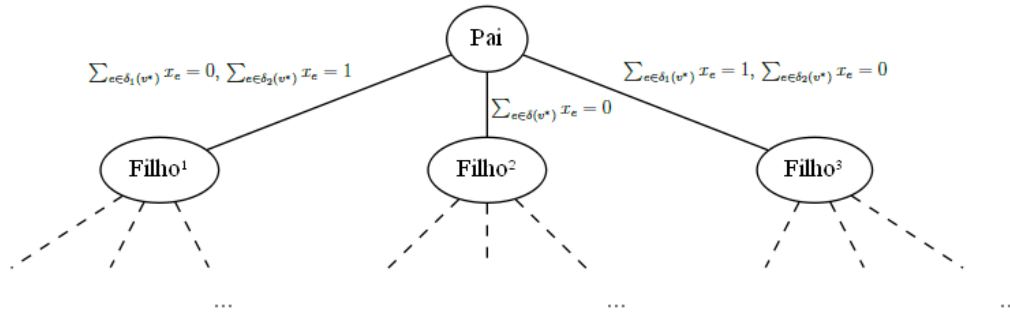
Encontrado o vértice v^*, realizamos uma divisão das arestas nele incidentes $\delta(v^*)$ em dois conjuntos disjuntos $\delta(v^*) = \delta_1(v^*) \cup \delta_2(v^*)$ que procurem equilibrar

$$\sum_{e \in \delta_1(v^*)} \bar{x}_e \quad \text{e} \quad \sum_{e \in \delta_2(v^*)} \bar{x}_e. \tag{5.4}$$

Com isso, geramos três nós filhos, como ilustra a Figura 17

Cada subproblema filho é definido a partir do pai com o acréscimo de uma das seguintes restrições:

Figura 17 – Ramificação



Fonte: Elaborada pelo autor.

- Filho 1: $\sum_{e\in\delta_1(v*)} x_e = 0$, $\sum_{e\in\delta_2(v*)} x_e = 1$;
- Filho 2: $\sum_{e\in\delta(v*)} x_e = 0$;
- Filho 3: $\sum_{e\in\delta_1(v*)} x_e = 1$, $\sum_{e\in\delta_2(v*)} x_e = 0$.

As restrições acrescentadas a um nó filho passam a compor a relaxação lagrangiana a ele associada. Note que cada um dos filhos cumpre explicitamente a restrição $\sum_{e\in\delta(v^*)} x_e \leq 1$. Dessa forma, o multiplicador λ_{v^*} pode ser fixado em zero nos nós descendentes.

Vale destacar que a ramificação da Figura 17 tende a fixar muitas variáveis em zero. Com isso espera-se que os nós gerados apresentem uma melhora no limite superior da relaxação lagrangiana e que isso possa contribuir para a obtenção de uma árvore de busca menor.

5.4 Implementações

Assim como todos os testes realizados neste trabalho, apresentados anteriormente nos capítulos 2 e 4, utilizamos o CPLEX para implementação e avaliação dos métodos descritos para a resolução do problema inteiro de EPMRC. Como visto na Seção 4.5, a qualidade dos limites superiores obtidos pela relaxação lagrangiana, bem como o esforço computacional dispendido para isso, foram elevados, comparados à relaxação linear. Então, como estratégia para compensar o esforço computacional utilizado na Versão 4.2.2 da relaxação lagrangiana, propomos uma ramificação tripla que fixa um número grande de variáveis em zero, acrescida, ainda, da ideia de empregar o vetor λ referente ao menor limite superior obtido no nó pai como λ_0 dos nós filhos.

Implementamos e apresentamos abaixo a resolução do problema de EPMRC (3.1)-(3.4), utilizando a relaxação lagrangiana (4.15)-(4.17) com o método de subgradientes, incorporado a heurística lagrangiana do modelo de EPM (2.1)-(2.3). Exibimos as partes principais dos códigos em C_{++} e a descrição do processo, evitando redundâncias nas explicações, tendo em vista o que já foi apresentado.

Para essa implementação, novamente interferimos no *branch-and-cut* do CPLEX. Todavia, diferentemente dos capítulos anteriores, não fazemos uso de *callbacks* para esse fim. Aproveitamos a oportunidade para ilustrar a aplicação de métodos similares, porém ainda mais flexíveis, disponibilizados pelo resolvedor. Tais métodos, chamados *goals*, permitem ao usuário controlar fortemente o processo de busca, de modo que é fundamental possuir domínio completo dos mesmos para não incorrer em erros no processo de solução. Detalhes podem ser encontrados em (CPLEX, 2017).

Para um bom entendimento de como funcionam os *goals*, vale esclarecer que cada nó da árvore de busca mantém uma *pilha de goals*. Depois que o CPLEX resolve a relaxação linear associada ao nó, ele desempilha os *goals* da pilha e os executa a partir daquele que está no topo.

Dentre os *goals* pré-definidos, que usaremos neste trabalho, encontram-se:

- *OrGoal*: Responsável por controlar a ramificação no processo de busca, tem o papel de criar subnós (nós filhos) do nó atual. Esta função exige de 2 até 6 argumentos, que podem ser do tipo *Goal* ou, no caso de ramificação em restrições, objetos do tipo *IloRange* ou *IloRangeArray*. Para cada um de seus argumentos, criará um subnó, a ser inicializado com uma cópia da pilha remanescente do pai. Em seguida, o nó pai é descartado. Quando um nó filho for avaliado, o *goal* ou restrição que o gerou estará no topo de sua pilha e será o(a) primeiro(a) a ser processado(a).
- *AndGoal*: Recebe como argumentos outros *goals* e os empilha na ordem reversa. Sendo assim, tais *goals* são processados na ordem em que são passados na chamada da função *AndGoal*.
- *FailGoal*: empilha um *goal* que, ao ser executado, poda o nó corrente.
- *SolutionGoal*: cria um *goal* que procura gerar soluções viáveis para o nó corrente, fornecendo uma lista de variáveis e que valores devem receber. Vale esclarecer que tal solução só é aceita após verificar que ela é compatível com o modelo (junto com cortes) e os *goals* associados ao nó, tando aqueles já processados quando aqueles ainda presentes na pilha do nó. Para avaliar essa compatibilidade, o CPLEX primeiro cria um subnó, que herda a pilha do seu pai, onde também são empilhados cortes locais fixando as variáveis da lista aos seus valores informados. Em particular, a solução será descartada se não for compatível com as ramificações que geraram o nó. Caso seja confirmada a viabilidade da solução, esta poderá ser candidata a melhor solução inteira encontrada.

Ao processar um nó, o CPLEX se mantém desempilhando e executando *goals* até encontrar um OrGoal ou FailGoal ou que a pilha se torne vazia. Neste último caso, ele continuará com suas estratégias de busca pré-estabelecidas.

Concretamente, considere o exemplo abaixo que cria dois subproblemas associados ao particionamento $x \leq \lfloor k \rfloor$ e $x \geq \lceil k \rceil$, onde $k \notin \mathbb{Z}$ é o valor corrente de uma variável inteira x:

return AndGoal (OrGoal (x $\leq$ IloFloor(k), x $\geq$ IloFloor(k)+1), this);

Nesse caso, o *AndGoal* primeiro empilha *this* (o *goal* em execução) e depois empilha *OrGoal*. Este último, no topo da pilha, será executado, criando dois nós e copiando a pilha remanescente em cada um deles. Cada um desses dois novos nós terá *this* no topo da pilha nesse momento. Em seguida, *OrGoal* empilha a desigualdade $x \leq IloFloor(k)$ na pilha de um deles e $x \geq IloFloor(k)+1$ na do outro. Depois que essas desigualdades são processadas em seus respectivos nós, *this* estará no topo da pilha de cada um, de modo que os nós filhos manterão a estratégia de ramificação do nó pai.

Passamos agora a implementação, usando *goals*, do método exato descrito neste capítulo.

Código-fonte 10 – Versão Final

```
1  #include<ilcplex/ilocplex.h>
2  #include<iostream>
3  ILOSTLBEGIN
4  using namespace std;
5
6  class dadosNo{
7      public:
8          IloNumArray Melhorlambda; \\(lambda) associado ao melhor LS da
               Relax. Lagrangiana
9          IloArray<IloNumArray>  indices_vizi;\\ conjuntos de indices das
               restricoes de particionamento que geraram o no
10         int tam; \\qtd. elementos em (indices_vizi)
11         int profundidade;
12
13         dadosNo(){}
14
15         dadosNo(const dadosNo& dadosPai, const IloEnv& env, const int n){
16             this->profundidade = dadosPai.profundidade;
17             this->tam = dadosPai.tam;
```

```
18              this->indices_vizi = IloArray<IloNumArray>(env);
19              for (int i = 0; i < dadosPai.tam; i++)
20                  this->indices_vizi.add(dadosPai.indices_vizi[i]);
21              this->Melhorlambda = IloNumArray(env, n);
22              for(int i = 0; i < n; i++)
23                  this->Melhorlambda[i] = dadosPai.Melhorlambda[i];
24          }
25  };
26
27  class ModelHL{
28  public:
29      IloEnv          env;
30      IloModel        modelo;
31      IloRangeArray   Rest;
32      IloObjective    obj;
33      IloCplex        cplex;
34      IloArray<IloNumVarArray> x_HL;
35
36      ModelHL(const int n) {
37          modelo = IloModel(env);
38          Rest = IloRangeArray(env);
39          x_HL = IloArray<IloNumVarArray>(env, n);
40          obj = IloMaximize(env);
41          cplex= IloCplex(env);
42      }
43  };
44
45  ILOCPLEXGOAL6(MEU_GOAL, IloArray<IloNumVarArray>&, x, double**, C_Ma, int
        **, Gc, int, mc, ModelHL*, Heur, dadosNo*, herancaNoRecebido) {
46      int n = x.getSize();
47      IloCplex::Goal resultado = 0; \\Inicializacao da variavel resultado,
           que atualiza a saida do Goal
48      \\A relaxacao linear tem solucao inteira?
49      if (!isIntegerFeasible()) {
50          \\Caso a relaxacao linear nao seja inteira, realizaremos a
               relaxacao lagrangiana
51          dadosNo* herancaNo; \\Objeto que salva informacoes do No para
               passar como heranca aos Nos filhos
52
```

```
53      \\Definicao das estruturas de programacao como na Versao 4.2.1,
            Secao 4.4, com excecao do modelo DL
54      \\Montagem do modelo RL(linha 56) semelhante a Versao 4.2.1 da
            Secao 4.4
55
56      if (herancaNoRecebido == NULL) { \\No raiz
57          herancaNo = new dadosNo;
58          \\Inicializando lambdaNo.
59          herancaNo->Melhorlambda = IloNumArray(getEnv(), n);
60          herancaNo->profundidade = 0; herancaNo->tam = 0;
61
62          \\ Obter o valor das variaveis duais (correspondentes as
                restricoes de emparelhamento) do modelo original
63
64          \\O vetor (herancaNo->Melhorlambda) recebe inicialmente os
                valores duais associados as restricoes relaxadas
65
66      }else{ \\Nao eh No raiz, logo temos informacoes geradas pelo No
            Pai
67          herancaNo = new dadosNo(*herancaNoRecebido, getEnv(), n);
68          herancaNo->profundidade++;
69          int tamL = herancaNo->indices_vizi.getSize();\\Quant de
                restricoes serao herdadas do NO Pai
70          IloExpr rest(env_RL);
71          for (int i = 0; i < tamL; i++) {
72              int tamC = herancaNo->indices_vizi[i].getSize(); \\ quant
                    de indices kl referentes a variaveis Xkl presentes na
                    restricao i
73              for (int j = 0; j < tamC - 1; j += 2)
74                  rest += x_RL[ herancaNo->indices_vizi[i][j] ][
                        herancaNo->indices_vizi[i][j + 1] ];\\Elaborando a
                        expressao com o somatorio das variaveis Xkl
75              Rest_RL.add(rest == herancaNo->indices_vizi[i][herancaNo
                    ->indices_vizi[i].getSize() - 1]);\\Definindo o bound
                    da restricao(se = 1 ou = 0)
76              rest.clear();
77          }
78          modelo_RL.add(Rest_RL);
79          cplex_RL.extract(modelo_RL);
```

```
80        }
81
82        \\Realizar os procedimentos de atualizacao: (UB), pelo metodo do
             Subgradientes, e (LB), pela funcao Heurist_EPM, descrito na
             Secao 4.4, Versao 4.2.2(linhas:9-74)
83
84        IloNumVarArray vars(getEnv());
85        IloNumArray val_vars(getEnv()); \\ Heuristica Lagrangeana
86
87        for (int i = 0; i < n; i++) {
88            for (int j = i + 1; j < n; j++) {
89                if (C_Ma[i][j] >= 0) {
90                    vars.add(x[i][j]);
91                    \\valor_xij = Heur->xHL[i][j] (valor da variavel x[i
                         ][j] na solucao correspondente ao melhor LB, dado
                         pelo heuristica lagrangiana
92                    val_vars.add(valor_xij);
93                }
94            }
95        }
96
97        \\A variavel (vertice) sera definida para armazenar o vertice (v
             *) dado pela expressao(5.3)
98
99        double maxLB = \\Max{LB, getIncumbentObjValue()}
100
101       if (UB <= maxLB) {
102           if(LB > getIncumbentObjValue())
103               resultado = SolutionGoal(vars, val_vars);
104           else
105               resultado = FailGoal(getEnv());
106       }else{
107           IloExpr expr_vizinhanca1(getEnv()), expr_vizinhanca2(getEnv()
                 ), expr_vizinhanca(getEnv());
108
109           IloNumArray indices_vizi1_rest1(getEnv()),
                 indices_vizi1_rest2(getEnv()), indices_vizi2_rest1(getEnv
                 ()), indices_vizi2_rest2(getEnv()), indices_viziT_rest(
                 getEnv());
```

```
110
111         \\Serao definidas as expressoes para posteriormente
               determinarmos as ramificacoes dos proximos Nos
112         double cst1 = 0, cst2 = 0;
113         for (int i = 0; i < n; i++) {
114             if (i < vertice && C_Ma[i][vertice] >= 0) {
115                 if (cst1 <= cst2) {
116                     expr_vizinhanca1 += x[i][vertice];
117                     cst1 += C_Ma[i][vertice];
118                     indices_vizi1_rest1.add(i);indices_vizi1_rest1.
                           add(vertice);
119                     indices_vizi2_rest1.add(i);indices_vizi2_rest1.
                           add(vertice);
120                 }else{
121                     expr_vizinhanca2 += x[i][vertice];
122                     cst2 += C_Ma[i][vertice];
123                     indices_vizi1_rest2.add(i);indices_vizi1_rest2.
                           add(vertice);
124                     indices_vizi2_rest2.add(i);indices_vizi2_rest2.
                           add vertice);
125                 }
126                 indices_viziT_rest.add(i);indices_viziT_rest.add(
                       vertice);
127             }else{
128                 if (i > vertice && C_Ma[vertice][i] >= 0) {
129                     if (cst1 <= cst2) {
130                         expr_vizinhanca1 += x[vertice][i];
131                         cst1 += C_Ma[vertice][i];
132                         indices_vizi1_rest1.add(vertice);
                               indices_vizi1_rest1.add(i);
133                         indices_vizi2_rest1.add(vertice);
                               indices_vizi2_rest1.add(i);
134                     }else {
135                         expr_vizinhanca2 += x[vertice][i];
136                         cst2 += C_Ma[vertice][i];
137                         indices_vizi1_rest2.add(vertice);
                               indices_vizi1_rest2.add(i);
138                         indices_vizi2_rest2.add(vertice);
                               indices_vizi2_rest2.add(i);
```

```
139                         }
140                         indices_viziT_rest.add(vertice);
                                indices_viziT_rest.add(i);
141                     }
142                 }
143             }
144
145             indices_vizi1_rest1.add(0); indices_vizi1_rest2.add(1);
146             indices_vizi2_rest1.add(1); indices_vizi2_rest2.add(0);
147             indices_viziT_rest.add(0);
148
149             IloRangeArray Vizinhanca1(getEnv());
150             Vizinhanca1.add(expr_vizinhanca1 == 0);Vizinhanca1.add(
                    expr_vizinhanca2 == 1);
151
152             IloRangeArray Vizinhanca3(getEnv());
153             Vizinhanca3.addheranca(expr_vizinhanca1 == 1);Vizinhanca3.add
                    (expr_vizinhanca2 == 0);
154
155             expr_vizinhanca = expr_vizinhanca1 + expr_vizinhanca2;
156             IloRangeArray Vizinhanca2(getEnv());
157             Vizinhanca2.add(expr_vizinhanca == 0);
158
159             dadosNo* herancaNo1 = new dadosNo(*herancaNo, getEnv(), n);
160             herancaNo1->indices_vizi.add(indices_vizi1_rest1);
161             herancaNo1->indices_vizi.add(indices_vizi1_rest2);
162             herancaNo1->tam += 2;
163
164             dadosNo* herancaNo3 = new dadosNo(*herancaNo, getEnv(), n);
165             herancaNo3->indices_vizi.add(indices_vizi2_rest1);
166             herancaNo3->indices_vizi.add(indices_vizi2_rest2);
167             herancaNo3->tam += 2;
168
169             dadosNo* herancaNo2 = new dadosNo(*herancaNo, getEnv(), n);
170             herancaNo2->indices_vizi.add(indices_viziT_rest);
171             herancaNo2->tam += 1;
172
173             if(LB > getIncumbentObjValue())
174                 resultado = OrGoal(SolutionGoal(vars, val_vars),
```

```
175                                     AndGoal(Vizinhanca1, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo1)),
176                                     AndGoal(Vizinhanca2, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo2)),
177                                     AndGoal(Vizinhanca3, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo3))
178                                     );
179                 else
180                     resultado = OrGoal(
181                                     AndGoal(Vizinhanca1, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo1)),
182                                     AndGoal(Vizinhanca2, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo2)),
183                                     AndGoal(Vizinhanca3, MEU_GOAL(getEnv(), x,
                                            C_Ma, Gc, mc, herancaNo3))
184                                     );
185             }
186
187             env_RL.end();
188         }
189         return resultado;
190 }
191
192 int main() {
193
194             \\Leitura da instancia e criacao das matrizes C_Ma e Gc como na
                    Versao 4.1(linhas 6-8) da Secao 4.4
195             IloEnv env_EPMRC;
196             IloModel modelo_EPMRC(env_EPMRC);
197             IloRangeArray Rest_EPMRC(env_EPMRC);
198             IloArray<IloNumVarArray> x(env_EPMRC, n);
199             for(int i = 0; i < n; i++){
200                 x[i] = IloNumVarArray(env_EPMRC, n, 0, 1, ILOINT);
201             }
202             IloObjective obj_EPMRC(env_EPMRC);
203             IloExpr expr_obj(env_EPMRC);
204
205             \\Montagem do modelo EPMRC como na versao 4.1, Secao 4.4 (linhas
                    24-49)
```

```
206
207         ModelHL *Heur = new ModelHL(n);
208         for (int i = 0; i < n; i++) {
209             Heur->x_HL[i] = IloNumVarArray(Heur->env, n, 0, 1, ILOINT);
210             for (int j = i + 1; j < n; j++) {
211                 if (C_Ma[i][j] >= 0)
212                     Heur->modelo.add(Heur->x_HL[i][j]);
213             }
214         }
215         Heur->modelo.add(Heur->obj);
216         IloExpr expr_emp(Heur->env);
217
218         for (int i = 0; i < n; i++) {
219             for (int j = i + 1; j < n; j++) {
220                 if (C_Ma[i][j] >= 0) {
221                     expr_emp += Heur->x_HL[i][j];
222                 }
223             }
224             for (int j = i - 1; j >= 0; j--) {
225                 if (C_Ma[j][i] >= 0) {
226                     expr_emp += Heur->x_HL[j][i];
227                 }
228             }
229             Heur->Rest.add(expr_emp <= 1);
230             expr_emp.clear();
231         }
232         expr_emp.end();
233         Heur->modelo.add(Heur->Rest);
234         Heur->cplex.extract(Heur->modelo);
235
236         dadosNo *herancaNo = nullptr;
237         IloCplex cplex_EPMRC(env_EPMRC);
238
239         \\Desabilitando os cortes gerados pelo CPLEX no processo de B&C
240         cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::FlowCovers, -1);
241         cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::Cliques, -1);
242         cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::Gomory, -1);
243         cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::Covers, -1);
244         cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::ZeroHalfCut, -1)
```

```
          ;
245       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::Disjunctive, -1)
          ;
246       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::MIRCut, -1);
247       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::LiftProj, -1);
248       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::BQP, -1);
249       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::GUBCovers, -1);
250       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::Implied, -1);
251       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::LocalImplied,
          -1);
252       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::MCFCut, -1);
253       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::PathCut, -1);
254       cplex_EPMRC.setParam(IloCplex::Param::MIP::Cuts::RLT, -1);
255
256       \\Desabilitando o uso de heuristicas do CPLEX ao processo de B&C
257       cplex_EPMRC.setParam(IloCplex::Param::MIP::Strategy::
          HeuristicFreq, -1);
258
259       \\Desabilitando o uso do presolve do CPLEX ao processo de B&C
260       cplex_EPMRC.setParam(IloCplex::Param::Preprocessing::Presolve, 0)
          ;
261
262       cplex_EPMRC.extract(modelo_EPMRC);
263       cplex_EPMRC.solve(MEU_GOAL(env_EPMRC, x, C_Ma, Gc, mc, Heur,
          herancaNo));
264
265           \\Inquirir solucao
266
267       env_EPMRC.end();
268     }
269     return 0;
270 }
```

Começamos descrevendo o procedimento principal, que se inicia na linha 192. Após a leitura das instâncias e alocação dos dados nas matrizes de custo (C_{Ma}) e de conflitos (Gc), definimos nas linhas 195-203 as estruturas de programação para o modelo EPMRC. A montagem desse modelo é realizada em seguida, conforme indicado na linha 205.

Para a utilização da heurística lagrangiana (a mesma empregada na Versão 4.2.2 da

Seção 4.5), construímos o modelo EPM (2.1)-(2.3), usando um procedimento similar ao que foi descrito na Versão 2.1, Seção 2.7. Agora, porém, definimos uma classe *ModelHL*(linhas 27-43) para conter todas as estruturas de programação necessárias para a construção desse modelo. Na linha 207, temos a criação de um ponteiro *Heur* para essa classe. *Heur* referenciará os objetos definidos nas linhas 29-34, que serão devidamente inicializados pelo construtor descrito nas linhas 36-42. A montagem do modelo de EPM ocorre efetivamente nas linhas 208-234. Ele será passado como parâmetro para a função ILOCPLEXGOAL. Nesse ponto reside a motivação para essa mudança na construção do modelo EPM: o número de parâmetros da função ILOCPLEXGOAL é limitado a 6.

Na linha 236, temos a criação do objeto *herancaNo*, da classe *dadosNo*. A classe *dadosNo* (ver linhas 6-25) contém os atributos necessários para não somente ir atualizando o vetor λ(linha 8) ao longo da árvore de subgradientes, como também guardar referências aos índices das variáveis(linhas 9-10) que irão participar das restrições de particionamento (que definem os nós filhos) a serem acrescidas ao modelo da relaxação lagrangiana do nó corrente. Além disso, mantemos um indicador da profundidade do nó (linha 11), essencialmente para diferenciar o nó raiz dos demais (dentro de ILOGCPLEXGOALn a atualização dos atributos de *herancaNo* é diferente dependendo se a profundidade é ou não igual a 0). A inicialização dos atibutos da classe *dadosNo* é realizada pelo construtor definido nas linhas 15-24, tomando como base os dados herdados do nó pai.

Voltando para a função *main*, temos, a partir da linha 240, modificações nos parâmetros do CPLEX. Desabilitamos uma série de cortes utilizados constantemente(linhas 240-254), a influência da heurística(linha 257) e o pré-processamento(linha 260) realizado inicialmente no processo de B&C. Com essa desabilitação, queremos avaliar a influência da regra de ramificação implementada frente à ramificação dicotômica padrão do CPLEX. Essa comparação será melhor detalha na próxima seção.

Na linha 263, temos a resolução do modelo de EPMRC, usando o método *solve* do tipo *IloCplex*. Note que a indicação para uso do *goal MEU_GOAL* acontece via passagem de parâmetro do método *solve*, diferentemente de uma chamada a *callbacks*, que utiliza o método *use*. Por outro lado, podemos notar grande semelhança entre a estrutura de declaração da macro *ILOGCPLEXGOALn* (linha 45) e de declaração de *callbacks*(ver Versão 2.2.2 Seção 2.7).

Após a definição do cabeçalho de *ILOGCPLEXGOALn* (linha 45), temos a declaração e inicialização do objeto *resultado* do tipo *IloCplex::Goal*(linha 47), com valor nulo. Esse objeto

é o resultado a ser retornado por essa macro, que, portanto, também será um *goal*.

Como o processo de B&C do CPLEX resolve sempre uma relaxação linear no nó corrente, a linha 49 realiza a verificação de integralidade da solução obtida. Caso seja inteira, o CPLEX salvará a solução como um limite inferior; caso contrário, aplicaremos o método proposto neste capítulo, a partir da linha 51.

Primeiro, definiremos um objeto do tipo *dadosNo* para receber informações do objeto *herancaNoRecebido*, passado como parâmetro. Na primeira chamada do *goal*, *herancaNoRecebido* é nulo, devido à inicialização na linha 236. Sendo nulo (não há dados a serem herdados do nó pai), *herancaNo->melhorlambda* é inicializado com os valores duais da relaxação linear (linha 62). Por outro lado, se estivermos tratando de um caso onde *herancaNoRecebido* é diferente de nulo, então o nó atual não é raiz e, por isso, existem informações do objeto *herancaNoRecebido* a serem transferidas para o objeto *herancaNo* do nó corrente(linha 67). Entre essas informações estão os conjuntos de índices das restrições de particionamento que definem o nó em processamento. Elas devem ser incluídas na relaxação lagrangiana desse nó. Isso é realizado entre as linhas 71-79.

A seguir, como indicado na linha 82, utilizamos o método estudado e explicado no capítulo anterior, código da Versão 2.2, que implementa o método de subgradientes, para atualização do limite superior (UB), bem como a heurística lagrangiana do modelo de EPM, para cálculo do limite inferior (LB). Vale destacar que referências ao modelo de EPM demandam agora uma mudança de sintaxe relevante (em relação ao mencionado código da Versão 2.2). Deve ser utilizado o ponteiro *Heur* quando da construção desse modelo no procedimento *main* (linhas 208-234) assim como para sua resolução e uso de sua resposta dentro do *Goal* (linhas 82 e 91). Por exemplo, para a utilização da variável *x_HL*, utilizamos a sintaxe $Heur \rightarrow x_HL$.

Após o processo para obtenção do limite superior(UB), pela relaxação lagrangiana, e limite inferior(LB), pela heurística lagrangiana, criamos os objetos *vars* e *val_vars*, que receberão as variáveis x_{ij} do modelo EPMRC e seus valores correspondentes à solução viável que define LB(linhas 84-95).

Na linha 101, verificamos a possível poda do nó, o que ocorrerá caso UB seja menor ou igual a *maxLB*, este definido na linha 99 como o maior valor entre LB e o valor da melhor solução viável obtida entre os nós já visitados (*getIncumbentObjValue()*). Além disso, se a solução obtida pela heurística lagrangiana é melhor que a melhor atualmente conhecida, ela é passada para o controle do CPLEX via *SolutionGoal* (linha 103), que atualizará a solução

incumbente. Por outro lado, se a solução incumbente não é melhorada, o nó é podado via *FailGoal* (linha 105). Vale lembrar que, se a solução da relaxação lagrangiana satisfaz as condições (5.1)-(5.2), esta solução é viável e candidata ao incumbente. Entretanto, nesse caso, a heurística lagrangiana aplicada (modelo de EPM) retornará esta mesma solução. Desse modo, não precisamos verificar a satisfação das condições (5.1)-(5.2).

Quando o critério de poda não for atendido, então iremos aplicar o processo de ramificação. Na linha 107, declaramos objetos do tipo *IloExpr* para a formação das expressões que irão definir as restrições para a ramificação. Na linha 109, declaramos vetores do tipo *IloNumArray* que irão armazenar os índices das restrições formadas pelas expressões, para serem passados posteriormente para os nós filhos através do *herancaNo*. As expressões das restrições são estruturadas nas linhas 112-143, a partir do conjunto de arestas incidentes ao vértice v^*, armazenado na variável *vertice*.

Procuramos particionar, iterativamente, esse conjunto de arestas, de modo a equilibrar as expressões (5.4), armazenadas em *expr_vizinhanca1* e *expr_vizinhanca2*. O valor desses somatórios são mantidos em $cst1$ e $cst2$, respectivamente. Uma por vez, cada aresta desse conjunto é avaliada e sua variável adicionada a *expr_vizinhanca1*, caso $cst1 \leq cst2$, ou a *expr_vizinhanca2*, caso contrário. Evidentemente, os valores de $cst1$ ou $cst2$ são atualizados, conforme o caso. Após todas as arestas incidentes a v^* terem sido analisadas, além dos índices de suas variáveis estarem salvas nas estruturas declaradas na linha 109, adicionamos por último o valor que irá representar o *lado direito* da restrição (0 ou 1) nas linhas 145-147. Posteriormente, construímos de fato as restrições, como podemos observar nas linhas 149-157, que apresentam a declaração dos objetos do tipo *IloRangeArray*, referentes a cada restrição a ser formada, e a adição das expressões à estrutura do objeto, por meio do método *add*.

Nas linhas 159-171, temos a criação de três objetos da classe *dadosNo*, um para cada ramo a ser criado. Eles irão armazenar as informações do nó a ser passada e herdada por seus nós descendentes. Após as estruturas e restrições estarem prontas, realizamos nas linhas 173-184 a devida ramificação, gerando três nós filhos, conforme Figura 17, usando *goals*. Caso LB seja melhor que a solução incumbente, informamos a solução associada a LB ao CPLEX também através de um *SolutionGoal*. Mais precisamente, o objeto *resultado* irá receber o resultado do seguinte processo de ramificação:

- 1º Passo: Utilizamos o método *OrGoal*, que recebe 4 parâmetros (caso LB seja maior que a solução incumbente) ou 3 parâmetros (caso contrário) para a criação dos nós;

- 2º Passo: Caso a condição testada na linha 175 seja verdadeira, o primeiro parâmetro do *OrGoal* é exatamente um *SolutionGoal*, para passarmos ao CPLEX a solução heurística construída no nó. Em qualquer caso, os outros três parâmetros do *OrGoal* correspondem à ramificação;
- 3º Passo: Cada um desses três parâmetros correspondentes à ramificação recebe um *AndGoal* entre a restrição que define o nó e *MEU_GOAL()*, que provocará a repetição do processo, recursivamente, em cada nó. Note que o último parâmetro de *MEU_GOAL* passa as informações do pai para o filho, que são diferentes entre os três filhos, dados pelos objetos *herancaNo1*, *herancaNo2* e *herancaNo3*.

Ao fim, apagamos e liberamos a memória do modelo RL, construído na linha 187, e posteriormente retornamos o *goal* resultado (linha 189). Após o desempilhamento de todos os *goals*, podemos, na linha 265, avaliar o *status* da solução obtida (*cplex_EPMRC.getStatus()*), assim como explicitá-la, retornando o valor da função objetivo (*cplex_EPMRC.getObjValue()*) e das variáveis (*cplex_EPMRC.getValues()*).

5.5 Experimentos computacionais

Para os testes, utilizamos algumas das instâncias citadas e usadas no Capítulo 4 para obtenção de limites superiores e inferiores. Desta vez, aplicaremos o código explicado na seção anterior para atingirmos a otimalidade. Para efeitos comparativos, resolveremos as mesmas instâncias, usando dos procedimentos: 1) o *branch-and-bound* padrão do CPLEX, 2) nosso *branch-and-bound*, implementado com *goals*. Em ambos os casos, desabilitamos o pré-processamento do CPLEX, seus métodos heurísticos e seus cortes (assim como descrito no código apresentado na seção anterior). Como já foi mencionado, o CPLEX inevitavelmente resolve a relaxação linear associada ao nó corrente. Aplicando ainda pré-processamento, cortes e heurística, a relaxação do nó raiz tende a sofrer alterações que melhoram limite gerado. Assim, como nossa versão da relaxação lagrangiana aplicada ao problema não utiliza e nem se beneficia de outros métodos de melhoramento, desabilitamos estes parâmetros do resolvedor para que a comparação dos resultados entre o CPLEX padrão e nossa versão com *goals* torne-se mais justa.

Para as implementações, utilizamos o mesmo ambiente descrito na Seção 2.8 com um limite máximo de tempo de duas horas (7200 segundos) estabelecido para cada instância. Na Tabela 4 é possível ver um resumo dos resultados obtidos nos testes. Além do nome da instância e do valor de sua solução ótima, apresentamos o tempo (em segundos) demandado por cada

uma das versões para a obtenção dos limites. Apresentamos também, na Tabela 4, o cálculo da distância relativa entre os limites superiores e inferiores para cada instância, dado pela fórmula $100\frac{LS-LI}{LI+0.0001}$.

Podemos notar que a versão testada com o CPLEX com desabilitações conseguiu atingir a otimalidade em um curto tempo de processamento para todas as instâncias testadas, encontrando maior dificuldade na instância match-100-0.75(3)_myci, cujo tempo de resolução foi aproximadamente 100 segundos. Por outro lado, nossa versão com *goals* não se mostrou eficiente, já que atingiu a otimalidade em apenas 8 instâncias, excedendo o tempo limite de 7.200 segundos nas 11 demais, ainda com um *gap* considerável entre os limites superior e inferior para alguns casos, como na instância match-100-0.75(3)_myci, que apresentou 14,86% de distância percentual relativa entre os limites.

Executamos também as mesmas instâncias com o CPLEX padrão, sem desabilitações. Nesse caso, o tempo médio de resolução baixou de 18,6 para 6,8 segundos, levando a um processo ainda mais rápido que o nosso.

O CPLEX é uma ferramenta poderosa e reconhecida no mundo acadêmico e científico pelo o desenvolvimento de seus métodos e estratégias durante anos. Então, obtermos um desempenho inferior com um método incipiente é, de certa forma, esperado. Por isso, o nosso trabalho focou principalmente em mostrar como podemos utilizar os recursos que o resolvedor nos disponibiliza para controle de seus próprios métodos e estratégias de acordo com o problema aplicado, como também em formas de incorporar ferramentas e metodologias desenvolvidas pelo usuário, deixando evidente que alterações ou interferências podem afetar fortemente o tempo da solução, como podemos ver pelos resultados da Tabela 4.

Tabela 4 – Resultados utilizando *goals*

		CPLEX				GOALS			
Instâncias	**Ótimo**	**LI**	**LS**	**Gap**	**Tempo**	**LI**	**LS**	**Gap**	**Tempo**
match-50-0.5-0.005	363,960	363,960	363,960	0,00%	0,088	363,960	363,960	0,00%	35,695
match-50-0.5-0.01	350,514	350,514	350,514	0,00%	0,140	350,514	350,514	0,00%	379,251
match-50-0.5-0.015	363,272	363,272	363,272	0,00%	0,119	363,272	363,272	0,00%	41,513
match-50-0.75-0.005	410,238	410,238	410,238	0,00%	0,058	410,238	410,238	0,00%	0,073
match-50-0.75-0.01	391,476	391,476	391,476	0,00%	0,109	391,476	391,476	0,00%	207,54
match-50-0.75-0.015	388,682	388,682	388,682	0,00%	0,131	388,682	388,682	0,00%	26,416
match-100-0.25-0.005	724,917	724,917	724,917	0,00%	0,139	724,917	724,917	0,00%	14,036
match-100-0.25-0.01	705,998	705,998	705,998	0,00%	1,659	697,635	710,011	1,77%	7200
match-100-0.25-0.015	701,471	701,471	701,471	0,00%	3,743	677,067	707,404	4,48%	7200
match-100-0.5-0.005	785,313	785,313	785,313	0,00%	0,642	785,313	785,313	0,00%	3697,440
match-100-0.5(1)_myci	791,341	791,341	791,341	0,00%	6,745	776,171	796,114	2,57%	7200
match-100-0.5(2)_myci	780,511	780,511	780,511	0,00%	14,662	745,215	795,751	6,78%	7200
match-100-0.5(3)_myci	788,323	788,323	788,323	0,00%	9,542	740,696	793,975	7,19%	7200
match-100-0.5(4)_myci	780,410	780,410	780,410	0,00%	18,603	750,046	791,524	5,53%	7200
match-100-0.75(1)_myci	817,351	817,351	817,351	0,00%	31,085	745,624	822,647	10,33%	7200
match-100-0.75(2)_myci	835,483	835,483	835,483	0,00%	59,231	741,688	851,934	14,86%	7200
match-100-0.75(3)_myci	815,285	815,285	815,285	0,00%	99,991	780,245	829,469	6,31%	7200
match-100-0.75(4)_myci	823,122	823,122	823,122	0,00%	82,794	751,615	837,635	11,44%	7200
match-100-0.75(5)_myci	819,544	819,544	819,544	0,00%	23,248	730,390	824,969	12,95%	7200

6 CONCLUSÕES

Antes de qualquer outro propósito, esta monografia procurou ilustrar, com exemplos concretos, o uso de resolvedores de programação matemática, em particular o CPLEX, para a resolução de problemas de programação inteira.

Algoritmos de solução para esse tipo de problema envolvem diversos procedimentos, entre as quais destacamos: fortalecimento de uma formulação básica através de desigualdades válidas, particionamento sucessivo do problemas em subproblemas potencialmente mais fáceis, determinação de limites primais e duais que contribuam para descartar subproblemas sem explorá-los explicitamente. Cada um desses procedimentos pode ser implementado segundo diversas formas, que podem levar a diferentes desempenhos dos algoritmos de solução.

Aqui, procuramos exemplificar e avaliar implementações variadas para cada um desses procedimentos, usando estratégias desde simples até bem elaboradas e que demandam maior conhecimento das ferramentas disponibilizadas pelos resolvedores.

O uso de desigualdades válidas para fortalecimento de uma formulação foi considerado no Capítulo 2. Elas podem ser adicionadas a priori ao modelo ou paulatinamente à medida que se mostrem úteis a sua resolução, através de um procedimento de separação de desigualdades violadas. A implementação do primeiro caso é similar à própria construção do modelo, vista na Seção 2.7. Para ilustrar o segundo caso, usamos o problema de emparelhamento máximo acrescido das restrições de blossom. Esse acréscimo foi realizado de duas maneiras: como restrições "preguiçosas"(*lazy constraints*) e como planos de corte. Para isso, trabalhamos com dois modelos simultaneamente ativos no resolvedor, sendo que o segundo era o responsável por identificar as restrições de *blossom* violadas.

No Capítulo 3, reformulamos o problema de emparelhamento de peso máximo com o acréscimo de restrições de conflito. Restrições estas que expressam uma incompatibilidade entre pares de elementos, exigindo que no máximo um deles poderá compor uma solução viável. Além disso, mostramos os efeitos que as restrições de conflitos causam no problema e assim aplicamos nossos testes posteriormente.

A forma usual para obtenção de um limite dual é através da resolução da relaxação linear do problema, o que é feito automaticamente pelos resolvedores. Já os limites primais são normalmente obtidos por heurísticas. Aqui, implementamos uma segunda abordagem, que tem potencial para fornecer melhores limites duais, mas demanda maior conhecimento teórico bem como maior domínio do ambiente disponibilizado pelo *software* de programação matemática.

Esse foi o foco do Capítulo 4, que teve como objetivo aplicar a técnica da relaxação lagrangiana para a obtenção de bons limitantes duais em diferentes versões do mesmo método, para uma comparação de esforço computacional. Além disso, testamos diferentes heurísticas lagrangianas, utilizadas para determinar soluções viáveis, cujo valor fornece um limitante primal para o problema.

O particionamento do problema em subproblemas menores se faz em geral pela divisão do conjunto viável. Possivelmente a forma mais simples de particionamento é pela ramificação em variável, feita automaticamente pelos resolvedores. No Capítulo 5, implementamos uma estratégia mais complexa de subdivisão do problema, baseada na relaxação lagrangiana usada para cálculo do limite dual. Apresentamos os resultados obtidos com essa estratégia, comparando-os aos fornecidos pelo o resolvedor. Em relação aos nossos experimentos computacionais, os tempos de resolução da relaxação lagrangiana das instâncias geradas, para obter limites, mostraram-se mais elevados comparados à relaxação linear; entretanto outras estratégias de solução poderiam ser utilizadas, como, por exemplo, fazer uma ramificação similar a que fazemos, porém baseada na solução da relaxação linear.

Nosso trabalho, mais do que estudar o problema de Emparelhamento de Peso Máximo com Restrições de Conflito, teve como objetivo e ponto mais forte conciliar a base teórica em otimização matemática com a prática, tratando com diferentes aspectos que devem ser considerados no momento de se resolver efetivamente um problema de otimização, especialmente quando temos diversas opções e possibilidades de abordagem, e ainda a escolha da forma segundo a qual se implementar o método escolhido, usando o resolvedor CPLEX. Além disso, analisar as consequências e os efeitos gerados por alterações no resolvedor, que pode levar a uma piora (ou melhora) do tempo computacional, seja pela a inserção de métodos ou até por alterações dos parâmetros realizadas pelo usuário, desabilitando ou fixando procedimentos já contidos no CPLEX.

O CPLEX, apesar de ser um *software* que disponibiliza inúmeros métodos, técnicas, além de permitir que o usuário possa alterar ou interferir diretamente no seu processo de resolução, junto com uma documentação acessível para o entendimento de suas funcionalidades, carece de exemplos mais ilustrativos que possam facilitar a compreensão do usuário, já que apenas a descrição de sua documentação pode levar a interpretações diferentes. A maior dificuldade deste trabalho foi justamente o entendimento e a aplicação dos *goals*, que foi um ponto vital para a elaboração e implementação da nossa ramificação não usual e o fornecimento de soluções

heurísticas ao processo de resolução.

Compartilhamos aqui nossa experiência, que esperamos possa ajudar outros iniciantes na área, especialmente alunos em formação. Desejamos que esta monografia possa constituir uma fonte de pesquisa acessível a esses estudantes, que venha a auxiliá-los no desenvolvimento de seus trabalhos para o mundo acadêmico ou profissional.

REFERÊNCIAS

ANDRADE, R. C.; FREITAS, A. T. Otimização em árvore de subgradiente para a árvore geradora mínima com restrição de grau nos vértices. In: **Anais do Simpósio Brasileiro de Pesquisa Operacional**. [S.l.: s.n.], 2008. p. 1751–1759.

BONDY, J. A.; MURTY, U. S. R. *et al.* **Graph theory with applications**. [S.l.]: Citeseer, 1976. v. 290.

CPLEX, I. I. **12.7, User's Manual for CPLEX**. [S.l.], 2017. Disponível em: <https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.1/ilog.odms.studio.help/pdf/usrcplex.pdf?origURL=SSSA5P_12.7.1/ilog.odms.studio.help/Optimization_Studio/topics/PLUGINS_ROOT/ilog.odms.studio.help/pdf/usrcplex.pdf>.

DARMANN, A.; PFERSCHY, U.; SCHAUER, J. Determining a minimum spanning tree with disjunctive constraints. In: ROSSI, F.; TSOUKIAS, A. (Ed.). **Proceedings of the Algorithmic Decision Theory Conference, ADT 2009**. [S.l.: s.n.], 2009. LNCS 5783, p. 414–423.

DARMANN, A.; PFERSCHY, U.; SCHAUER, J.; WOEGINGER, G. J. Paths, trees and matchings under disjunctive constraints. **Discrete Applied Mathematics**, v. 159, n. 16, p. 1726 – 1735, 2011. ISSN 0166-218X.

EDMONDS, J. Maximum matching and a polyhedron with $0,1$ vertices. **J. of Res. the Nat. Bureau of Standards**, v. 69 B, p. 125–130, 1965.

EDMONDS, J. Paths, trees, and flowers. **Canadian Journal of Mathematics**, v. 17, p. 449—-467, 1965.

EDMONDS, J.; KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. **J. ACM**, v. 19, n. 2, p. 248–264, abr. 1972.

FORD L. R.; FULKERSON, D. R. Maximal flow through a network. **Canadian Journal of Mathematics**, v. 8, p. 399—-404, 1956.

HOPCROFT, J. E.; KARP, R. M. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. **SIAM Journal on Computing**, v. 2, n. 4, p. 225–231, 1973.

KOLMOGOROV, V. Blossom v: a new implementation of a minimum cost perfect matching algorithm. **Mathematical Programming Computation**, v. 1, n. 1, p. 43–67, Jul 2009.

LOVÁSZ, L.; PLUMMER, M. **Matching theory**. [S.l.]: Akadémiai Kiadó, 1986. (North-Holland mathematics studies). ISBN 9789630541688.

MICALI, S.; VAZIRANI, V. V. An $o(|v|^{1/2}|e|)$ algorithm for finding maximum matching in general graphs. In: **21st Annual Symposium on Foundations of Computer Science**. [S.l.: s.n.], 1980. p. 17–27.

MILLER, C.; TUCKER, A.; ZEMLIN, R. Integer programming formulation of traveling salesman problems. **Journal of ACM**, v. 7, p. 326–329, 1960.

MYCIELSKI, J. Sur le coloriage des graphs. **Colloquium Mathematicae**, v. 3, n. 2, p. 161–162, 1955.

PADBERG, M. W.; RAO, M. R. Odd minimum cut-sets and b-matchings. **Mathematics of Operations Research**, v. 7, n. 1, p. 67–80, 1982.

PFERSCHY, U.; SCHAUER, J. The maximum flow problem with conflict and forcing conditions. In: PAHL, J.; REINERS, T.; VOSS, S. (Ed.). **Proceedings of the 5th International Conference on Network Optimization, INOC 2011**. [S.l.: s.n.], 2011. LNCS 6701, p. 289–294.

PFERSCHY, U.; SCHAUER, J. The maximum flow problem with disjunctive constraints. **Journal of Combinatorial Optimization**, v. 26, n. 1, p. 109–119, 2013.

RIZZI, R. A simple minimum t-cut algorithm. **Discrete Applied Mathematics**, v. 129, n. 2, p. 539 – 544, 2003. ISSN 0166-218X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0166218X03001823>.

SAMER, P.; URRUTIA, S. A branch and cut algorithm for minimum spanning trees under conflict constraints. **Optimization Letters**, v. 9, n. 1, p. 41–55, 2015. ISSN 1862-4480.

WOLSEY, L. A. **Integer programming**. [S.l.]: Wiley, 1998.

ZHANG, R.; KABADI, S. N.; PUNNEN, A. P. The minimum spanning tree problem with conflict constraints and its variations. **Discrete Optimization**, v. 8, n. 2, p. 191 – 205, 2011. ISSN 1572-5286.