BUSCA DE AGRUPAMENTOS DE DADOS UTILIZANDO GERAÇÃO DE COLUNAS EM PROGRAMAÇÃO INTEIRA

Rogério Gomes de Lima Tostas

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Nelson Maculan Filho

Rio de Janeiro
Março de 2012

BUSCA DE AGRUPAMENTOS DE DADOS UTILIZANDO GERAÇÃO DE COLUNAS EM PROGRAMAÇÃO INTEIRA

Rogério Gomes de Lima Tostas

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Nelson Maculan Filho, D.Sc.

Prof. Luiz Satoru Ochi, D.Sc.

Prof. Michael Ferreira de Souza, D.Sc.

Prof. Adilson Elias Xavier, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2012

Gomes de Lima Tostas, Rogério

Busca de agrupamentos de dados utilizando geração de colunas em programação inteira/Rogério Gomes de Lima Tostas. – Rio de Janeiro: UFRJ/COPPE, 2012.

X, 50 p.: il.; 29, 7cm.

Orientador: Nelson Maculan Filho

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 45 – 50.

1. Agrupamento. 2. Programação inteira. 3. Geração de colunas. 4. Modelos de fluxo. I. Maculan Filho, Nelson. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A Deus, família e amigos.

Agradecimentos

Gostaria de agradecer a minha família, em especial minha mãe Elisete pelos conselhos e orações.

A minha amada esposa Priscila por ter sido sempre companheira e ter compreendido os muitos momentos de ausência necessários para a execução deste trabalho.

Aos professores José Roberto Linhares de Mattos e Eulina Coutinho do Nascimento que têm me acompanhado desde a graduação e sempre acreditaram no meu pontecial, incentivando-me a prosseguir em meus estudos.

Aos amigos do CONTROLAB, Henrique Serdeira, Júlio Tadeu, e em especial, a professora Eliana Prado Lopes Aude (In memoriam) que me acolheu em seu laboratório antes mesmo de iniciar meus estudos no mestrado, pessoa rara e bondosa, a sua lembrança estará para sempre na minha memória e saudades no coração.

Ao professor Ernesto Prado Lopes, que foi inicialmente meu orientador neste programa de mestrado, e que gentilmente permitiu a mudança de orientação.

Aos colegas de curso e do LABOTIM, Renan, Jesus, Vinícius e em especial Helder Venceslau e Marilis Venceslau que sempre me aconselharam e ajudaram em meus estudos.

A empresa Light serviços de Eletricidade S/A que por intermédio de José Luiz Alqueres, concedeu uma bolsa de estudos que foi fundamental para a execução deste trabalho.

Aos meus orientadores os professores Nelson Maculan Filho, que em todos os momentos ajudou, sendo sempre um exemplo de educador, pesquisador e mestre, seus cuidados comigo foram muitos, não medindo esforços para me ajudar nesta jornada, e Michael Ferreira de Souza pela extrema paciência, bondade e compreensão, certamente sem sua ajuda este trabalho não seria possível.

Por fim e acima de tudo à Deus por ter colocado pessoas tão sábias em bondosas no meu caminho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

BUSCA DE AGRUPAMENTOS DE DADOS UTILIZANDO GERAÇÃO DE COLUNAS EM PROGRAMAÇÃO INTEIRA

Rogério Gomes de Lima Tostas

Março/2012

Orientador: Nelson Maculan Filho

Programa: Engenharia de Sistemas e Computação

Neste trabalho, nós apresentamos dois modelos exatos baseados na técnica de geração de colunas para o problema de agrupamento em grafos conhecido como *k-cluster* (ou *k-agrupamento*), onde o objetivo é particionar um dado grafo $G(V, E)$ em k componentes (subgrafos) conexas de tal forma que a soma das arestas no interior de cada componente seja mínima. Nossos modelos inspiraram-se nos problemas de fluxo em redes para garantir a conexidade dos subgrafos que representam os agrupamentos. Os dois modelos lineares e inteiros, foram denominados OneFlow, que se baseia-se na construção de um único fluxo e MultiFlow, baseado na construção de fluxos múltiplos. Foi utilizada uma abordagem de geração de colunas justificada pela característica combinatória e o elevado número de variáveis (e, consequentemente, colunas) associadas a cada instância do problema *k-cluster*. Para a geração do conjunto inicial de colunas, aplicamos uma versão modificada do algoritmo aproximativo proposto por Saran e Vazirani para a resolução do problema *min k-cut*. Nesta estratégia, partimos de uma árvore de *Gomory-Hu* para o grafo de entrada G e aplicamos sucessiva e de forma gulosa até k cortes associados às arestas da árvore de Gomory-Hu. Para a obtenção de soluções inteiras utilizamos as técnicas branch-and-bound. Para a etapa de branch, utilizamos a estratégia proposta por Ryan e Foster, desta forma obtemos árvores de ramificação geradas que tendem a ser mais balanceadas do que aquelas geradas pela imposição sucessiva de integralidade a cada uma das variáveis.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SEARCH FOR DATA CLUSTERS USING COLUMN GENERATION IN INTEGER PROGRAMMING

Rogério Gomes de Lima Tostas

March/2012

Advisor: Nelson Maculan Filho

Department: Systems Engineering and Computer Science

In this work, we present ourselves two exact models based on technique generation of columns for problem on grouping graphs known as *k-cluster*, where the goal and partition a given graph $G(V; E)$ k-components (subgraphs) connected in such a way that the sum of the edges within each component is minimum. Our models were inspired by problems of flow in networks to ensure connectivity of subgraphs representing the groups. The two models linear and integer, have been termed One-Flow, based on the construction of a single stream and Multiflow, based on the construction of multiple streams. Was used column generation method justified by the combinatorial characteristic and the high number of variables (and hence columns) associated with each instance of problem *k-cluster*. To generate the initial set of columns, we apply a modified version of the approximation algorithm proposed by Saran and Vazirani for the resolution of the problem *min k-cut*.In this strategy, we started from a tree *Gomory-Hu* for the input graph G and successively applied greedy way cuts up to k associated cutting with the edges of the tree Gomory-Hu. To obtain the integer solutions was used branch-and-bound technical . For the stage of branch, we used the strategy proposed by Ryan and Foster, thereby obtain branching tree generated which tend to be more balanced than those generated by the imposition of successive integral to each of the variables.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No problema de agrupamento, o objetivo é reconhecer grupos naturais no interior de um dado conjunto. Os métodos clássicos de agrupamento são baseados na representação dos elementos do conjunto utilizando vetores $x = (x_1, x_2, \ldots, x_n)$ [1, 2]. Nesta abordagem, cada uma das componentes x_i representa uma característica do elemento $x = (x_1, \ldots, x_n)$ do conjunto, e as dissimilaridades entre os elementos da população são definidas pelas distâncias entre os vetores que os representam. A partir destas representações vetoriais em um conjunto munido de uma função distância, os agrupamentos podem ser representados por centróides e pelos vetores (elementos do conjunto) a eles mais próximos. Grafos, por outro lado, apresentam estruturas (vértices e arestas) e conceitos (conexidade, grau, componente, etc) que, em geral, não podem ser convenientemente mapeados em vetores de um conjunto munido de uma função distância. Por exemplo, não há uma forma natural de medir dissimilaridade entre vértices que não sejam vizinhos sem que envolvamos outros vértices. No caso de uma representação vetorial, a função de distância teria que embutir os conceitos de caminhos mínimos ou outros ainda mais complexos/custosos. Como consequência, a redução do problema de agrupamento em grafos para o problema de agrupamento em um conjunto de vetores não parece ser a mais adequada, carecendo portanto de tratamento diferenciado.

A proposta deste trabalho consiste em um método exato baseado na técnica de geração de colunas para a resolução do problema de agrupamento em grafos conhecido como *k-cluster* (ou *k-agrupamento*). Neste contexto, um agrupamento é um particionamento que minimiza a soma das arestas no interior de cada uma das componentes da partição. No problema *k-cluster*, o objetivo é obter um agrupamento formado por exatamente k componentes. Adicionalmente, impõe-se como restrição a conexidade das componentes do agrupamento.

O problema de agrupamento é um caso particular do problema de particionamento e apresenta interessantes aplicações. Entre as áreas de aplicação das técnicas de particionamento de grafos destacam-se a multiplicação de matrizes esparsas, com-

putação paralela e a solução numérica de equações diferenciais parciais [3–5]. Por exemplo, na multiplicação de matrizes esparsas em computação paralela, uma das etapas é a distribuição das linhas entre diferentes processadores. Neste contexto, podemos definir o problema de particionamento do conjunto das linhas entre os processadores de forma a minimizar o tempo de comunicação entre eles. Outra grande área de aplicação do particionamento de grafos é LVSI CAD (do inglês *Very Large Scale Integration in Computer Aided Design*), onde o objetivo é particionar a especificação de um sistema em grupos de forma a minimizar o número de conexões entre os grupos [6].

Quanto à dificuldade, o problema *k-cluster* é equivalente ao problema conhecido como *max k-cut*, onde o objetivo é obter um corte (subconjunto de arestas) de custo máximo cuja remoção particione um dado grafo em k componentes conexas [7]. O custo do corte é dado pela soma das arestas pertencentes ao corte. A equivalência entre estes problemas decorre da correspondência entre minimizar a soma das arestas que permanecerão (i.e., arestas no interior das componentes) e maximizar a soma das arestas que serão removidas (as arestas do corte). O problema *max k-cut* postulado com um problema de existência (i.e. dado um número L, existe um corte cujo valor exceda L?) pertence à classe dos problemas NP-completos [7–9]. Grosso modo, a classe de complexidade NP é formada pelos problemas cujas candidatas a solução podem, em tempo polinomial, ser verificadas como sendo soluções de fato. Já a classe dos problemas NP-completos é um subconjunto especial da classe NP formada pelos problemas cuja resolução em tempo polinomial implica na resolução em tempo polinomial de todos os problemas da classe NP (Para maiores detalhes sobre o tema consulte [10]). Apesar de todo esforço em se provar que $P = NP$, até o presente momento, nenhum resultado nesta direção foi obtido. Postulado como um problema de otimização (i.e. determinar o corte máximo), o problema *max k-cut* pertence à classe NP-difícil o que significa, essencialmente, que a otimalidade de uma solução só pode ser estabelecida listando todas as demais soluções e seus respectivos custos [3, 11]. Sendo assim, pela equivalência entre o problema de agrupamento e o de corte máximo, não há muita esperança de obtenção de soluções ótimas em tempo polinomial para o problema de agrupamento em grafos.

Nesta dissertação, propomos a resolução do problema *k-cluster* através do método de geração de colunas utilizando dois modelos inspirados em problemas de fluxo em redes. Estes modelos permitem enumerar de maneira implícita os subgrafos conexos de um dado grafo. Como veremos, serão esses subgrafos conexos gerados pelos modelos de fluxo que definirão as colunas da nossa aplicação.

A utilização da estratégia de geração de colunas é particularmente útil no problema *k-cluster*, onde o tamanho do espaço viável (número de variáveis) inviabiliza uma representação explícita das restrições do problema. Cabe ainda ressaltar que

os métodos de geração de colunas são procedimentos iterativos que permitem a obtenção de soluções ótimas a partir da representação parcial (ou implícita) das restrições. Esta característica iterativa dos métodos de geração de colunas permite ainda a obtenção de soluções viáveis cada vez melhores ao longo das iterações. Desta forma, boas soluções podem estar disponíveis ainda que tenhamos de interromper a execução do algoritmo por, por exemplo, excesso de tempo.

Essa estratégia de decomposição de problemas complexos através de modelos de otimização não é nova. De fato, as origens dos métodos de geração de colunas remetem a década de 1960, quando Dantzig e Wolfe analisaram o problema de determinação de padrões de corte unidimensionais [12–14]. Deste então, esta técnica tem sido explorada em várias aplicações como, por exemplo, no problema de roteamento de veículos [15, 16], na escala de tripulações [17–19], em projetos de circuitos [20] e em problemas de empacotamento [21] entre outros.

Nas últimas décadas em particular, o número de aplicações de programação matemática em problemas complexos de grande porte vem aumentando. Segundo Lorena e Senne [22], isto em parte se deve ao desenvolvimento de pacotes comerciais de otimização como, por exemplo, CPLEX, Xpress-Mosel, Gurobi, SoPlex, LPSolver, GLPK e LINDO. Dadas a facilidade de integração/utilização e eficiência destes pacotes, o tratamento de problemas combinatórios de grande porte tem recebido grande impulso, já que estas ferramentas tem viabilizado a resolução de problemas de grande complexidade em um intervalo de tempo aceitável.

Contudo, em alguns problemas, até mesmo os pacotes comerciais mais eficientes não são capazes de obter uma solução ótima em tempo razoável. Uma das razões para esse fato é que, na sua forma mais geral, o problema de programação inteira pertence à classe dos problemas mais difíceis do ponto de vista computacional. Por isso, qualquer ferramenta que ataque o problema de programação inteira de uma perspectiva generalista, i.e., que não explore as estruturas particulares de cada problema, dificilmente será capaz (a menos que $P = NP$) de obter soluções ótimas em tempo razoável. Portanto, a definição de estratégias especializadas que explorem estruturas muito específicas de cada problema (algo que os pacotes comerciais não podem fazer para todos os problemas) pode reduzir consideravelmente o tempo necessário para a obtenção de boas soluções. É precisamente neste espaço gerado pelas abordagens generalistas que se insere a proposta dessa dissertação.

No capítulo 2, destacamos importantes contribuições que formataram a linha de pesquisa em programação linear além de alguns dos conceitos e resultados básicos na área. No capítulo 3, destacamos conceitos elementares de programação linear inteira como relaxação, decomposição de Dantzig e Wolfe e algoritmos branch-and-bound. No capítulo 4, apresentamos dois modelos de geração de colunas para o problema de agrupamento, uma heurística para a geração de colunas iniciais e a estratégia

para obtenção de soluções inteiras. O capítulo 5 é reservado para os resultados computacionais e discussões. Finalmente, o capítulo 6 destina-se às conclusões da presente dissertação.

Capítulo 2

Programação linear

A *programação linear* (ou *otimização linear*) tem desempenhado um importante papel na resolução de diversos problemas. Segundo Maculan e Fampa [23], este elemento importantíssimo da Pesquisa Operacional, tem aplicação direta nos mais variados setores desde indústrias, passando por empresas de transportes, na saúde [24], na educação, na agricultura, nas finanças, na economia, nas administrações públicas, na extração de petróleo, na medicina [25], entre outros.

A programação linear parte da representação em linguagem matemática dos problemas de interesse e, dessa forma, permite obter boas (ou até ótimas) soluções em tempo reduzido. Embora para muitas pessoas a idéia de modelar problemas em linguagem matemática para então resolvê-los utilizando estuturas matemáticas (como álgebra linear) e recursos computacionais pareça novidade, os fundamentos desta abordagem já estavam postos nas primeiras décadas do século XX.

Em 1928, John von Neuman publicou o teorema central da teoria dos jogos mais tarde foi formulado através da programação linear e interpretado à luz da teoria da dualidade. Em 1937, von Newman publicou *"A Model of General Economic Equilibrium"*, onde formula um modelo de programação linear dinâmica [26]. Alguns anos mais tarde, em 1944, von Neuman publicou *"Theory of Games and Economic Behaviour"* [27] novamente explorando a linguagem da programação linear para obtenção de muitos de suas conclusões. Os trabalhos de von Neumann estabeleceram a interação entre jogos e a economia e despertou um interesse ainda maior pela programação linear.

A análise insumo-produto (*input-output*) proposta por Leontief em termos matriciais em 1936 em seu artigo *"Quantitative input and output Relations in the Economic Systems of the United States"* [28] serviu para consolidar a programção como uma ferramenta viável para representação e análise de problemas em economia.

Entre os pioneiros da programação linear destaca-se também o matemático Kantorovich [29] que em 1939 na então União Sovética, já havia modelado e resolvido alguns problemas de otimização ligados ao planejamento econômico, tendo recebido

junto com Koopmans [30] prêmio Nobel de Economia em 1975 como resultado destes trabalhos. As contribuições de Hitchcock na modelagem do problema de transporte [31] e Stigler na formulação do problema de programação linear relacionado problema da Dieta [32] também ajudaram a formatar e consolidar a programação linear como uma importante ferramenta para solução de problemas reais e de forte apelo econômico.

O termo "programação numa estrutura linear"foi sugerido por Koopmans, e surgiu pela primeira vez em 1951, mas somente mais tarde, a expressão veio a se consagrar como programação linear [33].

Embora as primeiras aplicações expressivas do uso da programação linear tenham sido em problemas ligados ao setor econômico, foram problemas ligados à logística militar em tempos de guerra que promoveram o grande salto de publicidade da Pesquisa Operacional. Durante a 2ª Guerra Mundial, a força aérea americana organizou um grupo de pesquisadores em um projeto conhecido como *SCOOP (Scientific Computation of Optimum Program)* sob a direção de Marshall K. Wood. O objetivo do projeto era otimizar a alocação dos limitados recursos provenientes dos esforços de guerra [34]. Embora nenhum método de expressivo sucesso tenha sido descoberto durante a guerra, George B. Dantzig, um dos membros deste grupo, formulou o problema de *Programação Linear Geral* estabelecendo em termos definitivos o problema de interesse da programação linear.

Em 1947, o mesmo Dantzig inventou o primeiro algoritmo para a solução dos problemas de programação linear que foi denominado *Método Simplex*. Com a apresentação do método Simplex, não apenas a área de pesquisa em programação linear, mas a programação matemática em geral sofreu uma expansão quase inacreditável, despertando a curiosidade e interesse de muitos outos matemáticos e economistas da época. O interesse foi tanto que rapidamente novos avanços vieram. Em 1951, Tucker obteve os primeiros resultados na teoria da dualidade que mais tarde veio a se tornar uma parte fundamental da programação linear. Ainda em 1951, Dorfman publicou o seu livro *"Application of linear programming to the theory of the firm"* [35], onde expressou a teoria econômica da empresa em condições de concorrência perfeita e em monopólio em termos de programação linear. Um ano mais tarde, em 1952, Charnes e Lemkes desenvolveram o método Simplex Revisado, que foi muito importante para a implementação computacional do método Simplex [36].

Não podemos deixar de citar os trabalhos de Khachiyan em 1979 [37] que, utilizando os métodos dos elipsóides, propôs um algoritmo no qual o número de iterações para resolver um problema de programação linear é limitado por uma função polinomial. Porém, segundo [23], seu desempenho prático deixa bastante a desejar já que sua convergência é lenta quando aplicado a grande parte das instâncias do problema e comparado ao método Simplex.

A publicação de um novo algoritmo por Karmarkar [38], em 1984, deu início a outra importante linha de pesquisa em programação linear chamada de *métodos de pontos interiores*. Esses métodos vieram a partir da busca por um algoritmo que tivesse tempo de computação polinomial, já que Klee e Mintyapud [39] ilustraram que o algoritmo Simplex pode apresentar um crescimento exponencial no tempo de computação em seu pior caso. Após a publicação desse trabalho de Karmarkar [38], vários algoritmos de pontos interiores foram apresentados. Maiores explicações sobre os métodos de pontos interiores podem ser encontradas nas referências [40–42].

2.1 O problema de programação linear

Podemos definir o problema de programação linear (PPL) no seu formato usual como o seguinte problema de otimização:

$$
\begin{aligned}
(PPL) \quad \min \quad & z = cx && (2.1)\\
\text{s.a} \quad & Ax = b && (2.2)\\
& x \geq 0, && (2.3)
\end{aligned}
$$

onde $A \in \mathbb{R}^{m\times n}$, $c^T \in \mathbb{R}^n$, e $b \in \mathbb{R}^m$ são dados com $0 < m \leq n$. Vamos supor também sem perda de generalidade que $posto(A) = m$, isto é existem m colunas de A que são linearmente independentes.

Definição 2.1 *Seja $X = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$. O conjunto X é denominado conjunto ou região viável do (PPL) e se $x \in X$, então x é uma solução viável do mesmo problema. Dado $x^* \in X$, x^* é denominado uma solução ótima do (PPL) se $cx^* \leq cx$, para todo $x \in X$.*

Particionaremos a matriz A da seguinte maneira: $A = (B \quad N)$, onde B é uma matriz quadrada $m \times m$ e inversível, ou seja, $det(B) \neq 0$. Nesse caso, existe um conjunto I_B formado por m índices das n colunas de A, de tal forma que a matriz B formada com as colunas relativas a esses índices tem posto m. Chamaremos a matriz B de *matriz básica*. Podemos ainda alocar os índices das colunas de A não pertencentes a I_B em um outro conjunto I_N e com as colunas de A referentes a esses índices formar uma matriz $N \in \mathbb{R}^{m\times(n-m)}$, essa chamada de *matriz não básica*. Lembramos que $I_B \cap I_N = \emptyset$ e $I_B \cup I_N = \{1, 2, ..., n\}$.

De modo análogo podemos particionar os vetores x e c : $x^T = (x_B^T \quad x_N^T)$ e $c = (c_B \quad c_N)$, onde x_B e c_B possuirão m componentes associados a matriz B. Vejamos agora as seguintes definições:

- x_B: vetor de m componentes, formado pelas variáveis do vetor x associadas à matriz base, variáveis essas chamadas de *variáveis básicas*.

- x_N:vetor de $(n-m)$ componentes, formado pelas variáveis do vetor x associadas à matriz não básica N, variáveis essas chamadas de *variáveis não básicas*.
- c_B: vetor de m componentes, formado pelos coeficientes do vetor c referentes às variáveis básicas.
- c_N: vetor de $(n-m)$ componentes, formado pelos coeficientes do vetor c referentes às variáveis não básicas.

Dessa forma, podemos escrever o (PL) da seguinte maneira:

$$(PPL) \quad \min \quad z = c_B x_B + c_N x_N \tag{2.4}$$

$$\text{s.a} \quad Bx_B + Nx_N = b \tag{2.5}$$

$$x_B \geq 0 \quad x_N \geq 0 \tag{2.6}$$

Podemos explicitar x_B em função de x_N, multiplicando a restrição principal de (2.5) por B^{-1}, pois sabemos que B é inversível, logo temos:

$$x_B = B^{-1}b - B^{-1}Nx_N \tag{2.7}$$

Tomando $x_N = 0$ em (2.7) obtemos $\bar{x}_B = B^{-1}b$, sendo assim, definimos:

Definição 2.2 *$\bar{x}$ é uma* solução básica *para $Ax = b$ em (2.2) quando $x_N = 0$ e $\bar{x}_B = B^{-1}b$, ou seja, $\bar{x}^T = (\bar{x}_B^T \quad 0)$. Quando $\bar{x}_B$ possuir ao menos uma componente nula diremos que $\bar{x}$ é uma solução básica degenerada.*

Definição 2.3 *Uma solução básica em que x_B assume valores maiores ou iguais a zero é dita* ***Solução Básica Viável****.*

2.2 O método simplex

O algoritmo Simplex, constituiu um grande avanço científico e tecnológico, resultando em grande impulso ao campo da pesquisa operacional que estava dando os primeiros passos nas décadas de 30 e 40. O nome do algoritmo tem suas raízes no conceito de simplex: um plano que corta os vetores unitários. O algoritmo como é conhecido atualmente difere da versão original e tem servido de base para versões estendidas em diversas tarefas específicas. Podemos citar como exemplos o método dual Simplex que é amplamente adotado em implementações *branch-and-bound* e *branch-and-cut* para resolução de problemas inteiros, e também o método Simplex adaptado para o problema de fluxo em redes de custo mínimo.

O algoritmo Simplex pode ser visto como um processo combinatório que busca encontrar as colunas da matriz de restrições que induzem uma base e, portanto,

uma solução básica ótima. A dificuldade advém do fato que tipicamente existe um número exponencial de possíveis combinações de colunas, gerando portando um desempenho de pior caso de ordem exponencial. Apesar deste aspecto desfavorável, o algoritmo Simplex é eficaz para muitas instâncias, podendo em alguns casos ser mais rápido quando comparado com algoritmos de pontos interiores, que tem desempenho polinomial no pior caso. Em 2002 foi realizada por Illés e Terlaky em [43] uma análise comparativa entre o melhores métodos de pontos interiores e o método do Simplex onde concluem que, de uma forma geral, não há método vencedor. Na prática, a chave para o sucesso dos métodos é a utilização da estrutura dos problemas, da esparsidade e da arquitetura dos computadores.

Conforme mostraremos resumidamente a seguir, o método Simplex apresenta uma maneira eficiente de gerar soluções básicas viáveis a partir de uma solução básica dada. Durante o processo de buscas consecutivas por soluções básicas viáveis, uma matriz base B se diferencia da seguinte por apenas uma coluna. É um processo iterativo, que repete os mesmos passos a partir de uma solução básica viável inicial, até se chegar a uma solução ótima. Este método também detecta se o (PPL) tem infinitas soluções, se o problema é ilimitado ou se o problema é inviável.

Para apresentarmos alguns resultados importantes do método Simplex, vamos utilizar a notação da partição da matriz A e do vetor c utilizada na seção anterior e construir uma nova forma de escrevermos o (PPL).

Vamos iniciar substituindo a expressão de x_B dada por (2.7) na função objetivo do (PPL) (2.5), de onde segue que esse (PPL) pode ser reescrito como:

$$
\begin{aligned}
(PPL) \quad \min \quad & z = c_B B^{-1} b - (c_B B^{-1} N - c_N) x_N && (2.8)\\
\text{s.a} \quad & x_B = B^{-1} b - B^{-1} N x_N && (2.9)\\
& x_B \geq 0 \;\; x_N \geq 0 && (2.10)
\end{aligned}
$$

Temos que $c_B B^{-1} N - c_N$ são chamados *custos reduzidos*das variáveis não-básicas.

Para um problema de programação linear no caso de minimização, a condição de otimalidade requer que todos os custos reduzidos sejam não positivos. Para saber se a condição de otimalidade é atendida, o método *Simplex* executa o passo de *pricing* que consiste em encontrar a variável não básica de maior custo reduzido como veremos mais adiante.

Vamos escrever a matriz A por colunas, ou seja, $A = (a_1 \;\; a_2 \;\; ... \;\; a_n)$ onde $a_j^T = (a_{1j} \;\; a_{2j} \;\; ... \;\; a_{1m})$, $a_j \in \mathbb{R}^m$, representa a coluna de A.

Vamos definir $u = c_B B^{-1}$, onde $u^T \in \mathbb{R}^m$, $z_j = c_B B^{-1} a_j = u a_j$, $z_j \in \mathbb{R}$ e $\bar{z} = c_B B^{-1} b = ub = c_B \bar{x}_B$, $\bar{z} \in \mathbb{R}$ como sendo o valor da função objetivo para a solução básica viável corrente. Definimos ainda $y_j = B^{-1} a_j$, $j \in I_B \cup I_N$, $y_j \in \mathbb{R}^m$,

o problema de minimização anterior pode ser reescrito como

$$(PPL) \quad \min \quad z = \bar{z} - \sum_{j \in I_N} (c_j - z_j) x_j \tag{2.11}$$

$$\text{s.a} \quad x_B = \bar{x}_B - \sum_{j \in I_N} y_j x_j \tag{2.12}$$

$$x_B \geq 0, \quad x_j \geq 0, \quad j \in I_N \tag{2.13}$$

Definindo $y_j^T = (y_{1j} \quad y_{2j} \quad ... \quad y_{mj})$, $x_B^T = (x_{B(1)} \quad x_{B(2)} \quad ... \quad x_{B(m)})$, e $\bar{x}_B^T = (\bar{x}_{B(1)} \quad \bar{x}_{B(2)} \quad ... \quad \bar{x}_{B(m)})$ então (2.12) poderá ainda ser escrito como:

$$x_{B(i)} = \bar{x}_{B(i)} - \sum_{j \in I_N} y_{ij} x_j, \quad i = 1, ..., m \tag{2.14}$$

Teste de otimalidade e critério para entrada na base

O teste de otimalidade é utilizado para verificar se uma solução básica viável é ótima ou não. No caso de ter encontrado uma solução viável ótima, o método termina. Caso contrário, devemos indicar uma nova variável não básica para entrar na base de modo que a função objetivo diminua (problemas de minimização). Nesse sentido, tal passo do método mostra-se interessado na melhoria da função objetivo. A próxima proposição estabelece esse critério para dizer se uma uma solução básica viável é ótima a partir da análise do sinal de $(c_j - z_j)$

Proposição 2.1 *Se $x_B \geq 0$ e $(c_j - z_j) \leq 0$, $\forall j \in I_N$, então o vetor $x^* \in \mathbb{R}^n$, onde $x^*_{B_i} = \bar{x}_{B_i}$, $i = 1, 2, ..., m$ e $x^*_j = 0$, $j \in I_N$, será uma solução ótmia do (PPL).*

A demonstração dessa proposição pode ser encontrada em [23].

Assim sendo, ao se trabalhar com o método Simplex, de posse de uma solução básica viável faz-se necessário verificar se $(c_j - z_j) \leq 0$, $\forall j \in I_N$. Quando esta condição é verificada, o método pode ser finalizado, pois já que encontamos de uma solução ótima. É nesse sentido que as literaturas sobre o assunto chamam tal procedimento de *teste de otimalidade* ou *critério de otimalidade*.

No método Simplex, uma matriz base B se diferencia da matriz base usada na iteração seguinte por apenas uma coluna. Como cada variável x_j pode ser associada a coluna a_j de A. Podemos estabelecer um critério para a seleção de uma variável x_j entrar na base. Para tal, os valores de $(c_j - z_j)$, $j \in I_N$ desempenham papéis fundamentais, entre eles o de garantir a otimalidade de uma solução básica viável. Quando o critério de otimalidade não é satisfeito, existe pelo menos um índice $j \in I_N$ tal que $(c_j - z_j) > 0$. Se tornarmos positiva a variável não básica x_k e mantivermos nulas as outras variáveis não básicas, ou seja, $x_j = 0 \ \ \forall j \in I_N$, $j \neq k$, então a função

objetivo (2.11) pode ser expressa simplesmente por:

$$z = \bar{z} - (c_k - z_k)x_k \tag{2.15}$$

Sabemos que $x_k \geq 0$, logo devemos somente analisar os possíveis valores de $(c_k - z_k)$. Neste caso, temos as seguintes possibilidades:

1. Se $(c_k - z_k) > 0$ o valor de z nessa iteração irá diminuir. Como queremos minimizar a função objetivo, devemos considerar que x_r é candidata a entrar na base, ou seja, a coluna a_k deverá compor a matrix básica B;

2. Se $(c_k - z_k) = 0$ o valor de z se manterá inalterado. Supondo que o problema não apresenta solução degenerada;

3. Se $(c_k - z_k) < 0$ o valor de z aumenta, como estamos trabalhamos com um problema de minimização, não é conveniente colocar x_k na base.

Devemos lembrar que nem sempre existe um único $k \in I_N$ tal que $(c_k - z_k) > 0$. A princípio qualquer coluna a_k com custo reduzido positivo poderia ser inserida na base, já que permitiria a redução do valor da função objetivo em nosso problema de minimização. No entanto, uma escolha razoável é inserir a coluna com maior custo reduzido, ou seja,

$$(c_k - z_k) = \max_{j \in I_N}\{c_j - z_j \mid (c_j - z_j) > 0\} \tag{2.16}$$

A escolha da coluna de maior custo reduzido não é ótima no sentido de reduzir o número de iterações, sendo apenas uma forma gulosa de seleção de colunas [44].

Critério para saída da base ou teste da razão

Vimos na seção anterior que o critério para entrada na base estabelece uma variável x_k para entrar na base. No entanto, a base B pode possuir apenas m variáveis básicas, sendo assim, é necessário agora estabelecer um critério que defina uma variável básica para deixar a base, tornando-se assim uma variável não-básica. A partir dos critérios de entrada e saída da base, o método Simplex a cada iteração, troca uma coluna da matriz N por uma coluna da matriz B, caminhando assim por pontos extremos adjacentes que são soluções viáveis do nosso problema de minimização.

Quando escolhemos a variável x_k para entrar na base com o objetivo de diminuir a função objetivo é conveniente que x_k assuma o maior valor possível, já que quanto maior for x_k mais a função objetivo função objetivo diminuirá. Porém precisamos manter a viabilidade da solução e para isso é feito pelo teste da razão, que nos diz o

quanto a variável x_k pode crescer, assumindo que todas as variáveis básicas devem permanecer não negativas.

Temos que a equação dada por (2.14) poderá ser reescrita sem o somatório, considerando que somente a variável x_k que entrará na base assumirá um valor não negativo e todas as demais variáveis não básicas permanecerão nulas. Sendo assim temos:

$$x_{B(i)} = \bar{x}_{B(i)} - y_{ik}x_k, \quad i = 1, ..., m \tag{2.17}$$

Para que o método Simplex mantenha a viabilidade da solução é necessário garantir que $x_{B(i)} \geq 0$, $i = 1, ..., m$ (2.17). Como o valor de x_k sai do valor zero e cresce para um valor positivo, é preciso analisar o que acontece com o valor de cada variável básica. Pela expressão (2.17), podemos observar que isso depende apenas do sinal de y_{ik}.

Analisando o sinal de y_{ik} verificamos as seguintes possibilidades:

1. Se $y_{ik} < 0$, então $x_{B(i)}$ aumenta, uma vez que já era positivo. Isso quer dizer que $x_{B(i)}$ associado pode crescer indefinidamente com o valor de x_k.

2. Se $y_{ik} = 0$, então o valor $x_{B(i)}$ de permanecerá inalterado.

3. Se $y_{ik} > 0$, então o valor de $x_{B(i)}$ decresce à medida que x_k cresce. Porém, devemos observar que $x_{B(i)}$ só pode decrescer até atingir o valor zero. Neste caso, para satisfazer a condição de não negatividade, vamos impor que $x_{B(i)} \geq 0$, isto é,

$$\begin{aligned} \bar{x}_{B(i)} - y_{ik}x_k &\geq 0 \\ -y_{ik}x_k &\geq -\bar{x}_{B(i)} \\ y_{ik}x_k &\leq \bar{x}_{B(i)} \\ x_k &\leq \frac{\bar{x}_{B(i)}}{y_{ik}} \end{aligned}$$

Os valores de $\bar{x}_{B(i)}/y_{ik}$ limitam o crescimento da variável x_k. Como a nova variável básica só poderá crescer até que a primeira componente $x_{B(i)}$ atinja o valor zero, é necessário tomar o mínimo entre todos os $\bar{x}_{B(i)}/y_{ik}$, $y_{ik} > 0$. Se $x_{B(s)}$ for essa componente, então temos que:

$$x_{B(s)} = \min_{1 \leq i \leq m} = \left\{ \frac{\bar{x}_{B(i)}}{y_{ik}} | y_{ik} > 0 \right\} \tag{2.18}$$

Nesse caso, é a variável básica $x_{B(s)}$ que deixará a base, tornando-se variável não

básica, pois é a primeira a variável a se anular com o crescimento de x_k. A expressão (2.18) é conhecida na literatura como *teste da razão*.

Note que se $y_{ik} \leq 0$, $i = 1, ..., m$, então o valor de x_k pode crescer indefinidamente mantendo a viabilidade da solução. E, neste caso, o valor da função objetivo $z = \bar{z} - (c_k - z_k)x_k$, se x_k crescer indefinidamente, tenderá a menos infinito ($z \to \infty$), o que significa que o problema é ilimitado.

Capítulo 3

Programação linear inteira

A *otimização combinatória* é um conjunto de técnicas e modelos que visam a alocação eficiente de recursos limitados quando algumas ou todas as variáveis de decisão são restritas a assumirem apenas valores inteiros. O horizonte de aplicação dessas técnicas e modelos é amplo. Encontramos contribuições em domínios como a logística, a gestão de operações e a gestão de projetos. Devido à grande variedade de aplicações, nas últimas décadas, a área da *programação inteira* registrou progressos notáveis tanto ao nível da definição de modelos analíticos, como no domínio das técnicas de resolução. Nesta seção, veremos mais adiante o *método de Decomposição de Dantzig-Wolfe* que consiste na reformulação do modelo original em modelos compactos, e o *método de geração de colunas* que tem por objetivo resolver de forma eficiente os modelos decompostos. Estas duas técnicas estão diretamente ligadas a proposta desta dissertação.

O problema de programação linear inteira

Problemas de programação linear inteiros são aqueles em que uma ou mais variáveis de decisão são inteiras. Tais problemas são mais difíceis e precisam de algoritmos mais complexos que o *Simplex*.

Considere o problema de Programação Linear Inteira (PI).

$$
\begin{aligned}
(PLI) \quad & \min \quad z = cx && (3.1)\\
& \text{s.a} \quad Ax = b && (3.2)\\
& \qquad\quad x \geq 0 \;\; e \;\; inteiro
\end{aligned}
$$

Sendo S um poliedro convexo, por exemplo $S = \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$ podemos escrever nosso (PLI) da seguinte forma reduzida:

$$(PLI) : z = min\{cx : x \in S \cap \mathbb{Z}^n_+\} \tag{3.3}$$

Note que os pontos extremos de S podem não ser pontos inteiros, ou seja, pontos em $\mathbb{Z}^n$. Seja $X = S \cap \mathbb{Z}^n$. Idealmente gostaríamos de conhecer o menor poliedro convexo que contém X ao qual dá-se o nome de *envoltória convexa* de X e denotada-se por $conv(X)$, uma vez que todos os pontos extremos deste poliedro são inteiros e, então, poderíamos utilizar o algoritmo Simplex para resolver o problema de (PLI) correspondente. Nem sempre é facil determinar tal conjunto, logo na tentativa de tentar obter uma aproximação da solução candidata a solução viável podemos usar um tipo de relaxação do problema inteiro original.

Relaxações e limitantes primais e duais

A habilidade de se estimar a qualidade de uma solução candidata é fundamental para otimização, sendo assim, é de extrema importância conhecer sob quais condições uma solução candidata pode ter sua qualidade estimada em relação à solução ótima, mesmo sem conhecermos a solução ótima. Este tipo de estimação é geralmente obtida através da solução ótima de problemas mais fáceis que, entretanto, possuem espaço de soluções mais abrangente que o do problema original. Estes problemas são ditos *relaxações*, podendo ser obtidos por meio da desconsideração de algumas restrições, da desconsideração das restrições de integralidade ou pela simplificação do problema.

Uma das técnicas para resolução de programação linear inteira (PLI) são os algoritmos do tipo *branch-and-bound*, onde o espaço de busca é decomposto e limitantes mais precisos são gerados iterativamente. Nestes algoritmos, precisamos de procedimentos que gerem os limites *(bounds)* que potencialmente nos levem ao valor ótimo inteiro z. Para isto, o método mais comumente utilizado é a resolução da relaxação linear do problema inteiro, obtida quando desconsideramos as restrições de integralidade de sua formulação, trasformando o (PLI) em um problema linear, digamos (PL).

Assim, como o conjunto viável do problema de programação linear inteiro é um subconjunto das soluções viáveis de um problema linear, quando tentamos resolver um problema de minimização, o valor ótimo da relaxação linear é um limite inferior *(lower bound)* ou *limitante dual* denotado por $\underline{z}$ para o valor ótimo do problema inteiro. O limitante superior é chamado de (*upper bound)* ou *limitante Primal* denotado por $\overline{z}$.

Em geral, a relaxação linear é muito mais fácil de ser resolvida que o problema inteiro; no entanto, seu valor ótimo pode estar bastante distante do valor ótimo inteiro. Essa "distância"é usualmente denominada *gap de integralidade*.

Portanto, um algoritmo para resolver um (PLI) precisa encontrar limitantes duais e primais que são melhorados iterativamente até que $\underline{z} - \overline{z} = \epsilon$ onde ϵ é um

valor não-negativo pequeno suficiente para provar a otimalidade, isto é, $\underline{z} = \overline{z}$.

O valor de qualquer solução viável dá um limitante primal para z.

Definição 3.1 *Um problema* (PL) $z^R = min\{f(x) : x \in T \cap \mathbb{R}^n_+\}$ *é uma relaxação do problema* (PLI) $z = Max\ \{c(x) : x \in S \cap \mathbb{R}^n_+\}$ *se:*

i. $X \subseteq T$;

ii. $f(x) \geq c(x)$ *para todo* $x \in S$.

Proposição 3.1 *Se* (PL) *é uma relaxação de* (PLI) *então* $z^R \geq z$.

Proposição 3.2 *Dados um problema linear inteiro* PLI *e sua relaxação linear* PL, *temos:*

i. *Se a relaxação* PL *é infactível, então o problema original* PLI *é infactível*

ii. *Seja* x^* *uma solução ótima para* PL. *Se* $x^* \in S \bigcap \mathbb{Z}^n_+$, *então* x^* *e* $f(x^*) = c(x^*)$, *então é uma solução ótima para* PLI.

3.1 Decomposição de Dantzig-Wolfe

O princípio da decomposição de Dantzig-Wolfe foi formalizado nos anos 1960 por George Dantzig e Philip Wolfe [12], mas na realidade o *método de geração de colunas* ao qual a decomposição está diretamente associada tinha sido já usado, em 1958, por Ford e Fulkerson [45] para resolver problemas de fluxo multicomodidade. Algum tempo depois, em 1984, Desrosiers, Soumis e Desrochers [46] reforçaram a importância da *decomposição de Dantzig-Wolfe* a associando às técnicas de *geração de colunas* e ao *método de branch-and-bound* para a resolução de problemas de programação inteira. Outros exemplos de aplicações da decomposição de Dantzig-Wolfe em programação inteira podem ser vistos no trabalho Barnhart, Johnson, Nemhauser entre outros [47].

Para definirmos formalmente o princípio da decomposição de Dantzig-Wolfe, consideramos a seguinte problema de programação linear:

$$
\begin{aligned}
(P) \quad \min \quad & z = cx && (3.4)\\
\text{s.a} \quad & Ax = b && (3.5)\\
& x \geq 0
\end{aligned}
$$

onde $c^T \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, e $b \in \mathbb{R}^m$.

A decomposição de Dantzig-Wolfe aplicada a esse modelo de programação linear consiste em separar o modelo em partes: algumas restrições ficam explicitamente no modelo, enquanto outras são removidas, passando a ser expressas de um modo alternativo e tratadas separadamente em um problema auxiliar.

Particionaremos A e b da seguinte maneira:

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

onde $A_1 \in \mathbb{R}^{m_1 \times n}$, $A_2 \in \mathbb{R}^{m_2 \times n}$, $b_1 \in \mathbb{R}^{m_1}$ e $b_2 \in \mathbb{R}^{m_2}$.

Logo, nosso modelo (P) ficaria definido da seguinte forma:

$$\begin{aligned} (P) \quad \min \quad & z = cx && (3.6) \\ \text{s.a} \quad & A_1 x = b_1 && (3.7) \\ & A_2 x = b_2 && (3.8) \\ & x \geq 0 \end{aligned}$$

A idéia básica da decomposição de Dantzig-Wolfe é representar certo conjunto convexo e limitado de soluções viáveis como uma combinação linear convexa dos pontos extremos e também das direções extremas desse conjunto. Então, inicialmente devemos escolher o grupo de restrições que definem um tal subconjunto viável convexo e limitado, substituindo-o em seguida pela combinação apropriada de suas direções extremas.

Essa idéia é formalizada no resultado proposto por Minkowski [48] e cujas demonstrações podem econtradas em [23, 49, 50]]

Proposição 3.3 *Consideremos o conjunto* $\mathcal{X} = \{x \in \mathbb{R}^n_+ \mid A_2 x = b_2\} \neq \emptyset$ *e denotemos* $V(\mathcal{X}) = \{v^1, v^2, ..., v^p\}$ *o conjunto (finito) dos p vértices (ou pontos extremos) de* $\mathcal{X}$ *e por* $R(\mathcal{X}) = \{r^1, r^2, ..., r^q\}$ *o conjunto dos raios extremos (ou direções extremas). Então* $x \in \mathcal{X}$ *se , e somente se,*

$$x = \sum_{j=1}^{p} \lambda_j v^j + \sum_{i=1}^{q} \mu_i r^i$$

para

$$\sum_{j=1}^{p} \lambda_j = 1, \quad \lambda_j \geq 0, \quad j = 1, 2, ..., p, \quad \mu_i \geq 0, \quad i = 1, 2, ..., q.$$

Na decomposição de Dantzig-Wolfe, substituimos as variáveis x no problema (P) pela expressão dada na proposição anterior. A formulação resultante é o seguinte

problema de programação linear:

$$(PM) \quad \min \quad z = c\left(\sum_{j=1}^{p}\lambda_j v^j + \sum_{i=1}^{q}\mu_i r^i\right) \tag{3.9}$$

$$\text{s.a} \quad A_1\left(\sum_{j=1}^{p}\lambda_j v^j + \sum_{i=1}^{q}\mu_i r^i\right) = b_1 \tag{3.10}$$

$$\sum_{j=1}^{p}\lambda_j = 1 \tag{3.11}$$

$$\lambda_j \geq 0, \quad j = 1, 2, ..., p, \quad \mu_i \geq 0, \quad i = 1, 2, ..., q. \tag{3.12}$$

ou ainda,

$$(PM) \quad \min \quad z = \sum_{j=1}^{p}(cv^j)\lambda_j + \sum_{i=1}^{q}(cr^i)\mu_i \tag{3.13}$$

$$\text{s.a} \quad \sum_{j=1}^{p}(A_1 v^j)\lambda_j + \sum_{i=1}^{q}(A_1 r^i)\mu_i = b_1 \tag{3.14}$$

$$\sum_{j=1}^{p}\lambda_j = 1 \tag{3.15}$$

$$\lambda_j \geq 0, \quad j = 1, 2, ..., p, \quad \mu_i \geq 0, \quad i = 1, 2, ..., q. \tag{3.16}$$

O problema que resulta da decomposição de Dantzig-Wolfe do modelo compacto original, sob a forma (3.9)-(3.12), é chamado de *problema mestre* (PM).

Considerando o problema mestre (PM), vamos definir a matriz dos coeficientes por:

$$\bar{A} = \begin{pmatrix} A_1v^1 & A_1v^2 & ... & A_1v^p & A_1r^1 & A_1r^2 & ... & A_1r^q \\ 1 & 1 & ... & 1 & 0 & 0 & ... & 0 \end{pmatrix} \in \mathbb{R}^{(m_1+1)\times(p+q)}$$

e os vetores

$$\bar{c} = \begin{pmatrix} cv^1 & cv^2 & ... & cv^p & cr^1 & cr^2 & ... & cr^q \end{pmatrix} \in \mathbb{R}^{1\times(p+q)}, \tag{3.17}$$

$$\bar{b} = \begin{pmatrix} b \\ 1 \end{pmatrix} \in \mathbb{R}^{(m_1+1)}, \tag{3.18}$$

$$\lambda^T = \begin{pmatrix} \lambda_1 & \lambda_2 & ... & \lambda_p \end{pmatrix} \in \mathbb{R}^{1\times p} \text{ e} \tag{3.19}$$

$$\mu^T = \begin{pmatrix} \mu_1 & \mu_2 & ... & \mu_q \end{pmatrix} \in \mathbb{R}^{1\times q}. \tag{3.20}$$

Sendo assim, em notação matricial, podemos reescrever o problema mestre (PM)

como o seguinte problema de programação linear:

$$(PM) \quad \min \quad \bar{z} = \bar{c}\bar{x} \tag{3.21}$$

$$\text{s.a} \quad \bar{A}\bar{x} = \bar{b} \tag{3.22}$$

$$\bar{x} \geq 0,$$

em que

$$\bar{x} = \begin{pmatrix} \lambda \\ \mu \end{pmatrix} \in \mathbb{R}^{p+q}.$$

O problema mestre é o problema que permanece com as restrições originais do modelo compacto, e no qual é reformulada a definição das variáveis para passarem a ser escritas como uma combinação de pontos e raios extremos de $\mathcal{X}$. O modelo passa a ter um número exponencial de colunas. O subproblema (ou eventualmente subproblemas se a decomposição for múltipla) está ligado às restrições que saíram do modelo compacto original.

Dado que o problema mestre (PM) tem um número exponencial de colunas, não é viável, mesmo com os avanços em termos de processamento e memória, a enumeração completa de todas as colunas que o compõem para qualquer instância de média ou grande dimensão. Como tal, quando se pretende resolver um modelo desse tipo, apenas um conjunto restrito de coluna que são enumerado à partida. A esse problema, dá-se o nome de *problema mestre restrito*. O subproblema é o problema que é resolvido para identificar colunas atrativas (com potencial para melhorar o valor da função objetivo do problema mestre) que não tenham estejam no problema mestre. Esta interação entre problema mestre restrito e subproblema define o método de geração de colunas que veremos na próxima seção.

3.2 Geração de colunas

A geração de colunas é uma ferramenta poderosa para resolver problemas de programação linear que surgiu na década de 60, coma idéia introduzida por Dantzig e Wolfe para decompor programas lineares de grande porte[12]. Posteriormente, o método foi empregado para resolver problemas de programação linear inteira, com o trabalho pioneiro de Gilmore e Gomory para o clássico problema de corte e empacotamento [13].

Esta técnica pode ser usada quando as colunas do problema não são conhecidas com antecedência e uma enumeração completa de todas as colunas não é uma opção viável, ou quando o problema é reescrito usando decomposição Dantzig-Wolfe. A técnica de geração de colunas é uma escolha natural em diversas aplicações, combinado com o algoritmo de branch-and-bound, que veremos mais adiante, tem sido

aplicado com sucesso na resolução de diversos tipos de problemas difíceis tais como, programação de rotas (*vehicle routing*) [16], coloração em grafos (*graph colouring*) [51], atribuição generalizada (*generalized assignment*) [52], corte (*cutting stock*) [53], empacotamento (*bin-packing*) [54], e dimensionamento de lote (*lot sizing*) [55].

Definido um conjunto de índices $J = \{1, 2, ..., n\}$, considere o seguinte programa linear:

$$(PM) \quad \min \quad \sum_{j \in J} c_j x_j \tag{3.23}$$

$$\text{s.a} \quad \sum_{j \in J} a_j x_j = b \tag{3.24}$$

$$x_j \geq 0 \quad j \in J$$

O problema acima é conhecido como *Problema Mestre* (PM) no contexto de geração de colunas, onde $b \geq 0 \in \mathbb{R}^m$, $a_j \in \mathbb{R}^m$, $c_j \in \mathbb{R}$, para cada $j \in J$. Suponha que $a_j \in \mathcal{K} = \{a_1, a_2, ..., a_n\}$ e que existe uma função $f : \mathcal{K} \to \mathbb{R}$ tal que $c_j = f(a_j)$.

Tal como já foi visto na seção anterior, existem muito mais variáveis no problema mestre (PM), que na formulação original do problema P, uma vez que se associa uma nova variável a cada ponto extremo do conjunto $\mathcal{X}$. Sendo assim, mesmo com os excelentes progressos em termos de capacidade de memória dos computadores atuais, torna-se inviável, e algumas vezes impossível, considerar todos os dados do problema na busca por uma solução ótima.

A principal motivação para o uso da técnica de geração de colunas, consiste em tentar buscar resolução do problema mestre (PM), sem ter de se enumerar explicitamente todos as colunas a ele associadas. A idéia base subjacente à geração de colunas pode definir-se do seguinte modo: em vez de considerarmos todos as colunas do conjunto J no problema mestre (PM), vamos considerar apenas um conjunto restrito dessas colunas obtidas pelo conjunto $J' \subseteq J$, definindo assim um *Problema Mestre Restrito* (PMR) e avaliar se existem colunas que não estão atualmente presentes no problema (PM), que caso fossem incluídos no problema poderiam melhorar o valor da sua função objetivo. As colunas que se revelarem atrativas devem então ser adicionados ao *Problema Mestre*.

Para iniciar o processo de resolução usando geração de colunas é necessário introduzir no *problema mestre restrito* uma base inicial viável. Normalmente é utilizada uma solução inicial de boa qualidade obtida através de um método heurístico (a heurística que propomos no capítulo 4 para a geração de colunas iniciais nos modelos discutidos nesta dissertação é baseada na construção de uma árvore de Gomory-Hu). No entanto, na falta de base viável melhor, pode ser utilizada uma matriz identidade, com penalizações adequadas nos coeficientes da função objetivo e a partir desta solução básica ir gerando as colunas apenas se (e quando) forem

realmente necessárias.

$$(PMR) \quad \min \quad \sum_{j \in J'} c_j x_j \tag{3.25}$$

$$\text{s.a} \quad \sum_{j \in J'} a_j x_j = b \tag{3.26}$$

$$x_j \geq 0 \;\; para \;\; todo \;\; j \in J'$$

Em cada iteração do método, faz-se a otimização do (PMR), obtendo-se uma solução que é ótima em termos das variáveis que nele estão incluídas. Contudo poderão existir colunas no conjunto J, que não façam parte atualmente do (PMR) que melhoraria a solução do (PM). Para avaliar a atratividade destas colunas temos de recorrer à solução ótima dual do (PMR). Sabemos que a base viável corrente está associada a uma solução ótima para o problema de programação linear de minimização apenas quando todos os custos reduzidos são não negativos, isto é equivalente a dizer que a solução dual correspondente também é viável. Podemos calcular os custos reduzidos para cada $j \in J$, por $\overline{c}_j = c_j - u^T a_j$, onde $u = c_B^T B^{-1} \in \mathbb{R}^m$ é a solução dual associada a base B, com $c_B^T = [c_{B(1)}, c_{B(2)} ... c_{Bm}]$. No entanto, o cálculo dos valores de todos os custos reduzidos seriam inviável dada carnalidade do conjunto J ser muito grande. De fato, não é necessário saber todos os custos reduzidos $\overline{c}_j$, $j \in J$ para decidir qual coluna melhoraria a minha solução, é necessário saber apenas o menor custo reduzido, e para isso, bastaria recorre à resolução de um subproblema usualmente denominado de *problema auxiliar* [23] ou *Subproblema Pricing*.

$$(Pricing): \; \min \quad f(a_j) - u^T a_j \tag{3.27}$$

$$s.a. \quad a_j \in \mathcal{K}. \tag{3.28}$$

Na forma clássica de geração de colunas, o algoritmo itera entre um *Subproblema Pricing* que é gerador de novas colunas e um *Problema Mestre Restrito*. A solução do (PMR) produz uma determinada solução dual, que é utilizada no subproblema visando determinar se existe alguma coluna que pode ser acrescentada ao problema mestre. As soluções geradas no *Subproblema Pricing* são colunas podem ser incluídas como novas variáveis no *Problema Mestre Restrito*. Estes dois problemas são resolvidos alternadamente, trocando informações entre eles, até se atingir uma solução que corresponde à solução do problema original.

Enquanto os subproblema conseguir gerar soluções com custos reduzidos negativos, as novas variáveis são candidatas a entrar na base do *Problema Mestre Restrito*. Quando *Subproblema Pricing* não conseguirem gerar variáveis atrativas, isto é, se o custo reduzido não for negativo, teremos encontrado a solução ótima do problema

mestre.

3.3 Branch-and-bound

O algoritmo branch-and-bound foi proposto em 1960 por Land e Doig [56] para a resolução de problemas de programação inteira, já o termo *"Branch-and-bound"* como conhecemos hoje, foi empregado pela primeira vez em 1963 por Little, Murty, Sweeney e Karel et al. [57]. O algoritmo utiliza a estratégia "dividir para conquistar", no sentido em que está baseado na inspeção de tão somente partes do conjunto de soluções viáveis e, assim, obtendo estimativas para o valor da solução ótima do problema inteiro.

De acordo com o algoritmo *Branch-and-bound*, as divisões são feitas iterativamente, sempre observando que os subproblemas devem ser mais fáceis de serem resolvidos que o problema original, além de se procurar descartar subproblemas por meio de enumeração implícita, isto equivale a dizer que,de alguma forma, podemos garantir que a solução ótima não é solução de um determinado subproblema, e assim descartá-lo.

3.3.1 Dividir para conquistar

Dada uma matriz $A \in \mathbb{R}^{m\times n}$ e vetores $b \in \mathbb{R}^m$ e $c^T \in \mathbb{R}^n$ com $0 < m \leq n$ considere o seguinte problema de programação inteira:

$$\begin{aligned} (PI) \quad \min \quad & z = cx \\ \text{s.a} \quad & Ax = b \\ & x \geq 0 \text{ e inteiro} \end{aligned}$$

seu conjunto viável é $S_I = \{x \in \mathbb{Z}^n \mid Ax = b,\ x \geq 0\}$.

Para problemas de programação inteira, como é o caso do problema em estudo, não são conhecidas condições gerais de otimalidade como existem para a programação linear [58]. Na tentativa de obter uma solução viável para nosso problema vamos considerar a seguinte relaxação do problema linear (PI) dada por:

$$\begin{aligned} (P) \quad \min \quad & z = cx \\ \text{s.a} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

cujo conjunto viável é $S = \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$.

Como foi dito acima o problema (PI) pode ser dividido em subproblemas me-

nores e depois recombinado de forma a obter a solução do problema original. Para isso, considere o conjunto S de soluções viáveis de (P) obtido quando realizamos uma relaxação linear de (PI) . Seja $S_1 \cup S_2 \cup ... \cup S_K$ uma decomposição de S em conjuntos menores de modo que de modo que $S = S_1 \cup S_2 \cup ... \cup S_K$ e seja $z^k = \min\{cx : x \in S_k\}$ para $k = 1, 2, ...K$. Logo temos que o valor de $z = \min_k z_k$. Observe que cada subproblema pode ser decomposto em outros subproblemas menores ainda, e assim sucessivamente, até que cada subconjunto contenha um número suficientemente pequeno de pontos para que o subproblema seja resolvido trivialmente.

Podemos representar essa decomposição através de uma *árvore de enumeração*, na qual cada nó representa um subconjunto do espaço de soluções e cada nó filho, uma componente de sua decomposção. A raiz da árvore representa o conjunto S, portanto, todo o espaço de soluções do problema. Este processo de decomposição dos cunjuntos de soluções também é conhecido como "*ramificação*" ou *branching*.

Algoritmos de branch-and-bound tiram proveito da decomposição do problema e evitam a enumeração explícita de toda a árvore algo que seria muito custoso tanto em tempo quanto em espaço. Além disso, os algoritmos de branch-and-bound buscam a enumeração de suas soluções com auxílio de limitantes duais e primais.

Inicialmente, *árvore de branching* de um algoritmo branch-and-bound contém apenas o nó raiz, que está associado ao problema (P). Um limitante dual é obtido resolvendo-se a relaxação linear de (P). Caso a solução ótima da relaxação linear de (P) seja inteira, a solução ótima de (PI) foi encontrada. Caso contrário, uma operação de *branching* é realizada, isso consistem particionar o conjunto viável em dois subconjuntos através da imposição, respectivamente, das restrições $x_j \leq \lfloor \hat{x_j} \rfloor$ e $x_j \geq \lceil \hat{x_j} \rceil$, onde $\hat{x_j}$ é uma variável de valor fracionário da solução obtida através da relaxação linear (P), na iteração corrente. A árvore de branching, neste caso, é binária.

Cada subproblema, por sua vez, dá origem a outros subproblemas, sendo criada uma árvore neste processo de enumeração, cujos nós correspondem aos subconjuntos criados. Para um dado nó da árvore, a regra de partição deve eliminar a solução fracionária do respectivo subproblema, e garantir que nenhuma das soluções válidas do problema original é perdida.

Um nó pode sofrer uma "poda", ou *bounding*, que consiste em encontrarmos a melhor solução contida em cada subconjunto de S obtido na operação de *branching*.

Podemos realizar o a operação de *bounding* em uma árvore nos seguintes casos:

- Se o problema associado a ele é inviável;
- Já conhecemos sua solução ótima;
- Se o limitante dual do nó é menor que um limitante primal para (P).

Os nós que não foram podados e ainda não foram decompostos por operação de branching são chamados **nós ativos**. Os nós ativos são os únicos que podem gerar um limitante primal melhor para (P). Sendo assim, a otimalidade pode ser encontrada pelo algoritmo de *branch-and-bound* quando não há nós ativos ou quando o maior dos limitantes duais dos nós ativos for menor que o melhor limitante primal para (P). Portanto, quanto melhor forem os limitantes duais e primais, melhor será o desempenho de um algoritmo de *branch-and-bound*.

3.3.2 Critério de seleção do nó

Para determinarmos a escolha do proximo nó ativo a ser otimizado podemos escolher diversos critérios de pesquisa que variam de acordo com as características particulares de cada tipo problema. Os critérios de busca mais usados são: pesquisa em profundidade, pesquisa em largura e pesquisa do melhor limitante.

- Pesquisa em profundidade (*depth first search*) é um método em que são explorados em primeiro lugar os nós com uma maior profundidade em relação à raiz da árvore de pesquisa. Depois de cada partição é explorado o nó descendente mais à esquerda.

- Pesquisa em largura (*breadth first search*) é um método em que são explorados em primeiro lugar todos os nós de um mesmo nível de profundidade antes de passar aos nós do nível seguinte, começando pelo nó mais à esquerda.

- Pesquisa do melhor limitante (*best first*) é um método em que são explorados em primeiro lugar os nós com o valor de solução mais próximo do valor da solução na raiz da árvore de branching.

Em geral, o critério de seleção de nó com melhor limitante apresenta um resultado melhor. No entanto, a pesquisa por profundidade tem a vantagem de encontrar soluções primais mais cedo.

Além disso, estes critérios de pesquisa podem ainda ser combinados e dar lugar a regras de pesquisas compostas. Um exemplo de regra de pesquisa composta é a regra *depth first search* mais *best first* que funciona da seguinte forma: se o nó em avaliação não é abandonado usada a regra *depth first search* para selecionar o próximo nó da árvore de pesquisa a ser avaliado; caso contrário, é usada a regra de *best first*.

Capítulo 4

O problema de agrupamento

Considere um grafo conexo não direcionado $G(V, E)$ formado por vértices $v_i \in V$ com $|V| = n$ e por arestas $\{i, j\} \in E$ com pesos $w_{ij} \in \mathbb{R}$. Além disso, tome k como sendo o inteiro que define o número de componentes conexas em que desejamos particionar o grafo G.

Diremos que uma família de conjuntos $S_1, S_2, S_3, \ldots, S_k$ não vazios define uma k-partição de V (ou simplesmente do grafo G) se

(Particionamento.1) $S_1 \cup S_2 \cup \ldots \cup S_k = V$;

(Particionamento.2) $S_t \cap S_{t'} = \emptyset$, se $t \neq t'$.

Por conveniência, adotaremos uma representação binária para os subconjuntos $S_t \subset V$ utilizando vetores de incidência $a_t = (a_t^1, a_t^2, ..., a_t^n)^T \in \{0, 1\}^n$ tais que

$$a_t^i = \begin{cases} 1 & : \text{se o vértice } v_i \in S_t; \\ 0 & : \text{caso contrário.} \end{cases} \tag{4.1}$$

Com esta representação binária por vetores de incidência, é fácil ver que uma família $S_1, \ldots, S_k$ representada por vetores $a_1, \ldots, a_k$ define um particionamento se, e somente se, as sentenças abaixo forem verdadeiras

$$\sum_{i=1}^{n} a_t^i \;\; \geq \;\; 1, \;\; t = 1, \ldots, k, \tag{4.2}$$

$$\sum_{t=1}^{k} a_t \;\; = \;\; (1, 1, \ldots, 1)^T \in \{0, 1\}^n. \tag{4.3}$$

Dada a relação biunívoca entre um subconjunto S_t e sua representação binária a_t, não faremos distinção entre S_t e a_t. Logo, por $v_i \in a_t$ queremos dizer $v_i \in S_t$, o mesmo valendo para as demais operações envolvendo conjuntos.

No problema *k-cluster* (ou k-agrupamento), o objetivo é obter uma k-partição $a_1, \ldots, a_k$ de custo mínimo que induza subgrafos $G_1, \ldots, G_k$ conexos. Neste pro-

blema, o custo $c(a_1, \ldots, a_k) = \sum_{t=1}^{k} c(a_t)$ da k-partição $a_1, \ldots, a_k$ é dado pela soma dos pesos das arestas envolvendo apenas os vértices em uma mesma componente, ou seja,

$$c(a_1, a_2, \ldots, a_k) = \sum_{t=1}^{k} c(a_t) \tag{4.4}$$

$$= \sum_{t=1}^{k} \sum_{\{i,j\} \in E} w_{ij} a_t^i a_t^j. \tag{4.5}$$

Com as definições dadas anteriormente, o problema k-cluster pode ser modelado da seguinte forma

$$(P) : \min \quad \sum_{t \in \Gamma} c(a_t) x_t$$

$$s.a. \quad \sum_{t \in \Gamma} a_t x_t = \mathbf{e}, \tag{4.6}$$

$$\sum_{t \in \Gamma} x_t = k, \tag{4.7}$$

$$x \in \{0,1\}^{|\Gamma|}, \tag{4.8}$$

onde Γ é o conjunto formado pelos índices de todos os subconjuntos de V que induzem subgrafos conexos e $\mathbf{e} = (1, 1, \ldots, 1) \in \{0,1\}^n$.

A restrição (4.6) do problema (P) garante que os vetores a_t na solução (os associados a variáveis x_t tais que $x_t = 1$) formam um particionamento do conjunto de vértices de G. Já a restrição (4.7) faz com que, na solução, o número de subconjuntos (vetores a_t) seja igual a k. Finalmente, a restrição (4.8) força a integralidade da solução. Desta forma, o conjunto viável do problema (P) é formado por todas as k-partições de V que induzem subgrafos conexos.

Embora (P) seja um modelo bastante conciso para o problema da k-partição máxima, ele não é adequado pois supõe que o conjunto Γ e, consequentemente, todos os subgrafos conexos de G sejam conhecidos *a priori*. Em um grafo $G(V, E)$ com $|V| = n$, teríamos que checar quais dos $2^n - 1$ subconjuntos de V induzem subgrafos conexos. Algo inviável mesmo para problemas de tamanho reduzido.

Os dois modelos que proporemos nesta dissertação buscam a resolução de (P) sem a enumeração explícita do conjunto Γ. Para isso, ao invés de determinarmos *a priori* todas colunas (vetores a_t), procuramos gerar de forma iterativa apenas as colunas que podem contribuir para o aumento do custo do particionamento a cada iteração. A racionalidade de tal abordagem reside no fato conhecido de que as soluções dos problemas de programação linear não envolvem todas as colunas da definição do problema, mas apenas aquelas que definem a solução básica ótima (vide

capítulo 2). Em nosso problema em particular, isto é ainda mais evidente já que a solução deve ser composta por exatamente k colunas (componentes).

4.1 Propostas de enumeração

Nesta seção, derivaremos dois modelos baseados no método de geração de colunas para o problema k-cluster. Para isso, inicialmente, tomamos a relaxação linear do problema (P), obtendo desta forma o problema

$$(PL): \min \quad \sum_{t\in\Gamma} c(a_t)x_t \tag{4.9}$$

$$s.a. \quad \sum_{t\in\Gamma} a_t x_t = \mathbf{e}, \tag{4.10}$$

$$\sum_{t\in\Gamma} x_t = k, \tag{4.11}$$

$$x_t \geq 0, t \in \Gamma, \tag{4.12}$$

onde a restrição de integralidade (4.8) é substituída por um restrição de não-negatividade (4.12).

Note que o conjunto viável do problema (P) está contido no conjunto viável do problema relaxado (PL), logo a solução ótima de (PL) fornece um limitante inferior de (P).

Seja $A = \begin{bmatrix} a_1 & a_2 & \dots & a_{|\Gamma|} \\ 1 & 1 & \dots & 1 \end{bmatrix}$ a matriz $(n+1) \times |\Gamma|$ cujas colunas são formadas pelos vetores binários a_t com $t \in \Gamma$ acrescidos de uma componente de valor igual a uma unidade (última linha da matriz). Utilizando a matriz A, podemos reescrever o problema (PL) na forma compacta

$$(PL): \min \quad \sum_{t\in\Gamma} c(a_t)x_t \tag{4.13}$$

$$s.a. \quad Ax = [\mathbf{e}, k]^T \tag{4.14}$$

$$x_t \geq 0, \ t \in \Gamma \tag{4.15}$$

A matriz A pode ser decomposta em $A = [B, N]$, onde B é uma matriz $(n+1) \times (n+1)$ formada por $n+1$ colunas linearmente independentes de A, e N é a matriz formada pelas colunas restantes. A cada decomposição $[B, N]$, podemos associar uma solução básica da forma $x = (x_B, x_N)$ com $x_N = 0$ e $x_B = B^{-1}[\mathbf{e}, k]$.

Pelo *Teorema Fundamental da Programação Linear*, sabemos que a busca por soluções ótimas para o problema (PL) pode se concentrar exclusivamente no conjunto das soluções básicas viáveis, i.e., soluções $x = (x_B, 0) \geq 0$ [23, 59]. Desta forma, é suficiente determinar/gerar apenas as colunas de A que estejam associadas

à uma matriz B ótima. Ou seja, a princípio, não precisaríamos conhecer todos os subconjuntos indexados por Γ.

No entanto, resta ainda o desafio de determinar quais colunas devem ser geradas para a obtenção da solução ótima de (PL). A alternativa utilizada nos métodos de geração de colunas é partir de um conjunto inicial Γ' e gerar gradativamente novas colunas somente quando necessário. As colunas geradas são adicionadas ao subconjunto Γ' dando origem ao *problema mestre restrito* (PMR) definido pelas mesmas restrições de (PL) mas envolvendo apenas o subconjunto das colunas indexadas por Γ'.

Como destacado em [60], os métodos de geração de colunas seguem as seguintes linhas gerais:

a. Otimização do (PMR) para a determinação do valor da solução corrente

$$\bar{z} = \sum_{t \in \Gamma'} c(a_t) x_t \tag{4.16}$$

e dos multiplicadores u, v associados às restrições (4.11) e (4.12);

b. Determinar, caso exista, uma nova coluna $[a, 1]$ com custo reduzido

$$c_B(a) = c(a) - u^T a - v < 0, \text{ onde } \begin{bmatrix} u \\ v \end{bmatrix} = c_B B^{-1}$$

e inserí-la na base visando assim reduzir o valor da função objetivo.

O processo se repete até que não existam mais colunas com custo reduzido negativo. Note que o conjunto viável de (PL) é finito, portanto este procedimento sempre convergirá para uma solução ótima (caso exista).

A princípio, qualquer coluna com custo reduzido negativo poderia ser inserida na base, já que permitiria a redução do valor da função objetivo. No entanto, uma escolha razoável é inserir a coluna com menor custo reduzido. Apesar da razoabilidade da escolha da coluna com menor custo reduzido, vale a pena destacar que os estudos realizados por Forrest e Goldfarb [61] sobre os critérios de seleção para o pivoteamento no simplex mostram que se escolhermos a coluna com custo reduzido mais negativo para entrar na base nem sempre obteremos melhor desempenho na resolução do problema mestre linear. Ou seja, a escolha da coluna de menor custo reduzido não é ótima no sentido de reduzir o número de iterações, sendo apenas uma forma gulosa de seleção de colunas [44].

A busca pela coluna com menor custo reduzido dá origem ao subproblema

$$(Pricing) : \min \quad c(a) - u^T a - v \tag{4.17}$$

$$s.a. \quad a \in \{a_t : t \in \Gamma\}. \tag{4.18}$$

A função de custo do problema ($Pricing$) não é das mais convenientes já que não é linear, mas pode ser linearizada. A proposta de linearização de Fortet [62] para o termo quadrádico

$$c(a) = \sum_{\{i,j\} \in E} w_{ij} a^i a^j,$$

é utilizar variáveis s_{ij} ao invés do produto $a^i a^j$. Desta forma obtemos a linearização

$$(LPricing): \min_{a,s} \quad \sum_{\{i,j\} \in E} w_{ij} s_{ij} - u^T a - v \tag{4.19}$$

$$s.a. \quad a \in \{a_t : t \in \Gamma\} \tag{4.20}$$

$$s_{ij} \leq a^i \tag{4.21}$$

$$s_{ij} \leq a^j \tag{4.22}$$

$$a^i + a^j \leq s_{ij} + 1 \tag{4.23}$$

$$s_{ij} \geq 0, 1 \leq i < j \leq n. \tag{4.24}$$

As equações (4.21)-(4.24) garantem que $a^i a^j = s_{ij}$ para todos os valores possíveis de a^i e a^j. Esta linearização adiciona $|E| = m$ variáveis s_{ij} ao número de variáveis já existentes e $4m$ restrições às restrições do problema original.

No problema ($LPricing$) ainda temos uma representação implícita dos subgrafos conexos (eq. 4.20). Para obtermos uma representação explícita, propomos duas alternativas baseadas em modelos de fluxo para obtenção de subgrafos conexos. A primeira delas é baseada na construção de um modelo de fluxo único e a segunda em um modelo de multifluxo.

4.1.1 Modelo de fluxo único: OneFlow

A primeira alternativa chamada *OneFlow* baseia-se na construção de um grafo direcionado G' formado pelos vértices do grafo original G e um vértice artificial de rótulo zero. A cada aresta $\{i, j\}$ de G associamos dois arcos direcionados (i, j) e (j, i) em G'. Além destes arcos, criamos também arcos ligando o nó artificial zero a cada um dos vértices de G.

Formalmente, dado o grafo original não-orientado $G = G(V, E)$, construímos um grafo orientado $G' = G'(E', V')$, onde

$$V' = \{0\} \cup V \text{ e } E' = \{(i, j), (j, i) : \{i, j\} \in E\} \cup \{(0, j) : j \in V\}.$$

A figura 4.1 ilustra a relação entre o grafo original G e o grafo derivado G' para um exemplo com cinco vértices e sete arestas. Na ilustração da figura 4.1, o grafo G' associado ao modelo Oneflow possui uma vértice a mais que o grafo original G e $19 = 2 \times 7 + 5$ arestas, sendo duas arestas para cada uma das arestas de G e

mais cinco arestas que ligam o vértice artificial zero a cada um dos cinco vértices originais.

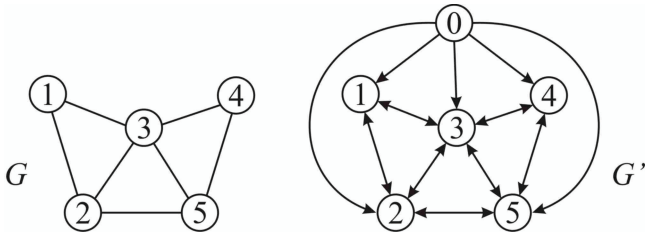

Figura 4.1: O grafo original G e o grafo G' associado ao modelo OneFlow.

No modelo OneFlow, utilizando as variáveis $z_{ij} \geq 0$ para representar o fluxo no arco $(i,j) \in E'$ e a variável $y_i \in \{0,1\}^n$ para indicar o vértice de entrada do fluxo, garantimos a conexidade do subgrafo induzido por a impondo as seguintes restrições:

$$\sum_{i=1}^{n} a^i \geq 1, \tag{4.25}$$

$$\sum_{j=1}^{n} y_j = 1, \tag{4.26}$$

$$z_{0j} \leq n y_j, \qquad j \in V, \tag{4.27}$$

$$\sum_{j \in V} z_{0j} = \sum_{i=1}^{n} a^i, \tag{4.28}$$

$$\sum_{j:(i,j)\in E'} z_{ij} - \sum_{j:(j,i)\in E'} z_{ji} = -a^i, \qquad i \in V, \tag{4.29}$$

$$z_{ij} + z_{ji} \leq n s_{ij}, \qquad [i,j] \in E. \tag{4.30}$$

A restrição (4.25) garante que o subgrafo induzido por $a = (a^1, \ldots, a^n)$ será formado por pelo menos um vértice. As restrições (4.26)-(4.27) fazem com que todo o fluxo saindo da raiz passe por um único vértice. A restrição (4.28) garante que o fluxo total saindo do vértice zero será igual ao número de vértices do subgrafo. A restrição (4.29) faz com que cada vértice utilizado no fluxo consuma uma unidade do fluxo. E, finalmente, a restrição (4.30) faz com que o fluxo só percorra arestas com extremos no subgrafo.

Em síntese, as restrições (4.25)-(4.31) modelam um fluxo que tem como fonte apenas o nó artificial zero e sumidouro os vértices por ele utilizados. A intensidade do fluxo é igual ao número de vértices cobertos por ele, cada um dos vértices cobertos consome uma unidade do fluxo e, além disso, só é permitido ao fluxo passar de um vértice para outro vértice vizinho. Como consequência, os vértices utilizados pelo fluxo formarão um subgrafo conexo. E, portanto, as restrições (4.25)-(4.27) fornecem um modelo explícito para os subgrafos conexos de G.

Substituindo a restrição (4.20) do ($LPricing$) pelas restrições (4.25)-(4.30) obtemos o problema

$$
\begin{aligned}
(OneFlow): \min_{a,s,y,z} \quad & \sum_{\{i,j\}\in E} w_{ij}s_{ij} - u^T a - v \\
s.a. \quad & s_{ij} \leq a^i, \\
& s_{ij} \leq a^j, \\
& a^i + a^j \leq s_{ij} + 1, \\
& s_{ij} \geq 0, 1 \leq i < j \leq n, \\
& \sum_{i=1}^{n} a^i \geq 1, \\
& \sum_{j=1}^{n} y_j = 1, \\
& z_{0j} \leq n y_j, \;\; j \in V, \\
& \sum_{j\in V} z_{0j} = \sum_{i=1}^{n} a^i, \\
& \sum_{j:(i,j)\in E'} z_{ij} - \sum_{j:(j,i)\in E'} z_{ji} = -a^i, \;\; i \in V, \\
& z_{ij} + z_{ji} \leq n s_{ij}, [i,j] \in E, \\
& z_{ij} \geq 0, \;\; \forall [i,j] \in E', \\
& y_j, a^j \in \{0,1\}, \;\; \forall j \in V'.
\end{aligned}
$$

4.1.2 Modelo de multifluxo: MultiFlow

Demos o nome *MultiFlow* à nossa segunda alternativa para a enumeração de subgrafos conexos. Esta alternativa, como o nome sugere, baseia-se na construção de um modelo de fluxo múltiplo. Enquanto que no modelo OneFlow construímos um fluxo único que parte do vértice zero e percorre os demais vértices, no modelo MultiFlow construímos n fluxos indexados por uma variável $k = 1, 2, \ldots, n$, uma para cada vértice de G. Neste modelo, o k-ésimo fluxo tem como fonte o nó artificial zero e sua intensidade será igual a um se o vértice k for coberto e zero caso contrário. O único sumidouro do k-ésimo fluxo é precisamente o vértice de índice k e, para que haja conservação, impomos que o sumidouro tem intensidade igual a menos uma unidade se o vértice k for coberto e zero caso contrário. A idéia é construir, para o k-ésimo vértice no subgrafo, um fluxo de uma unidade que parte do nó artificial zero e é absorvido exclusivamente pelo vértice k. Novamente, restrigimos a passagem do fluxo exclusivamente entre vértices vizinhos.

Formalmente, assim como no modelo OneFlow, considere o grafo orientado

$G(V', E')$, onde

$$V' = \{0\} \cup V \text{ e } E' = \{(i,j),(j,i) : \{i,j\} \in E\} \cup \{(0,j) : j \in V\}. \tag{4.31}$$

Além disso, sejam $z_{ij}^k \geq 0$ a variável que representa o k-ésimo fluxo no arco $(i,j) \in E'$, e $y_i \in \{0,1\}^n$ a variável que indica o vértice de entrada de todos os fluxos. Com estas definições, a conexidade do subgrafo induzido pelo vetor $a = (a^1, a^2, \ldots, a^n)$ no modelo MultiFlow é garantida pelo seguinte conjunto de restrições

$$\sum_{i=1}^{n} a^i \geq 1, \tag{4.32}$$

$$\sum_{j=1}^{n} y_j = 1, \tag{4.33}$$

$$z_{0j}^k \leq y_j, \qquad j \in V, \tag{4.34}$$

$$\sum_{j \in V} z_{0j}^k = a^k, \qquad k \in V, \tag{4.35}$$

$$\sum_{j:(i,j)\in E'} z_{ij}^k - \sum_{j:(j,i)\in E'} z_{ji}^k = 0, \qquad i \in V - \{k\}, k \in V, \tag{4.36}$$

$$\sum_{j:(k,j)\in E'} z_{kj}^k - \sum_{j:(j,k)\in E'} z_{jk}^k = -a^k, \qquad k \in V, \tag{4.37}$$

$$z_{ij}^k + z_{ji}^k \leq s_{ij}, \qquad [i,j] \in E. \tag{4.38}$$

A restrição (4.32) garante que o subgrafo induzido por $a = (a^1, \ldots, a^n)$ será formado por pelo menos um vértice. As restrições (4.33)-(4.34) fazem com que todo o fluxo saindo da raiz passe por um único vértice. A restrição (4.35) garante que o k-ésimo fluxo total saindo da fonte será igual a uma unidade se o vértice k for coberto pelo fluxo e zero em caso contrário. As restrições (4.36)-(4.37) fazem com que apenas o vértice k funcione como um sumidouro. E, finalmente, a restrição (4.38) faz com que o fluxo só percorra arestas com extremos no subgrafo.

Substituindo a restrição (4.20) do ($LPricing$) pelas restrições (4.32)-(4.38) ob-

temos o problema

$$
\begin{aligned}
(MultiFlow): \min_{a,s,y,z} \quad & \sum_{\{i,j\}\in E} w_{ij}s_{ij} - u^T a - v \\
s.a. \quad & s_{ij} \leq a^i, \\
& s_{ij} \leq a^j, \\
& a^i + a^j \leq s_{ij} + 1, \\
& s_{ij} \geq 0, 1 \leq i < j \leq n, \\
& \sum_{i=1}^{n} a^i \geq 1, \\
& \sum_{j=1}^{n} y_j = 1, \\
& z_{0j}^k \leq y_j, \ \ j \in V, \\
& \sum_{j\in V} z_{0j}^k = a^k, \ \ k \in V, \\
& \sum_{j:(i,j)\in E'} z_{ij}^k - \sum_{j:(j,i)\in E'} z_{ji}^k = 0, \ \ \in V - \{k\}, k \in V, \\
& \sum_{j:(k,j)\in E'} z_{kj}^k - \sum_{j:(j,k)\in E'} z_{jk}^k = -a^k, \ \ k \in V, \\
& z_{ij}^k + z_{ji}^k \leq s_{ij}, [i,j] \in E', \\
& z_{ij}^k \geq 0, \ \ \forall [i,j] \in E', \\
& y_j, a^j \in \{0,1\}, \ \ \forall j \in V.
\end{aligned}
$$

4.2 Obtendo soluções inteiras

Os dois modelos de fluxo apresentados possibilitam a geração das colunas necessárias para a solução do problema (*Pricing*). No entanto, um cojunto inicial de colunas dever ser fornecido *a priori* para a definição dos valores iniciais das variáveis duais (u e v). Por isso, propomos como alternativa a utilização de uma heurística bastante simples e relativamente rápida que para a geração do conjunto inicial de colunas viáveis.

4.2.1 Gerando um conjunto inicial de colunas

Antes de apresentarmos nossa heurística para geração do conjunto inicial de colunas, será útil definirmos os conceitos de *colapso de um subconjunto de vértices* e de *corte separador* (ou *corte*).

O colapso de um conjunto de vértices é a substituição de um dado conjunto de vértices por um único vértice. Ao colapsarmos um subconjunto $S \subset V$ de um

grafo não-orientado $G(V, E)$ em um único vértice s, damos origem a um novo grafo $G'(V', E')$ tal que $V' = (V - S) \cup \{s\}$ e

$$E' = \{\{i, j\} \in E : i, j \in V - S\} \cup \{\{i, s\} : \exists \{i, j\} \in E \text{ com } i \in V - S \text{ e } j \in S\}.$$

Os pesos das arestas $\{i, s\}$ geradas quando colapsamos $S \subset V$ são dados pela soma das arestas envolvendo o vértice i e os vértices em S, i.e.,

$$w(\{i, s\}) = \sum_{\substack{j:\{i,j\}\in E \\ j\in S}} w(\{i, j\}).$$

A figura 4.2 ilustra o colapso de um grafo G formado por 6 vértices e 10 arestas.

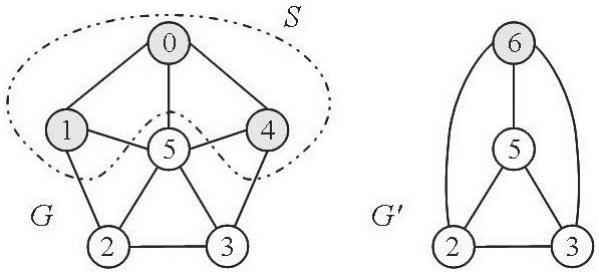

Figura 4.2: O grafo G' gerado pelo colapso do conjunto $S = \{0, 1, 4\}$.

Um *corte separador* (ou simplesmente *corte*) é um caso especial de particionamento. Dados um grafo $G(V, E)$ e dois vértices $i, j \in V$, diremos que um subconjunto de arestas define um corte $i - j$ se a sua remoção de G coloca os vértices i e j em partições distintas. Encontrar um corte $i - j$ mínimo, i.e., cujas somas dos pesos de suas arestas seja o menor possível ocupa um lugar central na otimização combinatória [63]. Existem diversas abordagens para determinar o corte mínimo em um grafo, sendo a mais antiga baseada na solução de problemas de *fluxo máximo*. Os primeiros algoritmos utilizados para solucionar o problemas envolvendo fluxos máximos em grafos foram propostos em 1962 por Ford e Fulkerson [64], e são baseados nas idéias de *caminhos elementares*. Segundo [65], novas implementações deste algoritmo tem melhorado o tempo de execução à medida que novas pesquisas são realizadas. A relação entre problemas de cortes e fluxos é estabelecida pelo teorema de *corte mínimo - fluxo máximo*, o qual garante que o fluxo máximo entre dois vértices i e j de um grafo é determinado pelo menor corte $i - j$ existente no mesmo.

A heurística que propomos para a geração de colunas iniciais é baseada na construção de uma árvore de *Gomory-Hu* para o grafo $G(V, E)$. Uma árvore de Gomory-Hu é uma representação dos cortes mínimos $i - j$ em G, para cada par de vértices $i, j \in V$ [66, 67]. Dizemos que uma árvore T é uma árvore de Gomory-Hu de um grafo $G(V, E)$ se

(a) para cada par de vértices $i, j \in V$, o peso do corte mínimo entre i e j em G for o mesmo que em T;

(b) para cada aresta $\{i, j\} \in T$, o peso $w(\{i, j\})$ de $\{i, j\}$ em T for igual ao peso do corte $i - j$ mínimo em G.

A partir destas propriedades, se a aresta de menor peso no caminho entre i e j em T for $\{u, v\}$, então o peso do corte $i - j$ mínimo em G será igual ao peso de $\{u, v\}$ em T e, mais ainda, as componentes (partições) produzidas ao removermos $\{u, v\}$ de T serão as mesmas produzidas pelo corte $i - j$ mínimo em G. Ou seja, há uma relação biunívoca entre os cortes mínimos e as arestas da árvore de Gomory-Hu.

Note que, em um grafo G com n vértices, necessitamos apenas das $n - 1$ arestas da árvore de Gomory-Hu para representarmos todos os cortes mínimos entre os $\binom{n}{2}$ pares de vértices em G. Por isso, dizemos que a árvore de Gomory-Hu é uma representação concisa dos cortes mínimos entre todos os pares de vértice de $G(V, E)$.

Na figura 4.3, exibimos uma árvore de Gomory-Hu para um grafo G não-direcionado formado por 6 vértices e 11 arestas.

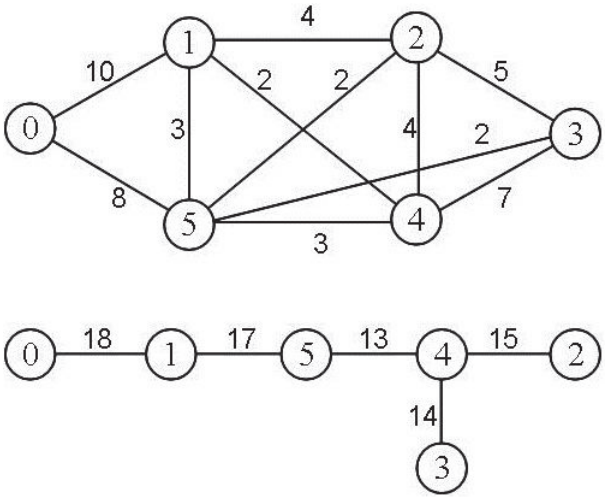

Figura 4.3: O grafo não-direcionado G e uma árvore de Gomory-Hu associada.

A árvore de Gomory-Hu de um grafo não-direcionado pode ser construída calculando-se exatamente $n - 1$ fluxos máximos [67]. A ideia para construção da árvore de Gomory-Hu é gerar uma árvore cujos vértices são subgrafos colapsados associados a conjunto de cortes mínimos. Especificamente, posicionamos o grafo original $G(V, E)$ na raiz da árvore. Em seguida, selecionamos dois vértices, digamos i e j em V, e determinamos o fluxo máximo entre i e j. O próximo passo é aplicar em $G(V, E)$ o corte mínimo associado. Desta forma, passamos a ter dois grafos, digamos G_i e G_j, gerados pelo colapso dos vértices na partição que contém i e o pelo colapso dos vértices na partição que contém j. Aos grafos G_i e G_j associamos uma aresta cujo peso é igual ao fluxo máximo entre i e j em G. Se um dos grafos

gerados tiver mais de dois vértices não-artificiais (vértices que não são resultado de colapso), repetimos o procedimento. Ao final, teremos uma árvore cujos nós são grafos com um único vértice não-artificial e arestas cujos pesos são os pesos dos cortes mínimos. A árvore de Gomory-Hu é obtida substituindo-se cada um destes grafos pelo seu respectivo vértice não-artificial (Ver figura 4.4).

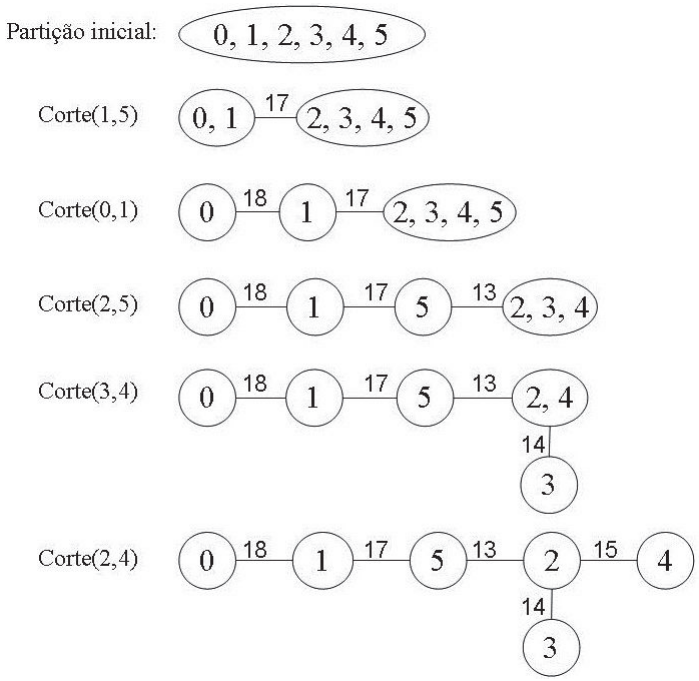

Figura 4.4: Construção de uma árvore de Gomory-Hu.

A ideia básica de nossa heurística é ordenar as arestas da árvore de Gomory-Hu e aplicar em G os cortes associados às arestas mais pesada até que o grafo G tenha sido particionado em k componentes. Esta heurística é uma adaptação do algoritmo aproximativo proposto por Saran e Vazirani para resolução do problema *min k-cut* [68].

O maior custo desta heurística é a determinação do número de componentes a cada iteração. Em um grafo $G(V, E)$ formado por $|V|$ vértices e $|E|$ arestas, utilizando o algoritmo de Kruskal, podemos fazer isto em $O(|E| + x \times \log(|V|))$, onde x representa o número de arestas menores ou iguais a maior aresta na árvore geradora mínima de G [69].

4.2.2 Obtendo soluções inteiras

Tendo o conjunto inicial de colunas, podemos aplicar o método de geração de colunas associado aos modelos OneFlow e MultiFlow para gerar todas as colunas necessárias à obtenção da solução ótima do problema relaxado (PL). No entanto, a solução obtida pode não ser inteira e, consequentemente, pode não representar uma solução

para o problema k-cluster. Uma alternativa para obtenção de soluções inteiras é a aplicação de um algoritmo de *branch-and-bound* baseado em relaxações lineares, onde soluções fracionárias são eliminadas através de sucessivas separações do espaço de soluções viáveis.

Os métodos de branch-and-bound são definidos por uma estratégia de branch, i.e., divisão do espaço de busca através da inserção de restrições e uma estratégia de obtenção de bounds, ou seja, limites superiores e inferiores para o valor da função objetivo do problema inteiro. Ao inserirmos uma restrição de branch, procuramos repartir o espaço viável em dois subespaços de forma que a solução fracionária corrente seja excluída. Essa separação do espaço pode ser feita de muitas formas. Por exemplo, se uma variável x_i possui valor fracionário, podemos repartir o espaço viável acrescentando alternadamente as restrições $x_i = 1$ e $x_i = 0$. Desta forma, obtemos dois subespaços: o espaço 0-*branch*, onde $x_i = 1$, e o espaço 1-*branch*, onde $x_i = 0$. Com esta estratégia teremos uma árvore binária, onde cada bifurcação representa a partição do espaço para uma dada variável. E, mais ainda, cada nó dessa árvore binária define um problema relaxado que fornecerá uma cota inferior (lower bound) para o seu o seu sucessor na a solução inteira que desejamos determinar. Podemos comparar duas estratégias de branch-and-bound a partir do número de partições (branch) necessárias para a obtenção da solução inteira ótima ou ainda compará-las a partir dos limites (bounds) gerados em cada nó.

Vale ressaltar que não existe um método definitivo de branch-and-bound. Algoritmos de branch-and-bound podem apresentar desempenho alternados em problemas diferentes, ou até em instâncias diferentes de um mesmo problema, não há uma estratégia de branch-and-bound que domine todas as demais. No entanto, uma propriedade conveniente de uma estratégia de branch é a de balanceamento. A ideia do balanceamento é a de que cada nó da árvore de branch em um dado nível deve representar um problema com espaço viável de tamanho relativamente igual. Assim como na estratégia de busca binária, a divisão do espaço de busca em partições de mesmo tamanho geralmente reduz o número de iterações. Infelizmente, como verificado por Ryan e Foster, a estratégia básica de atribuir 0's e 1's às variáveis fracionárias não produz árvores de branch balanceadas [70]. Pela restrição de particionamento da eq. (4.10), a restrição $x_i = 1$ implica em $x_j = 0, \forall j \neq i$; enquanto que a restrição $x_i = 0$ leva a $\sum_{j:j\neq i} x_j = 1$. De um lado, o espaço viável fica muito reduzido; enquanto no outro, a redução é muito pequena. Como consequência, o 0-branch tem pouco efeito no valor da função objetivo do problema (PL).

Como alternativa à estratégia ingênua de branch utilizando as restrições $x_i = 0$ e $x_i = 1$, Ryan e Foster propuseram uma estratégia mais elaborada baseada na restrição de particionamento. Obviamente, em uma solução inteira, dois vértices distintos ou pertencem ao mesmo cluster ou pertencem a clusters distintos. Em

termos das componentes dos vetores a_t, em uma solução inteira x, se dois vértices i e j pertencerem a mesma e única componente $a_{t'}$, então $x_t = 0$ para todo $t \neq t'$ tal que $a_t^i = a_t^j = 1$. Analogamente, se dois vértices i e j pertencerem a componentes distintas, então $x_t = 0$, para todo t tal que $a_t^i = a_t^j = 1$. Ou seja, uma solução x só poderá ser inteira se para qualquer par i e j de vértices tivermos

$$\sum_{t:a_t^i=a_t^j=1} x_t = 1,$$

quando i e j pertencerem à mesma componente, ou

$$\sum_{t:a_t^i=a_t^j=1} x_t = 0,$$

quando i e j pertencerem a componentes distintas. Note que, como consequência, em toda solução fracionária existirá ao menos um par de vértices i e j tal que

$$0 < \sum_{t:a_t^i=a_t^j=1} x_t < 1.$$

Esta característica das soluções fracionárias percebida por Ryan e Foster sugere uma estratégia de branch onde identificamos na solução fracionária um par de vértices i e j tais que $0 < \sum_{t:a_t^i=a_t^j=1} x_t < 1$ e impomos alternadamente as restrições $\sum_{t:a_t^i=a_t^j=1} x_t = 0$ (0-branch) e $\sum_{t:a_t^i=a_t^j=1} x_t = 1$ (1-branch).

Ao aplicarmos o 0-branch, devemos eliminar todos os vetores a_t tais que $a_t^i = a_t^j = 1$ e impedir que novos vetores (colunas) com esta característica sejam gerados. Para isto, adicionamos a restrição $a_t^i + a_t^j \leq 1$ à formulação do subproblema pricing. Já para a aplicação do 1-branch, eliminamos todos os vetores $a_t^i \neq a_t^j$ e impomos ao pricing a restrição $a_t^i = a_t^j$. Além da geração de estuturas balanceadas, a estratégia de branch de Ryan e Foster ainda apresenta a vantagem da simplicidade, já que as restrições de branch são simples.

Capítulo 5

Resultados e Discussões

As ideias apresentadas foram todas implementadas em liguagem Java (versão `1.6.02_2`), com chamadas ao solver `CPLEX` para a resolução dos problemas mestres reduzidos e dos subproblemas *pricing*. Os experimentos numéricos foram realizados em um computador `Intel Core 2 Quad Q9550`, `2.83GHz` e `4GB` de memória RAM.

As instâncias (grafos) utilizadas nos experimentos foram geradas a partir dos arquivos `gr17.tsp`, `gr21.tsp` e `gr24.tsp` da base de dados `TSPLIB` (`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`). Na tabela 5.1, podemos ver o número de vértices, $|V|$, a quantidade de arestas, $|E|$, o maior peso, $\lceil w_{ij} \rceil$, o menor peso, $\lfloor w_{ij} \rfloor$, e o peso médio, $\bar{w}_{ij}$, das arestas de cada um dos grafos completos definidos nestes arquivos. O peso de cada uma das arestas dos grafos definidos nos arquivos `gr17.tsp`, `gr21.tsp` e `gr24.tsp` são dados por matrizes simétricas com diagonais nulas.

No primeiro experimento, consideramos todos os vértices e arestas. Neste caso, não é necessário utilizar um modelo de fluxo para a obtenção de subrafos conexos já que todos os subgrafos serão conexos. Bastando, portanto, utilizar o modelo linearizado *LPricing* dado pelas eqs. (4.19)-(4.24). No entanto, apenas para comparação, aplicamos o modelo linearizado LPricing e também os modelos OneFlow e MultiFlow. Na tabela 5.2, vemos o tempo em segundos, o número de branchs, o número de colunas geradas, o número de subproblemas pricing resolvidos, o valor ótimo do

	`gr17.tsp`	`gr21.tsp`	`gr24.tsp`
$\|V\|$	17	21	24
$\|E\|$	136	210	276
$\lceil w_{ij} \rceil$	745	865	389
$\lfloor w_{ij} \rfloor$	27	27	22
$\bar{w}_{ij}$	274,60	363,89	147.61

Tabela 5.1: Número de vértices e arestas, pesos máximo, mínimo e médio das arestas de cada um dos grafos definidos em `gr17.tsp`, `gr21.tsp` e `gr24.tsp`.

	LPricing			OneFlow			MultiFlow		
	gr17	gr21	gr24	gr17	gr21	gr24	gr17	gr21	gr24
tempo	5,34	27,54	56,49	11,32	43,50	109,06	49,07	382,89	1122,48
#branchs	6	4	0	6	4	0	6	4	0
#colunas	56	100	105	56	100	105	55	100	105
#pricings	44	82	80	44	82	80	43	82	80
zPL	396,5	1201,5	811	396,5	1201,5	811	396,5	1201,5	811
zPI	421	1240	811	421	1240	811	421	1240	811

Tabela 5.2: Resultados dos experimentos considerando todas as arestas.

problema relaxado, zPL e o valor ótimo do particionamento, zPI, para cada uma das instâncias para agrupamentos formados dez partições ($k = 10$). Como esperado, o valor ótimo dos problemas relaxados e inteiros são os mesmos para todos os modelos. Isto é natural já que os modelos de pricing tem implicação apenas na ordem em que as colunas são geradas, mas não nos valores ótimos do problema inteiro original ou da sua relaxação linear. O número de colunas geradas difere apenas no experimento envolvendo a instância `gr17` (Utilizando o modelo OneFlow foram geradas 56 colunas e, utilizando o modelo MultiFlow, 55). Esta diferença pode ser explicada pela existência de múltiplas soluções para os subproblemas. Havendo duas ou mais soluções (colunas) de igual custo reduzido, a escolha de diferentes modelos para o subproblema pricing pode dar origem a diferentes sequências de colunas. Vale destacar que o número de colunas geradas em todas as instâncias foi muito inferior ao total de subgrafos conexos. Por exemplo, a instância `gr17` define um grafo completo formado por 17 vértices, portanto existem $2^{17} - 1 = 131.071$ subgrafos grafos conexos. Os modelos obtiveram uma solução ótima para a instância `gr17` gerando menos de 60 colunas, o que representa apenas 4,5% do número total de colunas (subgrafos conexos) possíveis. Isto ilustra a eficácia da abordagem de geração de colunas.

Com respeito ao tempo, o modelo OneFlow permitiu a obtenção de soluções em tempo muito inferior ao demandado pelo modelo MultiFlow. A discrepância entre os tempos demandados por cada um dos modelos cresce à medida que aumentamos o número de vértices. Para a instância associada ao arquivo `gr17.tsp`, formada por 17 vértices, o tempo demandado pela implementação com pricing dado pelo modelo MultiFlow correspondeu a cerca de 4,33 vezes o tempo utilizado pela implementação com o modelo OneFlow. Para a instância `gr21.tsp`, a implementação com fluxo múltiplo demandou quase nove vezes o tempo demandado pela implementação com fluxo único e, na instância `gr21.tsp`, mais de dez vezes.

No segundo experimento, buscamos verificar a sensibilidade dos modelos com respeito à variação do número de arestas nas instâncias. Para isto, excluímos aleatoriamente cerca de 25% e depois cerca de 50% das arestas de cada uma das instâncias, dando, desta forma, origem às instâncias `gr17-75`, `gr17-50`, `gr21-75`, `gr21-50` e `gr24-75`, `gr24-50`. A tabela 5.3 exibe um resumo das características destas instâncias.

	$\|V\|$	$\|E\|$	$\lceil w_{ij} \rceil$	$\lfloor w_{ij} \rfloor$	$\bar{w}_{ij}$
gr24-50	24	134	367	22	145,13
gr21-50	21	113	715	29	366,39
gr17-50	17	59	578	35	274,93
gr24-75	24	208	349	25	152,08
gr21-75	21	161	865	27	356,26
gr17-75	17	113	745	27	273,65

Tabela 5.3: Número de vértices e arestas, pesos máximo, mínimo e médio das arestas dos grafos gr17-75, gr17-50, gr21-75, gr12-50, gr24-75e gr24-50.

	OneFlow			MultiFlow		
	gr17	gr21	gr24	gr17	gr21	gr24
tempo	9,621	88,864	280,884	42,089	478,269	1350,786
#branchs	0	2	4	0	2	4
#colunas	47	83	109	47	81	110
#pricings	29	63	88	29	61	89
zPL	352	1204,5	654,444	352	1204,5	654,444
zPI	352	1210	655	352	1210	655

Tabela 5.4: Resultados dos experimentos com as instâncias formadas por 75% das arestas e 10 grupos ($k = 10$).

As tabelas 5.4 e 5.5 exibem os resultados dos experimentos envolvendo as instâncias com número reduzido de arestas. A redução do número de arestas não implicou na redução do tempo necessário para obtenção de soluções ótimas em todas as instâncias. Como se pode ver pelo resultados obtidos nas instâncias gr21 e gr24, o comportamento foi diametralmente oposto. Este comportamento pode ser explicado pela aumento da dificuldade na determinação de novos subgrafos conexos à medida que excluímos arestas. A etapa mais custosa da abordagem branch-and-price proposta é a resolução dos subproblemas pricing, por isso ao excluírmos arestas saímos de um grafo completo, onde qualquer subconjunto está associado a um grafo conexo, e caminhamos na direção de grafos esparsos, onde temos um número muito menor de subgrafos conexos, encarecendo a obtenção de soluções váiveis. A tabela 5.6 mostra os tempos totais em segundos e os tempos gastos na resolução dos subproblemas pricing na execução do método branch-and-price para $k = 10$ nas instâncias $gr - 24$, $gr24 - 75$ e $gr24 - 50$. Como podemos ver o tempo é quase que exclusivamente (mais de 99%) gasto na resolução dos subproblemas pricing e isto explica o aumento no tempo total à medida que os subproblemas pricing tornam-se mais difíceis.

No último experimento, avaliamos o impacto da variação do número de grupos no desempenho de ambos os modelos. Neste experimento, variamos no número de grupos para $k = 3, 5, 10$ na instância gr-24 formada por 24 vértices e 276 arestas. Na tabela 5.7 vemos os resultados obtidos. Como os resultados evidenciam, a redução do número de grupos (k) torna o problema ainda mais difícil, elevando

	OneFlow			MultiFlow		
	gr17	gr21	gr24	gr17	gr21	gr24
tempo	25,922	159,904	434,578	47,501	405,445	2160,348
#branchs	2	6	2	2	6	2
#colunas	53	86	85	53	86	83
#pricings	37	70	62	37	70	60
zPL	488	1176,5	678	488	1176,5	678
zPI	492	1210	680	492	1210	680

Tabela 5.5: Resultados dos experimentos com as instâncias formadas por apenas 50% das arestas e 10 grupos ($k = 10$).

	OneFlow		MultiFlow	
	tempo total	tempo pricing	tempo total	tempo pricing
gr24	109,062	108,451 (99,44%)	1122,476	1121,998 (99,96%)
gr24-75	280,884	280,206 (99,76%)	1350,786	1350,176 (99,95%)
gr24-50	434,578	434,176 (99,91%)	2160,348	2159,946 (99,98%)

Tabela 5.6: Tempos totais e tempos utilizados para resolução dos subproblemas pricing em segundos.

o tempo de execução. Novamente, podemos explicar este comportamento pelo aumento de dificuldade dos subproblemas princing. Ao reduzirmos o número de grupos do particionamento, exigimos que subgrafos conexos de maior cardinalidade sejam formados. Esta exigência dificuldade a obtenção de soluções para os subproblemas princing e, consequentemente, elevam o tempo de processamento.

	OneFlow			MultiFlow		
	tempo	#colunas	zPI	tempo	#colunas	zPI
$k = 3$	818,451	254	7671	9299,437	254	7671
$k = 5$	229,297	186	3122	2424,404	186	3122
$k = 10$	109,062	105	811	1122,476	105	811

Tabela 5.7: Tempos totais, número de colunas geras e o valor ótimo do agrupamento para diferentes valores de k na instância gr24.

Capítulo 6

Conclusões

Nesta dissertação, propomos dois modelos exatos baseados na técnica de geração de colunas para o problema de agrupamento em grafos conhecido como *k-cluster* (ou *k-agrupamento*), onde o objetivo é particionar um dado grafo $G(V, E)$ em k componentes (subgrafos) conexas de tal forma que a soma das arestas no interior de cada componente seja mínima.

Nossos modelos inspiraram-se nos problemas de fluxo em redes para garantir a conexidade dos subgrafos que representam os agrupamentos. Propomos dois modelos lineares e inteiros, um denominado OneFlow e o outro a que chamamos MultiFlow. Como os próprios nomes sugerem, o primeiro baseia-se na construção de um único fluxo e o segundo, na construção de fluxos múltiplos.

A abordagem de geração de colunas aqui adotada se justifica pela característica combinatória e o elevado número de variáveis (e, consequentemente, colunas) associadas a cada instância do problema *k-cluster*. Neste contexto, a representação explícita do espaço viável é impraticável mesmo para instâncias envolvendo um baixo número de vértices. Os métodos de geração de colunas por outro lado permitem contornar a barreira da representação explícita, através da geração iterativa das colunas que catacterizam o espaço viável.

Os método de geração de colunas partem de um subconjunto inicial, muito menor que o conjunto de soluções e geram colunas somente quando necessárias. A busca por colunas com menor custo reduzido dá origem a um outro subproblema conhecido como *Pricing*. Na exposição aqui apresentada, a função de custo do subproblema relacionado *Pricing* não era linear e para contornar essa dificuldade aplicamos a técnica de linearização proposta por Fortet em [62].

Para a geração do conjunto inicial de colunas, aplicamos uma versão modificada do algoritmo aproximativo proposto por Saran e Vazirani para a resolução do problema *min k-cut* em [68]. Nesta estratégia, partimos de uma árvore de *Gomory-Hu* para o grafo de entrada G e aplicamos sucessiva e de forma gulosa até k cortes associados às arestas da árvore de Gomory-Hu. Ao final deste procedimento, obtemos

um particionamento de G e k componentes conexas não necessariamente ótimas.

Para a obtenção de soluções inteiras aplicamos o arcabouço das técnicas branch-and-bound. Para a etapa de branch, utilizamos a estratégia proposta por Ryan e Foster em [70]. Com esta escolha as árvores de ramificação geradas tendem a ser mais balanceadas do que aquelas geradas pela imposição sucessiva de integralidade a cada uma das variáveis.

Nossas estratégias branch-and-bound com geração de colunas para os modelos OneFlow e Multiflow foram implementadas em linguagem Java utilizando o solver *CPLEX*. Realizamos experimentos numéricos com instâncias obtidas na TSPLIB. Os resultados mostraram que, apesar dos dois modelos apresentarem soluções inteiras iguais, o modelo OneFlow gera soluções em tempo muito inferior ao demandado pelo modelo MultiFlow. Além disso, vimos que a discrepância entre os tempos demandados por cada um dos modelos cresce à medida que aumentamos o número de vértices. Outro ponto que merece destaque é que o tempo de execução dos modelos é quase que exclusivamente gasto na resolução do subproblema pricing e aumenta gradativamente à medida que os grafos testados torna-se mais esparsos. Os resultados mostraram ainda que o número k de grupos influi de maneira significativa no tempo de execução, já que ao reduzirmos o número de partições a serem geradas exigimos uma maior cardinalidade dos subgrafos conexos, dificultando a obtenção de soluções inteiras.

Na direção de pesquisas futuras, a disponibilidade de métodos exatos para o problema de agrupamento em grafos é uma contribuição importante, já que permite checar a validade/qualidade de heurísticas em instâncias menores. Além disso, mesmo em instâncias maiores, podemos aplicar conjuntamente heurísticas para geração de bons pontos iniciais (e, consequentemente, bounds) para então aplicar métodos exatos como os aqui propostos. O trabalho aqui desenvolvido dá primeiros passos na direção de desenvolvimento de métodos eficientes para o problema de agrupamento em grafos. Os passos seguintes podem ser dados em direções relativamente ortogonais mas complementares como, por exemplo, definição de cortes mais eficientes e definição de heurísticas para a geração de bons pontos iniciais e para a resolução dos subproblemas entre outras.

Referências Bibliográficas

[1] JAIN, A. K., DUBES, R. C. *Algorithms for clustering data*. Prentice Hall, 1988.

[2] JAIN, A. K., MURTY, M. N., FLYNN, P. J. "Data clustering: a review", *ACM computing surveys (CSUR)*, v. 31, n. 3, pp. 264–323, 1999.

[3] VAN DONGEN, S. M. *Graph clustering by flow simulation*. Tese de Doutorado, Universiteit Utrecht, 2000.

[4] GUPTA, A. "Fast and effective algorithms for graph partitioning and sparse-matrix ordering", *IBM Journal of Research and Development*, v. 41, n. 1.2, pp. 171–183, 1997.

[5] ELSNER, U., OTHERS. "Graph partitioning-a survey", 1997.

[6] ALPERT, C., KAHNG, A. "Recent directions in netlist partitioning: a survey", *INTEGRATION, the VLSI journal*, v. 19, n. 1-2, pp. 1–81, 1995.

[7] FRIEZE, A., JERRUM, M. "Improved approximation algorithms for maxk-cut and max bisection", *Algorithmica*, v. 18, n. 1, pp. 67–81, 1997.

[8] MAHAJAN, S., RAMESH, H. "Derandomizing semidefinite programming based approximation algorithms". In: *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pp. 162–169. IEEE, 1995.

[9] PAPADIMITRIOU, C., YANNAKAKIS, M. "Optimization, approximation, and complexity classes", *Journal of Computer and system sciences*, v. 43, n. 3, pp. 425–440, 1991.

[10] PAPADIMITRIOU, C., STEIGLITZ, K. *Combinatorial optimization: algorithms and complexity*. Dover Pubns, 1998.

[11] GAREY, M., JOHNSON, D., STOCKMEYER, L. "Some simplified NP-complete graph problems", *Theoretical computer science*, v. 1, n. 3, pp. 237–267, 1976.

[12] DANTZIG, G., WOLFE, P. "Decomposition principle for linear programs", *Operations Research*, v. 8, pp. 101–111, 1960.

[13] GILMORE, P., GOMORY, R. "A linear programming approach to the cutting stock problem." *Operations Research*, v. 9, pp. 849–859, 1961.

[14] GILMORE, P., GOMORY, R. "A linear programming approach to the cutting stock problem: Part II", *Operations Research*, v. 11, pp. 863–888, 1963.

[15] DESROCHERS, M., SOUMIS, F. "A column generation approach to the urban transit crew scheduling problem", *Transportation Science*, v. 23, n. 1, pp. 1–13, 1989.

[16] DESROCHERS, M., DESROSIERS, J., SOLOMON, M. "A new optimization algorithm for the vehicle routing problem with time windows", *Operations research*, pp. 342–354, 1992.

[17] DAY, P. R., RYAN, M., D. "Flight attendant rostering for short-haul airline operations." *Operations Research*, v. 45, pp. 649–661, 1997.

[18] YUNES, T., MOURA, A., DE SOUZA, C. *Hybrid column generation approaches for solving real world crew management problems.* Relatório técnico, UNICAMP, 2000.

[19] YUNES, T., MOURA, A., DE SOUZA, C. "Solving very large crew scheduling problems to optimality." *Proceedings of the Symposium on Applied Computing.*, v. XV, pp. 446–451, 2000.

[20] MENESES, C., DE SOUZA, C. *Exact solutions of rectangular partitions via integer programming.* Relatório técnico, UNICAMP, 1998.

[21] MURITIBA, A., IORI, M., MALAGUTI, E., et al. "Algorithms for the bin packing problem with conflicts", *Informs Journal on computing*, v. 10, 2009.

[22] LORENA, L. A. N., SENNE, E. L. F. "A Column Generation Approach to Capacitated P-median Problems", *Computers and Operations Research*, v. 31, pp. 863–876, 2004.

[23] MACULAN, N., FAMPA, M. *Otimização linear.* EdUnB, 2006.

[24] MOREIRA, F. R. "Programação linear aplicada a problemas da área de saúde", *Einstein*, , n. 11, pp. 107–111, 2003.

[25] MANGASARIAN, O., SETIONO, R., WOLBERG, W. H. "Pattern Recognition Via Linear Programming: Theory And Application To Medical Diagnosis". 1990.

[26] NEUMANN, J. V. "A Model of General Economic Equilibrium", *The Review of Economic Studies*, v. 13, n. 1, pp. 1–9, 1945. ISSN: 00346527. doi: 10.2307/2296111. Disponível em: <http://dx.doi.org/10.2307/2296111>.

[27] MORGENSTERN, O., VON NEUMANN, J. *Theory of Games and Economic Behavior*. Princeton University Press, maio 1944. ISBN: 0691003629. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0691003629>.

[28] LEONTIEF, W. "Quantitative input-output relations in the economic system", *The Review of Economics and Statistics*, v. 18, pp. 105–125, 1936.

[29] KANTOROVICH, L. V. "Mathematical Methods of Organizing and Planning Production", *Management Science*, v. 6, n. 4, pp. 366–422, 1960. doi: 10.2307/2627082. Disponível em: <http://dx.doi.org/10.2307/2627082>.

[30] FANG, S.-C., PUTHENPURA, S. *Linear Optimization and Extensions: Theory and Algorithms*. Prentice Hall, New Jersey, 1993.

[31] HITCHCOCK, F. L. "The distribution of a product from several sources to numerous localities", *Journal of Mathematical Psychology*, v. 20, pp. 224–230, 1941.

[32] STIGLER, G. J. "The Cost of Subsistence", *Journal of Farm Economics*, v. 27, pp. 303–314, 1945.

[33] DANTZIG, G. *Linear Programming and Extensions*. Princeton University Press, ago. 1963. ISBN: 0691059136. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0691059136>.

[34] Cottle, R. W. (Ed.). *The basic George B. Dantzig*. Stanford University Press, Stanford, California, 2003.

[35] DORFMAN, R. *Application of linear programming to the theory of the firm including an analysis of monopolistic firms by non-linear programming*. Berkeley, University of California Press, 1951.

[36] CHARNES, A., C. E. LEMKE, A. *Modified Simplex Method for Control of Round-off Error in Linear Programming*. Relatório técnico, Carnegie Institute of Technology, Pittsburgh, PA, 1952.

[37] KHACHIAN, L. "A polynomial algorithm for linear programming", *Doklady Academiia Nauk SSSR*, v. 20, pp. 191–194, 1979.

[38] KARMARKAR, N. "A NEW POLYNOMIAL-TIME ALGORITHM FOR LINEAR PROGRAMMING", *COMBINATORICA*, v. 4, pp. 373–395, 1984.

[39] KLEE, V., MINTY, G. J. "How good is the simplex algorithm?" In: Shisha, O. (Ed.), *Inequalities*, v. III, Academic Press, pp. 159–175, New York, 1972.

[40] JANSEN, B. *Interior Point Techniques in Optimization Complementarity, Sensitivity and Algorithms*. Kluwer Academic Publishers, 1997.

[41] VANDERBEI, R. J. *Linear Programming, Foundations and Extensions*. Kluwer Academic Publishers, 1998.

[42] WRIGHT, S. J. *Primal-Dual Interior-Point Methods*. SIAM, 1997.

[43] ILLES, T., TERLAKY, T. "Pivot versus interior point methods: Pros and cons", *European Journal of Operational Research*, v. 140, n. 2, pp. 170–190, July 2002. Disponível em: <http://ideas.repec.org/a/eee/ejores/v140y2002i2p170-190.html>.

[44] BUENO, E. F. *Geração de Colunas em Problemas de Otimização Combinatória*. Tese de Mestrado, COPPE/UFRJ , Engenharia de Sistemas e Computação, 2005.

[45] FORD, L. R., FULKERSON, D. R. "A Suggested Computation for Maximal Multi-Commodity Network Flows", *MANAGEMENT SCIENCE*, v. 5, n. 1, pp. 97–101, out. 1958. doi: 10.1287/mnsc.5.1.97. Disponível em: <http://dx.doi.org/10.1287/mnsc.5.1.97>.

[46] DESROSIERS, J., SOUMIS, F., DESROCHERS, M. "Routing with time windows by column generation", *Networks*, v. 14, n. 4, pp. 545–565, 1984.

[47] BARNHART, C., JOHNSON, E. L., NEMHAUSER, G. L., et al. "Branch-and-Price: Column Generation for Solving Huge Integer Programs", *Operations Research*, v. 46, n. 3, pp. 316–329, 1998. ISSN: 0030364X. doi: 10.2307/222825. Disponível em: <http://dx.doi.org/10.2307/222825>.

[48] MINKOWSKI, H. *Gesammelte Abhanlungen*. Teubener and Leipzig, 1911.

[49] WOLSEY, L. A. *Integer Programming.* John Wiley & Sons, 1998.

[50] SCHRIJVER, A. *Theory of linear and integer programming.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999. ISBN: 978-0-471-98232-6.

[51] MEHROTRA, A., TRICK, M. A. "A Column Generation Approach For Graph Coloring", *INFORMS Journal on Computing*, v. 8, pp. 344–354, 1995.

[52] SAVELSBERGH, M. "A branch-and-price algorithm for the generalized assignment problem", *Operations Research*, pp. 831–841, 1997.

[53] VANCE, P. H. "Branch-and-Price Algorithms for the One-Dimensional Cutting Stock Problem", *Computational Optimization and Applications*, v. 9, n. 3, pp. 211–228, mar. 1998. ISSN: 09266003. doi: 10.1023/A:1018346107246. Disponível em: <http://dx.doi.org/10.1023/A:1018346107246>.

[54] VALÉRIO DE CARVALHO, J. M. "Exact solution of bin-packing problems using column generation and branch-and-bound", *Annals of Operations Research*, v. 86, n. 0, pp. 629–659, jan. 1999. ISSN: 02545330. doi: 10.1023/A:1018952112615. Disponível em: <http://dx.doi.org/10.1023/A:1018952112615>.

[55] KANG, S., MALIK, K., THOMAS, L. "Lotsizing and Scheduling on Parallel Machines with Sequence-Dependent Setup Costs", *Management Science*, v. 45, pp. 273–289, 1999.

[56] LAND, A. H., DOIG, A. G. "An Automatic Method for Solving Discrete Programming Problems". In: Jünger, M., Liebling, T. M., Naddef, D., et al. (Eds.), *50 Years of Integer Programming 1958-2008*, Springer Berlin Heidelberg, cap. 5, pp. 105–132, Berlin, Heidelberg, 2010. ISBN: 978-3-540-68274-5. doi: 10.1007/978-3-540-68279-0_5. Disponível em: <http://dx.doi.org/10.1007/978-3-540-68279-0_5>.

[57] LITTLE, J. D. C., MURTY, K. G., SWEENEY, D. W., et al. "An Algorithm for the Traveling Salesman Problem", *Operations Research*, v. 11, n. 6, 1963. ISSN: 0030364X. doi: 10.2307/167836. Disponível em: <http://dx.doi.org/10.2307/167836>.

[58] MURTY, K. G. *Operations Research: Deterministic Optimization Models.* Prentice-Hall, Inc., 1995.

[59] LUENBERGER, D., YE, Y. *Linear and nonlinear programming*, v. 116. Springer Verlag, 2008.

[60] DESAULNIERS, G., DESROSIERS, J., SOLOMON, M. *Column generation*, v. 5. Springer Verlag, 2005.

[61] FORREST, J. J., GOLDFARB, D. "Steepest-edge simplex algorithms for linear programming", *Mathematical Programing*, v. 57, pp. 341–374, 1992.

[62] FORTET, R. "Applications de l'algebre de boole en recherche opérationelle", *Revue Française de Recherche Opérationelle*, v. 4, pp. 17–26, 1960.

[63] GARG, N., VAZIRANI, V. V., YANNAKAKIS, M. "Multiway Cuts in Directed and Node Weighted Graphs". In: *ICALP*, pp. 487–498, 1994.

[64] FORD, L. R., FULKERSON, D. R. *Flows in Networks*. Relatório técnico, Princeton University Press, 1962.

[65] MONTENEGRO, A., RIBEIRO, C., ARAGÃO, M. *Um algoritmo randomizado para o problema de determinação ao do corte s-t mínimo em grafos direcionados.* Relatório técnico, Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio, 2001.

[66] GOMORY, R. E., HU, T. C. "Multi-Terminal Network Flows", *Journal of the Society for Industrial and Applied Mathematics*, v. 9, pp. 551–570, 1961.

[67] VAZIRANI, V. *Approximation algorithms*. Springer Verlag, 2001.

[68] SARAN, H., VAZIRANI, V. "Finding k Cuts within Twice the Optimal", *SIAM Journal on Computing*, v. 24, pp. 101, 1995.

[69] KRUSKAL, J. "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical society*, v. 7, n. 1, pp. 48–50, 1956.

[70] RYAN, D., FOSTER, B. "An integer programming approach to scheduling", *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pp. 269–280, 1981.