6
Algoritmos Exatos

6.1
Planos de Cortes

Nesta seção apresentaremos o método de *Planos-de-Corte* implementado para este trabalho. A idéia de se implementar este procedimento foi para a sua posterior utilização no método *Branch-and-Cut*. O método *Branch-and-Cut* combina as idéias do método de Planos-de-Corte com o método *Branch-and-Bound* para resolver ou obter bons limitantes inferiores e superiores para problemas difíceis de otimização combinatória. O método de Planos-de-Corte começa com uma solução do problema PL relaxado e, depois, de uma maneira sistemática, enquanto a solução obtida não é inteira, procura-se eliminá-la acrescentando-se novas inequações (chamadas *Planos-de-Corte*) ao sistema linear corrente. Esse processo nos permite encontrar um vértice inteiro que é uma solução do problema PLI original. Assim, a idéia do método é obter aproximações cada vez melhores do fecho inteiro P_I do poliedro P sobre o qual queremos otimizar, através da introdução de tais inequações. Abaixo é apresentando o pseudocódigo de um algoritmo Planos-de-Corte.

Procedimento *Planos-de-Corte*

1. Resolva a relaxação linear PL correspondente ao problema P, que é gerada eliminando a condição de integralidade da solução x. Se PL não tem solução ótima, então P é inviável ou ilimitado. Caso PL seja limitado, tome x^* uma solução ótima de PL;

2. Se x^* for inteiro, então pare e retorne como solução x^*. Caso contrário, encontre uma inequação que caracterize um plano de corte;

3. Junte essa nova inequação ao sistema de equações usando uma variável de folga e volte ao passo 1.

Tabela 6.1: Procedimento *Planos-de-Corte* - FONTE: adaptado de (Ferreira e Wakabayashi, 1996)

O método *Branch-and-Cut* utiliza para cada subproblema (nó da árvore de decisão) o método dos Planos-de-Corte com o objetivo de gerar bons limitantes inferiores. Contudo, conforme discutido a seguir, este procedimento não foi implementado devido ao fato de não encontrarmos inequações de ciclo ímpar violadas utilizando o procedimento de separação descrito mais adiante.

Na Seção 6.1.1 descrevemos as *inequações de ciclo ímpar* tratadas em (Baiou e Barahona, 2005) e (Baiou e Barahona, 2006). Na Seção 6.1.2 descrevemos o procedimento para separação desta classe de inequações. Este procedimento é definido em (Baiou e Barahona, 2005).

6.1.1
Definição das Inequações de Ciclo Ímpar

As inequações de ciclo ímpar são definidas da seguinte forma: seja $G = (V, A)$ um grafo direcionado e seja C um ciclo direcionado contido neste grafo definido como

$$C = v_1, (v_1, v_2), v_2, (v_2, v_3), ..., v_{k-1}, (v_{k-1}, v_k), v_k, (v_k, v_1), v_1 \quad (6\text{-}1)$$

Seja $A(C)$ o grupo de arcos em C. Se $|A(C)| = k$ for ímpar então C é ímpar. Seja y_j a variável associada ao vértice $j \in V$ e seja $x(a)$ a variável associada ao arco $a = (i, j) \in A$, conforme definição das variáveis da formulação primal descrita na Seção 2.2.1. As inequações do tipo

$$\sum_{a \in A(C)} x(a) \leq \frac{|A(C)| - 1}{2} \quad (6\text{-}2)$$

definidas para cada ciclo direcionado ímpar C são chamadas de inequações de ciclo ímpar. A adição destas inequações a relaxação linear do PMNC definida pela função objetivo (2-1) juntamente com as restrições (2-2)-(2-4), (2-6) e (2-7), fornece uma relaxação linear forte para o PMNC.

6.1.2
Procedimento para Separação das Inequações de Ciclo Ímpar

Descrição

Seja o *problema de separação*: dado o vetor x como sendo uma das componentes da solução ótima da relaxação linear da formulação primal definida por (2-1)-(2-4), (2-6) e (2-7) (a outra componente da solução ótima é o vetor y). Procurar inequações de ciclo ímpar violadas caso existam ou provar que tais inequações não existem. Visto que podem haver um número exponencial de inequações de ciclo ímpar, é necessário utilizar um algoritmo eficiente para resolver este problema de separação. Em (Baiou e Barahona, 2005) é descrito

este algoritmo e o mesmo foi implementado neste trabalho, conforme discutido a seguir.

O procedimento de separação trabalha com as inequações 6-2 descritas da seguinte forma

$$|A(C)| - 2 \sum_{a \in A(C)} x(a) \quad \geq \quad 1 \tag{6-3}$$

e trabalha, também, com um grafo auxiliar $G(V^{'}, A^{'})$ para procurar inequações deste tipo violadas. Este grafo é criado da seguinte forma: para cada vértice $u \in V$ são criados dois vértices $u^{'}$ e $u^{''}$ em $V^{'}$. Para cada arco $(u, v) \in A$ são criados dois arcos $(u^{'}, v^{''})$ e $(u^{''}, v^{'})$ em $A^{'}$, ambos com peso $1 - 2x(u, v)$. Após a criação de $G(V^{'}, A^{'})$, iniciamos a busca por inequações violadas da seguinte forma: para cada vértice $u \in V$ é encontrado o caminho mais curto P de $u^{'}$ para $u^{''}$ em $G(V^{'}, A^{'})$. Em seguida, cada par de vértices $(u^{'}, u^{''}) \in P$ é colapsado formando o vértice $u \in V$ e o novo caminho $P^{'}$ resultante desta operação sobre todos os pares de vértices em P forma um ciclo em $G(V, A)$. $P^{'}$ representa uma inequação de ciclo ímpar para o PMNC. Contudo, a classificação da inequação como violada ou não é feita da seguinte forma: se a soma das arestas do caminho P for menor do que 1 então a inequação referente a $P^{'}$ foi violada na solução ótima da relaxação linear. Caso contrário, esta inequação não foi violada.

Implementação

Antes de iniciarmos o procedimento de separação é necessário resolver na otimalidade a relaxação linear da formulação primal considerada. Para isto, foi utilizado o pacote comercial ILOG CPLEX 11.2 (ILOG, 2008). Após obtermos os vetores x e y que caracterizam a solução ótima encontrada, invocamos o procedimento de separação descrito na seção anterior. A primeira etapa deste procedimento consiste na construção do grafo auxiliar $G(V^{'}, A^{'})$. Esta construção consome um tempo $O(n^2)$, pois necessitamos percorrer toda a matriz de distâncias d_{ij} para definirmos as arestas que compõem $G(V^{'}, A^{'})$, juntamente com a definição de seus respectivos pesos. A etapa seguinte do método consiste no emprego do algoritmo de *Bellman Ford* descrito em Cormen et al. 2001 (Cormen et al., 2001) para encontrar os caminhos mais curtos a partir de cada vértice $u \in V$. Foi utilizado este algoritmo devido ao fato de poder haver pesos negativos nos arcos que compõem $G(V^{'}, A^{'})$. Cada execução do algoritmo de *Bellman Ford* gasta um tempo $O(|V^{'}||A^{'}|)$. Como são realizadas $|V|$ execuções, a complexidade final desta etapa é $O(|V||V^{'}||A^{'}|)$. Sendo $|V| = n$,

$|A| = n(n-1)$, $|V^{'}| = 2n$ e $|A^{'}| = 2n(n-1)$, a complexidade final desta etapa pode ser escrita como sendo $O(n^4)$. A etapa final do procedimento de separação consiste na montagem de cada ciclo $P^{'}$ obtido a partir da junção de cada par de vértices $(u^{'}, u^{''}) \in P$. A recuperação de cada caminho mínimo P é obtida a partir da análise da árvore de caminho mais curto desde $u^{'}$ até $u^{''}$ construída no algoritmo de *Bellman Ford*. O custo desta recuperação é $O(n^{'})$ por caminho recuperado, onde $n^{'}$ é o número de vértices em P (consultar (Cormen et al., 2001) para mais detalhes sobre este procedimento de recuperação). Como há $|V| = n$ recuperações a serem feitas, esta etapa possui complexidade $O(n^2)$. Contudo, a complexidade final do algoritmo de separação é $O(n^4)$.

O método de separação de inequações de ciclo ímpar utiliza o algoritmo de *Bellman Ford* descrito em (Cormen et al., 2001) para encontrar os caminhos mínimos a partir de cada vértice $v \in V$ no grafo $G(V, A)$. O algoritmo de *Bellman Ford* trata da resolução do problema de caminhos mínimos de única origem quando os pesos das arestas podem ser negativos. O tempo de execução deste algoritmo é $O(V^2A)$ sendo que, em um grafo denso, é $O(V^4)$.

As duas representações computacionais mais comuns para um grafo $G(V, A)$ são: listas de adjacências e matriz de adjacências. Optamos pela segunda forma de representação, pois tratamos com grafos densos (isto é, $|A|$ está próximo de $|V|^2$) em nossa aplicação. Para maiores detalhes sobre esta forma de representação, consultar (Cormen et al., 2001).

6.2 Branch-and-Ascent

Esta seção trata do detalhamento do algoritmo *Branch-and-Ascent*. Este algoritmo utiliza técnicas de enumeração implícita, assim como o *Branch-and-Bound*, e recebeu esta dominação em (Poggi de Aragão et al., 2001). A idéia do método *Branch-and-Bound* é enumerar todas as soluções viáveis de um problema em uma árvore de decisão. Porém, enumerar todas estas soluções pode ser inviável devido ao fato do número de nós na árvore poder crescer exponencialmente com o tamanho da instância de teste (depende do tamanho do problema). Assim, se dispormos de bons limitantes superiores e inferiores para o problema, a busca nesta árvore torna-se eficiente devido a eliminação de ramos que não precisam ser analisados. Para resolver os subproblemas que definem cada nó da árvore de decisão, são utilizadas as técnicas heurísticas descritas no capítulo 5. Optamos por utilizar estas técnicas devido ao baixo *Gap* encontrado pelos mesmos nas instâncias tratadas neste trabalho. Assim, dispondo-se de técnicas que encontram bons limites superiores e inferiores, a

abordagem mais recomendável para encontrar soluções ótimas e provar a sua otimalidade é através de técnicas de enumeração implícita, como a descrita a seguir.

6.2.1 Descrição

Tratamento do Nó

Para resolver o subproblema que define cada nó da árvore de decisão é executado o método *Fixação Ativa de Facilidades* cuja descrição está presente na Seção 5.6.2. A escolha deste método foi fortemente influenciada pela forma como são criados os subproblemas na árvore de decisão. Como veremos a seguir, estes subproblemas são gerados a partir da fixação de uma facilidade como aberta ou fechada. Assim, nada mais intuitivo do que tentar reduzir ao máximo o conjunto de facilidades candidatas no pré-processamento ou nos nós internos da árvore de enumeração. Foram feitos testes eliminando-se arcos na fixação ativa. Contudo, a eliminação de facilidades mostrou-se mais apropriado. Um passo mais além, que não foi testado neste trabalho, seria a utilização de ambas as fixações em cada nó da árvore. Esta abordagem, além de atender o quesito de diminuição do conjunto de candidatos a ramificação, aumenta o desempenho dos algoritmos duais devido ao fato de se eliminar variáveis y_j e x_{ij} da formulação ao qual estes métodos trabalham.

Branching

Para a geração dos dois subproblemas a partir de cada nó da árvore de decisão é feito o *branching* em alguma variável y_j não fixada em nenhum nó presente no caminho entre a raiz e o nó corrente da árvore de decisão. Esta variável é fixada em 0.0 para o primeiro subproblema e em 1.0 para o segundo subproblema. A estratégia para escolha da variável a ser fixada é feita seguindo a ordem de prioridade descrita abaixo.

1. É selecionada a primeira variável y_j encontrada que viola as condições de complementaridade de folga descritas em 5-2;

2. Caso nenhuma variável viole as restrições em 5-2 então verificamos as condições de complementaridade de folga descritas em 5-3. Para isto, seguimos a ordem das verificações descritas abaixo:

   (a) Selecionar a primeira facilidade não essencial encontrada durante a construção do conjunto $iMais$ e que não está fixada como aberta e nem como fechada (Condição 1);

(b) Se foram abertas menos do que p facilidades então selecionar a primeira facilidade aberta na solução primal que não está presente no conjunto $iMais$ e que não está fixada como aberta e nem como fechada (Condição 2);

(c) Se foram abertas mais do que p facilidades então selecionar a primeira facilidade fechada na solução primal que não está fixada como aberta e nem como fechada (Condição 3);

(d) Selecionar a primeira facilidade essencial encontrada durante a construção do conjunto $iMais$ e que não está fixada como aberta e nem como fechada (Condição 4);

(e) Selecionar a facilidade de menor custo reduzido que não esteja presente no conjunto $iEstrela$, ou seja, selecionar a facilidade com custo reduzido mínimo diferente de zero. Porém, a facilidade escolhida não deverá estar fixada como aberta e nem como fechada (Condição 5);

3. Caso nenhuma variável viole as restrições em 5-3 então seguimos a ordem das verificações das seguintes condições: Condição 1, Condição 4 e Condição 5.

Vale ressaltar que não verificamos as condições de complementaridade de folga descritas em 5-1 devido ao fato delas sempre serem satisfeitas para as soluções primais construídas pelo método construtivo *Primal-Dual* descrito na Seção 4.2.5.

Quando fixamos alguma variável y_j em 1.0 (facilidade fixada como aberta) ocorrem algumas alterações na formulação dual condensada descrita na Seção 2.2.3. Abaixo apresentamos a formulação englobando estas alterações.

$$\max \sum_{i \in U} \lambda_i + \gamma(p - |F^{Fix}|) \tag{6-4}$$

sujeito a

$$\sum_{i \in U} \max(0, \lambda_i - d_{ij}) \leq -\gamma, \forall j \in F \tag{6-5}$$

$$\lambda_i \leq \min_{j \in F^{Fix}} \{d_{ij}\}, \forall i \in U \tag{6-6}$$

$$\tag{6-7}$$

Nesta formulação $F^{Fix} \subseteq F$ é o conjunto de facilidades fixadas como aberta. A função objetivo (6-4) tenciona maximizar a soma das variáveis duais λ_i e o fator $\gamma(p - |F^{Fix}|)$. As restrições (6-5) são idênticas as restrições

descritas em (2-15). Por fim, as restrições (6-6) definem o limite superior para cada variável dual λ_i para manter cada facilidade $j \in F^{Fix}$ fixada como aberta. A inserção das restrições (6-6) é equivalente a inserção das restrições $\sum_{i \in U} \max(0, \lambda_i - d_{ij}) \leq 0, \forall j \in F^{Fix}$ na formulação descrita acima.

Quando modificamos a formulação dual em cada nó da árvore devido as fixações, a formulação primal derivada desta formulação dual também sofrerá alterações. Mais detalhadamente, cada restrição (6-6) inserida na formulação dual irá gerar uma variável primal k_i com coeficiente na função objetivo primal igual ao lado direito da restrição (6-6) inserida. Assim, a nova formulação primal será descrita como:

$$\min \sum_{i \in U} \sum_{j \in F} d_{ij} x_{ij} + \sum_{i \in U} \min_{j \in F^{Fix}} \{d_{ij}\} k_i \tag{6-8}$$

sujeito a

$$\begin{aligned}
\sum_{j \in F} x_{ij} + k_i &= 1, \forall i \in U && (6\text{-}9)\\
y_j - x_{ij} &\geq 0, \forall i \in U, \forall j \in F && (6\text{-}10)\\
\sum_{j \in F} y_j &= p && (6\text{-}11)\\
x_{ij} &\geq 0, \forall i \in U, \forall j \in F && (6\text{-}12)\\
y_j &\geq 0, \forall j \in F && (6\text{-}13)\\
k_i &\geq 0, \forall i \in U && (6\text{-}14)
\end{aligned}$$

A diferença desta formulação quando comparada com a formulação descrita em 2.2.1 é a inserção das variáveis k_i para todo $i \in U$, a substituição da função objetivo 2-1 pela função objetivo 6-8 e a substituição das restrições 2-2 pelas restrições 6-10. Contudo, como foi inserido um novo conjunto de variáveis, as restrições 6-14 também devem ser consideradas na nova formulação.

Como o método construtivo *Primal-Dual*, cuja descrição encontra-se na Seção 4.2.5, sempre designa um cliente a exatamente uma única facilidade aberta (este método sempre constrói soluções primais viáveis), uma única variável x_{ij} possuirá valor 1.0 em cada restrição 6-9 e, assim, todas as variáveis k_i possuirão valor 0.0 nas soluções primais geradas. Consequentemente, nenhuma adaptação é necessária para a construção de soluções primais a partir da nova formulação dual referente os subproblemas criados na árvore de decisão. Vale ressalta, contudo, que estas alterações criam um novo conjunto de restrições para a verificação das condições de complementaridade de folga. No entanto, como $k_i = 0.0$ para todo $i \in U$, estas condições nunca serão violadas não sendo

necessário verificá-las quando selecionamos uma variável para *branching*.

Quando fixamos a variável y_j em 0.0 (facilidade fixada como fechada), substituímos a constante $-\gamma$ para cada $j \in F^{Fix}$ nas restrições 2-15 por ∞. Esta substituição não acarreta nenhuma alteração na formulação dual gerada a partir desta substituição não sendo necessária nenhuma adaptação nos algoritmos duais empregados para a sua resolução.

Busca na árvore de decisão

É feito uma busca em profundidade na árvore de decisão. Iniciamos a busca no subproblema criado a partir da fixação da variável y_j em 0.0. Sempre que criamos os subproblemas, primeiro analisamos aquele onde a variável é fixada em 0.0 e, após o *backtrack*, analisamos o subproblema originado com a fixação da variável em 1.0.

Poda

Para finalizarmos a busca em um ramo da árvore de decisão verificamos as seguintes condições:

1. Se a solução primal encontrada pelo método *Primal-Dual* for inviável então podamos por inviabilidade;

2. Se o limite inferior encontrado for maior que o limite superior global então podamos devido ao fato de não encontramos soluções de melhora ramificando a partir do nó corrente;

3. Se o total de facilidades fixadas por custo reduzido mais o total de facilidades fixadas como fechada exceder o limite que permita abrir p facilidades então podamos devido ao fato das soluções derivadas do nó corrente serem inviáveis;

4. Se o total de facilidades fixadas como aberta for maior do que p então podamos devido ao fato das soluções derivadas do nó corrente serem inviáveis;

5. Se o limite superior global for igual ao limite inferior encontrado e a solução dual encontrada for inteira então podamos devido ao fato de termos encontrado um candidato a solução ótima para o problema.

6.2.2
Implementação

A Figura 6.1 descreve o pseudocódigo do método exato *Branch-and-Ascent* cuja complexidade será discutida nesta seção. Cada iteração do *loop* constituído pelas linhas 6 a 28 consiste na resolução de um nó da árvore de decisão e possui complexidade $O(\rho n^3)$ conforme analisado a seguir. Nas linhas 7 e 8 ocorrem, respectivamente, a recuperação do subproblema f_i a ser resolvido e a configuração de f_i com relação as variáveis fixadas para o mesmo (etapa de *branching*). Entre as linhas 9 e 14 são calculados o limitante inferior fO_π e o limitante superior fO_α para f_i. Esta etapa possui complexidade $O(\rho n^3)$ onde ρ é o total de iterações do método *Dual Adjustment*, pois são invocados os métodos *Dual Ascent*, *Dual Adjustment*, o método construtivo Primal-Dual e uma busca local, respectivamente, dentro desta etapa. Conforme já analisado, o *Dual Adjustment* é o método mais custoso dentre estes com complexidade $O(\rho n^3)$ por iteração definindo, assim, a complexidade de pior caso desta etapa. Nas linhas 15 e 16, respectivamente, poderão ser atualizas as melhores soluções primal α^* e dual π^* obtidas até então. Cada atualização possui complexidade $O(n)$. Entre as linhas 17 e 19, todas executas em tempo constante, estão definidas as condições para poda do nó. As linhas 20 a 27 dizem respeito a etapa de criação dos subproblemas $f_i^{'}$ e $f_i^{''}$ para posterior *branching*. Nesta etapa, a linha 21 é executada em tempo constante. Na linha 22 ocorre a seleção da variável para *branching* em tempo $O(n)$ e as linhas 23 a 26 são executas em tempo constante. Portanto, a complexidade final do método é $O(\rho n^3)$ por iteração.

Procedimento *Branch-and-Ascent*

1. **Entrada:** Solução dual vazia π^*, solução primal vazia α^*, lista de subproblemas ativos *pAtivos* e a *flag* δ informando a variação do método a ser utilizada (fixação ativa de arcos ou fixação ativa de facilidades)
2. **Saída:** Solução dual inteira ótima π^* e solução primal inteira ótima α^*
3. $fO_{\alpha^*} \leftarrow \infty$;
4. $fO_{\pi^*} \leftarrow -\infty$;
5. $pAtivos \leftarrow raiz$;
6. **enquanto** $pAtivos \neq \oslash$ **faça**
7. $f_i \leftarrow$ subproblema ativo em $pAtivos$;
8. Acerte as variáveis fixadas para f_i;
9. **se** $\delta = true$ **então**
10. $(\pi_{f_i}, \alpha_{f_i}) \leftarrow FixacaoAtivaArcos(\pi_{f_i}, \alpha_{f_i})$;
11. **fim-se**
12. **senão**
13. $(\pi_{f_i}, \alpha_{f_i}) \leftarrow FixacaoAtivaFacilidades(\pi_{f_i}, \alpha_{f_i})$;
14. **fim-senão**
15. **se** ($fO_{\alpha_{f_i}}$ for viável) $\wedge (fO_{\alpha_{f_i}} < fO_{\alpha^*})$ **então** $\alpha^* \leftarrow \alpha_{f_i}$ **fim-se**
16. **se** $(fO_{\pi_{f_i}} \leq fO_{\alpha^*}) \wedge (fO_{\pi_{f_i}} > fO_{\pi^*})$ **então** $\pi^* \leftarrow \pi_{f_i}$ **fim-se**
17. **se** $fO_{\alpha_{f_i}}$ for inviável **então** Retirar f_i de *pAtivos* **fim-se**
18. **senão se** $fO_{\pi_{f_i}} > fO_{\alpha^*}$ **então** Retirar f_i de *pAtivos* **fim-senão se**
19. **senão se** $fO_{\alpha^*} = fO_{\pi_{f_i}}$ **então** Retirar f_i de *pAtivos* **fim-senão se**
20. **senão**
21. Retirar f_i de *pAtivos*;
22. Selecionar uma variável y_j para *branching*;
23. **se** Foi selecionada uma variável y_j para *branching* **então**
24. Criar um subproblema $f_i^{'}$ com $y_j = 0.0$ e $pAtivos \leftarrow f_i^{'}$;
25. Criar um subproblema $f_i^{''}$ com $y_j = 1.0$ e $pAtivos \leftarrow f_i^{''}$;
26. **fim-se**
27. **fim-senão**
28. **fim-enquanto**
29. **retorne** (π^*,α^*);

Figura 6.1: *Algoritmo Branch-and-Ascent*

6.3 Resultados Computacionais

6.3.1 Soluções Obtidas

Planos de Cortes

O procedimento para separação das inequações de ciclo ímpar descrito na Seção 6.1.2 foi executado sobre as instâncias da classe *OR-Library*. Porém, foram encontradas somente inequações não violadas e inequações satisfeitas na igualdade. Não foi encontrada nenhuma inequação violada para estas instâncias. Uma validação da nossa implementação foi realizada através da inserção das inequações encontradas na formulação relaxada do PMNC e, em seguida, resolvendo esta nova formulação até a otimalidade utilizando o pacote comercial ILOG Cplex 2008 (ILOG, 2008). Como as inequações inseridas constituem inequações não violadas, a solução resultante desta nova formulação deve ser a mesma encontrada sem a inserção desta classe de inequações. Este fato foi confirmado para todas as instâncias da classe *OR-Library*. Contudo, a literatura não apresenta nenhum trabalho expondo inequações deste tipo separadas para o PMNC.

Branch-and-Ascent

A Tabela 6.2 apresenta os resultados obtidos com a aplicação da técnica *Branch-and-Ascent* sobre as instâncias da classe *OR-Library*. Nesta tabela, exibimos o nome e a dimensão da instância, nomenclatura esta já familiar ao leitor. Seguindo, expomos desvio primal dP_I, o desvio dual dD_I e o *Gap* Gap_I, ambos coletados no nó raiz da árvore. Mostramos o número de nós contidos na árvore de resolução do problema (*Nós*) juntamente com o tempo total ($T_{B\&A}$), em segundos, gasto para a finalização do método. Por último, exibimos o tempo (T_{CPLEX}), também em segundos, gasto pelo pacote comercial ILOG CPLEX 2008 (ILOG, 2008) para encontrar a solução ótima inteira para o problema. Em nenhum momento este trabalho teve como objetivo encontrar soluções inteiras mais rápido do que o CPLEX. Porém, como este pacote comercial consegue tratar as instâncias da classe *OR-Library*, julgamos relevante passar uma noção do tempo gasto pelo nosso procedimento quando comparado com este pacote. Vale enfatizar que o CPLEX não resolve nenhuma das instâncias da classe *TSP-Library* tratadas nesta dissertação devido a limitações de memória RAM da máquina utilizada nos experimentos.

Analisando o número de nós pesquisados pelo método nas diferentes instâncias, a instância *pmed7* foi aquela que abriu menos nós na árvore, sendo pesquisados 7 nós em 9,54 segundos. Em contrapartida, a instância que abriu mais nós foi *pmed31*, pesquisando 233 nós em 7032,38 segundos. Apesar de não ser o foco deste trabalho, é interessante observar o tempo gasto pelo nosso método quando comparado ao CPLEX. Para algumas instâncias, o *Branch-and-Ascent* encontrou o ótima mais rápido do que o CPLEX. Porém, para outras, o *Branch-and-Ascent* gastou um tempo bem superior aquele observado na execução do CPLEX. Com frequência, os melhores tempos estão ligados aos menores valores dos *Gaps* encontrados no nó raiz (Gap_I) da árvore. Instâncias com *Gaps* mais estreitos na raiz tendem a serem resolvidos mais rapidamente. Logicamente, a dimensão e a disposição dos pontos no plano cartesiano para um problema também interferem bastante no tempo gasto, sendo que dimensões maiores gastam tempos maiores.

A Figura 6.2 exibe os valores das soluções primais e duais encontradas no *Branch-and-Ascent* sempre que as melhores soluções encontradas até então foram atualizadas. Esta figura ilustra a convergência destas soluções, no decorrer do tempo, ao ótimo. A solução primal chega rapidamente ao ótimo enquanto a solução dual demora um pouco mais para chegar a este valor. Para quase todas as instâncias testadas, a solução primal converge rapidamente enquanto a convergência da solução dual é mais demorada. Esta lentidão na convergência da solução dual reflete no número de nós pesquisados pelo *Branch-and-Ascent*. Quanto mais lento for a convergência, maior será o número de nós. Outro fator que influência diretamente na quantidade de nós visitados pelo método é o critério de escolha da variável para *branching*. Um critério mais robusto implica em menos enumerações para encontrar o ótima de uma instância. Um último fator que tende a diminuir bastante a busca pelo ótimo é obter limites de excelente qualidade na raiz da árvore de enumeração, conforme analisado anteriormente.

A Figura 6.3 exibe a árvore de resolução para a instância *pmed21* da classe *OR-Library*. Nesta instância, foram examinados 11 nós para encontrar a solução ótima e este ótimo foi obtido no último nó pesquisado na árvore (nó 11). Na parte superior de cada nó colocamos os limites superior e inferior encontrado no respectivo nó e, abaixo, indicamos qual a variável que foi escolhida para a criação dos subproblemas. Neste exemplo, no nó raiz já encontramos a solução ótima para o problema. Contudo, esta otimalidade foi aprovada apenas no nó 11, quando foi encontrado um limite inferior com o mesmo valor do limite superior ótimo. Neste momento, foi finalizado a busca na árvore não por termos encontrado o ótimo, mas sim devido ao fato de não haver mais subproblemas

Instância		Branch-and-Ascent					CPLEX
Nome	**(n,p)**	dP_I	dD_I	Gap_I	**Nós**	$T_{B\&A}$	T_{CPLEX}
pmed1	(100,5)	0,00	0,78	0,78	47	2,39	8,69
pmed2	(100,10)	0,00	0,76	0,76	31	2,82	8,53
pmed3	(100,10)	0,00	0,78	0,78	15	1,74	7,55
pmed4	(100,20)	0,00	0,26	0,26	21	1,93	6,58
pmed5	(100,33)	0,00	0,82	0,82	21	2,22	6,74
pmed6	(200,5)	0,00	1,72	1,72	33	43,86	100,15
pmed7	(200,10)	0,00	0,63	0,63	7	9,54	29,58
pmed8	(200,20)	0,00	0,25	0,25	79	27,10	36,54
pmed9	(200,40)	0,00	0,26	0,26	15	12,01	44,54
pmed10	(200,67)	0,00	0,24	0,24	171	91,84	45,08
pmed11	(300,5)	0,00	1,80	1,80	101	248,75	170,71
pmed12	(300,10)	0,00	0,90	0,90	27	81,71	171,69
pmed13	(300,30)	0,00	0,21	0,21	67	113,72	112,82
pmed14	(300,60)	0,07	0,34	0,41	45	118,61	156,69
pmed15	(300,100)	0,00	0,23	0,23	19	50,38	109,78
pmed16	(400,5)	0,00	2,28	2,28	79	578,49	1109,44
pmed17	(400,10)	0,00	1,83	1,83	51	539,18	613,30
pmed18	(400,40)	0,08	1,18	1,26	45	379,80	410,36
pmed19	(400,80)	0,04	0,99	1,03	89	551,57	230,79
pmed20	(400,133)	0,06	0,56	0,62	83	697,70	228,74
pmed21	(500,5)	0,00	0,58	0,58	11	124,50	370,41
pmed22	(500,10)	0,00	1,62	1,62	25	727,59	1098,10
pmed23	(500,50)	0,17	0,94	1,11	67	787,99	367,16
pmed24	(500,100)	0,24	0,92	1,16	57	866,41	376,46
pmed25	(600,167)	0,22	1,05	1,27	31	443,55	606,01
pmed26	(600,5)	0,00	1,92	1,92	45	1187,30	2240,91
pmed27	(600,10)	0,00	1,39	1,39	45	1798,55	1783,25
pmed28	(600,60)	0,13	1,37	1,51	65	2115,81	516,95
pmed29	(600,120)	0,00	0,90	0,90	67	1562,22	510,35
pmed30	(600,200)	0,75	1,12	1,88	103	2638,38	510,82
pmed31	(700,5)	0,00	2,74	2,74	233	7032,38	3242,93
pmed32	(700,10)	0,04	1,97	2,02	45	5920,93	2700,70
pmed33	(700,70)	0,13	1,21	1,33	51	2548,20	731,55
pmed34	(700,140)	0,10	0,84	0,94	37	1411,48	709,35
pmed35	(800,5)	0,00	2,61	2,61	79	6900,11	5091,11
pmed36	(800,10)	0,00	2,63	2,63	93	10187,16	1151.42
pmed37	(800,80)	0,16	1,38	1,54	67	4584,01	2325,98
pmed38	(900,5)	0,00	2,61	2,61	117	15691,23	3295.97
pmed39	(900,10)	0,00	2,04	2,04	153	12836,81	575.20
pmed40	(900,90)	0,29	1,44	1,74	153	14464,13	390,95

Tabela 6.2: Resultados - *Branch-and-Ascent e CPLEX* aplicado sobre as instâncias da classe *OR-Library*

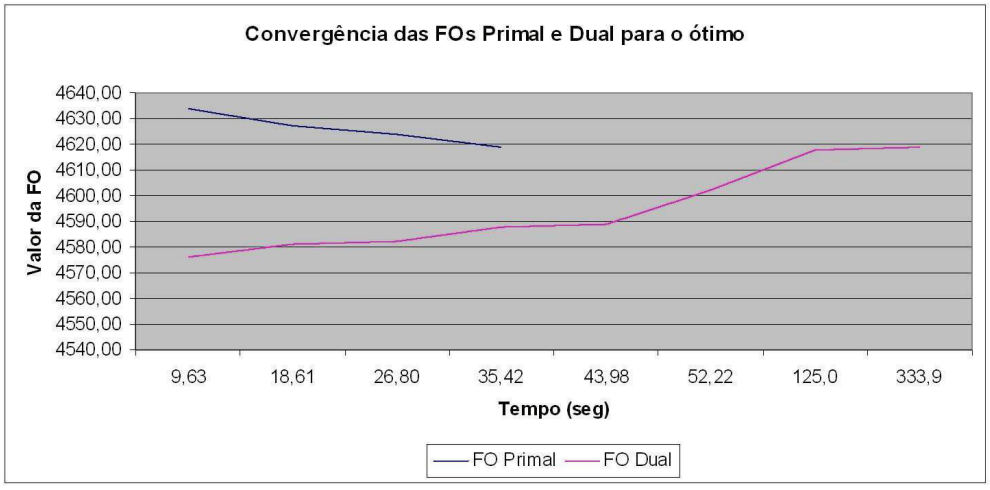

Figura 6.2: Convergência das soluções primal e dual para o ótimo na instância *pmed23* (n=500,p=50) da classe *OR-Library*

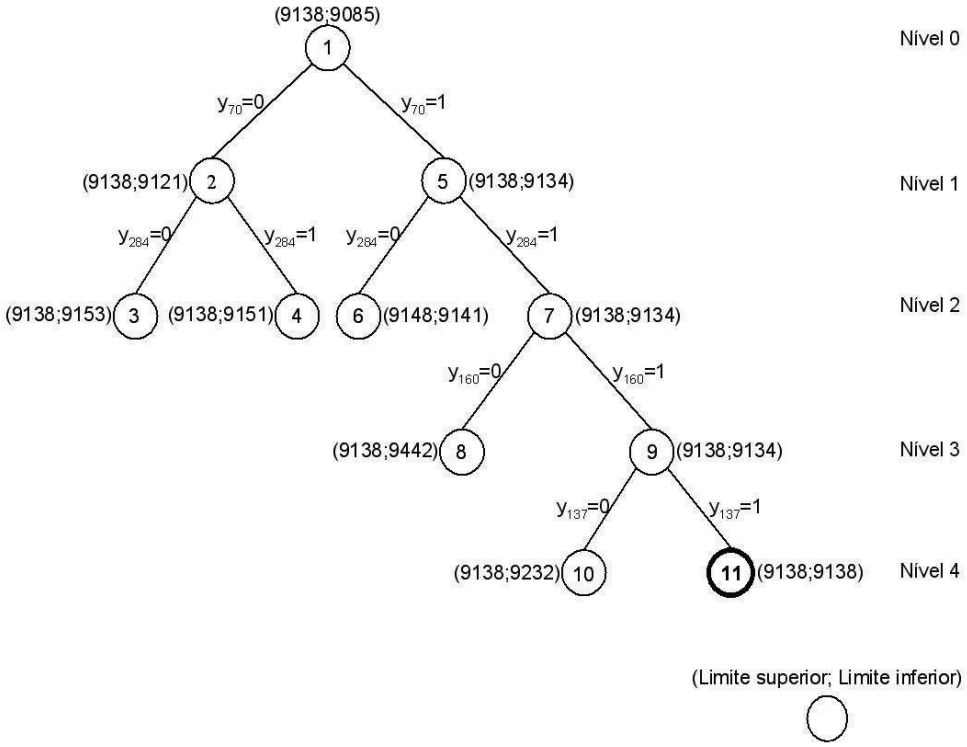

Figura 6.3: Árvore de enumeração do *Branch-and-Ascent* para a instância *pmed21* (n=500,p=5) da classe *OR-Library*

em aberto para serem analisados.

6.3.2
Complexidade

A Tabela 6.3 apresenta a complexidade de pior caso dos algoritmos exatos implementados para este trabalho.

Algoritmo	**Complexidade de Pior Caso**
B&A Fixaçao de Arcos	$O(\rho n^3)$ por nó
B&A Fixação de Facilidades	$O(\rho n^3)$ por nó
Algoritmo de Separação	$O(n^4)$

Tabela 6.3: Complexidade de pior caso dos algoritmos exatos

6.4
Conclusão

Não foi obtido sucesso com o procedimento de separação de inequações de ciclo ímpar, descrito na Seção 6.1.2, para as instâncias tratadas nesta dissertação. A não obtenção de sucesso diz respeito a não termos encontrado inequações violadas para a classe de inequações tratada por este procedimento. O objetivo deste trabalho era desenvolver o método *Branch-and-Cut*, cuja descrição está presente na Seção 6.1, utilizando internamente este procedimento de separação. Contudo, devido ao insucesso obtido com o procedimento de separação implementado, abandonamos esta metodologia e seguimos com a implementação do método *Branch-and-Ascent*.

Os resultados obtidos com o *Branch-and-Ascent* foram satisfatórios para as instâncias testadas. Observamos que os melhores tempos obtidos por este método estão ligados aos menores valores dos *Gaps* encontrados no nó raiz da árvore de resolução. Isto justifica a implementação desde método utilizando as heurísticas duais apresentadas na Seção 5. Como observado, estas heurísticas obtém *Gaps* bem estreitos, além de reduzir bastante algumas instâncias através da *Fixação Ativa*, cuja definição está presente na Seção 5.6. A *Fixação Ativa* acelerou o método *Branch-and-Ascent*, pois foram fixadas várias facilidades para determinadas instâncias e isto otimizou o critério de ramificação implementado, além de obtermos *Gaps* ainda mais estreitos após as fixações proporcionadas por esta técnica em várias das instâncias de teste. Esta técnica se mostrou promissora para tratamento de instâncias de grande porte para o PMNC.