Um estudo sobre heurísticas mergulho aplicada em Programação Inteira

Raniere Gaia Costa da Silva
Orientando

Margarida P. Mello
Orientadora

12 de dezembro de 2011

Resumo

O método utilizado pela maioria dos *solvers*, comerciais ou não, para problemas de programação linear interia são variantes do Método *Branch-and-Bound*. No caso de conhecermos uma solução viável para o problema de programação linear inteira, o Método *Branch-and-Bound* pode tirar proveito desta solução. Neste trabalho estudamos algumas heurísticas que buscam encontrar uma solução viável para um problema de programação linear inteira e implementamos-as utilizando a linguagem C e a API do GLPK.

A implementação foi testada com os problemas da biblioteca MIPLIB2. Como mais de 70% dos problemas foram resolvidos em menos de 10 segundos não conseguimos precisar o ganho de performance ao utilizar as heurísticas para encontrar uma solução viável.

Sumário

1 Programação Inteira

A programação linear inteira mista lida com problemas de maximizar ou minimizar uma função linear de várias variáveis sujeita a restrições de igualdade e/ou desigualdades lineares e de integralidade para algumas das variáveis, ver, por exemplo, [NW88, p. 3].

Sem perda de generalidade, ver maiores detalhes em [NW88, BJS04], podemos considerar que o problema de programação linear inteira mista é da forma

$$\begin{aligned} \min\ & c^T x + h^T y \\ \text{s.a}\ & Ax + Gy \leq b \\ & x \in \mathbb{Z}^n_+, y \in \mathbb{R}^p_+, \end{aligned}$$

onde (x, y) são as variáveis do problema, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $h \in \mathbb{R}^p$, $G \in \mathbb{R}^{m\times p}$ e $b \in \mathbb{R}^m$.

Costuma-se denominar $c^T x + h^T y$ de função objetivo, $(x, y) \in \{(x, y) : Ax + Gy \leq b, x \in \mathbb{Z}^n_+, y \in \mathbb{R}^p_+\}$ de solução viável e (x^*, y^*) que minimiza a função objetico no conjunto viável de solução ótima.

Para simplificar a abordagem dos problemas de programação linear inteira mista será considerado apenas o caso especial

$$\begin{aligned} \min\ & c^T x \\ \text{s.a}\ & Ax \leq b \\ & x \in \mathbb{Z}^n_+, \end{aligned} \tag{PLI}$$

onde x é a variável do problema, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m\times n}$, $b \in \mathbb{Z}^m$ e $\{x : Ax \leq b, x \in \mathbb{Z}^n_+\}$ é limitado, que é conhecido como problema de programação inteira.

Como exemplos de (PLI) temos

$$\begin{aligned} \min\ & -10x_1 - 15x_2 \\ \text{s.a}\ & 8x_1 + 4x_2 \leq 40 \\ & 15x_1 + 30x_2 \leq 200 \\ & x_1, x_2 \in \mathbb{Z}_+, \end{aligned} \tag{Exemplo 01}$$

retirado de [Tay09], e

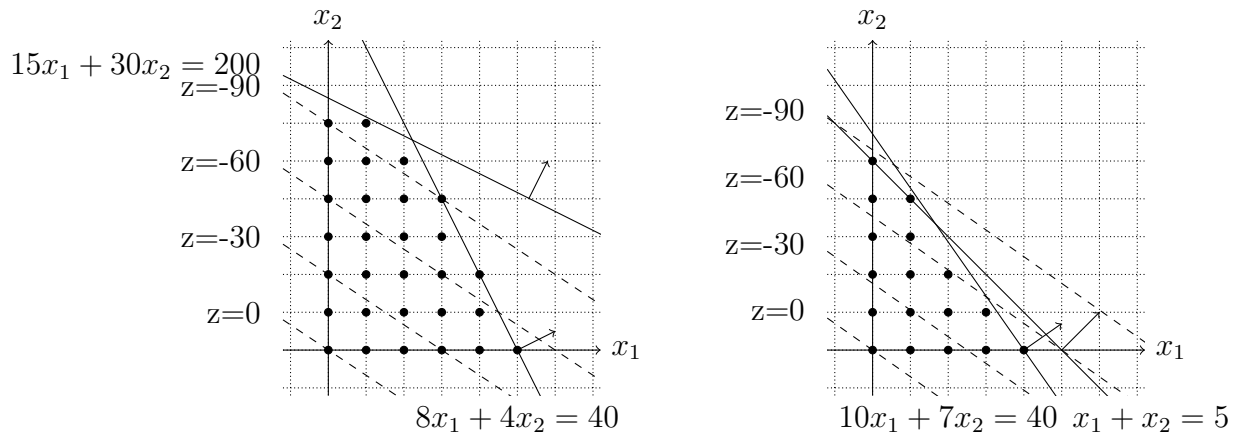
$$\begin{aligned} \min\ & -17x_1 - 12x_2 \\ \text{s.a}\ & 10x_1 + 7x_1 \leq 40 \\ & x_1 + x_2 \leq 5 \\ & x_1, x_2 \in \mathbb{Z}_+, \end{aligned} \tag{Exemplo 02}$$

retirado de [Van08].

A Figura 1 ilustra graficamente (Exemplo 01) e (Exemplo 02) sendo que as as linhas contínuas correspondem às restrições, as setas ao gradiente das restrições, as linhas tracejadas às curvas de nível da função objetivo e os círculos às soluções viáveis.

2 Resolução de PLIs

Uma das possíveis maneiras de resolver problemas de programação linear inteira é utilizando métodos baseados em uma abordagem enumerativa, de modo que todas as soluções viáveis para



(a) Exemplo 01 (b) Exemplo 02

Figura 1: Representação gráfica dos exemplos.

o problema de programação linear inteira são determinadas, e por comparação entre o valor da função objetivo correspondente às soluções viáveis enumeradas, é determinada a solução ótima do problema.

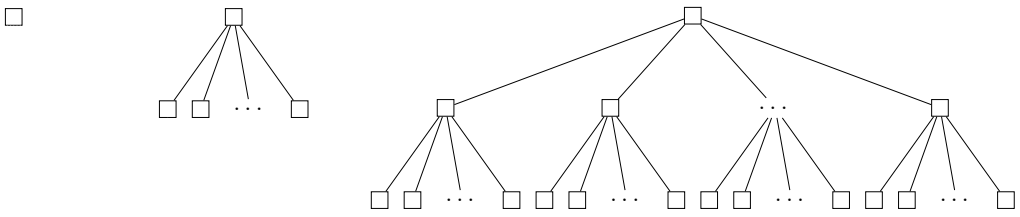
Para enumerar todas as soluções viáveis podemos utilizar, por exemplo, o procedimento estruturado nas seguintes fases da técnica "dividir para conquistar":

Divisão Seleciona-se uma variável que pode assumir mais de um valor e a cada elemento da divisão é atribuído um dos possíveis valores da variável selecionada e as demais variáveis não são alteradas.

Conquista Se todas as variáveis só podem assumir apenas um valor encontrou-se a solução do problema corrente.

Combinação Compara-se os valores da função objetivo para as soluções encontradas procurando pelo menor valor da função objetivo, que será proporcionado pela solução ótima.

A técnica "dividir para conquistar" pode ser representada por uma árvore na qual os filhos de um nó correspondem aos elementos obtidos da divisão do nó pai. Na Figura 2 ilustramos o crescimento de uma árvore proveniente da técnica "dividir para conquistar".



(a) Estágio 1. (b) Estágio 2. (c) Estágio 3.

Figura 2: Evolução da árvore proveniente da técnica "dividir para conquistar".

Na Figura 3 é representada parte da árvore correspondente à enumeração das soluções viáveis de (Exemplo 01). Os nós tracejados possuem alguma variável "livre" e os demais nós já não possuem variável "livre".

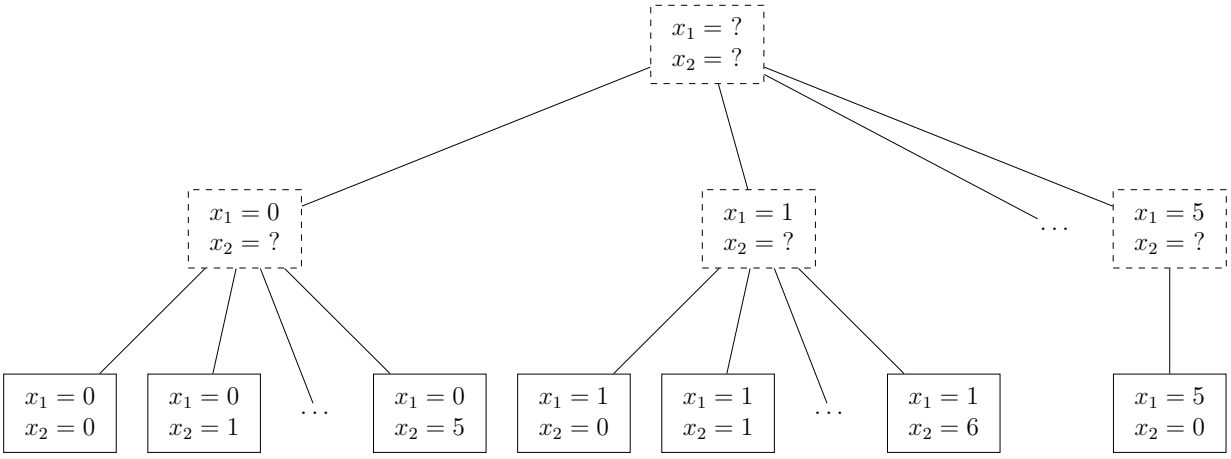


Figura 3: Árvore enumerando as soluções de (Exemplo 01).

Pela árvore apresentada na Figura 3 verificamos que, mesmo para um problema com poucas variáveis, a enumeração total das soluções viáveis de um problema de programação linear inteira é custosa devido ao número de soluções viáveis poder ser excessivamente grande.

Outra possível maneira de resolver problemas de programação linear inteira consiste em resolver uma sequência de relaxações de (PLI), i.e., um problema da forma $\min\{f(x) : x \in S_R\}$ tal que

1. $\{x : Ax \leq b, x \in \mathbb{Z}^n_+\} \subseteq S_R$ e
2. $f(x) \leq c^T x, \ \forall \ x \in \{Ax \leq b, x \in \mathbb{Z}^n_+\}$,

até que a solução da última relaxação da sequência seja viável para (PLI).

Por 1 e 2, concluímos que se a solução da relaxação é viável para (PLI) então ela é a solução ótima para (PLI).

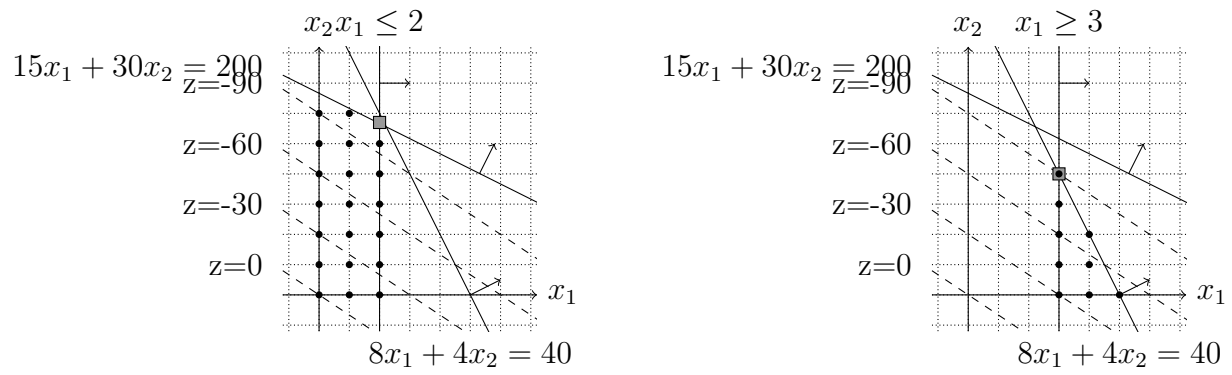
Um exemplo do uso de relaxações para resolver problemas de programação linear inteira é o *Fractional Cutting-Plane Algorithm* proposto por Ralph E. Gomory.

Além das duas maneiras apresentadas para resolver problemas de programação linear inteira também podemos combiná-las, i.e., utilizar relaxações em conjunto com métodos enumerativos.

Para facilitar o desenvolvimento dos métodos para problemas de programação linear inteira que utilizam relaxações em conjunto com métodos enumerativos vamos considerar o exemplo a seguir.

Dado (Exemplo 01) suponha que para enumear as soluções viáveis tenhamos particionado $\{x : Ax \leq b, x \in \mathbb{Z}^n_+\}$ nos conjuntos $S^1 = \{x : Ax \leq b, x \leq 2, x \in \mathbb{Z}^n_+\}$ e $S^2 = \{x : Ax \leq b, x \geq 3, x \in \mathbb{Z}^n_+\}$. A Figura 4 ilustra graficamente os problemas $\min\{c^T x : x \in S^1\}$ e $\min\{c^T x : x \in S^2\}$ sendo que as as linhas contínuas correspondem às restrições, as setas ao gradiente das restrições, as linhas tracejadas às curvas de nível da função objetivo, os círculos às soluções viáveis e o quadrado com preenchido de cinza a solução ótima da relaxação dos problemas $\min\{c^T x : x \in S^1\}$ e $\min\{c^T x : x \in S^2\}$ que corresponde aos problemas $\min\{c^T x : Ax \leq b, x \leq 2, x \in \mathbb{R}^n_+\}$ e $\min\{c^T x : Ax \leq b, x \geq 3, x \in \mathbb{R}^n_+\}$, respectivamente.

Observando a representação gráfica de $\min\{c^T x : x \in S^2\}$, ver Figura 4(b), verificamos que a solução ótima da relaxação é viável para $\min\{c^T x : x \in S^2\}$ e, portanto, é ótima para



(a) $\min\{c^T x : x \in \{Ax \leq b, x \leq 2, x \in \mathbb{Z}^n_+\}\}$ (b) $\min\{c^T x : x \in \{Ax \leq b, x \geq 3, x \in \mathbb{Z}^n_+\}\}$

Figura 4: Representação gráfica do particionamento de (Exemplo 01).

$\min\{c^T x : x \in S^2\}$. Como a solução ótima de $\min\{c^T x : x \in S^2\}$ já foi encontrada deixamos de investigar esse subproblema.

Em contrapartida, observando a representação gráfica de $\min\{c^T x : x \in S^2\}$, ver Figura 4(a), verificamos que a solução ótima da relaxação de $\min\{c^T x : x \in S^1\}$ não é viável para $\min\{c^T x : x \in S^1\}$. Deste modo devemos continuar investigando esse subproblema.

Como apresentado no exemplo, ao combinar relaxações com métodos enumerativos é possível utilizar a solução da relaxação para interromper a enumeração das soluções viáveis do problema de programação linear inteira e minimizar a dificuldade do método enumerativo, i.e., o número de soluções viáveis poder ser excessivamente grande.

Como o processo de enumeração das soluções viáveis de um problema de programação linear inteira pode ser representado por uma árvore, temos que interromper a enumeração das soluções viáveis corresponde a podar a árvore, por isso esses métodos são denominados de *Branch-and-Bound*. A seguir formalizamos quando podemos podar a árvore dos Métodos *Branch-and-Bound*, para simplificação denotada por árvore *B&B*.

Dado (PLI), seja, sem perda de generalidade, $\{S^1, S^2\}$ uma partição de $S \subset \{x \in \mathbb{Z}^n_+ : Ax \leq b\}$ e x^*_1 e x^*_2, respectivamente, a solução ótima da relaxação de $\min\{c^T x : x \in S^1\}$ e $\min\{c^T x : x \in S^2\}$, se existirem. Então ocorre apenas uma das seguintes situações para S^i, $i = 1$ ou $i = 2$:

1. $\nexists\, x^*_i$, a árvore *B&B* é podada no nó correspondente a S^i;
2. $x^*_i \in \mathbb{Z}^n_+$, a árvore *B&B* é podada no nó correspondente a S^i;
3. $x^*_i \notin \mathbb{Z}^n_+$, não é possível podar a árvore *B&B* no nó correspondente a S^i.

Sem perda de generalidade, supondo que $x^*_1 \in \mathbb{Z}^n_+$, $x^*_2 \notin \mathbb{Z}^n_+$ e que $c^T x^*_1 \leq c^T x^*_2$ é possível podar a árvore de *B&B* no nó correspondente a S^2 pois todas as soluções viáveis de $\min\{c^T x : x \in S^2\}$ apresentam valor da função objetivo maior que $c^T x^*_2$ e, por hipótese, maior que $c^T x^*_1$.

Para o caso do problema de programação linear inteira ser inviável, ao terminar a poda da árvore *B&B* não terá sido encontrada nenhuma solução viável e esse resultado é sinalizado por uma *flag* $v = 0$.

No Algoritmo 1 é apresentado um arcabouço escopo do Método *Branch-and-Bound* sintetizando as informações até o momento.

Algoritmo 1 Escopo do Método *Branch-and-Bound* para (PLI) limitado

Entrada: $\min\{c^T x : Ax \leq b, x \in \mathbb{Z}^n_+\}$, conjunto viável limitado.
Saída: (v^*, x^*, z^*). /* $v^* = 0$ se $\{Ax \leq b, z \in \mathbb{Z}^n_+\} = \emptyset$. */

1: $v^* \leftarrow 0$.
2: $S^0 \leftarrow \{Ax \leq b, x \in \mathbb{Z}^n_+\}$.
3: $\underline{z}^0 \leftarrow -\infty$.
4: $\mathcal{L} \leftarrow \{(\min\{c^T x : x \in S^0\}, \underline{z}^0)\}$.
5: $z^* \leftarrow \infty$.
6: **enquanto** $\mathcal{L} \neq \emptyset$ **faça**
7: **Selecione** elemento $(P^i, \underline{z}^i)$ de $\mathcal{L}$ e elimine-o.
8: $(v, x) \leftarrow$ **resolva** a **relaxação** de P^i. /* $v = 0$ se a relaxação de P^i é inviável. */
9: **se** $v \neq 0$ **então**
10: $z \leftarrow c^T x$.
11: **se** $z < z^*$ **então**
12: **se** $x \in S^i$ **então**
13: $v^* \leftarrow 1$.
14: $x^* \leftarrow x$.
15: $z^* \leftarrow z$.
16: Delete de $\mathcal{L}$ todos os elementos com $\underline{z}^i \geq z^*$.
17: **caso contrário**
18: $\{S^{ij}\}_{j=1}^k \leftarrow$ **Partição** de S^i.
19: $\underline{z}^{ij} \leftarrow z^i$ para $j = 1, \ldots, k$.
20: Adicione $(\min\{c^T x : x \in S^{ij}\}, \underline{z}^{ij})$, com $j = 1, 2, \ldots, k$, à $\mathcal{L}$.
21: **fim**
22: **fim**
23: **fim**
24: **fim**
25: **retorne** (v, x^*, z^*).

Pelo Algoritmo 1 verificamos que existem algumas questões em aberto. A primeira delas é o critério de seleção de um elemento de $\mathcal{L}$, ver linha 7, e será tratado em detalhes na Seção 3.

A segunda questão em aberto refere-se à forma como é feito o particionamento de S^i, ver linha 18. Maiores detalhes sobre o particionamento também serão tratados na Seção 3 mas é útil adiantar o fato de ser comum que o particionamento ser feito utilizando funções lineares para preservar a linearidade do problema.

A terceira e última questão em aberto refere-se à relaxação do problema P^i e a forma de resolvê-la, ver linha 8. Pelo informado na parágrafo anterior sobre o particionamento, é comum que P^i também seja um problema de programação linear inteira. Nemhauser[NW88] informa que é comum a utilização da relaxação linear, i.e., dado um (PLI) a relaxação linear é um caso particular de relaxação no qual $f(x) = c^T x$ e $S_R = \{Ax \leq b, x \in \mathbb{R}^n_+\}$. Como o resultado da relaxação linear é um problema de programação linear, este pode ser resolvido utilizando, por exemplo, o Método Simplex, ver maiores detalhes em [BJS04].

A maioria dos *solvers*, comerciais ou não, para PLIs utilizam variantes do Método *Branch-and-Bound* que diferem entre si, principalmente, quanto a seleção do elemento de $\mathcal{L}$ e o particionamento de S^i.

Até o momento sempre consideramos que dado um (PLI) não era conhecida nenhuma de suas soluções factíveis. No caso de se conhecer ao menos uma solução factível podemos verificar dentre elas a que possue o menor valor da função objetivo e utilizá-la desde o início do Método do *Branch-and-Bound*.

Ao fazer uso de uma solução viável conhecida temos a chance de gerar uma árvore menor e encontrar a solução ótima mais rapidamente pois pode-se podar a árvore *B&B* mais cedo.

Considerando que $\tilde{x}$ seja a melhor solução viável conhecida podemos mudar a entrada do Algoritmo 1 para

Entrada: $(\min\{c^T x : Ax \leq b, x \in \mathbb{Z}^n_+\}, \tilde{x})$. /* $\min\{c^T x : Ax \leq b, x \in \mathbb{Z}^n_+\}$ é limitado. */

e substituir a linha 5 por

$x^* \leftarrow \tilde{x}$.
$z^* \leftarrow c^T \tilde{x}$.

Mesmo que não seja conhecida uma solução viável para um problema de programação inteira existem algumas técnicas que podem ser aplicadas para determinar uma. Maiores detalhes sobre algumas destas técnicas serão abordadas na Seção 4.

3 Questões abertas no *Branch-and-Bound*

Na Seção 2 apresentamos algumas maneiras de resolver problemas de programação inteira mista. Uma das maneiras apresentadas é conhecida como Método *Branch-and-Bound* e foi sintetizada no arcabouço presente no Algoritmo 1.

No arcabouço estão em aberto

1. o critério de seleção de um elemento de $\mathcal{L}$,
2. a relaxação do problema P^i a ser utilizada e a forma de resolvê-la,
3. a construção da partição do conjunto S^i.

Ainda na Seção 2 foi informado que uma relaxação muito utilizada é a relaxação linear e uma forma de resolvê-la é utilizando o Método Simplex. A seguir será abordadado o critério de seleção de um elemento de $\mathcal{L}$ e a construção da partição de S^i.

Por último será apresentado um exemplo da árvore de *Branch-and-Bound* para (Exemplo 01) e (Exemplo 02).

3.1 Seleção de um nó de $\mathcal{L}$

Dado uma lista, não vazia, de elementos da forma $(P^i, \underline{z}^i)$ alguns dos possíveis critérios de seleção de um elemento dela são:

1. *best-first*: escolhe-se o nó com o menor valor de $\underline{z}^i$ na esperança de encontrar como candidato a ótimo a solução ótima e conseguir podar vários ramos da árvore *B&B*.
2. *depth-first search*: escolhe-se o nó de maneira a constituir uma busca em profundidade com a intenção de encontrar uma solução viável mais rapidamente pois a experiência indica que soluções viáveis são mais prováveis de serem encontradas próximo às folhas da árvore *B&B*.

3. *breadth-first search*: escolhe-se o nó de maneira a constituir uma busca em largura.

Outros critérios são encontrados em [LS99].

3.2 Particionamento do nó

Sem perda de generalidade, considerando o problema de programação linear inteira P^i que é viável, seja x^* a solução ótima da relaxação de P^i e $D(x^*)$ o conjunto dos índices de x^* cuja entrada correspondente não é inteira, i.e., $D(x^*) = \{j : x_j^* \notin \mathbb{Z}_+\}$.

Caso $D(x^*) \neq \emptyset$, desejamos construir uma partição de S^i dada por $\{S^{ij}\}_{j=1}^k$ tal que x^* não pertença a união das relaxações dos elementos da partição de S^i.

No caso da partição ser formada por apenas dois conjuntos, $k = 2$, podemos construí-la escolhendo (d, δ) tal que $\delta < d^T x^* < \delta + 1$ e fazendo $S^{i1} = S^i \cap \{x : d^T x \leq \delta\}$ e $S^{i2} = S^i \cap \{x : d^T x \geq \delta + 1\}$. Para $k \geq 3$, podemos reparticionar S^{i1} e/ou S^{i2} de modo a obter k conjuntos que formem uma partição de S^i.

Nemhauser [NW88] informa que apenas escolhas muito especiais de (d, δ) são utilizadas. A mais fácil e comumente utilizada, denominada de variável dicotômica, desigualdade trivial ou *branching on variable*, consiste em utilizar $d = e_j$ e $\delta = \lfloor x_j^* \rfloor$, [NW88, AKM05, LS99].

Normalmente existe em x^* mais de um $x_j^* \notin \mathbb{Z}$ e por isso precisamos definir um critério para a escolha do $j \in D(x^*)$ que será utilizado no particionamento. Não se conhece, ainda, um método robusto para a escolha do melhor j para o particionamento. Consequentemente, é comum a definição de um *score* para cada $j \in D(x^*)$, $s_d(j)$, e escolher o j correspondente ao maior *score*.

Um dos possíveis métodos para o cálculo do *score*, denominado de *most infeasible branching*, consiste em utilizar como *score* a distância de x_j^* ao inteiro mais próximo, i.e.,

$$s_d(j) = |x_j^* - \lfloor x_j^* + 0,5 \rfloor|.$$

e selecionar a variável com o maior *score*, ou seja, a variável cuja parte fracionária é mais próxima de $0,5$. Achterberg [AKM05] afirma que a razão por trás desta escolha é selecionar a variável cuja tendência de arredondamento é menos visível.

Outros métodos para o cálculo do *score* são encontrados em [AKM05] e [BGG+71].

3.3 Exemplo

Na Figura 5 é apresentado uma árvore *B&B* para (Exemplo 02) utilizando para a seleção de um nó de $\mathcal{L}$ o critério *depth-first search*, relaxações lineares resolvidas pelo Método Simplex e para o particionamento do nó variável dicotômica com o *score* sendo calculado de acordo com o *most infeasible branching*.

4 Heurísticas mergulho

Considerando (PLI), temos, como informado no fim da Seção 2, que, se for conhecida uma solução viável para (PLI), $\tilde{x}$, é possível iniciar o Algoritmo 1 com $x^* = \tilde{x}$ e $z^* = c^T \tilde{x}$.

Se a distância $||x^* - \tilde{x}||$ entre a solução ótima de (PLI), x^*, e a solução viável conhecida, $\tilde{x}$, for pequena, então espera-se que a árvore de *B&B* produzida seja menor do que aquela produzida sem o conhecimento de $\tilde{x}$.

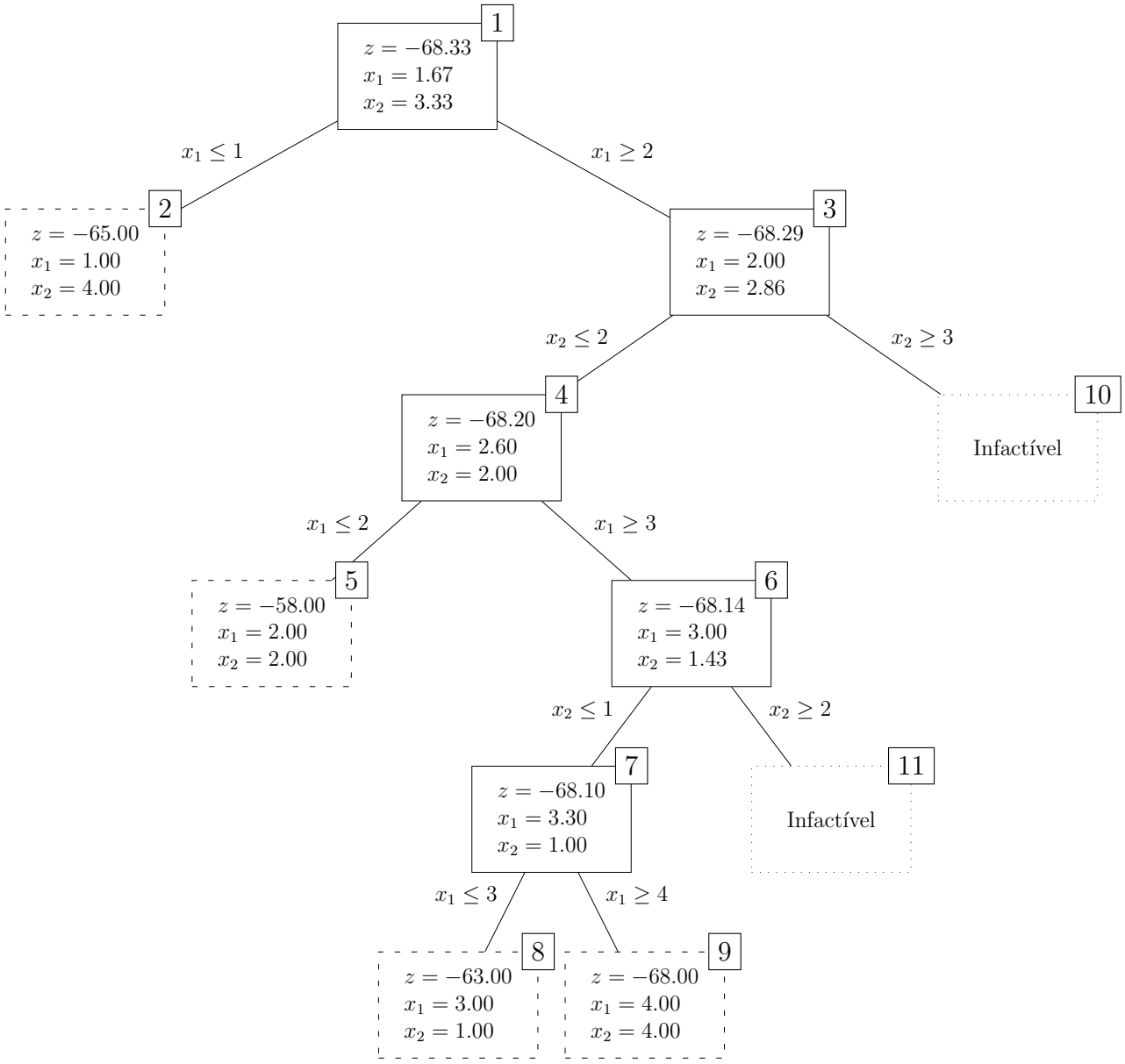


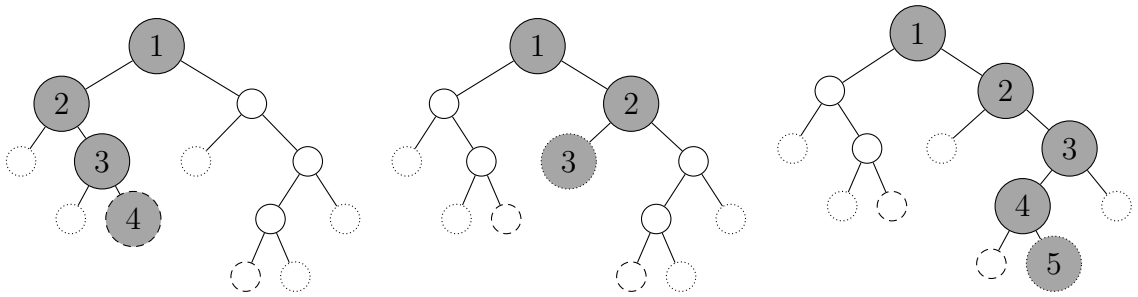
Figura 5: Árvore de *Branch-and-Bound* de (Exemplo 02).

Berthold [Ber06] apresenta algumas heurísticas, classificadas como heurísticas mergulho, que buscam fornecer $\tilde{x}$ tal que $||x^* - \tilde{x}||$ seja pequena.

Uma heurística mergulho assemelha-se ao Método *Branch-and-Bound* com a resalva de que a cada nó possui no máximo um filho, o que corresponde a simular uma trajetória de descida na árvore *B&B*, e por isso seu nome.

Na Figura 6 encontram-se representadas heurísticas mergulho, em destaque, e a árvore *B&B* correspondente a um problema de programação linear inteira em que os nós potilhados são iinviáveis, os tracejados correspondem a uma solução viável e os demais possuem alguma variável não inteira.

Os critérios de poda do Método *Branch-and-Bound* correspondem ao critério de parada para as heurísticas mergulho. Dado (PLI), seja $S^{i+1} \subset S^i \subset \{x \in \mathbb{Z}_+^n : Ax \leq b\}$ e x^* a solução ótima da relaxação de $\min\{c^T x : x \in S^{i+1}\}$, se existir. Então ocorre apenas uma das seguintes



(a) Exemplo de *diving*. (b) Exemplo de *diving*. (c) Exemplo de *diving*.

Figura 6: Exemplos de heurísticas mergulhos, em destaque, e árvore *B&B* correspondente.

situações:

1. $\nexists x^*$, a heurística mergulho é encerrada;
2. $x^* \in \mathbb{Z}^n_+$, a heurística mergulho é encerrada;
3. $x^* \notin \mathbb{Z}^n_+$, a heurística mergulho continua.

Um inconveniente das heurísticas mergulho é a possibilidade dela ser encerrada porque não existe x^* e, consequentemente, ter que iniciar o Método *Branch-and-Bound* sem uma solução viável conhecida. Para tentar minimizar esse inconveniente podemos modificar a heurística mergulho de maneira que seja permitido um grau de *backtracking*, i.e., supondo que tenha sido resolvido o problema correspondente ao i-ésimo nó, realizado o particionamento, construido o problema correspondente ao $(i+1)$-ésimo nó e ao resolve o novo problema descobriu-se que o mesmo é inviável, então oproblema dado por $\min\{c^T x : x \in \tilde{S}^{i+1}\}$, onde $\tilde{S}^{i+1} = S^i \setminus S^{i+1}$, é construído e resolvido, dando-se continuidade a heurística a partir do último problema construído.

É importante destacar, que ao permitir um grau de *backtracking*, caso não exista solução para a relaxação do problema $\min\{c^T x : x \in \tilde{S}^{i+1}\}$, a heurística é encerrada pois caso contrário a heurística iniciaria um *loop* infinito.

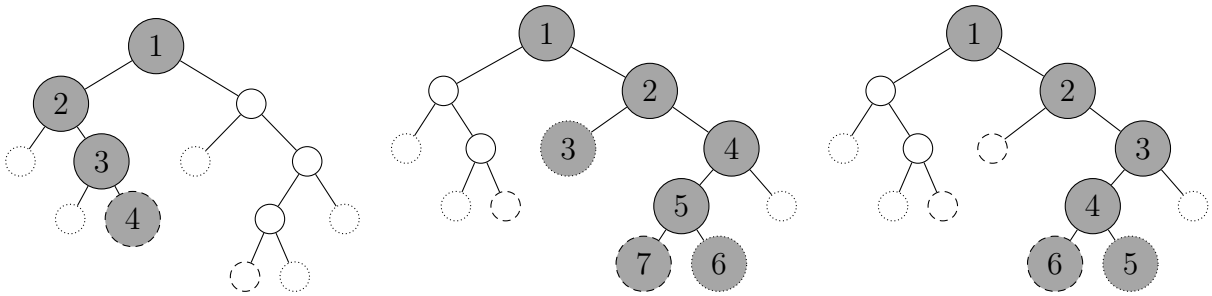
Na Figura 7 encontram-se representadas heurísticas mergulho permitindo um grau de *backtracking*, em destaque, e a árvore de *B&B* correspondente a um problema de programação linear inteira em que os nós pontilhados são infactíveis, os tracejados correspondem a uma solução viável e os demais possuem alguma variável não inteira.

No Algoritmo 2 é apresentado o arcabouço das heurísticas mergulho com possibilidade de permitir um grau de *backtracking*.

Pelo Algoritmo 2, verifica-se que existem algumas questões em aberto. Essas questões são semelhantes às presentes do arcabouço no Método *Branch-and-Bound*, Algoritmo 1, i.e., a relaxação do problema P a ser utilizada e a forma de resolvê-la, ver linhas 5, 15 e 19, e a construção do subconjunto de S, ver linha 13.

Como informado nas Seções 2 e 3, uma relaxação muito utilizada para o Método *Branch-and-Bound* é a relaxação linear e uma forma de resolvê-la é utilizando o Método Simplex. Para as heurísticas mergulho também será utilizado a relaxação linear e o Método Simplex.

Em relação à construção do subconjunto de S, efetua-se um particionamento do nó utilizando uma variável dicotômica. Como informado quando tratamos do particionamento do nó para o Método *Branch-and-Bound*, não se conhece, ainda, um método robusto para a escolha



(a) Exemplo de *diving*. (b) Exemplo de *diving*. (c) Exemplo de *diving*.

Figura 7: Exemplos de heurísticas mergulhos modificadas, em destaque, e árvore *B&B* correspondente.

Algoritmo 2 Arcabouço das heurísticas mergulho com possibilidade de um grau de *backtracking*.

Entrada: $(\min\{c^T x : Ax \leq b, x \in \mathbb{Z}^n_+\},\ back)$. /* $\min\{c^T x : Ax \leq b, x \in \mathbb{Z}^n_+\}$ é limitado e $back = 0$ caso não deseje-se utilizar um grau de *backtracking*. */

Saída: $(flag, \tilde{x})$. /* $flag = 0$ se e somente se não encontrou-se $\tilde{x}$ viável para (PLI). */

1: $S \leftarrow \{Ax \leq b, x \in \mathbb{Z}^n_+\}$.
2: $P \leftarrow \min\{c^T x : x \in S\}$.
3: $flag \leftarrow 0$.
4: $i \leftarrow 0$.
5: $(v, \tilde{x}) \leftarrow$ solução da relaxação de P. /* $v = 0$ se a relaxação de P é inviável. */
6: **enquanto** $i \leq MAXIT$ **faça**
7: $i \leftarrow i + 1$.
8: **se** $v \neq 0$ **então**
9: **se** $\tilde{x} \in S$ **então**
10: $flag \leftarrow 1$.
11: **vá para linha** 26.
12: **fim**
13: $\tilde{S} \leftarrow$ subconjunto de S.
14: $P \leftarrow \min\{c^T x : x \in \tilde{S}\}$.
15: $(v, \tilde{x}) \leftarrow$ solução da relaxação de P.
16: **se** $v = 0$ e $back \neq 0$ **então**
17: $\tilde{S} \leftarrow S \setminus \tilde{S}$.
18: $P \leftarrow \min\{c^T x : x \in \tilde{S}\}$.
19: $(v, \tilde{x}) \leftarrow$ solução da relaxação de P.
20: **fim**
21: $S \leftarrow \tilde{S}$.
22: **caso contrário**
23: **vá para linha** 26.
24: **fim**
25: **fim**
26: **retorne** $(flag, \tilde{x})$.

da melhor variável a ser utilizada no particionamento. Consequentemente, é comum a definição de um *score* para cada uma das variáveis, $s_h(j)$, e escolher aquela correspondente ao melhor *score*.

A seguir detalhamos as heurísticas mergulho fracionário, mergulho viável e mergulho sp, todas pertencentes à categoria de heurísticas mergulho e descritas por Berthold [Ber06].

4.1 Mergulho fracionário

Quando abordamos o particionamento do nó para o Método *Branch-and-Bound* apresentamos como opção para o cálculo do *score* o *most infeasible branching* que consiste em utilizar como *score* a distância de x_j^* ao inteiro mais próximo, i.e.,

$$s_d(j) = |x_j^* - \lfloor x_j^* + 0,5 \rfloor|.$$

A razão para essa escolha era privilegiar a variável cuja tendência de arredondamento fosse menos visível.

Nessa heurística estamos interessados em construir subconjuntos que contenham a solução ótima da relaxação após arredondarmos a variável cuja tendência for mais visível. Logo, utilizamos $s_h(j) = s_d(j)$.

O subconjunto é construindo utilizando a variável com o menor *score*, $s_h(j)$, pela adição da restrição $x_j \leq \lfloor x_j^* \rfloor$ se $x_j^* \leq \lfloor x_j^* \rfloor + 0.5$ e $x_j \geq \lceil x_j^* \rceil$ caso contrário.

Em caso de empate no valor do *score*, a variável com menor valor de j é utilizada.

Nas Figuras 8 e 9 é ilustrada a heurística mergulho fracionário para (Exemplo 01) e (Exemplo 02), respectivamente.

4.2 Mergulho viável

Nessa heurística procuramos evitar que a solução ótima da relaxação do problema de programação linear inteira correspondente a um nó seja inviável para a relaxação do problema de programação linear inteira correspondente de seu filho, por isso seu nome.

Para dar uma idéia, considera a i-ésima restrição de (PLI), e seja x^* a solução ótima da relaxação de (PLI). Se x_j^* não é inteiro e $a_{ij} \geq 0$ ($a_{ij} \leq 0$) é fácil verificar que ao substituir x_j^* por $\lfloor x_j^* \rfloor$ ($\lceil x_j^* \rceil$) a i-ésima restrição não é violada. Já ao substituir x_j^* por $\lceil x_j^* \rceil$ ($\lfloor x_j^* \rfloor$) é possível que a i-ésima restrição seja violada.

Logo, o número de $a_{ij} < 0$ ($a_{ij} > 0$) em $A_{\bullet j}$ corresponde ao número máximo de restrições que podem ser violadas ao substituir x_j^* por $\lfloor x_j^* \rfloor$ ($\lceil x_j^* \rceil$).

Portanto, nessa heurística utilizamos

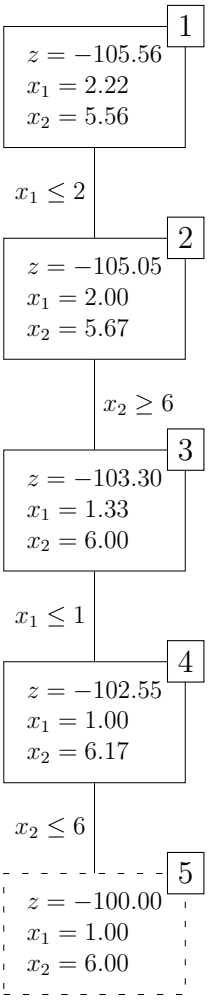
$$s_h(j) = \min\{|\{a_{ij} : a_{ij} < 0\}|, |\{a_{ij} : a_{ij} > 0\}|\}.$$

O subconjunto é construído utilizando a variável com o menor *score*, $s_h(j)$, pela adição da restrição $x_j \leq \lfloor x_j^* \rfloor$ se $|\{a_{ij} : a_{ij} < 0\}| \leq |\{a_{ij} : a_{ij} > 0\}|$ e $x_j \geq \lceil x_j^* \rceil$ caso contrário.

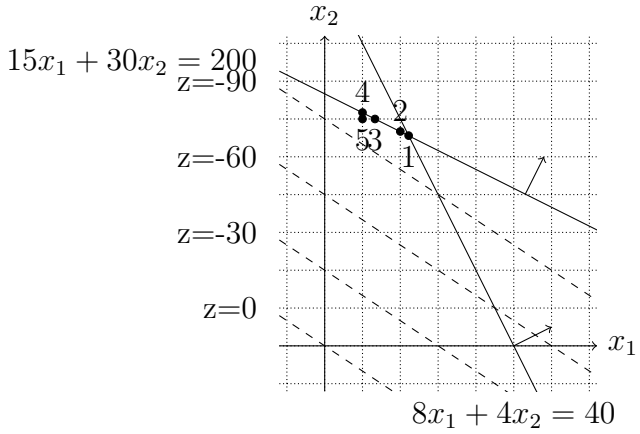
Em caso de empate escolhe-se a variável com menor valor de j.

Experimentos computacionais mostram que quando $|\{a_{ij} : a_{ij} < 0\}| = 0$ ($|\{a_{ij} : a_{ij} > 0\}| = 0$) costuma-se encontrar uma solução viável para (PLI) mais rapidamente adicionando a restrição $x_j \geq \lceil x_j^* \rceil$ ($x_j \leq \lfloor x_j^* \rfloor$).

Nas Figuras 10 e 11 é ilustrada a heurística mergulho viável para (Exemplo 01) e (Exemplo 02), respectivamente.



(a) Evolução da heurística mergulho fracionário.



(b) Representação gráfica da evolução da heurística mergulho fracionário.

Figura 8: Heurística mergulho fracionário para (Exemplo 01).

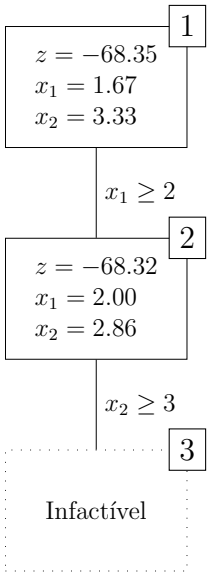
4.3 Mergulho sp

Essa heurística só pode ser aplicada em problema de programação linear inteira que apresente pelo menos uma restrição do tipo *set partition*, i.e., uma restrição da forma $a^T x = 1$, onde as variáveis são binárias e as entrads de a pertencem $\{0, 1\}$. Como muitos problemas reais apresentam esse tipo de restrição, e.g., *assignment and matching problems*, torna-se interessante considerar essa heurística.

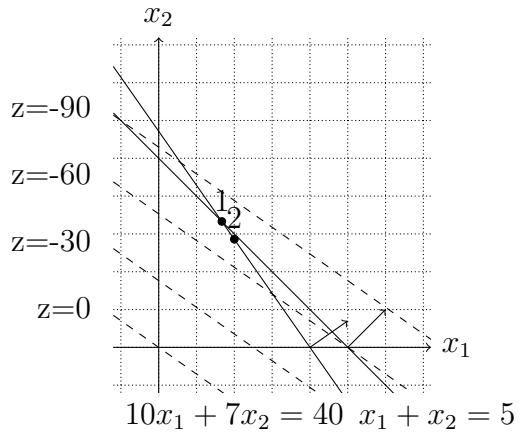
Pela restrição do tipo *set partition* é fácil verificar que, caso a solução ótima da relaxação do problema de programação linear inteira, x^*, apresente x_j^* fracionário, então existe pelo menos um x_i^*, $i \neq j$, fracionário. Também é fácil verificar que ao fixar uma variável fracionária em 0 ou 1 estaremos também fixando outras variáveis.

Nessa heurística procuramos fixar uma variável binária de modo que o maior número possível de variáveis também sejam fixadas e evitando grandes variações no valor da função objetivo.

É fácil notar que, ao fixar a variável x_j^* a variável x_i^*, $i \neq j$, só será afetada se existir k tal que a_{ki} e a_{kj} sejam diferentes de 0. Para um grande número de variáveis essa verificação pode

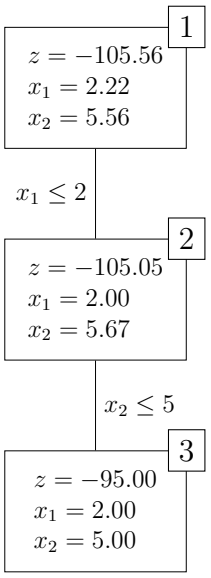


(a) Evolução da heurística mergulho fracionário.

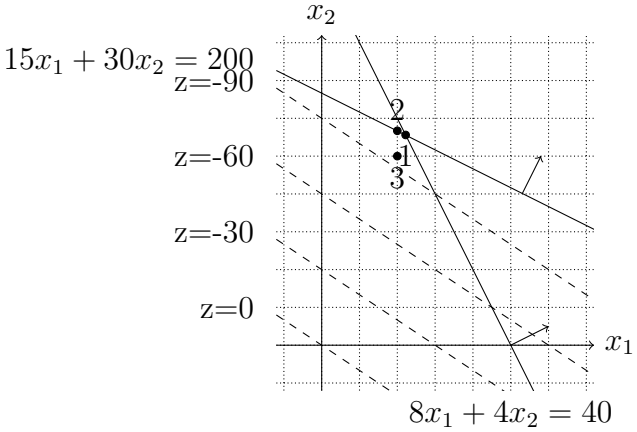


(b) Representação gráfica da evolução da heurística mergulho fracionário.

Figura 9: Heurística mergulho fracionário para (Exemplo 02).



(a) Evolução da heurística mergulho viável.



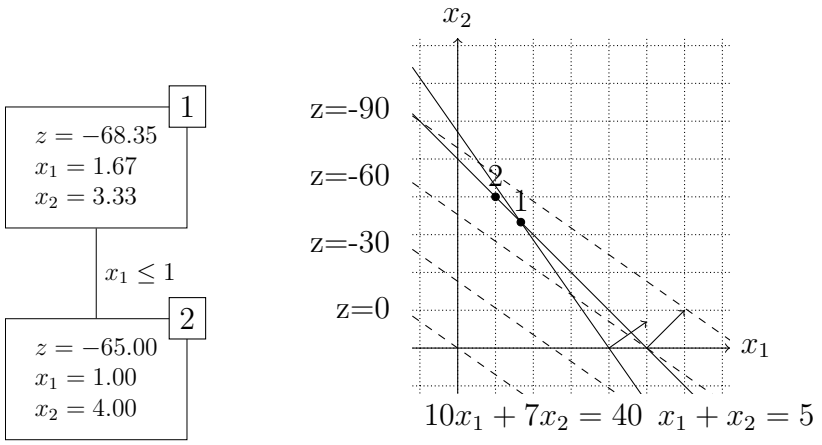
(b) Representação gráfica da evolução da heurística mergulho viável.

Figura 10: Heurística mergulho viável para (refE:Exemp01).

requerer um excessivo de testes e uma alternativa é verificar o número de restrições afetadas, i.e., a restrição k é afetada se $a_{kj} \neq 0$.

Portanto, nessa heurística utilizamos

$$s_h(j) = \begin{cases} ((\lceil x_j \rceil - x_j)c_j) \,/|\{a_{ij} : a_{ij} \neq 0\}|, & \text{se } c_j \geq 0, \\ ((\lfloor x_j \rfloor - x_j)c_j) \,/|\{a_{ij} : a_{ij} \neq 0\}|, & \text{se } c_j < 0. \end{cases}$$



(a) Evolução da heurística mergulho viável.

(b) Representação gráfica da evolução da heurística mergulho viável.

Figura 11: Heurística mergulho viável para (Exemplo 02).

que corresponde a razão da variação da função objetivo ao fixar x_j pelo número de restrições afetadas. O condicional é uma tentativa de evitar que ao fixar a variável x_j a solução seja inviável para a relaxação.

O subconjunto é construído utilizando a variável com o menor *score*, $s_h(j)$, pela adição da restrição $x_j \leq \lfloor x_j^* \rfloor$, se $c_j < 0$, e $x_j \geq \lceil x_j^* \rceil$, caso contrário.

Em caso de empate escolhe-se a variável com menor valor de j.

5 Implementação

O autor deste trabalho implementou as três heurísticas mergulho detalhadas na seção 4, i.e., mergulho fracionário, mergulho viável e mergulho sp, utilizando a lingaguem C e a API do GLPK, versão 4.47.

Os códigos referentes a implementação podem ser obtidos enviando um e-mail para `r.gaia.cs@gmail.com`.

6 Testes Computacionais

Testou-se a implementação das heurísticas mergulho detalhadas na seção 4, i.e., mergulho fracionário, mergulho viável e mergulho sp, com os problemas da biblioteca MIPLIB2, disponíveis em `http://miplib.zib.de/miplib2/miplib2.html`, em um computador equipado com um processador Intel(R) Pentium(R) D 3.40 GHz e 2 GB de memória RAM rodando com o sistema operacional UBUNTU 10.10 (Kernel Linux 2.6.35-30-generic e Gnome 2.32.0).

A implementação foi compilada utilizando a versão 4.4.4 do GCC.

Os resultados dos testes são apresentados nas Tabelas 1, 2, 3 e 4, presentes no Anexo A. Nas Tabelas 2, 3 e 4 não aparece o problema diamond.mps pois este é ilimitado não sendo utilizado o Método *Branch-and-Bound* e na Tabela 4 aparecem apenas os problemas com pelo menos uma restrição do tipo *set partition*.

Parte das informações das Tabelas 1, 2, 3 e 4 foram sintetizadas nas Figuras 12 e 13, presentes no Anexo A, que comparam os tempos de duração da busca para solução dos problemas com as diferentes abordagens, respectivamente, para os problemas que sempre foi encontrado a solução ótimas e para os problemas que pelo menos uma vez não encontrou-se a solução ótima.

7 Análise de dados e conclusões

Pelos dados das Tabelas 1, 2, 3 e 4 verificamos que as heurísticas mergulho sem um grau de *backtracking* costumam falhar bastante na tarefa de encontrar uma solução viável para o problema de programação linear inteira, a heurística mergulho fracionário falhou em aproximadamente 60% dos problemas enquanto que com um grau de *backtracking* falhou em apenas 30% dos problemas.

Como muitos dos problemas foram resolvidos em poucos segundos não conseguimos medir o ganho de eficiência ao utilizar alguma das heurísticas para procurar uma solução viável. De todo modo vale destacar o problema p6000 que com as heurísticas mergulho fracionário e mergulho sp foi resolvido enquanto que nos outros casos a solução ótima não foi encontrada.

O problema air06 apresentou um comportamento fora do padrão pela heurística mergulho fracionário demorar muito tempo. O motivo desse fato é desconhecido.

Concluimos que o uso de uma solução viável no início do Método *Branch-and-Bound* não garante um ganho de performance no processo de solução de um problema de programação linear inteira pois o custo do processo de encontrar uma solução viável pode ser elevado.

Referências

[AKM05] T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters **33** (2005), no. 1, 42–54.

[BBI92] R. E. Bixby, E. A. Boyd, and R. R. Indovina, *MIPLIB: A test set of mixed integer programming problems*, SIAM News **25** (1992), 16 (English).

[Ber06] Timo Berthold, *Primal heuristics for mixed integer programs*, Master's thesis, 2006.

[BGG+71] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent, *Experiments in mixed-integer linear programming*, Mathematical Programming **1** (1971), no. 1, 76–94, 10.1007/BF01584074.

[BJS04] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali, *Linear programming and network flows*, Wiley-Interscience, 2004.

[LS99] J.T. Linderoth and M.W.P. Savelsbergh, *A computational study of search strategies for mixed integer programming*, INFORMS Journal on Computing **11** (1999), 173–187.

[NW88] George L. Nemhauser and Laurence A. Wolsey, *Integer and combinatorial optimization*, Wiley-Interscience, New York, NY, USA, 1988.

[Tay09] B.W. Taylor, *Introduction to management science*, Pearson Custom Business Resources, Prentice Hall, 2009.

[Van08] R.J. Vanderbei, *Linear programming: foundations and extensions*, International series in operations research & management science, Springer, 2008.

A Tabelas e figuras dos testes computacionais

A seguir encontram-se as tabelas e figuras correspondentes aos testes computacionais.

Tabela 1: *Benchmark* do Método *Branch-and-Bound* para a biblioteca MIPLIB2.

Nome	Número de restrições	Número de variáveis	Número de variáveis inteiras	Resolveu relaxação inicial	Solução da relaxação inicial é viável	Tempo até resolver relaxação inicial	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado
air01.mps	24	771	771	Sim	Não	0	Sim	0	$6.7960e+03$
air02.mps	51	6774	6774	Sim	Não	0	Sim	1	$7.8100e+03$
air06.mps	826	8627	8627	Sim	Não	13	Sim	20	$4.9649e+04$
bell3b.mps	124	133	71	Sim	Não	0	Sim	2	$1.1786e+07$
bell4.mps	106	117	64	Sim	Não	0	Sim	5	$1.8541e+07$
bm23.mps	21	27	27	Sim	Não	0	Sim	0	$3.4000e+01$
cracpb1.mps	144	572	572	Sim	Não	0	Sim	2	$2.2199e+04$
diamond.mps	5	2	2	Sim	Não	0	Sim	0	Ilimitado
fixnet3.mps	479	878	378	Sim	Não	0	Não	7201	$5.5225e+04$
fixnet4.mps	479	878	378	Sim	Não	0	Não	7201	$1.2182e+04$
lp4l.mps	86	1086	1086	Sim	Não	0	Sim	1	$2.9670e+03$
misc01.mps	55	83	82	Sim	Não	0	Sim	1	$5.6350e+02$
misc02.mps	40	59	58	Sim	Não	0	Sim	0	$1.6900e+03$
misc04.mps	1726	4897	30	Sim	Não	1	Sim	1	$2.6667e+03$
misc05.mps	301	136	74	Sim	Não	0	Sim	1	$2.9845e+03$
mod013.mps	63	96	48	Sim	Não	0	Sim	0	$2.8095e+02$
p0040.mps	24	40	40	Sim	Não	0	Sim	0	$6.2027e+04$
p0291.mps	253	291	291	Sim	Não	0	Sim	0	$5.2237e+03$
p6000.mps	2177	6000	6000	Sim	Não	1	Não	7201	$-2.4129e+06$
pipex.mps	26	48	48	Sim	Não	0	Sim	1	$7.8826e+02$
sample2.mps	46	67	21	Sim	Não	0	Sim	0	$3.7500e+02$
sentoy.mps	31	60	60	Sim	Não	0	Sim	0	$-7.7720e+03$
set1al.mps	493	712	240	Sim	Não	0	Não	7201	$1.5915e+04$
set1cl.mps	493	712	240	Sim	Não	0	Não	7201	$6.4842e+03$
stein15.mps	37	15	15	Sim	Não	0	Sim	0	$9.0000e+00$
stein9.mps	14	9	9	Sim	Não	0	Sim	0	$5.0000e+00$

Tabela 2: *Benchmark* da heurística mergulho fracionário para a biblioteca MIPLIB2.

Nome Nome	Mergulho normal					Mergulho modificado (com um grau de *backtracking*				
	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado
air01.mps	Sim	0	Sim	0	$6.7960e+03$	Sim	0	Sim	0	$6.7960e+03$
air02.mps	Sim	1	Sim	2	$7.8100e+03$	Sim	1	Sim	2	$7.8100e+03$
air06.mps	Não	699	Sim	706	$4.9649e+04$	Não	707	Sim	713	$4.9649e+04$
bell3b.mps	Não	0	Sim	1	$1.1786e+07$	Não	0	Sim	1	$1.1786e+07$
bell4.mps	Não	0	Sim	5	$1.8541e+07$	Sim	0	Sim	2	$1.8541e+07$
bm23.mps	Não	0	Sim	0	$3.4000e+01$	Não	0	Sim	0	$3.4000e+01$
cracpb1.mps	Não	1	Sim	2	$2.2199e+04$	Não	0	Sim	2	$2.2199e+04$
fixnet3.mps	Não	0	Não	7200	$5.5225e+04$	Sim	1	Não	7201	$5.5225e+04$
fixnet4.mps	Não	1	Não	7201	$1.2182e+04$	Sim	0	Não	7201	$1.2182e+04$
lp4l.mps	Sim	0	Sim	0	$2.9670e+03$	Sim	0	Sim	1	$2.9670e+03$
misc01.mps	Não	0	Sim	1	$5.6350e+02$	Não	0	Sim	0	$5.6350e+02$
misc02.mps	Sim	0	Sim	0	$1.6900e+03$	Sim	0	Sim	1	$1.6900e+03$
misc04.mps	Não	1	Sim	1	$2.6667e+03$	Não	0	Sim	1	$2.6667e+03$
misc05.mps	Não	1	Sim	1	$2.9845e+03$	Sim	0	Sim	1	$2.9845e+03$
mod013.mps	Sim	1	Sim	1	$2.8095e+02$	Sim	0	Sim	0	$2.8095e+02$
p0040.mps	Sim	0	Sim	0	$6.2027e+04$	Sim	0	Sim	0	$6.2027e+04$
p0291.mps	Não	0	Sim	1	$5.2237e+03$	Sim	0	Sim	0	$5.2237e+03$
p6000.mps	Sim	133	Sim	403	$-2.4514e+06$	Sim	134	Sim	404	$-2.4514e+06$
pipex.mps	Não	0	Sim	0	$7.8826e+02$	Não	0	Sim	0	$7.8826e+02$
sample2.mps	Não	0	Sim	0	$3.7500e+02$	Não	1	Sim	1	$3.7500e+02$
sentoy.mps	Sim	0	Sim	0	$-7.7720e+03$	Sim	0	Sim	0	$-7.7720e+03$
set1al.mps	Não	0	Não	7200	$1.5915e+04$	Sim	1	Não	7202	$1.5915e+04$
set1cl.mps	Não	0	Não	7200	$6.4842e+03$	Sim	0	Não	7201	$6.4842e+03$
stein15.mps	Sim	0	Sim	0	$9.0000e+00$	Sim	1	Sim	1	$9.0000e+00$
stein9.mps	Sim	0	Sim	0	$5.0000e+00$	Sim	0	Sim	0	$5.0000e+00$

Tabela 3: *Benchmark* da heurística mergulho viável para a biblioteca MIPLIB2.

Nome	Mergulho normal					Mergulho modificado (com um grau de *backtracking*				
	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado
air01.mps	Sim	0	Sim	0	$6.7960e+03$	Sim	1	Sim	1	$6.7960e+03$
air02.mps	Sim	1	Sim	1	$7.8100e+03$	Sim	0	Sim	1	$7.8100e+03$
air06.mps	Sim	36	Sim	39	$4.9649e+04$	Sim	35	Sim	38	$4.9649e+04$
bell3b.mps	Não	0	Sim	2	$1.1786e+07$	Sim	0	Sim	1	$1.1786e+07$
bell4.mps	Não	0	Sim	4	$1.8541e+07$	Sim	0	Sim	4	$1.8541e+07$
bm23.mps	Não	0	Sim	0	$3.4000e+01$	Não	0	Sim	0	$3.4000e+01$
cracpb1.mps	Não	1	Sim	2	$2.2199e+04$	Sim	0	Sim	2	$2.2199e+04$
fixnet3.mps	Sim	1	Não	7201	$5.5225e+04$	Sim	0	Não	7201	$5.5225e+04$
fixnet4.mps	Sim	1	Não	7201	$1.2182e+04$	Sim	0	Não	7201	$1.2182e+04$
lp4l.mps	Sim	1	Sim	2	$2.9670e+03$	Sim	0	Sim	1	$2.9670e+03$
misc01.mps	Não	0	Sim	0	$5.6350e+02$	Sim	0	Sim	1	$5.6350e+02$
misc02.mps	Sim	1	Sim	1	$1.6900e+03$	Sim	0	Sim	0	$1.6900e+03$
misc04.mps	Não	1	Sim	1	$2.6667e+03$	Não	1	Sim	1	$2.6667e+03$
misc05.mps	Não	0	Sim	1	$2.9845e+03$	Sim	1	Sim	1	$2.9845e+03$
mod013.mps	Sim	0	Sim	0	$2.8095e+02$	Sim	1	Sim	1	$2.8095e+02$
p0040.mps	Não	1	Sim	1	$6.2027e+04$	Sim	0	Sim	0	$6.2027e+04$
p0291.mps	Sim	0	Sim	0	$5.2237e+03$	Sim	0	Sim	0	$5.2237e+03$
p6000.mps	Não	11	Não	7212	$-2.4131e+06$	Sim	387	Não	7587	$-2.4102e+06$
pipex.mps	Não	0	Sim	0	$7.8826e+02$	Não	0	Sim	0	$7.8826e+02$
sample2.mps	Não	0	Sim	0	$3.7500e+02$	Não	0	Sim	0	$3.7500e+02$
sentoy.mps	Sim	0	Sim	0	$-7.7720e+03$	Sim	0	Sim	0	$-7.7720e+03$
set1al.mps	Sim	1	Não	7201	$1.5913e+04$	Sim	0	Não	7201	$1.5913e+04$
set1cl.mps	Sim	1	Não	7201	$6.4842e+03$	Sim	0	Não	7201	$6.4842e+03$
stein15.mps	Sim	0	Sim	0	$9.0000e+00$	Sim	0	Sim	0	$9.0000e+00$
stein9.mps	Sim	0	Sim	0	$5.0000e+00$	Sim	0	Sim	0	$5.0000e+00$

Tabela 4: *Benchmark* da heurística mergulho sp para a biblioteca MIPLIB2.

Nome	Mergulho normal					Mergulho modificado (com um grau de *backtracking*				
Nome	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado	Heurística encontrou solução	Tempo até heurística terminar	*B&B* encontrou solução	Tempo até *B&B* terminar	Melhor valor encontrado
air01.mps	Sim	0	Sim	0	$6.7960e+03$	Sim	0	Sim	0	$6.7960e+03$
air02.mps	Sim	1	Sim	2	$7.8100e+03$	Sim	0	Sim	1	$7.8100e+03$
air06.mps	Sim	48	Sim	49	$4.9649e+04$	Sim	47	Sim	49	$4.9649e+04$
cracpb1.mps	Não	0	Sim	1	$2.2199e+04$	Sim	0	Sim	2	$2.2199e+04$
lp4l.mps	Sim	0	Sim	1	$2.9670e+03$	Sim	0	Sim	1	$2.9670e+03$
misc01.mps	Não	0	Sim	0	$5.6350e+02$	Não	0	Sim	0	$5.6350e+02$
misc02.mps	Sim	0	Sim	0	$1.6900e+03$	Sim	0	Sim	0	$1.6900e+03$
misc05.mps	Não	0	Sim	0	$2.9845e+03$	Não	0	Sim	1	$2.9845e+03$
p6000.mps	Sim	128	Sim	1699	$-2.4514e+06$	Sim	132	Sim	1727	$-2.4514e+06$
pipex.mps	Não	0	Sim	0	$7.8826e+02$	Não	0	Sim	0	$7.8826e+02$

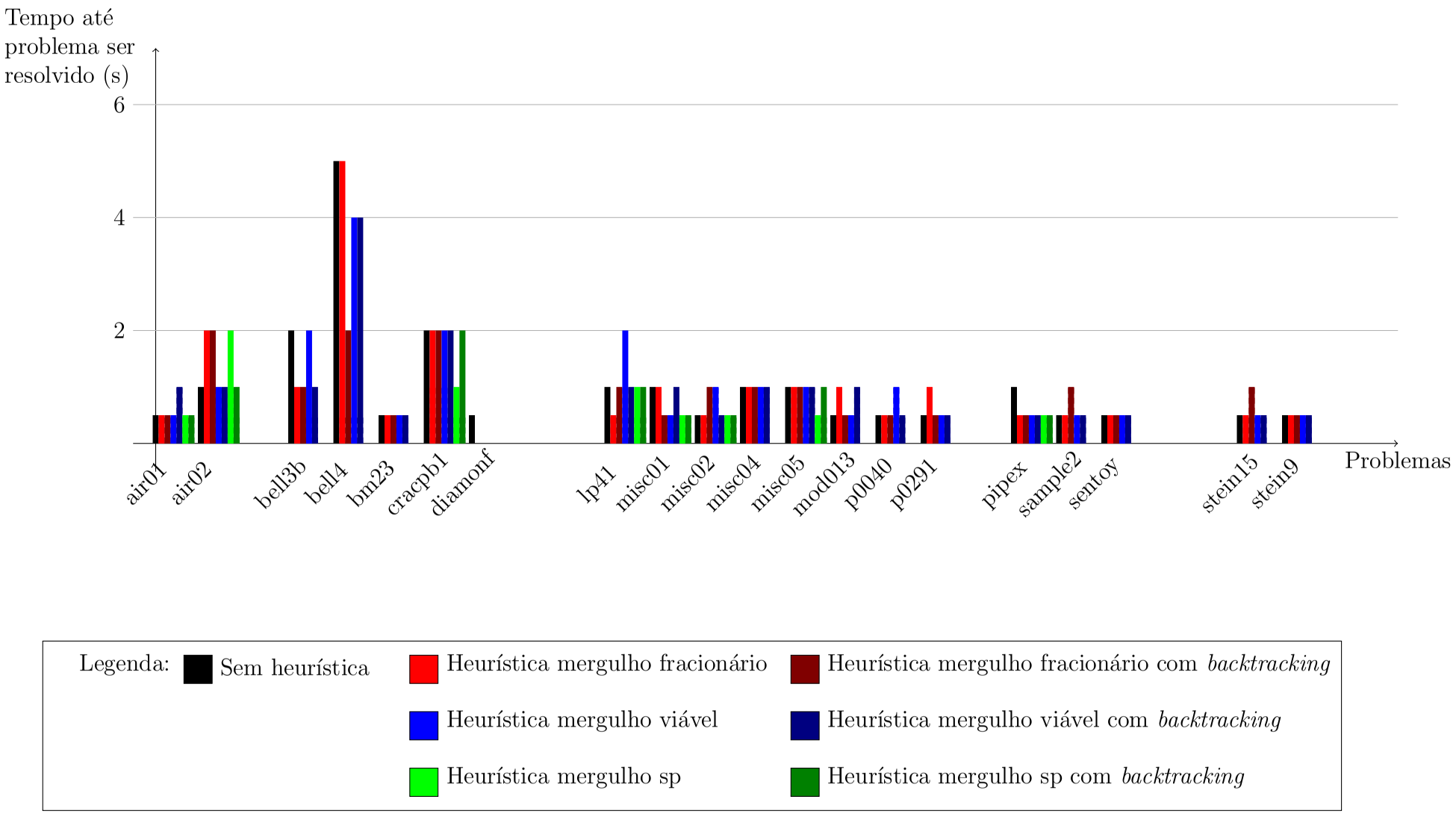


Figura 12: Gráfico comparativo do tempo de execução dos problemas da biblioteca MIPLIB2 sempre foram resolvidos.

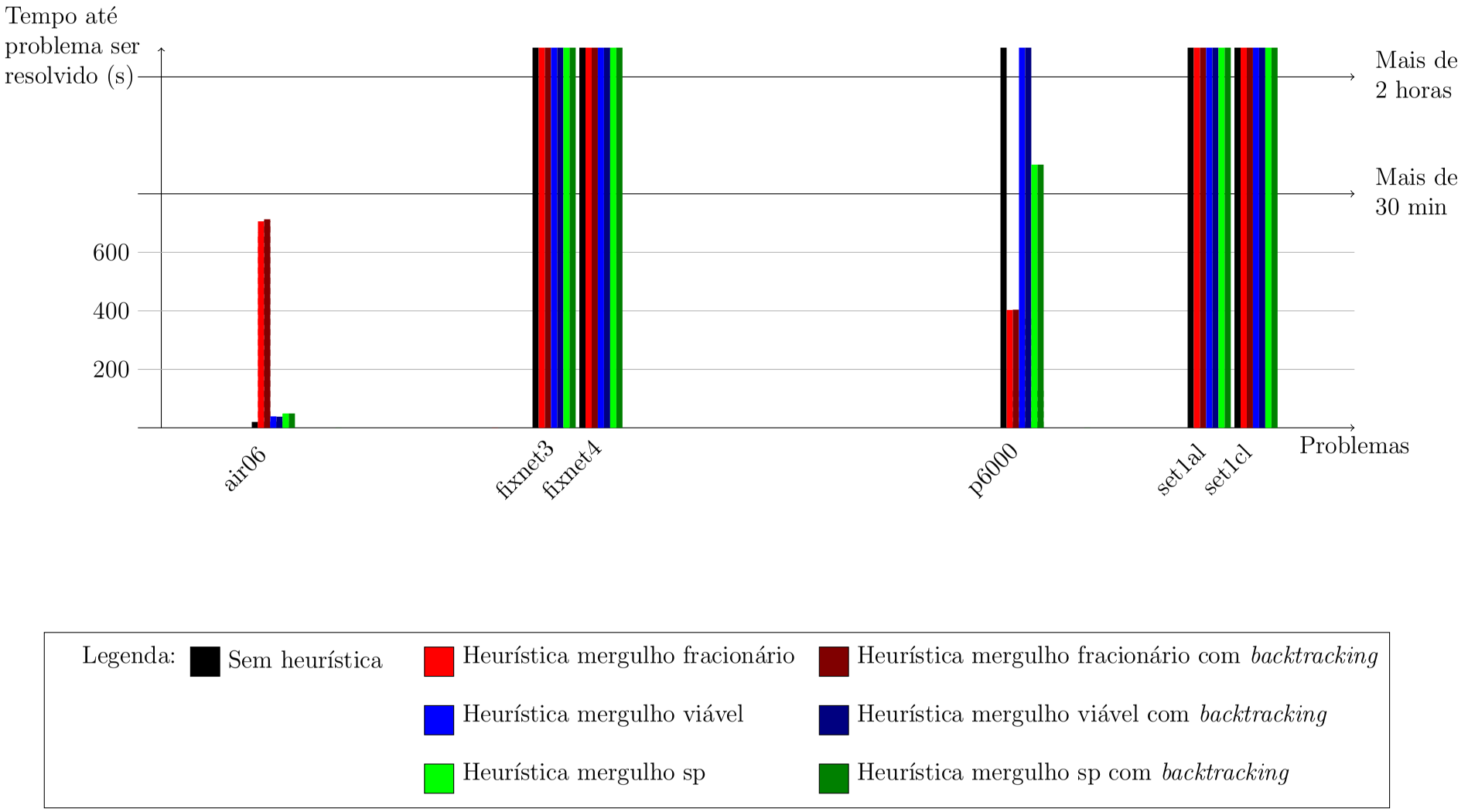


Figura 13: Gráfico comparativo do tempo de execução dos problemas da biblioteca MIPLIB2 que alguma vez não foi resolvido.