Uma Variante do Problema de Corte Unidimensional

Glauber Cintra [1] e Yoshiko Wakabayashi [2]

Instituto de Matemática e Estatística — Universidade de São Paulo
Rua do Matão, 1010 — CEP 05508-900 — São Paulo, SP
E-mail: glauber@ime.usp.br, yw@ime.usp.br

Resumo. Motivados por restrições de ordem prática que ocorrem no processo de corte manual do aço utilizado na indústria de construção civil, estudamos o problema de corte de estoque unidimensional (PCE_1) impondo uma nova restrição: o número de itens distintos que podem ocorrer em um padrão de corte é limitado por uma constante inteira δ. Investigamos esta variante do PCE_1, formulando-a como um problema de programação linear inteira, e propondo um algoritmo híbrido que combina o método de geração de colunas com um algoritmo exato.

Os resultados obtidos ao resolver um expressivo número de instâncias reais e instâncias geradas aleatoriamente indicam um desempenho bastante satisfatório do algoritmo híbrido. Ademais, verificamos que este desempenho praticamente não é afetado, em termos de tempo de execução e qualidade da solução, quando δ assume valores pequenos, como 3 ou 4. Foram obtidas soluções ótimas em praticamente quase todas as instâncias consideradas, dentro de um tempo de execução bastante razoável.

Palavras-chave: problema de corte de estoque unidimensional, geração de colunas, programação linear.

Abstract. In the manual process of cutting steel used in the building industry, some restrictions of practical nature arise naturally. This fact motivated us to study a variant of the One-dimensional Cutting Stock Problem (CSP_1), in which the number of distinct items that may occur in a cutting pattern is bounded by an integer constant δ. We investigated this new variant of CSP_1, modelling it as an integer linear program, and proposing a hybrid algorithm that combines the column generation method with an exact algorithm.

The huge number of practical as well as random instances we solved with our code indicate that its performance is very satisfactory. Moreover, we noted that the performance of the algorithm is practically unaffected, with respect to running time and quality of solution, when δ is very small, like 3 or 4. Optimum solutions were obtained for most of the instances, in a reasonable amount of time.

Keywords: one-dimensional cutting stock problem, column generation, linear programming.

Introdução

O problema de corte de estoque unidimensional (PCE_1) consiste em: dado um objeto, genericamente denominado de *barra*, de comprimento L, e uma lista de m itens, cada item i com comprimento l_i e demanda $d_i \in \mathbb{N}$ $(i = 1, \ldots, m)$, determinar o menor número de barras necessário para atender a demanda, ou seja, produzir d_i itens, cada qual de

[1] Bolsista de Doutorado do CNPq.

[2] Pesquisa parcialmente financiada pelo Projeto Pronex 0107/97, Projeto Temático FAPESP (Proc. 96/4505–2) e Bolsa de Pesquisa do CNPq (Proc. 304527/89-0).

comprimento l_i. Obviamente também estamos interessados em determinar como as barras devem ser cortadas. Cada possível forma de cortar uma barra é chamada de *padrão de corte* (ou simplesmente *padrão*).

O interesse por este problema é em parte explicado por sua aparente simplicidade e grande aplicabilidade prática. Trata-se porém de um problema de natureza complexa, $\mathcal{NP}$-*difícil* [5]. Nas últimas décadas, o PCE$_1$ tem sido largamente estudado por um número crescente de pesquisadores, gerando e tendo se beneficiado de avanços significativos em diversas áreas, tais como *programação linear*, *programação dinâmica*, *teoria da complexidade computacional* e *algoritmos de aproximação*. A pesquisa sobre este assunto deu origem a diversos modelos matemáticos e o desenvolvimento de variadas técnicas para lidar com sua intratabilidade.

Podemos citar inúmeras aplicações práticas para o PCE$_1$, sendo uma delas o corte do aço utilizado nas estruturas de concreto armado. Melhorias no processo de corte pode levar a ganhos substanciais, dependendo da escala de produção, e representar uma vantagem decisiva na competição com outras empresas do setor de construção civil.

Visando conhecer de perto o processo manual de corte do aço, visitamos construtoras e canteiros de obra e conversamos com engenheiros, mestres-de-obra e ferreiros. Percebemos então não ser desejável que a solução do PCE$_1$ apresente padrões com um grande número de itens distintos. No corte manual do aço, os padrões devem conter poucos itens diferentes, digamos no máximo 3 ou 4, pois do contrário os ajustes necessários no equipamento utilizado para o corte tornariam a solução desinteressante e, eventualmente, impraticável.

Resolvemos então estudar o PCE$_1$ impondo uma nova restrição: *o número de itens distintos que podem ocorrer em um padrão é limitado por uma constante inteira* δ. Esta restrição será chamada de *restrição* δ. É fácil perceber que a restrição δ faz com que o número de padrões viáveis seja limitado por $\lfloor \frac{L}{l_{min}} \rfloor^{\delta}$, onde L é o comprimento da barra e l_{min} o comprimento do menor item. Como desejamos que δ assuma valores pequenos, esta restrição limita bastante o número de padrão viáveis.

Podemos então esperar que, com esta restrição, os problemas possam ser resolvidos mais rapidamente, com o ônus de uma queda considerável na qualidade das soluções encontradas. Veremos no entanto que, empiricamente, estas duas expectativas não se confirmam.

A seguir, nas seções 1 e 2 discutimos a estratégia usada para o tratamento do PCE$_1$. Na Seção 3 apresentamos os resultados computacionais obtidos ao utilizar o código que implementamos. Mostramos os resultados obtidos ao resolver 6000 instâncias geradas aleatoriamente e cerca de 200 instâncias reais. Na Seção 4 tecemos algumas considerações finais.

1 Uma Estratégia de Resolução para o PCE$_1$

O PCE$_1$ pode ser formulado como o seguinte problema de programação linear inteira:

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &\text{sujeito a } Ax = d \\ &\qquad x_j \geq 0 \text{ e inteiro} \qquad j = 1, \ldots, n. \end{aligned} \tag{1}$$

Em [3] propusemos um algoritmo que combina o *método de geração de colunas* com um algoritmo de aproximação e um algoritmo exato para resolver o PCE$_1$. Chamamos este algoritmo de HÍBRIDO. Tal algoritmo consiste basicamente em:

1. Resolver a relaxação linear de (1), usando o método simplex revisado, onde a cada iteração, uma nova coluna é gerada resolvendo-se um problema da mochila. Chamamos tal implementação do simplex revisado de SimplexGC e resolvemos o problema da mochila utilizando um algoritmo de branch-and-bound [1], cuja implementação é chamada aqui de MOCHILA.

2. Arredondar para baixo a solução obtida obtendo um problema residual. Se este problema residual for diferente do problema original, voltamos ao passo 1 para resolvê-lo.

3. Tentar resolver o problema residual usando o algoritmo First Fit Decreasing With Backtracking (FFDWB).

4. Se não for possível resolver o problema residual com o FFDWB, utilizar o algoritmo First Fit Decreasing especializado (FFDe) para resolvê-lo.

O detalhamento dos algoritmos HÍBRIDO, SimplexGC, MOCHILA, FFDe e FFDWB pode ser encontrado em [3]. A seguir descrevemos as modificações que devem ser feitas nestes algoritmos de modo a incorporar a restrição δ.

2 Adaptando os Algoritmos

Basicamente precisamos evitar que padrões com mais do que δ itens distintos sejam gerados. Padrões são gerados no passo 1 do algoritmo SimplexGC, e nos algoritmos MOCHILA, FFDe e FFDWB. No passo 1 do algoritmo simplexGC os padrões gerados possuem apenas um item (eles constituem uma solução inicial, que é dada por uma matriz diagonal). Precisamos então adaptar apenas os algoritmos MOCHILA, FFDe e FFDWB, obtendo os algoritmos MOCHILA$_\delta$, FFDe_δ e FFDWB$_\delta$, respectivamente. As adaptações necessárias nos algoritmos SimplexGC e HÍBRIDO se resumem a utilizar os algoritmos MOCHILA$_\delta$, FFDe_δ e FFDWB$_\delta$.

2.1 O Algoritmo MOCHILA$_\delta$

O algoritmo MOCHILA executa uma busca em profundidade em uma árvore, sendo que a solução encontrada é um padrão viável descrito por um ramo da árvore, desde a raiz até uma folha. Cada nível desta árvore está associado a um item, e o valor de cada nó indica quantas vezes o item associado ao nível em que aquele nó se encontra deve ser cortado no padrão.

Portanto, o que precisamos fazer para incorporar a nova restrição ao algoritmo é, em cada ramo da árvore, limitar em δ o número de nós cujo valor é maior que zero. Isto pode ser facilmente implementado modificando os passos 2 e 3.2 do algoritmo MOCHILA. Introduzimos a variável d que representará o número de nós no ramo corrente com valor maior que zero. No passo 2 fazemos $d = 0$ e $j = 0$. A seguir calculamos a primeira solução fazendo, enquanto $j < m$ e $d < \delta$, $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então fazemos $d = d + 1$. Como este laço será repetido somente enquanto $d < \delta$, garantimos que a solução inicial gerada possuirá no máximo δ itens distintos.

No passo 3.2 introduzimos uma modificação similar. Fazemos $j = k + 1$, e se $z_k = 0$ então fazemos $d = d - 1$. De modo a atualizar o número de nós com valor maior que zero, para i variando de $k+1$ até m, se $z_i > 0$ fazemos $d = d - 1$. Calculamos o restante do ramo fazendo, enquanto $j < m$ e $d < \delta$, $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então fazemos $d = d + 1$. O fato deste laço também se repetir somente enquanto $d < \delta$, garante que as soluções geradas pelo algoritmo possuirão no máximo δ itens distintos.

Eventualmente a solução encontrada pelo algoritmo MOCHILA$_\delta$ apresentará compri-

mento não utilizado maior que l_{min}. Tais soluções podem fazer o método de geração de colunas parar antes de encontrar uma solução ótima do problema relaxado. No entanto os testes que apresentaremos na próxima seção indicam que, mesmo para valores pequenos de δ, digamos 3 ou 4, as soluções encontradas pelo algoritmo não comprometem, na maioria dos casos, o método de geração de colunas.

Na fase de testes verificamos que, ao contrário do que poderíamos esperar, a nova restrição imposta ao problema dificulta a resolução do problema da mochila. Isto pode parecer estranho pois a nova restrição limita o espaço de busca ao reduzir o número de padrões viáveis. No entanto, investigando mais profundamente este fenômeno, percebemos que ele pode ser explicado pelo seguinte fato, verificado experimentalmente: ao resolver o problema sem a restrição, apesar da árvore de busca ser maior, a solução é *quase sempre* encontrada nos primeiros ramos percorridos. Por outro lado, ao resolvermos o problema com a nova restrição precisamos percorrer uma parte significativamente maior da árvore.

A seguir apresentamos o algoritmo MOCHILA$_\delta$ que incorpora as modificações explicadas anteriormente.

Algoritmo MOCHILA$_\delta$

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m, \delta)$.
Saída: $z_1, \ldots, z_m \in \mathbb{N}$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$, $\sum_{i=1}^{m} y_i z_i$ é máximo e a cardinalidade do conjunto $\{z_i \mid z_i > 0 \ (i = 1, ..., m)\}$ é menor ou igual a δ.

1 Ordene os itens em ordem decrescente de custo relativo $(\frac{y_i}{l_i})$.
2 Faça $d = 0$ e $j = 0$.
  2.1 Enquanto $j < m$ e $d < \delta$ faça $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
  2.2 Faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
3 Faça $k = max\{i \mid z_i > 0 \ e \ \sum_{j=1}^{i} y_j z_j + \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i} l_j z_j) > M \ (i = m-1, \ldots, 1)\}$.
  3.1 Se não existe tal k então devolva z^* e pare.
  3.2 Caso contrário, faça $z_k = z_k - 1$ e $j = k + 1$. Se $z_k = 0$ faça $d = d - 1$.
    3.2.1 Para $i = k + 1, \ldots, m$ faça se $z_i > 0$ então $d = d - 1$.
    3.2.2 Enquanto $j < m$ e $d < \delta$ faça $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$, $j = j + 1$, e se $z_j > 0$ então faça $d = d + 1$.
4 Se $M \leq \sum_{i=1}^{m} y_i z_i$ faça $z^* = z$ e $M = \sum_{i=1}^{m} y_i z_i^*$.
5 Retorne ao passo 3.

2.2 O Algoritmo SimplexGC$_\delta$

Adaptamos o algoritmo simplexGC simplesmente substituindo a chamada ao algoritmo MOCHILA por uma chamada ao algoritmo MOCHILA$_\delta$.

Algoritmo SimplexGC$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução ótima de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0 \qquad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$ e possui no máximo δ elementos não nulos.

1 Para $i = 1$ até m faça.
1.1 Para $j = 1$ até m se $i = j$ então faça $a_{ij} = \lfloor \frac{L}{l_i} \rfloor$; caso contrário, faça $a_{ij} = 0$.
2 Faça $x_i = \frac{d_i}{a_{ii}}$ $(i = 1, \ldots, m)$.
3 Resolva $yA = c$.
4 Execute o algoritmo MOCHILA$_\delta$ com parâmetros $L, l_1, \ldots, l_m, y_1, \ldots, y_m, \delta$.
5 Se $\sum_{i=1}^{m} y_i z_i \leq 1$, retorne x e pare.
6 Caso contrário, resolva $Ab = z$.
7 Calcule $t = min\{\frac{x_i}{b_i} \mid b_i > 0 \ (i = 1, \ldots, m)\}$ e $s = min\{i \mid \frac{x_i}{b_i} = t \ (i = 1, \ldots, m)\}$.
8 Para $i = 1$ até m faça $a_{is} = z_i$ $(i = 1, \ldots, m)$.
8.1 Se $i = s$ então faça $x_i = t$; caso contrário, faça $x_i = x_i - b_i t$.
9 Retorne ao passo 3.

2.3 O Algoritmo FFDe_δ

Para adaptar o algoritmo FFDe precisamos apenas, ao gerar cada padrão, contar o número de itens utilizados no padrão, restringindo este número a δ. Isto pode ser implementado introduzindo três pequenas alterações nos passos 1, 2.2 e 2.3 do algoritmo. No passo 1 inicializamos com zero a variável d, que representará o número de itens distintos cortados no padrão até o momento. No passo 2.2, além de verificar se existe o j que procuramos no passo 2.1, verificamos também se $d \leq \delta$. Em caso afirmativo fazemos $d = d + 1$. Finalmente, ao terminar de gerar cada padrão, no passo 2.3 fazemos $d = 0$ novamente. A seguir apresentamos o algoritmo FFDe_δ.

Algoritmo FFDe_δ

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução com no máximo δ itens distintos em cada padrão.

1 Ordene $l_1, \ldots, l_m$ em ordem decrescente, faça $k = 1$, $\wp_k = \emptyset$ e $d = 0$.
2 Repita até que $d_i = 0$ $(i = 1, \ldots, m)$.
2.1 Procure $j = min\{h \mid l_h \leq c(\wp_k) \ (h = 1, \ldots, m)\}$.
2.2 Se existir tal j e $d \leq \delta$ então faça $d = d + 1$, $f = min(d_j, \lfloor \frac{c(\wp_k)}{l_j} \rfloor)$ e empacote f vezes o item j em $\wp_k$, fazendo f vezes $\wp_j = \wp_j \cup \{i\}$.
2.3 Caso contrário, faça $d = 0$ e $r_k = min\{\lfloor \frac{d_i}{f_i(\wp_k)} \rfloor, \ i \in \wp_k\}$.
2.3.1 Para todo $i \in \wp_k$ faça $d_i = d_i - f_i(\wp_k) r_k$. Faça $k = k + 1$ e $\wp_k = \emptyset$.
3 Retorne $\wp_1, \ldots, \wp_k$ e $r_1, \ldots, r_k$ e pare.

2.4 O Algoritmo FFDWB$_\delta$

No algoritmo FFDWB os padrões são gerados no procedimento PACKING. Precisamos então efetuar modificações apenas neste procedimento de modo a adequá-lo, e consequentemente também o FFDWB, à restrição δ. A natureza recursiva do procedimento PACKING facilita bastante sua adaptação.

Incluímos na chamada do procedimento os parâmetros δ e d, que representam o número máximo de itens distintos que podem ocorrer em um padrão e o número de itens já cortados no padrão corrente, respectivamente. É suficiente alterar os passos 3.1, 3.4, 5.1 e 5.2.

No passo 3.1, se existir item com demanda maior que zero que caiba em $\wp_k$, ou seja, se existir i' que procuramos no passo 3, verificamos se d é estritamente menor que δ. Em caso afirmativo, podemos empacotar mais itens no padrão; portanto calculamos $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$ e desviamos a execução para o passo 5. Do contrário, apesar de o padrão ainda deixar espaço para mais itens, não podemos aproveitar este espaço para que d não ultrapasse δ. Isto indica que a chamada ao procedimento mostrou-se infrutífera, e por isso desviamos o fluxo de execução para o passo 4.

No passo 3.4, incluímos na chamada ao procedimento PACKING os parâmetros δ e 0, visto que o padrão $\wp_{k+1}$ ainda está vazio. No passo 5.1, fazemos PACKING$_\delta(D, k, f', i', \delta, d)$, se $f' = 0$; caso contrário, executamos a chamada PACKING$_\delta(D, k, f', i', \delta, d+1)$. Se a chamada recursiva feita no passo 5.1 resultar em uma solução, o procedimento pára. Se não, como a chamada recursiva mostrou-se infrutífera, no passo 5.2 atualizamos a demanda fazendo $d_{i'} = d_{i'} + f'$ e removemos de $\wp_k$ as f' ocorrências do item de comprimento $l_{i'}$.

As modificações propostas nos dois parágrafos anteriores são suficientes para adaptar o procedimento PACKING à nova restrição. No entanto, podemos efetuar mais algumas mudanças de modo a melhorar a performance do procedimento.

Na subseção 2.1 comentamos que a dificuldade de resolver o problema da mochila aumenta consideravelmente ao introduzirmos a restrição δ. Podemos então deixar de utilizar o algoritmo MOCHILAe, visto que este algoritmo é utilizado no procedimento PACKING apenas como um dos testes usados na poda da árvore, podendo então ser descartado.

Por outro lado, podemos incluir ainda um novo teste para ser utilizado na poda da árvore. Ao gerarmos um novo padrão, calculamos R, que representa o número de itens remanescentes, ou seja, o número de itens com demanda maior que zero. Para isto, fazemos $R = 0$, no passo 3.2. Em seguida, no passo 3.3, para $i = l, \ldots, m$, fazemos $R = R + 1$ se $d_i > 0$. É necessário que $\delta \geq \lceil \frac{R}{C-k} \rceil$, senão pelo menos um dos padrões que ainda serão gerados terá que possuir mais do que δ itens distintos. Incluímos portanto mais esta condição no passo 3.4.

Como visto anteriormente, uma das condições que deve ser satisfeita para aceitarmos um padrão é $c(\wp_k) \leq \lceil \frac{D_k}{C-k+1} \rceil$. Esta condição determina que o desperdício no padrão gerado ($c(\wp_k)$) não pode exceder o desperdício médio aceitável no restante da solução (incluso o padrão gerado). Ao construir uma solução, ou seja, ao percorrer um ramo da árvore, optamos por começar pelos padrões que apresentam desperdício menor, permitindo um desperdício maior à medida que descemos no ramo da árvore. Em [2] provamos que tal procedimento não nos impede de encontrar uma solução, se ela existir. A seguir apresentamos o procedimento PACKING$_\delta$.

Procedimento PACKING$_\delta(D_k, k, f, i, \delta, d)$

1 Empacote f vezes o item i em $\wp_k$, fazendo f vezes $\wp_k = \wp_k \cup \{i\}$, e faça $d_i = d_i - f$.
2 Procure $j = min\{h \mid d_h > 0 \ (h = 1, \ldots, m)\}$. Se não existir tal j retorne $\wp_1, \ldots, \wp_k$ e pare.
3 Procure $i' = min\{h \mid d_h > 0 \ e \ l_h \leq c(\wp_k) \ (h = i+1, \ldots, m)\}$.
  3.1 Se existir i' então
    3.1.1 Se $d \leq \delta$ faça $f = min(d_{i'}, \lfloor \frac{c(\wp_k)}{l_{i'}} \rfloor)$ e vá para o passo 5; caso contrário, vá para o passo 4.
  3.2 Faça $R = 0$.
  3.3 Para $i = l$ até m se $d_i > 0$ faça $R = R + 1$.
  3.4 Se $k < C$ e $c(\wp_k) \leq \lfloor \frac{D_k}{C-k+1} \rfloor$ e $\delta \geq \lceil \frac{R}{C-k} \rceil$ então execute PACKING$_\delta(D_k - c(\wp_k), k+1, min(d_j, \lfloor \frac{L}{l_j} \rfloor), j, \delta, 0)$.
4 Faça $f = -1$. {Para que o laço no passo seguinte não seja efetuado}
5 Para $f' = f$ até 0 faça:
  5.1 Se $f' = 0$ execute PACKING$_\delta(D_k, k, f', i', \delta, d)$; caso contrário, execute PACKING$_\delta(D, k, f', i', \delta, d+1)$
  5.2 Faça $d_{i'} = d_{i'} + f'$ e remova de $\wp_k$ as f' ocorrências do item i', fazendo f vezes $\wp_k = \wp_k - \{i'\}$.

Para adaptar o algoritmo FFDWB precisamos apenas modificar o passo 3, fazendo uma chamada ao procedimento PACKING$_\delta$ com os parâmetros $D, 1, f', 1, \delta$ e 0. Obtemos assim o algoritmo FFDWB$_\delta$, descrito abaixo.

Algoritmo FFDWB$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, C, \delta)$.
Saída: Se existir, uma solução inteira de valor no máximo C na qual nenhum padrão contém mais do que δ itens distintos. Caso contrário, o valor 0.

1 Coloque os itens em ordem crescente de $min(d_i, \lfloor \frac{L}{l_i} \rfloor)$ $(i = 1, \ldots, m)$.
2 Faça $D = CL - \sum_{i=1}^{m} l_i d_i$ e $f = min(d_1, \lfloor \frac{L}{l_1} \rfloor)$.
3 Para $f' = f - 1$ até 0 execute PACKING$_\delta(D, 1, f', 1, \delta, 0)$.
4 Retorne 0 e pare.

2.5 O Algoritmo HÍBRIDO$_\delta$

O algoritmo HÍBRIDO$_\delta$ é obtido a partir do algoritmo HÍBRIDO substituindo-se as chamadas aos algoritmos SimplexGC, FFDWB e FFDe por chamadas aos algoritmos simplexGC$_\delta$, FFDWB$_\delta$ e FFDe_δ, respectivamente.

Algoritmo HÍBRIDO$_\delta$

Entrada: $(L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta)$.
Saída: Uma solução de: $\min \sum_{j=1}^{n} x_j$
sujeito a $Ax = d$
$x_j \geq 0$ e inteiro $\quad j = 1, \ldots, n,$
onde cada coluna j de A é tal que $\sum_{i=1}^{m} a_{ij} l_i \leq L$ e possui no máximo δ elementos não nulos.

1 Faça $v_{rl} = 0$ e $v_{ip} = 0$.
2 Execute o algoritmo SimplexGC$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$. Se $\sum_{i=1}^{m} x_i > v_{rl}$ então faça $v_{rl} = \sum_{i=1}^{m} x_i$.
3 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$. Faça $v_{ip} = v_{ip} + \sum_{i=1}^{m} x_i^*$
4 Se $x_i^* > 0$ para algum $i = 1, ..., m$ então
  4.1 Retorne A e $x_1^*, \ldots, x_m^*$.
  4.2 Para $i = 1$ até m faça
    4.2.1 Para $j = 1$ até m faça $d_i = d_i - a_{ij} x_j^*$.
  4.3 Faça $m' = 0$.
  4.4 Para $i = 1$ até m faça
    4.3.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$ e $d_{m'} = d_i$.
  4.5 Se $m' = 0$ então pare.
  4.6 Faça $m = m'$ e retorne ao passo 2.
5 Faça $C = \lceil v_{rl} \rceil - v_{ip}$. Execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C, \delta$.
6 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_C$ e pare.
7 Caso contrário, execute o FFDWB$_\delta$ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, C + 1, \delta$.
8 Se o algoritmo FFDWB$_\delta$ não retornou 0 então retorne $\wp_1, \ldots, \wp_{C+1}$ e pare.
9 Caso contrário, execute o algoritmo FFDe_δ com parâmetros $L, l_1, \ldots, l_m, d_1, \ldots, d_m, \delta$, retorne a solução encontrada e pare.

3 Resultados Computacionais

Avaliamos o algoritmo HÍBRIDO$_\delta$ resolvendo 6000 instâncias geradas aleatoriamente e mais cerca de duas centenas de instâncias tiradas das plantas de ferragem de obras de uma construtora. O algoritmo HÍBRIDO foi implementado na linguagem C e os testes executados numa estação Sun Sparc 1000, com dois processadores, clock de 50 mhz e 704 MB de memória principal. Utilizamos o CPLEX 2.0 [4] para resolver os sistemas de equações lineares que aparecem nos passos 3 e 6 do algoritmo SimplexGC$_\delta$.

Dividimos os testes com instâncias aleatórias em dois grupos: 4000 instâncias geradas utilizando o método delineado por Wäscher e Gau em [10], e mais 2000 instâncias geradas aleatoriamente usando parâmetros que julgamos mais próximos dos problemas reais. Na próxima subseção abordamos o primeiro grupo de instâncias, que chamaremos de *instâncias de Wäscher e Gau*, e na subseção seguinte abordamos o segundo grupo de instâncias, que chamaremos de *instâncias estratificadas*.

3.1 Reproduzindo os Testes de Wäscher e Gau

Para gerar várias classes de instâncias aleatórias, variamos três parâmetros: o tamanho do problema, o intervalo de tamanho dos itens e o valor total da demanda.

O tamanho de um problema de corte unidimensional é expresso pelo valor m, que significa o número de itens. Nos testes realizados, m recebeu os valores percebemos, 40 e 50. De acordo com Wäscher e Gau [9], todas as instâncias reais encontradas na literatura possuem tamanho menor que 50, o que justifica limitar o tamanho das instâncias aleatórias a este valor. Utilizamos $L = 10000$ unidades de comprimento.

Os comprimentos dos itens foram modelados como variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo $[1, \frac{\beta L}{100}]$. O tamanho dos itens em relação ao tamanho L da barra afeta significativamente a dificuldade inerente ao problema e consequentemente a performance dos algoritmos propostos para o problema. Variamos o intervalo tomando para β os valores 25, 50, 75 e 100.

Os valores d_i das demandas também foram tratadas como variáveis aleatórias inteiras. Elas foram geradas em duas etapas. Primeiramente calculamos a demanda total T fazendo $T = m\bar{d}$ ($\bar{d} \in I\!N$). Depois distribuímos a demanda total T pelos m itens. Podemos chamar $\bar{d}$ de *fator de multiplicação*. Variando este fator podemos mudar o caráter das instâncias, obtendo instâncias típicas do *bin-packing problem*, ao atribuir a $\bar{d}$ valores pequenos, ou instâncias típicas do problema de corte de estoque, atribuindo a $\bar{d}$ valores grandes. Os valores que utilizamos para $\bar{d}$ foram 10 e 50.

Para distribuir a demanda total T pelos m itens, geramos os números aleatórios $R_1, \ldots, R_m$, uniformemente distribuídos dentro do intervalo [0,1], e então calculamos a demanda de cada item fazendo $d_i = max(1, \lfloor \frac{R_i}{R_1+\ldots+R_m} \rfloor T)$ para $i = 1, \ldots, m-1$ e $d_m = max(1, T - \sum_{i=1}^{m-1} d_i)$.

Combinando os diferentes valores destes três parâmetros definimos 40 classes de instâncias, classes estas caracterizadas pela tripla $(m, \beta, \bar{d})$. Para cada classe, 100 instâncias do problema foram geradas, perfazendo um total de 4000 instâncias. As instâncias foram geradas usando o algoritmo *CUTGEN1*, descrito em [6], e que implementa os procedimentos descritos nos três últimos parágrafos. Tal algoritmo recebe como entrada os parâmetros m, β, $\bar{d}$ e um valor usado para inicializar a semente da função que gera números pseudo-aleatórios. De forma a obter as mesmas instâncias utilizadas por Wäscher e Gau, inicializamos a semente usando o número obtido pela concatenação de m, β (com 3 dígitos) e $\bar{d}$. Por exemplo, ao gerar as instâncias da classe caracterizada por $m = 10$, $\beta = 75$ e $\bar{d} = 10$, o número utilizado para inicializar a semente foi 1007510.

Apesar de o algoritmo FFDWB ser capaz de achar uma solução inteira ótima, se ela existir, os testes computacionais mostraram que em instâncias maiores o tempo de computação requerido pelo FFDWB era inaceitável. Por isso, em nossa implementação, limitamos em 250000 o número de nós que podem ser percorridos pelo FFDWB. Tal procedimento também foi adotado por Wäscher e Gau [10] ao restringir em 25000 o número de retrocessos efetuados pelo algoritmo MTP devido a Martello e Toth [7].

Na Tabela 1 comparamos o tempo médio e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDO$_\delta$ às instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Percebemos que, ao contrário do que poderíamos esperar, o tempo médio, de uma forma geral, tende a aumentar um pouco, à medida que δ diminui. No entanto, a Figura 1 mostra que este aumento no tempo médio é quase (visualmente) imperceptível para as classes nas quais $\beta \geq 75$. Observamos que este aumento no tempo médio é mais notório nas classes onde $\beta = 25$.

Na seção anterior comentamos a explicação para este fenômeno. À medida que δ decresce, o algoritmo MOCHILA$_\delta$ encontra maior dificuldade para achar uma solução, pois torna-se necessário percorrer uma largura muito maior da árvore de busca. Tal árvore, embora fique menor à medida que δ diminui, continua tendo tamanho exponencial (para

$\delta \geq 2$). Portanto a busca acaba por ser mais demorada.

m	Tamanho dos itens	Tempo (seg)				Desperdício (em %)			
		$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
10	0-100%	0,04	0,02	0,04	0,05	13,734	13,734	13,734	13,740
	0-75%	0,05	0,06	0,07	0,08	15,089	15,089	15,089	15,120
	0-50%	0,13	0,10	0,17	0,15	3,098	3,104	3,118	3,324
	0-25%	0,32	0,33	0,39	0,39	1,388	1,428	1,628	2,347
20	0-100%	0,17	0,14	0,18	0,14	9,873	9,873	9,873	9,881
	0-75%	0,24	0,23	0,28	0,34	7,452	7,452	7,452	7,483
	0-50%	0,70	0,89	0,95	0,94	0,698	0,701	0,756	1,007
	0-25%	1,05	1,31	2,52	5,33	0,665	0,732	0,964	1,774
30	0-100%	0,43	0,40	0,51	0,41	10,047	10,047	10,046	10,051
	0-75%	0,69	1,05	0,90	0,83	6,376	6,376	6,381	6,410
	0-50%	2,67	6,55	3,25	3,12	0,362	0,366	0,395	0,717
	0-25%	2,39	3,31	4,57	19,75	0,451	0,487	0,775	1,661
40	0-100%	0,94	0,95	0,93	0,93	9,905	9,904	9,905	9,908
	0-75%	1,79	1,96	1,94	1,83	4,304	4,304	4,306	4,533
	0-50%	6,63	7,08	8,10	7,68	0,210	0,213	0,253	0,565
	0-25%	4,12	5,83	9,29	34,79	0,348	0,374	0,607	1,547
50	0-100%	1,94	1,50	1,93	1,48	7,176	7,177	7,176	7,178
	0-75%	3,52	4,18	4,86	3,42	4,401	4,402	4,407	4,423
	0-50%	16,40	19,25	18,39	16,08	0,157	0,169	0,211	0,524
	0-25%	16,83	11,88	18,64	48,87	0,275	0,299	0,497	1,487

Tabela 1: Desempenho do algoritmo HÍBRIDO$_\delta$ aplicado às instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Em [2] mostramos que o algoritmo HÍBRIDO encontra uma solução cujo valor difere do valor de uma solução inteira ótima de no máximo 1, se verdadeira a conjectura MIRUP [8]. Observe que utilizar o algoritmo HÍBRIDO$_\delta$ com $\delta = m$ é a mesma coisa que utilizar o algoritmo HÍBRIDO. Ainda em [2] verificamos que o algoritmo HÍBRIDO resolveu exatamente 3988 instâncias e nas outras 12 instâncias o erro foi de no máximo 1. Assim, o desperdício verificado quando $\delta = m$, embora pareça grande para algumas categorias de instâncias, é praticamente o menor desperdício possível.

De uma forma geral, o desperdício médio aumentou pouco, à medida que δ diminuiu. Para as classes nas quais $\beta \geq 75$, este aumento foi quase (visualmente) imperceptível, como pode ser verificado pela Figura 2.

3.2 Mais Testes com Instâncias Geradas Aleatoriamente

Para gerar várias classes de instâncias aleatórias, variamos o tamanho do problema e o intervalo de tamanho dos itens. Diferentemente dos testes realizados por Wäscher e Gau, adotamos para m os valores 10, 20, 30, 50 e 100, de forma a analisar o comportamento dos algoritmos propostos para instâncias relativamente grandes.

Utilizamos $L = 10000$ unidades de comprimento. Os comprimentos dos itens foram modelados como variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo $[\frac{\alpha L}{100}, \frac{\beta L}{100}]$. Dessa forma o intervalo é definido pelo par (α, β). Escolhemos como valores

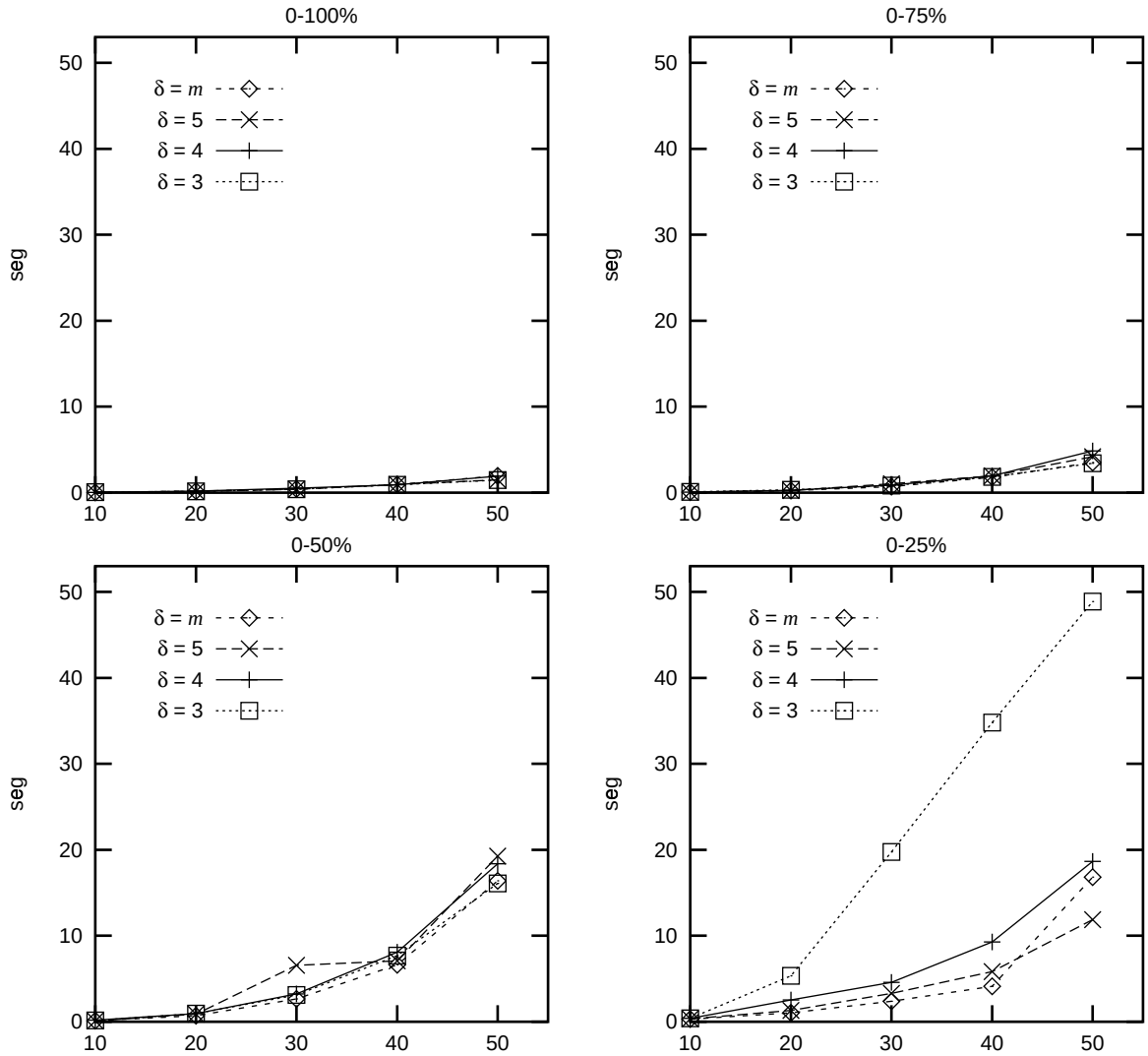

Figura 1: Tempo médio requerido pelo algoritmo HÍBRIDO$_\delta$ para resolver as instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

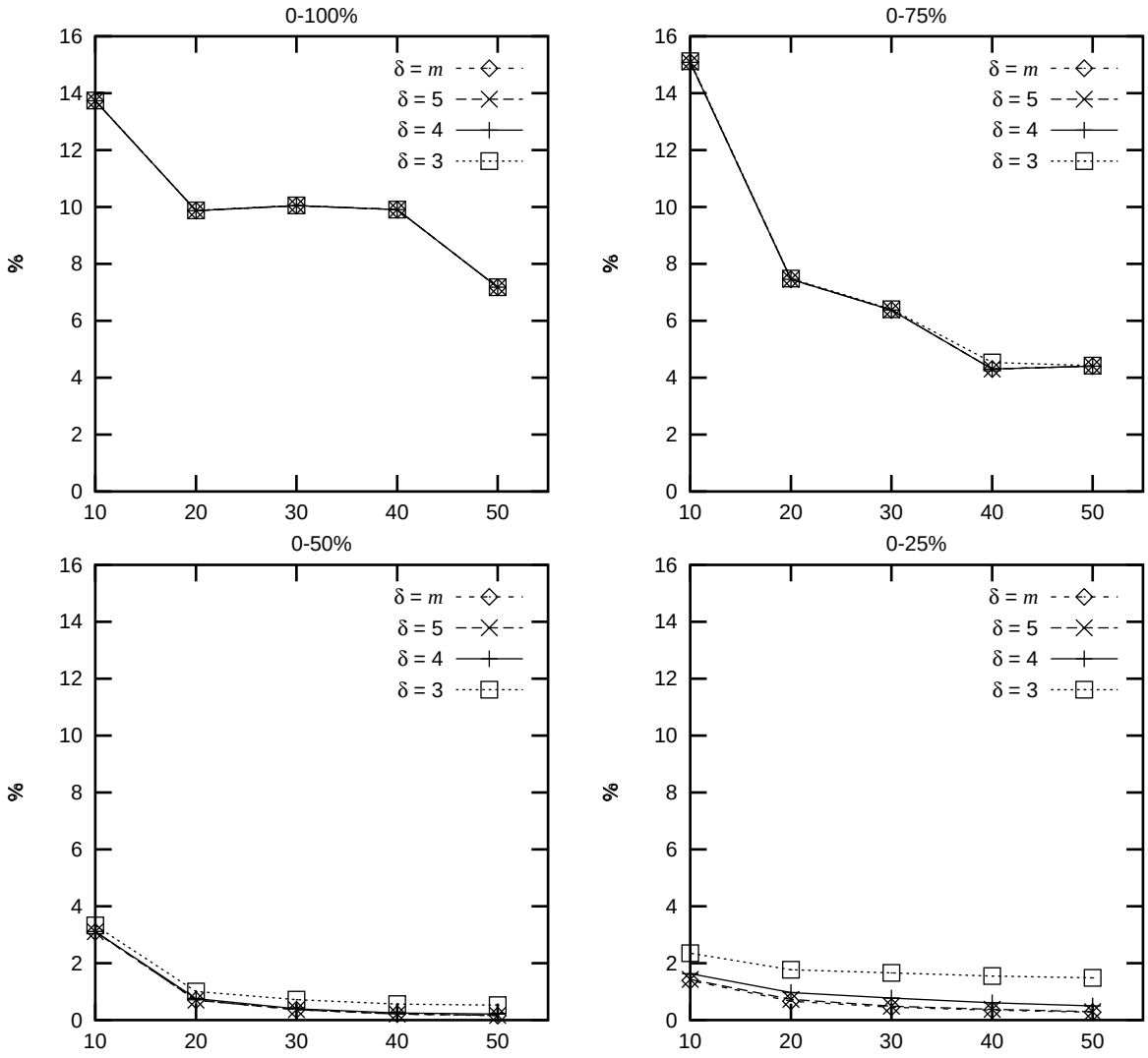

Figura 2: Desperdício médio obtido pelo algoritmo HÍBRIDO$_\delta$ ao resolver as instâncias de Wäscher e Gau, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

de (α, β) os pares (10,70), (20,50), (5,20) e (1,5). Estes quatro pares definem instâncias cujos itens têm comprimentos variados, médios, pequenos e muito pequenos, respectivamente, em relação ao comprimento L. Por este motivo resolvemos chamá-las de *instâncias estratificadas*.

Nossa experiência computacional demonstrou ser quase sempre trivial resolver instâncias nas quais a maioria dos itens são grandes, digamos maiores que $\frac{L}{3}$. Ademais, instâncias com esta característica não são típicas no mundo real. Dessa forma, escolhemos não executar testes com classes de instâncias onde os itens têm apenas tamanhos grandes. Os valores d_i das demandas também são variáveis aleatórias inteiras uniformemente distribuídas dentro do intervalo [1,10000].

Combinando os diferentes valores destes dois parâmetros definimos 20 classes de instâncias, classes estas caracterizadas pelo par $(m, (\alpha, \beta))$. Para cada classe, 100 instâncias do problema foram geradas, perfazendo um total de 2000 instâncias. Utilizamos como gerador de números pseudo-aleatórios a função *drand48*, que faz parte da biblioteca *stdlib.h* da linguagem C. Tal função gera números pseudo-aleatórios de precisão dupla em ponto

flutuante uniformemente distribuídos dentro do intervalo [0,1]. Inicializamos a semente da função *drand48* com o valor 1000.

Na Tabela 2 comparamos o tempo médio e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDO$_\delta$ às instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Verificamos que o tempo tende a aumentar à medida que δ diminui, especialmente para as classes de instâncias onde aparecem itens pequenos ($\alpha \leq 5$). Este aumento no tempo médio requerido pelo algoritmo HÍBRIDO$_\delta$ é mais sensível nas classes onde $\beta \leq 50$. A Figura 3 ilustra o tempo requerido pelo algoritmo HÍBRIDO$_\delta$ para encontrar a solução, em função de m.

m	Tamanho dos itens	Tempo (seg)				Desperdício (em %)			
		$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
10	10%-70%	0,05	0,07	0,04	0,05	11,258	11,258	11,258	11,265
	20%-50%	0,04	0,06	0,04	0,05	5,632	5,632	5,632	5,632
	5%-20%	0,34	0,36	0,37	0,35	0,057	0,057	0,065	0,112
	1%- 5%	0,12	0,16	0,15	0,65	0,034	0,034	0,034	0,042
20	10%-70%	0,23	0,29	0,30	0,25	7,198	7,198	7,198	7,205
	20%-50%	0,26	0,25	0,30	0,28	3,392	3,392	3,392	3,393
	5%-20%	1,72	1,29	1,98	5,85	0,004	0,004	0,005	0,019
	1%- 5%	0,26	0,30	0,53	3,15	0,017	0,017	0,017	0,020
30	10%-70%	0,71	0,87	0,76	0,88	3,992	3,992	3,992	3,995
	20%-50%	0,86	0,89	0,81	0,95	2,247	2,247	2,247	2,249
	5%-20%	2,11	2,62	3,19	25,71	0,003	0,003	0,004	0,010
	1%- 5%	0,53	0,53	0,86	3,39	0,012	0,012	0,012	0,013
50	10%-70%	2,99	3,16	2,50	2,72	2,227	2,227	2,227	2,229
	20%-50%	4,15	4,31	3,75	4,13	1,361	1,361	1,361	1,362
	5%-20%	5,12	6,25	9,84	44,50	0,002	0,002	0,002	0,007
	1%- 5%	2,13	2,35	3,41	9,69	0,007	0,007	0,007	0,010
100	10%-70%	26,17	24,28	25,49	23,01	0,731	0,731	0,731	0,732
	20%-50%	51,74	51,67	58,44	54,70	0,761	0,761	0,761	0,761
	5%-20%	21,89	28,43	64,77	56,92	0,001	0,001	0,002	0,006
	1%- 5%	42,67	19,83	24,22	26,98	0,004	0,004	0,004	0,007

Tabela 2: Desempenho do algoritmo HÍBRIDO$_\delta$ aplicado às instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

O desperdício médio quase não foi alterado à medida que δ diminuiu, como pode ser verificado pela Figura 4. Vale salientar que o desperdício médio obtido com $\delta = m$ é praticamente o menor desperdício possível.

3.3 Resultados dos Testes com Instâncias Reais

Apresentamos nesta subseção os resultados obtidos ao aplicar o algoritmo HÍBRIDO$_\delta$ a cerca de duas centenas de instâncias tiradas das plantas de ferragem de duas obras de uma construtora com atuação em várias cidades do país. Tratam-se de edifícios gêmeos de 9 andares, construídos lado a lado. As instâncias consideradas são relativas ao problema de determinar como cortar o aço a ser utilizado nas estruturas de concreto armado de modo a minimizar o desperdício de aço.

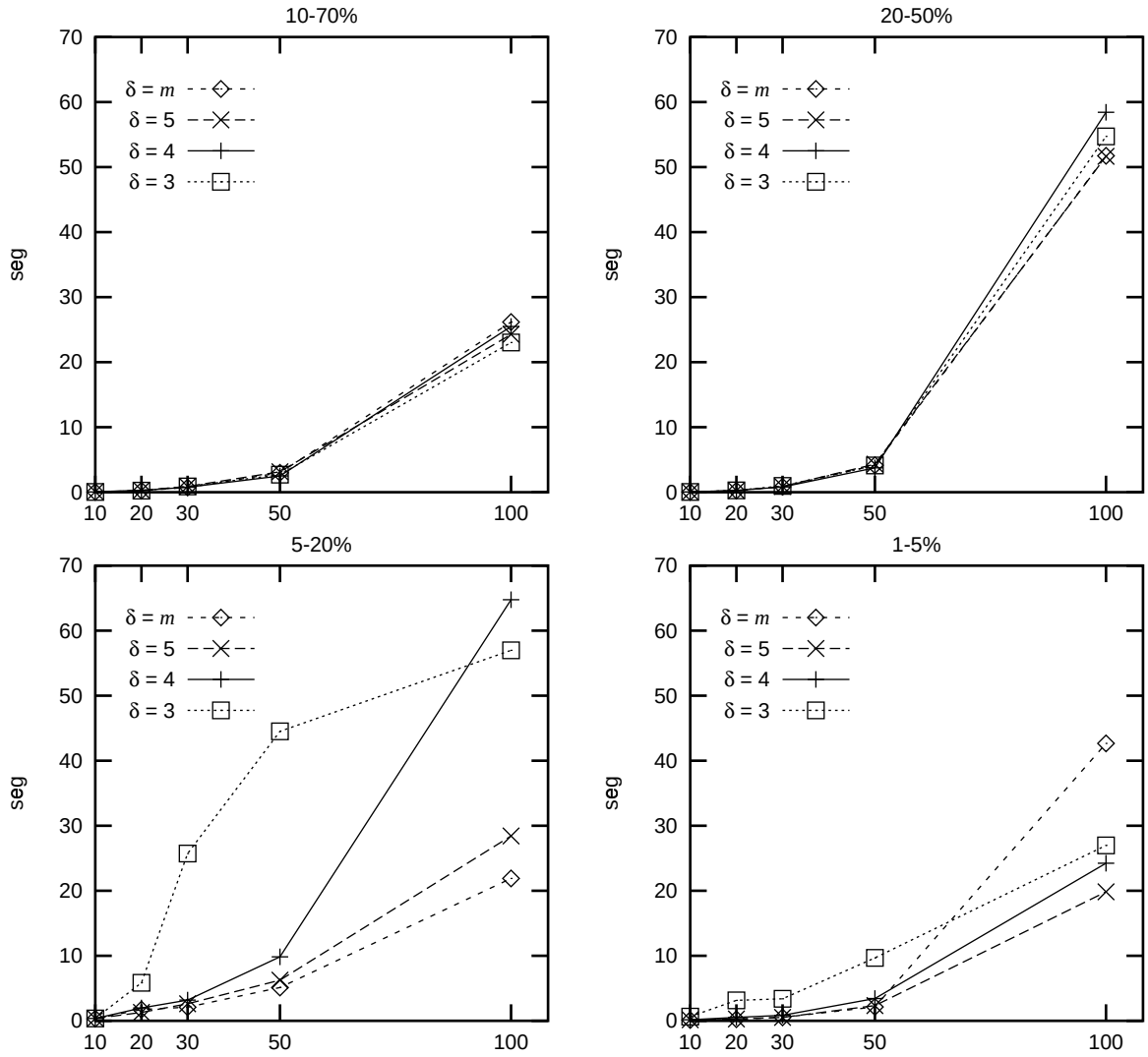

Figura 3: Tempo médio requerido pelo algoritmo HÍBRIDO$_\delta$ para resolver as instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

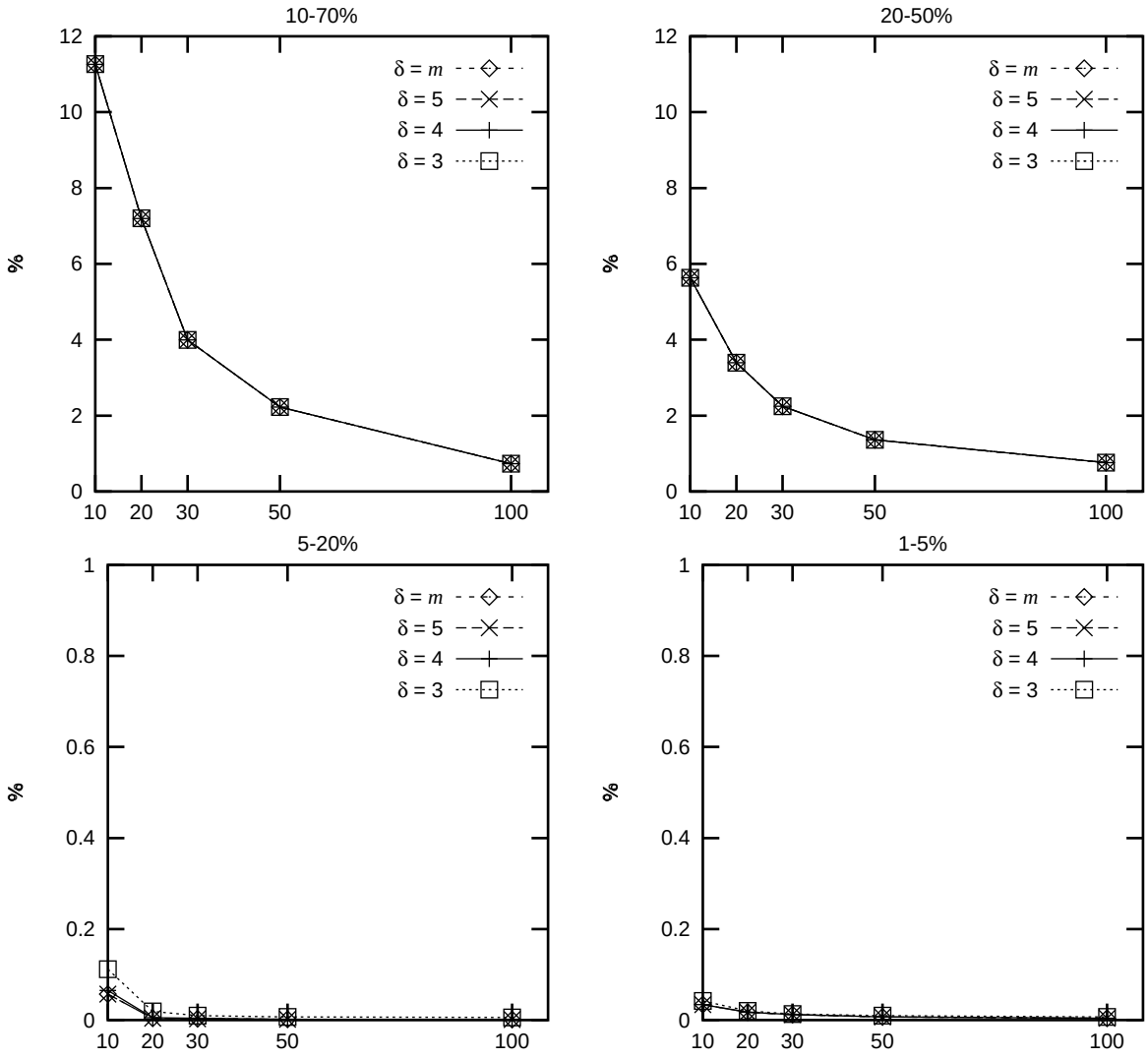

Figura 4: Desperdício médio obtido pelo algoritmo HÍBRIDO$_\delta$ ao resolver as instâncias estratificadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Nestas obras foram utilizados aço CA-60B com 5mm de bitola, e aço CA-50A com bitolas 6,3mm, 8mm, 10mm, 12,5mm, 16mm e 20mm. Consideramos então 7 tipos de barras de aço, que deram origem a 7 categorias de instâncias. Obtivemos 209 instâncias, que chamaremos de *instâncias reais*, agrupando a demanda de cada um destes 7 tipos de barras em cada uma das 42 plantas de ferragem analisadas.

Na Tabela 3 comparamos o tempo médio e o desperdício médio verificado ao aplicar o algoritmo HÍBRIDO$_\delta$ às instâncias reais com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

As instâncias reais foram resolvidas em tempo muito curto, de forma que devido à imprecisão na medição do tempo, fica difícil analisar o que acontece com o tempo à medida que δ diminui. Cabe ressaltar que o algoritmo HÍBRIDO$_\delta$ encontrou soluções ótimas para todas as instâncias quando foi utilizado $\delta = m$. Assim o desperdício médio verificado com $\delta = m$ é o menor possível. Observamos ainda que o desperdício médio permaneceu inalterado ou foi levemente superior, à medida que δ diminuiu.

Com o intuito de diminuir o desperdício, experimentamos agrupar todas as instâncias de cada categoria gerando apenas 7 instâncias, que chamaremos de *instâncias reais agrupadas*. Com isto obtivemos instâncias maiores, esperando obter soluções com menor desperdício.

Bitola	m médio	Tempo (seg)				Desperdício (em %)			
		$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
5.0	7	0,23	0,31	0,31	0,08	0,324	0,324	0,324	0,324
6.3	11	0,30	0,33	0,44	0,30	1,438	1,453	1,472	1,605
8.0	12	0,35	0,33	0,42	0,42	6,281	6,281	6,281	6,438
10.0	20	0,94	0,89	1,06	0,71	7,843	7,843	7,843	7,939
12.5	14	0,53	0,33	0,56	0,47	5,467	5,467	5,467	5,566
16.0	13	0,35	0,23	0,23	0,42	20,675	20,675	20,675	20,675
20.0	8	0,12	0,15	0,19	0,19	12,023	12,023	12,023	12,190

Tabela 3: Desempenho do algoritmo HÍBRIDO$_\delta$ aplicado às instâncias reais, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Na Tabela 4 comparamos o tempo e o desperdício verificado ao aplicar o algoritmo HÍBRIDO$_\delta$ às instâncias reais agrupadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

Observamos que o tempo de resolução não apresentou grandes alterações ao variarmos δ. Novamente o algoritmo HÍBRIDO$_\delta$ encontrou soluções ótimas para todas as instâncias quando foi utilizado $\delta = m$. Assim o desperdício médio verificado com $\delta = m$ é o menor possível. O desperdício médio apresentou alterações inferiores a 0,01% à medida que δ diminuiu.

Bitola	m	Tempo (seg)				Desperdício (em %)			
		$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$	$\delta = m$	$\delta = 5$	$\delta = 4$	$\delta = 3$
5.0	48	42	41	46	51	0,020	0,020	0,020	0,058
6.3	209	40	36	53	64	0,017	0,017	0,017	0,017
8.0	264	200	195	297	166	0,009	0,009	0,019	0,068
10.0	365	1601	1752	1586	1450	0,061	0,061	0,061	0,061
12.5	301	484	658	840	685	0,268	0,268	0,268	0,268
16.0	218	169	165	176	148	4,995	4,995	4,995	4,995
20.0	105	32	25	28	32	2,507	2,507	2,507	2,507

Tabela 4: Desempenho do algoritmo HÍBRIDO$_\delta$ aplicado às instâncias reais agrupadas, com $\delta = m$, $\delta = 5$, $\delta = 4$ e $\delta = 3$.

4 Considerações Finais

Vimos que o PCE$_1$ pode ser aplicado na modelagem e otimização do processo de corte manual do aço utilizado na indústria da construção civil. No entanto a solução obtida para o PCE$_1$ não deve apresentar padrões de corte com muitos itens distintos, do contrário tal solução exigiria uma quantidade muito grande de ajustes no equipamento usado para o corte, o que tornaria a solução desinteressante, em termos práticos. Estudamos então o PCE$_1$ impondo a restrição de que o número de itens distintos que podem ocorrer em um padrão é limitado por uma constante inteira δ

Poderíamos esperar que qualidade das soluções obtidas ao adotar esta restrição fosse sensivelmente inferior. No entanto, ao resolver um significativo número de instâncias ge-

radas aleatoriamente e instâncias reais, verificamos que a qualidade das soluções obtidas praticamente não se alterou, mesmo quando utilizamos valores pequenos para δ.

Finalmente, o algoritmo HIBRIDO$_\delta$ aqui delineado, mostrou ser uma ferramenta eficaz na resolução de instâncias de pequeno e médio porte. As soluções obtidas apresentaram desperdício muito próximo do menor desperdício possível e o tempo requerido foi tipicamente da ordem de segundos.

Referências

[1] V. Chvátal, *Linear Programming*, W. H. Freeman and Company, New York, 1980.

[2] G. F. Cintra, *Algoritmos híbridos para problemas de corte unidimensional*, master's thesis, Instituto de Matemática e Estatística, São Paulo, 1998.

[3] G. F. Cintra and Y. Wakabayashi, *Um algoritmo híbrido para o problema de corte unidimensional*, in XXX Simpósio Brasileiro de Pesquisa Operacional, Curitiba, 1998.

[4] CPLEX, *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, CPLEX Optimization, Inc, 1995.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*, W. H. Freeman and Co., San Fransisco, 1979.

[6] T. Gau and G. Wäsher, *CUTGEN1: A problem generator for the standard one-dimensional cutting stock problem*, European Journal of Operations Research, 84 (1995), pp. 572–579.

[7] S. Martello and P. Toth, *Knapsack problems*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd., Chichester, 1990. Algorithms and computer implementations.

[8] G. Scheithauer and J. Terno, *The modified integer round-up property of the one-dimensional cutting stock problem*, European Journal of Operations Research, 84 (1995), pp. 562–571.

[9] G. Wäscher and T. Gau, *Test data for the one-dimensional cutting stock problem*, Working Paper, Technische Universitaet Braunschweig, (1993).

[10] ——, *Heuristics for the integer one-dimensional cutting stock problem: a computational study*, OR Spektrum, 18 (1996), pp. 131–144.