O Problema de Corte de Guilhotina com Placas Variadas

Glauber F. Cintra [1]

Resumo: No problema de corte de guilhotina bidimensional com placas variadas (CGV) temos uma lista de placas retangulares com dimensões e custos variados e uma lista de itens retangulares que devem ser produzidos em quantidades especificadas. Desejamos determinar como cortar as placas, utilizando apenas cortes de guilhotina, de modo a produzir os itens solicitados, de tal forma que a soma dos custos das placas utilizadas seja o menor possível. Embora o CGV apareça com frequência em diversas situações práticas, existe bem pouca literatura a seu respeito.

Neste artigo mostramos como modelar o CGV como sendo um problema de programação linear inteira e como achar soluções *quase* ótimas para o modelo utilizando programação dinâmica e o método de geração de colunas. Investigamos também a variante do CGV onde os itens podem sofrer rotações ortogonais (CGV^r). Finalmente, introduzimos a idéia de "perturbar" as instâncias residuais, obtendo soluções ainda mais próximas do ótimo.

Os resultados computacionais obtidos ao resolver diversas instâncias do CGV e do CGV^r são analisados e indicam que os algoritmos que propomos apresentam um bom desempenho, em termos de tempo e de qualidade das soluções encontradas para as instâncias de teste utilizadas. Tais evidências empíricas mostram que estes algoritmos parecem ser apropriados para resolver instâncias associadas a situações reais.

Palavras-chave: corte de guilhotina, geração de colunas, programação dinâmica.

Abstract: In the two-dimensional cutting stock problem with multiple stock sizes (CGV) we have a list of rectangular bins with varied costs and sizes and a list of rectangular items that would be produced in specified quantities. We want to determine how to cut the bins, using only guillotine cuts, in order to produce the items requested, minimizing the sum of the costs of the used bins. Although the CGV appears frequently in many practical situations, there are very little amount of literature about it.

In this article we show how to formulate the CGV as a integer linear program and how to find *quasi-optimal* solutions for the model using dynamic programming and the column generation method. We also investigate the variant of the CGV in which the items may be rotated orthogonally (CGV^r). Finally, we introduce the idea of "disturbing" the residual instances, obtaining solutions that are closer to the optimal solutions.

The computational results obtained in solving various instances of the CGV and the CGV^r are analyzed and indicate that the algorithms that we proposed have a good performance, in terms of time and quality of the solutions for the used instances. Such empirical evidences indicate that these algorithms seem to be appropriate for solving real-world instances.

Keywords: guillotine cut, column generation, dynamic programming.

[1]Professor do Centro Federal de Educação Tecnológica do Ceará — glauberc@cefetce.br

Introdução

Existe uma grande diversidade de situações em que nos deparamos com o seguinte desafio: precisamos cortar objetos grandes, produzindo objetos menores, de forma a obter ganhos em termos econômicos. Em tais situações estão envolvidos problemas de corte. Frequentemente, os objetos grandes e os objetos pequenos têm apenas duas dimensões relevantes e possuem a forma retangular. Neste caso, chamaremos os objetos grandes de *placas* e o objetos menores de *itens*. Além disso, é comum a restrição de que os cortes feitos nas placas têm que ser paralelos a um de seus lados e se estender desde um lado da placa até o lado oposto. Chamamos este tipo de corte de *corte de guilhotina*. Os problemas de corte que envolvem essas restrições são chamados genericamente de problemas de corte de guilhotina bidimensional.

O interesse por tais problemas de corte é em parte explicado por sua grande aplicabilidade prática, especialmente nas indústrias. Pequenas melhorias nos processos que envolvem corte podem levar a ganhos substanciais, dependendo da escala de produção, e representar uma vantagem decisiva na competição com outras empresas do setor. Como exemplos de aplicações práticas de problemas de corte podemos citar os processos industriais envolvendo corte de chapas de metal e madeira, lâminas de vidro e fibra de vidro, peças de couro e carpete. Esses processos ocorrem na indústria metalúrgica, moveleira, vidraceira, na indústria da moda etc.

Abordaremos a variante do problema de corte de guilhotina bidimensional em que as placas possuem dimensões e custos variados e que chamaremos de *problema de corte de guilhotina bidimensional com placas variadas*, ou simplesmente CGV. Tal problema surge em diversos processos de corte de guilhotina, onde sobras de cortes anteriores devem ser reaproveitados, ou ainda em situações em que o material a ser cortado é fornecido em diversas medidas.

Os problemas de corte costumam ser fáceis de entender e formular. No entanto, sua aparente simplicidade costuma esconder sua natureza complexa, em termos computacionais. Queremos dizer com isto que a maioria dos problemas de corte abordados na literatura não podem ser resolvidos por algoritmos polinomiais, supondo P $\neq$ NP. Este também é o caso do CGV. A exata classe de complexidade computacional do CGV é desconhecida, no entanto sabe-se que o CGV é pelo menos tão difícil quanto os problemas NP-completos [Cin04]. De fato, Nelißen sustenta que os problemas de corte, em sua versão de decisão, não pertencem à classe NP mas a alguma classe abaixo de EXP-ESPAÇO [Nel93].

Ao contrário de outras variantes do problema de corte de guilhotina, existe bem pouca literatura a respeito do CGV. Podemos citar, entretanto, a heurística proposta por Yanasse, Zinober e Harris [YZH91] que resolve instâncias do CGV onde o objetivo é minimizar as sobras nas placas utilizadas[2]. Este objetivo é equivalente a minimizar a soma dos custos das placas utilizadas se considerarmos que o custo de cada placa é linearmente proporcional à sua área.

Este artigo está organizado da seguinte maneira. Na Seção 1 definimos formalmente o CGV e mostramos como formulá-lo como um problema de programação linear inteira (PLI). Em seguida, na Seção 2, discutimos como aplicar método de geração de colunas para resolver a relaxação linear do PLI correspondente ao CGV, indicando como usar programação dinâmica para gerar novas colunas. Na Seção 3, descrevemos como, a partir de uma solução fracionária do problema relaxado, podemos obter uma solução inteira sem que haja excesso de produção, e cujo valor se aproxime do ótimo. Na Seção 4 introduzimos a idéia de "perturbar" as instâncias residuais e mostramos como adaptar os algoritmos vistos anteriormente para a variante do CGV em que os itens podem sofrer rotações ortogonais. Na Seção 5 apresentamos os resultados computacionais obtidos e na última seção tecemos algumas considerações a respeito deste trabalho.

[2]De fato, o problema estudado por esses autores inclui duas restrições adicionais: a quantidade de cortes de guilhotina no primeiro estágio é limitada e as bordas da placa devem ser aparadas, pois durante o processo de corte elas são usadas para fixação da placa.

1 Formulação do Problema

Formalmente, o CGV consiste em: dada uma lista de k tipos de placas, onde cada placa do tipo j tem largura L_j, altura A_j e custo C_j, e uma lista de m itens, cada item i com largura l_i, altura a_i e demanda d_i, determinar como cortar as placas de modo a produzir d_i unidades de cada item i, utilizando apenas cortes de guilhotina, de forma que a soma dos custos das placas utilizadas seja a menor possível[3].

Podemos formular o CGV como um problema de programação linear inteira (PLI). Para fazer isso, representamos cada padrão de corte[4] j por um vetor p_j, cujo i-ésimo elemento indica o número de vezes que o item i ocorre nesse padrão. O problema agora consiste em considerar os padrões viáveis e decidir quantas vezes cada padrão deve ser utilizado de modo a atender a demanda, minimizando a soma dos custos das placas utilizadas. Supondo existir n padrões viáveis, introduzimos um vetor x cujos elementos são inteiros x_j $(j = 1, \ldots, n)$, onde x_j indica quantas placas devem ser cortadas de acordo com o padrão j.

Além disso, para cada padrão j, seja c_j o custo da placa associada a este padrão. Se tal placa é do tipo i, então $c_j = C_i$. Assim, denotando por P a matriz $m \times n$ cujas colunas são os vetores $p_1, \ldots, p_n$, representando por d o vetor das demandas e por c o vetor cujas entradas são $c_1, \ldots, c_n$, o problema pode ser assim formulado:

$$\begin{aligned} &\text{minimize} \quad c^T x \\ &Px = d \\ &x_j \geq 0 \quad \text{e inteiro} \qquad j = 1, \ldots, n. \end{aligned} \tag{1}$$

A relaxação linear de (1) é:

$$\begin{aligned} &\text{minimize} \quad c^T x \\ &Px = d \\ &x_j \geq 0 \qquad j = 1, \ldots, n. \end{aligned} \tag{2}$$

É fácil mostrar que a quantidade de padrões viáveis (n) pode ser muito grande, mesmo em instâncias de pequeno porte. Em tais casos, armazenar a matriz P é inviável, em termos práticos. Para resolver esta dificuldade, podemos utilizar o método de geração de colunas, descrito na próxima seção.

2 Resolvendo a Relaxação Linear

Podemos resolver (2) utilizando o conhecido método de geração de colunas, proposto por Gilmore e Gomory [GG61, GG63]. Para gerar novas colunas, podemos utilizar o algoritmo DP. Tal algoritmo, baseado em programação dinâmica, calcula a fórmula de recorrência proposta por Beasley [Bea85], que resolve o problema que chamaremos de Problema da Mochila Retangular e que consiste em: dada uma placa de largura L e altura A, e uma lista de m itens, cada item i com largura l_i, altura a_i e valor v_i, determinar um padrão guilhotinável[5] tal que a soma dos valores dos itens contidos no padrão seja a maior possível.

Para descrever a fórmula de recorrência proposta por Beasley, é conveniente adotar mais algumas notações. Denotaremos por $v(l, a)$ o valor de um item mais valioso que cabe numa placa de dimensões (l, a), ou 0 se nenhum item cabe na placa. Mais formalmente,

[3] Obviamente, para que uma instância do CGV tenha solução é preciso que para todo item i $(i = 1, \ldots, m)$ exista um tipo de placa j tal que $l_i \leq L_j$ e $a_i \leq A_j$.

[4] Chamamos de padrão de corte, ou simplesmente padrão, cada possível forma de cortar uma placa.

[5] ou seja, um padrão que pode ser obtido com uma sequência de cortes de guilhotina.

$$v(l,a) = max(\{v_i \mid 1 \leq i \leq m,\ \ l_i \leq l,\ a_i \leq a\} \cup \{0\}).$$

Chamamos de *ponto de discretização da largura* [Her72] um valor $i \leq L$ que pode ser obtido através de uma combinação cônica inteira[6] de $l_1, \ldots, l_m$. Os *pontos de discretização da altura* são definidos de forma análoga. Em [CW04] são apresentados dois algoritmos para calcular pontos de discretização. O primeiro utiliza a técnica de enumeração conhecida como *branch-and-bound*, enquanto que o segundo é baseado em programação dinâmica. Vamos denotar por $\mathcal{L}$ e por $\mathcal{A}$ o conjunto dos pontos de discretização da largura e da altura, respectivamente.

Denotamos por $p(l)$ o maior ponto de discretização da largura que é menor ou igual a l. Analogamente, denotamos por $q(a)$ o maior ponto de discretização da altura que é menor ou igual a a. Finalmente, denotamos por $O(l,a)$ o valor ótimo de um padrão guilhotinável numa placa de dimensões $(l,\ a)$[7]. Podemos calcular $O(l,a)$ usando a seguinte fórmula de recorrência:

$$O(l,a) = max(v(l,a), \{O(l',a)+O(p(l-l'),a) \mid l' \in \mathcal{L}\}, \{O(l,a')+O(l,q(a-a')) \mid a' \in \mathcal{A}\}). \quad (3)$$

A seguir apresentamos o algoritmo DP, que calcula (3). Em nossa implementação do algoritmo DP, os pontos de discretização são calculados utilizando-se programação dinâmica. Uma explicação mais detalhada desse algoritmo pode ser obtida em [CW04].

Algoritmo 1: DP

Entrada: Uma instância $I = (L, A, l, a, v)$ do Problema da Mochila Retangular, onde $l = (l_1, \ldots, l_m)$, $a = (a_1, \ldots, a_m)$ e $v = (v_1, \ldots, v_m)$.
Saída: Uma solução ótima de I.

1 Calcule $p_1 < \ldots < p_r$, os pontos de discretização da largura L.
2 Calcule $q_1 < \ldots < q_s$, os pontos de discretização da altura A.
3 Para $i = 1$ até r
  3.1 Para $j = 1$ até s
    3.1.1 $O(i,j) = max(\{v_k \mid 1 \leq k \leq m,\ l_k \leq p_i \text{ e } a_k \leq q_j\} \cup \{0\})$.
    3.1.2 $item(i,j) = max(\{k \mid 1 \leq k \leq m,\ \ l_k \leq p_i,\ \ a_k \leq q_j \text{ e } v_k = O(i,j)\} \cup \{0\})$.
    3.1.3 $guilhotina(i,j) = nil$.
4 Para $i = 2$ até r
  4.1 Para $j = 2$ até s
    4.1.1 $n = max(k \mid 1 \leq k \leq r \text{ e } p_k \leq \lfloor \frac{p_i}{2} \rfloor)$.
    4.1.2 Para $x = 2$ até n
      4.1.2.1 $t = max(k \mid 1 \leq k \leq r \text{ e } p_k \leq p_i - p_x)$.
      4.1.2.2 Se $O(i,j) < O(x,j) + O(t,j)$.
        4.1.2.2.1 $O(i,j) = O(x,j) + O(t,j)$, $posicao(i,j) = p_x$ e $guilhotina(i,j) =$ 'V'.
    4.1.3 $n = max(k \mid 1 \leq k \leq s \text{ e } q_k \leq \lfloor \frac{q_j}{2} \rfloor)$.
    4.1.4 Para $y = 2$ até n
      4.1.4.1 $t = max(k \mid 1 \leq k \leq s \text{ e } q_k \leq q_j - q_y)$.
      4.1.4.2 Se $O(i,j) < O(i,y) + O(i,t)$
        4.1.4.2.1 $O(i,j) = O(i,y) + O(i,t)$, $posicao(i,j) = q_y$ e $guilhotina(i,j) =$ 'H'.

As matrizes *item*, *guilhotina* e *posicao*, calculadas pelo algoritmo DP, nos permitem reconstruir o padrão correspondente à solução ótima encontrada. Fica implícito na descrição do algoritmo que esta informação deve ser devolvida, pois também estamos interessados em saber

[6] Uma combinação cônica inteira é uma combinação linear onde todos os coeficientes são números inteiros não-negativos.

[7] Obviamente, $O(l,0) = O(0,a) = 0$.

como cortar a placa. É possível mostrar que para instâncias do Problema da Mochila Retangular onde os itens não são "muito pequenos"[8] em relação ao tamanho das placas, o algoritmo DP requer tempo polinomial [Cin04].

Agora que vimos como gerar novas colunas, podemos descrever o algoritmo que resolve (2), e que chamaremos de SimplexGC. Na descrição do SimplexGC, o vetor b indica a placa associada a cada coluna da matriz B. Dessa forma, as colunas de B, junto com o vetor b e os vetores *guilhotina* e *posicao*, calculados pelo DP, permitem construir os padrões que constituem a solução do SimplexGC. Observe que em cada iteração do SimplexGC precisamos gerar um padrão cujo valor[9] seja maior que o custo da placa associada ao padrão. É possível gerar um tal padrão, se existir, fazendo no máximo k chamadas ao algoritmo DP. Em cada chamada utilizamos as dimensões de um dos k tipos de placas.

Algoritmo 2: SimplexGC

Entrada: $I = (L, A, l, a, d)$ onde $L = (L_1, \ldots, L_k)$, $A = (A_1, \ldots, A_k)$, $l = (l_1, \ldots, l_m)$, $a = (a_1, \ldots, a_m)$ e $d = (d_1, \ldots, d_m)$.

Saída: Uma solução ótima de (2): minimize $c^T x$

$$Px = d$$
$$x_j \geq 0 \qquad j = 1, \ldots, n.$$

1 Para $i = 1$ até m faça
  1.1 $x_i = d_i$, $b_i = min(h \mid l_i \leq L_h \text{ e } a_i \leq A_h)$ e $c_i = C_{b_i}$.
2 Seja B a matriz identidade de ordem m.
3 Resolva $y^T B = c^T$.
4 Para $i = 1$ até k faça
  4.1 Gere uma nova coluna z executando o algoritmo DP com parâmetros L_i, A_i, l, a, y.
  4.2 Se $y^T z > C_i$ vá para o passo 6.
5 Devolva B, b e x e pare (tal x corresponde apenas às colunas de B).
6 Resolva $Bw = z$.
7 Calcule $t = min(\frac{x_j}{w_j} \mid 1 \leq j \leq m,\ w_j > 0)$.
8 Calcule $s = min(j \mid 1 \leq j \leq m,\ \frac{x_j}{w_j} = t)$.
9 $b_s = i$ e $c_s = C_i$.
10 Para $i = 1$ até m faça
  10.1 $B_{i,s} = z_i$.
  10.2 Se $i = s$ então $x_i = t$; caso contrário, $x_i = x_i - w_i t$.
11 Retorne ao passo 3.

Dependendo dos dados de entrada, com alguns cuidados na implementação podemos obter um ganho substancial no tempo requerido pelo SimplexGC. Sejam L_{max} e A_{max} respectivamente a maior largura e a maior altura dentre todos os tipos de placas. Uma idéia que poderia ser usada é a de chamar o algoritmo DP apenas para uma placa (possivelmente fictícia) de dimensões (L_{max}, A_{max}). As matrizes O, *guilhotina* e *posicao*, calculadas pelo algoritmo, vão conter todas as soluções que seriam obtidas pelo DP se realizássemos k chamadas ao algoritmo utilizando as dimensões de cada um dos k tipos de placas.

A idéia descrita no parágrafo anterior nem sempre leva a uma diminuição no tempo gasto pelo algoritmo. Se houver grande diferença entre a largura e a altura de alguns dos tipos de placas, a chamada ao algoritmo DP com as dimensões L_{max} e A_{max} pode requerer mais tempo do

[8] Mais precisamente, consideramos que os itens não são "muito pequenos" se $l_i > \frac{L}{k}$ e $a_i > \frac{A}{k}$ (k fixo e $i = 1, \ldots, m$).

[9] O valor do padrão é a soma dos valores de cada um dos itens contidos no padrão, sendo que o valor de cada item é dado pelas variáveis duais de (2).

que k chamadas com as dimensões dos k tipos de placas. No entanto, é fato que, numa iteração do SimplexGC, podemos aproveitar as informações obtidas nas chamadas anteriores feitas ao DP, naquela iteração, para diminuir o tempo gasto numa nova chamada.

No SimplexGC utilizamos o primeiro padrão gerado pelo DP cujo valor é maior que o custo da placa associada ao padrão. Alternativamente, poderíamos executar todas as iterações do passo 4, e dentre os k padrões gerados pelo DP escolher aquele padrão cujo valor fosse máximo. Com isto, seria esperada uma diminuição no número de iterações do SimplexGC, pois o valor da função objetivo tenderia a convergir mais rapidamente para o valor ótimo. Por outro lado, o tempo gasto em cada iteração aumentaria. Seria interessante implementar essa idéia alternativa, o que corresponde a usar a regra de pivotação conhecida como *largest reduced cost* [TZ93], e verificar empiricamente o que acontece com o tempo requerido pelo SimplexGC.

Infelizmente, a formulação (2) pode não ter solução ótima onde todas as variáveis sejam inteiras. Na próxima seção discutimos como lidar com esta dificuldade.

3 Encontrando uma Solução Inteira

Explicaremos agora como podemos obter uma solução inteira a partir das soluções encontradas pelo SimplexGC. O processo é iterativo. Cada iteração inicia com uma instância I do CGV e consiste basicamente em resolver (2) com o SimplexGC obtendo B, b e x. Se todos os elementos de x forem inteiros, devolvemos B, b e x e paramos.

Caso contrário, calculamos $x^* = (x_1^*, \ldots, x_m^*)$ fazendo $x_i^* = \lfloor x_i \rfloor$ $(i = 1, \ldots, m)$. Adotando esta nova solução, uma parte da demanda dos itens não será atendida. Mais precisamente, a demanda não atendida de cada item i será $d_i^* = d_i - \sum_{j=1}^{m} B_{i,j} x_j^*$. Fazendo $d^* = (d_1^*, \ldots, d_m^*)$, temos então uma instância residual $I^* = (L, A, l, c, d^*)$ (podemos ter o cuidado de eliminar de I^* os itens que porventura tenham demanda nula). Se algum $x_i^* > 0$ $(i = 1, \ldots, m)$, uma parte da demanda é atendida pela solução x^*. Neste caso devolvemos B, b e x, fazemos $I = I^*$ e começamos uma nova iteração. Se $x_i^* = 0$ $(i = 1, \ldots, m)$, nenhuma parte da demanda é atendida por x^*. Resolvemos então a instância I^* com o algoritmo *Hybrid First Fit* (HFF), proposto por Chung, Garey e Johnson [CGJ82].

Apresentamos a seguir o algoritmo CG que implementa o processo iterativo que acabamos de descrever. Sobre a complexidade do algoritmo CG, é possível mostrar que para instâncias do CGV onde os itens não são "muito pequenos" em relação ao tamanho das placas, o algoritmo CG requer tempo polinomial, no caso médio [Cin04].

4 Perturbando as Instâncias Residuais

Experimentamos uma modificação no algoritmo CG com o intuito de obter soluções de melhor qualidade. Tal modificação consiste basicamente no seguinte: se ao resolver uma instância, a solução devolvida pelo SimplexGC, após ser arredondada para baixo, for igual a zero, em vez de submeter esta instância para o algoritmo HFF resolver, utilizamos o HFF para obter um *bom* padrão, atualizamos as demandas e, se houver demanda não atendida, retornamos ao passo 1. Em nossa implementação, consideramos como *bom* um padrão produzido pelo HFF que apresente o menor desperdício de área. Eventualmente, outros critérios podem ser utilizados.

Chamaremos o algoritmo obtido com esta modificação de CG^p. Observe que a idéia básica deste algoritmo é "perturbar" as instâncias residuais cuja solução de sua relaxação linear, arredondada para baixo, seja igual a zero. Com este procedimento, espera-se que a solução obtida

Algoritmo 3: CG

Entrada: Uma instância $I = (L, A, l, a, d)$ do CGV, onde $L = (L_1, \ldots, L_k)$, $A = (A_1, \ldots, A_k)$, $l = (l_1, \ldots, l_m)$, $a = (a_1, \ldots, a_m)$ e $d = (d_1, \ldots, d_m)$.

Saída: Uma solução para I

1 Execute o algoritmo SimplexGC com parâmetros L, A, l, a, d obtendo B, b e x.
2 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$.
3 Se $x_i^* > 0$ para algum i, $1 \leq i \leq m$, então
  3.1 Devolva B, b e $x_1^*, \ldots, x_m^*$ (mas não pare).
  3.2 Para $i = 1$ até m faça
    3.2.1 Para $j = 1$ até m faça $d_i = d_i - A_{i,j} x_j^*$.
  3.3 Faça $m' = 0$.
  3.4 Para $i = 1$ até m faça
    3.4.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$, $a_{m'} = a_i$ e $d_{m'} = d_i$.
  3.5 Se $m' = 0$ então pare.
  3.6 Faça $m = m'$ e volte ao passo 1.
4 $l' = \emptyset$, $a' = \emptyset$.
5 Para $i = 1$ até m faça
  5.1 Para $j = 1$ até d_i faça
    5.1.1 $l' = l' \cup \{l_i\}$, $a' = a' \cup \{a_i\}$. `/* l' e a' são multiconjuntos */`
6 Faça $j = min(\frac{C_i}{L_i A_i} \mid i = 1, \ldots, k)$ e $h = min(i \mid C_i = jL_iA_i)$.
7 Devolva a solução do algoritmo HFF executado com parâmetros L_h, A_h, l', a'.

pelo SimplexGC para a instância residual possua mais variáveis com valores maiores ou iguais a 1. Apresentamos a seguir uma descrição do algoritmo CG^p.

Podemos adaptar os algoritmos CG e CG^p para a variante em que rotações são permitidas, que chamamos de CGV^r. Para isso, basta modificar os parâmetros utilizados na chamada ao algoritmo DP, feita no passo 4.1 do SimplexGC. Tal modificação consiste em, para cada item i, de largura l_i, altura a_i e valor v_i, adicionar um item de largura a_i, altura l_i e valor v_i, desde que $l_i \neq a_i$, $l_i \leq A$ e $a_i \leq L$.

5 Resultados Computacionais

Não encontramos na literatura instâncias de teste do CGV. No entanto, encontramos instâncias do Problema da Mochila Retangular na OR-LIBRARY[10]. Tal biblioteca é uma coleção de instâncias de testes para uma grande variedade de problemas na área de pesquisa operacional. Nessas instâncias, denominadas de $gcut1, \ldots, gcut12$, as placas são quadradas, com dimensões variando entre 250 e 3000, a quantidade de itens varia entre 10 e 50, o valor de cada item é igual à sua área e rotações ortogonais não são permitidas.

Adaptamos tais instâncias para o CGV atribuindo a cada item uma demanda gerada aleatoriamente entre 1 e 100. As demandas foram geradas utilizando-se a função *rand* da linguagem PERL (versão 5.6.1) [WCO00] com a semente sendo inicializada com o número $(L + A) * m + m$, onde L e A são a largura e a altura da placa, respectivamente, e m é a quantidade de itens. Por exemplo, para a instância $gcut1$ a semente da função *rand* foi o número 5010. Além disso, para cada instância cujas placas tinham dimensões (L, A), adicionamos mais dois tipos de placas

[10]Esta biblioteca pode ser acessada através do endereço `http://mscmga.ms.ic.ac.uk/info.html`. Uma descrição dessa biblioteca e de seus objetivos é fornecida em [Bea90].

Algoritmo 4: CG^p

Entrada: Uma instância $I = (L, A, l, a, d)$ do CGV, onde $L = (L_1, \ldots, L_k)$, $A = (A_1, \ldots, A_k)$, $l = (l_1, \ldots, l_m)$, $a = (a_1, \ldots, a_m)$ e $d = (d_1, \ldots, d_m)$.

Saída: Uma solução para I

1 Execute o algoritmo SimplexGC com parâmetros L, A, l, a, d obtendo B, b e x.
2 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$.
3 Se $x_i^* > 0$ para algum i, $1 \leq i \leq m$, então
  3.1 Devolva B, b e $x_1^*, \ldots, x_m^*$ (mas não pare).
  3.2 Para $i = 1$ até m faça
    3.2.1 Para $j = 1$ até m faça $d_i = d_i - A_{i,j}x_j^*$.
  3.3 Faça $m' = 0$.
  3.4 Para $i = 1$ até m faça
    3.4.1 Se $d_i > 0$ faça $m' = m' + 1$, $l_{m'} = l_i$, $a_{m'} = a_i$ e $d_{m'} = d_i$.
  3.5 Se $m' = 0$ então pare.
  3.6 Faça $m = m'$ e volte ao passo 1.
4 $l' = \emptyset$, $a' = \emptyset$.
5 Para $i = 1$ até m faça
  5.1 Para $j = 1$ até d_i faça
    5.1.1 $l' = l' \cup \{l_i\}$, $a' = a' \cup \{a_i\}$. `/* l' e a' são multiconjuntos */`
6 Faça $j = min(\frac{C_i}{L_i A_i} \mid i = 1, \ldots, k)$ e $h = min(i \mid C_i = jL_iA_i)$.
7 Devolva um padrão gerado pelo algoritmo HFF, executado com parâmetros L_h, A_h, l', a', que apresente o menor desperdício de área e atualize as demandas.
8 Se houver demanda não atendida, volte ao passo 1.

com dimensões $(\frac{8}{10}L, \frac{12}{10}A)$ e $(\frac{9}{10}L, \frac{11}{10}A)$. Para cada placa, adotamos custo igual à sua área. Chamamos tais instâncias do CGV de $gcut1v, \ldots, gcut12v$.

Os algoritmos DP, SimplexGC, GC e GC^p foram implementados utilizando-se a linguagem C e o código executável gerado pelo compilador *gcc* versão 2.95.4 (*Debian prerelease*). Os testes executados num computador com dois processadores *AMD Athlon MP 1800+*, clock de 1.5 Ghz, 3.5 GB de memória principal e sistema operacional *Linux* (distribuição *Debian GNU/Linux 3.0*). Fizemos uso do software *Xpress-MP* [Xpr02] para resolver os sistemas de equações lineares que aparecem nos passos 2 e 5 do algoritmo SimplexGC.

Utilizamos o algoritmo CG para resolver as instâncias $gcut1v, \ldots, gcut12v$. Apresentamos na Tabela 1 o valor da solução encontrada pelo CG, e o limite inferior (LI) fornecido pela solução de (2) para o valor de uma solução inteira ótima e a diferença percentual entre a solução do CG e o valor do LI[11].

Detalhamos a utilização das placas na solução do algoritmo. Mostramos a quantidade de placas do tipo 1, do tipo 2, do tipo 3 e a quantidade total de placas utilizadas na solução do algoritmo e numa solução de (2) arrendondada para cima ($\lceil n \rceil$). Observe que o fato da quantidade de placas utilizadas na solução do algoritmo ser igual à $\lceil n \rceil$ não implica que a solução do algoritmo é necessariamente ótima. Lembre-se de que o nosso objetivo é minimizar a soma dos custos das placas utilizadas.

Exibimos ainda o tempo gasto e a quantidade de colunas geradas. Cada instância foi resolvida 10 vezes e os tempos apresentados foram obtidos calculando-se a média do tempo gasto nestas 10

[11] Seja n a quantidade de placas utilizadas numa solução ótima de (2). Calculamos LI somando o valor de uma solução ótima de (2) a $(\lceil n \rceil - n)C_{min}$, onde C_{min} é o menor custo dentre todos os tipos de placas.

Instância	Valor da Solução	Limite Inferior (LI)	Diferença em relação ao LI	Quantidade de Placas					Tempo (seg)	Colunas Geradas
				Tipo 1	Tipo 2	Tipo 3	Total Solução	Solução (2)		
$gcut1v$	14880000	14875313	0,032%	24	223	0	247	247	0,213	43
$gcut2v$	15863750	15728072	0,863%	59	106	94	259	257	6,104	585
$gcut3v$	19930000	19800108	0,656%	262	18	40	320	318	11,629	1145
$gcut4v$	46477500	46278728	0,430%	462	182	108	752	749	62,437	3745
$gcut5v$	42000000	41727500	0,653%	36	121	16	173	172	0,441	72
$gcut6v$	74187500	74177813	0,013%	116	14	169	299	299	3,810	316
$gcut7v$	123017500	122467968	0,449%	172	254	77	503	501	13,532	1063
$gcut8v$	156122500	155314120	0,520%	268	195	171	634	631	109,093	2256
$gcut9v$	129360000	129293847	0,051%	96	10	24	130	130	0,178	26
$gcut10v$	254130000	253055471	0,425%	126	118	15	259	258	1,340	117
$gcut11v$	295320000	293204167	0,722%	141	37	120	298	296	73,748	1437
$gcut12v$	601450000	600245987	0,201%	316	220	75	611	610	146,726	1482
Média			**0,418%**							

Tabela 1: Soluções do CG para as instâncias $gcut1v, \ldots, gcut12v$.

resoluções. Na média, a diferença entre o valor da solução encontrada pelo CG e o limite inferior foi de apenas 0,418%. Observamos ainda que o tempo gasto para resolver estas instâncias foi sempre inferior a 3 minutos.

Experimentamos também resolver as instâncias $gcut1v, \ldots, gcut12v$ com o algoritmo CG^p. A Tabela 2 contém os resultados obtidos. Percebemos que o desempenho do algoritmo CG^p foi um pouco melhor, em termos de qualidade das soluções obtidas, do que o do CG. A diferença entre o valor da solução encontrada pelo CG^p e o limite inferior caiu para 0,284%, na média. Por outro lado, o número de colunas geradas sofreu um aumento de aproximadamente 60%, na média, e o tempo gasto aumentou em cerca 25%, também na média.

Instância	Valor da Solução	Limite Inferior (LI)	Diferença em relação ao LI	Quantidade de Placas					Tempo (seg)	Colunas Geradas
				Tipo 1	Tipo 2	Tipo 3	Total Solução	Solução (2)		
$gcut1v$	14880000	14875313	0,032%	24	223	0	247	247	0,261	43
$gcut2v$	15738750	15728072	0,068%	57	106	94	257	257	7,203	749
$gcut3v$	19867500	19800108	0,340%	261	18	40	319	318	19,552	2128
$gcut4v$	46415000	46278728	0,294%	461	182	108	751	749	102,769	7770
$gcut5v$	42000000	41727500	0,653%	36	121	16	173	172	0,504	82
$gcut6v$	74187500	74177813	0,013%	116	14	169	299	299	3,939	316
$gcut7v$	122767500	122467968	0,245%	171	254	77	502	501	20,184	2006
$gcut8v$	155872500	155314120	0,360%	267	195	171	633	631	124,928	3450
$gcut9v$	129360000	129293847	0,051%	96	10	24	130	130	0,187	26
$gcut10v$	254130000	253055471	0,425%	126	118	15	259	258	1,912	190
$gcut11v$	295320000	293204167	0,722%	141	37	120	298	296	103,842	2551
$gcut12v$	601450000	600245987	0,201%	316	220	75	611	610	152,533	1965
Média			**0,284%**							

Tabela 2: Soluções do CG^p para as instâncias $gcut1v, \ldots, gcut12v$.

Resolvemos também as instâncias $gcut1v, \ldots, gcut12v$ permitindo rotações. Chamamos tais instâncias de $gcut1vr, \ldots, gcut12vr$. Apresentamos na Tabela 3 os resultados obtidos ao resolver tais instâncias com o CG. Na média, a diferença entre o valor da solução encontrada pelo CG e o limite inferior foi de 0,631%. Comparando-se o tempo gasto para resolver as instâncias $gcut1v, \ldots, gcut12v$ e as instâncias $gcut1vr, \ldots, gcut12vr$, verificamos que o aumento nestas últimas foi de aproximadamente 475%. Já o aumento no número de colunas foi de aproximadamente 50%. Isto se deve ao aumento na quantidade de padrões viáveis.

Resolvemos as instâncias $gcut1vr, \ldots, gcut12vr$ com o algoritmo CG^p. Apresentamos na Tabela 4 os resultados. O algoritmo CG^p apresentou desempenho um pouco superior ao CG, no que diz respeito à qualidade das soluções obtidas. Na média, a diferença entre o valor da solução

Instância	Valor da Solução	Limite Inferior (LI)	Diferença em relação ao LI	Quantidade de Placas					Tempo (seg)	Colunas Geradas
				Tipo 1	Tipo 2	Tipo 3	Total Solução	Solução (2)		
gcut1vr	13823750	13820625	0,023%	62	139	26	227	227	0,364	27
gcut2vr	15287500	15097046	1,262%	73	55	120	248	245	10,183	417
gcut3vr	19306875	19172691	0,700%	99	65	149	313	311	21,452	962
gcut4vr	45046875	44595200	1,013%	327	108	293	728	721	144,937	4601
gcut5vr	38890000	38618516	0,703%	40	75	44	159	158	2,581	122
gcut6vr	69942500	69668304	0,394%	56	65	163	284	283	11,219	405
gcut7vr	115385000	114610045	0,676%	215	164	90	469	466	45,930	999
gcut8vr	152280000	151472845	0,533%	330	134	152	616	613	449,135	4405
gcut9vr	119740000	119606667	0,111%	34	46	42	122	122	1,256	43
gcut10vr	247660000	247422500	0,096%	37	205	14	256	256	9,228	134
gcut11vr	284860000	281724141	1,113%	91	39	158	288	285	355,217	1621
gcut12vr	565870000	560529066	0,953%	202	147	225	574	569	1418,728	4779
Média			**0,631%**							

Tabela 3: Soluções do CG para as instâncias $gcut1vr, \ldots, gcut12vr$.

encontrada pelo CG^p e o limite inferior caiu para 0,333%. No entanto, o preço desta melhoria foi um aumento médio em torno de 71% no número de colunas geradas e de aproximadamente 47% no tempo gasto.

Instância	Valor da Solução	Limite Inferior (LI)	Diferença em relação ao LI	Quantidade de Placas					Tempo (seg)	Colunas Geradas
				Tipo 1	Tipo 2	Tipo 3	Total Solução	Solução (2)		
gcut1vr	13823750	13820625	0,023%	62	139	26	227	227	0,381	27
gcut2vr	15100000	15097046	0,020%	70	55	120	245	245	12,672	609
gcut3vr	19244375	19172691	0,374%	98	65	149	312	311	49,288	2582
gcut4vr	44734375	44595200	0,312%	322	108	293	723	721	179,137	7517
gcut5vr	38890000	38618516	0,703%	40	75	44	159	158	3,213	192
gcut6vr	70192500	69668304	0,752%	57	65	163	285	283	11,219	582
gcut7vr	114885000	114610045	0,240%	213	164	90	467	466	80,240	2191
gcut8vr	152030000	151472845	0,368%	329	134	152	615	613	708,471	8310
gcut9vr	119740000	119606667	0,111%	34	46	42	122	122	1,338	43
gcut10vr	247660000	247422500	0,096%	37	205	14	256	256	9,890	134
gcut11vr	282860000	281724141	0,403%	89	39	158	286	285	504,716	3060
gcut12vr	563870000	560529066	0,596%	200	147	225	572	569	2079,114	9046
Média			**0,333%**							

Tabela 4: Soluções do CG^p para as instâncias $gcut1vr, \ldots, gcut12vr$.

Considerações Finais

Estudamos a variante do problema de corte de guilhotina bidimensional na qual as placas podem não ser idênticas, chamada de CGV. Tal problema foi muito pouco abordado na literatura. Aplicamos o método de geração de colunas ao CGV, utilizando programação dinâmica para gerar as colunas, e o algoritmo HFF para resolver a última instância residual. Esta é a idéia básica do algoritmo CG. Vimos que este algoritmo requer tempo polinomial, no caso médio, se os itens não são "muito pequenos" em relação ao tamanho das placas.

O algoritmo CG reúne algoritmos de diversas naturezas, tais como o método Simplex com geração de colunas, o DP e o HFF. Esta abordagem diversificada tem se mostrado promissora, e foi explorada em diversos trabalhos envolvendo o problema de corte unidimensional [WG96, Gau97, Cin99].

Introduzimos a idéia de "perturbar" as instâncias residuais como forma de obter soluções de melhor qualidade. Incorporamos esta idéia ao CG obtendo o algoritmo CG^p. Implementamos

os algoritmos CG e CG^p, e resolvemos diversas instâncias do CGV, tendo obtido soluções *quase* ótimas para todas elas. O tempo requerido foi bastante satisfatório. Conforme esperávamos, o CG^p encontrou soluções um pouco melhores que o CG a custa de um pequeno aumento no tempo gasto.

Resolvemos também instâncias do CGV^r utilizando os algoritmos CG e CG^p, tendo encontrado soluções *quase* ótimas para todas as instâncias, em intervalos de tempo razoáveis. Novamente, o algoritmo que utiliza o método da perturbação das instâncias residuais, o CG^p, encontrou soluções de melhor qualidade, utilizando um pouco mais de tempo que o CG.

Verificamos que os algoritmos que propomos apresentaram um bom desempenho, em termos de tempo e de qualidade das soluções encontradas ao resolver diversas instâncias de pequeno e médio porte. Tais evidências empíricas indicam que estes algoritmos são adequados para resolver a maior parte das instâncias associadas a situações reais.

As idéias explicadas neste artigo podem ser adaptadas para diversas outras variantes do problema de corte. Por exemplo, no caso não guilhotinado, podemos obter uma solução inicial utilizando padrões homogêneos maximais[12]. Para gerar as colunas poderíamos utilizar diversos algoritmos propostos na literatura. Como exemplo, podemos citar os algoritmos propostos por Arenales e Morábito [AM95] e Lins, Lins e Morábito [LLM03]. Para resolver a última instância residual podemos utilizar o HFF.

Também podemos utilizar geração de colunas para a variante onde a quantidade de vezes que um item ocorre num padrão é limitada. Tal variante, proposta por Christofides e Whitlock [CW77], é chamada de problema de corte de estoque bidimensional restrito. Para gerar uma solução inicial podemos utilizar padrões homogêneos (não necessariamente maximais). Cada nova coluna pode ser obtida utilizando-se um dos diversos algoritmos propostos na literatura [OF90, DAD95, MA96]. A última instância residual pode ser resolvida com o HFF. Podemos também aplicar o método de geração de colunas também poderia ser aplicado para o caso com a restrição de cortes de guilhotina. Note que padrões homogêneos e os padrões produzidos pelo HFF são guilhotináveis. Poderíamos gerar colunas utilizando o algoritmo de Cung, Hifi e Le Cun [CHLC00].

Um passo mais audacioso seria adaptar o método de gerações de colunas para o problema de corte de estoque tridimensional. Soluções iniciais podem ser obtidas com padrões homogêneos. Existem alguns algoritmos de aproximação para o caso tridimensional que poderiam ser usados para resolver a última instância residual [LC92, MW00]. No entanto, não conhecemos algoritmos exatos propostos na literatura para gerar colunas. Já no caso guilhotinado, poderíamos adaptar o algoritmo DP e assim gerar novas colunas. Seria interessante descobrir se o tempo requerido seria aceitável.

Referências

[AM95] M. Arenales and R. Morabito, *An and/or-graph approach to the solution of two-dimensional non-guillotine cutting problems*, European Journal of Operations Research **84** (1995), 599–617.

[Bea85] J. E. Beasley, *Algorithms for unconstrained two-dimensional guillotine cutting*, Journal of the Operational Research Society **36** (1985), no. 4, 297–306.

[Bea90] J. E. Beasley, *Or-library: distributing test problems by electronic mail*, Journal of the Operational Research Society **41** (1990), no. 11, 1069–1072.

[12] padrões onde ocorre apenas um tipo de item, o maior número de vezes possível.

[CGJ82] F. R. K. Chung, M. R. Garey, and D. S. Johnson, *On packing two-dimensional bins*, SIAM J. Algebraic Discrete Methods **3** (1982), 66–76.

[CHLC00] Van-Dat Cung, Mhand Hifi, and Bertrand Le Cun, *Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm*, Int. Trans. Oper. Res. **7** (2000), no. 3, 185–210.

[Cin99] G. F. Cintra, *Algoritmos híbridos para o problema de corte unidimensional*, XXV Conferência Latinoamericana de Informática (Assunção), 1999.

[Cin04] ———, *Algoritmos para problemas de corte de guilhotina bidimensional*, Ph.D. thesis, Instituto de Matemática e Estatística, São Paulo, 2004.

[CW77] N. Christofides and C. Whitlock, *An algorithm for two dimensional cutting problems*, Operations Research **25** (1977), 30–44.

[CW04] G. F. Cintra and Y. Wakabayashi, *Dynamic programming and column generation based approaches for two-dimensional guillotine cutting problems*, Lecture Notes in Computer Science **3059** (2004), 175–190.

[DAD95] V. P. Daza, A. G. Alvarenga, and J. Diego, *Exact solutions for constrained two-dimensional cutting problems*, European Journal of Operations Research **84** (1995), 633–644.

[Gau97] T. Gau, *Solution methods for the standard one-dimensional cutting stock problem*, Physica, Heidelberg, 1997.

[GG61] P. Gilmore and R. Gomory, *A linear programming approach to the cutting stock problem*, Operations Research **9** (1961), 849–859.

[GG63] ———, *A linear programming approach to the cutting stock problem - part II*, Operations Research **11** (1963), 863–888.

[Her72] J. C. Herz, *A recursive computational procedure for two-dimensional stock-cutting*, IBM Journal of Research Development (1972), 462–469.

[LC92] K. Li and K-H. Cheng, *Heuristic algorithms for on-line packing in three dimensions*, Journal of Algorithms **13** (1992), 589–605.

[LLM03] Lauro Lins, Sóstenes Lins, and Reinaldo Morabito, *An l-approach for packing (l,w)-rectangles into rectangular and l-shaped pieces*, Journal of the Operational Research Society **54** (2003), 777–789.

[MA96] R. Morabito and M. N. Arenales, *Staged and constrained two-dimensional guilhotine cutting problems: an and/or-graph approach*, European Journal of Operational Research **94** (1996), 548–560.

[MW00] F. K. Miyazawa and Y. Wakabayashi, *Approximation algorithms for the orthogonal z-oriented three-dimensional packing problem*, SIAM Journal on Computing **29** (2000), no. 3, 1008–1029.

[Nel93] J. Nelißen, *New approaches to the pallet loading problem*, Tech. report, RWTH Aachen, Germany, August 1993.

[OF90] J. F. Oliveira and J. S. Ferreira, *An improved version of Wang's algorithm for two-dimensional cutting problems*, European Journal of Operations Research **44** (1990), 256–266.

[TZ93] Tamás Terlaky and Shu Zhong Zhang, *Pivot rules for linear programming: a survey on recent theoretical developments*, Ann. Oper. Res. **46/47** (1993), no. 1-4, 203–233, Degeneracy in optimization problems.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant, *Programming perl*, 3rd ed., O'Reilly & Associates, July 2000.

[WG96] G. Wäscher and T. Gau, *Heuristics for the integer one-dimensional cutting stock problem: a computational study*, OR Spektrum **18** (1996), 131–144.

[Xpr02] Xpress, *Xpress optimizer reference manual*, DASH Optimization, Inc, 2002.

[YZH91] Horácio Yanasse, Alan Zinober, and Reginald Harris, *Two-dimensional cutting stock with multiple stock sizes*, J. Oper. Res. Soc. **42** (1991), no. 8, 673–683.