Relatório de Estudos

MAC5701 - Tópicos em Ciência da Computação

Aplicação do Método de Geração de Colunas ao Problema de Corte de Guilhotina Bidimensional

Aluno: Glauber Ferreira Cintra

Orientadora: Profa. Dra. Yoshiko Wakabayashi

Área: Otimização Combinatória

— São Paulo, junho de 2003 —

Índice

Introdução

Apresentamos nesta monografia o relatório dos estudos realizados no decorrer do semestre como parte das atividades da disciplina. São dois os objetivos principais deste trabalho: o primeiro é apresentar algumas variantes do problema de corte de estoque bidimensional. O segundo objetivo é investigar a aplicação de programação dinâmica e do método de geração de colunas na resolução de tais variantes. Vamos supor aqui que o leitor tenha conhecimentos básicos de complexidade computacional, programação dinâmica e programação linear.

Este texto está organizado da seguinte maneira. Inicialmente, na Seção 1, definimos o problema pesquisado e apresentamos algumas de suas variantes. Em seguida, na Seção 2, mostramos como usar programação dinâmica para resolver algumas variantes. Na seção seguinte introduzimos o método de geração de colunas e mostramos como implementá-lo para solucionar outras variantes do problema. Finalmente, na Seção 4, apresentamos os resultados computacionais obtidos ao resolver diversas instâncias citadas na literatura.

1 Problemas de Corte de Estoque Bidimensional

Os problemas de corte de estoque bidimensional, na sua forma mais geral, consistem em: dada a largura L e a altura C de um objeto retangular, genericamente denominado de *placa*, e uma lista de m itens retangulares, cada item i com largura $l_i \leq L$ e altura $c_i \leq C$, determinar "a melhor maneira" de cortar as placas de forma a produzir os itens da lista. Dependendo do contexto onde estes problemas surgem, o termo "a melhor maneira" pode ter significados diferentes.

Em algumas situações estamos interessados em maximizar a soma dos valores dos itens produzidos ao cortar uma única placa. Neste caso um valor arbitrário v_i está associado a cada item i. Chamaremos de *problema de corte de estoque bidimensional com valor* (PCEV_2) a variante onde cada item i possui valor v_i $(i = 1, \ldots, m)$ e desejamos determinar como cortar uma única placa de modo a maximizar a soma dos valores dos itens produzidos.

Em outras situações precisamos produzir d_i unidades de cada item i. Dizemos que d_i é a demanda do item i. Neste caso, a "melhor maneira" de cortar as placas é uma que utiliza o menor número de placas possível de modo a atender a demanda de todos os itens. Chamaremos de *problema de corte de estoque bidimensional com demandas* (PCED_2) a variante onde cada item i possui demanda d_i $(i = 1, \ldots, m)$ e desejamos determinar como produzir d_i unidades de cada item i utilizando o menor número de placas possível.

Consideraremos a largura como a dimensão horizontal (eixo x) e a altura como a dimensão vertical (eixo y). Supomos aqui que os cortes são infinitamente finos, ou seja, têm largura zero. Apesar de isto não ocorrer na prática, esta suposição não é na verdade uma restrição, pois se num processo de corte do mundo real todos os cortes têm largura δ, basta somar δ a L, C, $l_1, \ldots, l_m, c_1, \ldots, c_m$, e resolver esta nova instância (agora supondo que todos os cortes têm largura zero).

O interesse por este problema e por outros correlatos tem motivações práticas e teóricas. Problemas de corte de estoque bidimensional surgem em diversas situações do mundo real. Podemos citar como exemplos os processos de corte de chapas de metal e madeira, lâminas de vidro, peças de carpete, etc. Empresas que conseguem melhorias nestes processos podem alcançar ganhos substanciais e, dependendo da escala de produção, podem obter assim uma vantagem decisiva na competição com outras empresas do setor. Além disso, o estudo dos problemas de corte e empacotamento por um número crescente de pesquisadores, tem gerado avanços significativos em diversas áreas, tais como *programação linear*, *programação dinâmica*, *complexidade computacional* e *algoritmos de aproximação*.

Apesar de sua aparente simplicidade, o problema de corte de estoque bidimensional é um problema de natureza complexa. Analisemos a variante do PCED_2 onde todas as alturas (da placa e dos itens) são iguais a zero e todos os itens possuem demanda igual a 1. Tal variante é chamada na literatura de *problema do empacotamento de bins* (PEB). Este problema tem como caso particular o *problema da 3-partição*[1], que é sabido ser é *$\mathcal{NP}$-completo no sentido forte* [11]. Deste fato, segue que o PEB é um problema *$\mathcal{NP}$-difícil*. Consequentemente o PCED_2 também é *$\mathcal{NP}$-difícil*.

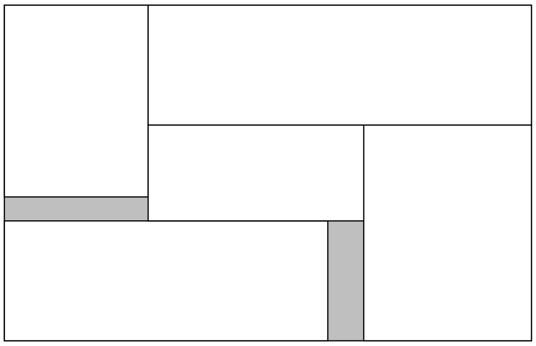
Descrevemos a seguir algumas características dos processos de corte que dão origem às variantes do PCEV_2 e do PCED_2 nas quais temos interesse.

1.1 Cortes de Guilhotina

Uma restrição bastante comum nos processos que envolvem corte de placas é exigir que todos os cortes sejam ortogonais a um dos lados da placa e se estendam em linha reta desde um lado até o lado oposto da placa. Tais cortes são chamados de *corte*

[1] O problema da 3-partição consiste em: dado um conjunto A com $3m$ elementos, onde cada elemento $a \in A$ possui valor valor $\frac{B}{4} < v_a < \frac{B}{2}$, de tal forma que $\sum_{a \in A} v_a = mB$, particionar A em subconjuntos disjuntos $A_1, \ldots, A_m$ tais que $\sum_{a \in A_i} v_a = B$ $(i = 1, \ldots, m)$.

de guilhotina. Chamamos de *padrão de corte* (ou simplesmente *padrão*) cada possível forma de cortar uma placa. Dizemos que um padrão é *guilhotinável* se pode ser obtido com uma sequência de cortes de guilhotina. Na Figura 1 temos, à esquerda, um padrão não-guilhotinável e, à direita, um padrão guilhotinável (a numeração indica a ordem em que os cortes de guilhotina podem ser feitos).



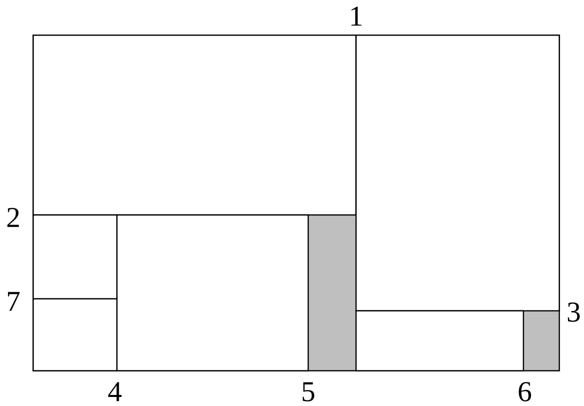


Figura 1:

(a) Padrão não-guilhotinável (b) Padrão guilhotinável

Dado um padrão guilhotinável, chamamos de *estágio* de corte uma sequência maximal de cortes consecutivos todos na mesma direção. No padrão da Figura 1b, no primeiro estágio é feito o corte 1, no segundo estágio são feitos os cortes 2 e 3, no estágio seguinte os cortes 4, 5 e 6, e no último estágio é feito o corte 7. Em muitos processos de corte do mundo real a quantidade de estágios pode ser limitada. Diversos autores investigaram o caso em que o número de estágios está limitado a dois ou três [15, 17]. Em nosso trabalho, vamos supor que a quantidade de estágios é ilimitada.

1.2 Rotações Ortogonais

Em diversas situações práticas é permitido que os itens sofram rotações ortogonais, ou seja, de 90 graus. Por brevidade, chamaremos rotações ortogonais simplesmente de *rotações*. Em outras situações, como no corte de tecido estampado, de vidro trabalhado, etc, rotações podem não ser permitidas.

É possível transformar uma instância onde rotações são permitidas numa instância onde rotações não são permitidas da seguinte maneira: para cada item i, de largura l_i e altura c_i, inserimos um item de largura c_i e altura l_i, desde que $l_i \neq c_i$, $l_i \leq C$ e $c_i \leq L$. Dessa forma, vamos supor restante deste texto que os itens não podem sofrer rotação.

1.3 Atribuindo Valores

Como citamos anteriormente, podemos associar a cada item i um valor v_i ($i = 1, \ldots, m$). Neste caso, o objetivo é determinar como cortar *uma placa* de modo a maximizar a soma dos valores dos itens produzidos. Dado um padrão, consideramos que todo retângulo que resulta do corte desse padrão cujas dimensões não coincidam com as dimensões de nenhum dos itens possui valor zero. Na Figura 1, representamos por retângulos escuros aqueles que possuem valor zero.

Se desejarmos simplesmente encontrar um padrão que minimize a sobra da placa, basta atribuir a cada item i valor $l_i.c_i$, ou seja, valor equivalente à sua área. Dessa forma, maximizar a soma dos valores dos itens produzidos equivale a minimizar a sobra.

Chamaremos de *problema de corte de guilhotina bidimensional com valor* (PCGV_2) a variante do PCEV_2 onde o padrão produzido tem que ser guilhotinável. Como veremos na seção seguinte, o PCGV_2 possui propriedades que nos permitem aplicar a técnica de programação dinâmica para resolvê-lo.

1.4 Introduzindo Demandas

No PCED_2 associamos a cada item i uma demanda não-nula $d_i \in I\!N$ ($i = 1, \ldots, m$). Neste caso, estamos interessados em determinar como cortar a quantidade mínima possível de placas de modo a produzir d_i unidades de cada item i ($i = 1, \ldots, m$). Chamaremos de *problema de corte de guilhotina bidimensional com demandas (PCGD_2)* a variante do PCED_2 onde todos os padrões têm que ser guilhotináveis.

Existem ainda outras variantes estudadas na literatura, mas que não abordaremos neste trabalho. Por exemplo, Christofides e Whitlock [6] propuseram associar a cada item i um inteiro b_i ($i = 1, \ldots, m$), que representa o número máximo de vezes que o item i pode aparecer num padrão qualquer. Tal restrição, em geral arbitrária, visa diminuir a quantidade de padrões viáveis, numa tentativa de facilitar a resolução do problema. Esta variante é conhecida como *problema de corte de estoque bidimensional restrito*.

Veremos que o PCGD_2 pode ser formulado como um problema de programação linear inteira (PLI). Mostraremos com resolver a relaxação linear desse PLI através do método de geração de colunas e a partir daí encontrar uma solução para o PCGD_2. Ao utilizar o método de geração de colunas na resolução do PCGD_2, o PCGV_2 surge como subproblema. Na seção seguinte mostramos como usar programação dinâmica para resolver o PCGV_2. Na Seção 3 introduzimos o método de geração de colunas e mostramos como usá-lo para resolver o PCGD_2.

As principais características dos problemas que trataremos nas seções seguintes estão resumidas na Tabela 1.

Característica	PCGV_2	PCGD_2
Cortes de guilhotina	Sim	Sim
Rotações ortogonais	Não	Não
Atribuição de valores aos itens	Sim	Não
Atribuição de demandas aos itens	Não	Sim

Tabela 1: Características do PCGV_2 e do PCGD_2.

2 Programação Dinâmica para o PCGV_2

Com o intuito de deixar claro o problema que vamos abordar nesta seção, vamos relembrar sua definição. O problema de corte de guilhotina bidimensional (PCGV_2) consiste em: dada uma placa, de largura L e altura C, e uma lista de m itens, cada item i com largura $l_i \leq L$, altura $c_i \leq C$ e valor v_i, queremos determinar como cortar a placa, fazendo apenas cortes de guilhotina, de modo a maximizar a soma dos valores dos itens produzidos.

Nesta seção, vamos considerar que $L, C, l_1, \ldots, l_m, c_1, \ldots, c_m$ são inteiros. Além disso, vamos supor que rotações não são permitidas (se necessário, replicamos cada item como explicado na Subseção 1.2). Veremos a seguir que o PCGV_2 pode ser resolvido através de programação dinâmica.

2.1 Aplicabilidade da Programação Dinâmica ao PCGV_2

Dada uma instância I de um problema, chamamos de decomposição de I o resultado da subdivisão de I em diversas instâncias menores (segundo alguma métrica) e que sejam do mesmo tipo que I. A programação dinâmica é uma generalização da bem conhecida técnica de *dividir-e-conquistar* e consiste basicamente em: dada uma instância I, decompor esta instância de várias maneiras, e para cada decomposição, calcular as soluções ótimas das instâncias que constituem a decomposição e a partir destas soluções parciais encontrar a solução ótima de I. Esta técnica de resolução de problemas é especialmente bem sucedida se a quantidade de decomposições que tivermos que examinar for *pequena*.

Alguns problemas têm em sua estrutura propriedades que permitem que eles sejam resolvidos por programação dinâmica. Seja $\mathcal{P}$ um problema de maximização ou de minimização cujas instâncias podem ser decompostas em instâncias de $\mathcal{P}$. Podemos resolver $\mathcal{P}$ usando programação dinâmica se, para toda instância $I \in \mathcal{P}$, o valor de uma solução ótima de I for igual à soma dos valores de soluções ótimas das instâncias obtidas através de alguma decomposição de I. Diversos problemas possuem esta propriedade, tais como o *problema da mochila* [10], o *problema da subcadeia comum máxima* [10], o *problema do alinhamento de sequências* [2], etc.

O PCGV_2 também possui a propriedade citada no parágrafo anterior, embora o número de decomposições que precisamos examinar nem sempre seja *pequeno*. Dada uma placa de dimensões (l, c), seja $v(l, c) = max(\{v_i \mid 1 \leq i \leq m,\ l_i \leq l \text{ e } c_i \leq c\} \cup \{0\})$, ou seja, $v(l, c)$ é o valor de um item mais valioso que cabe na placa, ou 0 se nenhum item cabe na placa. Denotando por $OPT(I)$ o valor de uma solução ótima de uma instância I do PCGV_2, vale o resultado que se segue.

Proposição 2.1. *Seja $I = (L, C, l, c, v)$ uma instância do $PCGV_2$ onde $l = \{l_1, \ldots, l_m\}$, $c = \{c_1, \ldots, c_m\}$ e $v = \{v_1, \ldots, v_m\}$. Sejam $I_1 = (L', C, l, c, v)$, $I_2 = (L - L', C, l, c, v)$, $I_3 = (L, C', l, c, v)$ e $I_4 = (L, C - C', l, c, v)$. Para algum $1 \leq L' \leq \lfloor \frac{L}{2} \rfloor$ e algum $1 \leq C' \leq \lfloor \frac{C}{2} \rfloor$ ($L', C' \in I\!N$) temos que:*

$$OPT(I) = max\{v(L, C),\ OPT(I_1) + OPT(I_2),\ OPT(I_3) + OPT(I_4)\}.$$

Prova. Seja $\mathcal{P}$ um padrão que corresponde a uma solução ótima de I. Se não existe nenhum corte de guilhotina em $\mathcal{P}$, claramente $OPT(I) = v(L, C)$.

Suponha que o primeiro corte de guilhotina é feito na vertical (como na Figura 1b), seccionando a placa em duas placas menores (subplacas) de dimensões (L', C) e $(L - L', C)$. Note que podemos considerar que $L' \leq \lfloor \frac{L}{2} \rfloor$, caso contrário trocamos L' por $L - L'$. Claramente os valores das soluções obtidas nestas duas subplacas são ótimos, caso contrário $\mathcal{P}$ não seria ótimo. Dessa forma, $OPT(I) = OPT(I_1) + OPT(I_2)$.

Suponha agora que o primeiro corte de guilhotina é feito na horizontal, seccionando a placa em duas subplacas de dimensões (L, C') e $(L, C - C')$. Usando um raciocínio análogo ao anterior, concluímos que $OPT(I) = OPT(I_3) + OPT(I_4)$. $\square$

Fazendo uso da Proposição 2.1 mostraremos como o PCGV_2 pode ser resolvido através de programação dinâmica. Na próxima subseção discutimos detalhadamente como isto pode ser feito.

2.2 O Algoritmo PCGV$_2$PD

Para representar um padrão de corte bidimensional usamos um sistema de coordenadas bidimensional (x, y). Vamos convencionar que a placa tem largura L ao longo do eixo x e altura C ao longo do eixo y. Cada item i tem largura l_i ao longo do eixo x e altura c_i ao longo do eixo y $(i = 1, \ldots, m)$. Convencionamos que o canto inferior esquerdo da placa está na posição $(0, 0)$ do nosso sistema de coordenadas.

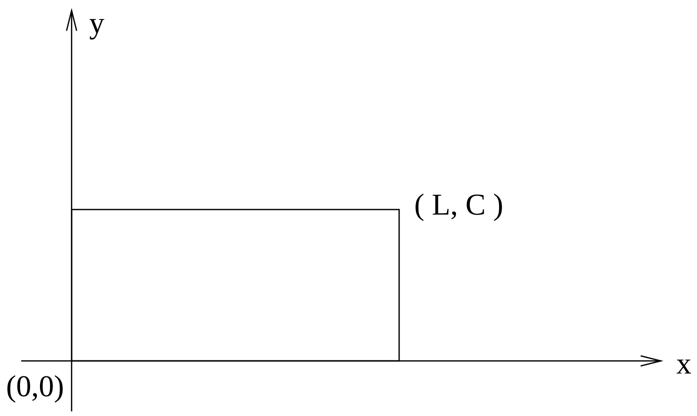


Figura 2: Sistema de coordenadas

Desse modo, para especificar a posição de um item k, na placa, basta especificar a posição do canto inferior esquerdo desse item em relação ao sistema de coordenadas. Neste caso, por causa da restrição de ortogonalidade (ver Subseção 1.1), se (x_k, y_k) denota um tal canto esquerdo e as dimensões deste item são $(l_i,\ c_i)$, a região ocupada por ele é dada por

$$\mathcal{R}_k = \{(x, y)\ :\ x_k \leq x \leq x_k + l_i \text{ e } y_k \leq y \leq y_k + c_i\}.$$

Com essas convenções, definimos um padrão de corte bidimensional como sendo uma coleção de itens $\wp$ que satisfaz as seguintes propriedades:

(a) Para todo item s em $\wp$, temos que $0 \leq x_s \leq L - l_i$ e $0 \leq y_s \leq C - c_i$ (l_i e c_i são as dimensões do item s);

(b) Para todo par de itens distintos s, t em $\wp$, temos que $\mathcal{R}_s \cap \mathcal{R}_t = \emptyset$.

Seja $A(l, c)$ o maior valor (isto é, a soma dos valores dos itens produzidos) que pode ser obtido ao cortar uma placa de largura l e altura c. O algoritmo baseado em programação dinâmica para solucionar o $PCGV_2$ consiste basicamente em resolver a seguinte fórmula de recorrência, onde $\bar{l} = \{1, \ldots, \lfloor \frac{l}{2} \rfloor\}$ e $\bar{c} = \{1, \ldots, \lfloor \frac{c}{2} \rfloor\}$:

$$A(l, c) = max_{i \in \bar{l},\ j \in \bar{c}}\{v(l, c),\ A(i, c) + A(l - i, c),\ A(l, j) + A(l, c - j)\} \qquad (2.1)$$

Ao calcular um padrão ótimo $\mathcal{P}$ para uma placa de dimensões $(l,\ c)$, temos três possibilidades:

- Nenhum corte de guilhotina é feito em $\mathcal{P}$. Neste caso, $A(l, c) = v(l, c)$.
- O primeiro corte de guilhotina em $\mathcal{P}$ é feito na posição i do eixo horizontal. Dessa forma, $A(l, c) = A(i, c) + A(l - i, c)$.
- O primeiro corte de guilhotina em $\mathcal{P}$ é feito na posição j do eixo vertical. Neste caso, $A(l, c) = A(l, j) + A(l, c - j)$.

Concluímos então que a fórmula de recorrência (2.1) é válida. Assim, dada uma instância I do $PCGV_2$, $A(L, C)$ nos fornece o valor de uma solução ótima de I.

Podemos modificar a recorrência (2.1) com o intuito de resolvê-la mais rapidamente. Chamemos de *ponto de discretização da largura* um valor $i \leq L$ que pode ser obtido através de uma combinação linear inteira[2] de $l_1, \ldots, l_m$. Analogamente, chamemos de *ponto de discretização da altura* um valor $j \leq C$ que pode ser obtido através de uma combinação linear inteira de $c_1, \ldots, c_m$. Tais pontos podem ser calculados como delineado a seguir.

Sejam $l = [l_1, \ldots, l_m]$, $c = [c_1, \ldots, c_m]$ e z um vetor de números inteiros não negativos. Sejam $P = \{z.l \mid z.l \leq L\}$ e $Q = \{z.c \mid z.c \leq C\}$. Os elementos de P e Q constituem os *pontos de discretização* da largura e da altura, respectivamente. Herz [14] definiu *padrão canônico* como sendo um padrão onde todos os cortes são feitos em pontos de discretização. Os padrões da Figura 1 são canônicos. Já o padrão da Figura 3 não é canônico, supondo que existem apenas os itens que aparecem no padrão e que um dos cortes de guilhotina é feito na linha pontilhada, correspondente à posição 23 no eixo horizontal.

Veremos agora que precisamos considerar somente padrões canônicos. Antes, vejamos uma definição. Dizemos que dois padrões, para uma dada instância, são *equivalentes* se eles possuem os mesmos itens, nas mesmas quantidades.

[2]Uma combinação linear inteira é uma combinação linear onde todos os coeficientes são números inteiros.

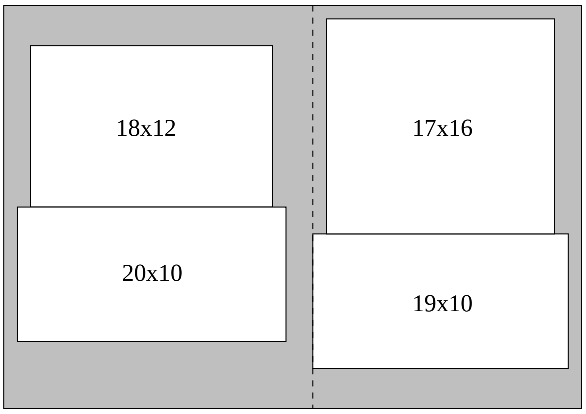


Figura 3: Padrão não-canônico

Proposição 2.2. *Seja $I = (L, C, l, c, v)$ uma instância do $PCGV_2$ e $\mathcal{P}$ um padrão guilhotinável para a instância I. Então existe um padrão guilhotinável canônico para I equivalente a $\mathcal{P}$.*

Prova. Seja k o número de cortes de guilhotina verticais e horizontais requeridos pelo padrão $\mathcal{P}$. Faremos indução em k. Se $k = 0$, obviamente a proposição é válida, pois nenhum corte de guilhotina precisa ser feito, e portanto $\mathcal{P}$ é canônico.

Suponha agora $k \geq 1$. Sem perda de generalidade, suponha que o primeiro corte de guilhotina de $\mathcal{P}$ é feito na vertical, numa posição i. Sejam $\mathcal{P}_1$ e $\mathcal{P}_2$ os padrões correspondentes às subplacas obtidas ao efetuar o primeiro corte de guilhotina de $\mathcal{P}$, sendo $\mathcal{P}_1$ o padrão correspondente à subplaca da esquerda. Se i é um ponto de discretização, aplicamos a hipótese de indução em $\mathcal{P}_1$ e $\mathcal{P}_2$, obtendo dois padrões guilhotináveis canônicos $\mathcal{P}'_1$ e $\mathcal{P}'_2$ equivalentes a $\mathcal{P}_1$ e $\mathcal{P}_2$, respectivamente. Colocando $\mathcal{P}'_1$ à esquerda de $\mathcal{P}'_2$ obtemos um padrão guilhotinável canônico equivalente a $\mathcal{P}$.

Suponha então que i não é um ponto de discretização. Seja $\mathcal{P}^*_1$ o padrão obtido movendo-se cada item contido em $\mathcal{P}_1$ para a posição mais à esquerda possível (sem que ocorra sobreposição). Seja i^* a posição mais à direita ocupada por um item de $\mathcal{P}^*_1$. Como i^* é um ponto de discretização, obviamente $i^* < i$. Efetue em $\mathcal{P}^*_1$ um corte de guilhotina vertical, na posição i^*, obtendo duas subplacas $\mathcal{S}_1$ e $\mathcal{S}_2$, sendo $\mathcal{S}_1$ a subplaca que contém o padrão equivalente a $\mathcal{P}^*_1$. Pela hipótese de indução, existe um padrão guilhotinável canônico $\mathcal{P}'_1$ equivalente a $\mathcal{P}^*_1$ e portanto equivalente a $\mathcal{P}_1$. Utilizando a mesma construção em $\mathcal{P}_2$, obtemos um padrão guilhotinável canônico $\mathcal{P}'_2$ equivalente a $\mathcal{P}_2$. Colocando $\mathcal{P}'_1$ à esquerda de $\mathcal{P}'_2$, de modo que i^* corresponda ao ponto mais à esquerda de $\mathcal{P}'_2$, obtemos um padrão guilhotinável canônico equivalente a $\mathcal{P}$. $\square$

Em vista do resultado acima, bastar procurarmos uma solução apenas entre os padrões canônicos. Portanto, na recorrência (2.1) somente precisamos exigir que $i \in P$ e $j \in Q$, onde P e Q contêm os pontos de discretização da largura e da altura, respectivamente.

Para reconstruir um padrão correspondente a uma solução ótima, basta anotar, para toda placa de largura $L' \in P$ e altura $C' \in Q$, qual a orientação e posição do primeiro corte de guilhotina a ser feito na placa. No caso de nenhum corte de guilhotina ser feito na placa, precisamos saber qual item deve ser colocado na placa. Descrevemos a seguir um algoritmo para o PCGV$_2$, que chamaremos de PCGV$_2$PD, baseado em programação dinâmica.

Algoritmo PCGV$_2$PD

Entrada: Uma instância $I = (L, C, l, c, v)$ do PCGV$_2$, onde $l = \{l_1, \ldots, l_m\}$,
$c = \{c_1, \ldots, c_m\}$ e $v = \{v_1, \ldots, v_m\}$.
Saída: Uma solução ótima de I.

Calcule $p_1 < \ldots < p_r$, os pontos de discretização da largura L.
Calcule $q_1 < \ldots < q_s$, os pontos de discretização da altura C.
Para $i = 1$ até r
  Para $j = 1$ até s
    $A(i,j) = max(\{v_k \mid 1 \le k \le m,\ l_k \le p_i \text{ e } c_k \le q_j\} \cup \{0\})$
    $item(i,j) = max(\{k \mid l_k \le p_i,\ c_k \le q_j \text{ e } v_k = A(i,j)\} \cup \{0\})$
    $guilhotina(i,j) = nil$
Para $i = 1$ até r
  Para $j = 1$ até s
    $n = max(\{k \mid 1 \le k \le r \text{ e } p_k \le \lfloor \frac{p_i}{2} \rfloor\} \cup \{0\})$
    Para $x = 1$ até n
      $t = max(\{k \mid 1 \le k \le r \text{ e } p_k \le p_i - p_x\} \cup \{0\})$
      Se $A(i,j) < A(x,j) + A(t,j)$
        $A(i,j) = A(x,j) + A(t,j)$, $posicao(i,j) = p_x$ e $guilhotina(i,j) =$ 'V'
    $n = max(\{k \mid 1 \le k \le s \text{ e } q_k \le \lfloor \frac{q_j}{2} \rfloor\} \cup \{0\})$
    Para $y = 1$ até n
      $t = max(\{k \mid 1 \le k \le s \text{ e } q_k \le q_j - q_y\} \cup \{0\})$
      Se $A(i,j) < A(i,y) + A(i,t)$
        $A(i,j) = A(i,y) + A(i,t)$, $posicao(i,j) = q_y$ e $guilhotina(i,j) =$ 'H'

No final do algoritmo PCGV$_2$PD, dada uma placa de dimensões $(p_i,\ q_j)$, $1 \le i \le r$ e $1 \le j \le s$, $A(i,j)$ contém o valor ótimo que pode ser obtido na placa, $guilhotina(i,j)$

indica a direção do primeiro corte de guilhotina e $posicao(i,j)$ é a posição, na vertical ou na horizontal, onde deve ser feito o primeiro corte de guilhotina. Se $guilhotina(i,j) = nil$, então nenhum corte de guilhotina deve ser feito na placa. Neste caso, $item(i,j)$ (se diferente de zero) indica qual item deve ser colocado na placa. O valor de uma solução ótima estará em $A(r,s)$.

Os pontos de discretização podem ser calculados em tempo $\mathcal{O}(m(L+C))$, utilizando-se uma variante do algoritmo baseado em programação dinâmica para o problema da mochila apresentado em [10]. Cada atribuição de valor à variável t pode ser feita em tempo constante. Observe que $r \leq L$ e $s \leq C$. Sendo assim, o algoritmo acima requer tempo $\mathcal{O}((LC+m)(L+C))$. Fazendo $N = max(L,\ C)$, no pior caso o $\mathrm{PCGV_2PD}$ requer tempo $\mathcal{O}(N^3 + mN^2)$. Trata-se portanto de um algoritmo *pseudo-polinomial*. Além disso, é fácil perceber que este algoritmo requer espaço proporcional a $r.s$, que no pior caso é $\mathcal{O}(N^2)$. Para valores *grandes* de L e C, este algoritmo se torna inviável.

Como veremos na Seção 4, implementamos e testamos o algoritmo $\mathrm{PGC_2PD}$ com diversas instâncias encontradas na literatura tendo obtido resultados bastante satisfatórios.

3 O Método de Geração de Colunas para o $\mathrm{PCGD_2}$

O método de geração de colunas foi pela primeira vez proposto para o problema de corte de estoque unidimensional ($\mathrm{PCE_1}$) por Gilmore e Gomory [12, 13]. O $\mathrm{PCE_1}$ 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 $(\leq L)$ e demanda $d_i \in I\!N$ $(i = 1, \ldots, m)$, determinar o menor número de barras necessário para atender a demanda. Obviamente também estamos interessados em determinar como as barras devem ser cortadas. Explicamos a seguir como resolver o $\mathrm{PCE_1}$ utilizando o método de geração de colunas e discutimos como adaptar esta técnica para o $\mathrm{PCGD_2}$.

3.1 Geração de Colunas

Para aplicar o método de geração de colunas, precisamos primeiramente formular o $\mathrm{PCE_1}$ como um problema de programação linear inteira. Para fazer isso, representamos cada padrão j por um vetor-coluna a_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 o número total de barras utilizadas. Supondo existir n padrões viáveis, introduzimos uma variável x cujos elementos são inteiros x_j ($j = 1, \ldots, n$), que indicam quantas vezes o padrão j é selecionado. Assim, denotando por A a matriz $m \times n$ cujas colunas são os vetores $a_1, \ldots, a_n$, e representando por d o vetor das demandas, o problema pode ser assim formulado:

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &Ax = d \\ &x_j \geq 0 \text{ e inteiro} \qquad j = 1, \ldots, n. \end{aligned} \tag{3.1}$$

A formulação acima traz consigo duas dificuldades em termos computacionais. A primeira é determinar a matriz A (que pode ter um número exponencial de colunas); a segunda é resolver um problema de programação linear inteira (que é sabido ser $\mathcal{NP}$-difícil).

Para se desvencilhar destas dificuldades, Gilmore e Gomory propuseram o método de geração de colunas que consiste em resolver a relaxação linear de (3.1):

$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &Ax = d \\ &x_j \geq 0 \qquad j = 1, \ldots, n. \end{aligned} \tag{3.2}$$

gerando gradativamente as colunas da matriz A. Podemos iniciar a resolução de (3.2) tomando como base da matriz A, que chamaremos de B, a matriz identidade de ordem m. No início da primeira iteração, uma solução viável é $x = d$.

Em cada iteração determinamos um vetor-linha y tal que $yB = c$, onde c é o vetor-linha com elementos $c_i = 1$ ($i = 1, \ldots, m$). Utilizando y, geramos uma nova coluna resolvendo o seguinte *problema da mochila*.

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{m} y_i z_i \\ &\textstyle\sum_{i=1}^{m} z_i l_i \leq L \\ &z_i \geq 0 \text{ e inteiro} \qquad i = 1, \ldots, m\ . \end{aligned} \tag{3.3}$$

Após resolver (3.3), se $yz > 1$, então resolvemos $Bw = z$ e calculamos:

$$t = min(\tfrac{x_j}{w_j} \mid 1 \leq j \leq m,\ w_j > 0) \tag{3.4}$$

e

$$s = min(j \mid 1 \leq j \leq m,\ \tfrac{x_j}{w_j} = t). \tag{3.5}$$

Em seguida substituímos os valores da coluna s de B pelos valores do vetor z (nova coluna), obtendo uma nova base B'. Utilizando t e a solução corrente x, calculamos uma nova solução viável x', como delineado na proposição a seguir.

Proposição 3.1. *Seja x solução viável de (3.2), t solução de (3.4), $x'_j = x_j - w_j t$ $(j = 1, \ldots, m)$, $x'_j = 0$ $(j = m+1, \ldots, n)$ e $x'_s = t$. Então x' é solução viável de (3.2).*

Prova. Primeiramente vamos mostrar que $x'_j \geq 0$ $(1 \leq j \leq n)$. Observe que $t \geq 0$, pois $x_s \geq 0$ e $w_s > 0$, portanto $x'_s \geq 0$. Resta mostrar que $x'_j \geq 0$ $(j = 1, \ldots, m)$. Note que $t = \frac{x_s}{w_s}$ e, para todo $1 \leq j \leq m$, $\frac{x_j}{w_j} \geq \frac{x_s}{w_s}$, logo $x_j w_s - x_s w_j \geq 0$. Dessa forma, temos que:

$$x'_j = x_j - w_j t = x_j - w_j \tfrac{x_s}{w_s} = \tfrac{x_j w_s - x_s w_j}{w_s} \geq 0.$$

Vamos mostrar agora que $Ax' = d$. Façamos $x_b = [x_1, \ldots, x_m]$ e $x'_b = [x'_1, \ldots, x'_m]$. Observe que $Bx = d$ e, para toda coluna $j > m$, $x'_j = 0$. Ademais, $x_s - w_s t = x_s - w_s \frac{x_s}{w_s} = 0$. Assim:

$$Ax' = B'x'_b = B(x_b - wt) + zt = Bx_b - Bwt + zt = d - zt + zt = d.$$

□

O valor da função objetivo de (3.2) no ponto x' não é pior que o valor da função objetivo de (3.2) no ponto x, conforme demonstramos abaixo.

Proposição 3.2. *Para x e x' mencionados na proposição 3.1, temos que $\sum_{j=1}^{n} x'_j \leq \sum_{j=1}^{n} x_j$.*

Prova. Façamos novamente $x_b = [x_1, \ldots, x_m]$ e $x'_b = [x'_1, \ldots, x'_m]$. Lembrando que $x_s - w_s t = 0$, $yB = c$, $Bw = z$, $t \geq 0$ e $yz > 1$, temos que $1 - yz < 0$. Dessa forma:

$$\sum_{j=1}^{n} x'_j = cx'_b = c(x_b - wt) + x'_s = cx_b - cwt + t$$
$$= \sum_{j=1}^{n} x_j - yBwt + t = \sum_{j=1}^{n} x_j + t(1 - yz) \leq \sum_{j=1}^{n} x_j.$$

□

Começamos então uma nova iteração com $x = x'$ e $B = B'$.

Se $yz \leq 1$, então a solução x corrente é ótima. Para demonstrar este fato podemos usar o dual de (3.2), formulado a seguir.

$$\begin{aligned} &\min \textstyle\sum_{i=1}^{m} y_i d_i \\ &yA \leq c' \end{aligned} \tag{3.6}$$

onde c' é o vetor-coluna com elementos $c'_i = 1$ $(j = 1, \ldots, n)$.

O *Teorema das Folgas Complementares*, enunciado abaixo de uma forma conveniente aos nossos propósitos, estabelece relações entre um par de soluções ótimas de (3.2) e (3.6).

Teorema 3.3. (Folgas Complementares) *Dada uma solução viável x de (3.2) e uma solução viável y de (3.6), x e y são soluções ótimas de (3.2) e (3.6), respectivamente, se e somente se $x_j(c_j - yA_j) = 0$ $(j = 1, \ldots, n)$ e $(d_i - A_i x)y_i = 0$ $(i = 1, \ldots, m)$.*

Uma demonstração deste teorema pode ser obtida em [7], pelo que deixamos de transcrevê-la aqui. Agora estamos em condições de provar que se a solução de (3.3) tiver valor menor ou igual a 1 então x é solução ótima de (3.2).

Proposição 3.4. *Dada uma solução viável x de (3.2) e um vetor y tal que $yB = c$, se a solução z de (3.3) for tal que $yz \leq 1$ então x é solução ótima de (3.2).*

Prova. Para todo $1 \leq j \leq n$, temos que $yA_j \leq 1$, caso contrário tal A_j seria uma solução de (3.3) com valor maior que 1. Portanto y é solução viável de (3.6). Observe que para todo $x_j > 0$, A_j é uma coluna de B; logo $yA_j = c_j$, pois $yB = c$. Ademais, $A_i x = b_i$, pois x é solução viável de (3.2). Portanto, pelo *Teorema das Folgas Complementares*, x é solução ótima de (3.2). $\square$

Temos então um algoritmo que chamaremos de *Simplex Revisado com Geração de Colunas para o PCE_1*, ou simplesmente *$SimplexGC_1$*, cujos detalhes são apresentados a seguir.

Algoritmo SimplexGC$_1$

Entrada: $(L, l = [l_1, \ldots, l_m], d = [d_1, \ldots, d_m])$.

Saída: Uma solução ótima de: $\min \sum_{j=1}^{n} x_j$

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

(onde A é a matriz dos padrões viáveis)

1 Faça $x = d$, $c_i = 1$ $(i = 1, \ldots, m)$ e seja B a matriz identidade de ordem m.
2 Resolva $yB = c$.
3 Gere uma nova coluna resolvendo:

$$\begin{aligned} & \max yz \\ & lz \leq L \\ & z_i \geq 0 \text{ e inteiro} \qquad i = 1, \ldots, m\ . \end{aligned}$$

4 Se $yz \leq 1$, devolva x e pare.
5 Caso contrário, resolva $Aw = z$.
6 Calcule $t = min(\frac{x_i}{w_i} \mid 1 \leq j \leq m,\ w_j > 0)$ e $s = min(i \mid 1 \leq j \leq m,\ \frac{x_j}{w_j} = t)$.
7 Para $i = 1$ até m faça
  7.1 $B_{i,s} = z_i$.
  7.2 Se $i = s$ então $x_i = t$; caso contrário, $x_i = x_i - w_i t$.
8 Retorne ao passo 2.

Para executar o passo 3 do algoritmo SimplexGC$_1$ podemos utilizar o algoritmo MOCHILA, explicado a seguir.

Podemos exigir que toda solução ótima de (3.3) satisfaça

$$L - \sum_{i=1}^{m} l_i z_i < l_m, \tag{3.7}$$

pois se isso não ocorresse seria possível incrementar z_m de uma unidade. Chamamos as soluções viáveis de (3.3) que satisfazem (3.7) de *soluções sensatas*. O algoritmo MOCHILA, consiste basicamente em enumerar as soluções sensatas, começando por aquelas que parecem ser mais promissoras, buscando entre elas uma solução ótima.

Podemos aperfeiçoar o algoritmo de modo a evitar que soluções claramente inferiores à melhor solução já encontrada sejam analisadas. Tal procedimento limita sobremaneira o espaço de busca, tornando o algoritmo sensivelmente mais rápido.

Algoritmo MOCHILA

Entrada: $(L, l_1, \ldots, l_m, y_1, \ldots, y_m)$.
Saída: $z_1, \ldots, z_m \in I\!N$ tais que $\sum_{i=1}^{m} l_i z_i \leq L$ e $\sum_{i=1}^{m} y_i z_i$ é máximo.

1 Faça $M = 0$ e $k = 0$.
2 Ordene os itens em ordem não-crescente de $\frac{y_i}{l_i}$ (custo relativo).
3 Para $j = k + 1$ até m faça
3.1 $z_j = \lfloor (L - \sum_{i=1}^{j-1} l_i z_i)/l_j \rfloor$.
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 Procure $k = max(\{i \mid 1 \leq i < m \ e \ z_i > 0 \ e \ \sum_{j=1}^{i-1} y_j z_j + y_i(z_i - 1) + \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i-1} l_j z_j - l_i(z_i - 1)) > M\} \cup \{0\})$.
6 Se $k = 0$ então devolva z^* e pare.
7 Faça $z_k = z_k - 1$ e retorne ao passo 3.

Observe que no passo 2 colocamos os itens em ordem não-crescente de custo relativo. No passo 5, ao procurar um valor para k, consideramos $\bar{z}$ tais que $\bar{z}_i = z_1$ $(i = 1, \ldots, k-1)$ e $\bar{z}_k = z_k - 1$. A idéia é determinar *a priori*, quaisquer que sejam os valores de $\bar{z}_{k+1}, \ldots, \bar{z}_m$, se $\bar{z}$ tem chance de ser melhor que z^*. Cada item $k+1, \ldots, m$ tem custo relativo menor ou igual a $\frac{y_{k+1}}{l_{k+1}}$, portanto $\sum_{i=k+1}^{m} y_i \bar{z}_i \leq \frac{y_{k+1}}{l_{k+1}}(L - \sum_{i=1}^{k-1} l_i \bar{z}_i - l_k z_k)$. Dessa forma, para todas as soluções $\bar{z}$ com os valores das primeiras k posições fixadas como acabamos de descrever, vale que

$$\sum_{i=1}^{m} y_i \bar{z}_i \leq \sum_{i=1}^{k} y_i \bar{z}_i \ + \ \frac{y_{k+1}}{l_{k+1}}(L - \sum_{i=1}^{k} l_i \bar{z}_i). \tag{3.8}$$

Como desejamos que $\sum_{i=1}^{m} y_i \bar{z}_i > M$, podemos impor que a seguinte desigualdade seja satisfeita ao escolher o valor de k, caso contrário $\bar{z}$ não tem chance ser melhor que z^*.

$$\sum_{j=1}^{i-1} y_j \bar{z}_j + y_i(\bar{z}_i - 1) \ + \ \frac{y_{i+1}}{l_{i+1}}(L - \sum_{j=1}^{i-1} l_j \bar{z}_j - l_i(\bar{z}_i - 1)) > M). \tag{3.9}$$

Podemos dizer que o algoritmo MOCHILA executa uma busca em profundidade numa árvore, onde cada ramo da árvore (um caminho desde a raiz até uma folha) representa um solução sensata. A desigualdade (3.9) permite podar esta árvore fazendo com que os algoritmo encontre uma solução mais rapidamente. Trata-se portanto de um algoritmo de enumeração tipo *branch-and-bound*. Apesar de ser exponencial no pior caso, este algoritmo se mostra bastante satisfatório na prática.

Voltando ao algoritmo SimplexGC_1, a condição suficiente para que a nova coluna possa entrar na base é $\sum_{i=1}^{m} y_i z_i > 1$. Dessa forma, não precisamos resolver (3.3) até a otimalidade. Implementamos o SimplexGC_1 e os testes práticos que realizamos (ver [8]) indicam que resolver (3.3) até a otimalidade faz com que seja necessário gerar um número menor de colunas, compensando assim o esforço para achar soluções ótimas.

Ainda sobre a quantidade de colunas geradas pelo SimplexGC_1, em 1972, Klee e Minty [16] exibiram uma família de problemas de programação linear, criada artificialmente a partir de deformações do cubo n-dimensional, para a qual o algoritmo Simplex, com regra de pivotação de Dantzig-Wolfe, executa um número exponencial de iterações. No entanto, testes computacionais indicam que o número médio de colunas geradas pelo SimplexGC_1 é $\mathcal{O}(m^2)$. Isto parece estar de acordo com resultados teóricos obtidos por diversos pesquisadores [1, 5] a respeito do tempo requerido pelo Simplex no caso médio.

Infelizmente a solução ótima de (3.2) não será necessariamente inteira. Para contornar este problema, Gilmore e Gomory propuseram, após achar a solução ótima, arredondar para cima o valor das variáveis (isto é, arredondar cada x_i para $\lceil x_i \rceil$), obtendo assim uma solução inteira. No entanto, este procedimento pode acarretar a produção de itens em quantidade superior à demanda, e conduz a uma solução inteira eventualmente longe da ótima. Diversas heurísticas têm sido propostas para obter uma solução inteira mais próxima da ótima [8, 19].

Discutiremos agora como adaptar SimplexGC_1 para o $PCGD_2$. No caso unidimensional, gerar uma nova coluna equivale a encontrar uma solução de (3.3) com valor maior que 1. Uma solução de (3.3) indica uma coleção de itens com a seguinte propriedade: a soma dos comprimentos dos itens da coleção não excede o comprimento da barra. Esta coleção é composta por x_i itens de comprimento l_i $(i = 1, \ldots, m)$. Esta propriedade garante a existência de um padrão contendo a coleção de itens.

No caso bidimensional, a propriedade acima não garante a existência de um padrão contendo a coleção de itens expressa pela solução de (3.3). Mesmo que alterássemos a desigualdade $\sum_{i=1}^{m} z_i l_i \leq L$ para $\sum_{i=1}^{m} z_i l_i c_i \leq LC$, ou seja, exigíssemos que a soma das áreas dos itens da coleção não excedesse a área da placa, ainda assim, tal propriedade não garantiria a existência de um padrão factível. Existe a necessidade de impedir a sobreposição dos itens dentro do padrão. Para tratar disso, usamos a convenção adotada na Subseção 2.2 sobre a representação de padrões.

O algoritmo $PCGV_2PD$, visto anteriormente, gera padrões guilhotináveis que possuem as propriedades citadas na Subseção 2.2, ou seja, padrões guilhotináveis factíveis. Além disso, se cada item i tem valor y_i e aparece z_i vezes vezes num padrão produzido pelo $PCGV_2PD$ $(i = 1, \ldots, m)$, então $\sum_{i=1}^{m} y_i z_i$ é máximo. Isto é tudo o que precisamos para gerar novas colunas. Abaixo detalhamos o algoritmo SimplexGC_2 que resolve a formulação (3.2) correspondente a uma instância do PCGD_2.

Algoritmo SimplexGC$_2$

Entrada: $(L, C, l = [l_1, \ldots, l_m], c = [c_1, \ldots, c_m], d = [d_1, \ldots, d_m])$.

Saída: Uma solução ótima de:
$$\begin{aligned} &\min \textstyle\sum_{j=1}^{n} x_j \\ &Ax = d \\ &x_j \geq 0 \qquad j = 1, \ldots, n. \end{aligned}$$

(onde A é a matriz dos padrões viáveis)

1 Faça $x = d$, $c_i = 1$ $(i = 1, \ldots, m)$ e seja B a matriz identidade de ordem m.
2 Resolva $yB = c$.
3 Gere uma nova coluna executando o algoritmo $\mathrm{PCGV}_2\mathrm{PD}$ com parâmetros L, C, l, c, y
sendo z a solução devolvida.
4 Se $yz \leq 1$, devolva x e pare.
5 Caso contrário, resolva $Aw = z$.
6 Calcule $t = min(\frac{x_i}{w_i} \mid 1 \leq j \leq m,\ w_j > 0)$ e $s = min(i \mid 1 \leq j \leq m,\ \frac{x_j}{w_j} = t)$.
7 Para $i = 1$ até m faça
7.1 $B_{i,s} = z_i$.
7.2 Se $i = s$ então $x_i = t$; caso contrário, $x_i = x_i - w_i t$.
8 Retorne ao passo 2.

Como já mencionamos anteriormente, a solução de (3.2) pode não ser inteira. Na próxima subseção discutimos como lidar com esta dificuldade.

3.2 O Algoritmo PCGD$_2$GC

Explicaremos agora como podemos obter uma solução inteira a partir de uma solução fracionária encontrada pelo SimplexGC$_2$. O processo é iterativo. Cada iteração consiste basicamente resolver (3.2) com o SimplexGC$_2$ e depois arredondar para baixo o valor das variáveis $x_1, \ldots, x_m$, obtendo uma nova solução $x_1^*, \ldots, x_m^*$. Adotando esta nova solução, uma parte da demanda dos itens não será atendida. Mais precisamente, a demanda não atendida de cada item i será $d_i^* = d_i - \sum_{j=1}^{m} A_{i,j} x_j^*$.

Fazendo $d^* = [d_1^*, \ldots, d_m^*]$, temos então uma instância residual $I^* = (L, C, l, c, d^*)$. Se $x_i^* > 0$ para algum $1 \leq i \leq m$, alguma parte da demanda foi atendida pela solução x^*. Neste caso, começamos uma nova iteração com I^*. Se $x_i^* = 0$ para todo $1 \leq i \leq m$, nenhuma parte da demanda foi atendida por x^*. Resolvemos então este último problema residual com o algoritmo FFDH, explicado na próxima subseção.

Apresentamos a seguir o algoritmo PCGD$_2$GC que implementa o processo iterativo que acabamos de descrever.

Algoritmo PCGD$_2$GC

Entrada: Uma instância $I = (L, C, l = [l_1, \ldots, l_m], c = [c_1, \ldots, c_m],$
$d = [d_1, \ldots, d_m])$ do PCGD$_2$.

Saída: Uma solução para I

1 Execute o algoritmo SimplexGC$_2$ com parâmetros L, C, l, c, d.
2 Para $i = 1$ até m faça $x_i^* = \lfloor x_i \rfloor$.
3 Se $x_i^* > 0$ para algum $1 \leq i \leq m$ então
  3.1 Devolva A 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 $m' = m' + 1$, $l_{m'} = l_i$, $c_{m'} = c_i$ e $d_{m'} = d_i$.
  3.5 Se $m' = 0$ então pare.
  3.6 Faça $m = m'$ e volte ao passo 1.
4 Devolva a solução do algoritmo FFDH executado com parâmetros L, C, l, c.

Observe que em cada iteração ou uma parte da demanda é atendida ou passamos para o passo 4. Isto garante que após um número finito de iterações, toda a demanda terá sido atendida (uma parte dela eventualmente no passo 4). Na realidade, sobre o número de iterações do algoritmo, vale o seguinte resultado.

Proposição 3.5. *O passo 3.6 do algoritmo PCGD$_2$GC é executado no máximo m vezes.*

Prova. Ao executar o passo 3.6, seja d^* a demanda da instância residual. Note que $d^* = Ax - Ax^*$. Faça $x' = x - x^*$. Observe que tal x' é solução da formulação (3.2) correspondente à instância residual pois $Ax' = Ax - Ax^* = d^*$.

Se uma nova iteração é iniciada, então $\sum_{i=1}^m x^* \geq 1$. Além disso, $\sum_{i=1}^m x' = \sum_{i=1}^m x' - \sum_{i=1}^m x^*$. Dessa forma, temos que $\sum_{i=1}^m x' \leq \sum_{i=1}^m x - 1$. Isto significa que ao final de cada iteração o valor de uma solução ótima da instância residual decresce de pelo menos 1. Como no final da primeira iteração $\sum_{i=1}^m x' < m$, após no máximo m iterações teremos $\sum_{i=1}^m x' < 1$. Portanto, na iteração seguinte $\sum_{i=1}^m x < 1$ e $\sum_{i=1}^m x^* = 0$. Dessa forma, o algoritmo não executará o passo 3.6, prosseguindo então para o passo 4. □

Vimos assim que o número de iterações do algoritmo é polinomial. Ademais, o número de colunas geradas no SimplexGC$_2$ também é polinomial, no caso médio, conforme discutimos na Subseção 3.1. No entanto, gerar uma nova coluna com o algoritmo PCGV$_2$PD requer tempo $\mathcal{O}((LC + m)(L + C))$, que pode ser exponencial em m. Portanto não podemos afirmar que o algoritmo PCGD$_2$GC é polinomial, mesmo considerando apenas o caso médio.

Resta-nos agora discutir o algoritmo FFDH utilizado no passo 4 do PCGD$_2$GC.

3.3 O Algoritmo FFDH

O algoritmo *First Fit Decreasing Height* (FFDH) foi proposto por Coffman *et al.* [9] para o PCGD$_2$, onde a demanda de todos os itens é exatamente 1. Trata-se de um algoritmo de aproximação com limite de desempenho assintótico 1.7 e que requer tempo $\mathcal{O}(m \log m)$.

Fazendo $D = \sum_{i=1}^{m} d_i$, é correto dizer que o FFDH requer tempo $\mathcal{O}(D \log D)$. Se as demandas são maiores ou iguais a 1, o FFDH pode não ser mais polinomial. Supondo que todos os dados de uma instância do PCGD$_2$ podem ser codificados com k bits, que todas as demandas são iguais a 2^k - 1 e que m é $\mathcal{O}(k)$, o tamanho da entrada é $\mathcal{O}(k)$, e $D = 2^k m - m$. Para tal instância o FFDH requer tempo exponencial. Mesmo assim, escolhemos usar o FFDH para obter uma solução inteira quando a solução fornecida pelo SimplexGC$_2$ para uma instância residual é constituída apenas de valores menores que 1.

Para entendermos a estratégia usada pelo FFDH, convém antes explicarmos outro algoritmo bastante similar, o *Next Fit Decreasing Height* (NFDH). Neste algoritmo dizemos que um item é *empacotado*, mas isto é equivalente a dizer que a placa é cortada produzindo um retângulo de dimensões iguais às do item. No NFDH, primeiramente colocamos os itens em ordem não-crescente de altura. Em seguida, empacotamos o primeiro item, na ordem estabelecida, no canto inferior esquerdo da placa.

Cada item i seguinte é empacotado imediatamente à direita do item anterior, se possível. Caso contrário, dizemos que o *nível* corrente está cheio e precisamos criar um novo nível e empacotar o item i no canto esquerdo do novo nível. Se a distância do nível corrente para o lado superior da placa for menor ou igual a c_i (altura do item i), o novo nível é criado logo acima do nível corrente, caso contrário, passamos a utilizar uma nova placa e o novo nível é criado a partir da canto inferior esquerdo da placa. Este procedimento é ilustrado pela Figura 4a, onde os níveis são as regiões delimitadas por duas linhas pontilhadas consecutivas.

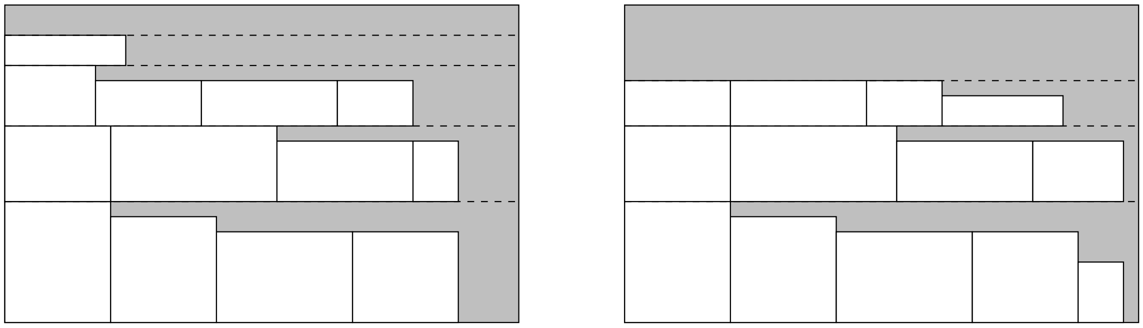


Figura 4:
(a) Exemplo de solução do NFDH (b) Exemplo de solução do FFDH

No FFDH é introduzida uma pequena variação. Cada item é empacotado no primeiro nível no qual ele caiba, considerando a ordem em que os níveis foram criados. A Figura 4b mostra a solução encontrada pelo FFDH para a mesma instância da Figura 4a. A seguir descrevemos o FFDH.

Algoritmo FFDH

Entrada: Uma instância $I = (L, C, l = [l_1, \ldots, l_n], c = [c_1, \ldots, c_n])$ do PCGD_2.
Saída: Uma solução para I.

Ordene os itens em ordem não-crescente de altura e faça $\mathcal{N} = 0$ e $\mathcal{P} = 1$.
Para $i = 1$ até n
  Faça $k = min(\{j \mid 1 \leq j \leq \mathcal{N} \text{ e } largura(j) \leq L - l_i\} \cup \{\mathcal{N} + 1\})$.
  Se $k = \mathcal{N} + 1$
    Se $altura(k-1) + c_i > C$
      Faça $\mathcal{P} = \mathcal{P} + 1$.
    Crie o nível k na placa $\mathcal{P}$ e faça $altura(k) = c_i$ e $\mathcal{N} = k$.
  Empacote o item i no nível k e faça $largura(k) = largura(k) + l_i$.

Observe que transformamos uma instância do PCGD_2 numa entrada adequada para o FFDH (onde todos os itens têm demanda igual a 1) fazendo d_i cópias de cada item i $(i = 1, \ldots, m)$. No descrição do FFDH, $\mathcal{N}$ é o índice do último nível criado e $\mathcal{P}$ é o indice da última placa utilizada. Além disso, $largura(j)$ e $altura(j)$ representam a largura e a altura do nível j, respectivamente.

4 Resultados Computacionais

Avaliamos os algoritmos $\mathrm{PCGV_2PD}$ e $\mathrm{PCGD_2GC}$, propostos nas seções 2 e 3, resolvendo as instâncias do problema de corte de guilhotina bidimensional disponíveis na OR-LIBRARY[3]. Tal biblioteca é uma coleção de instâncias de testes para uma grande variedade de problemas na área de pesquisa operacional. Uma descrição desta biblioteca e de seus objetivos é dada em [4].

Os algoritmos foram implementados utilizando-se a linguagem C e os testes executados numa estação 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). Utilizamos o software *Xpress-MP* [20] para resolver os sistemas de equações lineares que aparecem nos passos 2 e 5 do algoritmo $\mathrm{SimplexGC_2}$. Registramos aqui nossos agradecimentos à *Dash Optimization, Inc* que nos cedeu gratuitamente o *Xpress-MP* como parte de seu Programa de Parceiros Acadêmicos. A seguir apresentamos os resultados dos testes.

4.1 Resolvendo Instâncias do $\mathrm{PCGV_2}$

Na data da elaboração desta monografia, estavam disponíveis na OR-LIBRARY 13 instâncias do $\mathrm{PCGV_2}$, denominadas $gcut1, \ldots, gcut13$. Todas estas instâncias, exceto a instância $gcut13$, já haviam sido resolvidas até a otimalidade [3]. Em todas estas instâncias rotações ortogonais não são permitidas.

Na Tabela 2 apresentamos detalhes das instâncias (quantidade de itens e dimensões da placa), o valor da solução encontrada pelo algoritmo $\mathrm{PCGV_2PD}$ (que é uma solução ótima), o percentual de desperdício e o tempo médio gasto para resolvê-las. Cada instância foi resolvida 100 vezes e os tempos apresentados foram obtidos calculando-se a média do tempo gasto nestas 100 resoluções.

Destacamos a resolução da instância $gcut13$ cuja solução ótima era desconhecida. Exibimos na Figura 5 a solução encontrada pelo algoritmo $\mathrm{PCGV_2PD}$ para esta instância.

Experimentamos resolver as instâncias $gcut1, \ldots, gcut13$ permitindo rotações (chamamos tais instâncias de $gcut1r, \ldots, gcut13r$). Os resultados obtidos são apresentados na Tabela 3.

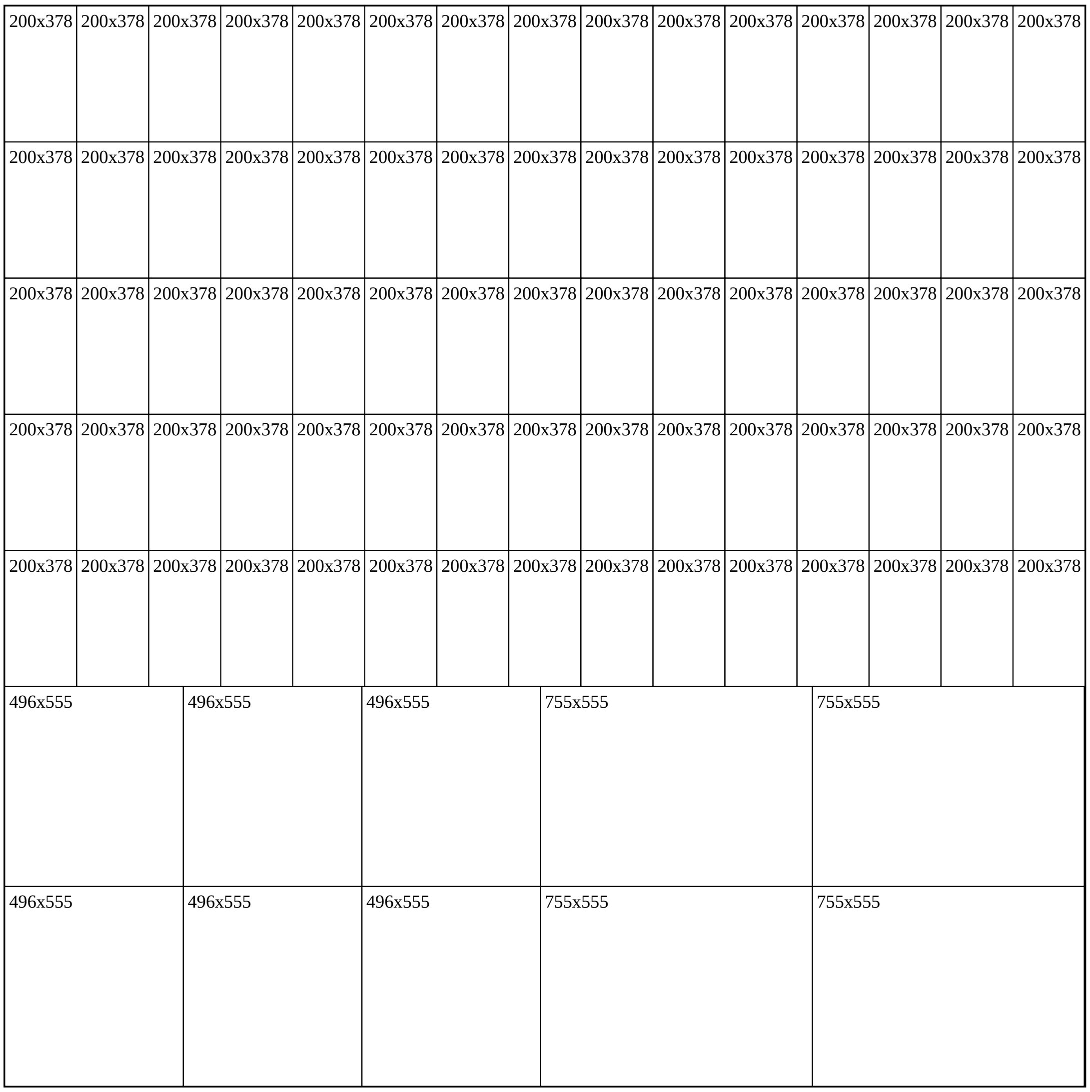
[3]http://mscmga.ms.ic.ac.uk/info.html

Instância	Quantidade de Itens	Dimensões da Placa	Solução Ótima	Desperdício	Tempo (seg)
gcut1	10	(250, 250)	56460	9,664%	0,003
gcut2	20	(250, 250)	60536	3,142%	0,010
gcut3	30	(250, 250)	61036	2,342%	0,012
gcut4	50	(250, 250)	61698	1,283%	0,022
gcut5	10	(500, 500)	246000	1,600%	0,004
gcut6	20	(500, 500)	238998	4,401%	0,008
gcut7	30	(500, 500)	242567	2,973%	0,017
gcut8	50	(500, 500)	246633	1,347%	0,062
gcut9	10	(1000, 1000)	971100	2,890%	0,006
gcut10	20	(1000, 1000)	982025	1,798%	0,009
gcut11	30	(1000, 1000)	980096	1,990%	0,066
gcut12	50	(1000, 1000)	979986	2,001%	0,140
gcut13	50	(3000, 3000)	8997780	0,025%	165,832

Tabela 2: Soluções do PCGV$_2$PD para as instâncias $gcut1, \ldots, gcut13$.

Instância	Quantidade de Itens	Dimensões da Placa	Solução Ótima	Desperdício	Tempo (seg)
gcut1r	10	(250, 250)	58136	6,982%	0,008
gcut2r	20	(250, 250)	60611	3,022%	0,021
gcut3r	30	(250, 250)	61626	1,398%	0,026
gcut4r	50	(250, 250)	62265	0,376%	0,042
gcut5r	10	(500, 500)	246000	1,600%	0,019
gcut6r	20	(500, 500)	240951	3,620%	0,032
gcut7r	30	(500, 500)	245866	1,654%	0,053
gcut8r	50	(500, 500)	247787	0,885%	0,135
gcut9r	10	(1000, 1000)	971100	2,890%	0,023
gcut10r	20	(1000, 1000)	982025	1,798%	0,071
gcut11r	30	(1000, 1000)	980096	1,990%	0,270
gcut12r	50	(1000, 1000)	979986	2,001%	0,140
gcut13r	50	(3000, 3000)	9000000	0,000%	280,247

Tabela 3: Soluções do PCGV$_2$PD para as instâncias $gcut1r, \ldots, gcut13r$.



Instancia: gcut13 L: 3000 C: 3000 Valor: 8997780

Figura 5:
Solução ótima encontrada pelo algoritmo $PCGV_2$ para a instância *gcut13*

4.2 Resolvendo Instâncias do $PCGD_2$

Não encontramos instâncias do $PCGD_2$ na OR-LIBRARY. Decidimos então usar as instâncias $gcut1, \ldots, gcut12$, atribuindo a cada item uma demanda entre 1 e 100. Chamamos tais instâncias de $gcut1d, \ldots, gcut12d$. As demandas foram geradas utilizando-se a função *rand* da linguagem PERL (versão 5.6.1) [18] com a semente sendo inicializada com o número $(L + C) * m + m$, onde L e C são a largura e a altura da placa, respectivamente, e m é a quantidade de itens. Por exemplo, para a instância $gcut1d$ a semente da função *rand* foi o número 5010.

Apresentamos na Tabela 4 o valor da solução encontrada pelo algoritmo $PCGD_2GC$, o limite inferior (LB) fornecido pela solução de (3.2) para o valor de uma solução inteira ótima, a diferença percentual entre a solução do $PCGD_2GC$ e o valor do LB, a quantidade de colunas geradas, o tempo gasto, o valor da solução encontrada pelo FFDH e o ganho percentual da solução do $PCGD_2GC$ em relação à solução do FFDH. Cada instância foi resolvida 10 vezes e os tempos apresentados foram obtidos calculando-se a média do tempo gasto nestas 10 resoluções.

Resolvemos também as instâncias $gcut1d, \ldots, gcut12d$ permitindo rotações. Chamamos tais instâncias de $gcut1dr, \ldots, gcut12dr$. Apresentamos na Tabela 5 os resultados obtidos.

Percebemos que, tanto no caso sem rotações como no caso com rotações, a solução do $PCGD_2GC$ é bastante próxima do valor do limite inferior fornecido pela solução de (3.2). Além disso, o ganho percentual da solução do $PCGD_2GC$ em relação à solução do FFDH é bastante significativo.

Nesta monografia investigamos a aplicação de programação dinâmica e do método de geração de colunas para o $PCGV_2PD$ e o $PCGD_2GC$, respectivamente. A estratégia que usamos na programação dinâmica difere de outras mencionadas na literatura. Propusemos então os algoritmos $PCGV_2PD$ e $PCGD_2GC$. Os resultados obtidos nos testes computacionais indicam que estes algoritmos são capazes de lidar satisfatoriamente com instâncias encontradas na literatura consideradas representativas de instâncias do mundo real.

Instância	Solução do $PCGD_2GC$	*Lower Bound* (LB)	Diferença em relação ao LB	Tempo (seg)	Colunas Geradas	Solução do FFDH	Ganho em relação ao FFDH
gcut1d	294	294	0,000%	0,023	9	385	23,636%
gcut2d	345	345	0,000%	0,843	93	450	23,333%
gcut3d	336	332	1,204%	2,311	213	488	31,148%
gcut4d	843	836	0,837%	12,426	671	1151	26,759%
gcut5d	198	197	0,507%	0,067	19	246	19,512%
gcut6d	345	343	0,583%	0,540	101	456	24,342%
gcut7d	594	592	0,337%	3,438	277	783	24,138%
gcut8d	697	691	0,868%	34,456	800	963	27,622%
gcut9d	132	131	0,763%	0,066	17	162	18,519%
gcut10d	293	293	0,000%	0,237	29	415	29,398%
gcut11d	334	330	1,212%	10,678	202	472	29,237%
gcut12d	675	672	0,446%	72,948	839	913	26,068%

Tabela 4: Soluções do $PCGD_2GC$ para as instâncias $gcut1d, \ldots, gcut12d$.

Instância	Solução do $PCGD_2GC$	*Lower Bound* (LB)	Diferença em relação ao LB	Tempo (seg)	Colunas Geradas	Solução do FFDH	Ganho em relação ao FFDH
gcut1dr	292	291	0,343%	0,052	9	385	24,156%
gcut2dr	286	282	1,418%	4,722	174	450	36,444%
gcut3dr	320	313	2,236%	7,456	311	488	34,426%
gcut4dr	844	836	0,956%	18,124	535	1151	26,672%
gcut5dr	177	174	1,724%	0,408	24	246	28,049%
gcut6dr	302	301	0,332%	3,740	184	456	33,772%
gcut7dr	547	542	0,992%	8,607	188	783	30,140%
gcut8dr	655	650	0,769%	96,844	800	963	31,983%
gcut9dr	124	122	1,639%	0,654	17	162	23,457%
gcut10dr	270	270	0,000%	1,943	29	415	34,940%
gcut11dr	302	298	1,342%	49,074	202	472	36,017%
gcut12dr	607	601	0,998%	384,337	839	913	33,516%

Tabela 5: Soluções do $PCGD_2GC$ para as instâncias $gcut1dr, \ldots, gcut12dr$.

Referências

[1] I. Adler, N. Megiddo, and M. J. Todd, *New results on the average behavior of simplex algorithms*, Bull. Amer. Math. Soc. (N.S.), 11 (1984), pp. 378–382.

[2] V. Bafna, S. Muthukrishnan, and R. Ravi, *Computing similarity between RNA strings*, in Combinatorial Pattern Matching (Espoo, 1995), vol. 937 of Lecture Notes in Comput. Sci., Springer, Berlin, 1995, pp. 1–16.

[3] J. E. Beasley, *Algorithms for unconstrained two-dimensional guillotine cutting*, Journal of the Operational Research Society, 36 (1985), pp. 297–306.

[4] J. E. Beasley, *Or-library: distributing test problems by electronic mail*, Journal of the Operational Research Society, 41 (1990), pp. 1069–1072.

[5] K.-H. Borgwardt, *Probabilistic analysis of the simplex method*, in Mathematical developments arising from linear programming (Brunswick, ME, 1988), vol. 114 of Contemp. Math., Amer. Math. Soc., Providence, RI, 1990, pp. 21–34.

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

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

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

[9] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan, *Performance bounds for level oriented two-dimensional packing algorithms*, SIAM Journal on Computing, 9 (1980), pp. 808–826.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, Cambridge, MA, second ed., 2001.

[11] 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 Francisco, 1979.

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

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

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

[15] M. Hifi, *Exact algorithms for large-scale unconstrained two and three staged cutting problems*, Comput. Optim. Appl., 18 (2001), pp. 63–88.

[16] V. Klee and G. J. Minty, *How good is the simplex algorithm?*, in Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin), New York, 1972, Academic Press, pp. 159–175.

[17] J. Riehme, G. Scheithauer, and J. Terno, *The solution of two-stage guillotine cutting stock problems having extremly varying order demands*, TR MATH-NM-05-1995, TU Dresden, 1995.

[18] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, O'Reilly & Associates, 3rd ed., July 2000.

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

[20] Xpress, *Xpress Optimizer Reference Manual*, DASH Optimization, Inc, 2002.