UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

SISTEMA DE APOIO AO CORTE BIDIMENSIONAL GUILHOTINADO APLICADO AO CORTE DE CHAPAS DE PAPELÃO UTILIZANDO PROGRAMAÇÃO LINEAR INTEIRA

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

MARCOS PAULO ZIMMERMANN

BLUMENAU, JUNHO/2002

2002/1-54

SISTEMA DE APOIO AO CORTE BIDIMENSIONAL GUILHOTINADO APLICADO AO CORTE DE CHAPAS DE PAPELÃO UTILIZANDO PROGRAMAÇÃO LINEAR INTEIRA

MARCOS PAULO ZIIMMERMANN

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Maurício Capobianco Lopes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Mauro Marcelo Mattos

_____

Prof. Ricardo Alencar Azambuja

AGRADECIMENTOS

Agradeço a todos que presenciaram minha jornada de graduação, dentre amigos, parentes e colegas de trabalho que compreendem o quão árduo caminho percorrido.

Em especial à minha família, a Deus, e à minha chefia profissional.

E principalmente, ao professor Maurício que auxiliou de forma essencial à concepção deste estudo.

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho apresenta o desenvolvimento de um software aplicativo para auxiliar a programação de corte de chapas (placas) de papelão, assim como o controle de estoque de placas e peças derivadas do corte destas placas, considerando o problema do corte bidimensional guilhotinado. Para isto foram utilizados o algoritmo de Herz e o método de geração de colunas para a otimização dos cortes. A especificação foi baseada na análise essencial de sistemas e a implementação foi desenvolvida no ambiente Borland C++ Builder.

ABSTRACT

This work aims a software aplicattion development to assist the cardboard plates cut programming, as well as the plates and parts derived from the cut of these plates supply control, considering the guillotine two-dimensional cut. For this, the Herz algorithm and the column generation method had been used for the cuts optimization. The specification was based on the essential analysis and the implementation was developed in the environment Borland C++ Builder.

SUMÁRIO

1 INTRODUÇÃO

Durante vários anos as teorias e métodos desenvolvidos por matemáticos e cientistas foram arquivados em livros e periódicos especializados e pouco foi utilizado no setor empresarial. Felizmente, contudo, essa situação vem se alterando. É cada vez maior o número de organizações que adotam modelos de otimização no seu dia-a-dia, reduzindo seus custos e, por conseguinte, aumentando os lucros. Além do mais, com a onda crescente de privatizações nos diversos setores da sociedade, a concorrência se fortifica e a sobrevivência dos negócios começa a depender seriamente do desempenho de cada um em relação aos demais. Quem estiver melhor preparado irá, sem dúvida alguma, superar os adversários. Neste contexto, uma das técnicas mais mencionadas atualmente, diz respeito a problemas de otimização.

Segundo Bronson (1985), em problemas de otimização busca-se maximizar ou minimizar uma quantidade específica, chamada objetivo, que depende de um número finito de variáveis de entrada. Estas variáveis podem ser independentes umas das outras ou podem ser relacionadas por meio de uma ou mais restrições. Os problemas de otimização são, com muita freqüência, formulados verbalmente. O procedimento para solucioná-los consiste em modelá-los sob a forma de problema de programação matemática e, em seguida, utilizar diferentes técnicas que possam fornecer tais soluções.

Para Loesch (1999), a programação linear é utilizada na resolução de problemas de maximização (como lucro) ou minimização (como custo) de algum objetivo, atendendo a um conjunto de restrições. Ela parte da modelagem do problema e culmina com a obtenção da solução ótima. As variáveis são representadas por números de ponto flutuante (números não necessariamente inteiros).

Contudo, o trabalho aqui proposto, trata do corte de unidades de peças inteiras com tamanhos pré-definidos e de forma bidimensional (retângulos), sendo, portanto, utilizada a técnica similar à programação linear, chamada de programação inteira, que conforme Loesch (1999), expande o alcance da programação linear permitindo a condição de que algumas (ou todas) as variáveis do modelo sejam números inteiros.

Conforme Nascimento (1999), o problema do corte bidimensional é clássico, e consiste na definição de um *layout* indicando como cortar uma lista de peças retangulares em placas de

tamanho padrão, também retangulares, de um certo produto (vidro, madeira, tecido, papel, etc.), de forma a reduzir a quantidade de placas a ser utilizada, minimizando o desperdício de material, ou, equivalentemente, maximizando o seu aproveitamento. Outras definições atribuem um valor de utilidade para cada peça e procuram maximizar o valor de utilidade total, sem restrições e com restrições no número de peças produzidas.

Como o corte das peças é feito por uma guilhotina, Nascimento (1999) atribui ao problema a definição de corte bidimensional guilhotinado.

Assim, desenvolveu-se um sistema de otimização do *layout* de corte de placas padrão de papelão reciclado em vários itens de diferentes dimensões, em formas retangulares, fabricados pelas indústrias de embalagens, assim como a realização do controle de estoques de itens a serem temporariamente armazenados. O sistema representa uma melhoria para o setor de produção de indústrias que necessitam de *layout* de corte de unidades retangulares (bidimensionais) e utilizam guilhotina para o corte da peça padrão de matéria-prima. A programação de corte de itens integrantes de pedidos facilita o cálculo da melhor forma de cortar as peças bidimensionais, acelerando assim o processo de produção, além de reduzir o desperdício de matéria-prima.

Para o setor industrial o software visa diminuir o tempo de supostas combinações feitas para otimizar o corte das caixas de papelão, assim como diminuir as perdas de matéria-prima. Desta forma, também pode-se controlar mais facilmente o estoque de placas de papelão.

Para o desenvolvimento deste sistema foi utilizado uma das técnicas da Pesquisa Operacional, a programação linear inteira. O sistema foi especificado utilizando-se a metodologia de desenvolvimento de sistemas Análise Essencial de Sistemas, e para a implementação foi utilizada a ferramenta de desenvolvimento Borland C++ Builder.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é apresentar o desenvolvimento de um sistema de apoio ao corte de placas bidimensionais de papelão que utilizam guilhotina, para um setor de produção de caixas de papelão, utilizando a programação linear inteira para resolução dos problemas propostos.

O trabalho tem como objetivos específicos:

a) disponibilizar um algoritmo que, utilizando a técnica de programação linear inteira, facilite o processo de corte das caixas de papelão, fornecendo um *layout* da disposição dos retângulos otimizado, para corte na guilhotina;
b) disponibilizar um módulo de controle de estoques que mantenha os dados principais dos itens cortados, da matéria-prima e da programação de cortes de itens.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está organizado da seguinte forma:

O primeiro capítulo trata da introdução, contextualização, justificativa e objetivos do trabalho.

O segundo apresenta o problema a ser tratado e uma visão sobre a complexidade do mesmo.

O terceiro capítulo apresenta os objetivos e áreas de aplicação da pesquisa operacional, a descrição do problema de corte, a fundamentação na programação matemática e os algoritmos a serem utilizados na solução do problema.

O quarto apresenta o desenvolvimento do sistema, contendo sua especificação e implementação.

O quinto finaliza o trabalho com as conclusões e sugestões futuras observadas no decorrer do estudo.

2 O PROBLEMA DO CORTE BIDIMENSIONAL

As indústrias de papelão reciclado, placas de aço, vidro, papel, filme, plástico, couro, tecido e madeira, fabricam seus produtos em placas de tamanho fixo, sendo que estas placas são adquiridas por outras indústrias, que as dividem em tamanhos menores definidos de acordo com suas necessidades.

Conforme Rangel (1990), o problema do corte bidimensional aparece no processo de produção dessas indústrias que usam matéria-prima fornecida em placas de tamanho fixo, onde uma placa retangular deve ser cortada em peças menores.

Para Rangel (1990), o problema do corte bidimensional (CB) consiste em determinar esquemas para cortar o menor número de placas retangulares de tamanho padrão $\boldsymbol{L}$ x $\boldsymbol{W}$, de forma a atender uma demanda b_i, de itens retangulares de dimensões $\boldsymbol{l}_i$ x $\boldsymbol{w}_i$, $i = 1...n$.

Para melhor entendimento considerar-se-á um problema onde se tem:

- tamanho padrão da placa: **L** = 85 cm (altura), e **W** = 170 cm (comprimento).
- itens retangulares menores: $\mathbf{l}_1$ x $\mathbf{w}_1$ = 50 x 20 cm, $\mathbf{l}_2$ x $\mathbf{w}_2$ = 30 x 60cm
- demanda para os itens menores: $\mathbf{b}_1$= 100, $\mathbf{b}_2$=150.

A partir destas entradas, supõe-se que os possíveis esquemas de corte das placas padrão combinando os itens menores, possam ser os mostrados nas figuras 1 e 2.

Figura 1 – Esquema de Corte 1.

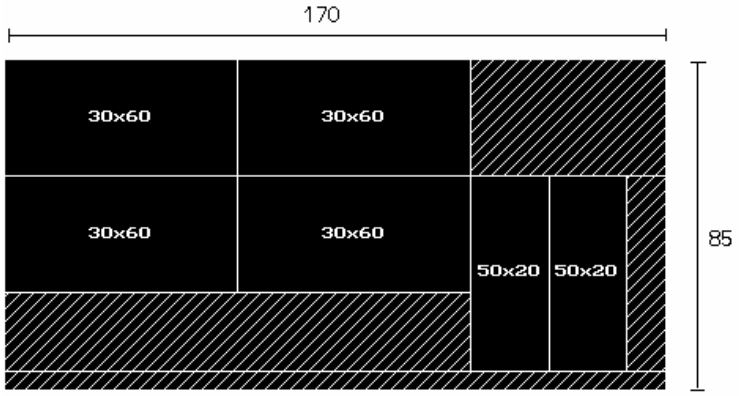
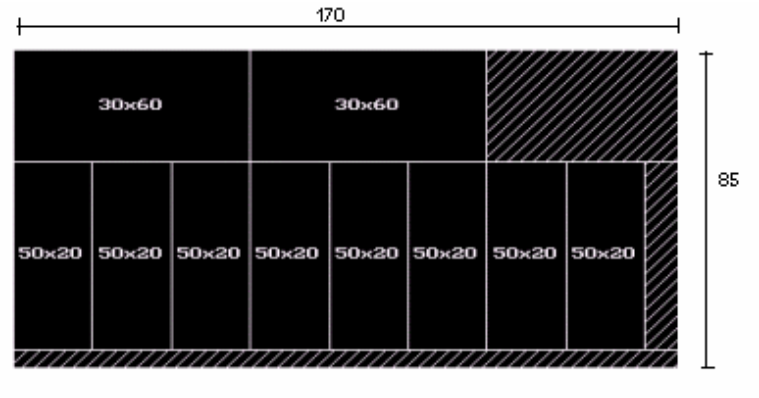


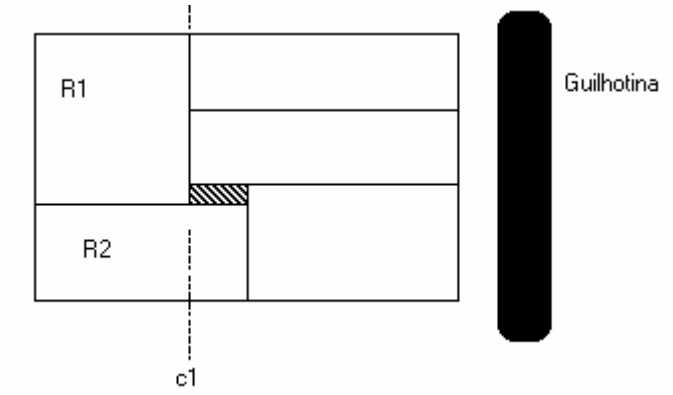
Figura 2 – Esquema de Corte 2.



Com os esquemas de corte definidos, o objetivo é minimizar o número de peças de tamanho padrão *L* x *W* a serem cortadas. Desta forma, este problema pode ser enquadrado como um problema de programação matemática, mais precisamente de programação linear inteira.

É importante salientar que este exemplo é apenas ilustrativo, não sendo utilizado ainda nenhum tipo de técnica para criação dos padrões de corte. Além disso, quando se utiliza guilhotina, o corte é feito a partir de uma aresta até a aresta oposta, eliminando-se por exemplo, o padrão de corte apresentado na fig. 3.

Figura 3 – Padrão não Permitido.



Na figura 3 observa-se que com o corte *c1*, obtém-se o item *R1* e não será possível obter-se *R2*.

2.1 COMPLEXIDADE DE IMPLEMENTAÇÃO

Conforme Goldbarg (2000), a dificuldade para se formular um problema de programação matemática é semelhante à de representar adequadamente um sistema do mundo real. Há um primeiro nível de dificuldade incidente sobre o modelador, que está associado à sua capacidade de perceber os relacionamentos entre causa e efeito e encontrar as causas fundamentais e um segundo nível que engloba o domínio das técnicas de representação do fenômeno em um contexto diferente do original, ou seja, o contexto do modelo.

Uma vez que os modelos devem ser implementáveis e na atualidade a ferramenta de consenso para a análise quantitativa é o computador, umas das representações mais utilizadas têm sido os algoritmos.

Segundo Goldbarg (2000), o objetivo maior de um algoritmo é solucionar um problema de forma rápida e econômica, não sendo portanto o algoritmo, apenas um conjunto de instruções que soluciona certo problema decidível. O termo rápido está associado à realidade do fim prático do algoritmo e o termo econômico está ligado à realidade da limitação dos recursos humanos ou computacionais. Uma análise da eficiência de um algoritmo significa predizer os recursos que irá requerer quando em passo de execução.

Segundo Salvetti (1998), somente com um estudo da complexidade de algoritmos é possível a comparação de dois algoritmos equivalentes, ou seja, desenvolvidos para resolver o mesmo problema. Há então, uma notação denominada *O,* utilizada para se expressar comparativamente o crescimento assintótico de duas funções. O termo assintótico está relacionado à velocidade com que uma função tende ao infinito. No estudo da complexidade de algoritmos é mais interessante saber como se comporta essa função à medida que é aumentado o valor das entradas representadas por *m*, do que conhecer valores da função correspondente a particulares valores de *m*. Como um exemplo, pode se citar que é mais importante saber que o número de operações se dobra ao duplicar-se o valor de *m*, do que saber que para *m* igual a 100 são executadas 300 operações.

Além disso, Goldbarg (2000) divide os problemas quanto à grandeza de solução que os algoritmos conhecidos apresentam, definindo então uma classe de problemas P como sendo a que compreende os problemas que admitem um algoritmo polinomial de solução.

Szwarcfiter (1998) define uma classe de problemas NP como sendo aquela que compreende problemas de decisão, tais que existe uma justificativa à resposta SIM para o problema, e que para o reconhecimento desta justificativa pode ou não existir uma solução, sendo que se existir, a mesma deve ser polinomial para a entrada fornecida. Já um problema NP-completo é um problema que faz parte de uma classe de problemas NP, de tal forma que os problemas da classe NP são redutíveis a ele em tempo polinomial. Por exemplo, se um problema de P2 é um problema redutível a um problema P1, então atribui-se que o problema de P1 é um problema NP-completo, ou seja, tem-se então problemas NP de complexidade equivalentes.

Assim entende-se que o problema de cortes bidimensionais tratado neste trabalho tem notação NP-completo, pois utiliza ramificação de árvores, sendo que os mesmos já possuem algoritmos que apresentam solução polinomial. Como exemplos de problemas de similar complexidade, pode-se citar:

a) problema de corte unidimensional: onde deseja-se otimizar as diversas seqüências de cortes de rolos em comprimentos *S1*,...,*Sn.* É aplicado ao corte de bobinas;
b) problema de empacotamento bidimensional: tenta-se obter um algoritmo aproximado que otimiza as possíveis formas de alocação de caixas em uma compartimento de cargas de um caminhão. Ocorre na alocação de *pallets* em caminhões de transporte de cargas;
c) problema clássico do caixeiro viajante: pretende-se achar um percurso de menor distância para um vendedor que deve visitar cada uma de *n* cidades, começando e terminando sua viagem na cidade 1. Aplicado a determinação de rotas.

Dentre uma das soluções encontradas na forma de algoritmos computacionais para o problema dos cortes bidimensionais sem o uso de PO, pode-se citar Nascimento (1999), que apresenta um método heurístico baseado em três regras principais enumeradas:

a) cortar sempre a maior peça que caiba na menor área de chapa disponível;
b) o corte deve ser sempre guilhotinado, retirando-se imediatamente a peça do canto inferior esquerdo da chapa;
c) escolhe-se o tipo de corte (vertical ou horizontal e com ou sem rotação) com base em algumas estratégias simples como, por exemplo, maximizar a área do menor ou do maior pedaço de chapa restante.

3 PESQUISA OPERACIONAL

Segundo Bronson (1985), a Pesquisa Operacional (PO), que diz respeito à alocação eficiente de recursos escassos, é tanto uma arte como uma ciência. A arte reside na habilidade de exprimir os conceitos de eficiência e de escassez por meio de um modelo matemático bem definido para uma determinada situação; a ciência consiste na dedução de métodos computacionais para solucionar tais modelos.

Para Goldbarg (2000), os modelos de PO são estruturados de forma lógica e amparados no ferramental matemático de representação, objetivando claramente a determinação das melhores condições de funcionamento para os sistemas representados. Os principais modelos de PO são denominados de programação matemática e constituem uma das mais importantes variedades dos modelos quantitativos.

O campo da programação matemática, conforme Goldbarg (2000), é enorme e suas técnicas consagraram-se em face à sua grande utilidade na solução de problemas de otimização. Em virtude das várias peculiaridades inerentes aos diversos contextos de programação, os métodos de solução sofreram especializações e particularizações. Porém, o processo de modelagem matemática, em si, pouco variava e as técnicas de solução acabaram agrupadas em várias sub-áreas como: programação linear, não-linear e inteira.

Sob este conceito, pode-se entender que organizações que procuram soluções para muitos de seus problemas de otimização, devem recorrer à pesquisa operacional para obter um método científico e fundamental de solução.

Dentre os exemplos mais comuns de aplicação da pesquisa operacional, pode-se enumerar as principais técnicas utilizadas citando alguns problemas (Wagner, 1985):

b) programação linear, na qual distribui-se uma quantidade fixa de recursos para uma determinada finalidade, sendo que alguma função deve ser otimizada. Exemplos:
   - problema de transportes: determinação de rotas para produtos de várias filiais de uma empresa a um número de áreas de mercado ao longo de percursos de transporte de mínimo-custo, reconhecendo fatores de custos diferenciais de produtos manufaturados, como encargos relativos de embarque e variações sazonais na demanda dos fregueses;

- problema de mistura de material: um produto é fabricado misturando-se vários ingredientes, desejando-se saber qual mistura atende às exigências a um custo mínimo.

c) programação não-linear: neste caso, trata-se de funções não lineares. Exemplos:
   - mistura de combustível: modelos para misturar gasolina dos estoques crus da refinaria comumente contém restrições não-lineares relativas à taxa de octanos de cada mistura, uma vez que esta característica varia não-linearmente com a quantidade de tetraetil-chumbo acrescentado à mistura;
   - controle de processos: num modelo para usina de processamento de aço, uma variável representando a temperatura de uma fornalha pode ser descrita por uma função não-linear de variáveis que indicam a quantidade e duração de energia calorífica aplicada, sendo que cada uma destas variáveis está contida em outras restrições, bem como na função-objetivo.

d) programação linear inteira: os objetivos são os mesmos da programação linear, porém algumas ou todas as soluções apresentadas devem ser números inteiros. Exemplo:
   - problema de seqüência: quando há *n* itens a serem fabricados, e cujos itens devem ser colocados em seqüência através de *k* máquinas, supondo que um item não pode possa prosseguir à máquina *j* até que ele não termine de ser processado na máquina *j-1*. O problema envolve otimização combinatória, pois consiste em achar, entre um conjunto finito de alternativas, uma que otimize o valor de uma função objetivo.

Basicamente, um modelo de otimização é construído seguindo um processo de análise quantitativa, resumido nos seguintes passos (Goldbarg, 2000):

a) formulação do problema: envolvendo definição de variáveis controláveis e não controláveis, elaboração da função objetivo e do critério de otimização e a formalização das restrições do modelo;
b) construção do modelo: englobando a elaboração da estrutura de entrada e saída de informações, as fórmulas de inter-relação e os horizontes de tempo;
c) execução das análises: compreendendo a análise da sensibilidade da solução, o levantamento da precisão dos dados, o estudo da estabilidade computacional e o levantamento das demais especificações do modelo;

d) implementação e utilização: envolvendo um processo de feedback repassando os passos anteriores, vivenciando o uso do modelo no sistema de produção ou prestação de serviços.

Contudo, embora dispondo de uma seqüência lógica para o processo de formulação matemática dos modelos, é importante salientar que essas etapas podem envolver certo grau de complexidade até a resolução do problema.

3.1 PROGRAMAÇÃO LINEAR

Conforme Loesch (1999), durante a segunda Guerra Mundial, a *United States Air Force - USAF* organizou um grupo de pesquisadores de nome SCOOP (*Scientific Computation of Optimum Program*), sob a direção de Marshall K. Wood, para tentar solucionar o problema de alocação de recursos limitados de modo a otimizar objetivos. George B. Dantzing era um dos membros deste grupo e embora nenhum método de expressivo sucesso fosse descoberto durante a guerra, ele formulou o problema de programação linear (PL) geral e inventou o Método Simplex em 1947.

A partir daí surgiram então métodos práticos e estudos para problemas de PL, aliados à maior capacidade de processamento de dados fornecida pelo contínuo processo evolutivo dos equipamentos de computação.

Para Goldbarg (2000), os modelos de PL são um tipo especial de modelos de otimização. Para que um determinado sistema possa ser representado por meio de um modelo de PL, ele deve possuir as seguintes características:

a) proporcionalidade: a quantidade de recurso consumido por uma atividade deve ser proporcional ao nível dessa atividade na solução final do problema. Além disso, o custo de cada atividade é proporcional ao nível de operação da atividade.
b) não negatividade: deve ser sempre possível desenvolver determinada atividade em qualquer nível não negativo e qualquer proporção de um dado recurso deve sempre poder ser utilizado.
c) aditivabilidade: o custo total é a soma das parcelas associadas a cada atividade.
d) separabilidade: pode-se identificar de forma separada o custo (ou consumo de recursos) específico das operações de cada atividade.

Um modelo de PL é um modelo matemático de otimização no qual todas as funções são lineares.

Para que se possa solucionar um problema de PL, Loesch (1999) define um modelo genérico apresentado no quadro 1.

Quadro 1 – Modelo Genérico.

$$\{Max,Min\}z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

sujeito a

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \{=,<=,>=\} b_1$$

$$a_{22}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \{=,<=,>=\} b_2$$

$$a_{m2}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \{=,<=,>=\} b_m$$

$$x_1, x_2, \ldots, x_n >= 0$$

Interpreta-se $x_1, x_{2,\ldots,}x_n$ como sendo as variáveis estruturais do problema e $c_1, c_{2,\ldots,}c_n$ os coeficientes da função objetivo. As restrições expressam limites a serem respeitados e são representadas por a_{ij} e b_m.

Na PL todas as variáveis devem ser quantidades reais. No caso de um problema de planejamento da produção em uma indústria discreta, as variáveis deverão representar quantidades a serem produzidas, necessariamente quantidades inteiras. Estes problemas são caracterizados então como problemas de programação linear inteira.

3.2 PROGRAMAÇÃO LINEAR INTEIRA

Segundo Loesch (1999), os problemas de programação linear inteira, são estruturados da mesma forma que os de PL, sendo que são caracterizados pela presença de ao menos uma restrição de integridade, ou seja, os possíveis valores assumidos pelas variáveis devem ser inteiros.

Conforme Goldbarg (2000), numa visão geral, podem ser enumeradas diversas técnicas para a busca da solução inteira de problemas de programação linear, citando:

a) técnicas de enumeração: baseadas na idéia de se desenvolver uma enumeração inteligente dos pontos candidatos à solução ótima inteira de um problema. Citam-se algumas:
   - separação e avaliação progressiva, conhecida como *Branch & Bound*;
   - enumeração implícita;
b) técnicas de cortes: extensão às técnicas de enumeração que reduz o número de iterações de um problema. Há um acréscimo de restrições a cada nova etapa de um algoritmo de solução para o problema. Pode-se citar:
   - cortes inteiros (primais e duais);
   - cortes combinatórios;
   - cortes de interseção;
   - método de geração de colunas;
c) técnicas Híbridas: neste caso são unidos algoritmos exatos à heurísticas:
   - técnica Branch & Cut;

Para a resolução do problema de cortes bidimensionais, podem ser utilizadas conjuntamente duas técnicas: *Branch & Bound*, conhecida como ramificação e limite, e um método que pode ser enquadrado numa das técnicas de cortes, denominado por Silva (1999), como método de geração de colunas.

3.2.1 TÉCNICA DA RAMIFICAÇÃO E LIMITE

Segundo Rangel (1999), utilizando o método particionar e limitar (entende-se ramificação e limite), os esquemas de corte são gerados desenvolvendo um procedimento de busca em árvore onde no nó raiz, as ramificações representam todos os possíveis cortes na

placa retangular de dimensões *L* x *W* e o nó ao fim de algum ramo, representa os itens produzidos pelo respectivo corte. A cada nó subseqüente, um retângulo é selecionado para futuros cortes. Para diminuir o número de nós que o algoritmo percorre na árvore são desenvolvidos limitantes superiores para o valor da solução. Obtém-se assim, um algoritmo aproximado adaptado ao problema de cortes dos retângulos.

Silva (1999), apresenta este algoritmo, denominando-o algoritmo de Herz, que se caracteriza por ter uma solução exata quando na determinação do melhor padrão possível de corte guilhotina, em uma única placa, dado um conjunto de retângulos, sendo que não há a obrigatoriedade de consideração de todos os retângulos. A seguir, são enumerados os passos a serem seguidos para a determinação do padrão de corte. Considera-se peso como sendo a área dos retângulos:

a) determinação do corte homogêneo dada uma certa região da placa (inicialmente a própria placa), ou seja, da inclusão do maior número de apenas um tipo de retângulo com maior peso. Peso total = A;
b) realização de um corte vertical, divisão da placa original em duas, dado um certo critério da posição exata de corte (posições do vetor P): verifica-se o lado esquerdo (peso total = X_e) e o direito (peso total = X_d), encontrando-se o peso total de preenchimento (peso total = $X = X_e+X_d$). Se $X > A$, então $A = X$;
c) faz-se o mesmo, realizando-se agora um corte horizontal (posições do vetor Q), dividindo o problema em duas partes: superior (peso total = Y_s) e inferior (peso total = Y_i), resultando em um outro peso total (peso total = $Y = Y_s+Y_i$). Se $Y > A$ então $A = Y$. A, agora, conterá o maior peso total considerado;
d) testa-se uma nova posição de corte na tentativa de encontrar um peso total maior de preenchimento, repetindo-se o processo para todas as possíveis posições oferecidas pelo vetor P (na vertical) e Q (na horizontal).

Para o funcionamento deste algoritmo deve-se inicialmente ter os seguintes dados: dimensões da placa, dimensões e quantidades de retângulos a serem cortados nesta placa, índice mínimo de preenchimento (atribui-se 100% para um melhor aproveitamento), além do peso (área) de cada um dos retângulos considerados. É importante ressaltar que as dimensões das peças não podem ser invertidas. Por exemplo, uma peça com dimensões 30x60 cm não é considerada equivalente a uma peça com dimensões 60x30 cm.

Basicamente, para que seja possível a utilização do algoritmo de Herz, é necessário seguir os passos abaixo:

a) entrada dos dados – placas e retângulos a serem cortados;

b) construção dos vetores P e Q;

c) aplicação de uma rotina recursiva para corte das placas;

d) traçado das linhas verticais e horizontais nas placas.

Silva (1999) apresenta então uma rotina (quadro 2), que em 2 chamadas gera os vetores P e Q respectivamente.

Quadro 2 – Algoritmo Gerador dos Vetores P e Q.

```
gera_PQ(pos, esq, dir)
início
      se (esq + S[pos] <= dir) e (pos < n)
      q = maior valor inteiro menor de ((dir – esq) / S[pos])
      for (i = 1 até q)
            valor = esq + i*S[pos]
            se (valor <= dir)
                  inclui_hash(valor)
            decrementa(i)
            for (j = 0 até i)
            gera_PQ(pos+1, esq+(i-j)*S[pos], dir)
fim
```

Fonte: Silva (1999).

Segundo Silva (1999), após uma primeira ordenação das dimensões *x* para P, assim como das *y* para *Q*, dos *n* retângulos utilizando-se de um vetor auxiliar temporário *(S)*, obtém-se como resultado do algoritmo 2 vetores, P e Q, que contém as posições de corte na placa retangular inicial. O vetor P representa as possíveis combinações lineares das dimensões horizontais dos retângulos e Q as verticais.

Tomando-se como exemplo uma placa de dimensões 10x10 cm e 2 retângulos que deverão ser originados dos cortes, cujas dimensões sejam 2x5 cm e 6x3 cm respectivamente, o resultado do algoritmo será a geração dos vetores P = [2 , 4] e Q = [3 , 5].

A seguir, no quadro 3 é apresentada a rotina *cortes* proposta por Silva (1999) para a resolução do problema, a qual determinará a disposição dos retângulos na placa. Esta deve ser chamada passando-se os parâmetros *cortes(0,0,A,B,0,0)*, onde *A* é o comprimento, e *B* a altura da placa.

Quadro 3 – Rotina Principal de Cortes.

```
cortes(x, y, A, B, i, j)
início
   peso = 0.0
   for (k = 0 até n)
      peso_aux = gera_homog(A, B, S[k])
      pega maior área homogênea (peso_aux) e o k respectivo (k_max)
   se (peso_aux > peso)
      peso  = peso_aux
   padrao = gera_padrao(x, y, A, B, S[k_max])
   se (peso = 0) ou (peso = A*B)
      retorna(padrao)
   for (laco1 = 0 até nP)           // nP é o núm. de pos. do vetor P
      se (i < nP) e (P[i] < A)
         padrao1 = cortes(x, y, P[i], B, i, 0)
         padrao2 = cortes(x+P[i], y, A-P[i], B, i, 0)
         peso_aux = padrao1.peso + padrao2.peso
         se (peso_aux > peso)
            peso = peso_aux
            padrao = concatena(padrao1, padrao2)
      incrementa(i)
   for (laco2 = 0 até nQ)           // nQ é o núm. de pos. do vetor Q
      se (j < nQ) e (Q[j] < A)
         padrao1 = cortes(x, y, A, Q[j], 0, j)
         padrao2 = cortes(x, y+Q[j], A, B-Q[j], 0, j)
         peso_aux = padrao1.peso + padrao2.peso
         se (peso_aux > peso)
            peso = peso_aux
            padrao = concatena(padrao1, padrao2)
      incrementa(j)
   retorna(padrao)
fim
```

Fonte: Silva (1999).

Conforme Silva (1999), na implementação do algoritmo do quadro 3, considera-se *padrao* como uma lista encadeada, onde cada nó armazena:

a) *x*,*y*: posição relativa de um conjunto homogêneo de determinado retângulo (pode ser apenas um) dentro da placa;

b) *k*: número de identificação do retângulo (um número de 0 a n-1);

c) *n_h*, *n_v*: quantidade deste retângulo, dentro do conjunto homogêneo, na horizontal (relativo ao comprimento) e vertical (relativo à altura);

d) *peso*: valor da área do retângulo.

e) *i e j*: posições nos vetores P e Q, respectivamente.

Assim, no algoritmo tem-se *k* que representa o k-ésimo retângulo fornecido, as variáveis *x* e *y* representam a posição relativa (como se fossem coordenadas cartesianas com a

origem no canto inferior esquerdo da placa) de um bloco de retângulos do mesmo tipo na placa, sendo *nh* retângulos iguais a este na horizontal e *nv*, na vertical (bloco homogêneo).

A função *gera_homog* do algoritmo apresentado no quadro 3, determina o peso resultante da inclusão do máximo de retângulos iguais na região em questão. A função *gera_padrao* cria um nó com o padrão homogêneo que melhor preencheu a mesma região. A função *concatena* acrescenta, ao final da lista do *padrao1*, a lista do *padrao2*, criando uma só lista, quando a realização de um corte guilhotina vertical ou horizontal determina duas metades cuja soma dos pesos é maior que o melhor peso obtido até então. Cada chamada recursiva de *cortes* determina uma região da placa, sendo que as duas primeiras dividem a placa em lado esquerdo e direito, e as outras duas, em lado superior e inferior. Silva (1999), utiliza a estrutura apresentada no quadro 4 para armazenamento da lista encadeada.

Quadro 4 – Estrutura da Lista Encadeada de Padrões.

```
typedef struct elemento_padrao {
   struct elemento_padrao *prox;
   float  x;
   float  y;
   int    k;
   int    n_h;
   int    n_v;
   float  peso;
} tipo_padrao;
```

Fonte: Silva (1999).

A partir deste ponto, já se tem definido um algoritmo que forneça padrões de corte distintos das placas, podendo-se então, pensar em minimização do total de placas de matéria-prima a serem consumidas, para atender às demandas de cada peça. A partir daí, utiliza-se o método de geração de colunas.

3.2.2 MÉTODO DE GERAÇÃO DE COLUNAS

Conforme Silva (1999), deve ser levado em consideração em determinadas situações de corte de placas padrão, a demanda dos retângulos a serem cortados, e portanto, a possibilidade de utilização de mais de uma placa. Como o objetivo é minimização do uso de placas retangulares, é utilizado o método de geração de colunas, estabelecendo-se o seguinte para o problema do corte bidimensional: dada uma lista L de retângulos a serem cortados em placas R e seja P_1, P_2, …, P_m os possíveis padrões de corte de retângulos L em uma placa R.

Um padrão de corte pode conter vários tipos de retângulos e ser representado como um vetor S*i*. Cada elemento deste vetor indica quantos retângulos de um determinado tipo estão cortados naquele padrão P_i de forma que, como agora o objetivo é minimizar a quantidade de placas usadas, pode-se formular o seguinte problema de programação inteira apresentado no quadro 5.

Quadro 5 - Modelo Formulado.

$$\{min\} z = x_1 + x_2 + x_3 + \cdots + x_n$$
$$\text{sujeito a} \quad P_1x_1 + P_2x_2 + \ldots + P_nx_n \geq b_k$$
$$x_i \in Z^+$$

Fonte: Adaptado de Silva (1999).

O vetor ***b*** representa a demanda de um retângulo, ou seja, um determinado retângulo deve ser cortado pelo menos ***bk*** vezes. Esta formulação apresenta duas restrições a serem destacadas: a enormidade de padrões possíveis que, na maioria dos casos, inviabiliza a resolução e a dificuldade em se resolver um problema de programação matemática procurando soluções com números inteiros.

Silva (1999), apresenta uma nova série de passos determinados a partir do método de geração de colunas e que devem ser seguidos, demonstrando a solução para o problema do corte bidimensional:

a) é escolhida a placa na qual serão cortados os retângulos. Todas as placas necessárias para cobrir a demanda *b* devem ser iguais a que foi escolhida.
b) define-se uma base, podendo ser uma matriz diagonal, quadrada, com ordem igual a *n* (n é o número total de retângulos considerados). Esta matriz é chamada de *B*. Cada elemento da diagonal corresponde a quantidade máxima de cada retângulo em uma placa. A demanda de cada retângulo dividida pela sua quantidade no padrão homogêneo determina, ainda de forma ineficiente, a quantidade necessária de placas para suprir a demanda de cada retângulo. Forma-se a solução inicial num vetor de *n* posições. Este vetor é chamado de *x*.
c) resolve-se o sistema de equações $yB = [1,1,...,1]$ (com n 1´s), obtendo os valores do vetor *y*.
d) acha-se um padrão de corte na placa (que pode ser agora heterogêneo). Ter-se-á

então um vetor temporário *Si* com a_1 retângulos do tipo 1, a_2 retângulo do tipo 2, ..., a_n retângulos do tipo *n*. A somatória dos valores dos pesos dos retângulos vezes seus respectivos valores de *a* deve ser menor ou igual ao peso da placa. A somatória dos valores encontrados em *y* vezes os respectivos valores de a deve ser maior que 1. Satisfeitas estas condições o padrão pode ser aceito, e *a1, a2, ..., an* formam um vetor que será chamado de *a*. O vetor transposto de *a* será a nova coluna a ser incluída em *B*. Se não foi possível determinar um padrão novo então não existe solução ou a solução ótima foi encontrada.

e) resolve-se o sistema $Bd = a$, encontrando-se *d*.
f) compara-se a divisão dos valores de *x* pelos valores de *d*. O menor valor encontrado correspondente é associado a uma variável *t*, e a coluna que gerou o valor de t é a coluna que será substituída pelo valores do vetor *a*.
g) a solução *x* é atualizada pegando-se o valor de *x* anterior menos os valores de t vezes *d*. Retorna-se ao passo c.

Segundo Silva (1999), o algoritmo de Herz é utilizado no passo *d* do método de geração de colunas. Basicamente o método de geração de colunas é utilizado devido à enormidade de padrões de corte que possam ser formados no vetor *S (P e Q)*. São primeiramente gerados os padrões homogêneos, e a cada iteração, é testado se um novo padrão de corte pode influenciar na otimização. Para que se tenha, a cada iteração, um padrão distinto cria-se então um novo vetor, onde cada posição apresenta uma lista de padrões já gerados, na qual cada nó apresenta somente a identificação do retângulo (um número de 0 à n) e sua quantidade. Ao final, ter-se-á *n* padrões de corte distintos com o padrão *i* tendo *xi* placas (*xi* é o *i*-ésimo elemento do vetor *x*).

Assim, verifica-se a necessidade de que seja utilizado em um primeiro momento, um método aproximado para se gerar os padrões de corte de forma não exaustiva, e somente num segundo momento é que se busca a otimização das quantidades de placas a serem consumidas.

Observa-se que novas iterações só se prosseguem, caso cada novo padrão de cortes satisfaça as condições impostas no passo *d* do método de geração de colunas. Caso contrário encontrou-se então uma solução adequada.

4 DESENVOLVIMENTO DO SISTEMA

Neste capítulo são apresentados os passos para o desenvolvimento do sistema, os quais seguem basicamente a ordem: especificação e implementação.

4.1 TRABALHOS CORRELATOS

Alguns trabalhos correlatos encontrados foram:

a) uma tese sobre o problema do corte bidimensional, apresentando uma visão geral de métodos exatos e heurísticos para o problema de corte de peças retangulares usando-se guilhotina (Rangel, 1990).
b) um trabalho de conclusão de curso sobre um sistema de controle de reservas de sala de aula da Universidade Regional de Blumenau, o qual é baseado principalmente em técnicas de alocação para a criação da grade de horários (Schoeffel, 2001).
c) um relatório do projeto de iniciação científica onde o autor relaciona de uma melhor forma o problema do corte bidimensional com técnicas da PO, apresentando algoritmos para a solução do problema (Silva, 1999).

4.2 REQUISITOS PRINCIPAIS

Este trabalho envolve basicamente a pré-seleção de retângulos de acordo com pedidos de peças e a geração dos padrões de corte em placas de papelão mantidas em estoque. As principais restrições estão ligadas:

a) ao número significativo de tamanhos diferentes dos retângulos existentes na linha de produção;
b) a quantidade significativa de padrões de corte que podem ser gerados em tempo hábil com estes diferentes retângulos;
c) à dificuldade de resolução de problemas de programação linear objetivando soluções inteiras.

4.3 ESPECIFICAÇÃO

A especificação é uma fase importante para o desenvolvimento do sistema. O principal objetivo desta fase é fornecer um documento onde são representados os modelos levantados pelo desenvolvedor do sistema.

Para este trabalho definiu-se que a especificação seguirá os passos da análise essencial, seguindo os passos propostos nas referências encontradas, em Pompilho (1994). Foi utilizada a ferramenta CASE *Power Designer* para a especificação neste trabalho.

4.3.1 MODELO AMBIENTAL

Conforme Pompilho (1994), no modelo ambiental são descritos os objetivos do sistema, bem como quais os estímulos que o sistema recebe do meio ambiente, que eventos ele aciona e qual resposta o sistema devolve ao meio.

Uma boa seqüência costuma ser seguida construindo-se a lista de eventos, desenhando-se o diagrama de contexto e finalmente declarando-se os objetivos do sistema (Pompilho, 1994). Assim, esta seqüência é apresentada a seguir.

4.3.1.1 LISTA DE EVENTOS

Segundo Pompilho (1994), a análise essencial propõe o particionamento do sistema por eventos, definidos informalmente como acontecimentos do mundo exterior que requerem do sistema uma resposta. Assim, o sistema é construído para responder a estímulos.

Desta forma, define-se que eventos serão abrangidos pelo sistema, sendo estes, apresentados no quadro 6.

Quadro 6 – Lista de Eventos.

Número	Nome	Tipo
1	Usuário cadastra placas padrão (*L x W*)	Fluxo
2	Usuário cadastra peças (itens *li x wi*)	Fluxo
3	Usuário cadastra pedidos de peças	Fluxo
4	Usuário solicita cortes de placas padrão	Fluxo
5	Usuário efetua entrada de placas padrão	Fluxo
6	Usuário efetua entrada de peças	Fluxo
7	Usuário solicita relatório de movimentos	Fluxo

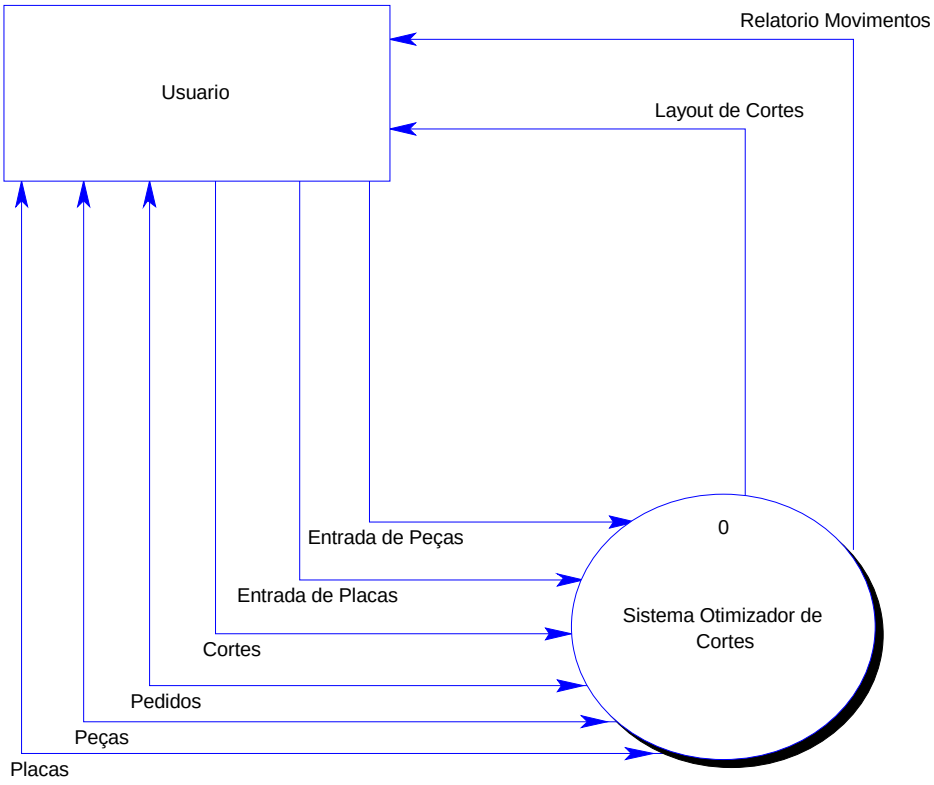
Apresenta-se uma descrição detalhada da lista de eventos:

1. Usuário cadastra placas padrão: são cadastrados, consultados, alterados e excluídos os dados das placas retangulares “padrão” utilizadas como matéria-prima;
2. Usuário cadastra peças: são cadastrados, consultados, alterados e excluídos os dados das dos diversos tipos de peças, que são retângulos originados do corte de placas padrão;
3. Usuário cadastra pedidos de peças: são cadastrados consultados, alterados e excluídos os dados básicos dos pedidos de peças;
4. Usuário solicita cortes de placas padrão: usuário seleciona o tipo de placa a ser usada e solicita o corte. Após subtrair da demanda dos pedidos, peças disponíveis em estoque é gerado um *layout* otimizado mostrando padrões de corte e quantidades necessárias de cada padrão;
5. Usuário efetua entrada de placas: usuário registra entrada de placas padrão (matéria-prima) no estoque;
6. Usuário efetua entrada de peças: usuário registra entrada de peças que sobraram do corte de placas padrão;
7. Usuário solicita relatório de movimentos: usuário solicita relatório de movimentos (entradas/saídas) das placas padrão e peças disponíveis em estoque.

4.3.1.2 DIAGRAMA DE CONTEXTO

No diagrama de contexto o sistema é apresentado como uma única função, sendo relacionadas as entidades e também os fluxos de dados. O diagrama de contexto está apresentado na fig. 4.

Figura 4 – Diagrama de Contexto.



4.3.1.3 DECLARAÇÃO DOS OBJETIVOS DO SISTEMA

O sistema proposto tem como objetivo principal a otimização do *layout* de corte de placas de papelão de tamanho padrão, em peças retangulares menores segundo uma demanda determinada pelo usuário. Assim, o sistema visa apresentar uma solução para o problema de cortes das placas fazendo com que sejam sugeridos ao usuário uma forma ótima, a qual talvez não seja a melhor possível (seria então uma abordagem exata), mas de tal modo que reduza a quantidade de placas a ser utilizada numa programação de cortes.

Para que o usuário faça os cortes das peças na guilhotina, o mesmo deve efetuar uma solicitação de cortes de itens, e assim será gerado um *layout* em tamanho reduzido.

Portanto, o sistema permite ao usuário manter os dados das placas e retângulos armazenados em uma base dados, que são selecionados e incluídos em pedidos com as respectivas demandas de peças, e posteriormente agrupados em programações de corte.

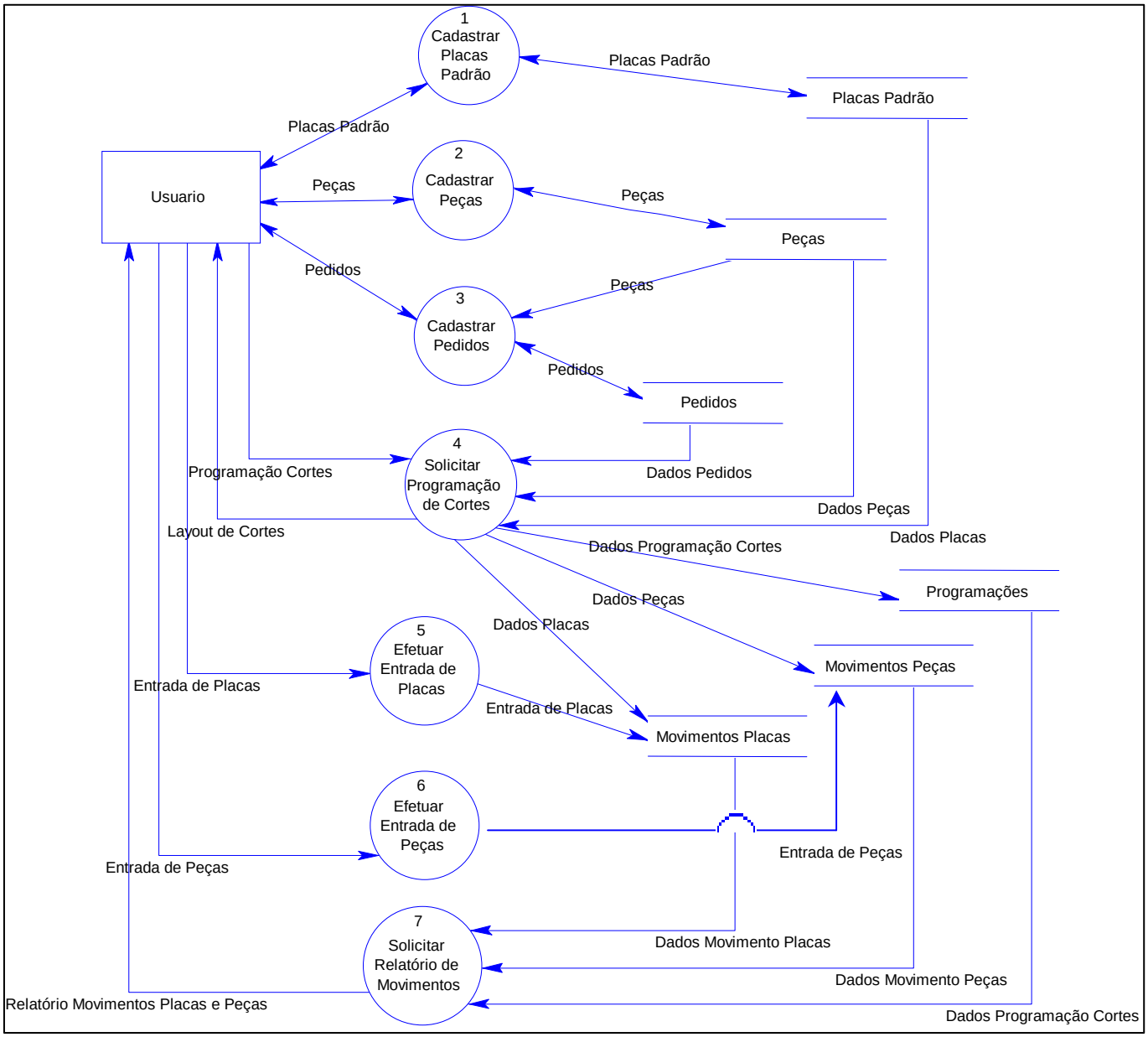
4.3.2 MODELO COMPORTAMENTAL

O modelo comportamental indica o que o sistema deve fazer para interagir satisfatoriamente com o ambiente em que o sistema se encontra. Para tanto, é apresentado então o diagrama de fluxo de dados (DFD).

4.3.2.1 DIAGRAMA DE FLUXO DE DADOS

O diagrama de fluxo de dados (DFD) apresenta de forma menos genérica os eventos e sua relação com as entidades e depósitos de dados. A fig. 5 ilustra o DFD para o sistema, onde pode-se verificar que para efetuar um corte estão envolvidas consultas aos depósitos de placas padrão e de peças.

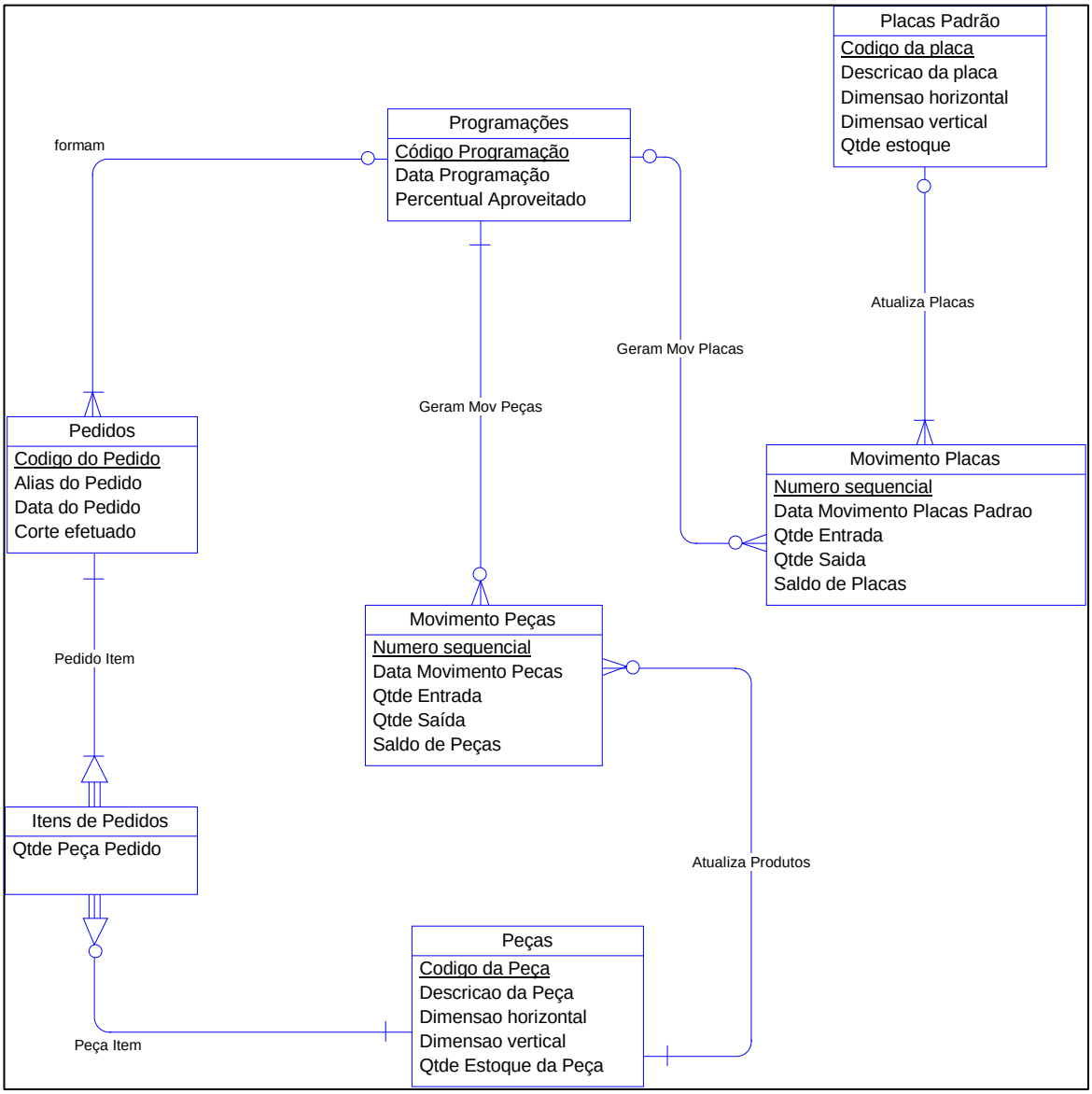
Figura 5 – Diagrama de Fluxo de Dados.



4.3.2.2 DIAGRAMA ENTIDADE-RELACIONAMENTO

Segundo Pompilho (1994), para se representar classes de entidades, relacionamentos e atributos, é usado o diagrama entidade–relacionamento (DER). Assim, através do DER é possível observar os ciclos que se referem aos relacionamentos entre as entidades, assim como os atributos das mesmas. É apresentado na fig. 6 o DER para o sistema.

Figura 6 – Diagrama Entidade-Relacionamento.



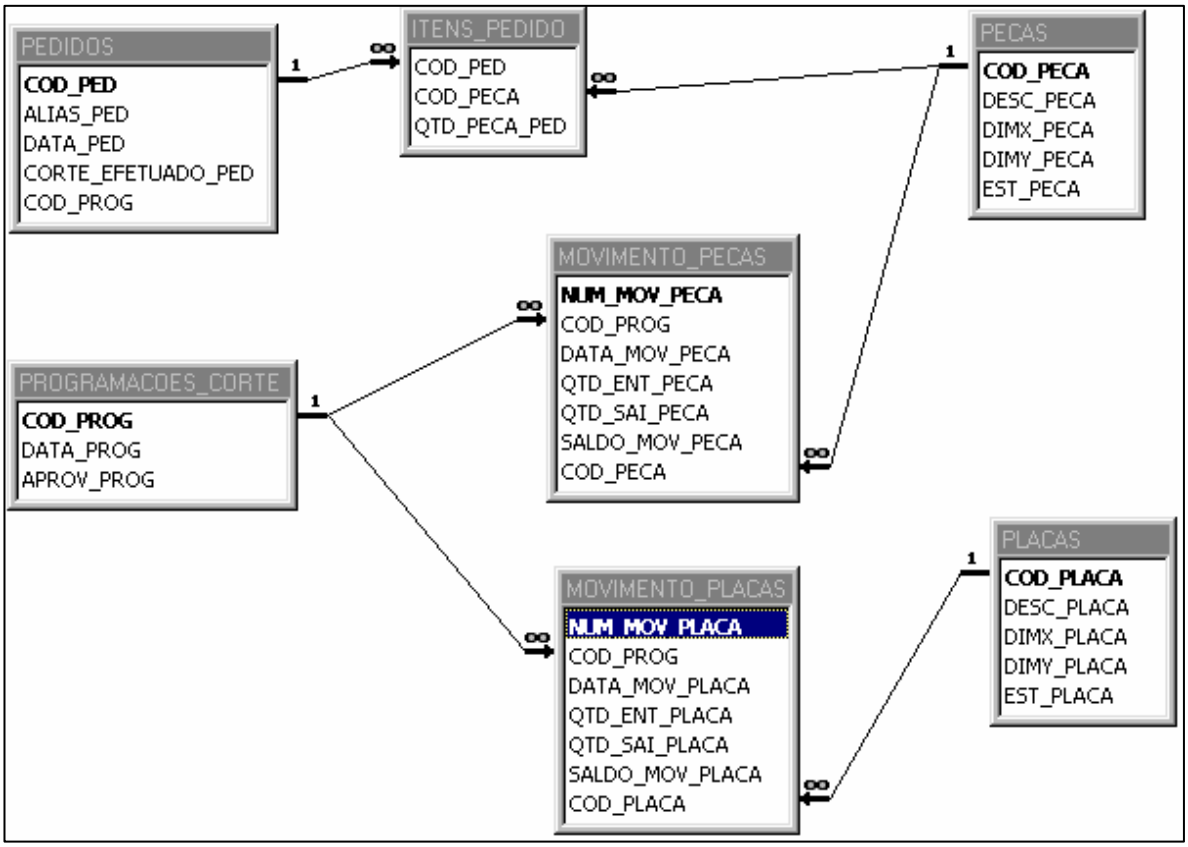
4.3.2.3 DICIONÁRIO DE DADOS

Conforme Pompilho (1994), um dicionário de dados é um repositório de informações sobre os componentes dos sistemas. Ele é o local estruturado onde são mantidos os conteúdos dos fluxos de dados, dos depósitos e dos processos. O dicionário de dados do sistema é apresentado no Anexo I.

4.3.3 MODELO DE IMPLEMENTAÇÃO

Conforme Pompilho (1994), o modelo de implementação corresponde ao modelo físico que aparecerá ao usuário. Para tanto, deve ser construído um modelo físico de dados, o qual varia conforme o Sistema Gerenciador de Banco de Dados a ser utilizado para a implementação do sistema. O SGBD utilizado neste trabalho foi o *Microsof Access*. É apresentado então o modelo físico de dados na fig. 7, extraído do SGBD.

Figura 7 – Modelo Físico de Dados



4.4 IMPLEMENTAÇÃO

A seguir são apresentados os passos do desenvolvimento do sistema, assim como algumas das telas demonstrando a operacionalidade do mesmo.

4.4.1 FERRAMENTAS E TÉCNICAS UTILIZADAS

O Sistema foi implementado no ambiente de programação visual Borland C++ Builder, utilizando o banco de dados Microsoft Access.

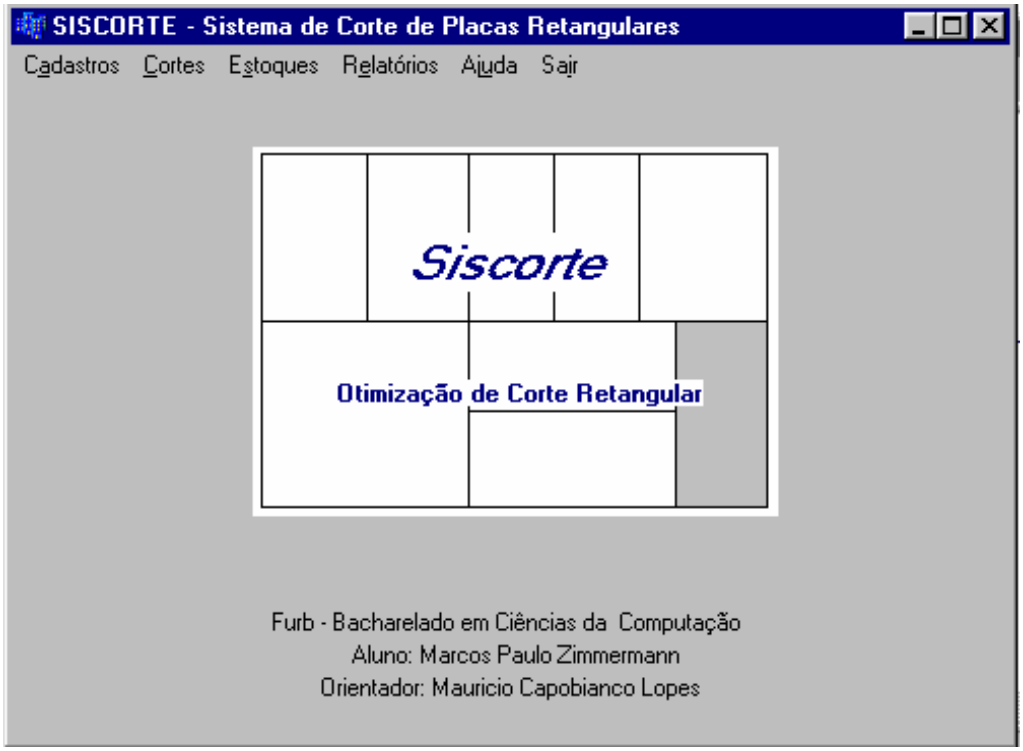
Os algoritmos implementados foram baseados no trabalho de Silva (1999), os quais apresentam adequada solução ao problema de cortes bidimensionais. Assim, no Anexo II são apresentadas as principais rotinas implementadas para o problema em questão.

4.4.2 INTERFACE COM O USUÁRIO

A interface com usuário é apresentada de forma simples, semelhante à alguns aplicativos desenvolvidos para plataformas Windows/32 bits.

A fig. 8 apresenta a tela principal inicial do sistema, onde o usuário pode facilmente escolher uma das funções que deseja efetuar.

Figura 8 – Tela Principal



As funções disponíveis são:

a) cadastros:
   - placas: nesta opção o usuário pode cadastrar, consultar, alterar e excluir as placas a serem utilizadas nos cortes;
   - peças: nesta opção o usuário pode cadastrar, consultar, alterar e excluir os diferentes tipos peças que necessita cortar;
   - pedidos: nesta opção o usuário registra pedidos de peças selecionadas;

a) cortes:
   - cortes de pedidos pendentes: nesta opção é efetuado o corte dos pedidos que estão pendentes para corte;

b) estoques:
   - entrada de placas em estoque: nesta opção é são registradas a entrada de placas no estoque;
   - entrada de peças em estoque: nesta opção são registradas as entradas de peças em estoque, provenientes das sobras de cortes efetuados;

c) relatórios:

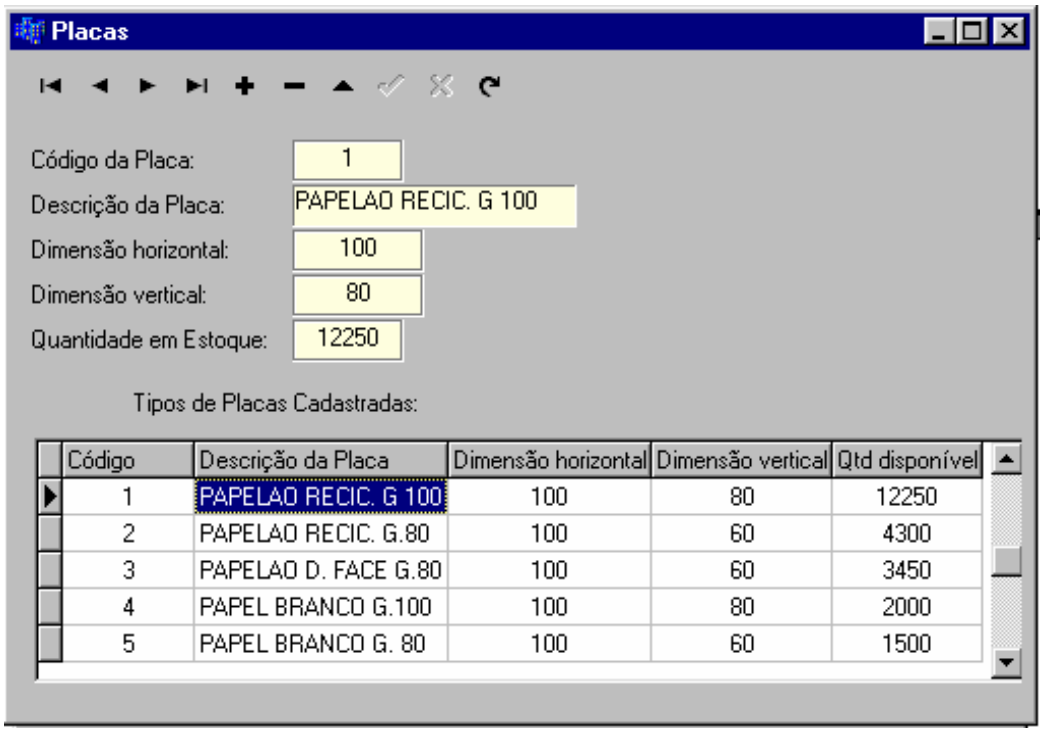
- relatório de movimentações: nesta opção o usuário solicita um relatório de movimentações das placas e peças (entradas/saídas) em estoque.

d) ajuda:

- passos do sistema: nesta opção o usuário consulta os passos necessários para a correta utilização do sistema.

Inicialmente o usuário deve cadastrar as placas padrão, utilizando a opção cadastros – placas. A fig. 9 apresenta a tela de cadastros das placas a serem utilizadas no corte.

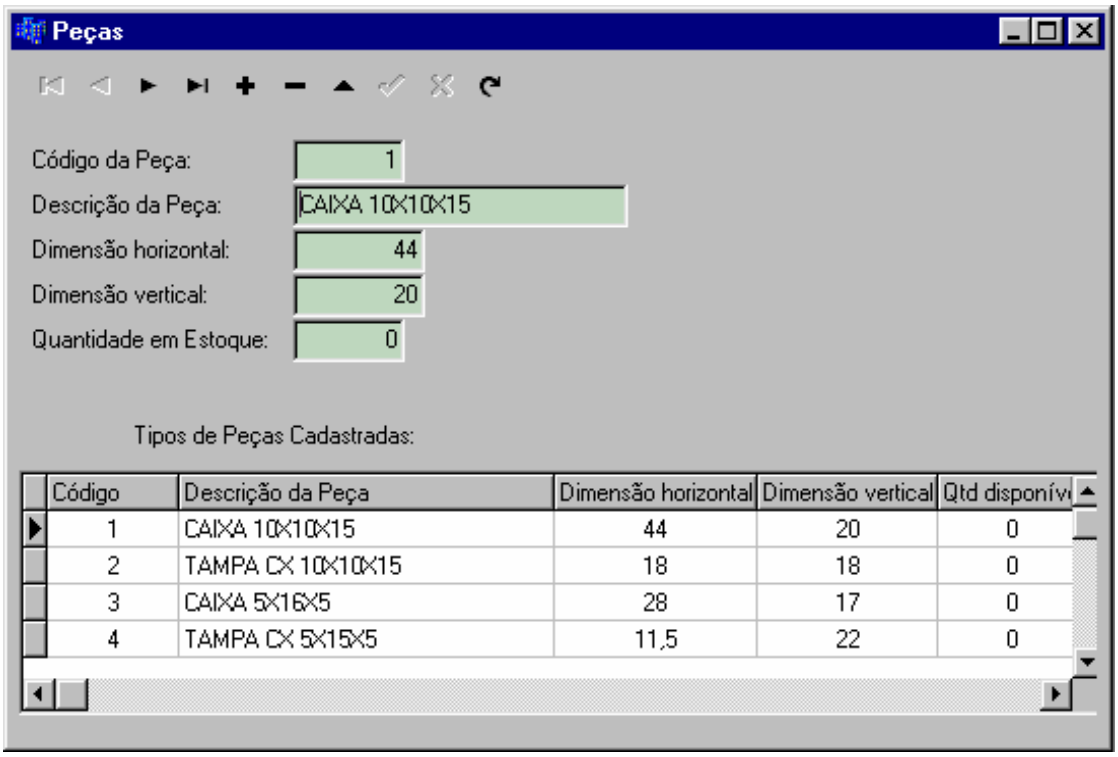
Figura 9 – Tela de Cadastro de Placas



Nesta tela é possível que o usuário observe os tipos de placas padrão já armazenadas no sistema. Inicialmente, as quantidades de placas são nulas e somente são atualizadas após o usuário registrar uma entrada no estoque de cada placa.

O próximo passo é inserir no sistema os dados dos tipos de peças que o usuário deseja obter no corte das placas. Deve-se então, utilizar a opção cadastros – peças. Apresenta-se a tela de cadastros de peças na fig. 10.

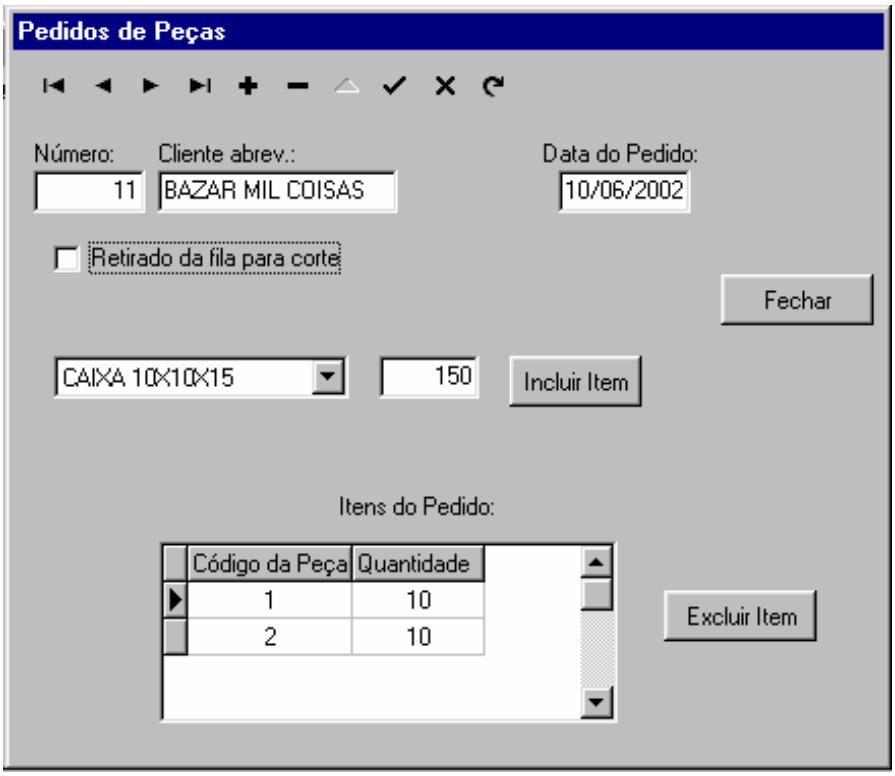
Figura 10 – Tela de Cadastro de Peças



Cadastradas as peças, o usuário já pode preencher um pedido das mesmas, que são baseados no exemplo em pedidos de produto final.

A seguir é apresentado a tela de cadastro de pedidos de peças na fig. 11.

Figura 11 – Tela de Cadastro de Pedidos

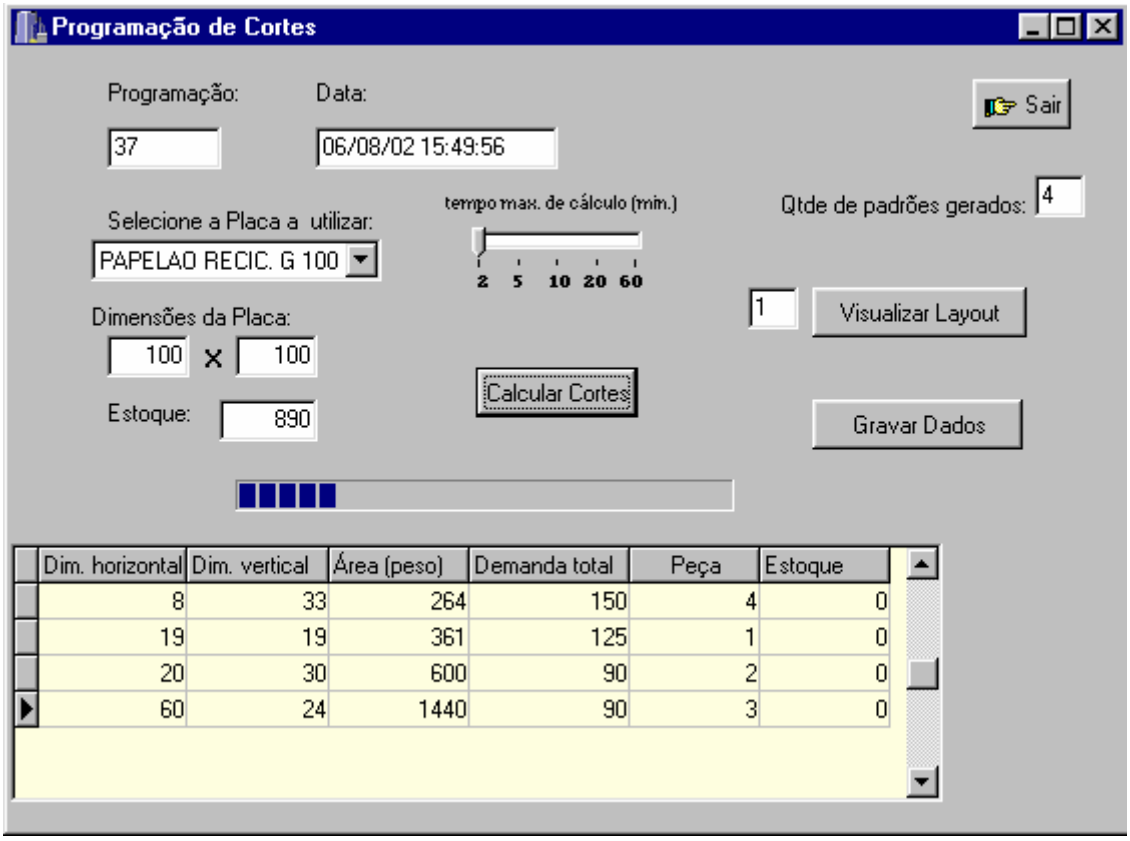


Uma vez incluídos os pedidos de peças, o usuário pode solicitar o corte das placas, utilizando a opção corte (corte de pedidos pendentes no menu principal).

A tela onde é registrada a programação de cortes é apresentada na fig. 12. Nesta tela o usuário seleciona a placa “padrão” a ser utilizada no corte, e solicita ao sistema que seja calculado o *layout* com as peças que se encontram pendentes para corte utilizando a opção “calcular cortes”. O sistema utiliza para o corte, uma consulta de todos os pedidos que estejam ”pendentes”, ou seja, que no componente *dbchecklist* esteja registrado o valor *false* (campo “Retirar da fila para corte” da tela cadastro de pedidos, fig. 11).

Portanto o usuário pode registrar o pedido, porém deixá-lo fora de uma programação de cortes. Para isso basta ativar o campo “Retirar da fila para corte” da tela cadastro de pedidos.

Figura 12 – Tela de Programação de Cortes



Ao utilizar o botão “calcular cortes” da tela de programação de cortes, o sistema verifica primeiramente o estoque de peças, e caso existam peças disponíveis, subtrai da demanda pendente as peças disponíveis. Então, o sistema calcula a quantidade de placas “padrão” necessárias e verifica se há disponibilidade das mesmas em estoque. Caso existam placas disponíveis, o processo é concretizado, e o usuário poderá visualizar em seguida um dos N *layout’s* (padrões de corte) formados. Caso não existam placas suficientes para o corte é exibida uma mensagem ao usuário e a programação de cortes é cancelada voltando-se ao menu principal.

Após a visualização/impressão dos mesmos, o usuário deve acionar o botão “gravar dados”, para que sejam atualizados os estoques de peças e placas, assim como o registro da programação de cortes efetuada, e dos movimentos de peças e placas realizados.

Na formação do *layout*, foi estabelecido que a quantidade de vezes que o padrão de corte deve ser utilizado, é arredondada para cima para obter-se soluções inteiras. Quando há sobra de peças, o usuário deve registrar entrada das mesmas no estoque utilizando a opção estoques – entrada de peças em estoque, para que sejam aproveitadas numa próxima programação de cortes.

4.4.3 ESTUDO DE CASO

A seguir é demonstrado num exemplo mais prático, o resultado do seguinte problema de cortes:

Utilizando-se uma placa 100x100 (desconsidera-se a unidade de medida, porém subentende-se que o usuário utilize a mesma unidade para as placas e para as peças desejadas), necessita-se das peças segundo o quadro 7:

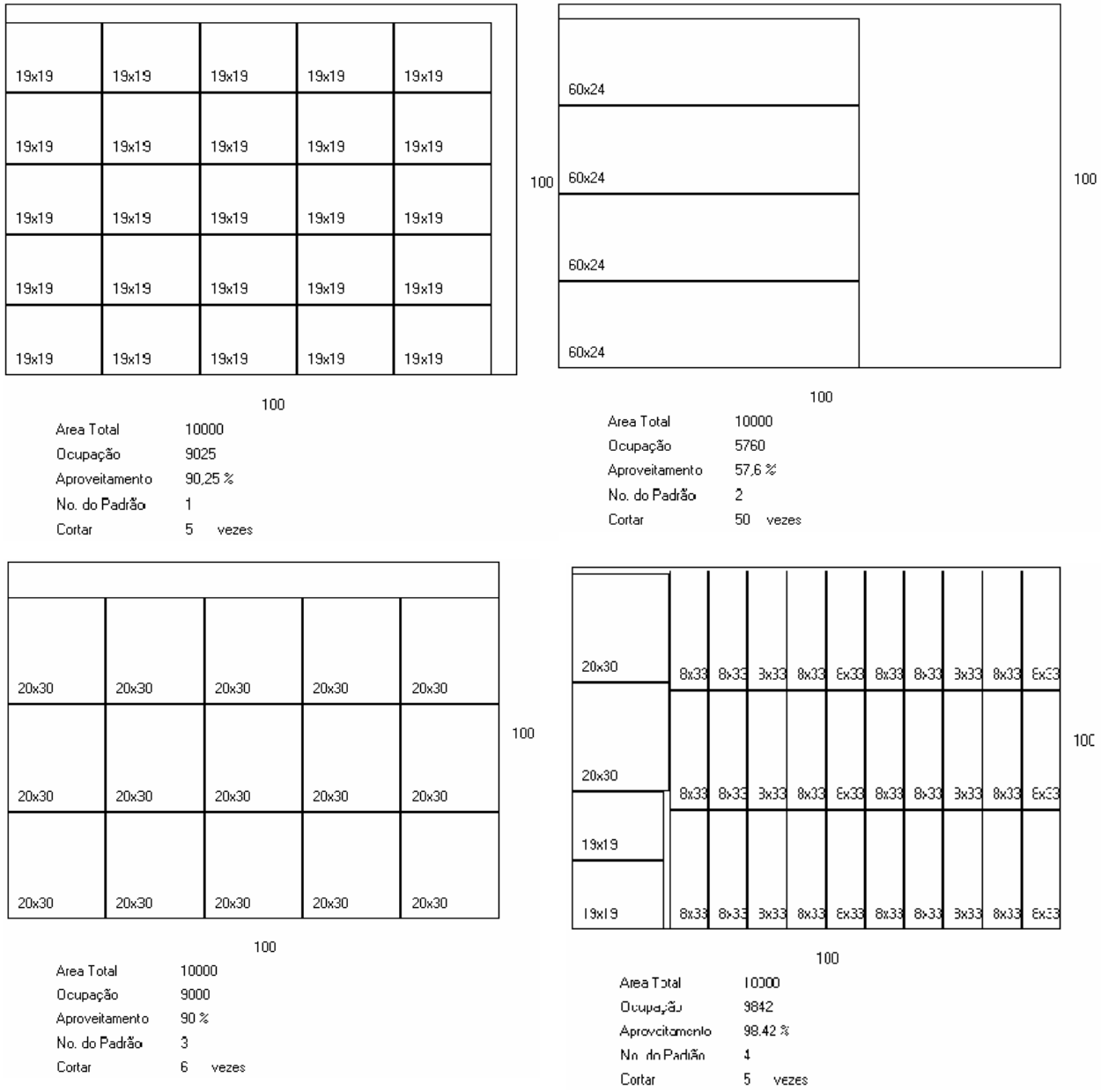
Quadro 7 – Exemplo de Caso

Peça	Dimensão horizontal	dimensão vertical	demanda
P01	19	19	125
P02	20	30	90
P03	60	24	200
P04	8	33	150

Partindo-se do princípio que não existam peças disponíveis em estoque, submetendo-se estes dados ao sistema e solicitando o cálculo de cortes e visualização dos *layout's*, serão apresentados os padrões de corte demonstrados na fig.13.

Com a otimização realizada foram obtidas as seguintes quantidades de peças para corte: P01 = 135, P02 = 100, P03 = 200, P04 = 150. Assim, as demandas serão atendidas, e sobrarão peças que deverão ser registradas no estoque.

Figura 13 – *Layout´s* Produzidos.



Para efeito de ilustração os quatro *layout's* gerados foram agrupados em apenas uma página, porém o sistema imprime um padrão de corte em cada página. Nota-se no quarto padrão que para garantir que sejam preservadas as dimensões das peças, após o corte vertical que origina as peças de dimensões 20x30 (além das 19x19) o usuário deve descartar temporariamente o pedaço à direita deste corte, tomar o pedaço à esquerda, rotacioná-lo em 90 graus e efetuar em seguida um corte horizontal neste, separando então as peças de dimensões 19x19.

5 CONCLUSÃO

Com o desenvolvimento deste trabalho pode-se concluir que o problema do corte em 2 dimensões é realmente um tema de relevância científica e de grande complexidade.

O objetivo principal foi atingido, pois o sistema disponibiliza ao usuário *layout´s* otimizados de padrões de corte. É importante ressaltar que estes *layout´s* podem algumas vezes não ser a melhor solução, mas sim uma solução adequada à necessidade do usuário.

Assim, como ponto forte pode-se citar que de maneira não exaustiva, o sistema apresenta soluções adequadas para o problema de cortes bidimensionais. O sistema apresenta melhores resultados quando utilizado com tamanhos de peças bem variados. Entretanto em alguns casos, mesmo resolvido o modelo matemático de minimização, pode ocorrer um excesso nas quantidades de peças que sobram. No caso de não serem mais utilizadas, poder-se-ia considerá-las desperdício.

Outro fator a considerar é que pode ocorrer no *layout* gerado, sobrar boa área aproveitável da placa (como pode ser visto no padrão 2 da fig. 13), devido ao término do tempo de cálculo estipulado pelo usuário. Estas por sua vez, poderiam ser consideradas novas "placas padrão" a serem utilizadas nos cortes.

A utilização de um armazenamento de sobras de peças em estoque permitiu o arredondamento (para cima) da quantidade de vezes que um padrão de cortes deve ser utilizado, solucionando assim o problema com as frações do sistema linear modelado. Ou seja, permite por exemplo, que ao invés de um padrão de cortes ser utilizado 1,7 vezes para atender uma demanda de peças, o mesmo seja utilizado 2 vezes e o usuário armazena as peças que sobram deste corte.

5.1 LIMITAÇÕES E CARACTERÍSTICAS

O sistema apresenta soluções para cada corte dentro de um prazo máximo de tempo estipulado pelo usuário no momento do corte. Nota-se também que uma diferença demasiada de proporções entre a placa a ser usada no corte e as peças em demanda, pode ocasionar problemas de qualidade de resolução na impressão dos *layouts* gerados, pois foi definido que cada padrão ocupa apenas uma página.

O sistema demonstrou-se estável rodando na plataforma Windows 98 em um microcomputador Pentium III 550 Mhz, com 64 Mbytes de memória RAM.

5.2 SUGESTÕES

Como sugestões para trabalhos futuros, ou mesmo o aperfeiçoamento do sistema, pode-se citar o uso de outros algoritmos que possam buscar além das soluções ótimas, uma maior proximidade às soluções exatas. Outra sugestão seria a atualização automática de estoques de placas e peças que sobram, além de um gerenciamento da programação de pedidos levando em consideração o prazo de entrega dos mesmos.

Interessante também seria a incorporação de alguns módulos que permitam o gerenciamento de custos de produção, como compras, contas a pagar, faturamento, contas a receber, além de estatísticas das incidências dos cortes efetuados.

5.3 REFERÊNCIAS BIBLIOGRÁFICAS

BRONSON, Richard. **Pesquisa operacional**. São Paulo: McGraw_Hill do Brasil, 1985.

CANTÚ, Marco. **Dominando o Delphi 5**. São Paulo: Makron Books, 2000.

FELICIANO NETO, A. **Engenharia da informação:** metodologia, técnicas e ferramentas. São Paulo: McGraw-Hill, 1988.

GOLDBARG, Marco Cesar. **Otimização combinatória e programação linear:** modelos e algoritmos. Rio de Janeiro: Campus, 2000.

LOESCH, Cláudio, Nelson Hein. **Pesquisa operacional:** fundamentos e modelos. Blumenau: Ed. da FURB, 1999.

MERCADO-GARDNER, Juanita. **Projetando bancos de dados com Access 2**. Tradução Elisa M. Ferreira. São Paulo: Berkeley, 1995.

NASCIMENTO, Hugo A. D. do, LONGO, Humberto Jose, ALOISE, Dario Jose. Uma heurística o(mn) para o corte bidimensional guilhotinado. In: Congresso da Sociedade Brasileira de Pesquisa Operacional, 31., 1999, Juiz de Fora. **Anais do XXXI Congresso da Sociedade Brasileira de Pesquisa Operacional.** Juiz de Fora: UFMG, 1999. Disponível em: <http://www.cs.usyd.edu.au/~hadn/>. Acesso em: 23 mar. 2002.

POMPILHO, S. **Análise essencial**. Rio de Janeiro: Infobook, 1994.

RAMALHO, Jose Antonio A. (Jose Antonio Alves). **Access 2 for Windows**. Sao Paulo: Makron Books, 1995.

RANGEL, Maria do Socorro Nougueira. **O problema do corte bidimensional.** 1990. 97 f. Tese (Matemática Aplicada) - Instituto de Matemática, Estatística e Ciências da Computação, Universidade Estadual de Campinas, Campinas.

SAVELTTI, Dirceu Douglas. **Algoritmos.** São Paulo: Makron Books, 1998.

SHILDT, Herbert. **Borland C++: completo e total**. São Paulo : Makron Books, 1997.

SILVA, Alexandre Gonçalves. **O problema do corte bidimensional.** Campinas, [1999]. Disponível em: <http://www.ic.unicamp.br/~alexgs/ic.html>. Acesso em: 08 mar. 2002.

SONNINO, Bruno. **Desenvolvendo aplicações com Delphi 5.** São Paulo: Makron Books, 2000.

SOUZA, André Luiz. **Criação de tabelas utilizando o Sybase**. Central, Uberaba, [2000]. Disponível em: < http://xfk.vilabol.uol.com.br/any/sqlany.htm >. Acesso em: 23 mar. 2002.

SZWARCFITER, Jayme Luiz. **Grafos e algoritmos computacionais.** Rio de Janeiro: Campus, 1984.

WAGNER, Harvey M. **Pesquisa operacional**. 2.ed. Rio de Janeiro : Prentice-Hall do Brasil, 1985.

ANEXO I

Dicionário de Dados do Sistema Otimizador de Corte de Placas Retangulares aplicado ao corte de chapas de papelão reciclado. O formato apresentado foi adaptado (traduzido) do relatório extraído através da opção ferramentas, analisar, documentador do SGBD utilizado.

Tabela: ITENS_PEDIDO

COLUNA:		Tipo	Tamanho
COD_PED		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Número do pedido		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
COD_PECA		Número (Inteiro)	2
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Código da peça		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
QTD_PECA_PED		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	0		
Descrição:	Quantidade da peça		
PermitirComprimentoZero:	Falso		
Regra de validação:	>0		
Requerido:	Verdadeiro		

Tabela: MOVIMENTO_PECAS

COLUNA:		Tipo	Tamanho
NUM_MOV_PECA		Número (Longo)	4
Atributos:	Tamanho fixo, AutoIncrementar		
Descrição:	Número de movimentação da peça		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
Tabela de origem:	MOVIMENTO_PECAS		
COD_PROG		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Código da programação/entrada (originou o mov.)		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DATA_MOV_PECA		Data/Hora	8
Atributos:	Tamanho fixo		
Descrição:	Data do movimento		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
QTD_ENT_PECA		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Quantidade entrada		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
QTD_SAI_PECA		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Quantidade saida		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
SALDO_MOV_PECA		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		

	Descrição:	Saldo	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
COD_PECA		Número (Inteiro)	2
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Código da peça movimentada	
	PermitirComprimentoZero:	Falso	
	Requerido:	Verdadeiro	

Tabela: MOVIMENTO_PLACAS

COLUNA:		Tipo	Tamanho
NUM_MOV_PLACA		Número (Longo)	4
	Atributos:	Tamanho fixo, AutoIncrementar	
	Descrição:	Número de movimentação da peça	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
COD_PROG		Número (Longo)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Código da programação/entrada (originou o mov.)	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
DATA_MOV_PLACA		Data/Hora	8
	Atributos:	Tamanho fixo	
	Descrição:	Data do movimento	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
QTD_ENT_PLACA		Número (Longo)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Quantidade entrada	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
QTD_SAI_PLACA		Número (Longo)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Quantidade saida	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
SALDO_MOV_PLACA		Número (Longo)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Saldo	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
COD_PLACA		Número (Inteiro)	2
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Código da placa movimentada	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	

Tabela: PECAS

COLUNA:		Tipo	Tamanho
COD_PECA		Número (Inteiro)	2
	Atributos:	Tamanho fixo	
	Casas decimais:	0	
	Descrição:	Código da Peça	
	ExibirControle:	Caixa de texto	
	Formato:	0000	
	PermitirComprimentoZero:	Falso	
	Requerido:	Verdadeiro	
DESC_PECA		Texto	30
	Atributos:	Comprimento variável	
	Descrição:	Descrição da Peça	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
	UnicodeCompression:	Verdadeiro	
DIMX_PECA		Número (Simples)	4
	Atributos:	Tamanho fixo	

Casas decimais:	Automático		
Descrição:	Dimensão horizontal da peça		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DIMY_PECA		Número (Simples)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Dimensão vertical da peça		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		

Tabela: PEDIDOS

COLUNA:		Tipo	Tamanho
COD_PED		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Número do pedido		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
ALIAS_PED		Texto	40
Atributos:	Comprimento variável		
Descrição:	Alias (apelido) do pedido		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DATA_PED		Data/Hora	8
Atributos:	Tamanho fixo		
Descrição:	Data do pedido		
Máscara de entrada:	99/99/00;0;_		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
CORTE_EFETUADO_PED		Sim/Não	1
Atributos:	Tamanho fixo		
Descrição:	Marcador		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
Valor padrão:	0		
COD_PROG		Número (Longo)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	codigo da programacao (após o corte)		
PermitirComprimentoZero:	Falso		
Requerido:	Falso		
Valor padrão:	0		

Tabela: PLACAS

COLUNA:		Tipo	Tamanho
COD_PLACA		Número (Inteiro)	2
Atributos:	Tamanho fixo		
Casas decimais:	0		
Descrição:	Código da placa padrão		
Formato:	0000		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DESC_PLACA		Texto	25
Atributos:	Comprimento variável		
Descrição:	Descrição da placa		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DIMX_PLACA		Número (Simples)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		
Descrição:	Dimensão horizontal		
PermitirComprimentoZero:	Falso		
Requerido:	Verdadeiro		
DIMY_PLACA		Número (Simples)	4
Atributos:	Tamanho fixo		
Casas decimais:	Automático		

	Descrição:	Dimensão vertical	
	PermitirComprimentoZero:	Falso	
	Requerido:	Verdadeiro	
EST_PLACA		Número (Longo)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Quantidade em estoque	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	

Tabela: PROGRAMACOES_CORTE

COLUNA:		Tipo	Tamanho
COD_PROG		Número (Longo)	4
	Atributos:	Tamanho fixo, AutoIncrementar	
	Descrição:	Número da Programação de Corte	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
DATA_PROG		Data/Hora	8
	Atributos:	Tamanho fixo	
	Descrição:	Data do Corte	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	
APROV_PROG		Número (Simples)	4
	Atributos:	Tamanho fixo	
	Casas decimais:	Automático	
	Descrição:	Percentual de Aproveitamento do Corte	
	Formato:	Porcentagem	
	PermitirComprimentoZero:	Falso	
	Requerido:	Falso	

ANEXO II

Função principal cortes que calcula os padrões de corte de forma a reduzir a quantidade de placas a ser cortada.

```
tipo_padrao *cortes(float posicao_x, float posicao_y, float dimensao_A, float dimensao_B, tipo_S *pS, int POS_P, int
POS_Q,  int *n, int *n_P, int *n_Q, int *n_PQ, int *n_D, float *P, float *Q, tipo_PQ *PQ,   int *ind, int D, tipo_padrao
**lista_padroes, tipo_padrao_gerado **PG, tipomatriz matriz_B, tipomatriz copiaB, tipovetor vetor_x, tipovetor vetor_y,
                tipovetor vetor_d, tipovetor res1, tipovetor vetor_a, tipohisto H, time_t *tempo)
{
float         peso=0.0; /*area do retangulo*/
float         peso_aux=0.0;
int                ind_preench=0; /* indice de preenchimento da area*/
tipo_padrao   *padrao0=NULL, *padrao1=NULL, *padrao2=NULL, *padrao_aux=NULL;
register int   k=0;   /* retângulo atual*/
int           k_max=-1; /*retângulo que melhor preencheu a a área atual */
register int   pos_P, pos_Q; /*posicao atual de cada vetor*/
register int   i_, j_; /*variaveis auxiliares para matrizes*/
register float soma; /* variavel para verificação da otimização*/
float         menor_t=0.0;  /* menor valor encontrado para entrada da nova coluna*/
int           col=0;  /*valor atual da coluna da matriz*/
/* n_D = profundidade máxima / D = profundidade atual*/

  FrmProgramacoes->ProgressBar1->Position++;
  if (FrmProgramacoes->ProgressBar1->Position == 100) FrmProgramacoes->ProgressBar1->Position = 0;
  while ((k < *n) && (ind_preench < *ind))
  {
    peso_aux = gera_homog(&dimensao_A, &dimensao_B, &pS[k].a, &ind_preench);
    if (peso_aux > peso)
    {
      k_max = k;
      peso = peso_aux;
    }
    k++;
  }

  k = k_max;
  padrao0 = gera_padrao(padrao0, &posicao_x, &posicao_y, &dimensao_A, &dimensao_B, &pS[k].a, &k_max, &peso);
  tempo[2] = time(&tempo[2]);  /* tempo atual (tempo[1] = tempo inicial) */
  if ( peso==0 || difftime(tempo[2],tempo[1]) > tempo[0])
```

```
 {
  return (padrao0);
 }
 for (pos_P=0; pos_P<*n_P; pos_P++)
 {
   if ((POS_P < *n_P) && (D <= *n_D) && (P[POS_P] < dimensao_A) && corte_ok(P[POS_P], dimensao_B, n_PQ, PQ))
   {
     padrao1 = cortes(posicao_x, posicao_y, P[POS_P], dimensao_B, pS, POS_P, 0, n, n_P, n_Q, n_PQ, n_D, P, Q, PQ, ind,
D+1, lista_padroes, PG,
                          matriz_B, copiaB, vetor_x, vetor_y, vetor_d, res1, vetor_a, H, tempo);
     if (corte_ok(dimensao_A-P[POS_P], dimensao_B, n_PQ, PQ))
          padrao2 = cortes(posicao_x+P[POS_P], posicao_y, dimensao_A-P[POS_P], dimensao_B, pS, POS_P, 0, n, n_P,
n_Q, n_PQ, n_D, P, Q, PQ, ind, D+1, lista_padroes, PG,
                            matriz_B, copiaB, vetor_x, vetor_y, vetor_d, res1, vetor_a, H, tempo);
     else
          padrao2 = NULL;
     if (padrao2 == NULL) peso_aux = padrao1->peso;
     else          peso_aux = padrao1->peso + padrao2->peso;
     if (peso_aux > peso)
     {
          peso = peso_aux;
          padrao0 = gera_padrao_concat(padrao0, padrao1, padrao2, &peso);
          if (!achou_padrao_gerado(padrao0,PG,H,n))
          {
           /* Cria vetor a */
           for (i_=0; i_<*n; i_++)
             vetor_a[i_] = (float) H[i_];
           /* Copia da matriz dimensao_B */
           for (i_=0; i_<*n; i_++)
             for(j_=0; j_<*n; j_++)
               copiaB[i_][j_] = matriz_B[i_][j_];
           /* Passo c - Resolucao do sistema yB = [1] */
           Transpoematriz(matriz_B,*n);
           ResolveSL(matriz_B, res1, vetor_y, *n);

           /* Sistema resolvido - B retorna a forma original */
           for (i_=0; i_<*n; i_++)
             for(j_=0; j_<*n; j_++)
               matriz_B[i_][j_] = copiaB[i_][j_];
           soma=0;
           for (i_=0; i_<*n; i_++)
             soma = soma + vetor_y[i_]*vetor_a[i_];
```

```
          if (soma > 1)
          {
            ResolveSL(matriz_B, vetor_a, vetor_d, *n);  /* Passo e - Resolucao do sistema Bd = a */
            /* Sistema resolvido - B retorna a forma original */
            for (i_=0; i_<*n; i_++)
              for(j_=0; j_<*n; j_++)
                    matriz_B[i_][j_] = copiaB[i_][j_];
            menor_t = Passof(vetor_x, vetor_d, &col, *n);
            Passog(matriz_B, vetor_x, vetor_a, vetor_d, menor_t, col, *n);
            if (!inclui_padrao(col, lista_padroes, padrao0))
            {
       ShowMessage(AnsiString("Memoria insuficiente"));
              exit(0);
            }
          }
        }
}
else
{
        padrao_aux = gera_padrao_concat(padrao_aux, padrao1, padrao2, &peso_aux);
        if (!achou_padrao_gerado(padrao_aux,PG,H,n))
        {
          /* Cria vetor a */
          for (i_=0; i_<*n; i_++)
            vetor_a[i_] = (float) H[i_];
          /* Copia da matriz B */
          for (i_=0; i_<*n; i_++)
            for(j_=0; j_<*n; j_++)
              copiaB[i_][j_] = matriz_B[i_][j_];
          /* Passo c - Resolucao do sistema yB = [1] */
          Transpoematriz(matriz_B,*n);
          ResolveSL(matriz_B, res1, vetor_y, *n);
          /* Sistema resolvido - B retorna a forma original */
          for (i_=0; i_<*n; i_++)
            for(j_=0; j_<*n; j_++)
              matriz_B[i_][j_] = copiaB[i_][j_];
          soma=0;
          for (i_=0; i_<*n; i_++)
            soma = soma + vetor_y[i_]*vetor_a[i_];
          if (soma > 1)
          {
```

```
            ResolveSL(matriz_B, vetor_a, vetor_d, *n);  /* Passo e - Resolucao do sistema Bd = a */
            /* Sistema resolvido - B retorna a forma original */
            for (i_=0; i_<*n; i_++)
              for(j_=0; j_<*n; j_++)
                    matriz_B[i_][j_] = copiaB[i_][j_];
            menor_t = Passof(vetor_x, vetor_d, &col, *n);
            Passog(matriz_B, vetor_x, vetor_a, vetor_d, menor_t, col, *n);
            if (!inclui_padrao(col, lista_padroes, padrao_aux))
            {
        ShowMessage(AnsiString("Memoria insuficiente"));
              exit(0);
            }
           }
          }
          apaga_padrao(&padrao_aux);
    }
    apaga_padrao(&padrao1);
    apaga_padrao(&padrao2);
   }
   POS_P++;
 }
 for (pos_Q=0; pos_Q<*n_Q; pos_Q++)
 {
   if ((POS_Q < *n_Q) && (D <= *n_D) && (Q[POS_Q] < dimensao_B) && corte_ok(dimensao_A, Q[POS_Q], n_PQ,
PQ))
   {
    padrao1 = cortes(posicao_x, posicao_y, dimensao_A, Q[POS_Q], pS, 0, POS_Q, n, n_P, n_Q, n_PQ, n_D, P, Q, PQ, ind,
D+1, lista_padroes, PG,
                    matriz_B, copiaB, vetor_x, vetor_y, vetor_d, res1, vetor_a, H, tempo);
    if (corte_ok(dimensao_A, dimensao_B-Q[POS_Q], n_PQ, PQ))
          padrao2 = cortes(posicao_x, posicao_y+Q[POS_Q], dimensao_A, dimensao_B-Q[POS_Q], pS, 0, POS_Q, n, n_P,
n_Q, n_PQ, n_D, P, Q, PQ, ind, D+1, lista_padroes, PG,
                     matriz_B, copiaB, vetor_x, vetor_y, vetor_d, res1, vetor_a, H, tempo);
    else
          padrao2 = NULL;
    if (padrao2 == NULL) peso_aux = padrao1->peso;
    else          peso_aux = padrao1->peso + padrao2->peso;
    if (peso_aux > peso)
    {
          peso = peso_aux;
          padrao0 = gera_padrao_concat(padrao0, padrao1, padrao2, &peso);
          if (!achou_padrao_gerado(padrao0,PG,H,n))
```

```
        {
          /* Cria vetor a */
          for (i_=0; i_<*n; i_++)
            vetor_a[i_] = (float) H[i_];
          /* Copia da matriz B */
          for (i_=0; i_<*n; i_++)
            for(j_=0; j_<*n; j_++)
              copiaB[i_][j_] = matriz_B[i_][j_];
          /* Passo c - Resolucao do sistema yB = [1] */
          Transpoematriz(matriz_B,*n);
          ResolveSL(matriz_B, res1, vetor_y, *n);
          /* Sistema resolvido - B retorna a forma original */
          for (i_=0; i_<*n; i_++)
            for(j_=0; j_<*n; j_++)
              matriz_B[i_][j_] = copiaB[i_][j_];
          soma=0;
          for (i_=0; i_<*n; i_++)
            soma = soma + vetor_y[i_]*vetor_a[i_];
          if (soma > 1)
          {
            ResolveSL(matriz_B, vetor_a, vetor_d, *n);  /* Passo e - Resolucao do sistema Bd = a */
            /* Sistema resolvido - B retorna a forma original */
            for (i_=0; i_<*n; i_++)
              for(j_=0; j_<*n; j_++)
                    matriz_B[i_][j_] = copiaB[i_][j_];
            menor_t = Passof(vetor_x, vetor_d, &col, *n);

            Passog(matriz_B, vetor_x, vetor_a, vetor_d, menor_t, col, *n);
            if (!inclui_padrao(col, lista_padroes, padrao0))
            {
      ShowMessage(AnsiString("Memoria insuficiente"));
              exit(0);
            }
          }
        }
}
else
{
        padrao_aux = gera_padrao_concat(padrao_aux, padrao1, padrao2, &peso_aux);
        if (!achou_padrao_gerado(padrao_aux,PG,H,n))
        {
```

```
            /* Cria vetor a */
            for (i_=0; i_<*n; i_++)
              vetor_a[i_] = (float) H[i_];
            /* Copia da matriz B */
            for (i_=0; i_<*n; i_++)
              for(j_=0; j_<*n; j_++)
                copiaB[i_][j_] = matriz_B[i_][j_];
            /* Passo1 - Resolucao do sistema yB = [1] */
            Transpoematriz(matriz_B,*n);
            ResolveSL(matriz_B, res1, vetor_y, *n);
            /* Sistema resolvido - B retorna a forma original */
            for (i_=0; i_<*n; i_++)
              for(j_=0; j_<*n; j_++)
                matriz_B[i_][j_] = copiaB[i_][j_];
            soma=0;
            for (i_=0; i_<*n; i_++)
              soma = soma + vetor_y[i_]*vetor_a[i_];
            if (soma > 1)
            {
          ResolveSL(matriz_B, vetor_a, vetor_d, *n);  /* Passo e - Resolucao do sistema Bd = a */
              /* Sistema resolvido - B retorna a forma original */
              for (i_=0; i_<*n; i_++)
                for(j_=0; j_<*n; j_++)
                      matriz_B[i_][j_] = copiaB[i_][j_];
              menor_t = Passof(vetor_x, vetor_d, &col, *n);
              Passog(matriz_B, vetor_x, vetor_a, vetor_d, menor_t, col, *n);
              if (!inclui_padrao(col, lista_padroes, padrao_aux))
              {
          ShowMessage(AnsiString("Memoria insuficiente"));
                exit(0);
              }
            }
           }
           apaga_padrao(&padrao_aux);
    }
    apaga_padrao(&padrao1);
    apaga_padrao(&padrao2);
  }
  POS_Q++;
}
return (padrao0); }
```