Livia Bastos Gratz
RA: 085.057

Projeto MT852 – 1º. Semestre 2009

1. Objetivo

O objetivo deste Projeto é aplicar o Método Heurístico Simulated Annealing em Problemas de Corte Unidimensional. A idéia inicial era a de replicar uma metodologia sugerida no artigo [1] *A simulated annealing heuristics for the one-dimensional cutting stock problem*. Porém, por não se conseguir resultados satisfatórios, esta metodologia sofreu algumas alterações que serão elucidadas ao longo do relatório.

2. Introdução

Simulated Annealing é um método computacional iterativo que se propõe a resolver grandes problemas combinatoriais de otimização através de processos aleatórios controlados. Ele simula o processo físico de fundição de metais que de forma geral, envolve os seguintes passos:

1. A temperatura do sistema é aumentada até um nível satisfatório.
2. A temperatura é mantida neste nível por um determinado período de tempo.
3. O sistema é resfriado de forma controlada até que se atinja o estado de energia desejado.

Se o sistema é resfriado bruscamente, ele pode solidificar em um indesejável nível de energia. E utilizando esta analogia para problemas de otimização, o nível energético do sistema seria ao valor da função objetivo. Assim, solidificar em um estado energético maior que o desejável significaria encontrar um mínimo local como solução do problema.

Para evitar que o sistema “solidifique” em um mínimo local, o algoritmo precisa mover-se devagar (em relação ao valor da função objetivo) dentro do conjunto das soluções possíveis. Este processo engloba aceitar os movimentos que aumentam a função objetivo, com certa probabilidade que decresce à medida que o algoritmo evolui.

O processo geral para implementação de um algoritmo de Simulated Annealing segue abaixo.

1. Selecione uma temperatura inicial t_0 e uma solução inicial x_0. $f_0 = f(x_0)$ é o valor inicial da função objetivo. Faça $i = 0$ e vá para o passo 2.

2. Faça $i = i + 1$, gere aleatoriamente uma solução x_i e calcule $f_i = f(x_i)$.

3. Se $f_i < f_{i-1}$ vá para o passo 5. Caso contrário, aceite f_i como nova solução de acordo com a probabilidade $e^{|f_i - f_{i-1}|/t}$.

4. Se f_i foi rejeitada como nova solução no passo 3, faça $f_i = f_{i-1}$.

5. Se estiver satisfeito com o valor da função objetivo f_i, então pare. Se não, decresça a temperatura t_0 é vá para o passo 2.

O problema de corte unidimensional surge de processos industriais que precisam cortar rolos de matéria-prima em rolos de tamanhos menores visando atender a uma demanda. O objetivo deste problema é determinar o conjunto de padrões de corte e a quantidade de vezes que serão utilizados de forma a minimizar o desperdício de matéria-prima e o custo de *setup*.

O custo de setup ocorre quando um padrão de corte é trocado por outro. Este custo é atribuído à mão-de-obra, e à perda de produtividade enquanto as facas são posicionadas nos seus lugares e precisamente ajustadas dentro da máquina de corte.

A maioria das técnicas utilizadas para resolução de Problemas de Corte Unidimensional envolve algoritmos baseados em Programação Linear. Porém, este problema se torna complicado quando existem diferentes tamanhos de rolos de matéria-prima ou diferentes tamanhos de rolos finais. Por exemplo,

como apontado em trabalhos prévios, um rolo de matéria-prima de 200” e demanda por 40 tipos de rolos finais variando de 20” até 80” geram uma quantidade de padrões de corte que excedem a grandeza dos milhões.

Tais problemas grandes não podem ser resolvidos por Métodos de Programação Linear Inteira dentro de um período de tempo computacional satisfatório. Por isso que as técnicas de Programação Linear permanecem como as mais aceitas. Nestas técnicas, as soluções inteiras são obtidas arredondando os valores das variáveis de decisão para o próximo inteiro. Contudo, o arredondamento pode levar a soluções não-ótimas ou até a soluções inviáveis.

3. Modelagem do Problema

A seguir seguem algumas definições necessárias à modelagem do Problema de Corte Unidimensional.

$L =$ largura da matriz de corte (matéria-prima).

$l_i =$ largura do item final *i* após processo de corte.

$a_{ij} =$ matriz que define a quantidade de itens *i* produzidos ao se utilizar o padrão de corte *j*.

$x_j =$ número de vezes que o padrão de corte *j* será utilizado.

$y_j =$ variável binária que vale um quando o padrão de corte *j* é utilizado e vale zero caso contrário.

$desp_j =$ desperdício de matéria-prima atribuído ao padrão de corte *j.*

$dem_i =$ demanda por itens de tamanho final *i*.

$fol.ga_i =$ excesso de produção dos itens de tamanho final *i*.

$cs =$ custo de setup.

$cmp =$ custo por unidade de desperdício de matéria-prima.

$M =$ constante muito maior que um.

$G =$ constante igual à máxima demanda.

O modelo pode ser expresso como:

$$\min\left\{ cmp.\sum_j desp_j x_j + cs.\sum_j y_j + M.\sum_i fol.ga_i \right\}$$

sujeito a

$$\sum_j a_{ij} x_j = dem_i + fol.ga_i \qquad (1)$$

$$x_j - Gy_j \le 0 \quad \forall j \qquad (2)$$

$$x_j \ge 0 \text{ e inteiro} \quad \forall j \qquad (3)$$

$$y_j \in \{0,1\} \qquad \forall j \qquad (4)$$

$$fol.ga_i \ge 0 \qquad \forall i \qquad (5)$$

Na formulação da função objetivo, os parâmetros $cmp = 10$, $cs = 100$ e $M = 10$ serão sempre fixos nestes valores. M funciona na verdade como uma penalidade ao excesso de produção, traduzido na variável $fol.ga_i$.

A restrição (1) garante o atendimento da demanda e permite um excesso de produção que será penalizado na função objetivo. Já a restrição (2) existe para garantir a definição da variável y_j e G é uma constante que irá valer o máximo valor da demanda por algum item *i*. Isto porque x_j nunca será maior ou igual a esta demanda. As outras restrições, de (3) a (5) garantem a não-negatividade das variáveis e define x_j como inteiro e y_j como variável binária.

4. Algoritmo para resolução do Problema

A metodologia sugerida na referência [1] consiste em três etapas detalhadas abaixo:

4.1 Geração dos padrões de Corte

A proposta é utilizar um processo enumerativo para identificar todos os possíveis padrões de corte. Alguns trabalhos sugerem que os padrões de corte com grande desperdício de matéria-prima podem ser eliminados, mas isto pode

levar a soluções não-ótimas. Como a idéia do projeto é utilizar um Método Heurístico eficiente capaz de lidar com problemas de grande porte, a eliminação de padrões de corte não será necessária.

A geração dos padrões de corte será baseada na metodologia proposta no artigo [2] *Pattern generating procedure for the cutting stock problem*. A idéia do artigo é gerar uma árvore de possibilidades onde cada nível representa o tamanho do item final, em ordem decrescente, e cada nó a largura da matriz de corte que falta ser cortada. A figura 1 fornece o exemplo de uma matriz de corte de 130 cm e itens finais de 50, 40 e 30 cm. O último nível é o desperdício de matéria-prima de cada padrão. A árvore gerou 8 padrões de corte diferentes que pode ser visualizados na tabela 1.

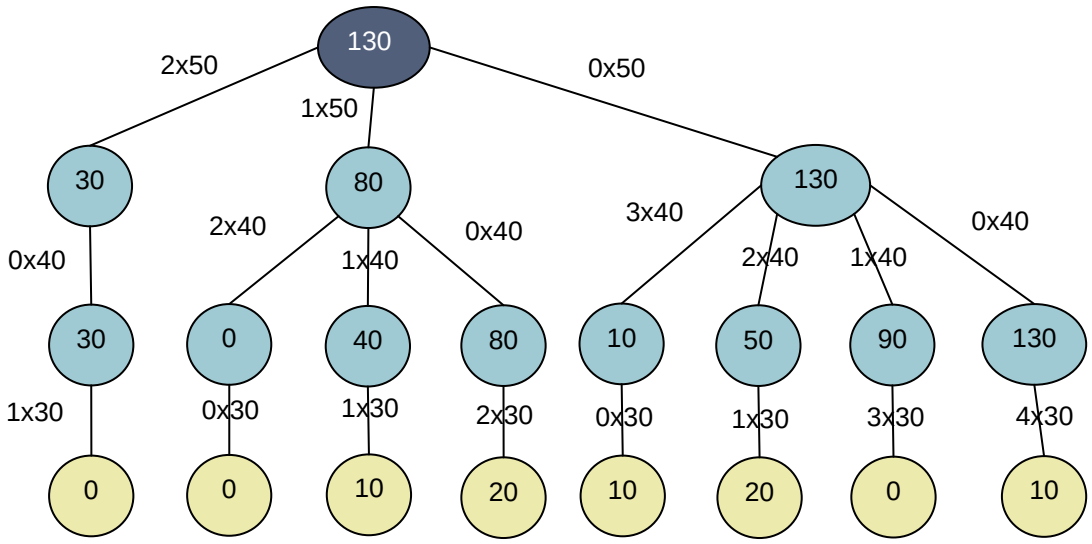

Figura 1: Árvore de possibilidades para matriz de 130cm e itens de 50, 40 e 30 cm.

Tabela 1: Padrões de corte gerados

Padrões	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
50 cm	2	1	1	1	0	0	0	0
40 cm	0	2	1	0	3	2	1	0
30 cm	1	0	1	2	0	1	3	4
Desperdício	**0**	**0**	**10**	**20**	**10**	**20**	**0**	**10**

Note que cada padrão de corte será uma coluna da matriz $[a_{ij}]$ definida anteriormente como a quantidade de itens finais *i* produzidos ao se utilizar o padrão de corte *j*. Neste exemplo, $[a_{ij}] \in \mathrm{M}_{3x8}(\Re)$. Foi criada em Matlab a *function* **gerador** para a geração dos padrões de corte possíveis dados o tamanho da matriz de corte L e o vetor $[l_i]$ com a largura de cada item em ordem decrescente. O código computacional encontra-se abaixo.

```
function [a,desp] = gerador (L,l)
%
%Vetor l: tamanhos dos itens onde l1 > l2 > l3 > ... > lm
%Peenchimento da 1a. coluna de A
%
[m] = length(l);
j = 1;
a(1,j) = floor(L/l(1));
sum = a(1,j)*l(1);
for i = 2:m
   a(i,j) = floor( (L - sum)/l(i) );
   sum = sum + a(i,j)*l(i);
end
desp(j) = L - sum;
%
i = m - 1;
%
while (i > 0 & j <= 20000)
   if ( a(i,j) == 0 )
      i = i - 1;
   else
      j = j + 1;
      zum = 0;
      for z = 1 : (i-1)
         a(z,j) = a(z,j-1);
         zum = zum + a(z,j)*l(z);
      end
      a(i,j) = a(i,j-1) - 1;
      zum = zum + a(i,j)*l(i);
      %
      for k = (i+1) : m
         a(k,j) = floor ((L-zum)/l(k));
         zum = zum + a(k,j)*l(k);
      end
      desp(j) = L - zum;
      i = m - 1;
   end
end
```

Note que no algoritmo estamos limitando a geração de padrões até 20.000 tipos diferentes. Isso foi necessário porque para problemas maiores o tempo de processamento era muito longo e decidiu-se por encontrar a solução ótima

trabalhando somente com esta quantidade de padrões. Assim, a matriz $[a_{ij}]$ possuirá no máximo 20.000 colunas. A quantidade de linhas será determinada pela quantidade de itens finais demandados.

4.2. Geração da solução inicial factível

A solução inicial factível será o x_0 do algoritmo de Simulated Annealing. Apesar de não ser obrigatório começar o processo com uma solução factível, alguns trabalhos já demonstraram que a convergência do método para o ótimo não é garantida e que o resultado é normalmente pior do que o encontrado nas técnicas de PL com arredondamento para o próximo inteiro.

Assim, para determinação de uma solução inicial factível, será resolvido o Problema de Programação Linear Relaxado sem considerar os custos de setup e x_0 será arredondada para o maior inteiro mais próximo. Ao fazer isto garanto a restrição (1) das demandas a serem atendidas. Neste passo, não será utilizada a variável y_j e conseqüentemente a restrição (2) é eliminada. O problema se resume então a:

$$\min\left\{ cmp.\sum_j desp_j x_j + M.\sum_i fol.ga_i \right\}$$

sujeito a

$$\sum_j a_{ij} x_j = dem_i + fol.ga_i \qquad (1)$$

$$x_j \geq 0 \qquad \forall j \qquad (2)$$

$$fol.ga_i \geq 0 \qquad \forall i \qquad (3)$$

A solução inicial factível foi processada no software *xpress*, cuja modelagem encontra-se abaixo. O arquivo *adesp.dat* irá fornecer a matriz $[a_{ij}]$, o vetor de desperdício $[desp_i]$ e o vetor de demanda $[dem_i]$.

```
model "Encontra solucao do problema inicial relaxado"
uses "mmxprs"; !gain access to the Xpress-Optimizer solver
!
parameters
    mlim = 38
    nlim = 3*mlim
    DATAFILE = "adesp.dat"
```

```
    OUTFILE = "result.dat"
end-parameters
!
declarations
m = 1..mlim
n = 1..nlim
x: array (n) of mpvar
folga: array (m) of mpvar
a: array (m,n) of integer
desp: array (n) of integer
d: array (m) of integer
end-declarations
!
initializations from "adesp.dat"
a desp d
end-initializations
!
forall (i in m) sum(j in n) a(i,j)*x(j) = d(i) + folga(i)
!
forall (j in n) x(j)>=0
forall (i in m) folga(i)>=0
!
desperdicio:= sum(j in n) desp(j)*x(j)
folgas:= sum ( i in m) folga(i)
minimize (10*desperdicio + 10*folgas)
!
writeln ("Desperdicio: ", getobjval)
write ("x: [ ")
forall (j in n) write (getsol(x(j))," ")
writeln ("]")
forall (i in m) writeln ("folga",i,": ", getsol(folga(i)))
!
fopen ("result", F_OUTPUT)
writeln ("Desperdicio: ", getobjval)
write ("x: [ ")
forall (j in n) write (getsol(x(j))," ")
writeln ("]")
forall (i in m) writeln ("folga",i,": ", getsol(folga(i)))
fclose (F_OUTPUT)
!
end-model
```

4.3. Implementação do Simulated Annealing

Primeiramente será definido um limitante inferior e um limitante superior (LB_j, UB_j) para cada variável de decisão x_j. O limitante inferior será sempre zero (restrição de não-negatividade) e o limitante superior será o maior valor que x_j pode assumir para atender a uma demanda.

Para ficar mais claro, tome a matriz $[a_{ij}] \in \mathrm{M}_{3x8}(\Re)$ da Tabela 1 e gere a matriz $\left[\frac{dem_i}{a_{ij}}\right]$ para $a_{ij} \neq 0$, cujos elementos serão zero se $a_{ij} = 0$.

$$\left[\frac{dem_i}{a_{ij}}\right] = \begin{bmatrix} 50 & 100 & 100 & 100 & 0 & 0 & 0 & 0 \\ 0 & 100 & 200 & 0 & 66{,}7 & 100 & 200 & 0 \\ 300 & 0 & 300 & 150 & 0 & 300 & 100 & 75 \end{bmatrix}, \text{ onde } [dem_i] = \begin{bmatrix} 100 \\ 200 \\ 300 \end{bmatrix}$$

Assim, $[UB_j] = \begin{bmatrix} 300 & 100 & 300 & 150 & 67 & 300 & 200 & 75 \end{bmatrix}$ que é o máximo valor inteiro que x_j pode assumir para atender a alguma demanda de $[dem_i]$.

Agora, seguem algumas definições que serão utilizadas logo a seguir.

$x^* =$ solução ótima.

$\hat{x} =$ solução vizinha a x.

$t_f =$ temperatura final de resfriamento.

$trial =$ número de tentativas de solução para cada temperatura t.

$\alpha =$ taxa de resfriamento.

De posse disto, segue o pseudo-algoritmo para o processo de Simulated Annealing utilizado neste projeto:

$x = x_0; \quad t = t_0; \quad f = f(x_0); \quad x^* = x;$
while $(t > t_f)$
  for $i = 1 : trial$
    $\widehat{x}$ ← Geração da solução vizinha a x
    $\Delta f = f(\widehat{x}) - f(x)$
    if $(\Delta f < 0) \quad x = \widehat{x}$
      else
        if $(rand(0,1) < exp(\frac{-\Delta f}{t})) \quad x = \widehat{x}$
          else $x = x$
        end
    end
    if $(f(\widehat{x}) < f(x^*)) \quad x^* = \widehat{x}$
      else $x^* = x^*$
    end
  end
  t $= \alpha.t$
end

Faltou apenas explicar como são geradas as soluções vizinhas $\hat{x}$. Segue abaixo a metodologia adotada no artigo [1], referência para este projeto.

1. Seja uma solução factível $x = \{x_1, x_2, ..., x_j, ..., x_k, ..., x_n\}$, onde *n* é a quantidade de padrões de corte possíveis.
2. Escolha aleatoriamente dois elementos de x (x_j e x_k).
3. Faça receberem um valor aleatório dentro do intervalo $x_j \in [0, UB_j]$ e $x_k \in [0, UB_k]$.
4. Se a nova solução for factível, chame este vetor de $\hat{x}$.
5. Se a nova solução não for factível, faça x_k voltar ao seu valor antigo e descubra o novo intervalo $[LB_j, UB_j]$ que x_j pode assumir de forma a garantir a factibilidade da solução.
6. Faça x_j receber um valor aleatório dentro do novo intervalo $[LB_j, UB_j]$.
7. Se a nova solução for igual a x, repita o processo desde o passo 2. Caso contrário, chame a nova solução de $\hat{x}$.

Logo abaixo encontra-se o algoritmo desenvolvido em Matlab para a resolução de Problemas de Corte Unidimensionais, como descrito até agora. Lembrando que o vetor x_{relax} é a solução do problema relaxado sem custo de setup e integralizado para o próximo maior inteiro, encontrada no *xpress*.

```
L = 500;
l = [190 188 164 158 146 144 139 135 125 114 111 108 76 74 64 61 52 12];
dem = [10 9 9 6 6 15 8 20 15 4 18 7 14 16 22 6 3 12];
%
[a,desp] = classes(L,l);

[m,n] = size(a);
%Solucao inicial gerada pelo xpress desconsiderando custos de setup
for j = 1 : n
   for i = 1 : m
      if (a(i,j) == 0)
         limx(i,j) = 0;
      else
         limx(i,j) = dem(i)/a(i,j);
      end
   end
   UBx(j)=ceil(max(limx(:,j)));
end
%
mp = 10;
```

```
M = 10;
setup = 100;
%
t0 = 10000;
tf = 0.001;
trial = 100;
alpha = 0.95;
%
%Solucao inicial gerada pelo xpress desconsiderando custos de setup
xrelax = [0 0 0 0 0 0 0 0 0 13 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 15 0 0 ];
for j = 1 : n
   x(j)=ceil(xrelax(j));
end
x
for i = 1:n
   if ( x(i) == 0 )
      y(i) = 0;
   else
      y(i) = 1;
   end
end
%
t = t0;
folga = a*x' - dem';
xtrial = x;
xbest = x;
ybest = y;
f = mp*desp*x' + setup*sum(y) + M*sum(folga)
fbest = f;
ftrial = f;
%
while (t > tf)
   for repetir = 1 : trial
         [xviz,yviz,folgviz] = vizinho (a,x,dem,UBx);
         fviz = mp*desp*xviz' + setup*sum(yviz) + M*sum(folgviz);
         delta = fviz - f;
         if (delta < 0)
            x = xviz;
            f = fviz;
         else
            k = rand(1);
            P = exp(-delta/(t));
            if (k <= P)
               x = xviz;
               f = fviz;
            end
         end
         if (fviz < ftrial)
           xtrial = xviz;
           ftrial = fviz;
         end
   end
   x = xtrial;
   f = ftrial;
   if (ftrial < fbest)
         xbest = xtrial;
         fbest = ftrial;
   end
   t = alpha*t;
```

```
end
    if (fviz < fbest)
        xbest = xviz;
        fbest = fviz;
    end
xbest
fbest
```

- Código computacional da *function classes*:

```
function [a,desp] = classes (L,l)
%
[a,desp]=gerador(L,l);
%
[y,ordem] = sort(desp);
%
[m,n] = size(a);
%
for i = 1:n
   aordem(:,i) = a(:,ordem(i));
end
%
a = aordem(:,1:3*m);
desp = y(1:3*m);
```

- Código computacional da *function vizinho*:

```
%Algoritmo para geracao de solucoes factiveis vizinhas para SA
function [xviz,yviz,folgviz] = vizinho (A,x,dem,UBx)
[m,n] = size (A);
%
xviz = x;
while (xviz == x)
   aux = x;
   rj = ceil(rand(1)*n);
   r2 = ceil(rand(1)*n);
   while ( r2 == rj )
      r2 = ceil(rand(1)*n);
   end
   rk = r2;
   %
   aux(rj) = round(rand(1)*UBx(rj));
   aux(rk) = round(rand(1)*UBx(rk));
   %
   folgax = A*aux' - dem';
   %
   if (min(folgax) >= 0)
      xviz = aux;
      folgviz = folgax;
   else
      aux(rk) = x(rk);
      folgax = A*aux' - dem';
      for i = 1 : m
         if (A(i,rj) == 0)
            varx(i) = 0;
         else
            varx(i) = floor(folgax(i)/A(i,rj));
         end
```

```
      end
      varrj = - min(varx);
      LBrj = varrj + aux(rj);
      aux(rj) = floor(rand(1)*(UBx(rj) - LBrj) + LBrj);
      xviz = aux;
      folgviz = A*aux' - dem';
   end
end
%
for i = 1 : n
   if ( xviz(i) == 0 )
      yviz(i) = 0;
   else
      yviz(i) = 1;
   end
end
```

Perceba que no algoritmo da *function classes* limitou-se mais uma vez o tamanho da matriz $[a_{ij}]$ para *3*m* colunas, onde *m* é a quantidade de itens finais demandados. Assim, dos 20.000 padrões gerados na *function gerador*, trabalhou-se apenas com os primeiro *3*m* padrões, cuja ordenação baseava-se no desperdício de matéria-prima, do menor para o maior.

Lembre que na definição da solução vizinha $\hat{x}$ escolhe-se apenas dois elementos de x para receberem novos valores. Quando o vetor x tem 20.000 posições (quantidade de padrões de corte), alterar dois elementos não era suficiente para sair do mínimo local fornecido pela solução do problema relaxado sem considerar custos de setup. Assim, foi tomada a decisão de se trabalhar somente com os melhores *3*m* padrões de corte.

Mesmo limitado *3*m* padrões, o algoritmo de geração de soluções vizinhas sugerido no artigo [1] não promoveu uma perturbação suficiente para escapar dos mínimos locais. Assim, foi criado um novo algoritmo para geração de vizinhos, a *function vizinho2*, cujo código computacional encontra-se abaixo.

```
Algoritmo para geracao de solucoes factiveis vizinhas para SA
function [xviz,fviz] = vizinho2 (A,c,x,dem,UBx,mp,setup,M)
%
[m,n]=size(A);
%
original = x;
indice = find (x ~= 0);
qtde = length(indice);
k = ceil(rand(1)*qtde);
```

```
kaux = k;
while(kaux == k)
   kaux = ceil(rand(1)*n);
end
%
for pulo = 1 : 4
if (pulo > 2)
   alter = kaux + pulo - 2;
   if (alter == n + 1)
      alter = 1;
   elseif (alter == n + 2)
      alter = 2;
   end
else
   alter = kaux - pulo;
   if (alter == 0)
      alter = n;
   elseif (alter == -1)
      alter = n - 1;
   end
end
   x(indice(k))=0;
   x(alter) = 0;
   folga0 = A*x' - dem';
   if (min(folga0) < 0)
       for i = 1 : m
          if (A(i,alter) == 0)
             inter(i) = 0;
          else
             inter(i) = ceil(folga0(i)/A(i,alter));
          end
       end
       varalter = - min(inter);
       x(alter) = round(rand(1)*(UBx(alter)-varalter) + varalter);
   else
       x(alter) = round(rand(1)*UBx(alter));
   end
   folgax = A*x' - dem';
   if (min(folgax) < 0)
       fteste(pulo) = Inf;
   else
       fteste(pulo) = mp*c*x' + setup*qtde + M*sum(folgax);
   end
   for j = 1 : n
       testex(pulo,j) = x(j);
   end
   x = original;
end
[fordem,index] = sort(fteste);
if (fordem(1) == Inf)
   for pulo = 1 : 4
       testex(pulo,indice(k)) = original(indice(k));
       folgax = A*testex(pulo,:)' - dem';
       fteste(pulo) = mp*c*testex(pulo,:)' + setup*(qtde+1) + M*sum(folgax);
   end
   [fordem,index] = sort(fteste);
   fviz = fordem(1);
   xviz = testex(index(1),:);
else
```

```
    fviz = fordem(1);
    xviz = testex(index(1),:);
end
```

Segue abaixo a metodologia da *function vizinho2*.

1. Seja uma solução factível $x = \{x_1, x_2, ..., x_j, ..., x_k, ..., x_n\}$, onde *n* é a quantidade de padrões de corte possíveis.
2. Escolha aleatoriamente dois elementos de x (x_j e x_k).
3. Faça $x_k = 0$ e gere valores aleatórios para os elementos $x_{j-2}, x_{j-1}, x_{j+1}, x_{j+2}$ dentro de seus respectivos intervalo $x_p \in [0, UB_p]$
4. Calcule o valor da função objetivo para os quatro vetores gerados e chame de $\hat{x}$ o vetor factível de menor valor.
5. Porém, se nenhum dos quatro vetores for factível, faça x_k voltar ao seu valor antigo.
6. Calcule novamente o valor da função objetivo para os quatro vetores gerados e chame de $\hat{x}$ o vetor de menor valor. Neste caso, o número de elementos não nulos de x será aumentado em uma unidade.

O programa principal da metodologia de Simulated Annealing previamente exposto sofreu somente a alteração abaixo para se adaptar à nova *function vizinho2*. O restante permaneceu o mesmo.

```
while (t > tf)
    for repetir = 1 : trial
        [xviz,fviz] = vizinho2 (a,desp,x,dem,UBx,mp,setup,M);
        delta = fviz - f;
```

A resolução dos problemas de Corte Unidimensional feita dessa forma começou a fornecer resultados satisfatórios, encontrando o valor da solução exata em vários casos. Os resultados que serão mostrados posteriormente neste projeto utilizaram a *function vizinho2* e as alterações no corpo do programa principal descritas logo acima.

5. Geração de Exemplos de Problemas de Corte

Uma vez implementado o modelo para resolução de Problemas de Corte Unidimensionais baseado em Simulated Annealing, precisava-se de exemplos para posterior análise dos resultados obtidos. Esses exemplos foram obtidos através da metodologia proposta no artigo [3] *CUTGEN1: A problem generator for the Standard One-dimensional Cutting Stock Problem*.

A idéia do artigo era gerar Classes de problemas de acordo com os critérios tamanho dos itens finais e demanda. Assim, uma classe poderia ser definida, por exemplo, como somente itens pequenos com demandas grandes, ou itens heterogêneos (pequenos, médios e grandes) com demandas médias. Neste projeto trabalhamos com cinco Classes diferentes e foram gerados cinco problemas para cada classe.

Para definição de cada Classe utilizou-se as variáveis definidas abaixo:

$m =$ quantidade de itens finais produzidos.

$L =$ tamanho da matriz de corte.

$l_i =$ largura do item *i*, onde $i = 1,2,...,m$.

$v_1 =$ limite inferior da relação l_i / L.

$v_2 =$ limite superior da relação l_i / L.

$\bar{d} =$ demanda média para cada item.

As cinco Classes definidas para este projeto foram:

Classe	v1	v2	m	d	itens	demanda
1	2%	40%	10	10	⇩	⇩
2	2%	40%	10	100	⇩	⇧
3	2%	40%	20	10	⇩	⇩
4	2%	40%	20	100	⇩	⇧
5	2%	40%	40	10	⇳	⇩

6. Resultados obtidos

Cada problema resolvido pela metodologia de Simulated Annealing foi comparado com a solução exata do PLI considerando os custos de setup. Esta solução foi obtida utilizando-se o software *Xpress*, cujo algoritmo encontra-se abaixo.

```
model "Encontra a solucao do problema exato"
uses "mmxprs"; !gain access to the Xpress-Optimizer solver
!
parameters
    mlim = 38
    nlim = 3*mlim
    DATAFILE = "adesp.dat"
    OUTFILE = "result.dat"
end-parameters
!
declarations
m = 1..mlim
n = 1..nlim
x: array (n) of mpvar
folga: array (m) of mpvar
y: array (n) of mpvar
a: array (m,n) of integer
desp: array (n) of integer
d: array (m) of integer
end-declarations
!
initializations from "adesp.dat"
a desp d
end-initializations
!
G:= 28
!
forall (i in m) sum(j in n) a(i,j)*x(j) = d(i) + folga(i)
!
forall (j in n) x(j) - G*y(j) <=0;
!
forall (j in n) x(j)>=0
forall (j in n) y(j) is_binary
forall (j in n) x(j) is_integer
!
forall (i in m) folga(i)>=0

!
desperdicio:= sum(j in n) desp(j)*x(j)
setup:= sum (j in n) y(j)
folgas:= sum ( i in m) folga(i)
minimize (10*desperdicio + 100*setup + 10*folgas)
!
writeln ("Desperdicio: ", getobjval)
write ("x: [ ")
forall (j in n) write (getsol(x(j))," ")
writeln ("]")
```

```
forall (i in m) writeln ("folga",i,": ", getsol(folga(i)))
!
fopen ("result", F_OUTPUT)
writeln ("Desperdicio: ", getobjval)
write ("x: [ ")
forall (j in n) write (getsol(x(j))," ")
writeln ("]")
forall (i in m) writeln ("folga",i,": ", getsol(folga(i)))
fclose (F_OUTPUT)
!
end-model
```

Os parâmetros utilizados para resolução dos problemas foram $\alpha = 0.95$, $trial = 100$ e $t_0 = 10.000$. A seguir serão mostrado alguns resultados por Classe de problemas.

6.1 Resultados Obtidos com as Classe 1 e 2

Os problemas gerados para as Classes 1 e 2 são iguais e o que muda é a demanda por item final produzido. Por isso, foram tratadas conjuntamente. As tabelas abaixo irão fornecer o valor da solução inicial ($x_{inicial}$) ao se resolver o PPL relaxado, o valor da melhor solução encontrada no Método de Simulated Annealing (x_{best}) e o valor da solução exata do problema (x_{otimo}).

Para cada solução, a primeira linha corresponde à posição do elemento no vetor e a segunda linha o valor do elemento. O valor da função objetivo foi representada por f e $nnulos$ é a quantidade de elementos não-nulos. O código à esquerda da tabela fornece a Classe e qual o número do problema resolvido.

C1 P1	xinicial	6	10	12	22	23	24	29	f	nnulos
		11,0	**1,4**	**1,5**	**8,5**	**3,0**	**0,9**	**0,6**	**1180**	**7**
	xbest	6	10	22	25				f	nnulos
		17	**2**	**10**	**3**				**900**	**4**
	xotimo	6	10	22	25				f	nnulos
		17	**2**	**10**	**3**				**900**	**4**

C1 P1: encontrou a solução ótima. Perceba que saiu do mínimo local gerado pela solução inicial e encontrou outra configuração de vetor e quantidade de elementos não-nulos.

C2 P1	xinicial	6	10	12	22	23	24	29	f	nnulos
		108,0	**15.25**	**14,4**	**85,0**	**30,0**	**8.25**	**5.75**	**4470**	**7**
	xbest	6	10	12	22	23	24	29	f	nnulos
		108	**15**	**15**	**85**	**30**	**9**	**6**	**4420**	**7**
	xotimo	6	10	22	23	25			f	nnulos
		108	**21**	**85**	**30**	**25**			**4300**	**5**

C2 P1: mesmos dados de P1, porém com demandas diferentes. Não saiu do mínimo local, mas melhorou o valor da função objetivo.

C1 P2	xinicial	2	3	5	15	23	25	26	30	f	nnulos
		2,3	**1,7**	**9,0**	**2,9**	**1,3**	**0,1**	**0,9**	**1,7**	**1020**	**8**
	xbest		3	4	5	15			30	f	nnulos
			2	**2**	**10**	**6**			**2**	**640**	**5**
	xbest	2			5	15			30	f	nnulos
		4			**12**	**4**			**4**	**640**	**4**
	xotimo	2	3	5	15				30	f	nnulos
		2	**2**	**10**	**5**				**3**	**630**	**5**

C1 P2: saiu do mínimo local, mas não chegou ao valor da solução exata. Porém, encontrou duas soluções muito próximas.

C2 P2	xinicial	2	3	5	15	23	26	30	f	nnulos
		18,9	**20,1**	**89,5**	**29,6**	**12,9**	**15,5**	**12,1**	**1220**	**7**
	xbest	2	3	5	15	23	26	30	f	nnulos
		19	**20**	**90**	**30**	**13**	**15**	**12**	**1050**	**7**
	xotimo	2	3	5	15	23	26	30	f	nnulos
		19	**20**	**90**	**30**	**13**	**15**	**12**	**1050**	**7**

C1 P2: não se pode comprovar a eficiência do Método neste caso, pois a solução exata pertencia ao mínimo local da solução inicial.

6.2 Resultados Obtidos com as Classe 3 e 4

Os problemas gerados para as Classes 3 e 4 são iguais e o que muda é a demanda por item final produzido. Por isso, foram tratadas conjuntamente. As considerações feitas no item 6.1 valem aqui também.

C3 P1														
	xinicial	4	5	8	20	21	25	27	28	46	47		f	nnulos
		9,0	**9,0**	**6,0**	**15,0**	**8,0**	**12,0**	**3,0**	**9,0**	**2,0**	**5,0**		**2380**	**10**
	xbest	4	5	8	20	21	25		28		47		f	nnulos
		9	**9**	**7**	**15**	**8**	**12**		**15**		**7**		**2220**	**8**
	xotimo	4	5	8	20	21	25		28		47		f	nnulos
		9	**9**	**7**	**15**	**8**	**12**		**15**		**7**		**2220**	**8**

C3 P1: a metodologia de SA encontrou a solução ótima.

C4 P1														
	xinicial	1	4	5	8	20	21	25	27	28	46	47	f	nnulos
		11,5	**93,0**	**92,0**	**64,0**	**153,0**	**83,0**	**117,0**	**22,8**	**102,5**	**13,0**	**48,5**	**15460**	**11**
	xbest	1	4	5	8	20	21	25		28	33	47	f	nnulos
		11	**93**	**92**	**64**	**153**	**83**	**117**		**148**	**13**	**49**	**15280**	**10**
	xotimo	1	4	5	8	20	21	25		28	33	47	f	nnulos
		11	**93**	**92**	**64**	**153**	**83**	**117**		**148**	**13**	**49**	**15280**	**10**

C4 P1: a metodologia de SA encontrou a solução ótima, porém esta estava muito próxima ao mínimo local gerado pela solução inicial.

6.3 Resultados Obtidos com a Classe 5

C5 P1																										
	xinicial	1	14	15	18	19	23	25	29	42	43	45	46	50	61	65	86	89	90	108	109	110	111	112	113	114	f	nnulos
		6	**3**	**1**	**1**	**1**	**15**	**4**	**4**	**2**	**18**	**6**	**11**	**9**	**13**	**6**	**6**	**2**	**2**	**8**	**4**	**4**	**2**	**13**	**16**	**3**	**5740**	**25**
	xbest	1	14	15	18	19	23	25	29	42	43		46	50	61	65	86	89		108	109	110	111	112	113	114	f	nnulos
		7	**2**	**1**	**1**	**1**	**16**	**4**	**4**	**2**	**18**		**13**	**9**	**12**	**7**	**6**	**8**		**8**	**3**	**4**	**2**	**13**	**16**	**3**	**5310**	**23**
	xotimo		14			19	20	23	29	42				50	61			89	90	108	109	110	111	112	113	114	f	nnulos
			10			**1**	**18**	**4**	**10**	**2**				**9**	**19**			**8**	**14**	**8**	**15**	**4**	**3**	**13**	**18**	**3**	**5090**	**17**

C5 P1: a metodologia de SA não encontrou a solução ótima, mas saiu do mínimo local e melhorou a função objetivo.

7. Conclusões

Com os exemplos mostrados neste projeto pode-se perceber que a solução exata do problema inteiro pode ser bastante diferente da solução do problema relaxado e integralizado para o maior inteiro mais próximo. Por isso, o algoritmo de geração de soluções vizinhas do Método de Simulated Annealig precisa ser capaz de gerar soluções que escapem do mínimo local fornecido pela solução inicial do problema relaxado.

A metodologia proposta no artigo [1] se mostrou ineficiente para escapar dos mínimos locais e a nova proposta da *function vizinho2* mostrou melhores resultados. Para as Classes 1 e 3 onde as demandas eram pequena, o algoritmo se mostrou eficaz, já para as Classes 2 e 4 nem sempre conseguia encontrar a solução exata dos problemas. Para a Classe 5 encontrava alguma solução intermediária entre a ótima e a inicial. Assim, conclui-se que esta *function* precisa ainda ser aprimorada de forma a sempre encontrar o ótimo.

Especificamente para os exemplos mostrados neste projeto, as soluções ótimas utilizaram *p* padrões de corte, onde $p < m$. Logo, a idéia do artigo [1] de se gerar todos os padrões possíveis (pode chegar a milhões), para depois se selecionar *m* (ou *3m* se for o caso) não parece ser uma alternativa inteligente de solucionar estes tipos de problema. Independente do algoritmo de geração de vizinhos utilizado, a metodologia de Simulated Annealing proposta no artigo [1] só foi possível de ser realizada limitando-se a geração de padrões em 20 mil tipos e depois escolhendo os *3*m* melhores.

A limitação dos padrões de corte não garante que a solução ótima seja aquela encontrada pelo *Xpress*. Mas, serviu como padrão de comparação para verificar a eficiência da Metodologia de Simulated Annealing. Logo, a proposta aqui demonstrada precisaria fazer parte de um algoritmo maior que também encontrasse o conjunto de padrões ótimos.

8. Bibliografia

[1] Chuen-Lung S. Chen, Stephen M. Hart, Wai Mui Tham, “A Simulated Annealing heuristic for the one-dimensional cutting stock problem”, European Journal of Operational Research, 93 (1996) 522-535.

[2] Saa M.A. Suliman, “Pattern generating procedure for the cutting stock problem”, International journal of production economics, 74 (2001) 293-301.

[3] T. Gau, G. Wäscher, “CUTGEN1: A problem generator for the Standard One-dimensional Cutting Stock Problem”, European Journal of Operational Research, 84 (1995) 572-579.