UNIVERSIDADE FEDERAL DO PARANÁ

GUSTAVO VALENTE NUNES

RESOLUÇÃO DE UM PROBLEMA DE ESCALONAMENTO DE MÁQUINAS USANDO PROGRAMAÇÃO LINEAR

CURITIBA - PR
28/12/2020

Contents

1 INTRODUÇÃO

O problema do escalonamento de tarefas entregue pelo orientador, consiste em achar o menor custo para uma quantidade de tarefas que uma empresa precisa executar. Cada tarefa vai ter uma quantidade de horas necessária para ser completada, e essa mesma tarefa pode ser dividia entre as máquinas de forma a gerar um menor custo no final. Cada máquina alugada (Máquinas próprias não possuem custo) vai ter um custo de operação por hora e uma quantidade máxima de horas que essa máquina pode ficar em operação. Essa máquina pode realizar quantas tarefas forem necessárias, considerando que, deve se respeitar o tempo máximo em horas que a máquina pode ficar em execução.

1.1 LIMITAÇÕES E DETALHES

- O custo mínimo é calculado sobre a quantidade de horas que as tarefas irão ficar em execução nas máquinas.
- Cada tarefa possui um tempo total que precisa ser executada para ser concluída.
- As máquinas possuem um tempo máximo que podem ficar em execução.
- Todas as máquinas alugadas (Máquinas próprias não possuem custo) possuem um custo por hora, para serem executadas.
- As máquinas podem executar uma ou mais tarefas, considerando que o tempo máximo que a máquina pode ficar ligado, não seja ultrapassado.

1.2 VÁRIAVEIS DO PROBLEMA

Neste capítulo é apresentado todas as variáveis que serão necessárias para fazer a modelagem do problema posteriormente.

- Váriaveis principais:
  - n = Quantidade de tarefas.
  - l = Quantidade de máquinas.
- Váriaveis referente as tarefas($1 \leq i \leq n$):
  - h_i = Hora da tarefa i.
- Váriaveis referente as máquinas($1 \leq i \leq l$):
  - c_i = Custo da máquina i.
  - u_i = Tempo máximo de uso da máquina i.
  - Caso $c_i = 0$, Significa que a máquina é alugada.

- Váriaveis referentes as tarefas e máquinas ($1 \leq$ i $\leq$ n e $1 \leq$ j $\leq$ l):
  - s_i = Número de tarefas que a máquina i vai realizar.
  - $t_{i,j}$ = Tarefa j que está sendo atribuído a máquina i.

2 MODELAGEM DO PROBLEMA

O objetivo do problema, e gerar o menor custo possível após todas as tarefas serem realizadas, e para poder calcular o custo mínimo precisa-se de um valor x em horas que cada tarefa vai ser executada em cada máquina. Pensando nisso podemos dizer então, que o custo mínimo vai ser, o custo da máquina c_i vezes o tempo que a tarefa vai ficar em execução nessa máquina $x_{t_{i,j}}$, sendo $t_{i,j}$ a tarefa atribuida a máquina m_i.

Generalizando agora, considerando todas as máquinas e todas as tarefas. O custo mínimo seria:

$$\text{min: } \sum_{i=1}^{l}(\sum_{j=1}^{n} c_i * x_{t_{i,j}})$$

Agora que sabemos a função do custo mínimo, precisamos identificar as restrições necessárias para o problema. Para fazer isso, o escalonamento de máquinas foi considerado como se fosse uma matriz n, l, onde l é a quantidade de máquinas e n é a quantidade de colunas.

Como cada tarefa é uma coluna, então a soma da coluna tem que ser igual a quantidade total de horas da tarefa. O mesmo racícinio vale para as máquinas, como cada linha da matriz equivale à uma máquina, então podemos dizer que, a soma da linha precisa ser menor igual a quantidade máxima que a máquina pode ficar em execução. Essas duas restrições precisam estar funcionando ao mesmo tempo para o problema dar certo. Elas ficariam da seguinte forma, respectivamente:

1. $\sum_{j=1}^{n} x_{t_{j,i}} = h_i$ para $1 \leq$ i $\leq$ l
2. $\sum_{j=1}^{l} x_{t_{i,j}} \leq u_i$ para $1 \leq$ i $\leq$ n

Detalhes:

- Na primeira função foi deixado de propósito a troca do i pelo j, para representar a soma das colunas.
- Foi considerado que a matriz por padrão tem todos seus valores zerados. E conforme as tarefas vão sendo atríbuidas seus valores vão mudando para um.

Sobre os limites dos valores de x, todos eles são valores maiores iguais a zero ($x \geq 0$), desconsiderando apenas os casos em que o valor de x for zero na matriz.

Modelagem Final:

$$\text{min: } \sum_{i=1}^{l}(\sum_{j=1}^{n} c_i * x_{t_{i,j}})$$
$$\text{s.t: } \sum_{j=1}^{n} x_{t_{j,i}} = h_i \text{ para } 1 \leq \text{i} \leq \text{l}$$
$$\text{s.t: } \sum_{j=1}^{l} x_{t_{i,j}} \leq u_i \text{ para } 1 \leq \text{i} \leq \text{n}$$
$$\text{lim: } x_{i,j} \geq 0 \text{ para todo } x_{i,j} \text{ com tarefa atribuida}$$
$$\text{lim: } x_{i,j} = 0 \text{ sem tarefa atribuida}$$

2.1 EXEMPLO PRÁTICO

Nesta seção, vai ser resolvido o problema do escalonamento de máquinas utilizando o exemplo proposto pelo orientador.

Entrada:
n = 2; l = 2
$h_1 = 10$
$h_2 = 10$
$c_1 = 100$; $u_1 = 20$
$c_1 = 50$; $u_1 = 10$
$s_1 = 2$
$t_{1,1} = 1$
$t_{1,2} = 2$
$s_2 = 1$
$t_{2,1} = 1$
Saída:
0.0 10.0
10.0 0.0
1500.0

Como foi dito nos capítulos anteriores, a ideia para resolver o problema, é construir uma matriz a partir dos dois primeiros valores de entrada. Ficaria assim: $M_{l,n} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ Pegando os valores $t_{i,j}$ da entrada, e atualizando os valores na matriz, ficaria da seguinte forma: $M_{l,n} = \begin{bmatrix} t_{1,1} & t_{1,2} \\ t_{2,1} & 0 \end{bmatrix}$ — $M_{l,n} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ Com isso conseguimos definir, quais serão os limites da função e quais valores de x terão valor igual a zero. Agora que a matriz está definida, podemos usar calcular o valor mínimo e utilizar as restrições.

Obs: Como já falado nos capitulos anteriores, o x representa a quantidade que cada máquina vai realizar a tarefa em horas.

Minímo:
$\sum_{i=1}^{l}(\sum_{j=1}^{n} c_i * x_{t_{i,j}})$

$$c_1 * x_{1,1} + c_1 * x_{1,2} + c_2 * x_{2,1} + c_2 * x_{2,2}$$

restrição 1:
$\sum_{j=1}^{n} x_{t_{j,i}} = h_i$ para 1 ≤ i ≤ l

1. $M_{1,1} * x_{1,1} + M_{2,1} * x_{2,1} = h_1$
2. $M_{1,2} * x_{1,2} + M_{2,2} * x_{2,2} = h_2$

restrição 2:
$\sum_{j=1}^{l} x_{t_{i,j}} \leq u_i$ para 1 ≤ i ≤ n

1. $M_{1,1} * x_{1,1} + M_{1,2} * x_{1,2} \leq u_1$
2. $M_{2,1} * x_{2,1} + M_{2,2} * x_{2,2} \leq u_2$

Limites:

$$x_{1,1}, x_{1,2}, x_{2,1} \geq 0$$
$$x_{2,2} = 0$$

Resolvendo as inequações, colocando os melhores valores, resultaria no seguinte:

Minímo:
$\sum_{i=1}^{l}(\sum_{j=1}^{n} c_i * x_{t_{i,j}})$

$$100 * 0 + 100 * 10 + 50 * 10 + 50 * 0 = 1500$$

restrição 1:
$\sum_{j=1}^{n} x_{t_{j,i}} = h_i$ para 1 ≤ i ≤ l

1) $1 * 0 + 1 * 10 = 10$
2) $1 * 10 + 0 * 0 = 10$

restrição 2:
$\sum_{j=1}^{l} x_{t_{i,j}} \leq u_i$ para 1 ≤ i ≤ n

1) $1 * 0 + 1 * 10 \leq 20$
2) $1 * 10 + 0 * 0 \leq 10$

Limites:

$$x_{1,1} \geq 0;\ x_{1,2} \geq 0;\ x_{2,1} \geq 0;\ x_{2,2} = 0$$

3 IMPLEMENTAÇÃO

3.1 FERRAMENTAS UTILIZADAS

O problema de escalonamento de máquinas é resolvido utilizando técnicas de programação linear. No problema do escalonamento, a melhor solução é aquela que se consegue resolver todas as tarefas, gerando o custo mínimo. O algoritimo utilizado para resolver o problema, é o simplex. Foi utilizado duas ferramentes para isso o python3 junto com a bilbioteca scipy e linprog, e a outra foi a IDE lpsolve.

3.1.1 USANDO O LPSOLVE

Como o lpsolve é usado para casos "estáticos", no sentindo de não poder automatizar a entrada, organização e a saída de dados. Foi utilizado apenas o caso do exemplo 1 fornecido pelo professor.

```
/* Objective function */
min: 100 x_1  + 100 x_2 + 50 x_3 + 50 x_4;

/* Variable bounds */
x_1 + x_3 = h_1;
x_2  + x_4  = h_2;
x_1 + x_2 <= u_1;
x_3 + x_4 <= u_2;


/* Limites */
x_1 >= 0;
x_2 >= 0;
x_3 >= 0;
x_4 = 0;
u_1 = 20;
u_2 = 10;
h_1 = 10;
h_2 = 10;
```

Imagem 1: Modelagem do problema 1 no LPsolve

```
Model name:  'LPSolver' - run #1
Objective:   Minimize(R0)

SUBMITTED
Model size:        4 constraints,       8 variables,          12 non-zeros.
Sets:                                   0 GUB,                 0 SOS.

Using DUAL simplex for phase 1 and PRIMAL simplex for phase 2.
The primal and dual simplex pricing strategy set to 'Devex'.


Optimal solution               1500 after          2 iter.

Relative numeric accuracy ||*|| = 0

 MEMO: lp_solve version 5.5.2.9 for 32 bit OS, with 64 bit REAL variables.
      In the total iteration count 2, 0 (0.0%) were bound flips.
      There were 0 refactorizations, 0 triggered by time and 0 by density.
       ... on average 2.0 major pivots per refactorization.
      The largest [LUSOL v2.2.1.0] fact(B) had 5 NZ entries, 1.0x largest basis.
      The constraint matrix inf-norm is 1, with a dynamic range of 1.
      Time to load data was 0.002 seconds, presolve used 0.004 seconds,
       ... 0.009 seconds in simplex solver, in total 0.015 seconds.
```

Imagem 2: Resultado do problema 1 no lpsolve

Com isso, podemos concluir que a modelagem do problema apresentada no capítulo anterior funciona.

3.1.2 UTILIZANDO O PYTHON3

Ó código feito para resolver a modelagem foi baseado na documentação da bilbioteca scipy e linprog do python3. Exemplo citado na documentação a seguir:

$$\begin{aligned} \min_{x_0, x_1} \quad & -x_0 + 4x_1 \\ \text{such that} \quad & -3x_0 + x_1 \leq 6, \\ & -x_0 - 2x_1 \geq -4, \\ & x_1 \geq -3. \end{aligned}$$

Imagem 3: Exemplo de modelagem scipy python3

```
>>> c = [-1, 4]                                                              >>>
>>> A = [[-3, 1], [1, 2]]
>>> b = [6, 4]
>>> x0_bounds = (None, None)
>>> x1_bounds = (-3, None)
>>> from scipy.optimize import linprog
>>> res = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds])
```

Imagem 4: Exemplo de resolução da modelagem scipy python3

Notações Importantes sobre o exemplo acima e sobre o lógica utilizada para desenvolvido no código:

- eq = Significa igualdade. Ex: a = b
- eb = Significa menor igual. Ex: $a \leq b$
- A = [[-1, 1], [1, 2]]. ((-1, -2) virou (1, 2) porque foi multiplicado por -1)
- b = [6, 4] é o lado direito da modelagem. (Mudou de -4 para 4, porque foi multiplicado por -1)
- c = [-1, 4] é o mínimo da função.
- x1_bounds(0,0) = Indica os limites da modelagem. Ex $x_1 = 0$
- x2_bounds(0,None) = Indica os limites da modelagem. Ex $x_2 \geq 0$

Agora que os pontos importantes foram citados, agora é possível explicar a linha de raciocínio e como o código foi desenvolvido.

A maior parte do código é organizar os dados, para poder utilizar eles na função que vai resolver o problema. A função "leInputs", apenas lê o input de dados de um arquivo e salva toda a entrada como se fosse um vetor de dados, no vetor inputs.

```python
def leInputs(inputs):
    #Abre o arquivo
    with open("./inputs/input2.txt", "r") as file:

        # Lê linha por linha
        for line in file:

            #salva os inputs em um vetor chamado inputs
            for word in line.split():
                inputs.append(int(word))

    return inputs
```

Imagem 5: Lê o input de dados de um arquivo

Nesta função "leTarefas", a função lê do vetor inputs os dados e salva apenas as tarefas que precisam ser realizadas. Que no caso seria, a quantidade de horas dessas tarefas.

```python
def leTarefas(inputs, tarefas):
    #inputs[0] indica a quantidade de tarefas
    quantidadeTarefas = inputs[0]
    i = 2
    k = 0
    # Aloca os valores das tarefas no dicionário tarefas
    while (quantidadeTarefas > 0):
        tarefas['H'+str(k)] = str(inputs[i])
        i += 1
        k += 1
        quantidadeTarefas -= 1

    return tarefas
```

Imagem 6: Lê as tarefas em um dicionário de tarefas

Nesta função "leMaquinas", a função lê do vetor "inputs" os dados das máquinas. Que no caso seria, o custo por hora e o tempo máximo de execução. Salva os dois valores dentro de um dicionário no estilo: ["Ci", CUSTO, "Ui", TEMPOMÁX].

```python
# Arruma os custos e tempo máximo de execução da máquina em um dicinário
def leMaquinas(inputs, maquinas):
    # inputs[1] indica a quantidade de máquinas
    quantidadeMaquinas = inputs[1]

    #inputs[0] indica a quantidade de tarefas
    quantidadeTarefas = inputs[0]

    i = 2 + quantidadeTarefas
    j = 0
    while (quantidadeMaquinas > 0):
        # Custo
        maquinas['C'+str(j)] = str(inputs[i])
        i += 1
        #Tempo máximo de cpu
        maquinas['U'+str(j)] = str(inputs[i])
        i += 1

        j += 1
        quantidadeMaquinas -= 1


    #retorna um dicinário máquinas
    return maquinas
```

Imagem 7: Lê os custos e tempo máximo e salva em um dicinário máquinas

Agora que os dados estão organizados, é possível utilizar o raciocínio da modelagem e criar uma matriz, para atribuir todas as tarefas para suas respectivas máquinas. Sendo uma matriz $M_{l,n}$ onde l é a quantidade de linhas (máquinas) e n é a quantidade de colunas (tarefas). Como a matriz é criada com todos seus valores zerados, podemos dizer que uma tarefa foi atribuída a uma máquina, quando seu valor é mudado de zero para um.

Ex. Tarefa 2 atríbuida a máquina 1:

$$M_{l,n} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ — } M_{l,n} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

A função “arrumandoMaquinas” faz o processo descrito acima e a função “min” descrita na imagem 8 cria a função de menor custo para o problema do escalonamento de máquinas.

```python
# Indica quais tarefas são atríbuidas as máquinas
def arrumandoMaquinas(inputs, matriZerada):
    # Indíce que indica início da atribuição de tarefas Ti,j
    i = 2 + inputs[0] + (inputs[1] * 2)
    k = 0
    while(i < len(inputs)):
        j = 0
        aux = inputs[i]
        while(j < aux):
            i += 1
            indice = inputs[i]
            matriZerada[k][indice-1] = 1 # Se a tarefa foi atríbuida, indica o valor de 1
            j += 1

        k += 1
        i += 1

    # retorna a matriz, com os valores de 0 ou 1 nela
    return matriZerada

# Indica os custos que vai precisar ser calculado, ex: 100 * x1
def min(matrizAtribuitiva, c, maquinas):
    for i in range(len(matrizAtribuitiva)):
        for j in range(len(matrizAtribuitiva[i])):
            c.append(int(maquinas['C'+str(i)]))


    return c
```

imagem 8: Arruma a matriz e a função mínima.

Às duas imagens abaixo, criam os vetores A, b, c citados nos detalhes acima.

```python
#Indica a 2° restrição da modelagem, em que a soma da coluna precisa ser igua a tarefa
def xizesEQ(matrizAtribuitiva, maquinas, tarefas, a, inputs):
    for i in range(inputs[0]):
        k = 0
        for j in range(inputs[1]):
            a[i][j+k+i] = matrizAtribuitiva[j][i]
            k += 1


    return a

#Aqui, indica um vetor de tarefas, aonde o xizesEQ, precisa ser igual aos valores atribuidos
#nesta função
def restricoesEQ(tarefas, maquinas, B, inputs):
    for i in range(inputs[0]):
        B.append(int(tarefas['H'+str(i)]))


    return B
```

imagem 9: Cria os vetores de igualdade.

```python
#Indica vetores que precisam ser menores iguais ao tempo máximo de execução da máquina
def xizesUB(matrizAtribuitiva, maquinas, tarefas, a, inputs):
    k = 0
    for i in range(inputs[1]):
        for j in range(inputs[0]):
            a[i][j+k] = matrizAtribuitiva[i][j]
        k += inputs[0]

    return a



#Indica um vetor de tempo de execução, que vai ser combinado com "xizesUB"
def restricoesUB(tarefas, maquinas, B, inputs):
    for i in range(len(maquinas)//2):
        B.append(int(maquinas['U'+str(i)]))


    return B
```

imagem 10: Cria os vetores de menores iguais.

3.2 USANDO O SIMPLEX:

Agora que os dados foram organizados, as restrições e limites arrumados. Podemos calcular o o melhor caso para o problema.

```python
# c retorna um vetor do custo mínimo
c = []
c = min(matrizAtribuitiva, c, maquinas)

# A_eq indica um vetor que precisa ser igual a outra coisa. Ex a = b.
# Porém, o A_eq precisa ser igual a quantidade máxima de tarefas
A_eq = []
colunas = inputs[0]*inputs[1]
linhas  = inputs[0]
for i in range(linhas):
    A_eq.append( [0] * colunas)

# A_ub indica um vetor que precisa ser menor igual a outra coisa. Ex a <= b.
# Porém, A_ub precisa ser menor igual a quantidade máxima que uma máquina pode ficar em execução.
A_ub = []
colunas = inputs[0]*inputs[1]
linhas  = ((inputs[0]*inputs[1]) // 2)
for i in range(linhas):
    A_ub.append( [0] * colunas)

A_ub = xizesUB(matrizAtribuitiva, maquinas, tarefas, A_ub, inputs)
A_eq = xizesEQ(matrizAtribuitiva, maquinas, tarefas, A_eq, inputs)

# Os valores do vetor A_ub precisam ser menores iguais que b_ub. Ex: A_ub[x,y] ; b_ub[k,l]
#Então, x <= k e y <= k
b_ub = []
b_ub = restricoesUB(tarefas, maquinas, b_ub, inputs)

# Os valores do vetor A_eq precisam ser iguais que b_eq. Ex: A_ub[x,y] ; b_ub[k,l]
#Então, x = k e y = k
b_eq = []
b_eq = restricoesEQ(tarefas, maquinas, b_eq, inputs)

# Os bounds são os limites da modelagem.
bounds = []
bounds = limites(matrizAtribuitiva, bounds)

res = linprog(c,
              A_ub=A_ub,
              b_ub=b_ub,
              A_eq=A_eq,
              b_eq=b_eq,
              bounds=bounds
              )
```

imagem 11: Coloca os resultados na função do scipy.

A imagem acima, segue os mesmo padrões descritos na documentação do scipy e linprog já descrito no começo do capítulo. Após arrumar as entradas para estilo que a função precisa, basta colocar tudo na função “linprog” que a função vai devolver o resultado esperado.