



Manual de uso da biblioteca Pyomo para Programação Matemática

Claudemir Woche V. Carvalho*
Anselmo R. Pitombeira Neto†

OPL - Pesquisa Operacional em Produção e Logística

UFC - Universidade Federal do Ceará

2020

*Matemática Industrial, Universidade Federal do Ceará

†Departamento de Engenharia de Produção, Universidade Federal do Ceará

Conteúdo

1	Visão geral do Pyomo	4
2	Instalação de componentes necessários	4
2.1	Anaconda	4
2.2	Python	4
2.3	Pyomo	4
2.4	Installando um <i>solver</i> livre	5
3	Tutorial de Python	6
3.1	Por que Python?	6
3.2	Variáveis	6
3.3	Números e Operações	6
3.4	Strings	6
3.5	Listas	7
3.6	Tuplas	7
3.7	Dicionários	7
3.8	Compressão de Listas	8
3.9	Estruturas de Controle	8
3.10	Indentação	9
3.11	Funções	9
3.12	Módulos	9
3.13	Leitura de arquivos	10
4	Tutorial de Pyomo	11
4.1	Modelo Abstrato e Modelo Concreto	11
4.2	Conjuntos	11
4.3	Parâmetros	12
4.4	Variáveis	13
4.5	Funções Objetivo	14
4.6	Restrições	14
5	O Problema da Mochila	16
5.1	Descrição do Problema	16
5.2	Exemplo Simples	16
5.2.1	Explicando o código	17
5.3	Exemplo Genérico	19
5.3.1	Explicando o código	20
6	O Problema de Transporte	23
6.1	Descrição do Problema	23
6.2	Exemplo Simples	23
6.2.1	Explicando o Código	25
7	O Problema de Alocação	28
7.1	Descrição do Problema	28
7.2	Exemplo Genérico	29
7.2.1	Explicando o Código	30

8	O Problema do Caixeiro Viajante	34
8.1	Descrição do Problema	34
8.2	Exemplo Genérico	35
8.2.1	Explicando o Código	36
9	O Problema de Localização de Facilidades Capacitadas	41
9.1	Descrição do Problema	41
9.2	Exemplo Genérico	42
9.2.1	Explicando o Código	44

1 Visão geral do Pyomo

Pyomo é uma biblioteca para a modelagem de problemas de otimização baseada em Python que tem como objetivos:

- Fornecer uma sintaxe natural para a descrição de modelos matemáticos.
- Formular modelo grandes com uma sintaxe concisa.
- Separar declaração de modelo e dados.

Pyomo não é um *solver* de programação matemática. Trata-se de uma *linguagem de modelagem*, ou seja, em Pyomo é possível definir um problema de programação matemática e então este é convertido para as estruturas de dados de entrada esperadas por algum dos solvers suportados. Em Pyomo é possível utilizar diferentes solvers, tais como IBM CPLEX, Gurobi, GLPK, CBC, dentre outros. Um modelo escrito em Pyomo é convertido para um dos formatos esperados pelo solver que se deseja utilizar. Dessa forma, um mesmo modelo pode ser resolvido por diferentes solvers com pouca ou nenhuma modificação.

2 Instalação de componentes necessários

Para utilizar o Pyomo, é necessário ter uma instalação do Python e uma instalação de algum *solver* de programação matemática. O Pyomo não inclui um solver.

2.1 Anaconda

Baixe e instale o Anaconda para o Python 3.7(ou mais recente) para o seu sistema operacional em <https://www.anaconda.com/download/>

2.2 Python

Com o Anaconda instalado, o Python já estará disponível. Teste a instalação do Python da seguinte forma:

- No Windows: Procure pelo "Anaconda Prompt" e nele digite `python --version`.
- No Linux ou MAC: Digite `python --version` no terminal.

Uma mensagem semelhante a `Python 3.6.8 :: Anaconda, Inc.` deverá ser exibida.

2.3 Pyomo

No Anaconda Prompt(Windows) ou no terminal(Linux ou MAC) digite `conda install -c conda-forge pyomo`. Teste a instalação digitando `pyomo --version`, algo parecido com: `Pyomo 5.6.6 (CPython 3.6.8 on Linux 4.15.0-45-generic)` deverá ser exibido.

2.4 Installando um *solver* livre

Dentre alguns solvers que podem ser instalados livremente, sem a necessidade da compra de uma licença, incluem-se o glpk e CBC. Para instalar por exemplo o glpk, pode-se fazer o seguinte: no Anaconda Prompt(Windows) ou no terminal(Linux ou MAC) digite `conda install -c conda-forge glpk`. Teste a instalação digitando `glpsol --version`, algo parecido com: `GLPSOL: GLPK LP/MIP Solver, v4.65` deverá aparecer.

3 Tutorial de Python

3.1 Por que Python?

- Linguagem interpretada;
- sintaxe intuitiva;
- orientada a objetos;
- simples e extremamente poderosa;
- grande quantidade de bibliotecas disponíveis;
- fácil aprendizado e com muitos recursos;
- integração com C e Java.

3.2 Variáveis

As variáveis no python não precisam ser declaradas, somente inicializadas.

```
1 a = 35
2 b = 'python'
3 type(a)      #int
4 type(b)      #str
```

A linguagem atribuiu os tipos certos às variáveis sem uma declaração explícita.

3.3 Números e Operações

```
1 x = 5
2 y = 3
3 x + y      #8          #SOMA
4 x - y      #2          #SUBTRACAO
5 x * y      #15         #PRODUTO
6 x / y      #1.6667     #DIVISAO
7 x ** y     #125        #POTENCIACAO
8 x // y     #1          #DIVISAO INTEIRA
9 x % y      #2          #RESTO DA DIVISAO
```

As operações também podem ser efetuadas para números não-inteiros.

3.4 Strings

```
1 string1 = 'Hello'
2 string2 = 'World'
3
4 string1 + string2      # 'HelloWorld'      - Concatenacao
5 string1 * 3            # 'HelloHelloHello' - Repeticao
6 string1[0]             # 'H'          - Indexacao
7 string1[-1]           # 'o'          - Ultimo item
8 string1[1:4]          # 'ell'        - Particao
9 len(string1)          # 5            - Tamanho
10 'l' in string1       # True          - Busca
```

3.5 Listas

Os vetores no Python são chamados de listas, as quais são estruturas heterogêneas que podem receber tipos diferentes de dados. No exemplo abaixo, a mesma lista temos um número inteiro, uma *string*, e outra lista com *strings*.

```
1 a = [99 , 'Pyomo', ['modelagem', 'matematica']]
2
3 a[0] #99
4 a[1] #Pyomo
5 a[2] #['modelagem', 'matematica']
6 a[2][1] # 'matematica'
7 del a[-1] #Deleta o ultimo item
```

As listas têm os mesmos operadores que as strings.

```
1 c = list(range(5)) #[0,1,2,3,4] - Cria uma lista de 0 a
5-1
2 c.append(5) #[0,1,2,3,4,5] - Insere o 5 no final da
lista
3 c.pop() #[0,1,2,3,4] - Retira o ultimo
elemento da lista
4 c.insert(3, 42) #[0,1,2,42,3,4] - Insere o 42 na posicao
3
5 c.pop(2) #[0,1,42,3,4] - Retira o elemento do
indice 2 da lista
6 c.reverse() #[4,3,42,1,0] - Inverte a lista
7 c.sort() #[0,1,2,3,4,42] - Ordena a lista
```

3.6 Tuplas

A principal diferença entre as tuplas e as listas é que as tuplas são imutáveis, ou seja, uma vez definidas, não podem ter seus valores alterados.

```
1 tupla = ('primeiro', 'segundo')
2 x,y = tupla
3 x #'primeiro'
4 y #'segundo'
5 tupla[0] = 3 #Erro, tupla nao pode ser alterada
```

3.7 Dicionários

As chaves(índices) dos dicionários são imutáveis, ou seja, não podem ser alteradas depois da criação. Listas ou outros dicionários não podem ser usados como keys(mas podem como values).

```
1 dicionario = {'item1': 30, 'item2':33.56, 'item3': 'item'}
2
3 dicionario[0] #Erro, o dicionario nao possui esse
indice
4 dicionario['item1'] #30
5 dicionario['item2'] #33.56
6 dicionario['item3'] #'item'
7
8 del dicionario['item2'] #{'item1': 30,'item3': 'item'}
9 dicionario['item3'] = 18.11 #{'item1': 30, item3': 18.11}
```

```

10 dicionario['novo_item'] = 'item4'    #{'item1': 30, 'item3': 18.11, '
    novo_item': 'item4'}
11
12 dicionario.keys()                    #['item1', 'item3', 'novo_item']
13 dicionario.values()                  #[30, 18.11, 'item4']

```

3.8 Compressão de Listas

No Python é possível criar listas de forma muito prática:

```

1 lista = [i for i in range(100)]      #Cria uma lista com numeros de 0 a
    99
2
3 # [0,1,2,...,99]
4
5 pares = [i for i in range(100) if i%2 == 0]    #Cria uma lista com os
    numeros pares de 0 a 99
6
7 # [0,2,4,6,...,98]
8
9 tuplas = [(i,j) for i in range(3) for j in range (5)] #Cria uma lista
    com tuplas (i,j)
10
11 # [(0,0), (0,1), (0,2), ..., (2,3), (2,4)]
12
13 soma = sum([i*j for i in range(3) for j in range (5)]) #Soma os
    elementos da lista com o produto de cada tupla
14
15 # 30

```

3.9 Estruturas de Controle

If-else-elif

```

1 a,b = 10,8
2
3 if a%2 == 0:
4     print(a, 'eh par')
5 else:
6     print(a, 'eh impar')
7
8 if b == 0:
9     print(b, 'eh 0')
10 elif b>0:
11     print(b, 'eh maior que zero')
12 else:
13     print(b, 'eh menor que zero')

```

while

```

1 while c < d:    #Enquanto a condicao for verdadeira, o laço sera
    executado
2     print(c)
3     c+=1

```

for

```

1 vetor = list(range(10))
2

```



```

3 for i in vetor:                                #A variavel i ira iterar no vetor
4     print(i)

```

3.10 Indentação

Python não utiliza chaves para delimitar blocos de código, ele faz isso através da indentação:

```

1 for i in range(20):                            #i de 0 a 19
2     if i % 3 == 0:
3         print(i)
4         if i % 5 == 0:
5             print('Bingo')
6             print('—')

```

3.11 Funções

```

1 def func(x):
2     if x <= 100:
3         return x**2
4
5 a = func(5)
6 a                                #25
7
8 def fibonnaci(n):                    #Calcula o n-esimo numero da sequencia de
9     if n==0 or n==1:                fibonnaci
10        return n
11    return fibonnaci(n-1) + fibonnaci(n-2)
12
13 f = fibonnaci(9)
14 f                                #34
15
16 def mdc(a, b):
17     while a != 0:
18         a, b = b%a, a
19     return b
20
21 m = mdc(120,145)                    #Retorna o maximo divisor comum entre
22 m                                  120 e 3
23 m                                  #5

```

3.12 Módulos

```

1 import numpy as np                        #Importa a biblioteca numpy e da um
2     apelido a ela
3
4 from numpy.random import randint         #Da biblioteca numpy.random importa so
5     o metodo randint
6
7 array1 = np.array([1,2,3])
8 array2 = np.array([3,4,5])
9 prod = np.dot(array1, array2)
10 prod                                    #Produto interno entre os dois vetores
11
12 aleatorio = randint(3,100)              #Retorna um numero inteiro aleatorio
13     entre 3 e 100

```

3.13 Leitura de arquivos

```
1 file = open('arq.txt')      #O arquivo precisa estar na mesma pasta do
   codigo
2 linhas = file.readlines()
3 linhas      #Lista onde cada elemento representa uma
   linha do arquivo
4
5 for linha in linhas:
6     print(linha[:-1])      #Printa as linhas sem o '\n'
```

4 Tutorial de Pyomo

4.1 Modelo Abstrato e Modelo Concreto

Um modelo de programação matemática pode ser definido usando símbolos que representam valores. Por exemplo, as equações a seguir representam um modelo de programação linear (na forma matricial) com vetor de variáveis de decisão \mathbf{x} e os parâmetros do modelo são dados pelos vetores \mathbf{b} e \mathbf{c} , e pela matriz \mathbf{A} :

$$\begin{aligned} \max \quad & \mathbf{z} = \mathbf{c}^t \mathbf{x} \\ \text{s.a.} \quad & \\ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Chamamos isso de modelo matemático *abstrato* ou *simbólico*, uma vez que se baseia em valores de parâmetros não especificados. Os valores dos dados podem ser usados para especificar uma *instância do modelo*. A classe `AbstractModel` fornece um contexto para definir e inicializar modelos de otimização abstrata no Pyomo quando os valores dos dados serão fornecidos no momento em que uma solução deve ser obtida.

Um modelo abstrato é declarado da seguinte forma:

```
1 import pyomo.environ as pyEnv
2
3 model = pyEnv.AbstractModel()
```

Em muitos contextos, um modelo matemático pode e deve ser definido diretamente com os valores dos dados fornecidos no momento da definição dele. Chamamos esses de modelos matemáticos *concretos*. Por exemplo, o seguinte modelo é uma instância concreta do modelo abstrato anterior:

$$\begin{aligned} \max \quad & z = 2x_1 + 3x_2 \\ \text{s.a.} \quad & \\ & 3x_1 + 4x_2 \leq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

A classe `ConcreteModel` é usada para definir modelos concretos de otimização no Pyomo:

```
1 model = pyEnv.ConcreteModel()
```

OBS: Nesse manual trabalharemos somente com a classe `ConcreteModel`.

4.2 Conjuntos

Conjuntos podem ser declarados usando as funções `Set` e `RangeSet`. Na [documentação](#) do Pyomo é possível checar os diversos parâmetros possíveis da função `Set`. Um deles é o `within`, o qual especifica o domínio dos valores. Exemplos de domínios que podem ser usados:

- Any - Qualquer valor possível.
- Reals - Reais.

- `PositiveReals` - Reais positivos.
- `NonNegativeReals` - Reais não negativos.
- `Integers` - Inteiros.
- `PositiveIntegers` - Inteiros positivos.
- `NonPositiveIntegers` - Inteiros não positivos.
- `NonNegativeIntegers` - Inteiros não negativos.
- `Binary` e `Boolean` - Valores booleanos: `False/True`, `0/1`.

A função `RangeSet` fornece um mecanismo para criar conjuntos com sequências simples. Essa função recebe como argumento um valor inicial, valor final e o tamanho do passo. Se só 1 argumento for fornecido, então o valor fornecido será o valor final na sequência; o primeiro valor e o tamanho do passo serão 1. Se dois valores forem fornecidos, eles serão o primeiro e último valor na sequência com o tamanho do passo padrão de 1. Por exemplo, o código a seguir cria um conjunto(`Set`) com os números 1.5, 5 e 8.5.

```
1 model.M = pyEnv.RangeSet(1.5, 10, 3.5)
```

Para mais informações consulte a [documentação](#).

4.3 Parâmetros

A palavra “parâmetro” é usada em muitos contextos. Ao discutir um modelo Pyomo, usamos a palavra para se referir a dados que devem ser fornecidos para encontrar uma atribuição ótima (ou boa) de valores para as variáveis de decisão. Os parâmetros são declarados com a função `Param`, que aceita argumentos que são um pouco semelhantes à função `Set`. Por exemplo, o seguinte código declara os conjuntos `model.A`, `model.B` e, em seguida, um parâmetro `model.P` que é indexado por `model.A` e `model.B`, e tem um tamanho de 5x3:

```
1 model.A = pyEnv.RangeSet(5)
2 model.B = pyEnv.RangeSet(3)
3 model.P = pyEnv.Param(model.A, model.B)
```

A função `Param` recebe alguns argumentos, os principais são:

- **`initialize`**: Uma função (ou objeto Python) que retorna os dados para inicializar os valores dos parâmetros.
- **`within`**: `Set` usado para validação; especifica o domínio dos valores do parâmetros.

Existem muitas maneiras de criar um parâmetro que seja uma matriz quadrada com 9, 16, 25 na diagonal principal e 0 nos outros lugares. Uma maneira de fazer é a seguinte: utilizando uma função de inicialização que é chamada uma vez para cada tupla de índices.

```

1 model.A = pyEnv.RangeSet(1,3)
2
3 def s_init(model, i, j):
4     if i == j:
5         return i*i
6     else:
7         return 0.0
8
9 model.S1 = pyEnv.Param(model.A, model.A, initialize=s_init)

```

`model.S1` será da seguinte forma:

```

In [4]: model.S1.display()
S1 : Size=9, Index=S1_index, Domain=Any, Default=None,
Mutable=False
Key   : Value
(1, 1) : 1
(1, 2) : 0.0
(1, 3) : 0.0
(2, 1) : 0.0
(2, 2) : 4
(2, 3) : 0.0
(3, 1) : 0.0
(3, 2) : 0.0
(3, 3) : 9

```

Nesse exemplo o conjunto de índices contém números inteiros, mas eles não precisam ser numéricos. É muito comum a utilização de strings.

4.4 Variáveis

As variáveis tem como objetivo receber valores de um pacote de otimização. Elas são declaradas e opcionalmente delimitadas, dadas valores iniciais e documentadas usando a função `Var` do Pyomo. Se conjuntos de índices são dados como argumentos para esta função, eles são usados para indexar a variável. Outras argumentos opcionais incluem:

- `bounds` - Uma função(ou objeto Python) que dá um par de limites (inferior, superior) para a variável.
- `domain` ou `within` - O conjunto de valores que a variável pode assumir.
- `initialize` - Uma função(ou objeto Python) que dá um valor inicial para a variável.

O código a seguir ilustra alguns aspectos dessas opções através da declaração de uma variável sem índices que poderá receber valores reais entre 0 e 6 e inicializada com 1.5:

```

1 model.Jack = pyEnv.Var(within=pyEnv.NonNegativeReals, bounds=(0,6),
   initialize=1.5)

```

Para variáveis indexadas, limites e valores iniciais são geralmente especificados por uma regra (uma função Python) que pode fazer referência a parâmetros ou outros

dados. Os argumentos formais para essas regras começam com o modelo, seguido pelos índices. Isso é ilustrado no seguinte trecho de código que utiliza dicionários Python declarados como `lb` e `ub`, usados por uma função para fornecer limites:

```
1 model.C = pyEnv.Set(initialize=['Scones', 'Tea'])
2 lb = {'Scones':2, 'Tea':4}
3 ub = {'Scones':5, 'Tea':7}
4
5 def fb(model, i):
6     return (lb[i], ub[i])
7
8 model.PriceToCharge = pyEnv.Var(model.C, domain=pyEnv.PositiveIntegers,
    bounds=fb)
```

4.5 Funções Objetivo

Uma função objetivo é uma função de variáveis que retorna um valor que um pacote de otimização tenta maximizar ou minimizar. A função `Objective` no Pyomo declara um objetivo. Embora outros mecanismos sejam possíveis, essa função geralmente recebe o nome de outra função que fornece a expressão. Abaixo está uma versão muito simples de uma função que assume que `model.x` foi declarado anteriormente como um `Var`:

```
1 def ObjRule(model):
2     return 2*model.x[1] + 3*model.x[2]
3
4 model.g = pyEnv.Objective(rule=ObjRule)
```

É mais comum que uma função objetivo faça referência a parâmetros, como neste exemplo, que pressupõe que `model.p` foi declarado como um `Param` e que `model.x` foi declarado com o mesmo conjunto de índices, enquanto `model.y` foi declarado como uma variável sem índice:

```
1 def profrul(model):
2     return pyEnv.summation(model.p, model.x) + model.y
3
4 model.Obj = pyEnv.Objective(rule=ObjRule, sense=pyEnv.maximize)
```

A função `summation` realiza a combinação linear entre os vetores `p` e `x`, ou seja, $p[1] * x[1] + p[2] * x[2] + \dots + p[n] * x[n]$. Nesse exemplo usamos o argumento `sense` para especificar maximização. O valor *default* do `sense` (ou seja, se ele não for especificado) é minimização.

4.6 Restrições

A maioria das restrições são especificadas usando expressões de igualdade ou desigualdade criadas usando uma regra, que é uma função Python. Por exemplo, se a variável `model.x` tiver os índices 'butter' e 'scones', então essa restrição limitará a soma desses índices a exatamente três:

```

1 def teaOKrule(model):
2     return(model.x['butter'] + model.x['scones'] == 3)
3
4 model.TeaConst = pyEnv.Constraint(rule=teaOKrule)

```

Em vez de expressões que envolvem igualdade (==) ou desigualdades (<= ou >=), as restrições também podem ser expressas usando uma tupla de três valores da forma (lb, expr, ub) em que lb e ub são o menor valor e o maior valor, respectivamente, que é interpretada como $lb \leq expr \leq ub$. O valor de lb e ub pode ser None, o qual significa que não há limitação. Variáveis podem aparecer apenas em expr. Por exemplo, as duas restrições a seguir são equivalentes:

```

1 model.x = pyEnv.Var()
2
3 def aRule(model):
4     return model.x >= 2
5
6 model.Bounds = pyEnv.Constraint(rule=aRule)
7
8 def bRule(model):
9     return (2, model.x, None)
10
11 model.boundx = pyEnv.Constraint(rule=bRule)

```

Para esse exemplo simples, também seria possível declarar model.x com a opção bounds que o resultado seria o mesmo.

Restrições(e funções objetivo) podem ser indexadas por listas ou conjuntos. Quando a declaração contém listas ou conjuntos como argumentos, os elementos são passados iterativamente para a rule function. Se houver mais de um, o produto cruzado será enviado. Por exemplo, a seguinte restrição pode ser interpretada como a colocação de um orçamento de i no i-ésimo item a ser comprado, onde o custo por item é fornecido pelo parâmetro model.a:

```

1 model.A = pyEnv.RangeSet(1,10)
2 model.a = pyEnv.Param(model.A, within=pyEnv.PositiveReals)
3 model.ToBuy = pyEnv.Var(model.A)
4
5 def bud_rule(model, i):
6     return model.a[i]*model.ToBuy[i] <= i
7
8 aBudget = pyEnv.Constraint(model.A, rule=bud_rule)

```

Python e Pyomo são *case sensitive*, ou seja, model.a não é o mesmo de model.A.

5 O Problema da Mochila

5.1 Descrição do Problema

O problema da mochila (Knapsack Problem) é um problema de Otimização Combinatória muito abordado na Pesquisa Operacional. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

O modelo de programação inteira do problema é o seguinte:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^m c_i x_i \\ & \text{sujeito a} && \sum_{i=1}^m p_i x_i \leq C \\ & && x_j \in \{0, 1\}, \quad j = 1, \dots, m \end{aligned}$$

Sendo \mathbf{c} o vetor de lucros, \mathbf{p} o vetor de pesos e \mathbf{C} a capacidade máxima da mochila.

5.2 Exemplo Simples

Usaremos a seguinte instância do problema:

Lucros: [92,57,49,68,60,43,67,84,87,72]

Pesos: [23,31,29,44,53,38,63,85,89,82]

Peso máximo suportado pela mochila: 165

Teremos a seguinte função objetivo:

$$\mathbf{Max} \quad 92x_1 + 57x_2 + 49x_3 + 68x_4 + 60x_5 + 43x_6 + 67x_7 + 84x_8 + 87x_9 + 72x_{10}$$

E a seguinte restrição:

$$23x_1 + 31x_2 + 29x_3 + 44x_4 + 53x_5 + 38x_6 + 63x_7 + 85x_8 + 89x_9 + 82x_{10} \leq 165$$

Segue na próxima página o código da modelagem e resolução do problema pelo Python utilizando o Pyomo.


```

1 import pyomo.environ as pyEnv
2 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
3
4 ##-----DADOS-----##
5 lucros = [92,57,49,68,60,43,67,84,87,72]
6 pesos = [23,31,29,44,53,38,63,85,89,82]
7 capacidade = 165
8 m = len(lucros)
9 ##-----DECLARACAO DO MODELO-----##
10 modelo = pyEnv.ConcreteModel()
11 modelo.I = range(m)
12
13 modelo.x = pyEnv.Var(modelo.I, within=pyEnv.Binary )
14
15 modelo.objetivo = pyEnv.Objective(expr = sum(lucros[i]*modelo.x[i] for
16     i in modelo.I)
17     ,sense = pyEnv.maximize)
18
19 modelo.rest = pyEnv.Constraint(expr = sum(pesos[i]*modelo.x[i] for i in
20     modelo.I) <= capacidade)
21
22 ##-----RESOLUCAO DO MODELO-----##
23 solver = pyEnv.SolverFactory('cplex',executable = path)
24 resultado_objetivo = solver.solve(modelo, tee=True)
25
26 ##-----PRINT DAS VARIAVEIS DE DECISAO-----##
27 lista = list(modelo.x.keys())
28 for i in lista:
29     print(i, '—', modelo.x[i]())
30
31 print('Valor objetivo = ',modelo.objetivo())

```

OBS: A variável path deve conter o caminho do arquivo executável do CPLEX ou outro solver como GLPK.

5.2.1 Explicando o código

Importamos o pyomo e o apelidamos como pyEnv, colocamos uma variável para receber o caminho do executável do solver.

```

1 import pyomo.environ as pyEnv
2 path = "/home/tito/cplex/bin/x86-64_linux/cplex"

```

Fornecemos os dados para o Python.

- **capacidade:** Variável com o peso máximo suportado pela mochila.
- **pesos:** Lista com os respectivos pesos de cada item.
- **lucros:** Lista com os respectivos lucros de cada item.
- **m:** variável com o número total de itens.

```

1 ##-----DADOS-----##
2 lucros = [92,57,49,68,60,43,67,84,87,72]
3 pesos = [23,31,29,44,53,38,63,85,89,82]
4 capacidade = 165
5 m = len(lucros)

```

Criamos o modelo.

```

1 modelo = pyEnv.ConcreteModel()

```

Criamos um índice $\mathbf{I}=(0,1,2,3,4,5,6,7,8,9)$ para as variáveis de decisão.

```

1 modelo.I = range(m)

```

Criamos \mathbf{m} variáveis de decisão binárias(só assumem os valores 0 ou 1) utilizando o índice \mathbf{I} .

```

1 modelo.x = pyEnv.Var(modelo.I, within=pyEnv.Binary )

```

Criamos a função objetivo e a restrição do problema utilizando os dados e as variáveis de decisão fornecidas ao modelo anteriormente.

```

1 modelo.objetivo = pyEnv.Objective(expr = sum(lucros[i]*modelo.x[i] for
2     i in modelo.I)
3     ,sense = pyEnv.maximize)
4 modelo.rest = pyEnv.Constraint(expr = sum(pesos[i]*modelo.x[i] for i in
5     modelo.I) <= capacidade)

```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk',executable=...)* para a utilização do solver GLPK em vez do CPLEX.

```

1 solver = pyEnv.SolverFactory('cplex',executable = path)

```

Resolvemos o modelo. O parâmetro **tee=True** indica que você quer ver na tela o log do solver, se não quiser basta fazer **tee=False**.

```

1 resultado_objetivo = solver.solve(modelo, tee=True)

```

Vemos os resultados da solução.

```

1 ##-----PRINT DAS VARIÁVEIS DE DECISÃO-----##
2 lista = list(modelo.x.keys())
3 for i in lista:
4     print(i, '—', modelo.x[i]())
5
6 print('Valor objetivo = ',modelo.objetivo())

```

5.3 Exemplo Genérico

```
1 #Problema da Mochila – Knapsack Problem
2 import pyomo.environ as pyEnv
3
4 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
5
6 ##-----LEITURA DA INSTANCIA-----##
7
8 filePesoMaximo = open('p01-c.txt')
9 pesoMaximo = filePesoMaximo.readline().split('\n')[0]
10 pesoMaximo = int(pesoMaximo)
11
12 filePesos = open('p01-w.txt')
13 pesos = filePesos.readlines()
14 pesos = pesos[0].split()
15 pesos = [int(i) for i in pesos]
16
17
18 fileLucros = open('p01-p.txt')
19 lucros = fileLucros.readlines()
20 lucros = lucros[0].split()
21 lucros = [int(i) for i in lucros]
22
23 m = len(pesos)
24
25 #-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
26 #Modelo
27 modelo = pyEnv.ConcreteModel()
28 #Indice para os pesos e precos
29 modelo.I = pyEnv.RangeSet(m)
30
31 #Variaveis de decisao xi
32 modelo.x = pyEnv.Var(modelo.I, within=pyEnv.Binary)
33
34 #Preco de cada produto
35 modelo.c = pyEnv.Param(modelo.I, initialize = lambda modelo,i: lucros[i
    -1])
36 #Peso de cada produto
37 modelo.p = pyEnv.Param(modelo.I, initialize = lambda modelo,i: pesos[i
    -1])
38
39 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICAO-----##
40 #Funcao objetivo do problema
41 modelo.objetivo = pyEnv.Objective(expr = sum(modelo.c[i]*modelo.x[i]
    for i in modelo.I), sense=pyEnv.maximize)
42
43 #Restricao do problema
44 modelo.restricao = pyEnv.Constraint(expr = sum(modelo.p[i] * modelo.x[i]
    ] for i in modelo.I) <= pesoMaximo)
45
46
47 ##-----RESOLUCAO DO MODELO-----##
48 solver = pyEnv.SolverFactory('cplex', executable=path)
49 resultado = solver.solve(modelo, tee= True)
50 print('\n-----\n')
51 modelo.pprint()
52 modelo.objetivo()
53 print(resultado)
54
```

```

55 ##-----PRINT DAS VARIÁVEIS DE DECISÃO-----##
56 lista = list(modelo.x.keys())
57 for i in lista:
58     print(i, '—', modelo.x[i]())

```

5.3.1 Explicando o código

Importamos o pyomo e o apelidamos como pyEnv, colocamos uma variável para receber o caminho do executável do solver.

```

1 #Problema da Mochila – Knapsack Problem
2 import pyomo.environ as pyEnv
3
4 path = "/home/tito/cplex/bin/x86-64_linux/cplex"

```

Efetuamos a leitura da instância. Nesse caso temos um arquivo para o peso máximo suportado pela mochila, um arquivo para os pesos de cada item e outro para os lucros de cada item.

- **pesoMaximo:** guardará o peso máximo suportado pela mochila.
- **pesos:** será uma lista com os respectivos pesos de cada item.
- **lucros:** será uma lista com os respectivos lucros de cada item.
- **m:** guardará o número total de itens.

```

1 ##-----LEITURA DA INSTANCIA-----##
2
3 filePesoMaximo = open('p01_c.txt')
4 pesoMaximo = filePesoMaximo.readline().split('\n')[0]
5 pesoMaximo = int(pesoMaximo)
6
7 filePesos = open('p01_w.txt')
8 pesos = filePesos.readlines()
9 pesos = pesos[0].split()
10 pesos = [int(i) for i in pesos]
11
12
13 fileLucros = open('p01_p.txt')
14 lucros = fileLucros.readlines()
15 lucros = lucros[0].split()
16 lucros = [int(i) for i in lucros]
17
18 m = len(pesos)

```

Criamos o modelo.

```

1 #-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
2 #Modelo
3 modelo = pyEnv.ConcreteModel()

```

Criamos um índice $I=(1,2,3,4,5,6,7,8,9,10)$ para os pesos e os preços(já que eles tem o mesmo tamanho). Diferente da função Range do Python que gera números de 0 a $m-1$, a função RangeSet do Pyomo gera números de 1 a m .

```
1 #Índice para os pesos e preços
2 modelo.I = pyEnv.RangeSet(m)
```

Criamos m variáveis de decisão binárias(só assumem os valores 0 ou 1) utilizando o índice I.

```
1 #Variaveis de decisao xi
2 modelo.x = pyEnv.Var(modelo.I, within=pyEnv.Binary)
```

Fornecemos ao modelo os parâmetros necessários. Peso e preço de cada item.

```
1 #Preco de cada produto
2 modelo.c = pyEnv.Param(modelo.I, initialize = lambda modelo,i: lucros[i
-1])
3 #Peso de cada produto
4 modelo.p = pyEnv.Param(modelo.I, initialize = lambda modelo,i: pesos[i
-1])
```

Fornecemos a função objetivo e a restrição do problema utilizando os parâmetros e as variáveis de decisão fornecidas ao modelo anteriormente.

```
1 #Funcao objetivo do problema
2 modelo.objetivo = pyEnv.Objective(expr = sum(modelo.c[i]*modelo.x[i]
for i in modelo.I), sense=pyEnv.maximize)
3
4 #Restricao do problema
5 modelo.restricao = pyEnv.Constraint(expr = sum(modelo.p[i] * modelo.x[i]
] for i in modelo.I) <= pesoMaximo)
```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk',executable=...)* para a utilização do solver GLPK em vez do CPLEX.

```
1 solver = pyEnv.SolverFactory('cplex', executable=path)
```

Resolvemos o modelo. O parâmetro **tee=True** indica que você quer ver na tela o log do solver, se não quiser basta fazer **tee=False**.

```
1 resultado = solver.solve(modelo, tee= True)
```

Vemos os resultados da solução.

```
1 modelo.pprint()
2 modelo.objetivo()
3 print(resultado)
```

Vemos os valores das variáveis de decisão.

```
1 lista = list(modelo.x.keys())
2 for i in lista:
3     print(i, '—', modelo.x[i]())
```

6 O Problema de Transporte

6.1 Descrição do Problema

O problema de transporte (Transportation Problem) consiste em um conjunto de m fábricas que produzem um produto, onde cada fábrica i pode fornecer no máximo s_i unidades. Um conjunto de n clientes para os quais o produto deve ser transportado, onde cada cliente j deve receber pelo menos d_j unidades do produto. Cada unidade produzida na fábrica i e transportada ao cliente j tem um custo de c_{ij} . O objetivo é minimizar os custos de transporte.

O modelo de programação linear do problema é o seguinte:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \\ \text{sujeito a} \quad & \sum_{j=1}^n x_{ij} \leq s_i, \quad i = 1, \dots, m \\ & \sum_{i=1}^m x_{ij} \geq d_j, \quad j = 1, \dots, n \\ & x_{ij} \geq 0 \end{aligned}$$

\mathbf{c} contém o custo de transporte da unidade da fábrica i para o cliente j , \mathbf{s} contém a capacidade máxima de cada fábrica e \mathbf{d} a demanda de cada cliente.

6.2 Exemplo Simples

Usaremos a seguinte instância do problema:

Capacidades: [1000,1500,1200]

Demandas: [2300,1400]

Custo de transporte da fabrica 1 para os clientes 1 e 2: [162,247]

Custo de transporte da fabrica 2 para os clientes 1 e 2: [117,193]

Custo de transporte da fabrica 3 para os clientes 1 e 2: [131,185]

Teremos a seguinte função objetivo:

$$\mathbf{Max} \quad 162x_{11} + 247x_{12} + 117x_{21} + 193x_{22} + 131x_{31} + 185x_{32}$$

E as seguintes restrições:

$$x_{11} + x_{12} \leq 1000$$

$$x_{21} + x_{22} \leq 1500$$

$$x_{31} + x_{32} \leq 1200$$

$$x_{11} + x_{21} + x_{31} \geq 2300$$

$$x_{12} + x_{22} + x_{32} \geq 1400$$

Segue na próxima página o código da modelagem e resolução do problema pelo Python utilizando o Pyomo.

```

1 import pyomo.environ as pyEnv
2 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
3
4 #Custo de transporte da fabrica i pro cliente j
5 cij = [[162,247],
6         [117,193],
7         [131,185]]
8
9 capacidades = [1000,1500,1200]
10 demandas = [2300,1400]
11
12 m = len(capacidades)
13 n = len(demandas)
14
15 ##-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
16 #Modelo
17 modelo = pyEnv.ConcreteModel()
18 #Indice para as fabricas
19 modelo.I = pyEnv.RangeSet(m)
20 #Indice para os clientes
21 modelo.J = pyEnv.RangeSet(n)
22
23 #Variaveis de decisao xij
24 modelo.x=pyEnv.Var(modelo.I,modelo.J, within=pyEnv.NonNegativeReals)
25
26 #Custo de transporte da fabrica i pro cliente j
27 modelo.c = pyEnv.Param(modelo.I, modelo.J, initialize=lambda modelo, i,
28                        j: cij[i-1][j-1])
29 #Capacidade de cada fabrica
30 modelo.b = pyEnv.Param(modelo.I, initialize = lambda modelo, i:
31                        capacidades[i-1])
32 #Demanda de cada cliente
33 modelo.d = pyEnv.Param(modelo.J, initialize = lambda modelo, j: demandas
34                        [j-1])
35
36 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##
37
38 def func_objetivo(modelo):
39     return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in
40               modelo.J)
41
42 modelo.objetivo = pyEnv.Objective(rule=func_objetivo, sense=pyEnv.
43 minimize)
44
45 ##-----##
46 #Cada fabrica nao exceder sua capacidade
47
48 def rule_rest1(modelo, i):
49     return sum(modelo.x[i,j] for j in modelo.J) <= modelo.b[i]
50
51 modelo.rest1 = pyEnv.Constraint(modelo.I, rule=rule_rest1)
52
53 ##-----##
54 #Cada cliente ter sua demanda atendida
55
56 def rule_rest2(modelo, j):
57     return sum(modelo.x[i,j] for i in modelo.I) >= modelo.d[j]

```



```

56
57 modelo.rest2 = pyEnv.Constraint(modelo.J, rule=rule_rest2)
58
59
60
61
62
63 ##-----RESOLUCAO DO MODELO-----##
64 solver = pyEnv.SolverFactory('cplex', executable=path)
65 resultado = solver.solve(modelo, tee = True)
66 print('\n-----\n')
67 #modelo.pprint()
68 modelo.objetivo()
69 print(resultado)
70
71 ##-----PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS-----##
72 ##(fabrica, cliente)
73 lista = list(modelo.x.keys())
74 for i in lista:
75     if modelo.x[i]() != 0:
76         print(i, '—', modelo.x[i]())

```

OBS: A variável path deve conter o caminho do arquivo executável do CPLEX ou outro solver como GLPK.

6.2.1 Explicando o Código

Importamos o pyomo e o apelidamos como pyEnv, colocamos uma variável para receber o caminho do executável do solver.

```

1 import pyomo.environ as pyEnv
2 path = "/home/tito/cplex/bin/x86-64_linux/cplex"

```

Fornecemos os dados para o Python.

- **capacidades:** Lista com a capacidade de cada fábrica i .
- **demandas:** Lista com a demanda de cada cliente j .
- **cij:** Matriz com o custo de transporte da unidade da fábrica i para o cliente j .
- **m:** variável com o número de fábricas.
- **n:** variável com o número de clientes.

```

1 #Custo de transporte da fabrica i pro cliente j
2 cij = [[162,247],
3         [117,193],
4         [131,185]]
5
6 capacidades = [1000,1500,1200]
7 demandas = [2300,1400]
8
9 m = len(capacidades)
10 n = len(demandas)

```

Criamos o modelo.

```

1 #Modelo
2 modelo = pyEnv.ConcreteModel()

```

Criamos um índice $\mathbf{I}=(1,\dots,m)$ para as fábricas.

```

1 #Índice para as fabricas
2 modelo.I = pyEnv.RangeSet(m)

```

Criamos um índice $\mathbf{J}=(1,\dots,n)$ para os clientes.

```

1 #Índice para os clientes
2 modelo.J = pyEnv.RangeSet(n)

```

Criamos $\mathbf{m}*\mathbf{n}$ variáveis de decisão utilizando os índices \mathbf{I} e \mathbf{J} .

```

1 #Variaveis de decisao xij
2 modelo.x=pyEnv.Var(modelo.I, modelo.J, within=pyEnv.NonNegativeReals)

```

Passamos a matriz \mathbf{cij} para o modelo usando os índices \mathbf{I} e \mathbf{J} criados anteriormente.

```

1 #Custo de transporte da fabrica i pro cliente j
2 modelo.c = pyEnv.Param(modelo.I, modelo.J, initialize=lambda modelo, i,
    j: cij[i-1][j-1])

```

Passamos as **capacidades** para o modelo usando o índice \mathbf{I} criado anteriormente.

```

1 #Capacidade de cada fabrica
2 modelo.b = pyEnv.Param(modelo.I, initialize = lambda modelo, i:
    capacidades[i-1])

```

Passamos as **demandas** para o modelo usando o índice \mathbf{J} criado anteriormente.

```

1 #Demanda de cada cliente
2 modelo.d = pyEnv.Param(modelo.J, initialize = lambda modelo, j: demandas
    [j-1])

```

Criamos a função objetivo utilizando os dados \mathbf{cij} e as variáveis \mathbf{x} fornecidas ao modelo anteriormente.

```

1 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##
2
3 def func_objetivo(modelo):
4     return sum(modelo.x[i, j] * modelo.c[i, j] for i in modelo.I for j in
    modelo.J)

```

```

5
6 modelo.objetivo = pyEnv.Objective(rule=func_objetivo, sense=pyEnv.
    minimize)

```

Criamos a primeira restrição que assegura que cada fabrica não exceda sua capacidade.

```

1 #Cada fabrica nao exceder sua capacidade
2
3 def rule_rest1(modelo, i):
4     return sum(modelo.x[i, j] for j in modelo.J) <= modelo.b[i]
5
6 modelo.rest1 = pyEnv.Constraint(modelo.I, rule=rule_rest1)

```

Criamos a segunda restrição que assegura que cada cliente tenha sua demanda atendida.

```

1 #Cada cliente ter sua demanda atendida
2
3 def rule_rest2(modelo, j):
4     return sum(modelo.x[i, j] for i in modelo.I) >= modelo.d[j]
5
6 modelo.rest2 = pyEnv.Constraint(modelo.J, rule=rule_rest2)

```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk', executable=...)* para a utilização do solver GLPK em vez do CPLEX.

```

1 solver = pyEnv.SolverFactory('cplex', executable=path)

```

Resolvemos o modelo. O parâmetro **tee=True** indica que você quer ver na tela o log do solver, se não quiser basta fazer **tee=False**.

```

1 resultado = solver.solve(modelo, tee = True)

```

Vemos os resultados da solução.

```

1 modelo.objetivo()
2 print(resultado)

```

Vemos quais as variáveis de decisão tiveram valores 1, ou seja, quais rotas (i,j) foram escolhidas.

```

1 ##-----PRINT DAS VARIÁVEIS DE DECISÃO ESCOLHIDAS-----##
2 ##(fabrica, cliente)
3 lista = list(modelo.x.keys())
4 for i in lista:
5     if modelo.x[i]() != 0:
6         print(i, '—', modelo.x[i]())

```

7 O Problema de Alocação

7.1 Descrição do Problema

O problema de alocação (Assignment Problem) consiste em alocar unidades de produção a tarefas a serem realizadas. Com custos de alocação de cada unidade para cada tarefa.

O modelo de programação linear do problema é o seguinte:

$$\begin{aligned} & \text{minimize} && \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \\ & \text{sujeito a} && \sum_{j=1}^n x_{ij} = 1, \quad i = 1, \dots, m \\ & && \sum_{i=1}^m x_{ij} = 1, \quad j = 1, \dots, n \\ & && x_{ij} \geq 0 \end{aligned}$$

c contém o custo de alocação da tarefa i para a unidade j .

Segue na próxima página o código da modelagem e resolução do problema pelo Python utilizando o Pyomo.

7.2 Exemplo Genérico

```
1 import pyomo.environ as pyEnv
2
3 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
4
5 #Alocar m tarefas [i] para n unidades[j]
6
7 ##-----LEITURA DA INSTANCIA-----##
8 file = open('assign300.txt')
9 linhas = file.readlines()
10 file.close()
11
12 cij = []
13 aux = []
14 aux2=[]
15
16 m=int(linhas[0].split('\n')[0])
17 linhas = linhas[1:]
18
19
20 for i in range(len(linhas)):
21     vet=linhas[i][: -1]
22     splitada = vet.split()
23     splitada = [int(i) for i in splitada if i!= '']
24     aux.append(splitada)
25
26 for sublist in aux:
27     for item in sublist:
28         aux2.append(item)
29
30 tam = len(aux2)
31 for i in range(m,tam+1,m):
32     cij.append(aux2[i-m:i])
33
34 n = len(cij[0])
35
36 ##-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
37
38 #Modelo
39 modelo = pyEnv.ConcreteModel()
40 #Indices para as tarefas a serem alocadas
41 modelo.I = pyEnv.RangeSet(n)
42 #Indices para unidades
43 modelo.J = pyEnv.RangeSet(m)
44
45 #Custo de alocao da tarefa i pra unidade j
46 modelo.c = pyEnv.Param(modelo.I,modelo.J, initialize=lambda modelo,i,j:
47     cij[i-1][j-1])
48
49 #Variaveis de decisao xij
50 modelo.x=pyEnv.Var(modelo.I,modelo.J, within=pyEnv.NonNegativeReals)
51
52 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##
53
54 def func_objetivo(modelo):
55     return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in
56     modelo.J)
```

```

57 modelo.objetivo = pyEnv.Objective(rule=func_objetivo , sense=pyEnv.
    minimize)
58
59
60
61
62
63 ##-----##
64 #Cada tarefa alocada para exatamente 1 unidade
65
66 def rule_rest1(modelo, i):
67     return sum(modelo.x[i, j] for j in modelo.J) == 1
68
69 modelo.rest1 = pyEnv.Constraint(modelo.I, rule=rule_rest1)
70
71 ##-----##
72 #Cada unidade realiza exatamente 1 tarefa
73
74 def rule_rest2(modelo, j):
75     return sum(modelo.x[i, j] for i in modelo.I) == 1
76
77 modelo.rest2 = pyEnv.Constraint(modelo.J, rule=rule_rest2)
78
79
80 ##-----RESOLUCAO DO MODELO-----##
81 solver = pyEnv.SolverFactory('cplex', executable=path)
82 resultado = solver.solve(modelo, tee = True)
83 print('\n-----\n')
84 #modelo.pprint()
85 modelo.objetivo()
86 print(resultado)
87 #-----PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS-----##
88 #(tarefa , unidade)
89 lista = list(modelo.x.keys())
90 for i in lista:
91     if modelo.x[i]() != 0:

```

OBS: A variável `path` deve conter o caminho do arquivo executável do CPLEX ou outro solver como GLPK.

7.2.1 Explicando o Código

Importamos o `pyomo` e o apelidamos como `pyEnv`, colocamos uma variável para receber o caminho do executável do solver.

```

1 import pyomo.environ as pyEnv
2
3 path = "/home/tito/cplex/bin/x86-64_linux/cplex"

```

Fornecemos os dados para o Python.

- **m:** quantidade de unidades.
- **n:** quantidade de tarefas.
- **cij:** matriz com o custo de alocar a tarefa i para a unidade j

```

1  ##-----LEITURA DA INSTANCIA-----##
2  file = open('assign300.txt')
3  linhas = file.readlines()
4  file.close()
5
6  cij = []
7  aux = []
8  aux2=[]
9
10 m=int(linhas[0].split('\n')[0])
11 linhas = linhas[1:]
12
13
14 for i in range(len(linhas)):
15     vet=linhas[i][: -1]
16     splitada = vet.split()
17     splitada = [int(i) for i in splitada if i!= '']
18     aux.append(splitada)
19
20 for sublist in aux:
21     for item in sublist:
22         aux2.append(item)
23
24 tam = len(aux2)
25 for i in range(m,tam+1,m):
26     cij.append(aux2[i-m:i])

```

OBS: O arquivo assign300.txt possui uma matriz 300x300 com os custos de alocação de 300 tarefas para 300 unidades

Criamos o modelo.

```

1 #Modelo
2 modelo = pyEnv.ConcreteModel()

```

Criamos um índice $\mathbf{I}=(1,\dots,n)$ para as tarefas a serem alocadas.

```

1 #Indices para as tarefas a serem alocadas
2 modelo.I = pyEnv.RangeSet(n)

```

Criamos um índice $\mathbf{J}=(1,\dots,m)$ para as unidades.

```

1 #Indices para unidades
2 modelo.J = pyEnv.RangeSet(m)

```

Passamos a matriz \mathbf{cij} para o modelo usando os índices \mathbf{I} e \mathbf{J} criados anteriormente.

```

1 #Custo de alocao da tarefa i pra unidade j
2 modelo.c = pyEnv.Param(modelo.I,modelo.J,initialize=lambda modelo,i,j:
    cij[i-1][j-1])

```

Criamos $m*n$ variáveis de decisão utilizando os índices **I** e **J**.

```
1 #Variaveis de decisao xij
2 modelo.x=pyEnv.Var(modelo.I, modelo.J, within=pyEnv.NonNegativeReals)
```

Criamos a função objetivo utilizando os dados **cij** e as variáveis **x** fornecidas ao modelo anteriormente.

```
1 def func_objetivo(modelo):
2     return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in
3         modelo.J)
4 modelo.objetivo = pyEnv.Objective(rule=func_objetivo, sense=pyEnv.
    minimize)
```

Criamos a primeira restrição que assegura que cada tarefa seja alocada para exatamente 1 unidade.

```
1 def rule_rest1(modelo, i):
2     return sum(modelo.x[i,j] for j in modelo.J) == 1
3
4 modelo.rest1 = pyEnv.Constraint(modelo.I, rule=rule_rest1)
```

Criamos a segunda restrição que assegura que cada unidade realiza exatamente 1 tarefa.

```
1 def rule_rest2(modelo, j):
2     return sum(modelo.x[i,j] for i in modelo.I) == 1
3
4 modelo.rest2 = pyEnv.Constraint(modelo.J, rule=rule_rest2)
```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk', executable=...)* para a utilização do solver GLPK em vez do CPLEX.

```
1 solver = pyEnv.SolverFactory('cplex', executable=path)
```

Resolvemos o modelo. O parâmetro **tee=True** indica que você quer ver na tela o log do solver, se não quiser basta fazer **tee=False**.

```
1 resultado = solver.solve(modelo, tee = True)
```

Vemos os resultados da solução.

```
1 modelo.objetivo()
2 print(resultado)
```

Vemos quais as variáveis de decisão tiveram valores 1, ou seja, quais tarefas foram alocadas para cada unidade.


```
1 #(tarefa , unidade)
2 lista = list(modelo.x.keys())
3 for i in lista:
4     if modelo.x[i]() != 0:
5         print(i, '—', modelo.x[i]())
```

8 O Problema do Caixeiro Viajante

8.1 Descrição do Problema

O Problema do Caixeiro Viajante (Travelling Salesman Problem) é um problema que tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas) e retornar à cidade de origem.

O modelo de programação inteira do problema é o seguinte:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^n \sum_{j \neq i, j=1}^n c_{ij} x_{ij} \\ \text{sujeito a} \quad & \sum_{i=1, i \neq j}^n x_{ij} = 1, & j = 1, \dots, n \\ & \sum_{j=1, j \neq i}^n x_{ij} = 1, & i = 1, \dots, n \\ & u_i - u_j + nx_{ij} \leq n - 1, & 2 \leq i \neq j \leq n \\ & 0 \leq u_i \leq n - 1, & 2 \leq i \leq n \\ & x_{ij} \in \{0, 1\} \\ & u_i \in \mathbb{Z} \end{aligned}$$

\mathbf{c} contém o custo de ir da cidade i para a cidade j , \mathbf{u} são variáveis auxiliares para a quebra de ciclos.

Segue na próxima página o código da modelagem e resolução do problema pelo Python utilizando o Pyomo.

8.2 Exemplo Genérico

```
1 import pyomo.environ as pyEnv
2 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
3
4
5 ##-----LEITURA DA INSTANCIA-----##
6 custoMatrix = []
7 aux = []
8 aux2 = []
9
10 file = open('53.txt')
11 linhas = file.readlines()
12 file.close()
13
14 for i in range(len(linhas)):
15     vet = linhas[i][: -1]
16     vet = vet.split('\t')
17     vet = [int(i) for i in vet if i != '']
18     aux2.append(vet)
19
20 for sublist in aux2:
21     for item in sublist:
22         aux.append(item)
23
24 tam = len(aux)
25 for i in range(53, tam+1, 53):
26     custoMatrix.append(aux[i-53:i])
27
28 custoMatrix = custoMatrix[: -1]
29 n = len(custoMatrix)
30
31 ##-----DECLARACAO DO MODELO E SEUS PARAMETROS
32 -----##
33 #Modelo
34 modelo = pyEnv.ConcreteModel()
35 #Indices para as cidades
36 modelo.M = pyEnv.RangeSet(n)
37 modelo.N = pyEnv.RangeSet(n)
38 #Indice auxiliar para as restricoes de quebra de ciclo
39 modelo.U = pyEnv.RangeSet(2, n)
40
41 #Variavel de decisao xij
42 modelo.x = pyEnv.Var(modelo.N, modelo.M, within=pyEnv.Binary)
43 #Variavel de decisao auxiliar u
44 modelo.u = pyEnv.Var(modelo.N, within=pyEnv.NonNegativeIntegers, bounds
45     =(0, n-1))
46
47 #Matriz de custo cij
48 modelo.c = pyEnv.Param(modelo.N, modelo.M, initialize=lambda modelo, i,
49     j: custoMatrix[i-1][j-1])
50
51 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES
52 -----##
53 def func_objetivo(modelo):
54     return sum(modelo.x[i, j] * modelo.c[i, j] for i in modelo.N for j in
55         modelo.M)
```

```

54 modelo.objetivo = pyEnv.Objective(rule=func_objetivo , sense=pyEnv.
    minimize)
55
56
57
58
59
60
61
62 ##-----##
63 #So sai 1 caminho de cada cidade
64
65 def rule_rest1(modelo , j):
66     return sum(modelo.x[i , j] for i in modelo.N if i!=j ) == 1
67
68 modelo.rest1 = pyEnv.Constraint(modelo.M, rule=rule_rest1)
69 ##-----##
70 #So entra 1 caminho em cada cidade
71
72 def rule_rest2(modelo , i):
73     return sum(modelo.x[i , j] for j in modelo.M if j!=i) == 1
74
75 modelo.rest2 = pyEnv.Constraint(modelo.N, rule=rule_rest2)
76 ##-----##
77 #Restricao de quebra de ciclos
78
79 def rule_rest3(modelo , i , j):
80     if i!=j:
81         return modelo.u[i] - modelo.u[j] + modelo.x[i , j] * n <= n-1
82     else:
83         #Sim, essa restricao nao diz nada, o solver vai ignora-la
84         return modelo.u[i] - modelo.u[i] == 0
85
86 modelo.rest3 = pyEnv.Constraint(modelo.U, modelo.N, rule=rule_rest3)
87
88
89 ##-----RESOLUCAO DO MODELO-----##
90 solver = pyEnv.SolverFactory('cplex', executable=path)
91 resultado = solver.solve(modelo, tee = True)
92 print('\n-----\n')
93 #modelo.pprint()
94 modelo.objetivo()
95 print(resultado)
96
97 ##-----PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS
98     -----##
99 ##(cidade i , cidade j)
100 lista = list(modelo.x.keys())
101 for i in lista:
102     if modelo.x[i]() != 0:
103         print(i, '—', modelo.x[i]())

```

OBS: A variável path deve conter o caminho do arquivo executável do CPLEX ou outro solver como GLPK.

8.2.1 Explicando o Código

Importamos o pyomo e o apelidamos como pyEnv, colocamos uma variável para receber o caminho do executável do solver.

```
1 | import pyomo.environ as pyEnv
2 | path = "/home/tito/cplex/bin/x86-64_linux/cplex"
```

Fornecemos os dados para o Python.

- **n:** número de cidades.
- **custoMatrix:** matriz com as distâncias entre as 53 cidades.

```

1  ##-----LEITURA DA INSTANCIA-----##
2  custoMatrix = []
3  aux = []
4  aux2 = []
5
6  file = open('53.txt')
7  linhas = file.readlines()
8  file.close()
9
10 for i in range(len(linhas)):
11     vet = linhas[i][: -1]
12     vet = vet.split('\t')
13     vet = [int(i) for i in vet if i != '']
14     aux2.append(vet)
15
16 for sublist in aux2:
17     for item in sublist:
18         aux.append(item)
19
20 tam = len(aux)
21 for i in range(53, tam+1, 53):
22     custoMatrix.append(aux[i-53:i])
23
24 custoMatrix = custoMatrix[: -1]
25 n = len(custoMatrix)

```

OBS: O arquivo 53.txt possui uma matriz 53x53 com as distâncias entre 53 cidades

Criamos o modelo.

```

1 #Modelo
2 modelo = pyEnv.ConcreteModel()

```

Criamos os índices $\mathbf{M}=(1,\dots,n)$ e $\mathbf{N}=(1,\dots,n)$ para as cidades.

```

1 #Indices para as cidades
2 modelo.M = pyEnv.RangeSet(n)
3 modelo.N = pyEnv.RangeSet(n)

```

Criamos um índice auxiliar $\mathbf{U}=(2,\dots,n)$ para as restrições de quebra de ciclo.

```

1 #Indice auxiliar para as restricoes de quebra de ciclo
2 modelo.U = pyEnv.RangeSet(2, n)

```

Criamos $\mathbf{n}*\mathbf{n}$ variáveis de decisão utilizando os índices \mathbf{N} e \mathbf{M} .

```

1 #Variavel de decisao xij
2 modelo.x=pyEnv.Var(modelo.N, modelo.M, within=pyEnv.Binary)

```

Criamos \mathbf{n} variáveis auxiliares inteiras utilizando o índice \mathbf{N} que serão limitadas

por 0 e n-1.

```
1 #Variavel de decisao auxiliar u
2 modelo.u=pyEnv.Var(modelo.N, within=pyEnv.NonNegativeIntegers, bounds
   =(0,n-1))
```

Passamos a matriz **custoMatrix** para o modelo usando os índices **N** e **M** criados anteriormente.

```
1 #Matriz de custo cij
2 modelo.c = pyEnv.Param(modelo.N, modelo.M, initialize=lambda modelo, i,
   j: custoMatrix[i-1][j-1])
```

Criamos a função objetivo utilizando os dados **custoMatrix** e as variáveis **x** fornecidas ao modelo anteriormente.

```
1 def func_objetivo(modelo):
2     return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.N for j in
   modelo.M)
3
4 modelo.objetivo = pyEnv.Objective(rule=func_objetivo, sense=pyEnv.
   minimize)
```

Criamos a primeira restrição que assegura que só sairá 1 rota de cada cidade.

```
1 #So sai 1 caminho de cada cidade
2
3 def rule_rest1(modelo, j):
4     return sum(modelo.x[i,j] for i in modelo.N if i!=j) == 1
5
6 modelo.rest1 = pyEnv.Constraint(modelo.M, rule=rule_rest1)
```

Criamos a segunda restrição que assegura que só entrará 1 rota em cada cidade.

```
1 #So entra 1 caminho em cada cidade
2
3 def rule_rest2(modelo, i):
4     return sum(modelo.x[i,j] for j in modelo.M if j!=i) == 1
5
6 modelo.rest2 = pyEnv.Constraint(modelo.N, rule=rule_rest2)
```

Criamos a restrição de quebra de ciclos que assegura que teremos uma rota que passa por todas as cidades.

```
1 #Restricao de quebra de ciclos
2
3 def rule_rest3(modelo, i, j):
4     if i!=j:
5         return modelo.u[i] - modelo.u[j] + modelo.x[i,j] * n <= n-1
6     else:
7         #Sim, essa restricao nao diz nada, o solver vai ignora-la
```

```

8         return modelo.u[i] - modelo.u[i] == 0
9
10 modelo.rest3 = pyEnv.Constraint(modelo.U, modelo.N, rule=rule_rest3)

```

Criamos a variável **solver** que poderia ser declarada como `pyEnv.SolverFactory('glpk', executable=...)` para a utilização do solver GLPK em vez do CPLEX.

```

1 solver = pyEnv.SolverFactory('cplex', executable=path)

```

Resolvemos o modelo. O parâmetro `tee=True` indica que você quer ver na tela o log do solver, se não quiser basta fazer `tee=False`.

```

1 resultado = solver.solve(modelo, tee = True)

```

Vemos os resultados da solução.

```

1 modelo.objetivo()
2 print(resultado)

```

Vemos quais as variáveis de decisão tiveram valores 1.

```

1 ##-----PRINT DAS VARIÁVEIS DE DECISÃO ESCOLHIDAS
2     -----##
3 ##(cidade i, cidade j)
4 lista = list(modelo.x.keys())
5 for i in lista:
6     if modelo.x[i]() != 0:
7         print(i, '—', modelo.x[i]())

```


9 O Problema de Localização de Facilidades Capacitadas

9.1 Descrição do Problema

No Problema de localização de facilidades capacitadas, cada cliente deve ser atribuído a uma instalação que atenda toda a sua demanda. A demanda total de clientes atribuídos a cada instalação não pode exceder sua capacidade. Um custo de abertura está associado a cada instalação, e é pago, se pelo menos um cliente é atribuído a ele. O objetivo é minimizar o custo total da abertura das instalações e suprir todos os clientes.

O modelo de programação inteira do problema é o seguinte:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^m f_i y_i + \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \\ \text{sujeito a} \quad & \sum_{j=1}^n a_j x_{ij} \leq b_i \quad , \quad i = 1, \dots, m \\ & \sum_{i=1}^m x_{ij} = 1 \quad , \quad j = 1, \dots, n \\ & x_{ij} \leq y_i \quad , \quad i = 1, \dots, m, j = 1, \dots, n \\ & x_{ij}, y_i \in \{0, 1\}, i = 1, \dots, m, j = 1, \dots, n \end{aligned}$$

\mathbf{f} contém o custo de abrir a facilidade i , \mathbf{c} é o custo de alocar o cliente j para a facilidade i , \mathbf{a} contém a demanda do cliente j . A variável de decisão binária x_{ij} diz se a facilidade i serve o cliente j , já a y_i diz se a facilidade i foi aberta ou não.

Segue na próxima página o código da modelagem e resolução do problema pelo Python utilizando o Pyomo.

9.2 Exemplo Genérico

```
1 import pyomo.environ as pyEnv
2 import numpy as np
3
4 #Single sourced facility location problem
5 #n clientes
6 #m facilidades
7 path = "/home/tito/cplex/bin/x86-64_linux/cplex"
8
9 capacidades = []
10 demandas = []
11 custoAbertura = []
12 cij = []
13
14 ##-----LEITURA DA INSTANCIA-----##
15 file = open('cap101')
16 linhas = file.readlines()
17 nFacilidades = int(linhas[0].split()[0])
18 nClientes = int(linhas[0].split()[1])
19
20 flag = 0
21 for i in linhas:
22     L = i.split()
23     if(0 < flag <= nFacilidades):
24         capacidades.append(float(L[0]))
25         custoAbertura.append(float(L[1]))
26
27     if(flag == nFacilidades +1 ):
28         demandas = [float(j) for j in L]
29
30     if(flag >= nFacilidades +2):
31         aux = []
32         for j in range(len(L)):
33             aux.append(float(L[j]))
34         cij.append(aux)
35
36     flag += 1
37
38 cij = np.transpose(cij)
39 file.close()
40
41
42 ##-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
43
44 #Modelo
45 modelo = pyEnv.ConcreteModel()
46
47 #Indices para as facilidades
48 modelo.M = pyEnv.RangeSet(nFacilidades)
49 #Indices para os clientes
50 modelo.N = pyEnv.RangeSet(nClientes)
51
52 #Variavel de decisao xij
53 modelo.x = pyEnv.Var(modelo.N, modelo.M, within=pyEnv.Binary)
54 #Variavel de decisao yi
55 modelo.y = pyEnv.Var(modelo.M, within=pyEnv.Binary)
56
57 #Matriz de custos de envio cij
```

```

58 modelo.c = pyEnv.Param(modelo.N, modelo.M, initialize=lambda modelo, i,
    j: cij[i-1][j-1])
59 #Demanda de cada cliente
60 modelo.d = pyEnv.Param(modelo.N, initialize = lambda modelo, i:
    demandas[i-1])
61 #Custo de abertura de cada facilidades
62 modelo.f = pyEnv.Param(modelo.M, initialize = lambda modelo, j:
    custoAbertura[j-1])
63 #Capacidade cada facilidade
64 modelo.b = pyEnv.Param(modelo.M, initialize = lambda modelo, j:
    capacidades[j-1])
65
66 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##
67
68 def func_objetivo(modelo):
69     return sum (modelo.f[f] * modelo.y[f] for f in modelo.M) + sum (
    modelo.c[c, f] * modelo.x[c, f] for c in modelo.N for f in modelo.M)
70
71 modelo.objetivo = pyEnv.Objective(rule = func_objetivo, sense = pyEnv.
    minimize)
72
73 ##-----##
74 #Facilidade nao exceder sua capacidade
75
76 def rule_rest1(modelo, j):
77     return sum (modelo.d[c] * modelo.x[c, j] for c in modelo.N) <=
    modelo.b[j]
78
79 modelo.rest1 = pyEnv.Constraint(modelo.M, rule=rule_rest1)
80
81
82 ##-----##
83 #Cada cliente recebe de exatamente 1 facilidade
84
85 def rule_rest2(modelo, i):
86     return sum (modelo.x[i, f] for f in modelo.M) == 1
87
88 modelo.rest2 = pyEnv.Constraint(modelo.N, rule=rule_rest2)
89
90
91 ##-----##
92 #As alocaoes sao feitas somente para facilidades abertas
93
94 def rule_rest3(modelo, i, j):
95     return modelo.x[i, j] <= modelo.y[j]
96
97 modelo.rest3 = pyEnv.Constraint(modelo.N, modelo.M, rule=rule_rest3)
98
99
100 ##-----RESOLUCAO DO MODELO-----##
101
102
103 solver = pyEnv.SolverFactory('cplex', executable=path)
104 resultado = solver.solve(modelo, tee = True)
105 print('\n-----\n')
106 modelo.objetivo()
107 print(resultado)
108
109 ##-----PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS-----##
110 ##(cliente, facilidade)

```

```

111
112 lista = list(modelo.x.keys())
113 for i in lista:
114     if modelo.x[i]() != 0:
115         print(i, '—', modelo.x[i]())

```

OBS: A variável path deve conter o caminho do arquivo executável do CPLEX ou outro solver como GLPK.

9.2.1 Explicando o Código

Importamos o pyomo e o apelidamos como pyEnv, importamos o numpy e o apelidamos como np, colocamos uma variável para receber o caminho do executável do solver.

```

1 import pyomo.environ as pyEnv
2 import numpy as np
3
4 #Single sourced facility location problem
5 #n clientes
6 #m facilidades
7 path = "/home/tito/cplex/bin/x86-64_linux/cplex"

```

Fornecemos os dados para o Python.

- **capacidades:** Lista com as capacidades de cada facilidade.
- **demandas:** Lista com as demandas de cada clientes.
- **custoAbertura:** Lista com os custos de abertura de cada facilidade.
- **cij:** Lista com o custo de alocação do cliente j para a facilidade i.

```

1 capacidades = []
2 demandas = []
3 custoAbertura = []
4 cij = []
5
6 ##-----LEITURA DA INSTANCIA-----##
7 file = open('cap101')
8 linhas = file.readlines()
9 nFacilidades = int(linhas[0].split()[0])
10 nClientes = int(linhas[0].split()[1])
11
12 flag = 0
13 for i in linhas:
14     L = i.split()
15     if(0 < flag <= nFacilidades):
16         capacidades.append(float(L[0]))
17         custoAbertura.append(float(L[1]))
18
19     if(flag == nFacilidades +1 ):
20         demandas = [float(j) for j in L]
21
22     if(flag >= nFacilidades +2):
23         aux = []
24         for j in range(len(L)):
25             aux.append(float(L[j]))

```

```

26         cij.append(aux)
27
28     flag += 1
29
30 cij = np.transpose(cij)
31 file.close()

```

OBS: O arquivo cap101 possui listas com as capacidades, demandas, custos de abertura e custo de alocação dos clientes.

Criamos o modelo.

```

1  ##-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##
2
3  #Modelo
4  modelo = pyEnv.ConcreteModel()

```

Criamos os índices $\mathbf{M}=(1,\dots,n)$ e $\mathbf{N}=(1,\dots,n)$ para respectivamente, as facilidades e os clientes.

```

1  #Índices para as facilidades
2  modelo.M = pyEnv.RangeSet(nFacilidades)
3  #Índices para os clientes
4  modelo.N = pyEnv.RangeSet(nClientes)

```

Criamos as variáveis de decisão utilizando os índices \mathbf{N} e \mathbf{M} .

```

1  #Variavel de decisao xij
2  modelo.x = pyEnv.Var(modelo.N, modelo.M, within=pyEnv.Binary)
3  #Variavel de decisao yi
4  modelo.y = pyEnv.Var(modelo.M, within=pyEnv.Binary)

```

Passamos a matriz **cij** para o modelo usando os índices \mathbf{N} e \mathbf{M} criados anteriormente.

```

1  #Matriz de custos de envio cij
2  modelo.c = pyEnv.Param(modelo.N, modelo.M, initialize=lambda modelo, i,
3  j: cij[i-1][j-1])

```

Passamos a lista **demandas** para o modelo usando o índice \mathbf{N} criado anteriormente.

```

1  #Demanda de cada cliente
2  modelo.d = pyEnv.Param(modelo.N, initialize = lambda modelo, i:
3  demandas[i-1])

```

Passamos a lista **custoAbertura** para o modelo usando o índice \mathbf{M} criado anteriormente.

```

1 #Custo de abertura de cada facilidades
2 modelo.f = pyEnv.Param(modelo.M, initialize = lambda modelo, j:
    custoAbertura[j-1])

```

Passamos a lista **capacidades** para o modelo usando o índice **M** criado anteriormente.

```

1 #Capacidade cada facilidade
2 modelo.b = pyEnv.Param(modelo.M, initialize = lambda modelo, j:
    capacidades[j-1])

```

Criamos a função objetivo utilizando os dados e as variáveis fornecidas ao modelo anteriormente.

```

1 ##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##
2
3 def func_objetivo(modelo):
4     return sum (modelo.f[f] * modelo.y[f] for f in modelo.M) + sum (
5         modelo.c[c, f] * modelo.x[c, f] for c in modelo.N for f in modelo.M)
6 modelo.objetivo = pyEnv.Objective(rule = func_objetivo, sense = pyEnv.
    minimize)

```

Criamos a primeira restrição que assegura que as facilidades não ultrapassem suas capacidades.

```

1 #Facilidade nao exceder sua capacidade
2
3 def rule_rest1(modelo, j):
4     return sum (modelo.d[c] * modelo.x[c, j] for c in modelo.N) <=
5         modelo.b[j]
6 modelo.rest1 = pyEnv.Constraint(modelo.M, rule=rule_rest1)

```

Criamos a segunda restrição que assegura que cada cliente seja atendido por exatamente 1 facilidade.

```

1 #Cada cliente recebe de exatamente 1 facilidade
2
3 def rule_rest2(modelo, i):
4     return sum (modelo.x[i, f] for f in modelo.M) == 1
5
6 modelo.rest2 = pyEnv.Constraint(modelo.N, rule=rule_rest2)

```

Criamos a terceira restrição que assegura que somente facilidades abertas sejam alocadas.

```

1 #As alocoes sao feitas somente para facilidades abertas
2

```

```

3 def rule_rest3(modelo, i, j):
4     return modelo.x[i, j] <= modelo.y[j]
5
6 modelo.rest3 = pyEnv.Constraint(modelo.N, modelo.M, rule=rule_rest3)

```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk', executable=...)* para a utilização do solver GLPK em vez do CPLEX.

```

1 solver = pyEnv.SolverFactory('cplex', executable=path)

```

Resolvemos o modelo. O parâmetro **tee=True** indica que você quer ver na tela o log do solver, se não quiser basta fazer **tee=False**.

```

1 resultado = solver.solve(modelo, tee = True)

```

Vemos os resultados da solução.

```

1 modelo.objetivo()
2 print(resultado)

```

Vemos quais as variáveis de decisão tiveram valores 1.

```

1 ##-----PRINT DAS VARIÁVEIS DE DECISÃO ESCOLHIDAS-----##
2 ##(cliente, facilidade)
3
4 lista = list(modelo.x.keys())
5 for i in lista:
6     if modelo.x[i]() != 0:
7         print(i, '—', modelo.x[i]())

```