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

2019

*Matemática Industrial, Universidade Federal do Ceará
†Departamento de Engenharia de Produção, Universidade Federal do Ceará

Conteúdo

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 e GLPK, 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.
  - Algo parecido com:***Python 3.6.8 :: Anaconda, Inc.*** deverá aparecer.

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 aparecer.

2.4 GLPK

- 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 Porque Python?

- Linguagem Interpretada.
- Sintaxe intuitiva.
- Orientada a objeto.
- 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, que são estruturas heterogêneas que podem receber tipos diferentes de dados. Como no exemplo abaixo onde na mesma lista temos um número inteiro, uma string, e outra lista com strings.

```
a = [99 , 'Pyomo', ['modelagem','matematica']]

a[0]                    #99
a[1]                    #Pyomo
a[2]                    #['modelagem','matematica']
a[2][1]                 # 'matematica'
del a[-1]               #Deleta o ultimo item
```

As listas têm os mesmos operadores que as strings.

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

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

3.7 Dicionários

```
dicionario = {'item1': 30, 'item2':33.56, 'item3': 'item'}

dicionario[0]                   #Erro, o dicionario nao possui esse indice
dicionario['item1']             #30
dicionario['item2']             #33.56
dicionario['item3']             #'item'

del dicionario['item2']         #{'item1': 30,'item3': 'item'}
dicionario['item3'] = 18.11     #{'item1': 30, item3': 18.11}
dicionario['novo_item'] = 'item4'   #{'item1': 30,'item3':18.11,'novo_item':'
    item4'}

dicionario.keys()               #['item1', 'item3', 'novo_item']
dicionario.values()             #[30, 18.11, 'item4']
```

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).

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 laco 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

```
def func(x):
    if x <= 100:
        return x**2

a = func(5)
a                                   #25

def fibonnaci(n):           #Calcula o n-esimo numero da sequencia de fibonnaci
    if n==0 or n==1:
        return n
    return fibonnaci(n-1) + fibonnaci(n-2)

f = fibonnaci(9)
f                                   #34

def mdc(a, b):
    while a != 0:
        a, b = b%a, a
    return b

m = mdc(120,145)                    #Retorna o maximo divisor comum entre 120 e 3
m                                   #5
```

3.12 Módulos

```
import numpy as np                  #Importa a biblioteca numpy e da um apelido a
    ela
from numpy.random import randint #Da biblioteca numpy.random importa so o metodo
     randint

array1 = np.array([1,2,3])
array2 = np.array([3,4,5])
prod = np.dot(array1,array2)
prod                                #Produto interno entre os dois vetores

aleatorio = randint(3,100)          #Retorna um numero inteiro aleatorio entre 3 e
    100
aleatorio
```

3.13 Arquivos

```
file = open('arq.txt')          #O arquivo precisa estar na mesma pasta do codigo
linhas = file.readlines()
linhas                          #Lista onde cada elemento representa uma linha do
    arquivo

for linha in linhas:
    print(linha[:-1])           #Printa as linhas sem o '\n'
```

4 Tutorial de Pyomo

4.1 Modelo Abstrato e Modelo Concreto

Um modelo matemático pode ser definido usando símbolos que representam valores. Por exemplo, as equações a seguir representam um modelo de programação linear para encontrar valores ótimos para o vetor **x** com os parâmetros **n** e **b**, e os vetores de parâmetros **a** e **c**:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall i = 1 \ldots m \\ & x_j \geq 0 & \forall j = 1 \ldots n \end{aligned}$$

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:

$$\begin{aligned} \min \quad & 2x_1 + 3x_2 \\ \text{s.t.} \quad & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

A classe **ConcreteModel** é usada para definir modelos concretos de otimização no Pyomo.

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

OBS: Nesse material trabalharemos com ConcreteModel.

4.2 Conjuntos

Conjuntos podem ser declarados usando as funções **Set** e **RangeSet**.
Na documentação do Pyomo é possível checar os diversos argumentos possíveis da função **Set**. Um deles é o within que 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.

```
model.A = pyEnv.RangeSet(1,3)

def s_init(model, i, j):
    if i == j:
        return i*i
    else:
        return 0.0

model.S1 = pyEnv.Param(model.A, model.A, initialize=s_init)
```

S1 será assim:

```
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:

```
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:

```
model.C = pyEnv.Set(initialize=['Scones', 'Tea'])
lb = {'Scones':2, 'Tea':4}
ub = {'Scones':5, 'Tea':7}

def fb(model, i):
    return (lb[i], ub[i])

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**:

```
def ObjRule(model):
    return 2*model.x[1] + 3*model.x[2]

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:

```
def profrul(model):
    return pyEnv.summation(model.p, model.x) + model.y

model.Obj = pyEnv.Objective(rule=ObjRule, sense=pyEnv.maximize)
```

A função **summation** performa a combinação linear entre os vetores p e x, ou seja, $p[1] * x[1] + p[2] * x[2] + ... + p[n] * x[n]$.
Nesse exemplo usamos o argumento sense para especificar maximização. O padrão 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:

```
def teaOKrule(model):
    return(model.x['butter'] + model.x['scones'] == 3)

model.TeaConst = pyEnv.Constraint(rule=teaOKrule)
```

Em vez de expressões que envolvem igualdade (==) ou desigualdades ($\leq$ ou $\geq$), as restrições também podem ser expressas usando uma tupla de três, se a forma (lb, expr, ub) em que lb e ub podem ser None, que é interpretada como lb ¡= expr ¡= ub. Variáveis podem aparecer apenas no meio expr. Por exemplo, as duas de restrições a seguir têm o mesmo significado:

```
model.x = pyEnv.Var()

def aRule(model):
    return model.x >= 2

model.Boundx = pyEnv.Constraint(rule=aRule)

def bRule(model):
    return (2, model.x, None)

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:

```
model.A = pyEnv.RangeSet(1,10)
model.a = pyEnv.Param(model.A, within=pyEnv.PositiveReals)
model.ToBuy = pyEnv.Var(model.A)

def bud_rule(model, i):
    return model.a[i]*model.ToBuy[i] <= i

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:

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

Sendo **c** o vetor de lucros, **p** o vetor de pesos e **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:
Max $92x_1 + 57x_2 + 49x_3 + 68x_4 + 60x_5 + 43x_6 + 67x_7 + 84x_8 + 87x_9 + 72x_10$

E a seguinte restrição:
$23x_1 + 31x_2 + 29x_3 + 44x_4 + 53x_5 + 38x_6 + 63x_7 + 85x_8 + 89x_9 + 82x_10 <= 165$

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

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

##-------------------------DADOS-------------------------##
lucros = [92,57,49,68,60,43,67,84,87,72]
pesos = [23,31,29,44,53,38,63,85,89,82]
capacidade = 165
m = len(lucros)
##-------------------DECLARACAO DO MODELO-----------------##
modelo = pyEnv.ConcreteModel()
modelo.I = range(m)

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

modelo.objetivo = pyEnv.Objective(expr = sum(lucros[i]*modelo.x[i] for i in
    modelo.I)
,sense = pyEnv.maximize)

modelo.rest = pyEnv.Constraint(expr = sum(pesos[i]*modelo.x[i] for i in modelo.I
    ) <= capacidade)


##-------------------RESOLUCAO DO MODELO------------------##
solver = pyEnv.SolverFactory('cplex',executable = path)
resultado_objetivo = solver.solve(modelo, tee=True)

##---------------PRINT DAS VARIAVEIS DE DECISAO-----------##
lista = list(modelo.x.keys())
for i in lista:
    print(i,'--', modelo.x[i]())

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.

```
import pyomo.environ as pyEnv
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.

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

Criamos o modelo.

```
modelo = pyEnv.ConcreteModel()
```

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

```
modelo.I = range(m)
```

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

```
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.

```
modelo.objetivo = pyEnv.Objective(expr = sum(lucros[i]*modelo.x[i] for i in
    modelo.I)
,sense = pyEnv.maximize)

modelo.rest = pyEnv.Constraint(expr = sum(pesos[i]*modelo.x[i] for i in modelo.I
    ) <= capacidade)
```

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

```
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.

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

Vemos os resultados da solução.

```
##-----------------PRINT DAS VARIAVEIS DE DECISAO-----------------##
lista = list(modelo.x.keys())
for i in lista:
    print(i,'--', modelo.x[i]())

print('Valor objetivo = ',modelo.objetivo())
```

5.3 Exemplo Genérico

```
#Problema da Mochila - Knapsack Problem
import pyomo.environ as pyEnv

path = "/home/tito/cplex/bin/x86-64_linux/cplex"

##---------------LEITURA DA INSTANCIA-------------------------------##

filePesoMaximo = open('p01_c.txt')
pesoMaximo  = filePesoMaximo.readline().split('\n')[0]
pesoMaximo = int(pesoMaximo)

filePesos = open('p01_w.txt')
pesos  = filePesos.readlines()
pesos = pesos[0].split()
pesos = [int(i) for i in pesos]


fileLucros = open('p01_p.txt')
lucros  = fileLucros.readlines()
lucros = lucros[0].split()
lucros = [int(i) for i in lucros]

m = len(pesos)

#-------------DECLARACAO DO MODELO E SEUS PARAMETROS-------------##
#Modelo
modelo = pyEnv.ConcreteModel()
#Indice para os pesos e precos
modelo.I = pyEnv.RangeSet(m)

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

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

##--------DECLARACAO DA FUNCAO OBJETIVO E RESTRICAO-------------##
#Funcao objetivo do problema
modelo.objetivo = pyEnv.Objective(expr = sum(modelo.c[i]*modelo.x[i] for i in
    modelo.I), sense=pyEnv.maximize)

#Restricao do problema
modelo.restricao = pyEnv.Constraint(expr = sum(modelo.p[i] * modelo.x[i] for i
    in modelo.I) <= pesoMaximo)


##-----------------RESOLUCAO DO MODELO-----------------------------##
solver = pyEnv.SolverFactory('cplex',executable=path)
resultado = solver.solve(modelo, tee= True)
print('\n-------------------------\n')
modelo.pprint()
modelo.objetivo()
print(resultado)

##---------------PRINT DAS VARIAVEIS DE DECISAO-----------------##
lista = list(modelo.x.keys())
```

```
for i in lista:
    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.

```
#Problema da Mochila - Knapsack Problem
import pyomo.environ as pyEnv

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.

```
##-----------------LEITURA DA INSTANCIA-----------------------------##

filePesoMaximo = open('p01_c.txt')
pesoMaximo  = filePesoMaximo.readline().split('\n')[0]
pesoMaximo = int(pesoMaximo)

filePesos = open('p01_w.txt')
pesos  = filePesos.readlines()
pesos = pesos[0].split()
pesos = [int(i) for i in pesos]


fileLucros = open('p01_p.txt')
lucros  = fileLucros.readlines()
lucros = lucros[0].split()
lucros = [int(i) for i in lucros]

m = len(pesos)
```

Criamos o modelo.

```
#-------------DECLARACAO DO MODELO E SEUS PARAMETROS-------------##
#Modelo
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.

```
#Indice para os pesos e precos
modelo.I = pyEnv.RangeSet(m)
```

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

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

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

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

```
#Funcao objetivo do problema
modelo.objetivo = pyEnv.Objective(expr = sum(modelo.c[i]*modelo.x[i] for i in
    modelo.I), sense=pyEnv.maximize)

#Restricao do problema
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=path)*, para a utilização do solver GLPK em vez do CPLEX.

```
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.

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

Vemos os resultados da solução.

```
modelo.pprint()
modelo.objetivo()
print(resultado)
```

Vemos os valores das variáveis de decisão.

```
lista = list(modelo.x.keys())
for i in lista:
    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:

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

c contém o custo de transporte da unidade da fábrica i para o cliente j, **s** contém a capacidade máxima de cada fábrica e **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:
Max $162x_{11} + 247x_{12} + 117x_{21} + 193x_{22} + 131x_{31} + 185x_{32}$

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.

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

#Custo de transporte da fabrica i pro cliente j
cij = [[162,247],
       [117,193],
       [131,185]]

capacidades = [1000,1500,1200]
demandas = [2300,1400]

m = len(capacidades)
n = len(demandas)

##-----------DECLARACAO DO MODELO E SEUS PARAMETROS----------##
#Modelo
modelo = pyEnv.ConcreteModel()
#Indice para as fabricas
modelo.I = pyEnv.RangeSet(m)
#Indice para os clientes
modelo.J = pyEnv.RangeSet(n)

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

#Custo de transporte da fabrica i pro cliente j
modelo.c = pyEnv.Param(modelo.I, modelo.J,initialize=lambda modelo, i, j: cij[i
    -1][j-1])
#Capacidade de cada fabrica
modelo.b = pyEnv.Param(modelo.I, initialize = lambda modelo,i: capacidades[i-1])
#Demanda de cada cliente
modelo.d = pyEnv.Param(modelo.J, initialize = lambda modelo,j: demandas[j-1])


##-------DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES---------##

def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in modelo.J
    )

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


##-------------------------------------------------------------##
#Cada fabrica nao exceder sua capacidade

def rule_rest1(modelo,i):
    return sum(modelo.x[i,j] for j in modelo.J) <= modelo.b[i]

modelo.rest1 = pyEnv.Constraint(modelo.I,rule=rule_rest1)


##-------------------------------------------------------------##
#Cada cliente ter sua demanda atendida

def rule_rest2(modelo,j):
    return sum(modelo.x[i,j] for i in modelo.I) >= modelo.d[j]

modelo.rest2 = pyEnv.Constraint(modelo.J,rule=rule_rest2)

```

```
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 **I**=(1,...,m) para as fábricas.

```
#Indice para as fabricas
modelo.I = pyEnv.RangeSet(m)
```

Criamos um índice **J**=(1,...,n) para os clientes.

```
#Indice para os clientes
modelo.J = pyEnv.RangeSet(n)
```

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

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

Passamos a matriz **cij** para o modelo usando os índices **I** e **J** criados anteriormente.

```
#Custo de transporte da fabrica i pro cliente j
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 **I** criado anteriormente.

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

Passamos as **demandas** para o modelo usando o índice **J** criado anteriormente.

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

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

```
##-------DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES---------##

def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in modelo.J
    )

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

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

```
#Cada fabrica nao exceder sua capacidade

def rule_rest1(modelo,i):
    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=path)*, 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 VARIAVEIS DE DECISAO 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:

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

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

```
import pyomo.environ as pyEnv

path = "/home/tito/cplex/bin/x86-64_linux/cplex"

#Alocar m  tarefas [i] para n unidades[j]

##----------------LEITURA DA INSTANCIA----------------##
file = open('assign300.txt')
linhas = file.readlines()
file.close()

cij = []
aux = []
aux2=[]

m=int(linhas[0].split('\n')[0])
linhas = linhas[1:]


for i in range(len(linhas)):
    vet=linhas[i][:-1]
    splitada = vet.split()
    splitada = [int(i) for i in splitada if i!= '']
    aux.append(splitada)

for sublist in aux:
    for item in sublist:
        aux2.append(item)

tam = len(aux2)
for i in range(m,tam+1,m):
    cij.append(aux2[i-m:i])

n = len(cij[0])

##-----DECLARACAO DO MODELO E SEUS PARAMETROS-----##

#Modelo
modelo = pyEnv.ConcreteModel()
#Indices para as tarefas a serem alocadas
modelo.I = pyEnv.RangeSet(n)
#Indices para unidades
modelo.J = pyEnv.RangeSet(m)

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

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


##-----DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-----##

def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in 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

```
##-----------------LEITURA DA INSTANCIA-----------------##
file = open('assign300.txt')
linhas = file.readlines()
file.close()

cij = []
aux = []
aux2=[]

m=int(linhas[0].split('\n')[0])
linhas = linhas[1:]


for i in range(len(linhas)):
    vet=linhas[i][:-1]
    splitada = vet.split()
    splitada = [int(i) for i in splitada if i!= '']
    aux.append(splitada)

for sublist in aux:
    for item in sublist:
        aux2.append(item)

tam = len(aux2)
for i in range(m,tam+1,m):
    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.

```
#Modelo
modelo = pyEnv.ConcreteModel()
```

Criamos um índice **I**=(1,...,n) para as tarefas a serem alocadas.

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

Criamos um índice **J**=(1,...,m) para as unidades.

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

Passamos a matriz **cij** para o modelo usando os indices **I** e **J** criados anteriormente.

```
#Custo de alocacao da tarefa i pra unidade j
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**.

```
#Variaveis de decisao xij
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.

```
def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.I for j in modelo.J
    )

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.

```
def rule_rest1(modelo,i):
    return sum(modelo.x[i,j] for j in modelo.J) == 1

modelo.rest1 = pyEnv.Constraint(modelo.I,rule=rule_rest1)
```

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

```
def rule_rest2(modelo,j):
    return sum(modelo.x[i,j] for i in modelo.I) == 1

modelo.rest2 = pyEnv.Constraint(modelo.J,rule=rule_rest2)
```

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

```
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.

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

Vemos os resultados da solução.

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

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

```
#(tarefa,unidade)
lista = list(modelo.x.keys())
for i in lista:
    if modelo.x[i]() != 0:
        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:

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

c contém o custo de ir da cidade i para a cidade j, **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

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


##-------------------------------LEITURA DA INSTANCIA-------------------------------##
custoMatrix = []
aux = []
aux2 = []

file = open('53.txt')
linhas = file.readlines()
file.close()

for i in range(len(linhas)):
    vet = linhas[i][:-1]
    vet = vet.split('\t')
    vet = [int(i) for i in vet if i!= '']
    aux2.append(vet)

for sublist in aux2:
    for item in sublist:
        aux.append(item)

tam = len(aux)
for i in range(53,tam+1,53):
    custoMatrix.append(aux[i-53:i])

custoMatrix = custoMatrix[:-1]
n = len(custoMatrix)

##-------------------------------DECLARACAO DO MODELO E SEUS PARAMETROS
    ----------------------##

#Modelo
modelo = pyEnv.ConcreteModel()
#Indices para as cidades
modelo.M = pyEnv.RangeSet(n)
modelo.N = pyEnv.RangeSet(n)
#Indice auxiliar para as restricoes de quebra de ciclo
modelo.U = pyEnv.RangeSet(2,n)

#Variavel de decisao xij
modelo.x=pyEnv.Var(modelo.N,modelo.M, within=pyEnv.Binary)
#Variavel de decisao auxiliar u
modelo.u=pyEnv.Var(modelo.N, within=pyEnv.NonNegativeIntegers,bounds=(0,n-1))

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

##-------------------------------DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES
    ----------------##

def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.N for j in modelo.M
    )

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 ##(cidade i,cidade j)
99 lista = list(modelo.x.keys())
100 for i in lista:
101     if modelo.x[i]() != 0:
102         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.

```
##-------------------------------LEITURA DA INSTANCIA-------------------------------##
custoMatrix = []
aux = []
aux2 = []

file = open('53.txt')
linhas = file.readlines()
file.close()

for i in range(len(linhas)):
    vet = linhas[i][:-1]
    vet = vet.split('\t')
    vet = [int(i) for i in vet if i!= '']
    aux2.append(vet)

for sublist in aux2:
    for item in sublist:
        aux.append(item)

tam = len(aux)
for i in range(53,tam+1,53):
    custoMatrix.append(aux[i-53:i])

custoMatrix = custoMatrix[:-1]
n = len(custoMatrix)
```

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

Criamos o modelo.

```
#Modelo
modelo = pyEnv.ConcreteModel()
```

Criamos os índices **M**=(1,...,n) e **N**=(1,...,n) para as cidades.

```
#Indices para as cidades
modelo.M = pyEnv.RangeSet(n)
modelo.N = pyEnv.RangeSet(n)
```

Criamos um índice auxiliar **U**=(2,...,n) para as restrições de quebra de ciclo.

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

Criamos **n*n** variáveis de decisão utilizando os índices **N** e **M**.

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

Criamos **n** variáveis auxiliares inteiras utilizando o índice **N** que serão limitadas por 0 e n-1.

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

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

```
#Matriz de custo cij
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.

```
def func_objetivo(modelo):
    return sum(modelo.x[i,j] * modelo.c[i,j] for i in modelo.N for j in modelo.M
    )

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

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

```
#So sai 1 caminho de cada cidade

def rule_rest1(modelo,j):
    return sum(modelo.x[i,j] for i in modelo.N if i!=j ) == 1

modelo.rest1 = pyEnv.Constraint(modelo.M,rule=rule_rest1)
```

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

```
#So entra 1 caminho em cada cidade

def rule_rest2(modelo,i):
    return sum(modelo.x[i,j] for j in modelo.M if j!=i) == 1

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.

```
#Restricao de quebra de ciclos

def rule_rest3(modelo,i,j):
    if i!=j:
        return modelo.u[i] - modelo.u[j] + modelo.x[i,j] * n <= n-1
    else:
        #Sim, essa restricao nao diz nada, o solver vai ignora-la
        return modelo.u[i] - modelo.u[i] == 0

modelo.rest3 = pyEnv.Constraint(modelo.U,modelo.N,rule=rule_rest3)
```

Criamos a variável **solver** que poderia ser declarada como *pyEnv.SolverFactory('glpk',executable=path)*, 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.

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

Vemos os resultados da solução.

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

Vemos quais as variáveis de decisão tiveram valores 1.

```
##-------------------------------PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS
    ---------------------##
##(cidade i,cidade j)
lista = list(modelo.x.keys())
for i in lista:
    if modelo.x[i]() != 0:
        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:

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

f contém o custo de abrir a facilidade i, **c** é o custo de alocar o cliente j para a facilidade i, **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

```
import pyomo.environ as pyEnv
import numpy as np

#Single sourced facility location problem
#n clientes
#m facilidades
path = "/home/tito/cplex/bin/x86-64_linux/cplex"

capacidades = []
demandas = []
custoAbertura = []
cij = []

##-------------------------------LEITURA DA INSTANCIA-------------------------------##
file = open('cap101')
linhas = file.readlines()
nFacilidades = int(linhas[0].split()[0])
nClientes = int(linhas[0].split()[1])

flag = 0
for i in linhas:
    L = i.split()
    if(0 < flag <= nFacilidades):
        capacidades.append(float(L[0]))
        custoAbertura.append(float(L[1]))

    if(flag == nFacilidades +1 ):
        demandas = [float (j) for j in L]

    if(flag >= nFacilidades +2):
        aux = []
        for j in range(len(L)):
            aux.append(float(L[j]))
        cij.append(aux)

    flag += 1

cij = np.transpose(cij)
file.close()


##----------DECLARACAO DO MODELO E SEUS PARAMETROS----------##

#Modelo
modelo = pyEnv.ConcreteModel()

#Indices para as facilidades
modelo.M = pyEnv.RangeSet(nFacilidades)
#Indices para os clientes
modelo.N = pyEnv.RangeSet(nClientes)

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

#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 alocacoes 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.

```
##----------DECLARACAO DO MODELO E SEUS PARAMETROS----------##

#Modelo
modelo = pyEnv.ConcreteModel()
```

Criamos os índices **M**=(1,...,n) e **N**=(1,...,n) para respectivamente, as facilidades e os clientes.

```
#Indices para as facilidades
modelo.M = pyEnv.RangeSet(nFacilidades)
#Indices para os clientes
modelo.N = pyEnv.RangeSet(nClientes)
```

Criamos as variáveis de decisão utilizando os índices **N** e **M**.

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

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

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

Passamos a lista **demandas** para o modelo usando o índice **N** criado anteriormente.

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

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

```
#Custo de abertura de cada facilidades
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.

```
#Capacidade cada facilidade
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.

```
##-------------DECLARACAO DA FUNCAO OBJETIVO E RESTRICOES-------------##

def func_objetivo(modelo):
    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)

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

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

```
#Facilidade nao exceder sua capacidade

def rule_rest1(modelo,j):
    return sum (modelo.d[c] * modelo.x[c,j] for c in modelo.N) <= modelo.b[j]

modelo.rest1 = pyEnv.Constraint(modelo.M, rule=rule_rest1)
```

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

```
#Cada cliente recebe de exatamente 1 facilidade

def rule_rest2(modelo,i):
    return sum (modelo.x[i,f] for f in modelo.M) == 1

modelo.rest2 = pyEnv.Constraint(modelo.N, rule=rule_rest2)
```

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

```
#As alocacoes sao feitas somente para facilidades abertas

def rule_rest3(modelo,i,j):
    return  modelo.x[i,j] <= modelo.y[j]

modelo.rest3 = pyEnv.Constraint(modelo.N,modelo.M, rule=rule_rest3)
```

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

```
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.

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

Vemos os resultados da solução.

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

Vemos quais as variáveis de decisão tiveram valores 1.

```
##----------------PRINT DAS VARIAVEIS DE DECISAO ESCOLHIDAS----------##
##(cliente,facilidade)

lista = list(modelo.x.keys())
for i in lista:
    if modelo.x[i]() != 0:
        print(i,'--', modelo.x[i]())
```