**Pesquisa Operacional para o Desenvolvimento**
Vol. 11, n. 3, p. 127–138, 2019

**ISSN 1984-3534**

# TUTORIAL DE DESENVOLVIMENTO DE MÉTODOS DE PROGRAMAÇÃO LINEAR INTEIRA MISTA EM PYTHON USANDO O PACOTE PYTHON-MIP[1]

**Haroldo G. Santos**[a*] **, Túlio A. M. Toffolo**[a]

[a]Instituto de Ciências Exatas e Biológicas, Departamento de Computação
Universidade Federal de Ouro Preto - UFOP, Ouro Preto-MG, Brasil

Recebido 11/11/2019, aceito 17/12/2019

## RESUMO

O pacote Python-MIP oferece um conjunto abrangente de ferramentas para modelagem e resolução de Problemas de Programação Inteira Mista em Python. Além de oferecer uma linguagem de modelagem de alto nível, o pacote permite o desenvolvimento de métodos avançados, habilitando a comunicação bidirecional com o pacote de otimização durante o processo de busca. Neste tutorial, desenvolveremos métodos de Programação Linear Inteira Mista para o Problema do Caixeiro Viajante. Iniciando com um método simples baseado em uma formulação compacta iremos evoluir para um método que combina heurísticas e planos de corte para a resolução de problemas maiores.

**Palavras-chave: Otimização combinatória, Caixeiro viajante, Programação linear inteira, Python.**

## ABSTRACT

The Python-MIP package offers a comprehensive set of tools for the modeling and solution of Integer Linear Programming Problems in Python. Besides providing a high level modeling language, the package allows the development of advanced solvers with bidirectional communication with the solver during the search process. In this tutorial we develop solvers for the Traveling Salesman Problem. Starting with a simple solver based on a compact formulation we evolve to a solver combining heuristics and cutting planes for the solution of larger instances.

**Keywords: Combinatorial optimization, Traveling salesman problem, Integer linear programming, Python.**

[*] Autor para correspondência. E-mail: haroldo@ufop.edu.br
DOI: 10.4322/PODes.2019.009

[1] *Todos os autores assumem a responsabilidade pelo conteúdo do artigo.*

## 1. Introdução

Python-MIP é um software livre de código aberto (*www.python-mip.com*) para modelagem de Problemas de Programação Linear Inteira Mista (Wolsey, 1998) em Python desenvolvido pelos autores deste tutorial. O projeto do pacote foi concebido com o objetivo de desenvolver uma ferramenta que atenda aos seguintes requisitos:

1. clareza de código e modelagem de alto nível;
2. alto desempenho;
3. extensibilidade e configurabilidade.

Tradicionalmente, os requisitos 1 e 2 são considerados conflitantes. Até recentemente, as opções mais recomendadas para os interessados no primeiro requisito eram linguagens algébricas de alto nível como AMPL (Fourer et al., 1987). A obtenção de desempenho máximo costumava requerer o uso de linguagens de mais baixo nível como C (Ritchie et al., 1978). Pacotes de otimização estado-da-arte como o CPLEX foram escritos nessa linguagem (Bixby, 2002). Desse modo, a biblioteca completa de funções estava originalmente disponível somente nela. Recentemente, soluções como JuMP (Dunning et al., 2017; Castellucci, 2017) demonstraram que os requistos 1 e 2 não são necessariamente conflitantes: linguagens de alto nível como Julia juntamente com compiladores *just-in-time* permitem o desenvolvimento rápido de métodos que apresentam alto desempenho. O objetivo do projeto Python-MIP é o desenvolvimento de um pacote de modelagem e desenvolvimento de algoritmos de Programação Linear Inteira Mista para a linguagem Python que atenda plenamente aos três requisitos destacados anteriormente.

Pesquisas recentes mostram que Python está se tornando a linguagem mais popular da atualidade (The Economist, 2018). O projeto Python-MIP foi inspirado em dois projetos de código aberto para programação linear inteira em Python. O primeiro é o PuLP (Mitchell, 2009), que oferece uma linguagem de modelagem de alto nível e interface para vários pacotes de otimização comerciais e de código aberto. No entanto, recursos que requerem uma integração maior com o pacote, como geração dinâmica de planos de cortes, não estão disponíveis neste pacote. O pacote CyLP, por outro lado, suporta geração dinâmica de planos de corte mas não oferece uma linguagem de modelagem de alto nível (Towhidi e Orban, 2016), mas é compatível apenas com o pacote de otimização COIN-OR CBC (Forrest e Lougee-Heimer, 2005). O pacote Python-MIP foi criado com o objetivo de prover a funcionalidade dos dois pacotes com máximo desempenho. A escrita de um um novo pacote de programação linear inteira em Python também permite que recursos relativamente novos da linguagem, como a tipagem estática e a comunicação direta com bibliotecas nativas (Python CFFI) sejam utilizados extensivamente no código.

Neste tutorial, apresentaremos versões sucessivamente mais sofisticadas de um método para resolver o clássico Problema do Caixeiro Viajante (Dantzig et al., 1954; Miller et al., 1960; Applegate et al., 2006), que é brevemente descrito na Seção 2. Os métodos foram desenvolvidos utilizando o pacote Python-MIP. Enquanto na primeira versão a comunicação com o pacote de otimização ocorre somente nos momentos de criação do modelo e coleta dos resultados, a versão final utiliza comunicação bidirecional com o pacote de otimização durante o processo de busca para o tratamento de uma formulação com um número exponencial de restrições em um método *branch&cut*. Resultados experimentais são apresentados na Seção 3 para demonstrar os ganhos substanciais de desempenho que podem ser obtidos com essa última versão.

## 2. Aplicação: Problema do Caixeiro Viajante

O problema do caixeiro viajante consiste em: dada uma malha viária e um conjunto de pontos que devem ser visitados, encontrar uma rota de custo mínimo (tempo ou distância, usualmente) que inclua todos os pontos percorrendo-os exatamente uma vez. Formalmente, temos como dados de entrada um grafo direcionado $G = (V, A)$ com custos associadas aos arcos:

$V$ é o conjunto de vértices, $V = \{0, 1, ..., n\}$;

$c_{i,j}$ é custo de percorrer a arco $(i, j) \in A$

Em todas as formulações consideradas neste tutorial, utilizamos as seguintes variáveis binárias de decisão que representam a escolha dos arcos que compõem a rota:

$$x_{i,j} = \begin{cases} 1 & \text{se o arco } (i,j) \text{ foi escolhido para a rota} \\ 0 & \text{caso contrário} \end{cases}$$

Como exemplo, considere o mapa da Figura 1 que inclui quatorze cidades turísticas da Bélgica.

Figure 1: Exemplo com 14 cidades turísticas da Bélgica.


Fonte: Elaborada pelos autores.

### 2.1. Uma Formulação Compacta

O problema do caixeiro viajante pode ser modelado utilizando-se uma formulação compacta, isto é, uma formulação com um número polinomial de variáveis e de restrições. Formulações desse tipo, apesar de usualmente não serem a melhor opção de resolução em termos de desempenho para o problema, são convenientes para uma primeira abordagem pois podem ser facilmente inseridas de uma vez só como entrada para um pacote de otimização. A formulação (1)-(6) foi proposta por Miller et al. (1960):

Minimize:

$$\sum_{i \in V} \sum_{j \in V} c_{i,j} . x_{i,j} \tag{1}$$

Sujeito a:

$$\sum_{j \in V \setminus \{i\}} x_{i,j} = 1 \ \ \forall i \in V \tag{2}$$

$$\sum_{i \in V \setminus \{j\}} x_{i,j} = 1 \ \ \forall j \in V \tag{3}$$

$$y_i - (n+1) . x_{i,j} \geq y_j - n \ \ \forall i \in V \setminus \{0\}, j \in V \setminus \{0, i\} \tag{4}$$

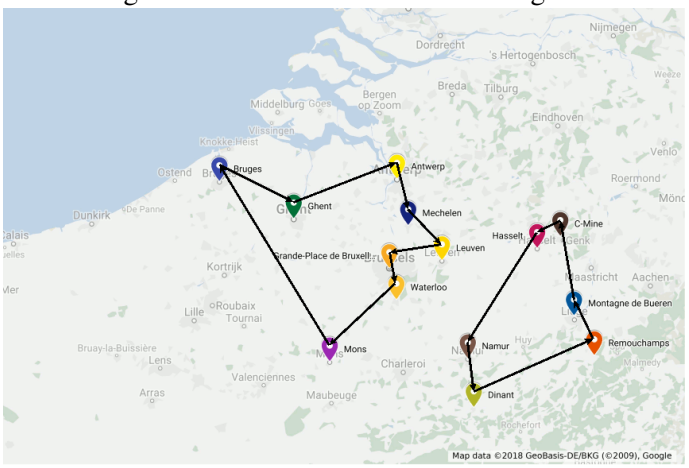
$$x_{i,j} \in \{0, 1\} \ \ \forall i \in V, j \in V \tag{5}$$

$$y_i \geq 0 \ \ \forall i \in V \tag{6}$$

A função objetivo (1) minimiza o custo total da rota. As equações (2) e (3), denominadas restrições de grau, garantem que cada vértice é visitado somente uma vez enquanto variáveis

auxiliares $y_i$ são utilizadas nas restrições (4) para garantir que uma vez que um arco $(i, j)$ seja selecionado, o valor de $y_j$ seja maior do que o valor de $y_i$ em uma unidade. Essa propriedade é garantida para todos os vértices exceto o vértice 0, que é arbitrariamente selecionado como origem de modo a evitar a construção de sub-rotas desconectadas, como no exemplo da Figura 2, onde os valores das variáveis $x_{i,j}$ indicados nos arcos representam uma solução viável se somente as restrições (2) e (3) fossem consideradas.

Figure 2: Rotas desconectadas da origem.




Fonte: Elaborada pelos autores.

### 2.2. Implementação do Modelo Usando o Python-MIP

Destacamos que a instalação do pacote é bastante simples. Recomenda-se fazer a instalação por meio do instalador de pacotes pip (*pypi.org/project/pip*). Utilizando o pip, para instalar o Python-MIP basta executar o seguinte comando no terminal do sistema:

```
pip install mip --user
```

A seguir temos um exemplo completo de um modelo escrito em Python-MIP para resolver o Problema do Caixeiro Viajante para o mapa da Figura 1. O código a seguir utiliza a formulação compacta dada pelas equações (1)-(6), descrita anteriormente.

Nas linhas 6-9 nomeamos nossos pontos turísticos. Nas linhas 12-25 informamos o tempo estimado de deslocamento entre cada par de cidades $(i, j)$ onde $i < j$, visto que para nosso exemplo consideramos que as distâncias de ida e volta são iguais, ou seja, temos um Problema do Caixeiro Viajante Simétrico. Convertemos a matriz triangular `dists` em uma matriz completa nas linhas 31-34.

A linha 36 cria um modelo de programação linear inteira. As variáveis do modelo são criadas nas linhas 39 e 43 utilizando o método `add_var` em nosso modelo `model`. Durante a criação das restrições, é necessário referenciar as variáveis criadas. Por isso, utilizamos a matriz `x` para mapear cada arco do grafo com sua respectiva variável binária. O vetor `y` é utilizado para mapear cada nó com sua respectiva variável auxiliar contínua para eliminação de sub-rotas.

A função objetivo que minimiza o tempo total gasto para percorrer a rota é informada na linha 46. Nesse caso, para cada arco multiplicamos sua respectiva variável binária de seleção pelo tempo de deslocamento do arco armazenado em `c`.

```
1  from itertools import product
2  from sys import stdout as out
3  from mip import Model, xsum, minimize, BINARY
4
5  # lugares de visitação
6  places = ['Antwerp', 'Bruges', 'C-Mine', 'Dinant', 'Ghent',
7            'Grand-Place de Bruxelles', 'Hasselt', 'Leuven',
8            'Mechelen', 'Mons', 'Montagne de Bueren', 'Namur',
9            'Remouchamps', 'Waterloo']
10
11 # distâncias em matriz triangular superior
12 dists = [[83, 81, 113, 52, 42, 73, 44, 23, 91, 105, 90, 124, 57],
13           [161, 160, 39, 89, 151, 110, 90, 99, 177, 143, 193, 100],
14           [90, 125, 82, 13, 57, 71, 123, 38, 72, 59, 82],
15           [123, 77, 81, 71, 91, 72, 64, 24, 62, 63],
16           [51, 114, 72, 54, 69, 139, 105, 155, 62],
17           [70, 25, 22, 52, 90, 56, 105, 16],
18           [45, 61, 111, 36, 61, 57, 70],
19           [23, 71, 67, 48, 85, 29],
20           [74, 89, 69, 107, 36],
21           [117, 65, 125, 43],
22           [54, 22, 84],
23           [60, 44],
24           [97],
25           []]
26
27 # número de nós e lista de vértices
28 n, V = len(dists), set(range(len(dists)))
29
30 # matriz de distâncias completa
31 c = [[0 if i == j
32       else dists[i][j-i-1] if j > i
33       else dists[j][i-j-1]
34       for j in V] for i in V]
35
36 model = Model()
37
38 # variáveis binárias indicando se arco (i,j) é usado na rota
39 x = [[model.add_var(var_type=BINARY) for j in V] for i in V]
40
41 # variáveis contínuas para prevenção de sub-rotas: cada cidade terá
42 # um identificador numérico maior na rota, excetuando-se a primeira
43 y = [model.add_var() for i in V]
44
45 # função objetivo: minimizar custo total
46 model.objective = minimize(xsum(c[i][j]*x[i][j] for i in V for j in V))
47
48 # restrição : sair de cada cidade somente uma vez
49 for i in V:
50     model += xsum(x[i][j] for j in V - {i}) == 1
51
52 # restrição : entrar em cada cidade somente uma vez
53 for i in V:
54     model += xsum(x[j][i] for j in V - {i}) == 1
55
56 # eliminação de sub-rotas
57 for (i, j) in product(V - {0}, V - {0}):
58     if i != j:
59         model += y[i] - (n+1)*x[i][j] >= y[j]-n
60
61 # otimização com limite de tempo de 30 segundos
62 model.optimize(max_seconds=30)
63
64 # verificando se ao menos uma solução válida foi encontrada
65 if model.num_solutions:
66     out.write('route with total distance %g found: %s'
67               % (model.objective_value, places[0]))
68     nc = 0
69     while True:
70         nc = [i for i in V if x[nc][i].x >= 0.99][0]
71         out.write(' -> %s' % places[nc])
72         if nc == 0:
73             break
74     out.write('\n')
```

As restrições são criadas nas linhas 48-59. Em todos os casos utilizamos o operador += sobre o modelo `m` para adicionar restrições lineares. Note que assim como na função objetivo,

o somatório é efetuado com a função `xsum`. Esta função é similar a função `sum` disponível na linguagem Python mas otimizada para a situação específica de escrita de restrições lineares no pacote Python-MIP.

A linha 62 dispara a otimização do modelo indicando um limite de tempo para a otimização de 30 segundos. Na linha 65, verificamos se uma solução viável foi encontrada e, em caso positivo, a escrevemos nas linhas 65-74.

Os pacotes de programação linear inteira executam uma busca em árvore onde são utilizados *limites* para a poda de nós. O limite superior é obtido a partir de qualquer solução viável que for encontrada durante a busca e o limite inferior corresponde ao custo obtido com a resolução do problema com as restrições de integralidade das variáveis relaxadas. No nosso caso considerando domínio das variáveis $x$ como contínuo entre 0 e 1. A formulação aqui utilizada tem uma grave deficiência: o limite inferior por ela produzido é distante do custo ótimo da solução. Desse modo, o desempenho dos pacotes de programação linear inteira em sua resolução é bastante pobre. Essa deficiência não é visível quando resolvemos problemas pequenos como o de exemplo (apenas 14 cidades), mas faz com que o desempenho do pacote degenere muito rapidamente a medida que o tamanho do grafo de entrada aumenta. Na seção seguinte veremos o tratamento de uma formulação mais apropriada.

### 2.3. Formulações Fortes com Geração Dinâmica de Restrições

A formulação do caixeiro viajante utilizada nos métodos de melhor desempenho inclui restrições de eliminação de sub-rotas do tipo:

$$\sum_{(i,j)\in S\times S} x_{(i,j)} \leq |S| - 1 \;\; \forall S \subset V \tag{7}$$

O problema das desigualdades acima é que elas devem ser geradas para *cada subconjunto* $S$ de vértices do grafo, ou seja, para um grafo com $n$ vértices temos $2^n - 1$ subconjuntos não vazios. Inseri-las no modelo inicial é inviável, exceto para instâncias pequenas. Uma solução para esse problema é inserir somente as restrições *necessárias* no modelo, com o método dos Planos de Corte (Dantzig et al., 1954). O pacote Python-MIP permite uma comunicação bi-direcional com o resolvedor para que as restrições necessárias sejam inseridas *durante* a busca. Para isso precisamos criar uma classe derivada da classe `ConstrsGenerator` que implemente o método `generate_constrs`. Esse método recebe como parâmetro um modelo, onde a solução corrente pode ser consultada e restrições adicionais podem ser inseridas na medida do necessário, caso a solução corrente não as satisfaça.

O pacote Python-MIP permite um controle sobre em que ponto da busca as restrições faltantes são verificadas. Duas propriedades do modelo podem ser preenchidas com objeto(s) do tipo `ConstrsGenerator`:

**`cuts_generator`** : caso o gerador de restrições seja informado com essa propriedade do modelo, a verificação por restrições faltantes (cortes violados) é realizada sempre que uma *solução fracionária* for gerada durante a resolução de um nó na árvore de busca. Nesse caso, cortes são inseridos para *reforçar* a formulação inserida inicialmente, ou seja, caso somente um `cuts_generator` seja informado é necessário que a formulação inicial seja completa; assim, caso a restrição (7) seja inserida dessa forma, as variáveis auxiliares $y$ e as restrições (fracas) de eliminação de sub-ciclos devem ser mantidas na formulação inicial visto que as restrições de grau não são suficientes para definir uma solução factível.

**`lazy_constrs_generator`** : um gerador de restrições informado com essa propriedade é chamado sempre que uma *solução inteira* for gerada. Nesse modo de uso é possível iniciar a busca com uma formulação incompleta, ou seja, sem as variáveis auxiliares $y$

e as restrições (4). Desse modo, a formulação inicial pode ficar muito mais leve. Uma possível desvantagem dessa abordagem é que como o pacote de otimização deve considerar que a formulação inicial está *incompleta*, algumas fases do pré-processamento do mesmo precisam ser desligadas e o limite inferior inicial pode ser menor.

Assim, uma vez que tenhamos um gerador de restrições implementado (especialização da classe `ConstrsGenerator`), podemos decidir em que situação o mesmo será chamado. É possível também especificar geradores de restrições para ambas as situações, ou seja, a formulação inicial é incompleta e portanto toda solução inteira deve ser checada mas soluções fracionárias também são verificadas. A melhor abordagem a ser utilizada deve ser determinada com experimentos para o problema de interesse. Na Seção 3 são incluídos experimentos para diferentes configurações do nosso código.

O código a seguir implementa um gerador dinâmico de restrições de eliminação de sub-rotas em soluções inteiras para nossa formulação compacta previamente implementada.

```
1  from mip.callbacks import ConstrsGenerator, CutPool
2  from typing import Set, List
3  from collections import defaultdict
4
5  def subtour(N: Set, outa: defaultdict, node) -> List:
6      """verifica se 'node' pertence a alguma sub-rota e retorna os
7      nós envolvidos"""
8      queue = [node]
9      visited = set(queue)
10     while queue:
11         n = queue.pop()
12         for nl in outa[n]:
13             if nl not in visited:
14                 queue.append(nl)
15                 visited.add(nl)
16
17     if len(visited) != len(N):
18         return [v for v in visited]
19     else:
20         return []
21
22 class SubTourLazyGenerator(ConstrsGenerator):
23     def __init__(self, xv):
24         self._x = xv
25
26     def generate_constrs(self, model: Model):
27         """restrições de eliminação de sub-rotas em soluções inteiras"""
28         x_, N = model.translate(self._x), range(len(self._x))
29         cp = CutPool()
30         outa = [[j for j in N if x_[i][j].x >= 0.99] for i in N]
31
32         for node in N:
33             S = set(subtour(N, outa, node))
34             if S:
35                 AS = [(i, j) for (i, j) in product(S, S) if i != j]
36                 cut = xsum(x_[i][j] for (i, j) in AS) <= len(S)-1
37                 cp.add(cut)
38         for cut in cp.cuts:
39             model += cut
```

Em soluções inteiras como a da Figura 2, a identificação de sub-rotas pode ser facilmente realizada executando-se uma busca em profundidade a partir de cada nó e verificando sua conectividade (função `subtour`). A existência ou não de um arco depende do valor da respectiva variável $x$ na solução inteira. Dentro do método `generate_constrs`, as linhas (25-28) consultam as variáveis do modelo (`model.vars`) pelo nome, identificando as varáveis `x` e os respectivos pontos de saída e chegada de cada arco que são então armazenados nos vetores `U` e `V`. Nesse ponto pode-se questionar se a matriz `x` previamente definida não poderia ser usada para a consulta das variáveis relacionadas aos arcos. Essa abordagem é desencorajada dentro do gerador de restrições pois alguns resolvedores criam um novo problema durante a busca, possivelmente removendo algumas variáveis na fase de pré-processamento e as referências originais podem se

tornar inválidas. Para cada nó, caso uma sub-rota desconectada seja identificada (linha 36) uma restrição do tipo (7) é gerada para o subconjunto `S` envolvido. Os cortes são temporariamente armazenados em um objeto do tipo `CutPool` pois o mesmo descarta cortes repetidos. Finalmente, nas linhas 42-43, os cortes são adicionados ao modelo.

Para que esse código seja executado durante a busca, antes de chamar a otimização do modelo no código anterior (linha 60), precisamos informar o gerador atribuindo um objeto desse tipo à propriedade `lazy_constrs_generator` com o código:

```
model.lazy_constrs_generator = SubTourLazyGenerator()
```

A versão completa do código que inclui a geração dinâmica de restrições de eliminação de sub-rotas em soluções inteiras pode ser obtida no repositório do Python-MIP. Um exemplo `tsp-lazy.py` está disponível no repositório Python-MIP em `www.python-mip.com`.

A geração de desigualdades que eliminam *soluções fracionárias* determina o problema conhecido como *separação de cortes*. No nosso caso queremos encontrar sub-conjuntos de vértices pouco conectados (não necessariamente desconectados) do restante da rota. Esse problema pode ser resolvido solucionando-se o problema do *corte mínimo* considerando o grafo da malha viária do problema com o valor das variáveis $x$ como capacidades nos arcos. Para nosso exemplo, utilizaremos a implementação do algoritmo de corte mínimo disponível no pacote Python `networkx` para resolver esse problema.

```
1  import networkx as nx
2
3  class SubTourCutGenerator(ConstrsGenerator):
4      def __init__(self, Fl: List[Tuple[int, int]]):
5          self.F = Fl
6
7      def generate_constrs(self, model: Model):
8          G = nx.DiGraph()
9          r = [(v, v.x) for v in model.vars if v.name.startswith('x(')]
10         U = [int(v.name.split('(')[1].split(',')[0]) for v, f in r]
11         V = [int(v.name.split(')')[0].split(',')[1]) for v, f in r]
12         cp = CutPool()
13         for i in range(len(U)):
14             G.add_edge(U[i], V[i], capacity=r[i][1])
15         for (u, v) in F:
16             if u not in U or v not in V:
17                 continue
18             val, (S, NS) = nx.minimum_cut(G, u, v)
19             if val <= 0.99:
20                 arcsInS = [(v, f) for i, (v, f) in enumerate(r)
21                            if U[i] in S and V[i] in S]
22                 if sum(f for v, f in arcsInS) >= (len(S)-1)+1e-4:
23                     cut = xsum(1.0*v for v, fm in arcsInS) <= len(S)-1
24                     cp.add(cut)
25                     if len(cp.cuts) > 256:
26                         for cut in cp.cuts:
27                             model += cut
28                         return
29         for cut in cp.cuts:
30             model += cut
31         return
```

Na criação de nosso gerador de cortes informamos uma lista `Fl` de pares $(i, j)$ de vértices cuja conectividade deve ser verificada em toda solução gerada. No método `generate_constrs` consultamos as variáveis do modelo (`model.vars`) e identificamos a qual arco cada variável se refere considerando o nome das variáveis (linhas 9-11), utilizando o seu valor na solução (propriedade `x`) para construção do grafo onde iremos procurar sub-rotas desconexas para geração das restrições (6). A descoberta de sub-rotas desconexas é feita nas linhas 18-19. Na linha 25, inserimos um critério de parada para a geração dos cortes: caso um número suficientemente grande já tenha sido gerado, inserimos os mesmos no modelo e prosseguimos a otimização sem procurar por cortes adicionais. Nesse ponto convém ressaltar a função dos cortes e sua relação com o

restante do modelo. Como a formulação compacta que estamos utilizando define completamente o problema, a adição de cortes é opcional, ou seja, somente feita para melhorar o desempenho do modelo. A inserção de um número muito grande de cortes por iteração pode gerar o efeito indesejado de perda de desempenho na resolução. Para utilizarmos nosso gerador de cortes com a formulação anterior basta atribuir à propriedade `cuts_generator` um objeto da nossa classe `SubTourCutGenerator` antes da otimização do modelo. O exemplo completo está disponível no repositório Python-MIP (exemplo tsp-cuts.py, www.python-mip.com).

### 2.4. Integração com Heurísticas

Pacotes de programação linear inteira iniciam o processo de resolução computando uma solução possivelmente fracionária, obtida através da relaxação do problema. Em instâncias difíceis, a obtenção da primeira solução *inteira* válida pode requerer a exploração de um grande número de nós na árvore de busca. Para essas instâncias, muitas vezes uma heurística simples e rápida pode ser utilizada para geração de uma solução inicial. No pacote Python-MIP, soluções iniciais podem ser facilmente informadas ao pacote de otimização através da propriedade `start` do modelo que recebe uma lista de pares `(x, v)` onde `x` é uma referência para uma variável de decisão e `v` o seu valor na solução factível.

O código abaixo exemplifica a utilização de um algoritmo construtivo e de uma meta-heurística de busca local para construção de uma solução inicial factível para o problema considerado.

```
1  seq = [0, max((c[0][j], j) for j in V)[1]] + [0]
2  Vout = V-set(seq)
3  while Vout:
4      (j, p) = min([(c[seq[p]][j] + c[j][seq[p+1]], (j, p)) for j, p in
5                    product(Vout, range(len(seq)-1))])[1]
6
7      seq = seq[:p+1]+[j]+seq[p+1:]
8      Vout = Vout - {j}
9
10
11 def delta(d: List[List[float]], S: List[int], p1: int, p2: int) -> float:
12     p1, p2 = min(p1, p2), max(p1, p2)
13     e1, e2 = S[p1], S[p2]
14     if p1 == p2:
15         return 0
16     elif abs(p1-p2) == 1:
17         return ((d[S[p1-1]][e2] + d[e2][e1] + d[e1][S[p2+1]])
18                 - (d[S[p1-1]][e1] + d[e1][e2] + d[e2][S[p2+1]]))
19     else:
20         return ((d[S[p1-1]][e2] + d[e2][S[p1+1]] +
21                  d[S[p2-1]][e1] + d[e1][S[p2+1]])
22                - (d[S[p1-1]][e1] + d[e1][S[p1+1]] +
23                  d[S[p2-1]][e2] + d[e2][S[p2+1]]))
24
25 L = [cost for i in range(50)]
26 sl, cur_cost, best = seq.copy(), cost, cost
27 for it in range(int(1e7)):
28     (i, j) = rnd.randint(1, len(sl)-2), rnd.randint(1, len(sl)-2)
29     dlt = delta(c, sl, i, j)
30     if cur_cost + dlt <= L[it % len(L)]:
31         sl[i], sl[j], cur_cost = sl[j], sl[i], cur_cost + dlt
32         if cur_cost < best:
33             seq, best = sl.copy(), cur_cost
34     L[it % len(L)] = cur_cost
35
36 m.start = [(x[seq[i]][seq[i+1]], 1) for i in range(len(seq)-1)]
```

Nosso algoritmo construtivo (linhas 3-8) é o algoritmo da inserção mais barata: iniciamos uma rota parcial incluindo arbitrariamente as ligações $(0, j)$ e $(j, 0)$ (linha 6), onde $j$ é o ponto mais distante do ponto inicial e prosseguimos aumentando a rota. Na inserção de um novo ponto verificamos o custo de inserir *cada vértice* ainda fora da rota (conjunto `Vout` em *cada posição* intermediária possível (`p`) e selecionamos a opção mais barata. A inserção de um novo ponto utiliza os recursos de fatiamento e adição de listas da linguagem Python (linha 7).

Para melhoria da solução inicial, utilizamos a meta-heurística baseada em busca local Late Acceptance Hill Climbing (Burke e Bykov, 2017) por sua simplicidade. O gargalo de métodos de busca local usualmente é a avaliação do custo da solução resultante da aplicação de dado movimento. Por isso, nas linhas 11-22 incluímos uma função que dada uma solução de entrada (sequência de cidades) `s` e duas posições dessa sequência, `p1` e `p2` calcula em *tempo constante* a variação de custo que será obtida. Dessa forma, podemos executar rapidamente um grande número de movimentos cuja aceitação é controlada pelo arcabouço da meta-heurística que é implementada nas linhas 25-33. Finalmente, informamos as variáveis de decisão relacionadas aos arcos existentes na melhor solução encontrada na linha 35.

## 3. Experimentos Computacionais

Para analisar a melhoria de desempenho que pode ser obtida com a geração dinâmica de restrições e com a integração com heurísticas, incluímos neste tutorial experimentos com diferentes versões de nosso método:

**MTZ** : método que utiliza a formulação compacta somente;

**MTZ+H** : método que utiliza a formulação compacta e a heurística para produção de uma solução inicial factível;

**MTZ+C** : método que utiliza a formulação compacta e a geração dinâmica de restrições para eliminação de soluções fracionárias (planos de corte);

**LAZY** : método em que somente as restrições de grau (2) e (3) são inicialmente informadas, enquanto as restrições de eliminação de sub-rotas são inseridas por demanda;

**LAZY+C+H** : método em que somente as restrições de grau (2) e (3) são inicialmente informadas, enquanto as restrições de eliminação de sub-rotas e planos de corte são inseridas por demanda e uma solução inicial inteira é informada.

Os testes utilizaram a versão 1.5.3 do pacote Python-MIP. Foram avaliados como motor de busca ou o pacote de otimização COIN-OR CBC, incluso no pacote, ou o pacote comercial Gurobi 8.1 (www.gurobi.com). Uma das vantagens do pacote Python-MIP é que nenhuma linha de código precisa ser modificada caso o usuário queira trocar de motor de busca: todas as funcionalidades são compatíveis com ambos os motores. Os experimentos foram executados em um computador com processador Intel Core i7-4960X 3.6 GHz com 32 Gb de RAM. Cada teste foi realizado de maneira sequencial, sendo que o pacote GNU Parallel (Tange, 2011) foi empregado para que os núcleos fossem utilizados em paralelo, executando diferentes testes. Um tempo limite de dez horas (36.000 segundos) foi estipulado para cada execução. Algumas instâncias com até 202 nós da coleção de problemas TSPLIB95 (Reinelt, 1995) foram selecionadas. Os resultados (tempos de execução em segundos) são apresentados na Tabela 1. O sufixo numérico no nome das instâncias indica o número de pontos que cada uma inclui. O melhor resultado para cada instância é enfatizado em negrito. Células com execuções que foram truncadas por tempo foram sombreadas.

Como podemos observar, instâncias com algumas centenas de nós são claramente difíceis de resolver utilizando a formulação fraca MTZ, independente do pacote de otimização utilizado. A adição por demanda de restrições de eliminação de sub-rotas (versões com +C e/ou LAZY) oferece ganhos expressivos, de modo similar para os dois pacotes de otimização. A aceleração média obtida para o CBC foi de mais de 80 vezes e para o Gurobi de mais de 60 vezes. Esse número é uma estimativa baixa da aceleração obtida, pois as execuções foram truncadas por tempo. Esses resultados ilustram a importância do estudo e implementação computacional de diferentes formulações para o tratamento de um problema de otimização combinatória. Leitores interessados no estado-da-arte em métodos de resolução para o problema do caixeiro viajante podem consultar o texto de Cook (2019).

Table 1: Resultados de experimentos computacionais para os métodos desenvolvidos utilizando os pacotes de otimização CBC e Gurobi.

| instância | Python-MIP 1.5.3 com CBC | | | | |
|---|---|---|---|---|---|
| | MTZ | MTZ+H | MTZ+C | LAZY | LAZY+C+H |
| ulysses22 | 778,0 | 1.292,5 | 7,4 | 36.000,0 | 3,3 |
| att48 | 13.578,6 | 12.749,2 | 14,8 | 14.546,7 | **5,9** |
| bier127 | 36.000,0 | 36.000,0 | 2.637,6 | 36.000,0 | 136,3 |
| gr202 | 36.000,0 | 36.000,0 | 2.613,0 | 36.000,0 | **867,9** |
| *média* | 17.271,3 | 17.208,3 | 1.054,6 | 24.509,3 | **202,7** |

| instância | Python-MIP 1.5.3 com GUROBI | | | | |
|---|---|---|---|---|---|
| | MTZ | MTZ+H | MTZ+C | LAZY | LAZY+C+H |
| ulysses22 | 148,1 | 214,1 | **1,7** | 66,5 | 3,1 |
| att48 | 186,5 | 185,7 | 6,5 | 257,7 | 6,4 |
| bier127 | 36.000,0 | 25.508,8 | 209,3 | 884,0 | **79,7** |
| gr202 | 36.000,0 | 36.000,0 | 1.057,2 | 36.000,0 | 1.084,8 |
| *média* | 14.466,9 | 12.381,7 | 254,9 | 7.441,6 | 234,8 |

Fonte: Elaborada pelos autores.

## 4. Conclusões

Neste tutorial, foram desenvolvidos e avaliados computacionalmente diferentes métodos exatos para o clássico Problema do Caixeiro Viajante utilizando o pacote Python-MIP. Experimentos computacionais foram realizados demonstrando os ganhos expressivos que podem ser obtidos com melhorias relativamente simples do método inicial. As técnicas aqui aplicadas são facilmente adaptáveis a problemas similares de otimização combinatória, para os quais diferentes formulações estão disponíveis. Nesse sentido, enfatizamos a vantagem de utilização da linguagem de alto nível de modelagem do pacote Python-MIP para acelerar o desenvolvimento de métodos de alto desempenho para problemas de otimização combinatória. Ressaltamos, por fim, que o pacote tem uma extensa documentação disponível, incluindo vários outros exemplos de modelagem, disponíveis em www.python-mip.com.

***Agradecimentos.*** Os autores agradecem o apoio do grupo CODeS (Combinatorial Optimization and Decision Support) da KU Leven pela bolsa de pesquisador sênior disponibilizada ao Prof. Haroldo no período de 2018-2019, ao CNPq e a FAPEMIG.

## References

Applegate, D. L., Bixby, R. E., Chvatal, V. e Cook, W. J. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

Bixby, R. E. Solving real-world linear programs: A decade and more of progress. *Operations Research*, v. 50, n. 1, p. 3–15, 2002.

Burke, E. K. e Bykov, Y. The late acceptance Hill-Climbing heuristic. *European Journal of Operational Research*, v. 258, n.1, p. 70–78, 2017.

Castellucci, P. B. Julia e JuMP: Novas ferramentas para programação matemática. *Pesquisa Operacional para o Desenvolvimento*, v. 9, n.2, p. 48–61, 2017.

Cook, W. Computing in Combinatorial Optimization. In: Steffen B. e Woeginger, G. (ee.) *Computing and Software Science.* Lecture Notes in Computer Science, vol 10000. Springer, Cham, 2019.

Dantzig, G., Fulkerson, R. e Johnson, S. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, v. 2, n. 4, p. 393–410, 1954.

Dunning, I., Huchette, J. e Lubin, M. JuMP: A modeling language for mathematical optimization. *SIAM Review*, v. 59, n. 2, p. 295–320, 2017.

Forrest, J. e Lougee-Heimer, R. CBC user guide. In: *INFORMS Tutorials in Operations Research*, p. 257-277. 2005.

Fourer, R., Gay, D. e Kernighan, B. *AMPL: A mathematical programming language*. Murray Hill: AT & T Bell Laboratories, 1987.

Miller, C. E., Tucker, A. W. e Zemlin, R. A. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, v. 7, n. 4, p. 326–329, 1960.

Mitchell, S. An introduction to PuLP for python programmers. *Python Papers Monograph*, v. 1, n. 14, 2009.

Reinelt, G. *TSPLIB95*. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*, v. 338, 1995.

Ritchie, D. M., Johnson, S. C., Lesk, M. E. e Kernighan, B. W. Unix time-sharing system: The C programming language. *The Bell System Technical Journal*, v. 57, n. 6, p. 1991–2019, 1978.

Tange, O. GNU parallel: the command-line power tool. *The USENIX Magazine*, v. 36, n. 1, p. 42–47, 2011.

The Economist. *Python is becoming the world's most popular coding language.* 2018. Disponível em: www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language. Acesso em : 17/12/2019.

Towhidi, M. e Orban, D. Customizing the solution process of COIN-OR's linear solvers with Python. *Mathematical Programming Computation*, v. 8, n. 4, p. 377–391, 2016.

Wolsey, L. A. *Integer Programming*. Wiley-Interscience, 1998.