Pesquisa Operacional para o Desenvolvimento
Vol. XX, n. YY, pp. ZZ-ZZ, ANO: XXXX

ISSN 1984-3534

DESENVOLVENDO RESOLVEDORES 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, Ouro Preto, MG, Brasil

Recebido xx/xx/xxxx, aceito xx/xx/xxxx

RESUMO

O pacote Python-MIP oferece um conjunto abrangente ferramentas para a 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 resolvedores avançados, habilitando comunicação bidirecional com o resolvedor durante o processo de busca. Neste tutorial desenvolveremos resolvedores de Programação Linear Inteira Mista para o Problema do Caixeiro Viajante. Iniciando com um resolvedor simples baseado em uma formulação compacta iremos evoluir para um resolvedor que combina heurísticas e planos de corte para a resolução eficaz de problemas de grande porte.

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 di-directional 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 effective solution of large scale instances.

Keywords: Combinatorial Optimization, Traveling Salesman Problem, Integer Linear Programming, Python.

* Autor para correspondência. E-mail: haroldo@ufop.edu.br
DOI: xx.xxxx/PODes.xxxx.xxx

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

Python-MIP é um pacote para modelagem e resolução de Problemas de Programação Linear Inteira Mista (PLIM) (Wolsey, 1998) em Python. O projeto do pacote foi feito com o objetivo de desenvolver uma ferramenta que atendesse os seguintes requisitos:

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

Tradicionalmente, os objetivos 1 e 2 foram considerados conflitantes. Até recentemente, as opções mais recomendada para os interessados em 1 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). Resolvedores 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) demonstraram que os objetivos 1 e 2 não são necessariamente conflitantes: linguagens de alto nível como Julia juntamente com compiladores *just-in-time* permitem o desenvolvimento rápido de resolvedores que apresentam alto desempenho. O objetivo do projeto Python-MIP é o desenvolvimento de um pacote de Programação Linear Inteira Mista para a linguagem Python que atenda plenamente os requisitos 1-3.

Pesquisas recentes mostram que Python está se tornando a linguagem mais popular da atualidade (pyt, 2018). O projeto Python-MIP foi primariamente 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 resolvedores comerciais e de código aberto. Recursos que requerem uma integração maior com o resolvedor, 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) e somente suporta o resolvedor 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 desenvolveremos versões sucessivamente mais sofisticadas de um resolvedor para o clássico problema do Caixeiro Viajante (G. Dantzig e Johnson, 1954; Miller et al., 1960; Applegate et al., 2006) no pacote Python-MIP. Enquanto na primeira versão a comunicação com o resolver somente ocorre no momento em que o modelo criado é informado e na coleta dos resultados, a versão final utiliza comunicação bidirecional com o resolvedor durante o processo de busca para o tratamento de uma formulação com um número exponencial de restrições em um método de *branch-&-cut*. Resultados experimentais são apresentados na seção XXX para demonstrar os ganhos substanciais de desempenho que podem ser obtidos com essa última versão.

1. 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 conjunto de vértices numerados sequencialmente a partir de 0

$c_{(i,j)}$ custo de percorrer a ligação $(i, j) \in V \times V$

Figura 1: 14 cidades turísticas da Bélgica

Em todas as formulações que serão apresentadas, utilizaremos as seguintes variáveis binárias de decisão que representam a escolha dos arcos que compõe 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 algumas cidades turísticas da Bélgica.

1.1. 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 restrições. Formulações desse tipo, apesar de usualmente não serem a melhor opção de resolução em termos de desempenho para problemas deste tipo, são convenientes para uma primeira abordagem pois podem ser facilmente inseridas de uma vez só como entrada para um software resolvedor. A formulação abaixo foi proposta por Miller et al. (1960):

Minimize:

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

Sujeito a:

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

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

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

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

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

As equações (2) e (3) 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

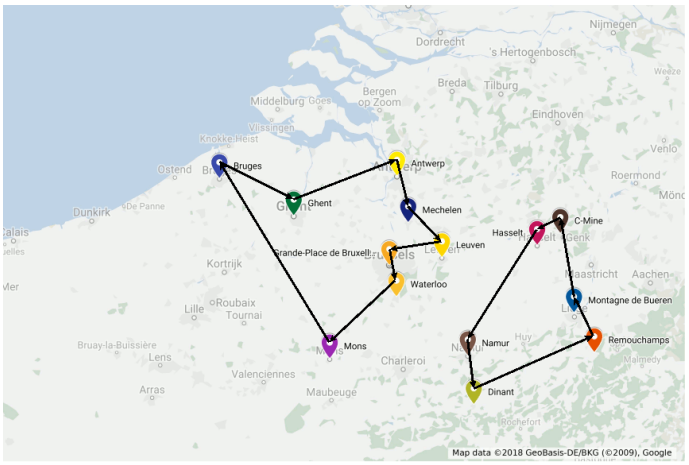

Figura 2: Rotas desconectadas da origem

(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 caso somente as restrições (2) e (3) fossem consideradas.

A seguir temos um exemplo completo de um resolvedor em Python-MIP para o problema do caixeiro viajante para o mapa Figura 1, onde o resolvedor utilizará a formulação compacta descrita anteriormente:

```
from itertools import product
from sys import stdout as out
from mip import Model, xsum, minimize, BINARY

# locais a visitar
places = ['Antwerp', 'Bruges', 'C–Mine', 'Dinant', 'Ghent',
          'Grand–Place de Bruxelles', 'Hasselt', 'Leuven',
          'Mechelen', 'Mons', 'Montagne de Bueren', 'Namur',
          'Remouchamps', 'Waterloo']

# matriz triangular superior com tempos de deslocamento
dists = [[83, 81, 113, 52, 42, 73, 44, 23, 91, 105, 90, 124, 57],
         [161, 160, 39, 89, 151, 110, 90, 99, 177, 143, 193, 100],
         [90, 125, 82, 13, 57, 71, 123, 38, 72, 59, 82],
         [123, 77, 81, 71, 91, 72, 64, 24, 62, 63],
         [51, 114, 72, 54, 69, 139, 105, 155, 62],
         [70, 25, 22, 52, 90, 56, 105, 16],
         [45, 61, 111, 36, 61, 57, 70],
         [23, 71, 67, 48, 85, 29],
         [74, 89, 69, 107, 36],
         [117, 65, 125, 43],
         [54, 22, 84],
         [60, 44],
         [97],
         []]

# nr. de pontos e conjunto de pontos
n, V = len(dists), set(range(len(dists)))

# matriz com tempos
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 0/1 indicando se um arco (i,j) participa da rota ou não
39  x = [[model.add_var(var_type=BINARY) for j in V] for i in V]
40
41  # variáveis auxiliares
42  y = [model.add_var() for i in V]
43
44  # função objetivo: minimizar tempo
45  model.objective = minimize(xsum(c[i][j]*x[i][j] for i in V for j in V))
46
47  # restrição: selecionar arco de saída da cidade
48  for i in V:
49      model += xsum(x[i][j] for j in V - {i}) == 1
50
51  # restrição: selecionar arco de entrada na cidade
52  for i in V:
53      model += xsum(x[j][i] for j in V - {i}) == 1
54
55  # eliminação de sub-rotas
56  for (i, j) in product(V - {0}, V - {0}):
57      model += y[i] - (n+1)*x[i][j] >= y[j]-n
58
59  # chamada da otimização com limite tempo de 30 segundos
60  model.optimize(max_seconds=30)
61
62  # verificando se ao menos uma solução foi encontrada e a imprimindo
63  if model.num_solutions:
64      out.write('route with total distance %g found: %s'
65                % (model.objective_value, places[0]))
66      nc = 0
67      while True:
68          nc = [i for i in V if x[nc][i].x >= 0.99][0]
69          out.write(' -> %s' % places[nc])
70          if nc == 0:
71              break
72      out.write('\n')
```

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 em nosso exemplo consideramos que a distância de ida e volta é igual. Convertemos a matriz triangular `dists` em uma matriz completa nas linhas 31-34.

A linha 36 cria o modelo de programação linear inteira. As variáveis do modelo são criadas nas linhas 39 e 42 utilizando o método `add_var` em nosso modelo `m`. Durante a criação das restrições, será necessário referenciar as variáveis criadas. Por isso, utilizamos a matriz `x` para mapear cada arco do grafo com sua respectiva variável binária e vetor `y` para mapear cada nó com sua respectiva variável auxiliar contínua para eliminação de sub-rotas.

A função objetivo que minimiza o tempo total dos arcos selecionados é informada na linha 45. Nesse caso, para cada arco multiplicamos sua respectiva variável binária de seleção pelo tempoo de deslocamento do arco armazenado em `c`.

As restrições são criadas nas linhas 48-57. 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 60 dispara a otimização do modelo. Na linha 63 verificamos se uma solução viável foi encontrada e se positivo a escrevemos nas linhas 64-72.

Os resolvedores 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 resolvedores de programação linear inteira em sua resolução é bastante pobre.

Um desempenho muito melhor pode ser obtido com a inserção das seguintes desigualdades:

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

O problema com as desigualdades acima é que elas devem ser geradas para cada subconjunto S de vértices do grafo, ou seja, para uma 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 o método dos planos de corte (G. Dantzig e Johnson, 1954), onde somente as restrições *necessárias* são inseridas. 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.

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

`cuts_generator` : caso o gerador de restrições seja informado nessa propriedade do modelo, a verificação por restrições faltantes (cortes violados) será 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.

`lazy_constrs_generator` : um gerador de restrições informado nessa propriedade será 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 desta abordagem é que como o resolvedor 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.

Uma vez que tenhamos um gerador de restrições implementado, 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 será incompleta e portanto toda solução inteira deve ser checada mas soluções fracionárias também serão checadas. A melhor abordagem a ser utilizada deve ser determinada com experimentos no problema de interesse.

1.2. Geração Dinâmica de Restrições

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.

O código abaixo 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, out: defaultdict, node) -> List:
6      """verifica se 'node' pertence a uma sub-rota.
7      se positivo retorna os elementos dessa sub-rota"""
8      queue = [node]
9      visited = set(queue)
10     while queue:
11         n = queue.pop()
12         for nl in out[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     """Gera restricões de eliminacão de sub-rotas em solucões inteiras"""
24     def generate_constrs(self, model: Model):
25         r = [(v, v.x) for v in model.vars
26              if v.name.startswith('x(') and v.x >= 0.99]
27         U = [int(v.name.split('(')[1].split(',')[0]) for v, f in r]
28         V = [int(v.name.split(')')[0].split(',')[1]) for v, f in r]
29         N, cp = set(U+V), CutPool()
30         # output nodes for each node
31         out = defaultdict(lambda: list())
32         for i in range(len(U)):
33             out[U[i]].append(V[i])
34
35         for n in N:
36             S = set(subtour(N, out, n))
37             if S:
38                 arcsInS = [(v, f) for i, (v, f) in enumerate(r)
39                            if U[i] in S and V[i] in S]
40                 if sum(f for v, f in arcsInS) >= (len(S)-1)+1e-4:
41                     cut = xsum(1.0*v for v, fm in arcsInS) <= len(S)-1
42                     cp.add(cut)
43         for cut in cp.cuts:
44             model += cut
```

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()
```

```
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 checada 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 22 e 23), utilizando o seu valor na solução (propriedade `x`) para construção do grafo onde iremos procurar sub-rotas desconexas para geração das restrições (6). A identificação dessas sub-rotas é feita resolvendo-se o problema do corte mínimo, com o algoritmo que está disponível no pacote `networkx` (linha 18). A descoberta de sub-rotas desconexas (linhas 19) gera a restrição de restrições (cortes) que são primeiramente armazenados em um conjunto (linha 24) e posteriormente adicionados ao modelo. Separamos esses dois passos pois restrições repetidas podem ser geradas. 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 utilizamos 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.

1.3. Integração com heurísticas

Resolvedores 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 resolvedor 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 demonstra a utilização de um algoritmo construtivo e de uma metaheurística de busca local para construção de uma solução inicial factível para nosso modelo.

```
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 (
21         (d[S[p1-1]][e2] + d[e2][S[p1+1]] + d[S[p2-1]][e1] + d[e1][S[p2+1]])
22         - (d[S[p1-1]][e1] + d[e1][S[p1+1]] + d[S[p2-1]][e2] + d[e2][S[p2+1]]))
23
24 L = [cost for i in range(50)]
25 sl, cur_cost, best = seq.copy(), cost, cost
26 for it in range(int(1e7)):
27     (i, j) = rnd.randint(1, len(sl)-2), rnd.randint(1, len(sl)-2)
28     dlt = delta(c, sl, i, j)
29     if cur_cost + dlt <= L[it % len(L)]:
30         sl[i], sl[j], cur_cost = sl[j], sl[i], cur_cost + dlt
31         if cur_cost < best:
32             seq, best = sl.copy(), cur_cost
33     L[it % len(L)] = cur_cost
34
35 m.start = [(x[seq[i]][seq[i+1]], 1) for i in range(len(seq)-1)]
```

Nosso algoritmo construtivo (linhas 262-269) será o algoritmo da inserção mais barata: dada uma rota que não inclua todos as cidades, na inserção de uma nova cidade verificamos o custo de inserir cada cidade ainda fora da rota (elemento `j` conjunto `Vout` em cada posição intermediária possível (`p`) e selecionamos a opção mais barata.

Para melhoria da nossa solução inicial utilizaremos a metaheurística baseada em busca local Late Acceptance Hill Climbing (**?**) 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 metaheurística que é implementada nas linhas 25-33. Finalmente, informamos as variáveis de decisão relacionadas aos arcos existentes na

melhor solução encontrada na linha 35.

Referências

Python is becoming the world's most popular coding language. *The Economist*, v. , 2018. https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language.

Applegate, D. L., Bixby, R. E., Chvatal, V., e Cook, W. J. *The Traveling Salesman Problem: A Computational Study*. *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. ISSN 0030364X.

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. *INFORMS Tutorials in Operations Research.*, v. p. 257–277, 2005.

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

G. Dantzig, R. F. 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.

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. ISSN 1557735X.

Mitchell, S. An Introduction to PuLP for Python Programmers. *Python Papers Monograph*, v. 1, n. 14, 2009.

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.

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. ISSN 18672957.

Wolsey, L. A. *Integer Programming*. *Integer Programming*: Wiley-Interscience. ISBN 0471283665, 1998.