

DESENVOLVENDO RESOLVEDORES DE PROGRAMAÇÃO LINEAR INTEIRA MISTA EM PYTHON USANDO O PACOTE PYTHON-MIP¹

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 resolvidor durante o processo de busca. Neste tutorial desenvolveremos resolvedores de Programação Linear Inteira Mista para o Problema do Caixeiro Viajante. Iniciando com um resolvidor simples baseado em uma formulação compacta iremos evoluir para um resolvidor que combina heurísticas e planos de corte para a resolução eficaz 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 effective solution of larger 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 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 2 para demonstrar os ganhos substanciais de desempenho que podem ser obtidos com essa última versão.

A instalação do pacote é bastante simples. Depois da instalação da linguagem Python, o gerenciador de pacotes da mesma (pip) pode ser invocado para instalação. Para isso, pode-se usar o seguinte comando no terminal do sistema:

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

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:



Figura 1: 14 cidades turísticas da Bélgica

V conjunto de vértices numerados sequencialmente a partir de 0

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

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 resolvidor. A formulação abaixo foi proposta por Miller et al. (1960):

Minimize:

$$\sum_{i \in I, j \in I} c_{i,j} \cdot x_{i,j} \quad (1)$$

Sujeito a:

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

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

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

$$x_{i,j} \in \{0, 1\} \quad \forall i \in V, j \in V \quad (5)$$

$$y_i \geq 0 \quad \forall i \in V \quad (6)$$

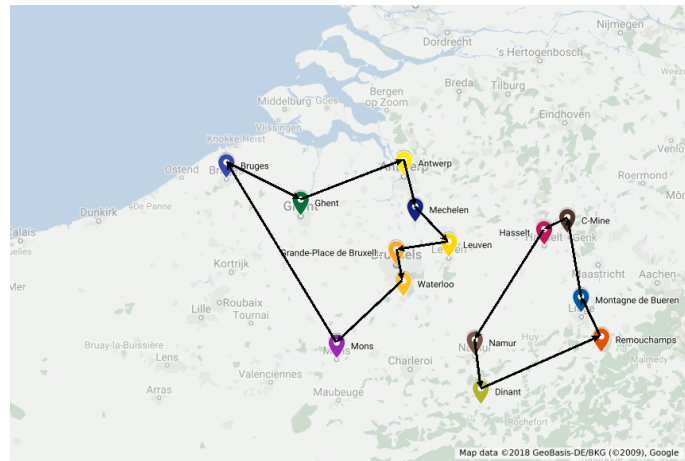


Figura 2: Rotas desconectadas da origem

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

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

```

1  from itertools import product
2  from sys import stdout as out
3  from mip import Model, xsum, minimize, BINARY
4
5  # locais a visitar
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 # matriz triangular superior com tempos de deslocamento
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 # nr. de pontos e conjunto de pontos
28 n, V = len(dists), set(range(len(dists)))

```

```

29
30 # matriz com tempos
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 0/1 indicando se um arco (i,j) participa da rota ou não
39 x = [[model.add_var(var_type=BINARY, name='x(%d,%d)' % (i, j)) 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 time %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 `model`. 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 binária¹ e o 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 de percorrimento da rota é informada na

¹Se um grafo esparso fosse informado poderíamos utilizar um vetor de dicionários para indicar os arcos de saída de cada nó.

linha 45. Nesse caso, para cada arco multiplicamos sua respectiva variável binária de seleção pelo tempo 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 $\times\text{sum}$. 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. Essa deficiência não é visível quando resolvemos problemas pequenos como o de exemplo (apenas 14 cidades), mas faz com que o desempenho do resolvedor degenere muito rapidamente a medida que o tamanho do grafo de entrada aumenta. Na seção seguinte veremos o tratamento de uma formulação mais apropriada.

1.2. Tratamento de Formulações Fortes com Geração Dinâmica de Restrições

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

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

O problema com as 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. Inserir-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 caso a solução corrente não as satisfaça.

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 dessa abordagem é que como o resolvidor 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 será incompleta e portanto toda solução inteira deve ser checada mas soluções fracionárias também serão cheçadas. A melhor abordagem a ser utilizada deve ser determinada com experimentos no problema de interesse. Na seção (2) são incluídos experimentos para diferentes configurações do nosso código.

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 restrições de eliminação de sub-rotas em soluções 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         out = defaultdict(lambda: list())
31         for i in range(len(U)):
32             out[U[i]].append(V[i])
33
34         for n in N:
35             S = set(subtour(N, out, n))
36             if S:
37                 arcsInS = [(v, f) for i, (v, f) in enumerate(r)
38                           if U[i] in S and V[i] in S]
39                 cut = xsum(1.0*v for v, fm in arcsInS) <= len(S)-1
40                 cp.add(cut)
41         for cut in cp.cuts:
42             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 variáveis 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-rotas 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 baixada no seguinte endereço: <https://raw.githubusercontent.com/coin-or/python-mip/master/docs/podes/tsp-lazy.py>.

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. Em 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, F1: List[Tuple[int, int]]):
5         self.F = F1
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 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 na seguinte ligação: <https://raw.githubusercontent.com/coin-or/python-mip/master/docs/podes/tsp-cuts.py>.

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 3-8) será 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 V_{out} 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 nossa solução inicial utilizaremos a metaheurística 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 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.

2. Experimentos Computacionais

Para demonstrar a melhoria de desempenho que pode ser obtida com a geração dinâmica de restrições e com a integração com heurísticas, incluímos experimentos com diferentes versões de nosso algoritmo:

MTZ : resolvidor que somente utiliza a formulação compacta;

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

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

LAZY : resolvidor onde somente as restrições de grau (2) e (3) são inicialmente informadas e restrições de eliminação de sub-rotas são inseridas por demanda;

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

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
<i>média</i>	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
<i>média</i>	14.466,9	12.381,7	254,9	7.441,6	234,8

Tabela 1: Resultados de experimentos com diferentes configurações do resolvidor utilizando CBC e Gurobi

Os testes utilizaram a versão 1.5.3 do pacote Python-MIP. Foram avaliados como motor de busca ou o resolvidor COIN-OR CBC, incluso no pacote, ou o resolvidor comercial Gurobi 8.1². 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 executaram em um computador com processador Intel ® Core i7-4960X 3.6 GHz com 32 Gb de RAM. Cada teste executou de maneira sequencial, sendo que o pacote GNU Parallel (Tange, 2011) foi utilizado para que 2 núcleos fossem utilizados em paralelo em 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 2. 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 pode-se observar, instâncias com algumas centenas de nós são claramente difíceis de resolver utilizando a formulação fraca MTZ, independente do resolvidor utilizado. A adição por demanda de restrições de eliminação de sub-rotas (versões com +C e/ou LAZY) oferece ganhos expressivos, de modo similar nos dois resolvidores. A aceleração média obtida com o resolvidor CBC foi de mais de 80 vezes e no resolvidor 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).

3. Conclusões

Neste tutorial foram desenvolvidos e avaliados computacionalmente diferentes resolvidores exatos para o problema clássico do caixeiro viajante utilizando o pacote Python-MIP. Experimentos computacionais foram realizados demonstrando os ganhos expressivos que podem ser obtidos com

²<https://www.gurobi.com/>

adições relativamente simples ao resolvedor inicial. As técnicas aqui aplicadas são facilmente adaptáveis a problemas similares de otimização combinatória, onde diferentes formulações estão disponíveis. Nesse sentido, enfatiza-se a vantagem de utilização da linguagem de alto nível de modelagem do pacote Python-MIP para acelerar o desenvolvimento de resolvedores de alto desempenho para problemas de otimização combinatória. O pacote tem uma extensa documentação disponível³, incluindo vários exemplos de modelagem.

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.

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

Cook, W. *Computing in Combinatorial Optimization. Computing in Combinatorial Optimization*. Springer International Publishing, 2019.

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.

Reinelt, G. *TSPLIB 95*. D-69120, Universität Heidelberg, 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. 43–47, 2011.

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.

³<https://python-mip.readthedocs.io/en/latest/>

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