

Relatório de Iniciação Científica

Otimização linear: aplicações e modelagem
matemática.

Maryanna Conceição Silva

email: maryanna.silva@unesp.br

Universidade Estadual Paulista

“Júlio de Mesquita Filho”

- Câmpus São José do Rio Preto

Relatório de Iniciação Científica

Relatório de Iniciação Científica em matemática aplicada desenvolvido da forma online devido a pandemia do coronavírus(COVID-19).

Docente orientadora: Profa. Dra. Maria do Socorro Nogueira Rangel.

Sumário

1	Introdução	2
2	Modelos de Otimização	3
3	Sistema de Resolução GUROBI.	14
4	Resolvendo um problema da dieta pelo GUROBI.	18
5	O Problema da Mochila.	22
6	O Problema do Caixeiro Viajante.	32
7	Conclusão	45
	Appendices	49
	Apêndice A <i>Python</i>	49
A.1	<i>Print e Import</i>	49
A.2	Condições: <i>If</i> e <i>else</i>	49
A.3	Estrutura de repetição: <i>for</i>	49
A.4	Variáveis compostas: <i>list()</i> e <i>disc()</i>	50
A.5	Função: <i>def</i>	50

Lista de Figuras

1	Construção de um modelo matemático. Adaptado de [2]	3
2	Representação gráfica da reta $4x_1 + 3x_2 = 240$	10
3	Representação gráfica da inequação $4x_1 + 3x_2 \leq 240$	11
4	Região factível do modelo matemático do Exemplo 2.2	12
5	Modelo matemático do Exemplo 2.2 com o Vetor Gradiente.	12
6	Região factível do Exemplo 2.2 com o Vetor Gradiente e com a curva de nível $k = 500$	13
7	Ponto ótimo (P) do Exemplo 2.2.	13
8	Erro na instalação do Gurobi.	15
9	Exemplo 2.1(Problema da Fábrica de Trailers) escrito na sintaxe Python-GUROBI.	16
10	Detalhes do processo de solução em Python-Gurobi: Exemplo 2.1(Pro- blema da Fabrica de Trailers)	17
11	Comando para imprimir e a solução Exemplo 2.1(Problema da Fábrica de Trailers)	18
12	Implementação em Python-Gurobi do Problema da Dieta.	22
13	Resultado do Exemplo 4.1(Problema da Dieta)	22
14	Dados do Exemplo 5.1 na sintaxe Python-Gurobi.	26
15	Implementação do modelo do Problema da Mochila em Python- Gurobi.	28
16	Solução do Exemplo 5.1 em Python-Gurobi.	28
17	Instância do Exemplo 5.1.	29
18	A função <code>le_dados</code> para ler instâncias do Problema da Mochila.	30
19	Função <code>solve_ProblemaMochila()</code>	31
20	Programa principal para resolver o Problema da Mochila	31
21	Solução do Exemplo 5.1 em Python-Gurobi.	31
22	Exemplo 6.1(Problema do Caixeiro Viajante).	34
23	Grafo do Exemplo 6.1	34
24	Instância do Exemplo 6.1(Problema do Caixeiro Viajante).	36
25	Função <code>ler_dados</code> para o Exemplo 6.1(PVC) em Python-Gurobi.	37

26	Função <code>solve_CaixeiroViajante</code> com o comando para a matriz binária do Exemplo 6.1(PVC).	39
27	Resolução do Exemplo 6.1(PCV) em Gurobi(Passo 4).	39
28	Solução do Exemplo 6.1(PCV) em Python-Gurobi.	39
29	Sub-rotas do Exemplo 6.1.	40
30	Reformulação do Passo 3 - incluindo as restrições para eliminar sub-rotas	44
31	Ciclo hamiltoniano ótimo para o Exemplo 6.1(PCV) obtido usando o Python-Gurobi.	44
32	O melhor percurso para o Ramon seguir.	45
33	Comando <code>print()</code>	49

Lista de Tabelas

1	Dados dos três trailers	6
2	Disponibilidade, custo e quantidade de matéria-prima necessários para fazer as bolsas	8
3	Informações nutricionais dos alimentos da Dieta	19
4	Peso e importância dos itens	23

Lista de Quadros

1	Distância entre cidades 6.1(Problema do Caixeiro Viajante)	33
---	--	----

1 Introdução

A Otimização é uma área da matemática que envolve minimizar ou maximizar uma função que representa crítico de tomada de decisão de um determinado problema ou situação, esta função é chamada de função objetiva[3]. Os modelos matemáticos de otimização linear que estudamos no desenrolar desta pesquisa, envolvem três elementos: as variáveis de decisões, a função objetiva e as restrições de um problema[22].

O objetivo deste projeto é apresentar o Gurobi um solucionador de otimização, fundado por Zonghao **Gu**, Edward **Rothberg** e Robert **Bixby**, o nome Gurobi é inspirado no sobrenome dos fundadores. Ele pode ser utilizado para ajudar a encontrar soluções de vários problemas clássicos de Otimização, como o Problema da Mistura, o Problema Mochila e Problema do Caixeiro Viajante, tais problemas serão discutidos no decorrer do estudo. A metodologia usada nesta pesquisa consiste em apresentar, compreender alguns problemas e também encontrar e demonstrar como obter a soluções pelo o sistema de resolução Gurobi. A linguagem de programação Python será usada como interface com o sistema. Com o desenvolvimento deste projeto tivemos um entendimento mais amplo da área de Pesquisa Operacional e principalmente a área de Otimização.

2 Modelos de Otimização

Os modelos de otimização podem ser usados em várias áreas da indústria, pois é um modo a mais de melhorar o desempenho e os processos das empresas. Podemos dizer que os modelos de otimização linear ou otimização linear inteira representam um problema ou situação usando várias ferramentas da matemática e estas ferramentas são as funções, equações, inequações e dependências lógicas[22].

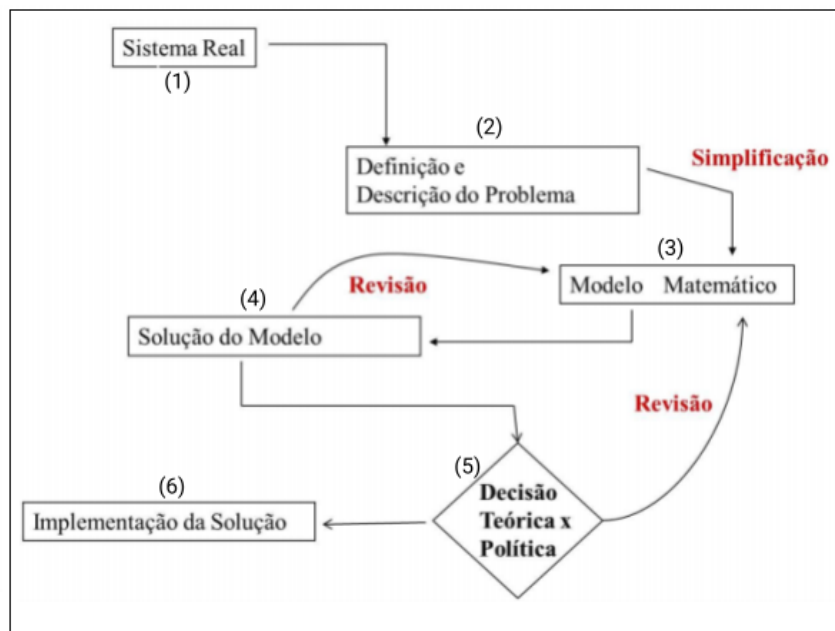


Figura 1: Construção de um modelo matemático. Adaptado de [2]

O processo para resolução de problemas pela pesquisa operacional é dividido em várias fases ou etapas [3], como destacado na Figura 1 (existem outros modelos disponíveis, por exemplo [3, 10]). As empresas ou a indústria apresenta os dados, a equipe responsável tem que analisar e estudar qual é o problema a ser resolvido, etapa (1). Na etapa (2) definimos e descrevemos o problema a ser resolvido e a finalidade deste problema, geralmente o problema é simplificado para chegar num modelo matemático na etapa (3). Neste momento com o modelo definido podemos com o auxílio de softwares ou algoritmos, por exemplo o algoritmo do método simplex, chegar na solução de uma instância do modelo que é a etapa (4), porém nem sempre a solução é satisfatória, por isso pode ser necessário voltar

para etapa (3). O processo de Decisão teórica ou política, na etapa (5), precisa verificar se o modelo proposto está representando adequadamente o problema e talvez seja necessário adicionar novas restrições. Neste caso, será necessário voltar para a etapa (3). Depois de todas análises, simplificações, reformulações e revisões seguiremos para a implementação da solução na prática na etapa (6) ([22, 3, 2]).

Para começar a construir um modelo é necessário identificar alguns elementos.

- i) *Elementos conhecidos*: o que temos;
- ii) *Elementos desconhecidos*: o que queremos determinar, qual a decisão a ser tomada;
- iii) *Função Objetiva*: qual é o critério para tomar a decisão;
- iv) *Restrições*: qual são as restrições que limitam a tomada de decisão.

Os *elementos conhecidos* podem ser definidos como o conhecimento prévio ou as informações que já temos sobre o problema e os *elementos desconhecidos* seriam as informações que queremos obter ao resolver o problema. Estes elementos são importantes na resolução, pois são deles que tiramos quais constantes, incógnitas e funções que representaram o problema. As incógnitas são nomeadas de “variáveis de decisão”. Para reconhecer esses elementos é necessário muito empenho dos indivíduos e geralmente este empenho é mais coletivo do que individual [22].

Quando falamos em variáveis usualmente associamos a diversos valores, porém precisaremos definir critério para optar pela alternativa mais adequada e esse critério é descrito como uma função das variáveis de decisão que será chamada de *Função Objetiva*. Entretanto, para o desenvolvimento da construção do modelo que representa o problema é importante esclarecer quais são os impedimentos para a tomada de decisão, pois são eles que definem as equações e inequações que formam o conjunto de *Restrições* do modelo.

Existem algumas classes de modelos de otimização:

- Modelos de Otimização Linear Contínua;
- Modelos de Otimização Inteira;

- Modelos de Otimização Inteira Mista;
- Modelos de Otimização Não linear.

O que diferencia estas classes de modelos é o domínio das variáveis, por exemplo Modelos de Otimização Linear Inteira, na qual as variáveis pertencem ao conjunto dos números inteiros e o tipo de função que representam a função objetiva e as restrições. Neste estudo vamos utilizar os *Modelos de Otimização Linear Contínua e Inteira*.

Definição 2.1 (Modelo de Otimização Linear Contínuo). *A forma do problema de otimização linear apresentada em (2.1)-(2.3) é chamada de forma padrão.*

$$\text{Min } f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n \quad (2.1)$$

S.a:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{m1}x_{m1} + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \quad (2.2)$$

$$x_1 \geq 0, x_2 \geq 0, \dots, x_n \geq 0 \quad (2.3)$$

A função linear f (2.1) é chamada de *função objetiva*, que pode ser para maximizar ou minimizar dependendo do problema. As equações (2.2) são as *restrições do problema* e por fim temos a (2.3) que são as *condições de não negatividade*[3]. O Modelo de Otimização Linear Contínuo também pode ser apresentado de forma matricial como apresentado em 2.4.

$$\begin{aligned} \text{Min } f(\mathbf{x}) &= \mathbf{c}^T \mathbf{x} \\ \mathbf{Ax} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned} \quad (2.4)$$

em que \mathbf{x} e \mathbf{c}^T são vetores n -dimensionais, na qual \mathbf{c}^T é o vetor de custos e \mathbf{x} é o vetor das variáveis ou incógnitas, \mathbf{b} um vetor m -dimensional dos termos independentes ou de recursos e

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad \text{é uma matriz } m \times n, \text{ chamada matriz dos coeficientes.}$$

Nos exemplos 2.1 e 2.2 ilustramos a formulação de modelos de otimização linear.

Exemplo 2.1 (Problema da Fábrica de Trailers[2]). *O proprietário de uma fábrica de trailers para automóveis pretende determinar a melhor combinação para seus três produtos: flat-bed trailers, economy trailers e luxury trailers. Sua loja está limitada a trabalhar 24 dias/mês em metalurgia e 60 dias / mês em madeira para esses produtos. A tabela a seguir indica dados de produção para os trailers:*

Tabela 1: Dados dos três trailers

	Uso por unidade produzida			Disponibilidade de recursos
	<i>Flat-bed</i>	<i>Economy</i>	<i>Luxury</i>	
Dias de trabalho em metalurgia	0.5	2	1	24
Dias de trabalho em madeira	1	2	4	60
Contribuição (\$ x 100)	6	14	13	

Fonte: Silvio [2]

Solução. Construindo um modelo para o Problema da Fábrica de Trailers. Neste problema temos:

Elementos conhecidos: Disponibilidade, composições e contribuição de cada trailer;

Elemento desconhecido: Quantos dias trabalhar em cada tipo de trailer;

Objetivo a ser alcançado: Maximizar a contribuição total;

Restrições: Disponibilidade de dias de trabalho em metalurgia e em madeira.

- Índice:

- A distribuição dos dias de trabalho pode ser feita entre três trailers ($j = 1, 2, 3$). Estes trailers seriam *Flat-bed*, *Economy* e *Luxur*.
- Variáveis de decisão:
 - x_j : quantidade dias/mês usada na fabricação do trailer j .
- Função Objetiva:
 - maximizar $6x_1 + 14x_2 + 13x_3$
 - Os valores 6, 14 e 13 são a contribuição ao lucro do itens 1, 2 e 3, respectivamente.
 - multiplicando por 100, como mostrado na Tabela 1:
 $600x_1 + 1400x_2 + 1300x_3$
- Restrições de Composição de Dias trabalhados:
 - $0.5x_1 + 2x_2 + 1x_3 \leq 24$: Metalurgia
 - $1x_1 + 2x_2 + 4x_3 \leq 60$: Madeira
- Restrições de Não Negatividade das Variáveis:
 - $x_1 \geq 0; x_2 \geq 0; x_3 \geq 0$
- Modelo Matemático

$$Max \quad 600x_1 + 1400x_2 + 1300x_3$$

S. a :

$$0.5x_1 + 2x_2 + x_3 \leq 24$$

$$x_1 + 2x_2 + 4x_3 \leq 60$$

$$x_1 \geq 0; x_2 \geq 0; x_3 \geq 0$$

Solução: O Exemplo 2.1 teve a solução obtida através de um software, na próxima seção conheceremos a ferramenta que pode ser utilizada para encontrar o resultado.

Distribuição dos dias de trabalho entre os trailers: 36 dias para o trailer *Flat-bed*, 0 dias para o *Economy* e 6 dias para o *Luxury*. Com um lucro total de 29.400,00.

Exemplo 2.2. [2] Uma empresa produz dois tipos de bolsas de plástico (B_1, B_2) cujos mercados não absorvem mais do que 80 e 40 unidades diárias, respectivamente. O processo de produção consome dois tipos de matéria-prima: folhas de plásticos e fechos. Cada unidade de B_1 consome duas folhas de plástico e quatro fechos. Cada unidade de B_2 consome três folhas de plástico e três fechos. São disponíveis diariamente 200 folhas de plástico e 240 fechos. Os lucros unitários pelas vendas dos produtos são, respectivamente, R\$20 e R\$25. Qual deve ser o esquema de produção que conduza ao maior lucro possível?

Solução. Vamos construir o modelo para resolução deste exemplo.

Elementos conhecidos: os custos e a disponibilidade da matéria-prima, demanda e composição necessária de cada bolsa;

Elemento desconhecido: qual é a quantidade de cada bolsa que será produzida;

Objetivo a ser alcançado: obter o maior lucro possível;

Restrições: cada bolsa tem que ser feitas com uma determinada quantidade de matéria-prima sem ultrapassar a disponibilidade e também não ultrapassado a demanda dos mercados.

x_1 = quantidades de bolsa 1.

x_2 = quantidades de bolsa 2.

Tabela 2: Disponibilidade, custo e quantidade de matéria-prima necessários para fazer as bolsas

Matéria-prima	Bolsa 1	Bolsa 2	Disponibilidade
Folhas	2	3	200
Feches	4	3	240
Demanda	80	40	
Custo	20	25	

Fonte: Elaborada pela autora

Modelo Matemático:

$$\text{Max } f(\mathbf{x}) = 20x_1 + 25x_2$$

S.a:

$$2x_1 + 3x_2 \leq 200$$

$$4x_1 + 3x_2 \leq 240$$

$$x_1 \leq 8$$

$$x_2 \leq 40$$

$$x_1, x_2 \geq 0$$

Solução:

Mistura: $x_1 = 30; x_2 = 40$

$Val : 1600$

Logo, a empresa deve produzir 30 bolsas do tipo B_1 e 40 bolsas do tipo B_2 com lucro total de 1600 reais.

Resolução Gráfica de um problema de otimização linear contínua

Existem várias formas de obter as soluções de um modelo de otimização linear contínua e uma delas é a resolução gráfica. Para resolver os problemas graficamente, será necessário apresentar algumas definições importantes[5].

Definição 2.2 (Solução factível e região factível). *Uma solução (x_1, x_2, \dots, x_n) de um problema de otimização é dita factível se satisfizer todas as restrições (2.2) e as condições de não-negatividade (2.3). O conjunto de todas as soluções factíveis é chamado região factível.*

Definição 2.3 (Solução ótima). *Uma solução factível que fornece o menor valor à função objetivo f é chamada solução ótima, denotada por $(x_1^*, x_2^*, \dots, x_n^*)$. Uma solução factível é ótima se:*

$$f(x_1^*, x_2^*, \dots, x_n^*) \leq f(x_1, x_2, \dots, x_n)$$

para qualquer solução factível (x_1, x_2, \dots, x_n) .

Definição 2.4 (Vetor Gradiente). *Sejam $A \subset \mathbb{R}^2$ um aberto e $f : A \rightarrow \mathbb{R}$ uma função que possui derivadas parciais em (x_0, y_0) o vetor*

$$\nabla f(x_0, y_0) = \left(\frac{\partial f}{\partial x}(x_0, y_0), \frac{\partial f}{\partial y}(x_0, y_0) \right)$$

chama-se vetor gradiente de f em (x_0, y_0)

Definição 2.5 (Curva de nível). *Sejam $D \subset \mathbb{R}^2$ e $f : D \rightarrow \mathbb{R}$. A curva de nível k de f é o conjunto*

$$f(k) = \{(x, y) \in D : f(x, y) = k\}$$

Recordamos que uma **equação** e uma **inequação** podem ser representadas graficamente como uma **reta** e um **semi-planos**, respectivamente. No Exemplo 2.3 ilustramos essas representações.

Exemplo 2.3. A Figura 2 representa uma **equação** $4x_1 + 3x_2 = 240$ e a Figura 3 representa **inequação** $4x_1 + 3x_2 \leq 240$.

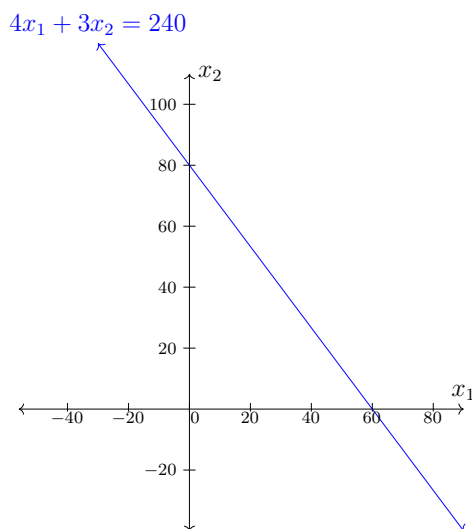


Figura 2: Representação gráfica da reta $4x_1 + 3x_2 = 240$

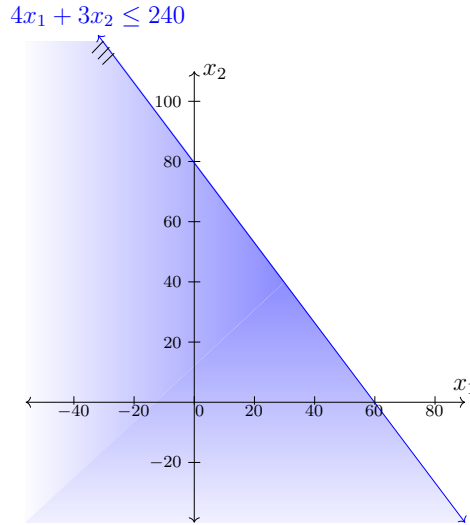


Figura 3: Representação gráfica da inequação $4x_1 + 3x_2 \leq 240$

Utilizaremos o modelo do *Exemplo 2.2*, o qual possui uma função objetiva de maximizar, duas variáveis, quatro restrições e duas condições de não negatividade. Resolver um de modelo otimização é equivalente a achar a solução ótima, ou seja, determinar uma solução factível que maximize o valor da função objetiva. Isto é, determina uma solução factível do tipo \mathbf{x}^* tal que $f(\mathbf{x}^*) \geq f(\mathbf{x}), \forall \mathbf{x}$.

Modelo Matemático do Exemplo 2.2.

$$\text{Max } f(\mathbf{x}) = 20x_1 + 25x_2 \quad (2.5)$$

S.a:

$$2x_1 + 3x_2 \leq 200 \quad (2.6)$$

$$4x_1 + 3x_2 \leq 240 \quad (2.7)$$

$$x_1 \leq 80 \quad (2.8)$$

$$x_2 \leq 40 \quad (2.9)$$

$$x_1, x_2 \geq 0 \quad (2.10)$$

Na Figura 4, o polígono azul é definido pela intersecção desses semiplanos representados pelas restrições (2.6)-(2.10). Os pontos que pertencem a essa região são os *pontos viáveis* ou *pontos factíveis* do problema e a região é chamada de

região factível([7, 19]).

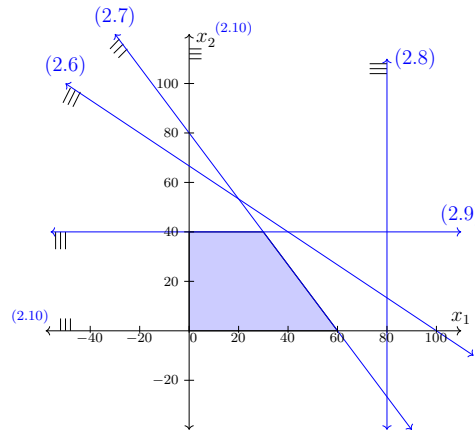


Figura 4: Região factível do modelo matemático do Exemplo 2.2

O *Vetor Gradiente* define a direção de crescimento de uma função, logo como o problema que estamos trabalhando é de maximizar iremos utilizar o mesmo sentido, se o problema fosse de minimizar era só utilizar o sentido inverso. O Vetor Gradiente da função objetiva (∇f) do modelo estudado está representado na Figura 5.

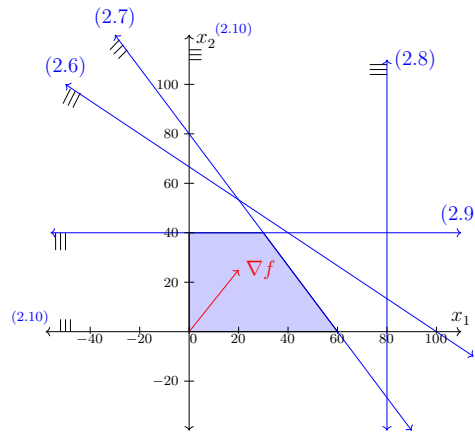


Figura 5: Modelo matemático do Exemplo 2.2 com o Vetor Gradiente.

Uma curva de nível com valor 500 está ilustrada por uma reta tracejada na Figura 6. Todos os pontos que pertençam a curva de nível são os pontos que satisfazem a função objetiva com $k = 500$ e formam a reta $20x_1 + 25x_2 = 500$.

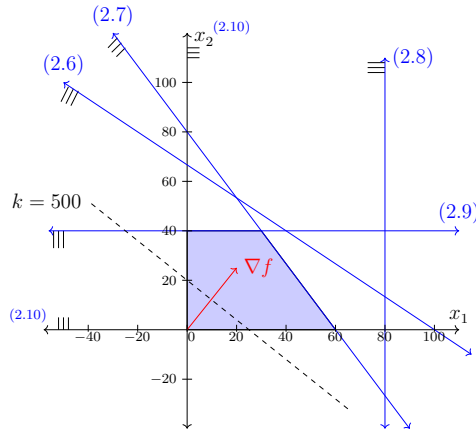


Figura 6: Região factível do Exemplo 2.2 com o Vetor Gradiente e com a curva de nível $k = 500$.

Para resolver o problema, “deslocaremos” a curva de nível no sentido do vetor gradiente até chegar ao limite da região factível, ou seja, encontrando uma curva de nível que passa sobre o ponto ótimo(P). A solução ótimo é $(30,40)$ e o valor ótimo é 1600, como mostrado na Figura 7.

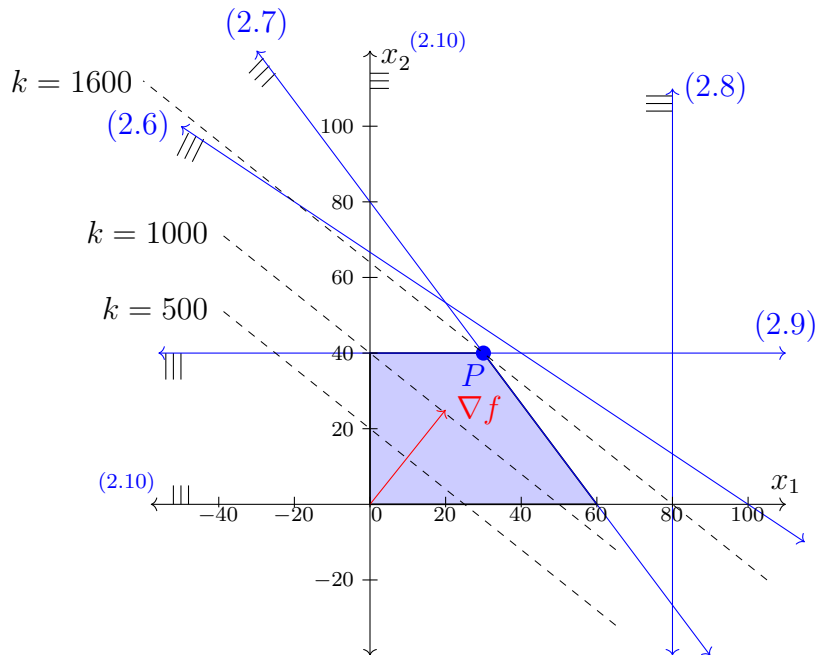


Figura 7: Ponto ótimo (P) do Exemplo 2.2.

3 Sistema de Resolução GUROBI.

Para obter a solução de um problema de otimização é necessário alguns métodos, algoritmos ou ferramentas computacionais. Estas ferramentas podem ser CPLEX, GUROBI, XPRESS-MP, LPSOLVE e CLP. Os softwares de otimização matemática CPLEX, GUROBI e XPRESS-MP são comerciais e a LPSOLVE e CLP não são comerciais [20, 2, 22, 26].

Neste estudo, será usado o sistema de otimização GUROBI. O **Gurobi** é um software de otimização matemática que oferece acesso livre para estudantes. O solucionador funciona em uma grande variedade de linguagens de programação, tal como C++, JAVA, Python etc. Utilizaremos a linguagem **Python** no desenrolar desse estudo[20]. O termo “Python-Gurobi” será usado para indicar que a linguagem Python será usada como interface com o sistema de otimização Gurobi.

Instalação do Gurobi.

O Gurobi pode ser executado em várias sistemas operacionais como: *Windows64*, *linux* e *Mac*[20]. Neste projeto operaremos pelo sistema **Windows64**. É possível solicitar uma licença acadêmica no *Registro de licença acadêmica* [23], esta licença tem um tempo limitado. A fim de criar uma conta, basta acessar o *Cadastre-se* [6] com o e-mail institucional. É necessário instalar antes o Anaconda que é uma distribuição das linguagens de programação Python. Acesse o *Gurobi e Anaconda para Windows* [13] que explica passo a passo ou pode assistir o vídeo do Professor Rafael Lima[15]

Observação 3.1. *Um dos problemas que podem ocorrer durante a instalação é o erro mostrado na Figura 8. O problema está ocorrendo porque o Anaconda foi atualizado para a **versão** 3.8 do Python, no entanto o pacote do Gurobi ainda não foi atualizado, como mostra na Figura 8 o “**python < 3.8**”.*

*Quando for instalar o python é bom verificar a versão aceita pelo GUROBI. Essa informação está no Quais versões Python são compatíveis com Gurobi? [21]. Neste estudo usamos o Gurobi **versão** 9.0, e o Anaconda com a versão do **Python** 3.7.*

```

Selecionar Anaconda Prompt (anaconda3)

(base) C:\Users\marya>conda config --add channels http://conda.anaconda.org/gurobi
Warning: 'http://conda.anaconda.org/gurobi' already in 'channels' list, moving to the top

(base) C:\Users\marya>conda install gurobi
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Solving environment: failed with repodata from current_repodata.json, will retry with next repodata source.
Collecting package metadata (repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Solving environment: /
Found conflicts! Looking for incompatible packages.
This can take several minutes. Press CTRL-C to abort.
failed

UnsatisfiableError: The following specifications were found
to be incompatible with the existing python installation in your environment:

Specifications:

- gurobi -> python[version='2.7.*|3.5.*|>=2.7,<2.8.0a0|>=3.6,<3.7.0a0|>=3.7,<3.8.0a0|3.4.*']
Your python: python=3.8

If python is on the left-most side of the chain, that's the version you've asked for.
When python appears to the right, that indicates that the thing on the left is somehow
not available for the python version you are constrained to. Note that conda will not
change your python version to a different minor version unless you explicitly specify
that.

The following specifications were found to be incompatible with your CUDA driver:

- feature:/win-64::_cuda==10.2=0
- feature:|@/win-64::_cuda==10.2=0

Your installed CUDA driver is: 10.2

```

Figura 8: Erro na instalação do Gurobi.

Depois de ser feito o download do anaconda, pode-se observar que o anaconda veio com algumas ferramentas instaladas, como o Jupyter Notebook. O **Jupyter Notebook** é um editor de código python que funciona pelo navegador e é utilizado no desenrolar deste estudo.

Resolução de um problema de Otimização Linear Contínua pelo GUROBI

O modelo do Exemplo 2.2 contém duas variáveis, logo é possível encontrar uma solução pelo método gráfico. Entretanto, em pesquisa operacional os problemas geralmente tem $n > 3$ variáveis, e será necessário ferramentas computacionais como o GUROBI. No presente estudo, estudaremos algumas funcionalidades do GUROBI. Alguns conceitos de Python serão importantes na continuação.

Resolução do Exemplo 2.1 pelo Gurobi

O problema do Exemplo 2.1:

$$\text{Max } f(\mathbf{x}) = 600x_1 + 1400x_2 + 1300x_3$$

S.a:

$$0.5x_1 + 2x_2 + x_3 \leq 24$$

$$x_1 + 2x_2 + 4x_3 \leq 60$$

$$x_1, x_2, x_3 \geq 0$$

Logo, precisa maximizar este modelo que contém três restrições e quatro condições de não negatividade.

Na Figura 9 apresenta-se exemplo escrito na sintaxe do Gurobi usando o Python.

```

1 m = gp.Model("Exemplo 2.1")
2
3 x1 = m.addVar()
4 x2 = m.addVar()
5 x3 = m.addVar()
6
7
8 m.setObjective(600 * x1 + 1400 * x2 + 1300 * x3, sense=gp.GRB.MAXIMIZE)
9
10
11 c1 = m.addConstr(0.5 * x1 + 2 * x2 + x3 <= 24)
12 c2 = m.addConstr(x1 + 2 * x2 + 4 * x3 <= 60)
13
14
15 m.optimize()
```

Figura 9: Exemplo 2.1(Problema da Fábrica de Trailers) escrito na sintaxe Python-GUROBI.

Depois de abrir um notebook no Jupyter, o primeiro passo para implementação do modelo é importar o GUROBI na primeira célula do Jupyter, utilizando comando `import gurobipy as gp`, o `gp` é uma forma mais curta de nomear um determinado módulo, tal que quando necessariamente quisermos nos referir ao módulo do GUROBI digitaremos `gp.comando`.

Na segunda célula `In[3]`, será implementado o modelo do problema. A linha 1 se refere ao comando de criação do modelo, e nomeamos `Exemplo 2.1`. O segundo passo é inserir as variáveis de decisão, o comando `.addVar()` adiciona uma variável

por vez, como no modelo utilizaremos 3 variáveis, logo inserimos 3 vezes o comando `.addVar()` como mostrado na linha 3 até a linha 6 da célula [In\[3\]](#). O comando `.addVar()` pode também definir qual tipo da variável, por exemplo se as variáveis for do tipo inteiro, utilizamos o código `vtype=gp.GRB.INTEGER` dentro dos parênteses do `.addVar()`. Todavia, no Exemplo 2.1 não será necessário definir o tipo da variável, pois Problema da Fábrica de Trailers pode ser definido como contínuo ou inteiro que não vai mudar seu valor ótimo.

O terceiro passo será fixar a *função objetiva* pelo comando `.setObjective()`, dentro dos parênteses digitamos o critério de otimização para MINIMIZAR ou MAXIMIZAR. Visto que o modelo que estamos trabalhando é para MAXIMIZAR logo usaremos `sense=gp.GRB.MAXIMIZE`. O quarto passo é inserir as *restrições*, o comando `.addConstr()` adiciona uma restrição por vez. Os comandos ilustrados nas linhas 1-12 foram usados para definir o problema. Para resolvê-lo usamos o comando `.optimize()`.

Na Figura 10 temos a solução com o valor ótimo e na Figura 11 os comandos necessários para visualizar o valores das variáveis e o valor ótimo.

```
Using license file C:\Users\marya\gurobi.lic
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 2 rows, 3 columns and 6 nonzeros
Model fingerprint: 0x88f5aba3
Coefficient statistics:
  Matrix range      [5e-01, 4e+00]
  Objective range   [6e+02, 1e+03]
  Bounds range      [0e+00, 0e+00]
  RHS range         [2e+01, 6e+01]
Presolve time: 0.02s
Presolved: 2 rows, 3 columns, 6 nonzeros
```

Iteration	Objective	Primal Inf.	Dual Inf.	Time
0	3.2500000e+33	4.750000e+30	3.250000e+03	0s
3	2.9400000e+04	0.000000e+00	0.000000e+00	0s

```
Solved in 3 iterations and 0.03 seconds
Optimal objective 2.940000000e+04
```

Figura 10: Detalhes do processo de solução em Python-Gurobi:
Exemplo 2.1(Problema da Fábrica de Trailers)

```
1 print("x1 =", x1.X)
2 print("x2 =", x2.X)
3 print("x3 =", x3.X)
4 print("Valor da solução:", m.ObjVal)

x1 = 36.0
x2 = -0.0
x3 = 6.0
Valor da solução: 29400.0
```

Figura 11: Comando para imprimir e a solução Exemplo 2.1(Problema da Fábrica de Trailers)

4 Resolvendo um problema da dieta pelo GUROBI.

O Problema da Dieta (ou Problema da Mistura), consiste em encontrar a melhor seleção de alimentos para compor uma dieta diária atendendo critérios nutricionais apresentados e de o custo mínimo. O Exemplo 4.1 foi inspirado no Problema da Dieta do GOLDBARG (2015, p.39)[11] e do Problema da Dieta da Rangel(2012, p.6)[22].

Exemplo 4.1 (Problema da Dieta). *Suponha-se que, por motivos justificáveis, a dieta alimentar diária da Gicelma esteja restrita a Crepioca de banana, arroz com brócolis, feijão carioca e omelete simples. Sabe-se que os componentes nutricionais são expressos por três componentes encontrados normalmente nas informações nutricionais dos alimentos, os componentes são carboidrato, proteína e gordura total . A Tabela 3 tem as informações necessárias para saber qual a quantidade de cada alimento que a Gicelma deve consumir, que atenda os requisitos mínimos nutricionais para uma melhor qualidade de vida e tenha um custo mínimo*

Tabela 3: Informações nutricionais dos alimentos da Dieta

Componentes	Crepioca de banana (110g)	Arroz com brócolis (200g)	Feijão carioca (165g)	Omelete(3un)	Requisito Mínimo nutricional
Carbs	28,34g	54g	35,66g	3,45g	100g
Proteína	6,37g	6,10g	12,08g	20,43g	45g
Gord.	4,22g	4,20g	2,04g	21,42g	22g
Custo	2,5 reais	3 reais	1,3 real	1,5 real	

Fonte: Elaborada pela autora

Solução. Vamos discutir quais serão os elementos conhecidos, elementos desconhecidos, função objetiva e as restrições[22], pois com estas informações ajuda a construir um modelo para o Exemplo 6.1.

Elementos conhecidos: custos, composição e requisitos mínimos de cada alimento;

Elemento desconhecido: quanto de cada alimento será necessário para fazer esta dieta;

Função objetiva: minimizar o custo total;

Restrição: esta dieta deve ter uma quantidade mínima de composição pré-específicas.

- Índices:
 - ($j = 1, 2, 3, 4$) representa os alimentos: crepioca de banana, arroz com brócolis, feijão carioca e omelete, respectivamente;
 - ($i = 1, 2, 3$) representa as componentes: carboidrato, proteína e gordura total, respectivamente.
- Variáveis de decisão:
 - x_j : n° de unidades de cada alimento j utilizado na dieta.
- Função Objetiva:
 - minimizar $2.5x_1 + 3x_2 + 1.3x_3 + 1.5x_4$.

- Restrições:

$$28.34x_1 + 54x_2 + 35.66x_3 + 3.45x_4 \geq 100: \text{Carboidrato};$$

$$6.37x_1 + 6.10x_2 + 12.08x_3 + 20.43x_4 \geq 45: \text{Proteína};$$

$$4.22x_1 + 4.20x_2 + 2.04x_3 + 21.42x_4 \geq 22: \text{Gordura total}.$$

- Restrições de Não Negatividade das Variáveis:

$$x_1 \geq 0; x_2 \geq 0; x_3 \geq 0; x_4 \geq 0$$

- Modelo Matemático

$$\text{Min } f(\mathbf{x}) = 2.5x_1 + 3x_2 + 1.3x_3 + 1.5x_4$$

S.a:

$$28.34x_1 + 54x_2 + 35.66x_3 + 3.45x_4 \geq 100$$

$$6.37x_1 + 6.10x_2 + 12.08x_3 + 20.43x_4 \geq 45$$

$$4.22x_1 + 4.20x_2 + 2.04x_3 + 21.42x_4 \geq 22$$

$$x_1 \geq 0; x_2 \geq 0; x_3 \geq 0; x_4 \geq 0$$

Resolução do Exemplo 4.1(Problema de Dieta) pelo Gurobi.

O processo de Resolução do Problema da Dieta foi feito utilizando a mesma lógica da **Seção 3**, como fonte o guia do Gurobi[20]. Vamos discutir a implementação do Exemplo 4.1, linha por linha, para compreender melhor a implementação.

Passo 1(Importar o Gurobi): antes de começar a implementar o Gurobi é necessário importar o Gurobi:

```
import gurobipy as gp
```

Este comando permite que o Gurobi disponibilize todas as funções e classes.

Passo 2(Construir o modelo): é necessário criar as variáveis de decisão, a função objetiva e as restrições. Inicialmente nomeando modelo usando o comando:

```
m = gp.Model("Problema da Dieta")
```


Como vamos trabalhar com variável contínua não é necessário definir o tipo da variável, pois o Gurobi considera automaticamente que o modelo é contínuo se não definir o *vtype*. Temos que o Exemplo 4.1 tem 4 elementos que compõe a dieta, logo precisaremos de 4 variáveis de decisão e então procederemos implementando 4 vezes o comando *.addVar()*, na qual cada comando representara um elemento da dieta.

$$x1 = m.addVar()$$
$$x2 = m.addVar()$$
$$x3 = m.addVar()$$
$$x4 = m.addVar()$$

Função objetiva: será necessário definir o critério de otimização usando o comando *sense=*. Para a implementação, iremos digitalizar os valores dos custos de cada variável dentro dos parênteses do comando *.setObjective()*:

$$m.setObjective(2.5*x1+3*x2+1.3*x3+1.5*x4, \textit{sense} = gp.GRB.MINIMIZE)$$

Restrição:

$$c1 = m.addConstr(28.34 * x1 + 54 * x2 + 35.66 * x3 + 3.45 * x4 >= 100)$$
$$c2 = m.addConstr(6.37 * x1 + 6.10 * x2 + 12.08 * x3 + 20.43 * x4 >= 45)$$
$$c3 = m.addConstr(4.22 * x1 + 4.20 * x2 + 2.04 * x3 + 21.42 * x4 >= 22)$$

Como visto na **Seção 3** adicionamos cada restrição digitalizando o comando *.addConstr()*. O **Passo 3** precisamos do comando *.optimize()* para executar a otimização do modelo:

$$m.optimize()$$

A Figura 12 apresenta o código do Exemplo 4.1(Problema da Dieta).

```

1 import gurobipy as gp

1 m = gp.Model("Exemplo Diet")
2
3 x1 = m.addVar()
4 x2 = m.addVar()
5 x3 = m.addVar()
6 x4 = m.addVar()
7
8 m.setObjective(2.5 * x1 + 3 * x2 + 1.3 * x3 + 1.5 * x4, sense=gp.GRB.MINIMIZE) #Função Obj.
9
10 c1 = m.addConstr(28.34 * x1 + 54 * x2 + 32.66 * x3 + 3.45 * x4 >= 100)#Carboidrato
11 c2 = m.addConstr(6.37 * x1 + 6.10 * x2 + 12.08 * x3 + 20.43 * x4 >= 45)#Proteína
12 c3 = m.addConstr(4.22 * x1 + 4.2 * x2 + 2.04 * x3 + 21.42 * x4 >= 22)#Gordura total
13
14 m.optimize()

```

Figura 12: Implementação em Python-Gurobi do Problema da Dieta.

Passo 4(Resultado) usando o comando *print* para visualizar o resultados na tela.

As Figuras 12 e 13 mostram o código implementado no Jupiter.

```

1 print("x1 =", x1.X)
2 print("x2 =", x2.X)
3 print("x3 =", x3.X)
4 print("x4 =", x4.X)
5 print("Valor da solução:", m.ObjVal)

x1 = 0.0
x2 = 0.0
x3 = 2.983369028063682
x4 = 0.7429471140406203
Valor da solução: 4.992800407543717

```

Figura 13: Resultado do Exemplo 4.1(Problema da Dieta)

Portanto, $x_1 = 0$; $x_2 = 0$; $x_3 = 2.98$; $x_4 = 0.74$ e $Val : 4.99$ Logo, a Gicelma deve consumir 2.98 de Feijão carioca e 0.74 de Omelete com um custo total mais ou menos de 4,99 reais.

5 O Problema da Mochila.

Apresentamos no Exemplo 5.1 o Problema da Mochila ([17, 3]).

Exemplo 5.1 (Problema da Mochila). *Um garoto chamado Ramon foi convocado às pressas pelo técnico para um Campeonato Regional de Badminton que ocorreu*

no dia seguinte. Como foi avisado em cima da hora o técnico disponibilizaria os equipamentos e seria necessário que o aluno levasse somente uma mochila pequena para passar um dia fora. O Ramon chegou em casa e avisou a mãe, porém ela avisou que não teria tempo para escolher e colocar os itens na mochila, logo esta tarefa ficou para o garoto. Para ajudar o garoto, a mãe sugeriu que ele usasse a Tabela 4 para selecionar os objetos. A capacidade da mochila era de 12.

Tabela 4: Peso e importância dos itens

Itens	Peso	Importância
1	2	1
2	6	2
3	3	4
4	2	1
5	3	5
6	4	6

Fonte: Elaborada pela autora

Solução. O objetivo do Ramon é escolher a melhor seleção de itens para colocar na mochila, considerando o peso e a importância de cada item, vamos chamar a importância dos itens de *valor*. Pelo enunciado do exemplo temos que existem 6 itens e a mochila tem capacidade de 12, logo os *elementos conhecidos* são a capacidade da mochila, a quantidade de itens, os pesos e os valores.

Um problema que possui uma única restrição além das restrições de não-negatividade é conhecido como o **Problema da Mochila**. Os *elementos desconhecidos* quais itens vamos colocar na mochila do Ramon. O objetivo do problema é encontrar a melhor seleção de itens, considerando a importância ou como vamos chamar de valor de cada item. Cada item tem um peso e a *restrição* do problema é não ultrapassar a capacidade da mochila de 12.

- **Índices:**

$$\text{itens } j = 1, \dots, 6$$

- **Variáveis de decisão:**

$$x_j = \begin{cases} 1, & \text{se o item } j \text{ é colocado na mochila;} \\ 0, & \text{caso contrário.} \end{cases}$$

- **Função Objetiva:**

$$\text{Maximizar } f(\mathbf{x}) = x_1 + 2x_2 + 4x_3 + 1x_4 + 5x_5 + x_6$$

- **Restrições de capacidade da mochila:**

A quantidade de itens escolhidos não podem ultrapassar a capacidade máxima da mochila.

$$2x_1 + 6x_2 + 3x_3 + 2x_4 + 3x_5 + 4x_6 \leq 12$$

- **Restrições de domínio das Variáveis:**

$$x_j \in \{0, 1\}, j = 1, \dots, 6.$$

Logo, o modelo a seguir é o modelo de otimização binário para o Problema da mochila do Ramon:

$$\text{Max } f(\mathbf{x}) = x_1 + 2x_2 + 4x_3 + 1x_4 + 5x_5 + x_6$$

S.a:

$$2x_1 + 6x_2 + 3x_3 + 2x_4 + 3x_5 + 4x_6 \leq 12$$

$$x_j \in \{0, 1\}, j = 1, \dots, 6$$

Existem também outras aplicações do Problema da Mochila (Knapsack Problem) como a *seleção de projetos* que envolvem capitais e investimentos [22, 3]. Afirmção geral do problema envolve n itens e uma capacidade b . O item $j = 1 \dots n$ tem um pesos a_j e um valor de p_j [22, 3, 24], esta expressado em 5.1-5.2

$$\text{Maximizar } z = \sum_{j=1}^n p_j x_j \quad (5.1)$$

S.a:

$$\sum_{j=1}^n a_j x_j \leq b \quad (5.2)$$

$$x_j \in \{0, 1\}, j = 1, \dots, n \quad (5.3)$$

Resolvendo o Problema da Mochila pelo Python-Gurobi.

Como estamos operando com Gurobi-Python será necessário conhecimentos intermediários do Python para algumas funções essenciais, como por exemplo a importação de instâncias, na qual podemos editar os dados de um problema sem ter que mudar o código principal. Primeiro usaremos um modo em que não é necessário muito entendimento de Python. Utilizaremos como fonte o estudo do Professor Rafael Lima[15] e o guia do Gurobi[20].

Passo 1(Importar o Gurobi): importar a biblioteca do Gurobi

```
import gurobipy as gp
```

Passo 2(Parâmetro): Colocar os dados do Exemplo 5.1.

A quantidade de itens: *qtd_itens = 6*

A capacidade da mochila: *capacidade = 12*

O vetor de valores: *vet_valores = [1, 2, 4, 1, 5, 6]*

O vetor de pesos: *vet_pesos = [2, 6, 3, 2, 3, 4]*

O Gurobi utiliza os dicionário e rótulos do Python, logo precisamos criar rótulos para nomear cada item, criamos uma lista com o nome de *itens*:

```
itens = list()
```

```
for i in range(qtd_itens):
```

```
    rotulo = 'Item-{}'.format(i+1)
```

```
    itens.append(rotulo)
```

Agora, os dicionários serão necessários, pois conectam os elementos da lista *itens* aos elementos dos vetores pesos e valores:

```
pesos = dict()
```

```
for idx,peso in enumerate(vet_pesos):
```

```

    rotulo = itens[idx]

    pesos[rotulo] = peso

valores = dict()

for idx,valor in enumerate(vet_valores):

    rotulo = itens[idx]

    valores[rotulo] = valor

```

Os comandos necessários no **Passo 2** são apresentados na Figura 14

```

1 import gurobipy as gp

1 qtd_itens = 6
2 capacidade = 12
3
4 #temos que rótular os itens
5
6 itens = list()
7
8 for i in range(qtd_itens):
9     rotulo = 'Item_{}'.format(i+1)
10    itens.append(rotulo)#inserir na lista
11
12 vet_pesos = [2, 6, 3, 2, 3, 4]
13 pesos = dict()#dicionario(pesquisar)
14 for idx,peso in enumerate(vet_pesos):#idx=indice
15     rotulo = itens[idx]#rotulo(pesquisar mais)
16     pesos[rotulo] = peso
17
18 vet_valores = [1, 2, 4, 1, 5, 6]
19 valores = dict()
20 for idx,valor in enumerate(vet_valores):
21     rotulo = itens[idx]
22     valores[rotulo] = valor

```

Figura 14: Dados do Exemplo 5.1 na sintaxe Python-Gurobi.

Passo 3(Construir o modelo): Vamos utilizar a mesma lógica da **Seção 3** para construir o modelo do Exemplo 5.1.

Criar o modelo:

```
m = gp.Model()
```

Criar as variáveis de decisão, tal que o tipo da variável é binária(1 ou 0)5.3:

$$x = m.addVars(itens, vtype = gp.GRB.BINARY)$$

O comando `.addVars()` inclui varias variáveis ao mesmo tempo, tal que no nosso código vai adicionar a cada elemento da lista `itens` uma variável.

Função objetiva:

$$m.setObjective(gp.quicksum(x[i] * valores[i] \text{ for } i \text{ in } itens), \\ sense= gp.GRB.MAXIMIZE)$$

O comando `.setObjective()` adiciona uma função objetiva, pode se observar que na equação 5.1 tem uma somatória. No Gurobi existe uma função que faz exatamente papel da somatória da matemática que é a função `gp.quicksum()`, isto é, a `gp.quicksum(x[i] * pesos[i] \text{ for } i \text{ in } itens)` multiplicará para cada elemento da lista `itens` por um elemento vetor de `pesos` e depois somar tudo. O sentido da otimização é definido por `sense=gp.GRB.MAXIMIZE`.

Restrição: para a restrição temos que utilizar o comando `.addConstr()` que vai adicionar uma restrição e como na função objetiva, utilizaremos a função `gp.quicksum()` cada elemento da lista `itens` será multiplicado por elemento no vetor de `pesos`

$$c = m.addConstr(gp.quicksum(x[i] * pesos[i] \text{ for } i \text{ in } itens) \leq capacidade)$$

Para resolver o modelo que criamos usamos o comando:

$$m.optimize()$$

As Figuras 15 e 16 ilustram o código implementado e a resolução do exemplo.

```

1  #construir o modelo
2
3  m = gp.Model()
4
5  #variaveis
6
7  x = m.addVars(itens, vtype=gp.GRB.BINARY)
8
9  #função objetiva
10 m.setObjective(
11     gp.quicksum( x[i] * valores[i] for i in itens),
12     sense=gp.GRB.MAXIMIZE)#quicksum = somatoria, sense = sentido
13
14 #restrição
15
16 c = m.addConstr(gp.quicksum( x[i] * pesos[i] for i in itens) <= capacidade )
17
18 #executar
19
20 m.optimize()

```

Figura 15: Implementação do modelo do Problema da Mochila em Python-Gurobi.

```

1  for item in itens:
2      if round(x[item].X) == 1:
3          print(item)
4
5  print('Valor Total', m.objVal)

```

Item_1
Item_3
Item_5
Item_6
Valor Total 16.0

Figura 16: Solução do Exemplo 5.1 em Python-Gurobi.

O Exemplo 5.1 é um problema de 6 itens. Se a mãe do Ramon percebesse que tem outros itens que podem ser relevantes levar para a viagem seria necessário fazer algumas mudanças nos parâmetros do código principal das Figuras 14 e 15. Nestas situações a importação de instância é fundamental. Modificamos o código do Exemplo 5.1 para que não seja necessário editar o código principal.

Podemos observar que o código do Exemplo 5.1 já está “maleável”, pois os parâmetros e o modelo estão separados. Diante disso, somente mudaremos os **Passo 2(Parâmetro)** e o **Passo 4(Resultados)** e um pouco do **Passo 3(Construir o modelo)**.

Para continuar é importante entender o que é uma instâncias. A instância é um caso particular de um problema obtido quando atribuímos valores específicos para os parâmetros de um modelo. Uma instância para ser lida em um arquivo de dados separado. A Figura 17 mostra como é a instâncias do Exemplo 5.1.

quantidade de itens	valores dos itens	pesos dos itens
6	12	
2	1	
6	2	
3	4	
2	1	
3	5	
4	6	

Figura 17: Instância do Exemplo 5.1.

Para que o código possa resolver diferentes instâncias de um problema, criaremos uma função chamada *le_dados(nome_arq)* para que o programa possa ler um arquivo dos parâmetros do problema(instância). Esta função, vai ler cada linha do arquivo(*linhas = f.readlines()*) vai compreender que a primeira linha se refere a quantidade de itens e capacidade da mochila, respectivamente. Os demais parâmetros são lidos por colunas na primeira coluna o vetor de pesos e na segunda representa o vetor de valores. A lógica para os dicionários e rótulos será a mesma que utilizamos no **Passo 2(Parâmetro)** na Figura 14. Para concluir a mudança no **Passo 2(Parâmetro)**, retornaremos os dados que serão usados no **Passo 3(Construir o modelo)** com o *return itens, capacidade, pesos, valores*. O código da função *le_dados* é exibido na Figura 18.

Também colocaremos o modelo dentro de uma nova função, na qual chamaremos de *solve_ProblemaMochila(nome_arq)* a diferença que nesta nova função chamará a função *le_dados(nome_arq)* para importar os dados das instâncias necessários para resolver o problema. O novo código é mostrado na Figura 19.

```

1 import gurobipy as gp

2 def le_dados(nome_arq):
3     with open(nome_arq, 'r') as f:
4         linhas = f.readlines()#Ler Linhas
5         #strip= tirar espacamento
6         #split= quebrar
7         valores = linhas[0].strip().split(' ')#elemento da linha 0
8         qtd_itens = int(valores[0])#Linha 0 e coluna 0
9         capacidade = int(valores[1])#Linha 0 e coluna 1
10        #vetores de valor e peso
11        vet_pesos = list()
12        vet_valores = list()
13        del(linhas[0]) #apagar a primeira linha, como se não contasse mais a primeira linha
14        for linha in linhas:
15            valores = linha.strip().split(' ')
16            vet_pesos.append(int(valores[0]))
17            vet_valores.append(int(valores[1]))
18
19        #copiar e mudar algumas coisas do GurobiEx01Lista2.py
20        #rotulo
21        itens = list()#nitens
22        for i in range(qtd_itens):
23            rotulo = 'Item_{}'.format(i + 1)
24            itens.append(rotulo)
25
26        #dicionario peso
27        pesos = dict()
28        for idx, peso in enumerate(vet_pesos):
29            rotulo = itens[idx]
30            pesos[rotulo] = peso
31
32        # dicionario valores
33        valores = dict()
34        for idx, valor in enumerate(vet_valores):
35            rotulo = itens[idx]
36            valores[rotulo] = valor
37
38    return itens, capacidade, pesos, valores#qtd_itens

```

Figura 18: A função `le_dados` para ler instâncias do Problema da Mochila.

O código principal chama a função (`solve_ProblemaMochila(nome_arq)`) e imprime o resultado como mostrado na Figura 20.

```

1 def solve_ProblemaMochila(nome_arq):
2     #ler dados
3     itens, capacidade, pesos, valores = le_dados(nome_arq)
4
5     # construir o modelo
6     m = gp.Model()
7
8     # variaveis
9     x = m.addVars(itens, vtype=gp.GRB.BINARY)
10
11     # função objetiva
12     m.setObjective(gp.quicksum(x[i] * valores[i] for i in itens),
13                    sense=gp.GRB.MAXIMIZE)
14
15     # restrição
16     c = m.addConstr(gp.quicksum(x[i] * pesos[i] for i in itens) <= capacidade)
17
18     # executar
19     m.optimize()
20
21     # lista dos itens incluídos na mochila
22     lista_mochila = list()
23     for item in itens:
24         if round(x[item].X) == 1:
25             lista_mochila.append(item)
26
27     return m.objVal, lista_mochila

```

Figura 19: Função `solve_ProblemaMochila()`

```

1 arq = 'InstanciaTeste.Ex5.1.txt'
2 resultado, lista_mochila = solve_ProblemaMochila(arq)
3 print(resultado)
4 print(lista_mochila)

```

Figura 20: Programa principal para resolver o Problema da Mochila

O valor ótimo e a seleção de itens está mostrado na Figura 21. Portanto, o Ramon terá que levar para a viagem os itens 1, 3, 5 e 6.

```

Solution count 2: 16

Optimal solution found (tolerance 1.00e-04)
Best objective 1.600000000000e+01, best bound 1.600000000000e+01, gap 0.0000%
16.0
['Item_1', 'Item_3', 'Item_5', 'Item_6']

```

Figura 21: Solução do Exemplo 5.1 em Python-Gurobi.

Observação 5.1. O código das Figuras 18, 19 e 20 está construído de uma forma que podem importar um arquivo (instância), porém não necessariamente este arquivo (instância) tem que ser igual a instância representada na Figura 17, este código pode importar outras instâncias com outras determinadas quantidades finitas de itens, capacidade da mochilas, valores ou pesos de um Problema da Mochila.

6 O Problema do Caixeiro Viajante.

Um grafo composto por um conjunto discreto V , cujos os elementos do conjunto são chamados de vértices (ou nós) e um conjunto A cujos elementos são arcos ou aresta, tal que cada aresta se associa a dois vértices e é denotado como $G(V, A)$, ou simplesmente G [14]. Algumas definições são importantes para a continuidade deste estudo[1].

Definição 6.1 (Subgrafo). *Dado um $G(V, A)$, um subgrafo é um grafo $H(V', A')$, tal que os vértices e arestas de H são subconjuntos dos vértices e arestas de G , (ou seja, $V' \subseteq V, A' \subseteq A$) e cada aresta de H tem a sua extremidade em G .*

Definição 6.2 (Caminho). *Dado um $G(V, A)$, um caminho em um grafo G é uma sequência de vértices e arestas alternados, na qual não há repetição de vértices e nem de aresta.*

Definição 6.3 (Ciclo). *Dado um $G(V, A)$, um ciclo ou circuito em um grafo G é uma sequência de vértices e arestas alternados, onde não tem repetição de aresta e nem de vértice (com exceção do inicial e do final) e que começa e termina pelo mesmo vértice.*

Um Grafo $G(V, A)$ é *hamiltoniano* se possuir um *ciclo hamiltoniano*, um *ciclo hamiltoniano* é um ciclo onde parti de um vértice, visita todos os elementos do V uma única vez e retorna no vértice de partida.

Considere um conjunto de cidades distintas, o Problema do Caixeiro Viajante(PCV) ou travelling salesman problem (TSP) consiste em encontrar o menor custo tal que o condutor sai de uma cidade e passe por todas as $(n - 1)$ demais cidades e retorne para cidade que saiu sem passar pela mesma cidade mais do que uma vez[3]. Se representarmos o problema através de um grafo em que os vértices representam as n cidades e as arestas representam a possibilidade de viajar entre duas cidades, logo para resolver o problema precisaremos de um *ciclo hamiltoniano*.

O PCV é um problema profundamente estudado na Pesquisa Operacional, pois é um problema que possui uma solução complexa, mas o enunciado é de fácil

compreensão [25, 22]. Vamos ilustrar o problema usando um exemplo do Problema do Caixeiro Viajante(PCV)[3].

Exemplo 6.1 (Problema do Caixeiro Viajante(PCV)). *Um caixeiro viajante chamado Damon tem clientes em seis cidades: Ubatuba(UBA), Caraguatatuba(CAR), São José dos Campos(SJC), São Luiz do Paraitinga(SLP), Paraty(PAR) e Taubaté(TAU), a Figura 22 exibe o mapa da região litorânea do Estado de São Paulo, na qual enquadra as cidades que o Ramon tem clientes. O caixeiro mora em Ubatuba e precisa partir em uma viagem de negócios e a cidade de partida e destino final será Ubatuba, nesta viagem ele precisará passar por todas as outras cinco cidades uma única vez. Os custos que consiste nas distancia entre as cidades, em quilômetros, estão colocado no Quadro 1.*

Quadro 1: Distância entre cidades 6.1(Problema do Caixeiro Viajante)

	UBA	CAR	SJC	SLP	PAR	TAU
UBA	0	53.4	∞	53.7	72.7	∞
CAR	53.4	0	87.1	∞	∞	∞
SJC	∞	87.1	0	∞	∞	45.3
SLP	53.7	∞	∞	0	126	46.2
PAR	72.7	∞	∞	126	0	137
TAU	∞	∞	45.3	46.2	137	0

Fonte: Elaborada pela autora

Solução. O problema pode ser representado através de um grafo $G(V, A)$, onde $V = \{UBA, CAR, SJC, SLP, PAR, TAU\}$ é o conjunto de cidades que o Ramon vai visitar e o conjunto de aresta, A , representa as possibilidades de viajar de uma cidade a outra e este grafo está exibido na Figura 23.

Para o desenvolvimento da solução do Problema do Caixeiro Viajantes é importante identificar os elementos conhecidos, elementos desconhecidos, função objetiva e as restrições[22], pois com estas informações podemos construir um modelo para o Exemplo 6.1.

Elementos conhecidos: custos, as cidades e a quantidade de cidades;

Elemento desconhecido: qual será a sequência de cidades que o Damon vai visitar;



Figura 22: Exemplo 6.1(Problema do Caixeiro Viajante).

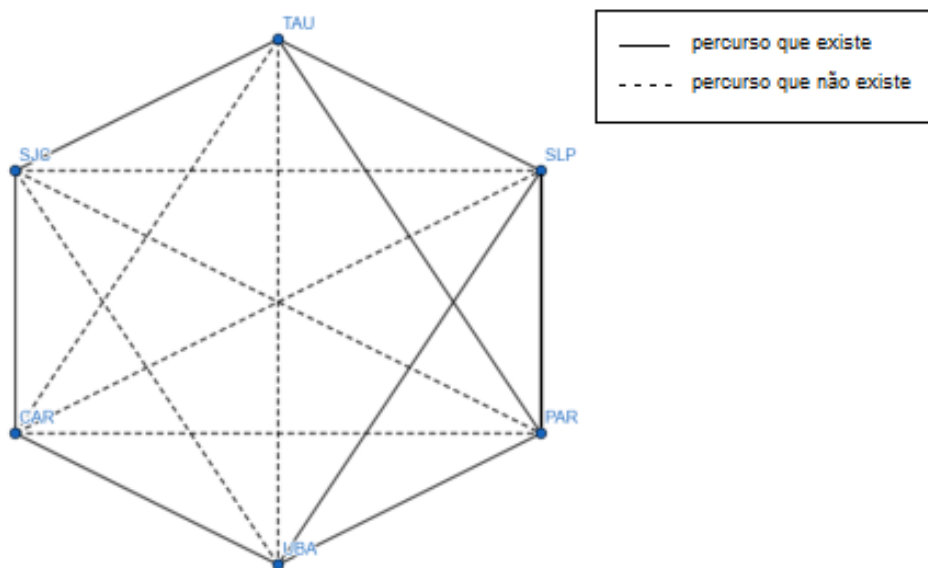


Figura 23: Grafo do Exemplo 6.1

Função objetiva: é minimizar o custo da viagem;

Restrição: o caixeiro tem que visitar todas as cidades, tal que faça um único caminho que visite todas as cidades uma única vez ver.

Índice: Para representar as cidades utilizaremos os índices i, j ($j = 1, \dots, 6$ e $i = 1, \dots, 6$).

Variáveis de decisão:

$$x_{ij} = \begin{cases} 1, & \text{se a cidade } i \text{ visitada antes da } j \text{ (ou se a aresta } (i, j) \\ & \text{esta incluída no ciclo hamiltoniano);} \\ 0, & \text{caso contrário.} \end{cases}$$

Função Objetiva:

Como queremos o menor ciclo hamiltoniano, logo vamos *minimizar*.

O c_{ij} é a distância da cidade i a cidade j .

$$\text{Min } z = \sum_{j=1}^6 \sum_{i=1}^6 c_{ij} x_{ij} \quad (6.1)$$

Restrição de origem:

Queremos que o Ramon sai de cada cidade uma única vez, logo

$$x_{1j} + x_{2j} + \dots + x_{6j} = 1, \quad j = 1, \dots, 6 \quad (6.2)$$

Restrição de destino:

Vai ser necessário que cada cidade tenha apenas um antecessor, a análogo a *restrição de origem*.

$$x_{i1} + x_{i2} + \dots + x_{i6} = 1, \quad i = 1, \dots, 6 \quad (6.3)$$

Restrição de domínio das Variáveis:

$$x_j \in \{0, 1\}, j = 1, \dots, 12 \quad (6.4)$$

Portanto, a primeira tentativa de representar o Exemplo 6.1(Problema do Caixeiro Viajante).

$$\text{Min } z = \sum_{j=1}^6 \sum_{i=1}^6 c_{ij} x_{ij} \quad (6.1)$$

S.a:

$$x_{1j} + x_{2j} + \dots + x_{6j} = 1, \quad j = 1, \dots, 6 \quad (6.2)$$

$$x_{i1} + x_{i2} + \dots + x_{i6} = 1, \quad i = 1, \dots, 6 \quad (6.3)$$

$$x_j \in \{0, 1\}, j = 1, \dots, 6 \quad (6.4)$$

Considerando n cidades, temos:

$$\text{Min } z = \sum_{j=1}^n \sum_{i=1}^n c_{ij} x_{ij} \quad (6.5)$$

S.a:

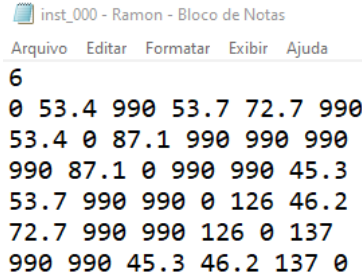
$$\sum_{i=1, i \neq j}^n x_{ij} = 1, \forall j \quad (6.6)$$

$$\sum_{j=1, j \neq i}^n x_{ij} = 1, \forall i \quad (6.7)$$

$$x_j \in \{0, 1\}, j = 1, \dots, n \quad (6.8)$$

Exemplo 6.1(Problema do Caixeiro Viajante) na sintaxe Python-Gurobi.

Neste exemplo a instância está construído da forma mostrada na Figura 24, na qual a primeira linha contém a quantidade de cidades e as outras linhas representa os custos das cidades, pode se observar que a representação dos custos é quase igual o Quadro 1, onde trocamos o infinito do Quadro 1 por um valor alto, no qual foi de 990.



```

6
0 53.4 990 53.7 72.7 990
53.4 0 87.1 990 990 990
990 87.1 0 990 990 45.3
53.7 990 990 0 126 46.2
72.7 990 990 126 0 137
990 990 45.3 46.2 137 0

```

Figura 24: Instância do Exemplo 6.1(Problema do Caixeiro Viajante).

Para a construção do código utilizaremos os estudos do Professor Rafael Lima[15] e o guia do Gurobi[20], com finalidade de implementar o modelo. Nesta construção, utilizaremos as instâncias para importar os dados e este modelo que será apresentado pode não só resolver o Exemplo 6.1 como também qualquer do tipo Problema do Caixeiro Viajante.

Passo 1(Importar o Gurobi):

```
import gurobipy as gp
```

Passo 2: Vamos utilizar a mesma analogia do **Seção 5**, porém nesse exemplo não será necessário rotular e nem criar dicionários na função de importar instância, somente a função `le_dados(nome_arq)` vai importar o vetor *custo* e a quantidade de cidades, como mostrado na Figura 25.

```
1 import gurobipy as gp

1 def le_dados(nome_arq):
2     with open(nome_arq, 'r') as f:
3         linhas = f.readlines()#ler linhas
4         valores = linhas[0].strip().split(' ')
5         qtd_cidades = int(valores[0])
6         del (linhas[0])
7         custo = list()
8         for linha in linhas:
9             valores = linha.strip().split(' ')
10            for i in range(0, len(valores)):
11                valores[i] = float(valores[i])
12            custo.append(valores)
13        #retornar dados
14        return qtd_cidades, custo
```

Figura 25: Função ler_dados para o Exemplo 6.1(PVC) em Python-Gurobi.

Passo 3: Criando uma função de solução `solve_Caixeiro Viajante(nome_arq):`, nesta função importaremos a função de ler dados da instancias do exemplo:

```
cidades, mat_custos = le_dados(nome_arq)
```

Agora, temos que criar os dicionários dos custos e os índices dos pontos de origem e destino, como na **Seção 5**, na qual os índices vão de 1 até a quantidade de cidade e os dicionários que vai ligar as coordenadas aos custos, como ilustrado na Figura 26.

Logo, depois das importações de funções e a criação dos índices e dicionários, vamos iniciar o modelo pelo comando `gp.Model()`. Para implementar o modelo são

necessárias as variáveis de decisões, a x tem que ser para origem e destino, tal que variáveis binárias como na restrição 6.8.

```
x = m.addVars(origens, destinos, vtype = gp.GRB.BINARY)
```

Função objetiva: vamos utilizar desta vez o `prod(custos)` encontrará o produto dos elementos do custos e o sentido é minimizar como definido na equação 6.1.

```
m.setObjective(x.prod(custos), sense = gp.GRB.MINIMIZE)
```

Restrições: para adicionar uma restrição utilizaremos o comando `.addConstrs()`.

- **Restrições de origem(6.2):** como é para todos os i vamos utilizar um *for*, com o `gp.quicksum` somaremos todos os $x[i, j]$ se i for diferente de j e somatoria tem que ser igual a 1:

```
c1 = m.addConstrs( gp.quicksum(x[i, j] for j in destinos if i != j) == 1 for i in origens)
```

- **Restrições de destino(6.3):** mesma lógica das restrições de origem
- ```
c2 = m.addConstrs(gp.quicksum(x[i, j] for i in origens if i != j) == 1 for j in destinos)
```

Para terminar o **Passo 3** executaremos o modelo pelo comando

```
m.optimize()
```

Mas também precisamos saber o que vamos retornar desta função `solve`, nesta implementação vamos retornar a solução ótima e imprimir a matriz binária da variável  $x$ , para imprimira matriz será necessário implementar alguns comandos. Este comando mostrará a matriz onde  $x_{ij} = 1$ , tal que  $i$  origem e o  $j$  o destino, como mostrado na Figura 26. Por exemplo, a primeira linha, onde  $x_{15} = 1$  pois é a linha 1 na coluna 5.

**Passo 4(Resolução:)** para finalizar imprimiremos o resultado e a matriz binária do modelo com os dados do Exemplo 6.1, como ilustrado na Figura 27.

A solução está na Figura 28, podemos observar que a solução está incorreta, pois a solução inclui várias sub-rotas. As sub-rotas estão ilustradas na Figura 29.

```

1 def solve_CaixeiroViajante(nome_arq):
2 #ler dados
3 cidades, mat_custos = le_dados(nome_arq)
4
5 # Índices dos pontos de origem e destino
6 origens = [i + 1 for i in range(cidades)]
7 destinos = [i + 1 for i in range(cidades)]
8 # Dicionário dos custos
9 custos = dict()
10 for i, origem in enumerate(origens):
11 for j, destino in enumerate(destinos):
12 custos[origem, destino] = mat_custos[i][j]
13
14 # iniciar o modelo
15 m = gp.Model()
16
17 # variáveis de decisão
18 x = m.addVars(origens, destinos, vtype=gp.GRB.BINARY)
19
20 # Função Obj
21 m.setObjective(x.prod(custos), sense=gp.GRB.MINIMIZE)
22
23 # restrições de origem
24 c1 = m.addConstrs(gp.quicksum(x[i, j] for j in destinos if i != j) == 1 for i in origens)
25
26 # restrições de destino
27 c2 = m.addConstrs(gp.quicksum(x[i, j] for i in origens if i != j) == 1 for j in destinos)
28
29 # restrições de eliminação de subrotas(MTZ)
30 m.optimize()
31
32 for i in origens:
33 print(f'{i:02d}: ', end='')
34 for j in destinos:
35 print(round(x[i, j].X), ' ', end='')
36 print('')
37
38 return m.objVal

```

Figura 26: Função solve\_CaixeiroViajante com o comando para a matriz binária do Exemplo 6.1(PVC).

```

1 arq = 'inst_000 - Ramon.txt'
2 resultado = solve_CaixeiroViajante(arq)
3 print(resultado)

```

Figura 27: Resolução do Exemplo 6.1(PCV) em Gurobi(Passo 4).

```

Solution count 2: 412 2269.1

Optimal solution found (tolerance 1.00e-04)
Best objective 4.120000000000e+02, best bound 4.120000000000e+02, gap 0.0000%
01:0 0 0 0 1 0
02:0 0 1 0 0 0
03:0 1 0 0 0 0
04:0 0 0 0 0 1
05:1 0 0 0 0 0
06:0 0 0 1 0 0
412.0

```

Figura 28: Solução do Exemplo 6.1(PCV) em Python-Gurobi.

## Reformulando o modelo de otimização e a implementação no Gurobi do Exemplo 6.1(Problema do Caixeiro Viajante)

Novas variáveis e a restrição de sub-rotas:



Figura 29: Sub-rotas do Exemplo 6.1.

Como pode se observar a solução apresentada para esta instância do modelo não possui um único ciclo com todas as cidades que precisam ser visitadas. Dizemos que inclui “sub-rotas”. Neste contexto a solução apresentou três ciclos e estamos querendo um único ciclo que inclua todas as cidades (ciclo hamiltoniano). Necessitamos de uma restrição para não ocorrer sub-rotas, uma possibilidade incluir as restrições propostas por [16] e [18] descrita a seguir. Será necessário definir um novo conjunto de variáveis.

$u_i$  = variáveis contínua

$$u_i - u_j + 6x_{ij} \leq 5, \quad 2 \leq i \neq j \leq 6 \quad (6.9)$$

$$u_i \leq 5, \quad \forall i = 2, \dots, 6 \quad (6.10)$$

**Formulação de Miller-Tucker-Zemlin(1960).** Caso geral

$$u_i - u_j + nx_{ij} \leq n - 1, \quad 2 \leq i \neq j \leq n \quad (6.11)$$

$$u_i \leq n - 1, \quad \forall i = 2, \dots, n \quad (6.12)$$

As restrições 6.9 e 6.10 eliminam as sub-rotas[16] e [4]. Para demonstrar a formulação de **Miller-Tucker-Zemlin**, na qual eliminam as sub-rotas, porém

não eliminam o ciclo hamiltoniano, utilizaremos uma contradição, em que consideraremos que existe a sub-rota:  $r_1 - r_2 - r_3$ . Se aplicarmos a restrição 6.11, temos

$$\begin{aligned} u_{r_1} - u_{r_2} + nx_{r_1r_2} &\leq n - 1 \\ u_{r_2} - u_{r_3} + nx_{r_2r_3} &\leq n - 1 \\ u_{r_3} - u_{r_1} + nx_{r_3r_1} &\leq n - 1 \end{aligned} \tag{6.13}$$

Se somar todas as restrições 6.13, temos que

$$u_{r_1} - u_{r_2} + nx_{r_1r_2} + u_{r_2} - u_{r_3} + nx_{r_2r_3} + u_{r_3} - u_{r_1} + nx_{r_3r_1} \leq 3n - 3 \tag{6.14}$$

Logo, as variáveis  $u_{r_1}$ ,  $u_{r_2}$  e  $u_{r_3}$  se anulam e também podemos isolar o  $n$ :

$$n(x_{r_1r_2} + x_{r_2r_3} + x_{r_3r_1}) \leq 3n - 3 \tag{6.15}$$

Se chamarmos  $h = x_{r_1r_2} + x_{r_2r_3} + x_{r_3r_1}$  e  $p = 3$  da inequação 6.15, temos a inequação 6.16

$$nh \leq pn - p \tag{6.16}$$

Com algumas manipulações algébrica, temos

$$nh \leq pn - p \Rightarrow h \leq p - \frac{p}{n} \tag{6.17}$$

Em que  $p$  representa a quantidade de cidades que a sub-rota visita e  $n$  representa a quantidade de cidades, por conseguinte  $p < n$ , logo  $\frac{p}{n} < 1$  (\*).

Sabemos que para não ter sub-rotas as variáveis  $x_{r_1r_2}$ ,  $x_{r_2r_3}$  e  $x_{r_3r_1}$  não podem ser iguais a 1 ao mesmo tempo, o  $p$  representa a quantidade de cidades que a sub-rota visita e o  $h$  é a soma das variáveis que pertença a sub-rota. Assim, como  $r_1 - r_2 - r_3$  é uma sub-rota, logo  $h = 3$ . Absurdo, por causa de (\*) e 6.17. Por exemplo, considere um  $n = 6$ , na qual possui uma sub-rota  $r_1 - r_2 - r_3$ , logo

$$h \leq p - \frac{p}{n} \Rightarrow 3 \leq 3 - \frac{3}{6}$$

Assim um absurdo, pois  $\frac{p}{n} < 1 \Rightarrow \frac{3}{6} < 1$ .

Portanto, o modelo que representa o problema descrito no Exemplo 6.1(Problema do Caixeiro Viajante):

$$\text{Min } z = \sum_{j=1}^6 \sum_{i=1}^6 c_{ij} x_{ij}$$

S.a:

$$x_{1j} + x_{2j} + \dots + x_{6j} = 1, \quad j = 1, \dots, 6$$

$$x_{i1} + x_{i2} + \dots + x_{i6} = 1, \quad i = 1, \dots, 6$$

$$u_i - u_j + 6x_{ij} \leq 5, \quad 2 \leq i \neq j \leq 6$$

$$u_i \leq 5, \quad \forall i = 2, \dots, 6$$

$$x_j \in \{0, 1\}, j = 1, \dots, 6$$

O modelo de otimização inteira mista para o problema do Caixeiro Viajante considerando  $n$  cidades descrito em (6.19)-(6.23).

$$\text{Min } z = \sum_{j=1}^n \sum_{i=1}^n c_{ij} x_{ij} \tag{6.18}$$

S.a:

$$\sum_{i=1, i \neq j}^n x_{ij} = 1, \forall j \tag{6.19}$$

$$\sum_{j=1, i \neq j}^n x_{ij} = 1, \forall i \tag{6.20}$$

$$u_i - u_j + nx_{ij} \leq n - 1, \quad 2 \leq i \neq j \leq n \tag{6.21}$$

$$u_i \leq n - 1, \quad \forall i = 2, \dots, n \tag{6.22}$$

$$x_j \in \{0, 1\}, j = 1, \dots, n, \tag{6.23}$$

$$u_{i,j} \geq 0, i, j = 1, \dots, n, i \neq j$$

Para a construção do código utilizaremos o código das Figuras 25, 26 e 27 e somente modificando o **Passo 3**, na qual definiremos uma nova variável, está variável é uma restrição que utiliza a **Formulação de Miller-Tucker-Zemlin(1960)** e no lugar de imprimir a matriz binária vai imprimir o ciclo hamiltoniano ótimo.

Para implementar o novo modelo na sintaxe Python-Gurobi serão necessárias

as variáveis auxiliares para eliminar sub-rotas, neste caso adicionaremos um limite superior( $ub$ ) para adicionar a restrição 6.22.

```
x = m.addVars(origens, destinos, vtype = gp.GRB.BINARY)
u = m.addVars(origens[1:], vtype = gp.GRB.INTEGER, ub = cidades - 1)
```

Utilizaremos o comando `.addConstrs()` para definir a nova restrição 6.21, porém tem que colocar as restrição de intervalo do  $u$  também, na qual utilizaremos um *for* para o  $i$  e o  $j$  ir de  $2...n$  tal que  $i$  seja diferente  $j$ :

```
c3 = m.addConstrs(u[i] - u[j] + cidades * x[i, j] <= cidades - 1
for i in origins[1:] for j in destinos[1:] if i != j)
```

Portanto, o código final está descrito nas figuras 25, 30 e 27.

```

1 def solve_CaixeiroViajante(nome_arq):
2 #ler dados
3 cidades, mat_custos = le_dados(nome_arq)
4
5 # Índices dos pontos de origem e destino
6 origens = [i + 1 for i in range(cidades)]
7 destinos = [i + 1 for i in range(cidades)]
8 # Dicionário dos custos
9 custos = dict()
10 for i, origem in enumerate(origens):
11 for j, destino in enumerate(destinos):
12 custos[origem, destino] = mat_custos[i][j]
13
14 # iniciar o modelo
15 m = gp.Model()
16
17 # variáveis de decisão
18 x = m.addVars(origens, destinos, vtype=gp.GRB.BINARY)
19 u = m.addVars(origens[1:], vtype=gp.GRB.INTEGER, ub=cidades - 1)
20
21 # Função Obj
22 m.setObjective(x.prod(custos), sense=gp.GRB.MINIMIZE)
23
24 # restrições de origem
25 c1 = m.addConstrs(gp.quicksum(x[i, j] for j in destinos if i != j) == 1 for i in origens)
26
27 # restrições de destino
28 c2 = m.addConstrs(gp.quicksum(x[i, j] for i in origens if i != j) == 1 for j in destinos)
29
30 # restrições de eliminação de subrotas(MTZ)
31 c3 = m.addConstrs(u[i] - u[j] + cidades * x[i, j] <= cidades - 1 for i in origens[1:] for j in destinos[1:] if i != j)
32
33 # restrições de eliminação de subrotas(MTZ)
34 m.optimize()
35
36 # Constrói o vetor com o circuito,
37 circuito = [1]
38 anterior = 1
39 for ponto in range(cidades):
40 for j in destinos:
41 if round(x[anterior, j].X) == 1:
42 circuito.append(j)
43 anterior = j
44 break
45
46 return m.objVal, circuito

```

Figura 30: Reformulação do Passo 3 - incluindo as restrições para eliminar sub-rotas .

A nova solução ótima está na Figura 31 e a rota que o Damon segurará está na Figura 32.

```

Solution count 3: 430.7 2178.6 2277.4

Optimal solution found (tolerance 1.00e-04)
Best objective 4.307000000000e+02, best bound 4.307000000000e+02, gap 0.0000%
430.7
[1, 2, 3, 6, 4, 5, 1]

```

Figura 31: Ciclo hamiltoniano ótimo para o Exemplo 6.1(PCV) obtido usando o Python-Gurobi.



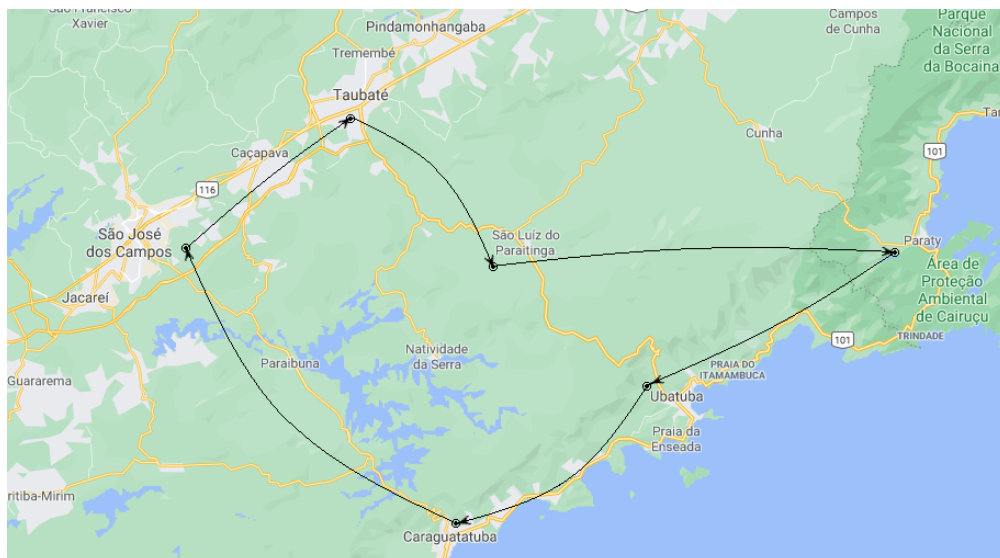


Figura 32: O melhor percurso para o Ramon seguir.

## 7 Conclusão

No presente relatório foram apresentados vários problemas de Otimização Linear contínua e inteira com as suas soluções obtidas pelo sistema de resolução Gurobi e a linguagem de programação Python. A estudo de otimização desenvolvido nos proporcionou a conquistar um raciocínio organizado e lógico. Este raciocínio facilita a compreensão e interpretação dos problemas reais do dia a dia. No desenrolar da pesquisa aprendemos sobre modelos matemáticos de otimização linear, resoluções gráficas e alguns problemas clássicos da otimização, como o Problema da Dieta, Problema do Mochileiro e o Problema Caixeiro Viajante. Estes problemas tiveram a suas soluções obtidas pela ferramenta Gurobi, que pode servir na solução de problemas que possuem grande quantidade de variáveis e restrições.

Portanto, este projeto possibilitou compreender um pouco sobre otimização e suas aplicações e aperfeiçoar o conhecimento sobre ferramentas computacionais como Gurobi com a linguagem Python e proporcionou o entendimento sobre Pesquisa Operacional.

## Referências

- [1] Silvio A de Araujo e Socorro Rangel. *Matemática Aplicada ao Planejamento da Produção e Logística*. SBMAC, 2014.
- [2] Silvio Alexandre de Araujo. *Curso de Otimização Linear Contínua*. Notas de aulas, Departamento de Matemática, Ibilce/UNESP São José do Rio Preto, 2020.
- [3] Marcos Arenales, Vinicius Armentano et al. *Pesquisa operacional*. Elsevier Brasil, 2006.
- [4] E Balas e P Toth. *The Traveling Salesman Problem*. EL Lawler, JK Lenstra, AHG Rinnoy Kan and DB Schmoys. 1985.
- [5] Claudio Aguinaldo Buzzi. *Cálculo Diferencial e Integral II*. Notas de aulas, Departamento de Matemática, Ibilce/UNESP São José do Rio Preto.
- [6] *Cadastrase*. Disponível em: <https://pages.gurobi.com/registration> (acesso em 14/05/2021).
- [7] Professor Ernée. *Método gráfico*. Youtube. 2020. Disponível em: [https://www.youtube.com/watch?v=GfKLi62spYg&list=PLQvfItXkNUvI4L5RodobI\\_0LB7tQRHtoQ](https://www.youtube.com/watch?v=GfKLi62spYg&list=PLQvfItXkNUvI4L5RodobI_0LB7tQRHtoQ) (acesso em 28/01/2021).
- [8] Python Software Foundation. *Documentação do Python 3.7.9*. Disponível em: <https://docs.python.org/3.7/> (acesso em 28/01/2021).
- [9] Haroldo Gambini Santos e Túlio A. M. Toffolo. “tutorial de desenvolvimento de métodos de programação linear inteira mista em python usando o pacote python-mip”. Em: *Pesquisa Operacional para o Desenvolvimento* 11.3 (2019), pp. 127–138. DOI: 10.4322/P0Des.2019.009. Disponível em: <https://www.podesenvolvimento.org.br/podesenvolvimento/article/view/629>.
- [10] Marco Goldbarg. *Otimização combinatória e programação linear-2a edic.* Vol. 2. Elsevier Brasil, 2005.
- [11] Marco Goldbarg, Henrique Pacca Luna e Elizabeth Goldbarg. *Programação linear e fluxos em redes*. Elsevier Brasil, 2015.

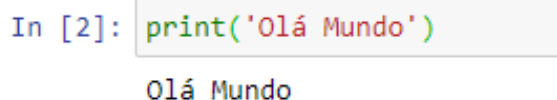
- 
- [12] Prof. Gustavo Guanabara. *Curso de Python 3 - Mundo 1: Fundamentos*. Youtube. 2017. Disponível em: [https://www.youtube.com/watch?v=S9uPNppGsGo&list=PLHz\\_AreHm4dlKP6QQCekuIPky1CiwmdI6](https://www.youtube.com/watch?v=S9uPNppGsGo&list=PLHz_AreHm4dlKP6QQCekuIPky1CiwmdI6) (acesso em 28/01/2021).
- [13] *Gurobi e Anaconda para Windows*. Disponível em: <https://www.gurobi.com/gurobi-and-anaconda-for-windows/> (acesso em 14/05/2021).
- [14] Makswell Seyiti Kawashima. “Relax and cut: Limitantes duais para o problema do caixeiro viajante”. Diss. de mestr. Universidade Estadual Paulista Julio de Mesquita Filho, Instituto de Biociências, Letras e Ciências Exatas, mai. de 2014.
- [15] Prof. Rafael Lima. *Tutorial Gurobi + Python*. Youtube. 2020. Disponível em: <https://www.youtube.com/playlist?list=PL9H2pvV0741ZxpxH36aCC1PUZ6YHL9fYn> (acesso em 28/01/2021).
- [16] Gustavo Valentim Loch. *Problema do Caixeiro Viajante (TSP) - Restrições de Miller-Tucker-Zemlin (MTZ)*. Youtube. 2020. Disponível em: <https://www.youtube.com/watch?v=mQ5TFXXrMtc> (acesso em 25/02/2021).
- [17] Silvano Martello. “Knapsack problems: algorithms and computer implementations”. Em: *Wiley-Interscience series in discrete mathematics and optimization* (1990).
- [18] Clair E Miller, Albert W Tucker e Richard A Zemlin. “Integer programming formulation of traveling salesman problems”. Em: *Journal of the ACM (JACM)* 7.4 (1960), pp. 326–329.
- [19] Pedro Munari. *Solução Gráfica em Programação Linear - Otimização, Pesquisa Operacional, UFSCar*. Youtube. 2020. Disponível em: [https://www.youtube.com/watch?v=SydwBk\\_ZzHQ&feature=emb\\_title](https://www.youtube.com/watch?v=SydwBk_ZzHQ&feature=emb_title) (acesso em 28/01/2021).
- [20] Gurobi Optimization. *Gurobi optimizer quick start guide*. Rel. técn. Technical report, 2014.

- 
- [21] *Quais versões Python são compatíveis com Gurobi?* Disponível em: <https://support.gurobi.com/hc/en-us/articles/360013195212-Which-Python-versions-are-supported-by-Gurobi-> (acesso em 14/05/2021).
- [22] Socorro Rangel. *Introdução à construção de modelos de otimização linear e inteira*. SBMAC, 2005.
- [23] *Registro de licença acadêmica*. Disponível em: <https://www.gurobi.com/downloads/end-user-license-agreement-academic/> (acesso em 14/05/2021).
- [24] Natália S Rodrigues e Socorro Rangel. *Otimização Matemática usando a Linguagem Julia*. XXXI Semana da Matemática, Ibilce/UNESP São José do Rio Preto, 2019.
- [25] Fernando Soares Gomes Taufer e Elaine Correa Pereira. “Aplicação do problema do caixeiro viajante na otimização de roteiros”. Em: *XXXI Encontro Nacional De Engenharia De Produção* (2011).
- [26] James Taylor. *First Look – Gurobi Optimization*. Fev. de 2011. Disponível em: <https://jtonedm.com/2011/03/02/first-look-gurobi-optimization/> (acesso em 28/01/2021).

## Apêndice A. *Python*

### A.1 *Print e Import*

Existem formas de exibir valores numéricos ou texto para usuário e uma destas forma pode ser feito pelo comando ***print()***, como na Figura 33.



```
In [2]: print('Olá Mundo')
Olá Mundo
```

Figura 33: Comando *print()*

Construímos o código do *python* em um módulo e para acessar outros módulos utilizamos a importação, o ***\_import\_()*** é a forma mais comum de invocar a máquina de importação porém não é a única. Existe duas formas de usar o *import*: *from\_import\_()* e *import\_()*. O valor de retorno de *\_import\_()* é usado para executar a operação de vinculação de nome da importinstrução[8].

### A.2 Condições: *If e else*

Nas estruturas condicionais simples(*if*) e compostas(*if e else*) somente será executado um o bloco ou outro, estas condições são utilizadas para determinar se instruções serão ou não executadas[12].

***if*** condição :

*block-True(bloco-verdade)*

*executado varias instruções se a condição for avaliada como verdadeira.*

***else***(opcional):

*block-False(bloco-falso)*

*executará se a condição do ***if*** for falsa.*

### A.3 Estrutura de repetição: *for*

O ***for*** é um laço de repetição ou *loops*, existe também o ***while*** que é um laço de repetição, porém não vamos por enquanto utilizar no decorrer do estudo.

A estrutura de repetição **for** é usualmente executada quando temos um bloco de código que tem como objetivo repetir diversas vezes até uma quantidade determinada pelo programador ou usuário[8].

```
for x in range(0, 3):
 print("We're on time %d"% (x))
```

#### A.4 Variáveis compostas: *list()* e *disc()*

As listas(*list()*) e os dicionários(*disc()*) são dois dos 4 tipos de dados integrados no *Python*. As listas utilizadas para armazenar uma quantidade de itens em uma mesma estrutura é acessada por chaves individuais, os dicionários são similares as listas, porém as diferenças são que os *disc()* é mutável e podemos nomear os índices ou elementos da estrutura[12].

#### A.5 Função: *def*

Em *Python*, as funções são fragmentos de um código que executa algumas operações pré-definidas e que podem ser executadas em algum momento do código quando for solicitado. Utilizamos a palavra **def** quando queremos definir alguma função[8].

```
def nome_da_função(parâmetros):
```