

SME0110 - Programação Matemática

# Trabalho: Um Enigma das Galáxias

Profas. Franklina Toledo e Marina Andretta

Daniel Sá Barreto Prado Garcia - 10374344  
Fernando Gorgulho Fayet - 10734407  
Laura Genari Alves de Jesus - 10801180  
Tiago Marino Silva - 10734748

Parte 1 - corrigida

(08 de novembro)

# 1. Visão Geral

O problema consiste em: apontar um telescópio para  $N$  galáxias uma única vez, minimizando a movimentação necessária de ser operada no telescópio. Este problema, como indicado na própria especificação do trabalho, pode ser comparado ao Problema do Caixeiro Viajante.

Uma distinção válida de se notar entre o problema presente e o do Caixeiro Viajante é que, no segundo, a distância percorrida é a distância euclidiana entre as cidades visitadas, enquanto que no problema presente, a movimentação do telescópio se dá por um par de ângulos, representando a rotação em volta do próprio eixo e a inclinação do telescópio.

No entanto, para fins didáticos, a movimentação do telescópio foi calculada usando a distância euclidiana entre as galáxias, posicionando estas num plano do  $\mathbb{R}^2$ , visto que, via de regra, a minimização desta distância também implica na minimização da movimentação do telescópio, e torna o problema consideravelmente mais simples de ser modelado.

## 1.1 Descrição do Problema e definições

Dado um grafo completo direcionado  $G = (V, A)$ , sendo cada vértice a posição no  $\mathbb{R}^2$  de uma galáxia e as arestas a conexão entre as galáxias, com seu custo igual a distância euclidiana entre elas, deseja-se encontrar um caminho hamiltoniano somando o menor custo possível entre as arestas.

$V$  é o conjunto de vértices,

$$V = \{0, 1, \dots, n\}$$

$A$  é o conjunto de arestas  $(i, j)$ ,

$$i, j \in V$$

$c_{ij}$  é o custo da aresta do vértice  $i$  ao vértice  $j$ ,

$$(i, j) \in A$$

$$x_{ij} = \{1, \text{ se a aresta } (i, j) \in A \text{ for escolhida para o caminho}; 0, \text{ caso contrário}\}$$

$T$  é o conjunto com todos os subconjuntos  $S$  de  $V$  com número de elementos entre 2 e  $n - 2$ .

## 1.2 Função objetiva e restrições

Minimizar:

$$(1) \quad \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

Sujeito a:

$$(2) \quad \sum_{i \in V} \sum_{j \in V, j \neq i} x_{ij} = 1 \quad i = 1, \dots, n$$

$$(3) \quad \sum_{i \in V} \sum_{j \in V, j \neq i} x_{ji} = 1 \quad i = 1, \dots, n$$

$$(4) \quad \sum_{i \in S, j \notin S} x_{ij} \geq 1 \quad \forall S \in T$$

$$x_{ij} \in \{0, 1\} \quad i = 1, \dots, n, j = 1, \dots, n, i \neq j$$

## 1.3 Análise da modelagem

A expressão (1) representa a função a ser minimizada. A idéia é que, de todas as arestas disponíveis, apenas aquelas escolhidas para constituir o caminho possuirão um  $x_{ij} = 1$ . Dessa forma, o custo total do caminho é dado pela somatória de todos os  $x_{ij}$ , multiplicados pelos seus respectivos custos  $c_{ij}$ .

As restrições (2) e (3), por sua vez, garantem que cada vértice terá exatamente uma aresta chegando e uma saindo dele. Se para cada  $i$ , em relação a todos os  $j$ , existe um e apenas um  $x_{ij} = 1$ , significa que haverá uma única aresta saindo daquele vértice. O mesmo raciocínio se aplica às arestas de entrada. Assim, todos os vértices pertencerão ao caminho final e serão percorridos apenas uma vez.

Porém, isso não impede que haja subciclos. Para evitar isso tem-se a restrição (4), que garante que todo subconjunto de vértices  $S$  possuirá pelo menos uma aresta direcionada a um vértice que não pertença àquele subconjunto. A ideia é que se evite ciclos desconexos garantindo que haja pelo menos uma aresta entre eles.

## 2. Resolução via software

### 2.1 Implementação

Para a resolução do modelo, adotou-se a implementação sugerida em [1], fazendo as adaptações necessárias para que o código condissesse com as restrições definidas na sessão 1.2.

A maior alteração deu-se na forma de eliminação dos sub-ciclos desconexos. O artigo utiliza variáveis auxiliares  $y_i$  e  $y_j$  para atingir este objetivo, enquanto o nosso código utilizou-se dos subconjuntos de vértices indicados na restrição (4) (sessão 1.2).

A implementação foi feita utilizando a linguagem python com o pacote **Python-MIP** [2], sendo sua instalação bem simples, basta usar o pip e executar o seguinte comando no terminal:

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

O código completo consta no arquivo *caixeiro.py*

OBS.: o código só foi testado em Ubuntu 20.04

## 2.2 Toy Problem

### 2.2.1 Descrição

Para fins de demonstração, criou-se um Toy Problem com 5 galáxias:

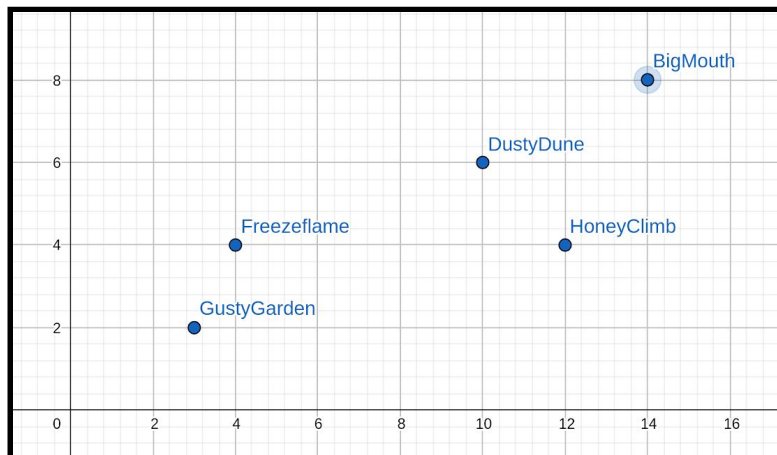
*Galáxias = {'Gusty Garden Galaxy', 'Freeze Flame Galaxy', 'Dusty Dune Galaxy', 'Honeyclimb Galaxy', 'Bigmouth Galaxy'}*

Estas sendo respectivamente posicionadas nos pontos do plano:

$V = \{(3, 2), (4, 4), (10, 6), (12, 4), (14, 8)\}$

A matriz de distâncias possui tamanho  $N \times N$ , sendo cada valor  $c_{ij}$  a distância euclidiana entre o vértice  $i$  e o vértice  $j$ .

A figura a seguir ilustra a distribuição das galáxias no plano:

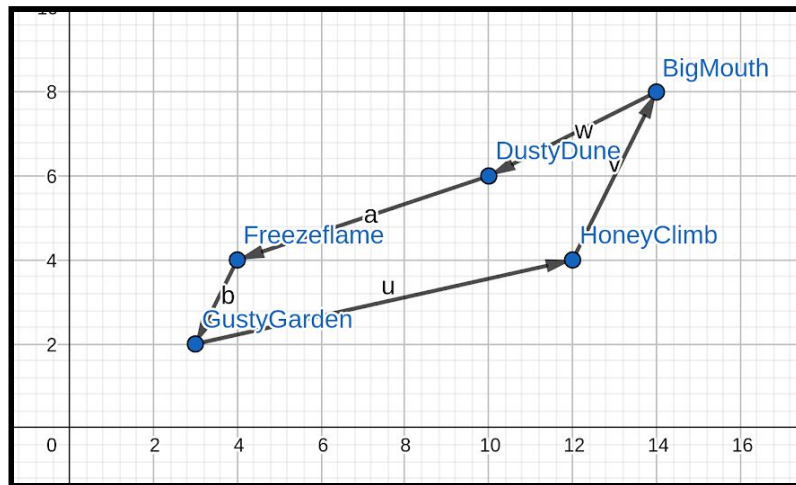


### 2.2.2 Resultados

O caminho apontado pelo algoritmo como mais curto foi:

*Gusty Garden ----> Honey Climb ----> Big Mouth ----> Dusty Dune ----> Freeze Flame  
----> Gusty Garden*

Com um custo total de 26.7244. O caminho encontrado pode ser visto na imagem a seguir:



## Parte 2

(06 de dezembro)

### 3. Alterações na implementação

Na implementação da 1ª etapa, a restrição responsável por evitar o surgimento de subciclos era representada pela expressão (4) (sessão 1.2), e consistia em garantir que todo subconjunto de vértices  $S$  possuiria pelo menos uma aresta direcionada a um vértice que não pertença àquele subconjunto, como sugerido em [3].

Apesar de ser uma forma de modelagem eficaz e de simples compreensão, ela dependia da geração prévia de todos os subconjuntos possíveis de um determinado conjunto de cidades. Esse número cresce exponencialmente, sendo que, para um problema com 29 cidades, era necessário 54secs para gerar todos os subconjuntos possíveis, e para um problema com 38 cidades, 7 horas e 38 minutos. Para  $N = 734$ , tamanho da instância *Uruguay*, seriam necessários  $7,962 * 10^{43}$  anos, ou  $10^{33}$  vezes a idade do universo.

Dado essa limitação, essa restrição foi alterada para a formulação sugerida em [1], e sua nova implementação se deu da seguinte forma:

$$(5) \quad y_i - (n + 1) * x_{ij} \geq y_j - n \quad \forall i \in V \setminus \{0\}, \quad j \in V \setminus \{0, i\}$$
$$y_i \geq 0 \quad \forall i \in V$$

Dessa forma, é garantido que, uma vez que um arco  $(i, j)$  é selecionado, o valor de  $y_j$  é sempre uma unidade maior que o valor de  $y_i$ . Isso se mantém para todos os vértices, com exceção do vértice 0, arbitrariamente selecionado, o que acaba por garantir que haja apenas um ciclo na solução.

Além disso, a implementação foi também recriada utilizando a biblioteca **Python OR-Tools** [4], utilizando o solver **SCIP**, disponibilizado pela própria biblioteca. Esta mudança se deu principalmente pelo fato de esta biblioteca possuir uma melhor documentação, o que facilitou o desenvolvimento.

Por fim, foi também realizado um pré-processamento sobre as instâncias, utilizando a heurística 2-OPT [5].

Modelo completo consta em *caixeiro.py*

Script de pré-processamento consta em *2opt.py*

OBS.: código testado em ambiente Ubuntu 20.04 e Windows 10

### 4. Experimentos

Inicialmente, processou-se cada uma das instâncias sem nenhum pré-processamento, deixando a resolução do problema completamente a cargo do modelo. Enquanto as instâncias pequenas já obtiveram bons resultados deste modo, as instâncias maiores (*Quatar* e *Uruguay*) obtiveram resultados bastante insatisfatórios. Parte disso se dá também pelas limitações das máquinas nas quais o processamento foi feito. No caso da instância do *Uruguay*, por exemplo,



o processamento ocupava 6,8GB de um total de 8GB de RAM. Como consequência, o processamento foi consideravelmente lento, e o número de nós explorados foi muito pequeno.

Visando-se melhorar o desempenho, as instâncias foram submetidas a uma segunda sequência de testes. Dessa vez, efetuou-se um pré-processamento sobre os dados, que consistiu na aplicação da heurística 2-OPT sobre cada uma das instâncias. A solução resultante desta heurística foi utilizada como ponto de partida para o modelo. Neste segundo caso, apesar das limitações técnicas se manterem, observou-se uma melhora considerável, principalmente nas instâncias maiores, visto que o modelo gastava o tempo de processamento explorando soluções mais promissoras.

## 5. Resultados e Discussões

### 5.1 Valores Numéricos

	wi29		dj38		qa194		uy734	
Melhor valor conhecido	27603		6656		9352		73114	
	Sem prépro.	Com 2-opt	Sem prépro.	Com 2-opt	Sem prépro.	Com 2-opt	Sem prépro.	Com 2-opt
Melhor valor obtido	31457,93	30130,14	6662,82	6659,91	22700,13	10953,09	843853	90054,4
Nós Explorados	5486	730	1538	1557	54	9	8	1
Gap total	25,97%	22,47%	5,56%	3,46%	161,62%	22,47%	1170,2%	39,04%
Gap Relativo	13,97%	9,16%	0,10%	0,06%	142,73%	17,12%	1054,2%	23,17%
Tempo de Busca	30'	10'	30'	10'	30'	10'	30'	10'
Tempo de Execução	49'27"	10'	30'31"	10'	29'51"	9'52"	56'41"	3'56"

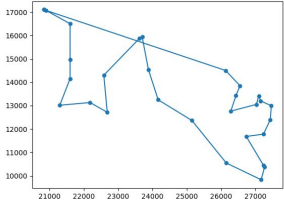
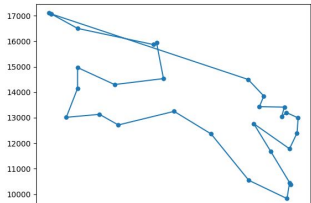
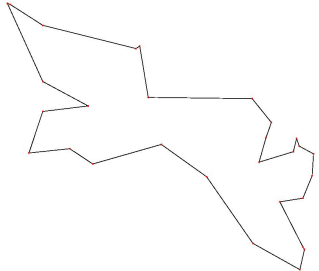
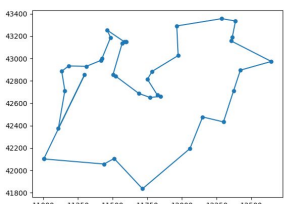
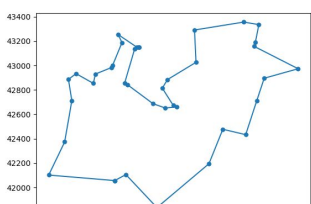
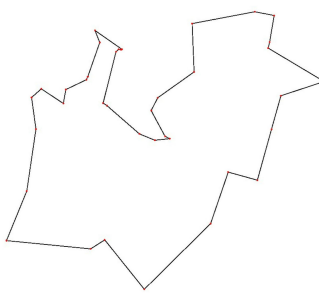
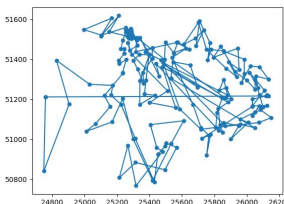
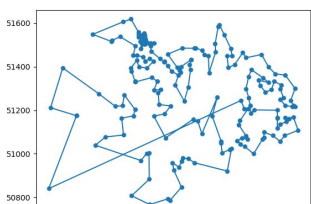
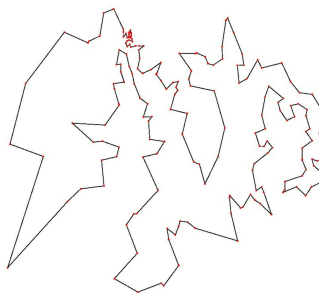
Logo de início, é possível observar que as instâncias menores (*wi29* e *dj38*) apresentaram bons resultados mesmo sem o pré-processamento. No caso do *dj38*, por exemplo, o modelo chegou a encontrar uma solução extremamente próxima da ótima, sem precisar explorar um número tão grande de nós. No caso do *wi29*, o resultado também foi satisfatório, com um GAP relativo de 13,97%.

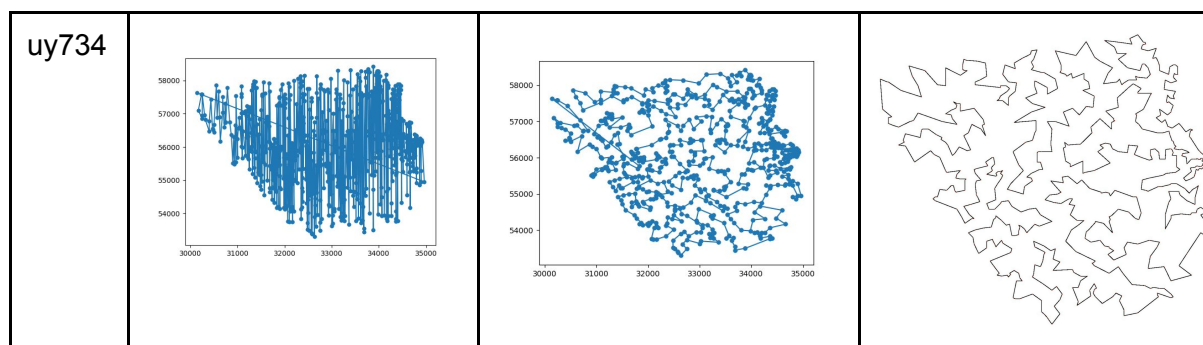
No caso das instâncias maiores, os resultados foram menos satisfatórios, sendo que o GAP final do caso *uy734* foram impressionantes 1170%, muito acima da solução ideal. Isso pode ser observado na representação gráfica da sessão seguinte: o percurso encontrado praticamente fica alternando entre as extremidades do plano, tirando quase que nenhum proveito dos nós mais próximos. Parte desta performance tão baixa pode ser explicada pelas limitações técnicas já mencionadas na sessão 4: a lentidão de processamento e o alto consumo da RAM fizeram com que, no caso do *uy734*, o modelo explorasse apenas 8 nós em um total de 56 minutos de processamento, ou seja, a resposta fornecida foi praticamente a primeira tentativa feita pelo modelo.

Já nos casos onde foi feito o pré-processamento com a heurística 2-OPT, o cenário é diferente. Para as instâncias menores, houve uma pequena melhora (de 2% a 3%), num tempo de processamento 3 vezes menor (apenas 10 minutos). No caso do *dj38* a solução encontrada foi praticamente a ótima, com um GAP relativo de apenas 0,06%.

No entanto, são nas instâncias maiores que os efeitos do 2-OPT foram mais observados. No caso do *qa194*, o modelo encontrou uma solução quase que a metade da solução anterior, explorando apenas 1/6 do número de nós explorados anteriormente. Para o *uy734*, a utilização da heurística foi fundamental. O resultado final foi quase 10 vezes menor que o encontrado anteriormente, resultando num GAP relativo de apenas 23,17%. No entanto, vale notar que o número de nós explorados limitou-se a 1, ou seja, o resultado fornecido foi o resultado da própria heurística. Para se explorar mais possibilidades, seria recomendável uma máquina mais potente ou um tempo maior de processamento.

## 5.2 Representação gráfica

	Sem pré-processamento	Com 2-opt	Melhor solução conhecida
wi29			
dj38			
qa194			



Devido ao tamanho, a sequência exata de cidades visitadas para cada instância não será exposta neste relatório. No entanto, podem ser encontradas na pasta “./código/solutions”, disponível junto com este relatório.

## Referências

[1] Santos H. G, Toffolo, T. A. M, *Tutorial de Desenvolvimento de Métodos de Programação Linear Inteira Mista em Python Usando o Pacote Python-MIP*, Instituto de Ciências Exatas e Biológicas, Departamento de Computação - Universidade Federal de Ouro Preto - UFOP, Ouro Preto-MG, Brasil - 2019

[2] Python-MIP is a modelling tool developed to provide: Ease of use; High performance; Extensibility. Disponível em: <<https://www.python-mip.com/>>

[3] Andretta M., *Problema do Caixeiro Viajante* - ICMC-USP - 2 de março de 2019. Disponível em: <<https://sites.icmc.usp.br/andretta/ensino/aulas/sme0241-1-19/aula3-TSP.pdf>>

[4] GOOGLE. OR-Tools is an open source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer and linear programming, and constraint programming. Disponível em: <<https://developers.google.com/optimization/introduction/overview>>. Acesso em: 3 de dez de 2020.

[5] PYPI. Find, install and publish Python packages with the Python Package Index. Disponível em: <<https://pypi.org/project/py2opt/>>. Acesso em: 3 de dez de 2020.