Um Enigma das Galáxias

SME0110 - Programação Matemática

Julia Carolina Frare Peixoto (Nº USP 10734727) - juliafrare@usp.br
Michelle Wingter da Silva (Nº USP 10783243) - mwingter@usp.br
Matheus Carvalho Raimundo (Nº 10369014) - mcarvalhor@usp.br
Marcelo Duchêne (Nº 8596351) - marcelo.duchene@usp.br

Prof. Franklina Toledo e Prof. Marina Andretta

USP - São Carlos

Dezembro/2020

Índice Analítico

1. Modelagem

Iniciaremos nossa modelagem analisando o problema. O astrônomo deve mover o telescópio o mínimo possível, passando por todas as galáxias, e deseja descobrir qual caminho deve fazer para isso.

Esse problema se assemelha muito ao Problema do Caixeiro Viajante, e a modelagem relembra tal.

1.1 Variáveis e Parâmetros

Vamos denotar em nossa modelagem como L o conjunto das galáxias (coordenadas *x* e *y* das galáxias). Ou seja, L_1 é o *(x, y)* da primeira galáxia, L_2 é o *(x, y)* da segunda, e assim por diante. Esse conjunto *L* é conhecido ao programa, pois é um dado lido e armazenado no início da execução - é um parâmetro. Define-se, então, como *N* o número de galáxias em *L*:

$$N = |L|$$

Além disso, definimos a matriz C como o custo, ou seja, C_{ij} guarda a distância que o telescópio leva para se locomover da galáxia *i* para a galáxia *j*. Esse conjunto *C* também é um parâmetro conhecido ao programa, pois é calculado usando a função de Distância Euclidiana[1]:

$$C_{ij} = \sqrt{(L_j.x - L_i.x)^2 + (L_j.y - L_i.y)^2}$$

Agora partimos para a variável que o programa deve descobrir como solução: o caminho. Vamos descrever esse caminho no formato de uma matriz Z. Basicamente Z_{ij} indica se, estando na galáxia L_i, deve-se mover em seguida para a galáxia L_j (1) ou não (0). Ou seja, para cada linha de Z, apenas uma coluna adota o valor 1, enquanto que todas as outras colunas ficam como 0 - indicando o caminho adequado a ser seguido. Tem-se então:

$$Z_{ij} = \begin{cases} 1, & se\ vai\ de\ L_i para\ L_j \\ 0, & caso\ contrário \end{cases}$$

A matriz Z é desconhecida no começo da execução do algoritmo. Ao fim da execução, ela estará preenchida. Mais semanticamente, com essa variável pode-se traçar um caminho que o astrônomo deve seguir de maneira a minimizar os movimentos no telescópio.

1.2 Função-objetivo

A função-objetivo é bem simples: minimizar os movimentos do telescópio. Para isso, é só somar os caminhos andados, multiplicado pela distância percorrida. Dessa forma, se um caminho não é tomado (0), ele não é somado. Mas se um caminho é tomado (1), a distância dele é somada. Como queremos minimizar essa distância somada, na modelagem tem-se:

$$min: \sum_{i=1}^{N} \sum_{j=1}^{N} C_{ij} \times Z_{ij}$$

Perceba que o parâmetro C é usado porque já se possui ele calculado no programa, enquanto que a variável Z é o que será descoberto como solução.

1.3 Restrições

Primeiro, vamos definir que não é possível sair e entrar na mesma galáxia:

$$\bullet\ Z_{ii} = 0 \qquad , \forall\ i\ \in\ \{1,\ 2,\ ...,\ N\}$$

Agora definimos que, a partir de cada galáxia, deve-se existir exatamente um caminho possível que sai dela:

$$\bullet\ \sum_{j=1}^{N} Z_{ij} = 1 \qquad , \forall\ i\ \in\ \{1,\ 2,\ ...,\ N\}$$

Analogamente, definimos que partindo para qualquer galáxia, deve-se existir exatamente um caminho possível que chegue até ela:

$$\bullet\ \sum_{i=1}^{N} Z_{ij} = 1 \qquad , \forall\ j\ \in\ \{1,\ 2,\ ...,\ N\}$$

Agora precisamos eliminar os **subciclos ilegais**[2]. Para isso usaremos a mesma formulação de Miller–Tucker–Zemlin para o Problema do Caixeiro Viajante[3]. Essa formulação consiste em usar variáveis inteiras extras μ_i que satisfaçam esta restrição:

$$\bullet\ \mu_i - \mu_j + N \times Z_{ij} \leq N - 1 \quad , \forall\ i, j\ \in\ \{2,\ 3,\ ...,\ N\},\ i \neq j$$

Estas variáveis não são relevantes para nós, e serão usadas apenas na eliminação dos subciclos ilegais. A explicação de como isso ocorre é porque estas restrições garantem que:

1. Toda solução viável do problema contém apenas um ciclo com uma sequência de cidades fechadas - forçando que todo ciclo passe pelo ponto L_1;
2. Para toda solução viável, existam valores para μ_i que satisfazem a restrição acima.

Por fim, devemos definir o domínio das nossas variáveis:

- $Z_{ij} \in \{0, 1\}$, $\forall\ i, j \in \{1, 2, ..., N\}$
- $\mu_i \in \mathbb{Z},\ 1 \leq \mu_i \leq N$, $\forall\ i \in \{2, 3, ..., N\}$

No programa, é possível notar que as restrições de domínio são adicionadas durante a própria alocação das variáveis. Não é necessário definir o domínio das variáveis *C* e *L* porque são parâmetros cujos valores já são fixos e conhecidos.

1.4 Resultado

Juntando tudo o que foi descrito acima, tem-se o modelo:

$$min: \sum_{i=1}^{N} \sum_{j=1}^{N} C_{ij} \times Z_{ij}$$

Sujeito a :

$$Z_{ii} = 0 \qquad , \forall\ i \in \{1, 2, ..., N\}$$

$$\sum_{j=1}^{N} Z_{ij} = 1 \qquad , \forall\ i \in \{1, 2, ..., N\}$$

$$\sum_{i=1}^{N} Z_{ij} = 1 \qquad , \forall\ j \in \{1, 2, ..., N\}$$

$$\mu_i - \mu_j + N \times Z_{ij} \leq N - 1 \qquad , \forall\ i, j \in \{2, 3, ..., N\},\ i \neq j$$

$$Z_{ij} \in \{0, 1\} \qquad , \forall\ i, j \in \{1, 2, ..., N\}$$

$$\mu_{ij} \in \mathbb{Z},\ 1 \leq \mu_{ij} \leq N \qquad , \forall\ i \in \{2, 3, ..., N\}$$

2. Toy-problem

O programa foi testado com diversos casos de teste, mas determinamos um problema-exemplo (Toy-problem). O problema-exemplo escolhido se trata de localizações reais de pontos no universo observável, com dados obtidos do website Google Sky[4]. Veja:

Imagem 1: imagem do website Google Sky[4].

Definimos então nosso conjunto *L* como:

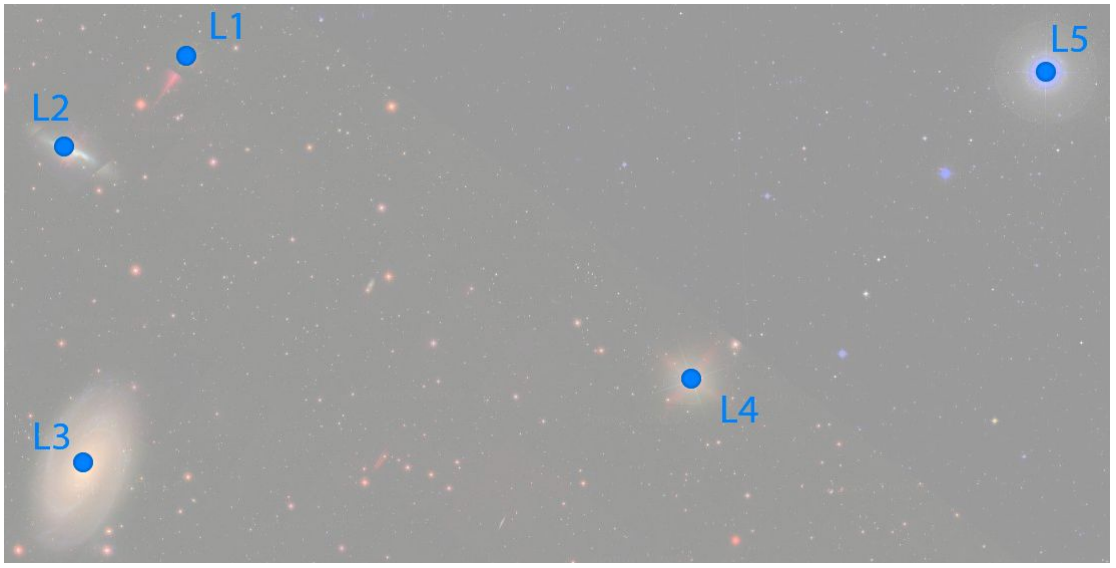

Imagem 2: conjunto *L* a partir de imagem do website Google Sky.

Por fim, temos as seguintes coordenadas (medida: segundos de arco[5]):

L	x	y
1	35598	251501
2	35756	250852
3	35734	248644
4	34936	249270
5	34470	251399

Tabela 1: coordenadas do conjunto *L*.

O arquivo de entrada fornecido já está em formato textual que o programa consegue ler e reconhecer.

2.1 Guia de Execução

Para executar o programa, primeiramente é necessário instalar o ambiente de execução da linguagem Python 3, além de todas as bibliotecas necessárias - foi usada a biblioteca OR-Tools (com *solver* SCIP)[6]. Isso só é feito uma única vez antes da primeira execução. Execute no terminal:

```
> sudo apt-get install python3 python3-pip
> sudo pip3 install -r requirements.txt
```

Para iniciar o programa é muito simples. Em um terminal, execute:

```
> python3 Solucao.py
```

O programa vai solicitar como entrada os dados para gerar os conjuntos *L* e *C*. Se for conveniente, é possível fornecer a entrada a partir de um arquivo. Dessa maneira não é necessário digitar os dados necessários toda vez que for executar o programa. Por exemplo, para fornecer como entrada o problema-exemplo do trabalho, em um terminal execute:

```
> python3 Solucao.py < CasosDeTeste/Toy-problem.tsp
```

2.2 Solução

Ao executar o programa, da maneira descrita na seção anterior para o problema-exemplo escolhido, obtém-se como resposta uma solução-ótima:

```
== Solução Ótima ==
    Interações do Simplex: 16
    Nós explorados: 1
    Valor da função objetivo: 7202.309
    Valor da função objetivo após heurísticas: 7202.309
    Tempo de execução do solver: 0min 0s
    Tempo de execução da heurística: 0min 0s
    Caminho resultado: L1 -> L5 -> L4 -> L3 -> L2 -> L1
```

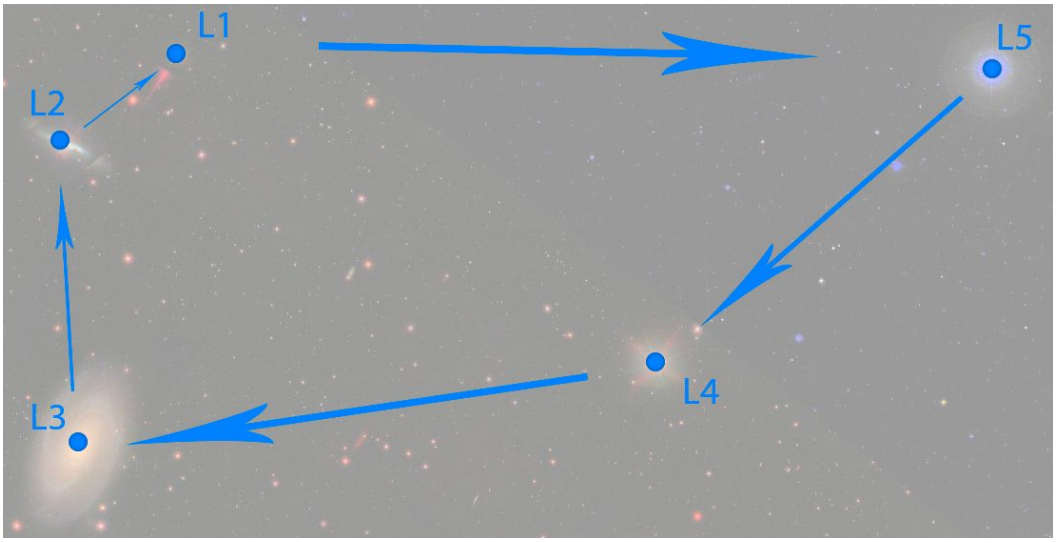

Imagem 3: representação gráfica da resposta do programa.

É evidente que tomar o caminho exatamente contrário a esse (caminhar na direção oposta) também é uma solução-ótima.

3. Remodelagem ou Aprimoramento do Programa

O modelo matemático explicitado nas seções anteriores é funcional, mas não é prático. Ele funciona muito bem para instâncias pequenas, como o problema-exemplo demonstrado neste projeto. Mas quando o número de galáxias aumenta, o tempo de execução do programa também aumenta, chegando a ser impossível só o uso desta modelagem para encontrar alguma solução. Exemplificando, observe que quando temos mais de 20 galáxias, o tempo de execução sobe exponencialmente, e para apenas algumas centenas de pontos o programa pode levar anos para ser executado: e no mundo real não temos todo esse tempo.

Sendo assim, é necessário **remodelar** o problema, de forma a tornar mais prático seu uso, **ou** usar de outras técnicas para que a solução encontrada se aproxime ao máximo possível de uma solução ótima, executando em um tempo mais factível.

Infelizmente, este problema está entrelaçado ao clássico Problema do Caixeiro Viajante - bastante conhecido na Ciência da Computação e no qual ainda não existe uma modelagem perfeita capaz de executar em tempo factível. Dessa maneira, não é possível para este projeto encontrar exatamente a única solução ótima.

Por outro lado, podemos utilizar técnicas de programação matemática para encontrar uma solução que **se aproxime** da solução ótima. Não é garantido que a solução que vamos encontrar vai ser a ótima: muito provavelmente não será. Mas é garantido que o tempo de execução será factível e a solução encontrada será minimamente boa - e é isto o que vamos fazer agora.

3.1 Solução Inicial

A técnica que usaremos será fornecer para nosso modelo uma **solução inicial**. Dessa maneira, o modelo terá apenas que reajustar os valores já fornecidos, poupando parte do tempo de execução. Além disso, será limitado o tempo de execução, de forma que o algoritmo pare de executar e se contente com a melhor resposta que ele foi capaz de obter naquele tempo limite. Para este projeto, este tempo será de **10 minutos**.

O desafio agora é descobrir uma solução inicial, pois essa solução deve ser gerada pelo nosso programa interativamente. Para isso, podemos usar uma função aleatória (como por exemplo, partindo de L_0, aleatoriamente ir para qualquer outro L_j), ou uma função de menor distância (partindo de L_0, ir para o L_j mais próximo). Mas optamos por um algoritmo mais complexo que se baseia em abordagens gulosas.

Basicamente, para construir nosso algoritmo personalizado, consideramos que existem dois viajantes no nosso esquema, e eles estão competindo pelas cidades. Os dois viajantes começam em L_0. A cada interação, o viajante 1 escolhe o ponto mais próximo dele disponível e vai para lá (de L_0 para o L_j mais próximo), e o viajante 2 escolhe o ponto mais próximo dele disponível e vai para lá (de L_0 para o segundo L_j mais próximo). Sendo assim, os viajantes gulosamente competem pelas cidades, andando pelas mais próximas o máximo possível. Perceba que, dessa maneira, dois caminhos estão sendo formados. No fim do algoritmo, os viajantes se encontram em cidades diferentes, e finalmente elas são interligadas, formando um caminho único - nossa solução inicial.

3.2 Heurística 2-OPT

Fornecendo a solução inicial da forma descrita acima e limitando o tempo de execução de nosso programa, conseguimos obter resultados impressionantemente superiores. Contudo, para os casos em que não atingimos a solução-ótima, é possível melhorar e aproximar ainda mais a solução viável atingida da solução-ótima. Sendo assim, usaremos uma heurística a mais: a otimização 2-opt de busca local.

Esta heurística será usada como complemento à solução obtida anteriormente. O algoritmo é simples:

```
função trocar(caminho, i, j):
        alocar vetor_caminho novo_caminho[]
        incluir caminho[início : i] em novo_caminho
        incluir inverso de caminho[i : j + 1] em novo_caminho
        incluir caminho[j+1 : fim] em novo_caminho
        retornar novo_caminho;

repetir enquanto houver melhorias no vetor caminho:
        melhor_distancia = caminho.distanciaTotal()
        para i de 0 até n - 2:
                para j de i + 1 até n - 1:
                        novo_caminho = trocar(caminho, i, j)
                        nova_distancia = novo_caminho.distanciaTotal()
                        se nova_distancia < melhor_distancia, então:
                                caminho = novo_caminho
                                melhor_distancia = novo_caminho.distanciaTotal()
                                retornar para estrutura enquanto
```

Vale lembrar que este algoritmo deve ter o tempo limitado também, caso contrário, a depender de como nosso *solver* deixou a solução final, ele pode acabar executando por tempo inviável - e neste trabalho queremos limitar o tempo de execução do programa. Sendo assim, foi optado por utilizar apenas o que sobrou do tempo limitado para o *solver* somado à 1 minuto. Ou seja, se o *solver* executou por apenas 4 minutos dos 10 minutos de limite, o 2-OPT executará por 7 minutos (os 6 minutos que restaram do *solver* + 1 minuto extra). Obviamente, pode ser que ele acabe ainda antes disso se não conseguir fazer mais otimizações (por exemplo, já conseguiu chegar na solução-ótima).

4. Experimentos

Dado as modificações feitas acima, agora é necessário testar o programa, usando como base maior número de pontos - e não apenas 5, como no problema-exemplo. Para isso, usaremos como banco de dados as instâncias disponibilizadas no repositório "National Traveling Salesman Problems" da Universidade de Waterloo[6]. Os pontos representados, na verdade, não se referem à galáxias, mas sim à cidades - apesar de que o propósito é o mesmo. As instâncias utilizadas serão:

- Western Sahara (29 cidades);
- Djibouti (38 cidades);
- Qatar (194 cidades);
- Uruguay (734 cidades).

Para cada uma das instâncias acima, a solução obtida e respectiva imagem gerada serão explicitadas. Na imagem gerada, a **estrela** indica o início do caminho (L_0) enquanto que o **'x' vermelho** indica o fim do caminho (última cidade visitada). Além disso, a distância euclidiana calculada foi arredondada para os resultados baterem com o repositório, em que esta também foi arredondada.

O **ambiente de execução** tem influência nestes resultados. Por exemplo, um processador bem mais potente é capaz de realizar mais interações do Simplex do que um processador bem menos potente em um mesmo intervalo de tempo. Sendo assim, foi instanciada uma mesma máquina na nuvem para processar todos esses dados. Tal máquina possui as seguintes especificações:

- ❑ **CPU:** Intel Hexa-core (permite execução paralela das instâncias)
- ❑ **GPU:** NVIDIA Tesla P100 (caso a OR-Tools utilize otimização em GPU)
- ❑ **Memória RAM:** 16GB (para evitar paginação em disco)

4.1 Western Sahara

Executando o programa para o conjunto de pontos *Western Sahara*, obtemos a seguinte solução:

```
> python3 Solucao.py < CasosDeTeste/Waterloo-WesternSahara-wi29.tsp
== Solução Viável ==
    Interações do Simplex: 1161811
    Nós explorados: 79817
    Valor da função objetivo: 27623.000
    Valor da função objetivo após heurísticas: 27603.000 (Waterloo: 27603)
    Tempo de execução do solver: 10min 0s
    Tempo de execução da heurística: 0min 0s
    Caminho resultado: (oculto para melhor visualização abaixo)
```

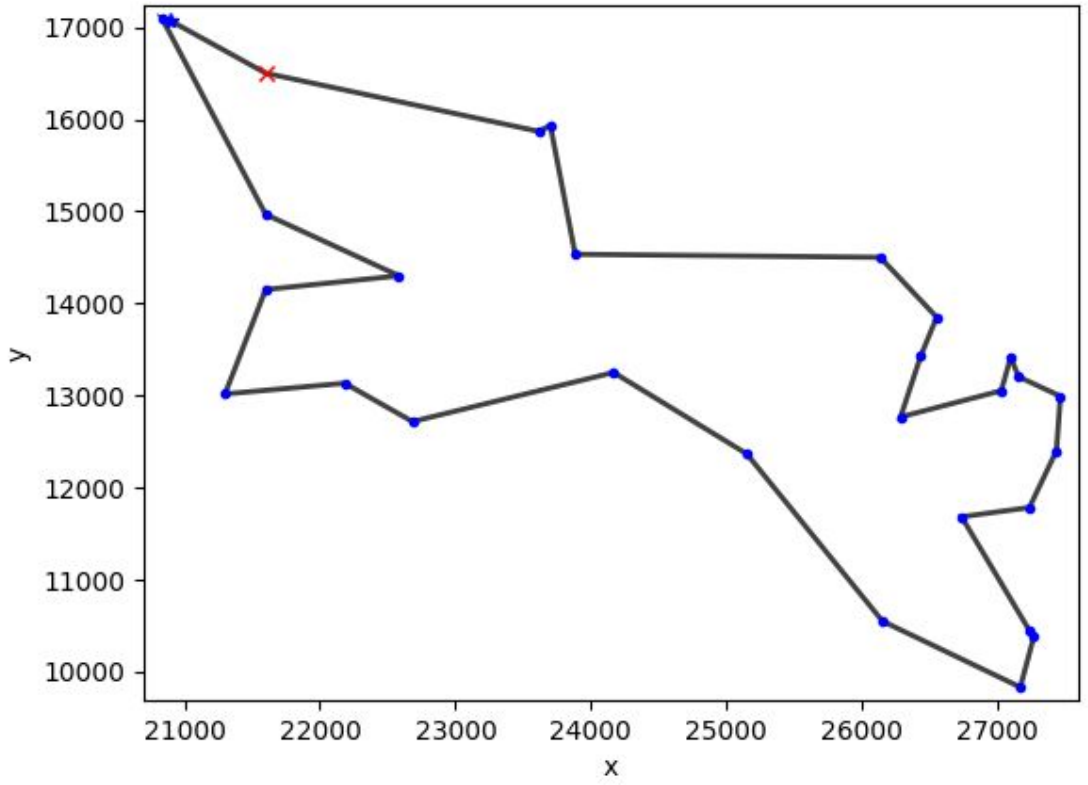

Imagem 4: Solução gerada para *Western Sahara*.

4.2 Djibouti

Executando o programa para o conjunto de pontos *Djibouti*, obtemos a seguinte solução:

```
> python3 Solucao.py < CasosDeTeste/Waterloo-Djibouti-dj38.tsp
== Solução Ótima ==
    Interações do Simplex: 958072
    Nós explorados: 42714
    Valor da função objetivo: 6656.000
    Valor da função objetivo após heurísticas: 6656.000 (Waterloo: 6656)
    Tempo de execução do solver: 5min 51s
    Tempo de execução da heurística: 0min 0s
    Caminho resultado: (oculto para melhor visualização abaixo)
```

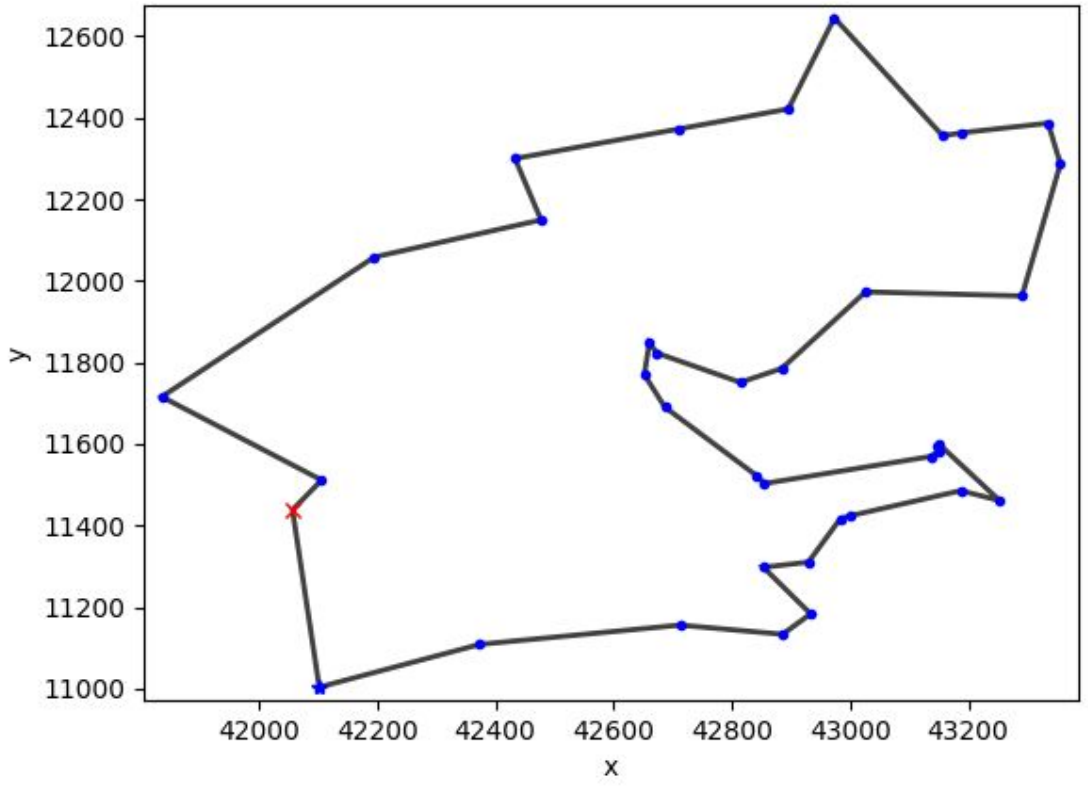

Imagem 5: Solução gerada para *Djibouti*. Eixos *x* e *y* invertidos para assemelhar-se a imagem do repositório público.

4.3 Qatar

Executando o programa para o conjunto de pontos *Qatar*, obtemos a seguinte solução:

```
> python3 Solucao.py < CasosDeTeste/Waterloo-Qatar-qa194.tsp
== Solução Viável ==
    Interações do Simplex: 13521
    Nós explorados: 7
    Valor da função objetivo: 12303.000
    Valor da função objetivo após heurísticas: 10051.000 (Waterloo: 9352)
    Tempo de execução do solver: 9min 51s
    Tempo de execução da heurística: 0min 17s
    Caminho resultado: (oculto para melhor visualização abaixo)
```

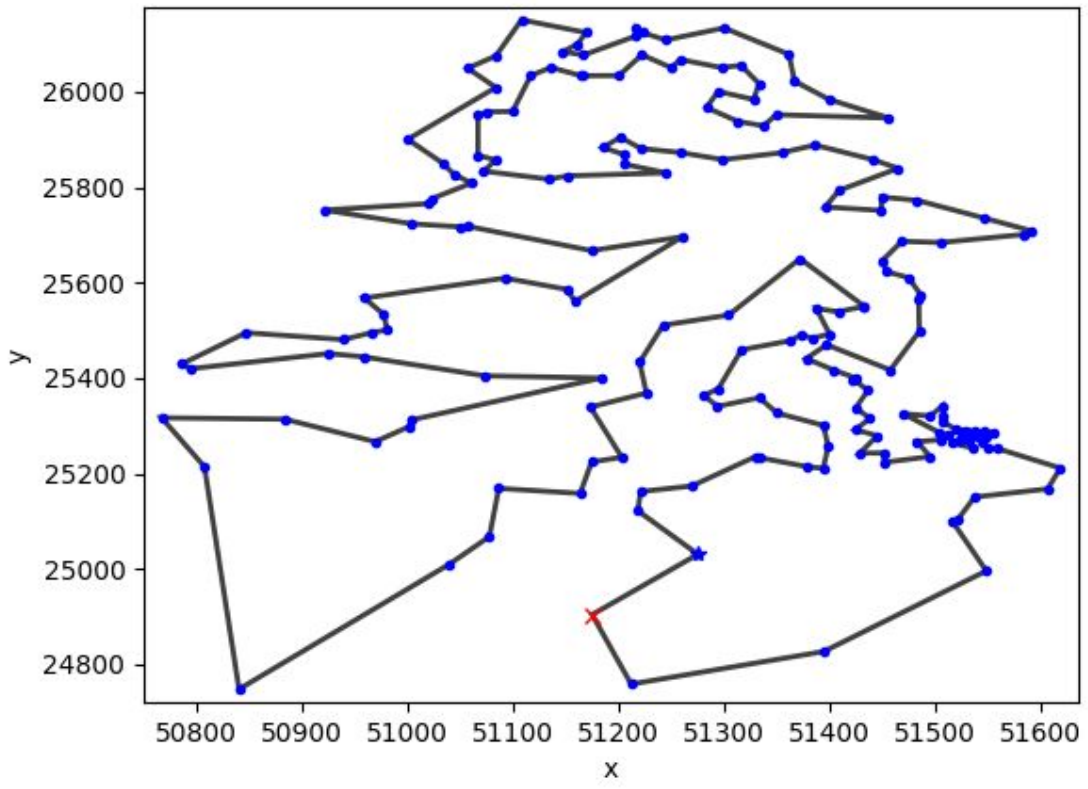

Imagem 6: Solução gerada para *Qatar*. Eixos *x* e *y* invertidos para assemelhar-se a imagem do repositório público.

4.4 Uruguay

Executando o programa para o conjunto de pontos *Uruguay*, obtemos a seguinte solução:

```
> python3 Solucao.py < CasosDeTeste/Waterloo-Uruguay-uy734.tsp
== Solução Viável ==
    Interações do Simplex: 2790
    Nós explorados: 1
    Valor da função objetivo: 106215.000
    Valor da função objetivo após heurísticas: 99352.000 (Waterloo: 79114)
    Tempo de execução do solver: 4min 10s
    Tempo de execução da heurística: 6min 53s
    Caminho resultado: (oculto para melhor visualização abaixo)
```

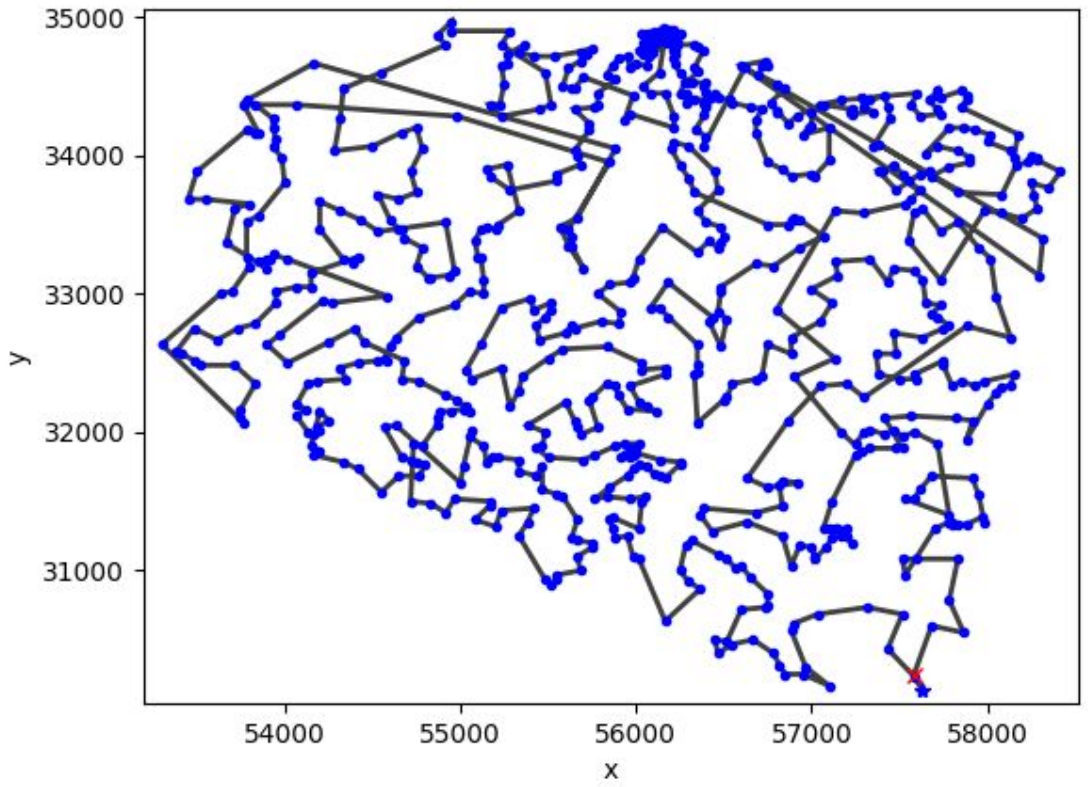

Imagem 7: Solução gerada para *Uruguay*. Eixos *x* e *y* invertidos para assemelhar-se a imagem do repositório público.

Um segundo experimento foi realizado com esta instância, estendendo o tempo limite de execução do programa para 30 minutos. Foi possível alcançar, neste segundo experimento, um valor da função objetivo de 92632.000.

5. Conclusão

Nesta seção será feita uma comparação dos resultados obtidos na seção anterior com os resultados públicos do repositório da Universidade de Waterloo[7]. Tal comparação será feita tanto em relação ao caminho - analisando visualmente as imagens geradas como solução, como em relação ao GAP calculado - ou seja, a relação entre os limitantes primais do nosso programa e do repositório da Universidade de Waterloo.

A equação usada para cálculo do GAP será:

$$GAP = 100 \times \frac{LP_{nosso} - LP_{site}}{LP_{site}}$$

Note, porém, que nesta análise não será considerado o "erro externo". Definimos o "erro externo" como qualquer afastamento da solução-ótima provocado por fenômenos não controlados pelo projeto (fenômenos externos). Por exemplo:

- Imprecisão computacional nos cálculos provocada por tipagem de variáveis e arquitetura computacional em execução;
- Imprecisão nos dados disponibilizados no repositório.

Dessa maneira, a instância **Western Sahara** obteve uma excelente solução. O *solver* não conseguiu encontrar uma solução-ótima, mas encontrou uma factível. Contudo, ao executar a heurística 2-OPT, a solução foi otimizada. Analisando a imagem gerada e a disponível no repositório, notamos inclusive que tal é a **solução-ótima**. Ao realizar o cálculo do GAP com o repositório público, obtém-se o valor **0.0**.

A instância **Djibouti** também obteve uma excelente solução. Inclusive, o *solver* conseguiu encontrar diretamente a **solução-ótima**. Ao realizar o cálculo do GAP com o repositório público, obtém-se o valor **0.0**.

A instância **Qatar** também obteve uma excelente solução. O *solver* não conseguiu encontrar uma solução-ótima e encerrou antes do fim do tempo limite, enquanto que a heurística 2-OPT conseguiu melhorar a solução encontrada. Ao realizar o cálculo do GAP com o repositório público, obtém-se o valor **7.47**.

Por fim, a instância **Uruguay** obteve uma boa solução. O *solver* não conseguiu encontrar uma solução-ótima e encerrou, enquanto que a heurística 2-OPT foi interrompida por conta do tempo limite de execução. Sendo assim, foi encontrada uma **solução factível**. Ao realizar o cálculo do GAP com o repositório público, obtém-se o valor **25.58**. Apesar de não ter sido encontrada aqui a solução-ótima, considerando o valor do GAP podemos concluir que a solução factível encontrada já é consideravelmente boa e razoavelmente próxima da solução-ótima. Isso se comprova no segundo experimento realizado na instância **Uruguay**, em que o tempo limite é estendido para 30 minutos e conseguimos alcançar um melhor resultado, com GAP de **17.09**.

Interessante: neste projeto estamos utilizando a biblioteca OR-Tools com o *solver* SCIP. Ao usar a OR-Tools com o *solver* CP-SAT, **todas** as instâncias analisadas acima obtiveram resultados superiores e ainda mais rapidamente - em um computador com CPU de 16 cores. Isso acontece porque o *solver* CP-SAT suporta processamento paralelo, enquanto que o SCIP não. A instância **Djibouti**, por exemplo, conseguiu encontrar a solução-ótima em apenas 20s de execução ao usar o CP-SAT neste ambiente.

6. Referências Bibliográficas

1. “Distância Euclidiana”, disponível no website <https://en.wikipedia.org/wiki/Euclidean_distance> em 25 de outubro de 2020.
2. “Variáveis inteiras como ferramenta de modelagem”, por Prof. Franklina Toledo (aula do dia 09/09/2020), disponível no website <https://edisciplinas.usp.br/> em 25 de outubro de 2020.
3. “The Travelling Salesman Problem”, disponível no website <https://en.wikipedia.org/wiki/Travelling_salesman_problem> em 25 de outubro de 2020.
4. Google Sky, disponível no website <https://www.google.com/sky/#latitude=69.27276405942452&longitude=-32.94125194551579&zoom=8> em 30 de outubro de 2020.
5. “Conhecimento: Aprenda a estimar as distâncias que envolvem os objetos no céu”, disponível no website <https://www.apolo11.com/distancias_no_ceu.php> em 30 de outubro de 2020.
6. “OR-Tools”, disponível no website <https://developers.google.com/optimization> em 30 de outubro de 2020.
7. “National Traveling Salesman Problems”, disponível no website <http://www.math.uwaterloo.ca/tsp/world/countries.html> em 2 de dezembro de 2020.
8. “2-opt”, disponível no website <https://en.wikipedia.org/wiki/2-opt> em 2 de dezembro de 2020.