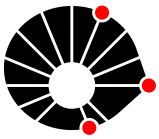
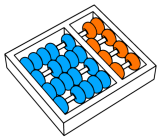




Algoritmos e Heurísticas para o Problema do Caixeiro Viajante com Minimização de Energia

J. P. B. Viccari *F. K. Miyazawa*

Relatório Técnico - IC-PFG-21-40
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Algoritmos e Heurísticas para o Problema do Caixeiro Viajante com Minimização de Energia

João Pedro Viccari* Flávio Keidi Miyazawa†

10 de fevereiro de 2022

Resumo

O propósito deste trabalho é o de analisar o comportamento de algoritmos exatos e heurísticas para uma versão do Problema do Caixeiro Viajante com Minimização de Energia, investigando-o na abordagem de Programação Linear Inteira. Para a realização dos experimentos foi usada a biblioteca de modelagem e otimização Gurobi em um ambiente Jupyter Notebook.

1 Introdução

A circunstância do Problema do Caixeiro Viajante com Minimização de Energia, daqui em diante abreviada para PCVME, discutida nas seções a seguir considera uma malha de pontos distribuídos através de um mapa bidimensional. Nesta malha, um ponto é informado como sendo representativo do depósito de onde um veículo sairá e todos os outros pontos contêm cargas que devem ser obrigatoriamente coletadas por ele. Este então deve passar por todos os pontos coletando as cargas e por fim retornando ao depósito. O caminho entre dois pontos está associado a uma função de energia gasta pelo veículo, que relaciona o custo de transporte ao peso carregado pelo veículo, ambas grandezas vinculadas somente a este trecho do trajeto. Então, solucionar este problema consiste sumariamente em minimizar uma função de custo associada ao trajeto total percorrido pelo veículo. Para obter uma solução ótima, o formularemos como um problema de Programação Linear Inteira, cuja função objetivo será minimizar uma função linear de gasto de energia e as restrições lineares permitirão que as variáveis assumam apenas valores que estejam associados a soluções viáveis do problema.

Em Teoria da Computação, Programação Linear (PL) é usada para a resolução de problemas de otimização em que suas restrições são expressas como relações lineares e cuja modelagem assume a seguinte forma canônica:

$$\textit{Encontrar} \quad \mathbf{x}$$

$$\textit{Que maximiza} \quad \mathbf{c}^T \cdot \mathbf{x}$$

*Aluno do Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP.
†Professor do Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP.

$$Sujeito\ a \quad \mathbf{A} \cdot \mathbf{x} = \mathbf{B};\ \mathbf{x} \geqslant 0,$$

sendo $\mathbf{x}$ o conjunto de variáveis que solucionam determinada instância do problema e $\mathbf{c}$, $\mathbf{b}$ e $\mathbf{A}$ são respectivamente dois vetores e uma matriz de coeficientes que configuram tal instância a ser otimizada.[1] Note que problemas de minimização, como o PCVME, podem ser descritos de maneira análoga à forma acima explicitada, uma vez que maximizar $c^T \cdot x$ é equivalente a minimizar $-c^T \cdot x$.

A Programação Linear Inteira (PLI), por sua vez, consiste na resolução de problemas abordados por Programação Inteira em que o conjunto de variáveis deve assumir valores inteiros respeitando um conjunto de restrições lineares. Dentre os principais métodos de solução de problemas de PLI, temos a utilização do método Branch-and-Bound com a resolução de programas lineares.[2]

Temos acima um problema que corresponde a uma variação do *Travelling Salesman Problem* (TSP) e para realizar sua enunciação na seção seguinte, vamos nos basear na formulação de *Miller-Tucker-Zemlin* (MTZ)[3] para o TSP, de maneira que passaremos a considerar um conjunto de restrições que visam a eliminação de subrotas no percurso do veículo.

Ao longo deste projeto vamos propor uma alternativa heurística ao algoritmo exato inicialmente descrito e investigar como e em quais conjuntos de instâncias o uso destas heurísticas é mais apropriado em relação ao do algoritmo exato.

2 Formulação

Como tratamos de um problema em que temos diversos pontos de coleta e rotas que os ligam, a formulação do problema será feita utilizando a linguagem da Teoria de Grafos. Como dados de entrada, temos o seguinte:

- $G = (V, E)$, um grafo completo direcionado;
- $V = \{0, 1, ..., n\}$, o conjunto de vértices do grafo, sendo o vértice correspondente ao depósito sempre indexado pelo vértice 0;
- E, o conjunto de arestas;
- $c_{i,j}$, o custo associado a percorrer a aresta $(i, j) \in E$;
- w_i, o peso do item a ser coletado no vértice $i \in V$;
- P, o peso do veículo.

Para formular o PCVME como um problema linear inteiro, utilizaremos as seguintes variáveis:

- $x_{i,j} \in \{0, 1\}$ é uma variável binária que terá valor 1 se o veículo percorre a aresta $(i, j) \in E$ e 0 em caso contrário;
- $p_{i,j}$, terá o peso do veículo somado ao total das cargas sendo transportadas ao longo da aresta $(i, j) \in E$.

Denotaremos por S, a soma dos pesos de todas as cargas a serem coletadas. Note que o TSP é um caso particular do PCVME quando $P = 1$ e $w_i = 0$ para todo $i \in V \setminus \{0\}$. Finalmente, a formulação do problema é dada por:

$$min \sum_{(i,j) \in E} c_{i,j} \cdot p_{i,j}, \tag{1}$$

$$\sum_{i=1,\, i \neq j}^{n} x_{i,j} = 1, \;\; \forall j \in V, \tag{2}$$

$$\sum_{j=1,\, j \neq i}^{n} x_{i,j} = 1, \;\; \forall i \in V, \tag{3}$$

$$u_i - u_j + n \cdot x_{i,j} \leqslant n - 1, \;\; 1 \leqslant i \neq j \leqslant n, \tag{4}$$

$$\sum_{j \in \delta^+(i)} p_{i,j} = \sum_{j \in \delta^-(i)} p_{i,j} + w_i, \; \forall i \in V \setminus \{0\}, \tag{5}$$

$$\sum_{j \in \delta^+(0)} p_{0,j} = P, \tag{6}$$

$$p_{i,j} \leqslant [S + P] \cdot x_{i,j}, \; \forall (i,j) \in E, \tag{7}$$

$$x_{i,j} \in \{0, 1\}, \; \forall (i,j) \in E, \tag{8}$$

$$p_{i,j} \geq 0, \; \forall (i,j) \in E. \tag{9}$$

Conforme exposto, o objetivo é o de minimizar a função de custo (1) do gasto de energia ao longo de todas as arestas que são percorridas pelo veículo. O conjunto de restrições impostas aos algoritmos nos garantem que o veículo sai (2) e entra (3) de todos os vértices somente uma vez. Note que obedecer às restrições (2) e (3) permite que sejam encontradas rotas cíclicas que satisfazem o problema. No entanto, tais rotas não são ciclos hamiltonianos, mas caminhos cíclicos desconexos que não satisfazem a especificação do problema de coletar cargas em cada um dos pontos. Para tanto, adicionamos a restrição (4) introduzida por Miller, Tucker e Zemlin, que nos garante a eliminação dos possíveis subciclos do trajeto (4). Além disso, asseguramos com as outras restrições que o peso carregado ao longo de uma aresta que entra em um vértice é sempre igual ao peso carregado ao longo de uma aresta que sai de um vértice mais o peso da carga contida naquele vértice, com exceção do vértice correspondente ao depósito (5). Garantimos também que a soma do peso transportado ao longo de todas as arestas que saem do depósito é igual ao peso do veículo (6) e que o peso transportado ao longo de uma aresta é sempre menor ou igual ao peso do veículo mais a soma de todas as cargas a serem coletadas (7).

3 Heurísticas

3.1 Vizinho Mais Próximo Probabilístico

O primeiro algoritmo empregado na obtenção de uma solução heurística foi uma variação do algoritmo do Vizinho Mais Próximo (VMP), que por sua vez é um algoritmo guloso, ou seja, um algoritmo que constrói uma solução de maneira iterativa e a cada iteração faz escolhas que buscam minimizar o custo da solução parcial sendo construída, na tentativa de se obter uma solução de valor próximo ao ótimo. Historicamente uma das primeiras tentativas de resolução do TSP, o VMP é um algoritmo de simples implementação mas que apresenta a fragilidade de raramente obter uma solução ótima devido a sua característica de não levar em consideração o escopo maior dos pontos a serem visitados, não capturando assim caminhos menos custosos e facilmente perceptíveis.

O algoritmo em pseudo-código do VMP, como enunciado por Masehian e Ellips [4], é dado a seguir:

Algorithm 1 Nearest Neighbor Heuristic

1: Seja i um vértice aleatório do grafo
2: $V \leftarrow \{0, 1, 2, ..., n\} \setminus \{i\}$
3: **while** $V \neq \emptyset$ **do**
4: Seja $j \in V$ um vértice tal que $c_{i,j}$ é mínimo
5: Insira a aresta (i, j) na solução
6: $V \leftarrow V \setminus \{j\}$
7: $i \leftarrow j$
8: **end while**
9: Insira a aresta (j, i_0) na solução, sendo i_0 o vértice i escolhido na linha 1

Analisando o pseudo-código, podemos descrever o VMP, em linhas gerais, como um algoritmo que inicia a rota em um vértice aleatório e cria uma lista de vértices a visitar contendo todos os vértices do grafo exceto o inicial. Então escolhe entre as arestas que saem deste vértice uma com o menor custo. Esta é adicionada à rota e seu vértice destino é removido da lista de vértices a visitar. Este processo repete-se até que esta lista esteja vazia. Por fim, a aresta que parte do último vértice visitado e chega ao vértice inicial é adicionada à solução para que seja formado um ciclo hamiltoniano.

Neste projeto vamos aplicar uma variação do VMP que atribui probabilidades de escolha aos próximos vértices a visitar, em vez de simplesmente escolher o vértice cuja aresta até ele apresenta o menor custo. Isto é feito por ordenar as arestas que partem do vértice atual de acordo com seus custos, atribuir a elas pesos de escolha inversamente proporcionais a tais custos e segundo eles, realizar uma escolha aleatória usando uma função *weighted_random* que retorna o vértice escolhido. Esta função que aleatoriamente faz uma escolha entre as arestas de acordo com seus pesos será detalhada na seção 4.3, em que são esclarecidos os passos tomados para a implementação da solução heurística como um todo. Chamaremos, daqui em diante, a heurística do VMP Probabilístico por VMPP.

Ademais, vale ressaltar que o início da rota não se dá em um vértice aleatório, pois

conforme definido acima, a rota inicia sempre no depósito, ou seja, no vértice de índice 0. Dessa forma, o pseudo-código da variação do VMP usada neste projeto torna-se:

Algorithm 2 Vizinho Mais Próximo Probabilístico

```
1: function VMPP(V, c)
2:     i ← 0
3:     V ← {1, 2, ..., n}
4:     while V ≠ ∅ do
5:         Ordene V de acordo com o peso das arestas (i, k), ∀k ∈ V
6:         j ← weighted_random(V, c)
7:         Insira a aresta (i, j) na solução
8:         V ← V \ {j}
9:         i ← j
10:    end while
11:    Insira a aresta (j, 0) na solução
12: end function
```

3.2 Busca Local 2-Opt

O 2-Opt é um algoritmo de busca local simples desenvolvido para resolver o TSP. [5] Algoritmos de busca local têm a característica de começar a partir de uma solução inicial e então usar uma estrutura de repetição para tentar encontrar melhorias nas vizinhanças dessa solução. Essa estrutura de repetição pode ser executada por uma quantidade pré-determinada de iterações ou até que nenhum resultado melhor que o atual seja obtido. Este algoritmo tem como entrada uma solução inicial viável qualquer, sendo que no nosso caso, será utilizada a solução gerada pelo VMP Probabilístico, um algoritmo construtivo.

Para dar uma ideia do algoritmo, considere uma entrada dada por um grafo onde os vértices são pontos no plano euclidiano e os custos são dados pela distância euclidiana. O algoritmo funciona por reordenar trechos de uma rota onde arestas se sobrepõe uma a outra de maneira que tal sobreposição seja eliminada. Repare que no cenário do PCVME, encontrar uma solução melhor pode inclusive gerar novas sobreposições a serem processadas pelo 2-Opt.

De forma resumida, os passos tomados são: selecionar dois arcos da rota, reconectar tais arcos de uma forma diferente e calcular se houve diminuição no custo total da rota. Em caso afirmativo, uma função *swap2opt* se encarrega de reorganizar a rota e devolver a nova rota mais barata, que por sua vez torna-se a rota atual.

Antes de apresentar o algoritmo, apresentaremos as notações usadas para sua definição. Uma rota r é dada pela sequência $r = (r_0, r_1, ..., r_n, r_{n+1})$, sendo $r_0 = r_{n+1} = 0$, o ponto da rota correspondente ao depósito. Denotaremos por $w_0 = P$, o peso do veículo ao sair do depósito, por $r[i, ..., j]$ a sequência $(r_i, r_{i+1}, ..., rj)$ e por $\overline{r[i, ..., j]}$ a inversão da sequência $(r_i, r_{i+1}, ..., rj)$, ou seja, $(r_j, r_{j-1}, ..., r_i)$. Dadas duas sequências r' e r'', denotaremos sua concatenação por $r' \| r''$. Enfim, dada a matriz de custos c e a rota r, denotaremos por $c(r)$ o valor $\sum_{i=0}^{n} c[r_i, r_{i+1}]$.

Os pseudo-códigos de *swap2opt* e 2-Opt seguem respectivamente abaixo:

Algorithm 3 swap2opt

1: **function** SWAP2OPT(r, i, j)
2: $r' \leftarrow r[0, ..., i-1] \,\|\, \overline{r[i, ..., j]} \,\|\, r[j+1, ..., n]$
3: *return* r'
4: **end function**

Algorithm 4 2-Opt Local Search

1: Seja n a quantidade de vértices a visitar
2: Seja r uma solução inicial
3: Seja c a matriz de custos do grafo
4: **function** 2OPT($r,\ n,\ c$)
5: **repeat**
6: $melhoria \leftarrow 0$
7: **for each** i, j tal que $1 \leq i < j \leq n$ **do**
8: $r' \leftarrow swap2opt(r, i, j)$
9: **if** $c(r') < c(r)$ **then**
10: $r' \leftarrow r$
11: $melhoria \leftarrow 1$
12: **break** loop da linha 7
13: **end if**
14: **end for**
15: **until** $melhoria = 0$
16: **end function**

Como no caso do PCVME queremos minimizar uma função de energia ao invés da distância percorrida, utilizaremos uma versão alternativa do algoritmo Busca Local 2-Opt, a qual chamaremos de *Energy Minimization 2-Opt* ou *EM 2-Opt*. Para tanto, denotaremos por $w(r, i)$ o peso total do caminhão mais as cargas transportadas do ponto 0 ao ponto i na rota r, isto é, $\sum_{i=0}^{i-1} w_i$, e por $\varepsilon(r)$ a energia gasta para transportar as cargas pela rota r, isto é, $\varepsilon(r) = \sum_{i=0}^{n} c[r_i, r_{i+1}] \cdot w(r, i+1)$.

Feitas tais explicitações, segue o algoritmo em pseudo-código do *EM 2-Opt*:

Algorithm 5 Energy Minimization 2-Opt

```
1: Seja n a quantidade de vértices a visitar
2: Seja r uma solução inicial
3: Seja ε a função de custo energético ao longo de uma rota
4: function EM-2-OPT(r, n, ε)
5:     repeat
6:         melhoria ← 0
7:         for each i, j tal que 1 ≤ i < j ≤ n do
8:             r' ← swap2opt(r, i, j)
9:             if ε(r') < ε(r) then
10:                r ← r'
11:                melhoria ← 1
12:                break loop da linha 7
13:            end if
14:        end for
15:    until melhoria = 0
16: end function
```

3.3 Multi-Start VMP Probabilístico e Energy Minimization 2-Opt

A abordagem heurística que de fato será usada neste projeto é uma combinação dos algoritmos descritos nas duas subseções anteriores: O VMPP fornece uma solução de entrada para o *Energy Minimization 2-Opt* e este processo se repete por uma quantidade determinada por meio de testes computacionais. Dessa forma, a heurística usada neste projeto caracteriza-se como Multi-Start. Multi-Start é uma metaheurística que busca gerar e aprimorar soluções usando alguma heurística de refinamento. A solução inicial é gerada de forma aleatória, sendo continuamente suplantada cada vez que uma melhor é encontrada. Assim, cada iteração global produz uma solução (normalmente um ótimo local) e a melhor solução ao final de todas elas é a saída do algoritmo.[6]

4 Experimentos Computacionais

O código para os experimentos desse projeto foi escrito na linguagem Python em um ambiente Jupyter Notebook. Nas subseções seguintes, vamos descrever como foram as três fases de experimentos para 6 instâncias fixadas. Primeiro buscamos a solução ótima por meio do Solver Gurobi, em seguida aplicamos a abordagem heurística e finalmente, rodamos novamente o solver Gurobi, porém agora informando um valor de Cutoff dado pela melhor solução obtida pela abordagem heurística.

4.1 Instâncias

As instâncias são parametrizadas pelo peso do veículo e o número de pontos n a serem visitados. A partir deste valor, são gerados de forma aleatória e simulando um plano bidimensional, n pares de coordenadas x e y cujos valores variam de -100 a 100 no conjunto dos

números racionais. Em seguida, é calculada a matriz de custos a partir dos n pares de pontos. Isso é feito por criar uma matriz de adjacências '*costs*' em que cada elemento $costs[i][j]$ representa a distância euclidiana ($\sqrt{(i_x - j_x)^2 + (i_y - j_y)^2}$) da aresta (i, j) multiplicada por um fator aleatório entre 0.1 e 1. Além disso, são gerados também de maneira aleatória os n pesos das cargas a serem coletadas em cada um dos n pontos, sendo estes valores inteiros entre 50 e 100 kilogramas. Em todos os experimentos realizados neste projeto, mantivemos o peso do veículo fixo em 1000 kilogramas.

Finalmente, é importante ressaltar que os experimentos não foram feitos com instâncias maiores que 30 vértices em razão da licença para estudantes do otimizador Gurobi não permitir a execução de modelos com instâncias maiores que tal valor. Assim, fixaremos instâncias com 5, 10, 15, 20, 25 e 30 pontos.

4.2 Solver Gurobi

Nessa seção vamos expor os resultados obtidos por um algoritmo exato otimizado pelo Solver Gurobi, mais especificamente sua biblioteca para Python: Gurobipy. Para todas as instâncias o parâmetro *TimeLimit*, que corresponde ao tempo máximo para a otimização do modelo, foi setado para 3600 segundos. Em cada linha da tabela abaixo temos a instância por número de pontos, o custo da melhor solução, a solução gerada pelo solver e o tempo de execução.

Instância	Custo da Melhor Solução	Rota Melhor Solução	Tempo de execução
5 pontos	180750	[0, 1, 4, 3, 2, 0]	0.15 segundos
10 pontos	263510	[0, 5, 7, 4, 9, 2, 3, 8, 1, 6, 0]	13.54 segundos
15 pontos	361966	[0, 7, 5, 14, 3, 4, 6, 13, 9, 2, 1, 8, 10, 12, 11, 0]	3600 segundos
20 pontos	417299	[0, 9, 14, 18, 3, 4, 8, 7, 5, 10, 15, 2, 11, 6, 17, 1, 12, 16, 19, 13, 0]	3600 segundos
25 pontos	407613	[0, 13, 20, 5, 21, 18, 24, 3 16, 2, 23, 6, 19, 8, 11, 4, 17 1, 14, 10, 22, 15, 9, 7, 12, 0]	3600 segundos
30 pontos	564997	[0, 9, 19, 24, 6, 29, 28, 15, 7, 21 20, 22, 18, 10, 17, 12, 5, 14, 13, 26 23, 2, 1, 11, 3, 8, 27, 4, 25, 16, 0]	3600 segundos

4.3 A Solução Heurística

Inicialmente nesta seção vamos esclarecer como foi implementada a escolha aleatória baseada em pesos da função que representa o VMPP. Em cada iteração, inicializamos um

dicionário Python '*cost_to_vertex_sorted*' com os custos de trânsito por uma aresta que parte do vértice atual e chega a um vértice ainda não visitado, sendo cada chave dada pelo vértice destino da aresta e seu valor dado pelo custo da aresta. Em seguida, cada par chave-valor deste dicionário é ordenado de acordo com a ordem crescente de seus valores. O algoritmo então usa um parâmetro $k \in \{1, ..., n\}$ de maneira que, em cada iteração, os até k primeiros vértices da ordenação são candidatos a serem o próximo vértice a visitar. Dada uma ordenação $(v_1, ..., v_t)$ com t candidatos, a probabilidade de se escolher o vértice i é dada por:

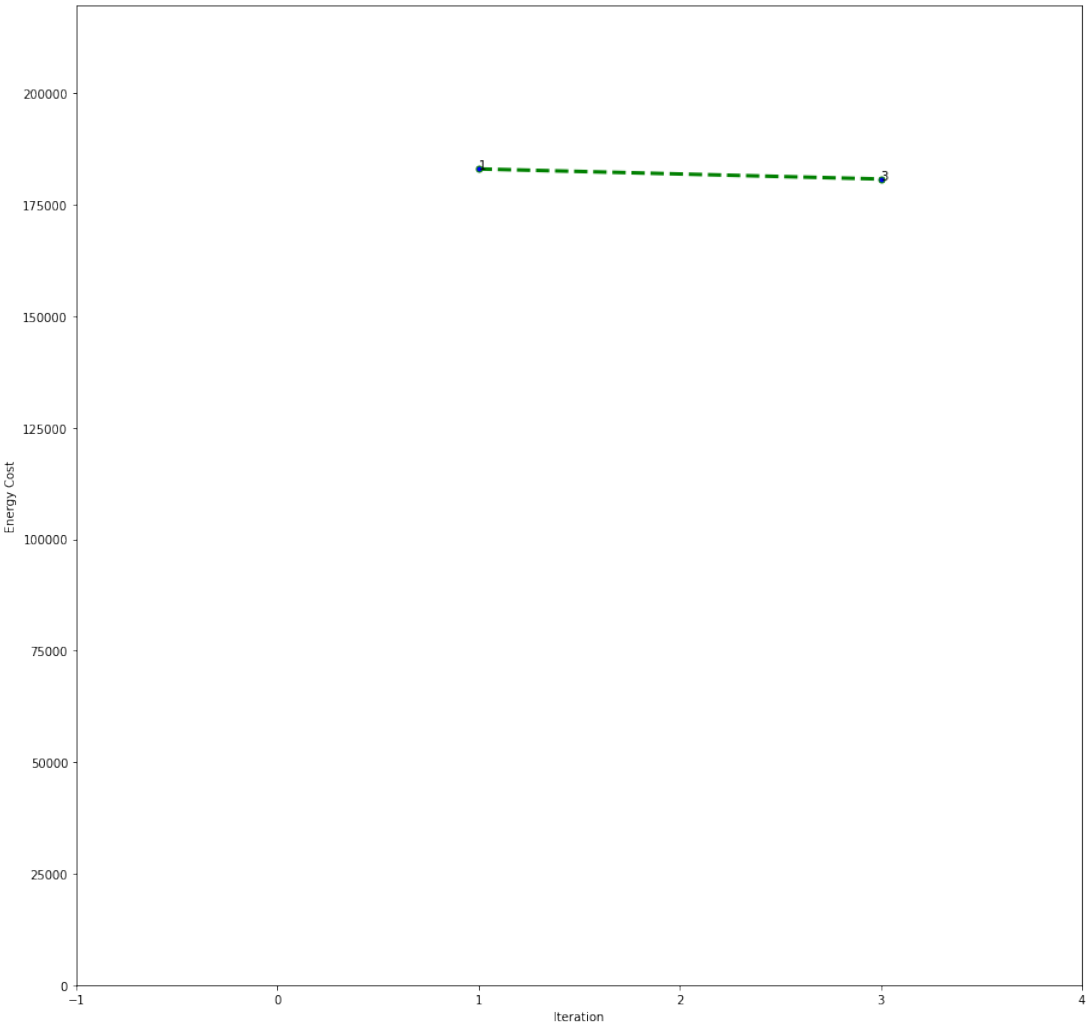
$$\frac{\frac{1}{i}}{\sum_{j=1}^{t} \frac{1}{j}}.$$

Isto posto, é criado um vetor de pesos p tal que $p = [\frac{1}{1}, \frac{1}{2}, ..., \frac{1}{k}]$, sendo $k \in \{1, ..., n\}$ o número de vértices não visitados. Finalmente, a função '*choices*' da biblioteca *built-in* '*random*' se encarrega de atribuir os pesos em p de maneira respectiva a cada elemento de *cost_to_vertex_sorted* e realiza a escolha probabilística. Tal função, no que lhe diz respeito, realiza o trabalho da função *weighted_random*, mencionada na seção 3.1.

Para a escolha da quantidade de repetições com o propósito de obter uma solução heurística satisfatória, fazemos uma análise dos gráficos das seis subseções abaixo. É possível notar que a maior parte das descobertas de melhores soluções ocorre no quarto inferior das iterações da abordagem heurística, havendo portanto uma similaridade nos perfis das curvas dos gráficos a partir de 10 pontos. Note também que as últimas iterações que entregam melhores soluções comumente apresentam cada vez menos diferença em termos de custo energético em relação a melhor solução anterior. Dessa forma, definir o número de iterações totais do algoritmo como uma quantidade levemente superior ao número da iteração em que o gráfico de custo em energia por iterações demonstra sinais de estabilização foi a estratégia adotada neste projeto. Tal limitante foi escolhido conforme o comportamento das instâncias com o maior número de pontos (30, neste caso), que por sua vez apresentam maiores custos a minimizar e consequentemente precisam de mais iterações. Após diversas execuções da instância de 30 pontos, identificamos que os custos provenientes da descoberta de uma nova melhor solução sempre alcançava um patamar constante antes das 100 mil iterações, logo, este foi o parâmetro de repetição utilizado em todos os seis tipos de instâncias deste projeto.

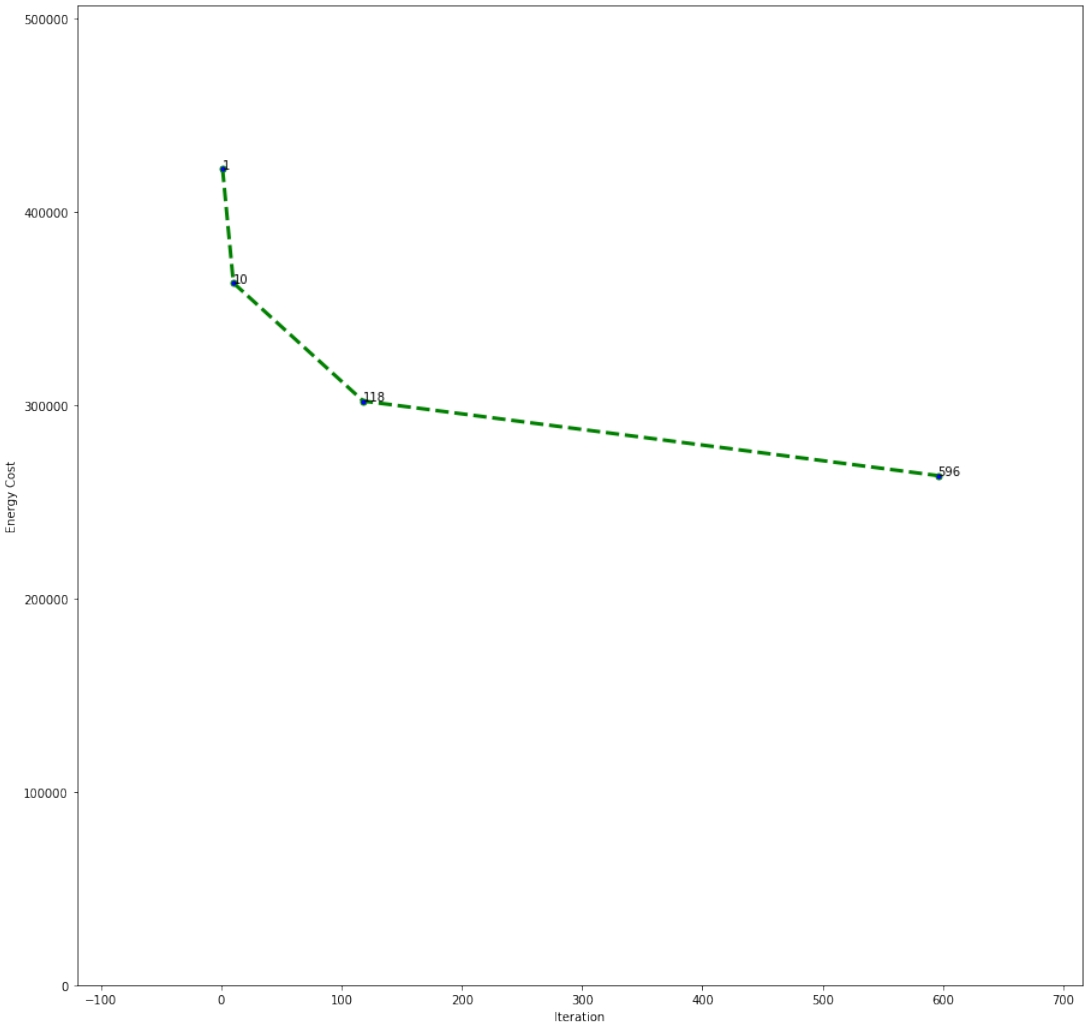
Seguem nas subseções a seguir os gráficos mencionados acima, bem como o número de substituições do algoritmo por uma melhor solução, o tempo de execução, a rota encontrada como solução e seu custo energético.

4.3.1 5 Pontos



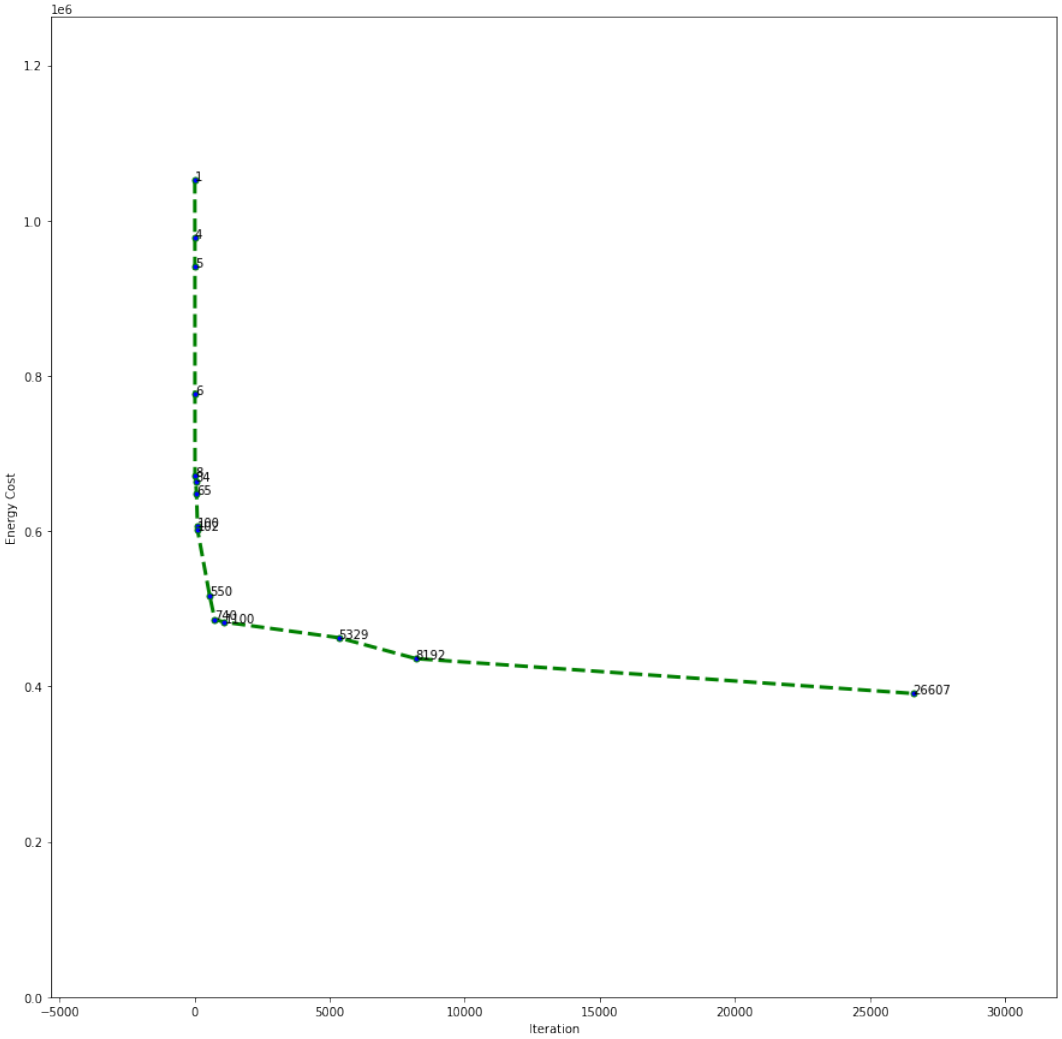

```
There were 2 iterations where a best cost was found.
The heuristic approach took 8 seconds to complete.
Best path found by Heuristics: [0, 1, 4, 3, 2, 0]
Best cost found by Heuristics: 180750
```

4.3.2 10 Pontos



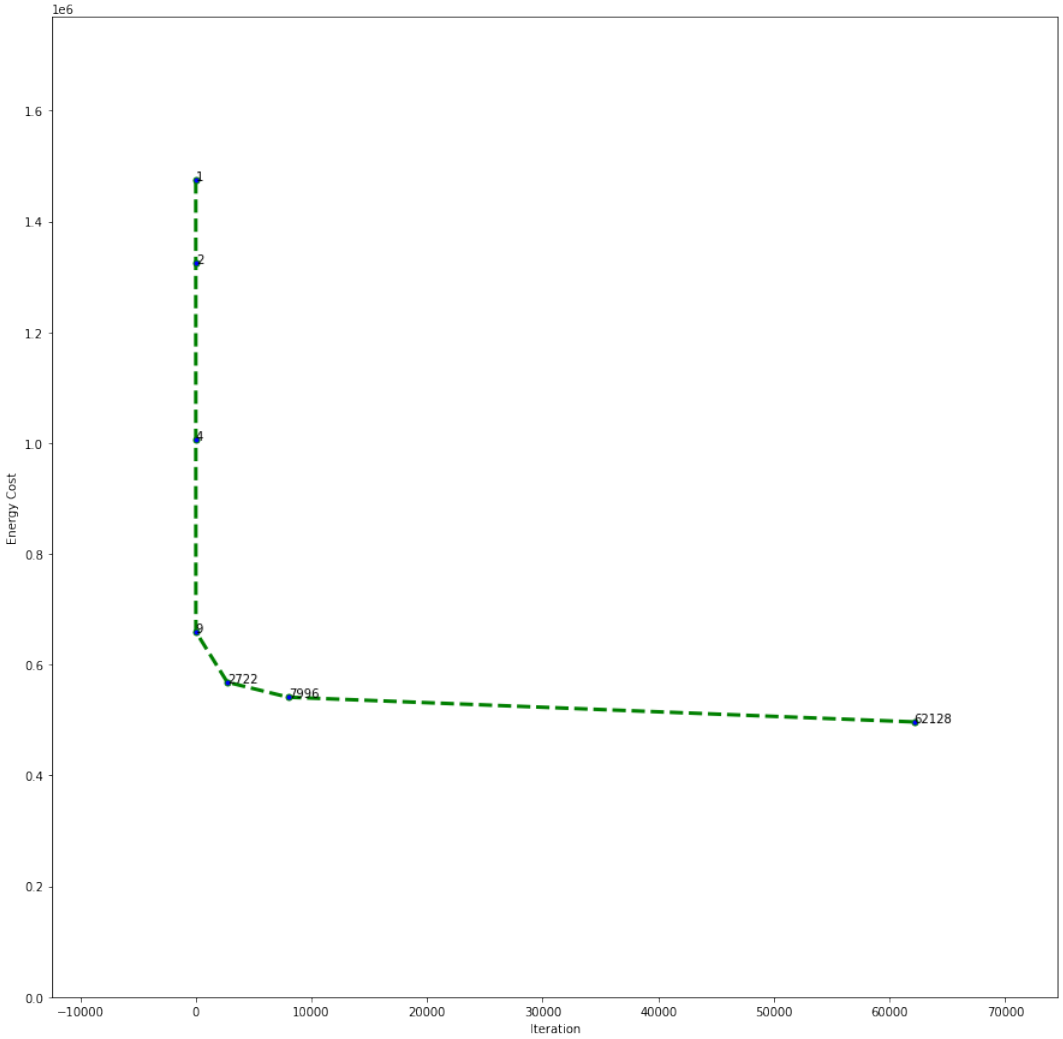

```
There were 4 iterations where a best cost was found.
The heuristic approach took 24 seconds to complete.
Best path found by Heuristics: [0, 5, 7, 4, 9, 2, 3, 8, 1, 6, 0]
Best cost found by Heuristics: 263510
```

4.3.3 15 Pontos



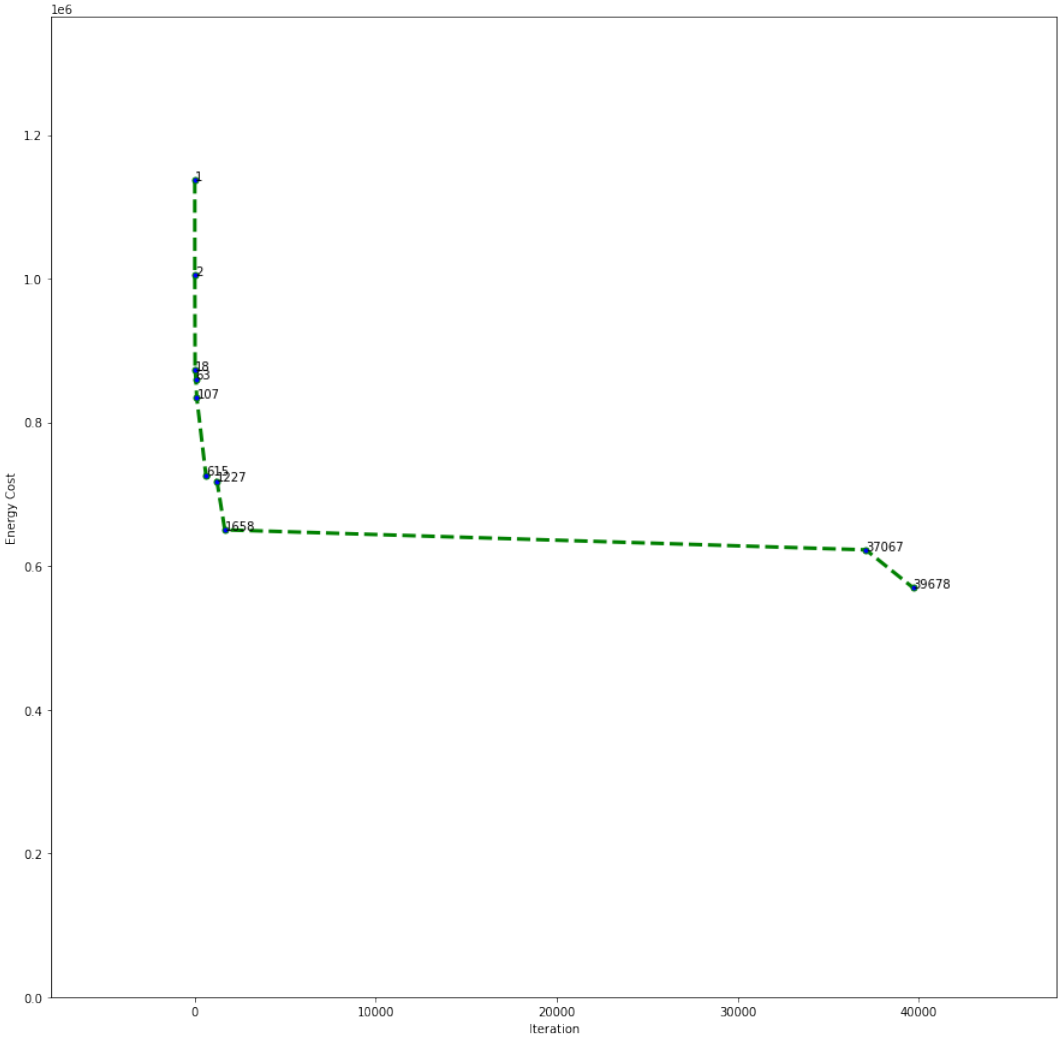

```
There were 15 iterations where a best cost was found.
The heuristic approach took 51 seconds to complete.
Best path found by Heuristics: [0, 11, 7, 5, 14, 3, 4, 6, 13, 9, 2, 1, 8, 10, 12, 0]
Best cost found by Heuristics: 391020
```

4.3.4 20 Pontos



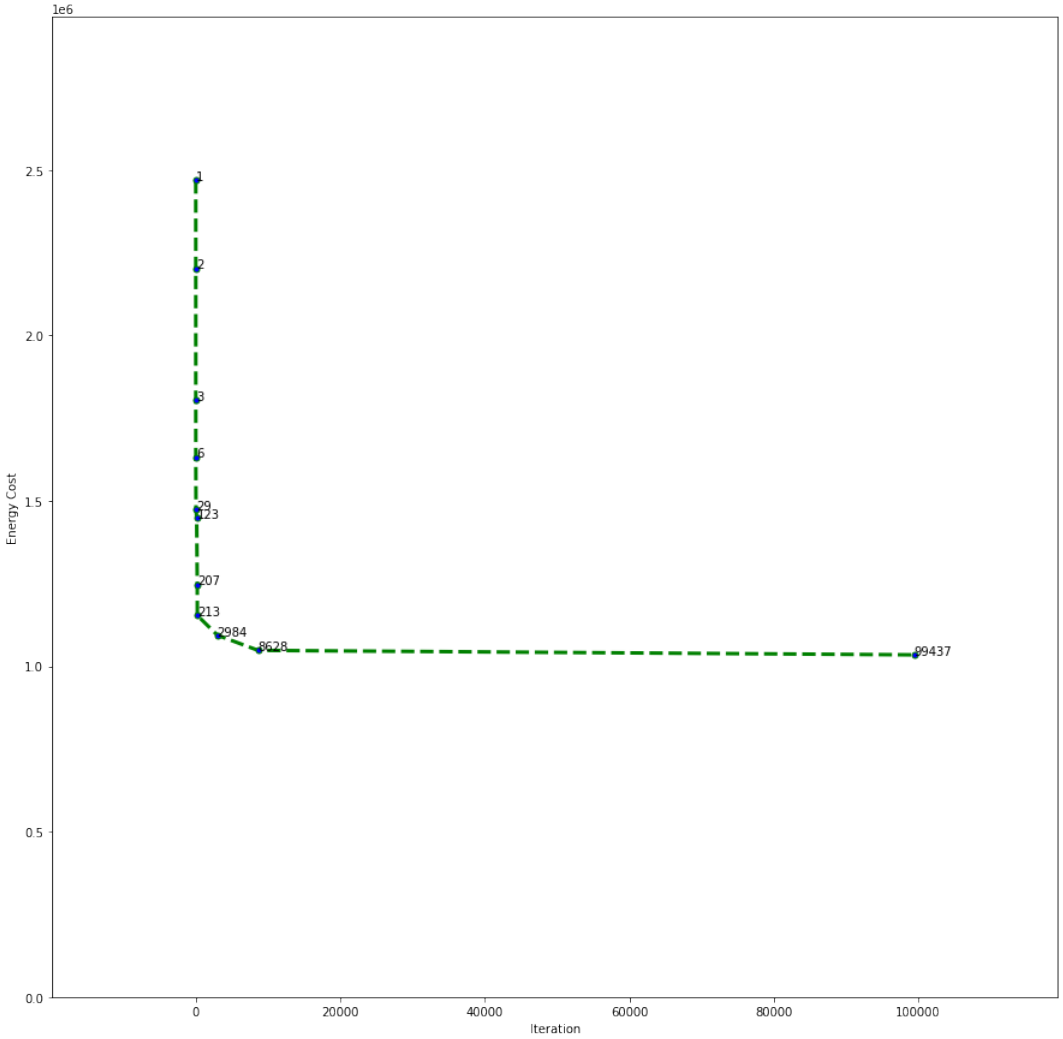

```
There were 7 iterations where a best cost was found.
The heuristic approach took 87 seconds to complete.
Best path found by Heuristics: [0, 10, 5, 17, 1, 9, 16, 11, 12, 6, 14, 18, 3, 4, 15, 19, 13, 2, 8, 7, 0]
Best cost found by Heuristics: 496665
```

4.3.5 25 Pontos




```
There were 10 iterations where a best cost was found.
The heuristic approach took 133 seconds to complete.
Best path found by Heuristics: [0, 14, 22, 15, 17, 2, 6, 19, 8, 4, 11, 16, 23, 18, 1, 24, 3, 10, 13, 20, 5, 7, 9, 12, 21, 0]
Best cost found by Heuristics: 569652
```

4.3.6 30 Pontos




```
There were 11 iterations where a best cost was found.
The heuristic approach took 194 seconds to complete.
Best path found by Heuristics: [0, 9, 19, 24, 13, 22, 18, 11, 28, 15, 7, 21, 20, 2, 1, 26, 8, 27, 23, 17, 12, 5, 14, 29, 25
16, 3, 10, 4, 6, 0]
Best cost found by Heuristics: 1034633
```

4.4 Solver Gurobi com Valor Cutoff

Outro importante parâmetro que pode ser configurado no otimizador Gurobi é o valor de *Cutoff* do modelo. Por meio desse valor, é possível fazer com que o solver só busque soluções abaixo dele. A razão para executar novamente o otimizador com um valor de *cutoff* é o pressuposto de que fornecendo um valor de custo energético abaixo do qual o modelo deve procurar soluções, este possa ser executado mais rapidamente e encontrar soluções menos

custosas. Assim sendo, determinamos que o valor de *cutoff* passado ao modelo Gurobi é aquele dado pelas soluções heurísticas, expostos nas subseções da seção 4.3. O parâmetro *TimeLimit* foi mantido em 3600 segundos para as segundas execuções dos modelos Gurobi. Segue abaixo a tabela com os resultados obtidos após a reexecução do modelo para cada uma das instâncias:

Instância	Custo da Melhor Solução	Rota Melhor Solução	Tempo de execução
5 pontos	180750	[0, 1, 4, 3, 2, 0]	0.15 segundos
10 pontos	263510	[0, 5, 7, 4, 9, 2, 3, 8, 1, 6, 0]	4.07 segundos
15 pontos	Não encontrada	Não encontrada	3600 segundos
20 pontos	Não encontrada	Não encontrada	3600 segundos
25 pontos	405136	[0, 21, 18, 24, 3, 16, 2, 23 6, 19, 8, 11, 4, 17, 1, 14, 10 22, 15, 13, 20, 5, 9, 7, 12, 0]	3600 segundos
30 pontos	560536	[0, 9, 1, 2, 27, 4, 10, 17, 12, 3 8, 5, 14, 11, 28, 15, 7, 21, 20, 22 18, 29, 6, 13, 26, 23, 24, 19, 25, 16, 0]	3600 segundos

5 Análise dos Resultados

Avaliando os resultados obtidos na seção 4, pudemos constatar alguns padrões interessantes. A solução heurística consegue se equiparar a instâncias consideradas pequenas em nosso cenário (até 10 pontos), se aproximar das instâncias médias (15 a 20 pontos) e a partir de instâncias com 20 pontos é possível notar uma maior divergência entre as abordagens heurística e exata. O grande ganho do procedimento heurístico se dá no tempo gasto para sua execução, uma vez que produz resultados quase instantâneos e muito precisos para instâncias pequenas e, embora divirja bastante em termos do custo energético encontrado para instâncias grandes (25 ou mais pontos), leva aproximadamente 19 vezes menos tempo para emitir um resultado nesses casos. Outra particularidade encontrada está relacionada à parametrização do solver Gurobi com um valor *cutoff*. Em contrapartida ao que se suspeitava, informar um valor de *cutoff* não torna a execução das otimizações mais rápidas, com exceção de instâncias com 10 pontos, em que foi obtida uma notável melhoria no runtime. Por outro lado, confirmou-se a estimativa de que um valor *cutoff* possibilitaria que soluções melhores fossem obtidas em relação a primeira execução do solver Gurobi. Como se pode verificar, a segunda execução (com valor *cutoff*) costuma sempre retornar um valor de custo energético idêntico ou melhor (em instâncias maiores) quando comparado à primeira execução.

6 Conclusão

Encerramos este projeto tendo analisado tanto teórica, quanto experimentalmente, uma variação do TSP bastante condizente com os problemas de roteamento de veículos encontrados no mundo real. Além disso, foi possível estudar duas heurísticas amplamente conhecidas e empregadas na resolução do TSP, modificá-las e combiná-las para testar seu desempenho em uma versão do TSP que embora simples, é também menos recorrente em literaturas sobre o assunto.

Por restrições de tempo, este projeto se restringiu a poucas heurísticas e algoritmos. Para futuros estudos, é desejável explorar implementações de outras heurísticas e algoritmos, bem como testá-las em instâncias mais próximas das aplicações reais.

É importante ressaltar que não somente pelo fato da licença Gurobi para estudantes limitar o tamanho das instâncias investigadas, o fato da formulação usada ser deveras trivial também não torna viável que o problema seja executado para instâncias maiores (superior a 50 pontos). A implementação de uma formulação diferente e mais robusta não pôde ser realizada em razão da curta janela de tempo para a entrega do projeto, não obstante permitiria que instâncias mais próximas de problemas do cotidiano fossem utilizadas, como por exemplo as instâncias das mais famosas bibliotecas do TSP. A escolha de outra linguagem como C++ em detrimento de Python também se provaria proveitosa para este projeto, visto que Python é amplamente conhecida como uma linguagem lenta pelo fato do código ser interpretado em tempo de execução, diferente de linguagens menos *high-level*, como a já mencionada C++. Contudo, por motivos de maior familiaridade com a linguagem, Python foi escolhida para a escrita do código. Ademais, existem vários estudos mais aprofundados a respeito de variações semelhantes a este problema e que produzem bons resultados, como por exemplo o artigo escrito por Shijin Wang, Ming Liu e Feng Chu, em que é usado um algoritmo branch-and-bound integrado a um 1-tree based lower bound.[7]

Referências

[1] B. Gärtner and J. Matoušek. Understanding and using linear programming. *Journal of Cleaner Production (Springer, Berlin)*, 2006.

[2] Hadley and George. Linear programming. *Eighth Printing, Addison-Wesley*, 1974.

[3] C.E. Miller, A.W. Tucker, and R.A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960. doi: 10.1145/321043.321046.

[4] Masehian and Ellips. New heuristic algorithms for solving single-vehicle and multi-vehicle generalized traveling salesman problems (gtsp). *Journal of Optimization in Industrial Engineering*, 2, 2010.

[5] G. A. Croes. A method for solving traveling salesman problems. *Exploration and Production Research Division, Shell Development Company, Houston, Texas*, 1958.

[6] R. Martí. Multi-start methods, handbook of metaheuristics. *Kluwer Academic Publishers, Boston*, page 355–367, 2003.

[7] S. Wang, M. Liu, and F. Chu. Approximate and exact algorithms for an energy minimization traveling salesman problem. 2020.