UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

WESLEY MAMORU SONOMURA

OTIMIZAÇÃO DE ROTAS PARA REALIZAÇÃO DE PESQUISAS DE PREÇOS EM SUPERMERCADOS

LONDRINA

2022

WESLEY MAMORU SONOMURA

OTIMIZAÇÃO DE ROTAS PARA REALIZAÇÃO DE PESQUISAS DE PREÇOS EM SUPERMERCADOS

Optimization Of Routes For Price Collecting In Supermarkets

Trabalho de conclusão de curso de Graduação apresentada como requisito para obtenção do título de Bacharel em Engenharia de Produção da Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador(a): Prof. Dr. Rafael Henrique Palma Lima

LONDRINA

2022

WESLEY MAMORU SONOMURA

OTIMIZAÇÃO DE ROTAS PARA REALIZAÇÃO DE PESQUISAS DE PREÇOS EM SUPERMERCADOS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Produção da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 06/junho/2022

_____

Rafael Henrique Palma Lima
Doutor
Universidade Tecnológica Federal do Paraná

_____

Bruno Samways dos Santos
Doutor
Universidade Tecnológica Federal do Paraná

_____

Pedro Rochavetz de Lara Andrade
Doutor
Universidade Tecnológica Federal do Paraná

LONDRINA

2022

RESUMO

Na cidade de Londrina/PR, a inflação da cesta básica é pesquisada e divulgada mensalmente pelo Núcleo de Pesquisas Econômicas Aplicadas (NUPEA) da Universidade Tecnológica Federal do Paraná (UTFPR). Nessa pesquisa, são levantados preços em doze supermercados espalhados pela cidade, sempre no último dia útil do mês vigente. A divisão das rotas entres os pesquisadores do NUPEA deve considerar uma distribuição equilibrada do tempo total que será gasto por cada pesquisador. Diante deste cenário, é proposto um algoritmo baseado no *hill-climbing* que faça a busca por uma solução deste problema que foi caracterizado como um *Traveling Salesman Problem* assimétrico com múltiplos viajantes. A solução deste problema aborda a divisão das rotas em k pesquisadores e a minimização do tempo máximo entre as k rotas. O primeiro passo do algoritmo é gerar uma solução inicial pelo VMP estocástico que depois é dividida em sub-rotas de custo menor que 180 minutos. O algoritmo tem três tipos de estrutura de vizinhanças diferentes, baseados em inserção aleatória de pontos em rotas, trocas de pontos aleatórias entre rotas ou sorteio entre um dos dois métodos. A quantidade de iterações realizadas é um parâmetro do problema e após a realização de todas as iterações, o algoritmo retorna a melhor solução global encontrada. Além da instância real, o desempenho do algoritmo foi avaliado em 19 instâncias da biblioteca tsplib95, após o qual foi concluído que o algoritmo performa melhor em instâncias pequenas e que seu custo computacional cresce exponencialmente em função da quantidade de pontos sem melhora significativa da solução global. Quanto aplicado à instância da pesquisa de inflação, o algoritmo encontrou uma solução utilizando 3 pesquisadores, com custo de tempo total de 368 minutos onde a maior rota que compõe a solução tem custo de 125 minutos.

Palavras-chave: *hill-climbing*; problema do caixeiro viajante*;* análise combinatória; algoritmo de otimização.

ABSTRACT

In the city of Londrina/PR, the basket of goods price inflation is monthly researched and published by the Núcleo de Pesquisas Econômicas Aplicadas (NUPEA) from Universidade Tecnológica Federal do Paraná (UTFPR). In this research prices are collected on the last workday of the current month in 12 different supermarkets scattered throughout the city. The distribution of routes between travelers must be evenly balanced considering the total time cost assigned to each researcher. In order to solve this problem, a hill-climbing based algorithm to search for a solution of a Multiple Asymmetric Traveling Salesman Problem is proposed. The solution to this particular problem must consider the division of the routes between k number of researchers and the minimization of total time of the k routes. The first step to this algorithm is to generate an initial solution with another algorithm called Stochastic Nearest Neighbor and slice it in sub-routes in a way that each one of them has a time cost lower than 180 minutes. The algorithm has three different neighborhood structures based in random insertion of points, random points swap and a random choice between one of the two other methods. The number of iterations is an input parameter of the problem and after going through all iterations the algorithm returns the best global solution it could find. Besides the real-world instance, the algorithm's performance was evaluated in 19 instances from tsplib95 instance library. After which, it was possible to see that the algorithm performs better in small instances and its computational cost grows exponentially with the number of points with no significant improvement of the global solution. Applied to the basket of good price inflation research, the algorithm found a solution with 3 travelers with total time cost of 368 minutes in which the biggest route takes 125 to complete.

Keywords: hill-climbing; traveling salesman problem; combinatorial analysis; optimization problem.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A inflação dos preços e ajuste do salário mínimo são informações importantes que geram grande expectativa na população. Autores como Agarwal, Chua e Song (2020), assim como Rambalducci e Feltrin (2018) afirmam que a inflação nos produtos alimentícios força as famílias a rearranjar os orçamentos domésticos, tendo impactos negativos em sua alimentação, tanto em qualidade como em quantidade.

Sobretudo, a inflação é sentida com maior peso pela população de baixa renda. Um estudo do Banco Central mostra que em 2020 a inflação acumulada para gastos de alimentação para famílias com renda de 1 até 3 salários mínimos foi de 10,27%, enquanto famílias com renda de 10 a 40 salários mínimos tiveram uma inflação acumulada de 8,16% para a mesma categoria (BANCO CENTRAL DO BRASIL, 2021).

Na cidade de Londrina, a inflação da cesta básica é pesquisada e divulgada mensalmente pelo Núcleo de Pesquisas Econômicas Aplicadas (NUPEA) da Universidade Tecnológica Federal do Paraná (UTFPR) em seu projeto de extensão Pesquisa da Variação do Valor da Cesta Básica em Londrina. Esse projeto compara a inflação da cesta básica com o salário-mínimo nacional e paranaense a fim de estabelecer séries históricas e analisar a oscilação temporal do poder de compra do cidadão londrinense.

A pesquisa é realizada pelo levantamento e análise dos preços em doze supermercados da cidade, cobrindo os quatro pontos cardeais e o centro da cidade. Os itens analisados são aqueles que compõem a cesta básica definidos pelo decreto 399 de 1938 e são coletados com base no menor valor encontrado em cada supermercado, desconsiderando marcas.

O levantamento dos preços é geralmente feito no último dia do mês pelos alunos e professores responsáveis pelo projeto. Os pesquisadores visitam os supermercados, fazem a coleta dos preços, tabulam e analisam os dados e confeccionam um relatório que será divulgado à imprensa local.

A maior parte do tempo gasto para a realização desta pesquisa corresponde ao deslocamento entre os pontos de coleta. Visto que estão espalhados pela cidade, torna-se um problema definir a melhor sequência de visitas, pois o sentido das vias e

localização geográfica das regiões da cidade fazem com que as distâncias e os tempos de deslocamento entre os supermercados se tornem fatores essenciais na realização desta pesquisa.

Este problema pode ser visto com um caso especial do Problema do Caixeiro Viajante (*Traveling Salesman Problem)* em que os pesquisadores são múltiplos caixeiros e os supermercados, os pontos a serem visitadas. O TSP é um dos problemas clássicos de otimização combinatória e consiste em um visitante que precisa passar por diversos pontos exatamente uma vez e retornar ao local de início (DANTZIG; FULKERSON; JOHNSON, 1954).

Por se tratar de um problema amplamente estudado, existem diversas heurísticas publicadas para a sua resolução, incluindo a de suas variantes. Mais especificamente, este trabalho considera o problema da roteirização dos pesquisadores como um TSP assimétrico com múltiplos viajantes, ou seja, um m-ATSP. Dessa maneira, o objetivo deste trabalho é propor um algoritmo de otimização baseado no *hill-climbing* que ofereça uma solução dividida em sub-rotas, que obedeçam a um limite de tempo, e que minimize o tempo máximo de deslocamento das sub-rotas que a compõem.

Este trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta uma revisão bibliográfica do TSP, apresentando trabalhos e heurísticas publicadas do problema clássico assim como de suas variantes. O Capítulo 3 faz uma descrição mais detalhada do problema, apresentando suas restrições, parâmetros, formato dos dados e o método de resolução proposto. O Capítulo 4 mostra os resultados e discussões da implementação do algoritmo. O Capítulo 5 traz as conclusões e considerações finais. Os apêndices trazem o algoritmo programado na linguagem *Python*.

2 REFERENCIAL TEÓRICO

A seguinte seção apresenta a fundamentação teórica na qual este trabalho se baseia. A seção 2.1 aborda o TSP, a seção 2.2 traz algumas das principais heurísticas para sua resolução, as seções 2.3 e 2.4 apresentam, respectivamente, o MTSP e as heurísticas para a resolução deste caso específico.

2.1 *Traveling salesman problem* (TSP)

O problema do caixeiro viajante, ou *traveling salesman problem* (TSP), é um dos problemas de otimização mais famosos e estudados na literatura especializada. Historicamente, não tem suas origens muito bem definidas, porém, Merril Flood da Universidade de Columbia pode ser creditado por estimular o crescimento de interesse em volta do TSP por meio de seu trabalho em 1937, no qual ele tentou obter soluções quase ótimas para o problema de roteirização de ônibus escolares (DANTZIG; FULKERSON; JOHNSON, 1954).

Este problema pode ser resumido a partir da seguinte pergunta: dado um grafo com n pontos, qual a rota que passa exatamente uma vez por todos os pontos, formando um circuito fechado que se inicia e termina no mesmo ponto, de modo a minimizar o custo/distância total de viagem?

Matematicamente, o problema pode ser definido por um grafo $G = (V, E)$, direcionado ou não direcionado, e por F, sendo a família de todos os ciclos hamiltonianos em G, em que $V = \{1,2, \dots, n\}$ é o conjunto de pontos a serem visitados e E é o conjunto de arestas. Para cada aresta $e \in E$, um custo (distância) c_e é estabelecido. O problema do caixeiro viajante é encontrar uma rota no grafo G tal que a soma dos custos das arestas presentes na rota seja a menor possível (PUNEN, 2002).

As classificações do TSP são extremamente variadas, porém, em geral, tem-se o TSP simétrico, onde a distância $d_{ij} = d_{ji}$. Há o TSP assimétrico, para o qual esta condição não é verdadeira. E também existe o TSP múltiplo (mTSP), no qual se apresentam múltiplos vendedores para percorrer o grafo.

Para o TSP simétrico, Dantzig, Fulkerson e Johnson (1954) caracterizaram a formulação matemática mais citada para este problema, representada abaixo:

Minimizar: $$D(x) = \sum_{i>j} d_{ij} x_{ij} \quad (1)$$

Sujeito a: $$\sum_{j=1}^{n} x_{ij} = 2 \quad (x_{ij} \geq 0; i = 1, 2, \dots, n; i \neq j;\ x_{ij} = x_{ji}) \quad (2)$$

$$\sum_{s\bar{s}} x_{ij} \geq 2 \quad (3)$$

$$x_{ij} \in \{0, 1\} \quad (4)$$

Onde d_{ij}é a distância ou custo de visitar o ponto j a partir de i. A variável x_{ij} se refere à decisão de realizar esta rota, sendo 0 caso negativo e 1 se a rota entrar na solução do problema – condição definida pela restrição (4). As restrições (2) e (3) são referentes ao grau de cada ponto$n$ e à eliminação de sub-rotas, respectivamente.

O número total de rotas possíveis factíveis, que contenham todos os pontos a serem visitados, dado uma quantidade n de cidades é calculado pela expressão abaixo (DANTZIG; FULKERSON; JOHNSON,1954):

$$\frac{(n-1)!}{2} \quad (5)$$

Uma aplicação deste problema foi desenvolvida por Ratliff e Rosenthal (1983), que realizaram um trabalho para minimizar a distância percorrida por veículos dentro de um armazém logístico. O problema se inicia quando um pedido é emitido e um veículo é designado para buscar os itens demandados e trazê-los de volta para a área de expedição. Cada item tem uma posição específica dentro do armazém e corresponde a um vértice do grafo em questão. Dessa maneira, este problema de aumentar a velocidade de coleta dos itens pôde ser resolvido como um TSP (RATLIFF; ROSENTHAL, 1983).

Para o caso do TSP assimétrico, temos um grafo G = (V, A), onde $V = \{1, 2, \dots, n\}$ é o conjunto de vértices e $A = \{(i,j): i,j \in V\}$ é o conjunto dos arcos formados por pares de vértices do grafo. Deve-se achar um circuito hamiltoniano passando cada ponto de V apenas uma vez de modo a minimizar o custo/distância total da viagem.

Nesse caso, é apresentado um grafo direcional, portanto temos que os custos $c_{ij} \neq c_{ji}$ (ROBERTI; TOTH, 2012).

Roberti e Toth (2012) também adequaram a formulação linear inteira do TSP proposto por Dantzig, Fulkerson e Johnson (1954) para o TSP assimétrico (ATSP), representada nas equações abaixo:

Minimizar:
$$\sum_{i,j=1}^{n} d_{ij} x_{ij} \quad (6)$$

Sujeito a:
$$\sum_{i=1}^{n} x_{ij} = 1, j = 1, 2, \dots, n \quad (7)$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, 2, \dots, n \quad (8)$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \; S \subset V : S \neq \emptyset \quad (9)$$

$$x_{ij} \in \{0, 1\}, \; i, j = 1, 2, \dots, n \quad (10)$$

Na formulação apresentada, as restrições (7) e (8) se referem ao grau de entradas e saídas de cada vértice e a restrição (9) é para eliminação de sub-rotas.

2.2 Heurísticas para o TSP

Heurísticas são estratégias, critérios e métodos utilizados para decidir uma linha de ação que seja a mais eficaz para atingir um objetivo. Segundo a autora, a resolução de muitos problemas complexos requer um tempo de processamento muito alto, assim, requerendo heurísticas que possam diminuir a quantidade de análises e avaliações e que levem à uma solução em um intervalo de tempo razoável (JUDEA, 1984). A seguir estão listadas algumas heurísticas para a resolução do TSP.

2.2.1 Vizinho mais próximo (VMP)

A heurística mais difundida para resolução do TSP é a chamada “Vizinho mais próximo” (VMP) ou *Nearest Neighbor* (NN). Essa heurística faz a escolha do ponto seguinte com base na menor distância disponível. Ela consiste nos seguintes passos (ROSENKRANTZ; STEARNS; LEWIS, 1977):

a) Iniciar a partir de um vértice aleatório;

b) Identificar o vértice mais próximo e adicionar uma aresta conectando os dois vértices – escolher arbitrariamente em caso de empate;
c) Quando todos os vértices estiverem incluídos no trajeto, ligue o ponto final ao inicial com uma aresta.

Kizilateş e Nuriyeva (2013) conseguiram resultados melhores com uma variação do NN considerando não somente o ponto final de um lado, mas os dois vértices finais de cada lado. Assim, o algoritmo – denominado *Nearest Neighbor Algorithm from both end-points* (NND) - se consiste em:

a) Iniciar a partir de um ponto arbitrário;
b) Visitar o vértice não visitado mais próximo;
c) Visitar o vértice não visitado mais próximo, considerando os dois vértices;
d) Quando todos os vértices forem visitados, ligar os pontos finais.

2.2.2 Inserção mais próxima

A premissa básica dos métodos de *insertion*, é construir uma rota inserindo nela arcos progressivamente maiores. (ROSENKRANTZ; STEARNS; LEWIS, 1977). Desse modo, este algoritmo pode ser resumido nos seguintes passos NILSSON (2003):

a) Selecionar a menor aresta e fazer uma sub-rota com ela;
b) Identificar o ponto ainda não incluso na sub-rota que tenha a menor distância de alguma das cidades na sub-rota
c) Inserir o ponto na sub-rota na posição que minimiza o custo de inserção deste ponto;
d) Repetir passo a) e b) até que não restem mais pontos

O *nearest insertion* se difere do VMP no que tange à escolha dos arcos adicionados na solução. Enquanto o algoritmo VMP trabalha de maneira a encontrar o nó imediatamente mais próximo do ponto atual, o *nearest insertion* procura a menor aresta dentre todas as possíveis e procura encaixá-la na solução de modo a minimizar o custo de sua inserção.

2.2.3 Heurísticas para ATSP

Johnson *et al.* (2002) propuseram quatro grupos de heurísticas para este caso de TSP: heurísticas clássicas de construção de rotas, heurísticas baseadas em soluções para o problema de designação, heurísticas de busca local e heurísticas de busca local com repetição.

Entre as heurísticas clássicas tem-se o VMP, já referenciado neste trabalho, e o *Greedy* (guloso). No algoritmo *Greedy* para o ATSP, a instância é um grafo completamente direcional e os arcos têm comprimentos correspondentes às distâncias entre os pontos. O algoritmo consiste nos seguintes passos (JOHNSON *et al*., 2002):

a) Ordenar crescentemente os arcos por comprimento;
b) Identificar um arco como aceitável, caso possa ser adicionado no *set* atual de arcos sem criar um ciclo não hamiltoniano ou causar um grau de entrada ou saída maior que 1;
c) Escolher aleatoriamente entre os dois menores arcos aceitáveis, repetidamente até que a rota esteja completa.

Os algoritmos baseados em problemas de designação são o *Cycle Patching* – que calcula o *minimum cycle cover*, para a instância inteira e depois emenda os ciclos da seguinte maneira: repetidamente selecionar os dois ciclos que contenham a maior quantidade de cidades e quebrar um arco em cada ciclo para juntar os dois ciclos em um. Outras heurísticas mencionadas são a *Repeated Assignment*, a *Contractor Patch*, e a Heurística de Zhang (JOHNSON *et al*., 2002).

As heurísticas de busca local são métodos que começam com uma solução e buscam soluções melhores por buscas em vizinhanças (AARTS e LENSTRA, 1997). Os métodos de busca local mencionados por Johnson *et al.* (2002) são o 3-opt e a heurística de Kanellakis-Papadimitrou, que foi uma variação do Lin-Kernighan para o caso assimétrico (KANELLAKIS e PAPADIMITROU, 1980).

As heurísticas de busca local com iterações são aplicações repetidas das heurísticas de busca local, buscando resultados melhores a cada iteração realizada.

2.3 *Multiple traveling salesman problem* (MTSP)

O mTSP pode ser definido, com um conjunto de nós, com m caixeiros localizados em um único depósito. Os outros nós são cidades a serem visitadas e denominadas “nós intermediários”. O mTSP consiste em encontrar rotas para todos os m caixeiros, saindo e retornando ao nó depósito, tal que cada nó intermediário e é visitado exatamente uma vez. O custo total da viagem deve ser minimizado (BEKTAS, 2006).

As variações possíveis para esse problema incluem: variações na quantidade de depósitos, quantidade de caixeiros, custos adicionais para cada caixeiro, janelas de tempo (*time windows*) (BEKTAS, 2006). Nota-se também que um caso especial do mTSP é aquele com um único viajante, tornando-se então um problema TSP.

Além disso, BEKTAS (2006) ainda enuncia alguns problemas nos quais o mTSP pode ser aplicado, como a programação de prensas de impressão (GORENSTEIN, 1970) e o roteamento de ônibus escolares (ANGEL, *et al.*, 1972).

O trabalho realizado por Gorenstein (1970) consistia na programação de prensas de impressão de periódicos com variadas edições de modo a diminuir custos. Uma prensa impressora tem 5 pares de cilindros e cada edição é impressa em conjuntos de 4, 6 ou 8 páginas. Em seu trabalho, foi discutido o cenário usando um conjunto de 8 páginas. O problema consiste em decidir quais conjuntos de páginas estarão em cada corrida (*run*) e o comprimento de cada corrida, de modo a minimizar os custos de troca de placas impressoras.

Angel *et al.*, (1972) realizaram uma aplicação do problema de roteamento de ônibus escolares. O objetivo deste trabalho foi obter um padrão de carregamento de ônibus (*bus loading pattern*), tal que minimize o número de rotas e a distância percorrida, respeite a capacidade dos ônibus e não ultrapasse limite de tempo, impostos por políticas, para percorrer qualquer rota.

Segundo Bektas (2006), o mTSP pode ser definido formalmente por um grafo $G = (V, A)$, sendo que V é o conjunto de n vértices (nós), A é o conjunto de arestas (arcos) que ligam os nós e $C = (c_{ij})$ é a matriz de custos (distâncias) associada a A. Caso $c_{ij} = c_{ji}, \forall (i,j) \in A$, considera-se um caso simétrico. Caso contrário, torna-se um problema de caixeiro viajante assimétrico. O problema satisfaz a desigualdade triangular se e somente se $c_{ij} + c_{jk} \geq c_{ik}, \forall i, j, k \in V, C$.

Bektas (2006), ainda apresenta uma formulação para o mTSP para programação linear baseada em designações. Primeiramente, definindo uma variável binária x_{ij}, a qual toma valor 1, caso o arco (i,j) seja usado na rota e 0 caso não seja usado.

Minimizar: $$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \qquad (11)$$

Sujeito a:

$$\sum_{j=2}^{n} x_{1j} = m\,, \tag{12}$$

$$\sum_{i=2}^{n} x_{i1} = m\,, \tag{13}$$

$$\sum_{i=1}^{n} x_{ij} = 1, j = 2, \dots, n \tag{14}$$

$$\sum_{j=1}^{n} x_{ij} = 1, i = 2, \dots, n \tag{15}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \tag{16}$$

$$x_{ij} \in \{0, 1\} \tag{17}$$

Onde, as restrições (12) e (13) fazem com que exatamente m vendedores saiam e retornem do ponto de partida. As restrições (14) e (15) eliminam a ocorrências de bifurcações na rota. A restrição (16) é a eliminação de sub-rotas, proposta por Dantzig *et al.* (1954), que também é válida para o mTSP (BEKTAS, 2006).

2.4 Heurísticas para o mTSP

Algumas abordagens com algoritmos heurísticos para o mTSP foram propostas por Russel (1977) e por Potvin *et al.* (1989). Russel (1976) propôs um algoritmo para o caso simétrico do mTSP, com algumas condições de capacidade, distância e tempo e restrições de sequência. Seu algoritmo foi uma extensão da heurística do TSP com único caixeiro, de Lin e Kernighan. Potvin *et al.* (1989) propuseram uma generalização dos métodos *k-opt* para o mTSP simétrico, considerando mudanças numa rota que levem à partição entre sub rotas, que são combinadas de volta em uma rota novamente.

Bellmore e Hong (1974) mostraram que é possível converter o mTSP pela resolução do TSP em um grafo expandido que contém $(m-1)$ nós a mais que o grafo original. Pelo método proposto por Bellmore e Hong (1974), é realizado a expansão do grafo $G(N, A)$ para o $G(N', A')$. O conjunto de nós N' é obtido pela adição de $(m-1)$ nós ao conjunto N, que são contados por números negativos $-1, -2, \dots, -(m-1)$. A partir do qual é gerado o novo conjunto de arcos A', que contém:

(i) Todo arco presente em A;
(ii) Os arcos $(-i, j)$ para todo novo nó $(-i)$, se A contém o arco $(0, j)$, para $j = 1,2, \dots, n$;
(iii) Os arcos $(j, -i)$ para todo novo nó $(-i)$, se A contém o arco $(j, 0)$, para $j = 1,2, \dots, n$;
(iv) Os arcos $(-i, -(i-1))$ para $i = 1, 2, \dots, (m-1)$.

O cálculo dos custos destes novos arcos, é feito em função dos custos fixos para a designação de cada vendedor e dos custos já conhecidos dos arcos pertencentes em A. Seja C_i o custo associado ao i-ésimo vendedor, os custos dos arcos são calculados da seguinte maneira (BELLMORE e HONG, 1974):

(i) $d'(i, j) = d(i, j)$ para $i = 1,2, \dots, n$ e $j = 1,2, \dots, n$;
(ii) $d'(-i, j) = d(0, j) + \frac{1}{2} c_i$ para $i = 1,2, \dots, (m-1)$ e $j = 1,2, \dots, n$;
(iii) $d'(j, -i) = d(j, 0) + \frac{1}{2} c_i$para $i = 1,2, \dots, (m-1)$ e $j = 1,2, \dots, n$;
(iv) $d'(-i, -(i-1)) = \frac{1}{2} c_{i-1} - \frac{1}{2} c_i$ para $i = 1,2, \dots, (m-1)$.

Tendo construído o grafo expandido, Bellmore e Hong (1974) afirmam que o mTSP pode ser resolvido usando métodos de resolução do TSP padrão em para $G(N', A')$. E também, caso cada vendedor precise visitar uma cidade, pode ser resolvido a partir da remoção dos arcos $(-i, -(i-1))$ para $i = 1,2, \dots, (m-1)$ do grafo expandido $G(N', A')$. Dessa maneira, cada nó adicional é forçado a ser seguido por um nó cliente em uma rota pertencente a $G(N', A')$.

3 MÉTODO PROPOSTO

Esta seção traz, em ordem, uma descrição mais profunda sobre o problema real tratado neste trabalho, os parâmetros escolhidos para a solução das várias instâncias escolhidas e a explicação do algoritmo proposto.

3.1 Descrição do problema

A pesquisa da inflação da cesta básica, realizada pelo Núcleo de Pesquisas Econômicas Aplicadas da UTFPR é sempre realizada em 12 supermercados de Londrina, cobrindo o centro e os quatro pontos cardeais da cidade. Esses pontos de coleta são fixos, sem mudança de um mês para outro. Para sua escolha foram levantadas todas as redes de supermercado que atuam na cidade e listadas todas as unidades de cada rede, segregando-as por região. E então, foi realizada a escolha dos pontos de coleta de modo a levar em conta a distribuição da população londrinense, a não ter dois pontos de coleta da mesma rede e a abranger todas as redes de supermercados listadas inicialmente.

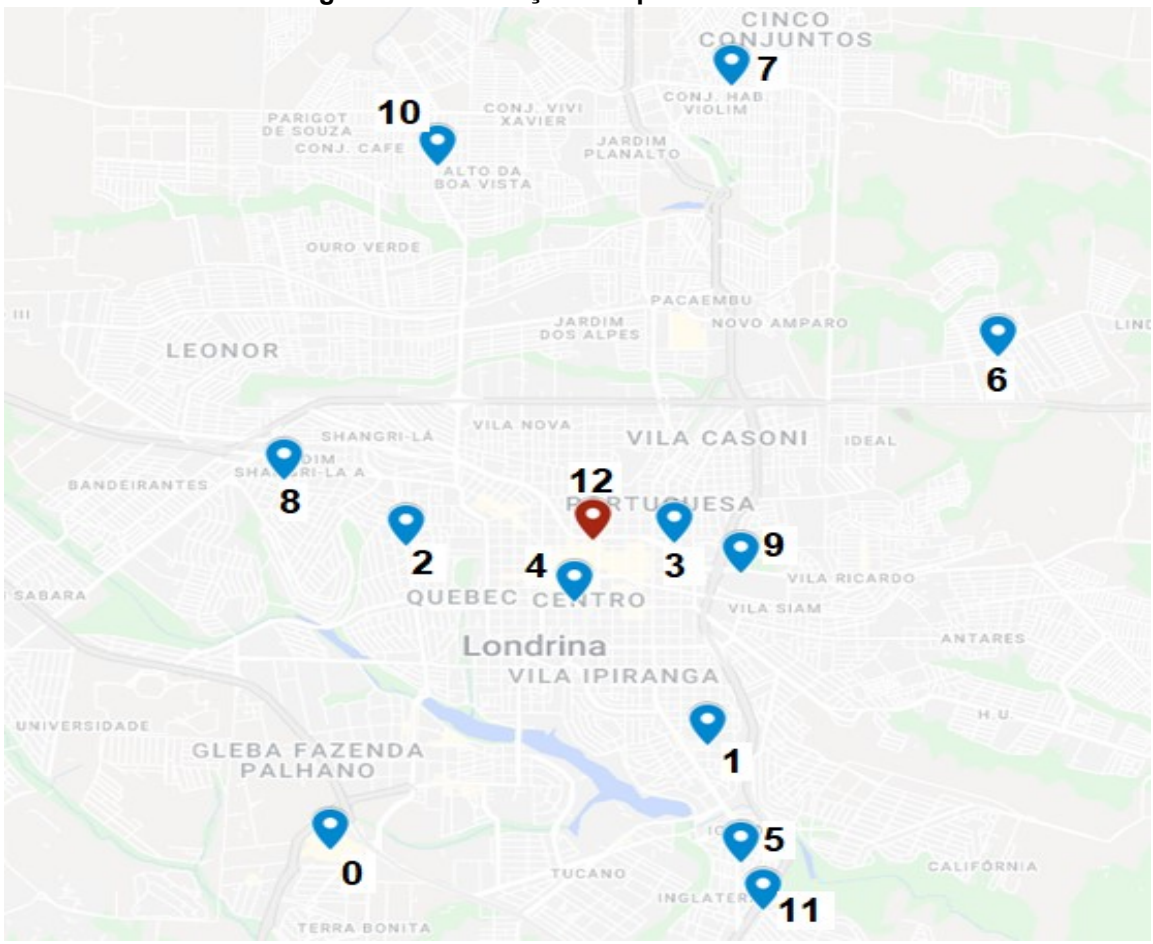
Essa pesquisa é realizada pela coleta e tabulação dos preços dos 13 itens da cesta básica nacional, definidos pelo Decreto-Lei nº399, de 30 de abril de 1938. Sua frequência é mensal, e é feita no último dia útil do mês vigente. É importante que a coleta em todos supermercados seja realizada no mesmo dia, para evitar possíveis flutuações de preço.

Após a coleta, é feito o tratamento e análise dos dados coletados e é formulado um relatório que é enviado à imprensa local para divulgação dos dados à população. Por conta disso, é fundamental que a pesquisa seja realizada com certa agilidade, principalmente durante o deslocamento entre os pontos de coleta. Os diferentes trajetos entre os supermercados devem ocorrer da maneira mais eficiente possível para que os pesquisadores tenham mais tempo para a realização da análise de dados e possam coletar e tabular os preços com calma, sem comprometer a integridade de dados.

Os supermercados visitados são sempre os mesmos e são localizados por toda a cidade (Figura 1). A localização de cada ponto de coleta está disponível no site do NUPEA (NUPEA, 2021). Por conta da disposição das principais vias que conectam as regiões, entre um ponto e outro frequentemente é necessário escolher entre tomar

uma rota entre bairros residenciais – com velocidade média menor, porém distância mais curta – ou utilizando vias maiores – com velocidade média maior, porém possivelmente maior distância a ser percorrida.

Figura 1 - Localização dos pontos de coleta



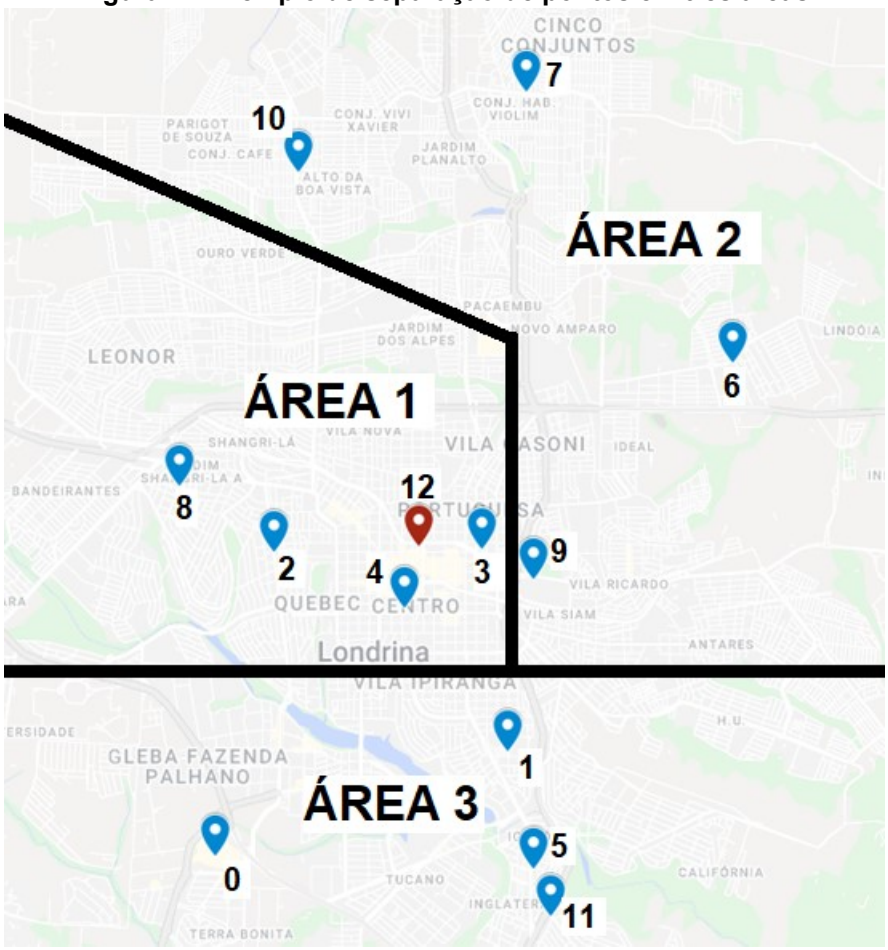
Fonte: o autor.

A localização de certos supermercados, como os pontos 10, 7, 6 e 0, caracteriza outro ponto de tomada de decisão referente a ordem de visita. Por se localizarem em uma região mais afastada das outras, acabam sendo um dos pontos críticos para a solução deste problema, pois formam arcos longos com outros pontos.

Para realizar a coleta de dados, os pesquisadores saem todos ao mesmo horário e devem retornar dentro de um limite de tempo para compilar os dados e elaborar o relatório da inflação mensal. Por esse motivo, deve-se atribuir uma carga balanceada de ponto de coleta para cada pesquisador, evitando atrasos e tornando-se necessário minimizar o tempo máximo de campo de um colaborador.

A fim de ilustrar exemplo de designação de pontos para três pesquisadores, poderia-se dividir o mapa em três áreas. Neste caso, a área 1 conteria os pontos 2, 3, 4, 8 e 12. A área 2, os pontos 6, 7, 9 e 10. E a área 3, os pontos 0, 1, 5 e 11. Ilustrado abaixo na Figura 2.

Figura 2 - Exemplo de separação de pontos em três áreas



Fonte: o autor.

Esta configuração permite separar os mercados em quantidade igual para cada pesquisador. Porém a qualidade desta divisão pode ser questionada, já que na área 1, os pontos estão muito mais próximos entre si que na área 2, assim, fazendo que o tempo de pesquisa seja desbalanceado entre cada área.

Levanta-se a possibilidade, portanto, de realizar a designação de pontos baseada nos custos dos arcos adjacentes a cada ponto e na região na qual está localizado.

3.2 Parâmetros do problema

Os primeiros parâmetros levantados foram as coordenadas de cada supermercado e as distâncias reais entre cada um deles, caracterizados na Tabela 1.

Tabela 1 - Coordenadas decimais dos pontos de coleta

	Coordenadas	
Ponto	Latitude	Longitude
0	-23,3408819634613	-51,1860528934160
1	-23,3305009529451	-51,1502330204008
2	-23,3086327013108	-51,1779041492371
3	-23,3082902409724	-51,1528850897139
4	-23,3148939298122	-51,1623887106110
5	-23,3434299708686	-51,1464573725184
6	-23,2877722516596	-51,1228816492376
7	-23,2579872455883	-51,1474154878638
8	-23,3013937860301	-51,1884615786069
9	-23,3117534721844	-51,1466534897140
10	-23,2669072898947	-51,1747651204019
11	-23,3486516803458	-51,1445947129954
12	-23,3085245374148	-51,1607279627628

Fonte: Elaborado pelo autor com os dados disponíveis em <https://www.nupea.org >. Acesso em: 07 de maio de 2021

As distâncias reais estão representadas nas Tabelas 2 e 3, que mostram dados coletados de distância, em quilômetros, e tempo de deslocamento, em minutos, da melhor rota disponível no momento da coleta. Para isso, foi utilizado o aplicativo de mapas *Google Maps* e os parâmetros que serão utilizados são aqueles referentes ao transporte por carro. O carro foi escolhido por ser atualmente o principal meio de transporte utilizado pelos pesquisadores e para evitar inúmeros retornos ao Terminal Central antes de acabar uma rota, como aconteceria com o ônibus. Nelas é possível observar a falta de simetria em referência à diagonal principal, característica do problema de ATSP. Tal assimetria se dá por conta dos diferentes sentidos de cada rua, presença de semáforos, vias preferenciais e outros aspectos do transporte urbano.

Tabela 2 - Tempos de deslocamento entre pontos de coleta em minutos

De/para	0	1	2	3	4	5	6	7	8	9	10	11	12
0	999	12	10	16	13	10	23	25	12	15	19	10	18
1	12	999	11	7	9	6	14	16	14	6	18	7	15
2	9	11	999	9	7	13	15	18	5	9	12	13	10
3	15	7	7	999	5	9	8	12	8	4	11	10	4
4	11	7	8	4	999	8	13	15	9	5	12	10	6
5	10	4	14	8	10	999	14	16	15	6	18	3	14
6	22	13	15	12	13	16	999	10	14	10	13	17	18
7	22	13	17	12	16	16	12	999	16	11	7	17	19
8	12	13	4	9	9	16	13	16	999	10	10	15	11
9	14	6	9	4	8	8	9	12	11	999	14	9	9
10	19	15	11	11	12	17	13	8	10	12	999	19	14
11	10	5	13	7	10	3	13	15	15	6	17	999	13
12	22	9	11	7	6	13	16	25	13	5	19	14	999

Fonte: O autor.

Tabela 3 - Distância em quilômetros entre pontos de coleta

De/para	0	1	2	3	4	5	6	7	8	9	10	11	12
0	999	6,3	4,8	6,7	5,4	5,3	12,4	16,1	7,6	9,2	9,8	6,4	7,3
1	6,3	999	5,6	2,7	2,9	2,3	7,9	10,1	6,9	3,1	8,8	3	3,4
2	4,4	5,9	999	4	2,7	8,1	9,7	11,1	2,3	4,4	7,1	10,8	2,8
3	6,8	4	2,7	999	1,8	5,4	4,9	7,5	4,5	1,4	6,3	6,2	0,8
4	5	3,7	2,8	1,7	999	4,3	7	9,1	4,5	2,2	6,3	5,5	1
5	5,8	1,9	7,6	5	4,3	999	9,2	11,4	12	4,5	12,7	1,5	6
6	15,1	9,2	9,6	6,9	7,3	10,6	999	6,2	9,6	6,7	8,7	11,5	6,4
7	15,5	9,6	9,1	7,3	8,5	11	7,1	999	10,8	7,1	3,9	11,9	8
8	6,1	8	1,7	4,7	4,5	9,8	8,7	10,1	999	5,4	6,2	11,8	3,5
9	9,7	3,9	4,2	1,6	3,3	5,3	4,9	7,8	5,8	999	9,1	6,2	2,8
10	10,1	11,3	5,7	7	6,5	12,7	8,7	3,9	6,2	8,7	999	13,5	6
11	7	2,9	7,7	5,1	5,5	1,1	9,3	11,5	9,4	4,6	12,8	999	6,3
12	7,5	4,1	2,3	1,3	1	5,5	5,7	8,5	4	1,5	5,9	6,4	999

Fonte: O autor.

Outro parâmetro a ser analisado foi a escolha do ponto base, de onde as rotas teriam seu início e fim. Para a instância focada neste trabalho foi escolhido o Terminal Central de Londrina por conta de sua localização e por ser um ponto de comum conhecimento entre os estudantes. Nas tabelas de custos acima, o Terminal Central é representado na última linha e na última coluna, sendo ele o ponto de número 12.

Por conta do horário de abertura dos supermercados e da necessidade de se realizar a pesquisa em um único dia, há uma restrição de tempo máximo que deve ser respeitado para a coleta de dados. Caso se torne muito extenso pode atrasar o relatório final. A abertura dos supermercados em questão é por volta das 8:00 da manhã e é preferível que o relatório seja enviado até meio-dia. A tabulação, análise

dos dados e confecção do relatório costuma demandar 1 hora, quando feito de maneira sem pressa. Por isso, é necessário que os trajetos tenham um custo de tempo máximo de 3 horas.

Além do custo de deslocamento, existe o tempo de visita em cada ponto de coleta. Estes costumam ser constantes entre si, com baixa variação entre cada supermercado. Para cada visita é demandado aproximadamente 20 minutos e, como a variação entre cada local é baixa, será considerado uma constante.

A quantidade de pesquisadores também é limitada, visto que o grupo de pesquisas econômicas divide várias demandas de pesquisas entre si, e este número costuma variar de dois a quatro. Para este trabalho, será feita uma análise do menor tempo máximo e qual a quantidade mínima de pesquisadores que satisfaz esta condição.

3.2.1 Instâncias do tsplib95

Com o intuito de analisar o comportamento do algoritmo proposto em instâncias com maiores quantidades de pontos, também foram usadas outras 19 instâncias. Essas instâncias foram coletadas a partir da biblioteca conhecida como *tsplib95*, que reúne diversas instâncias para problemas de TSP e seus derivados. A Tabela 4 descreve a nomenclatura de cada instância assim como a respectiva quantidade de pontos disponíveis. Para essas instâncias, o ponto base escolhido foi sempre o último ponto da matriz de distâncias.

Tabela 4 - Instâncias do tsplib95

Nome da instância	Quantidade de pontos
br17	17
ftv33	34
ftv35	36
ftv38	39
p43	43
ftv44	45
ftv47	48
ry48p	48
ft53	53
ftv55	56
ftv64	65
ft70	70
ftv70	71
ftv90	91
kro124	100
ftv170	171
rbg323	323
rbg358	358
rbg403	403
rbg443	443

Fonte: Elaborado pelo autor a partir das instâncias disponíveis em < http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>. Acesso em: 20 de maio de 2022.

Os parâmetros para aplicação do algoritmo proposto a essas instâncias foram feitas com base no tamanho de cada instância. Primeiramente as instâncias foram divididas em tercis, categorizando os pontos entre instâncias pequenas “P”, médias “M” ou grandes “G”. E a partir disso, os valores de quantidade de iteração e de quantidade de vizinhos gerados por iteração dados. A quantidade de iteração foi atribuída como 300 para instâncias pequenas, 200 para médias e 20 para grandes. Já a quantidade de vizinhos foi de 50 para instâncias pequenas, 100 para médias e 50 para grandes. Esses parâmetros foram determinados para que todas as instâncias fossem solucionadas em no máximo 24 horas.

3.2.2 Conversão das instâncias tsplib95

Antes de gerar uma solução inicial, foi realizado um método de padronização das matrizes de custos a serem utilizadas pelo algoritmo. Para transformar as instâncias do TSPLIB em *arrays* comuns, além da biblioteca do *TSPLIB,* o *tsplib95*, foram

utilizadas as bibliotecas *NetworkX* e *SciPy*. O tsplib95 foi utilizado para realizar a leitura dos arquivos de instâncias. Isso resultava num grafo direcionado, que o NetworkX transformava numa matriz esparsa e que era convertido para uma matriz convencional utilizando os métodos de *arrays sparse* do *SciPy*.

Após isso, insere-se M grandes na diagonal principal da matriz. E também a linha e coluna do ponto base eram retirados e atribuídos a uma variável global para serem utilizados como consulta para a função de cálculo de custos, assim como para evitar que esse ponto fosse alocado dentro de alguma rota, que não nas suas extremidades.

3.3 Proposta dos algoritmos

Esta seção apresenta o algoritmo proposto, estruturado no *hill-climbing* e seu objetivo busca minimizar o custo máximo da solução. Cada subseção seguinte contém uma parte do algoritmo e seu respectivo pseudocódigo.

3.3.1 Geração da solução inicial

A geração da solução inicial se dá por um algoritmo baseado no VMP (descrito na seção 2.2.1) com o acréscimo da aleatoriedade no que tange à escolha dos vizinhos mais próximos. Tal algoritmo será chamado de “VMP estocástico”, no qual o primeiro ponto é escolhido aleatoriamente e a escolha do ponto seguinte é feita com um sorteio.

Este sorteio é realizado com base em um vetor de probabilidades acumuladas, que se baseia no custo dos arcos que o ponto analisado pode formar (sendo ele a origem).

Para a formação do vetor de probabilidade acumulada, primeiro é necessário fazer o cálculo da qualidade de cada ponto candidato a ser o próximo destino. Essa qualidade q_{ij} é dada pelo quadrado da diferença entre a maior distância dentro da lista de destinos, $c_{ij\,max}$, e distância de cada destino, c_{ij}. Ou seja:

$$q_{ij} = (c_{ij\,max} - c_{ij})^2 \tag{18}$$

A probabilidade p_{ij}, portanto, de um destino j ser sorteado é calculada da seguinte maneira:

$$p_{ij} = \frac{q_{ij}}{\sum_{j=1}^{n} q_{ij}} \quad (18)$$

Então, gera-se um vetor de probabilidades acumuladas. Dado por:

$$p\_acum_{ij} = \sum_{j=1}^{j} p_{ij} \quad (18)$$

Após isso, a escolha do próximo ponto é feito pelo sorteio de um número aleatório entre 0 e 1. E o próximo ponto a ser visitado será dado pela posição do vetor a qual está atribuído o valor a do intervalo de pontos adjacentes (a, b) que contém, o número aleatório sorteado.

A Figura 3, 4 e 5 mostram os pseudocódigos dos passos que compõem o VMP estocástico.

Figura 3 - Pseudocódigo do vetor de probabilidades acumuladas

```
1.  def vetor_probabilidades_acumuladas(vetor_inicial):
2.              # Fazendo o vetor qualidade
3.              Para cada ponto em vetor_inicial:
4.                        Vetor_qualidade[i] = max(vetor_inicial) ^ 2
5.                        i += 1
6.              # Fazendo o vetor probabilidade
7.              Para cada custo em vetor_qualidade:
8.                        Vetor_probabilidade[i] = custo /
    soma_total(vetor_qualidade)
9.                        i += 1
10.             # Fazendo o vetor de probabilidades acumuladas
11.             Acumulado = 0
12.             Para probabilidade em vetor_probabilidade:
13.                       Acumulado = acumulado + probabilidade
14.                       Vetor_probabilidade_acumulada[i] = acumulado
15.                       i += 1
16.             Retorna vetor_probabilidade_acumulada
```

Fonte: O autor.

Figura 4 - Pseudocódigo do próximo ponto do VMP estocástico

```
1.  def próximo_ponto(vetor_probabilidade_acumulada):
2.    x = número aleatório entre 0 e 1
3.    i = 0
4.    Enquanto vetor_probabilidade_acumulada[i] < x:
5.              i += 1
6.
7.    Retorna i
```

Fonte: O autor.

Figura 5 - Pseudocódigo do VMP estocástico

```
1.  def vmp_estocastico(matriz_custos, ponto_inicial):
2.              solução = []
3.              solução[0] = ponto_inicial
4.              pontos_faltantes = vetor de todos os pontos da instância menos o
    ponto                                   inicial
5.              i = 0
6.              Enquanto tamanho da solução < tamanho da matriz_custos:
7.                        ponto_atual = solução[i]
8.                        destinos = destinos possíveis do ponto atual
9.                        vetor_prob = vetor_probabilidade_acumulada(destinos)
10.                       próximo_ponto = próximo_ponto(vetor_prob)
11.                       coloca m grandes na matriz de custos
12.                       atualiza lista pontos_faltantes
13.                       se lista_pontos tiver somente 1 ponto,
              adiciona-o na solução sem repetir o algoritmo
      todo.
14.             Retorna solução
```

Fonte: O autor.

Após a geração de uma solução pelo VMP estocástico, é realizada uma agregação de pontos adjacentes em *chunks*, onde cada um representa uma rota atribuída a um viajante e que não ultrapassam o limite de custo de factibilidade. Cada ponto é inserido em uma rota até que se alcance o limite máximo de tempo para cada viajante. Quando um *chunk* chega no custo limite ou o ultrapassa, esta rota é fechada

e é criada uma nova. Esse processo se repete até que todos os pontos do vetor inicial forem alocados em um *chunk*. A Figura 6 traz o pseudocódigo para esta etapa da geração da solução inicial.

Figura 6 - Pseudocódigo da geração de solução inicial

```
1.  def gera_solucao_inicial(instancia, rota_inicial):
2.    rota_hipotetica = []
3.    solucao = []
4.    Para cada ponto em rota_inicial:
5.              Insere ponto em rota_hipotetica
6.              Se custo não excede limite:
7.                        Continua
8.              Se excede limite:
9.                        Insere rota_hipotetica sem ponto atual na solucao
10.                       Limpa rota_hipotetica
11.                       Acrescenta ponto atual
12.                       Continua
13.   Se restam pontos na rota_hipotetica
14.             Insere na solucao
15.
16.   Retorna solucao
```

Fonte: O autor.

A avaliação do custo total de uma rota k é feita a partir da inclusão dos arcos referentes a saída e ao retorno da rota ao ponto base e também da soma de n_k tempos de coleta para cada ponto dentro da rota. O pseudocódigo desta avaliação está representado na Figura 7.

Figura 7 – Pseudocódigo do cálculo de custo de rotas

```
1.  def calcula_custo_rota(instancia, rota):
2.    r = tamanho da rota
3.    custo_total = 0
4.    Para i = 0 até r - 1:
5.              ponto_atual = rota[i]
6.              proximo_ponto = rota[i + 1]
7.              custo_total = custo_total +
    instancia[ponto_atual][próximo_ponto]
8.
9.    Somar custos de ida e volta ao ponto base ao custo_total
10.   Retorna custo_total
```

Fonte: O autor.

3.3.2 Estrutura de vizinhança - inserção

A primeira estrutura de vizinhança apresentada é a baseada em algoritmos de inserção. A inserção é feita a partir da escolha de um ponto aleatório de uma rota aleatória. Este ponto, então, é retirado de sua rota de origem e reinserido na solução numa posição a ser escolhida pela avaliação do melhor ponto de inserção dentro desta

rota. Este método de escolha da posição de inserção é denominado “melhor inserção” e seu pseudocódigo está representado na Figura 8.

Figura 8 - Pseudocódigo da melhor inserção

```
1.  def best_insertion(instancia, rota1 = [], rota2 = [], ponto1 = int())
2.    Remove ponto1 da rota1
3.    Para cada posição em rota 2:
4.              Insere ponto1 na posição
5.              Calcula custo da rota
6.    Escolher a posição de inserção do ponto1 na rota2 que minimiza o custo de inserção
7.    Retorna rota2_com_inserção
```

Fonte: O autor.

A melhor inserção, portanto, funciona fazendo o cálculo do custo da solução em todas as posições possíveis de se inserir o ponto e escolhendo aquela posição na qual se obtém o melhor resultado.

Para essa estrutura de geração de vizinhos, foi realizada uma adaptação do VMP estocástico, que será chamada de “remoção com peso”. Onde, assim como o VMP estocástico, é feito um vetor de probabilidades acumuladas, mas a qualidade de remoção, neste caso, é determinada pelo custo da solução. Quanto maior for, maior será a chance de ter um ponto retirado. Além disso, foi atribuída uma regra para que um vetor com somente um ponto tenha a mesma chance de ser sorteada que o vetor mais caro do conjunto de rotas. Esta regra foi inserida para que haja um incentivo ao algoritmo para a eliminação de rotas de um ponto, prezando pela eficiência da quantidade de viajantes necessários. O pseudocódigo deste algoritmo está representado na Figura 9.

Figura 9 – Pseudocódigo da remoção com peso

```
1.  def remocao_com_peso(solucao, instancia):
2.    Atribui pesos para cada rota dentro da solucao
3.    peso = custo_da_rota ^ 2
4.    Armazena os pesos em um vetor
5.    Caso uma rota tenha somente um ponto, atribuir a ela o maior peso
      calculado
6.    Criar um vetor de probabilidade acumulada para o vetor de pesos
7.    Sorteia uma rota para ter um ponto removido
8.    Retorna rota sorteada
```

Fonte: O autor.

Dessa maneira, a inserção, em sua totalidade se apresenta na Figura 10:

Figura 10 – Pseudocódigo do algoritmo de inserção

```
1.  def gera_vizinhos_inserção(solucao_input, qtde_vizinhos, mat_dist):
2.    vizinhança = []
3.    Para i = 0 até qtde_vizinhos:
4.              Rota1 = remocao_com_peso
5.              Rota2 = sorteira outra rota dentro da solução
6.              Sorteia um ponto da rota1
7.              Calcula a sua melhor inserção na rota2
8.              Retira o ponto da rota1 e o insere na melhor inserção
    encontrada na rota2
9.              Elimina rotas vazias da solução
10.             Verifica factibilidade do vizinho novo
11.             Se for factível:
12.                       Insere na vizinhança
13.             Se não for factível:
14.                       Próxima iteração
15.
16.              Retorna vizinhança
```

Fonte: O autor.

Dentro de uma vizinhança, a solução inicial será a mesma e cada operação do algoritmo realizada gera um novo vizinho. A quantidade de vizinhos gerados é dada pelos parâmetros apresentados na seção 3.2.1.

É importante ressaltar que tanto para a inserção, quanto para a vizinhança de troca, vista a seguir, é realizada uma verificação de factibilidade da resposta. Na qual, se houver alguma rota com custo total acima do limite parametrizado anteriormente, essa solução é dada como infactível e não é adicionada ao conjunto de vizinhos gerados.

3.3.3 Estrutura de vizinhança – troca

A segunda estrutura de vizinhança abordada é a troca de pontos sorteados entre duas rotas. Neste método são sorteadas duas rotas e dois pontos aleatórios dentro de cada uma. Então os pontos são removidos de sua rota de origem e recolocadas dentro da outra rota, em uma posição definida pelo método de melhor inserção. Esses passos estão retratados na Figura 11 em formato de pseudocódigo.

Figura 11 – Pseudocódigo algoritmo troca

```
1. def gera_vizinhos_troca(solucao_input, qtde_vizinhos, mat_dist):
2.   vizinhança = []
3.   Para i = 0 até qtde_vizinhos:
4.           Sorteia rota1 e rota2
5.           Sorteia um ponto na rota 1
6.           Sorteia um ponto na rota 2
7.           Realiza a troca de acordo com a melhor inserção
8.           Elimina rotas vazias
9.           Verifica factibilidade do vizinho novo:
10.          Se for factível:
11.                  Insere na vizinhança
12.          Se não for factível:
13.                  Próxima iteração
14.    Retorna vizinhança
```

Fonte: O autor.

E a melhor inserção para a troca está na Figura 12.

Figura 12 - Algoritmo melhor inserção para swap

```
1. def best_insertion(rota1, rota2, ponto1, ponto2)
2.   Remove ponto1 da rota1 e ponto2 da rota2
3.   Para cada posição em rota2:
4.           Insere ponto1 na posição
5.           Calcula custo após inserção
6.   Escolhe o melhor cenário e realiza a inserção
7.   Realiza mesmos passos para a rota1 com o ponto2
8.   Retorna rota1_após_inserção, rota2_após_inserção
```

Fonte: O autor.

3.3.4 Estrutura de vizinhança – sorteio

A terceira estrutura de geração de vizinhos abordada neste trabalho é um sorteio entre a opção de inserção ou de troca. Neste método é realizada a escolha da probabilidade de acontecer a inserção, tendo a troca como vizinhança padrão e esse sorteio acontece durante a geração de cada vizinho. Como padrão para a resolução das instâncias utilizadas, foi utilizada uma chance de 20% para realizar a inserção.

Figura 13 - Algoritmo sorteio

```
1.  def gera_vizinhos_prob(insertion_prob = 0.2):
2.    vizinhança = []
3.    Para i = 0 até qtde_vizinhos:
4.            Método = sorteio entre inserção ou troca de acordo com
                              probabilidade input
5.            Sorteia rota1 e rota2
6.            Sorteia um ponto na rota 1
7.            Sorteia um ponto na rota 2
8.            Realiza melhor inserção de acordo com método sorteado
9.            Elimina rotas vazias
10.           Verifica factibilidade do vizinho novo:
11.           Se for factível:
12.                   Insere na vizinhança
13.           Se não for factível:
14.                   Próxima iteração
15.   Retorna vizinhança
```

Fonte: O autor.

3.3.5 Iteração do hill-climbing

A iteração proposta do *hill-climbing*, acontece a partir da determinação do método de vizinhança que será realizado. Após, são gerados a quantidade de vizinhos dada como argumento de entrada. Para cada conjunto de vizinhos gerados, pega-se o melhor vizinho. O critério de seleção do melhor vizinho é o menor valor máximo do custo das rotas que cada vizinho contém. Caso esse vizinho tenha uma resposta melhor que o já registrado, atualiza-se o valor de melhor solução local. Essas operações são repetidas até que se chegue a um ótimo local. Se for detectado um ótimo local, a iteração é encerrada e os melhores valores encontrados até então são dados como retorno dessa função. O ótimo local definido para este algoritmo é detecção de 3 conjuntos de vizinhanças seguidos sem melhora no melhor valor encontrado. Abaixo são retratadas as funções que retornam o melhor vizinho dentre o conjunto de vizinhança e a que realiza a iteração do *hill-climbing*.

Figura 14 – Pseudocódigo do algoritmo de avaliação de vizinhos

```
def get_melhor_vizinho (vizinhança, instancia):
  custos_maximos = []

  Para cada vizinho na vizinhança:
            Calcula custos de cada sub-rota
            Escolhe o maior custo entre eles
            Insere o maior custo em custos_maximos

  Escolhe o menor custo dentro de custos_maximos
  Pega a sua posição
  melhor_vizinho = vizinhança [posição melhor vizinho]

  Retorna melhor_vizinho, melhor_vizinho_custo
```

Fonte: O autor.

3.3.6 Estrutura do *hill-climbing*

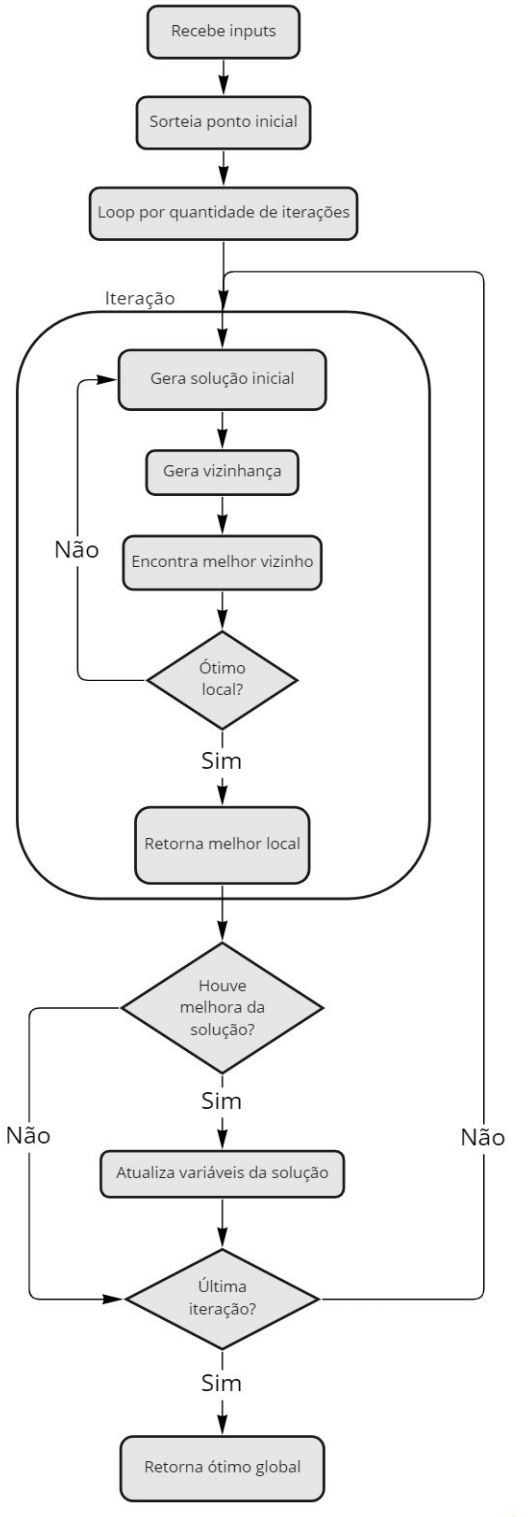
O *hill-climbing* proposto neste trabalho recebe como *inputs* a matriz de custos, a quantidade de iterações a realizar, a quantidade de vizinhos que devem ser gerados a cada iteração e o tipo de vizinhança que se deseja gerar. Após receber os inputs, é sorteado qual será o ponto inicial desta instância e iniciado o *loop* de iterações. A cada iteração, é gerada a quantidade de vizinhos determinada e analisado o melhor vizinho até chegar na definição de ótimo local proposta anteriormente. Ao cair no ótimo local, retorna o melhor vizinho encontrado nesta iteração. Caso haja melhoria na solução, salva-se a nova solução encontrada numa variável global e avança para a próxima iteração. Caso não ocorra melhoria, somente pula-se para a próxima iteração. A seguir nas figuras 15 e 16 estão retratados o código usado para essa estrutura, assim como um fluxograma que resume o *hill-climbing* proposto.

Figura 15 – Pseudocódigo do *hill-climbing*

```
def hill_climbing(instancia, qtde_iterações, qtde_vizinhos, tipo_vizinhança):


    melhor_solucao = None
    melhor_solucao_valor = infinito
    ponto_inicial_rnd = ponto aleatório

    Para i = 0 até qtde_iterações:
            Realiza uma iteração retornando:
                    solução_iteracao
                    custo_maximo_solução
            # Atualiza o valor da solução
                    Se custo_maximo_solução < melhor_solucao_valor:
                            melhor_solucao = solução_iteracao
                            melhor_solucao_valor = custo_maximo_solução


    Retorna melhor_solucao, melhor_solucao_valor,
```

Fonte: O autor.

Figura 16 - Fluxograma do *hill-climbing* proposto



Fonte: O autor.

4 RESULTADOS E DISCUSSÕES

Nesta seção serão apresentados os resultados da implementação do algoritmo proposto e análise de seu comportamento dentro das instâncias pré-definidas assim como na instância real proposta por este trabalho. Os valores de solução são sempre referentes ao máximo custo da solução.

4.1 Resultados instâncias tsplib95

As seguintes seções, destinam-se à análise do comportamento do algoritmo nas instâncias de teste do tsplib95. Os conjuntos foram separados por tamanho, como mencionado na seção 3.2.1, e serão analisados separadamente.

4.1.1 Resultados das instâncias pequenas

Para as instâncias consideradas pequenas, entre 17 e 48 pontos, foram encontrados os seguintes desempenhos. Essas instâncias foram resolvidas com parâmetro de geração de 50 vizinhos e 300 iterações do *hill-climbing*. A Tabela 5 mostra os resultados obtidos, onde em destaque está o melhor resultado global encontrado para este conjunto de instâncias.

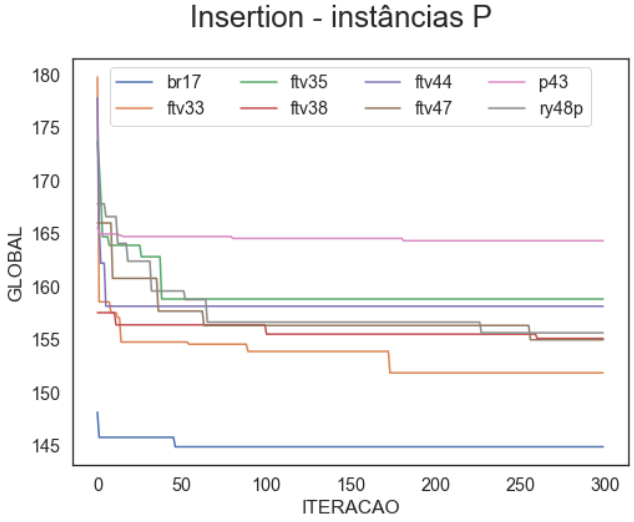
Tabela 5 - Resultados obtidos instâncias pequenas

	INSERÇÃO			TROCA			SORTEIO		
instância	inicial	global	melhoria	inicial	global	melhoria	inicial	global	melhoria
br17	179.16	144.86	19.14%	174.7	145.76	16.57%	179.16	143.97	19.64%
ftv33	179.79	151.85	15.54%	179.58	146.63	18.35%	177.35	143.66	19.0%
ftv35	176.81	158.82	10.18%	170.72	149.07	12.68%	176.81	148.46	16.03%
ftv38	174.64	155.1	11.19%	171.41	145.96	14.85%	176.74	145.35	17.76%
p43	165.55	164.32	0.74%	164.79	164.12	0.41%	165.02	164.08	0.57%
ftv44	179.04	158.14	11.67%	173.78	146.77	15.54%	170.26	146.63	13.88%
ftv47	169.68	154.97	8.67%	177.03	145.81	17.64%	**173.45**	**143.23**	**17.43%**
ry48p	167.81	155.63	7.26%	178.47	149.66	16.14%	179.59	148.22	17.47%
		Média	10.55%		Média	14.02%		Média	15.22%

Fonte: o autor.

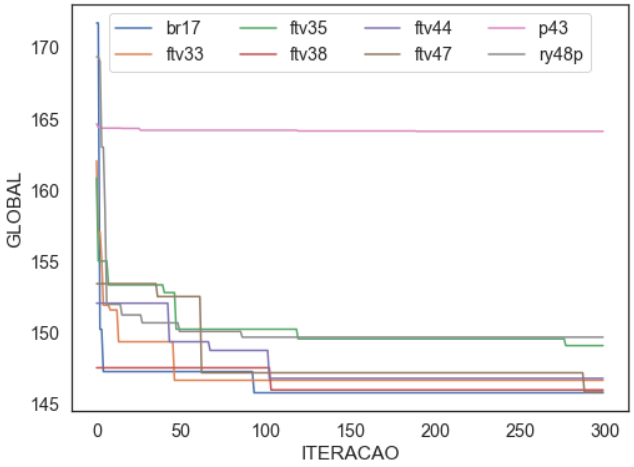
A evolução do valor do ótimo global é mostrada pelos Gráficos 17, 18 e 19 abaixo:

Figura 17 - Evolução do resultado global. Instâncias P - inserção



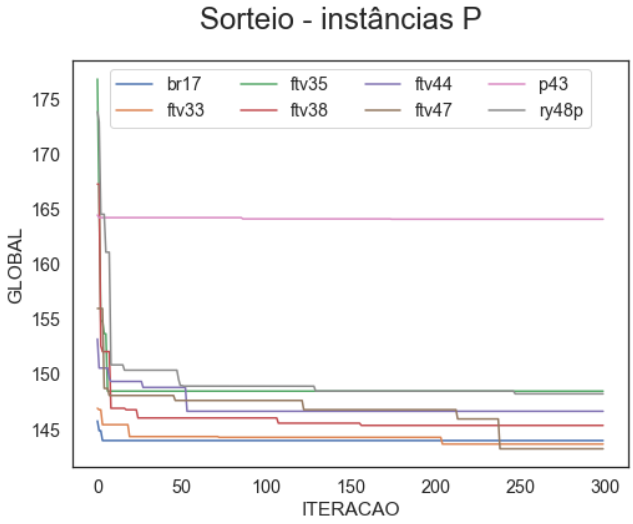
Fonte: o autor.

Figura 18 - Evolução do resultado global. Instâncias P - troca



Fonte: o autor.

Figura 19 - Evolução do resultado global. Instâncias P - sorteio



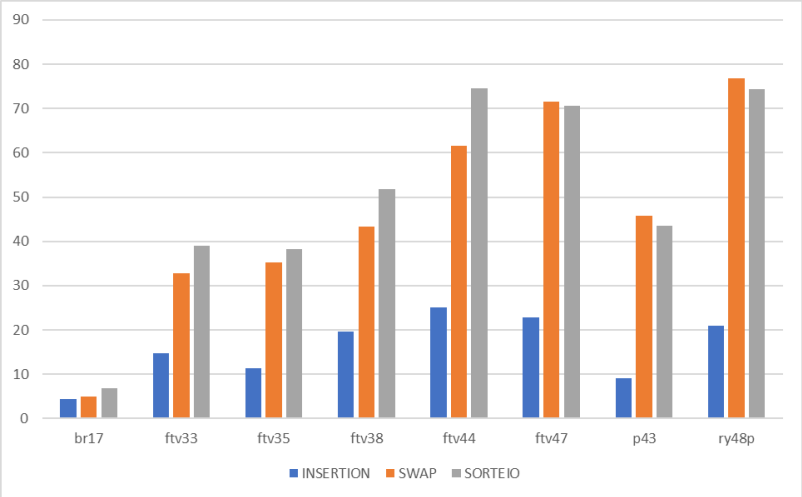
Fonte: o autor.

Observando os dados acima, percebe-se que os algoritmos que tiveram maior desempenho foram os que continham o método de troca em sua estrutura. Como ele gera duas trocas, ele usa a melhor inserção duas vezes dentro de um único vizinho. Isso acarreta em maior otimização do custo além de aumentar a heterogenia da vizinhança, que é favorável para o *hill-climbing*, pois dificulta que o algoritmo encontre um ótimo local.

Pelos gráficos das linhas de evolução, observa-se que em sua maioria, tiveram melhoria acentuada durante as primeiras iterações. Após isso, o algoritmo teve dificuldades em encontrar melhores respostas.

No que se refere ao custo computacional, os tempos de execução de 300 instâncias estão representados, em segundos, na Figura 20.

Figura 20 - Tempos de execução - instâncias P



Fonte: o autor.

4.1.2 Resultados das instâncias médias

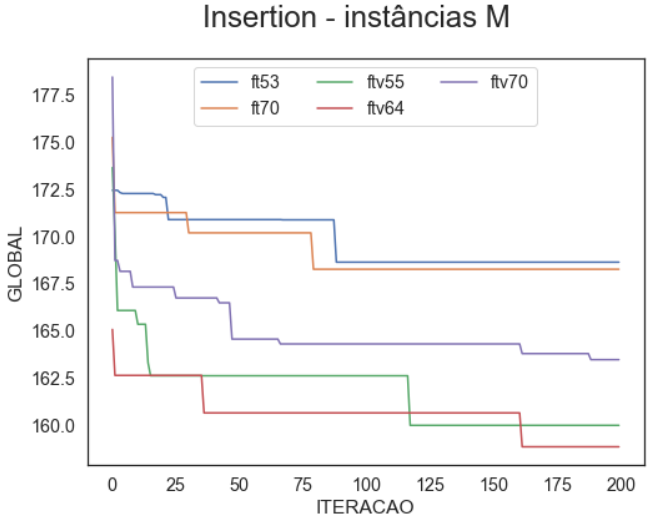
As instâncias médias são aquelas que contém entre 53 e 70 pontos em sua totalidade. Para essa categoria, foram realizadas 200 iterações, com gerações de 100 vizinhos por conjunto. Os resultados globais estão apresentados na Tabela 9 e gráficos da evolução do ótimo global de cada instância estão disponíveis nas Figuras 21, 22 e 23.

Tabela 6 - Resultados obtidos instâncias médias

	INSERÇÃO			TROCA			SORTEIO		
instância	inicial	global	melhoria	inicial	global	melhoria	inicial	global	melhoria
ft53	179.78	168.64	6.2%	175.32	163.53	6.72%	179.19	162.82	9.14%
ftv55	176.49	159.99	9.35%	176.56	150.31	14.87%	172.75	148.03	14.31%
ftv64	178.39	158.86	10.95%	176.02	148.79	15.47%	**179.36**	**147.44**	**17.79%**
ft70	178.69	168.27	5.83%	179.81	164.37	8.59%	179.45	164.47	8.35%
ftv70	180.0	163.48	9.18%	175.96	150.39	14.53%	178.52	151.8	14.97%
		Média	8.3%		Média	12.04%		Média	12.91%

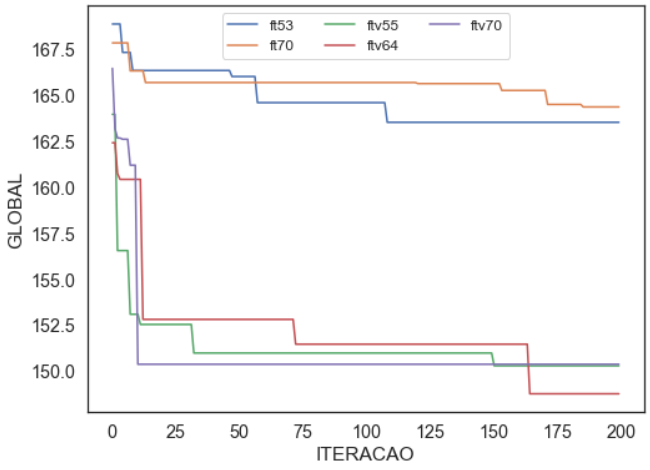
Fonte: o autor.

Figura 21 - Evolução do resultado global. Instâncias M - inserção



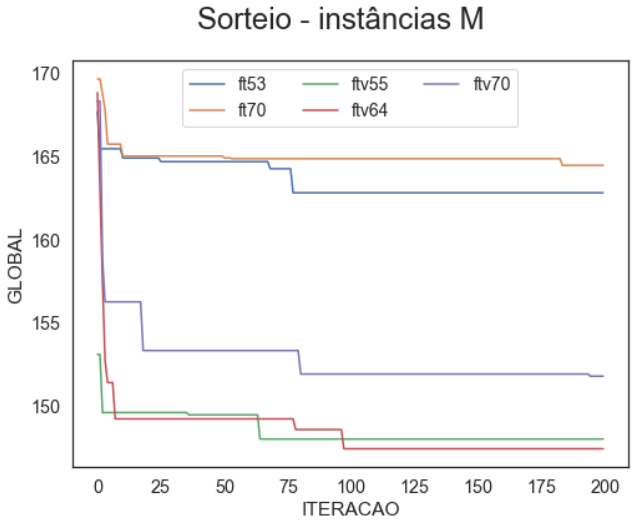
Fonte: o autor.

Figura 22 - Evolução do resultado global. Instâncias M - troca



Fonte: o autor.

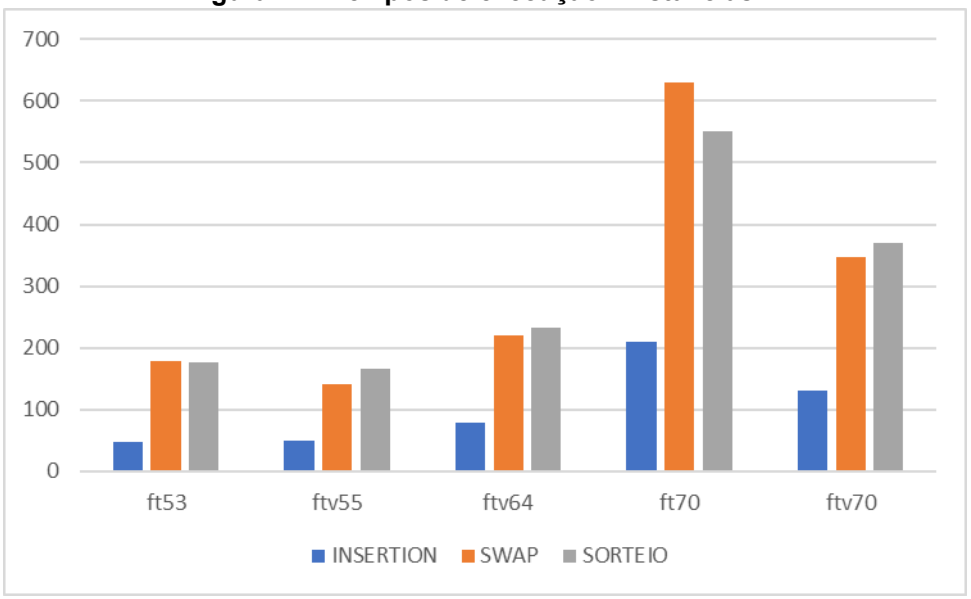
Figura 23 - Evolução do resultado global. Instâncias M - sorteio



Fonte: o autor.

Nas instâncias médias, pôde-se observar, em média, uma piora nos índices de melhoria das soluções. Cada algoritmo perdeu cerca de 2 pontos percentuais em sua média, o que pode ser explicado pelo aumento da complexidade do problema ao aumentar a quantidade de pontos em cada instância. Para que houvesse índices de melhorias mais positivos, seria necessário aumentar a quantidade de iterações ou de vizinhos gerados. Porém, como pode-se observar pelo gráfico de tempo de execução abaixo, o custo computacional seria muito maior.

Figura 24 - Tempos de execução - instâncias M



Fonte: o autor.

4.1.3 Resultados das instâncias grandes

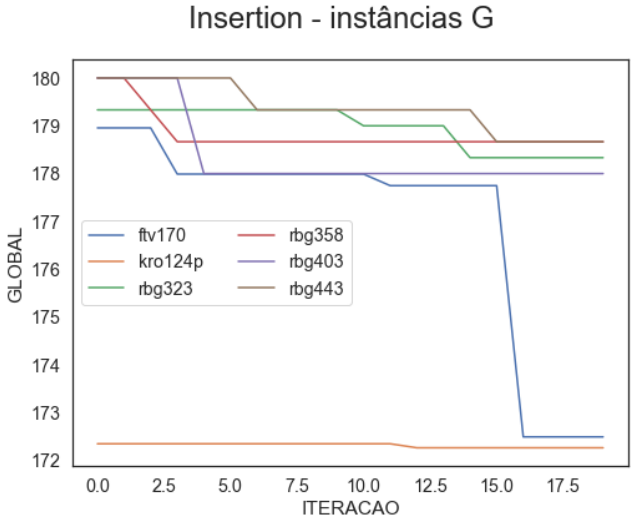
As instâncias grandes abrangem aquelas que contém de 170 a 443 pontos. Elas foram resolvidas com 20 iterações e gerando 50 vizinhos em cada conjunto. Esses valores foram menores por demandarem mais esforço computacional que as outras instâncias, o que resultaria em um tempo de execução do algoritmo muito grande. Abaixo seguem os resultados da implementação do algoritmo nessas instâncias, ilustrados na Tabela 7 e nas Figuras 25, 26 e 27.

Tabela 7 - Resultados obtidos instâncias grandes

	INSERÇÃO			TROCA			SORTEIO		
instância	inicial	global	melhoria	inicial	global	melhoria	inicial	global	melhoria
kro124p	179.5	172.26	4.03%	179.17	165.52	7.62%	**176.36**	**163.91**	**7.06%**
ftv170	178.96	172.49	3.61%	179.53	165.66	7.72%	179.68	168.8	6.05%
rbg323	179.33	178.33	0.56%	179.67	177.33	1.3%	180.0	178.0	1.11%
rbg358	180.0	178.67	0.74%	180.0	177.33	1.48%	179.67	178.0	0.93%
rbg403	180.0	178.0	1.11%	180.0	178.67	0.74%	179.0	178.0	0.56%
rbg443	180.0	178.67	0.74%	180.0	177.33	1.48%	178.0	176.0	1.12%
		Média	1.8%		Média	3.39%		Média	2.8%

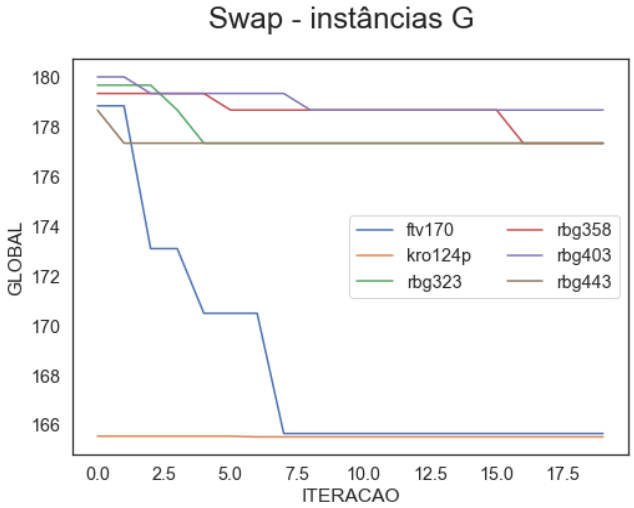
Fonte: o autor.

Figura 25 - Evolução do resultado global. Instâncias G - inserção



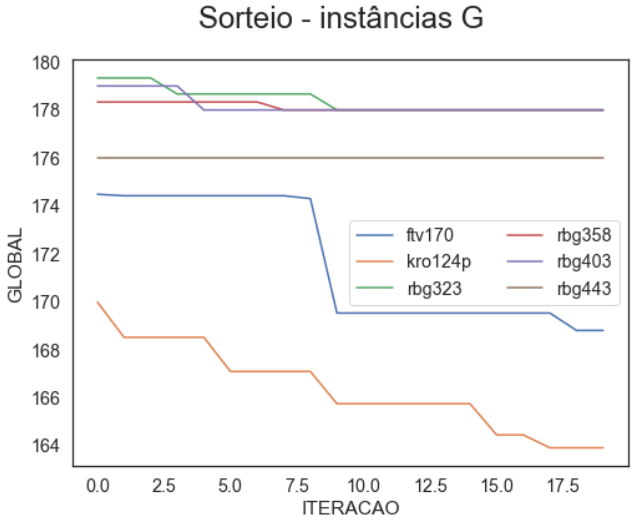
Fonte: o autor.

Figura 26 - Evolução do resultado global. Instâncias G - troca



Fonte: o autor.

Figura 27 - Evolução do resultado global. Instâncias G - sorteio

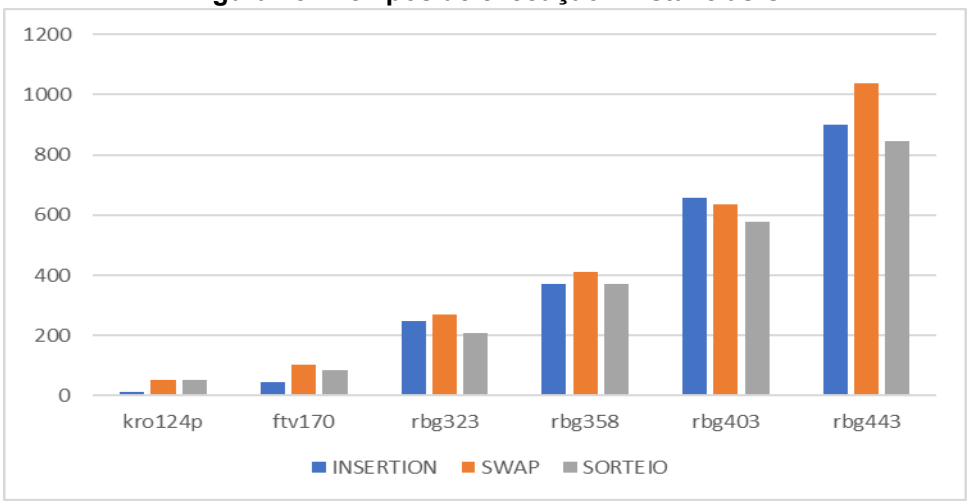


Fonte: o autor.

Com os dados apresentados acima, pode-se observar uma grande piora na performance do algoritmo em instâncias maiores. Isso se explica pela característica das vizinhanças, que mudam apenas um ponto ou par de pontos por vez. Além disso, outro fator que contribui é a pequena quantidade de iterações para instâncias grandes, limitando a área de busca do algoritmo. Para instâncias menores, essa mudança pode ser uma fração significativa da resposta. Para grandes instâncias, pode acabar sendo quase insignificante. Pela primeira vez neste trabalho, pode-se observar uma piora dos resultados médios da vizinhança de sorteio em relação ao algoritmo de troca.

Os custos computacionais também se tornam maiores, como já mencionado, e isso pode ser observado no seguinte gráfico.

Figura 28 - Tempos de execução - instâncias G



Fonte: o autor.

Neste caso, fica evidente a tendência de crescimento exponencial do tempo de processamento do algoritmo em função do número de pontos da solução.

4.2 Resultados da instância real

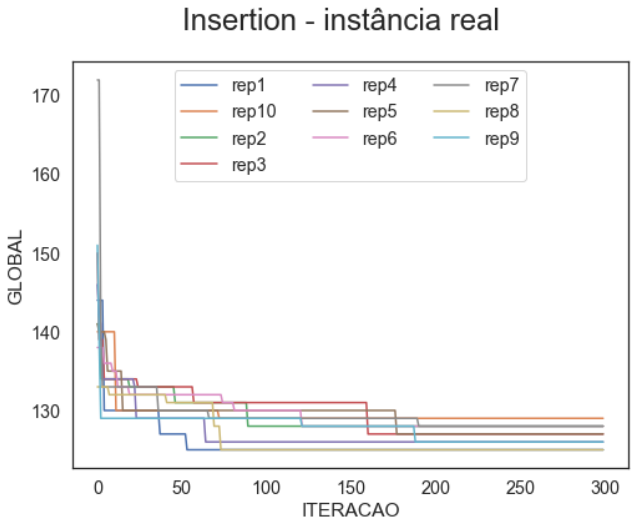
Considerando que a instância real tem no total 13 pontos, foram adotados como parâmetros os mesmos para as instâncias do *tsplib95* de tamanho pequeno. Sendo eles 300 iterações e com geração de 50 vizinhos. Para essa instância foram feitas 10 repetições do *hill-climbing* completo a fim de se obter uma quantidade de dados que retratasse melhor o comportamento do algoritmo.

Os resultados seguem na Tabela 17 e nas Figuras 29, 30 e 31.

Tabela 8 - Resultados obtidos instância real

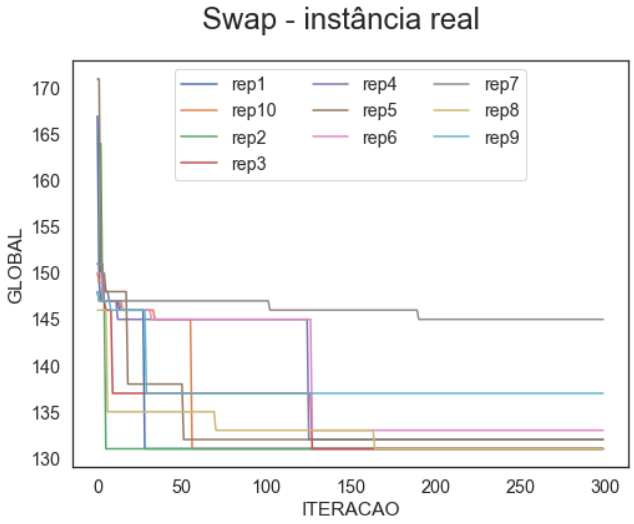
	INSERÇÃO			TROCA			SORTEIO		
instância	inicial	global	melhoria	inicial	global	melhoria	inicial	global	melhoria
1	168	125	25.6%	175	131	25.14%	171	125	26.9%
2	178	128	28.09%	178	131	26.4%	**176**	**125**	**28.98%**
3	154	127	17.53%	161	131	18.63%	162	125	22.84%
4	171	126	26.32%	169	132	21.89%	158	125	20.89%
5	170	127	25.29%	179	132	26.26%	175	125	28.57%
6	154	128	16.88%	168	133	20.83%	156	125	19.87%
7	172	128	25.58%	161	145	9.94%	160	125	21.88%
8	166	125	24.7%	165	131	20.61%	166	125	24.7%
9	174	126	27.59%	164	137	16.46%	166	125	24.7%
10	172	129	25.0%	163	131	19.63%	163	125	23.31%
		Média	24.26%		Média	20.58%		Média	24.26%

Figura 29 - Evolução do resultado global. Instâncias real - inserção



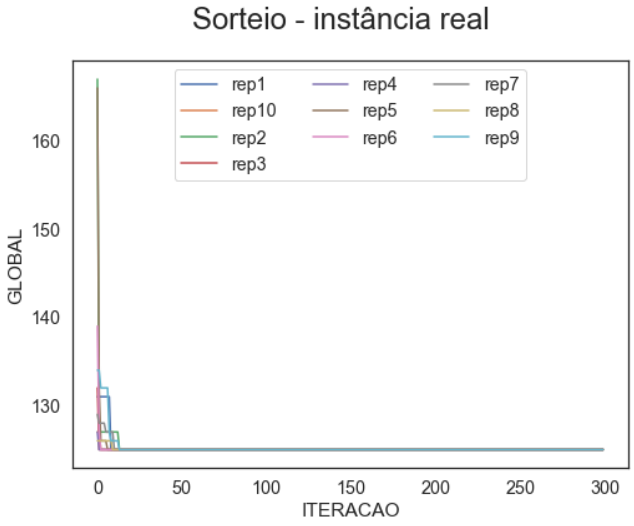
Fonte: o autor.

Figura 30 - Evolução do resultado global. Instâncias real - troca



Fonte: o autor.

Figura 31 - Evolução do resultado global. Instâncias real - sorteio

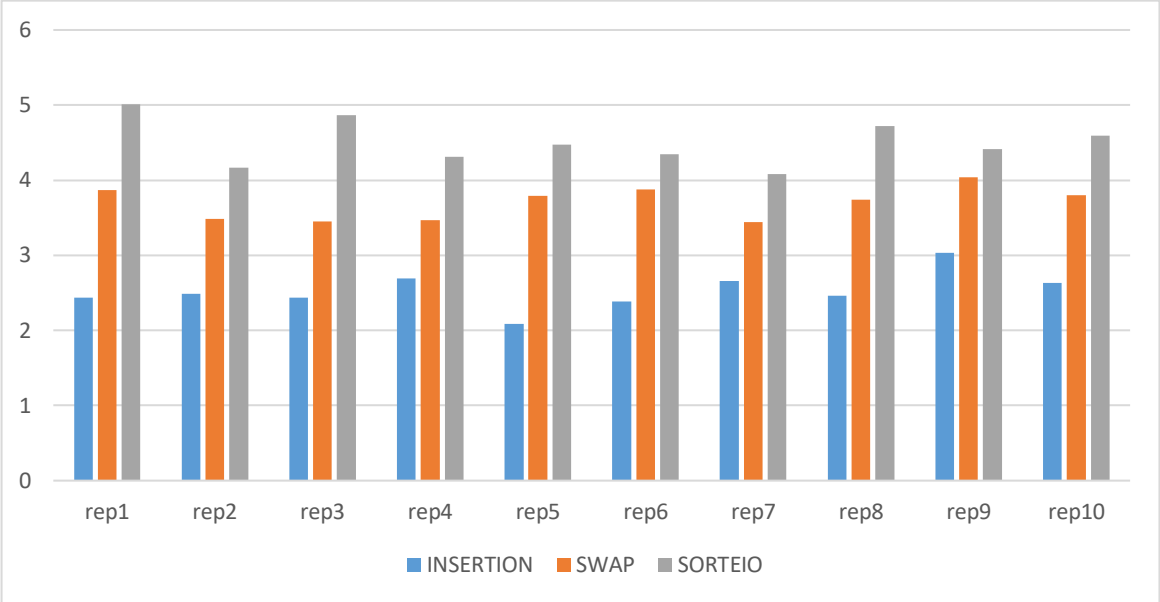


Fonte: o autor.

Aqui observa-se que o *swap* teve a pior performance de resultado entre os três tipos de vizinhança. Embora a inserção e o sorteio tenham ficado com médias iguais, o destaca-se o comportamento de melhoria brusca logo nas primeiras iterações no algoritmo com o método de sorteio, o que pode ser explicado pela mistura de vizinhanças ocasionando uma busca mais ampla dentro do domínio do problema.

Os tempos de execução desta instância, retratadas na Figura 32, e foram condizentes ao das menores instâncias do *tsplib95* no que tange à ordem de grandeza.

Figura 32 - Tempos de execução - instância real



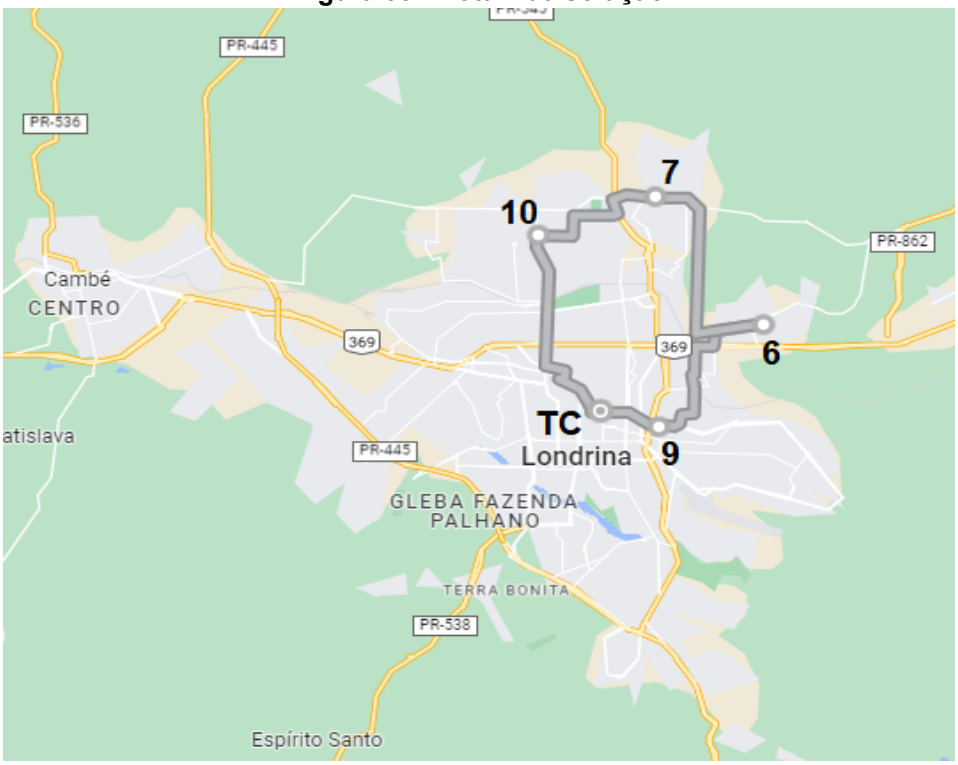
Fonte: o autor

A melhor solução, aquela que teve menor tempo máximo e menor distância percorrida entre as outras, para a rota da pesquisa da cesta básica teve a seguinte configuração:

- Quantidade de alunos necessários: 3
- Distância total percorrida = 331.8 km
- Tempo total (todas as rotas) = 368 minutos
- Rota 1: [9, 6, 7, 10]
  - Custo de deslocamento = 45 minutos
  - Custo de coleta de dados = 80 minutos
  - Custo de tempo total = 125 minutos
  - Custo de distância = 22,5 km
- Rota 2: [1, 0, 11, 3]
  - Custo de deslocamento = 42 minutos
  - Custo de coleta de dados = 80 minutos
  - Custo de tempo total = 122 minutos
  - Custo de distância = 22,7 km
- Rota 3: [5, 8, 2, 4]
  - Custo de deslocamento = 45 minutos
  - Custo de coleta de dados = 80 minutos
  - Custo de tempo = 125 minutos
  - Custo de distância = 22,9 km

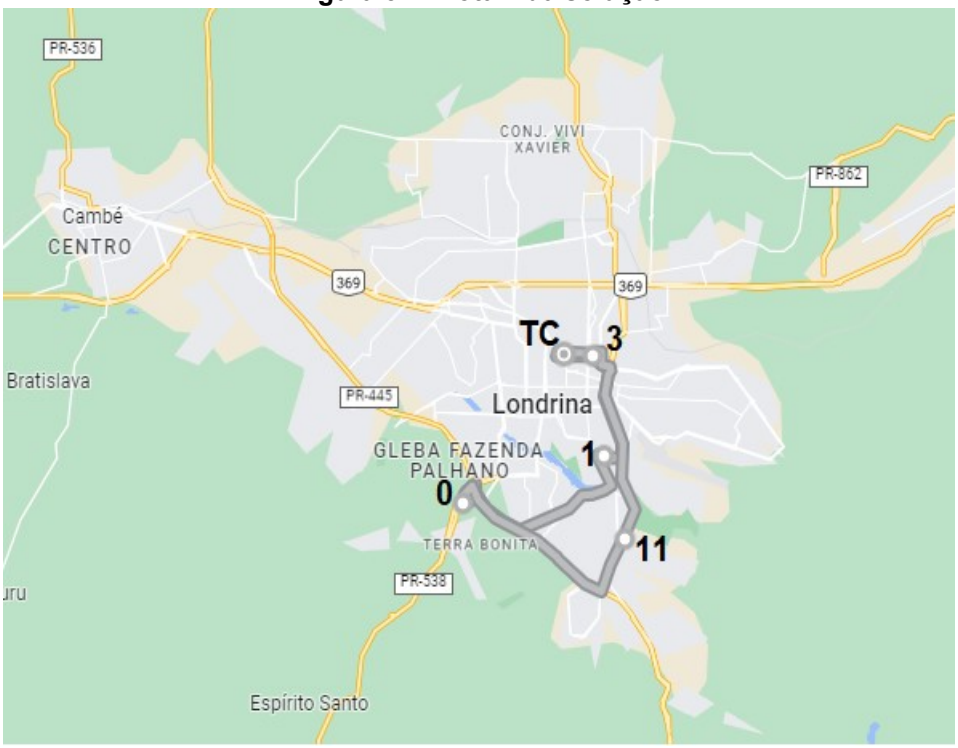
Os mapas correspondentes de cada rota estão representados abaixo. Onde TC é o Terminal Central, ponto de início e fim de todas as rotas.

Figura 33 - Rota 1 da solução



Fonte: o autor

Figura 34 - Rota 2 da solução



Fonte: o autor

Figura 35 - Rota 3 da solução

Fonte: o autor

5 CONCLUSÃO

Este trabalho teve como objetivo a proposta de um algoritmo *hill-climbing* para um problema real de mTSP assimétrico e para isso foi necessário realizar uma pesquisa exploratória dos métodos e heurísticas consolidados na literatura, assim como o levantamento de dados e outras instâncias. Os dados de tempo e distâncias entre os pontos instância real foram coletados a partir do *Google Maps* e o tsplib95 foi escolhido como fonte das demais instâncias de teste.

Começando pela geração da solução inicial e a inclusão de um fator de aleatoriedade em um algoritmo determinístico, o vizinho mais próximo, dando origem ao VMP estocástico. Em seguida, foram desenvolvidas as estruturas geradoras de vizinhanças pelos métodos de inserção, troca e sorteio. Então, foram estruturadas as iterações do *hill-climbing* e o *loop* de iterações, que retorna o melhor resultado global. Além disso, antes de resolver as instâncias, foram determinados parâmetros do algoritmo para cada instância, com base em seu tamanho.

Para a resolução das instâncias, cada uma delas foi convertida, por meio de bibliotecas públicas do *Python*, para um formato compatível ao programa feito e correspondente ao problema real analisado.

Após o cálculo das soluções de todas as instâncias e com base nos resultados apresentados, pode-se concluir que o algoritmo tem melhor performance quando utilizado para calcular instâncias pequenas, com até aproximadamente 50 pontos. Neste cenário, pode-se esperar melhora do maior custo da solução inicial em média de 10,5% para a vizinhança inserção, 14,02% para a vizinhança utilizando a troca e 15,22% para o sorteio com 20% de chance para realizar inserção.

Após isso, aumenta-se muito o custo computacional para realizar este algoritmo com pouco retorno no que tange à melhoria da solução inicial. Como foi o caso da instância “rbg443”, de 443 pontos, que levou 898 segundos para realizar 20 iterações, tendo uma melhoria de 0,74% da solução inicial. Isso ocorre por conta de uma limitação do algoritmo, que não produz uma vizinhança suficientemente diversa para direcionar as próximas iterações do *hill-climbing*, fazendo com que, mesmo com muitas iterações, o algoritmo não se aprofunde significantemente no domínio de soluções possíveis.

Essa limitação pode ser explorada em pesquisas futuras pelo aumento da quantidade de trocas ou inserções para cada vizinho gerado ou realizando trocas com conjuntos de pontos maiores. Além disso, é sugerido que algoritmos futuros atribuam maior peso à minimização da quantidade de viajantes, melhorando a eficiência de recursos geral do problema.

Em relação à solução da instância real, tendo ela 13 pontos em sua totalidade, se enquadrou na faixa das instâncias pequenas, apresentando resultados satisfatórios de melhoria do maior custo e chegando a apresentar uma média de 24,26% para uma amostra de 10 repetições do *hill-climbing* proposto. Com base na solução encontrada, a coleta de dados em supermercados pode ser realizada por três pesquisadores simultâneos percorrendo as rotas e coletando dados paralelamente em até 125 minutos, 30,56% abaixo do tempo limite de 3 horas.

As rotas aqui propostas poderão ser utilizadas pelos pesquisadores do NUPEA, mesmo que seccionadas, pois majoritariamente apresentam os trechos mais rápidos que se pode percorrer entre dois pontos de coleta. É necessário, contudo, se atentar ao sentido das rotas, por conta da característica assimétrica do deslocamento na cidade. Além disso, pode-se adicionar novos pontos e formar uma nova matriz de tempos a fim de englobar novos pontos de coleta ou outros pontos base, como as residências de cada pesquisador ou a própria UTFPR. E para calcular as rotas, o algoritmo proposto estará disponível nos apêndices deste trabalho a quem possa interessar.

REFERÊNCIAS

AARTS, E. H. L.; LENSTRA, J. K. ***Local search in combinatorial optimization.*** Nova Jersey: *John Wiley & Sons Ltd.,* 1997.

AGARWAL, S; CHUA, Y.H.; SONG, C. ***Inflation expectations of households and the upgrading channel.*** Disponível em: <http://dx.doi.org/10.2139/ssrn.3559176>. Acesso em 27 de maio de 2022.

ANGEL, R. D. *et al*. *Computer assisted school bus scheduling.* ***Management Science.*** Vol. 18, n. 6, p.b279-b288, 1972.

BEKTAS, T. *The multiple traveling salesman problem: an overview of formulations and solutions procedures.* ***Omega the international journal of management science.*** Vol. 34, p. 209-219, 2006.

BELLMORE, M.; HONG, S. *Transformation of multisalesman problem to the standard traveling salesman problem.* ***Journal of the association for computing machinery.*** Vol. 20, n. 3, p. 500-504, 1974.

DANTZIG, G.; FULKERSON, D. R.; JOHNSON, Selmer M. *Solution of a large-scale travelling salesman problem.* ***Journal of the operations research society of America.*** Vol. 2, n. 4, p.393-410, 1954.

GORENSTEIN, S. *Printing press scheduling for multi-edition periodicals.* ***Management Science.*** Vol. 16, n. 6, p.b373-b383, 1970.

INFLAÇÃO por faixa de renda familiar em 2020. **Banco Central do Brasil**, 2020. Disponível em: <https://www.bcb.gov.br/conteudo/relatorioinflacao/EstudosEspeciais/EE098_Inflacao_por_faixa_de_renda_familiar_em_2020.pdf>. Acesso em 08 de maio de 2021.

JOHNSON, D. S. *et al. Experimental analysis of heuristics for the ATSP. In:* ***The traveling salesman problems and its variations.***Nova York: *Springer Science & Business Media,* 2002. p.445-486.

KANELLAKIS, P.C.; PAPADIMITROU, C.H. *Local search for the asymmetric traveling salesman problem.* ***Operations Research.*** Vol. 28, n. 5, p. 1086-1099, 1980.

KIZILATEŞ-EVIN, G.; NURIYEVA, F. *On the nearest neighbor algorithms for the traveling salesman problem.* ***Advances in Intelligent Systems and Computing.*** vol. 225, p. 111-118, 2013.

NILSSON, C. *Heuristics for the traveling salesman problem. LinköpingUniversity*.

NUPEA. Inflação da cesta básica em Londrina. [S.l] [2021]. Disponível em: <https://www.nupea.org/infla%C3%A7%C3%A3o-da-cesta-b%C3%A1sica/infla%C3%A7%C3%A3o-da-cesta-b%C3%A1sica-em-londrina>. Acessoem 08 de maio de 2021.

PEARL, Judea. ***HEURISTICS:*** *intelligent search strategies for computer problem solving. Reading, MA: Addison-Wesley Publishing Company,* 1984.

POTVIN, J-Y.; LAPALME, G.; ROUSSEAU, J-M. *A generalized k-opt exchange procedure for the mtsp.* ***INFOR: Informations Systems and Operational Research.*** Vol.27, n. 4, p. 474-481, 1989.

PUNNEN, A.P. *The traveling salesman problem: applications, formulations and variations*. *In*: GUTIN, G. (Ed.); PUNNEN, A.P. (Ed.). The ***traveling salesman problem and its variations***. Nova York: *Springer Science & Business Media*, 2002. p.1-24.

RAMBALDUCCI, M. J. G.; FELTRIN, B. R. *The impact of the monetary value of the basket of goods in the economy of Londrina/PR – Brazil.* ***Journal of International Scientific Publications.***Vol.12, p.37-51, 2018.

RATLIFF, H. D.; ROSENTHAL, A. S.*Order picking in a rectangular warehouse: a solvable case of the traveling salesman problem.* ***Operations Research.*** Vol.31, n.3, p.507-521, 1983.

ROBERTI, R.; TOTH, P. *Models and algorithms for the assymetric traveling salesman problem: an experimental comparison.* ***Euro Journal on Transportation and Logistics.*** Vol.1, n.2, p.113-133, 2012.

ROSENKRANTZ, D.J.; STEARNS, R.E.; LEWIS II, P.M. *An analysis of several heuristics for the traveling salesman problem.****SIAM Journal on Computing.*** vol.6, n.3, p. 563-581,1977.

RUSSEL, R. A.*An effective heuristic for the M-Tour traveling salesman problem with some side conditions.* ***Operations Research.*** Vol. 25, n.3, p. 517-524, 1977.

APÊNDICE A - Códigos em *python* para o VMP Estocástico

```
import random as rd
global m_grande
m_grande = 999999

def vetor_probabilidade(vetor):
    copia = vetor.copy()
    for i in copia:
        if i == m_grande:
            copia[copia.index(i)] = 0

    valor_max = max(copia)
    vetor_qualidade = list()

    for i in copia:
        if i != 0:
            vetor_qualidade.append((valor_max - i) ** 2)
        else:
            vetor_qualidade.append(0)

    soma_total = sum(vetor_qualidade)
    vetor_probab = list()

    if soma_total == 0 and valor_max != 0:
        return False

    else:
        for i in vetor_qualidade:
            vetor_probab.append(i / soma_total)

        prob_acum = 0
        vetor_prob_acum = list()
        for i in vetor_probab:
            prob_acum += i
            vetor_prob_acum.append(prob_acum)

        return vetor_prob_acum

def prox_ponto(vetor_prob):

    x = rd.random()
    i = 0
    while vetor_prob[i] < x:
        i += 1

    return i

def coloca_m_grande(instancia, i, j):

    matriz = [linha.copy() for linha in instancia]

    for arc in matriz[i]:
        matriz[i][matriz[i].index(arc)] = m_grande

    for line in matriz:
        line[j] = m_grande

    matriz[j][i] = m_grande

    return matriz

def vmp_estocastico(mat_custos, ponto_inicial):
    matriz = [linha.copy() for linha in mat_custos]
    solucao = list()
    solucao.append(ponto_inicial)
    i = 0
    ponto_i = ponto_inicial
    pontos_faltantes = [i for i in range(len(mat_custos))]
    pontos_faltantes.remove(ponto_i)
    for linha in matriz:
```

```
70.         linha[ponto_i] = m_grande
71.
72.     while len(solucao) < len(mat_custos):
73.
74.         ponto_atual = solucao[i]
75.
76.         destinos = matriz[ponto_atual]
77.         vetor_prob = vetor_probabilidade(destinos)
78.
79.         if vetor_prob == False:
80.             proximo_ponto = rd.choice(pontos_faltantes)
81.             pass
82.         else:
83.             proximo_ponto = prox_ponto(vetor_prob)
84.
85.
86.         solucao.append(proximo_ponto)
87.         matriz = coloca_m_grande(instancia=matriz, i=ponto_atual, j=proximo_ponto)
88.
89.         pontos_faltantes.remove(proximo_ponto)
90.         i += 1
91.
92.         if len(pontos_faltantes) == 1:
93.             solucao.append(pontos_faltantes[0])
94.
95.
96.     return solucao
97.
```

APÊNDICE B - Códigos em *python* para o *hill-climbing*

```
import random
import NearestNeighbor
import time
import LerArrays

global tempo_coleta, tempo_maximo, m_grande
m_grande = 999999
tempo_maximo = 180
tempo_coleta = 20

#apaga o ponto base da matriz original e faz uma matriz com os custo de saída e chegada no ponto base
def format_matrizes(matriz_instancia, ponto_base):
    matriz_retorno = [linha.copy() for linha in matriz_instancia]
    i = 0
    for linha in matriz_retorno:
        linha[i] = m_grande
        i += 1

    global custos_base
    custos_base = [[], []]

    custos_base[0] = matriz_retorno[ponto_base].copy()
    matriz_retorno.pop(ponto_base)

    for n1 in matriz_retorno:
        custos_base[1].append(n1[ponto_base])
        n1.pop(ponto_base)

    i = 0

    return matriz_retorno

def format_matrizes_km(matriz_instancia, ponto_base):
    matriz_retorno = [linha.copy() for linha in matriz_instancia]
    i = 0
    for linha in matriz_retorno:
        linha[i] = 999999
        i += 1

    global custos_base_km
    custos_base_km = [[], []]

    custos_base_km[0] = matriz_retorno[ponto_base].copy()
    matriz_retorno.pop(ponto_base)

    for n1 in matriz_retorno:
        custos_base_km[1].append(n1[ponto_base])
        n1.pop(ponto_base)

    i = 0

    return matriz_retorno


#calculo do custo da solução:
def get_rout_cost(instancia, rota):
    matriz = [linha.copy() for linha in instancia]
    vetorsolucao = rota.copy()
    custoTotal = 0
    qtde_pontos = len(vetorsolucao)
    ponto_zero = vetorsolucao[0]
    ponto_fim = vetorsolucao[len(vetorsolucao) - 1]

    # print(f'rota analise custo: {vetorsolucao}')

    for i in range(len(vetorsolucao) - 1):
        pontoAtual = vetorsolucao[i]
        proxPonto = vetorsolucao[i + 1]
```

```
69.         arco = matriz[pontoAtual][proxPonto]
70.         custoTotal = custoTotal + arco
71.         # print(f'Ponto atual: {pontoAtual}, Próx ponto: {proxPonto}')
72.
73.
74.     custoTotal = custoTotal + custos_base[0][ponto_zero] + custos_base[1][ponto_fim]
75.
76.     #aplica tempos de coleta
77.     custoTotal = custoTotal + (qtde_pontos * tempo_coleta)
78.
79.     return custoTotal
80.
81. def hill_climbing(instancia, qtde_iterações, qtde_vizinhos, tipo_vizinhança):
82.     start_time = time.time()
83.     mat_dist = [linha.copy() for linha in instancia]
84.     melhor_solucao = None
85.     melhor_solucao_valor = float('inf')
86.
87.     relatorio = list()
88.
89.     ponto_inicial_rnd = random.randint(0, len(mat_dist) - 1)
90.
91.     for i in range(qtde_iterações):
92.         # Pega o melhor valor dentro da iteração
93.         sol_iteracao, sol_iteracao_max_cost, sol_inicial_max_cost, qt_viajantes =
    iteracao_hillclimbing(tipo_vizinhança=tipo_vizinhança,
94.
    qtde_vizinhos=qtde_vizinhos,
95.
    instancia=mat_dist,
96.
    ponto_inicial=ponto_inicial_rnd)
97.
98.         if sol_iteracao_max_cost < melhor_solucao_valor:
99.             melhor_solucao = sol_iteracao
100.              melhor_solucao_valor = sol_iteracao_max_cost
101.
102.          relatorio.append((i, sol_inicial_max_cost, sol_iteracao_max_cost,
    melhor_solucao_valor))
103.
104.     tempo_execucao = time.time() - start_time
105.     return melhor_solucao, melhor_solucao_valor, relatorio, tempo_execucao
106.
107.  def iteracao_hillclimbing(tipo_vizinhança, qtde_vizinhos, instancia, ponto_inicial):
108.
109.     sol_vmp = NearestNeighbor.vmp_estocastico(instancia, ponto_inicial)
110.     qtde_viajantes, solucao_candidata = gera_solucao_inicial(rota=sol_vmp,
    instancia=instancia)
111.
112.     custos_solucao_candidata = [get_rout_cost(instancia, rota) for rota in
    solucao_candidata]
113.     solucao_inicial_valor = melhor_custo_max = max(custos_solucao_candidata)
114.
115.     no_improvement_iteractions = 0
116.     otimo_local = False
117.     while not otimo_local:
118.
119.         if tipo_vizinhança == 'swap':
120.             vizinhança = gera_vizinhos_swap(solucao_candidata, qtde_vizinhos,
    instancia)
121.
122.         elif tipo_vizinhança == 'insertion':
123.             vizinhança = gera_vizinhos_insertion(solucao_candidata, qtde_vizinhos,
    instancia)
124.
125.         elif tipo_vizinhança == 'probability':
126.             vizinhança = gera_vizinhos_prob(solucao_candidata, qtde_vizinhos,
    instancia, 0.2)
127.
```

```
128.
129.          qtde_vizinhos_gerados = len(vizinhança)
130.          # print(f'qtde de vizinhos factíveis = {qtde_vizinhos_gerados}')
131.
132.          if qtde_vizinhos_gerados > 0:
133.              melhor_vizinho, melhor_vizinho_max_cost = get_melhor_vizinho(vizinhança,
    instancia)
134.          else:
135.              melhor_vizinho = solucao_candidata
136.              custos_solucao_candidata = [get_rout_cost(instancia, rota) for rota in
    melhor_vizinho]
137.              melhor_vizinho_max_cost = max(custos_solucao_candidata)
138.
139.
140.          if melhor_vizinho_max_cost < melhor_custo_max:
141.              melhor_custo_max = melhor_vizinho_max_cost
142.              solucao_candidata = melhor_vizinho
143.              no_improvement_iteractions = 0
144.          else:
145.              no_improvement_iteractions += 1
146.
147.              if no_improvement_iteractions >= 3:
148.                  otimo_local = True
149.
150.          qtde_viajantes_apos_iteracao = len(melhor_vizinho)
151.
152.      return melhor_vizinho, melhor_custo_max, solucao_inicial_valor,
    qtde_viajantes_apos_iteracao
153.
154.
155.  def gera_solucao_inicial(instancia, rota):
156.
157.      rota_copy = rota.copy()
158.      rota_atual = list()
159.      rota_hipotética = rota_atual.copy()
160.      rotas_definitivas = list()
161.
162.      ponto_anterior = 'a'
163.      for i in rota_copy:
164.
165.          if not type(ponto_anterior) == str:
166.              rota_hipotética.append(ponto_anterior)
167.
168.          rota_hipotética.append(i)
169.          custo_rota_hipotética = get_rout_cost(instancia, rota_hipotética)
170.
171.
172.          if custo_rota_hipotética <= tempo_maximo:
173.              rota_atual = rota_hipotética.copy()
174.              ponto_anterior = 'a'
175.
176.          else:
177.              rota_append = rota_atual.copy()
178.              rotas_definitivas.append(rota_append)
179.              rota_hipotética.clear()
180.              rota_atual.clear()
181.              ponto_anterior = i
182.
183.      #só coloca na resposta se sobrou pontos no último vetor
184.      if not len(rota_hipotética) == 0:
185.          rotas_definitivas.append(rota_hipotética)
186.
187.
188.      qtde_min_viajantes = len(rotas_definitivas)
189.      return qtde_min_viajantes, rotas_definitivas
190.
191.  def gera_vizinhos_prob(solucao_input, qtde_vizinhos, mat_dist, insertion_prob = 0.2):
192.
193.      vizinhos = []
```

```
194.
195.      for _ in range(qtde_vizinhos):
196.          aleatorio = random.random()
197.          if aleatorio <= insertion_prob:
198.              metodo = 'insertion'
199.          else:
200.              metodo = 'swap'
201.
202.          # vetor pra usar durante uma iteração
203.          rotas_iteracao = [linha.copy() for linha in solucao_input]
204.
205.          # sorteia duas rotas diferentes
206.          rota_sorteado1 = random.choice(rotas_iteracao)
207.          rota_sorteado2 = random.choice(rotas_iteracao)
208.          while rota_sorteado2 == rota_sorteado1:
209.              rota_sorteado2 = random.choice(rotas_iteracao)
210.
211.          # sorteia uma posiçao em cada rota e salva o valor do elemento neste índice
212.          sorteio_ind_rota1 = random.randint(0, len(rota_sorteado1) - 1)
213.          sorteio_ind_rota2 = random.randint(0, len(rota_sorteado2) - 1)
214.
215.          ponto_sorteio1 = rota_sorteado1[sorteio_ind_rota1]
216.          ponto_sorteio2 = rota_sorteado2[sorteio_ind_rota2]
217.
218.          # Faz o swap OU insertion
219.          rota1_pos_swap, rota2_pos_swap = best_insertion(mat_dist, list1=rota_sorteado1,
    list2=rota_sorteado2,
220.                                                          point1=ponto_sorteio1,
    point2=ponto_sorteio2, method=metodo)
221.
222.          # remonta o vetor de rotas
223.          rotas_iteracao.remove(rota_sorteado1)
224.          rotas_iteracao.remove(rota_sorteado2)
225.          rotas_iteracao.append(rota1_pos_swap)
226.          rotas_iteracao.append(rota2_pos_swap)
227.
228.          #elimina listas com nenhum ponto
229.          for rota in rotas_iteracao:
230.              if len(rota) == 0:
231.                  rotas_iteracao.remove(rota)
232.
233.          # Verifica factibilidade
234.          fact = factivel(rotas_iteracao, mat_dist)
235.          if fact:
236.              vizinhos.append(rotas_iteracao)
237.          elif not fact:
238.              pass
239.
240.      return vizinhos
241.
242.  def gera_vizinhos_swap(solucao_input, qtde_vizinhos, mat_dist):
243.
244.      vizinhos = []
245.
246.      for _ in range(qtde_vizinhos):
247.          #vetor pra usar durante uma iteração
248.          rotas_iteracao = [linha.copy() for linha in solucao_input]
249.
250.          # sorteia duas rotas diferentes
251.          rota_sorteado1 = random.choice(rotas_iteracao)
252.          rota_sorteado2 = random.choice(rotas_iteracao)
253.          while rota_sorteado2 == rota_sorteado1:
254.              rota_sorteado2 = random.choice(rotas_iteracao)
255.
256.          # sorteia uma posiçao em cada rota e salva o valor do elemento neste índice
257.          sorteio_ind_rota1 = random.randint(0, len(rota_sorteado1) - 1)
258.          sorteio_ind_rota2 = random.randint(0, len(rota_sorteado2) - 1)
259.
260.          ponto_sorteio1 = rota_sorteado1[sorteio_ind_rota1]
```

```
261.          ponto_sorteio2 = rota_sorteado2[sorteio_ind_rota2]
262.
263.          #Faz o swap com BEST INSERTION
264.          rota1_pos_swap, rota2_pos_swap = best_insertion(mat_dist, list1=rota_sorteado1,
    list2=rota_sorteado2,
265.                                                          point1=ponto_sorteio1,
    point2=ponto_sorteio2, method='swap')
266.
267.          #remonta o vetor de rotas
268.          rotas_iteracao.remove(rota_sorteado1)
269.          rotas_iteracao.remove(rota_sorteado2)
270.          rotas_iteracao.append(rota1_pos_swap)
271.          rotas_iteracao.append(rota2_pos_swap)
272.
273.          #elimina listas com nenhum ponto
274.          for rota in rotas_iteracao:
275.              if len(rota) == 0:
276.                  rotas_iteracao.remove(rota)
277.
278.          #Verifica factibilidade
279.          fact = factivel(rotas_iteracao, mat_dist)
280.          if fact:
281.              vizinhos.append(rotas_iteracao)
282.          elif not fact:
283.              pass
284.
285.      return vizinhos
286.
287.  def gera_vizinhos_insertion(solucao_input, qtde_vizinhos, mat_dist):
288.
289.      vizinhos = []
290.
291.      for _ in range(qtde_vizinhos):
292.          #vetor pra usar durante uma iteração
293.          rotas_iteracao = [linha.copy() for linha in solucao_input]
294.
295.          # sorteia duas rotas diferentes
296.          rota_sorteado1 = removal_com_peso(solucao=rotas_iteracao, instancia=mat_dist)
297.          options_rota2 = rotas_iteracao.copy()
298.          options_rota2.remove(rota_sorteado1)
299.          rota_sorteado2 = random.choice(options_rota2)
300.
301.          # sorteia uma posiçao em cada rota e salva o valor do elemento neste índice
302.          sorteio_ind_rota1 = random.randint(0, len(rota_sorteado1) - 1)
303.          ponto_sorteio1 = rota_sorteado1[sorteio_ind_rota1]
304.
305.          #Faz o insertion com BEST INSERTION
306.          rota1_apos_ins, rota2_apos_ins = best_insertion(mat_dist, list1=rota_sorteado1,
    list2=rota_sorteado2,
    point1=ponto_sorteio1, method='insertion')
307.
308.          #remonta o vetor de rotas
309.          rotas_iteracao.remove(rota_sorteado1)
310.          rotas_iteracao.remove(rota_sorteado2)
311.          rotas_iteracao.append(rota1_apos_ins)
312.          rotas_iteracao.append(rota2_apos_ins)
313.
314.          #elimina listas com nenhum ponto
315.          for rota in rotas_iteracao:
316.              if len(rota) == 0:
317.                  rotas_iteracao.remove(rota)
318.
319.          #Verifica factibilidade
320.          fact = factivel(rotas_iteracao, mat_dist)
321.          if fact:
322.              vizinhos.append(rotas_iteracao)
323.          elif not fact:
324.              pass
325.
```

```
326.      return vizinhos
327.
328.  def factivel(rotas, mat_dist=[]):
329.      #Retorna True para factivel, False para nao factível
330.      qtde_pontos_real = len(mat_dist)
331.      qtde_pontos_rota = 0
332.      custos = []
333.      for sub_rota in rotas:
334.          custos.append(get_rout_cost(mat_dist, sub_rota))
335.          qtde_pontos_rota += len(sub_rota)
336.
337.
338.      if qtde_pontos_rota != qtde_pontos_real:
339.          return False
340.
341.      for custo in custos:
342.
343.          if custo > tempo_maximo:
344.              return False
345.
346.      return True
347.
348.  def get_melhor_vizinho(vizinhança, instancia):
349.      # retorna o melhor vizinho e o maior custo de uma das suas rotas
350.      custos_maximos = []
351.
352.      #monta um vetor com todos os custos máximos de cada vizinho
353.      for vizinho in vizinhança:
354.          lista_custos = []
355.
356.          for rota in vizinho:
357.              lista_custos.append(get_rout_cost(instancia, rota))
358.
359.          max_cost_vizinho = max(lista_custos)
360.          custos_maximos.append(max_cost_vizinho)
361.      menor_max_cost = min(custos_maximos)
362.      posicao_menor_max_cost = custos_maximos.index(menor_max_cost)
363.      melhor_vizinho = vizinhança[posicao_menor_max_cost]
364.
365.      return melhor_vizinho, menor_max_cost
366.
367.  def best_insertion(instancia, list1 = [], list2 = [], point1 = int(), point2 = int(),
    method=['swap', 'insertion']):
368.      #INSERTION
369.      if method == 'insertion':
370.          listas2_rotas = list()
371.          listas2_custos = list()
372.
373.          list1_retorno = list1.copy()
374.          list1_retorno.remove(point1)
375.          # insere em cada posição, menos no final do vetor
376.          for i in list2:
377.              list2_c = list2.copy()
378.              current_index = list2_c.index(i)
379.              list2_c.insert(current_index, point1)
380.              listas2_rotas.append(list2_c.copy())
381.              listas2_custos.append(get_rout_cost(instancia, list2_c))
382.
383.          # insere o ponto no final do vetor
384.          list2_c = list2.copy()
385.          list2_c.append(point1)
386.          listas2_rotas.append(list2_c.copy())
387.          listas2_custos.append(get_rout_cost(instancia, list2_c))
388.
389.          #pega melhor cenário de inserção
390.          melhor_insertion_cost = min(listas2_custos)
391.          melhor_insertion_cost_posicao = listas2_custos.index(melhor_insertion_cost)
392.
393.          melhor_insercao_rota2 = listas2_rotas[melhor_insertion_cost_posicao]
```

```
394.
395.         return list1_retorno, melhor_insercao_rota2
396.
397.     # SWAP
398.     elif method == 'swap':
399.         #remove os pontos de cada vetor
400.         list1_rotas = []
401.         list1_custos = []
402.         list2_rotas = []
403.         list2_custos = []
404.
405.         list1_apoio = list1.copy()
406.         list1_apoio.remove(point1)
407.
408.         list2_apoio = list2.copy()
409.         list2_apoio.remove(point2)
410.
411.         #insere ponto2 na lista 1 até a penultima posição
412.         for i in list1_apoio:
413.             list1_apoio_c = list1_apoio.copy()
414.             current_index = list1_apoio_c.index(i)
415.             list1_apoio_c.insert(current_index, point2)
416.             list1_rotas.append(list1_apoio_c.copy())
417.             list1_custos.append(get_rout_cost(instancia, list1_apoio_c))
418.         #insere ponto2 na lista1 na última posição
419.         list1_apoio_c = list1_apoio.copy()
420.         list1_apoio_c.append(point2)
421.         list1_rotas.append(list1_apoio_c.copy())
422.         list1_custos.append(get_rout_cost(instancia, list1_apoio_c))
423.
424.         #insere ponto1 na lista 2 até a penultima posição
425.         for i in list2_apoio:
426.             list2_apoio_c = list2_apoio.copy()
427.             current_index = list2_apoio_c.index(i)
428.             list2_apoio_c.insert(current_index, point1)
429.             list2_rotas.append(list2_apoio_c.copy())
430.             list2_custos.append(get_rout_cost(instancia, list2_apoio_c))
431.
432.         #insere ponto1 na lista2 na última posição
433.         list2_apoio_c = list2_apoio.copy()
434.         list2_apoio_c.append(point1)
435.         list2_rotas.append(list2_apoio_c.copy())
436.         list2_custos.append(get_rout_cost(instancia, list2_apoio_c))
437.
438.         #pega melhor cenário de swap
439.         melhor_cost1 = min(list1_custos)
440.         melhor_cost1_posicao = list1_custos.index(melhor_cost1)
441.         melhor_swap_rota1 = list1_rotas[melhor_cost1_posicao]
442.
443.         melhor_cost2 = min(list2_custos)
444.         melhor_cost2_posicao = list2_custos.index(melhor_cost2)
445.         melhor_swap_rota2 = list2_rotas[melhor_cost2_posicao]
446.
447.         return melhor_swap_rota1, melhor_swap_rota2
448.
```

APÊNDICE C – Código referente a abertura de instâncias do tsplib95 e normalização das instâncias em *python*

```
def abre_instancia_tsplib(filepath):
    import tsplib95 as tsp
    import networkx as nx
    from scipy import sparse

    a = tsp.load(filepath)
    G = a.get_graph()
    mat = nx.adjacency_matrix(G)
    my_array = sparse.lil_array(mat).data
    my_list = my_array.tolist()
    mat_dist = my_list

    return mat_dist

def normalizar_instancia(array_normalizar, array_base):

    base = [linha.copy() for linha in array_base]
    new = [linha.copy() for linha in array_normalizar]


    i = 0
    for linha in new:
        linha[i] = m_grande
        i += 1
        # for j in linha:
        #     if j == 0:
        #         linha[linha.index(j)] = m_grande

    base_valor_max = -m_grande
    base_valor_min = m_grande
    new_valor_max = -m_grande
    new_valor_min = m_grande

    # pega valores x min e x max
    for i in base:
        linha_copia = i.copy()
        linha_copia.sort()
        qtde_m_grande = linha_copia.count(m_grande)
        if linha_copia[-(qtde_m_grande + 1)] > base_valor_max:
            base_valor_max = linha_copia[-2]
            # print(f'maior valor: {base_valor_max}')

        if linha_copia[0] < base_valor_min:
            base_valor_min = linha_copia[0]

    # pega valores do novo max e min
    for i in new:
        linha_copia = i.copy()
        linha_copia.sort()
        qtde_m = linha_copia.count(m_grande)
        índice_maior_valor = -(qtde_m + 1)
        if linha_copia[índice_maior_valor] > new_valor_max:
            new_valor_max = linha_copia[índice_maior_valor]


        if linha_copia[0] < new_valor_min:
            new_valor_min = linha_copia[0]

    novo_array_normalizado = list()

    for linha in new:
        linha_normalizada = list()
        for x in linha:
            if x == m_grande:
                linha_normalizada.append(x)

            else:
                taxa = ((base_valor_max - base_valor_min) / (new_valor_max -
new_valor_min))
```

```
                x_scaled = base_valor_min + (x * taxa)
                linha_normalizada.append(x_scaled)

        novo_array_normalizado.append(linha_normalizada.copy())

    return novo_array_normalizado

```