UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Tecnologia

Pedro Henrique Bianchi

Otimização de Rotas Usando o Google OR-Tools

Limeira
2023

Pedro Henrique Bianchi

Otimização de Rotas Usando o Google OR-Tools

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Luis Augusto Angelotti Meira

Este trabalho corresponde à versão final da Monografia defendida por Pedro Henrique Bianchi e orientada pelo Prof. Dr. Luis Augusto Angelotti Meira.

Limeira
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Mariana Xavier - CRB 8/9615

Bianchi, Pedro Henrique, 1992-
B47o Otimização de rotas usando o Google OR-Tools / Pedro Henrique Bianchi. – Limeira, SP : [s.n.], 2023.

Orientador: Luis Augusto Angelotti Meira.
Trabalho de Conclusão de Curso (graduação) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Problema de roteamento de veículos. 2. Otimização. I. Meira, Luis Augusto Angelotti,1979-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações adicionais, complementares

Título em outro idioma: Routes optimization using Google OR-Tools
Palavras-chave em inglês:
Vehicle routing problem
Optimization
Titulação: Bacharel
Banca examinadora:
Luis Augusto Angelotti Meira [Orientador]
Ieda Geriberto Hidalgo
Plínio Roberto Souza Vilela
Data de entrega do trabalho definitivo: 04-12-2023

FOLHA DE APROVAÇÃO

Abaixo se apresentam os membros da comissão julgadora da sessão pública de defesa de dissertação para o Título de Bacharel em Sistemas de Informação na área de concentração , a que se submeteu o aluno Pedro Henrique Bianchi, em 04 de dezembro de 2023 na Faculdade de Tecnologia – FT/UNICAMP, em Limeira/SP.

Prof. Dr. Luis Augusto Angelotti Meira
Presidente da Comissão Julgadora

Profa. Dra. Ieda Geriberto Hidalgo
FT/UNICAMP

Prof. Dr. Plínio Roberto Souza Vilela
FT/UNICAMP

Ata da defesa, assinada pelos membros da Comissão Examinadora, encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria de Graduação da Faculdade de Tecnologia.

Agradecimentos

Primeiramente gostaria de agradecer aos meus pais que sempre se esforçaram para que eu pudesse ter uma educação de qualidade e sempre me instruiram sobre e me levaram a sonhar em cursar uma grande universidade brasileira, como a Unicamp. E apesar de todos os obstáculos no caminho, meus pais sempre me apoiaram e me incentivaram a continuar. Também gostaria de agradecer a minha irmã e meus amigos que sempre estiveram ao meu lado, me incentivando e acreditando que conseguiria concluir esta etapa na minha vida. A todos meus professores, desde antes das universidades, por toda ajuda, pelo conhecimento compartilhado e por todos os conselhos que guiaram o meu aprendizado. Mas em especial ao meu orientador que sempre acrescentou compartilhando conhecimentos sobre o tema, apontando as direções que faziam sentido para o trabalho, me ajudando com as contruções e revisões dos textos deste trabalho e durante todos estes anos nunca perdeu a paciência em me guiar. E acima de tudo, agradecer a Deus, por todas as oportunidades a mim oferecidas e por todas as bençãos a mim concedidas, nunca me faltando saúde para nada na vida.

Resumo

O problema de definir rotas para veículos é enfrentado diariamente por empresas que realizam entregas, coletas, atendimentos em domicílios, transporte de cargas, pessoas, alimentos, entre outros. Também é um problema clássico em computação, sendo estudado por décadas e sendo desafiador até os dias atuais. O problema de roteamento pode ser dividido em dois casos. Se houver apenas um veículo, temos o Problema do Caixeiro Viajante, conhecido pela sigla *TSP*. Se tivermos que atender os clientes com uma frota de veículos, temos o Problema de Roteamento de Veículos, conhecido pela sigla *VRP*. Temos o surgimento de novas ferramentas para resolver tanto o *TSP* quanto o *VRP*, porém é necessário executar experimetos com as tais ferramentas para validá-las. Neste trabalho realizamos um estudo da ferramenta Google *OR-Tools*, uma ferramenta lançada pelo Google para resolver uma série de problemas de otimização. O Google *OR-Tools* resolve problemas como Otimização Linear, Otimização Inteira, Otimização de Restrições, Agendamento, Roteamento entre outros. Nosso foco foi utilizar a ferramenta para problemas de roteamento, tanto para o *TSP* quanto para o *VRP*. Este trabalho utilizou a linguagem Python e trabalhou com dois conjuntos de instâncias, algumas vindas do *TSPLIB95* e outras vindas do PostVRP, que são instâncias feitas para carteiros nas cidades de Artur Nogueira e Rio Claro. Foi necessário fazer a leitura e processamento dos arquivos das instâncias, deixá-los no formato adequado para o Google *OR-Tools* e depois ajustar uma série de parâmetros para poder realizar os experimentos. Por exemplo, existem estratégias de resolução como *GLOBAL_CHEAPEST_ARC*, *PATH_CHEAPEST_ARC* e a *GUIDED_LOCAL_SEARCH*. Executamos as instâncias com cada uma das estratégias e comparamos os resultados. Também comparamos os resultados com o ótimo, quando este era conhecido. O resultado deste TCC consiste em analisar o esforço necessário para se executar instâncias utilizando o Google *OR-Tools* e também analisar a qualidade da solução *versus* o tamanho da solução *versus* o tempo de execução dos algoritmos de otimização. Além de analisar o esforço necessário para se utilizar a ferramenta Google *OR-Tools*, pudemos atestar a qualidade dos resultados em instâncias com ótimo conhecido e em instâncias reais de roteamento.

Abstract

The issue of defining vehicles routes is faced daily by companies that works with deliveries, collection, home care, cargo transport, passenger transport, food transport, among others. It is also a traditional computing issue, which has been studied for decades and remains challenging until nowadays. The routing issues could be divided in two cases. If only one vehicle is available, it defines the Travel Salesman Problem, also know by the acronym TSP. If the customers will be attended by a vehicle fleet, it defines another routing issue know by the acronym VRP. New tools has being emerged to solve both TSP and VRP issues, but it is necessary to execute experiments with those tools to validate them. On this work we carried out a study of the Google OR-Tools, a tool released by Google to solve a lot of optimization issues. The Google OR-Tools solve issues as Linear Optimization, Integer Optimization, Constraint Optimization, Scheduling, Routing, among others. Our focus was to validate the tool on routing issues, both for TSP and VRP. This work used Python language and worked with two sets of instances, some came from TSPLIB95 and others came from PostVRP, that are instances made to postmen in the cities of Artur Nogueira and Rio Claro. It was necessary make the parse from those instances files, them process it, transform the data on the proper format to Google OR-Tools and them adjust a lot of parameters to be able to realize all the experiments. As example there is solve strategies as GLOBAL_CHEAPEST_ARC, PATH_CHEAPEST_ARC and GUIDED_LOCAL_SEARCH. We executed those instances with each of the strategies and we compared the results. We also compared the results with the optimum solution, when it was know. The result of this final paper consists on analyze the necessary effort to execute those instances using Google OR-Tools and also analyze how good was the solution against the solution size and against the cost from the execution time from the optimization algorithms. In addition to analyzing the effort required to use the Google OR-Tools, we could certify the results quality on instances with a know optimum tour and also on real routing issue instances.

Sumário

1 Introdução **9**

2 Implementação **12**
2.1 Passo 1. Familiarização com a Ferramenta 13
2.2 Passo 2. Preparação dos dados 14
2.3 Passo 3. Google OR-Tools em instâncias da TSPLIB 18
2.4 Passo 4. Desenvolvimento de logs e imagens 23
2.5 Passo 5. Google OR-Tools em instâncias de cidades do interior paulista 28
2.6 Passo 6. Google OR-Tools em instâncias de VRP 30
2.7 Passo 7. Desenvolvimento de logs e imagens para instâncias de VRP 32

3 Experimentos e Resultados **34**
3.1 Resultados *TSPLIB95* 35
3.1.1 Instância a280 35
3.1.2 Instância pr2392 39
3.2 Resultados das instâncias das cidades de Rio Claro e Artur Nogueira como TSP 41
3.2.1 Instância *TSP* Rio Claro 20 42
3.2.2 Instância *TSP* Rio Claro 200 43
3.2.3 Instância *TSP* Rio Claro 2000 45
3.2.4 Instância *TSP* Artur Nogueira 20 47
3.2.5 Instância *TSP* Artur Nogueira 200 48
3.2.6 Instância *TSP* Artur Nogueira 2000 50
3.3 Resultados das instâncias da cidade de Rio Claro como VRP 52
3.3.1 Instância *VRP* Rio Claro 20 52
3.3.2 Instância *VRP* Rio Claro 200 54
3.3.3 Instância *VRP* Rio Claro 2000 55

4 Conclusões **58**

Referências bibliográficas **60**

Capítulo 1

Introdução

O setor de transporte é um setor crítico na sociedade atual por diversas razões, como a globalização, a poluição atmosférica (COLVILE et al., 2001) e as mudanças climáticas (WORLD ECONOMIC FORUM, 2022). Primeiramente, o crescimento da globalização faz com que seja cada vez mais necessária a coordenação de meios de transporte de pessoas e cargas visando uma integração ampla nos mais diversos níveis (i.e. global, nacional, regional e interurbano) (O'CONNOR, 2009). Em segundo lugar, todos os meios de transporte motorizado emitem poluentes na atmosfera que têm sido relacionados a doenças como ataques cardíacos, derrames, câncer de pulmão e infecções respiratórias em crianças. Terceiro, há muitas discussões sobre como o setor de transporte pode contribuir para a ação climática, dado que este emite globalmente 20% dos gases de efeito estufa, principais responsáveis pelas mudanças climáticas (WORLD ECONOMIC FORUM, 2022).

Dentro desse contexto, a otimização de rotas emerge como uma proposta de solução para esses problemas. Com relação à globalização, essa técnica promove maior eficiência no setor de transporte, ao indicar os melhores trajetos de acordo com quaisquer parâmetros desejados como tempo ou urgência. A diminuição do tempo de trajetos contribui significativamente para a redução de emissão de poluentes na atmosfera (DEWI; UTAMA, 2021). Além disso, essa técnica tem o potencial de contribuir para a redução do uso de combustíveis fósseis vinculados ao setor e principal emissor de gases de efeito estufa (DEWI; UTAMA, 2021).

A busca pela melhor rota é um tópico antigo, apesar de ser mais recente o uso de subsídios matemáticos na otimização. Historicamente, o mais famoso problema de otimização de rota é o problema do caixeiro viajante (*Travelling Salesman Problem* - TSP) apresentada pela primeira vez por Karl Menger in 1930 (ENAMI et al., 2017). A resolução do TSP tem como objetivo

encontrar o melhor trajeto que o caixeiro deveria percorrer. Em termos matemáticos isso significa encontrar a menor sequência, sob um conjunto finito de pontos, sabendo a distância entre eles. O TSP é classificado como NP-Difícil. Garey e Johnson definem como um problema improvável de se resolver usando algoritmos polinomiais (ENAMI et al., 2017).

In 1959, Dantzig e Ramser foram além do TSP e definiram pela primeira vez o problema de roteamento de veículos (*Vehicle Routing Problem* - VRP), que nada mais é do que uma generalização do TSP considerando múltiplos veículos (NÉIA; ARTERO; CUNHA, 2017). Em termos simples, o VRP tinha como objetivo entregar gasolina para diversos postos (NÉIA; ARTERO; CUNHA, 2017). Desde então diferentes tipos de abordagens têm sido aplicadas a esse problema como heurística, metaheurística e algorítmos genéticos.

Variantes desse problema também têm sido constantemente propostas, como a *Post Office Deliveries VRP* (PostVRP), que lida com a distribuição de correios por carteiros, modelados como veículos (MEIRA et al., 2019). A Figura 1.1 exemplifica a resolução de um VRP com objetivo de diminuição de custos. Nesse problema o veículo precisa passar por quatro pontos usando a menor rota possível.

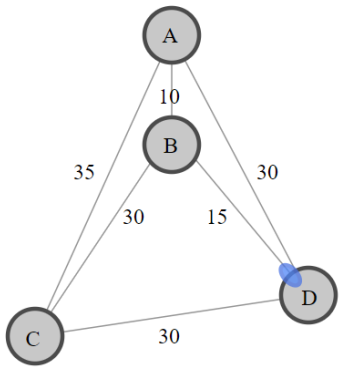

Figura 1.1: TSP com 4 pontos. Fonte: (GOOGLE, 2023)

Mais recentemente diversas ferramentas computacionais têm sido produzidas para auxiliar no desafio do roteamento de veículos. Dentre as opções, esse trabalho se concentra no Google OR-Tool pelas seguintes razões:

- A ferramenta é gratuita e *open-source*, o que auxilia a ser amplamente utilizada;
- É uma ferramenta premiada (ouro na *International Constraint Programming Competition* desde 2013);
- Usa algoritmos que são estado-da-arte;

- É pouco explorada em trabalhos acadêmicos (37 documentos encontrados no Scopus)[1].

O Google OR-Tool é um software de otimização combinatória (GOOGLE, 2023). Seu principal uso é achar uma solução otimizada de um problema. Seu uso é para roteamento de veículos mas também é capaz de resolver outros problemas como *Linear Optmization*, *Integer Optmization*, *Constraint Optimization*, *Assignment Problem*, *Packing Problem*, *Scheduling Problem* and *Network flow problem* (GOOGLE, 2023).

Dada a importância da busca de soluções dentro do setor de transporte, esse trabalho de conclusão de curso em Sistemas da Informação tem como objetivo utilizar o Google OR-Tools em problemas de roteamento de veículos.

O restante desse trabalho de conclusão de curso se organiza em outros 3 capítulos. Capítulo 2 apresenta toda a implementação para nos familiarizarmos com o Google OR-Tool e depois utilizar a ferramenta em nossos experimentos. Capítulo 3 apresenta os experimentos e resultados que geramos utilizando o Google OR-Tools. Capítulo 4 conclui esse trabalho e aponta novos caminhos que podem ser seguidos em futuras aplicações.

[1]Dia 17 de setembro de 2022

Capítulo 2

Implementação

Neste capítulo será abordado o desenvolvimento realizado para utilizar o *OR-Tools* em problemas de roteamento de veículos.

A implementação buscou: *(i)* modelar os dados de entrada; *(ii)* passar ao *OR-Tools* todas as informações necessárias; *(iii)* escolher uma ou mais estratégias de otimização para o *OR-Tools*; *(iv)* e ao final obter logs das execuções e imagens da solução encontrada pelo *OR-Tools*.

A Figura 2.1 descreve passo a passo o desenvolvimento deste trabalho. Cada etapa da figura será abordada em um sub-tópico deste capítulo. Toda a implementação realizada para este trabalho de conclusão de curso, incluindo as imagens e logs, estão versionados publicamente no github[1].

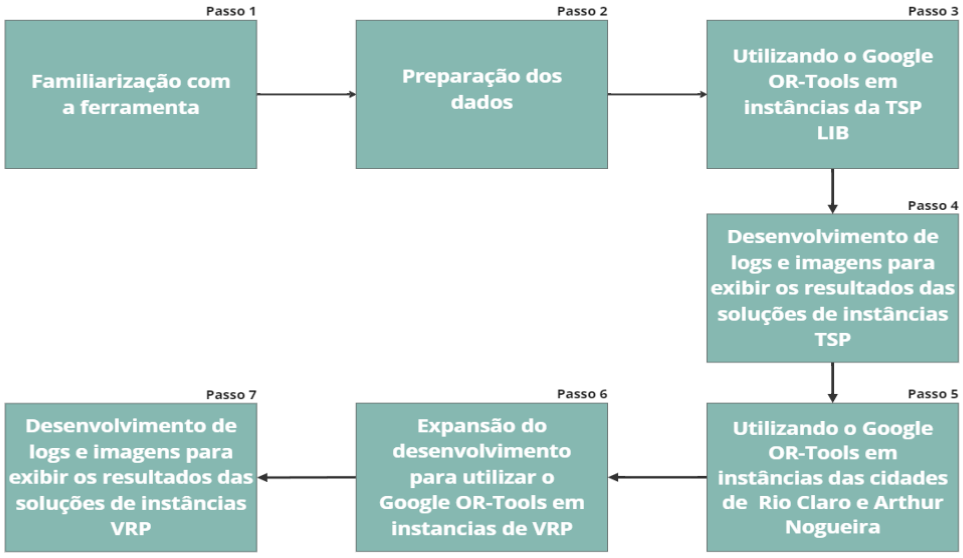

Figura 2.1: Passos da Implementação

[1] https://github.com/PedroHenriqueBianchi/Google-OR-Tools-Routing-Issues

2.1 Passo 1. Familiarização com a Ferramenta

Os primeiros passos com o *OR-Tools* foram para compreensão da ferramenta, ou seja, como utilizar o *OR-Tools*, quais as diferentes estratégias que a ferramenta provê para solucionar o problema proposto e quais informações o *OR-Tools* precisa para atacar o problema e resultar em uma solução.

Para compreensão da ferramenta, seguimos os exemplos de problemas de roteamento que o Google disponibiliza em sua página oficial do *OR-Tools*[2]. Seguindo os exemplos na linguagem Python, construímos as soluções para algumas situações problemas que foram propostas. Estes exemplos, que solucionamos de forma guiada, abordam diferentes variantes de problemas de roteamento de veículos. Dentre estas diferentes variantes, seguimos os tutoriais para solucionar situações do tipo: *TSP*, *VRP*, *CVRP* (VRP com capacidade do veículo), *Pickup-Delivery VRP* (VRP com pontos de coleta e entrega) e *VRPTW* onde cada ponto de entrega tem uma janela de horário para que a entrega seja realizada.

Desta forma ficou claro algumas coisas básicas do uso da ferramenta: *(i)* é necessário passar uma matriz de distâncias entre os pontos a serem visitados; *(ii)* definir qual estratégia será utilizada para solucionar o problema; *(iii)* e passar a referência de uma função distância, que, dados dois pares ordenados, retorna a distância entre estes dois pontos. Tal distância pode ser obtida diretamente da matriz ou indiretamente, caso seja usada alguma métrica específica.

Neste ponto também procuramos entender o que são estas estratégias utilizadas para solucionar o problema e esta página[3], explica brevemente um pouco sobre cada estratégia disponível no *OR-Tools*. Segue a explicação para as 3 estratégias utilizadas na implementação deste trabalho.

Primeiro é necessário citar que as estratégias se dividem em dois tipos: *First Solution Strategy* e *Local Search Options*, que também é chamada de *metaheuristics*.

As estratégias do tipo *First Solution Strategy* têm o objetivo de encontrar uma solução inicial para o problema (GOOGLE, 2023). Estas estratégias seguem um passo a passo que sempre encontra a mesma solução, entregam a solução em um baixo tempo de execução e não garantem que a solução encontrada seja ótima.

Já as estratégias do tipo *Local Search Options* utilizam metaheurística para encontrar uma solução. Nestas estratégias, diferente das *First Solution Strategy*, é necessário informar um

[2]Segue o endereço desta página, `https://developers.google.com/optimization/routing`
[3]`https://developers.google.com/optimization/routing/routing_options`

tempo limite para permancer executando a estratégia e ao final do tempo limite é retornada a melhor solução encontrada até o momento, as soluções encontradas podem diferir conforme o tempo limite difere. E as soluções encontradas, podem melhorar, com o aumento do tempo limite, podendo chegar na solução ótima, se dado tempo o suficiente.

Foram utilizadas duas estratégias de *First Solution Strategy* e uma de *Local Search Options*. As estratégias de *First Solution Strategy* são as *GLOBAL_CHEAPEST_ARC* e *LOCAL_CHEAPEST_ARC*. E a estratégia de *Local Search Options* foi a *GUIDED_LOCAL_SEARCH*.

GLOBAL_CHEAPEST_ARC é uma estratégia que iterativamente conecta dois nós que produzem o menor segmento de rota (GOOGLE, 2023).

LOCAL_CHEAPEST_ARC é uma estratégia que seleciona o primeiro nó que tenha um sucessor desconectado e o conecta ao nó que produzir o menor segmento de rota (GOOGLE, 2023).

GUIDED_LOCAL_SEARCH é uma estratégia que utiliza a metaheurística chamada de *Guided Local Search* para escapar dos mínimos locais. Geralmente esta é a metaheurística mais eficiente para roteamento de veículos (GOOGLE, 2023).

2.2 Passo 2. Preparação dos dados

Depois dos primeiros passos com o *OR-Tools*, seguindo os exemplos guiados, como comentado anteriormente, há alguns itens necessários para utilizar a ferramenta em problemas de rotemanto. O principal é a matriz de distâncias. Pois, tendo a matriz de distâncias, temos todas as informações necessárias para construir um método que nos retorne a distância entre os vértices, que é o que de fato é passado ao *OR-Tools* para resolver o problema. Então, a fim de facilitar o uso de matrizes, foi construída uma classe em Python, chamada de *Matrix*, onde se concentraram algumas informações e funcionalidades relacionadas às matrizes.

Para facilitar o entendimento das funcionalidades da classe *Matrix*, ou seja seus métodos, primeiro é necessário entender quais são as informações, ou seja seus atributos, que são passados no construtor da classe *Matrix*. Os atributos da classe *Matrix* são: *(i)* número de linhas (*rows_size*); *(ii)* número de colunas (*columns_size*); *(iii)* tipo da matriz (*matrix_type*); *(iv)* tipo das coordenadas cartesianas (*coordinates_type*); e *(v)* tipo do cálculo para obter a distância entre os pontos (*calc_dist_type*). E há mais dois atributos que não são passados no construtor

da classe, porém são instanciandos ao se chamar o método construtor da classe, estes atributos são: *(i)* a matriz de pontos (*points_matrix*); e *(ii)* a matriz (*matrix*).

Dos atributos passados no construtor da classe, alguns podem ser identificados pelo próprio nome. Por exemplo, o *número de linhas*, que é o número de linhas da matriz de distâncias. Outro bem explicativo é o *número de colunas*, que é o número de colunas da matriz de distâncias. Já os demais merecem uma explicação mais detalhada.

O *tipo da matriz*: é esperado que seja passado uma *string* não sensível a maiúsculas e minúsculas com três possíveis valores distintos, que são: *(i) TSPLIB*; *(ii) REAL_WORLD*; e *(iii) RIO_CLARO*.

Das instâncias TSPLIB, selecionamos o problema TSP e distância Euclidiana. Ou seja, os pontos são coordenadas cartesianas e a distância vem da métrica Euclidiana. As instâncias tipo *REAL_WORLD* e *RIO_CLARO* já possuem a matriz de distâncias explicitas no arquivo.

A partir destes três valores (*TSPLIB*, *REAL_WORLD* e *RIO_CLARO*) sabemos como construir a matriz de distâncias. Nas matrizes do tipo *TSPLIB* temos as coordenadas de cada ponto a ser visitado. Para cada par de coordenadas, utilizamos a métrica Euclidiana para descobrir as distâncias. *TSPLIB95* é um conjunto de instâncias de várias fontes e vários tipos para problemas de *Travelling Salesman Problem* e problemas relacionados a *TSP* (HEIDELBERG, 2023).

Já nas matrizes dos tipos *REAL_WORLD* e *RIO_CLARO* lemos um arquivo gerado por um software, construido no trabalho (MEIRA et al., 2019). Nestas instâncias de cidades, além das coordenadas de cada ponto, também temos a distância entre cada par de pontos já calculada de uma forma que se aproxime da distância utilizando um mapa de ruas. A geração destes arquivos será abordada, com mais detalhes, na Seção 2.5 *Google OR-Tools em instâncias de cidades do interior paulista*.

A classe *Matrix* tem também o atributo *tipo da coordenada cartesiana*, onde é esperado que seja passado uma *string*, não sensível a maiúsculas e minúsculas, com dois possíveis valores distintos, que são: *(i) INT*; e *(ii) SCIENTIFIC_NOTATION*. A ideia aqui é facilitar a expansão para novos tipos de coordenadas cartesianas. No caso *INT* é utilizado o valor diretamente. No caso *SCIENTIFIC_NOTATION* temos uma trabalho maior a fazer. Primeiro é necessário dividir a string que representa a coordenada, para que seja possível identificar o número base da notação científica e o seu expoente. Tendo estas duas strings, a base é convertida para ponto flutuante e o expoente convertido para inteiro, fazemos o cálculo da notação científica, fazendo dez elevado ao expoente e multiplicando este resultado pelo número base da notação.

E por fim temos o *tipo do cálculo da distância* onde também é esperado que seja passada uma *string*, não sensível a maiúsculas e minúsculas com um único possível valor, que é o *(i) EUCLIDEAN*. Vale ressaltar que a ideia aqui é facilitar a expansão para novos tipos de cálculo da distância, mas, nas instâncias que fizemos experimentos, este foi o único cálculo necessário. Acredito que neste momento esteja claro qual será o uso deste atributo, basicamente é saber como realizar o cálculo da distância dado dois pontos, ou seja, duas coordenadas. E no caso o cálculo do tipo *EUCLIDEAN* é feito o cálculo da distância euclidiana bidimensional entre estes dois pontos.

Dados todos os atributos, fica mais fácil de entender os métodos desta classe *Matrix*. Há 12 diferentes métodos nesta classe, mas vamos passar pelos principais, que são os:

- *build_matrix*
- *make_points_matrix_from_a_file*
- *construct_distance_matrix_from_points_matrix*
- *load_points_and_distance_matrices_from_a_file*
- *split_coordinates_improved*

O método *build_matrix* é o primeiro a ser chamado depois de instanciar a classe. Este método é responsável por popular a lista de pontos, com cada coordenada a ser visitada e a matriz em si, que no caso é a matriz de distâncias, com a distância entre cada par de pontos. Este método tem em sua assinatura um atributo chamado *file_path*, que é o caminho para o arquivo que precisamos ler para popular a matriz de distâncias e a lista de pontos. A partir do tipo da matriz, que foi passado no construtor, este método decide qual caminho tomar para a construção destas matrizes.

Se o tipo da matriz for *TSPLIB*, vamos ler um arquivo de alguma instância da TSPLIB95, obtendo os pontos a serem visitados. E vamos calcular cada distância para popular a matriz de distâncias. Já se o tipo da matriz for *REAL_WORLD* ou *RIO_CLARO* vamos somente ler o arquivo, que já nos fornece os pontos a serem visitados e cada distância entre estes pontos. Então, se for do tipo *TSPLIB*, dois outros métodos serão chamados, que é o *make_points_matrix_from_a_file* e o *construct_distance_matrix_from_points_matrix*. Já se o tipo for *REAL_WORLD* ou *RIO_CLARO* somente um outro método será chamado que é o *load_points_and_distance_matrices_from_a_file*.

O método *make_points_matrix_from_a_file* é responsável por ler os arquivos de instâncias da TSPLIB95 e popular a lista de pontos. Este método tem em sua assinatura dois atributos, o *file_path* e o *coord_type*. O *file_path* é o caminho para um arquivo de instância da TSPLIB95. E o *coord_type* é o tipo da coordenada que vamos ler neste arquivo, que pode ser *INT* ou *SCIENTIFIC_NOTATION*. Então o método *make_points_matrix_from_a_file* abre o arquivo de uma instância da TSPLIB95, passa linha a linha deste arquivo para obter as coordenadas, e para isso ele chama o método *split_coordinates_improved* que recebe a linha do arquivo e o tipo da coordenada e nos devolve uma lista com par ordenado destas coordenadas. Então, com a coordenada em mãos, adicionamos esta coordenada na lista de pontos, compondo uma nova linha da matriz de pontos para cada nova coordenada. Ao final deste método temos a lista de pontos populada.

O método *construct_distance_matrix_from_points_matrix* é responsável por popular a matriz de distâncias, para a matriz do tipo *TSPLIB95*. Para isso ele precisa calcular a distância entre cada par de pontos. Então utilizamos a lista de pontos, de forma que iteramos sobre a lista de pontos fixando a primeira coordenada e percorrendo as coordenadas subsequentes, assim calculando todas as distâncias para a primeira coordenada. Desta forma, já populamos a matriz de distâncias em todas as posições da primeira coluna e da primeira linha. Na próxima iteração fixamos a segunda coordenada e percorremos somente as coordenadas subsequentes, ou seja, da terceira coordenada em diante. Logo, vamos ter todas as distâncias faltantes para a segunda coordenada. E assim já populamos todas as posições faltantes da segunda coluna e da segunda linha. Iteramos este processo N vezes, sendo N a quantidade de pontos a serem visitados. E sabemos como realizar o cálculo da distância a partir do atributo que contém o tipo do cálculo de distância.

O método *load_points_and_distance_matrices_from_a_file* é responsável por ler os arquivos das instâncias de cidades do interior paulista. Estes arquivos, além das coordenadas de cada ponto a ser visitado, já nos fornece as distâncias entre cada par de pontos. Este método vai passar linha a linha do arquivo, fazendo algumas verificações para identificar em qual parte do arquivo estamos, obtendo as informações necessárias para popular a matriz de distâncias e a lista de pontos. Quando este método está lendo as distâncias, ele converte a string da distância para inteiro e popula a matriz nas posições correspondentes. Já quando está lendo as coordenadas ele chama o método *split_coordinates_improved* passando a linha com as

coordenadas e o tipo da coordenada. E o método *split_coordinates_improved* nos devolve uma lista com par ordenado destas coordenadas, para popularmos a lista de pontos.

E por fim temos o método *split_coordinates_improved* que recebe a string contendo o par ordenado da coordenada de algum ponto a ser visitado e devolve uma lista de inteiros contendo, na primeira posição, o valor do eixo X desta coordenada. E na segunda posição, o valor do eixo Y desta coordenada. Além da string contendo a coordenada, este método também recebe o tipo da coordenada. Depois de obter a string que representa a coordenda, é preciso saber como converter esta string em uma lista de inteiros. Para isso o método percorre char a char da string recebida, fazendo algumas verificações para que possa obter duas strings que representem o valor do eixo X e o valor do eixo Y, desta coordenada. E então, chama outro método, que faz a conversão destas strings, que representam os eixos, para inteiro.

2.3 Passo 3. Google OR-Tools em instâncias da TSPLIB

Já com a classe *Matrix* implementada, foi pensada uma forma de abstrair o uso do *OR-Tools* para resolver problemas do tipo *TSP*. Abstrair no sentido de facilitar o uso do *OR-Tools* e centralizar todos os dados e todas as funcionalidades necessárias para solucionar o problema proposto. Para isso foi construída uma classe chamada *OrToolsTSPSolver*. A Figura 2.2 mostra o uso desta classe para encontrar soluções da instância *pr2392* da *TSPLIB95*. Na Figura 2.2 podemos ver o construtor da classe sendo utilizado e depois de fato a classe utilizando do *OR-Tools* para executar as estratégias que foram passadas no construtor e encontrar as soluções para cada estratégia executada.

Desta forma fica mais clara todas as informações necessárias para conseguir utilizar o *OR-Tools*. E também fica mais objetivo seu uso, pois, basicamente, precisamos invocar o construtor com todas as informações necessárias e depois invocar o método *execute_strategies* para que obtenhamos as soluções.

Mas, por trás de tudo isso, foram feitas algumas implementações na classe *OrToolsTSPSolver* para que essa facilidade fosse possível. O construtor utiliza de outros atributos e métodos privados[4], para deixar a classe pronta para executar as estratégias definidas e prover as soluções.

[4]Não existe, de fato, métodos e atributos privados em Python. Mas existe uma convenção de se utilizar __ no início do atributo ou método que deveria ser privado, assim indicando que seu uso deve ser restrito a classe pai.

```python
from com.utils.orToolsTSPSolver import OrToolsTSPSolver

if __name__ == '__main__':
    solver = OrToolsTSPSolver(
        problem_name='pr2392',
        matrix_rows_size=2392,
        matrix_columns_size=2392,
        matrix_type='TSPLIB',
        coordinates_type='scientific_notation',
        calc_dist_type='EUCLIDEAN',
        strategies=['GLOBAL_CHEAPEST_ARC', 'PATH_CHEAPEST_ARC', 'GUIDED_LOCAL_SEARCH'],
        marker_size_on_image_solution=0.5,
        line_width_on_image_solution=0.3,
        time_limit_on_seconds_to_metaheuristics=259200
    )

    solver.execute_strategies()
```

Figura 2.2: Construtor da classe OrToolsTSPSolver

Antes de chegarmos aos principais atributos e métodos privados, vamos passar pelos atributos públicos do construtor da classe *OrToolsTSPSolver*. Temos onze atributos públicos pertencentes a classe *OrToolsTSPSolver* que são passados no construtor da classe, destes onze atributos, três serão abordados na próxima seção em que vamos falar sobre os logs e as imagens que geramos a partir da soluções encontrada pelo *OR-Tools*. E cinco destes atributos já foram citados e explicados no seção anterior, em que explicamos a classe *Matrix*, pois internamente a classe *OrToolsTSPSolver* utiliza da classe *Matrix*. Ou seja, a classe *OrToolsTSPSolver* é composta pela classe *Matrix*.

Dito isso, para não ficar repetitivo, vamos passar pelos três atributos exclusivos da classe *OrToolsTSPSolver* e que não foram explicados anteriormente. Estes três atributos são os: *(i)* nome do problema (*problem_name*); *(ii)* estratégias a serem executadas para solucionar o problema (*strategies*); e *(iii)* tempo limite que o OR-Tools parmenecerá executando as estratégias que utilizam de metaheurística (*time_limit_on_seconds_to_metaheuristics*).

Destes atributos temos o nome do problema, que é esperado que seja passado uma *string*, sensível a maiúsculas e minúsculas e esta string pode ter qualque valor possível. Este atributo será utilizado para construir a string do caminho do arquivo referente a instância que estamos abordando, no caso, do arquivo que precisamos ler para obter as informações do problema. Observando a Figura 2.3 fica mais claro como este nome será usado para construir este caminho

do arquivo que precisamos ler. Pois, como podemos observar, na árvore do projeto, há uma pasta chamada *pr2392* e dentro desta pasta há um arquivo chamado *pr2392* e este é o arquivo que a *TSPLIB95* nos provê, ou seja é o arquivo que precisamos ler para construir a matriz dos pontos a serem visitados e a matriz de distâncias. Logo *pr2392* é o nome de uma das instâncias que solucionamos, então passaríamos a string "pr2392" no atributo nome do problema, ao instânciar um objeto da classe *OrToolsTSPSolver* para resolver a instância *pr2392* da *TSPLIB95*.

Inclusive, vale comentar neste momento, que esta estrutura de organização do projeto se repete para os demais tipos de instâncias que vamos resolver. Ou seja, dentro do diretório do projeto sempre vamos ter uma pasta com o nome do tipo da instância que estamos resolvendo, por exemplo *tsplib*, é um tipo de instância, das instâncias da *TSPLIB95*. Logo temos uma pasta chamada *tsplib*. E dentro desta pasta *tsplib* haverá outra pasta com o nome de uma instância deste tipo, por exemplo *pr2392*, que é o nome de uma das instâncias do tipo *tsplib*. E dentro desta pasta chamada *pr2392* se encontram os arquivos, logs e imagens referentes a esta instância.

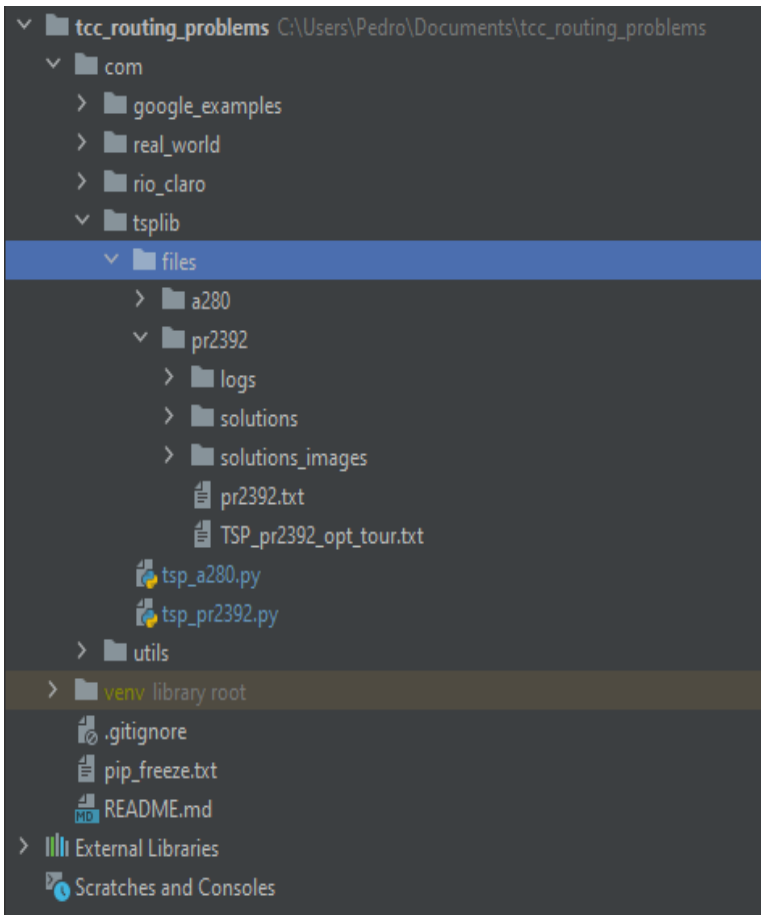

Figura 2.3: Árvore da estrutura do projeto

Outro atributo relevante é o que contém as estratégias que serão executadas pelo *OR-Tools*. Neste atributo é esperado que seja passada uma lista de strings, podendo passar uma lista vazia. Caso seja passada uma lista vazia somente a estratégia *GLOBAL_CHEAPEST_ARC* será executada. As strings, nesta lista, farão com que novas estratégias sejam executadas. Estas strings são: *(i) GLOBAL_CHEAPEST_ARC*; *(ii) PATH_CHEAPEST_ARC*; e *(iii) GUIDED_LOCAL_SEARCH*.

Já o atributo do tempo limite indica quanto tempo que o OR-Tools permenecerá executando as estratégias que utilizam de metaheurística. Neste atributo é esperado que seja passado um inteiro e este inteiro vai representar o tempo de execução em segundos.

Vale comentar também alguns atributos privados desta classe. Há seis atributos privados nesta classe. Destes, dois serão elencados quando falarmos sobre os logs e imagens. Então temos quatro atributos privados que são importantes para entender alguns métodos desta classe, estes atributos são os: *(i)* diretório comun *(__common_directory)*; *(ii)* modelo de dados *(__model_data)*; *(iii)* gerenciador *(__manager)*; e *(iv)* roteador *(__routing)*.

O atributo privado diretório comun *(__common_directory)* é um atributo onde no contrutor da classe *OrToolsTSPSolver* utilizamos o pacote "*os*" do Python, que nos provê recursos do sistema operacional. Através de métodos do pacote "*os*" é obtido uma string do diretório raiz do código fonte deste projeto, salvamos esta string do diretório raiz no atributo diretório comun *(__common_directory)*. Sabendo o diretório raiz e como organizamos os pacotes e arquivos dentro do projeto, conseguimos construir a string do caminho dos arquivos que precisamos ler durante as execuções das instâncias que resolvemos.

O atributo privado modelo de dados *(__model_data)* é um dicionário. Em Python um dicionário seria algo similar a um mapa onde os itens deste dicionário são sempre compostos pelo par chave e valor. Este dicionário contém dados da classe *OrToolsTSPSolver* que são gerados a partir dos atributos passados no construtor. Temos alguns dados mais simples que ficam neste dicionário, como o número de veículos que é fixado como um para a classe *OrToolsTSPSolver*. E o mais importante no modelo de dados são as referências para as matrizes de distância e matriz de pontos, que são geradas dinamicamente no construtor da classe *OrToolsTSPSolver*.

Já os atributos gerenciador *(__manager)* e roteador *(__routing)*, serão utilizados para guardar referências a objetos que sempre são instanciados pelo próprio *OR-Tools*. De forma simplória podemos dizer que o *__manager* é um gerente do *OR-Tools* que prove informações

como qual o ponto de partida, quantos pontos precisamos percorrer e quantos veículos estamos utilizando para solucionar o problema. E o *__routing* seria o responsável do *OR-Tools* por dizer qual a distância entre dois pontos, tanto que quando este objeto é instanciado passamos a referência do método capaz de responder a distância entre dois pontos.

Depois de passar pelos atributos da classe *OrToolsTSPSolver*, fica mais objetivo comentar sobre os métodos, privados e públicos, que foram implementados nesta classe. Temos quatorze métodos implementados, mas seis foram exclusivamente implementados para gerar os logs e imagens das soluções encontradas e serão comentados no sub capítulo *2.4 Passo 4. Desenvolvimento de logs e imagens.* Então estes oito métodos, que não são exclusivos de log e imagens, são os:

- *__init__*
- *__create_data_model*
- *__distance_callback*
- *__prepare_solution*
- *execute_strategies*
- *global_chepeast_arc*
- *path_chepeast_arc*
- *guided_local_search*

O método *__init__* é o construtor da classe. Ele recebe todos os atributos públicos que comentamos anteriormente e os atributos públicos da classe *Matrix* em sua assinatura. Este método construtor já deixa a classe pronta para executar as estratégias passadas no construtor. Um ponto importante é que ao final do construtor, utilizamos do *OR-Tools* para instanciar o atributo *(__manager).* E ele invoca o método *__create_data_model* que será abordado a seguir.

O método *__create_data_model* é responsável por instanciar a matriz de distâncias e a matriz de pontos. Ele não recebe nada em seu construtor, pois utiliza dos atributos da classe *OrToolsTSPSolver* e dos métodos classe *Matrix* para realizar as ações necessárias. E depois de instanciar as matrizes, o método cria o dicionário do atributo *(__model_data)*, já com todas as informações necessárias.

O método *__distance_callback* é capaz de acessar a matriz de distâncias e retornar a distância entre dois pontos. Este método recebe em sua assinatura dois inteiros, um chamado de *from_index*, que é o ponto de partida, e outro chamado de *to_index*, que é o ponto de chegada. E a partir destes dois pontos o método retorna distância entre eles. Uma referência deste método será passada ao *OR-Tools* para que ele utilize este método para saber a distância entre dois pontos. Como comentamos anteriormente, caso queira, ou necessite, é possível alterar este método para fazer mais do que somente consultar a matriz de distância.

O método *__prepare_solution* contém um trecho de código que sempre se repetia e por isso foi extraído num método a parte. Neste método utilizamos o *OR-Tools* para instanciar o atributo *(__routing)* passando o *(__manager)* que foi criado no construtor e passando a referência do método *__distance_callback*.

O método *execute_strategies* é bem simples, ele basicamente verifica quais estratégias foram passadas no vetor *strategies* e chama outros métodos que vão de fato executar aquelas estratégias. Por isso, depois de instanciar a classe *OrToolsTSPSolver*, sempre será chamado o método *execute_strategies*, para que de fato as estratégias passadas no construtor sejam executadas.

E por fim temos os três métodos que são chamados em *execute_strategies*, que são os métodos que de fato utilizam do *OR-Tools* para executar tal estratégia e encontrar uma solução. E utilizando de outros métodos que serão abordados no próximo sub capítulo, geram logs contendo a solução encontrada e também geram uma imagem do trajeto desta solução. Estes métodos levam o nome da estratégia que será executada. Então este métodos são os: *(i) global_chepeast_arc*; *(ii) path_chepeast_arc*; e *(iii) guided_local_search*.

2.4 Passo 4. Desenvolvimento de logs e imagens

Depois que a classe *Matrix* e a classe *OrToolsTSPSolver* foram implementadas, percebeu-se a necessidade complementar a solução encontrada com alguns metadados, como por exemplo, quanto tempo a execução demorou para que a solução fosse encontrada. Além de somente complementar a solução com mais informações, também seria interessante salvar em arquivos estas soluções encontradas e estes metadados sobre a solução. E para complementar tudo isso, surgiu a necessidade de visualizar esta solução, ou seja, tornar gráfica a rota da solução encontrada.

Para atender estas necessidades, alguns métodos foram implementados na classe *OrToolsTSPSolver* para que, durante a execução de alguma estratégia, seja feito um log desta solução com os metadados necessários e com a rota da solução encontrada. E depois de executar, já com a solução encontrada e os logs gerados, um outro método é chamado passando a solução. Assim uma imagem da rota da solução é gerada.

Vamos passar pelos atributos que foram criados na classe *OrToolsTSPSolver* para que fosse possível gerar estes logs e estas imagens. Temos cinco atributos que foram adicionados a classe durante este desenvolvimento, são eles os: *(i) dpi_on_image_solution*; *(ii) marker_size_on_image_solution*; *(iii) line_width_on_image_solution*; *(iv) __logger*; e *(v) __file_name*.

O atributo *dpi_on_image_solution* espera receber um inteiro. Tem por padrão o valor mil duzentos e indica qual será a dpi da imagem do gráfico da rota da solução.

O atributo *marker_size_on_image_solution* espera receber um float. Tem por padrão o valor sete e indica qual será o tamanho dos pontos da imagem do gráfico da rota da solução.

O atributo *line_width_on_image_solution* espera receber um float. Tem por padrão o valor dois e indica qual será a grossura da linha da imagem do gráfico da rota da solução.

O atributo *__logger* espera receber a referência do objeto que usamos para gerar os arquivos de *log*. Este objeto vem do pacote *logging*, que é um pacote padrão de python, usado para gerar arquivos de *log*. Através de métodos deste objeto referenciado em *__logger*, adicionamos novas linhas ao arquivo de *log*, já salvando o horário em que estes *inputs* são feitos no *log*.

E o atributo *__file_name* espera receber uma string. Esta string é montada no construtor da classe *OrToolsTSPSolver*. Este atributo será utilizado para compor o nome do arquivo de *log* e das imagens geradas para uma solução. Vale comentar que é gerado um arquivo de *log* e uma imagem da rota para cada estratégia executada em uma dada instância. Podemos observar como ficam os nomes dos arquivos na Figura 2.4.

Os seis métodos que foram implementados para que as soluções das instâncias fossem capaz de gerar os logs e as imagens são os:

- *__setup_logger*
- *__log_solution*
- *__plot_solution*
- *__log_optimum_solution*

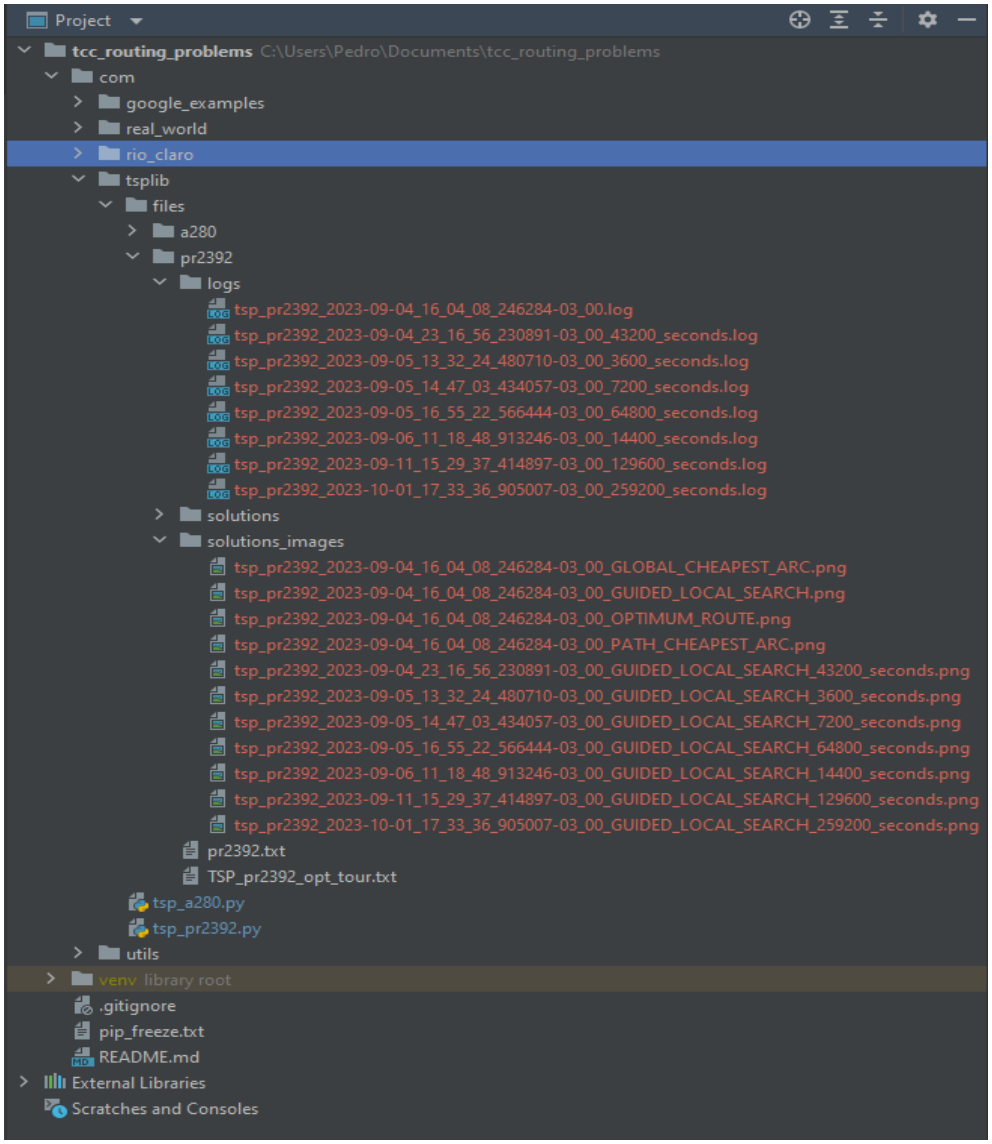

Figura 2.4: Exemplo de como ficam os nomes dos logs e imagens

- *__plot_optimum_solution*
- *log_and_plot_optimum_solution*

O método *__setup_logger* faz as configurações básicas para conseguirmos gerar os arquivos de *log*. Utilizamos o pacote *logging*, um pacote nativo do Python 3, que nos provê as ferramentas necessárias para gerarmos arquivos de *log* em Python. Esta configuração inicial se dá através de um método chamado *basicConfig* do pacote *logging*. Neste método *basicConfig* definimos: como vai ser o nome do arquivo de *log*; o modo como o arquivo de *log* vai ser aberto; qual é o nível de *log* que queremos gerar; e formatamos como serão as linhas ao passar um *input* para o *logger* adicionar ao *log*. E este método sempre será chamado no construtor da classe *OrToolsTSPSolver*.

O método *__log_solution* é o que de fato salva as principais informações. É o método que recebe a solução encontrada, percorre ponto a ponto da solução e vai gerando o *log* da rota da solução encontrada. Ao percorrer cada ponto, também faz um acumulado da distância percorrida para adcionar ao *log*. A única informação salva que não é originária deste método é o *log* do início e do fim da execução de uma estratégia em uma dada instância. Estes *logs* ficam no método que de fato executa a estratégia no *Or-Tools*. Então o *log* do início se dá logo antes de chamarmos o *Or-Tools* para executar uma estratégia. E na sequência, ou seja, quando o *Or-Tools* já executou a estratégia, salvamos o fim da execução. Podemos ver como fica um *log* de uma execução com múltiplas estratégias na Figura 2.5

O método *__plot_solution* é o que gera a imagem que representa a rota da solução encontrada. Este método recebe a solução e percorre item a item os pontos da rota. Imprime um círculo vermelho na coordenada correspondente daquele ponto. Depois de imprimir todas as coordenadas, executa um laço para percorrer todos os pontos da solução, na ordem. Neste laço sempre estamos olhando para um dado ponto e temos o ponto anterior salvo em uma variável, para que possamos traçar a reta entre estes dois pontos. Depois de passar por todos os pontos, traçando uma reta com seu ponto antecessor. Removemos os eixos X e Y da imagem e a salvamos. Podemos ver como fica a imagem de uma estratégia na Figura 2.6.

Os métodos *__log_optimum_solution* e *__plot_optimum_solution* são utilizados somente nas instâncias da *TSPLIB95* para logar e gerar uma imagem da rota que a *TSPLIB95* nos provê como solução ótima para a dada instância.

O método *__log_optimum_solution* lê um arquivo que já contém a solução ótima e então percorre esse arquivo logando ponto a ponto da solução.

```
2023-09-04 15:03:07,724 - root - INFO - Start to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - End to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - Solution achieved by GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - Objective: 2648 Unit of Measure
2023-09-04 15:03:09,802 - root - INFO - Route for vehicle 1:
 1 -> 2 -> 242 -> 243 -> 244 -> 241 -> 240 -> 239 -> 238 -> 237 -> 236 -> 235 -> 234 -> 233 -> 232 -> 231 -> 246 -> 245 ->

2023-09-04 15:03:12,018 - root - INFO - Start to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - End to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - Solution achieved by PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - Objective: 2742 Unit of Measure
2023-09-04 15:03:13,023 - root - INFO - Route for vehicle 1:
 1 -> 2 -> 280 -> 3 -> 279 -> 4 -> 277 -> 276 -> 275 -> 274 -> 273 -> 272 -> 271 -> 16 -> 17 -> 18 -> 133 -> 134 -> 270 ->

2023-09-04 15:03:14,931 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-09-04 15:16:34,932 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-09-04 15:16:34,932 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 800 seconds
2023-09-04 15:16:34,932 - root - INFO - Objective: 2579 Unit of Measure
2023-09-04 15:16:34,933 - root - INFO - Route for vehicle 1:
 1 -> 280 -> 3 -> 279 -> 278 -> 248 -> 249 -> 256 -> 255 -> 254 -> 257 -> 258 -> 259 -> 260 -> 261 -> 262 -> 263 -> 264 ->

2023-09-04 15:16:36,900 - root - INFO - Optimum Solution
2023-09-04 15:16:36,900 - root - INFO - Objective: 2579 Unit of Measure
2023-09-04 15:16:36,900 - root - INFO - Route on opt file:
1 -> 2 -> 242 -> 243 -> 244 -> 241 -> 240 -> 239 -> 238 -> 237 -> 236 -> 235 -> 234 -> 233 -> 232 -> 231 -> 246 -> 245 ->

```

Figura 2.5: Exemplo de como fica o *log* da solução da instância a280 com múltiplas estratégias

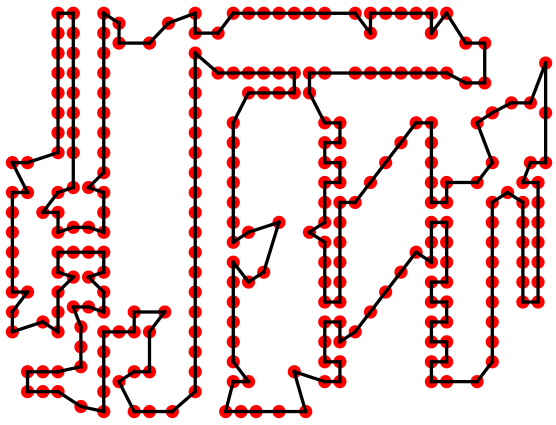

Figura 2.6: Imagem da solução encontrada para a instância a280 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 800 segundos

O método *__plot_optimum_solution* também lê um arquivo que já contém a solução ótima, mas primeiro ele percorre a lista de pontos para imprimir todos os pontos que a solução vai percorrer. Depois ele percorre todos os pontos da solução ótima. Neste laço sempre estamos olhando para um dado ponto e temos o ponto anterior salvo em uma variável, para que possamos traçar a reta entre estes dois pontos. Depois de percorrer todos os pontos e traçar uma reta com seu ponto antecessor, removemos os eixos X e Y da imagem e a salvamos.

O método *log_and_plot_optimum_solution* é o método público que deve ser chamado fora da classe *OrToolsTSPSolver*. Ou seja é o método que podemos chamar quando utilizamos a classe *OrToolsTSPSolver* para resolver alguma instância da *TSPLIB95*. Este método simplemente chama os métodos *__log_optimum_solution* e *__plot_optimum_solution*.

2.5 Passo 5. Google OR-Tools em instâncias de cidades do interior paulista

Esse trabalho utiliza duas cidades do interior paulista como estudo de caso: Artur Nogueira e Rio Claro. A Figura 2.7 mostra a localização de Artur Nogueira no Estado de São Paulo. E a Figura 2.8 mostra a localização de Rio Claro no Estado de São Paulo.

Figura 2.7: Artur Nogueira.

Figura 2.8: Rio Claro.

As principais características de população e economia dessas cidades estão apresentadas na Tabela 2.1.

Tabela 2.1: Dados demográficos de Artur Nogueira e Rio Claro (REF).

	Artur Nogueira	Rio Claro
População	56.247 hab	209.548 hab
Densidade demográfica	248,15 hab/km^2	373,69 hab/km^2
PIB	23.354,82 R$	50.923,39 R$
IDH	0,749	0,803

Para cada uma das cidades analisadas foram utilizadas instâncias de problemas de roteamento. Estas instâncias possuem um par ordenado para cada ponto de entrega e uma matriz de distâncias, que contém as distâncias entre cada par ordenado. As Figuras 2.9 e 2.10 mostram os mapas das cidades usados para criar as instâncias contendo os pares ordenados e a matriz de distância.

Figura 2.9: Artur Nogueira.

O procedimento para gerar estas instâncias utilizou-se de Meira *et al.* 2019, que descreve as equações utilizadas em detalhes. Mas em resumo, cada rua St é modelada como uma cadeia poligonal de coordenadas planas. Usando o exemplo desenvolvido por Meira *et al.* 2019, assumindo uma Rua Universidade, que foi modelada como $P = (c_1, \ldots, c_n')$ com $c \in \mathbb{R}^2$ para todos os $c \in P$. Dessa forma, o mapa completo é $\mathbf{P} = (P_1, ..., P_2)$ de cadeias poligonais, para cada uma das ruas que serão incluídas na análise.

Com base nesse $\mathbf{P}$, é possível construir o $G(V, E)$,

- $v \in V$ é associado com a coordenada cartesiana $(x_v, y_v) \in \mathbb{R}^2$ e é um vértice
- cada borda $e = (u, v)$ é um segmento reto entre os dois pontos.

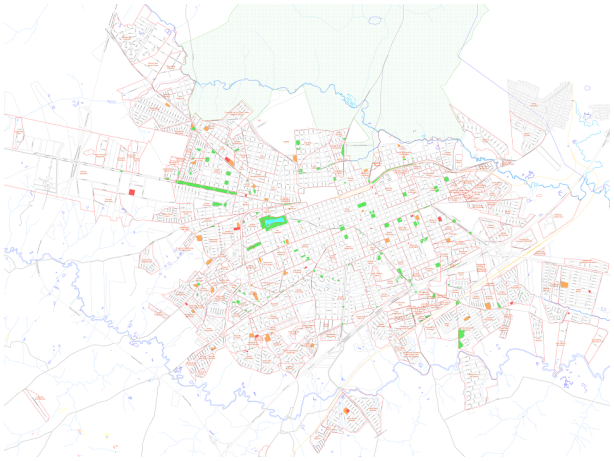

Figura 2.10: Rio Claro.

Para uma reta e, a rua é definida como $St(e)$.

O motivo de se trabalhar com cada rua descrita como uma cadeia poligonal é calcular as distâncias como distâncias num mapa rodoviário e não métricas Euclidianas. Assim, as distâncias das instâncias utilizadas são mais realistas, pois são os caminhos mínimos em um grafo de ruas. Estes valores já estão prontos. Apenas utilizamos a matriz de distância criada por (MEIRA et al., 2019).

Sendo assim, para seguir com o trabalho, o *OR-Tools* foi aplicado em instâncias de problemas reais e de cidades de Artur Nogueira e Rio Claro. Para tal, foi utilizado um software já construido em (MEIRA et al., 2019), para gerar instâncias, referente às cidades de Arthur Nogueira e Rio Claro. Foram geradas instâncias com 20, 200 e 2000 pontos a serem visitados, em cada cidade. Chamarei estes pontos a serem visitados de vértices. E estas instâncias foram utilizadas para fazer experimentos, buscando soluções passando por todos os vértices, com um único veículo, ou seja, um TSP. Mas também foram utilizadas as instâncias em situações com mais de um veículo para percorrer todos os vértices, ou seja, um VRP.

2.6 Passo 6. Google OR-Tools em instâncias de VRP

Depois de implementar e fazer experimentos com instâncias do tipo *TSP*, foi implementada uma classe muito semelhante à *OrToolsTSPSolver*, porém, foi pensado que seu uso fosse em instâncias do tipo *VRP*. Ou seja, foi pensada uma classe para facilitar o uso da ferramenta *Or-Tools* em instâncias do tipo VRP. Chamamos esta classe de *OrToolsVRPSolver*. Vamos mostrar

como fica o uso da *OrToolsVRPSolver* e quais as diferenças para a *OrToolsTSPSolver*. Algumas das principais diferenças entre a *OrToolsTSPSolver* e a *OrToolsVRPSolver* se dá nos métodos para gerarmos os *logs* e as imagens das soluções, mas vamos abordar estas diferenças no próximo subcapítulo *2.7 Passo 7. Desenvolvimento de logs e imagens para instâncias de VRP*.

A Figura 2.11 mostra o uso desta classe para encontrar soluções da instância de Rio Claro com vinte pontos, usando três veículos. Ou seja mostra o uso do construtor da classe para gerar um novo objeto desta classe e depois este objeto chamando o método *execute_strategies* para executar as estratégias que é onde de fato utilizamos o *Or-Tools* para encontrar as soluções.

```
from com.utils.orToolsVRPSolver import OrToolsVRPSolver

if __name__ == '__main__':
    solver = OrToolsVRPSolver(
        problem_name='rio_claro_20',
        matrix_rows_size=21,
        matrix_columns_size=21,
        matrix_type='rio_claro',
        coordinates_type='int',
        calc_dist_type=None,
        dimension_name='Hours',
        num_vehicles=3,
        upper_limit_coefficient=0.7,
        strategies=['GLOBAL_CHEAPEST_ARC', 'PATH_CHEAPEST_ARC', 'GUIDED_LOCAL_SEARCH'],
        time_limit_on_seconds_to_metaheuristics=30,
        log_search_on_terminal=True,
        marker_size_on_image_solution=4,
        line_width_on_image_solution=1
    )

    solver.execute_strategies()
```

Figura 2.11: Construtor da classe OrToolsVRPSolver

Podemos notar que a diferença entre os construtores da *OrToolsTSPSolver*, da Figura 2.2, e a *OrToolsVRPSolver*, da Figura 2.11, é o atributo *num_vehicles* e o atributo *upper_limit_coefficient*.

O atributo *num_vehicles*, é o número de veículos que vamos utilizar para encontrar a solução. Basicamente, quando queremos que o *Or-Tools* resolva uma dada instância de um VRP, precisamos passar um número de veículos diferente de um. Na classe *OrToolsTSPSolver*, quando instanciamos o objeto que fica em seu atributo privado *__manager* sempre passamos o número de veículos como sendo o número um. Já na classe *OrToolsVRPSolver* quando instanciamos o *__manager* passamos o número de veículos que foi passado no construtor da classe.

O outro atributo em que há diferença é o *upper_limit_coefficient*. Este atributo é utilizado para algo que não precisávamos nos preocupar quando estávamos trabalhando com instâncias

do tipo *TSP*, que é o limite que um veículo pode percorrer. Nas instâncias de problemas VRP que foram feitos experimentos, que são instâncias provenientes do trabalho de (MEIRA et al., 2019), sempre recebemos um valor correspondente ao máximo que uma rota de um veículo poderia percorrer.

Então pegamos este valor, do máximo de uma rota de um veículo, e aplicamos o coeficiente que é passado em *upper_limit_coefficient*. Poderíamos passar direto o valor que obtemos das instâncias, mas, para poder ir restringindo aos poucos o limite do trajeto dos veículos, foi pensado nesse coeficiente. Pois, se deixamos muito alto, este valor do trajeto máximo de qualquer veículo, o *Or-Tools* fica desbalencado, ou seja, o *Or-Tools* acaba sobrecarregando o trajeto de um veículo e deixando os demais veículos com um trajeto curto. Então, com este coeficiente, conseguimos ir polindo, para que a distribuição dos trajetos em cada veículo sejam semelhantes. Logo quando trabalhamos com instâncias do tipo *VRP* precisamos ir ajustando este valor para que encontremos soluções melhores.

2.7 Passo 7. Desenvolvimento de logs e imagens para instâncias de VRP

Por fim, foi necessário implementar novos métodos para gerar os *logs* e as imagens quando estamos abordando um problema do tipo *VRP*. Apesar dos métodos seguirem ideias parecidas dos métodos usados em problemas do tipo *TSP*, a solução que obtemos em instâncias do tipo *VRP* são bem diferentes, vamos ter múltiplas rotas, uma rota para cada veículo contendo os pontos que aquele dado veículo deve percorrer.

A diferença destes métodos é que ao invés de aplicar a lógica somente uma vez, pois tínhamos somente um veículo, aplicamos a lógica diversas vezes, dependendo do número de veículos. Ou seja a complexidade aumenta, pois realizamos um laço a mais para percorrer os veículos e dentro deste laço a lógica que seguimos para cada veículo é a mesma que era feita para um veículo e foi descrita no subcapítulo *2.4*.

Podemos ver como fica o log de uma execução de uma instância *VRP* na Figura 2.12

E Podemos ver como fica a imagem de uma estratégia, de uma instância *VRP*, na Figura 2.13.

```
2023-08-28 18:51:17,720 - root - INFO - Start to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-08-28 18:51:47,435 - root - INFO - End to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-08-28 18:51:47,435 - root - INFO - Solution achieved by GLOBAL_CHEAPEST_ARC strategy
2023-08-28 18:51:47,435 - root - INFO - Max Allowed Route: 4.199959944444444  Hours
2023-08-28 18:51:47,435 - root - INFO - Max Possible Route: 5.999942777777778  Hours
2023-08-28 18:51:47,435 - root - INFO - Route for vehicle 0:
 1 -> 4 -> 2 -> 19 -> 6 -> 10 -> 1
Distance of the route: 3.7679377777777776 Hours

2023-08-28 18:51:47,436 - root - INFO - Route for vehicle 1:
 1 -> 9 -> 8 -> 7 -> 15 -> 14 -> 20 -> 1
Distance of the route: 3.9564208333333335 Hours

2023-08-28 18:51:47,436 - root - INFO - Route for vehicle 2:
 1 -> 11 -> 3 -> 12 -> 16 -> 18 -> 17 -> 13 -> 21 -> 5 -> 1
Distance of the route: 3.7726261111111112 Hours

2023-08-28 18:51:47,436 - root - INFO - Maximum of the route distances: 3.9564208333333335 Hours
```

Figura 2.12: Exemplo de como fica o log da solução da instância VRP_RIO_CLARO_20 com a estratégia GLOBAL_CHEAPEST_ARC

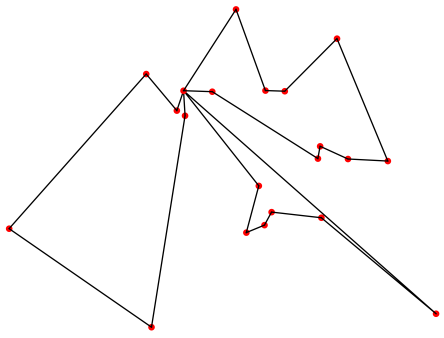

Figura 2.13: Imagem da solução encontrada para a instância VRP_RIO_CLARO_20 utilizando a estratégia GLOBAL_CHEAPEST_ARC

Capítulo 3

Experimentos e Resultados

Neste capítulo serão abordados os experimentos realizados utilizando o Google *OR-Tools* e como foram os resultados destes experimentos. Para realizar estes experimentos com o *OR-Tools*, utilizamos tudo que foi implementado neste trabalho. E todo o desenvolvimento está explicado em detalhes no Capítulo 2. Estes experimentos nos trazem os resultados em forma de *logs*, contendo os tempos de execução e a rota encontrada em cada experimento. E também geram uma imagem, que seria uma representação gráfica da rota encontrada. Vale resaltar, que todos os *logs* e imagens dos experimentos estão versionadas publicamente no github[1].

Os resultados dos experimentos serão abordados na mesma sequência em que os experimentos foram realizados. Inicialmente traremos os resultados obtidos com instâncias da *TSPLIB95*. Depois os resultados obtidos com as instâncias das cidades de Rio Claro e Artur Nogueira utilizando somente um veículo, ou seja estas instâncias abordadas como problemas do tipo *TSP*. E por fim traremos os resultados obtidos das instâncias de Rio Claro, porém abordando o problema com múltiplos veículos, ou seja estas instâncias abordadas como problemas do tipo *VRP*. Logo, este capítulo será organizado em três subtópicos:

1. Com os resultados dos experimentos com a *TSPLIB95*;
2. Com os resultados dos experimentos das instâncias das cidades de Rio Claro e Artur Nogueira utilizando somente um veículo;
3. E por fim os resultados dos experimentos das instâncias da cidade de Rio Claro utilizando mais de um veículo.

[1] https://github.com/PedroHenriqueBianchi/Google-OR-Tools-Routing-Issues

3.1 Resultados *TSPLIB95*

Inicialmente pensamos em realizar experimentos com problemas populares e que já tivessem sua solução ótima conhecida, pois desta forma conseguimos comparar as soluções encontradas com a solução ótima e também tentar chegar na solução ótima utilizando o Google *OR-Tools*. Logo pensamos nas instâncias da *TSPLIB95*, das quais, todos os problemas já foram solucionados, encontrando-se a solução ótima para cada instância. Sendo assim trabalhamos com duas instâncias da *TSPLIB95*: *(i)* a instância a280; *(ii)* e a instância pr2392.

3.1.1 Instância a280

A instância a280 é uma instância do tipo *TSP* que contém duzentos e oitenta pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e descreve que a distância entre estes pontos deve ser calculada utilizando a distância Euclidiana.

A Tabela 3.1 contém todos os resultados dos experimentos com a instância a280. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* o custo da solução encontrada; *(iii)* e quanto foi a diferença do custo da solução encontrada para o custo da solução ótima em cada experimento, chamaremos esta diferença de *gap*[2].

Para a estratégia *PATH_CHEAPEST_ARC* podemos ver na Figura 3.1 a rota desta solução e podemos ver na Figura 3.2 o *log* desta execução.

Para a estratégia *GLOBAL_CHEAPEST_ARC* podemos ver na Figura 3.3 a rota desta solução e podemos ver na Figura 3.4 o *log* desta execução.

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 750 segundos, qual chegou na solução ótima, podemos ver na Figura 3.5 a rota desta solução e podemos ver na Figura 3.6 o *log* desta execução.

Na Figura 3.7 temos a solução ótima, fornecida pela *TSPLIB95*. A solução ótima difere da rota encontrada na execução da estratégia *GUIDED_LOCAL_SEARCH* por 750 segundos, porém os custos são os mesmos. Sendo assim, temos duas rotas diferentes que são ótimas. Se observarmos a Figura 3.5 e a Figura 3.7 podemos ver pequenas modificações na rota.

[2] *Gap* seria a diferença do custo da solução encontrada para a solução ótima, por exemplo, imagine um custo da estratégia x como sendo 12 e uma solução ótima de custo 10, o *gap* da estratégia x para a solução ótima é de 20%

Tabela 3.1: Resultados dos experimentos com a instância a280.

Estratégias	Tempo de Execução	Custo	*Gap*
Solução ótima	-	2579	-
PATH_CHEAPEST_ARC	1 segundo	2742	6,3%
GLOBAL_CHEAPEST_ARC	2 segundos	2648	2,6%
GUIDED_LOCAL_SEARCH	10 segundos	2642	2,4%
GUIDED_LOCAL_SEARCH	30 segundos	2635	2,1%
GUIDED_LOCAL_SEARCH	60 segundos	2618	1,5%
GUIDED_LOCAL_SEARCH	120 segundos	2598	0,7%
GUIDED_LOCAL_SEARCH	180 segundos	2592	0,5%
GUIDED_LOCAL_SEARCH	240 segundos	2592	0,5%
GUIDED_LOCAL_SEARCH	300 segundos	2592	0,5%
GUIDED_LOCAL_SEARCH	360 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	420 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	480 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	540 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	600 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	660 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	720 segundos	2588	0,3%
GUIDED_LOCAL_SEARCH	750 segundos	2579	0%

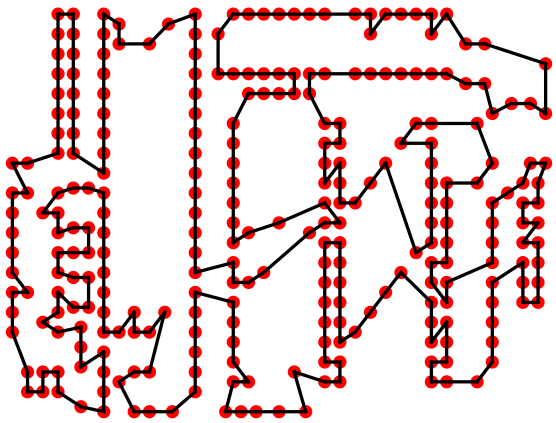

Figura 3.1: Figura com a solução da instância a280 utilizando a estratégia *PATH_CHEAPEST_ARC*.

```
2023-09-04 15:03:12,018 - root - INFO - Start to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - End to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - Solution achieved by PATH_CHEAPEST_ARC strategy
2023-09-04 15:03:13,022 - root - INFO - Objective: 2742 Unit of Measure
2023-09-04 15:03:13,023 - root - INFO - Route for vehicle 1:
 1 -> 2 -> 280 -> 3 -> 279 -> 4 -> 277 -> 276 -> 275 -> 274 -> 273 -> 272 -> 271 -> 16 -> 17 -> 18 -> 133 -> 134 -> 270
```

Figura 3.2: *Log* da solução da instância a280 utilizando a estratégia *PATH_CHEAPEST_ARC*.

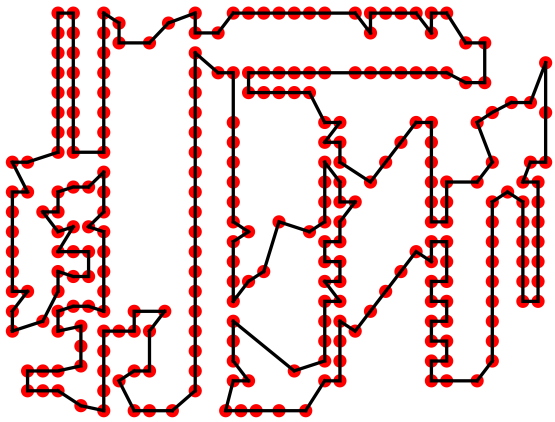

Figura 3.3: Figura com a solução da instância a280 utilizando a estratégia *GLOBAL_CHEAPEST_ARC*.

```
2023-09-04 15:03:07,724 - root - INFO - Start to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - End to solve problem with GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - Solution achieved by GLOBAL_CHEAPEST_ARC strategy
2023-09-04 15:03:09,802 - root - INFO - Objective: 2648 Unit of Measure
2023-09-04 15:03:09,802 - root - INFO - Route for vehicle 1:
 1 -> 2 -> 242 -> 243 -> 244 -> 241 -> 240 -> 239 -> 238 -> 237 -> 236 -> 235 -> 234 -> 233 -> 232 -> 231 -> 246 -> 245
```

Figura 3.4: *Log* da solução da instância a280 utilizando a estratégia *GLOBAL_CHEAPEST_ARC*.

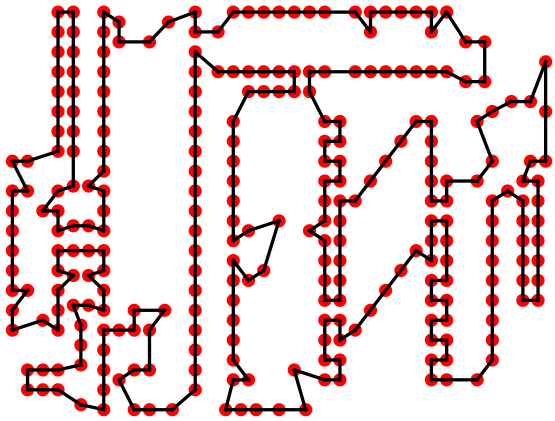

Figura 3.5: Figura com a solução da instância a280 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 750 segundos.

```
2023-09-04 20:09:04,626 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-09-04 20:21:34,627 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-09-04 20:21:34,627 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 750 seconds
2023-09-04 20:21:34,627 - root - INFO - Objective: 2579 Unit of Measure
2023-09-04 20:21:34,627 - root - INFO - Route for vehicle 1:
 1 -> 280 -> 3 -> 279 -> 278 -> 248 -> 249 -> 256 -> 255 -> 254 -> 257 -> 258 -> 259 -> 260 -> 261 -> 262 -> 263 -> 264 -
```

Figura 3.6: *Log* da solução da instância a280 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 750 segundos.

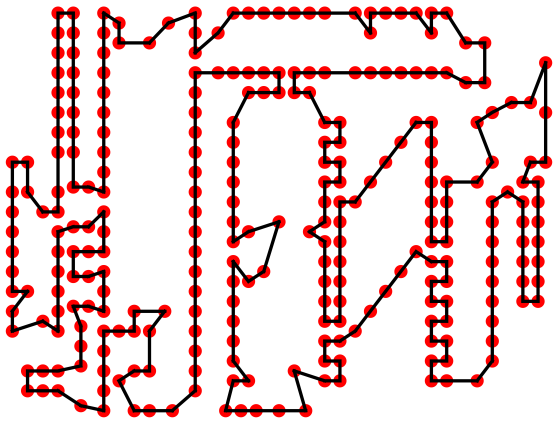

Figura 3.7: Figura com a solução ótima da instância a280.

3.1.2 Instância pr2392

A instância pr2392 é uma instância do tipo TSP que contém dois mil trezentos e noventa e dois pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e descreve que a distância entre estes pontos deve ser calculada utlizando a distância Euclidiana.

A Tabela 3.2 contém todos os resultados dos experimentos com a instância pr2392. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* o custo da solução encontrada; *(iii)* e quanto foi a diferença do custo da solução encontrada para o custo da solução ótima em cada experimento, o *gap*.

Tabela 3.2: Resultados dos experimentos com a instância pr2392.

Estratégias	Tempo de Execução	Custo	*Gap*
Solução ótima	-	378032	-
PATH_CHEAPEST_ARC	3,5 minutos	395969	4,7%
GLOBAL_CHEAPEST_ARC	5 minutos	399729	5,7%
GUIDED_LOCAL_SEARCH	1 minuto	398866	5,5%
GUIDED_LOCAL_SEARCH	10 minutos	395581	4,6%
GUIDED_LOCAL_SEARCH	20 minutos	394946	4,4%
GUIDED_LOCAL_SEARCH	1 hora	392444	3,8%
GUIDED_LOCAL_SEARCH	2 horas	392071	3,7%
GUIDED_LOCAL_SEARCH	4 horas	390451	3,2%
GUIDED_LOCAL_SEARCH	12 horas	389280	2,9%
GUIDED_LOCAL_SEARCH	18 horas	388069	2,6%
GUIDED_LOCAL_SEARCH	36 horas	386280	2,1%
GUIDED_LOCAL_SEARCH	72 horas	385095	1,8%

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 72 horas, que chegou mais próximo da solução ótima, podemos ver na Figura 3.8 a rota desta solução e podemos ver na Figura 3.9 o *log* desta execução.

E na Figura 3.10 temos a solução ótima fornecida pela *TSPLIB95*.

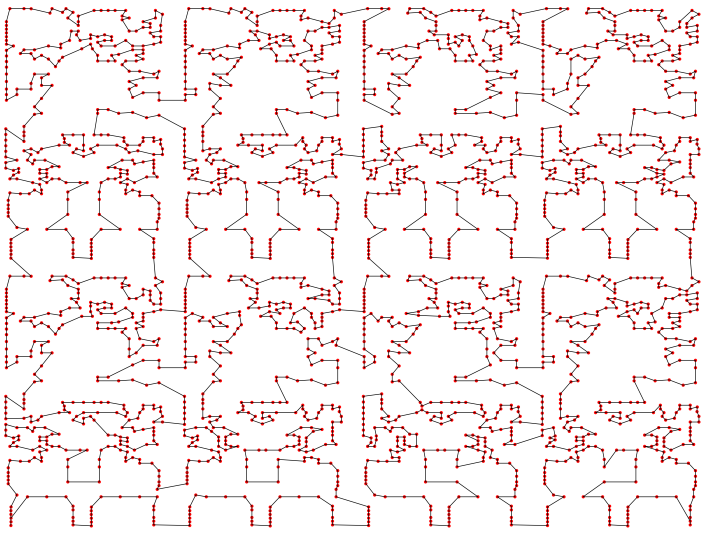

Figura 3.8: Figura com a solução da instância pr2392 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

```
2023-10-01 17:33:39,443 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-04 17:33:39,456 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-04 17:33:39,458 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 259200 seconds
2023-10-04 17:33:39,458 - root - INFO - Objective: 385095 Unit of Measure
2023-10-04 17:33:39,475 - root - INFO - Route for vehicle 1:
 1 -> 2392 -> 2390 -> 2389 -> 2388 -> 2387 -> 2386 -> 2385 -> 2384 -> 2383 -> 2382 -> 2381 -> 2380 -> 2379 -> 2378 -> 2377
```

Figura 3.9: *Log* da solução da instância pr2392 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

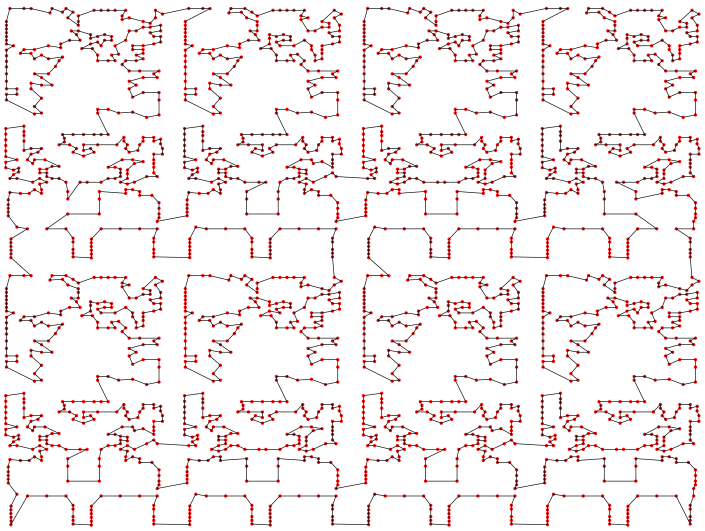

Figura 3.10: Figura com a solução ótima da instância pr2392.

3.2 Resultados das instâncias das cidades de Rio Claro e Artur Nogueira como TSP

Depois de realizar experimentos com as instâncias da *TSPLIB95*, pensamos em utilizar as instâncias do trabalho (MEIRA et al., 2019), pois estas instâncias refletem uma situação real de entrega de encomendas, com as distâncias reais entre cada entrega, como se estivéssemos percorrendo as ruas das cidades de Artur Nogueira e Rio Claro. E estas distâncias são dadas em forma de tempo, que seria o tempo de deslocamento de um veículo de um ponto ao outro. Ou seja o grafo ponderado que representa a rota destas soluções possui tempo nas arestas e o tempo representa o tempo de deslocamente de um veículo.

Porém, neste momento utilizamos estas instâncias para encontrar soluções com um único veículo, ou seja, como instâncias do tipo TSP. E trabalhamos com três tamanhos de instâncias para cada cidade, uma instância com vinte e um pontos de entrega, outra com duzentos e um pontos de entrega e uma com dois mil e um pontos de entrega.

3.2.1 Instância *TSP* Rio Claro 20

A instância *TSP* Rio Claro 20 contém vinte e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e também todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos.

A Tabela 3.3 contém todos os resultados dos experimentos com a instância *TSP* Rio Claro 20. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo da solução encontrada.

Tabela 3.3: Resultados dos experimentos com a instância *TSP* Rio Claro 20.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	0,05 segundos	9,053765
GLOBAL_CHEAPEST_ARC	0,4 segundos	9,091665
GUIDED_LOCAL_SEARCH	10 segundos	9,053765
GUIDED_LOCAL_SEARCH	30 segundos	9,053765
GUIDED_LOCAL_SEARCH	1 minuto	9,053765
GUIDED_LOCAL_SEARCH	2 minutos	9,053765
GUIDED_LOCAL_SEARCH	5 minutos	9,053765
GUIDED_LOCAL_SEARCH	10 minutos	9,053765

Como podemos observar na Tabela 3.3, o custo para realizar as vinte entregas desta instância seria algo muito próximo a nove horas. Logo, pensando que estas fossem as vinte entregas que um carteiro precisa realizar, seria impossível para este carteiro realizar estas vinte entregas em um dia de trabalho, sem horas extras. Então, para que estas entregas fossem realizadas em um único dia seria necessário mais veículos nesta rota, ou seja, precisariamos abordar esta instância como um *VRP*.

Para a estratégia *PATH_CHEAPEST_ARC*, que atingiu a melhor solução que encontramos, em apenas 5 centésimos de segundo, podemos ver na Figura 3.11 a rota desta solução e podemos ver na Figura 3.12 o *log* desta execução.

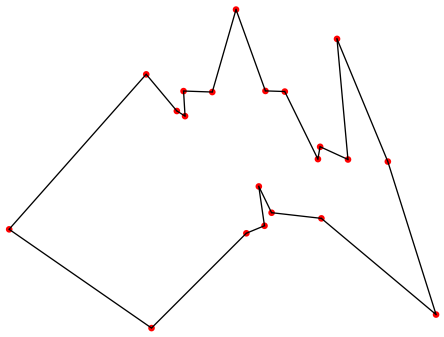

Figura 3.11: Figura com a solução da instância Rio Claro 20 utilizando a estratégia *PATH_CHEAPEST_ARC*.

```
2023-11-13 18:30:24,879 - root - INFO - Start to solve problem with PATH_CHEAPEST_ARC strategy
2023-11-13 18:30:24,884 - root - INFO - End to solve problem with PATH_CHEAPEST_ARC strategy
2023-11-13 18:30:24,884 - root - INFO - Solution achieved by PATH_CHEAPEST_ARC strategy
2023-11-13 18:30:24,884 - root - INFO - Objective: 9.053765555555556 Horas
2023-11-13 18:30:24,884 - root - INFO - Max Allowed Route: 5.999942777777778 Horas
2023-11-13 18:30:24,884 - root - INFO - Length per max route: 1.508975317079413
2023-11-13 18:30:24,884 - root - INFO - Route for vehicle 1:
 1 -> 4 -> 10 -> 6 -> 19 -> 2 -> 8 -> 7 -> 9 -> 15 -> 14 -> 20 -> 18 -> 16 -> 17 -> 13 -> 21 -> 12 -> 3 -> 11 -> 5 -> 1
```

Figura 3.12: *Log* da solução da instância Rio Claro 20 utilizando a estratégia *PATH_CHEAPEST_ARC*.

3.2.2 Instância *TSP* Rio Claro 200

A instância *TSP* Rio Claro 200 contém duzentos e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos.

A Tabela 3.4 contém todos os resultados dos experimentos com a instância *TSP* Rio Claro 200. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo acumulado na solução encontrada.

Como podemos observar na Tabela 3.4 o custo para realizar as duzentas entregas desta instância seria algo muito próximo a vinte e quatro horas. Logo, pensando que estas fossem as duzentas entregas que um carteiro precisa realizar, seria impossível para este carteiro realizar estas duzentas entregas em um dia de trabalho, na verdade pensando em uma jornada de seis horas de trabalho, tomaria quatro dias úteis para um carteiro realizar todas as entregas. Então, para que estas entregas fossem realizadas em um único dia seria necessário mais veículos nesta rota, ou seja, precisariamos abordar esta instância como um *VRP*.

Tabela 3.4: Resultados dos experimentos com a instância *TSP* Rio Claro 200.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	1 segundo	24,08623
GLOBAL_CHEAPEST_ARC	1 segundo	24,02724
GUIDED_LOCAL_SEARCH	10 segundos	23,97914
GUIDED_LOCAL_SEARCH	30 segundos	23,88201
GUIDED_LOCAL_SEARCH	1 minuto	23,86396
GUIDED_LOCAL_SEARCH	2 minutos	23,69625
GUIDED_LOCAL_SEARCH	5 minutos	23,40522
GUIDED_LOCAL_SEARCH	7 minutos	23,29801
GUIDED_LOCAL_SEARCH	10 minutos	23,25918
GUIDED_LOCAL_SEARCH	13 minutos	23,21643

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 800 segundos, qual atingiu a melhor solução que encontramos, podemos ver na Figura 3.13 a rota desta solução e podemos ver na Figura 3.14 o *log* desta execução.

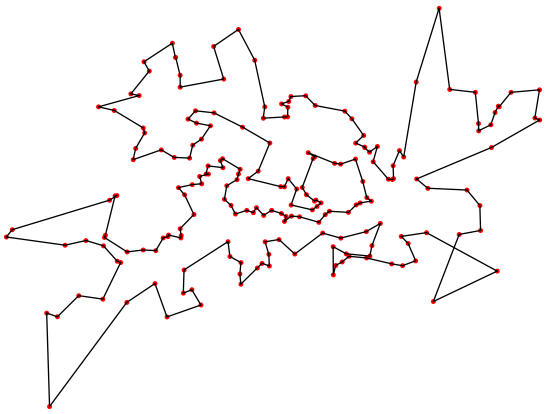

Figura 3.13: Figura com a solução da instância Rio Claro 200 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 800 segundos.

```
2023-11-13 19:52:26,395 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-13 20:05:46,396 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-13 20:05:46,396 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 800 seconds
2023-11-13 20:05:46,396 - root - INFO - Objective: 23.216430833333334 Horas
2023-11-13 20:05:46,396 - root - INFO - Max Allowed Route: 12.0 Horas
2023-11-13 20:05:46,396 - root - INFO - Length per max route: 1.9347025694444444
2023-11-13 20:05:46,397 - root - INFO - Route for vehicle 1:
 1 -> 181 -> 200 -> 30 -> 129 -> 142 -> 158 -> 177 -> 10 -> 91 -> 192 -> 51 -> 16 -> 138 -> 13 -> 157 -> 99 -> 115 -> 194
```

Figura 3.14: *Log* da solução da instância Rio Claro 200 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 800 segundos.

3.2.3 Instância *TSP* Rio Claro 2000

A instância *TSP* Rio Claro 2000 contém dois mil e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos.

A Tabela 3.5 contém todos os resultados dos experimentos com a instância *TSP* Rio Claro 2000. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo acumulado na solução encontrada.

Tabela 3.5: Resultados dos experimentos com a instância *TSP* Rio Claro 2000.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	2 minutos	70,144886
GLOBAL_CHEAPEST_ARC	4 minutos	70,496005
GUIDED_LOCAL_SEARCH	1 minuto	70,634155
GUIDED_LOCAL_SEARCH	2 minutos	70,496005
GUIDED_LOCAL_SEARCH	5 minutos	70,481078
GUIDED_LOCAL_SEARCH	10 minutos	70,474148
GUIDED_LOCAL_SEARCH	20 minutos	70,474148
GUIDED_LOCAL_SEARCH	1 hora	70,344005
GUIDED_LOCAL_SEARCH	2 horas	70,131533
GUIDED_LOCAL_SEARCH	4 horas	69,900446
GUIDED_LOCAL_SEARCH	12 horas	69,859621
GUIDED_LOCAL_SEARCH	24 horas	69,671533
GUIDED_LOCAL_SEARCH	72 horas	69,089033

Como podemos observar na Tabela 3.5 o custo para realizar as duas mil entregas desta instância seria algo muito próximo a setenta horas. Logo, pensando que estas fossem as duas

mil entregas que um carteiro precisa realizar, seria impossível para este carteiro realizar estas duas mil entregas em um dia de trabalho, na verdade pensando em uma jornada de seis horas de trabalho, tomaria doze dias úteis para um carteiro realizar todas as entregas. Então, para que estas entregas fossem realizadas em um único dia seria necessário mais veículos nesta rota, ou seja, precisariamos abordar esta instância como um *VRP*.

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 72 horas, que atingiu a melhor solução que encontramos, podemos ver na Figura 3.15 a rota desta solução e podemos ver na Figura 3.16 o *log* desta execução. É interessante comentar que na Figura 3.15 apesar de temos arestas se cruzando, não é um indício que esta não seja uma solução ótima, pois as distâncias não são euclidianas. No caso das instâncias com distância euclidiana entre os pontos, o cruzamento de arestas no grafo da solução comumente indica que a solução não é ótima.

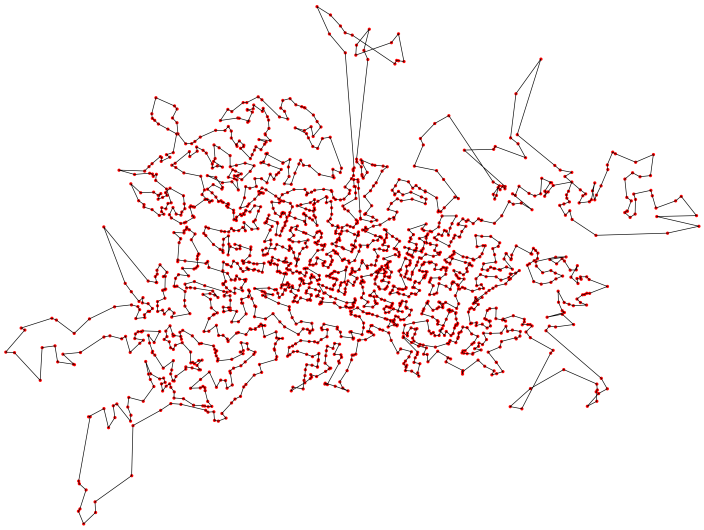

Figura 3.15: Figura com a solução da instância Rio Claro 2000 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

```
2023-10-12 19:12:28,023 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-15 19:12:28,031 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-15 19:12:28,034 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 259200 seconds
2023-10-15 19:12:28,035 - root - INFO - Objective: 69.08903333333333 Horas
2023-10-15 19:12:28,035 - root - INFO - Max Allowed Route: 5.999942777777778 Horas
2023-10-15 19:12:28,035 - root - INFO - Length per max route: 11.51494870738119
2023-10-15 19:12:28,052 - root - INFO - Route for vehicle 1:
 1 -> 901 -> 1037 -> 946 -> 856 -> 271 -> 1108 -> 830 -> 1047 -> 206 -> 1691 -> 1460 -> 994 -> 438 -> 1592 -> 1443 -> 1653 -
```

Figura 3.16: *Log* da solução da instância Rio Claro 2000 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

3.2.4 Instância *TSP* Artur Nogueira 20

A instância *TSP* Artur Nogueira 20 contém vinte e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e também todas as distâncias entre cada ponto, com as distâncias dadas em microsegundos.

A Tabela 3.6 contém todos os resultados dos experimentos com a instância *TSP* Artur Nogueira 20. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo da solução encontrada.

Tabela 3.6: Resultados dos experimentos com a instância *TSP* Artur Nogueira 20.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	0,04 segundos	3,119208
GLOBAL_CHEAPEST_ARC	0,5 segundos	3,119208
GUIDED_LOCAL_SEARCH	10 segundos	3,119208
GUIDED_LOCAL_SEARCH	30 segundos	3,119208
GUIDED_LOCAL_SEARCH	1 minuto	3,119208
GUIDED_LOCAL_SEARCH	2 minutos	3,119208
GUIDED_LOCAL_SEARCH	5 minutos	3,119208
GUIDED_LOCAL_SEARCH	10 minutos	3,119208

Como podemos observar na tabela 3.6 o custo para as vinte entregas desta instância seria algo próximo a três horas. O tempo é menor se compararmos com a instância de vinte entregas de Rio Claro, pois Artur Nogueira é uma cidade menor que Rio Claro. Então, seria possível para um único carteiro realizar estas vinte entegas em um único dia de trabalho e ainda teria tempo para mais entregas.

Para a estratégia *PATH_CHEAPEST_ARC*, qual atingiu a melhor solução que encontramos em apenas 4 centésimos de segundo, podemos ver na Figura 3.17 a rota desta solução e podemos ver na Figura 3.18 o *log* desta execução.

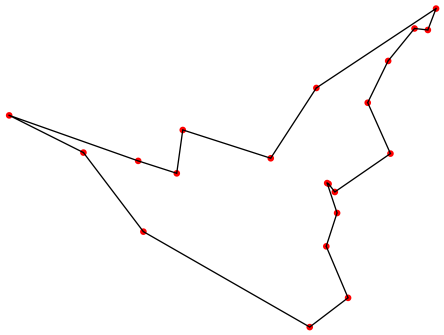

Figura 3.17: Figura com a solução da instância Artur Nogueira 20 utilizando a estratégia *PATH_CHEAPEST_ARC*.

```
2023-09-06 19:38:28,229 - root - INFO - Start to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-06 19:38:28,233 - root - INFO - End to solve problem with PATH_CHEAPEST_ARC strategy
2023-09-06 19:38:28,233 - root - INFO - Solution achieved by PATH_CHEAPEST_ARC strategy
2023-09-06 19:38:28,233 - root - INFO - Objective: 3.1192081519444446 Horas
2023-09-06 19:38:28,233 - root - INFO - Max Allowed Route: 5.999933333333333 Horas
2023-09-06 19:38:28,233 - root - INFO - Length per max route: 0.5198738016996486
2023-09-06 19:38:28,233 - root - INFO - Route for vehicle 1:
 1 -> 21 -> 8 -> 9 -> 12 -> 16 -> 17 -> 7 -> 11 -> 3 -> 2 -> 20 -> 5 -> 18 -> 13 -> 15 -> 14 -> 6 -> 4 -> 10 -> 19 -> 1
```

Figura 3.18: Exemplo do *log* da solução da instância Artur Nogueira 20 utilizando a estratégia *PATH_CHEAPEST_ARC*.

3.2.5 Instância *TSP* Artur Nogueira 200

A instância *TSP* Artur Nogueira 200 contém duzentos e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e também todas as distâncias entre cada ponto, com as distâncias dadas em microsegundos.

A Tabela 3.7 contém todos os resultados dos experimentos com a instância *TSP* Artur Nogueira 200. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo da solução encontrada.

Como podemos observar na Tabela 3.7 o custo para realizar as duzentas entregas desta instância seria algo muito próximo a nove horas. Logo, pensando que estas fossem as

Tabela 3.7: Resultados dos experimentos com a instância *TSP* Artur Nogueira 200.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	1 segundo	9,115908
GLOBAL_CHEAPEST_ARC	1 segundo	9,178792
GUIDED_LOCAL_SEARCH	10 segundos	9,052085
GUIDED_LOCAL_SEARCH	30 segundos	8,797569
GUIDED_LOCAL_SEARCH	1 minuto	8,797569
GUIDED_LOCAL_SEARCH	2 minutos	8,797569
GUIDED_LOCAL_SEARCH	5 minutos	8,797569
GUIDED_LOCAL_SEARCH	7 minutos	8,797569
GUIDED_LOCAL_SEARCH	10 minutos	8,694470
GUIDED_LOCAL_SEARCH	13 minutos	8,694470

duzentas entregas que um carteiro precisa realizar, seria impossível para este carteiro realizar estas duzentas entregas em um dia de trabalho, sem horas extras. Então, para que estas entregas fossem realizadas em um único dia seria necessário mais veículos nesta rota, ou seja, precisariamos abordar esta instância como um *VRP*.

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 10 minutos, que atingiu a melhor solução que encontramos, podemos ver na Figura 3.19 a rota desta solução e podemos ver na Figura 3.20 o *log* desta execução.

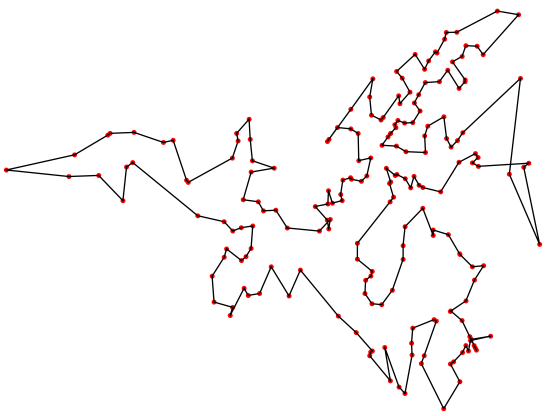

Figura 3.19: Figura com a solução da instância Artur Nogueira 200 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 10 minutos.

```
2023-11-13 20:19:05,231 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-13 20:29:05,232 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-13 20:29:05,232 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 600 seconds
2023-11-13 20:29:05,232 - root - INFO - Objective: 8.694470900277778 Horas
2023-11-13 20:29:05,232 - root - INFO - Max Allowed Route: 5.999933333333333 Horas
2023-11-13 20:29:05,232 - root - INFO - Length per max route: 1.449094584430568
2023-11-13 20:29:05,232 - root - INFO - Route for vehicle 1:
 1 -> 189 -> 148 -> 124 -> 90 -> 22 -> 142 -> 56 -> 59 -> 158 -> 147 -> 111 -> 55 -> 177 -> 60 -> 84 -> 146 -> 107 -> 41
```

Figura 3.20: *Log* da solução da instância Artur Nogueira 200 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 10 minutos.

3.2.6 Instância *TSP* Artur Nogueira 2000

A instância *TSP* Artur Nogueira 2000 contém dois mil e um pontos a serem visitados por um único veículo. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado e também todas as distâncias entre cada ponto, com as distâncias dadas em microsegundos.

A Tabela 3.8 contém todos os resultados dos experimentos com a instância *TSP* Artur Nogueira 2000. Trazendo os resultados para as diferentes estratégias que utilizamos. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* e o custo da solução encontrada.

Tabela 3.8: Resultados dos experimentos com a instância *TSP* Artur Nogueira 2000.

Estratégias	Tempo de Execução	Custo em Horas
PATH_CHEAPEST_ARC	4 minutos	24,579019
GLOBAL_CHEAPEST_ARC	4 minutos	24,781086
GUIDED_LOCAL_SEARCH	1 minuto	24,908276
GUIDED_LOCAL_SEARCH	2 minutos	24,809419
GUIDED_LOCAL_SEARCH	5 minutos	24,579019
GUIDED_LOCAL_SEARCH	10 minutos	24,579019
GUIDED_LOCAL_SEARCH	20 minutos	24,579019
GUIDED_LOCAL_SEARCH	1 hora	24,579019
GUIDED_LOCAL_SEARCH	2 horas	24,579019
GUIDED_LOCAL_SEARCH	4 horas	24,555521
GUIDED_LOCAL_SEARCH	12 horas	24,468670
GUIDED_LOCAL_SEARCH	24 horas	24,358616
GUIDED_LOCAL_SEARCH	72 horas	24,311314

Como podemos observar na Tabela 3.8 o custo para realizar as duas mil entregas desta instância seria algo muito próximo a vinte e quatro horas. Logo, pensando que estas fossem as

duas mil entregas que um carteiro precisa realizar, seria impossível para este carteiro realizar estas duas mil entregas em um dia de trabalho, na verdade pensando em uma jornada de seis horas de trabalho, tomaria quatro dias úteis para um carteiro realizar todas as entregas. Então, para que estas entregas fossem realizadas em um único dia seria necessário mais veículos nesta rota, ou seja, precisariamos abordar esta instância como um *VRP*.

Para a estratégia *GUIDED_LOCAL_SEARCH* executada durante 72 horas, que já atingiu a melhor solução que encontramos, podemos ver na Figura 3.21 a rota desta solução e podemos ver na Figura 3.22 o *log* desta execução.

Figura 3.21: Figura com a solução da instância Artur Nogueira 2000 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

```
2023-10-12 19:12:05,327 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-15 19:12:05,342 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-10-15 19:12:05,347 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 259200 seconds
2023-10-15 19:12:05,348 - root - INFO - Objective: 24.311314435833335 Horas
2023-10-15 19:12:05,348 - root - INFO - Max Allowed Route: 5.999933333333333 Horas
2023-10-15 19:12:05,348 - root - INFO - Length per max route: 4.051930760758453
2023-10-15 19:12:05,383 - root - INFO - Route for vehicle 1:
 1 -> 588 -> 189 -> 1183 -> 991 -> 1064 -> 1689 -> 400 -> 1116 -> 831 -> 885 -> 1235 -> 815 -> 1352 -> 1831 -> 1279 -> 1906 -> 332 -> 908
```

Figura 3.22: *Log* da solução da instância Artur Nogueira 2000 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

3.3 Resultados das instâncias da cidade de Rio Claro como VRP

Depois de realizar experimentos com as instâncias da *TSPLIB95*, e das instâncias de Rio Claro e Artur Nogueira como *TSP*, gostaríamos de realizar experimentos com instâncias do tipo *VRP* e como vimos nas soluções das instâncias de Rio Claro, até a menor das instâncias, com vinte entregas, não seria possível para um carteiro realizar todas as entregas em um único dia de trabalho. Então, pensamos em abordar estas instâncias com múltiplos veículos, ou seja, como *VRP*.

Esta seção será divida em três subseções que trarão os resultados dos experimentos para as instâncias: *(i) VRP* Rio Claro 20; *(ii) VRP* Rio Claro 200; e *(iii) VRP* Rio Claro 2000.

3.3.1 Instância *VRP* Rio Claro 20

A instância *VRP* Rio Claro 20 contém vinte e um pontos a serem visitados por três veículos. O objetivo é fazer com que as rotas de cada veículo tenham custo máximo de seis horas, que é um tempo aceitável para uma pessoa realizar as entregas durante um dia de trabalho. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado, todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos e também nos dá um limite de tempo, ou seja um limite do custo para cada veículo. Mas em todos os testes foram aplicados um coeficiente ao limite do custo para deixar este limite em quatro horas e trinta minutos, ficando com um custo máximo dentro do objetivo a se alcançar com este experimento.

A Tabela 3.9 contém todos os resultados dos experimentos com a instância *VRP* Rio Claro 20. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* o custo da solução encontrada para o veículo com a menor rota; *(iii)* e o custo da solução encontrada para o veículo com a maior rota.

Como pudemos observar na Tabela 3.9 com três veículos conseguimos deixar os custos de cada veículo próximo a quatro horas, com isso seria possível que cada veículo fizesse todas as entregas em um dia de trabalho e ainda sobraria tempo para mais entregas.

A estratégia que chegou na melhor solução que encontramos foi a *GUIDED_LOCAL_SEARCH* executada durante 20 segundos. Esta é a melhor solução pois é a que melhor utilizou os três veículos, deixando uma pequena diferença do maior custo

Tabela 3.9: Resultados dos experimentos com a instância *VRP* Rio Claro 20 com três veículos.

Estratégias	Tempo de Execução	Menor Custo (h)	Maior Custo (h)
PATH_CHEAPEST_ARC	18 segundos	3,767937	3,958297
GLOBAL_CHEAPEST_ARC	32 segundos	3,767937	4,091346
GUIDED_LOCAL_SEARCH	20 segundos	3,767937	3,956420
GUIDED_LOCAL_SEARCH	30 segundos	3,767937	3,956420
GUIDED_LOCAL_SEARCH	60 segundos	3,767937	3,956420

para o menor custo. Podemos ver na Figura 3.23 a rota desta solução e podemos ver na Figura 3.24 o log desta execução.

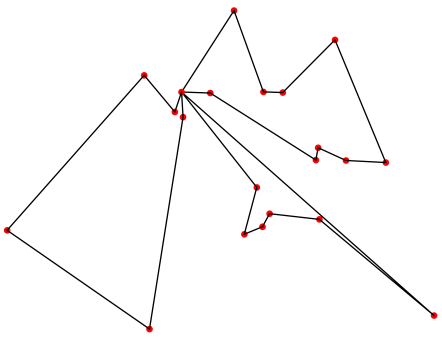

Figura 3.23: Figura com a solução da instância *VRP* Rio Claro 20 utilizando a estratégia *GUIDED_LOCAL_SEARCH*.

```
2023-11-18 02:38:46,280 - root - INFO - Start to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-18 02:39:06,280 - root - INFO - End to solve problem with GUIDED_LOCAL_SEARCH strategy
2023-11-18 02:39:06,280 - root - INFO - Solution achieved by GUIDED_LOCAL_SEARCH strategy with time limit on 20 seconds
2023-11-18 02:39:06,280 - root - INFO - Max Allowed Route: 4.499957083333333  Hours
2023-11-18 02:39:06,280 - root - INFO - Max Possible Route: 5.999942777777778  Hours
2023-11-18 02:39:06,280 - root - INFO - Route for vehicle 0:
 1 -> 11 -> 3 -> 12 -> 16 -> 18 -> 17 -> 13 -> 21 -> 5 -> 1
Distance of the route: 3.7726261111111112 Hours

2023-11-18 02:39:06,281 - root - INFO - Route for vehicle 1:
 1 -> 9 -> 8 -> 7 -> 15 -> 14 -> 20 -> 1
Distance of the route: 3.9564208333333335 Hours

2023-11-18 02:39:06,281 - root - INFO - Route for vehicle 2:
 1 -> 4 -> 2 -> 19 -> 6 -> 10 -> 1
Distance of the route: 3.7679377777777776 Hours

2023-11-18 02:39:06,281 - root - INFO - Maximum of the route distances: 3.9564208333333335 Hours
```

Figura 3.24: *Log* da solução da instância *VRP* Rio Claro 20 utilizando a estratégia *GUIDED_LOCAL_SEARCH*.

3.3.2 Instância *VRP* Rio Claro 200

A instância Rio Claro *VRP* 200 contém duzentos e um pontos a serem visitados por oito veículos. O objetivo é fazer com que as rotas de cada veículo tenham custo máximo de seis horas, que é um tempo aceitável para uma pessoa realizar as entregas durante um dia de trabalho. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado, todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos e também nos dá um limite de tempo, ou seja um limite do custo para cada veículo. Mas em todos os testes foram aplicados um coeficiente ao limite do custo para deixar este limite em seis horas, que é o objetivo a se alcançar com este experimeto.

A Tabela 3.10 contém todos os resultados dos experimentos com a instância *VRP* Rio Claro 200. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* o custo da solução encontrada para o veículo com a menor rota; *(iii)* e o custo da solução encontrada para o veículo com a maior rota.

Tabela 3.10: Resultados dos experimentos com a instância *VRP* Rio Claro 200 com oito veículos.

Estratégias	Tempo de Execução	Menor Custo (h)	Maior Custo (h)
PATH_CHEAPEST_ARC	2 minutos	3,732790	4,360771
GLOBAL_CHEAPEST_ARC	9 minutos	3,672291	4,341368
GUIDED_LOCAL_SEARCH	1 minuto	3,761809	4,797021
GUIDED_LOCAL_SEARCH	10 minutos	4,220505	4,275276
GUIDED_LOCAL_SEARCH	30 minutos	3,511645	4,274291
GUIDED_LOCAL_SEARCH	1 hora	4,108852	4,22524
GUIDED_LOCAL_SEARCH	2 horas	4,120195	4,208580
GUIDED_LOCAL_SEARCH	4 horas	4,120711	4,196206

Como pudemos observar na Tabela 3.10 com oito veículos conseguimos deixar os custos de cada veículo próximo a quatro horas, com isso seria possível que cada veículo fizesse todas as entregas em um dia de trabalho e ainda sobraria tempo para mais entregas.

A estratégia que chegou na melhor solução que encontramos foi a *GUIDED_LOCAL_SEARCH* executada durante 4 horas. Esta é a melhor solução pois é a que melhor utilizou os oito veículos, deixando uma pequena diferença do maior custo para o menor custo. Podemos ver na Figura 3.25 a rota desta solução.

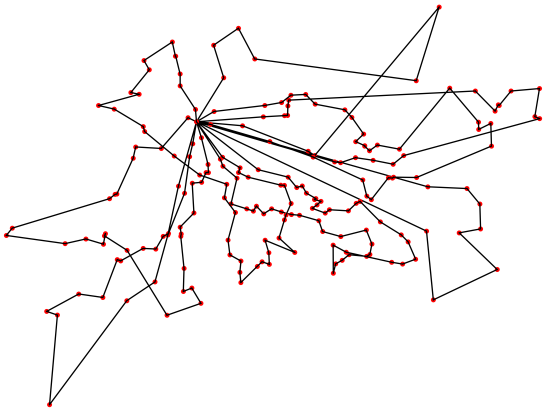

Figura 3.25: Figura com a solução da instância *VRP* Rio Claro 200 utilizando a estratégia *GUIDED_LOCAL_SEARCH*.

3.3.3 Instância *VRP* Rio Claro 2000

A instância Rio Claro *VRP* 2000 contém dois mil e um pontos a serem visitados por vinte veículos. O objetivo é fazer com que as rotas de cada veículo tenham custo máximo de seis horas, que é um tempo aceitável para uma pessoa realizar as entregas durante um dia de trabalho. A instância nos provê as coordenadas cartesianas de cada ponto a ser visitado, todas as distâncias entre cada ponto, com as distâncias dadas em milisegundos e também nos dá um limite de tempo, ou seja um limite do custo para cada veículo. Mas em todos os testes foram aplicados um coeficiente ao limite do custo para deixar este limite em seis horas, que é o objetivo a se alcançar com este experimeto.

A Tabela 3.11 contém todos os resultados dos experimentos com a instância *VRP* Rio Claro 2000, porém as estratégias de *PATH_CHEAPEST_ARC* e *GLOBAL_CHEAPEST_ARC* não chegaram a nenhuma solução com as mesmas restrições impostas na execução da estratégia *GUIDED_LOCAL_SEARCH*. Estes resultados serão compostos: *(i)* pelo tempo que o Google *OR-Tools* demorou para encontrar a solução; *(ii)* o custo da solução encontrada para o veículo com a menor rota; *(iii)* e o custo da solução encontrada para o veículo com a maior rota.

Como pudemos observar na Tabela 3.11 com vinte veículos conseguimos deixar os custos de cada veículo próximo a 4 horas na melhor solução, com isso seria possível que cada veículo fizesse todas as entregas em um dia de trabalho e ainda sobraria tempo para mais entregas. E vale comentar que o menor custo da solução encontrada durante a execução da estratégia

Tabela 3.11: Resultados dos experimentos com a instância *VRP* Rio Claro 2000 com vinte veículos.

Estratégias	Tempo de Execução	Menor Custo (h)	Maior Custo (h)
GUIDED_LOCAL_SEARCH	1 minuto	0	5,999850
GUIDED_LOCAL_SEARCH	10 minutos	1,506358	5,992518
GUIDED_LOCAL_SEARCH	30 minutos	3,282786	5,778851
GUIDED_LOCAL_SEARCH	1 hora	2,938316	5,766971
GUIDED_LOCAL_SEARCH	2 horas	2,667207	5,758034
GUIDED_LOCAL_SEARCH	4 horas	2,482388	5,609237
GUIDED_LOCAL_SEARCH	12 horas	3,544281	5,18562
GUIDED_LOCAL_SEARCH	24 horas	3,820106	5,066823
GUIDED_LOCAL_SEARCH	36 horas	3,882942	4,952057
GUIDED_LOCAL_SEARCH	72 horas	4,197423	4,773487

GUIDED_LOCAL_SEARCH por um minuto foi zero pois um dos veículos não foi utilizado, deixando-o parado no *depot*.

A estratégia que chegou na melhor solução que encontramos foi a *GUIDED_LOCAL_SEARCH* executada durante 72 horas. Esta é a melhor solução pois é a que melhor utilizou os vinte veículos, deixando uma pequena diferença do maior custo para o menor custo. Podemos ver na Figura 3.26 a rota desta solução.

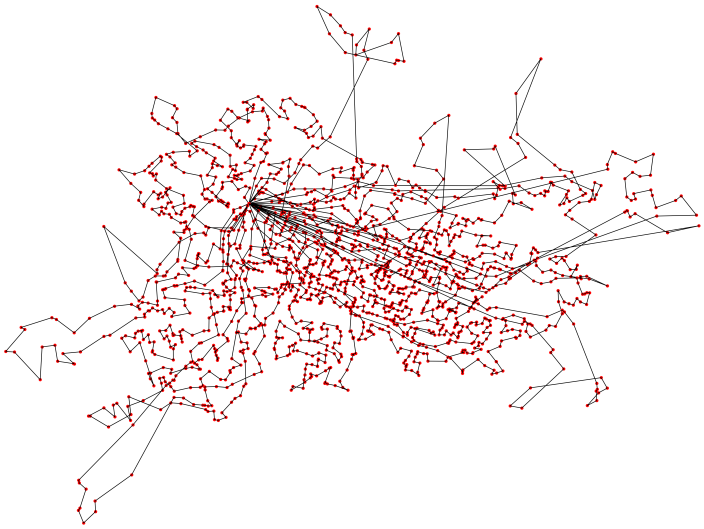

Figura 3.26: Figura com a solução da instância *VRP* Rio Claro 2000 utilizando a estratégia *GUIDED_LOCAL_SEARCH* por 72 horas.

Capítulo 4

Conclusões

Neste trabalho realizamos um estudo da ferramenta Google *OR-Tools*. Apesar da ferramenta ser preparada para otimizar problemas clássicos e algumas variantes, sua utilização não é tão imediata. Por exemplo, a ferramenta possui diversas técnicas de otimização disponíveis, mas é necessário um certo conhecimento de otimização para poder ajustar os parametros necessários para se realizar uma boa otimização.

Utilizamos a linguagem Python para escrever os códigos necessários para poder executar as instâncias do *TSPLIB95* e do PostVRP no Google *OR-Tools*. Uma vez que o código ficou pronto, executamos experimentos computacionais para avaliar a qualidade da solução e o tempo utilizado na otimização frente a instâncias de tamanhos diversos.

Fizemos experimentos variando o algoritmo de otimização. Foram testados o *GLOBAL_CHEAPEST_ARC*, *PATH_CHEAPEST_ARC* e a *GUIDED_LOCAL_SEARCH*. Os melhores resultados vieram do algoritmo *GUIDED_LOCAL_SEARCH*, porém o tempo de execução deste algoritmo foi maior.

Da *TSPLIB95*, utilizamos e executamos a instância a280 e a pr2392 com um único veículo. Como são instâncias com o ótimo conhecido, pudemos avaliar o *gap*. Para a instância a280, o algoritmo *GUIDED_LOCAL_SEARCH* chegou na solução ótima em 750s. Na instância pr2392, deixamos o otimizador executar por 72h, mas a solução ótima não foi alcançada. O *gap* encontrado nesta execução foi de 1,8%.

Para o PostVRP, utilizamos instâncias com 21, 201 e 2001 pontos nas cidades de Artur Nogueira e Rio Claro, sendo que um dos pontos é o depósito. Trabalhamos incialmente com um único veículo, ou seja, utilizando um *TSP*. Para Artur Nogueira, o custo da rota ficou em 3,11h, 8,69h e 24,31h para instâncias de tamanho 21, 201 e 2001 respectivamente. Isso já mostra

que um único carteiro conseguiria realizar as 20 entregas na cidade. Para 200 entregas, um carteiro não é sufiente. Para duas mil entregas, precisamos de mais de 6 carteiros, supondo uma jornada de 6h para cada carteiro.

Já para Rio Claro e um único veículo, as instâncias com 21, 201 e 2001 pontos tiveram custo 9,05h, 23,21h e 69,08h, respectivamente. Estas entregas são geradas de maneira probabilística e espalhadas por toda a cidade. Os custos de Rio Claro são maiores que os custos de Artur Nogueira, pois a cidade é bem maior. Com estes custos, podemos concluir que o limitante inferior do número de carteiros é 2, 4 e 12, considerando uma jornada de 6h para cada carteiro.

Após estas otimizações com um único veículo, foram feitas otimizações com uma frota de veículos. Note que cada carteiro é considerado como um veículo no modelo utilizado. Para Rio Claro, foram definidos 3, 8 e 20 carteiros para as instâncias de tamanho 20, 200, e 2000 respectivamente. Os tempos de execução do algoritmo foram 20s, 12h e 72h para as instâncias de tamanho 20, 200, e 2000 respectivamente.

A rotas geradas pelo O Google *OR-Tools* tiveram tamanho máximo 3,95h, 4,19h e 4,77h para as instâncias de tamanho 20, 200 e 2000 respetivamente. Tais rotas dizem respeito à cidade de Rio Claro.

Este trabalho focou em instâncias para *TSP* e *VRP*, mas o Google *OR-Tools* tem muitas outas funcionalidades que não foram exploradas neste trabalho, como Otimização Linear e Otimização Inteira. O Google *OR-Tools* também possui outras variantes do *VRP* que podem ser exploradas em trabalhos futuros.

Referências bibliográficas

COLVILE, R.; HUTCHINSON, E. J.; MINDELL, J.; WARREN, R. The transport sector as a source of air pollution. **Atmospheric environment**, Elsevier, v. 35, n. 9, p. 1537–1565, 2001.

DEWI, S. K.; UTAMA, D. M. A new hybrid whale optimization algorithm for green vehicle routing problem. **Systems Science & Control Engineering**, Taylor & Francis, v. 9, n. 1, p. 61–72, 2021.

ENAMI, L. M. et al. O Problema do caixeiro viajante através de algoritmo genético. **IX Congresso Brasileiro de Engenharia de Produção**, 2017.

GOOGLE. **OR-Tools**. [S.l.: s.n.], 2023. Disponível em: <https://developers.google.com/optimization>. Acesso em: 18 set. 2023.

HEIDELBERG, G. R. .-. U. TSPLIB95, 2023. Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf>. Acesso em: 2 out. 2023.

MEIRA, L. A.; MARTINS, P. S.; MENZORI, M.; ZENI, G. A. How to assess your Smart Delivery System? **Smart Delivery Systems: Solving Complex Vehicle Routing Problems**, Elsevier, p. 227, 2019.

NÉIA, S. N. d. A. S.; ARTERO, A. O.; CUNHA, C. B. da. Avaliação de roteiros de veículos usando técnicas de geometria computacional. **XLIX Simpósio Brasileiro de Pesquisa Operacional**, 2017.

O'CONNOR, K. Transport and Globalization. **International Encyclopedia of Human Geography**, Elsevier, p. 424–428, jan. 2009. DOI: 10.1016/B978-008044910-4.01032-4.

WORLD ECONOMIC FORUM. **Why green transport is vital for meeting global climate targets**. [S.l.: s.n.], jun. 2022. Disponível em: <https://www.weforum.org/agenda/2022/06/green-transport-and-cleaner-mobility-are-key-to-meeting-climate-goals/>. Acesso em: 17 out. 2022.