Otimização da Escala Semanal de Atendimento das Lojas Renner: Solução de um Problema de Roteamento de Veículos com Janelas de Tempo, Descanso, Turnos e Frota Heterogênea através da Heurística do Vizinho Mais Próximo

Gabriel Telles Missailidis
Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 - Vila das Acácias, São José dos Campos - SP
gabriel.missailidis@ga.ita.br

Rafael Silva de Oliveira
Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 - Vila das Acácias, São José dos Campos - SP
rafael.rsoliveira@ga.ita.br

Samir Nunes da Silva
Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 - Vila das Acácias, São José dos Campos - SP
samir.silva@ga.ita.br

RESUMO

Este artigo faz um estudo de caso de um Problema de Roteamento de Veículos para uma varejista vestuária brasileira, entre seu Centro de Distribuição (CD) e 133 lojas no estado de São Paulo. São consideradas restrições de janelas de tempo para atendimento, dois tipos de veículos com custos fixos e variáveis distintos, capacidade máxima de expedição por turno de trabalho no CD e tempo de descanso obrigatório entre viagens de um mesmo veículo. Buscou-se uma solução ótima no formato de uma escala de turnos com as rotas de cada veículo, abastecendo todas as lojas semanalmente. Abordou-se o problema dos pontos de vista de Programação Linear e de Heurísticas, através de um algoritmo guloso. O algoritmo heurístico utilizado conseguiu uma otimização de custo em relação à solução trivial, apesar de ainda serem possíveis maiores otimizações, as quais também foram discutidas no artigo.

PALAVRAS CHAVE. Problema de Roteamento de Veículos, Programação Linear, Heurística.

Problema de Roteamento de Veículos (VRP)

ABSTRACT

This paper studies a Vehicle Routing Problem case of a Brazilian clothing retailer, between their Distribution Center (DC) and 133 stores across the state of São Paulo. We considered Time Windows restrictions, as well as different vehicle types, with different fixed and variable costs, and also a maximum expedition capacity per work shift and mandatory vehicle rest between trips. We sought an optimized solution in the form of a weekly schedule of routes that supply each store. Linear Programming and Heuristics approaches have been taken, with the later through a greedy algorithm, which got a cheaper solution than the trivial one, although further work could have been done, seeking better optimizations, which were discussed in the paper.

KEYWORDS. Vehicle Routing Problem, Linear Programming, Heuristics

Vehicle Routing Problem (VRP)

1. Introdução

1.1. Apresentação do Case

O presente *case* de otimização foi proposto pelas Lojas Renner S.A. aos alunos da disciplina de PO-202 - Programação Linear, ministrada como disciplina eletiva no ITA (Instituto Tecnológico de Aeronáutica), em parceria com a Unifesp (Universidade Federal de São Paulo), e se trata do clássico *Vehicle Routing Problem* (Problema de roteamento de veículos) com a adição de *Time Windows* (janelas de tempo), tempos de descanso dos veículos, dias da semana, restrições envolvendo capacidades de expedição de turnos e dois tipos distintos de veículos: tocos e carretas, que possuem custos fixos e variáveis distintos, bem como capacidades e tempos de carregamento e descarregamento diferentes. As características de cada tipo de veículo são mostradas na Figura 1. Além disso, uma breve descrição do problema e a série principal de restrições associadas foram cedidas pela Renner na apresentação do *case* e são mostradas, respectivamente, nas Figuras 2 e 3.

Criação de dicionários para acesso mais fácil

```
In [ ]: custo_km = {'TOCO': 1.6, 'CARRETA': 2.4} # em reais/km
        capacidade_max = {'TOCO': 5000, 'CARRETA': 10000} # em unidades
        custos_fixos = {'TOCO': 100, 'CARRETA': 150} # em reais/km
        velocidades = {'TOCO': 50000, 'CARRETA': 50000} # em metros/hora
        carregamento = {'TOCO': 0.5, 'CARRETA': 1} # em horas
        descarregamento = {'TOCO': 0.5, 'CARRETA': 1} # em horas
```

Figura 1: Dicionários com as características de cada tipo de veículo.

Case de Otimização

- O objetivo é do case é definir a escala semanal de atendimento das lojas.

Contexto

- O case apresenta um conjunto de lojas localizadas no Estado de SP que possuem uma demanda semanal de peças que deve ser atendida pelo centro de distribuição (CD). Cada loja possui uma janela em que pode receber as peças e também existe um limite mínimo e máximo de peças que cada loja pode receber em cada entrega.
- O CD fica localizado na cidade de Arujá e funciona em três turnos de 8 horas por dia, totalizando 18 turnos por semana. Cada turno possui uma capacidade máxima de expedição de peças.
- Existem dois tipos de veículos que podem realizar o transporte das peças até as lojas. Não há um limite no número de veículos disponíveis e cada veículo pode transportar peças de mais de uma loja em uma viagem. Entretanto, a cada 12 horas percorridas, o veículo deve permanecer parado por 12 horas para descanso.
- O atendimento da demanda da loja pode ser quebrada em mais de uma entrega durante a semana.
- Dessa forma, a resposta esperada deverá dizer quais lojas estão em cada rota, e qual é a frequência semanal da rota. Por exemplo, a rota do veículo 1, uma carreta, acontece às segundas, quartas e sextas (TT1, TT7 e TT13), e atende as lojas 88, 92, 94 e 95, sendo que às segundas-feiras a rota é 88 - 92 - 94 - 95, às quartas-feiras 92 - 94 - 95 e às sextas-feiras 88 - 92 - 94.

Figura 2: Descrição do *case* dada pela Renner.

Tipo	Restrição	Descrição
CD	Capacidade de Expedição	O CD possui uma capacidade máxima de expedição de itens em cada turno.
Lojas	Janela de Atendimento	Cada loja possui uma janela disponível em que pode receber os veículos com os itens demandados.
	Capacidade de Recebimento	Cada loja possui um máximo e um mínimo de itens que podem receber em cada entrega.
	Demanda Semanal	Cada loja possui uma demanda de itens que deve ser atendida no decorrer de uma semana.
	Rota	Cada loja só pode estar em uma rota. Entretanto, o atendimento da loja não precisa ser na mesma frequência da rota
Veículos	Capacidade de Ocupação	Cada tipo de veículo possui uma capacidade máxima de ocupação, e pode transportar carga para atender mais de uma loja.
	Descanso	A cada 12 horas percorridas, o veículo deve permanecer parado por 12 horas para descanso.
	Custos	Cada tipo de veículo possui um custo fixo por dia de viagem, e um custo variável aplicado à quilometragem percorrida.
	Tempo de Carregamento/Descarregamento	Cada tipo de veículo possui um tempo fixo de carregamento e descarregamento que pode ser adicionado ao tempo de rota independentemente do número de peças transportado

Figura 3: Restrições principais do problema de programação linear proposto pela Renner.

Para o projeto construído com base nesse *case*, foi criado o repositório *case_renner* contendo o arquivo *notebook.ipynb* com a modelagem do problema, bem como todos os outros arquivos pertinentes, como os dados, as imagens e o artigo-base utilizados. Tal repositório encontra-se na página do GitHub `https://github.com/Samirnunes/case_renner`.

Inicialmente, propôs-se, com base em [Dell'Amico et al., 2007], em [Panggabean et al., 2018] e em [Onut et al., 2014], a implementação, através da biblioteca *PuLP* da linguagem *Python*, de restrições necessárias, mas ainda não suficientes, para a resolução do problema. No entanto, verificou-se que a solução do problema de programação linear com tais restrições não poderia ser resolvido em tempo hábil, o que já era previsto pelo artigo. Nesse sentido, propôs-se, ao invés dessa solução, uma que utiliza a heurística do vizinho mais próximo, que se baseia na procura, a partir de uma loja, daquela mais próxima que possa ser atendida por um dado veículo logo em seguida, conforme utilizado por [Yoshizaki e Belfiore, 2009] e por [Nugroho et al., 2020]. Tal solução foi ainda formada de outras heurísticas simplificadoras que compuseram um algoritmo que construiu as rotas de cada veículo, bem como definiu o tipo e a quantidade de cada veículo utilizado. Por fim, análises dos dados cedidos pelas Lojas Renner e adições manuais de veículos aos resultados foram realizadas para procurar e corrigir os casos que não eram englobados pelas heurísticas simplificadoras propostas.

Dividiu-se a estruturação do *case* em três partes principais: o pré-processamento dos dados, a tentativa de solução do case através de programação linear e a solução heurística, que de fato gerou um resultado final. Tais partes serão detalhadas mais adiante.

2. Pré-processamento dos Dados

Conforme a Figura 4, primeiramente importou-se as bibliotecas utilizadas para o pré-processamento dos dados e para a modelagem e resolução do problema. Em seguida, utilizou-se a biblioteca *Pandas* da linguagem *Python* para ler cada uma das abas da planilha de dados, de tal forma a se extrair 5 *dataframes* que foram alterados e utilizados conforme necessário.

Bibliotecas

Instalando e importando as bibliotecas necessárias

geopy: para o cálculo de distância entre coordenadas.

pandas: para criação e manipulação de dataframes.

pulp: para modelar e resolver problemas de programação linear.

numpy: para utilizar em conjunto com os dataframes do Pandas.

matplotlib: para plot de figuras.

datetime: para manipulação de datas.

math: biblioteca de funções matemáticas.

In [ ]:
```
!pip install geopy
!pip install pulp

import pandas as pd
from pulp import *
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import image as mpimg
import geopy.distance
import datetime
import math
```

Figura 4: Importação das bibliotecas e leitura da planilha por meio da biblioteca *Pandas*.

Logo após isso, como mostra a Figura 5, juntou-se os *dataframes* do centro de distribuição (CD) e das lojas, de tal forma que o índice 0 do *dataframe* foi associado ao CD em Arujá, para facilitar a referência a ele durante a resolução, já que é o ponto de partida de todos os veículos entregadores.

Unindo os dataframes do CD (Centro de Distribuição) e das lojas

In [ ]:
```
df_lojas.head()
```

In [ ]:
```
df_CD
```

In [ ]:
```
cd_row = pd.DataFrame({'ID Loja': 0,
                       'Município':df_CD['Cidade'][0].upper(), 'UF':df_CD['UF'],
                      'Latitude':df_CD['Latitude'][0], 'Longitude':df_CD['Longitude'],
                      'SEG': 'x','TER': 'x','QUA': 'x','QUI': 'x','SEX': 'x',
                      'SAB': 'x', 'Hora Recebimento Inicial': datetime.time(0, 0),
                      'Hora Finalização Recebimento': datetime.time(23, 59),
                      'Min. Peças p/ Entrega': 0,
                      'Máx. Peças p/ Entrega': 0,
                      })
```

In [ ]:
```
df_lojas = pd.concat([cd_row,df_lojas.loc[:]]).reset_index(drop = True)
```

In [ ]:
```
df_lojas.head(5)
```

Figura 5: Junção dos *dataframes* do CD e das lojas através da biblioteca *Pandas*.

Calculou-se, então, as distâncias entre as coordenadas de cada uma das lojas, incluindo o CD, de tal forma a se gerar uma matriz de distâncias. Utilizou-se, para tal, a biblioteca *Geopy*, conforme mostra a Figura 6. Em seguida, limpou-se o *dataframe* das lojas de forma a deixá-lo apenas com as informações que seriam extraídas dele para a criação das variáveis do problema futuramente, o que é também mostrado na Figura 6.

Obtendo a matriz de distâncias entre as coordenadas

```
In [ ]: def calculate_distance(coords_1: np.float64, coords_2: np.float64) -> float:
          return float(geopy.distance.geodesic(coords_1, coords_2).m)

In [ ]: def generate_distance_matrix(df: pd.core.frame.DataFrame) -> np.ndarray:
          distance_matrix = np.zeros((len(df), len(df)))
          for i in range(len(df)):
            for j in range(len(df)):
              coords_1 = (df['Latitude'].iloc[i], df['Longitude'].iloc[i])
              coords_2 = (df['Latitude'].iloc[j], df['Longitude'].iloc[j])
              distance_matrix[i][j] = calculate_distance(coords_1, coords_2)
          return distance_matrix

In [ ]: dmatrix = generate_distance_matrix(df_lojas)
```

Limpando o dataframe das lojas para deixá-lo com apenas as informações que serão usadas adiante

```
In [ ]: df_lojas.drop(['Latitude', 'Longitude'], axis = 1, inplace = True)
```

Figura 6: Cálculo das distâncias entre as lojas e limpeza do *dataframe* das lojas.

Por fim, modificou-se alguns valores do *dataframe* das lojas de forma a adequá-los a um padrão que pudesse ser utilizado ao longo do projeto. Tais modificações são facilmente visualizadas na Figura 7.

Trocando 'x' por seu equivalente booleano na escala de atendimentos das lojas

```
In [ ]: # Para df_lojas, onde tem X ou x, singnifica que a Loja funciona. Logo, será substituído por 1. Em contrapartida, NaN será 0, poi
        df_lojas_turnos = ['SEG', 'TER', 'QUA', 'QUI', 'SEX', 'SAB']

        df_lojas[df_lojas_turnos] = df_lojas[df_lojas_turnos].replace('X', 1)
        df_lojas[df_lojas_turnos] = df_lojas[df_lojas_turnos].replace('x', 1)

        df_lojas[df_lojas_turnos] = df_lojas[df_lojas_turnos].replace(np.nan, 0)
```

Tranformando as horas em float e transformando os '0's das horas de finalização em '24's, para criar uma escala de tempo adequada.

```
In [ ]: df_lojas['Hora Recebimento Inicial'] = df_lojas['Hora Recebimento Inicial'].apply(lambda x: x.hour + x.minute/60)
        df_lojas['Hora Finalização Recebimento'] = df_lojas['Hora Finalização Recebimento'].apply(lambda x: x.hour + x.minute/60)
        df_lojas['Hora Finalização Recebimento'] = df_lojas['Hora Finalização Recebimento'].apply(lambda x: float(24) if x == 0 else x)
```

Figura 7: Adequação de dados do *dataframe* das lojas.

3. Resolução do Case

3.1. Tentativa de Solução Via Programação Linear

Após o pré-processamento, definiu-se variáveis como apresentadas em [Dell'Amico et al., 2007], de tal forma a resolver um problema mais simples envolvendo *Time Windows* antes de se adicionar as restrições extras. Tais variáveis são definidas no código disponibilizado no arquivo *notebook.ipynb* e são resumidas a seguir:

- K é o conjunto dos 266 veículos que seriam utilizados no caso da solução trivial do problema, isto é, caso fosse utilizado 1 veículo por ponto: 133 veículos do tipo "toco"(ocupando os índices de 0 a 132) e o mesmo número do tipo "carreta"(índices de 133 a 265), totalizando uma lista com 266 índices.

- Q é a lista com as capacidades de cada veículos em número de itens, ou seja, 5000 para os tocos e 10000 para as carretas.
- c é o custo por quilômetro dos veículos, em reais: 1,6 para tocos e 2,4 para carretas.
- F representa os custos fixos de cada veículo (isto é, o custo por dia só por sair da garagem) em reais: 100 para tocos e 150 para carretas.
- v são as velocidades em metros por hora (para facilitar operações com a matriz de distâncias, inteiramente em metros): 50000 para ambos os tipos de veículos.
- s armazena os tempos de serviço de cada veículo, diferente do $paper$, que associa o tempo aos clientes.
- Os caminhos, isto é, qualquer dupla de pontos (*i, j*) foram definidas como A.
- As rotas, isto é, a matriz tridimensional com 133 linhas, 133 colunas e profundidade 266, é a variável de resolução x_{ijk}, que é binária. Ou seja: caso o veículo k percorra a rota (*i, j*), o elemento (*i, j, k*) da matriz receberá 1. Caso contrário, 0.
- Outra variável binária criada foi y_{ik}, que define se o cliente i foi servido (recebendo 1) ou não (recebendo 0) pelo veículo k.
- Além disso, a variável binária z_k define se o veículo k foi ou não utilizado.
- As variáveis t e τ indicam, respectivamente, o mínimo instante no qual o veículo k pode chegar num nó i e o serviço em tal nó pode começar.
- Finalmente, a variável π é o instante de tempo no qual o veículo pode começar sua rota.

Propõe-se a resolução do problema em termos de tempo, como em [Dell'Amico et al., 2007], e não em termos de custo. Conhecido o tempo, será necessário apenas utilizar relações lineares para obter o custo mínimo da operação. Para isso, trabalhou-se com as variáveis transformando-se suas unidades para tempo e, nesse sentido, definiu-se: $\tilde{F}$ é o custo fixo em termos de tempo, obtido dividindo-se o custo fixo F pelo custo por quilômetro, e depois dividindo pela velocidade, de forma a obter o tempo equivalente de viagem que custe o mesmo que o custo fixo.

Notou-se que o tempo de resolução do problema é muito alto utilizando-se a biblioteca PuLP, passando da ordem de horas de processamento. Isso ocorreu devido ao fato da ordem de complexidade do problema ser extremamente elevada, o que pode ser notado através da grande quantidade de variáveis que compõem o problema, juntamente de 134 localidades (CD e lojas) que deveriam ser resolvidas simultaneamente aos veículos. De fato, o artigo-base utilizado indica que os tempos para resolução do problema proposto são altíssimos devido à ordem de complexidade da resolução do problema.

Portanto, será preciso utilizar outros métodos de resolução para o problema, particularmente a resolução através do uso de heurísticas, tornando o procedimento uma abordagem gulosa perante o problema mas de simples complexidade computacional

3.2. Solução Heurística

3.2.1. Obtenção da Solução

Em particular, escolheu-se trabalhar com a heurística do vizinho mais próximo juntamente de outras heurísticas simplificadoras. A heurística do vizinho mais próximo se resume a três passos:

1. Parte-se do depósito com um novo veículo até a cidade mais próxima.

2. Calcula-se a cidade mais próxima da última cidade inserida na rota e verifica-se se é possível atender sua demanada.

3. Se for possível atender a demanda dessa cidade, adiciona-se ela à rota. Do contrário, retorna-se o veículo ao depósito e volta-se ao passo 1.

Ademais, as outras heurísticas simplificadoras consideradas são listadas a seguir:

- Todos os veículos inicialmente são carretas, para que consigamos atender todas as lojas. Ao fim, será proposta (apesar de não realizada) uma otimização para trocar as carretas por tocos onde for possível.

- Todos os veículos atenderão as lojas logo que possível, isto é, nos turnos mais cedo. Caso não seja possível, atenderão em turnos mais tarde. Por conta disso, os turnos iniciais de cada dia serão preenchidos mais rapidamente.

- Os veículos partirão do Centro de Distribuição (CD) em turnos que estão localizados no mesmo dia em que as lojas podem ser atendidas. Caso não seja possível atender todas as lojas seguindo essa heurística, será realizada uma análise dos dados obtidos e um ajuste manual dos casos que não a respeitam.

Com o fim de implementar a heurística do vizinho mais próximo e aplicá-la ao problema, modelou-se a função $n\_nearest\_neighbors$, que acha o índice dos n vizinhos mais próximos da loja i e a distância mínima de cada um a essa loja, dada a matriz de distâncias calculada anteriormente. Em seguida, definiu-se algumas variáveis globais úteis para a resolução heurística, que permitiram o acesso mais fácil à capacidade máxima de recebimento de cada loja e aos seus horários de início e finalização de recebimento.

Para facilitar o acesso às características e valores mutáveis de cada veículo, definiu-se a classe *vehicle*, a partir da qual criou-se as classes filhas *toco* e *carreta*, cujas implementações podem ser vistas nas Figuras 8 e 9. A implementação da classe *vehicle*, por ser demasiado longa, não será mostrada na íntegra no presente artigo, mas pode ser vista no *notebook* disponível no repositório do projeto.

```
In [ ]: class toco(vehicle):
            m_cost = 0.0016
            max_capacity = 5000
            fixed_cost = 100
            velocity = 50000
            charge_time = 0.5
            discharge_time = 0.5

            def __init__(self, vehicle_index: int, routes, delivery_time, start_time = 0, current_load = max_capacity, current_cost = fi
                self.vehicle_index = vehicle_index
                self.current_load = current_load
                self.current_cost = current_cost
                self.current_work_time = current_work_time
                self.routes = routes
                self.start_time = start_time
                self.delivery_time = delivery_time

            def __str__(self):
                return f'Número do veículo: {self.vehicle_index}\nTipo do veículo: TOCO\nCarga atual: {self.current_load}\nCusto atual: {

            def get_type(self):
                return 'TOCO'
```

Figura 8: Classe filha *toco*, que herda da classe *vehicle*.

```
In [ ]: class carreta(vehicle):
            m_cost = 0.0024
            max_capacity = 10000
            fixed_cost = 150
            velocity = 50000
            charge_time = 1
            discharge_time = 1
            current_load = max_capacity
            current_cost = fixed_cost
            #{'MON': list(), 'TUE': list(), 'WED': list(), 'THU': list(), 'FRI': list(), 'SAT': list()}

            def __init__(self,vehicle_index: int, routes, delivery_time, start_time = 0, current_load = max_capacity, current_cost = fixe
                self.vehicle_index = vehicle_index
                self.current_load = current_load
                self.current_cost = current_cost
                self.current_work_time = current_work_time
                self.routes = routes
                self.start_time = start_time
                self.delivery_time = delivery_time

            def __str__(self):
                return f'Número do veículo: {self.vehicle_index}\nTipo do veículo: CARRETA\nCarga atual: {self.current_load}\nCusto atual

            def get_type(self):
                return 'CARRETA'
```

Figura 9: Classe filha *carreta*, que herda da classe *vehicle*.

Após isso, definiu-se as funções auxiliares get_next_store, $get\_n\_next\_stores$ e can_supply. A primeira retorna a loja mais próxima da loja i, com base na função $n\_nearest\_neighbors$, cuja demanda ainda não foi atendida e que pode ser atendida no dia em questão; a segunda, por sua vez, retorna as n lojas mais próxima da loja i nas condições da função anterior; e a terceira, por fim, retorna um booleano que indica se uma dada loja pode ser atendida pelo veículo v, saindo de uma loja em direção a uma outra. Suas implementações podem ser visualizadas nas Figuras 10 e 11.

```
In [ ]: def get_next_store(n: int, demanda: list, dmatrix: np.ndarray, day: str, stores_per_day: dict) -> int:
            '''Retorna a loja mais próxima da loja i cuja demanda ainda não foi atendida, e que pode ser atendida no dia em questão'''
            closest_stores = n_nearest_neighbors(len(dmatrix),n,dmatrix)
            for i in range(0,len(dmatrix)-1):
                if demanda[closest_stores[i][0]] != 0 and stores_per_day[day][i] != 0:
                    return closest_stores[i][0]
            return 0
```

```
In [ ]: def get_n_next_stores(n: int, i: int, demanda: list, dmatrix: np.ndarray, day: str, stores_per_day: dict) -> list:
            '''Retorna as n lojas mais próximas da loja i, que ainda não foram atendidas, e que podem ser atendidas no dia em questão'''
            stores = n_nearest_neighbors(len(dmatrix),i,dmatrix) # ordenadas por proximidade à loja i
            result = list()
            for j in range(0,len(dmatrix)-1):
                if demanda[stores[j][0]] != 0 and stores_per_day[day][j] != 0 and len(result) < n:
                    result.append(stores[j][0])
            return result
```

Figura 10: Funções auxiliares de localização das próximas lojas a serem atendidas.

```
In [ ]: def can_supply(current_store: int, next_store:int, v:vehicle, demanda:list, comeco:list, termino: list, cap_exp: int) -> bool:
            ''' Retorna se uma dada loja pode ser atendida pelo veículo v, saindo da current_store e indo à next_store.'''
            # A nova viagem não pode exceder a capacidade máxima de carga do veículo
            if demanda[next_store] > v.get_current_load():
                return False

            # o veículo não pode chegar antes da loja abrir
            if v.get_start_time() + v.get_current_work_time() + dmatrix[current_store][next_store]/ v.get_velocity() < comeco[next_store]
                return False

            # o veículo não pode chegar na loja após o horario de fechamento
            if v.get_start_time() + v.get_current_work_time() + dmatrix[current_store][next_store]/ v.get_velocity() > termino[next_store
                return False

            # o tempo de uso total do veículo não pode ultrapassar o limite de 12 horas (incluindo o retorno ao depósito?)
            if v.get_current_work_time() + dmatrix[current_store][next_store]/ v.get_velocity() + v.get_discharge_time() + dmatrix[next_s
                return False

            # A nova viagem não pode exceder a capacidade de expedição do turno
            if demanda[next_store] > cap_exp:
                return False

            return True
```

Figura 11: Função auxiliar que determina se uma loja consegue ser atendida por um dado veículo partindo de uma outra loja.

De posse dessas funções e subtraindo uma carga de 5000 unidades do único carregamento (loja 77) que não pode ser atendido com um único veículo, o qual foi descoberto via análise dos dados da planilha e que será tratado individualmente ao fim através do envio de um toco para a loja em questão, construiu-se um algoritmo guloso que permitiu a obtenção da solução parcial do problema, visto que nem todas as lojas puderam ser atendidas com as heurísticas propostas - e essas foram tratadas uma a uma após a obtenção e otimização do resultado. O algoritmo implementado pode ser visto na Figura 12 e permitiu, além da obtenção da escala semanal parcial de atendimento das lojas, o cálculo do custo dos transportes feitos.

```
# Algoritmo
for v in vehicles:
    for day in days:
        #Reset do veículo no dia
        v.set_current_load(10000)
        v.set_current_work_time(0)

        #Primeira loja do dia
        next_store = get_next_store(current_store, current_demand, dmatrix, day, stores_per_day)
        v.set_start_time(hora_recebimento_inicial[next_store] - dmatrix[current_store][next_store]/ v.get_velocity() - v.get_char
        while can_supply(current_store,next_store,v,current_demand, hora_recebimento_inicial, hora_finalizacao_recebimento, cap_e
            #Alterações do veículo
            v.add_store_to_route(day,next_store)
            v.reduce_load_while_discharging(current_demand[next_store])
            v.increase_cost(dmatrix,current_store,next_store)
            v.increase_work_time(dmatrix,current_store,next_store)
            v.save_delivery_time()

            #Alterações das lojas/CD
            cap_exp[vehicle.get_turns(day,v.get_start_time()) - 1] -= current_demand[next_store]
            current_demand[next_store] = 0.0
            #Continuação do ciclo
            current_store = next_store
            n_next_stores = get_n_next_stores(n,current_store, current_demand, dmatrix, day, stores_per_day)
            n_next_stores = ranked_by_cost(current_store,n_next_stores,v,dmatrix)
            next_store = get_next_store(current_store,current_demand,dmatrix,day,stores_per_day)
            for store in n_next_stores:
                if can_supply(current_store,store,v,current_demand, hora_recebimento_inicial, hora_finalizacao_recebimento, cap_e
                    next_store = store
                    break

        if len(v.routes[day]) > 0:
            #Finaliza rotas iniciadas, retornando-as para o CD
            v.end_route(dmatrix,current_store)
        v.set_current_load(0)
    custo_total += v.get_current_cost()
```

Figura 12: Algoritmo guloso utilizado para chegar na solução através das heurísticas propostas.

Em seguida, enviou-se um toco adicional para a loja 77 em horário apropriado e escreveu-se o resultado obtido até então de uma forma mais visual, através de um *dataframe* da biblioteca *Pandas*. Para tal, ainda traduziu-se os dias da semana para o português e transformou-se os horários no formato de horas e minutos.

Com o *dataframe* pronto, verificou-se se todas as lojas foram atendidas com as heurísticas propostas, e concluiu-se, de fato, que não. Dessa forma, buscou-se, através do teste de diferentes valores de n na função $get\_n\_next\_stores$ (já que cada n gera uma solução diferente, devido à forma com que o algoritmo foi construído), aquela solução que atende o maior número de lojas, descobrindo-se que $n = 1$ é o valor ótimo. De posse desse valor, procurou-se, então, as lojas não atendidas e descobriu-se que as lojas $113, 117, 121, 122, 123$ e 124 eram as únicas que não haviam sido atendidas e que eram justamente aquelas que eram muito distantes do centro de distribuição e que deveriam ser atendidas por turnos do dia anterior: situação essa que não era englobada pelas heurísticas simplificadoras. Notou-se, assim, que outras heurísticas mais complexas deveriam ser aplicadas para se conseguir atender todas as lojas. Porém, como o número de lojas não atendidas foi muito pequeno, buscou-se preencher manualmente esses valores baseando-se nas capacidades restantes dos turnos, nos horários disponíveis de atendimento das lojas e na otimização do transporte através do volume previsto (se esse valor fosse menor que 5000, seria enviado um toco, por exemplo).

Com todas as lojas atendidas, obteve-se, finalmente, a resposta final do problema, que pode ser visualizada no *dataframe* da Figura 13, e somou-se ao custo total os transportes que foram definidos manualmente. Nesse contexto, conseguiu-se um custo total de aproximadamente $R\$\ 56745,03$. Por fim, transferiu-se o *dataframe* para o formato de uma planilha de Excel através da biblioteca *Pandas*.

Finalmente, a resposta final do problema

```
In [96]: resultado_final
```

Out[96]:

	CD	LOJA	ROTA	DIA DA SEMANA	TURNO	PLAN. HORA	VOLUME PREVISTO	VEÍCULO
0	Arujá	129	1	SEG	2	8:00	3983.0	CARRETA
1	Arujá	74	2	SEG	3	20:00	9196.0	CARRETA
2	Arujá	76	2	TER	5	13:26	7510.0	CARRETA
3	Arujá	71	2	QUA	8	11:57	7821.0	CARRETA
4	Arujá	60	2	QUI	11	14:57	4041.0	CARRETA
...	...	...	...	...	...	...	...	...
129	Arujá	117	56	QUA	9	7:30	1012.0	TOCO
130	Arujá	121	56	QUI	9	8:00	3371.0	CARRETA
131	Arujá	122	56	QUI	9	10:00	2122.0	CARRETA
132	Arujá	123	57	QUI	9	9:00	9103.0	CARRETA
133	Arujá	124	58	SEX	12	7:00	5227.0	CARRETA

Figura 13: *Dataframe* contendo a resposta final do problema de roteamento de veículos das Lojas Renner.

3.2.2. Possíveis Melhorias da Solução

Futuras melhorias podem incorporar, entre outros aspectos, um algoritmo que faça não apenas uma busca em largura, mas também uma busca em profundidade (*depth search*), de modo a obter soluções potencialmente otimizadas, a um custo maior de tempo de computação. Assim, ao invés de apenas procurar para a próxima loja que vá minimizar o custo, olha-se para o próximo par de lojas que, em conjunto, minimiza o custo. Ou então, para o próximo trio, quarteto, quinteto, etc. Note que o número de combinações cresce extremamente rápido, o que torna inviável uma busca em profundidade muito grande. Ainda assim, técnicas como *alpha-beta pruning* poderiam ser utilizadas, visto que só faria sentido procurar por lojas próximas.

Outro ponto de melhoria seria atualizar o algoritmo utilizado para garantir que todas as lojas sejam atendidas, fazendo mudanças em rotas anteriores se necessário acomodar uma loja com horário mais restrito em algum turno cuja capacidade de expedição ja foi esgotada. Embora tal mudança seja fácil de ser feita manualmente, atualizar isso gera complexidades computacionais que fogem do escopo deste trabalho.

Por fim, uma série de pequenas mudanças podem ser implementadas, que embora façam pouca diferença individual, podem contribuir para uma solução mais otimizada, e até mesmo mais realista. Por exemplo, utilizar distâncias reais entre lojas, ou seja, a distância ao longo de ruas e estradas, e não apenas a distância entre as coordenadas. Também pode-se aprimorar a troca de carreta por toco quando possível, incorporando as mudanças de tempo de carregamento e descarregamento na busca de rotas otimizadas.

4. Conclusões

Neste trabalho foi encontrada uma solução melhor que a trivial (embora não totalmente otimizada) para a escala semanal de turnos e rotas de de entrega envolvendo 133 lojas e 1 Centro de Distribuição das Lojas Renner no estado de São Paulo, cada um com seu respectivo horário de atendimento, em diferentes dias da semana, com suas demandas de peças de roupa, sendo supridas por uma frota heterogênea de veículos, formada por veículos do tipo Toco e Carreta, os quais além de distintas cargas e custos, necessitavam ter um tempo obrigatório de descanso entre viagens. Tentou-se inicialmente uma abordagem com Programação Linear, mas devido à quantidade intrabalhável de variáveis no problema em questão, partiu-se para uma abordagem via Heurísticas, em especial com o desenvolvimento de um algoritmo guloso que fazia busca em largura a partir de uma dada rota, buscando minimizar o custo da dada rota. Por meio dessas heurísticas, determinou-se o custo semanal em 56 mil reais.

Agradecimentos

Gostaríamos de agradecer enormemente aos dois maiores possibilitadores deste projeto: ao docente da Unifesp prof. Dr. Luiz Leduino de Salles Neto, por ministrar a disciplina PO-202 - Programação Linear e nos introduzir ao mundo da pesquisa operacional; e à empresa Lojas Renner S.A., por nos propor a resolução do *case* de otimização de rotas que norteou este artigo e fornecer informações pertinentes para a resolução do problema.

Referências

Dell'Amico, M., Vigo, D., Monaci, M. Pagani, C. (2007). Heuristic Approaches for the Fleet Size and Mix Vehicle Routing Problem with Time Windows. *Transportation Science*, *41*(4), 516-526. https://www.doi.org/10.1287/trsc.1070.0190

Panggabean, E. M., Mawengkang, H., Azis, Z., Sari, R. F. (2018). Periodic heterogeneous vehicle routing problem with driver scheduling. *IOP Conference Series: Materials Science and Engineering*, *300*(1), https://www.doi.org/10.1088/1757-899X/300/1/012017

Onut, S., Kamber, M. R., Altay, G. (2014, March). A heterogeneous fleet vehicle routing model for solving the LPG distribution problem: A case study. *Journal of Physics: Conference Series 490*(1), 12043. https://www.doi.org/10.1088/1742-6596/490/1/012043

Nugroho, S. M., Nafisah, L., Khannan, M. S. A., Mastrisiswadi, H., Ramdhani, M. N. (2020, April). Vehicle routing problem with heterogeneous fleet, split delivery, multiple product, multiple trip, and time windows: A case study in fuel distribution. *IOP Conference Series: Materials Science and Engineering 847*(1), 12066. https://www.doi.org/10.1088/1757-899X/847/1/012066

Yoshizaki, H., Belfiore, P. (2009). Scatter search for a real-life heterogeneous fleet vehicle routing problem with time windows and split deliveries in Brazil. *European Journal of Operational Research*, *199*(3), 750-758. https://doi.org/10.1016/j.ejor.2008.08.003