Universidade Federal de São Carlos

Processo FAPESP #2020/06103-0

Heurísticas e Meta-Heurísticas para Problemas de Roteamento de Veículos

Relatório Final

Orientador: Prof. Dr. Mário César San Felice
Bolsista: Matheus Teixeira Mattioli

Agosto de 2020 a Julho de 2021
São Carlos - SP, Brasil

Sumário

1 Introdução

Em problemas de otimização combinatória existem grandes quantidades de escolhas a serem tomadas, que podem levar a muitas soluções distintas e o objetivo é alcançar uma solução cujo valor é mínimo ou máximo. Um exemplo deste tipo de problema é a tarefa de traçar rotas de custo mínimo para uma determinada frota de veículos. Este problema é conhecido como Problema do Roteamento de Veículos (Vehicle Routing Problem - VRP) e foi o objeto de estudos desta iniciação científica.

O VRP é uma generalização do Problema do Caixeiro Viajante (TSP), porém, o VRP apresenta mais complicações que o TSP, já que pode levar em consideração fatores como capacidade dos veículos da frota, emissão de poluentes, janelas de tempo no atendimento de clientes, frotas heterogêneas, entre outras variantes. Essas variantes permitem que nos aproximemos mais de problemas da vida real, oriundos da área de logística e de outras partes da indústria.

Segundo Vidal, Laporte e Matl [17] o VRP é um problema computacionalmente muito desafiador para métodos exatos e aproximativos. Parte disso pelo fato dele ser NP-difícil, ou seja, não poder ser resolvido de forma ótima em tempo polinomial a menos que P=NP. Por conta desta dificuldade, muitos pesquisadores que estudam o VRP passaram a dar um foco maior a heurísticas e meta-heurísticas, conforme indica Laporte [9].

Em decorrência disso, diversos algoritmos heurísticos e meta-heurísticos são conhecidos para ele. Por exemplo, a heurística construtiva Nearest Neighbour [7]; heurísticas de intensificação como a busca local, a qual pode ser trabalhada com diversas estruturas de vizinhança, como k-opt, swap e shift, conforme indica Munhoz et. al [13]. Com relação às meta-heurísticas, segundo Thibaut et. al [16], várias já foram testadas e obtiveram resultados interessantes, tais como, Greedy Randomized Adaptative Search Procedure (GRASP), Simulated Annealing, Tabu Search, Variable Neighbourhood Search (VNS) e Variable Neighbourhood Descent (VND).

Neste projeto também trabalhamos com a versão capacitada do problema de roteamento de veículos (CVRP), em que os veículos apresentam uma capacidade máxima de carga [2]. Também estudamos uma versão verde bi-objetiva do VRP, chamada de roteamento de veículos verde (GVRP), na qual devemos minimizar custos operacionais e emissões de poluentes [10].

Neste relatório, são apresentadas brevemente as atividades desenvolvidas na primeira etapa do projeto, de agosto de 2020 a janeiro de 2021, e apresentadas mais detalhadamente as atividades realizadas na segunda etapa, de fevereiro de 2021 a julho de 2021. Também são descritas as atividades propostas para as próximas etapas, que envolvem o pedido de extensão da bolsa. A Seção 2 contém a definição do VRP, bem como a descrição das variantes estudadas. Na Seção 3 são descritos os algoritmos estudados para abordar o TSP. A Seção 4 apresenta uma estratégia heurística de resolução do CVRP. A Seção 5 apresenta uma técnica de geração de fronteiras de pareto para o GVRP. Na Seção 6 são descritas as meta-heurísticas estudadas para o VRP. A Seção 7 descreve a implementação feita de alguns dos algoritmo estudados. Na Seção 8 são apresentados os resultados experimentais obtidos. A Seção 9 trata do pedido de renovação de bolsa e apresenta o planejamento para as próximas etapas do projeto. A Seção 10 corresponde ao cronograma atualizado do projeto. Por fim, a Seção 11 traz a conclusão.

2 Definição do Problema

O objetivo no VRP é formar rotas para que os veículos possam transportar as demandas, respeitando as restrições operacionais e buscando minimizar os custos envolvidos. No VRP, cada rota deve iniciar no depósito e terminar no mesmo, sendo cíclica. Como dito anteriormente, no CVRP temos de respeitar a capacidade dos veículos. A seguir apresentamos uma formulação em programação linear inteira (PLI) para o CVRP, que é baseada na formulação de Fisher e Jaikumar [3]. Consideramos um grafo não direcionado $G = (V, E)$ com n vértices, m arestas e h veículos. Esta formulação supõe que o depósito está no vértice com índice 1 e utiliza as seguintes varíaveis e constantes:

- x_{ijk} é uma variável binária que assume valor 1 quando o veículo k visita o cliente j imediatamente após visitar o cliente i, 0 caso contrário.
- y_{ik} é uma variável binária que assume valor 1 se o cliente i é visitado pelo veículo k, 0 caso contrário.
- q_i é a demanda do cliente i.
- Q_k é a capacidade do veículo k.
- c_{ij} é o custo de percorrer o trecho que vai do cliente i ao j.

O problema pode ser formulado como segue:

$$\min \sum_{i,j} \left(c_{ij} \sum_{k=1}^{h} x_{ijk} \right)$$

$$\text{s.a.} \sum_{k=1}^{h} y_{ik} = 1 \qquad i = 2, \ldots, n \qquad (1)$$

$$\sum_{k=1}^{h} y_{1k} = h \qquad (2)$$

$$\sum_{i=2}^{n} q_i y_{ik} \leqslant Q_k \qquad k = 1, \ldots, h \qquad (3)$$

$$\sum_{j \neq i} x_{ijk} = \sum_{j \neq i} x_{jik} = y_{ik} \qquad i = 2, \ldots, n \quad k = 1, \ldots, h \qquad (4)$$

$$\sum_{i,j \in S} x_{ijk} \leqslant |S| - 1 \qquad \forall S \subseteq \{2, \ldots, n\}, \quad k = 1, \ldots, h \qquad (5)$$

$$y_{ik} \in \{0, 1\} \qquad i = 1, \ldots, n \quad k = 1, \ldots, h \qquad (6)$$

$$x_{ijk} \in \{0, 1\} \qquad i, j = 1, \ldots, n \quad k = 1, \ldots, h \qquad (7)$$

A restrição (1) assegura que cada cliente é visitado por exatamente um veículo, a restrição (2) garante que todos os veículos visitam o depósito, a restrição (3) evita que a capacidade dos veículos seja ultrapassada, as restrições (4) garantem que os veículos não param suas rotas em um cliente e conectam as variáveis x e y, e a restrição (5) elimina subrotas. Para o PLI descrito acima se aplicar para o VRP basta removermos a restrição 3.

A variante bi-objetiva GVRP também foi estudada durante este projeto. Nela queremos construir rotas que minimizem tanto a distância percorrida, quanto a emissão de poluentes [6]. No caso, a emissão é representada por fatores de poluição atrelados a cada aresta do grafo.

3 Algoritmos para o TSP

Vale notar que podemos encontrar soluções para o VRP usando algoritmos para seu caso particular, o TSP, se considerarmos que podemos utilizar um veículo da frota para atender todas as demandas. Além disso, mesmo quando tratamos do CVRP ou outras variantes mais complexas, algoritmos para o TSP podem ser subrotinas importantes para a construção das soluções.

3.1 Algoritmo Guloso

Um algoritmo guloso trabalhado é baseado no *nearest neighbor* apresentado no artigo de Rosenkrantz et. al [14]. A ideia principal por trás dele é a inserção do vizinho mais próximo. O nosso algoritmo é adaptado para o VRP, portanto, insere na primeira posição da lista L o depósito d, a partir daí, sendo C o conjunto de clientes que ainda não foram atendidos, o algoritmo seleciona o cliente c mais próximo do último cliente em L, que chamamos l_{ultimo}, o remove do conjunto C e o insere no final de L. O algoritmo continua até que não restem clientes em C. Ao final, adicionamos uma aresta conectando o último cliente visitado e o depósito, obtendo uma lista que mostra a ordem de visitação de todos os clientes, partindo e terminando no depósito. O pseudocódigo desse algoritmo guloso é mostrado no Algoritmo 1.

Algoritmo 1 TSP-Guloso

Input: Um grafo conexo $G = (C, E)$
Output: L, a lista contendo os clientes na ordem a serem visitados

```
1: function TSP-Guloso(G = (C, E), w, d)
2:     L ← d
3:     while C ≠ ∅ do
4:         Encontre o cliente c ∈ C mais próximo de l_ultimo
5:         L ← L ∪ {c}
6:         Remova c do conjunto C
7:     L ← L ∪ {d}
8:     return L
```

3.2 Algoritmo Aleatorizado

Algoritmo para geração de circuitos aleatórios baseado no Knuth *Shuffle*[8]. A ideia é colocar todos os clientes em uma lista e gerar uma permutação aleatória dela. A partir disso inserimos duas arestas, que conectam o primeiro cliente da lista com o depósito e o

último cliente da lista com o depósito. Seu pseudocódigo é apresentando em Algoritmo 2.

Algoritmo 2 KnuthShuffle

Input: Um vetor V contendo todos os clientes C

Output: L, uma permutação aleatória do vetor original, contendo a ordem dos clientes a serem visitados, com o depósito no início e final da permutação

```
1: function KnuthShuffle(V)
2:     for i ← 1 to |C| do
3:         Escolha um índice j com probabilidade uniforme entre i e |C|
4:         Troque de posição o valor da posição i com o da posição j
5:     Insira o depósito no começo e final do vetor L
6:     return L
```

3.3 Busca Local

Antes de apresentar a Busca Local é muito importante definir o conceito de vizinhança de soluções ou soluções vizinhas: consideremos um grafo em que cada vértice é uma solução do problema e existe uma aresta entre dois vértices se as soluções correspondentes atendem a algum critério de similaridade. Entende-se por vizinhança o conjunto de vértices adjacentes ao vértice que está sendo analisado.

Podemos dizer que os algoritmos de busca local se baseiam em escolhas que obtêm melhorias locais sem garantia de atingir um ótimo global. Eles começam recebendo uma solução viável e, iterativamente, fazem mudanças pequenas nesta solução através da análise de soluções vizinhas, de modo a sempre transformá-la numa solução um pouco melhor. A escolha de solução vizinha segue algum critério, como *Best Fit* ou *First Fit*. No primeiro, a cada iteração escolhemos a solução vizinha que oferece a melhor melhoria possível. No segundo escolhemos a primeira solução encontrada que oferece melhoria. Como as melhorias promovidas a cada iteração podem ser muito pequenas, estes algoritmos não necessariamente terminam em um número polinomial de iterações. Por isso, em geral um nível mínimo de melhoria a cada iteração ou número máximo de iterações são adotados como critérios de parada. Nos Algoritmos 3 e 4 são apresentados pseudocódigos da busca local *Best Fit* e *First Fit*, respectivamente.

Algoritmo 3 LS-BestFit

Input: Solução inicial S

Output: Devolve um mínimo local S^*

```
1: S* ← S
2: while Critério de parada não atingido do
3:     Analise a vizinhança de S* e devolva seu melhor vizinho S'
4:     if S' for melhor que S* then
5:         S* ← S'
6: return S*
```

Algoritmo 4 LS-FirstFit

Input: Solução inicial S
Output: Retorna um mínimo local S^*
1: $S^* \leftarrow S$
2: **while** Critério de parada não atingido **do**
3: Analise a vizinhança de S^* até encontrar o primeiro vizinho S' melhor que S^*
4: $S* \leftarrow S'$
5: **return** S^*

3.3.1 2-OPT

Para a busca local funcionar, ela deve explorar uma estrutura de vizinhanças com o objetivo de encontrar a melhor solução vizinha, best fit, ou a primeira solução que melhora, first fit. Para isso, uma estrutura de vizinhança possível é a chamada de 2-OPT, na qual realizamos uma troca entre duas arestas selecionadas, explicitada pelas imagens 1 e 2

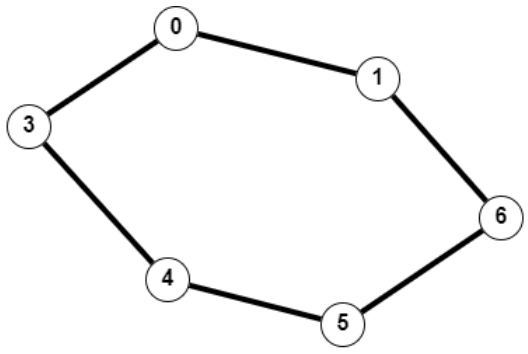

Figura 1: Ilustração de uma possível solução inicial para um VRP. O vértice 0 representa o depósito.

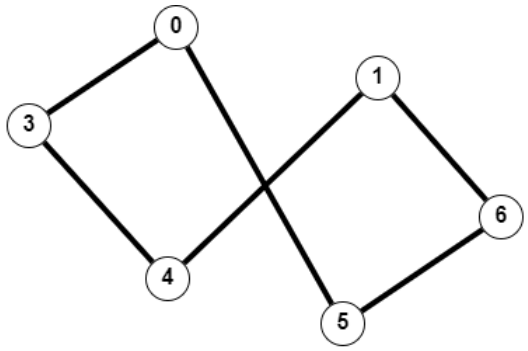

Figura 2: Um vizinho 2-OPT da solução inicial. Onde houve a troca das arestas E(0, 1) e E(4, 5) pelas arestas E(0, 5) e E(1, 4). O vértice 0 representa o depósito.

Todas as combinações possíveis desta troca entre duas arestas no circuito são realizadas por meio de um laço externo que seleciona uma aresta por iteração, e testamos as possibilidades de troca desta aresta com todas as outras $m-1$ arestas através de um laço

interno. O laço externo é executado por m iterações. Como, ao total, temos $m(m-1)/2$ combinações de arestas para testarmos na solução, temos este mesmo tanto de soluções vizinhas à serem analisadas. Portanto, o custo deste algoritmo é de $O(m^2)$, mas o grafo é completo, o que implica em uma complexidade $O(n^2)$, com n sendo o número de vértices.

3.3.2 3-OPT

A estrutura de vizinhança 3-OPT trabalha com a troca entre três arestas de uma solução inicial, as imagens 3, 4, 5 e 6 mostram todas as possiblidades de troca.

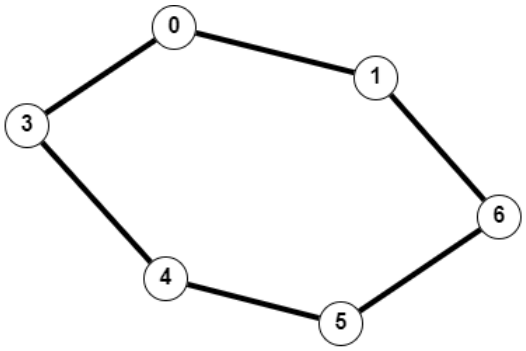

Figura 3: Ilustração de uma possível solução inicial para um VRP. O vértice 0 representa o depósito.

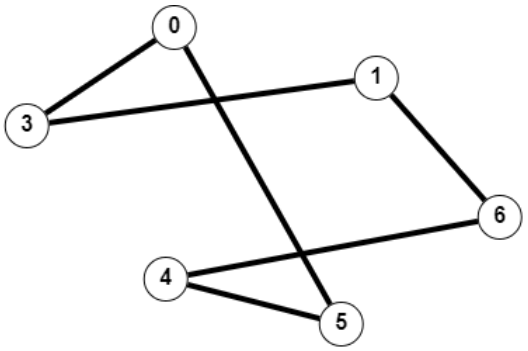

Figura 4: Um possível vizinho 3-OPT da solução inicial. Em que as arestas E(0, 1), E(6, 5) e E(4, 3), foram substituídas pelas arestas E(0,5), E(4, 6) e E(1, 3).

Todas as combinações possíveis da troca entre três arestas no circuito são realizadas por meio de três laços aninhados. O que configura uma complexidade de $O(m^3)$, porém o grafo é completo e podemos escrever em ordem do número de vértices, portanto $O(n^3)$.

Para essas estruturas de vizinhança k-OPT, com k sendo o número de arestas trocadas. Vale destacar que, conforme aumentamos o número de arestas fixadas para serem trocadas, cresce o número de vizinhos da estrutura de vizinhança e a probabilidade de encontrarmos uma solução vizinha que melhore bastante a qualidade da solução inicial. Contudo, com mais arestas analisadas, o tempo consumido em uma execução do algoritmo aumenta, pois a ordem da função de crescimento sobe de categoria, por exemplo da 2-OPT para a

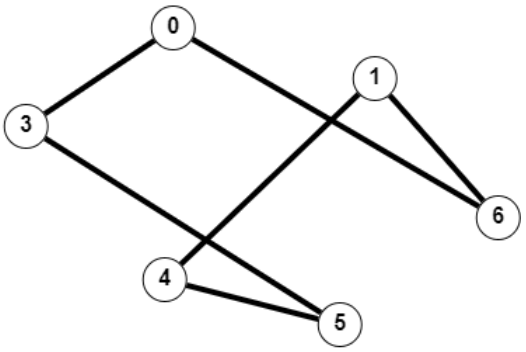

Figura 5: Um possível vizinho 3-OPT da solução inicial. Em que as arestas E(0, 1), E(6, 5) e E(4, 3), foram substituídas pelas arestas E(0,6), E(1, 4) e E(5, 3).

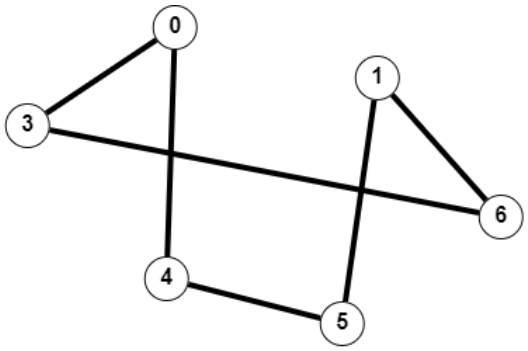

Figura 6: Um possível vizinho 3-OPT da solução inicial. Em que as arestas E(0, 1), E(6, 5) e E(4, 3), foram substituídas pelas arestas E(0,4), E(5, 1) e E(6, 3).

3-OPT, passamos de O(n^2) para O(n^3). Em uma 4-OPT podemos obter resultados mais expressivos, porém o preço a pagar é de O(n^4).

3.3.3 Vizinhanças Swap e Shift

Essas duas estruturas de vizinhança são alternativas as apresentadas anteriormente. A estrutura 2-Swap seleciona 2 vértices da solução inicial e simplesmente inverte suas posições, que é uma operação feita em tempo constante, O(1), mas testando-se para todas as combinações de dois vértices, leva tempo O(n^2), com n sendo o número de vértices. Como exemplo, considere a lista inicial $\{0, 1, 2, 3, 4, 5\}$, um movimento 2-Swap entre os vértices 0 e 3 leva à lista $\{3, 1, 2, 0, 4, 5\}$.

A estrutura 2-Shift realiza um movimento de deslocamento de duas posições para a direita em um vértice selecionado. É bem eficiente, pois basta percorrer todos os vértices n, O(n), e realizar um deslocamento de duas posições, que é uma operação bem barata, levando à complexidade O(n). Por exemplo, vamos utilizar a mesma lista anterior, se fizermos um 2-Shift no vértice 0, obtemos $\{1, 2, 0, 3, 4, 5\}$.

As duas estruturas ofertam movimentos distintos da vizinhança 2-OPT, e são mais eficientes que a 3-OPT, o que leva à oportunidades distintas na hora de implementar buscas locais com múltiplas estruturas de vizinhança.

4 Algoritmos de Clusterização para o CVRP

Para trabalhar com o CVRP, estudamos a técnica da clusterização. Nela formamos uma partição do conjunto de clientes, onde cada parte pode ser atendida por um veículo da frota levando em consideração a capacidade dos veículos. Dessa forma, reduzimos o CVRP a diversos subproblemas do TSP. Então usamos algoritmos que resolvem o TSP, para formar rotas entre clientes de cada parte e o depósito.

Existem várias implementações para esta técnica. Uma delas é a ideia de formar aglomerados ao redor de vértices iniciais selecionados por algum critério, como a aleatoriedade. Esses vértices são chamados centros de *cluster*. A partir dessa seleção, verificamos de qual centro cada cliente está mais perto, o inserindo no *cluster* do mesmo. O *cluster* estará completo quando nenhum cliente couber na capacidade W dos veículos da frota. O algoritmo termina quando todos os clientes são inseridos em algum cluster. Ao final, obtemos uma partição que divide os clientes entre os veículos da frota. O pseudocódigo dessa ideia pode ser visto no Algoritmo 5.

Algoritmo 5 Clusterização com centros em Paralelo

Input: Um grafo conexo $G = (C, E)$, a capacidade W da frota e a quantidade T de veículos da frota

Output: P, um conjunto de partições P' do conjunto C respeitando a capacidade W dos veículos

1: **function** PARALELO($G = (C, E)$, W, T)
2: **for** $i = 1$ até T **do**
3: $c \leftarrow random(C)$
4: $P'_i \leftarrow \{c\}$
5: Remova c do conjunto C
6: **while** $C \neq \emptyset$ **do**
7: Selecione um cliente $c \in C$ e encontre o centro de *cluster* da partição P'_i mais próximo
8: $P'_i \leftarrow P'_i \cup \{c\}$
9: Remova c do conjunto C
10: $P \leftarrow \emptyset$
11: **for** $i = 1$ até T **do**
12: $P \leftarrow P \cup \{P'\}$
13: **return** P

Outra ideia para algoritmo de clusterização é a construção de uma parte por rodada do algoritmo, por meio do algoritmo caminho guloso, que é uma adaptação do *Nearest Neighbor*. Primeiro selecionamos aleatoriamente um cliente c do conjunto C para ser o início do *cluster*. A partir disso, adicionamos o cliente mais próximo do cliente que está sendo analisado atualmente. O pseudocódigo dessa abordagem é mostrado no Algoritmo 6.

Algoritmo 6 Caminho Guloso

Input: Um grafo conexo $G = (C, E)$ e a capacidade W da frota
Output: P, uma parte do conjunto C respeitando a capacidade W dos veículos

function CAMINHOGULOSO($G = (C, E)$, W)
    $P \leftarrow random(C)$
    while $C \neq \emptyset$ ou $w(P) \leq W$ **do**
        Encontre o cliente $c \in C$ mais próximo de $P_{ultimoInserido}$
        $P \leftarrow P \cup \{c\}$
        Remova c do conjunto C
    return L

5 Algoritmo de Geração de Pesos para o GVRP

Com relação ao GVRP, por ser um problema multi-objetivo, utilizamos um método de geração de pesos, a fim de gerar várias soluções para popular a fronteira de pareto. Essa fronteira corresponde ao conjunto de soluções que não são dominadas por nenhuma outra solução do conjunto de soluções.

Então, adaptamos nossas heurísticas e meta-heurísticas. Isso foi feito através do algoritmo para geração de pesos, o qual se utiliza da estrutura de dados fila. Nele, no começo resolvemos o problema com um esquema de pesos p_1 e p_2, que valem respectativamente $[1, 0]$ e $[0, 1]$ e inserimos essa dupla na primeira posição da fila. Depois, no primeiro laço do algoritmo retiramos esse esquema de pesos da fila e calculamos um novo valor p_3 que é a média de p_1 e p_2, resolvemos o problema com esse novo valor e inserimos na fila duas novas duplas de pesos, que se tratam de $[p_1, p_3]$ e $[p_2, p_3]$. O algoritmo segue até que um critério de parada seja alcançado, como por exemplo, número de soluções que queremos obter. O pseudocódigo se encontra em Algoritmo 7.

Algoritmo 7 Algoritmo de geração de pesos

Input: p_1, p_2
Output: Conjunto de soluções S

function GERADORPESOS
    $S \leftarrow ~~S \cup~~ \{resolvaGVRP(p_1), resolvaGVRP(p_2)\}$
    Insira $[p_1, p_2]$ na fila
    while Critério de parada não atingido **do**
        Retire um esquema de peso da fila, digamos [pi, pj]
        $p_k \leftarrow (p_i + p_j)/2$
        $S \leftarrow S \cup \{resolvaGVRP(p_k)\}$
        Insira $[p_i, p_k]$ na fila
        Insira $[p_j, p_k]$ na fila
    return S

6 Metaheurísticas

Nesta seção são apresentadas metaheurísticas estudadas, algumas das quais foram implementadas para os problemas abordados.

6.1 Multistart

A meta-heurística *Multistart*, também conhecida como reinício aleatório, funciona por meio da combinação de soluções iniciais aleatórias com uma heurística de busca local. O objetivo é explorar melhor o espaço de soluções para o problema, porém, mantendo a característica das heurísticas de resolver instâncias em tempo hábil e sem garantias do ótimo. Para isso, o critério de parada pode ser implementado de várias formas, como: um limite de iterações ou quantidade de iterações sem melhoria na qualidade da melhor solução já encontrada. O pseudocódigo do *Multistart* é apresentado no Algoritmo 8.

Algoritmo 8 Multistart

Input: Uma entrada I
Output: Retorna um mínimo local S^*

```
1: function MULTISTART(I)
2:     while Critério de parada não atingido do
3:         S ← solucaoaleatoria(I)
4:         S ← buscalocal(S)
5:         if S for melhor que S* then
6:             S* ← S
7:     return S*
```

6.2 Recozimento Simulado

O Recozimento Simulado, do inglês *Simulated Annealing*, funciona por meio de decisões aleatórias e uma função de aceitação do vizinho baseada em um Cronograma de Recozimento. Mais especificamente, a cada iteração uma solução analisa toda sua vizinhança e escolhe alguma solução vizinha de forma aleatória. Se a temperatura estiver alta, a probabilidade de aceitação de uma solução que piore a qualidade da solução é maior. Conforme a temperatura abaixa, essa probabilidade diminui e o algoritmo toma comportamento parecido com o de uma busca local simples. O Cronograma de Recozimento decide como a temperatura é reduzida ao longo das iterações. Um pseudocódigo para esse método é apresentado no Algoritmo 9.

6.3 Busca Tabu

A Busca Tabu é assim denominada por empregar estratégias que proibem certas ações ou movimentos de busca durante a exploração de vizinhanças. Essa meta-heurística tem como princípios criar uma busca eficiente evitando revisitar soluções. Para tanto, ela registra as alterações realizadas ao criar determinadas soluções, usando memórias de curto e médio prazo. A memória de curto prazo é conhecida como lista de movimentos, uma lista que

Algoritmo 9 Simulated Annealing

Input: Uma solução inicial S
Output: Retorna um mínimo local S^*

```
1:  function SimulatedAnnealing(S)
2:      S* ← S
3:      while Critério de parada não atingido do
4:          S' ← solucaoVizinhaAleatoria(S)
5:          if S for melhor que S* then
6:              S* ← S
7:          else
8:              if temperatura > 0 then
9:                  Decida por meio da temperatura se S* ← S
10:     return S*
```

registra movimentos que levaram à formação da solução em foco. Parte dessa lista é a lista tabu, responsável por armazenar movimentos proibidos. No algoritmo clássico, movimentos proibidos são aqueles que ocorreram recentemente e que, se fossem realizados novamente, levariam a ciclos e movimentos comprovadamente ineficientes. Mas a proibição é flexível, pois, em determinadas ocasiões algum movimento da lista tabu pode levar a soluções boas. Se este for o caso, a restrição do movimento é ignorada. A memória de médio prazo registra vizinhanças promissoras que foram descartadas por decisões locais do algoritmo e soluções de elite, isto é, as melhores soluções encontradas até então. A imagem 7 ilustra o comportamento que a lista tabu trás ao algoritmo.

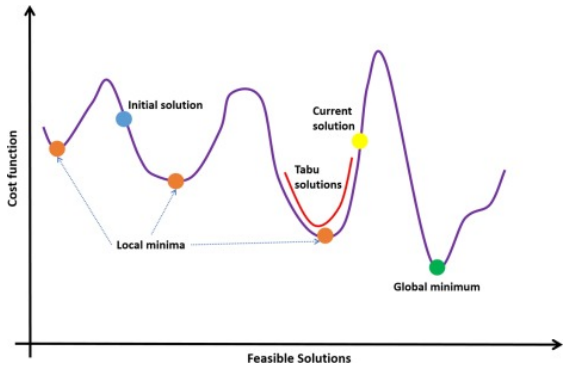

Figura 7: Neste gráfico, de solução viável por custo da solução, considere que o problema seja de minimização. A bolinha amarela é a solução corrente, as laranjas são mínimos locais, a verde é o mínimo global e a azul se trata da solução inicial, já a linha laranja se trata de todas as soluções que estão na lista tabu, elas são tabu pois, apesar de melhorarem o custo da solução, elas nos levam em um mínimo local que queremos escapar. Por questões de eficiência na implementação, a lista tabu não armazena soluções inteiras, apenas as trocas que levam até elas. A imagem foi retirada do artigo de Ibarra-Rojas et. al [5].

6.4 GRASP

O GRASP, do inglês *Greedy Randomized Adaptive Search Procedure*, é uma meta-heurística formada por um procedimento construtivo e um procedimento de busca local. O procedimento construtivo é projetado por meio de uma estratégia gulosa aleatorizada. O GRASP busca obter, em sua primeira fase, soluções diversificadas e de melhor qualidade que soluções puramente aleatórias, lançando mão de diferentes estratégias. Uma das mais utilizadas é a estratégia semigulosa de Hart e Shogan [4], na qual a clássica escolha gulosa determinística é substituída por um critério de escolha aleatória em um conjunto restrito definido por um critério guloso, que é chamado de Lista Restrita de Candidatos (LRC). A solução obtida na primeira fase é melhorada por uma busca local. Estes procedimentos são repetidos diversas vezes e em cada iteração uma nova solução para o problema é obtida. No final escolhemos a melhor solução encontrada. O algoritmo é adaptativo, pois a LRC é atualizada ao longo das iterações. Na maioria dos algoritmos gulosos, a avaliação das variáveis candidatas a compor a solução é feita uma única vez e permanece imutável ao longo do processo construtivo. Assim, o critério guloso é, na maioria dos casos, insensível ao processo de formação da solução. Já no GRASP, o algoritmo adapta esses critérios de acordo com a solução encaminhada, evitando armadilhas e viéses em que o processo guloso puro costuma cair. O Algoritmo 10 apresenta seu pseudocódigo.

Algoritmo 10 GRASP

Input: Uma entrada I
Output: Retorna uma solução S^*

```
1: function GRASP(I)
2:     while Critério de parada não atingido do
3:         S ← greedyRand(I)
4:         S ← localSearch(S)
5:         if S for melhor que S* then
6:             S* ← S
7:     return S*
```

Na construção da solução gulosa aleatorizada, um possível critério para selecionar elementos i em um problema de minimização para a LRC é analisar se o custo de i está dentro do intervalo definido pela equação: $LRC = \{i \leq menorCusto + (maiorCusto - menorCusto)\alpha\}$. Com "maiorCusto"sendo o elemento i mais custoso e "menorCusto"o menos custoso. O α é um valor que pertence ao intervalo $[0, 1]$ e define uma quantidade de elementos que pertencerão à lista, se ele valer 0, acontece a decisão gulosa tradicional, mas se ele valer 1, todos os elementos i estarão presentes na lista, o que torna o método um algoritmo totalmente aleatório. Uma possível implementação deste algoritmo guloso aleatorizado para o VRP é apresentado no Algoritmo 11.

6.5 Busca em Vizinhança Variável

A busca em Vizinhança Variável (VNS) é uma generalização da busca local que explora múltiplas estruturas de vizinhanças de uma solução. Denomina-se busca em vizinhança

Algoritmo 11 Guloso Aleatorizado

Input: Um grafo $G = (V, E)$
Output: Retorna uma solução S

```
1: function GULOSOALEATORIZADO(G)
2:     while Não visitou todos os vértices do
3:         Seja u o último vértice adicionado em S
4:         Percorra todos os vértices fora de S e encontre menorCusto e maiorCusto
5:         LRC = {i ≤ menorCusto + (maiorCusto − menorCusto)α}
6:         escolha um i aleatoriamente em LRC
7:         adicione i no final de S
8:     return S
```

variável, pois propõe explorar o espaço de busca variando sistematicamente a vizinhança, sendo que as vizinhanças mais simples e eficientes possuem maiores chances de utilização durante a execução do algoritmo. Mais precisamente, a VNS primeiro determina uma solução que é ótimo local em uma vizinhança, então busca por melhorias em outra vizinhança, na esperança de se aproximar do ótimo global. O algoritmo termina quando a estrutura de vizinhança mais complexa utilizada alcança o mínimo local.

6.6 Variable Neighbourhood Descent

Uma variante particularmente eficiente à VNS é a Variable Neighbourhood Descent (VND). A VND também usa várias vizinhanças, em ordem crescente de complexidade. Quando uma estrutura de vizinhança menos complexa alcança o mínimo local, avançamos para uma mais complexa. Se esta conseguir algum resultado que melhore a solução, retornamos à estrutura anterior e continuamos a busca com ela. O algoritmo termina quando alcançamos o mínimo local em todas as estruturas. O Algoritmo 12 apresenta uma VND para o TSP, utilizando as vizinhanças 2-OPT e 3-OPT.

7 Implementação

Dentre os algoritmos descritos nas seções anteriores, no primeiro semestre desta iniciação científica foram implementados em linguagem Python o algoritmo guloso, o aleatorizado e a busca local com estrutura de vizinhança 2-OPT para o TSP. Também foi implementado uma metaheurística multistart para o VRP. Os detalhes de implementação e resultados foram apresentados no relatório parcial.

No segundo semestre foram implementados, também em Python, a busca local 3-OPT, os dois algoritmos de clusterização descritos e uma combinação de GRASP com a VND (GRASP-VND) para o CVRP. Para atacar o GVRP, adaptamos os algoritmos de clusterização e o GRASP-VND, além de implementarmos o algoritmo de geração de pesos.

Os algoritmos foram testados com diversas instâncias para avaliar seus desempenhos tanto em tempo quanto em qualidade de soluções. Os resultados obtidos nesses testes

Algoritmo 12 VND

Input: A entrada I e uma solução inicial S
Output: Retorna uma solução S^* mínimo local em todas as vizinhanças

```
1: function VND(I, S)
2:     S* ← S
3:     while True do
4:         S ← 2opt(S)
5:         if S melhor que S* then
6:             S* ← S
7:         else
8:             S ← 3opt(S)
9:             if S melhor que S* then
10:                S* ← S
11:            else
12:                Encerre o laço, pois um mínimo local da 2-OPT e da 3-OPT foi encontrado.
13:    return S*
```

estão descritos na Seção 8 e podem ser conferidos junto aos códigos no github [1] [2].

7.1 Clusterização

Para implementar os algoritmos de clusterização, trabalhamos com um grafo completo $G = (V, E)$ e distâncias euclidianas entre os vértices para representar o peso das arestas.

Para implementar os dois métodos de clusterização inserimos todos os vértices, exceto o depósito, em um dicionário, a fim de facilitar as operações de inserção, remoção e verificação de clientes. Selecionamos um cliente aleatoriamente no dicionário por meio da função *random.choice(dicionario.keys()))*. A partir disso os caminhos mudam dependendo do procedimento, no nomeado Caminho Guloso, entramos no laço principal que dura até que o dicionário esteja vazio, ou que a capacidade W do veículo acabe. A cada iteração do laço, o cliente mais próximo do cliente inserido anteriormente é colocado no *cluster* se sua capacidade respeitar a capacidade restante do veículo. No final, inserimos o depósito na lista e devolvemos essa partição do conjunto V para o programa resolver como um TSP entre depósito e clientes.

Já no algoritmo de clusterização com centros em Paralelo, selecionamos vértices aleatórios para ser centro de *cluster* h vezes, com h sendo o tamanho da frota, então, trabalhamos com a formação h *clusters* ao mesmo tempo. Após essa seleção, entramos no laço principal, que dura enquanto ou existem clientes para serem atendidos, ou há capacidade disponível para inserir um cliente em algum *cluster*. Daí, analisamos em qual centro o cliente escolhido neste laço está mais próximo e o inserimos na partição que este centro se encontra. No final, inserimos o depósito em todas as listas, que representam as partições no programa, e, também, as retornamos para que o programa resolvá-as como um TSP.

[1] https://github.com/matheustmattioli/ICCVRPheuristic
[2] https://github.com/matheustmattioli/ICMultiObjGVRP

As adaptações para atacar o GVRP foram feitas nas linhas que se referem à decisões de proximidade, em que pesos entre 0 e 1, ponderam as escolhas conforme o peso das funções objetivos nesta rodada do programa.

7.2 Multistart

O multistart funciona decidindo a melhor solução obtida após uma quantidade estipulada de reinícios do programa, naturalmente, para termos soluções distintas, a construção delas deve possuír alguma etapa aleatorizada. Felizmente, nosso programa apresenta aleatorização na construção das partições e na etapa de construção das soluções. Então, para controlar esses reinícios, implementamos algumas variáveis de controle, tais como o *N_ITE_MS_CLUST*, que define a quantidade de iterações do multistart na clusterização, isto é trabalhamos o problema com formações de várias partições distintas. E temos o *N_ITE_GRASP* com o *MAX_ITER_W_NO_IMPROV* que definem quantas vezes executamos o GRASP, eles serão melhor explicados na próxima subseção.

7.3 GRASP-VND

O GRASP funciona com uma etapa construtiva gulosa aleatorizada, uma etapa de busca local e um laço principal que une as duas etapas anteriores em uma espécie de multistart. Na nossa implementação, utilizamos as variáveis *N_ITE_GRASP* e *MAX_ITER_W_NO_IMPROV* para controlar o número de iterações desse laço principal. Dentro dele, chamamos o método construtivo e a busca local, sendo que a busca local é um *Variable Neighbourhood Descent*, no final de cada laço avaliamos se a solução obtida é melhor que a melhor solução encontrada em rodadas anteriores. Também atualizamos o valor α utilizado no método construtivo para definir o tamanho da lista restrita de candidatos, seu valor é incrementado até o valor máximo definido de $0,5$. Este valor é independente da variável *N_ITE_GRASP*, ela apenas influencia em quanto o α é incrementado por iteração, porém, pode ser que o valor máximo do α não seja atingido caso a variável *MAX_ITER_W_NO_IMPROV* seja violada.

O método construtivo é guloso aleatorizado e adaptativo. É guloso aleatorizado por conta da LRC, adaptativo porque a atualizamos em cada laço do método guloso. Sendo assim, o algoritmo foi implementado através de um laço principal que opera enquanto existirem vértices fora da rota. A cada iteração é analisado o vértice mais distante e mais próximo do vértice inserido anteriormente na rota, daí calculamos o intervalo de valores possíveis de serem inseridos na LRC e inserimos os vértices presentes nesse intervalo na LRC, então selecionamos um desses vértices aleatoriamente e inserimos na rota. É adaptativo, pois a cada iteração atualizamos a lista restrita de candidatos de acordo com o último vértice inserido. O α só é atualizado no multistart do GRASP, na etapa construtiva ele entra no cálculo da LRC.

Após a construção da solução, ocorre a etapa de intensificação do GRASP conhecido como busca local, porém no nosso caso, utilizamos a VND como procedimento de busca local. É um procedimento simples que funciona com um laço principal, que dura enquanto não encontramos o mínimo local de todas as estruturas de vizinhança, e as estruturas 2-opt e 3-opt. A 2-opt é utilizada a cada laço até que ela não melhore mais a solução, a partir

desse ponto, utilizamos a 3-opt, se ela melhorar a solução, na próxima iteração retornamos a utilizar a 2-opt, senão, o algoritmo encerra e retorna a melhor solução encontrada.

A adaptação feita para o GVRP é presente na etapa construtiva, com os pesos das funções objetivo influenciando nas escolhas de vizinho mais distante e mais próxima, e portanto, no cálculo do intervalo da LRC. Também é presente na etapa de intensificação, dentro das estruturas 2-opt e 3-opt os pesos participam do cálculo de escolha menos custosa.

8 Resultados Experimentais

Nesta seção são apresentados os resultados obtidos pelos algoritmos implementados para os problemas CVRP e GVRP. Todos os testes foram executados em um processador AMD FX-8350 CPU 4.0GHz, com o sistema operacional Windows 10, versão 21H1 e *OS build* 19043.1110. Os testes foram realizados com as variáveis de controle nas seguintes configurações: $N\_ITE\_MS\_CLUST = 100$, $MAX\_ITER\_W\_NO\_IMPROV = 100$ e $N\_ITE\_GRASP = 500$.

8.1 Testes para o CVRP

A seguir são apresentadas as Tabelas 1 e 2 que contêm os resultados da aplicação dos algoritmos implementados em instâncias selecionadas para o CVRP. As Figuras 8 e 9 ilustram a solução obtida para a instância "vrp_16_3_1", respectivamente, pelo GRASP-VND com clusterização *Greedy Path* e pelo GRASP-VND com clusterização em Paralelo.

Instância	**Clientes**	**Veículos**	**Capacidade**	**Valor Ótimo**	**Custo**	**Tempo (s)**
vrp_16_3_1	16	3	90	279	284,10	5,18
vrp_26_8_1	26	8	48	622	628,17	6,80
vrp_51_5_1	51	5	160	525	594,80	72,37
vrp_101_10_1	101	10	200	829	879,71	201,37
vrp_200_16_1	200	16	200	1400	1624,96	517,52
vrp_421_41_1	421	41	200	2000	2057,30	768,78

Tabela 1: Custos das soluções produzidas pelo GRASP-VND com clusterização Greedy Path.

8.2 Testes para o GVRP

Como as instâncias trabalhadas do GVRP são adaptações das do CVRP, não sabemos sua fronteira de pareto ótima, ou seja, o conjunto ótimo de soluções não dominadas. As Figuras 10 a 15 apresentam o conjunto de soluções encontrado com o GRASP-VND adaptado para as funções distância e emissão de poluentes, ambas de minimização.

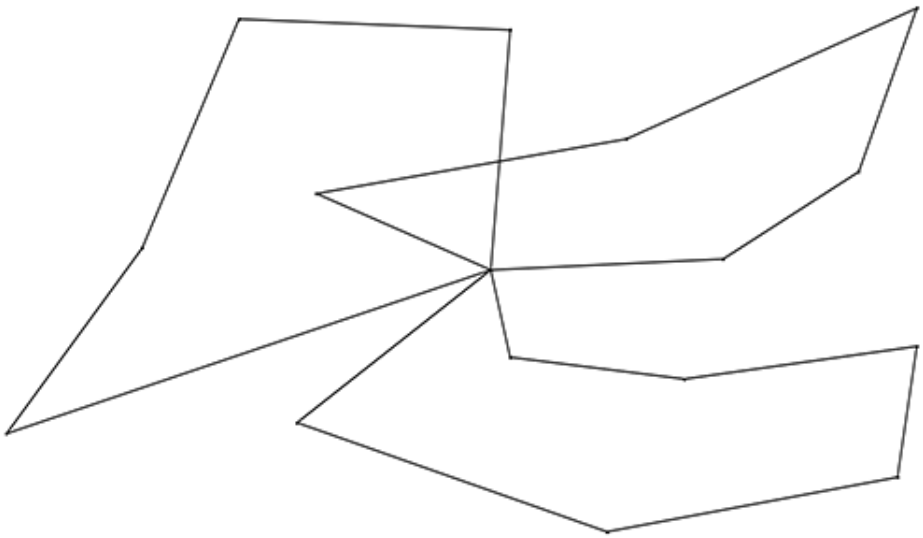

Figura 8: Solução encontrada pelo GRASP-VND com clusterização Greedy Path para a instância de 16 clientes do CVRP.

Instância	**Clientes**	**Veículos**	**Capacidade**	**Valor Ótimo**	**Custo**	**Tempo (s)**
vrp_16_3_1	16	3	90	279	284,61	5,52
vrp_26_8_1	26	8	48	622	Sem resposta	Sem resposta
vrp_51_5_1	51	5	160	525	597,57	78,09
vrp_101_10_1	101	10	200	829	1013,03	183,32
vrp_200_16_1	200	16	200	1400	Sem resposta	Sem resposta
vrp_421_41_1	421	41	200	2000	2334,74	841,78

Tabela 2: Custos das soluções produzidas pelo GRASP-VND com clusterização em Paralelo. Por conta da natuzera da clusterização em paralelo, alguns clientes não foram atendidos nas instâncias de 26 clientes e de 200, portanto, o algoritmo não retornou resposta válida.

9 Pedido de Renovação da Bolsa

Esta seção trata do pedido de renovação da bolsa e do planejamento das próximas etapas do projeto. Durante este projeto de iniciação científica, o bolsista estudou diversas abordagens heurísticas e metaheurísticas, usando o VRP como linha condutora deste estudo. No final, algoritmos baseados em algumas das abordagens estudadas foram implementados e testados. Destacamos que alguns dos algoritmos implementados foram para o TSP, caso particular do VRP, enquanto outros foram para generalizações, como o CVRP e o GVRP. Assim, o projeto permitiu ao estudante construir uma base de conhecimentos sobre problemas de roteamento e métodos heurísticos. Nas etapas seguintes queremos aplicar esses conhecimentos para desenvolver, implementar e testar métodos heurísticos para o Problema do Roteamento da Mula de Dados (PRMD).

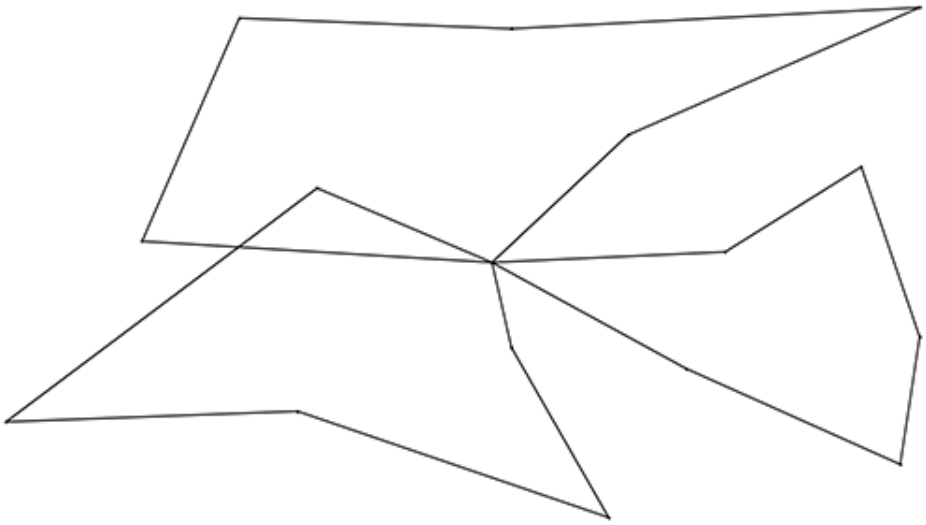

Figura 9: Solução encontrada pelo GRASP-VND com clusterização em Paralelo para a instância de 16 clientes do CVRP.

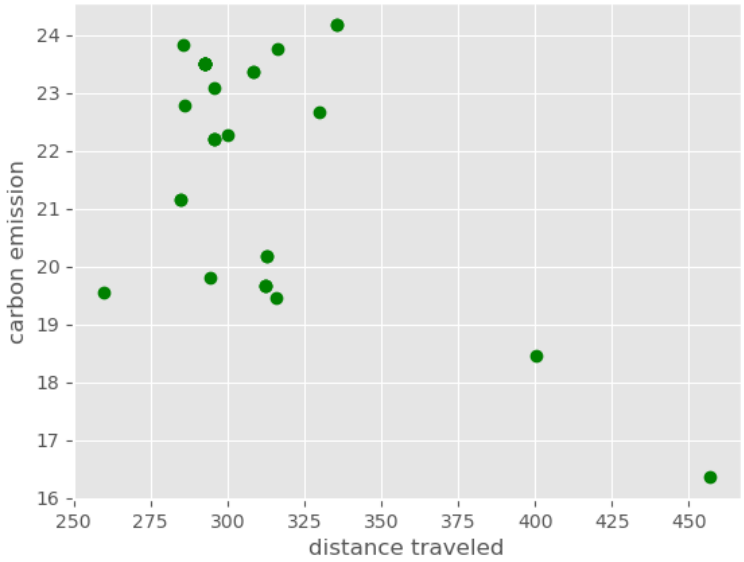

Figura 10: Conjunto de soluções para a instância vrp_16_3_1

O PRMD é um problema de roteamento em que um agente, chamado de mula de dados, deve coletar dados de sensores esparsamente espalhados em uma certa região e retornar para a base da qual partiu, formando um circuito [12]. A principal diferença deste problema para outros problemas de roteamento, como TSP e VRP, é o fato dos sensores terem wi-fi, permitindo que a coleta seja feita a uma certa distância do aparelho, além da possibilidade de comunicação entre os sensores. Por conta dessas características, há a possibilidade de atender mais de um sensor a partir de uma mesma localização.

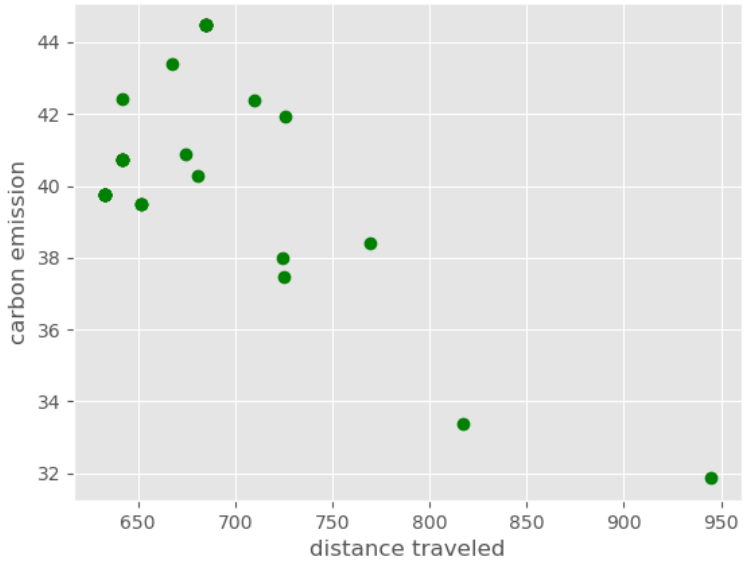

Figura 11: Conjunto de soluções para a instância vrp_26_8_1

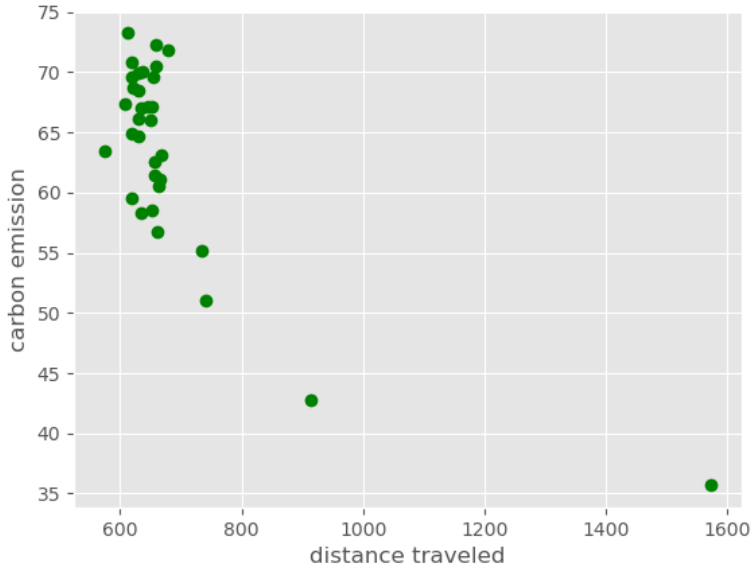

Figura 12: Conjunto de soluções para a instância vrp_51_5_1

Para atacar este problema, pretendemos projetar e implementar um GRASP-VND e combiná-lo com o BRKGA. Acreditamos que essas abordagens são promissoras pois, além do estudante já ter alguma experiência com a metaheurística GRASP-VND, esta produz uma coleção de soluções variadas, por conta do algoritmo guloso aleatorizado, e de boa qualidade, por conta da intensificação da busca VND. Assim, as soluções produzidas pelo GRASP-VND podem ser usadas como população inicial do BRKGA. Isso porque o

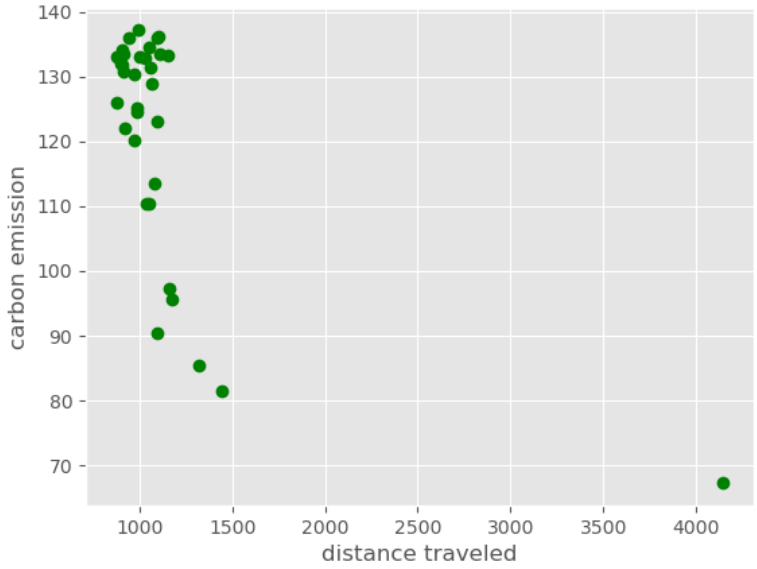

Figura 13: Conjunto de soluções para a instância vrp_101_10_1

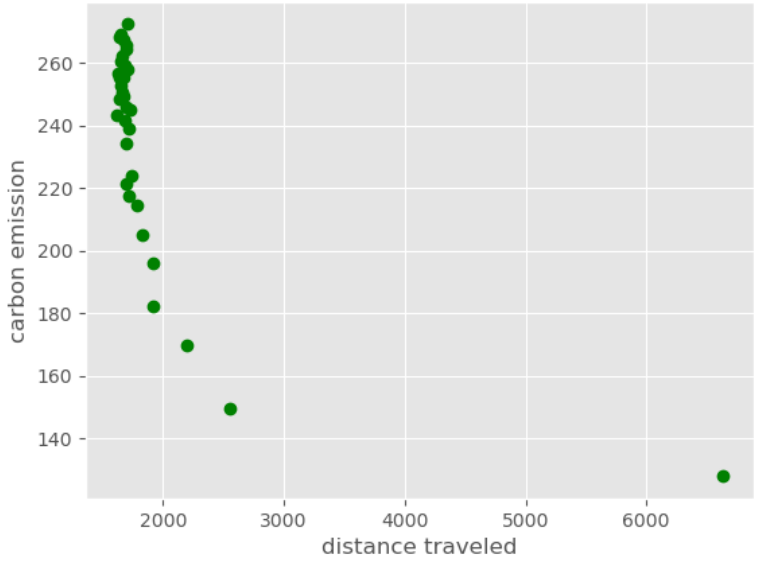

Figura 14: Conjunto de soluções para a instância vrp_200_16_1

BRKGA, que é uma metaheurística evolutiva aleatorizada, costuma se beneficiar ao receber algumas soluções de boa qualidade em sua população inicial, ao invés de usar apenas indivíduos gerados aleatoriamente. Também existe a possibilidade de trabalharmos com uma definição multiobjetiva do problema e com variantes que consideram outras restrições na rede de sensores, como as encontradas em Ma et al. [11] e Sugihara e Gupta [15].

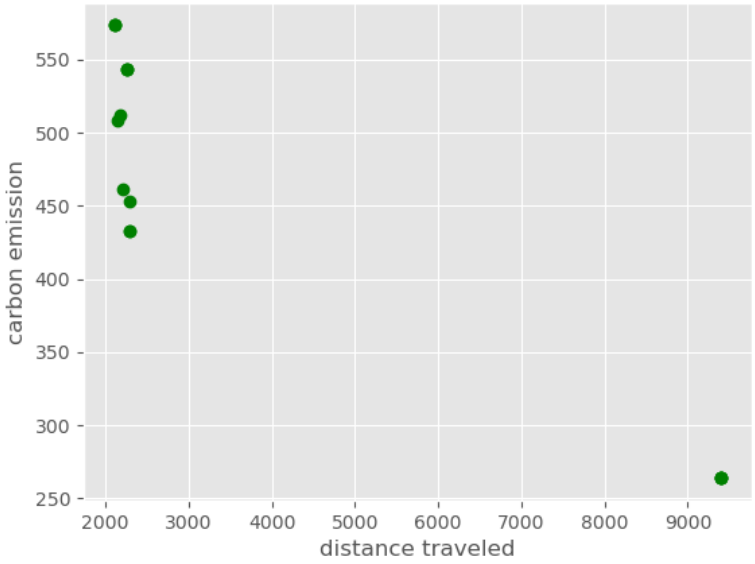

Figura 15: Conjunto de soluções para a instância vrp_421_41_1

10 Cronograma

A Tabela 3 contém o cronograma completo do projeto original.

Tabela 3: Cronograma das atividades.

Atividades	Meses											
	1	2	3	4	5	6	7	8	9	10	11	12
Heurísticas	•	•	•	•								
Meta-Heurísticas					•	•	•	•				
Variantes do problema									•	•	•	•
Implementação	•	•	•	•	•	•	•	•	•	•	•	•

O primeiro semestre foi destinado ao estudo de bases teóricas, de algoritmos heurísticos clássicos para o TSP e sua generalização, o VRP, bem como de abordagens metaheurísticas. Também teve início nesse período a fase de implementação, com foco em algoritmos heurísticos. Além disso, foram realizados testes que avaliam a qualidade das soluções produzidas por esses algoritmos e a eficiência da implementação dos mesmos.

No segundo semestre desta iniciação científica, continuou-se o estudo e implementação de algoritmos para o VRP com mais profundidade, explorando combinações de heurísticas e meta-heurísticas estudadas, como o GRASP com a fase de intensificação sendo uma Variable Neighbourhood Descent. Também iniciou-se os estudos e implementações de algoritmos para a variante CVRP. E dentre eles, abordou-se algoritmos de agrupamento, ou clusterização, para lidar com a questão da capacidade dos veículos.

Os últimos dois meses da IC foram dedicados ao tema da otimização multiobjetivo,

no qual, estudou-se seus principais conceitos, como fronteira de pareto. A variante bi-objetiva GVRP foi escolhida para ser atacada com os métodos de clusterização, heurísticas e meta-heurísticas adaptadas para problemas deste tipo.

Durante todo o período do projeto, foram feitas reuniões com os professores orientadores e elas foram compartilhadas com outro orientando, o qual trabalha com o problema do VRP usando métodos exatos. Isto foi proveitoso para desenvolver trabalhos em conjunto, onde cada um dos orientandos teve a oportunidade de contribuir com a pesquisa do outro. Dentre essas contribuições vale destacar que publicamos um artigo nomeado Spanning Cover Inequalities for the Capacitated Vehicle Routing Problem [1] nos anais do VI Encontro de Teoria da Computação (ETC 2021), conquistando o prêmio de melhor trabalho publicado.

Em caso de resposta positiva da FAPESP sobre o pedido de prorrogação da bolsa, haverá uma próxima etapa do projeto, com duração de 12 meses, na qual será estudado o problema PRMD, que é uma variante do VRP. Pretende-se aplicar técnicas meta-heurísticas ao problema, em particular, GRASP-VND e BRKGA. A Tabela 4 apresenta o planejamento do período requisitado da prorrogação da bolsa.

Tabela 4: Cronograma para as próximas atividades.

Atividades	Meses											
	1	2	3	4	5	6	7	8	9	10	11	12
Revisão Bibliográfica sobre o PRMD	•	•										
GRASP-VND para o PRMD			•	•	•							
BRKGA para o PRMD						•	•	•	•			
Variante Multiobjetivo do PRMD										•	•	•

11 Conclusão

Foram estudados e implementados em Python algoritmos conhecidos para o problema do caixeiro viajante e do roteamento de veículos, como algoritmos gulosos, aleatorizados e heurísticas de busca local com estrutura de vizinhaça 2-OPT. Na busca local foram implementadas as escolhas best fit e first fit, a fim de comparar a qualidade das soluções produzidas por cada uma delas. A heurística construtiva aleatorizada foi usada junto da busca local, na implementação de uma metaheurística multistart.

A partir desses primeiros passos, adaptamos o algoritmo guloso, proposto na Seção 3.1, inserindo uma lista restrita de candidatos e a escolha aleatória de um elemento dessa lista, chegando na meta-heurística GRASP. Através da combinação da estrutura de vizinhança 3-OPT com a da 2-OPT, obteu-se a meta-heurística de busca local VND. Substituindo a busca local do GRASP pela VND, formou-se uma meta-heurística mais robusta chamada de GRASP-VND.

Então, para atacar o CVRP houve uma redução ao TSP através de métodos de clusterização, no qual se formou subconjuntos de clientes do conjunto original para cada caminhão, respeitando suas capacidades. Para o GVRP, os algoritmos foram adaptados

para ponderar a influência nas escolhas de cada função objetivo, através da escolha de um esquema de pesos a cada iteração.

Os resultados obtidos com os algoritmos implementados corresponderam ao que era esperado de heurísticas e meta-heurísticas. Isso porque os algoritmos implementados foram bem eficientes e, mesmo que sem garantias de otimalidade, apresentaram soluções de boa qualidade. Por conta dos estudos mencionados, houve o desenvolvimento de mais familiaridade com a linguagem Python, estruturas matemáticas, como grafos, tipos abstratos de dados, como dicionários, e problemas NP-Difíceis, enriquecendo meu conhecimento em ciência da computação, projeto e análise de algoritmos e otimização combinatória.

Fiquei muito feliz com a oportunidade de desenvolver este projeto e ainda existem muitas atividades relevantes sugeridas a serem realizadas, como a implementação de mais meta-heurísticas e a oportunidade de trabalhar com o problema de roteamento de mula de dados, o que deixará o projeto ainda mais completo. Dito isto, considero a conclusão dessa etapa do projeto bem sucedida e espero que as próximas etapas também possam ser realizadas.

Referências

[1] Guilherme G Arcencio, Matheus T Mattioli, Pedro HDB Hokama, and Mário César San Felice. Spanning cover inequalities for the capacitated vehicle routing problem. In *Anais do VI Encontro de Teoria da Computação*, pages 86–89. SBC, 2021.

[2] Burak Eksioglu, Arif Volkan Vural, and Arnold Reisman. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering*, 57(4):1472–1483, 2009.

[3] Marshall L Fisher and Ramchandran Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124, 1981.

[4] J Pirie Hart and Andrew W Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107–114, 1987.

[5] Omar J Ibarra-Rojas, Felipe Delgado, Ricardo Giesen, and Juan Carlos Muñoz. Planning, operation, and control of bus transport systems: A literature review. *Transportation Research Part B: Methodological*, 77:38–75, 2015.

[6] Jaber Jemai, Manel Zekri, and Khaled Mellouli. An nsga-ii algorithm for the green vehicle routing problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 37–48. Springer, 2012.

[7] Sourabh Joshi and Sarabjit Kaur. Nearest neighbor insertion algorithm for solving capacitated vehicle routing problem. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 86–88. IEEE, 2015.

[8] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[9] Gilbert Laporte. Fifty years of vehicle routing. *Transportation science*, 43(4):408–416, 2009.

[10] Canhong Lin, King Lun Choy, George TS Ho, Sai Ho Chung, and HY Lam. Survey of green vehicle routing problem: past and future trends. *Expert systems with applications*, 41(4):1118–1138, 2014.

[11] Ming Ma, Yuanyuan Yang, and Miao Zhao. Tour planning for mobile data-gathering mechanisms in wireless sensor networks. *IEEE transactions on vehicular technology*, 62(4):1472–1483, 2012.

[12] Pablo LA Munhoz, Felipe P do Carmo, Uéverton S Souza, Lúcia MA Drummond, Pedro Henrique González, Luiz S Ochi, and Philippe Michelon. Locality sensitive algotrithms for data mule routing problem. In *International Conference on Algorithmic Applications in Management*, pages 236–248. Springer, 2019.

[13] Pablo Luiz Araújo Munhoz, Pedro Henrique González, Uéverton dos Santos Souza, Luiz Satoru Ochi, Philippe Michelon, and Lúcia Maria de A Drummond. General variable neighborhood search for the data mule scheduling problem. *Electronic Notes in Discrete Mathematics*, 66:71–78, 2018.

[14] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

[15] Ryo Sugihara and Rajesh K Gupta. Path planning of data mules in sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 8(1):1–27, 2011.

[16] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *European Journal of Operational Research*, 231(1):1–21, 2013.

[17] Thibaut Vidal, Gilbert Laporte, and Piotr Matl. A concise guide to existing and emerging vehicle routing problem variants. *European Journal of Operational Research*, 286(2):401–416, 2020.