
UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

MARCOS ANTONIO DE SOUZA

# ESTUDO DO PROBLEMA DE ROTEAMENTO COM BICICLETAS

CAMPINAS

2023

MARCOS ANTONIO DE SOUZA

# ESTUDO DO PROBLEMA DE ROTEAMENTO COM BICICLETAS

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na área de Automação.

Supervisor/Orientador: TAKAAKI OHISHI

Este trabalho corresponde à versão final da dissertação defendida pelo aluno Marcos Antonio de Souza, orientada pelo Prof. Dr. Takaaki Ohishi.

CAMPINAS

2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

So89e

Souza, Marcos Antonio de, 1971-
Estudo do problema de roteamento com bicicletas / Marcos Antonio de Souza. – Campinas, SP : [s.n.], 2023.

Orientador: Takaaki Ohishi.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Bicicletas. 2. Problema de roteamento de veículos. 3. Meta-heurística. I. Ohishi, Takaaki, 1955-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações Complementares

**Título em outro idioma:** Study of the routing problem with bicycles
**Palavras-chave em inglês:**
Bicycles
Vehicle routing problem
Meta-heuristic
**Área de concentração:** Automação
**Titulação:** Mestre em Engenharia Elétrica
**Banca examinadora:**
Takaaki Ohishi [Orientador]
Thayze D'Martin Costa da Silva
Cleber Damião Rocco
**Data de defesa:** 26-05-2023
**Programa de Pós-Graduação:** Engenharia Elétrica

**Identificação e informações acadêmicas do(a) aluno(a)**
- ORCID do autor: https://orcid.org/0009-0003-7298-6681
- Currículo Lattes do autor: https://lattes.cnpq.br/6850381898372592

COMISSÃO JULGADORA – DISSERTAÇÃO DE MESTRADO

Candidato: Marcos Antonio de Souza  RA: 900874
Data da defesa: 26 de maio de 2023
Título da Dissertação: “Estudo do problema de roteamento com bicicletas”

Prof. Dr. Takaaki Ohishi (Presidente)
Profa. Dra. Thayze D’Martin Costa da Silva
Prof. Dr. Cleber Damião Rocco

A Ata de Defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

Dedico esse trabalho à minha amada esposa Leatrice, que tanto se empenha em me ajudar a me tornar uma pessoa melhor, e por não desistir disso por mais de duas décadas.

## AGRADECIMENTOS

Agradeço a Deus pela sua vasta criação com incontáveis campos para estudo e pesquisa. Agradeço a minha esposa Leatrice por seu amoroso apoio e por ter estado ao meu lado já por mais de metade de minha vida. Agradeço ao meu pai por ter criado em mim o gosto pela programação de computadores quando eu ainda era criança e a minha mãe por ter me ensinado valores que impediram de me desviar de um bom caminho. Infelizmente, ambos meus pais agora descansam depois de uma jornada produtiva e da qual eu e minha irmã damos continuidade. Agradeço ao meu orientador professor Dr. Takaaki por me conduzir durante esse empreendimento para fora de minhas zonas de conforto.

# RESUMO

A utilização de bicicletas para serviços de entrega para clientes vem ganhando popularidade devido a um custo menor e pode alcançar tempos de entrega mais rápidos e mais confiáveis, especialmente em grandes cidades. Nesse trabalho consideramos como modificar o problema de roteamento de veículos em três de suas variantes clássicas (restrição de capacidade, janelas de tempo e frota mista) para incorporar um cenário de bicicletas que penalize os trajetos onde uma subida poderia exigir um esforço maior por parte do ciclista. São analisados os modelos matemáticos para essas formulações bem como três meta-heurísticas clássicas (busca tabu, algoritmo genético e colônia de formigas). Por fim, um algoritmo genético é aprimorado para gerar soluções para o problema completo, compreendendo uma frota mista de bicicletas e motocicletas, janelas de tempo, restrições de capacidade e penalidades de subidas íngremes.



Palavras-chave:

BICICLETAS, PROBLEMA DE ROTEAMENTO DE VEÍCULOS, META-HEURÍSTICAS.

ABSTRACT

Using bicycles for delivery services to customers is gaining popularity due to lower cost and can also achieve faster and more reliable delivery times, mainly inside city centers. In this work we consider how the vehicle routing problem can be modified in three classical variations (capacitated problem, time windows and mixed fleet) in order to incorporate a scenario with bicycles and penalties for paths with steep climbs, which may be harder for a biker. We analyze the mathematical models for these formulations as well as three classical meta-heuristics (tabu search, genetic algorithm and ant colony). Finally, a genetic algorithm is improved to generate a solution for the complete problem including a mixed fleet with bicycles and motorcycles, time windows, capacity restrictions and also a penalty scheme for steep climbs.

Key Words:
BICYCLES, VEHICLE ROUTING PROBLEM, META-HEURISTICS.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

# CAPÍTULO 1 INTRODUÇÃO

## 1.1 Motivação

Você alguma vez já se encontrou numa situação em que você estava dentro de um carro possante, capaz de ir de zero a 100 km/h em apenas alguns poucos segundos, mas completamente parado por causa de um congestionamento? Aí você olha para o semáforo e o vê mudando de vermelho para verde, depois de verde para amarelo, depois vermelho novamente, e você não se moveu nem mesmo um único metro! Então, uma bicicleta passa por você e você começa a pensar quão bom seria se estivesse com uma bicicleta ao invés do seu carro possante! Você começa a dizer a si mesmo que se estivesse com uma bicicleta então a essa hora você já estaria descansando tranquilamente em sua casa. Se você vive ou trabalha num centro de cidade grande, provavelmente já se viu numa situação como essa. Nem precisamos dizer, é fácil constatar, que bicicletas estão ganhando cada vez mais presença no dia a dia das grandes cidades.

Bicicletas são usadas hoje não apenas para passeios, mas também locomoção das pessoas e inclusive para a entrega de mercadorias. Nesse último caso, podemos ver bicicletas fazendo entregas para padarias, "*pet shops*", farmácias, e outros tipos de comércio. O mesmo aconteceu com os serviços de entrega de refeições que aumentaram seus negócios e ganharam ainda mais popularidade durante a pandemia do Covid-19. Ainda mais, com o aumento do negócio de compras *online* podemos ver algumas empresas de serviços logísticos usando bicicletas para cumprir com o "*last mile delivery*", ou a entrega na ponta, ao invés das vans e caminhões tradicionais. Justamente nesse campo podemos ter uma mudança de mentalidade e quebra de paradigmas passando a enxergar bicicletas como uma alternativa bem vantajosa para os serviços de logística.

A grande pergunta aqui é: as bicicletas podem oferecer um serviço de entrega equivalente, ou até melhor, em algumas circunstâncias do que vans ou outros veículos motorizados com maior capacidade de entrega? Dependendo da situação a resposta é bem positiva. Uma boa parte das entregas em cidades não está relacionada com volumes grandes se movendo de um ponto A para um ponto B. Ao invés disso, temos uma parcela significativa que é composta de volumes pequenos para serem entregues em vários pontos espalhados numa determinada vizinhança (o que constitui um clássico problema de roteamento de veículos). Além do mais, temos os fatores

presentes no ambiente de muitas cidades grandes que são obstáculos para vans e caminhões, mas que podem ser contornados por bicicletas: ruas congestionadas, horas de pico, restrições para circular com vans e caminhões durante certos horários, dificuldade para se estacionar em frente ao destino, custo elevado com estacionamentos e o próprio custo com combustível. Assim, a realidade é que atender a ponta ("*last mile*") no ambiente atual de cidades é um tópico complexo e oneroso para os serviços de logística. Para uma clareza maior, podemos visualizar uma primeira situação em que temos uma van que necessita entregar 3 itens pequenos em 3 localidades diferentes, separadas por 3 quilômetros uma da outra, durante o horário de pico e que não encontra local fácil para estacionar nessas regiões. Agora visualizamos uma segunda situação em que temos uma bicicleta capaz de carregar esses 3 itens juntos, que ultrapassa carros parados em ruas congestionadas e facilmente estaciona em frente às localidades onde necessita fazer as entregas. Comparando essas duas situações hipotéticas, mas realistas, fica claro quem irá completar o serviço primeiro. Um dos desafios de um bom serviço de logística é oferecer previsões de entrega confiáveis. As bicicletas podem oferecer uma melhor previsibilidade e satisfação dos clientes em alguns cenários.

Há ainda uma outra vantagem que as bicicletas trazem para os serviços logísticos. Bicicletas custam menos. O preço de aquisição é muito menor, os custos de manutenção e seguro são menores, e não há consumo de combustível, o que é uma grande vantagem em termos de custo operacional. Além do mais, as bicicletas se constituem numa alternativa "verde" para serviços de entrega logística. Elas não aumentam a poluição nas cidades, o que por outro lado não é o caso com vans, carros e caminhões dirigindo de maneira muito ineficiente por ruas congestionadas. Assim, as bicicletas ajudam a não aumentarem as emissões de $CO_2$.

Considerando todos os fatores mencionados acima, as bicicletas trazem ganhos que as tornam uma solução viável para alguns cenários.

Diante do exposto, esse trabalho explora a viabilidade da utilização de bicicletas em serviços logísticos, aborda o problema do roteamento de veículos para algumas de suas variantes, discute como essas variantes podem ser modificadas para o emprego de bicicletas diante de um esforço em subidas íngremes, e explora técnicas para o melhoramento da qualidade da solução do problema por meio de meta-heurísticas.

## 1.2 Objetivos do trabalho

O presente trabalho tem por objetivo o desenvolvimento de um modelo de planejamento de entregas que utilize bicicletas e que leve em conta possíveis restrições que sejam particulares a essa modalidade de entrega. Para atingir esse objetivo, alguns objetivos secundários são abordados e listados abaixo:

- Analisar a bibliografia existente sobre utilização de bicicletas como solução logística e sua viabilidade ou limitações. Com base nessa análise, mostrar que as bicicletas não são uma solução tão limitada como talvez pareça num olhar inicial.
- Apresentar modelagem matemática do problema clássico do roteamento de veículos com algumas de suas variações como o problema básico apenas com restrição de capacidade, sua expansão para a inclusão de janelas de tempo e frota híbrida.
- Propor uma adaptação do problema de roteamento para a utilização de bicicletas que leve em consideração o esforço adicional do ciclista em percorrer algumas rotas em função da inclinação da subida. Dependendo da inclinação, uma rota pode até se tornar inviável. Com essa restrição, o problema deverá buscar uma solução que suavize possíveis subidas.
- Estudar a implementação desses modelos matemáticos numa linguagem de programação moderna como o Python com o emprego de *solvers*.
- Analisar soluções heurísticas e/ou meta-heurísticas para a solução do problema de roteamento de veículo básico (apenas com restrição de capacidade), como a heurística do algoritmo de Clarke & Wright e as meta-heurísticas como busca-tabu, algoritmo genético e colônia de formigas (CLARKE; WRIGHT, 1964).
- Escolher uma meta-heurística e explorar técnicas de melhoramento da qualidade da solução bem como explorar sua implementação para as demais variantes do problema de roteamento - janelas de tempo e frota híbrida.

A principal justificativa para a seleção desse tema é buscar desenvolver a compreensão de métodos de otimização combinatória aplicados num problema atual e presente em muitas empresas, e com um apelo da cada vez mais presente que é a preocupação com a sustentabilidade e a preservação do nosso planeta.

### 1.3 Visão geral da dissertação

O Capítulo de Revisão da Literatura foi escrito em três partes. A primeira parte se concentrou em investigar a viabilidade do emprego de bicicletas nos serviços logísticos, em que situações isso pode ocorrer, em que situações as bicicletas podem ser uma solução melhor do que os veículos motorizados tradicionais, e quais vantagens podem oferecer. A segunda parte se concentrou no estudo do problema do roteamento de veículos, seu histórico, suas variantes, e seu modelamento matemático. A terceira parte abordou a busca de soluções para o problema, com um foco em soluções heurísticas e meta-heurísticas (em particular busca tabu, algoritmo genético e colônia de formigas).

Na sequência, o Capítulo sobre Metodologia foi elaborado compreendendo dois tópicos básicos que dão o alicerce para todos os experimentos conduzidos:

- Modelamento matemático:
    - Apresentação dos modelos matemáticos para o problema de roteamento de veículo com restrições de capacidade, janelas de tempo e frota híbrida.
    - Proposta de um modelo para o problema considerando o emprego de bicicletas com um esquema de penalização em função do ângulo de subida de um cliente para outro.
- Soluções heurísticas e meta-heurísticas:
    - Explanação sobre a importância de soluções heurísticas.
    - Discussão sobre a heurística de Clarke & Wright (CLARKE; WRIGHT, 1964).
    - Apresentação e discussão das meta-heurísticas de busca tabu, algoritmo genético e colônia de formigas.
    - Discussão sobre como aprimorar a qualidade da solução da meta-heurística do algoritmo genético para o problema do roteamento.

Os Capítulos de Experimentos, Resultados e Discussões, apresentam os resultados das várias implementações feitas durante a execução desse trabalho com o objetivo de validar os modelos matemáticos, mostrar a capacidade das soluções meta-heurísticas em entregar uma solução relativamente boa para o problema, e mostrar o aprimoramento da qualidade da solução com o emprego de técnicas adicionais. Como ferramenta para a condução dos experimentos utilizou-se a linguagem de programação Python. Para a realização de experimentos lançou-se mão

de instâncias conhecidas na literatura para o problema de roteamento, que estão disponíveis na Internet e que possuem soluções ótimas já conhecidas, além da geração de dados aleatórios permitindo a criação de instâncias próprias para esse trabalho.

Tanto o Capítulo sobre a Metodologia como o sobre Experimentos apresentam o problema de forma gradual, iniciando com sua versão mais básica e gradualmente aumentando sua complexidade por incluir novas variantes. Com isso, essa dissertação apresenta o problema de roteamento de veículos de uma forma didática, culminando com a proposta do modelo que engloba três variantes clássicas do problema (restrição de capacidade de veículo, janelas de tempo, frota híbrida) mais a proposta do modelo de penalizações para subidas íngremes a ser aplicado para bicicletas. Essa abordagem didática foi utilizada tanto para a apresentação dos modelos matemáticos do problema de roteamento, como também para o desenvolvimento das meta-heurísticas.

Finalmente, a conclusão do trabalho fecha essa dissertação com as principais conclusões obtidas e abrindo a porta para futuras investigações.

Até onde o autor pesquisou o tema, essa dissertação traz uma originalidade para o problema de roteamento de veículos por (1) apresentar uma proposta de penalidades em função do ângulo de subida que pode existir em uma rota e incorporar esse cenário no modelamento matemático, e (2) modificar o algoritmo *Split* de Prins (2004) para um problema de roteamento que também possui janelas de tempo, frota mista e penalidades de subida.

# CAPÍTULO 2 REVISÃO DA LITERATURA

## 2.1 Emprego de bicicletas como alternativa para serviços logísticos

Há uma boa quantidade de referências sobre pesquisas feitas com o objetivo de analisar a viabilidade do emprego de bicicletas para entregas logísticas. Estudos feitos estimam que cerca de 25% de todos os produtos e cerca de 50% de todos os produtos leves podem migrar dos veículos tradicionais de entrega para bicicletas nas áreas urbanas (FERGUNSON, 2012). Em algumas cidades esse potencial pode ser ainda maior. Um outro estudo indicou que o peso médio transportado por veículos motorizados em cidades europeias é menor que 100kg, enquanto bicicletas podem carregar pesos até acima de 200kg dependendo do tipo da bicicleta empregada (DE DECKER, 2012). Esse estudo quebra o paradigma que bicicletas servem apenas para volumes pequenos ou de peso leve. Algumas bicicletas têm sido desenhadas para uma boa capacidade de carga. Portanto, dependendo do tipo de bicicleta empregado, o potencial de mover volumes de vans para bicicletas pode ser consideravelmente alto. Esse mesmo estudo também indicou que bicicletas podem custar 98% menos que vans, cobrindo não apenas o preço de aquisição, mas também todos os custos associados para ambos os tipos de veículos. Um outro estudo comparou alguns cenários e demonstrou que bicicletas adaptadas para o transporte de cargas são mais eficazes em termos de custo do que vans e caminhões quando se trata de entregas acontecendo numa área próxima do centro de distribuição enquanto que vans e caminhões são uma solução melhor para os cenários de longas distâncias e entregas de grande volume numa única parada (SHETH *et al*., 2019).

Há também a preocupação de que ao se utilizar bicicletas teríamos um aumento do custo de pessoas envolvidas porque um número maior de bicicletas seria necessário para substituir uma única van, mas isso não é uma regra definida e pode ser visto como mais um paradigma a ser quebrado. Quando consideramos o ambiente de entregas dentro de cidades, bicicletas podem ser mais rápidas e assim realizar tantas entregas quanto uma van. Um estudo analisou a diferença no tempo de viagem entre bicicletas e carros no ambiente de muitas cidades e constatou que cerca de metade das entregas podem mudar de carros para bicicletas com um atraso de não mais que 10 minutos (GRUBER e NARAYANAN, 2019). Além disso, o custo da entrega na ponta (“*last mile*”) corresponde a 28% do custo total de entrega e é percebido como o estágio menos eficiente da cadeia de suprimentos (RAINERI *et al*.,

2018). Assumindo que um pouco mais que metade da população vive em cidades e que a tendência é que essa parcela aumente nos próximos anos, então buscar soluções e oportunidades para melhorar a eficiência na entrega na ponta é algo que não pode ser ignorado (NAUMOV e PAWLUS, 2021).

Reconhecendo que hoje há uma preocupação, e extremamente correta, com questões de sustentabilidade, então há ainda outro fator favorável para a utilização de bicicletas. O setor de transportes responde por aproximadamente 23% das emissões globais de $CO_2$. As bicicletas são uma alternativa “verde” que pode ajudar muito a reduzir essas emissões (MASSINK *et al.*, 2011).

Além dos benefícios acima mencionados como custos, velocidade, previsibilidade e sustentabilidade, também podemos mencionar os benefícios de saúde para os ciclistas. O exercício físico produzido por se andar de bicicleta tem inúmeros benefícios para a capacidade cardiovascular, melhora da resistência física e imunológica, e até ajuda no combate da depressão e obesidade.

Uma boa revisão de referências bibliográficas na utilização de bicicletas para entregar produtos em diferentes cidades e com diferentes abordagens é indicada em (VASIUTINA *et al.*, 2021).

Com base em todos esses pontos acima considerados, podemos olhar o cenário de bicicletas para a distribuição de mercadorias para vários clientes espalhados numa determinada vizinhança e partindo de um centro de distribuição como origem. Trata-se do clássico problema de roteamento de veículos amplamente estudado nos domínios da pesquisa operacional e da otimização combinatória. Há a necessidade de alguma modificação para que o problema seja adaptado para bicicletas? A resposta vai depender da topologia geográfica da região. Uma região com muitas subidas e algumas bem íngremes poderia dificultar o trajeto por exigir um esforço físico em demasia por parte do ciclista. Dependendo de quão íngreme for a subida, talvez o próprio trajeto seja impraticável para um ciclista. Num caso como esse, talvez uma rota de um ponto A para um ponto B não seja factível. Por outro lado, é possível desenhar uma rota que conecte os pontos A e B mas passando por outros pontos intermediários onde a subida não seja tão íngreme e que possa surgir como uma alternativa mais viável. Nesse trabalho apresentaremos uma metodologia simples que pode ser facilmente adaptada e inserida na formulação dos problemas de roteamento e que levaria em conta a dificuldade em subidas mais acentuadas.

## 2.2 Problema do roteamento de veículos: histórico, variações, modelos matemáticos

O problema do roteamento de veículos foi introduzido em 1959 por Dantzig e Ramser, o que abriu a porta para uma extensa pesquisa sobre esse problema (DANTZIG e RAMSER, 1959). De fato, esse problema de otimização combinatória com suas inúmeras variações tem uma ampla gama de aplicação prática no universo da logística e se tornou um dos problemas mais explorados no campo da pesquisa operacional.

Podemos analisar o problema de roteamento de veículos clássico – PRV (ou VRP do inglês *vehicle routing problem*) com a ajuda de um grafo *G = (V, A)*. Nesse grafo temos o conjunto de vértices *V* = {0,1,2,…,*n*} onde 0 representa o centro de distribuição (ou depósito) e de onde todos os veículos partem e finalmente retornam. Todos os outros vértices representam os clientes e cada um desses clientes possui uma demanda $q_i$ associada com ele. Além do conjunto de vértices, temos o conjunto de arcos $A = \{(i,j): i,j \in V, i \neq j\}$ que contém todas a conexões entre os vértices, ou seja, todos os caminhos ligando um cliente diretamente a todos os demais vértices e inclusive ao centro de distribuição. Cada veículo tem uma capacidade de carga máxima *Q* e uma premissa importante é que $q_i \leq Q, \ \forall\, i \in V/\{0\}$, que significa que a demanda de cada cliente não pode exceder a capacidade do veículo. O PRV é um problema NP-difícil e que lembra o problema clássico do caixeiro viajante, ou *traveling salesman problem* conforme a literatura inglesa (EL-SHEBERNY, 2010). O problema do caixeiro viajante pode ser visto como um caso particular do problema de roteamento de veículos onde se é empregado apenas um único veículo e com capacidade infinita. Dessa forma, esse único veículo percorre uma rota que passa por todos os clientes.

O PRV é um problema que tem por objetivo encontrar uma programação de rotas partindo do depósito de forma a atender a demanda de todos os clientes e minimizando o custo total (ou tempo total, ou distância total). Algumas restrições são impostas:

- Cada cliente é atendido uma única vez e por um único veículo;
- Os veículos têm o depósito como ponto inicial e como ponto final de suas rotas;
- A demanda total dos clientes atendidos numa rota não pode exceder a capacidade do veículo.

Há variações bem conhecidas do problema, mas todas respeitando as três restrições acima.

- Restrição de capacidade

Na formulação básica, o problema conforme descrito acima é conhecido como PRVC (problema de roteamento de veículos capacitado, ou CVRP – *capacitated vehicle routing problem*).

- Janelas de tempo

Nessa variação, cada cliente *i* possui uma janela de tempo $[a_i, l_i]$ bem como um tempo de serviço $s_i$. A partir da matriz de distâncias $c_{ij}$ podemos facilmente calcular uma matriz de tempos $t_{ij}$, que representa o tempo de deslocamento entre um cliente *i* e um cliente *j,* assumindo uma velocidade média constante. Logicamente elaborações mais complicadas podem ser feitas com outras premissas para assumir velocidades e tempos de deslocamentos diferentes dependendo da situação. Para respeitar a janela de tempo, o veículo somente pode iniciar o serviço de descarga num cliente *i* a partir do momento $a_i$, o tempo de serviço é $s_i$. O fim da janela de tempo é $l_i$, onde podemos equacionar o problema tanto para que o veículo não possa chegar mais no cliente a partir desse momento (embora possa chegar pouco antes dele e continuar o serviço de descarga mesmo após ele) como também para que o serviço de descarga seja todo executado antes dele.

- Frota híbrida

Nas variações anteriores assumíamos um mesmo tipo de veículo para todas as rotas, ou seja, mesma capacidade máxima, mesma velocidade, mesmo custo. A variação com a frota híbrida permite usar veículos que possuam capacidades diferentes, velocidades diferentes, custos fixos diferentes, custos variáveis diferentes. Nessa dissertação apresentaremos uma situação em que podemos dispor de uma quantidade de bicicletas e de uma quantidade de motocicletas, e o problema irá buscar uma solução que minimize o custo total empregando um *mix* dos dois tipos de veículos e escolhendo que rotas deverão percorrer.

- Múltiplos depósitos

Nas variantes acima considerávamos um único depósito a partir do qual todos os veículos partiam. Em algumas situações práticas podemos ter mais que um depósito disponível. Nessa formulação o problema passa a buscar uma solução que

minimize o custo total escolhendo de que depósito o veículo deve partir, sempre retornando para o mesmo depósito de partida.

Toth e Vigo (2002) apresentam a formulação matemática para o problema com restrição de capacidade com variável $x_{ij}$ com dois índices e equações de Miller-Tucker-Zemlin (conhecidas como equações MTZ) para eliminação de sub-rotas. Trata-se de um modelo simples onde o número de veículos utilizados é obtido por inspeção sobre o resultado. Cordeau e*t al*. (2002) apresentam a formulação do problema com janelas de tempo usando a variável $x_{ijk}$ com três índices, onde um deles é utilizado para a identificação do veículo (ou da rota). Baldacci *et al*. (2008) trazem a formulação matemática para o problema com frota híbrida. Afshar-Nadjafi e Afshar-Nadjafi (2014) bem como Hanum *et al.* (2018) apresentam a formulação para o problema com multi-depósitos mantendo também incorporadas as variantes de frota heterogênea e janelas de tempo.

## 2.3 Métodos de resolução

O problema de roteamento tem sido amplamente explorado e a literatura disponível aborda as variações do problema de roteamento descritas acima além de outras, e também oferece uma vasta gama de abordagens de resolução (TOTH e VIGO, 2002; LAPORTE, 2007; LAPORTE *et al*., 2013; GOEL e MAINI, 2017). Há vários exemplos de resolução por métodos exatos bem como por métodos heurísticos. Sobre métodos exatos temos várias técnicas que podem ser empregadas como o tradicional *branch & bound*, como *branch & cut* que adiciona a técnica de planos de corte, e como o *branch & price* que utiliza a técnica de geração de colunas.

A grande vantagem dos métodos exatos é sua capacidade de poder garantir uma solução ótima para a instância de um problema. Por outro lado, na medida em que o tamanho da instância cresce, o tempo de execução pode crescer demasiadamente inviabilizando o emprego do método exato. Para fazer frente a essa limitação dos métodos exatos, temos os métodos heurísticos. Os métodos heurísticos não garantem a solução ótima, mas podem chegar a uma solução razoavelmente boa em um período curto. Considerando a realidade de muitos problemas de negócios da atualidade, executar uma solução que seja ótima, mas que leve muito tempo para ser encontrada não é tão importante quanto executar logo uma solução que seja boa. Aliás, nesses ambientes costuma-se dizer que “o ótimo é inimigo do bom”. Assim, em

muitas empresas a busca por soluções boas, mas que sejam obtidas rapidamente é primordial. Juntando isso com o fato de que geralmente os problemas reais de muitas empresas envolvem uma grande quantidade de dados e variáveis, podemos dizer que os métodos exatos muitas vezes não irão satisfazer às expectativas. Justamente por esse fato o campo das heurísticas e das meta-heurísticas tem recebido bastante atenção e muitas pesquisas têm sido realizadas para explorar seu potencial.

A heurística de Clarke e Wright, ou algoritmo de *savings*, é conhecida como uma das primeiras heurísticas desenvolvidas para o problema do roteamento de veículos e até hoje é empregada como um método de resolução, ou parte de um algoritmo maior (como uma meta-heurística), devido à sua simplicidade, rapidez e boa eficiência (CLARKE; WRIGHT, 1964). Adaptações dessa heurística também foram desenvolvidas para abordar as características pertinentes às variações do problema original de roteamento. O trabalho de Atinel e Öncan (2005) expandiu a formulação de Clarke e Wright por adicionar parâmetros que levam em conta não só a importância relativa de um arco unindo dois clientes, mas também a assimetria das distâncias do depósito aos dois clientes a serem unidos, e ainda o benefício de se utilizar melhor a capacidade do veículo. No entanto esses três parâmetros necessitam ser ajustados e o procedimento realizado foi enumerativo variando-se seus valores com passos fixos dentro de um intervalo. Battara *et al*. (2008) se valeu de um algoritmo genético para rapidamente gerar o vetor com os três parâmetros para utilização dessa heurística melhorada.

Com relação ao problema de roteamento básico, com a restrição de capacidade, Gendreau *et al*. (1994) apresentam uma implementação desse problema através da busca tabu e com resultados bem satisfatórios. Em Simas e Gómez (2005) temos um algoritmo de busca tabu que explora a vizinhança da solução através de três tipos de movimentos possíveis, em cada um deles a busca tabu é realizada levando-se em conta movimentos tabu e critérios de aspiração. A força do algoritmo de busca tabu vem justamente de explorar diversificação e intensificação ao longo de sua execução. Prins (2004) indicou que as implementações de busca tabu se mostravam superiores em desempenho quando comparadas com os algoritmos genéticos e formulou um algoritmo genético que reverteu essa situação e demonstrou superioridade a muitos dos algoritmos em busca tabu até o momento. A solução de Prins baseia-se no conceito de *route-first, cluster-second*, onde ele codifica o cromossomo do algoritmo genético como uma sequência de clientes sem

delimitadores de rotas. Essa é a sequência de clientes percorrida no que ele chama de *giant-tour*. Depois ele aplica um algoritmo *Split* que define as rotas nessa sequência. A grande vantagem desse algoritmo *Split* é que acaba definindo para essa sequência qual é a configuração ótima de rotas. Ao fazer isso, nós temos um procedimento de otimização incluído dentro do algoritmo genético, o que aumenta consideravelmente sua eficiência. Além disso, o fato de o cromossomo não possuir delimitadores de rota torna a implementação dos procedimentos de *crossover* e mutação do algoritmo genético muito mais fáceis de serem implementados. Ainda outra vantagem é que uma codificação do cromossomo com delimitadores implica em adicionar um teste de viabilidade na solução, por outro lado a codificação sem delimitadores torna esse teste desnecessário já que o algoritmo se encarrega de fazer a tradução do cromossomo em rotas que sejam viáveis, ou seja, sem violação de restrições. Esse algoritmo foi utilizado nos experimentos dessa dissertação. O algoritmo de Prins se tornou uma base para trabalhos posteriores. Como exemplo, Néia *et al*. (2013) aplicam essa solução para a implementação de um algoritmo genético e resolvem o problema do cruzamento de cromossomos com o emprego de operadores PMX (*partially matched crossover*), OX (*order crossover*) e CX (*cycle crossover*) que realizam o *crossover* e ao mesmo tempo executam um procedimento de reparo. Em adição ao algoritmo genético, seu trabalho também inclui uma análise da meta-heurística de colônia de formigas para o mesmo problema. Liu *et al*. (2009) modificaram o algoritmo *Split* de Prins (2004) para utilizá-lo num algoritmo genético para solução do problema com frota heterogênea. Nesse caso, além da decodificação da sequência de clientes em rotas, o algoritmo também informa qual o tipo de veículo empregado.

Ombuki *et al*. (2002) propuseram um algoritmo genético híbrido onde também se utiliza uma codificação de cromossomo para o problema de roteamento sem delimitadores de rotas, porém mais simples. A sequência de clientes é definida no cromossomo. Partindo do depósito, o primeiro cliente no cromossomo é o primeiro cliente a ser visitado pela primeira rota. Se a adição do cliente seguinte do cromossomo nessa rota não violar a restrição de capacidade e nem de janela de tempo, então esse cliente é adicionado nessa rota. Caso contrário se inicia uma nova rota tendo a ele como cliente inicial. Esse processo continua até o fim do cromossomo. Trata-se de uma implementação simples para se traduzir a sequência do cromossomo em rotas factíveis para o problema. Embora seja uma solução prática, rápida e com

bons resultados, esse método não contém a grande vantagem do algoritmo proposto por Prins que fornece a solução de rotas ótima para a sequência do cromossomo.

O problema do roteamento de veículos também pode ser desenhado com a existência de mais de um depósito de onde partem as rotas. Trata-se da variante de múltiplos depósitos. Rybicková *et al*. (2015) apresentaram um algoritmo genético para resolver um cenário de distribuição de peças de reposição para garagens na República Tcheca. A codificação do cromossomo é um pouco mais complexa e pode ser vista na Figura 1 abaixo. Nessa figura cada linha representa uma rota diferente (ou veículo), a primeira coluna representa o número do depósito, e na sequência aparecem os clientes visitados por essa rota. Como pode-se imaginar, a operação de *crossover* também é um pouco mais complexa. A Figura 2 ilustra dois cromossomos pais. Um trecho do Pai 1 é selecionado e então inserido no cromossomo do Pai 2 para formar o cromossomo Filho. Logicamente, após essa inserção nós temos dois genes duplicados em outras posições e que devem ser deletados. A designação dos depósitos é a mesma do cromossomo pai, mas após essa etapa verifica-se para cada rota alterada, seja pela inserção ou pela remoção de genes duplicados, qual o depósito que apresenta a menor distância (ou custo) total para a rota e muda-se para o depósito que apresentar a melhor solução.

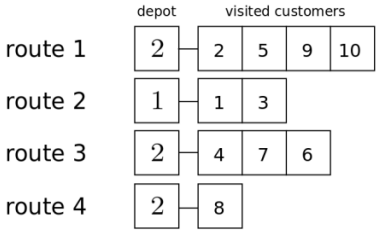



Figura 1 – Codificação do cromossomo, extraído de Rybicková (2015).

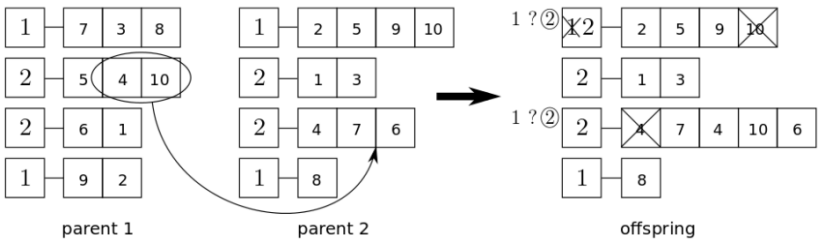



Figura 2 - Operação de *crossover*, extraído de Rybicková (2015).

No entanto, a solução acima apresenta maior complexidade para implementação computacional. Ombuki-Berman e Hanshar (2009) apresentam uma solução sem a delimitação das rotas na estrutura do cromossomo. Conforme a Figura 3, a estrutura do cromossomo é bem mais simples, basicamente uma sequência de clientes para cada depósito do problema. Um *route scheduler* é criado para montar as rotas caminhando-se pela sequência de clientes e respeitando as restrições de limite de capacidade ou limite de distância.

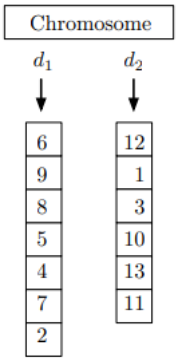



Figura 3 - Codificação do cromossomo, extraído de Ombuki-Berman e Hanshar (2009).

Uma variação importante do problema de roteamento e com grande aplicação prática tem a ver com a inclusão de janelas de tempo para entregas nos clientes. Blanton e Wainwright (1993) elaboraram uma proposta para operadores de *crossover* (MX1 e MX2) para o emprego em algoritmos genéticos. Basicamente, a ideia envolve selecionar dois cromossomos pais e executar o *crossover* por compará-los com um vetor de precedência. Para o problema do roteamento com janelas de tempo, esse vetor pode ser facilmente construído ordenando-se os tempos de início de janela ou os tempos de fim de janela. Na realidade, o algoritmo genético pode ser executado usando essas duas possibilidades intercambiando-as ao longo das iterações. A grande vantagem que esse método traz é que a operação de *crossover* não se torna meramente aleatória, mas tenha consigo uma lógica de busca de viabilidade de solução. Com isso, os movimentos de *crossover* tendem a buscar soluções viáveis em relação às restrições de janelas de tempo. Tan *et al*. (2001) documentam uma interessante investigação sobre meta-heurísticas desenvolvidas para a solução do problema com janelas de tempo, como a busca local, *simulated annealing*, busca tabu

e algoritmo genético. Costa *et al*. (2005) apresentam uma metodologia que parte de uma solução com base no algoritmo de Clarke e Wright e que depois é refinado com a busca tabu. Mais recentemente, Mirshekarian *et al*. (2015) implementaram uma adaptação da heurística de Clarke e Wright para uma solução rápida e razoável para o problema de roteamento com janelas de tempo, o que confirma que mesmo com 50 anos de existência a heurística de economias (*savings*) de Clarke e Wright não ficou ultrapassada.

A meta-heurística de colônia de formigas também tem sido explorada para a busca de soluções de qualidade para o problema de roteamento. Dorigo *et al*. (1991) abordaram esse tema com a proposta do modelo chamado *Ant System* (AS) com aplicação no problema do caixeiro viajante. Bullnheimer *et al*. (1997) empregaram a meta-heurística da colônia de formigas pela primeira vez para resolver o problema do roteamento de veículos e seu trabalho serviu como base para outros que se seguiram. Gambardella *et al*. (1999) apresentam um algoritmo de colônia de formigas para o problema do roteamento com a adição de janelas de tempo. Santos e Leal (2007) fazem um retrospecto e análise dos modelos de *Ant System* (AS), *Ant Colony System* (ACS) e *Multiple Ant Colony System* (MACS) e discutem um algoritmo que é iniciado com uma heurística de *Nearest Neighbor* e depois repete sucessivamente ciclos com um algoritmo de colônia de formigas buscando minimizar o número de veículos e outro algoritmo de colônia de formigas buscando minimizar o tempo total do deslocamento que se revezam entre si. Além disso, os resultados de cada execução de um desses algoritmos influenciam na execução do outro.

Nessa dissertação abordaremos as meta-heurísticas de busca-tabu, algoritmos genéticos e colônia de formigas. Uma vez analisadas, iremos descrever como implementá-las no problema de roteamento de veículos básico, com apenas a restrição de capacidade. Selecionaremos uma meta-heurística para ser aprimorada com base em algumas técnicas apresentadas nesse Capítulo. Essa meta-heurística será desenvolvida para resolver as demais variantes do problema de roteamento levando ao problema completo, com restrição de capacidade, janelas de tempo, frota heterogênea de bicicletas e motocicletas com penalidades para subidas íngremes.

# CAPÍTULO 3 METODOLOGIA

O foco deste Capítulo é detalhar como o problema de roteamento de veículos pode ser resolvido. Basicamente, a metodologia está dividida em duas partes:

- Apresentação de modelos matemáticos para serem implementados através de programas em Python e resolvidos com a utilização de *solvers*, bem como a adaptação desses modelos para contemplarem o cenário de bicicletas com a penalização de subidas íngremes.
- Soluções heurísticas e meta-heurísticas para a obtenção de soluções bem como o emprego de técnicas para melhorar a qualidade da solução.

## 3.1 Modelamento matemático

### 3.1.1 Problema com restrição de capacidade

O problema do roteamento de veículos com a restrição de capacidade (PRVC), ou como consta na bibliografia em inglês "*capacited vehicle routing problem*" (CVRP), considera que os veículos partem de um depósito através de rotas que percorrem os *N* clientes e sem exceder a capacidade *Q* do veículo. O problema faz uma distinção entre dois conjuntos: o conjunto de clientes {1,...,*N*}, onde *N* é o número de clientes, e o conjunto de nós que engloba todos os clientes incluindo o depósito. Assumimos 0 como o índice do depósito e com isso temos o conjunto de nós {0,...,*N*}.

Para o modelamento matemático, temos as seguintes variáveis:

$x_{ij}$ – Variável binária que vale 1 se houver uma rota ligando o cliente *i* ao cliente *j*, e 0 caso contrário;

$u_i$ – Variável auxiliar que acumulará a capacidade do veículo utilizada ao longo da rota.

Temos os seguintes parâmetros:

$c_{ij}$ – Parâmetro que indica o custo (ou distância) entre o cliente *i* e o cliente *j*;

$q_i$ – Parâmetro que indica a demanda exigida pelo cliente *i*.

Temos as seguintes constantes:

*Q* – Constante que indica a capacidade do veículo;

*N* – Constante que indica o número de clientes;

*M* – Número suficientemente grande para satisfazer uma restrição auxiliar.

O modelamento matemático é apresentado a seguir.

$$\text{Minimizar} \quad z = \sum_{i=0}^{N} \sum_{j=0, i\neq j}^{N} (c_{ij} x_{ij}) \tag{1}$$

Sujeito a:

$$\sum_{j=0}^{N} x_{ij} = 1 \qquad \forall\, i \in \{1, \dots, N\}, \quad i \neq j \tag{2}$$

$$\sum_{i=0}^{N} x_{ij} = 1 \qquad \forall\, j \in \{1, \dots, N\}, \quad i \neq j \tag{3}$$

$$\left.\begin{aligned} u_j &\geq u_i + q_j - (1 - x_{ij})M \\ u_j &\leq u_i + q_j + (1 - x_{ij})M \end{aligned}\right\} \quad \forall\, i, j \in \{1, \dots, N\}, \quad i \neq j \tag{4}$$

$$q_i \leq u_i \qquad \forall\, i \in \{1, \dots, N\} \tag{5}$$

$$0 \leq u_i \leq Q \qquad \forall\, i \in \{0, \dots, N\} \tag{6}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i, j \in \{0, \dots, N\}, \quad i \neq j \tag{7}$$

A Equação (1) define a função objetivo que busca minimizar o custo das rotas selecionadas. O parâmetro $c_{ij}$ é multiplicado pela variável $x_{ij}$ para o cálculo do custo do trajeto. Esse custo pode ser convertido na própria distância entre esses dois pontos e aí o problema transforma-se em minimizar a distância total percorrida pelas rotas.

A Equação (2) garante que de cada cliente saia uma única rota para outro cliente ou para o depósito como ponto final da rota. Por outro lado, a Equação (3) garante que para cada cliente chegue apenas uma única rota vinda de outro cliente ou do depósito. A duas equações juntas garantem que todos os clientes sejam visitados e que isso ocorra apenas uma vez. Assim, o mesmo cliente não poderá pertencer a mais de uma rota. Entretanto, a maneira como as equações foram construídas permitem a saída bem como a chegada de múltiplas rotas no depósito.

As Equações (4), (5) e (6) trabalham juntas utilizando uma variável auxiliar *u* que será utilizada para calcular a capacidade utilizada por cada veículo na medida em que se desloca por cada cliente ao longo da rota. A demanda de cada cliente *i* dada

pelo parâmetro $q_i$ deve obrigatoriamente ser inferior à capacidade do veículo dada pela constante *Q*. Na Equação (4) *M* é uma constante de valor grande para satisfazer a relação quando $x_{ij}$ for 0. Quando a variável for 1, a equação anula *M* multiplicando-o por 0. Essa equação também garante a eliminação de sub-rotas uma vez que ao se fechar uma sub-rota entre clientes apenas, sem passar pelo depósito, nós teremos uma violação da equação. A Equação (6) garante que a capacidade do veículo não será excedida ao limitar o valor máximo da variável auxiliar *u* em *Q*.

Não há imposição da variável auxiliar *u* ser discreta, assim ela é contínua e o problema deixa de ser de programação inteira para ser de programação mista.

A condição (7) define a integralidade do problema através do caráter binário da variável $x_{ij}$ que poderá assumir como valores apenas 0 ou 1.

Não há uma variável para registrar o número de veículos empregados, assim esse número é determinado por inspeção na solução.

### 3.1.2 Problema com restrição de capacidade e janelas de tempo

Basicamente, essa variação do VRP contempla janelas de tempo que definem que um veículo só poderá servir um cliente dentro de uma janela especificada de antemão. Na implementação nesse trabalho um veículo pode chegar num cliente antes que sua janela de tempo se inicie, porém deverá esperar até o início da janela para que a operação de serviço ao cliente comece a ser executada. A formulação do problema com janelas de tempo emprega variáveis, parâmetros e constantes já descritos na Seção anterior e traz novos que estão descritos a seguir.

Variáveis para a formulação com janelas de tempo:

$x_{ijk}$ – Variável binária que vale 1 se houver uma rota ligando o cliente *i* ao cliente *j* sendo percorrida pelo veículo *k,* e 0 caso contrário*;*

$b_{ik}$ – Variável para armazenar o momento do início do atendimento do cliente *i* pelo veículo *k.*

Parâmetros adicionais para janelas de tempo:

$s_i$ – Parâmetro dado para indicar o tempo de atendimento no cliente *i*;

$t_{ij}$ – Parâmetro dado (ou calculado) para indicar o tempo de deslocamento entre o cliente *i* e o cliente *j*;

$e_i$ – Parâmetro dado para indicar o início da janela de tempo em que o cliente *i* pode ser atendido;

$l_i$ – Parâmetro dado para indicar o término da janela de tempo em que o cliente *i* pode ser atendido ou receber um veículo.

Constante necessária para essa implementação com janelas de tempo:

*K* – Constante para armazenar o número de veículos.

O modelo matemático é apresentado abaixo:

$$\text{Minimizar} \quad z = \sum_{i=0}^{N} \sum_{j=0,\, i\neq j}^{N} \sum_{k=1}^{K} (c_{ij} x_{ijk}) \tag{8}$$

Sujeito a:

$$\sum_{k=1}^{K} \sum_{j=1}^{N} x_{ijk} = 1 \qquad \forall\, i \in \{1, \dots, N\}, \quad i \neq j \tag{9}$$

$$\sum_{k=1}^{N} \sum_{i=1}^{N} x_{ijk} = 1 \qquad \forall\, j \in \{1, \dots, N\}, \quad i \neq j \tag{10}$$

$$\sum_{j=1}^{N} x_{0jk} = 1 \qquad \forall\, k \in \{1, \dots, K\} \tag{11}$$

$$\sum_{i=1}^{N} x_{i0k} = 1 \qquad \forall\, k \in \{1, \dots, K\} \tag{12}$$

$$\sum_{i=0, i\neq h}^{N} x_{ihk} - \sum_{j=0, j\neq h}^{N} x_{hjk} = 0 \qquad \forall\, k \in \{1, \dots, K\},\ \forall\, h \in \{1, \dots, N\} \tag{13}$$

$$\sum_{i=1}^{N} q_i \sum_{j=0, i\neq j}^{N} x_{ijk} \leq Q \qquad \forall\, k \in \{1, \dots, K\} \tag{14}$$

$$b_{ik} + s_i + t_{ij} - M(1 - x_{ijk}) \leq b_{jk} \qquad \begin{cases} \forall\, i \in \{0, \dots, N\}, \\ \forall\, j \in \{1, \dots, N\}, \\ \forall\, k \in \{1, \dots, K\}, \\ i \neq j \end{cases} \tag{15}$$

$$s_0 = 0, \qquad b_{0k} = 0 \tag{16}$$

$$e_i \leq b_{ik} \leq l_i \qquad \forall\, i \in \{1, \dots, N\}, \forall\, k \in \{1, \dots, K\} \tag{17}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i, j \in \{1, \dots, N\},\ i \neq j,\ \forall\, k \in \{1, \dots, K\} \tag{18}$$

A Equação (8) é a função objetivo e nela temos a variável $x_{ijk}$ que passou a ser tridimensional com um índice para indicar o veículo *k* pertencente ao conjunto de veículos {1,...,*K*}.

A Equação (9) determina que de um cliente *i* sai somente uma rota para um cliente *j* através de um único veículo. Por outro lado, a Equação (10) determina que no cliente *j* chega somente uma rota de um cliente *i* para cada veículo *k*.

A Equação (11) determina que todos os veículos partam da origem estabelecida como 0. Já a Equação (12) determina que todos os veículos retornem a origem como destino final.

A Equação (13) garante que o fluxo de veículos que entra num cliente é o mesmo que sai desse cliente.

A Equação (14) garante que a somatória de todas as demandas $q_i$ de todos os clientes *i* atendidos pelo veículo *k* não ultrapassem a capacidade *Q* do veículo.

Com essas definições acima, as Equações (15), (16) e (17) garantem que as rotas bem como o número de veículos sejam calculados observando a restrição das janelas de tempo bem como que a formação de sub-rotas seja eliminada.

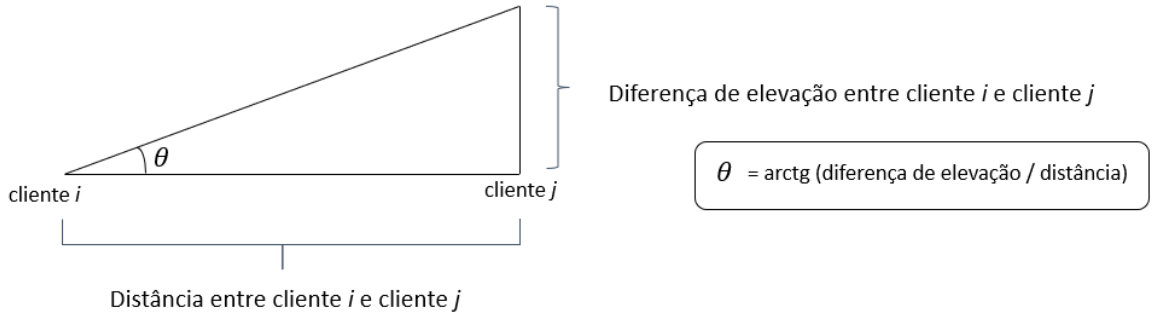
A condição (18) define a integralidade através do caráter binário de $x_{ijk}$.

Uma observação importante sobre esse modelo matemático é que ele pressupõe o conhecimento de *K*, ou seja, do número de veículos necessário para a solução do problema. Na prática, esse é um número que queremos descobrir. Isso pode ser acertado com um *loop* que comece com um valor de *K* que seja igual a somatória das demandas de todos os clientes dividida pela capacidade do veículo. Se o programa não encontrar uma solução, incrementa-se o valor de *K* até que o programa encontre uma solução ótima.

### 3.1.3 Modelo matemático com penalização de subidas

Uma vez que estamos considerando bicicletas como o tipo de veículo empregado, necessitamos considerar o impacto que uma subida poderia ter na escolha de uma rota. Dependendo da inclinação da subida o trajeto entre um cliente *i* e um cliente *j* poderá se tornar mais custoso ou até mesmo inviável. Dessa forma o modelo apresentado na Figura 4 sugere uma penalização que depende do grau de inclinação das subidas que podem existir de um cliente para outro. Essa penalização pode ser vista como se houvesse um incremento no custo ou na distância entre esses

dois clientes e que passa a ser levado em conta no cálculo da função objetivo. Dependendo da inclinação, a penalização poderá tornar um caminho inviável para um ciclista.




| | |
|---|---|
| $\theta \leq 2^o$ | Sem impacto na função objetivo |
| $2^o < \theta \leq 4^o$ | Função objetivo penalizada com acréscimo de 30% da distância entre os clientes *i* e *j* |
| $4^o < \theta \leq 6^o$ | Função objetivo penalizada com acréscimo de 70% da distância entre os clientes *i* e *j* |
| $\theta > 6^o$ | Restrição de impossibilidade de rota entre os clientes *i* e *j* |

Figura 4 - Modelo de penalidades para emprego de bicicletas.

Tendo a diferença de elevação entre dois clientes bem como a distância entre eles, podemos facilmente calcular o ângulo da inclinação. De posse do ângulo podemos ver como ele se encaixa nas faixas de penalização da Figura 4. Os valores limites para as faixas bem como a porcentagem de penalização foram definidas arbitrariamente com base em verificação de esforço do próprio autor para pedalar em subidas com alguns ângulos de inclinação. Logicamente numa aplicação prática esses valores podem variar em função do condicionamento físico do ciclista e podem ser calibrados para uma melhor aproximação. Faixas adicionais de ângulos com as respectivas penalizações também podem ser acrescentadas. Não é escopo desse trabalho fazer uma análise exaustiva desse esforço para apresentar ângulos e penalizações que reflitam uma realidade média com maior acuracidade.

Tomando por base o problema com a restrição de janelas de tempo, o modelo matemático pode ser adaptado para o problema com bicicletas e penalizações de subidas alterando-se apenas duas equações: a Equação (8), ou função objetivo, e a Equação (15).

No caso da Equação (8), ou função objetivo, temos a modificação:

$$\text{Minimizar} \quad z = \sum_{i=0}^{N} \sum_{j=0, i \neq j}^{N} \sum_{k=1}^{K} \left(c_{ijk} x_{ijk}\right) + \sum_{i=0}^{N} \sum_{j=0, i \neq j}^{N} \sum_{k=1}^{K} \left(p_{ijk} x_{ijk}\right)$$

Onde $p_{ij}$ representa uma matriz construída com as penalizações de acréscimo de distância em função do ângulo de subida. No caso do problema com restrição de capacidade, esse termo de penalidades acrescido na função objetivo é a única modificação na modelagem matemática que se mostra necessária. A única diferença em relação ao modelo de janelas de tempo é que no problema com restrição apenas de capacidade a variável $x_{ij}$ é bidimensional, sem o índice *k*.

A Equação (15) do problema com a restrição das janelas de tempo passa a ser:

$$b_{ik} + s_i + t_{ij} + \alpha . p_{ij} - M(1 - x_{ijk}) \leq b_{jk}$$

Basicamente o que está sendo feito aqui é adicionar um fator $\alpha . p_{ij}$ ao tempo $t_{ij}$ que é o tempo de deslocamento entre os clientes *i* e *j* para aumentar esse tempo caso haja a penalização do aumento de custo (ou distância) em função do aumento de elevação entre esses clientes. A constante $\alpha$ é utilizada para traduzir essa penalidade num aumento de tempo e assim poder ser somado ao parâmetro $t_{ij}$.

A proposta descrita nessa Seção considera o esforço adicional no caso de uma subida. É correto afirmar que numa descida haja uma redução do esforço por parte do ciclista. Tal redução pode ser considerada no mesmo modelo assumindo um valor de $p_{ij}$ negativo, de forma a reduzir o custo e até o tempo de deslocamento entre o cliente *i* e o cliente *j*. No entanto, na abordagem dada ao problema nessa dissertação assumimos que não há tal redução e que o comportamento da descida equivale ao de uma rota plana entre os dois clientes.

### 3.1.4 Problema com penalização de subidas e frota heterogênea

Uma formulação um pouco mais complexa do problema do roteamento com janelas de tempo e penalização de subidas pode ser feita considerando-se uma frota heterogênea. Passamos a ter um conjunto de veículos {1,...,*K*} onde cada veículo possui características próprias. No caso em questão o depósito pode dispor de uma frota de bicicletas e de motocicletas. Nessa formulação dependendo da escolha do veículo a ser utilizado, bicicleta ou motocicleta, teremos custos e capacidades diferentes para o veículo, tempos de traslados mais rápidos e sem penalização de subida caso a escolha seja feita para a motocicleta.

Variável adicional para a formulação com frota heterogênea:

$y_k$ – Variável binária que valerá 1 se o veículo *k* for utilizado e 0 caso contrário.

Parâmetros necessários para a formulação com frota heterogênea:

$f_k$ – Parâmetro que indica o custo fixo do veículo *k* caso ele seja utilizado;

$g_{ijk}$ – Parâmetro que representa o custo da rota ligando o cliente *i* ao cliente *j* feita pelo veículo *k*;

$Q_k$ – Parâmetro que indica a capacidade do veículo *k.*

A formulação matemática segue abaixo:

$$\text{Minimizar} \quad z = \sum_{k=1}^{K} f_k y_k + \sum_{i=0}^{N} \sum_{j=0, i\neq j}^{N} \sum_{k=1}^{K} (g_{ijk} x_{ijk}) + \sum_{i=0}^{N} \sum_{j=0, i\neq j}^{N} \sum_{k=1}^{K} (p_{ijk} x_{ijk}) \tag{19}$$

Sujeito a:

$$\sum_{k=1}^{K} \sum_{j=1}^{N} x_{ijk} = 1 \qquad \forall\, i \in \{1, \dots, N\}, \quad i \neq j \tag{20}$$

$$\sum_{k=1}^{K} \sum_{i=1}^{N} x_{ijk} = 1 \qquad \forall\, j \in \{1, \dots, N\}, \quad i \neq j \tag{21}$$

$$\sum_{j=1}^{N} x_{0jk} = y_k \qquad \forall\, k \in \{1, \dots, K\} \tag{22}$$

$$\sum_{i=1}^{N} x_{i0k} = y_k \qquad \forall\, k \in \{1, \dots, K\} \tag{23}$$

$$\sum_{i=0, i\neq h}^{N} x_{ihk} - \sum_{j=0, j\neq h}^{N} x_{hjk} = 0 \qquad \forall\, k \in \{1, \dots, K\},\ \forall\, h \in \{1, \dots, N\} \tag{24}$$

$$\sum_{i=1}^{N} q_i \sum_{j=0, i\neq j}^{N} x_{ijk} \leq Q_k y_k \qquad \forall\, k \in \{1, \dots, K\} \tag{25}$$

$$b_{ik} + s_i + t_{ijk} + \alpha . p_{ijk} - M(1 - x_{ijk}) \leq b_{jk} \qquad \begin{cases} \forall\, i \in \{0, \dots, N\}, \\ \forall\, j \in \{1, \dots, N\}, \\ \forall\, k \in \{1, \dots, K\}, \\ i \neq j \end{cases} \tag{26}$$

$$s_0 = 0, \qquad b_{0k} = 0 \tag{27}$$

$$e_i \leq b_{ik} \leq l_i \qquad \begin{matrix} \forall\, i \in \{1, \dots, N\}, \\ \forall\, k \in \{1, \dots, K\} \end{matrix} \tag{28}$$

$$x_{ij}, y_k \in \{0,1\} \qquad \forall\, i, j \in \{1, \dots, N\},\ i \neq j,\ \forall\, k \in \{1, \dots, K\} \tag{29}$$

A Equação (19) é a função objetivo. A variável $y_k$ é multiplicada pelo parâmetro $f_k$ que indica o custo fixo do veículo *k*, que será um valor para a bicicleta e outro para a moto. A variável $x_{ijk}$ será multiplicado pelo parâmetro $g_{ijk}$ que representa o custo da rota ligando o cliente *i* ao cliente *j* feita pelo veículo *k*. Mais uma vez temos a diferenciação entre os tipos de veículos na equação. Por exemplo, se o veículo for uma bicicleta então o custo do deslocamento entre o cliente *i* e o cliente *j* será um, se o veículo for uma moto o custo para esse mesmo trecho será outro. Por fim, temos o parâmetro $p_{ijk}$ que representa a penalidade da subida entre o cliente *i* e o cliente *j* se o veículo *k* for uma bicicleta, sendo zero caso o veículo *k* seja uma moto. Dessa maneira a função objetivo leva em conta que bicicletas e motos possuem um custo fixo diferente, que motos e bicicletas podem ter um custo diferente de deslocamento (por exemplo, consumo de combustível), e que bicicletas são penalizadas quando enfrentam subidas íngremes.

Como na formulação de janelas de tempo, a Equação (20) determina que de um cliente *i* sai somente uma rota para um cliente *j* através de um único veículo. Por outro lado, a Equação (21) determina que no cliente *j* chega somente uma rota de um cliente *i* e por um único veículo.

A Equação (22) estabelece que todos os veículos, sejam bicicletas ou motos, partem da origem enquanto a Equação (23) estabelece que todos os veículos retornam para a origem.

A Equação (24) garante que o fluxo que entra num cliente é o mesmo que sai desse cliente.

A Equação (25) garante que a somatória das demandas $q_i$ de todos os clientes *i* atendidos pelo veículo *k* não deve exceder a capacidade *Q* do veículo *k* se ele for utilizado. Aqui podemos ter a diferenciação entre capacidade de carga da bicicleta e a da moto através de $Q_k$.

As Equações (26), (27) e (28) garantem que as rotas bem como o número de veículos sejam calculados observando a restrição das janelas de tempo bem como que a formação de sub-rotas seja eliminada. Aqui a constante $t_{ijk}$ estabelece que o tempo de deslocamento do cliente *i* até o cliente *j* poderá ser diferente dependendo do tipo do veículo empregado. No caso, motos podem se deslocar mais rapidamente que bicicletas.

A Equação (29) garante que as variáveis sejam binárias.

Como exemplo de aplicação podemos atribuir os valores abaixo (esses valores são apenas ilustrativos):

- $f_k$ vale 100 para moto e 5 para bicicleta;
- $g_{ijk} = \omega . c_{ij}$ onde ω vale 20 para moto e 1 para bicicleta;
- $p_{ijk} = \phi . p_{ij}$ onde $\phi$ vale 0 para moto e 1 para bicicleta;
- $t_{ijk} = \beta . t_{ij}$ onde β vale 1/4 para moto e 1 para bicicleta;
- $Q_k$ vale 20 para moto e 15 para bicicleta.

A vantagem dessa formulação é que com as constantes $\omega, \phi, \beta$ nós podemos facilmente calcular as matrizes $g_{ijk}$, $p_{ijk}$ e $t_{ijk}$ a partir das matrizes $c_{ij}$, $p_{ij}$ e $t_{ij}$ já vistas nas formulações anteriores.

### 3.1.5 Problema com múltiplos depósitos

Uma outra variação interessante do problema considera as restrições anteriores e adiciona a possibilidade de haver mais que um depósito de origem. Nesse cenário, um veículo sai de um depósito, percorre alguns clientes e volta sempre para o mesmo depósito de onde partiu. Trata-se de um modelo bem completo e versátil que será apresentado nessa Seção, porém não fará parte do escopo do restante da dissertação quanto à implementação do modelo em Python nem a construção de soluções heurísticas.

A formulação apresentada nessa Seção baseia-se nos artigos de Hanum *et al*. (2018) e de Afshar-Nadjafi e Afshar-Nadjafi (2014) e possibilita alocarmos um número arbitrário de veículos por depósitos onde esses veículos podem variar em capacidade e custo. Novos conjuntos passam a ser definidos para essa formulação:

*W* – Conjunto de todos os depósitos e com cardinalidade *M*. Assim, *W* = {1,...,*M*} e não usamos mais o índice 0 para o depósito;

*V* – Conjunto de todos os clientes e com cardinalidade *N*, porém iniciando-se logo após o índice *M* referente ao último depósito, *V* = {*M*+1,*M*+2,...,*M*+*N*};

*T* – Conjunto de todos os nós, $T = W \cup V$;

$F_i$ – Conjunto dos veículos alocados ao depósito *i*. Além disso, *F* representa o conjunto de todos os veículos disponíveis considerando todos os depósitos, nesse caso a cardinalidade do conjunto é *P*, onde *P* representa o número de veículos.

Segue o modelo matemático:

$$\text{Minimizar} \quad z = \sum_{i=1}^{M+N}\sum_{j=1}^{M+N}\sum_{k=1}^{P} c_{ijk}\, x_{ijk} + \sum_{i=1}^{M}\sum_{j=M+1}^{M+N}\sum_{k=1}^{P} f_k\, x_{ijk} \tag{30}$$

Sujeito a:

$$\sum_{i=1}^{M+N}\sum_{k=1}^{P} x_{ijk} = 1 \qquad \forall\, j \in V \tag{31}$$

$$\sum_{j=1}^{M+N}\sum_{k=1}^{P} x_{ijk} = 1 \qquad \forall\, i \in V \tag{32}$$

$$\sum_{j=M+1}^{M+N} x_{ijk} \leq 1 \qquad \forall\, k \in F_i, \quad \forall\, i \in W \tag{33}$$

$$\sum_{i=M+1}^{M+N} x_{ijk} \leq 1 \qquad \forall\, k \in F_j, \quad \forall\, j \in W \tag{34}$$

$$\sum_{j=M+1}^{M+N} x_{ijk} = 0 \qquad \forall\, k \notin F_i, \forall\, i \in W \tag{35}$$

$$\sum_{i=M+1}^{M+N} x_{ijk} = 0 \qquad \forall\, k \notin F_j, \forall\, j \in W \tag{36}$$

$$\sum_{i=1}^{M+N} x_{irk} = \sum_{j=1}^{M+N} x_{rjk} \qquad \forall\, k \in F, \forall\, r \in V \tag{37}$$

$$\sum_{i=M+1}^{M+N} q_i \sum_{j=1}^{M+N} x_{ijk} \leq Q_k \qquad \forall\, i \in V, \quad \forall\, k \in F_i \tag{38}$$

$$e_i \leq b_{ik} + s_i \leq l_i \qquad \forall\, k \in F, \quad \forall\, i \in T \tag{39}$$

$$b_{ik} + s_i + t_{ij} - M(1 - x_{ijk}) \leq b_{jk} \qquad \forall\, k \in F, \forall\, i \in T, \forall\, \mathrm{j} \in V \tag{40}$$

$$x_{ijk} = 0 \qquad \forall\, k \in F, \forall\, i \in W, \forall\, \mathrm{j} \in W \tag{41}$$

$$x_{ijk} \in \{0,1\} \qquad \forall\, i,j \in T, \quad \forall\, k \in F \tag{42}$$

Na Equação (30) temos a função objetivo que é composta por dois termos. O primeiro termo envolve o custo do trajeto percorrido pela rota e o segundo envolve o custo fixo dos veículos escolhidos. Logicamente o custo do trajeto também pode depender do tipo do veículo escolhido e é por isso que utilizamos a variável $c_{ijk}$ onde podemos fazer $c_{ijk} = d_{ij}v_k$ tendo $d_{ij}$ como uma matriz de distâncias entre os pontos *i* e *j*, e tendo $v_k$ como um vetor contendo os fatores de conversão de distância para custo para cada veículo *k*.

As Equações (31) e (32) contém as primeiras restrições que estabelecem que cada cliente é visitado por apenas um único veículo e que depois de ser visitado por esse veículo que ele seguirá para outro cliente ou depósito.

As Equações (33) e (34) estabelecem que um veículo *k* só pode sair do depósito em que ele está alocado e só pode retornar para esse depósito, embora não seja necessário usar todos os veículos que estejam alocados a um depósito. Na otimização de um problema talvez um determinado depósito possa acabar não sendo utilizado.

As Equações (35) e (36) estabelecem que um veículo que não pertence a um depósito não venha a sair dele nem voltar para ele.

A Equação (37) garante a continuidade da rota ou, como apresentada nos modelos anteriores, a conservação do fluxo onde o mesmo fluxo que sair de um nó é o mesmo fluxo que entra nele.

A Equação (38) garante que a capacidade de cada veículo não seja violada ao passar por todos os clientes que estão contidos em sua rota.

A Equação (39) garante que as janelas de tempo sejam respeitadas. Da forma como a equação foi escrita, não apenas a chegada do veículo, mas também a execução do serviço deve ficar dentro da janela de tempo. Logicamente, um ajuste nessa equação possibilitará o cenário em que não necessariamente a execução do serviço tenha de ocorrer dentro da janela de tempo. Com isso teremos o cenário em que o instante final da janela de tempo limitará apenas a chegada do veículo no cliente. A Equação (40) elimina as sub-rotas.

A Equação (41) elimina a possibilidade de um veículo sair de um depósito e ir para outro uma vez que definimos $x_{ijk}$ como zero para qualquer índice *i* e índice *j* pertencentes ao conjunto *W* de depósitos.

A Equação (42) é a definição de integralidade da variável $x_{ijk}$.

## 3.2 Métodos heurísticos e meta-heurísticas

### 3.2.1 Necessidade de métodos heurísticos

A modelagem matemática e o emprego de algoritmos exatos têm se mostrado um campo vasto para pesquisa e com a obtenção de resultados notáveis. No entanto, na medida em que a complexidade dos problemas e seu tamanho aumentam, a obtenção de uma solução por meio de métodos exatos torna-se impraticável devido a um demasiado esforço computacional, que se traduz em tempo, exigido para a resolução.

Diante desse cenário, as soluções heurísticas aparecem como uma abordagem realista e atraente. Uma solução heurística não garante a solução ótima, mas pode apresentar uma solução que seja factível e ao mesmo tempo boa num prazo de tempo curto. Aqui é importante destacar o ponto de que não há garantia de que a solução seja ótima, mas para muitos problemas de natureza combinatória na vida real podemos nos contentar com uma solução que seja consideravelmente boa. Aliás, podemos perder muito tempo na busca de uma solução ótima quando já poderíamos haver implementado uma solução boa e lucrado com ela.

Goldbarg *et al*. (2016) apresentam a etimologia da palavra heurística como tendo origem grega e vinda do verbo *heuriskein,* que possui o significado de descobrir, e onde a conjugação em primeira pessoa é a conhecida palavra *eureka* (eu descobri).

Além da grande vantagem do tempo de execução, que pode ser curto para uma solução heurística e totalmente irrealista para uma solução com base em métodos exatos, também deve-se destacar que a implementação de um método heurístico pode ser feita bem mais facilmente.

Como conceito geral, podemos dizer que um método heurístico consiste numa busca de soluções por meio de tentativa e erro, onde soluções do problema passam a ser experimentadas quanto à sua viabilidade (ou aderência às restrições do problema) e a como se comparam com a melhor solução viável encontrada até o presente momento. Naturalmente, no limite poderíamos pensar numa análise que enumerasse todas as possíveis soluções do problema e aí nosso método heurístico garantiria a otimalidade, porém esbarraríamos no mesmo problema dos métodos exatos. O tempo para se testar todas as possíveis soluções seria irrealisticamente longo na medida em que o tamanho do problema aumenta. Visto dessa forma, podemos dizer de uma maneira simplista que um método heurístico produz um

conjunto de soluções para serem testadas e seleciona a melhor dentre elas. Dessa colocação podemos então deduzir facilmente que a qualidade do método virá de quão bem ele consegue gerar esse conjunto de soluções de forma que ele produza soluções consideravelmente boas e que não estejam distantes da solução ótima.

Diante do exposto acima, podemos identificar três pontos que poderão fazer parte de um método heurístico:

- Gerar soluções aleatórias para o problema a partir de uma solução já existente, o que confere um caráter estocástico e não determinista para o método, mas que ao mesmo tempo haja um racional por detrás que crie um potencial de melhora da solução;
- Capacidade de testar essas soluções em relação às restrições que definem os limites do problema;
- Capacidade de armazenar a melhor solução factível encontrada até o momento.

Com relação ao primeiro ponto mencionado, onde o método heurístico deve gerar soluções a partir de uma solução já existente, podemos dizer que é nesse ponto que reside o que poderíamos chamar de “a inteligência” do método. Basicamente estamos buscando melhorar uma solução a partir de uma já existente. Com esse intuito não é difícil visualizar que uma das armadilhas que podem comprometer a qualidade de uma solução heurística é que o algoritmo fique preso numa solução ótima local. Assim, “a inteligência” do método pode ser aprimorada por mecanismos que permitam uma fuga de uma região de soluções do problema para outra que possa ter sua vizinhança explorada. É nesse ponto da “inteligência” do método em gerar possíveis soluções que temos uma vasta área de pesquisa e que levou ao desenvolvimento de meta-heurísticas.

Entre as soluções heurísticas já desenvolvidas para o problema de roteamento de veículos, uma delas é considerada “clássica” na literatura. Ela não é estocástica, pelo contrário, sempre irá fornecer a mesma solução. No entanto, ela apresenta boa eficácia e é relativamente simples, o que a faz merecer uma menção nesse trabalho. Muitos trabalhos para o problema de roteamento já foram desenvolvidos a partir dessa heurística.

### 3.2.2 Algoritmo de Clarke e Wright

O algoritmo de Clarke e Wright é muito provavelmente a solução heurística mais popular para o problema de roteamento de veículos (CLARKE; WRIGHT, 1964). Ela é capaz de rapidamente fornecer uma solução boa para o problema. Dito de uma maneira simples, esse algoritmo parte de uma solução onde um veículo parte do depósito, vai para um cliente, e retorna para o depósito. Assim, com *N* clientes teremos *N* veículos. Logicamente essa é a solução mais custosa para o problema. No entanto, a partir desse ponto ela analisa a possibilidade de juntar clientes diferentes na mesma rota, partindo da junção de maior ganho até a de menor ganho. Essa junção de clientes na mesma rota, possibilita chegarmos numa solução boa no final da execução do algoritmo.

Um conceito importante nesse algoritmo é o conceito de economia, ou de ganho, por se juntar dois clientes na mesma rota. A Figura 5 ilustra esse conceito. A Figura 5a mostra a solução inicial onde cada cliente é servido por meio de um veículo individual, já a Figura 5b nos ajuda a entender a economia por se juntar os dois clientes na mesma rota. Nessa figura bem como nas equações abaixo o depósito é indicado pelo índice 0.

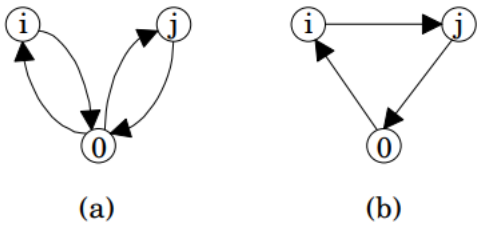



Figura 5 - Lógica do algoritmo de Clarke & Wright.

Na Figura 5a, a distância total percorrida vale (assumindo o problema simétrico onde a distância da ida é a mesma da volta):

$$D_a = D_{0i} + D_{i0} + D_{0j} + D_{j0} = 2D_{0i} + 2D_{0j}$$

Na Figura 5b, a distância total passa a ser:

$$D_b = D_{0i} + D_{ij} + D_{0j}$$

A economia por se juntar os clientes *i* e *j* na mesma rota passa a ser a diferença entre $D_a$ e $D_b$.

$$S_{ij} = D_a - D_b = D_{0i} - D_{ij} + D_{0j}$$

A Equação $S_{ij} = D_{0i} - D_{ij} + D_{0j}$ é calculada para todos os pares *i-j* representando todas as possíveis combinações de pares de clientes do problema. Após isso, uma tabela de $S_{ij}$ ordenada do maior valor para o menor é montada. O maior valor representa a maior economia por se conectar dois clientes numa mesma rota. A tabela criada contém três colunas (economia $S_{ij}$, cliente *i*, cliente *j*) e é ordenada pela primeira coluna, do maior valor para o menor.

O algoritmo de Clarke e Wright percorre essa tabela, da primeira linha até a última, e analisa a possibilidade de se ligar os clientes *i* e *j* na mesma rota. Para isso há alguns casos específicos que o algoritmo considera.

- Caso 1: No caso de ambos o cliente *i* e o cliente *j* não fazerem parte de nenhuma outra rota com outros clientes, então uma nova rota é criada contendo os clientes *i* e *j* se a demanda somada dos clientes *i* e *j* não ultrapassar a capacidade máxima do veículo do problema.
- Caso 2: Se um dos dois clientes (e apenas um deles) já fizer parte de uma rota com outros clientes e não for um ponto interior da rota, ou seja, se for um ponto de conexão da rota com o depósito, então o outro cliente do par *i-j* em questão será inserido entre ele e o depósito. Logicamente apenas se a inserção desse cliente na rota não vier a exceder a capacidade máxima do veículo do problema.
- Caso 3: Se ambos os clientes *i* e *j* fizerem parte de rotas distintas, já com outros clientes nessas rotas, e ambos os clientes *i* e *j* não forem pontos interiores dessas rotas, então essas duas rotas poderão ser conectadas através da junção *i-j*, passando assim a se tornar uma única rota. Logicamente, isso acontecerá apenas se a demanda consolidada dessas duas rotas não exceder a capacidade máxima do veículo do problema.

O algoritmo de Clarke e Wright executa a verificação dos três casos acima para cada uma das linhas da tabela de $S_{ij}$, até chegar na sua última linha. Caso ainda permaneçam clientes sem pertencer a nenhuma rota com outros clientes, então cada um desses clientes terá uma rota apenas contendo esse cliente e o depósito.

Altinel e Öncan (2005) propuseram uma modificação do algoritmo de Clarke e Wright com a adição de três parâmetros (λ,μ,ν), onde o primeiro permite um controle da importância relativa do arco ligando os clientes *i* e *j*, o segundo leva em consideração a assimetria da distância do depósito para cada um desses clientes

candidatos a fazerem parte da mesma rota, e o terceiro que leva em consideração o agrupamento das demandas na mesma rota.

$$s_{ij} = c_{i0} + c_{0j} - \lambda c_{ij} + \mu\left|c_{0i} - c_{j0}\right| + \nu\left(d_i + d_j\right)/\bar{d}$$

Logicamente, essa nova fórmula requer o ajuste dos três parâmetros (λ,μ,ν). Podemos facilmente ver que o vetor (1,0,0) retorna a fórmula acima para o algoritmo original de Clarke e Wright.

### 3.2.3 Meta-heurísticas

Goldbarg *et al*. (2016) apresentam a etimologia do prefixo grego *meta* relacionando-o com um nível superior. Com isso, caracterizamos as meta-heurísticas como métodos heurísticos de alto nível. Goldbarg *et al*. (2016) também apresentam a definição de meta-heurística como uma “arquitetura geral de regras que, formada a partir de um tema comum, pode servir de base para o projeto de uma ampla gama de heurísticas computacionais”.

Entre as meta-heurísticas mais conhecidas podemos citar a busca tabu, *simulated annealing*, algoritmo genético, colônia de formigas, entre outros. Cada uma dessas meta-heurísticas é uma “arquitetura geral de regras”, ou poderíamos até mesmo dizer um *framework*, a partir do qual um algoritmo heurístico específico pode ser desenhado e implementado. Por exemplo, o algoritmo genético possui um *framework* básico que inclui técnicas como geração de população, *crossover*, mutação, elitismo, entre outras. Ele pode ser implementado para vários problemas e pode ser implementado de diversas maneiras e com variações para o mesmo problema. Todas essas implementações serão diferentes, mas todas serão consideradas como algoritmos genéticos enquanto mantiverem aderência ao *framework* básico evidenciando que a implementação gere populações, realize *crossover*, faça mutações, embora, por exemplo, o elitismo possa não constar em alguma delas.

Três meta-heurísticas são consideradas nesse trabalho: busca tabu, algoritmo genético e colônia de formigas. Para tanto, vamos considerar um exemplo de um problema de otimização combinatória simples, o bem conhecido problema do caixeiro viajante que precisa percorrer *N* cidades, cada uma representada por um número, sem passar pela mesma cidade uma segunda vez, de forma a minimizar a distância total percorrida. Se considerarmos que cada cidade tenha coordenadas (*x,y*) então

podemos calcular a distância entre cada par de cidades utilizando o método de distância Euclidiana. Nesse método calculamos a distância entre dois pontos *(i,j)* por $d_{ij} = [(x_i - x_j)^2 + (y_i - y_j)^2]^{1/2}$ . Feito isso construímos uma matriz *N* x *N* onde cada elemento (*i,j*) da matriz representa a distância entre a cidade *i* e a cidade *j*. Como o objetivo é apenas usar esse problema para explicar os conceitos fundamentais das meta-heurísticas, não adicionaremos restrições ao problema. Um exemplo de restrição poderia ser o de que algumas cidades precisem ser visitadas de maneira prioritária. Com base no exposto, supondo apenas 10 cidades podemos dizer que um exemplo de uma solução para o problema seria a rota 1-2-3-4-5-6-7-8-9-10, outro exemplo seria a rota 3-4-5-1-2-10-9-6-7-8. Poderíamos pensar que conseguimos enumerar todas as soluções dessa forma, mas o problema é entender quantas possíveis soluções temos. Se partimos de uma cidade como a origem, então nos restam 9 possibilidades para a próxima cidade. Chegando nela, teremos agora 8 possibilidades para a próxima. Chegando nela, restam 7 para a próxima e assim sucessivamente. Com isso temos 9x8x7x...x1=9! = (10-1)! = (*N*-1)! = 362.880 possibilidades. Contudo, como o trajeto 1-2-3 é o mesmo que 3-2-1 para esse tipo de problema, então a fórmula para o caixeiro viajante é (*N*-1)!/2 para a obtenção do número de soluções possíveis. Convém ressaltar que essa última afirmação é válida para o problema simétrico, em que a distância entre o ponto *i* e o ponto *j* seria exatamente a mesma se o movimento for feito do ponto *j* para o ponto *i*. Para *N*=10 temos 181.440 possíveis soluções. Certamente, não conseguimos enumerar na mão todas as possibilidades. Para entendermos um pouco mais como o aumento do tamanho do problema aumenta o esforço computacional, vamos ver o que significa aumentar apenas uma cidade nesse problema. Agora, ao invés de 10 temos 11 cidades. Com isso o número de possibilidades passa de 181 mil para 1,8 milhão! Se formos para 12 cidades, então aumentamos para quase 20 milhões de possíveis soluções. Se formos para 13 cidades, então aumentamos para mais de 239 milhões de possíveis soluções. Se formos para 14 cidades, então passamos de 3 bilhões.

Esse simples exercício numérico nos ajuda a entender o quão complicado pode se tornar o emprego de um método exato que busca uma solução ótima para um problema de otimização combinatória na medida em que o tamanho do problema aumenta. Num problema simples como o caixeiro viajante, um aumento de uma única cidade, que soa como algo pequeno, acarreta um aumento de ordem fatorial no

universo de soluções do problema, aumentando consideravelmente o tempo computacional empregado. Uma solução heurística que entrega uma solução boa num prazo de tempo curto torna-se bem mais atraente e que atende perfeitamente a demanda de muitos problemas reais. Agora esse exercício também nos ajuda a vislumbrar um dos problemas relacionados com a eficiência dos métodos heurísticos. Como os métodos heurísticos são muitas vezes estocásticos ao gerar possíveis soluções, podemos ficar repetindo que uma mesma solução seja testada na execução do algoritmo, o que diminuiria sua eficiência e ainda poderia contribuir para prender a busca num ótimo local. Essa foi uma das preocupações abordadas pela meta-heurística da busca tabu.

#### 3.2.3.1 Busca Tabu

A busca tabu foi proposta por Glover (1986) e tem como característica básica buscar evitar que algumas soluções sejam revisitadas e assim garantir a eficiência do algoritmo. O mecanismo básico para a execução dessa estratégia envolve a implementação de uma memória que permita que movimentos recentes sejam armazenados e usados para evitar recorrência. Em cada iteração uma nova solução é gerada a partir de uma solução pré-existente. Essa nova solução pode ser gerada a partir de um movimento simples, ou então vários movimentos simples podem ser verificados e o melhor dentre eles selecionado. Nessa última opção usamos o conceito de exploração de uma vizinhança da solução pré-existente para a seleção da melhor opção nessa vizinhança. Uma vez tendo a nova solução selecionada, verifica-se se o movimento que a gerou consta na memória como tendo ocorrido recentemente. Entretanto, usar uma memória para a armazenagem de soluções completas para evitar o retorno às mesmas eleva o custo computacional. Implementações de busca tabu normalmente armazenam movimentos recentes e não soluções completas. O exemplo do problema do caixeiro viajante de 10 cidades pode ser usado para explicar a lógica da busca tabu e o uso da memória para armazenagem de movimentos recentes.

No nosso problema do caixeiro viajante vamos assumir uma solução inicial 3-4-5-1-2-10-9-6-7-8. Uma nova solução é gerada a partir dessa solução invertendo-se a ordem da cidade 4 com a cidade 9. Dessa forma passamos a ter a solução 3-9-5-1-2-10-4-6-7-8. Nossa estrutura de memória passou a armazenar o par 4-9 como um movimento recente. Com isso, na próxima iteração caso as cidades 4 e 9 sejam

novamente selecionadas para a inversão, o algoritmo sinalizará que esse é um movimento tabu, ou seja proibido, e não o realizará. Se o realizasse, voltaríamos para a solução 3-4-5-1-2-10-9-6-7-8, o que seria uma perda de eficiência na execução do algoritmo. Vamos supor que nossa memória tenha a capacidade para armazenar cinco pares de movimentos e opere no modelo FIFO (*first in, first out*). Dessa forma, depois de 5 movimentos o par 4-9 sai da memória e deixa de ser um movimento tabu. Ao realizar cada movimento e gerar uma nova solução, o custo dessa solução é calculado e comparado com o custo da melhor solução até o momento. Se o custo da nova solução for melhor, então esse novo custo bem como essa nova solução serão armazenados como a melhor solução até o momento.

Mencionamos que a memória para esse exemplo tem a capacidade de armazenar até cinco pares de movimentos. Implementações da busca tabu podem aumentar ou diminuir o tamanho dessa memória ao longo da execução.

É muito importante ressaltar que a memória não está armazenando soluções completas e sim movimentos, e isso traz uma implicação. O primeiro movimento foi a inversão das cidades 4 e 9, o que gerou a solução 3-9-5-1-2-10-4-6-7-8. Vamos agora assumir que o algoritmo selecionou as cidades 1 e 10 para o próximo movimento de inversão e com isso passamos a ter a nova solução 3-9-5-10-2-1-4-6-7-8. A memória terá armazenados os pares 4 e 9, bem como 1 e 10. Supomos que o algoritmo agora recomende novamente o par 4 e 9 para o movimento seguinte. Esse par ainda consta na memória e, portanto, esse será um movimento tabu. Por outro lado, executar esse movimento, embora tabu, não nos leva de volta para uma solução já verificada. A solução 3-4-5-10-2-1-9-6-7-8 é uma nova solução. Isso é a implicação do fato que a memória não armazena soluções completas, mas sim movimentos. Para levar essa implicação em conta, introduzimos o que chamamos de critério de aspiração. Basicamente avaliamos qual o custo da solução se o movimento tabu for executado, se o custo for melhor que o custo da melhor solução até o momento, então a restrição tabu é relaxada e o movimento é executado. Caso não seja melhor, então a restrição tabu permanece e o movimento não é realizado.

O procedimento descrito até aqui já caracteriza um algoritmo de busca tabu. No entanto, ainda há possibilidades de aprimoramento do algoritmo. Por exemplo, a memória que contém a lista tabu é considerada uma memória de curto prazo. Basicamente ela vai armazenar os movimentos recentes até um limite, chegando-se nesse limite o movimento mais antigo é eliminado e deixa de ser considerado um

movimento tabu. Algumas implementações de busca tabu adicionam uma memória de longo prazo. Essa memória possui uma capacidade de armazenamento maior e uma utilização dela pode ser para armazenar a frequência com que movimentos são executados. Trata-se de uma memória de longo prazo porque as informações dela não necessitam ser apagadas durante a execução. Por exemplo, vamos voltar no nosso problema do caixeiro viajante de 10 cidades. O par 4 e 9 entrou na lista tabu, ou seja, na memória de curto prazo. Em paralelo a isso, uma entrada poderá ser criada na memória de longo prazo para o par 4 e 9, e associar a essa entrada o valor 1. Isso significa que durante a execução do algoritmo esse movimento já ocorreu uma vez. O algoritmo continua sua execução, novos movimentos são realizados, e o par 4 e 9 por fim sai da lista tabu, ou seja, da memória de curto prazo. Numa outra iteração, o movimento inverte novamente as cidades 4 e 9 de posição na rota do caixeiro viajante. Com isso, o par 4 e 9 entra novamente na lista tabu, ou seja, a memória de curto prazo. Em paralelo, na memória de longo prazo a entrada do par 4 e 9 é incrementada passando a ter o valor 2. Isso significa que essas duas cidades já foram invertidas na rota duas vezes durante a execução do algoritmo. Assim, ao longo da execução mantemos um registro da frequência com que pares de cidades sofreram movimentos de inversão. A vantagem dessa memória de longo prazo é que podemos aprimorar o algoritmo de forma que ele passe a privilegiar movimentos que tiveram uma frequência de inversão baixa e comece a penalizar os movimentos que já têm uma frequência de inversão alta. O objetivo por detrás dessa estratégia é reforçar que o algoritmo escape de ótimos locais e busque a exploração de outras áreas do universo de soluções do problema.

Um fluxograma da busca tabu é apresentado na Figura 6. Esse fluxograma descreve o algoritmo básico de busca tabu com a criação e verificação da lista tabu e do critério de aspiração.

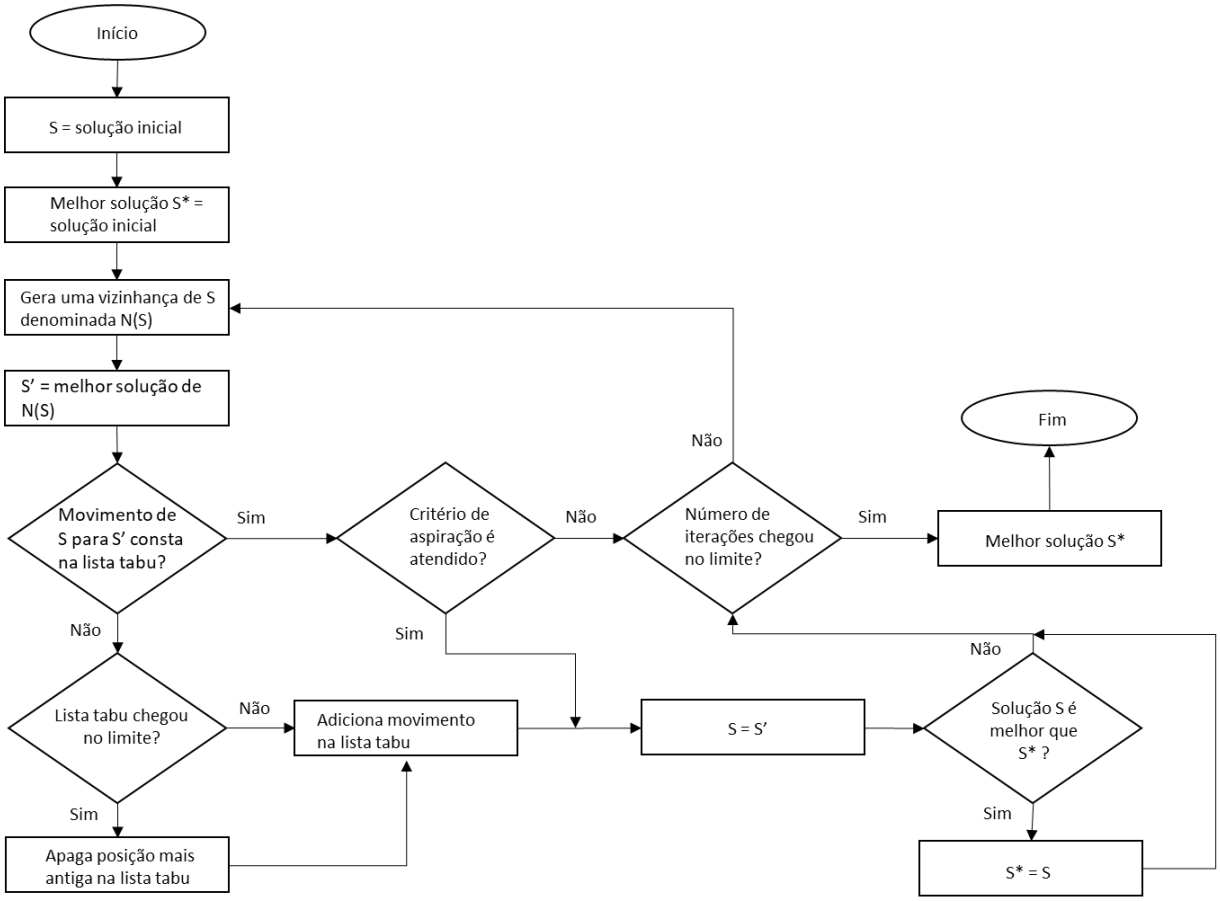



Figura 6 - Fluxograma da busca tabu.

Uma aplicação bem-sucedida de uma meta-heurística envolve não apenas o conhecimento da meta-heurística, mas também um conhecimento do problema e dos possíveis movimentos que podem ser feitos para a exploração de vizinhança e de melhora do resultado visando aproximar-se da solução ótima. No caso do problema do roteamento de veículos há uma variedade de movimentos que podem ser feitos e que estão descritos a seguir.

Troca da posição de dois clientes dentro da mesma rota. Uma função é chamada para executar um *loop* que, limitado a um número definido de tentativas, buscará uma troca de dois clientes que melhore o custo da solução atual. Encontrando essa troca, a função retorna a rota da troca com os dois clientes invertidos. Caso o *loop* tenha chegado no limite de tentativas, a função retorna que não encontrou nenhuma troca. Como exemplo, na Figura 7 os clientes 3 e 2, ambos da mesma rota, são invertidos de posição nessa rota.

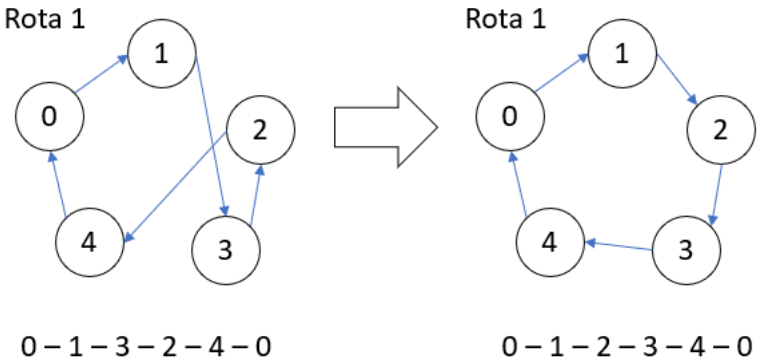



Figura 7 - Inversão de dois clientes na mesma rota.

Troca de dois clientes em rotas diferentes. Uma função é chamada para executar um *loop* que, limitado a um número definido de tentativas, buscará aleatoriamente um cliente de uma rota e outro cliente de outra rota para fazer a troca entre eles. Se a troca resultar numa solução melhor que a solução antes da troca, então a troca é feita. Se o *loop* chegar no limite de tentativas, a função retorna que não encontrou nenhuma troca. Como exemplo, a Figura 8 mostra duas rotas e o cliente 1 de uma rota é trocado com o cliente 5 de outra rota.

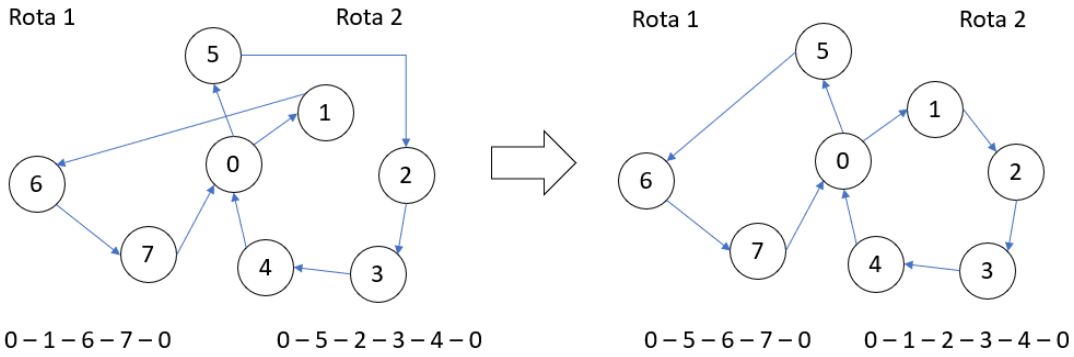



Figura 8 – Troca de dois clientes em rotas diferentes.

Tira um cliente de uma rota e o acrescenta em outra. Uma função é chamada que escolhe aleatoriamente duas rotas. Então um cliente aleatório da primeira rota é escolhido e removido dela. Um *loop* é feito para avaliar em que posição esse cliente pode ser inserido na segunda rota e então a inserção é feita na posição em que resulta no menor custo para essa rota. Na Figura 9 o cliente 1 é removido de uma rota e inserido em outra.

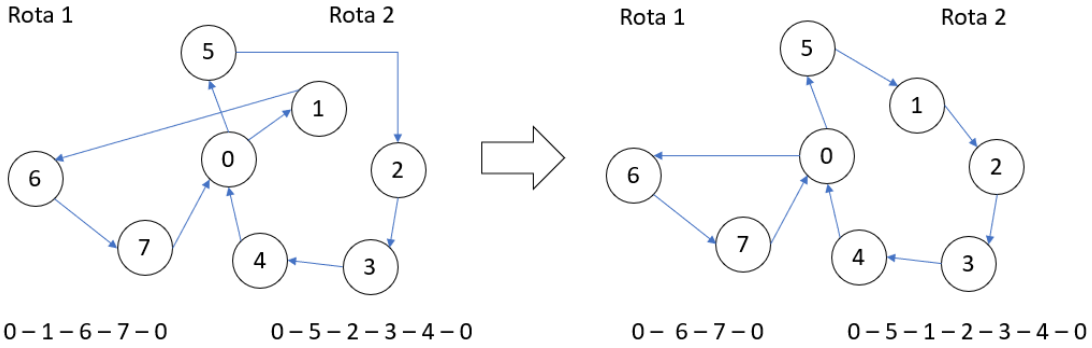



Figura 9 - Retirada de um cliente de uma rota e inserção em outra.

Tira o primeiro cliente de uma rota que esteja com capacidade excedida e o insere no final de uma rota em que sobra capacidade. Uma função é chamada e verifica se entre todas as rotas há alguma onde a capacidade do veículo esteja excedida. Encontrando, remove-se o primeiro cliente dela e o insere no final de uma rota onde haja uma sobra de capacidade. Enquanto os movimentos anteriores são movimentos que primariamente buscam reduzir o custo, ou distância percorrida, da solução, esse movimento busca primariamente tornar uma solução não-factível em uma solução factível. Na Figura 10, a segunda rota tem sua capacidade excedida, assim retira-se seu primeiro cliente, que no caso é o cliente 1, e ele é inserido no final da primeira rota.

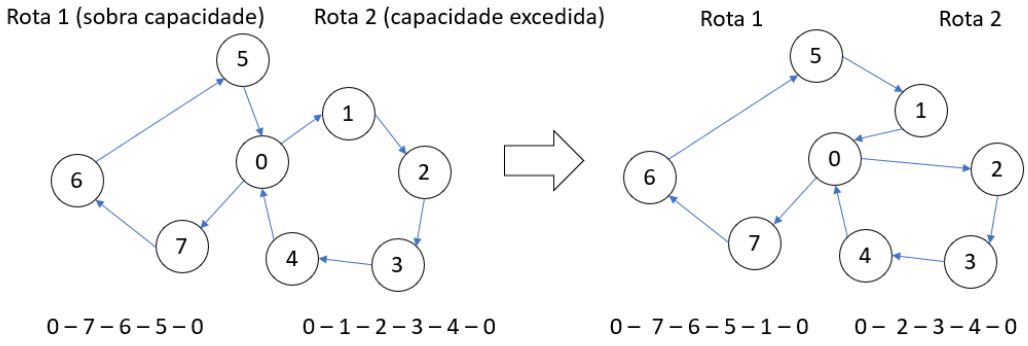



Figura 10 - Retirada do primeiro cliente de uma rota com capacidade excedida e inserção no final de outra.

O algoritmo começa com a criação de uma solução inicial, a partir da qual inicia-se a exploração de vizinhança com a execução dos movimentos descritos acima. Para essa implementação duas memórias tabu foram empregadas.

1) Memória de rotas: Essa memória armazena a rota ou as duas rotas que foram utilizadas no movimento. Por exemplo, no movimento de inversão de dois clientes na mesma rota, apenas uma rota vai para essa memória. Já no caso da inversão de dois clientes onde cada um está numa rota, temos duas rotas indo para a memória.

2) Memória de clientes: Essa memória armazena o cliente ou os dois clientes envolvidos no movimento. Por exemplo, no movimento de inversão de dois clientes na mesma rota, os dois clientes vão para essa memória. Já no caso do movimento em que um cliente é retirado de uma rota e acrescentado em outra, apenas um cliente vai para a memória.

Na sequência apresentam-se os pseudocódigos para as funções de movimentos para a exploração de vizinhança e do algoritmo busca tabu chamando todas essas funções. Durante os testes de implementação percebeu-se não só a possibilidade bem como também a vantagem de se intensificar a busca de soluções melhores durante certos movimentos. Por exemplo, na chamada da função para a inversão de dois clientes na mesma rota há a possibilidade de se inverter dois clientes e a solução resultante ser pior. Dessa forma, na implementação feita como parte desse trabalho, a função desfaz a inversão quando isso ocorre e seleciona um outro par para avaliar a inversão. A função repete o ciclo até que encontre um par que resulte numa solução de custo inferior ou até o limite estabelecido. Para a função de troca de dois clientes em duas rotas diferentes, o funcionamento é um pouco diferente, a função é executada por um número determinado de vezes sendo que se a troca resulta em uma solução melhor então é mantida, caso contrário ela é desfeita. Com isso, na mesma chamada de função poderemos ter várias inversões de clientes em rotas diferentes se cada uma delas resultar numa melhoria do custo da solução. Nesse caso, apenas as últimas rotas e os últimos clientes que sofreram essa operação passam para a atualização das memórias das listas tabu.

A seguir temos o pseudocódigo das funções dos quatro movimentos de exploração de vizinhança que foram descritos e o pseudocódigo da busca tabu que foi implementada para os experimentos da dissertação. O valor de 500 empregado nas duas primeiras funções e de 10.000 para o número de iterações da busca tabu são ilustrativos e refletem o que foi usado na implementação dessa dissertação.

Pseudocódigo das funções chamadas pelo algoritmo de busca tabu

```
função troca de posição na mesma rota
    Armazena custo da solução atual
    Contador = 0; Máximo = 500; Troca = false
    enquanto (Contador<Máximo) ou (Troca == False) faça
        Seleciona uma rota que contenha 2 ou mais clientes
        Seleciona aleatoriamente dois clientes da rota
        Inverte a posição dos dois clientes
        Calcula custo da nova rota
        Se custo da nova rota < custo da solução atual faça
            Troca = true
        caso contrário
            Inverte a posição dos dois clientes
        Incrementa Contador

função troca de posição de dois clientes em rotas diferentes
    Contador = 0; Teto = +∞
    Armazena custo da solução atual
    enquanto Contador < 500 faça
        Seleciona aleatoriamente duas rotas
        Seleciona um cliente de cada uma dessas duas rotas
        Inverte os clientes nessas rotas
        Calcula custo da nova solução
        se custo da nova solução < Teto faça
            Teto = custo da nova solução
            Armazena rota 1, rota 2, cliente 1, cliente 2
        caso contrário
            Desfaz troca
        Incrementa Contador
```

**função** tira cliente de uma rota e acrescenta na melhor posição de outra

Seleciona uma rota com pelo menos dois clientes

Seleciona um cliente dessa rota

Seleciona uma segunda rota

Percorre segunda rota e insere cliente na posição onde custo seja menor

**função** tira cliente de rota com capacidade excedida e coloca em outra

Percorre as rotas e identifica uma rota com a capacidade excedida

Remove o primeiro cliente da rota com capacidade excedida

Insere o cliente no final de outra rota

Pseudocódigo do algoritmo de busca tabu

```
Contador = 0; Máximo = 10.000;
enquanto Contador < Máximo faça
    Chama função troca de posição na mesma rota
    se (custo da solução < melhor custo) e (solução viável == true) faça
        Melhor custo = custo da solução
        Melhor solução = solução atual
        Adiciona rota e clientes nas listas tabus
    caso contrário
        se rota ou clientes desse movimento estão nas listas tabu faça
            Não executa a troca
        caso contrário
            Adiciona rota e clientes nas listas tabus
    Chama função troca de posição de dois clientes em rotas diferentes
    se (custo da solução < melhor custo) e (solução viável == true) faça
        Melhor custo = custo da solução
        Melhor solução = solução atual
        Adiciona rotas e clientes nas listas tabu
```

**caso contrário**

**se** rotas ou clientes desse movimento estão nas listas tabu **faça**

Não executa a troca

**caso contrário**

Adiciona rotas e clientes nas listas tabus

Chama **função** tira cliente de uma rota e insere na melhor posição de outra

**se** (custo da solução < melhor custo) **e** (solução viável == true) **faça**

Melhor custo = custo da solução

Melhor solução = solução atual

Adiciona rotas e cliente nas listas tabu

**caso contrário**

**se** rotas ou cliente desse movimento estão nas listas tabu **faça**

Não executa a troca

**caso contrário**

Adiciona rotas e cliente nas listas tabu

Chama **função** tira cliente de rota com capacidade excedida e coloca em outra

**se** (custo da solução < melhor custo) **e** (solução viável == true) faça

Melhor custo = custo da solução

Melhor solução = solução atual

Adiciona rotas e cliente nas listas tabu

**caso contrário**

**se** rotas ou cliente desse movimento estão nas listas tabu **faça**

Não executa a troca

**caso contrário**

Adiciona rotas e cliente nas listas tabu

Incrementa Contador

---

O Capítulo de Experimentos irá abordar a implementação do algoritmo, porém conforme já indicado no Capítulo de Revisão da Literatura, o algoritmo de busca tabu tem se mostrado bem eficiente. A Figura 11 ilustra o resultado de uma instância conhecida, a A-n32-K5, que possui solução ótima de 784 e que foi obtida pela implementação do algoritmo conforme concebido nessa Seção em apenas 381 segundos.

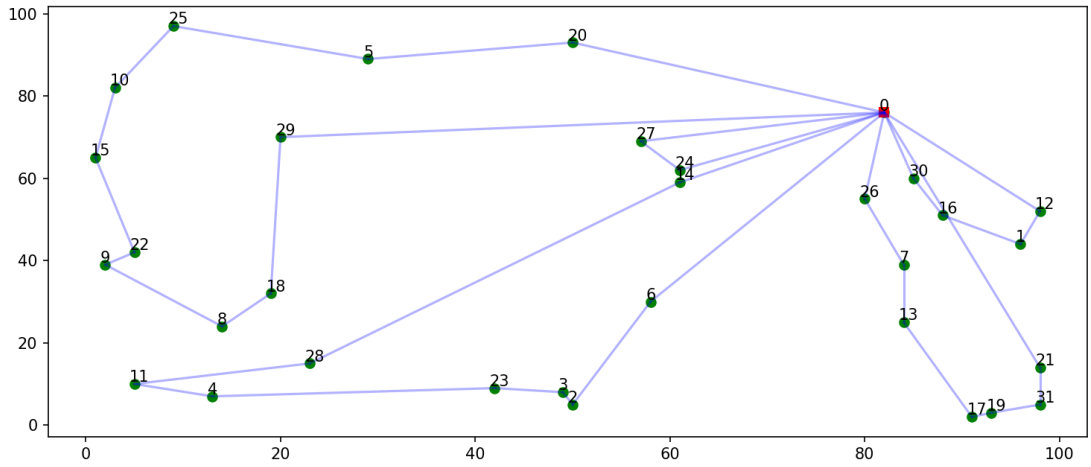



Figura 11 - Solução ótima encontrada através de busca tabu para A-n32-k5.

### 3.2.3.2 Algoritmos genéticos

Os algoritmos genéticos estão entre as meta-heurísticas mais populares para a resolução de problemas de otimização combinatória e foram introduzidos por John Holland (HOLLAND, 1975). O paralelo com a genética é claro. As possíveis soluções do problema são traduzidas para um formato de cromossomo e um conjunto de possíveis soluções constitui uma população. Passamos a chamar as iterações de gerações. Em cada geração selecionamos dois indivíduos (possíveis soluções) para a reprodução. Para a reprodução determinamos um ponto de *crossover* para um corte no cromossomo que constitui a solução. Os dois indivíduos pais irão gerar dois indivíduos filhos, cada um com um dos cortes dos cromossomos de ambos os pais. Podemos ainda selecionar indivíduos da nova população de filhos e realizar uma mutação no cromossomo. Essas mutações podem, por exemplo, ter como objetivo o aprimoramento do cromossomo melhorando o custo da solução representada por ele, ou então corrigir o cromossomo de uma solução não-factível para que passe a obedecer a alguma restrição do problema. Na geração de cada indivíduo filho e na realização de cada mutação calculamos o custo da solução representada pelo cromossomo e comparamos com a melhor solução encontrada até o momento, se for melhor atualizamos a melhor solução. Os indivíduos filhos dessa geração tornam-se os indivíduos pais da geração seguinte. O processo se repete por várias gerações. Podemos trabalhar com elitismo por incluirmos a melhor solução encontrada até o

momento com frequência variável na reprodução para indivíduos filhos em cada geração.

Podemos exemplificar o processo dos operadores do algoritmo genético como o *crossover* e a mutação na Figura 12. Para esse exemplo estamos considerando que podemos traduzir a solução do problema como uma sequência de 0s e 1s. Na Figura 12 vemos que a operação de crossover gera dois indivíduos chamados de Filhos, cada um com parte da solução dos indivíduos Pai. A Figura também mostra um exemplo de mutação com uma operação de inversão entre duas posições da solução.

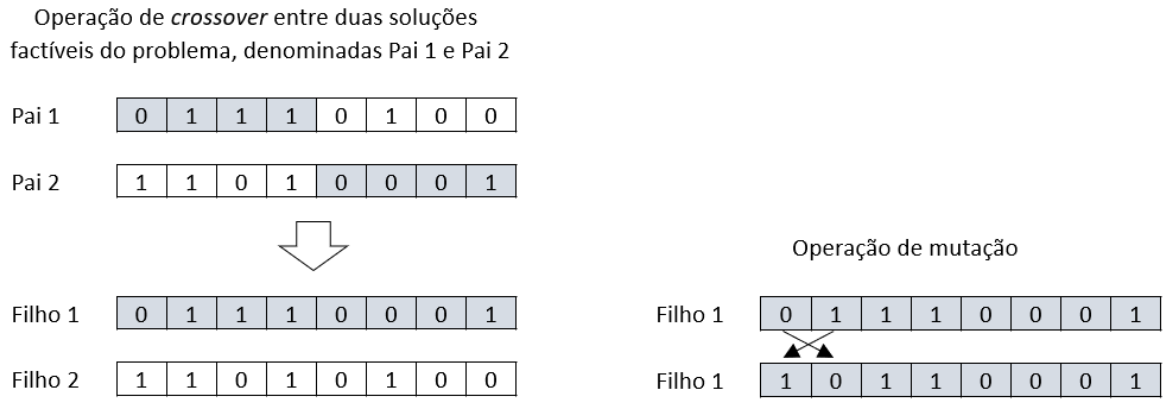



Figura 12 - Operação de *crossover* e de mutação para algoritmo genético.

Embora haja problemas que possam ser facilmente formulados da forma descrita, existem muitos problemas, especialmente os que envolvem algum tipo de sequenciamento onde a técnica de *crossover* requer cuidados específicos. Por exemplo, esse cuidado fica claro com o problema do caixeiro viajante que estamos usando como base para a discussão de meta-heurísticas. Conforme a Figura 13, uma operação de *crossover* criaria cromossomos com soluções não existentes para o problema porque repetiria algumas cidades e não incluiria outras. Para acertar essa situação, implementações especiais de *crossover* foram desenhadas e exploradas pela literatura. Basicamente, o objetivo é realizar um *crossover* que traga uma parte do cromossomo dos pais para a próxima geração e ao mesmo tempo garantir que o restante do cromossomo não tenha cidades faltando ou repetidas. Uma dessas técnicas está descrita no esquema ilustrado na Figura 13 e que pode ser usada em

problemas de sequenciamento como o problema do caixeiro viajante (TSP) e do roteamento de veículos.

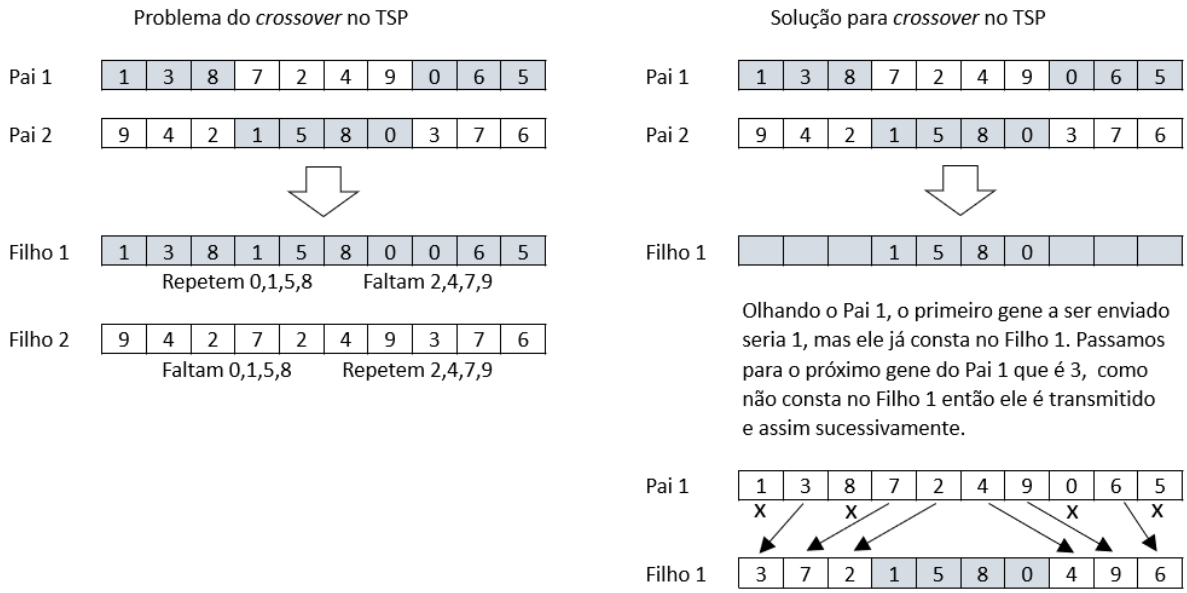



Figura 13 - Problema do *crossover* em problemas de sequenciamento.

Uma vez realizada a etapa da geração do cromossomo filho, o mesmo pode ser submetido a um processo de mutação. A mutação pode servir para diversificar a vizinhança do problema a ser explorada bem como buscar um aprimoramento da solução. No nosso problema do caixeiro viajante, uma maneira simples de se implementar a mutação é selecionando aleatoriamente duas cidades da solução e inverter suas posições no cromossomo. Essa etapa pode ser feita independente do que isso resulte no valor da solução, visando uma diversificação da vizinhança, ou então estar condicionada a que essa mutação ocorra somente se ela acarretar numa melhora do valor da solução.

O fluxograma básico de um algoritmo genético é apresentado na Figura 14. Esse fluxograma indica as principais etapas como geração de população inicial, seleção de indivíduos para reprodução, *crossover*, mutação, comparação com melhor solução até o momento, composição da nova população, elitismo, e número de gerações para execução do algoritmo.

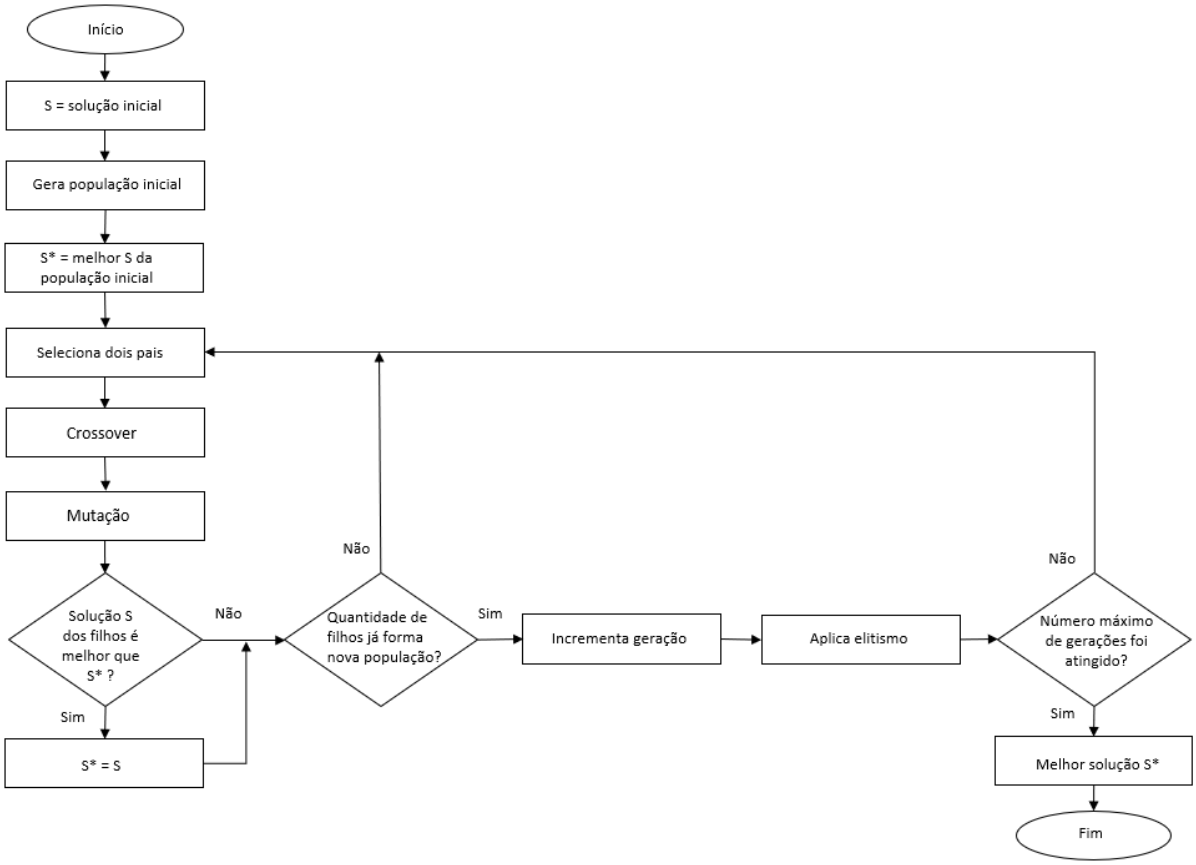



Figura 14 - Fluxograma do algoritmo genético.

Para a implementação do problema de roteamento de veículos através de um algoritmo genético, precisamos começar com uma definição do formato do cromossomo que representa uma solução viável. Por exemplo:

0 – 1 – 3 – 6 – 7 – 0 – 2 – 8 – 5 – 0 – 9 – 4 – 0

Nessa solução temos 0 representando o depósito e uma rota que sai do depósito e vai até o cliente 1, dele para o cliente 3, daí para o 6, depois para o 7 e então retorna ao depósito. Uma outra rota que sai do depósito e vai até o cliente 2, de lá para o 8, de lá para o 5 e retorna para o depósito. Por fim, a terceira e última rota que sai do depósito e vai para o cliente 9, de lá para o 4 e deste para o depósito. Com base nesse cromossomo podemos executar as operações de *crossover*, reprodução e mutação que formam a base do algoritmo genético.

O algoritmo genético para o problema do CVRP foi implementado utilizando o formato de cromossomo acima para representar as possíveis soluções. Uma solução inicial foi gerada e a partir dela uma população inicial de *N* cromossomos foi gerada embaralhando-se todos os elementos do cromossomo, mas com o cuidado de se

manter tanto o primeiro como o último elemento do cromossomo como zero, determinando-se assim o início e destino final no depósito. Nesse embaralhamento, uma sequência de dois zeros (0 – 0) simplesmente indica que um veículo deixou de ser utilizado. Uma vez a população criada, enquanto o algoritmo não tiver chegado no número de gerações máximo permitido teremos todos os processos de *crossover*, reprodução, mutação, verificação de viabilidade, comparação com o melhor resultado obtido até o momento sendo executados geração após geração.

Para a etapa de *crossover*, selecionamos aleatoriamente dois indivíduos da população corrente, bem como um ponto aleatório dentro do comprimento do cromossomo. Tendo por base esse último ponto, que definimos como ponto de *crossover*, dividimos cada um dos dois indivíduos selecionados. Juntamos então a primeira metade do indivíduo 1 com a segunda metade do indivíduo 2, e a segunda metade do indivíduo 1 com a primeira metade do indivíduo 2. Logicamente, como vimos no exemplo do algoritmo genético do caixeiro viajante, essa etapa gera o que podemos chamar de cromossomos defeituosos pois poderemos ter casos em que o mesmo cliente aparece duas vezes na mesma solução, bem como clientes ausentes na solução. Assim, sempre que tivermos a geração dos dois filhos a partir de dois pais, teremos que executar um procedimento de acerto dos filhos. Na implementação feita geramos uma lista dos clientes repetidos e outra dos clientes faltantes na solução. Identificados esses casos, fazemos a substituição de um cliente repetido por um cliente faltante até que todos os casos sejam eliminados. Dessa forma teremos dois filhos que contemplam todos os clientes, embora ainda assim a solução possa não ser factível em função da restrição de capacidade do veículo.

Uma vez tendo os dois filhos gerados e acertados para contemplarem todos os clientes, passamos para a etapa de mutação. Para essa implementação criamos alguns operadores de mutação que são aleatoriamente executados de acordo com uma probabilidade pré-estabelecida para cada um deles.

Operador de mutação para acerto de carga: Esse operador basicamente começa criando um vetor onde é lançado nele a soma da demanda dos clientes de cada rota da solução específica. Por exemplo, considerando a solução:

0 – 1 – 3 – 6 – 7 – 0 – 2 – 8 – 5 – 0 – 9 – 4 – 0

Vemos que temos nela três rotas. Se a soma das demandas dos clientes 1, 3, 6 e 7 vale 40; se a soma das demandas dos clientes 2, 8 e 5 vale 30; e a soma das demandas dos clientes 9 e 4 vale 20; teríamos o vetor: [40 30 20].

Agora supondo que a capacidade de cada veículo seja 35, então teríamos uma violação de capacidade no primeiro veículo. Criamos um procedimento de mutação que enquanto houver uma rota com a capacidade excedida no vetor ele identifica no vetor qual é essa rota e remove um dos clientes nessa rota e o insere na rota de menor capacidade utilizada. Essa operação é realizada sucessivamente até que não haja mais nenhuma rota com valor de capacidade excedida no vetor. Com isso passamos a ter um cromossomo filho, oriundo da etapa de *crossover* e reprodução, que após essa operação torna-se uma solução factível para o problema.

Operador de mutação para melhoria de custo: Nessa operação selecionamos aleatoriamente dois pontos do cromossomo e os invertemos se essa inversão resultar num custo menor do que o original. Deve-se notar que essa execução não necessariamente resulta numa solução factível do ponto de vista de capacidade. No entanto, ela contribui para a diversificação e exploração de vizinhança.

Operador de mutação para reversão do cromossomo: Essa operação reverte a ordem do cromossomo. Ao invés de ser:

0 – 1 – 3 – 6 – 7 – 0 – 2 – 8 – 5 – 0 – 9 – 4 – 0

Ele passa a ser:

0 – 4 – 9 – 0 – 5 – 8 – 2 – 0 – 7 – 6 – 3 – 1 – 0

Embora a solução tenha o mesmo custo, o efeito dela é gerar um pouco mais de diversificação na etapa futura de *crossover* e reprodução.

Operador de mutação para redução de veículo: Nessa operação a rota com a menor demanda no vetor de demandas é eliminada.

Para cada operação realizada ao longo do algoritmo, seja de geração de filhos seja de mutação nesses filhos, o cromossomo resultante é verificado para testar se é factível (significando que não viola restrição de capacidade de veículo) e se o seu custo é menor que o custo da melhor solução factível encontrada até o momento. Caso essas duas condições se mostrem verdadeiras, então a melhor solução encontrada até o momento passa a ser atualizada.

As etapas de seleção aleatória de dois indivíduos da população corrente para *crossover*, reprodução, correção, mutação, são repetidas $X/2$ vezes onde $X$ é o tamanho da população. Após essas repetições temos a população nova que será utilizada para a próxima geração onde todo o processo se repetirá até chegar no número máximo de gerações.

Elitismo: De forma a acelerar a convergência podemos empregar elitismo por substituir um número aleatório de indivíduos da população pela melhor solução encontrada até o momento. Assim, essa solução contribuirá de forma mais significativa para a produção de novos filhos.

Segue o pseudocódigo do algoritmo genético implementado para o problema de roteamento com restrição de capacidade. Nele foram assumidos alguns valores como 200 para o tamanho da população e 5.000 para o limite de gerações. Esses valores podem ser modificados. Logicamente, quanto maior for o tamanho da instância do problema (número de clientes), melhor será ter um número maior para o tamanho da população. A consequência disso será vista no tempo de execução do programa.

Pseudocódigo do algoritmo genético

```
Define uma solução inicial
Tamanho da população = 200; Total de gerações = 5.000; Contador = 1
Gera 200 soluções por embaralhar a solução inicial para compor população inicial
Corrige soluções que não tenham zero (depósito) como elementos iniciais e finais
Inicializar melhor custo e melhor solução com a melhor solução viável da população
enquanto Contador < Total de gerações faça
    count_população = 1
    enquanto count_população < Tamanho da população faça
        Seleciona dois pais e define um ponto de crossover
        Faz o crossover e gera dois filhos
        Executa correção: remove clientes duplicados e insere clientes faltantes
        Executa correção para ter 0 como elemento inicial e final
        se (custo do filho 1 ou custo do filho 2 < menor que melhor custo) faça
            se solução é viável faça
                Melhor custo = custo do filho
                Melhor solução = filho
        Início da mutação
        Executa melhora de capacidade com probabilidade de 90%
            enquanto uma das rotas do filho 1 exceder a capacidade faça
                Remove um cliente da rota que excede a capacidade
```

Insere numa rota com capacidade disponível

**se** o resultado for melhor que a melhor solução **faça**

**se** solução é viável **faça**: Atualiza a melhor solução

Repete a mutação acima mas agora para o filho 2

Executa melhora de custo com probabilidade de 90%

Seleciona filho 1

Contador auxiliar =1

**enquanto** contador auxiliar < 50 **faça**

Seleciona dois elementos do filho

Calcula custo da solução com os elementos invertidos

**se** custo diminuiu, independente de viabilidade, **faça**

Faz a troca

**caso contrário**

Desfaz a troca

Incrementa contador auxiliar

**se** o custo novo < a melhor solução **faça**

**se** solução é viável **faça**: Atualiza a melhor solução

Repete mutação acima mas agora para o filho 2

Executa procedimento de inversão com probabilidade de 50%

Inverte a ordem do filho 2

Executa a mutação de acertar a carga

**se** o resultado for melhor que a melhor solução **faça**

**se** solução é viável **faça**: Atualiza a melhor solução

Carrega população nova [count_população] = filho 1

Incrementa count_população

Carrega população nova [count_população] = filho 2

Incrementa count_população

Início do elitismo

**para** as posições de 1 a 70 da população **faça**

Com probabilidade de 30% copie a melhor solução para a posição

Incrementa contador

Comentários sobre a implementação do pseudocódigo em Python: Como a codificação do cromossomo envolveu o uso de delimitadores, a complexidade do código de programa se mostrou alta. A operação de *crossover* exige a correção dos clientes duplicados ou faltantes mas na codificação do cromossomo o nó zero já aparece como duplicado e isso está correto uma vez que indica as várias saídas e chegadas no depósito considerando que isso ocorre em cada rota. Geralmente nos algoritmos genéticos a taxa de mutação é baixa, nessa implementação usamos três formas de mutação e todas com uma taxa alta, como 90%. O motivo dessa taxa alta é que os dois tipos de mutações com taxa de 90% são operações bem dirigidas. A primeira tem o objetivo de evitar que uma rota exceda a capacidade do veículo, o que contribui para tornar soluções não-viáveis em soluções viáveis. Já a segunda tem o objetivo de realizar uma busca local visando a melhoria do custo. Dessa forma, uma taxa alta para esses dois tipos de mutação colaborou para uma convergência mais rápida e um resultado de qualidade aceitável.

#### 3.2.3.3 Colônia de formigas

A meta-heurística da colônia de formigas tem base nos trabalhos de Marco Dorigo e seus colegas e baseia-se na técnica de utilização de feromônio pelas formigas para marcar seus caminhos para que sejam seguidos pelas próximas formigas (DORIGO; MANIEZZO; COLORNI, 1991). Podemos imaginar que as primeiras formigas saem do ninho para a busca de alimento e escolhem aleatoriamente seus caminhos. Na medida em que caminham elas depositam feromônio marcando suas trilhas. Esse feromônio também evapora com o passar do tempo. É fácil imaginar que a primeira formiga a encontrar o alimento e retornar para o ninho, no seu caminho de volta ela novamente deposita feromônio sobre o feromônio já depositado no trajeto de ida, reforçando-o. Como ela chegou antes de outras, seu caminho possui mais feromônio que o caminho feito pelas outras que ainda não voltaram. Assim, uma nova formiga saindo do ninho terá uma maior probabilidade de seguir por esse caminho pois identifica a maior quantidade de feromônio depositado pela sua antecessora. Ao fazer isso, o feromônio depositado é reforçado ainda mais por essa nova formiga, tornando esse caminho mais atrativo para as próximas. Assim, a meta-heurística da colônia de formigas fornece uma técnica estocástica que envolve um mecanismo de reforçar caminhos ao mesmo tempo em que o balanceia com um conceito de atratividade para o próximo ponto da trajetória. Uma aplicação imediata

encontra-se nos problemas de busca do menor caminho ligando pontos, como por exemplo o problema do caixeiro viajante ou do roteamento de veículos.

Com base no exposto acima, um algoritmo de colônia de formigas conterá:

- Uma matriz que indicará a quantidade de feromônio depositado entre o ponto *i* e o ponto *j*;
- Uma matriz que indicará a atratividade de se ir para um ponto *j* a partir de um ponto *i*. Usualmente emprega-se o inverso da distância entre esses dois pontos para indicar essa atratividade;
- Parâmetros para calibrar o algoritmo para que a probabilidade de se ir de um ponto a outro tenda a ser maior em função da quantidade de feromônio depositado entre esses pontos ou da atratividade de um ponto em relação ao outro;
- Fórmula que calcule a probabilidade de se ir de um ponto *i* a um ponto *j* em função dos parâmetros acima;
- Número de formigas que percorrerá trajetos com base nas probabilidades calculadas durante a execução do algoritmo;
- Ajuste da matriz de feromônio reforçando as arestas percorridas;
- Ajuste da matriz de feromônio considerando uma taxa de evaporação;
- Possibilidade de elitismo por manter a melhor solução encontrada até o momento reforçando a matriz de feromônio para seu trajeto.

Para uma implementação do algoritmo de otimização por colônia de formigas para o problema do roteamento de veículos podemos considerar que uma formiga sairá do depósito e selecionará o próximo cliente para sua rota com base numa probabilidade que considera tanto a matriz de feromônio como a atratividade do cliente em função da sua distância, clientes mais próximos são considerados mais atrativos. Uma vez no cliente seguinte o processo se repete para a escolha do próximo cliente. Entretanto, para cada cliente escolhido acumulamos a demanda e se excedemos a capacidade de carga, então ao invés do deslocamento para o próximo cliente realizamos um deslocamento obrigatório para o depósito, fechamos uma rota e iniciamos outra. A demanda acumulada é zerada e todo o processo se repete até que todos os clientes tenham sido percorridos e terminando a última rota também no depósito.

A probabilidade da formiga na localidade *i* escolher a localidade *j* é dada pela fórmula:

$$P_{ij} = \frac{(\tau_{ij})^{\alpha}.(\eta_{ij})^{\beta}}{\sum_{l \in N_i^k}(\tau_{il})^{\alpha}.(\eta_{il})^{\beta}} \qquad \forall j \in N_i^k$$

Nessa fórmula temos os seguintes parâmetros:

- $\tau_{ij}$ representa a quantidade de feromônio no caminho entre os pontos *i* e *j*;
- $\alpha$ representa um parâmetro para controlar o peso relativo do feromônio no cálculo da probabilidade;
- $\eta_{ij}$ representa a atratividade do caminho entre os pontos *i* e *j*. Em muitas implementações a atratividade é calculada como $\eta_{ij} = 1/d_{ij}$ onde $d_{ij}$ é a distância entre os pontos *i* e *j*, dessa forma uma maior distância possui uma menor atratividade e vice-versa;
- $\beta$ representa um parâmetro para controlar o peso relativo da atratividade no cálculo da probabilidade;
- $N_i^k$ representa o conjunto de todos os pontos do problema para onde a formiga *k* na posição *i* possa ir e que ainda não foram visitados.

Com base no acima, vemos que o problema pode ser calibrado pelos parâmetros $\alpha$ e $\beta$ para pender para uma escolha com base na quantidade de feromônio entre cada aresta ou pender para uma escolha com base numa atratividade de aresta baseada no inverso do seu tamanho.

O algoritmo necessita de uma matriz $\tau_{ij}$ para indicar a quantidade de feromônio entre cada aresta ligando os pontos *i* e *j*. Essa matriz pode ser inicializada com zeros ou com um valor aleatório igual para cada um de seus elementos. Depois de cada iteração a matriz deve ser atualizada com base nos caminhos escolhidos por cada formiga. Ou seja, cada formiga depositará uma quantidade adicional de feromônio. Na implementação feita, a quantidade adicional de feromônio que é somada em cada aresta do caminho feito pela formiga é dada pela fórmula:

$\Delta\tau_{ij} = \frac{1}{L_k}$ onde $L_k$ é a distância total da rota feita pela formiga *k*

Logicamente, se uma aresta não faz parte do caminho dessa formiga, então nada é somado ao elemento $\tau_{ij}$ correspondente na matriz de feromônio.

Além da atualização da matriz de feromônio, o algoritmo também realiza a evaporação do feromônio e para isso usamos a taxa de evaporação $\rho$. A equação da evaporação de feromônio é:

$\tau_{ij}^{novo}$= (1- $\rho$). $\tau_{ij}$

Na implementação feita, calculamos o caminho percorrido por cada uma das formigas e comparamos com o melhor caminho encontrado até o momento. Se uma das formigas teve um caminho menor, então atualizamos o melhor caminho encontrado até o momento.

Para cada iteração, podemos soltar um número *X* de formigas onde cada formiga percorrerá todos os clientes da forma descrita nessa Seção, cada formiga representando uma solução viável para o problema. Definimos também um número máximo de iterações *N*.

É possível implementar um mecanismo de elitismo nesse problema. Em cada iteração geramos uma lista de *X* formigas que serão utilizadas para a atualização da matriz de feromônio. Entretanto, teremos também a melhor formiga dessa lista para a iteração corrente bem como a melhor formiga encontrada desde o início. Dessa maneira, podemos criar um elitismo onde podemos substituir algumas das formigas da iteração corrente pela melhor formiga encontrada até o momento e algumas pela melhor formiga da iteração corrente, e as demais permanecem como estão. Essas substituições farão com que a matriz de feromônio seja atualizada privilegiando boas formigas e acelerando a convergência do problema.

Uma limitação que encontramos nessa implementação é que as primeiras rotas sempre serão construídas procurando-se alcançar o limite da capacidade de carga enquanto a última poderá ficar com uma carga mínima. Ao agir assim, ignoramos que na solução ótima possamos ter uma configuração de carga mais equilibrada. Numa tentativa de reduzir esse impacto criamos um operador de mutação onde ele olha a capacidade faltante para a última rota e busca um dos clientes da penúltima rota que possa ser inserido nela. Se o custo resultante dessa mutação for superior ao custo dessa formiga antes da mutação, então a mutação é desfeita. A Figura 15 apresenta o fluxograma genérico do algoritmo de colônia de formiga e na sequência temos o pseudocódigo do algoritmo implementado. Apenas para efeito de entendimento do pseudocódigo atribuiu-se nele alguns valores como 1.000 para o número máximo de iterações (variável *maxiter* no pseudocódigo) e um número de formigas igual a 100. No entanto esses valores são ilustrativos e podem ser variados nas execuções.

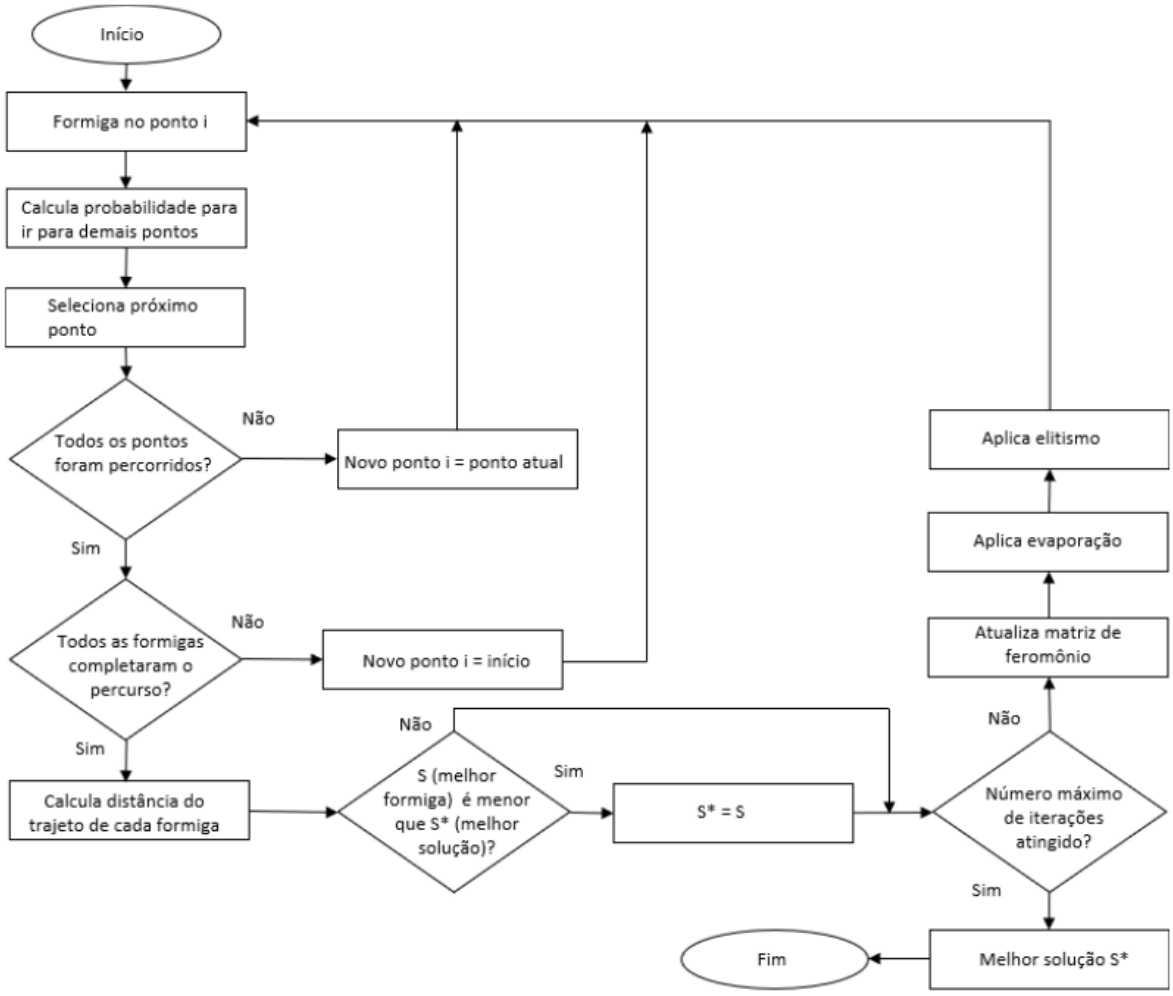



Figura 15 - Fluxograma do algoritmo de colônia de formigas.

Pseudocódigo do algoritmo de formigas

Define maxiter = 1.000; número de formigas = 100; t = 1; melhor valor = +∞

Cria um vetor para cada formiga com o depósito como ponto de partida

**enquanto** t < maxiter **faça**

**para** cada formiga do número de formigas (colônia) **faça**

**enquanto** o vetor de formiga não tiver percorrido todos os clientes **faça**

Seleciona último elemento do vetor formiga para a posição atual

Calcula a probabilidade de ir dele para outro cliente

Probabilidade de ir para elementos já visitados = zero

Probabilidade de ir para cliente sem capacidade da rota = zero

Usa roleta para próximo cliente e insere no vetor formiga

**se** soma das probabilidades para próximos clientes == zero **faça**
Adicione o depósito no vetor da formiga
Identifica a formiga com o menor custo na colônia
**se** menor custo na colônia < melhor valor **faça**
Melhor valor = menor custo na colônia
Melhor formiga = formiga com menor custo na colônia
Início do procedimento de mutação
Calcula o delta de capacidade para completar a capacidade da última formiga
**se** o último cliente da penúltima rota tiver uma capacidade inferior ao delta **faça**
Mover cliente para o final da última rota
**se** o custo da nova formiga for maior que o custo da formiga antes **faça**
Desfaz a mutação
**se** o custo da nova formiga for menor que melhor valor **faça**
Melhor valor = custo da formiga após mutação
Melhor formiga = formiga após mutação
Início do procedimento de elitismo
**para** as 10 primeiras formigas da colônia **faça**
Copiar a melhor formiga até o momento nessas posições
**para** as 20 próximas formigas da colônia **faça**
Copiar a melhor formiga da colônia nessas posições
Atualizar a matriz de feromônio e executa-se a evaporação
Incrementa t

---

### 3.2.4 Aprimoramento da qualidade da solução de uma meta-heurística

Conforme será apresentado no Capítulo de Experimentos, as meta-heurísticas de busca tabu, algoritmo genético e colônia de formigas foram implementadas para o problema básico que contém apenas a restrição de capacidade. Os resultados mostraram uma performance melhor do algoritmo busca tabu e que o algoritmo genético, embora com soluções que podem ser consideradas relativamente boas, apresentou soluções inferiores quando comparadas as outras duas meta-heurísticas. Essa conclusão já era esperada conforme indicada no Capítulo de Revisão da Literatura. Não é difícil de entender o motivo, a busca tabu e a colônia de formigas,

embora tenham um caráter estocástico, contém também movimentos de busca que fazem o algoritmo caminhar de maneira mais decisiva na direção de um aprimoramento da qualidade da solução. Por outro lado, o algoritmo genético tem um caráter muito mais aleatório na geração de novos indivíduos (soluções) do que de busca dirigida. Em função dessa constatação, optou-se pela seleção do algoritmo genético para a investigação de como aprimorar a sua eficiência. Uma das propostas para a melhoria do algoritmo genético no problema do roteamento de veículos foi encontrada no Capítulo da Revisão da Literatura e é o algoritmo *Split* (PRINS, 2004).

#### 3.2.4.1 Algoritmo *Split*

Os algoritmos genéticos abriram as portas de um vasto campo de pesquisa e aplicações. No entanto, inicialmente os resultados para o problema do roteamento de veículos não foram competitivos quando comparados ao desempenho poderoso de algoritmos de busca tabu. Isso levou a implementações híbridas, acoplando o algoritmo genético à heurísticas para uma busca local e levando à uma convergência mais rápida e a resultados melhores. Uma das dificuldades encontradas na implementação do algoritmo genético do problema de roteamento tem que ver com a codificação do cromossomo para representar uma possível solução do problema. Realmente essa dificuldade foi verificada na implementação do algoritmo genético básico feita nesse trabalho. O cromossomo com delimitadores de rotas, conforme apresentado anteriormente e exemplificado abaixo, traz dificuldades para a elaboração do código do programa. Basicamente, fazer os movimentos de *crossover* e de mutação levando em conta esses delimitadores aumenta a complexidade do código de programa.

Exemplo de um cromossomo de solução: 0-3-2-6-0-1-7-0-4-5-8-10-9-0. Nesse exemplo os valores zero representam o depósito e atuam como delimitadores de rotas. Os demais valores representam os clientes do problema. Assim temos três rotas onde a primeira é 0-3-2-6-0, a segunda rota é 0-1-7-0, a terceira é 0-4-5-8-10-9-0.

O trabalho de Prins (2004) resolveu esse problema com a apresentação de um algoritmo *Split* para ser aplicado a uma sequência de clientes sem os delimitadores de rota. Para o exemplo acima o cromossomo seria codificado apenas como: 3-2-6-1-7-4-5-8-10-9.

A grande questão agora passa a ser como traduzir essa sequência em rotas? Certamente há várias possibilidades, mas o algoritmo de Prins resolveu essa questão

de uma maneira elegante, por indicar qual a configuração de rotas ótima que mantém essa sequência. Ao fazer isso, o algoritmo já passa a ter um mecanismo de busca de solução ótima inerente e que contribui para uma maior eficiência. Além disso, a implementação das operações de *crossover* e de mutação no código de programa se torna mais fácil. Isso reduz o esforço requerido para codificação em linguagem de programação e permite um foco maior na busca de aprimoramentos do algoritmo genético. A Figura 16 apresenta a lógica do algoritmo *Split*.

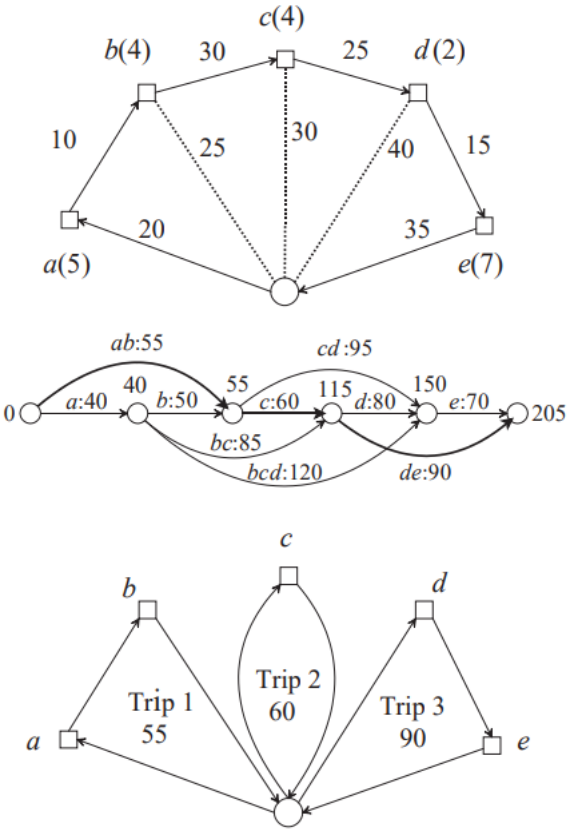



Figura 16 - Lógica do algoritmo *Split*, extraído de Prins (2004).

Na primeira parte da Figura 16 temos um depósito localizado na parte inferior representado pelo círculo e 5 clientes: *a,b,c,d,e*. O ponto importante aqui é que a sequência já está definida como *a-b-c-d-e*. Com isso, temos agora que identificar qual a composição de rotas ótima mas mantendo essa sequência. As distâncias entre cada ponto são indicadas pelos valores nas retas ligando esses pontos. Por exemplo, temos uma distância de 20 entre o depósito e o cliente *a*, temos uma distância de 10 entre o cliente *a* e o cliente *b*, e assim por diante. Os números entre parênteses ao lado da letra de cada cliente indicam as respectivas demandas. Por exemplo, o cliente *a* tem

demanda 5 e o cliente *e* tem demanda 7. Para esse exemplo, assumimos uma capacidade de 10 para cada veículo.

Na segunda parte da Figura 16 temos um grafo auxiliar que nos ajuda a entender a lógica que estamos buscando. Partindo do depósito na extremidade esquerda podemos chegar novamente ao depósito passando pelo cliente *a* percorrendo uma distância total de 40 (20+20) e essa seria uma primeira rota possível. Por outro lado, também podemos sair do depósito e retornar a ele tendo passado pelos clientes *a* e *b*, nesse caso a distância total percorrida é 55 (20+10+25). Essa também é uma rota possível uma vez que a demanda de *a* e *b* somadas é 9 enquanto que a capacidade do veículo é 10 conforme dito inicialmente. Não é possível ter uma rota saindo do depósito e passando pelos clientes *a, b* e *c* uma vez que a demanda somada dos três clientes passa da capacidade do veículo. Dessa forma, temos somente duas rotas possíveis passando pelo cliente a. Aplicando essa mesma lógica para o segundo ponto do depósito do grafo, que é o depósito após ter vindo do cliente a, temos três possíveis rotas: uma primeira somente com o cliente *b* e distância total de 50; uma segunda indo ao cliente *b* e depois passando pelo cliente *c*, retornando ao depósito e com distância total de 85; e uma terceira passando pelos clientes *b, c, d,* e retornando ao depósito com distância total de 120. Seguimos essa lógica até completar o grafo. Uma vez tendo esse grafo, o mesmo pode ser resolvido pelo algoritmo de Bellman chegando num caminho ótimo (PRINS, 2004). Essa solução ótima pode ser visualizada no grafo com as linhas em negrito e que dão um total de 55+60+90=205.

A solução ótima é melhor visualizada na parte inferior da Figura 16 onde vemos as três rotas. Assim, a sequência do cromossomo *a*-*b*-*c*-*d*-*e* para esse problema conteria três rotas dando uma distância total de 205: 0-*a*-*b*-0, 0-*c*-0, 0-*d*-*e*-0.

O grafo ajuda a visualizar a lógica por detrás do algoritmo *Split* mas na prática não é necessário montá-lo. Ao invés disso, podemos executar o algoritmo *Split* apresentado por Prins (2004) e transcrito na sequência. Nesse algoritmo temos algumas variáveis como *load* que carrega a carga acumulada de uma determinada rota, *cost* que carrega o custo acumulado dessa mesma rota, temos $S_j$ que indica o cliente que estiver na *j*-ésima posição na sequência de rotas que compõe a solução específica, $d_{S_j}$ é usado para indicar um custo específico para atender esse cliente

(pode ser zero), *W* que indica a carga máxima total de um veículo e *L* que indica a distância máxima que um veículo pode percorrer.

Algoritmo *Split* original

```
V0 := 0
for i := 1 to n do Vi := +∞ endfor
for i := 1 to n do
    load := 0; cost := 0; j := i
    repeat
        load := load + q_{S_j}
        if i = j then
            cost := c_{0,S_j} + d_{S_j} + c_{S_j,0}
        else
            cost := cost − c_{S_{j−1},0} + c_{S_{j−1},S_j} + d_{S_j} + c_{S_j,0}
        endif
        if (load ≤ W) and (cost ≤ L) then
            if V_{i−1} + cost < V_j then
                V_j := V_{i−1} + cost
                P_j := i − 1
            endif
            j := j+1
        endif
    until (j > n) or (load > W) or (cost > L)
endfor
```

O pseudocódigo acima gera um vetor auxiliar *P* que será usado para extrair a informação de quebra das rotas. Enquanto o vetor *V* indicará os custos, *P* é necessário para traduzir a sequência original sem delimitadores em rotas distintas, mas para isso essa informação precisará ser extraída de *P*. Prins (2004) também apresenta o algoritmo para fazer essa extração. Diferente do algoritmo *Split* que deve ser executado para o cálculo do custo de cada solução, esse algoritmo de extração de rotas somente precisa ser executado no final quando já tivermos a melhor solução e que deverá ser apresentada com a quebra de rotas. O algoritmo de extração é

apresentado na sequência. Nesse algoritmo a função *enqueue(trip(t), $S_k$)* adiciona o cliente $S_k$ na rota *trip(t).*

Algoritmo de extração de rotas a partir de $P_n$

```
for i := 1 to n do trip(i) := 0 endfor
t := 0
j := n
repeat
        t := t + 1
        i := Pj
        for k := i+1 to j do enqueue(trip(t), Sk) endfor
        j := i
until i = 0
```

No algoritmo *Split* temos uma variável *V* indexada para cada cliente e ela armazena o custo (ou distância) para se chegar em cada cliente. A variável $V_e$ terá o valor de 205 que é a distância percorrida desde o início até a chegada nela. Esse é o valor da melhor distância total para essa sequência de rotas. A variável *P* é uma variável criada para armazenar o predecessor. Para o problema da Figura 16 teremos $P_a = P_b = 0$, $P_c = P_d = 2$, $P_e = 3$.

O algoritmo de extração faz a tradução desses valores armazenados em P nas rotas do problema. Basicamente a execução desse algoritmo indicará o seguinte:

O menor custo para se chegar no último cliente (cliente *e*) retornar ao depósito é 205. Como $P_e = 3$, então isso indica que a rota que chega no cliente *e* partiu do depósito depois de ele ter visitado o cliente *c* ($P_e = 3$, e o cliente *c* é o terceiro cliente na sequência). Assim, a última rota é 0-*d*-*e*-0.

Como $P_c = 2$, então a rota que vai para o cliente *c* se inicia no depósito depois que passou pelo cliente *b* (segundo cliente na sequência), assim a rota é 0-*c*-0.

Como $P_b = 0$, então a rota que vai para o cliente *b* se inicia no depósito antes de ter visitado qualquer cliente, e com isso a rota é 0-*a*-*b*-0.

Com a inclusão desse algoritmo e da codificação do cromossomo de solução do problema de roteamento sem delimitadores, o desempenho do algoritmo genético torna-se mais eficiente.

#### 3.2.4.2 Aprimoramento através de operadores de *crossover*

Além do método de *Split* proposto por Prins (2004), uma segunda forma de buscar aprimorar a eficiência do algoritmo genético baseia-se no trabalho de Blanton e Wainwright (1993). Eles propuseram dois operadores de *crossover* (MX1 e MX2) que basicamente comparam cada gene dos cromossomos pais com um vetor de precedência e selecionam o gene do Pai 1 ou do Pai 2 dependendo de quem aparece primeiro no vetor de precedência. Essa solução torna-se muito útil para o problema com janelas de tempo. Nessa variante pode-se criar um vetor de precedência ordenando todos os clientes com base no parâmetro do instante final da janela de tempo, do cliente que possui a janela de tempo que se encerra mais cedo para o cliente que tem a janela de tempo que se encerra por último. Fica claro quais são os clientes que devem ser atendidos primeiro.

Para o problema de roteamento apenas com a restrição de capacidade não temos a questão da precedência. No entanto, empregando-se esse mesmo conceito podemos criar um algoritmo que realiza o *crossover* entre os dois cromossomos pais utilizando uma regra de preferência com base na distância (ou custo) com o gene precedente.

- Passo 1: Selecionam-se dois cromossomos, Pai 1 e Pai 2.
- Passo 2: Seleciona-se o primeiro gene do Pai 1 para ser o primeiro gene do cromossomo Filho.
- Passo 3: No cromossomo Pai 2, busca-se em que posição se encontra o gene que aparece na primeira posição do Pai 1 e troca-se ele pelo primeiro gene do cromossomo do Pai 2.
- Passo 4: Passamos para o próximo gene do Pai 1, comparamos esse gene com o gene na mesma posição no Pai 2, selecionando dentre esses dois qual tem a menor distância ou custo em relação ao gene na posição anterior no cromossomo Filho. Esse gene é adicionado ao cromossomo Filho.
- Passo 5: No caso do pai que não teve seu gene selecionado, busca-se nele o gene escolhido e faz-se a troca com o gene não escolhido.

- Passo 6: Volta-se ao passo 4 até o fim do cromossomo.

Esse procedimento fica mais claro com um exemplo. Nele temos dois cromossomos que serão utilizados como cromossomos Pai para gerar um cromossomo Filho.

Exemplo:

- Pai 1: 2-4-6-3-1-5
- Pai 2: 3-5-1-4-2-6

Ao executarmos o passo 1, selecionamos o primeiro gene do Pai 1 e que vale 2 para ser o primeiro gene do cromossomo filho. Além disso, o cromossomo inicial do Pai 2 necessita ser mudado de posição com o gene selecionado para o *crossover*. Como isso passamos a ter a seguinte configuração:

- Pai 1: 2-4-6-3-1-5
- Pai 2: 2-5-1-4-3-6
- Filho: 2

Agora pegamos o próximo gene de ambos os pais (4 para o Pai 1 e 5 para o Pai 2) e comparamos ambos com o último gene no cromossomo Filho (2 no caso) e escolhemos o gene que possui a menor distância em relação a ele. Vamos supor que é o gene do Pai 2. Nesse caso, esse gene é adicionado ao cromossomo Filho. Além disso, esse gene no cromossomo do Pai 1 deverá ser substituído pelo gene não escolhido. Com isso passamos a ter a seguinte configuração:

- Pai 1: 2-5-6-3-1-4
- Pai 2: 2-5-1-4-3-6
- Filho: 2-5

Agora passamos para o próximo gene nos cromossomos pais (6 para o Pai 1 e 1 para o Pai 2). Eles necessitam ser comparados em distância (ou custo) com o último gene do cromossomo Filho (no caso o gene 5). Vamos supor que o gene do cromossomo do Pai 1 foi o selecionado, então ele é adicionado ao cromossomo Filho. Além disso, esse mesmo gene mas no cromossomo do Pai 2 precisa ser substituído pelo gene não selecionado. Assim, passamos a ter:

- Pai 1: 2-5-6-3-1-4
- Pai 2: 2-5-6-4-3-1
- Filho: 2-5-6

E assim sucessivamente até o cromossomo Filho ficar completo. Com isso, teremos uma operação de crossover que será executada com uma lógica inerente de tentar reduzir o custo da solução, e não totalmente aleatória.

#### 3.2.4.3 Implementação das técnicas de melhoramento no algoritmo genético do problema com restrição de capacidade

As duas técnicas descritas acima, do algoritmo de *Split* de Prins e do operador de *crossover* com a comparação das distâncias do próximo gene com o gene antecessor foram implementadas num algoritmo genético conforme a descrição abaixo.

- População inicial: Selecionam-se 100 indivíduos totalmente aleatórios usando o comando "*shuffle*" no Python.
- Para cada indivíduo, executa-se uma função que calcula o melhor *Split* (usando o algoritmo de Prins) e seu custo. O indivíduo com o melhor custo já passa a ser a melhor solução até o momento.
- Para cada geração executam-se operações de *crossover*. Em cada operação, um par de indivíduos é selecionado aleatoriamente para servirem como Pai 1 e como Pai 2.
    - A primeira operação de *crossover* é conhecida como OX (*order crossover*). Nela selecionamos duas posições do cromossomo Pai 1 e copiamos os genes compreendidos por elas para o cromossomo Filho, nas mesmas posições. Depois copiam-se os genes do cromossomo do Pai 2 para o cromossomo Filho a partir da posição final do intervalo de seleção tomando-se o cuidado para não repetir genes já recebidos do Pai 1.
    - Uma segunda operação de *crossover* é feita utilizando-se a técnica descrita na Seção 3.2.4.2 de comparação de distâncias.
    - Uma terceira operação de *crossover* é feita utilizando-se a técnica de comparação de distâncias, mas dessa vez começando-se de trás para frente. Ou seja, selecionamos o último gene do Pai 1 para ser o último gene do Filho. Daí seguimos para comparar os penúltimos genes dos cromossomos pais para selecionar qual será o penúltimo gene do cromossomo Filho, e assim sucessivamente.
- Depois ter termos efetuado as operações de *crossover* e gerado novos indivíduos para a nova população, executamos três operações de mutação.

- Na primeira operação de mutação selecionamos um indivíduo aleatoriamente e depois dois genes desse indivíduo que são trocados de posição. Se essa troca resulta num indivíduo com um custo menor, então a troca é mantida. Caso contrário, a troca é desfeita.
- A segunda operação de mutação é idêntica a primeira, com a exceção de que a troca é mantida mesmo que o resultado seja uma piora do custo.
- A terceira operação de mutação é conhecida como 2OPT, nela selecionamos um indivíduo e depois duas posições desse indivíduo. A sequência de genes compreendida entre essas duas posições é invertida. Caso essa inversão resulte num custo inferior ao indivíduo original, a mutação é mantida.

- Importante destacar que cada vez que um indivíduo novo é gerado, seja por uma operação de *crossover*, seja por uma operação de mutação, executamos o algoritmo *Split* para esse novo indivíduo para verificar seu custo. Caso o custo seja inferior ao da melhor solução até o momento, então essa melhor solução até o momento é atualizada.
- Uma função de elitismo é adicionada onde escolhe-se um indivíduo da geração corrente e ele é trocado pela melhor solução até o momento.

Essa proposta de algoritmo genético apresenta algumas vantagens:

- Implementação em código de programa Python muito mais fácil em função de não necessitarmos lidar com os delimitadores de rota na codificação do cromossomo de solução. O programa de computador torna-se muito mais claro e menor.
- O algoritmo de *Split* já garante para uma determinada sequência de clientes qual é a divisão ótima em rotas, o que já contribui por si só para a eficiência do programa. Podemos dizer que temos um algoritmo de otimização para uma sequência específica inerente ao algoritmo completo.
- A operação de *crossover* OX bem como a mutação independentemente de melhorar a solução ou não contribuem para a diversificação das soluções e para escapar de ótimos locais. Por outro lado, as operações de *crossover* com comparação de distâncias bem como a mutação somente quando há uma melhoria da solução contribuem para termos uma busca local dentro do algoritmo de forma a promover a busca de uma melhoria da solução e não deixar essa busca totalmente aleatória.

Segue o pseudocódigo para o algoritmo genético aprimorado conforme concebido acima. Os valores de número de gerações, tamanho de população, número

de execuções de *crossover* e mutações são ilustrativos e refletem uma implementação específica durante os experimentos.

Pseudocódigo do algoritmo genético aprimorado

População = 100; geração = 0
Gera população inicial com 100 indivíduos aleatórios (sem delimitadores de rota)
**função** *Split* calculada para cada indivíduo
Melhor solução e melhor custo recebem o indivíduo com o menor custo da população
Número de gerações = 500
**enquanto** geração < Número de gerações **faça**
    *Início do crossover de diversificação OX*
    Número de reproduções por geração = 50
    Contador = 1
    **enquanto** Contador < Número de reproduções **faça**
        Seleciona aleatoriamente 2 indivíduos
        Executa *crossover* OX e gera dois filhos
        **se** o custo dos filhos for inferior aos dois pais faça
            Troca os pais pelos filhos na população
        Incrementa Contador
    **se** mínimo da população < melhor custo **faça**
        Melhor custo = Custo do mínimo da população
        Melhor solução = Mínimo da população
    *Início do crossover de intensificação com ordem crescente*
    Número de reproduções = 30
    Contador = 1
    **enquanto** Contador < Número de reproduções **faça**
        Seleciona aleatoriamente 2 indivíduos
        *Crossover* MS1 crescente
        **se** custo do Pai 1 > custo do Filho **faça**
            Pai 1 = Filho
        **se** custo do Pai 2 > custo do Filho faça
            Pai 2 = Filho
        **se** custo do Filho < Melhor custo **faça**

Melhor custo = custo do Filho

Melhor solução = Filho

*Início do crossover de intensificação com ordem decrescente*

Número de reproduções = 30

Contador = 1

**enquanto** Contador < Número de reproduções **faça**

Selecionar aleatoriamente 2 indivíduos

*Crossover* MS1 decrescente

**se** custo do Pai 1 > custo do Filho **faça**

Pai 1 = Filho

**se** custo do Pai 2 > custo do Filho **faça**

Pai 2 = Filho

**se** custo do Filho < Melhor custo **faça**

Melhor custo = custo do Filho

Melhor solução = Filho

*Início do procedimento de mutação para intensificação*

**Faça** 50 vezes

Seleciona indivíduo aleatoriamente e calcula *Split* para ele

Seleciona dois pontos aleatórios

Executa a mutação trocando os dois genes

Calcula *Split* para o indivíduo

**se** custo novo indivíduo < Melhor custo **faça**

Melhor custo = custo novo indivíduo

Melhor solução = novo indivíduo

**se** solução antes < solução depois **faça**

Desfaz mutação

*Início do procedimento de mutação para diversificação*

**Faça** 50 vezes

Seleciona indivíduo aleatoriamente

Seleciona um ponto aleatório e o ponto seguinte na sequência

Executa a mutação trocando os dois genes

Calcula *Split* para o indivíduo

**se** custo novo indivíduo < Melhor custo **faça**

Melhor custo = custo novo indivíduo

Melhor solução = novo indivíduo
Início do procedimento de mutação 2OPT
**Faça** 50 vezes
Seleciona indivíduo aleatoriamente
Seleciona dois pontos aleatórios do cromossomo
Executa inversão da sequência entre esses dois pontos
**se** custo novo indivíduo < custo indivíduo original **faça**
Atualiza indivíduo na população
**se** custo novo indivíduo < Melhor custo **faça**
Melhor custo = custo novo indivíduo
Melhor solução = novo indivíduo
*Início do procedimento de elitismo*
Seleciona dois indivíduos aleatórios da população
Troca esses dois indivíduos pela melhor solução até o momento
Incrementa geração

---

### 3.2.4.4 Implementação das técnicas de melhoramento no algoritmo genético do problema com janelas de tempo

Uma das vantagens do algoritmo de *Split* de Prins (2004) é que ele pode ser adaptado para outras variantes do problema de roteamento de veículos. Chang e Chen (2007) apresentaram uma variação do algoritmo original para problemas com janelas de tempo com $e_i > 0$ e $l_i = \infty$ para todos os clientes, ou seja, janelas de tempo com o momento de início e sem término de janela. Adicionalmente, assume-se $d_{S_j}$ como o tempo de serviço do cliente que estiver na j-ésima posição na sequência da solução, e não mais como o custo pelo seu atendimento. Para o problema com janelas de tempo devemos além do custo entre o cliente *a* e o cliente *b*, também ter o tempo de deslocamento entre esses clientes. Além disso, devemos ter a variável que acumule o tempo decorrido desde a origem até passar por cada cliente da rota. Chang e Chen (2007) assumiram que $t_{ij} = c_{ij}$. Dessa forma, a variável *cost* que no algoritmo *Split* original acumulava o custo da rota, agora passa a acumular o tempo de deslocamento da rota. Logicamente, outras implementações podem ser feitas assumindo $t_{ij}$ como uma função de $c_{ij}$.

Algoritmo *Split* para problema com janela de tempo com instante inicial e sem final

$V_0$ := 0
for i := 1 to n do $V_i$ := +∞ endfor
for i := 1 to n do
    load := 0; cost := 0; j := i
    repeat
        load := load + $q_{S_j}$
        if i = j then
            cost := $\max [c_{0,S_j}, e_{S_j}] + d_{S_j} + c_{S_j,0}$
        else
            cost := $\max [cost - c_{S_{j-1},0} + c_{S_{j-1},S_j}, e_{S_j}] + d_{S_j} + c_{S_j,0}$
        endif
        if (load ≤ W) and (cost ≤ L) then
            if $V_{i-1} + cost < V_j$ then
                $V_j := V_{i-1} + cost$
                $P_j := i - 1$
            endif
            j := j+1
        endif
    until (j > n) or (load > W) or (cost > L)
endfor

O algoritmo para a extração da informação das rotas a partir do vetor *P* segue idêntico ao original.

Para esse trabalho apresentamos a seguir a modificação do algoritmo considerando janelas de tempo com momentos definidos de início e de fim. Pode-se ver isso pela inclusão do parâmetro $l_{S_j}$, que determina o fim da janela de tempo, no corpo do algoritmo. Utilizamos a simplificação de Chang e Chen (2007) de que $t_{ij} = c_{ij}$. Também mantemos a variável *cost* para acumular o tempo decorrido pela rota desde a partida no depósito. Novamente, o algoritmo de extração da informação das rotas a partir do vetor *P* permanece idêntico ao original, para ser executado somente no final do programa para a melhor solução encontrada.

Algoritmo *Split* para problema com janelas de tempo com início e fim

$V_0$ := 0
for i := 1 to n do $V_i$ := +∞ endfor
for i := 1 to n do
    load := 0; cost := 0; j := i
    repeat
        load := load + $q_{S_j}$
        if i = j then
            cost := $\max [c_{0,S_j}, e_{S_j}] + d_{S_j} + c_{S_j,0}$
        else
            cost := $\max [cost - c_{S_{j-1},0} + c_{S_{j-1},S_j}, e_{S_j}] + d_{S_j} + c_{S_j,0}$
        endif
        if (load ≤ W) and ($cost - c_{S_j,0} - d_{S_j} \leq l_{S_j}$) then
            if $V_{i-1} + cost < V_j$ then
                $V_j := V_{i-1} + cost$
                $P_j := i - 1$
            endif
            j := j+1
        endif
    until (j > n) or (load > W) or ($cost - c_{S_{j-1},0} - d_{S_{j-1}} > l_{S_{j-1}}$)
endfor

Com a adaptação do algoritmo *Split* para contemplar janelas de tempo, a codificação do cromossomo é exatamente a mesma do algoritmo genético para o problema do roteamento capacitado. Com isso o programa pode ser facilmente modificado por se chamar a execução do algoritmo *Split* como uma função do problema. Em outras palavras, toda a complexidade adicional causada pelo acréscimo das janelas de tempo fica restrita à função do algoritmo *Split*, e a estrutura do algoritmo genético permanece a mesma. Logicamente, para melhorarmos a eficiência do algoritmo genético podemos acrescentar uma inteligência que reconheça a nova

característica do problema oriunda das janelas de tempo. Podemos fazer isso por implementar a técnica descrita por Blanton e Wainwright (1993) do *crossover* que leva em conta a precedência de genes com relação ao término da janela de tempo. No problema de roteamento capacitado selecionávamos o gene que possuía a distância mais próxima do gene anterior no cromossomo filho, agora dentre as duas opções selecionamos a que possui o menor instante de término de sua janela de tempo. Para a implementação desse trabalho, mantivemos o *crossover* OX que contribui para a diversificação de soluções e mantivemos o *crossover* que seleciona o gene mais próximo de forma a contribuir para a intensificação das soluções.

#### 3.2.4.5 Implementação das técnicas de melhoramento no algoritmo genético do problema com frota mista

Da mesma forma que acontece na variação com janelas de tempo, o algoritmo *Split* também pode ser modificado para contemplar a existência de uma frota mista onde os veículos possam diferir em termos de capacidade, custo fixo e custo variável. Prins (2009) explica como o algoritmo pode ser modificado para considerar diferentes tipos de veículos. Seguindo a explicação dada, o algoritmo foi modificado e segue abaixo. Mais uma vez, o algoritmo para extração das rotas a partir do vetor *P* permanece inalterado. Temos alguns parâmetros novos: *t* é utilizado para indicar o número de tipos diferentes de veículos, *Type* que é um vetor alinhado com *P* e que indicará o tipo do veículo para a rota, *f[k]* indica o custo fixo do veículo *k* e *v[k]* indica o custo variável.

Algoritmo *Split* para problema com frota mista

```
t := {número de tipos diferentes de veículos}
V0 := 0;
for i := 1 to n do Vi := +∞ endfor
Type[0]= +∞
for i :=1 to n do Type[i] := 0 endfor
for i := 1 to n do
        load := 0; cost := 0; j := i
        repeat
```

load := load + $q_{S_j}$

if i = j then

cost := $c_{0,S_j} + c_{S_j,0}$

else

cost := $cost - c_{S_{j-1},0} + c_{S_{j-1},S_j} + c_{S_j,0}$

endif

if (load ≤ W) then

Z := +∞

for k:=1 to t do

$aux := f[k] + v[k].cost$

if $(aux < Z)\ and\ (load \leq Q[k])$ then

$Z := aux$

$car := k$

if $V_{i-1} + Z < V_j$ then

$V_j := V_{i-1} + Z$

$P_j := i - 1$

$Type[j] := car$

endif

j := j+1

endif

until (j > n) or (load > W)

endfor.

---

### 3.2.5 Algoritmo genético para o problema de uma frota mista de bicicletas e motocicletas, com janelas de tempo e penalidades para subidas

Todas as seções anteriores contribuíram para uma compreensão do problema do roteamento de veículos bem como de técnicas de solução. Aqui o algoritmo genético apresentado anteriormente é expandido para abordar as variações de janelas de tempo e de frota mista bem como incorporar o modelamento de bicicletas com penalidades para subidas e chegar no objetivo final que é modelar o problema completo. Seguindo os exemplos anteriores, o algoritmo *Split* de Prins (2004) é modificado. Basicamente toda a complexidade da formulação desse problema se

limita na modificação do algoritmo *Split* uma vez que o algoritmo genético (geração de população, reproduções, mutações) permanece inalterado. Uma consequência dessa abordagem é que toda a preocupação com as características do problema e suas especificidades se concentra no desenvolvimento do algoritmo *Split* mantendo o código do algoritmo genético na sua formulação básica.

A seguir encontra-se a modificação do algoritmo *Split* para esse modelo. Nessa modificação adotou-se que o tempo de deslocamento das motocicletas é 4 vezes mais rápido do que das bicicletas. Em algumas linhas o parâmetro $t_{ij}$ é dividido por 4 para levar essa diferenciação em conta. Pode-se perceber isso nas linhas que envolvem as variáveis *time_b* e *time_m*, onde a primeira é usada para acumular o tempo de deslocamento de uma bicicleta numa determinada rota e a outra o tempo de deslocamento de uma motocicleta para a mesma rota. Também passamos a ter *cost* e *costb,* onde a primeira carrega a distância acumulada da rota considerando $c_{ij}$ como uma matriz de distâncias e a segunda carrega a distância acumulada da rota acrescida de eventuais penalizações de subidas íngremes para a utilização por bicicleta. Percebe-se no algoritmo que ele calcula de forma paralela tanto a distância como o tempo de deslocamento para os dois tipos de veículos: motocicleta e bicicleta. Uma variável auxiliar “janela” assume o valor de “1” quando uma rota viola a janela de tempo de um cliente no caso do emprego de bicicleta. Isso se mostrou necessário para evitar que o algoritmo mudasse uma rota em análise de motocicleta para bicicleta se isso causasse uma violação na janela de tempo de algum cliente dessa rota. No caso dos vetores $tf[\,], f[\,], v[\,]$, o índice “0” se refere ao emprego de bicicleta e “1” ao emprego de motocicleta. O algoritmo escolhe qual o melhor veículo verificando o atendimento de todas as restrições (capacidade e janelas de tempo) e privilegiando o que possuir o melhor custo, sendo que o custo da bicicleta é penalizado no caso das subidas íngremes. A seleção de qual é a melhor opção é armazenada no vetor *Type_car* alinhado com o vetor *P*. Dessa forma, quando se executar a extração das rotas a partir de *P*, pode-se também extrair a informação de qual tipo de veículo cada rota deve usar.

Comparando o algoritmo a seguir com o original de Prins (2004) vemos um aumento de sua complexidade uma vez que adicionamos restrições e especificidades não presentes no algoritmo original.

Algoritmo *Split* para problema com frota mista e janelas de tempo

$V_0$ := 0

for i := 1 to n do $V_i$ := +∞ endfor

for i := 1 to n do

load := 0; cost := 0; costb :=0; j :=i; car :=0; janela :=0

repeat

load := load + $q_{S_j}$

if i = j then

cost := $c_{0,S_j} + c_{S_j,0}$

costb := $p_{0,S_j} + p_{S_j,0}$

time_b := $\max\left[t_{0,S_j}, e_{S_j}\right] + d_{S_j} + t_{S_j,0}$

time_m := $\max\left[t_{0,S_j}/4, e_{S_j}\right] + d_{S_j} + t_{S_j,0}/4$

else

cost := $cost - c_{S_{j-1},0} + c_{S_{j-1},S_j} + c_{S_j,0}$

costb := $cost - p_{S_{j-1},0} + p_{S_{j-1},S_j} + p_{S_j,0}$

time_b := $\max\left[time\_b - t_{S_{j-1},0} + t_{S_{j-1},S_j}, e_{S_j}\right] + d_{S_j} + t_{S_j,0}$

time_m := $\max\left[time\_m - \frac{t_{S_{j-1},0}}{4} + \frac{t_{S_{j-1},S_j}}{4}, e_{S_j}\right] + d_{S_j} + \frac{t_{S_j,0}}{4}$

endif

$tf[0] := time\_b - t_{S_j,0} - d_{S_j}$

$tf[1] := time\_m - t_{S_j,0}/4 - d_{S_j}$

if $(tf[0] > l_{S_j})$ then

janela := 1

if (load ≤ W) and (($tf[0] \leq l_{S_j}$) or ($tf[1] \leq l_{S_j}$)) then

$Z_{ij} = 9999$

if $janela == 0$ and $load \leq Q[0]$ then

custo_bi := $f[0] + v[0] * costb$

else

custo_bi := 9999

custo_mo := $f[1] + v[1] * cost$

if $custo\_bi \leq custo\_mo$ then

$car := 0$

$Z_{ij} := custo\_bi$

else

$car := 1$

$Z_{ij} := custo\_mo$

if $V_{i-1} + Z_{ij} < V_j$ then

$V_j := V_{i-1} + Z_{ij}$

$P_j := i - 1$

$Type\_car[j] := car$

endif

$j := j + 1$

endif

until (j > n) or (load > W) or ($time\_m - (t_{S_{j-1},0}/4) - d_{S_{j-1}} > l_{S_{j-1}}$)

endfor

# CAPÍTULO 4 EXPERIMENTOS

Nesse Capítulo são apresentados os experimentos com as implementações dos modelos e algoritmos abordados no Capítulo de Metodologia.

## 4.1 Modelamento matemático

Os modelos matemáticos foram implementados em Python e rodados em um PC com processador Core i7, clock 1.3GHz, 8GB de memória RAM, 256GB SSD. Em Python usamos dois *frameworks* para a resolução do problema: PuLP e também o Pyomo. No caso do Pyomo, utilizamos como *solver* o CPLEX bem como o Gurobi. Tanto o PuLP como o Pyomo se mostraram excelentes ferramentas para a solução dos modelos empregados.

### 4.1.1 Modelo com restrição de capacidade

O problema de roteamento com restrição de capacidade (CVRP) foi solucionado através do PuLP conforme descrito abaixo e apresentado na Figura 17.

- Número de clientes = 10 na Figura 17 (a) e (b); 11 na Figura (c), 12 na Figura (d), 13 na Figura (e) e 15 na Figura (f);
- Geração aleatória das coordenadas do depósito e dos clientes;
- Geração aleatória da demanda de cada cliente indicada pelo número ao lado de cada cliente;
- Capacidade do veículo = 15;
- Soluções ótimas encontradas com tempo de execução inferior a 5 segundos para a instância de 10 clientes e quase 700 segundos para a instância com 15 clientes.

Tomando a Figura 17a como exemplo, vemos que a solução ótima é encontrada utilizando-se 4 veículos. Como a capacidade do veículo foi indicada acima como 15, temos uma rota atendendo uma demanda total de 11, outra de 12, outra de 13 e outra de 14. Vemos que a demanda total é de 50 e não há como essa demanda ser atendida por menos que 4 veículos se a capacidade de cada um deles é 15. O restante da Figura 17 mostra as soluções ótimas encontradas para a minimização do custo total. Problemas com mais de 15 clientes começam a demandar um tempo de execução bem maior e que pode se tornar inviável.

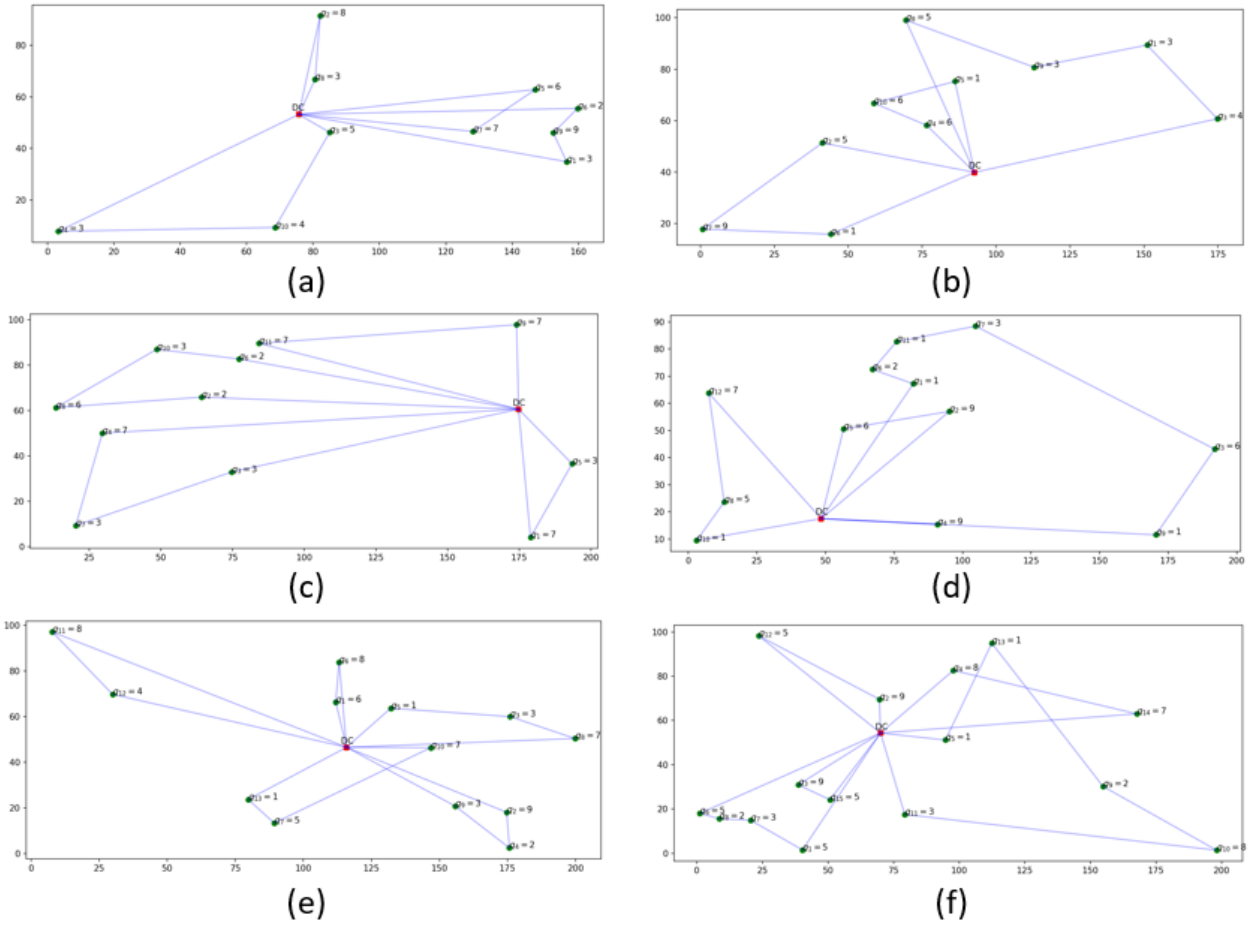



Figura 17 - Soluções de problemas com restrição de capacidade pelo PuLP.

### 4.1.2 Análise da adaptação para bicicletas com penalização de subidas

Apresentamos a seguir os resultados utilizando a instância P-n19-k2 do problema com restrição de capacidade e que apresenta 18 clientes mais a origem. Essa instância P-n19-k2 é uma instância disponível em http://vrp.galgos.inf.puc-rio.br/index.php/en/ para utilização em problemas de roteamento de veículos capacitado (CVRP) e que foi usada nesse experimento para a validação do modelo matemático e de sua implementação em Python. A capacidade máxima dos veículos é de 160. A instância possui solução ótima de 212 com a utilização de dois veículos. A Figura 18 mostra a distribuição dos 18 clientes em relação à origem. A instância fornece as coordenadas *(x,y)* dos clientes e as distâncias entre eles são calculadas pelo método Euclidiano.

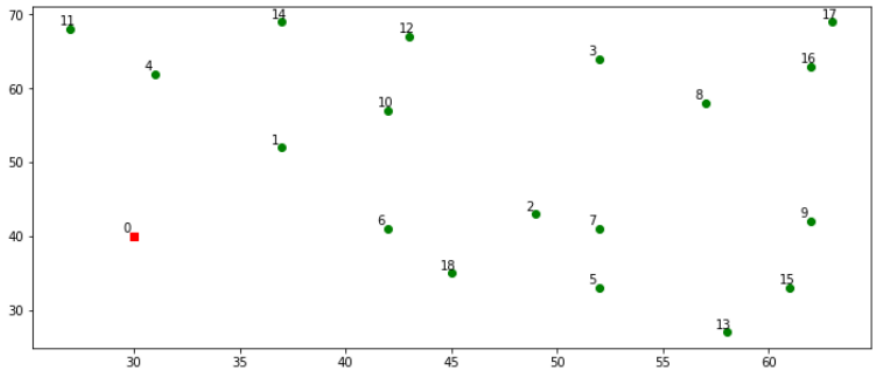



Figura 18 - Posicionamento dos clientes na instância P-n19-k2.

Utilizamos o modelo matemático apresentado para o problema do roteamento de veículos com restrição de capacidade (PRVC) implementado em Python através do Pyomo com o Gurobi como *solver* e confirmamos a solução ótima de 212 com 2 veículos conforme a Figura 19.

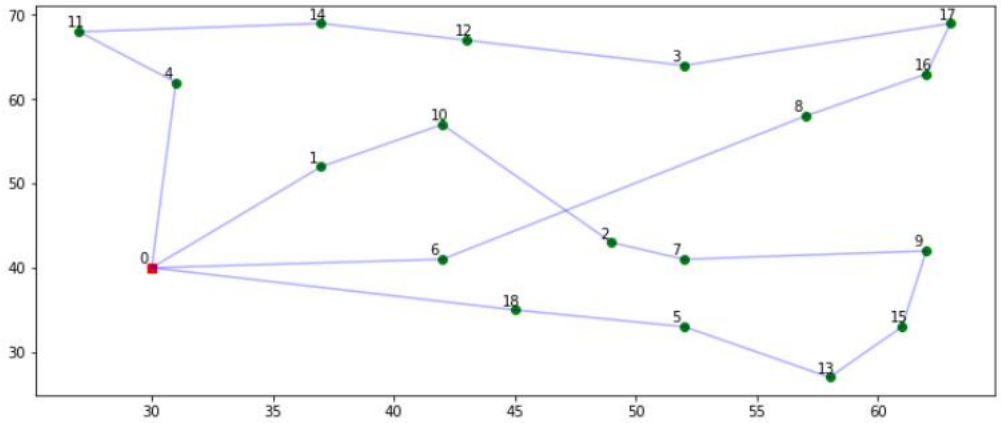



Figura 19 - Solução ótima do P-n19-k2 encontrada pelo Pyomo-Gurobi.

Percebemos duas rotas:

Primeira rota: 0 – 4 – 11 – 14 – 12 – 3 – 17 – 16 – 8 – 6 – 0

Segunda rota: 0 – 18 – 5 – 13 – 15 – 9 – 7 – 2 – 10 – 1 – 0

Com base na solução acima, podemos acrescentar a adaptação para o problema das bicicletas considerando diferenças de elevação. Podemos atribuir altura 0 para a origem bem como todos os clientes, com exceção dos clientes 7, 9, 15 e 13. Iremos atribuir altura 1,5 para os clientes 7 e 13, 0.8 para o cliente 9 e 1,2 para o cliente 13. Dessa forma criamos uma dificuldade para o ciclista da segunda rota seguir do cliente 5 diretamente para o cliente 13 (conforme rota da solução ótima original) e essa dificuldade é refletida na função objetivo com o acréscimo de uma penalidade

forte. Por outro lado, o ciclista pode chegar no cliente 13 por meio de uma outra rota passando por subidas intermediárias, mas mais suaves, e a partir delas chegar no cliente 13. Vemos a nova solução ótima encontrada na figura 20.

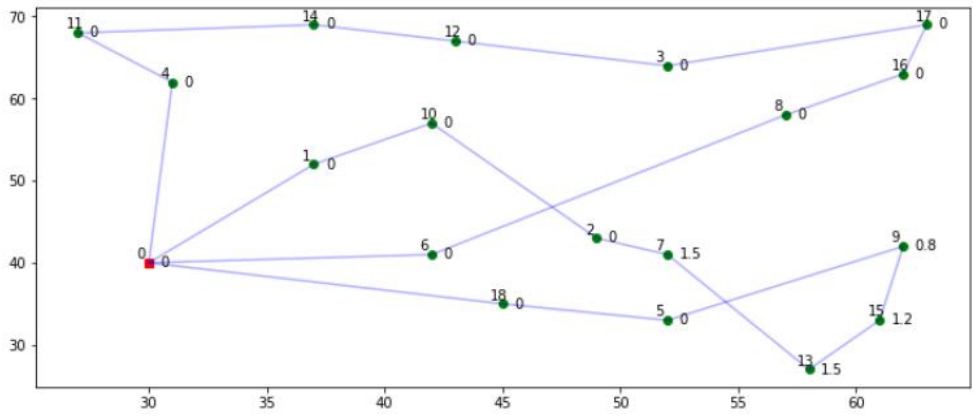



Figura 20 - Solução ótima para o problema com bicicletas.

Nessa nova configuração, há um ângulo de subida de pouco mais de 10 graus entre os clientes 5 e 13, penalizando fortemente a função objetivo. Diante dessa situação, a rota prefere ir do cliente 5 para o cliente 9 onde o ângulo de subida é bem menor, de lá para o cliente 15, e do cliente 15 para finalmente o cliente 13, e daí em diante até o destino. Dessa maneira vemos como o problema buscou uma solução ótima que suavizasse as subidas. Na Figura 20 os números à esquerda dos pontos se referem aos números dos clientes e os números à direita dos clientes se referem aos valores de elevação.

### 4.1.3 Problema com janelas de tempo

O problema com janelas de tempo foi configurado com os seguintes parâmetros:

- 10 clientes com demanda aleatória inteira entre 1 e 10;
- Coordenadas aleatórias para os clientes e para o depósito;
- Capacidade do veículo igual a 15;
- Início da janela de tempo aleatória com valores inteiros entre 5 e 30 para cada cliente;
- Fim da janela de tempo igual ao início da janela somado com um valor inteiro aleatório entre 10 e 20;

- Tempo de serviço em cada cliente com o valor inteiro aleatório entre 3 e 7.

A Figura 21 apresenta uma solução encontrada para o problema respeitando tanto a restrição de capacidade como as restrições de janelas de tempo. Percebe-se que a solução encontrada emprega 3 veículos.

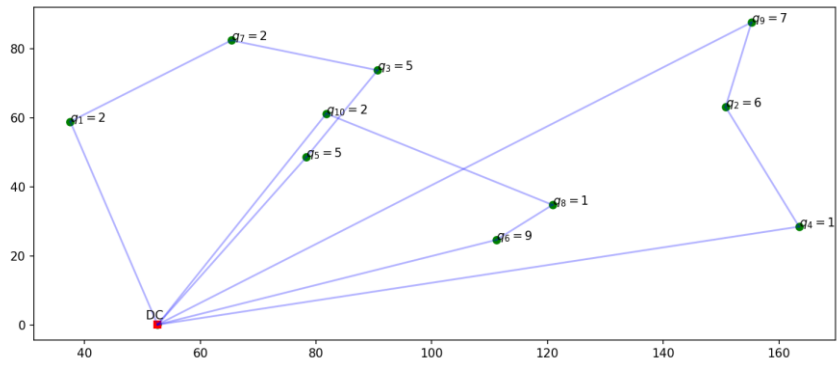
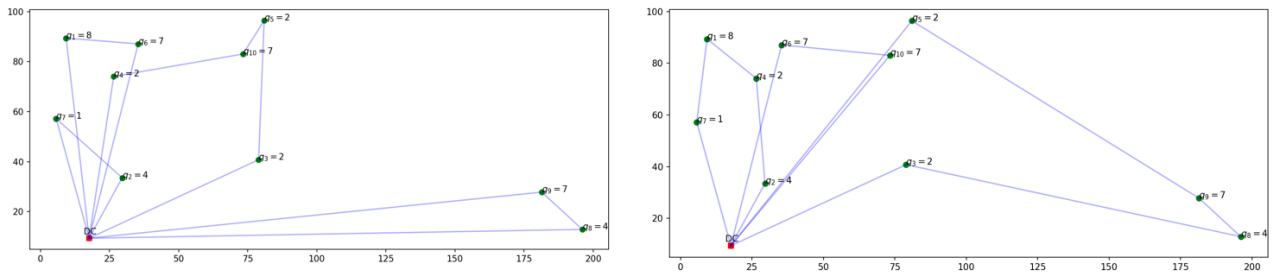



Figura 21 - Solução ótima para 10 clientes com janelas de tempo.

A Figura 22a apresenta uma solução ótima de 913,62 encontrada para o problema com janelas de tempo com 10 clientes e utilizando 4 veículos, já a Figura 22b mostra uma solução ótima de 822,28 e com apenas 3 veículos para o mesmo problema, mas sem a restrição de janelas de tempo. Essa comparação ilustra bem que a adição de janelas de tempo pode penalizar o problema exigindo uma distância total maior e talvez aumente o número de veículos, como é o caso desse exemplo. A solução pode também mudar muito dependendo do tamanho que for definido para as janelas de tempo, onde janelas de tempo pequenas poderão exigir um número maior de veículos. No limite, o número de veículos será igual ao número de clientes.




(a) (b)

Figura 22 - Exemplo com e sem janelas de tempo.

### 4.1.4 Problema com frota heterogênea com penalidades para subidas

O problema com frota heterogênea foi implementado no Python e configurado com os seguintes parâmetros:

- 9 clientes com demanda aleatória inteira entre 1 e 10 para cada cliente e com coordenadas aleatórias, altura aleatória com valores inteiros entre 0 e 25;
- Frota composta de bicicletas com capacidade de 15, custo fixo de 5, custo variável de 1, e de motos com capacidade de 20, custo fixo de 100 e custo variável de 20;
- Início da janela de tempo atribuída aleatoriamente com valores inteiros entre 5 e 30 para cada cliente;
- Fim da janela de tempo igual ao início da janela somado com um valor inteiro aleatório entre 10 e 100;
- Tempo de serviço em cada cliente atribuído com o valor inteiro aleatório entre 3 e 7;
- Para o cálculo de $t_{ij}$ consideramos o próprio $t_{ij}$ se o veículo for uma bicicleta e consideramos $t_{ij}$ / 4 se o veículo for uma moto;
- No gráfico as bicicletas são plotadas com linha azul e as motos com linha vermelha.

Como nos exemplos anteriores e nos demais desse Capítulo, não estamos indicando unidades de medida nos experimentos. Numa aplicação prática poderemos ter quilômetros para as distâncias, minutos para tempo, custos em Reais. Para a realização dos experimentos a preocupação é a validação do modelo em si. Para o desenvolvimento de aplicação prática, os valores e unidades de medida deverão ser calibrados para uma melhor representação da realidade.

As Figuras 23 e 24 ilustram duas soluções para esse problema. Na Figura 23 vemos 3 rotas de bicicletas. Numa rota a bicicleta sai do depósito, vai para o cliente 1 e retorna para o depósito. Pode-se verificar que as duas outras rotas de bicicletas já estão no limite da capacidade, o que exige essa rota atendendo um único cliente. Também pode-se observar uma rota de motocicleta que se mostrou necessária para justamente atender os 3 clientes (clientes 4, 7 e 8) nas alturas mais elevadas.

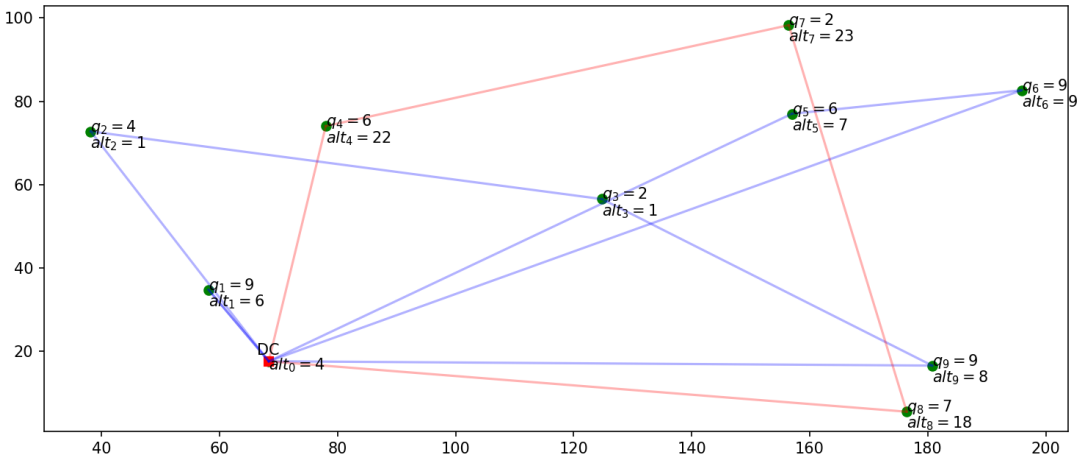



Figura 23 - Solução ótima com 3 bicicletas e 1 motocicleta para 9 clientes.

Já na Figura 24 vemos uma solução empregando 3 motocicletas e uma bicicleta. As 3 rotas de motocicletas atendem os clientes em elevações maiores, onde o ângulo de subida penaliza demais o uso de bicicletas para atendê-los. Como resultado, um número maior de motocicletas se mostra necessário para a solução ótima dessa configuração de clientes.

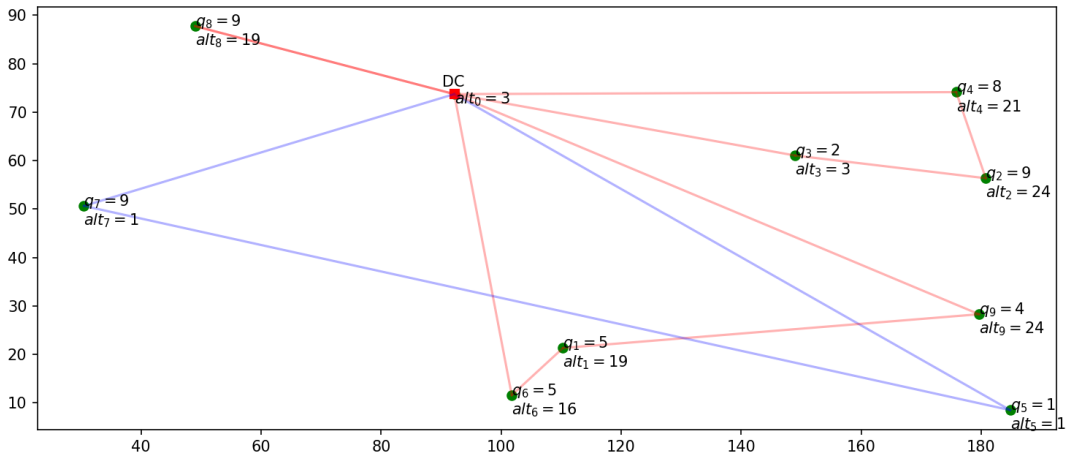



Figura 24 - Solução ótima com 1 bicicleta e 3 motocicletas para 9 clientes.

## 4.2 Modelos implementados com meta-heurísticas

### 4.2.1 Análise com base na instância A-n32-k5

A Tabela 1 apresenta uma compilação de resultados da execução dos algoritmos discutidos no Capítulo de Metodologia para o problema do roteamento de veículos com a restrição de capacidade apenas. São apresentados os resultados das

implementações de busca tabu, algoritmo genético, colônia de formigas e o algoritmo genético aprimorado com o procedimento de *Split* e operações especiais de *crossover*. Para uma comparação de resultados, utilizamos a instância A-n32-k5 disponível em http://vrp.galgos.inf.puc-rio.br/index.php/en/ e bem empregada no estudo do CVRP. Essa instância possui 31 clientes mais o depósito, a solução ótima vale 784 e é obtida com 5 veículos. A instância na Internet contém as coordenadas *(x,y)* de todos os pontos e as distâncias são calculadas pelo método Euclidiano. De posse do conhecimento da solução ótima dessa instância, podemos comparar os resultados de cada uma das implementações das meta-heurísticas com esse padrão para avaliar sua eficiência. Cada uma dessas implementações foi executada 10 vezes. Com os resultados dessas 10 execuções foi possível calcular o desvio da média bem como o desvio da melhor solução através das fórmulas abaixo.

$$Desvio\ médio = \frac{f(s^*) - \frac{1}{10}(\sum_{i=1}^{10} f(s_i)}{f(s^*)}$$

$$Desvio\ da\ melhor\ solução = GAP = \frac{f(s^*) - mín\{f(s_i): i = 1, \dots, 10\}}{f(s^*)}$$

Onde $f(s^*)$ é a solução ótima ou a melhor solução conhecida para a determinada instância.

Tabela 1 - Análise para A-n32-k5.

| Implementação | Melhor solução | Desvio médio de 10 soluções | Desvio da melhor solução |
|---|---|---|---|
| Busca Tabu | 784 | 3,84% | 0,00% |
| Algoritmo Genético | 799 | 9,69% | 1,91% |
| Colônia de Formigas | 832 | 8,16% | 6,12% |
| Algoritmo Genético Aprimorado | 784 | 0,61% | 0,00% |

O algoritmo de busca tabu rodou com 10.000 iterações e sua execução durou por cerca de 300 segundos. Importante destacar que em uma das execuções o algoritmo chegou na solução ótima 784 em apenas 42 segundos, atestando uma boa eficiência dessa implementação da busca tabu. Já o algoritmo genético simples rodou com uma população de 200 indivíduos e 5.000 gerações, e tempo de cerca de 420 segundos. O algoritmo de colônia de formigas durou cerca de 350 segundos, enquanto que o algoritmo genético aprimorado redou por 1.000 gerações levando cerca de 340 segundos. Assim, as quatro implementações puderam ser comparadas

com tempos de execução semelhantes. Considerando os resultados de todas as execuções, o algoritmo genético aprimorado apresentou a melhor performance, tanto no desvio médio das 10 execuções como também em chegar na solução ótima. Durante as 10 execuções, o algoritmo de busca tabu chegou na solução ótima de 784 apenas uma única vez, enquanto o algoritmo genético aprimorado chegou na solução ótima em 4 das 10 execuções. Na ocasião em que o algoritmo genético aprimorado chegou mais rapidamente na solução ótima o tempo foi de apenas 62 segundos. Isso comprovou o que foi apresentado por Prins (2004), que o algoritmo genético básico não tem uma performance tão boa quanto o algoritmo de busca tabu, mas que se for devidamente aprimorado então poderá superá-lo em performance.

A Figura 25 mostra a solução ótima para a instância A-n32-k5 encontrada pelo algoritmo genético aprimorado.

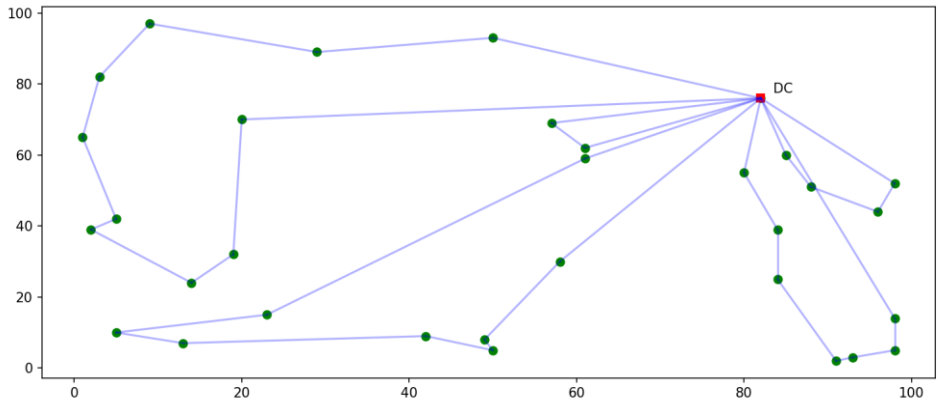



Figura 25 - Solução do algoritmo genético aprimorado para A-n32-k5.

Os resultados obtidos com as meta-heurísticas comprovam o que foi falado anteriormente. Um método heurístico pode ser capaz de entregar em segundos uma solução boa, bem próxima da solução ótima que talvez leve muitas horas, ou até mesmo dias, para ser obtida através dos métodos exatos. Para muitas situações práticas, é muito mais importante ter uma solução boa e rápida do que esperar por horas por uma solução que não chega.

### 4.2.2 Experimentos do algoritmo genético aprimorado para A-n80-k10

O algoritmo genético aprimorado foi executado para a instância A-n80-k10, também disponível em http://vrp.galgos.inf.puc-rio.br/index.php/en/ para ser usada em problemas de roteamento com a restrição de capacidade de veículos (CVRP), que possui 79 clientes em adição ao depósito e apresenta solução ótima de 1.763 com 10 veículos. A performance encontrada pode ser considerada de boa qualidade. A tabela abaixo apresenta o resultado de 10 execuções onde o melhor resultado possui um desvio de apenas 4,42% do valor ótimo conhecido para essa instância e que também foi obtido com a utilização de 10 veículos. A Tabela 2 apresenta os resultados obtidos pela implementação e a Figura 26 mostra a melhor solução encontrada durante os testes para essa instância.

Tabela 2 - Análise para A-n80-k10.

| Implementação | Melhor solução | Desvio médio de 10 soluções | Desvio da melhor solução |
|---|---|---|---|
| Algoritmo Genético Aprimorado | 1.841 | 9,09% | 4,42% |

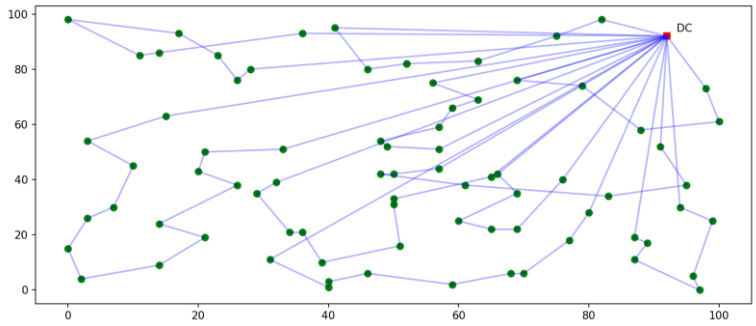



Figura 26 - Melhor solução encontrada para a instância A-n80-k10.

### 4.2.3 Experimentos com R101 com 25 clientes e janelas de tempo

A modificação do algoritmo *Split* para o problema com janelas de tempo foi testada utilizando a instância R101 de Solomon com 25 clientes. Essa instância, disponível em http://web.cba.neu.edu/~msolomon/problems.htm é própria para problemas de roteamento com janelas de tempo. Ela fornece as coordenadas dos clientes *(x,y)*, a demanda de cada cliente, os instantes iniciais e finais das janelas de tempo, tempo de serviço e capacidade dos veículos. Para essa instância, a solução

ótima vale 617,1 e é obtida com 8 veículos. Essa instância apresenta uma característica interessante, todas as janelas de tempo são bem estreitas e com valor de 10, o que impõe uma restrição obrigando os veículos a serem subutilizados em termos de capacidade. O algoritmo foi executado por 500 gerações e cada uma com 200 indivíduos. A Tabela 3 apresenta os resultados da implementação e a Figura 27 mostra a melhor solução encontrada durante os testes para essa instância.

Tabela 3 - Análise para R101 com 25 clientes.

| Implementação | Melhor solução | Desvio médio de 10 soluções | Desvio da melhor solução |
|---|---|---|---|
| Algoritmo Genético Aprimorado | 625 | 2,37% | 1,30% |

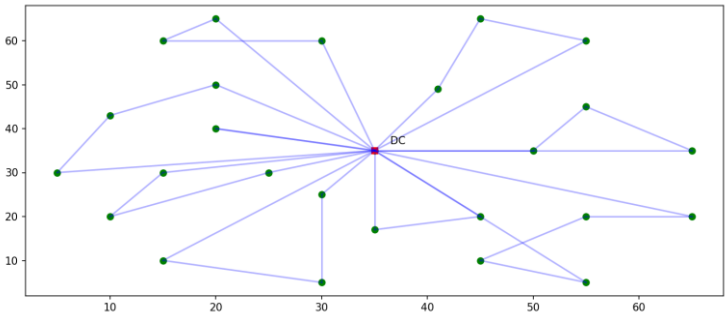



Figura 27 - Solução do algoritmo genético para R101 com 25 clientes.

### 4.2.4 Experimentos com R101 com 100 clientes e janelas de tempo

A modificação do algoritmo *Split* para o problema com janelas de tempo também foi testada utilizando a instância R101 de Solomon com 100 clientes (disponível em http://web.cba.neu.edu/~msolomon/r101.htm). Essa instância apresenta solução ótima de 1.651 com 19 veículos. Essa instância mantém a característica de todas as janelas de tempo bem estreitas e com valor de 10, na mesma forma como a instância anterior de apenas 25 clientes. O algoritmo foi executado por 500 gerações e cada uma com 200 indivíduos. A melhor solução encontrada pelo algoritmo genético foi 1.986, com um desvio de 20,3% da solução ótima conhecida. A Figura 28 mostra a melhor solução encontrada durante os testes para essa instância. Logicamente os resultados podem melhorar por se aumentar o número de indivíduos da população e o número de gerações. Como se trata de uma instância de maior tamanho, a convergência também leva mais tempo.

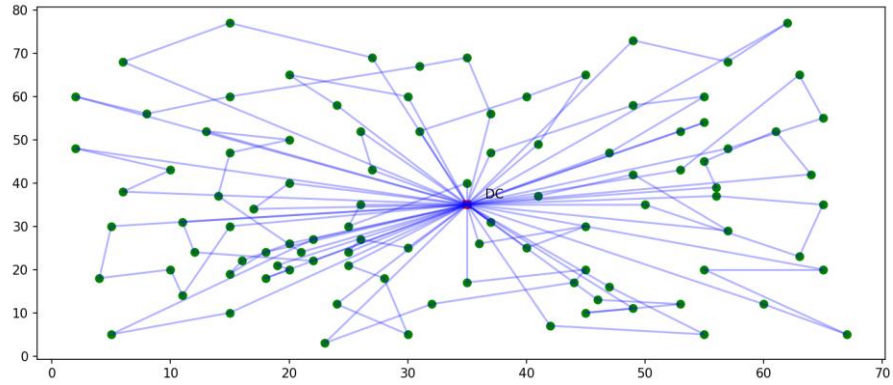



Figura 28 - Solução do algoritmo genético para R101 com 100 clientes.

### 4.2.5 Experimentos com VFMP-F 3 com 20 clientes e frota mista

Da mesma forma como o algoritmo *Split* foi modificado para o problema com janelas de tempo, ele também foi modificado conforme apresentado na Seção 3.2.4.5 e testado com a instância VFMP-F número 3 de Golden (disponível em https://perso.isima.fr/~lacomme/hvrp/hvrp.html). Essa instância apresenta solução ótima de 961 para uma configuração de 20 clientes além do depósito, uma frota mista de 5 tipos de veículos onde cada tipo de veículo tem uma capacidade e um custo fixo específicos. Não há restrição de quantidade para cada tipo de veículo.

A Figura 29 mostra o resultado ótimo de 961 obtido através da implementação do algoritmo genético aprimorado para essa instância específica. Cada tipo de veículo é identificado por uma cor. Conforme observa-se na Figura 29, a solução ótima é obtida com um total de 6 veículos.

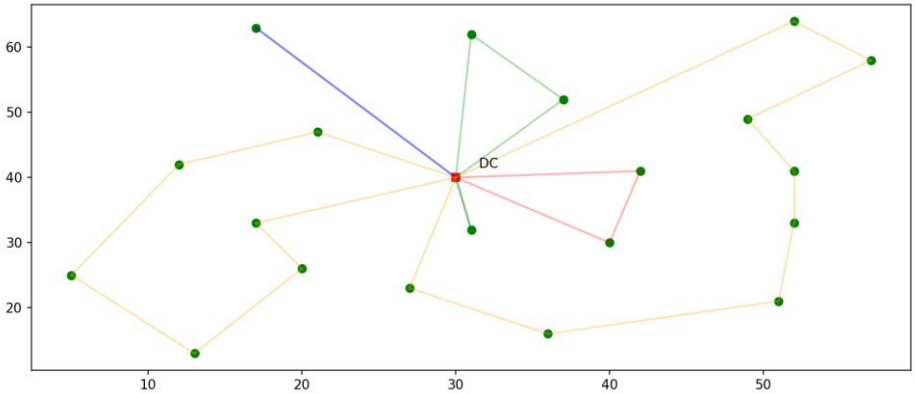



Figura 29 - Solução ótima para a instância 3 de Golden com frota mista.

### 4.2.6 Algoritmo genético aprimorado incorporando bicicletas

O algoritmo genético aprimorado pode ser adaptado para bicicletas com o modelo de penalidades para subidas íngremes conforme proposto nesse trabalho. Considerando o problema de restrição de capacidade apenas, acrescentamos no programa em Python: (1) o vetor com os dados de elevação para cada um dos clientes; (2) cálculo da matriz de ângulos entre o cliente *i* e o cliente *j*; (3) cálculo da matriz de penalidades; e (4) atualização da matriz $c_{ij}$ onde cada elemento dela é acrescentado do valor da penalidade $p_{ij}$. Todo o restante do algoritmo da meta-heurística é exatamente o mesmo. O algoritmo foi executado para a instância P-n19-k2 com as elevações usadas na Seção 4.1.2 e conseguiu chegar no resultado ótimo que havia sido apresentado nessa Seção. A tela de solução é apresentada na Figura 30. Importante ressaltar que a meta-heurística nesse caso entregou a solução ótima, mas sem poder garantir que ela era ótima. Aliás, nem todas as execuções da meta-heurística entregaram essa solução ótima. A garantia que essa era a solução ótima veio do emprego do *solver* na Seção 4.1.2.

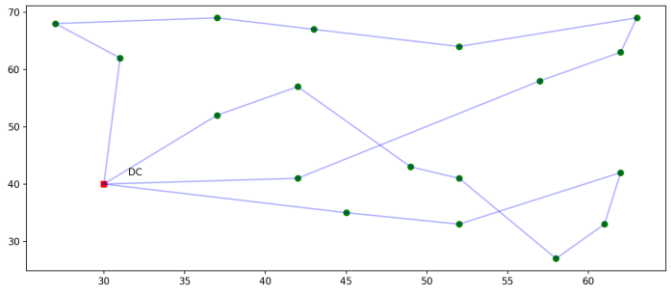



Figura 30 - Solução do problema P-n19-k2 com bicicletas e subidas.

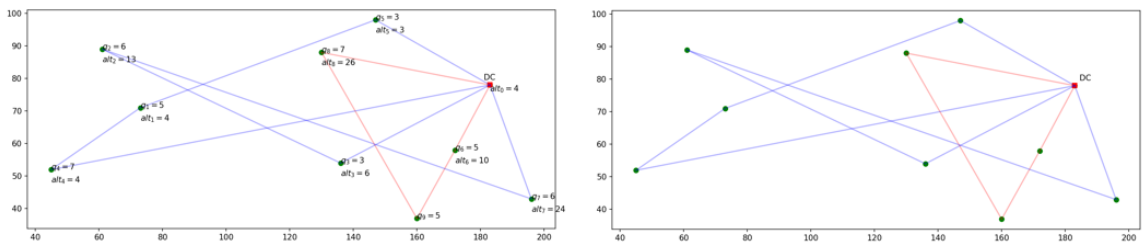
## 4.3 Experimentos com o algoritmo genético para o problema de uma frota mista de bicicletas e motocicletas com janelas de tempo e penalidades de subida

Nessa Seção temos os experimentos conduzidos com a implementação do algoritmo genético para o problema completo conforme a modificação do algoritmo *Split* apresentado na Seção 3.2.5. Na falta de instâncias para esse problema específico, utilizamos o programa da Seção 4.1.4 para gerar dados aleatórios e calcular a solução ótima. Esses dados são usados pelo algoritmo genético e a sua solução é comparada com a solução ótima.

Os seguintes parâmetros foram usados para os experimentos de ambos os programas:

- Frota mista e não limitada de bicicletas e motocicletas;
- Custos fixos: 5 para bicicleta e 100 para motocicleta;
- Custos variáveis: 1 para bicicleta e 20 para motocicleta;
- Capacidade: 15 para bicicleta e 20 para motocicleta;
- Penalização para bicicletas:
    - 0 para ângulos inferiores a 2 graus
    - 30% para ângulos entre 2 e 4 graus
    - 70% para ângulos entre 4 e 6 graus
    - Impossível para ângulos superiores a 6 graus;
- Sem penalização para motocicletas;
- Matriz de tempos de deslocamento $t_{ij}$ obtida a partir da matriz de distâncias $c_{ij}$ dividindo-a por 10;
- Motocicletas são 4 vezes mais rápidas que bicicletas, com isso o tempo $t_{ij}$ entre o cliente *i* e o cliente *j* é dividido por 4 para motocicletas;
- Cada cliente possui uma demanda, uma janela de tempo e um tempo de serviço que lhe são específicos, bem como coordenadas de localização e uma altura.

Na Figura 31a o problema foi executado para uma instância de 9 clientes através do PuLP que encontrou a solução ótima de 4.095,6 em aproximadamente 73 segundos. Na Figura (b), o mesmo problema foi executado com o algoritmo genético que encontrou a mesma solução 4.095,6 em apenas 4,8 segundos. As linhas em azul representam rotas de bicicletas e a linha em vermelho uma rota de motocicleta, dessa forma a solução ótima possui duas bicicletas e uma motocicleta. A Tabela 4 contém os dados para esse problema.




(a) (b)

Figura 31 - Problema com 9 clientes.

Tabela 4 - Dados da instância de 9 clientes da Figura 31.

| Demanda | 0 | 5 | 6 | 3 | 7 | 3 | 5 | 6 | 7 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|
| Coordenadas X | 183 | 73 | 61 | 136 | 45 | 147 | 172 | 196 | 130 | 160 |
| Coordenadas Y | 78 | 71 | 89 | 54 | 52 | 98 | 58 | 43 | 88 | 37 |
| Altura | 4 | 4 | 13 | 6 | 4 | 3 | 10 | 24 | 26 | 21 |
| Início da janela | 0 | 10 | 13 | 10 | 12 | 24 | 5 | 13 | 10 | 27 |
| Fim da janela | 9999 | 96 | 27 | 83 | 105 | 46 | 75 | 51 | 65 | 91 |
| Tempo serviço | 0 | 5 | 3 | 6 | 3 | 6 | 5 | 3 | 5 | 6 |

Na Figura 32 (a) o problema foi executado com o PuLP para 10 clientes e o valor ótimo de 8.985,5 foi encontrado em 185 segundos. Na Figura (b) o algoritmo genético foi capaz de encontrar a mesma solução em 2,6 segundos. Vemos que a solução foi encontrada com duas rotas de bicicletas e duas rotas de motocicletas. Podemos perceber que ao passarmos de 9 para 10 clientes o tempo de execução mais que dobrou para a solução através de métodos exatos, mas a rapidez da meta-heurística chegar na mesma solução foi mantida. A Tabela 5 contém os dados para esse problema.

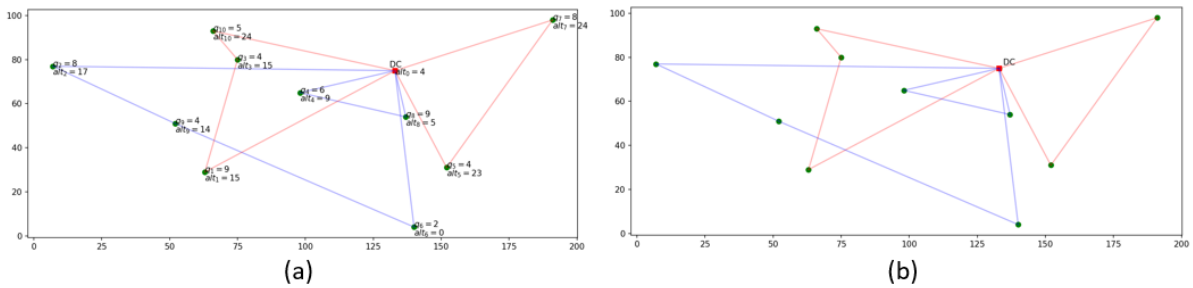



Figura 32 - Problema com 10 clientes.

Tabela 5 - Dados da instância de 10 clientes da Figura 32.

| Demanda | 0 | 9 | 8 | 4 | 6 | 4 | 2 | 8 | 9 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordenadas X | 133 | 63 | 7 | 75 | 98 | 152 | 140 | 191 | 137 | 52 | 66 |
| Coordenadas Y | 75 | 29 | 77 | 80 | 65 | 31 | 4 | 98 | 54 | 51 | 93 |
| Altura | 4 | 15 | 17 | 15 | 9 | 23 | 0 | 24 | 5 | 14 | 24 |
| Início janela | 0 | 12 | 22 | 24 | 7 | 22 | 24 | 11 | 25 | 22 | 28 |
| Fim janela | 9999 | 73 | 111 | 85 | 55 | 36 | 123 | 27 | 59 | 108 | 106 |
| Tempo serviço | 0 | 4 | 6 | 5 | 5 | 6 | 4 | 3 | 3 | 3 | 5 |

Já na Figura 33, o problema foi analisado considerando-se 13 clientes. A Figura 33 (a) mostra a solução através do Pulp que obteve o resultado de 3.348,83 em 740 segundos, enquanto que a Figura 33 (b) mostra a mesma solução obtida através do algoritmo genético em 57,8 segundos em uma execução e em 23.2 segundos em outra execução. A solução é composta de uma rota com motocicleta e quatro rotas com bicicletas. Percebe-se o aumento do tempo de execução para o método exato e a contrapartida de uma solução rápida e boa oferecida pelo algoritmo genético. A Tabela 6 contém os dados para esse problema.

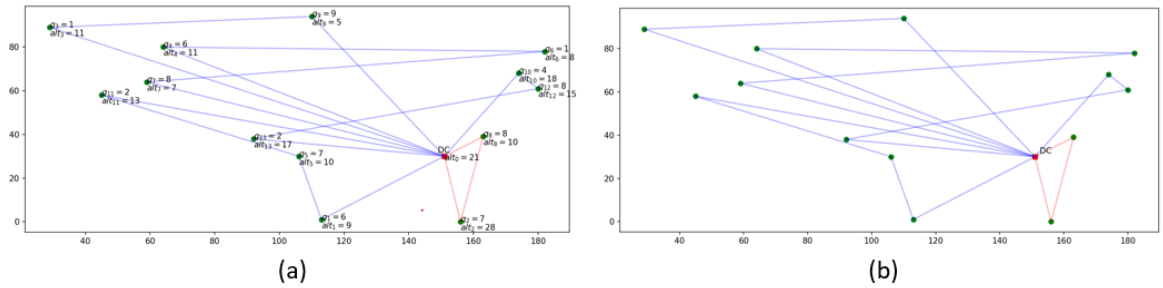



Figura 33 - Problema com 13 clientes.

Tabela 6 - Dados da instância de 13 clientes da Figura 33.

| Demanda | 0 | 6 | 7 | 1 | 6 | 7 | 1 | 8 | 8 | 9 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Coordenadas X | 151 | 113 | 156 | 29 | 64 | 106 | 182 | 59 | 163 | 110 | 174 |
| Coordenadas Y | 30 | 1 | 0 | 89 | 80 | 30 | 78 | 64 | 39 | 94 | 68 |
| Altura | 21 | 9 | 28 | 11 | 11 | 10 | 8 | 7 | 10 | 5 | 18 |
| Início janela | 0 | 10 | 21 | 10 | 20 | 23 | 21 | 12 | 22 | 22 | 5 |
| Fim janela | 9999 | 47 | 77 | 37 | 74 | 88 | 50 | 23 | 71 | 41 | 15 |
| Tempo serviço | 0 | 6 | 6 | 3 | 4 | 4 | 6 | 4 | 4 | 3 | 3 |

Continuação da Tabela 6

| Demanda | 2 | 8 | 2 |
|---|---|---|---|
| Coordenadas X | 45 | 180 | 92 |
| Coordenadas Y | 58 | 61 | 38 |
| Altura | 13 | 15 | 17 |
| Início janela | 14 | 27 | 6 |
| Fim janela | 43 | 43 | 99 |
| Tempo serviço | 4 | 6 | 6 |

# CAPÍTULO 5 RESULTADOS E DISCUSSÕES

A implementação dos modelos matemáticos em Python utilizando o PuLP bem como o Pyomo apresentou resultados satisfatórios, levando às soluções ótimas. Foram implementados os modelos com restrição de capacidade, janelas de tempo e frota heterogênea. O modelo de penalidades relacionadas aos ângulos de subida para o problema de bicicletas também se mostrou satisfatório. Esse modelo foi implementado para o problema com restrição de capacidades e para o modelo com as três variantes juntas: capacidade, janelas de tempo e frota heterogênea. Por outro lado, na medida em que se aumenta o número de clientes, percebe-se que o tempo de execução dos programas também aumenta consideravelmente. Ou seja, com um aumento pequeno do número de clientes, o tempo de execução já se torna bastante longo. Isso comprova que os métodos exatos, embora tenham a vantagem de conduzir a uma solução ótima e garantir que ela seja realmente ótima, têm o inconveniente de serem lentos na medida em que o tamanho do problema cresce. Para problemas de larga escala, torna-se totalmente impraticável a utilização desses métodos.

Embora já sabido que soluções heurísticas e meta-heurísticas são mais adequadas para problemas de tamanho maior, o que se aproxima de muitos problemas reais que são encontrados na prática, é importante ressaltar a vantagem de se fazer o modelamento matemático e até buscar sua resolução para instâncias menores antes de se partir para as soluções heurísticas. O modelamento matemático permite uma melhor compreensão do problema e de suas restrições, compreensão essa que pode se mostrar importante no desenvolvimento de métodos heurísticos. Ademais, uma implementação eficiente de uma meta-heurística exige um bom entendimento não apenas do algoritmo básico da meta-heurística, mas também do problema em questão. Certamente o algoritmo deverá fazer movimentos buscando tanto a intensificação bem como a diversificação, além dos testes de viabilidade para garantir que as restrições não sejam violadas. Muito da eficiência virá da realização de uma boa gama de movimentos e nesse ponto o conhecimento matemático do problema é vantajoso. Por isso que esse trabalho começou com o modelamento matemático para depois avançar para os métodos heurísticos.

A implementação das meta-heurísticas de busca tabu, algoritmo genético e colônia de formigas para o problema básico apenas com restrição de capacidade

mostrou resultados superiores para a busca tabu, sendo seguido da colônia de formigas e, por último, do algoritmo genético. Embora o algoritmo genético tenha entregado uma solução melhor que a colônia de formigas, mas analisando todo o conjunto de soluções obtidas, o desvio médio da solução ótima foi melhor para a colônia de formigas. Não é difícil explicar esses resultados. O algoritmo de busca tabu é bastante poderoso porque executa uma série de movimentos para a exploração da vizinhança e onde esses movimentos são executados visando a melhoria da solução. O algoritmo de colônia de formigas também tem uma lógica de privilegiar a escolha de cidades mais próximas para o próximo movimento. Por outro lado, o algoritmo genético tem um caráter predominantemente aleatório, privilegiando muito mais a diversificação do que a intensificação da solução. Consequentemente, um algoritmo genético no seu conceito puro tende a não ter uma performance tão boa quanto os demais algoritmos. Fica evidente que para uma melhor performance, o algoritmo genético precisa incorporar algum mecanismo de busca local ou outra técnica que permita a execução de movimentos que privilegiem a intensificação além da diversificação das soluções.

A implementação do algoritmo genético para o problema básico usando delimitadores de rota na codificação do cromossomo adicionou uma complexidade ao código e estrutura do programa. As operações para se gerar novos indivíduos (soluções) e testar sua viabilidade diante da restrição única de não estourar a capacidade do veículo se mostraram complexas na escrita do código de programa.

A solução proposta por Prins (2004) se mostrou muito eficiente para melhorar o desempenho do algoritmo genético para o problema de roteamento de veículos. Ela elimina o uso de delimitadores no código do cromossomo, o que facilita sobremaneira a escrita do código de programa e as operações a serem feitas com os cromossomos. Além disso, o algoritmo *Split* garante que a decomposição do cromossomo em rotas seja ótima para aquela sequência. Isso por si só já garante uma intensificação para o algoritmo. Todas as operações de *crossover* e de mutação podem ser exploradas de uma maneira mais simples e sem a preocupação de violação da restrição do problema porque o algoritmo *Split* se encarrega de decompor o cromossomo em rotas sem que haja violação de restrição. A elegância do algoritmo *Split* é ainda mais notável quando vemos que ele foi originalmente concebido para o problema com restrição de capacidade apenas, mas pode ser expandido para o problema com restrição de janelas de tempo bem como o problema com frota híbrida.

Uma outra técnica para o aprimoramento do desempenho do algoritmo genético foi abordada em Blanton e Wainwright (1993). Conforme explicado no Capítulo da Revisão da Literatura, essa técnica realiza a operação de *crossover* não de forma puramente aleatória (conforme sua concepção original) mas ela busca produzir um cromossomo filho a partir de dois cromossomos pais aplicando uma análise de precedência na escolha de que gene passa do pai para o filho. Essa técnica foi elaborada com base no critério de precedência que pode ser construída a partir dos dados de entrada de um problema com janelas de tempo. O Capítulo de Metodologia propôs um método inspirado nessa técnica para o problema apenas com restrição de capacidade. Embora não haja um critério de precedência, estabeleceu-se uma regra com base nas distâncias dos próximos genes dos cromossomos pais em relação ao gene anterior do cromossomo filho. O ponto interessante nessas técnicas é que passamos a ter uma operação de *crossover* no algoritmo genético que passa a privilegiar a intensificação no caso do problema com restrição de capacidade, ou a viabilidade no caso do problema com janelas de tempo, ao invés da pura diversificação que ocorre nos operadores tradicionais de *crossover*. Para um problema de janelas de tempo, mesclar o operador tal como está em Blanton e Wainwright (1993) com o operador proposto nesse trabalho para o problema com restrição de capacidade é uma estratégia interessante.

Essa dissertação elaborou um algoritmo genético que faz uso do algoritmo *Split* para a codificação do cromossomo bem como para decompô-lo numa solução de rotas. Os operadores OX, 2OPT, bem como operadores baseados no critério de precedência de Blanton e Wainwright (1993) e baseados na menor distância para o gene seguinte no cromossomo compõem a estrutura do algoritmo genético. As variações do problema capacitado, com janelas de tempo, com frota mista e, por fim, o problema completo com todas essas variações foram implementados mantendo-se o algoritmo genético inalterado e modificando apenas o algoritmo *Split*.

O algoritmo genético aprimorado desenvolvido no Capítulo de Metodologia foi testado com as seguintes instâncias: A-n32-K5, A-n80-k10, R101 com 25 clientes, R101 com 100 clientes, número 3 de Golden com 20 clientes. Os resultados obtidos comprovam a eficiência do algoritmo.

Como etapa final o algoritmo *Split* foi expandido para considerar o problema de frota mista de bicicletas e motocicletas com a penalidade de subidas íngremes. Na falta de instâncias com soluções ótimas para *benchmark*, essas instâncias foram

geradas aleatoriamente e executadas com o PuLP para a obtenção das soluções ótimas. Na sequência, essas mesmas instâncias foram executadas com o algoritmo genético e as soluções comparadas. O algoritmo genético foi capaz de entregar as mesmas soluções ótimas num intervalo de tempo inferior ao *solver* do PuLP. Logicamente, sem a garantia de otimalidade.

Comparando todo o trabalho com os objetivos delineados na Seção 1.2 podemos dizer que os objetivos foram alcançados. Esse trabalho culminou com a entrega de um modelo que permite o planejamento de entregas utilizando bicicletas e que acomoda o esforço adicional do ciclista com subidas íngremes. Para alcançar esse objetivo, o trabalho analisou uma quantidade ampla de referências bibliográficas sobre a utilização de bicicletas como solução logística de entregas bem como do problema de roteamento de veículos, apresentou a modelagem matemática para o problema com algumas de suas variantes, propôs um modelo para considerar os ângulos de subida entre um cliente e outro na formação de rotas, implementou os modelos em Python para a resolução por meio de *solvers*, analisou soluções heurísticas (Clarke & Wright) e meta-heurísticas (busca tabu, algoritmo genético e colônia de formigas) para a geração de soluções, explorou técnicas de como aprimorar a eficiência de uma meta-heurística (algoritmo genético) e como desenvolvê-la para resolver o problema completo da frota mista de bicicletas e motocicletas.

# CAPÍTULO 6 CONCLUSÕES

Bicicletas podem ser uma solução alternativa para muitas entregas que acontecem dentro de grandes cidades. Quando comparadas aos veículos motorizados, como vans e caminhões, as bicicletas trazem vantagens para o transporte de mercadorias de pequeno volume dentro do ambiente de trânsito congestionado, horas de pico, dificuldade de estacionar, que são comuns em grandes cidades. As bicicletas custam menos, não utilizam combustível, não poluem o meio ambiente. Certamente elas não são uma solução para todas as situações de entregas mas têm seu espaço.

O problema de entregas por bicicletas é um problema de roteamento de veículos. O trabalho apresentou uma proposta de modelamento do problema por considerar penalidades em função do ângulo de subida para a rota entre um cliente e outro. Embora a rota possa ser irregular e a inclinação possa variar ao longo da mesma, esse trabalho assume uma aproximação linear que pode ser bem razoável para muitos cenários. Essa adaptação se mostrou simples e sua eficácia reside justamente na sua simplicidade. Para cenários onde as localidades são praticamente planas, o problema de roteamento tradicional já atende o emprego de bicicletas. Uma alternativa interessante seria o emprego de bicicletas elétricas, nesse caso o esforço com subidas é eliminado e retornamos ao problema de roteamento tradicional.

Conforme evidenciado na resolução dos problemas através do modelamento matemático e *solvers*, a obtenção de uma solução ótima começa a ficar impraticável na medida em que o tamanho do problema aumenta. Com isso, soluções heurísticas passam a ser necessárias. De fato, para muitos problemas do quotidiano é mais importante obter uma solução boa e rápida do que a ótima com muita espera. O trabalho focou no aprimoramento do algoritmo genético utilizando duas técnicas fundamentais: o algoritmo *Split* de Prins (2004) e o desenvolvimento de operadores de *crossover* que privilegiam a intensificação e a viabilidade de uma solução ao invés da pura diversificação aleatória de muitos operadores tradicionais. Com isso, bons resultados puderam ser alcançados num intervalo curto de tempo.

O algoritmo genético aprimorado, conforme concebido nessa dissertação, foi executado para o problema capacitado, com janelas de tempo, com frota mista e finalmente numa aplicação que incorporou todas essas variações mais a proposta de penalidades para subidas no caso de bicicletas. Em todos esses casos o algoritmo

genético foi exatamente o mesmo e apenas o algoritmo *Split* foi modificado para contemplar as variações. Podemos destacar as vantagens do algoritmo *Split:*

- Codificação de cromossomo sem delimitadores, o que simplifica sobremaneira o código de programa do algoritmo genético para a execução das operações sobre o cromossomo;
- Garante a otimalidade da sequência indicada no cromossomo;
- Permite modificações para outras variantes do problema;
- Padrão do cromossomo permite que o algoritmo genético seja exatamente o mesmo.

Como trabalho futuro podemos mencionar a expansão desse problema para outras variações do problema de roteamento, como por exemplo o caso de múltiplos depósitos que foi apresentado aqui no modelamento matemático, mas que não foi explorado nos métodos heurísticos. Um modelo que crie *clusters* de clientes próximos de depósitos, encontre soluções boas para cada um desses *clusters*, depois realize movimentos de trocas de clientes entre *clusters* diferentes pode ser uma abordagem para a busca de soluções.

## REFERÊNCIAS

AFSHAR-NADJAFI, B. e AFSHAR-NADJAFI, A. (2014). **A constructive heuristic for time-dependent multi-depot vehicle routing problem with time-windows and heterogeneous fleet**. Journal of King Saud University - Engineering Sciences.

ALTINEL, I. e ÖNCAN, T. (2005). **A new enhancement of the Clarke and Wright savings heuristic for the capacitated vehicle routing problem**. Journal of the Operational Research Society 56: p.954-961.

BALDACCI, R., BATTARRA, M., e VIGO, D. (2008) **Routing a heterogeneous fleet of vehicles, in The Vehicle Routing Problem: Latest Advances and New Challenges**, Golden, B., Raghavan, S. e Wasil, E., eds., vol. 43 of Operations Research/Computer Science Interfaces Series, Springer, New York, 2008, pp. 3–27

BATTARA, M., GOLDEN, B. e VIGO, D. (2008). **Tuning a parametric Clarke–Wright heuristic via a genetic algorithm**. Journal Operational Research Society 59(11), p.1568–1572

BLANTON, J. e WAINWRIGHT, R. (1993). **Multiple vehicle routing with time and capacity constraints using genetic algorithms**. Proceedings of the 5th International Conference on Genetic Algorithms, p. 452-459.

BULLNHEIMER, B., HARTL, R. e STRAUSS, C. (1997). **Applying the ant system to the vehicle routing problem**. Proceedings of Second Metaheuristics International Conference, MIC'97, Sophia-Antipolis, France.

CHANG, Y. e CHEN, L. (2007). **Solve the vehicle routing problem with time windows via a genetic algorithm**. Discrete and continuous dynamical systems supplement, 2007, pp. 240-249.

CLARKE, G. e WRIGHT, J. (1964). **Scheduling of vehicles from a central depot to a number of delivery points**. Operations Research, v.12, p. 568-581.

CORDEAU, J., DESAULNIERS, G., DESROSIERS, J., SOLOMON, M. e SOUMIS, F. (2002). VRP with time windows, In **The vehicle routing problem**, Toth, P. e Vigo, D., SIAM.

COSTA, T., SOUZA, S. e SOUZA, M. (2005). **Um algoritmo baseado em grasp e busca tabu aplicado à resolução do problema de roteamento de veículos com janela de tempo**. SPOLM.

DANTZIG, G. e RAMSER, J. (1959). **The truck dispatching problem**. Management Science 6, p.81-91.

DE DECKER, K. (2012). Cargo cyclists replace truck drivers on European city streets. **Low-Tech Magazine**. Web page. https://www.lowtechmagazine.com/2012/09/jobs-of-the-future-cargo-cyclist.html . Acesso: 22/05/2022.

DORIGO, M., MANIEZZO, V. e COLORNI, A. (1991). **A positive feedback as a search strategy**. Milão: Dipartimento di Elettronica e Informatica. Politecnico di Milano, Itália, Relatório Técnico.

EL-SHERBENY, N. (2010). **Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods**. Journal of King Saud University 22, p.123-131.

FERGUNSON, J. (2012). **Cargo Bike Crazy: The Potential of Delivering Goods by Bike**. ECF – European Cyclists' Federation. Página na Internet. https://ecf.com/news-and-events/news/cargo-bike-crazy-potential-delivering-goods-bike-0 . Acesso: 22/05/2022.

GAMBARDELLA, L., TAILLARD, É. e AGAZZO, G. (1999). MACS-VRPTW: A multiple ant colony system for vehicle routing problem with time windows. In: **New Ideas in Optimization**. Londres, McGraw-Hill, p.63-76

GENDREAU, M., HERTZ, A. e LAPORTE, G. (1994). **A tabu search heuristic for the vehicle routing problem**. Management Science, V.40, N.10, p. 1276-1290.

GLOVER, F. (1986). **Future paths for integer programming and links to artificial intelligence**. Computer & Operations Research, v.5, p. 549-553.

GOEL, R. e MAINI, R. (2017). **Vehicle routing problem and its solution methodologies: a survey**. Int. J. Logistics Systems and Management, Vol.28, No.4, p.419-435.

GOLDBARG, M., GOLDBARG, E. e LUNA, H. (2016). **Otimização Combinatória e Meta-Heurísticas: Algoritmos e Aplicações**. Editora LTC, p.277-323.

GRUBER, J. e NARAYANAN, S. (2019). **Travel Time Differences between Cargo Cycles and Cars in Commercial Transport Operations**. Transportation Research Record, 2673, p.623–637.

HANUM, F., HARTONO, A. e BAKHTIAR, T. (2018). **On the multiple depots vehicle routing problem with heterogeneous fleet capacity and velocity**. IOP Conf. Series: Materials Science and Engineering 332 012052.

HOLLAND, J. (1975). **Adaptation in natural and artificial systems**. Ann Arbor, USA: University of Michigan Press, 1975.

KUMAR, S. e PANNEERSELVAM, R. (2015). **A time-dependent vehicle routing problem with time windows for e-commerce supplier site pickups using genetic algorithm**. Intelligent Information Management 7, p.181-194.

LAPORTE, G. (1992). **The vehicle routing problem: An overview of exact and approximate algorithms**. European Journal of Operational Research 59, p. 77-85.

LAPORTE, G. (2007). **What you should know about the vehicle routing problem**. Naval Research Logistics, Vol. 54, No. 8, p.811-819.

LAPORTE, G., TOTH, P. e VIGO, D. (2013). **Vehicle routing: historical perspective and recent contributions**. The EURO Journal on Transportation and Logistics.

LIONG, C., WAN ROSMANIRA, I., HAIRUDDIN, O. e ZIROUR, M. (2008). **Vehicle routing problem: Models and solutions**. Journal of Quality Measurement and Analysis 4(1), p.205-218.

LIU, S., HUANG, W. e MA, H. (2009). **An effective genetic algorithm for the fleet size and mix vehicle routing problems**. Transportation Research Part E 45, p. 434-445.

MASSINK, R., ZUIDGEEST, M., RIJNSBURGER, J., SARMIENTO, O. e VAN MAARSEVEEN, M. (2011). **The Climate Value of Cycling**. Natural Resources Forum. A United Nations Sustainable Development Journal 35(2), p.100-111.

MIRSHEKARIAN, S., ÇELIKBILEK, C., e SÜER, G. (2015). **A Heuristic for the Vehicle Routing Problem with Tight Time Windows and Limited Working Times**. Proceedings of the 26th Annual Production and Operations Management Society Conference.

NAUMOV, V. e PAWLUS, M. (2021). **Identifying the Optimal Packing and Routing to Improve Last-Mile Delivery Using Cargo Bicycles**. Energies 14, 4132. https://doi.org/10.3390/en14144132 .

NÉIA, S., ARTERO, A., CANTÃO, L. e CUNHA, C. (2013). Roteamento de veículos utilizando otimização por colônia de formigas e algoritmo genético. **Meta-Heurísticas em Pesquisa Operacional**. Cap.14, p.219-236. Editora Lopes.

OMBUKI, B., NAKAMURA, M. e OSAMU, M. (2002). **A hybrid search based on genetic algorithms and tabu search for vehicle routing**. 6th IASTED International Conference on Artificial Intelligence and Soft Computing, p. 176-181.

OMBUKI-BERMAN, B. e HANSHAR, F. (2009). Using genetic algorithms for multi-depot vehicle routing. In: Pereira, F. e Tavares, J. (Eds). **Bio-inspired algorithms for the vehicle routing problem**. Studies in Computational Intelligence (Vol. 161, p. 77-99). Springer, Berlin, Heidelberg.

PRINS, C. (2004). **A simple and effective evolutionary algorithm for the vehicle routing problem**. Computers & Operations Research, 31, p. 1985-2002.

PRINS, C. (2009). **Two memetic algorithms for heterogeneous fleet vehicle routing problems**. Engineering Applications of Artificial Intelligence 22 (2009), p. 916-928.

RANIERI, L., DIGIESI, S., SILVESTRI, B. e ROCCOTELLI, M. (2018). **A Review of Last Mile Logistics Innovations in an Externalities Cost Reduction Vision**. Sustainability, 10(3): 782. https://doi.org/10.3390/su10030782 .

RYBICKOVÁ, A., BRODSKÝ, J., KARÁSKOVÁ, A. e MOCKOVÁ, D. (2015). **A genetic algorithm for the multi-depot vehicle routing problem**. Applied Mechanics and Materials, October 2015.

SANTOS, R. e LEAL, J. (2007). **Solução de um problema de roteirização com janelas de tempo através de um algoritmo de múltiplas colônias de formigas**. Transportes, volume XV, n.2, dezembro de 2007.

SHETH, M., BUTRINA, P., GOODCHILD, A. e MCCORMACK, E. (2019). **Measuring delivery route cost trade-offs between electric-assist cargo bicycles and delivery trucks in dense urban areas**. European Transport Research Review 11,11. https://doi.org/10.1186/s12544-019-0349-5 .

SIMAS, E. e GÓMEZ, A. (2005). **Uma solução para o problema de roteamento de veículos através da pesquisa tabu**. SBPO. XXXVII Simpósio Brasileiro de Pesquisa Operacional.

TAN, K., LEE, L., ZHU, Q. e OU, K. (2001). **Heuristic methods for vehicle routing problem with time windows**. Artificial Intelligence in Engineering, v15, p. 281-295.

TOTH, P. e VIGO, D. (2002). An overview of vehicle routing problems, In **The vehicle routing problem**, Toth, P. e Vigo, D., SIAM.

VASIUTINA, H., SZARATA, A. e RYBICKI, S. (2021). Evaluating the Environmental Impact of Using Cargo Bikes in **Cities: A Comprehensive Review of Existing Approaches**. Energies 14, 6462. https://doi.org/10.3390/en14206462 .

**APÊNDICE A – Programa em Python para solução com PuLP**

```
import numpy as np
import matplotlib.pyplot as plt
from pulp import *
import math

# Lista de n clientes
# Nó 0 representa a origem
n = 10
clientes = [x for x in range(1,n+1)]
nodes = [0] + clientes

# Variável para número de veículos
Motos = 4
Bikes = 4
Frota = Motos + Bikes
veiculos = [x for x in range(1,Frota+1)]

rnd = np.random

# Lista q gerada com a demanda aleatória para cada cliente
q = [0,9,8,4,6,4,2,8,9,4,5]

# Capacidade e custo fixo e variável de moto e bicicleta
Qmoto=20
Qbike=15
Fmoto=100
Fbike=5
Vmoto=20
Vbike=1

# Coordenadas aleatórias

loc_x = [133,63,7,75,98,152,140,191,137,52,66]
loc_y = [75,29,77,80,65,31,4,98,54,51,93]

# Lista alt de alturas geradas aleatóriamente para cada cliente
alt =  [4,15,17,15,9,23,0,24,5,14,24]

plt.figure(figsize=(12,5))
plt.scatter(loc_x, loc_y, color="green")
for i in clientes:
        plt.annotate('$q_{%d}=%d$'%(i,q[i]),(loc_x[i],loc_y[i]))
        plt.annotate('$alt_{%d}=%d$'%(i,alt[i]),(loc_x[i],loc_y[i]-4))
plt.plot(loc_x[0],loc_y[0], color='red',marker='s')
plt.annotate('DC',(loc_x[0]-2, loc_y[0]+1.5))
plt.show()

arcos = {(i,j) for i in nodes for j in nodes if i!=j}
```

```
# Cálculo da matriz de distância entre dois nós
distancia={(k,i,j): np.hypot(loc_x[i]- loc_x[j],loc_y[i]-loc_y[j])
                    for k in veiculos for i in nodes for j in nodes if i!=j}

# Cálculo da matriz g
matrizg={(k,i,j):0 for k in veiculos for i in nodes for j in nodes if i!=j}

for k in veiculos:
    if k<=Motos:
        for (i,j) in arcos:
            matrizg[(k,i,j)]=distancia[(k,i,j)]*Vmoto
    else:
        for (i,j) in arcos:
            matrizg[(k,i,j)]=distancia[(k,i,j)]*Vbike


# Cálculo do ângulo de elevação entre dois nós
angulo_rad = {(i,j):np.arctan((alt[j]-alt[i])/distancia[(k,i,j)]) for k in veiculos for i in nodes
for j in nodes if i!=j}
angulo_graus = {(i,j):np.degrees(angulo_rad[(i,j)]) for i in nodes for j in nodes if i!=j}

# Cálculo da matriz de penalidades
penalidade={(k,i,j):0 for k in veiculos for i in nodes for j in nodes if i!=j}

for k in veiculos:
    if k<=Motos:
        for i in nodes:
            for j in nodes:
                if i!=j:
                    penalidade[(k,i,j)]=0
    else:
        for i in nodes:
            for j in nodes:
                if i!=j:
                    if angulo_graus[(i,j)] <= 2:
                        penalidade[(k,i,j)] = 0
                    elif angulo_graus[(i,j)] <=4:
                        penalidade[(k,i,j)] = distancia[(k,i,j)]*0.3
                    elif angulo_graus[(i,j)] <=6:
                        penalidade[(k,i,j)] = distancia[(k,i,j)]*0.7
                    else:
                        penalidade[(k,i,j)] = distancia[(k,i,j)]*50000


# Definições de janelas de tempo
# e = início da janela de tempo
# l = final da janela de tempo
# s = tempo de serviço
```

```
e = [0,12,22,24,7,22,24,11,25,22,28]
l = [9999,73,111,85,55,36,123,27,59,108,106]
s = [0,4,6,5,5,6,4,3,3,3,5]


# Cálculo do tempo de viagem entre dois nós
timetravel={(i,j,k):(np.hypot(loc_x[i]- loc_x[j],loc_y[i]-loc_y[j])//10)
                        for k in veiculos for i in nodes for j in nodes if i!=j}

for k in veiculos:
        if k<=Motos:
                for (i,j) in arcos:
                        timetravel[(i,j,k)]=timetravel[(i,j,k)]//4
        else:
                for (i,j) in arcos:
                        timetravel[(i,j,k)]=timetravel[(i,j,k)]*1
print("timetravel")
#print(timetravel)
for i in nodes:
        for j in nodes:
                if i!=j:
                        print(timetravel[(i,j,7)])


#Vetor do custo fixo e da capacidade por veículo
f=list(range(1,Frota+1))
Q=list(range(1,Frota+1))
cont=1
for x in range(Frota):
        if cont<=Motos:
                f[x]=Fmoto
                Q[x]=Qmoto
                cont=cont+1
        else:
                f[x]=Fbike
                Q[x]=Qbike

# Definição do problema
prob=LpProblem("VRPTW",LpMinimize)

rotas = {(i,j,k) for i in nodes for j in nodes if i!=j for k in veiculos}
bik = {(i,k) for i in nodes for k in veiculos}
transporte = {(k) for k in veiculos}

# Definição da variável x que será 1 se a localidade i estiver ligada a localidade j
x = LpVariable.dicts('x',rotas,0,1,LpBinary)

# Variável para início do atendimento do veículo k no cliente i
b = LpVariable.dicts('b',bik,lowBound=0,cat='Integer')
```

```
# Variável y que será 1 se o veículo k for utilizado
y = LpVariable.dicts('y',transporte,0,1,LpBinary)


prob+=lpSum(f[k-1]*y[k] for k in veiculos) + lpSum(x[(i,j,k)]*matrizg[(k,i,j)] for i,j,k in
rotas) + lpSum(x[(i,j,k)]*penalidade[(k,i,j)] for i,j,k in rotas)

for i in clientes:
        prob+=lpSum(x[i,j,k] for j in nodes if i!=j for k in veiculos)==1

for j in clientes:
        prob+=lpSum(x[i,j,k] for i in nodes if i!=j for k in veiculos)==1

for k in veiculos:
        prob+=(lpSum(q[i]*(lpSum(x[i,j,k] for j in nodes if i!=j))for i in clientes))<=Q[k-
1]*y[(k)]

for k in veiculos:
        prob+=lpSum(x[0,j,k] for j in clientes)==y[(k)]

for h in clientes:
        for k in veiculos:
                prob+=(lpSum(x[i,h,k] for i in nodes if i!=h)-lpSum(x[h,j,k] for j in nodes if
h!=j))==0

for k in veiculos:
        prob+=lpSum(x[i,0,k] for i in clientes)==y[(k)]

M=999999
alfa=0

for k in veiculos:
        b[0,k]=0
        s[0]=0

print(b)

for k in veiculos:
        for i in nodes:
                for j in clientes:
                        if i!=j:
                                prob+=b[i,k]+s[i]+timetravel[i,j,k]+(alfa*penalidade[k,i,j])-
M*(1-x[i,j,k])<=b[j,k]

for k in veiculos:
        for i in clientes:
                prob+=e[i]<=b[i,k]
                prob+=b[i,k]<=l[i]

prob.solve()
```

```
print(LpStatus[prob.status])

# Imprimindo as rotas para cada veículo
for k in veiculos:
        if y[k].varValue>0.5:
                i = 0
                j=1
                fim=0
                rota = [0]
                while fim==0:
                        while x[i,j,k].varValue<0.9:
                                j=j+1
                                if j==i:
                                        j=j+1
                                if j==n+1:
                                        j=0
                        rota.append(j)
                        if j==0:
                                fim=1
                        i=j
                        j=1
                        if j==i:
                                j=j+1
                if k<=Motos:
                        print("Moto :      ",rota)
                else:
                        print("Bicicleta : ",rota)


# Gráfico da solução ótima
arcos_ativos=[p for p in rotas if x[p].varValue>0.9]
plt.figure(figsize=(12,5))
plt.scatter(loc_x, loc_y, color="green")
for i in clientes:
        plt.annotate('$q_{%d}=%d$'%(i,q[i]),(loc_x[i],loc_y[i]))
        plt.annotate('$alt_{%d}=%d$'%(i,alt[i]),(loc_x[i],loc_y[i]-4))
plt.plot(loc_x[0],loc_y[0], color='red',marker='s')
plt.annotate('DC',(loc_x[0]-2, loc_y[0]+1.5))
plt.annotate('$alt_{%d}=%d$'%(0,alt[0]),(loc_x[0],loc_y[0]-2))
for i,j,k in arcos_ativos:
        if k<=Motos:
                plt.plot([loc_x[i],loc_x[j]],[loc_y[i],loc_y[j]], color="red",alpha=0.3)
        else:
                plt.plot([loc_x[i],loc_x[j]],[loc_y[i],loc_y[j]], color="blue",alpha=0.3)
plt.show()
```

## APÊNDICE B – Programa em Python com algoritmo genético

```
import numpy as np
import copy
import matplotlib.pyplot as plt
import time

ini = time.time()
rnd=np.random

# Definição do problema

# Lista de clientes
n = 10
clientes = [x for x in range(1,n+1)]
nodes = [0] + clientes

c = np.zeros((len(nodes),len(nodes)))
tij = np.zeros((len(nodes),len(nodes)))
pen = np.zeros((len(nodes),len(nodes)))

loc_x = [133,63,7,75,98,152,140,191,137,52,66]
loc_y = [75,29,77,80,65,31,4,98,54,51,93]

demanda = [0,9,8,4,6,4,2,8,9,4,5]

alt =  [4,15,17,15,9,23,0,24,5,14,24]
jinicio = [0,12,22,24,7,22,24,11,25,22,28]
jfim = [9999,73,111,85,55,36,123,27,59,108,106]
tservico = [0,4,6,5,5,6,4,3,3,3,5]

# Dados dos veículos
t=2
f=[5,100]
v=[1, 20]
Q=[15,20]
tf=[0,0]

# Calculo das distancias
for i in nodes:
    for j in nodes:
        xi,yi=loc_x[i],loc_y[i]
        xj,yj=loc_x[j],loc_y[j]
        c[i][j]=((np.hypot(xi-xj,yi-yj)))
        tij[i][j]=c[i][j]//10

# Cálculo do ângulo de elevação entre dois nós
angulo_rad = {(i,j):np.arctan((alt[j]-alt[i])/c[i][j]) for i in nodes for j in nodes if i!=j}
angulo_graus = {(i,j):np.degrees(angulo_rad[(i,j)]) for i in nodes for j in nodes if i!=j}
```

```
# Construção da matriz de penalidades
for i in nodes:
	for j in nodes:
		if i!=j:
			if angulo_graus[(i,j)]<=2:
				pen[i][j]=c[i][j]
			elif angulo_graus[(i,j)]<=4:
				pen[i][j]=c[i][j]*1.3
			elif angulo_graus[(i,j)]<=6:
				pen[i][j]=c[i][j]*1.7
			else:
				pen[i][j]=c[i][j]*50000


# Função Split
def SplitPrins(rota):

	V=[]
	P=[]

	V=np.zeros([n+1,1])
	V[0]=0
	W=max(Q)
	P=np.zeros([n+1,1])
	P[0]=99999
	Tipo=np.zeros([n+1,1])
	Tipo[0]=99999

	for i in rota:
		V[i]=99999
		P[i]=99999

	for i in range(1,len(rota)+1):
		load=0
		cost=0
		costb=0
		j=i
		car=0
		janela=0

		while True:
			load=load+demanda[rota[j-1]]
			if i==j:
				cost=c[0][rota[j-1]]+c[rota[j-1]][0]
				costb=pen[0][rota[j-1]]+pen[rota[j-1]][0]
				time_b=max(tij[0][rota[j-1]],jinicio[rota[j-1]])+tservico[rota[j-
1]]+tij[rota[j-1]][0]
				time_m=max(tij[0][rota[j-1]]/4,jinicio[rota[j-
1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]/4
```

```
                else:
                        cost=cost-c[rota[j-2]][0]+c[rota[j-2]][rota[j-1]]+c[rota[j-1]][0]
                        costb=costb-pen[rota[j-2]][0]+pen[rota[j-2]][rota[j-
1]]+pen[rota[j-1]][0]
                        time_b=max(time_b-tij[rota[j-2]][0]+tij[rota[j-2]][rota[j-
1]],jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]
                        time_m=max(time_m-tij[rota[j-2]][0]/4+tij[rota[j-2]][rota[j-
1]]/4,jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]/4
                tf[0]=time_b-tij[rota[j-1]][0]-tservico[rota[j-1]]
                tf[1]=time_m-tij[rota[j-1]][0]/4-tservico[rota[j-1]]

                if (tf[0]>jfim[rota[j-1]]):
                        janela=1

                if load<=W and ((tf[0]<=jfim[rota[j-1]]) or (tf[1]<=jfim[rota[j-1]])):
                        Zij=99999

                        if janela==0 and load<=Q[0]:
                                custo_bi=f[0]+v[0]*costb
                        else:
                                custo_bi=99999

                        custo_mo = f[1]+v[1]*cost

                        if custo_bi<=custo_mo:
                                car=0
                                Zij=custo_bi
                        else:
                                car=1
                                Zij=custo_mo


                        if (V[i-1]+Zij)<V[j]:
                                V[j]=V[i-1]+Zij
                                P[j]=i-1
                                Tipo[j]=car

                        j=j+1


                if (j>n) or (load>W) or ((time_m-tij[rota[j-2]][0]/4-tservico[rota[j-
2]])>(jfim[rota[j-2]])):
                        break
        resultado = V[n]
        return resultado


# Geracao da populacao inicial
rota=[]
i=1
```

```
while i<=n:
        rota.append(i)
        i+=1
N=10
pop={}
valor={}
p=1
pop[p]=copy.deepcopy(rota)
valor[p]=int(SplitPrins(rota))
while p<N:
   p=p+1
   rnd.shuffle(rota)
   pop[p]=copy.deepcopy(rota)
   valor[p]=int(SplitPrins(rota))

# Melhor solução inicial
melhor_minimo=min(valor.values())
melhor_posicao=min(valor, key=valor.get)
melhor_rota=copy.deepcopy(pop[melhor_posicao])


# Algoritmo genético

# 100 gerações
Num_geracao = 1
while Num_geracao<100:

# Selecao de dois pais para crossover OX
        reproducao=1
        while reproducao<50:
                p=rnd.randint(1,N)
                q=rnd.randint(1,N)
                while p==q:
                        q=rnd.randint(1,N)
                pai_1 = pop[p]
                pai_2 = pop[q]
                filho_1=[]
                filho_2=[]
                i=0
                while i<n:
                                filho_1.append(0)
                                filho_2.append(0)
                                i+=1
                ponto_crossover1=rnd.randint(1,n-1)
                ponto_crossover2=rnd.randint(1,n-1)
                while ponto_crossover1==ponto_crossover2:
                        ponto_crossover2=rnd.randint(1,n-1)
                if (ponto_crossover1)<ponto_crossover2:
                        P1=ponto_crossover1
                        P2=ponto_crossover2
```

```
else:
        P1=ponto_crossover2
        P2=ponto_crossover1
indice=P1
while indice<=P2:
        filho_1[indice]=pai_1[indice]
        filho_2[indice]=pai_2[indice]
        indice+=1
indpai=P2+1
if indpai==n:
        indpai=0
indfilho=P2+1
if indfilho==n:
        indfilho=0
while indfilho!=P1:
        if pai_2[indpai] in filho_1:
                indpai+=1
                if indpai==n:
                        indpai=0
        if pai_2[indpai] not in filho_1:
                filho_1[indfilho]=pai_2[indpai]
                indpai+=1
                if indpai==n:
                        indpai=0
                indfilho+=1
                if indfilho==n:
                        indfilho=0
indpai=P2+1
if indpai==n:
        indpai=0
indfilho=P2+1
if indfilho==n:
        indfilho=0
while indfilho!=P1:
        if pai_1[indpai] in filho_2:
                indpai+=1
                if indpai==n:
                        indpai=0
        if pai_1[indpai] not in filho_2:
                filho_2[indfilho]=pai_1[indpai]
                indpai+=1
                if indpai==n:
                        indpai=0
                indfilho+=1
                if indfilho==n:
                        indfilho=0
a=int(SplitPrins(filho_1))
if a<=valor[p]:
        pop[p]=copy.deepcopy(filho_1)
        valor[p]=a
```

```
                if a<=valor[q]:
                        pop[q]=copy.deepcopy(filho_1)
                        valor[q]=a
                a=int(SplitPrins(filho_2))
                if a<=valor[p]:
                        pop[p]=copy.deepcopy(filho_2)
                        valor[p]=a
                if a<=valor[q]:
                        pop[q]=copy.deepcopy(filho_2)
                        valor[q]=a
                reproducao+=1

        minimo_geracao=min(valor.values())
        minpos_geracao=min(valor, key=valor.get)
        if minimo_geracao<melhor_minimo:
                melhor_minimo=minimo_geracao
                melhor_rota=copy.deepcopy(pop[minpos_geracao])

# Crossover ordenado crescente
        contador=1
        while contador<30:
                p=rnd.randint(1,N)
                q=rnd.randint(1,N)
                while p==q:
                        q=rnd.randint(1,N)
                pai_1 = copy.deepcopy(pop[p])
                pai_2 = copy.deepcopy(pop[q])
                filho=[]
                i=0
                while i<n:
                        filho.append(0)
                        i+=1
                filho[0]=pai_1[0]
                ind=pai_2.index(pai_1[0])
                pai_2[ind]=pai_2[0]
                pai_2[0]=pai_1[0]
                i=0
                while i<n-1:
                        co1=filho[i]
                        co2=pai_1[i+1]
                        co3=pai_2[i+1]
                        if c[co1][co2]<=c[co1][co3]:
                                filho[i+1]=co2
                                ind=pai_2.index(co2)
                                pai_2[i+1]=co2
                                pai_2[ind]=co3
                        if c[co1][co2]>c[co1][co3]:
                                filho[i+1]=co3
                                ind=pai_1.index(co3)
                                pai_1[i+1]=co3
```

```
                        pai_1[ind]=co2
                i+=1
        ax=SplitPrins(pop[p])
        bx=SplitPrins(pop[q])
        cx=SplitPrins(filho)
        if ax>bx:
                pop[p]=copy.deepcopy(filho)
                valor[p]=cx
                posicao=p
        else:
                pop[q]=copy.deepcopy(filho)
                valor[q]=cx
                posicao=q
        if cx<melhor_minimo:
                melhor_minimo=cx
                melhor_rota=copy.deepcopy(filho)
        contador+=1

# Crossover ordenado decrescente
    contador=1
    while contador<30:
        p=rnd.randint(1,N)
        q=rnd.randint(1,N)
        while p==q:
                q=rnd.randint(1,N)
        pai_1 = copy.deepcopy(pop[p])
        pai_2 = copy.deepcopy(pop[q])
        filho=[]
        i=0
        while i<n:
                filho.append(0)
                i+=1
        filho[n-1]=pai_1[n-1]
        ind=pai_2.index(pai_1[n-1])
        pai_2[ind]=pai_2[n-1]
        pai_2[n-1]=pai_1[n-1]
        i=n-1
        while i>0:
                co1=filho[i]
                co2=pai_1[i-1]
                co3=pai_2[i-1]
                if c[co1][co2]<=c[co1][co3]:
                        filho[i-1]=co2
                        ind=pai_2.index(co2)
                        pai_2[i-1]=co2
                        pai_2[ind]=co3
                if c[co1][co2]>c[co1][co3]:
                        filho[i-1]=co3
                        ind=pai_1.index(co3)
                        pai_1[i-1]=co3
```

```
                        pai_1[ind]=co2
                    i-=1
                ax=SplitPrins(pop[p])
                bx=SplitPrins(pop[q])
                cx=SplitPrins(filho)
                if ax>bx:
                    pop[p]=copy.deepcopy(filho)
                    valor[p]=cx
                    posicao=p
                else:
                    pop[q]=copy.deepcopy(filho)
                    valor[q]=cx
                    posicao=q
                if cx<melhor_minimo:
                    melhor_minimo=cx
                    melhor_rota=copy.deepcopy(filho)
                contador+=1

# Mutação 1 de troca de genes
        Qtd_mut=50
        n_mut=1
        while n_mut<Qtd_mut:
                y=rnd.randint(1,N+1)
                vantes=SplitPrins(pop[y])
                ponto_mutacao1=rnd.randint(0,n)
                ponto_mutacao2=rnd.randint(0,n)
                while ponto_mutacao1==ponto_mutacao2:
                    ponto_mutacao2=rnd.randint(0,n)
                pop[y][ponto_mutacao1], pop[y][ponto_mutacao2] =
pop[y][ponto_mutacao2], pop[y][ponto_mutacao1]
                vdepois=SplitPrins(pop[y])
                if vdepois<melhor_minimo:
                    melhor_minimo=vdepois
                    melhor_rota=copy.deepcopy(pop[y])
                if vantes<vdepois:
                    pop[y][ponto_mutacao1], pop[y][ponto_mutacao2] =
pop[y][ponto_mutacao2], pop[y][ponto_mutacao1]
                n_mut+=1

# Mutação 2 de troca de genes
        Qtd_mut=50
        n_mut=1
        while n_mut<Qtd_mut:
                y=rnd.randint(1,N+1)
                ponto_mutacao1=rnd.randint(0,n-1)
                pont_mutacao2=ponto_mutacao1+1
                pop[y][ponto_mutacao1], pop[y][ponto_mutacao2] =
pop[y][ponto_mutacao2], pop[y][ponto_mutacao1]
                vdepois=SplitPrins(pop[y])
                if vdepois<melhor_minimo:
```

```
                    melhor_minimo=vdepois
                    melhor_rota=copy.deepcopy(pop[y])
            n_mut+=1
        minimo_geracao=min(valor.values())
        minpos_geracao=min(valor, key=valor.get)
        if minimo_geracao<melhor_minimo:
            melhor_minimo=minimo_geracao
            melhor_rota=copy.deepcopy(pop[minpos_geracao])

# Mutação 2-OPT
        Qtd_mut=50
        n_mut=1
        while n_mut<Qtd_mut:
            y=rnd.randint(1,N+1)
            x1=rnd.randint(1,n)
            while (n-x1)<=1:
                x1=rnd.randint(1,n)
            x2=rnd.randint(1,(n-x1))
            x2=x1+x2
            mut2pot=[]
            for a in range(0,x1):
                mut2pot.append(pop[y][a])
            teta=x2-x1
            for a in range(0,teta+1):
                mut2pot.append(pop[y][x2-a])
            for a in range(x2+1,n):
                mut2pot.append(pop[y][a])
            vmut=SplitPrins(mut2pot)
            if vmut<valor[y]:
                pop[y]=copy.deepcopy(mut2pot)
                valor[y]=vmut
            if valor[y]<melhor_minimo:
                melhor_minimo=valor[y]
                melhor_rota=copy.deepcopy(pop[y])
            n_mut+=1
        minimo_geracao=min(valor.values())
        minpos_geracao=min(valor, key=valor.get)
        if minimo_geracao<melhor_minimo:
            melhor_minimo=minimo_geracao
            melhor_rota=copy.deepcopy(pop[minpos_geracao])

# Elitismo
        Elite=2
        cont_elite=1
        while cont_elite<Elite:
            x=rnd.randint(1,N)
            pop[x]=copy.deepcopy(melhor_rota)
            valor[x]=melhor_minimo
            cont_elite+=1
```

```
        Num_geracao+=1
        tempo = time.time()
        print("Num geracao ",Num_geracao,"      Melhor valor : ",melhor_minimo," Tempo ",tempo-ini)
print(melhor_rota)

# Extração das rotas a partir do vetor P
rota=copy.deepcopy(melhor_rota)


V=[]
P=[]

V=np.zeros([n+1,1])
V[0]=0
W=max(Q)
P=np.zeros([n+1,1])
P[0]=99999
Tipo=np.zeros([n+1,1])
Tipo[0]=99999

for i in rota:
        V[i]=99999
        P[i]=99999

for i in range(1,len(rota)+1):
        load=0
        cost=0
        costb=0
        j=i
        car=0
        janela=0

        while True:
                load=load+demanda[rota[j-1]]
                if i==j:
                        cost=c[0][rota[j-1]]+c[rota[j-1]][0]
                        costb=pen[0][rota[j-1]]+pen[rota[j-1]][0]
                        time_b=max(tij[0][rota[j-1]],jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]
                        time_m=max(tij[0][rota[j-1]]/4,jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]/4
                else:
                        cost=cost-c[rota[j-2]][0]+c[rota[j-2]][rota[j-1]]+c[rota[j-1]][0]
                        costb=costb-pen[rota[j-2]][0]+pen[rota[j-2]][rota[j-1]]+pen[rota[j-1]][0]
                        time_b=max(time_b-tij[rota[j-2]][0]+tij[rota[j-2]][rota[j-1]],jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]
                        time_m=max(time_m-tij[rota[j-2]][0]/4+tij[rota[j-2]][rota[j-1]]/4,jinicio[rota[j-1]])+tservico[rota[j-1]]+tij[rota[j-1]][0]/4
```

```
                tf[0]=time_b-tij[rota[j-1]][0]-tservico[rota[j-1]]
                tf[1]=time_m-tij[rota[j-1]][0]/4-tservico[rota[j-1]]

                if (tf[0]>jfim[rota[j-1]]):
                        janela=1

                if load<=W and ((tf[0]<=jfim[rota[j-1]]) or (tf[1]<=jfim[rota[j-1]])):
                        Zij=99999

                        if janela==0 and load<=Q[0]:
                                custo_bi=f[0]+v[0]*costb
                        else:
                                custo_bi=99999

                        custo_mo = f[1]+v[1]*cost

                        if custo_bi<=custo_mo:
                                car=0
                                Zij=custo_bi
                        else:
                                car=1
                                Zij=custo_mo


                        if (V[i-1]+Zij)<V[j]:
                                V[j]=V[i-1]+Zij
                                P[j]=i-1
                                Tipo[j]=car

                        j=j+1


                if (j>n) or (load>W) or ((time_m-tij[rota[j-2]][0]/4-tservico[rota[j-
2]])>(jfim[rota[j-2]])):
                        break
resultado = V[n]




trip=[]
trucktype=[]
for i in range(0,n):
        trucktype.append(0)
for i in range(0,n):
        l=[]
        trip.append(l)
```

```
t=0
j=n

while True:

        i=int(P[j])
        for k in range(i+1,j+1):
                trip[t].append(rota[k-1])
                trucktype[t]=int(Tipo[j])
        j=i
        if i==0:
                break
        t=t+1

print(trip)
print(trucktype)


# Gráfico da solução
plt.figure(figsize=(12,5))
plt.scatter(loc_x,loc_y, color="green")
plt.plot(loc_x[0], loc_y[0], color='red',marker='s')
plt.annotate('DC',(loc_x[0]+1.5, loc_y[0]+1.5))
z=len(trip)
for i in range(0, z):
        if trucktype[i]==0:
                cor="blue"
        elif trucktype[i]==1:
                cor="red"
        if len(trip[i])>0:
                PontoA=0
                for j in range(0, len(trip[i])):
                        PontoB=trip[i][j]

        plt.plot([loc_x[PontoA],loc_x[PontoB]],[loc_y[PontoA],loc_y[PontoB]],
color=cor,alpha=0.3)
                        PontoA=PontoB
                plt.plot([loc_x[PontoB],loc_x[0]],[loc_y[PontoB],loc_y[0]],
color=cor,alpha=0.3)
plt.show()
```