Maximizando a Eficiência de Entrega em Sistemas de Transporte por Drones: Exploração de Estratégias de Otimização de Rotas

Vinícius S. Lima[1], Vinícius da S. Gomes[1], Vinícius F. Ferraz[1], e Heitor L. L. Pereira[1]

[1]*Departamento de Estatística e Informática (DEINFO), Universidade Federal Rural de Pernambuco, Brasil*

Submetido: 4 Setembro 2023

Abstração

Este estudo aborda as dificuldades enfrentadas na implementação de um sistema de entregas via drones que se enquadra no contexto do problema do caixeiro-viajante. O objetivo principal é propor soluções eficientes que visam minimizar custos, reduzir impactos ambientais e otimizar o tempo de entrega. Por meio de uma análise detalhada das variáveis críticas, como distância das rotas, duração da bateria dos drones e capacidade de processamento, busca-se encontrar uma solução que facilite e contorne as limitações existentes nessa problemática. Ao considerar o desafio da implementação de um sistema de entregas via drones, é crucial garantir que as rotas sejam planejadas de forma eficiente, de modo que cada drone possa percorrer todos os pontos de entrega de maneira otimizada e retornar à base para recarregar a bateria. Isso se dará através da implementação de três algoritmos para roteirização dos drones: o algoritmo da força bruta, o algoritmo genético e o algoritmo da colônia de formigas. Por meio desses algoritmos, foram encontradas soluções ótimas ou quase ótimas para o PCV em tempo hábil.

Palavras-chave: FlyFood, Sistema de entregas, Drones, Caixeiro viajante, Otimização, Rotas de entrega, Consumo de bateria, Algoritmo de força bruta, Menor caminho hamiltoniano, Complexidade computacional, Meta-heurísticas, Algoritmo genético, Algoritmo da colÔnia de formigas

1 Introdução

1.1 Apresentação e Motivação

A Agência Nacional de Aviação Civil (ANAC) autorizou, em 2020, o uso experimental de drones para serviços de entrega. Atualmente, essas operações de entrega envolvem o transporte dos produtos para locais intermediários, onde são posteriormente repassados para os entregadores que realizam a entrega final ao cliente. Além disso, em virtude da pandemia, a procura por serviços de delivery cresceu bastante. Nesse contexto, a empresa FlyFood propõe uma estratégia promissora para contornar o trânsito caótico nas cidades: a utilização de drones para fazer entregas de comida aos clientes.

Esta abordagem tem um grande potencial, considerando o previsto crescimento no uso doméstico de drones em setores como segurança, agricultura, eventos e serviços de entrega. A entrega de produtos por drones oferece vantagens como velocidade aumentada, menor custo de transporte, acesso facilitado a áreas remotas e maior conveniência para os consumidores, o que desperta interesse tanto de empresas quanto de consumidores e impulsiona o desenvolvimento e a adoção de soluções de entrega baseadas em drones.

No entanto, um obstáculo central precisa ser superado para a adoção em larga escala deste serviço: a necessidade de encontrar rotas de entrega de forma eficiente em um curto período de tempo. A capacidade limitada das baterias dos drones, geralmente em torno de 30 minutos de voo, torna inviável percorrer longas distâncias sem a necessidade de recarregá-las. Este fator representa um desafio significativo para as operações de entrega, pois é necessário garantir que os drones passem por todos os pontos de entrega de maneira eficiente, retornando ao ponto de partida para recarga.

Dessa forma, torna-se essencial o desenvolvimento de estratégias que permitam calcular as rotas de entrega de maneira rápida e otimizada, garantindo a máxima utilização do tempo disponível. A otimização dessas rotas não só economiza tempo, mas também permite uma utilização mais adequada da bateria dos drones, prolongando sua vida útil. O desenvolvimento de métodos e algoritmos eficazes para a determinação das melhores rotas é fundamental para viabilizar a adoção em larga escala dos drones na entrega de produtos, proporcionando melhorias significativas na logística e trazendo benefícios tanto para as empresas quanto para os consumidores.

1.2 Formalização do problema

1.2.1 Modelagem via otimização

O problema em questão é conhecido como o problema do caixeiro viajante (PCV) e é um tipo impor-

tante de problema na área da otimização combinatória. No PCV, temos um caixeiro cujo objetivo é encontrar a rota mais curta que passe por todas as cidades e volte para a cidade de origem. Considerando esse cenário, é possível resolver esse problema por meio de modelagem via otimização. Como o objetivo é encontrar o menor caminho hamiltoniano, que é o caminho onde se passa por todos os vértices somente uma vez e retorna ao vértice de origem, este problema é de minimização.

1.2.1.1 Variáveis de decisão

- Seja P uma sequência de cidades a serem visitadas na qual há n pontos de entrega. $P = (p1, p2 \ldots pn)$.
- Sejam p_i, p_j as coordenadas cartesianas da cidade qualquer.
- Seja $d(i, j)$ a distância entre duas cidades, definida como $|p_{i1} - p_{j1}| + |p_{i2} - p_{j2}|$.
- Seja r um ponto que representa o restaurante, de coordenadas do restaurante: p_r, p_r.

1.2.1.2 A função objetivo

A função utilidade u(P) é um laço no qual soma todas as permutações de caminhos de pi até pn e retorna os índices do menor caminho por meio da função argmin pu(P).

$$P = (p_1, p_2, \ldots, p_n) \tag{1}$$

$$d(p_i, p_j) = |p_{i1} - p_{j1}| + |p_{i2} - p_{j2}| \tag{2}$$

$$u(P) = d(r, p_1) + \sum_{i=1}^{n-1} d(p_i, p_{i+1}) + d(p_n, r) \tag{3}$$

$$\text{argmin}_P \, u(P) \tag{4}$$

1.2.1.3 Restrições

- O caixeiro deve passar por cada cidade exatamente uma vez.
- O caixeiro deve sair de cada cidade exatamente uma vez.
- O caixeiro não pode visitar uma cidade duas vezes consecutivas.
- O caixeiro deve retornar à cidade de origem depois de passar por todas as outras cidades exatamente uma vez.

Ao abordar esse problema, a solução ideal seria testar todas as rotas possíveis para garantir a melhor rota usando o algoritmo de força bruta. No entanto, existem três impedimentos para a solução deste problema em tempo hábil.

Em primeiro lugar, não há um algoritmo que garanta a resolução do problema em tempo hábil. Em segundo lugar, ao adicionar uma cidade ao problema, será necessário calcular $(n + 1)$ vezes mais rotas. Por último, é preciso calcular todas as rotas possíveis para garantir o menor caminho hamiltoniano, e isso pode se tornar muito demorado à medida que o número de cidades aumenta. No entanto, para os fins de roteirização de drones essa abordagem ainda é viável.

1.3 Objetivos

O objetivo principal é desenvolver estratégias que possibilitem o cálculo eficiente das rotas de entrega, maximizando o tempo de voo disponível ao encontrar uma rota ótima para o percurso dos drones do FlyFood. Para alcançar esse objetivo, serão realizadas investigações teóricas sobre o PCV e suas implicações computacionais. Além disso, será criado um algoritmo que garanta a solução ótima do PCV. Será estudado a variação do tempo de execução por meio de algoritmos de força bruta.

Espera-se que este trabalho contribua para a adoção em larga escala dos drones na entrega de produtos, extraindo ao máximo sua capacidade por meio das melhores rotas. Ademais, busca-se ampliar o conhecimento sobre a complexidade computacional do PCV e oferecer soluções eficientes para o desafio da roteirização de drones.

1.4 Organização do Trabalho

Este artigo está estruturado em seções para facilitar a leitura e a compreensão do conteúdo. Na seção 2, de Referencial Teórico, são explicados os conceitos de problemas computacionais que têm relação com o problema do FlyFood. Na seção 3 ,de Trabalhos Relacionados, serão discutidos outros estudos que complementam a análise do PCV e, consequentemente, do FlyFood. Na seção 4, de Metodologia, serão descritos os materiais, ferramentas e dados utilizados de forma apropriada para encontrar a solução. Na seção 5,de Experimentos, serão explicados os procedimentos experimentais realizados para comprovar a eficácia do método escolhido. A seção 6, dos Resultados, apresentará as descobertas obtidas por meio deste estudo. Por fim, a seção 7, de conclusão, encerra o trabalho, sintetizando todas as ideias apresentadas e destacando os impactos que essas ideias terão na resolução deste tipo de problema.

2 Referencial Teórico

Tendo em vista a complexidade do algoritmo do PCV aplicado ao contexto do FlyFood, se faz necessário introduzir conceitos de classes de complexidade e análise de algoritmos. Nesta análise, serão citados conceitos como algoritmo, classes de completude,satisfabilidade, redutibilidade, grande O e tempo de execução. Além disso, serão apresentadas o conceito de meta-heurística e dois algoritmos meta-heurísticos para o PCV: o algoritmo da colônia de formigas e o algoritmo genético.

2.1 Classes de problemas

2.1.1 Algoritmos

Antes de definir classes de complexidade, é importante definir o conceito de algoritmo."Um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída"(Cormen,2002.P.3). Atualmente, existem vários algoritmos que nos dão respostas boas em tempo hábil ao PCV, mas não nos garantem o menor caminho possível.

2.1.2 Tempo Polinomial

Quando o tempo de execução é polinomial o problema é considerável tratável.Mesmo que a ordem do programa seja de n elevado a 100, por exemplo, este problema ainda é considerável filosoficamente tratável. Pois, para Cormen (Cormen, 2012. P.779)"a experiência mostrou que, tão logo seja descoberto o primeiro algoritmo de tempo polinomial para um problema, em geral algoritmos mais eficientes vêm logo atrás.".

2.1.3 Classes de complexidade

As classes de complexidade são um conjunto de problemas que têm complexidades similares em relação ao uso de recursos tais quais tempo, memória e energia. Neste trabalho, serão estudadas as quatro principais classes de complexidade.

2.1.3.1 Classe P

A classe P é formada por problemas que têm algoritmos de solução eficientes por resolverem em tempo polinomial. Os problemas da classe P são problemas de decisão que podem ser respondidos com sim-ou-não. Alguns exemplos são o algoritmo do caminho mínimo e algoritmos de ordenação.Os problemas da classe P são problemas de decisão que podem ser respondidos com sim-ou-não.

Tem um tipo especial de problema de decisão que não pertence à classe P: a Satisfabilidade (SAT), que usa proposições como x_n e operadores lógicos como conjunção ($\wedge$) e disjunção ($\vee$). Tal como no exemplo:

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_2) \tag{5}$$

Neste tipo de problema os valores booleanos que garantem que a expressão retorne sim são chamados de Certificado. Neste exemplo o certificado é $x_1 = \mathrm{V}$, $x_2 = \mathrm{V}$ e $x_3 = \mathrm{V}$ ou F.

2.1.3.2 Classe NP

Antes de introduzir a definição de problemas NP é necessário o conceito de verificabilidade de algoritmos. Para Cormen (Cormen,2012. P.784):

> "Definimos algoritmo de verificação como um algoritmo de dois argumentos A, onde um argumento é uma cadeia de entrada comum x e o outro é uma cadeia binária y denominada certificado. Um algoritmo A de dois argumentos verifica uma cadeia de entrada x se existe um certificado y tal que A(x, y) = 1. A linguagem verificada por um algoritmo de verificação A é".

$$L = \{x \in \{0,1\}^* : \text{ existe } y \in \{0,1\}^* \text{ tal que } A(x,y) = 1\}. \tag{6}$$

> "Intuitivamente, um algoritmo A verifica uma linguagem L se, para qualquer cadeia $x \in L$, existe um certificado y que A pode utilizar para provar que $x \in L$. Além disso, para qualquer cadeia $x \notin L$, não deve existir nenhum certificado que prove que $x \in L$.".

Deste modo, a definição formal de Cormen (Cormen,2012,P.784) para problemas NP é:

> "A classe de complexidade NP é a classe de linguagens que podem ser verificadas por um algoritmo de tempo polinomial. Mais precisamente, uma linguagem L pertence a NP se e somente se existe um algoritmo de tempo polinomial de duas entradas A e uma constante c tal que".

$$L = \{x \in \{0,1\}^* : \text{ existe um certificado } y \text{ com } |y| = O(|x|^c) \text{ tal que } A(x,y) = 1\} \tag{7}$$

Ou seja, na classe dos problemas NP, o certificado pode ser verificado em tempo polinomial. Contudo, não existe um algoritmo que garanta solução ótima em tempo polinomial.

2.1.3.3 Classe NP-completo

Para provar que um problema é NP-completo é necessária a noção de redutibilidade. Um problema A é redutível a um problema $B(A\alpha B)$ se para qualquer instância d_A de A podemos construir uma instância d_B de B em tempo polinomial de modo que dA resulta em sim para A se, e somente se, dB resulta em sim para B.

O problema A é NP completo se B pertence a NP dado qualquer A pertencente a NP, então $A\alpha B$. Dados A e B em NP, tal que $A\alpha B$. Se B pertence a P, então A pertence a P. Se A pertence a NP-completo,então B pertence a NP-completo.

2.1.3.4 Classe NP-difícil

Esta classe de complexidade não tem apenas problemas de decisão, mas também tem problemas de otimização que consistem em encontrar a melhor solução

de todas as possíveis respostas. De maneira simples, o problema NP difícil é tão difícil quanto os problemas mais difíceis em NP.

Para a definição formal de problema NP-difícil é necessário o conceito de oráculo. Em um problema de decisão ou otimização A, dizemos que um algoritmo possui A como oráculo quando este algoritmo resolve uma instância de A com uma única instrução. Um problema A é NP-difícil se existe um algoritmo de tempo polinomial para um problema B NP-completo quando o algoritmo A como oráculo.

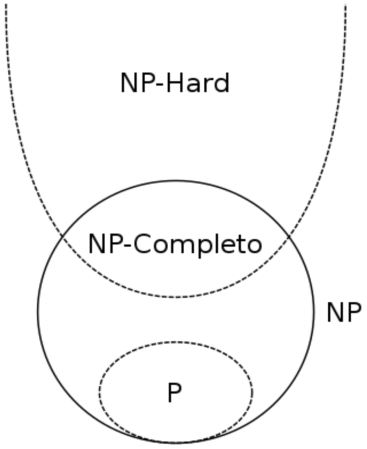

Figura 1: Relação de pertinência das classes

A relação de pertinência das classes é ilustrada na figura 1. Vale destacar que a classe P está contida na classe NP, mas que a classe NP-difícil não está na classe NP. Quando um problema é NP-difícil e NP, ele é NP -completo.

O PCV viajante pertence a classe de problemas NP-difícil, portanto não há nenhum algoritmo que garanta a solução ótima em tempo polinomial. A seguir será discutido o crescimento do tempo de execução do PCV.

2.2 Tempo de execução

2.2.1 A análise de pior caso

O PCV é um problema computacional de complexidade NP-difícil. No algoritmo de força bruta são permitidos $n!$ caminhos, sendo n o número de vértices . Em virtude do número de caminhos crescer de maneira fatorial se torna muito demorado para calcular o melhor percurso possível à medida que o número de vértices aumenta.

2.2.1.1 Notação Big O

A notação big O denota uma abstração do comportamento de uma função. Considerando apenas o termo mais significativo quando a função tende ao infinito. Por exemplo, considerando a função $f(x) = 2x^4 + 7x^2 + x + 4$, quando o x tende ao infinito o termo $2x^4$ sobrepuja a imagem dos outras termos, devido ao seu maior crescimento, por isso a notação big(O). Neste caso, o big(O) do PCV é igual a $n!$.

A notação big O também serve para classificar os algoritmos por ordem de eficiência. Como ilustrado na figura 2 que mostra a relação entre o número de elementos e o número de operações para cada valor de big(O). Nesta figura, nota-se o rápido crescimento no número de operações para problemas de big O $n!$.

2.2.1.2 Tempo de Execução pelo Número de Vértices

É importante destacar que o algoritmo de força bruta é uma abordagem exata, garantindo a solução ótima para o cálculo do menor caminho hamiltoniano possível. No entanto, esta abordagem tem limitações em termos de tempo de processamento. O número de rotas a serem avaliadas cresce de forma fatorial à medida que pontos de entrega são adicionados.

Para ilustrar esta complexidade, considere o exemplo de uma rota com n vértices. Se $n = 5$, existem 120 possibilidades de rotas (5!). Ao adicionar um ponto de entrega, o número de possibilidades aumenta de acordo com a fórmula recursiva: $f(n) = n * (n-1)!$. Assim, para $n = 6$, existem 720 possibilidades ($6 * 120$), e para $n = 7$, existem 5040 possibilidades ($7 * 720$). O número de possibilidades aumenta significativamente conforme mais pontos de entrega são adicionados.

Portanto, quando o número de paradas que o drone precisa fazer no percurso é grande, outras abordagens, como algoritmos heurísticos e meta-heurísticos, podem ser exploradas. Essas abordagens buscam soluções aproximadas e eficientes, permitindo encontrar rotas aceitáveis em um tempo razoável. Dessa forma, é possível lidar com o problema de forma mais fácil, mesmo quando o algoritmo de força bruta se torna inviável devido à sua complexidade fatorial.

A tabela 1 representa o tempo decorrido para a solução de um problema de n vértices. Até 10 vértices é possível a solução em tempo hábil no contexto do problema do FlyFood, contudo o tempo de execução cresce de maneira fatorial e logo se torna um problema intratável.

Tabela 1: Tempo decorrido para a solução do PCV para n vértices

n	**Nº de Caminhos**	**Tempo (1 μs/Cam.)**
5	12	12 μs
10	182,440	0.18 s
15	4.359×10^{10}	12h
20	6.082×10^{16}	1,928 anos
61	4.160×10^{81}	13.19×10^{67} anos

2.3 Meta-heurísiticas e Heurísticas

Como visto, o PCV se torna inviável para o problema do FlyFood quando há muitos vértices. Para diminuir o tempo de execução do PCV se fazem necessários métodos meta-heuristicos. Vale a pena lembrar que meta-heurísticas são diferentes de heurísticas, segundo Chopard:

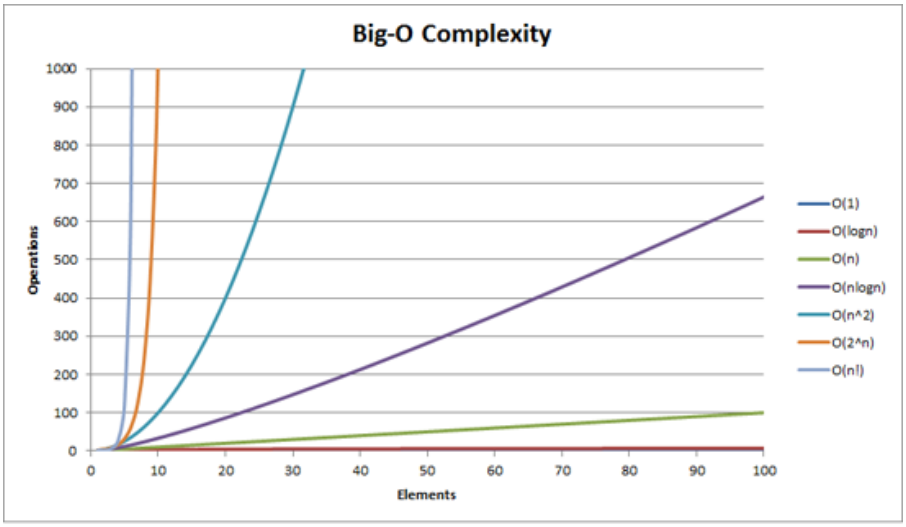

Figura 2: Complexidade Big-O.

> "Métodos heurísticos são usados quando métodos rigorosos não podem ser aplicados, tipicamente porque seriam muito lentos. Uma meta-heuristica é uma forma de otimização geral que é usada para controlar uma heurística específica do problema subjacente, de modo que o método possa ser facilmente aplicado a diferentes problemas. (CHOPARD, 2018, p.1, tradução nossa)."

Estas técnicas não garantem uma solução ótima, contudo, na maioria das vezes geram uma solução boa o suficiente em tempo hábil. Por este motivo, serão exploradas duas meta-heurísticas neste trabalho.

2.4 Algoritmo genético

Um algoritmo genético(AG) é um algoritmo que imita a seleção natural, teoria proposta pelo naturalista inglês Charles Darwin, na qual os indivíduos mais adaptados ao meio que estão inseridos têm maiores chances de se reproduzir. Uma vantagem deste método é que não é necessário saber resolver o problema, apenas é necessário gerar e avaliar as soluções que serão otimizadas pela seleção natural.

A implementação do AG é simples. Pode ser explicada em seis etapas: gerar população inicial, avaliar a população, selecionar os indivíduos mais aptos, reprodução dos indivíduos mais aptos, mutação aleatória e repetição até que haja uma situação de parada.

2.4.1 Gerar população inicial

A população inicial é gerada de maneira aleatória. No caso do PCV, um indivíduo seria uma permutação qualquer dos vértices de um ciclo hamiltoniano. Vale ressaltar que não é permitida a repetição de vértices no PCV.

2.4.2 Avaliar a população

Esta é uma das etapas que mais se repetirá no AG. Nela, é calculada a aptidão(fitness) que serve como base para a chance de reprodução de cada indivíduo. Geralmente, o fitness é definido como um critério para julgar possíveis soluções do problema a ser resolvido. Assim, o indivíduo que é a resposta mais adequada ao problema tende a se reproduzir mais e passar seus genens favoráveis adiante.

2.4.3 Seleção dos indivíduos mais aptos

Nesta fase são selecionados os indivíduos que serão pais. A seleção é bastante influenciada pela aptidão. Como dito por Chopard:

> "A seleção atua nos indivíduos da população de acordo com sua aptidão: aqueles com maior aptidão têm maior probabilidade de serem reproduzidos, enquanto os piores têm maior probabilidade de desaparecer.(CHOPARD, 2018, p.116, tradução nossa).".

Ou seja, não é comum indivíduos com fitness menor serem selecionados para passar adiante suas características. Mas ainda é possível.

2.4.4 Reprodução dos indivíduos mais aptos

Esta fase é baseada no fenômeno da reprodução sexuada. Quando ocorre, a partir do crossover (ponto do trajeto escolhido aleatoriamente), cada um dos pais contribui com uma parte do seu DNA. No caso do PCV, cada pai contribui com uma sequência de vértices, gerando um filho diferente dos pais.

2.4.5 Mutação Aleatória

Esta é uma característica dos filhos que não foi herdada diretamente dos pais. Mas, adquirida por meio de um erro de transcrição do material genético. Como

é um erro decorrente do crossover, a taxa de mutação geralmente é baixa.

2.4.6 Repetição

As etapas anteriores, com exceção da primeira, repetem-se até que seja satisfeita uma condição de parada. Essa condição pode ocorrer de diversas maneiras: se um número específico de gerações é atingido, se um número específico de indivíduos é atingido ou se um indivíduo é a resposta ótima para o problema.

O AG é baseado no método de população, em que se parte de um conjunto de soluções iniciais (a população inicial) e tenta encontrar uma solução melhor alterando-se elementos dessa população. O método a seguir é baseado no método construtivo, no qual, partindo de um conjunto solução vazio, adicionam-se elementos a esse conjunto até obter uma solução viável para o problema.

2.5 Algoritmo da Colônia de Formigas

Este método se baseia em uma característica biológica de certos tipos de animais: a inteligência de enxame. Uma colônia de formigas é capaz de resolver problemas de otimização tais como achar a menor distância entre a comida e o formigueiro, coisa que uma formiga sozinha é incapaz de fazer.

No contexto do PCV, no algoritmo da colônia de formigas (ACO) cada formiga representa uma possível solução. Cada formiga decide que vértice será visitado por meio de feromônios depositados por outras formigas e distância entre os vértices, a cada iteração os feromônios são atualizados de acordo com as soluções de menor custo.

Isso incentiva as formigas a visitar vértices que estão associadas a soluções melhores.De maneira simples, o ACO tem 4 etapas: inicialização, movimento, atualização das trilhas de feromônios e repetição.

2.5.1 Inicialização

Para cada formiga, é sorteado um vértice inicial para ela. O número inicial de formigas não pode ser muito pequeno para gerar poucas soluções, nem muito grande para que haja soluções demais a serem avaliadas. Na primeira iteração, as formigas passam por todos os vértices de maneira aleatória.

2.5.2 Movimento

Cada formiga seleciona um vértice para se deslocar tomando em conta o nível de feromônio deixado pelas outras formigas. Por outro lado, as arestas mais curtas são favorecidas, pois guardam mais feromônios por unidade de distância.

2.5.3 Atualização da trilha de feromônios

A trilha de feromônios saindo de cada vértice é atualizada baseada na qualidade das soluções das formigas que trilharam este caminho. Apenas a formiga que cursou um ciclo hamiltoniano com a menor distância da iteração, tem sua trilha de feromônios salva.

Devido a evaporação dos feromônios, a cada iteração as trilhas salvas perdem sua desejabilidade. Deste modo, soluções mais atuais são mais desejáveis do que soluções mais antigas.

2.5.4 Finalização

O algoritmo encerra se um número de iterações pré-estabelecido é excedido. Ou se houver uma solução satisfatória para o problema. Ou ainda se um tempo de execução máximo é excedido. É válido ressaltar que o número de iterações afeta diretamente o tempo de execução.

3 Trabalhos Relacionados

Buscando compreender melhor o funcionamento dos drones e sua relação com o problema do caixeiro viajante, realizou-se uma pesquisa sobre trabalhos que abordam esta mesma problemática.

Ao considerar o estudo de Raivi et al. (2023), que abordou o desafio dos altos custos das rotas de drones, especialmente em cidades menores, é possível identificar uma similaridade com o cenário do FlyFood. Assim como no problema investigado por esses autores, a necessidade de otimizar o uso das baterias dos drones e encontrar rotas eficientes é um desafio crucial na entrega de alimentos.

O trabalho de Picolo e Silva (2021), que discutiu o PCV no contexto de análise de rotas turísticas, também apresenta uma conexão interessante com o estudo do FlyFood. A escolha do algoritmo do vizinho mais próximo pelos autores, visando fornecer respostas satisfatórias em um tempo razoável, pode ser uma abordagem promissora a ser explorada nesse contexto de roteirização de entrega de alimentos. Essa estratégia, que busca uma solução aproximada em vez de uma solução ótima, pode ser uma alternativa viável para lidar com a complexidade exponencial do PCV e com certeza um bom método comparado ao de força bruta.

Além disso, o trabalho de Bispo (2018) também oferece uma perspectiva valiosa para a nossa pesquisa. A discussão sobre o uso do método de força bruta para resolver o PCV, especialmente para um número limitado de pontos na rota, destaca a importância de considerar diferentes abordagens em relação ao tamanho do problema. Essa análise pode auxiliar na identificação de limitações e quando o método de força bruta se torna vantajoso para resolver o FlyFood.

Os estudos analisados foram de grande importância para compreender o PCV. No entanto, nenhum deles conseguiu garantir uma rota ótima. Para abordar essa questão, foi desenvolvida uma abordagem utilizando o método de força bruta, que será detalhada na seção de metodologia. Com este método, é garantida a rota mais curta para o FlyFood.

4 Metodologia

4.1 Algoritmo de força bruta

Um algoritmo de força bruta é uma técnica de solução de problemas que envolve o cálculo de todas as possibilidades deste problema. Este método sempre encontrará uma solução ótima. Entretanto, se torna cada vez mais difícil com o aumento de elementos, pois o big O FlyFood usado neste algoritmo é $n!$.

4.2 Prova que o PCV é NP-difícil

Para provar que o PCV é NP-difícil é necessário provar que ele é NP. Pois, para ser NP-difícil é suficiente que o problema pertença às classes NP-completo e NP. Considerando esse problema como de decisão e que possa ser representado por um grafo completo. Cormen (2014, p.169) prova que o PCV pertence à NP.

> "Perguntamos se o grafo tem um ciclo que contém todos os vértices cujo peso total é, no máximo, k. É bem fácil mostrar que esse problema está em NP. Um certificado consiste nos vértices do ciclo, em ordem. Podemos facilmente verificar em tempo polinomial se as arestas nesse ciclo visitam todos os vértices e têm peso total de k ou menos".

Agora, se faz necessário provar que o problema é NP-completo. Isto se faz possível quando se reduz a partir do problema do ciclo hamiltoniano que se sabe que é NP completo. Cormen (Cormen,2014, p.169) prova que o problema do caixeiro-viajante pertence à NP-completo.

> "Dado um grafo G como entrada para o problema do ciclo hamiltoniano, construímos um grafo completo G' com os mesmos vértices de G. Igualamos o peso de aresta (u, v) em G' a 0 se. (u, v) é uma aresta em G e o igualamos a 1 se não há nenhuma aresta (u, v) em G. Igualamos k a 0. Essa redução leva tempo polinomial no tamanho de G, visto que acrescenta, no máximo, $n(n-1)$ arestas.
>
> Para mostrar que a redução funciona, precisamos mostrar que G tem um ciclo hamiltoniano se e somente se G' tem um ciclo de peso 0 que inclui todos os vértices. Mais uma vez, o argumento é fácil. Suponha que G tenha um ciclo hamiltoniano. Então cada aresta no ciclo está em G e, assim, cada uma dessas arestas obtém um peso de 0 em G'. Desse modo, G' tem um ciclo que contém todos os vértices, e o peso total desse ciclo é 0. Ao contrário, agora suponha que G' tenha um ciclo que contém todos os vértices e cujo peso total é 0. Então cada aresta neste ciclo deve também estar em G, e portanto G tem um ciclo hamiltoniano".

De maneira mais simples, considerando um grafo $G(V, E)$ que contém uma instância do problema do ciclo hamiltoniano, neste caso o caminho 12345 é hamiltoniano. É possível criar a partir dele uma instância do PCV. Com a adição de um grafo complementar $G'(V, E')$ é possível criar um grafo completo. Como ilustram as figuras 3, 4 e 5.

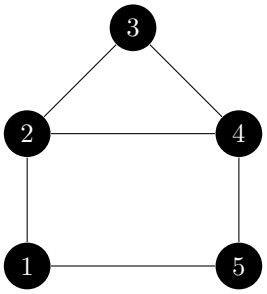

Figura 3: Grafo $G(V, E)$

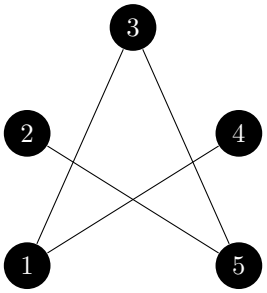

Figura 4: Grafo $G'(V, E')$

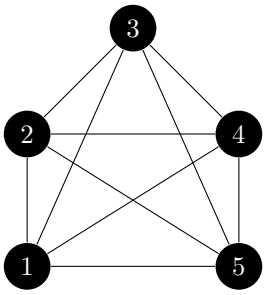

Figura 5: Grafo completo

Então, é necessário definir o custo do caminho deste grafo completo. Seja c a distância entre os vértices i e j. Temos:

- c(i,j) = 0, se i, j pertence as arestas do grafo original(E)
- c(i,j) = 1, se i, j não pertence as arestas do grafo original(E)

Para mostrar que a redução funciona basta que o grafo completo tenha um ciclo hamiltoniano de custo 0. No grafo G, que está contido no grafo completo, há um ciclo hamiltoniano em que todas as arestas pertencem a E. Por outro lado, se no grafo completo há um caminho de custo 0, logo suas arestas pertencem a E.

Portanto, está provado que o PCV é NP completo. Sendo NP e NP-completo, este problema está na classe dos problemas NP-difícil.

4.3 Funcionamento do algoritmo de força bruta

Para a execução deste algoritmo, é necessário, primeiramente, receber os dados de entrada. A entrada é dividada em duas partes o tamanho do mapa da cidade e o mapa propriamente dito. Para lê-los serão criadas duas funções, *adiquirir_tamanho_mapa* e *adiquirir_vertices*, as quais são descritas a seguir.

Algoritmo 1: adiquirir_tamanho_mapa

```
Função adiquirir_tamanho_mapa(separador):
    leia(entrada)
    entrada_dividida ←
     entrada.dividir(separador)
    para i de 1 até 2 faça
        se entrada_dividida[i] ≤ 0 então
            levantar ErroDeValor("O tamanho
             do mapa deve ser positivo.")
        fim
        tamanho_mapa[i] ← entrada_dividida[i]
    fim
    retorna tamanho_mapa
fim
```

A função*adiquirir_tamanho_mapa* lê a entrada do usuário, separa o tamanho horizontal e vertical em um vetor de tamanho 2, em seguida verifica se ambos os valores são positivos. Caso algum valor seja negativo ou nulo, a função gera uma exceção. Esta função é utilizada para obter as dimensões corretas do mapa antes de realizar outras operações relacionadas.

Algoritmo 2: esta_contido

```
Função esta_contido(sequencia, alvo):
    para cada i em sequencia faça
        se i == alvo então
            retorne Verdadeiro
        fim
    fim
    retorna Falso
fim
```

A função *esta_contido* é projetada para verificar se um determinado elemento, denominado alvo, está presente em uma sequência de elementos, representada por um vetor. O pseudocódigo apresenta uma estrutura simples, mas eficiente, para realizar essa verificação. O algoritmo inicia percorrendo cada elemento da sequência, utilizando um laço "para cada". A cada iteração, o algoritmo verifica se o elemento atual é igual ao alvo. Se essa condição for verdadeira, a função retorna Verdadeiro, indicando que o alvo está contido na sequência. Caso contrário, o algoritmo continua a percorrer os elementos da sequência. Se nenhum elemento for igual ao alvo, o algoritmo chega ao final do laço "para cada"e retorna Falso, indicando que o alvo não está presente na sequência. Essa abordagem permite uma verificação rápida e eficiente da existência do alvo na sequência, e é especialmente útil quando se trata de vetores grandes.

Algoritmo 3: adiquirir_vertices

```
Função adiquirir_vertices(tamanho_mapa,
 separador):
    para i de 1 até tamanho_mapa[1] faça
        leia(entrada)
        linha ← en-
         trada.dividir(separador)[:tamanho_mapa[2]]
        se não linha ou
         |linha| < tamanho_mapa[2] então
            levante ErroDeValor("O tamanho
             real do mapa não corresponde ao
             tamanho fornecido.")
        fim
        para j de 1 até tamanho_mapa[2] faça
            vertice ← linha[j]
            se vertice for alfabético então
                se não esta_contido(chaves de
                 vertices, vertice) então
                    vertices[vertice] ← (i, j)
                senão
                    levante
                     ErroDeValor("Vértices
                     duplicados!")
                fim
            fim
        fim
    fim
    retorna vertices
fim
```

Por sua vez, a função *adiquirir_vertices* recebe os dados restantes do input, que representam o mapa da cidade, onde cada linha representa uma unidade vertical e cada espaço, uma unidade horizontal. Primeiramente é verificado se o tamanho real do mapa corresponde ao tamanho fornecido como parâmetro, se não é levantada uma exceção se sim verifica-se se a célula atual é alfabética, ou seja, se corresponde ao nome de um vértice, se sim, é realizada outra verificação para determinar se já existe um vertice de mesmo nome, senão, o vertice atual é adicionado a lista juntamente com suas coordenadas. Agora se faz nescessário criar uma classe Grafo, a qual irá representar o mapa da cidade no programa. Esta classe terá 4 métodos, os quais são descritos a seguir:

Algoritmo 4: Grafo.novo

```
Função novo(isso, vertices):
    isso.grafo ← isso.gerar_grafo(vertices)
fim
```

O primeiro deles é o método construtor da classe o qual apenas irá atribuir o valor da propriedade *grafo* para o retorno da função *gerar_grafo*, a qual sera explicada logo em seguida:

Esta função itera por cada um dos vértices e calcula a distâcia do vértice atual para todos os restantes,

Algoritmo 5: Grafo.gerar_grafo

```
Função gerar_grafo(isso, vertices):
    distancias ← {}
    para cada i em chaves de vertices faça
        para cada j em chaves de vertices faça
            se i ≠ j então
                distancias[i][j] ←
                 módulo(vertices[i][0] -
                 vertices[j][0]) +
                 módulo(vertices[i][1] -
                 vertices[j][1])
            fim
        fim
    fim
    retorna distancias
fim
```

gerando como resultado um dicionário com todas as distâncias entre todos os vértices.

Algoritmo 6: Grafo.adiquirir_vertices

```
Função adiquirir_vertices(isso):
    retorna chaves de isso.grafo
fim
```

Algoritmo 7: Grafo.adiquirir_distancia

```
Função adiquirir_distancia(isso, origem,
 destino):
    retorna isso.grafo[origem][destino]
fim
```

As duas funções anteriores apenas retornam a lista de nomes dos vértices e a distância entre dois vértices respectivamente. Estas funções servem de base para a função que será descrita logo em seguida.

Algoritmo 8: Grafo.calcular_custo

```
Função calcular_custo(isso, rota):
    custo ← 0
    para i de 1 até |rota| − 1 faça
        vertice_atual ← rota[i]
        proximo_vertice ← rota[i + 1]
        custo ← custo +
         isso.adiquirir_distancia(vertice_atual,
         proximo_vertice)
    fim
    retorna custo
fim
```

A função *calcular_custo* é a pricipal função da classe *Grafo*. Por meio dela calcula-se a distância percorrida em uma sequencia de vérices. O método funciona da seguinte maneira, utilizando a "tabela"de distâncias gerada pela função *gerar_grafo* o algoritmo soma a distancia entre um determinado vértice da sequência e o seu sucessor, após percorrer por todos os vértices o valor total da distância é retornado. Com isso, conclui-se a classe *Grafo*

Algoritmo 9: gerar_permutacoes

```
Função gerar_permutacoes(vertices,
 profundidade):
    se |vertices| == 1 então
        retorna [vertices]
    fim
    result ← []
    para i de 1 até |vertices| faça
        vertice_atual ← vertices[i]
        vertices_restantes ← vertices[:i] +
         vertices[i + 1:]
        permutacoes_restantes ←
         gerar_permutacoes(vertices_restantes,
         profundidade + 1)
        para cada permutacao em
         permutacoes_restantes faça
            nova_permutacao ← [vertice_atual] +
             permutacao
            se profundidade ≠ 1 ou não
             esta_contido(result,
             nova_permutacao[::-1]) então
                result.adicionar(nova_permutacao)
            fim
        fim
    fim
    retorna result
fim
```

Esta é a última função desenvolvida para este algoritmo e uma das principais. Utilizando recursividade a função *gerar_permutacoes* gera uma lista com todas as permutações possíveis dado um determinado conjunto de vérices. Por fim temos o algoritmo principal do programa:

O pseudocódigo apresenta um algoritmo para encontrar o menor caminho em um grafo, com exceção de um ponto de partida chamado "R". Ele começa adquirindo os vértices do grafo e verificando se "R"está presente. Caso contrário, é lançada uma exceção. Em seguida, são inicializadas variáveis para armazenar a menor distância e o melhor caminho. Os vértices são permutados e, para cada permutação, é calculada a distância total. Se a distância for menor do que a menor distância atual, os valores são atualizados. Ao final, o algoritmo exibe a menor distância encontrada e o melhor caminho.

O algoritmo utiliza uma abordagem de força bruta para resolver o problema. Ele testa todas as possíveis permutações dos vértices restantes, excluindo o ponto de partida "R". Calcula a distância total para cada

Algoritmo 10: Algoritmo Principal

```
grafo ←
 adiquirir_vertices(adiquirir_tamanho_mapa(), )
vertices ← grafo.adiquirir_vertices()
se não esta_contido(vertices, "R") então
    levantar Excecao("Ponto de retorno não
     encontrado.")
fim
menor_distancia ← +∞
melhor_caminho ← None
vertices.remover("R")
permutacoes ← gerar_permutacoes(vertices, 1)
para i in 1 até —permutacoes— faça
    caminho ← ["R"] + permutations[i] + ["R"]
    distancia ← grafo.calcular_custo(caminho)
    se distancia ¡ min_distance então
        min_distance ← menor_distancia
        melhor_caminho ← permutations[i]
    fim
fim
Escrever("Menor distância:", min_distance)
Escrever("Melhor caminho: ")
para i de 1 até —melhor_caminho— faça
    Escrever(melhor_caminho[i], )
fim
Escrever("\n")
```

permutação e compara com a menor distância encontrada até o momento. Isso garante que o algoritmo encontre o menor caminho possível dentro do grafo.

4.3.1 Instância de execução do algoritmo de força bruta

4.3.1.1 Exemplo de entrada

Nesta seção, o exemplo da entrada da figura 6 será usado para estudar o funcionamento do programa RouteXplorer que resolve o PCV. Nesta entrada, cada algarismo '0' representa uma unidade de medida, as letras maiúsculas representam os vértices e o ponto 'R' é o ponto de partida do percurso.

```
4 5
0 0 0 0 D
0 A 0 0 0
0 0 0 0 C
R 0 B 0 0
```

Figura 6: Matriz de entrada

4.3.1.2 Funcionamento do algoritmo

Em primeiro lugar, o RouteXplorer lê um vetor de tamanho dois, neste caso a tupla '(4, 5)' . Essa tupla gera uma matriz de 4 linhas e 5 colunas. Em seguida, verifica uma lista para ver se um argumento 'alvo' está presente. Este algoritmo recebe os dados da matriz e retorna uma lista de vértices.

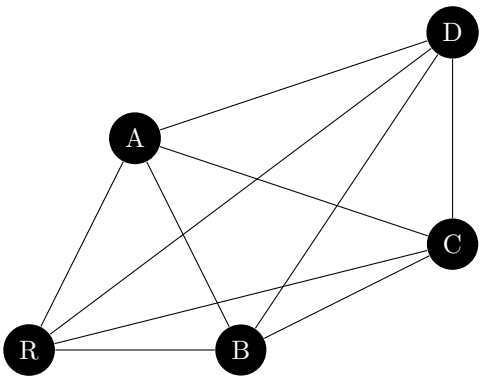

Figura 7: Grafo gerado a partir da entrada

Então, é criada uma classe abstrata Grafo com alguns métodos internos, tais como o método gerar_grafo , o metódo adquirir vértices e o método calcular_custo. Estas funções criam o grafo (figura 7) e calculam o comprimento das arestas. Como ilustrado na tabela 2.

Tabela 2: Comprimento das arestas

	R	**A**	**B**	**C**	**D**
R	-	3	2	5	7
A	3	-	3	4	4
B	2	3	-	3	5
C	5	4	3	-	2
D	7	4	5	2	-

Em seguida é calculada a distância de cada vértice como na tabela 2. É calculada a distância percorrida em uma sequência de vértices. E retorna o valor total. Por meio da recursividade, é gerada todas as permutações do percurso no algoritmo *gerar_permutações*. Por fim, *algoritmo_principal* retorna os vértices do menor caminhos e a menor distância percorrida.

4.4 Análise do Custo do Algoritmo

Para obter uma compreensão mais profunda do funcionamento do algoritmo, será demonstrado como o tempo de execução do código varia de acordo com a entrada. Será feito uma análise detalhada das permutações, linha por linha, considerando (n) como a variável que representa o tamanho da entrada e variáveis como $(c1, c2, c3\ldots)$ representando o custo constante de cada linha. Com esta análise, é possível entender melhor a eficiência do algoritmo em diferentes cenários e a sua escalabilidade para tamanhos maiores de entrada.

Ao chamar a função *gerar_permutacoes*, observa-se que até a linha c3, cada comando é executado apenas uma vez, portanto, o custo dessas linhas segue o padrão c1 + c2 + c3.

Na segunda parte do algoritmo, há um loop "para"que percorre todos os elementos contidos em "vértices"(n). Dessa forma, o custo de cada linha dentro desse loop se repete n vezes, exceto pela linha c7, que trata da recursão do algoritmo, que resulta em um custo de n!.

Algoritmo 11: gerar_permutacoes

```
Função gerar_permutacoes(vertices,
 profundidade):
    se |vertices| == 1 então                    #C1
        retorna [vertices]                      #C2
    fim
    result ← []                                 #C3
    para i de 1 até |vertices| faça             #C4 × n
        vertice_atual ← vertices[i]             #C5 × n
        vertices_restantes ← vertices[:i] +
         vertices[i + 1:]                       #C6 × n
        permutacoes_restantes ←
         gerar_permutacoes(vertices_restantes,
         profundidade + 1)                      #C7 × n!
        para cada permutacao em
         permutacoes_restantes faça             #C8 × n!
            nova_permutacao ← [vertice_atual] +
             permutacao                         #C9 × n!
            se profundidade ≠ 1 ou não
             esta_contido(result,
             nova_permutacao[::-1]) então
             #C10 × n!
                result.adicionar(nova_permutacao)
                 #C11 × n!
            fim
        fim
    fim
    retorna result                              #C12
fim
```

Na terceira parte do algoritmo, todos os comandos dentro do loop são repetidos $|permutacoes\_restantes|$ vezes, ou seja,$n!$.

Ao finalizar a função, ela retorna o resultado, que acontece apenas uma vez. Com isso, o tempo de execução do algoritmo é expresso por meio da função T(n) a seguir:

$$\begin{aligned} T(n) &= n!(c7 + c8 + c9 + c10 + c11) \\ &+ n(c4 + c5 + c6) \\ &+ c1 + c2 + c3 + c12 \end{aligned} \tag{8}$$

Analisando a complexidade de tempo, o termo dominante é n!, que representa uma complexidade exponencial. Portanto, a notação Grande O do algoritmo é O(n!). Esta complexidade exponencial indica que o tempo de execução cresce de forma significativa com o tamanho da entrada.

4.5 Funcionamento e instância de execução do Algoritmo Genético

Tome como exemplo o grafo da figura 6 e 7, nela consta uma instância do PCV. Onde R é o ponto de retorno e os demais pontos representam um ponto de entrega. Cada quadrado representa o deslocamento de uma unidade. Para a solução usando o AG se faz necessário a criação de uma população inicial.

4.5.1 Criação da população inicial

Cada indivíduo é gerado sorteando um vértice da lista de vértices que ainda não foram visitados, excetuando o vértice de retorno. O processo é repetido até que não haja mais vértices a serem visitados.

A criação de indivíduos se repete até que a população inicial esteja formada. O número de indivíduos iniciais deve ser escolhido com cuidado. Uma vez que se forem muitos o algoritmo torna-se lento e se forem poucos a diversidade fica prejudicada.

Para fins de simplicidade, serão usados quatro indivíduos na população inicial. São eles os caminhos:ABCD, BACD, BADC, CADB. Esta etapa não se repetirá mais. As outras etapas se repetirão por um número determinado de gerações.

Algoritmo 12: geraçãoInicial

```
Função geraçãoInicial(pontos,
 tamanhoPopulação):
    primeiraGeração ← []
    pontos ← chaves de pontos
    pontos ← pontos[1..|pontos|]
    para i de 0 até tamanhoPopulação faça
        pontosAleatórios ← copia(pontos)
        embaralhar(pontosAleatórios)
        novoElemento ← ['1'] + pontosAleatórios
         + ['1'] primeira-
         Geração.adicionar(novoElemento)
    fim
    retorna primeiraGeração
fim
```

Esta função é responsável por criar a primeira geração de indivíduos para o AG que resolverá o PCV . Esta função recebe como entrada os pontos que representam as cidades a serem visitadas e o tamanho desejado para a população inicial.

Primeiro, a função inicializa uma lista vazia chamada "primeiraGeração" para armazenar os indivíduos gerados. Em seguida, são processados os pontos de entrada. A variável "pontos" é inicialmente criada como uma lista das chaves dos pontos, o que essencialmente a transforma em uma lista contendo os índices das cidades.

Após essa etapa, a função remove o primeiro ponto da lista de pontos, uma vez que é comum que o primeiro ponto seja considerado o ponto de partida e chegada para o PCV, e ele não precisa ser incluído novamente no final da rota.

O algoritmo entra em um loop que se repete do índice 0 até o tamanho da população desejada. Dentro desse loop, uma lista chamada "pontosAleatórios" é criada como uma cópia da lista de pontos. A função "embaralhar" é usada para aleatorizar a ordem dos

pontos na lista "pontosAleatórios", o que resulta em uma ordem aleatória de visita das cidades.

Em seguida, um novo elemento (indivíduo) é construído. Esse novo elemento começa com o primeiro ponto (geralmente identificado como "1"), seguido da lista de pontos aleatoriamente embaralhados e, por fim, o ponto de chegada, que também é "1". Esse novo elemento representa uma rota que inicia e termina na mesma cidade, visitando as demais em uma ordem aleatória.

Após a construção do novo elemento, ele é adicionado à lista "primeiraGeração", que armazena a primeira geração de indivíduos.

O algoritmo continua a gerar indivíduos até que a primeira geração alcance o tamanho desejado. Em seguida, a lista "primeiraGeração", que contém a primeira geração de indivíduos, é retornada como resultado da função.

4.5.2 Avaliação dos indivíduos

Nesta fase é calculada a aptidão. Como o PCV é um problema de minimização, quanto menor o custo dos caminhos maior a aptidão do indivíduo. As aptidões constam na tabela a seguir:

Tabela 3: Aptidão dos indivíduos

Vértices	**ABCD**	**BACD**	**BADC**	**CADB**
Custo	18	18	16	20

Neste exemplo, o indivíduo BADC pode ser considerado mais apto que CABD, pois seu custo é menor. A aptidão afeta a chance de um indivíduo passar suas características para frente. A escolha de quais indivíduos deixarão descendentes estará na fase de seleção. O cálculo da aptidão é feito da seguinte forma:

Algoritmo 13: calcularAptidao

```
Função calcularAptidao(rota, coordenadas):
    aptidão ← 0
    para índice de 0 até |rota| − 1 faça
        aptidão ← aptidão +
          calcularDistancia(rota[indice],
          rota[indice+1], coordenadas)
    fim
    retorna aptidão
fim
```

A função "calcularAptidao" tem como objetivo calcular a aptidão de uma dada rota em um problema de otimização.

Essa função recebe dois parâmetros: "rota" e "coordenadas." A "rota" representa uma sequência de índices que indicam a ordem em que os pontos devem ser visitados, e "coordenadas"contém as coordenadas (geralmente em um espaço bidimensional) dos pontos a serem visitados.

A variável "aptidão" é inicializada com zero, representando a aptidão inicial da rota. A função, então, entra em um loop que itera do índice 0 até o comprimento da rota menos um, o que significa que percorrerá cada par de pontos consecutivos na rota.

Dentro desse loop, a função chama outra função chamada "calcularDistancia" para calcular a distância entre os pontos representados pelos índices "rota[indice]" e "rota[indice+1]" usando as coordenadas fornecidas. Essa distância é somada à variável "aptidão."

O loop continua até que todas as distâncias entre os pontos consecutivos na rota tenham sido somadas à "aptidão."

Finalmente, a função retorna o valor da "aptidão", que representa a qualidade da rota em termos de distância total percorrida.

Algoritmo 14: calcularDistância

```
Função calcularDistância(primeiroVertice,
  segundoVertice, coordenadas):
    primeiroVerticeX, primeiroVerticeY ←
      coordenadas[primeiroVertice]
    segundoVerticeX, segundoVerticeY ←
      coordenadas[segundoVertice]
    distância ← ((primeiroVerticeX −
      segundoVerticeX)^2 + (primeiroVerticeY −
      segundoVerticeY)^2)^(1/2)
    retorna distância
fim
```

A função "calcularDistância" tem como finalidade calcular a distância Euclidiana entre dois vértices em um espaço bidimensional.

Essa função recebe três parâmetros: "primeiroVertice," "segundoVertice," e "coordenadas." Os parâmetros "primeiroVertice" e "segundoVertice" representam os índices dos dois vértices entre os quais a distância deve ser calculada, enquanto "coordenadas" contém as coordenadas de todos os vértices do problema.

A função começa descompactando as coordenadas dos vértices referentes aos índices "primeiroVertice" e "segundoVertice" em suas componentes X e Y. Isso é feito atribuindo os valores correspondentes das coordenadas a variáveis separadas: "primeiroVerticeX", "primeiroVerticeY", "segundoVerticeX", e "segundoVerticeY".

Em seguida, a função calcula a distância Euclidiana entre os dois vértices usando a fórmula matemática padrão de distância entre dois pontos em um plano cartesiano. A fórmula consiste na raiz quadrada da soma dos quadrados das diferenças das coordenadas X e Y dos dois vértices.

Por fim, a função retorna o valor calculado da "distância,"que representa a distância linear entre os dois vértices especificados.

4.5.3 Seleção dos indivíduos mais aptos

Há mais de uma forma de seleção. Neste caso, será usada a seleção-torneio. Este tipo de seleção, além de

ser simples de explicar, não fica tão presa a mínimos locais quanto a seleção roleta, por exemplo.

Nela, todos os indivíduos são sorteados aos pares para uma batalha. Quem vence a batalha é quem tem a aptidão maior. No caso de empate o primeiro competidor será escolhido. Sorteando os indivíduos, teremos as seguintes duas batalhas:

Tabela 4: Sorteio de Indivíduos

Competidor 1	Competidor 2	Vencedor
ABCD	**BACD**	**ABCD**
BADC	**CADB**	**BADC**

Uma disputa em um torneio ocorre conforme o seguinte algoritmo:

Algoritmo 15: ganhadorTorneio

```
Função ganhadorTorneio(população,
 coordenadas):
    primeiroCompetidor ←
     escolher_aleatoriamete(população)
    segundoCompetidor ←
     escolher_aleatoriamete(população)
    se calcularAptidão(primeiroCompetidor,
     coordenadas) <
     calcularAptidão(segundoCompetidor,
     coordenadas) então
        retorna primeiroCompetidor
    fim
    senão
        retorna segundoCompetidor
    fim
fim
```

A função "ganhadorTorneio" implementa um processo de seleção tipo torneio.

Essa função recebe dois parâmetros: "população" e "coordenadas". A "população" é uma lista que contém indivíduos ou rotas candidatas, representados geralmente por sequências de índices de vértices, enquanto "coordenadas" armazena as coordenadas (tipicamente em um espaço bidimensional) dos pontos a serem visitados.

O funcionamento da função começa com a seleção aleatória de dois competidores da população usando a função "escolherAleatório". Esses competidores são representados pelas variáveis "primeiroCompetidor" e "segundoCompetidor".

Em seguida, a função calcula a aptidão dos dois competidores, chamando a função "calcularAptidão" para cada um deles, passando as "coordenadas" como parâmetro. A aptidão é uma medida da qualidade da rota, geralmente representando a distância total percorrida no contexto do PCV.).

A função compara as aptidões dos dois competidores. Se o "primeiroCompetidor" tiver uma aptidão menor (ou seja, uma rota mais curta) do que o "segundoCompetidor", o "primeiroCompetidor" é considerado o vencedor do torneio, e a função retorna o "primeiroCompetidor"como o ganhador. Caso contrário, se o "segundoCompetidor" tiver uma aptidão menor, ele é considerado o ganhador e é retornado.

Os vencedores estão aptos a se reproduzir entre si. A reprodução é o tema da próxima sessão e é a etapa responsável pela geração de novos indivíduos. Nesta etapa é necessário que seja mantida a diversidade. A diversidade é importante, pois evita que o algoritmo fique preso a um mínimo local. Dessa maneira o algoritmo deixa de explorar o resto do espaço de busca e a solução ótima se torna muito mais difícil.

4.5.4 Reprodução dos indivíduos mais aptos

Depois da etapa de seleção vem a parte de reprodução propriamente dita. Vale ressaltar que, assim como na natureza, nem toda cópula resulta em novos indivíduos (esta chance é chamada de taxa de fecundidade). Mas a chance geralmente é alta. O cruzamento dos indivíduos segue no seguinte algoritmo:

Algoritmo 16: cruzamento

```
Função cruzamento(primeiroIndividuo,
 segundoIndividuo):
    separador ← inteiro_aleatorio(2,
     |primeiroIndividuo|)
    primeiroIndividuo, segundoIndividuo ←
     PMX(primeiroIndividuo,
     segundoIndividuo, separador)
    primeiroFilho ←
     primeiroIndividuo[1..separador] +
     segundoIndividuo[separador..final]
    segundoFilho ←
     segundoIndividuo[1..separador] +
     primeiroIndividuo[separador..final]
    retorna (primeiroFilho, segundoFilho)
fim
```

A função "cruzamento" implementa um operador de cruzamento (crossover) utilizado em algoritmos genéticos para combinar dois indivíduos (ou cromossomos) e produzir dois novos indivíduos, muitas vezes chamados de filhos. Abaixo, segue uma explicação em formato de parágrafo:

Essa função recebe dois parâmetros: "primeiroIndivíduo" e "segundoIndivíduo." os quais representam soluções candidatas em um espaço de busca.

O processo de cruzamento começa com a escolha aleatória de um "separador" entre 2 e o tamanho do "primeiroIndivíduo."O objetivo desse separador é dividir os dois indivíduos em duas partes que serão trocadas para criar novos indivíduos.

Supondo que a cópula tenha sido bem sucedida. Para este caso será usado o algoritmo Partially-Mapped Crossover (PMX). Neste algoritmo é sorteado um ponto de Crossover. Este ponto dita a porção de cada indivíduo que será mantido na sua prole.

No PMX, para cada pai, todos os vértices antes do ponto de Crossover são mantidos e incorporados ao filho. E os vértices restantes são retirados do pai restante de modo a evitar repetição. Para o exemplo

abaixo, mantendo os vértices iniciais do primeiro pai (ABCD) , será retirado o ponto 'A'. E incorporada o resto da sequência sem o 'A'. Com isso será gerado o filho ABDC.

Algoritmo 17: PMX

```
Função PMX(primeiroIndividuo,
  segundoIndividuo, separador):
    primeiroIndividuo ← primeiroIndivi-
      duo[2..(|primeiroIndividuo| − 1)]
    segundoIndividuo ←
      segundoIndividuo[2..(|segundoIndividuo| −
      1)]
    para cada indicePrincipal, ponto_i em
      primeiroIndividuo[até separador] faça
        para cada indiceSecundário, ponto_j em
          segundoIndividuo faça
            se ponto_i = ponto_j então
              temp ← segundoIndivi-
                duo[indicePrincipal]
              segundoIndividuo[indicePrincipal]
                ← segundoIndivi-
                duo[indiceSecundário]
              segundoIndividuo[indiceSecundário]
                ← temp
            fim
        fim
    fim
    primeiroIndividuo ← ['1'] +
      primeiroIndividuo + ['1']
    segundoIndividuo ← ['1'] +
      segundoIndividuo + ['1']
    retorna (primeiroIndividuo,
      segundoIndividuo)
fim
```

A função "PMX" (Partially-Mapped Crossover) implementa o operador de cruzamento PMX.

A função "PMX" recebe três parâmetros: "primeiroIndivíduo", "segundoIndivíduo", e "separador". Esses indivíduos representam possíveis soluções no espaço de busca, enquanto o "separador" é um índice que determina onde os cromossomos dos pais serão divididos para criar os filhos.

Primeiro, a função remove o primeiro e o último ponto dos cromossomos dos pais, pois esses pontos são geralmente fixos em problemas como o Problema do Caixeiro Viajante (TSP) e representam o ponto de partida e de retorno.

Em seguida, a função inicia dois loops aninhados. O primeiro loop, com o índice "indicePrincipal" e a variável "ponto i", itera sobre os pontos no cromossomo do "primeiroIndivíduo" que estão à esquerda do "separador". O segundo loop, com o índice "indiceSecundário" e a variável "ponto j", itera sobre todos os pontos no cromossomo do "segundoIndivíduo".

Dentro desses loops, a função verifica se o "ponto i" do "primeiroIndivíduo" é igual ao "ponto j" do "segundoIndivíduo". Quando uma correspondência é encontrada, indica-se que houve uma troca de informações entre os pais. Portanto, a função realiza uma troca nos cromossomos dos filhos para manter a consistência das informações herdadas.

Após a conclusão dos loops, os cromossomos dos filhos são modificados e atualizados de acordo com as trocas realizadas. Ambos os cromossomos são agora complementados com o primeiro e o último ponto, representando o ponto de partida e de retorno, garantindo que a solução resultante seja completa e viável.

Por fim, a função retorna uma tupla que contém os cromossomos dos dois filhos gerados após a aplicação do operador PMX

Após a execução do "PMX", os indivíduos resultantes são divididos em duas partes: "primeiroFilho" e "segundoFilho". O "primeiroFilho" consiste na parte do "primeiroIndivíduo" que vai do início até o "separador", concatenada com a parte do "segundoIndivíduo" que vai do "separador" até o final. O "segundoFilho" é construído da maneira oposta.

Finalmente, a função retorna uma tupla contendo os dois filhos gerados, "primeiroFilho"e "segundoFilho."Esses filhos resultantes do cruzamento são então adicionados à população para dar continuidade ao processo de evolução do algoritmo genético, com a esperança de que eles herdem boas características dos pais e contribuam para uma melhoria nas soluções ao longo das gerações.

No final desse processo para cada indivíduo-pai será criado um indivíduo-filho. Neste problema será usado a técnica de substituição geracional na qual cada indivíduo-filho substitui o seu pai. Veja o exemplo do PMX abaixo (o pai com o ponto de Crossover fixo fica em vermelho):

Tabela 5: Técnica de substituição geracional

1ª Filho	2ª Filho
A|B C D – 1º Pai	A|B C D – 1º Pai
B|A D C – 2º Pai	B|A D C – 2º Pai
A B D C – 1º Fil.	B A C D – 2º Fil.

Portanto, devido a substituição geracional. Teremos a população formada pelos indivíduos ABDC, BACD,CADB, BACD.

4.5.5 Mutação aleatória

No entanto, assim como na vida real. Ocorrem erros na formação de indivíduos, estes erros naturalmente são raros, logo a taxa de mutação é baixa. No entanto, mutações não devem produzir soluções inválidas para o PCV como ABAD (na qual um vértice se repete duas vezes).

Neste caso, quando a mutação se dá. O vértice em que a mutação ocorreu é trocado de lugar com o próximo vértice. Supondo que os dois indivíduos-filhos sofreram mutação na posição 2.

A função "mutacao" implementa o operador de mutação em um algoritmo genético.

A função "mutacao" recebe dois parâmetros: "individuo" e "taxaMutacao". O "individuo" é o cro-

Tabela 6: Mutação de Indivíduos

Filho	Antes	Mutação	Depois
1	ABDC	–	ADBC
2	BACD	–	BCAD

Algoritmo 18: mutação

```
Função mutacao(individuo, taxaMutacao):
    para indice de 1 até |individuo| - 2 faça
        se inteiro_aleatorio(1, 100) ≤
         taxaMutacao então
            temp ← individuo[indice]
            individuo[indice] ← individuo[indice
             + 1]
        fim
    fim
    retorna individuo
fim
```

mossomo que será sujeito à mutação, e a "taxaMutacao"determina a probabilidade de ocorrer uma mutação em cada posição do cromossomo.

Em um loop que percorre todas as posições do cromossomo, exceto a primeira e a última (uma vez que essas posições são geralmente fixas em problemas como o Problema do Caixeiro Viajante - TSP), a função realiza o seguinte procedimento:

Gera um número aleatório entre 1 e 100. Isso representa uma chance percentual de ocorrer uma mutação na posição atual do cromossomo.

Compara o número gerado aleatoriamente com a "taxaMutacao." Se o número for menor ou igual à taxa de mutação, uma mutação será realizada nessa posição.

Se a mutação for acionada, a função realiza uma troca entre o elemento na posição atual do cromossomo (índice "indice") e o elemento na posição seguinte (índice "indice + 1"). Essa troca é uma forma simples de realizar a mutação.

O loop continua a percorrer todas as posições do cromossomo, verificando a probabilidade de mutação em cada uma delas.

Após o término do loop, a função retorna o cromossomo modificado, que pode ou não ter sofrido mutações em suas posições, dependendo da taxa de mutação definida.

4.5.6 Algoritmo Genético

Com todos os passos definidos pode-se montar o algoritmo principal do algoritmo genético. representado pela função encontrar_melhor_rota.

Essa função recebe como entrada os pontos que representam as cidades a serem visitadas.

A função começa definindo algumas variáveis importantes para o algoritmo, como o número da geração inicial ("geração"), o tamanho da população inicial ("tamanhoPopulação"), o número máximo de gerações a serem executadas ("máximoGerações"), a taxa de cruzamento ("taxaCruzamento"), e a taxa de mutação

Algoritmo 19: Algoritmo Genético

```
Função encontrar_melhor_rota(pontos):
    geração ← 1
    tamanhoPopulação ← 30
    máximoGerações ← 1500
    taxaCruzamento ← 95
    taxaMutação ← 2
    menorCustoRota ← infinito
    melhorRota ← lista vazia
    população ← geraçãoInicial(pontos,
     tamanhoPopulaçãp)
    enquanto geração < maxGerações faça
        população ←
         criarGeração(população,
         taxaCruzamento, taxaMutação,
         pontos)
        para individuo em população faça
            aptidãoIndividuo ←
             calcularAptidão(individuo,
             pontos)
            se aptidãoIndividuo <
             menorCustoRota então
                menorCustoRota ←
                 aptidãoIndividuo
                melhorRota ← individuo
            fim
        fim
        geração ← geração + 1
    fim
    retorna melhorRota, menorCustoRota
fim
```

("taxaMutação"). Além disso, a função inicializa as variáveis "menorCustoRota"e "melhorRota"com valores iniciais adequados. "menorCustoRota" é definida como infinito, e "melhorRota" como uma lista vazia.

A próxima etapa do algoritmo é criar a primeira geração de indivíduos. Isso é feito chamando a função "geraçãoInicial", que gera indivíduos aleatórios para iniciar o processo. A população inicial é armazenada na variável "população".

Em seguida, o algoritmo entra em um loop que se repete até atingir o número máximo de gerações definido. Dentro desse loop, ele chama a função "criarGeração" para gerar uma nova geração de indivíduos a partir da população atual. Essa função utiliza as taxas de cruzamento e mutação para criar novos indivíduos.

Após a criação da nova geração, o algoritmo avalia cada indivíduo da população atual calculando sua aptidão (custo da rota) usando a função "calcularAptidão". Se a aptidão do indivíduo atual for menor do que o "menorCustoRota" encontrado até o momento, ele atualiza o "menorCustoRota" com o novo valor e armazena a rota desse indivíduo como a "melhorRota"encontrada até agora.

O loop continua a ser executado até que o número máximo de gerações seja alcançado. Após o término do loop, o algoritmo retorna a "melhorRota" encontrada e o seu custo, representado por "menorCustoRota".

4.6 Funcionamento do Algoritmo ACO

4.6.0.1 Funções Auxiliares

Antes de entender o algoritmo da otimização da colônia de formigas se faz necessário definir algumas funções que serão essenciais para o desenvolvimento do algoritmo.

Algoritmo 20: ler_arquivo_tsp

```
Função ler_arquivo_tsp(caminho):
    assegure que arquivo em caminho existe
     senão exiba caminho + " - arquivo não
     existe."
    abrir arquivo em caminho para leitura
     como arquivo:
        informações ← {}
        para cada linha em arquivo faça
            se linha =
              "NODE_COORD_SECTION"
             então
                parar
            fim
            linha_dividida ← linha.dividir(":")
             informações[linha_dividida[1]] ←
             linha_dividida[2]
        fim
        assegure que in-
         formações["EDGE_WEIGHT_TYPE"]
         = "EUC_2D" senão exiba "tipo de
         distância não suportado: " + in-
         formações["EDGE_WEIGHT_TYPE"]
        n ←
         Inteiro(informações["DIMENSION"])
        vertices ← {}
        para i de 1 até n faça
            v ← f.lerlinha().dividir(" ")
            vertices[v[1]] ← (Real(v[2]),
             Real(v[3]))
        fim
    fim
    retorna Grafo(vertices)
fim
```

A função chamada "ler_arquivo_tsp" recebe como entrada o caminho do arquivo TSP. Primeiro, ele verifica se o arquivo existe; caso contrário, exibe uma mensagem de erro. Em seguida, o algoritmo abre o arquivo para leitura e inicia um loop que percorre cada linha do arquivo TSP. Ele procura por uma linha especial que marca o início da seção de coordenadas dos nós, identificada como "NODE COORD SECTION". Uma vez encontrada essa marca, o loop é interrompido. Durante a leitura das linhas subsequentes, as informações são divididas com base em ":" e armazenadas em um dicionário chamado "informações". O código assegura que o tipo de distância seja "EUC 2D,"caso contrário, exibe uma mensagem de erro. Em seguida, ele obtém o número de dimensões (nós) do grafo e inicia a leitura das coordenadas dos nós. As coordenadas são armazenadas em um dicionário chamado "vertices"com o índice do nó como chave e as coordenadas (x, y) como valor. Finalmente, o código retorna um grafo construído com base nas coordenadas lidas do arquivo TSP.

Algoritmo 21: obter_máximo

```
Função obter_máximo(sequencia, chave ←
 lambda(x){x}):
    indice_maximo ← 0
    para i de 1 até |sequencia| faça
        se chave(sequencia[i]) >
         chave(sequencia[indice_maximo]) então
            indice_maximo ← i
        fim
    fim
    retorna sequencia[indice_maximo]
fim
```

A função "obter_máximo" tem como objetivo encontrar o elemento máximo em uma sequência. Ela recebe dois parâmetros: a sequência de elementos que deseja-se analisar e uma função de chave opcional que determina como os elementos são comparados. Por padrão, a função de chave é uma função lambda que retorna o próprio elemento, o que significa que a comparação é baseada nos valores dos elementos. A função inicia inicializando uma variável chamada "Índice máximo" com 0. Em seguida, um loop percorre a sequência, comparando cada elemento com o atual máximo com base nos critérios estabelecidos pela função de chave. Se um elemento for considerado maior, o índice do máximo é atualizado. Ao final do loop, a função retorna o elemento na sequência que possui o maior valor resultante da aplicação da função de chave, tornando-a flexível para encontrar o máximo com base em critérios específicos.

Algoritmo 22: obter_maior

```
Função obter_maior(a, b):
    se b > a então
        retorna b
    fim
    retorna a
fim
```

A função chamada "obter_maior" aceita dois argumentos, "a" e "b". O objetivo desta função é determinar e retornar o maior valor entre os dois argumentos fornecidos. O código começa comparando "b"com "a"para verificar se "b" é maior do que "a". Se essa condição for verdadeira, a função retorna o valor de "b", indicando que "b" é o maior dos dois. Caso contrário, se "b" não for maior que "a" (ou seja, "a" é maior ou ambos são iguais), a função retorna o valor de "a".

Algoritmo 23: min_heapify

```
Função min_heapify(arr, comprimento, i,
 chave):
    maior ← i
    esquerda ← 2 × i + 1
    direita ← 2 × i + 2
    se esquerda < comprimento e chave(arr[i]) ¡
     chave(arr[esquerda]) então
        maior ← esquerda
    fim
    se direita < comprimento e chave(arr[i]) ¡
     chave(arr[direita]) então
        maior ← direita
    fim
    se maior ≠ i então
        temp ← arr[i]
        arr[i] ← arr[maior]
        arr[maior] ← arr[temp]
        min_heapify(arr, comprimento, maior,
         chave)
    fim
fim
```

Este código apresenta uma função denominada "min_heapify" que opera em uma estrutura de dados chamada "heap mínimo" (min heap). A função aceita quatro parâmetros: uma sequência (arr), o comprimento da sequência (comprimento), um índice (i) e uma função de chave (chave).

O objetivo da função é manter a propriedade de min heap na subárvore enraizada no índice "i" da matriz "arr". A propriedade de min heap implica que o valor de um nó é menor ou igual ao valor de seus filhos. A função faz isso verificando se o elemento atual é maior do que seus filhos (esquerdo e direito) e, se necessário, troca o elemento com o menor dos filhos para manter a propriedade do min heap. Ela começa definindo "maior"como o índice atual "i" e calcula os índices dos filhos esquerdo e direito.

Em seguida, a função verifica se o filho esquerdo existe (ou seja, seu índice é menor do que o comprimento da matriz) e se o valor do filho esquerdo é menor que o valor do nó atual, com base na função de chave. Se isso for verdadeiro, atualiza o índice "maior" para apontar para o filho esquerdo.

A mesma verificação é feita para o filho direito, se ele existir e seu valor for menor que o valor atual. Se o índice "maior" for diferente do índice "i", isso significa que uma troca é necessária para manter a propriedade do min heap. Portanto, os elementos em "arr[i]" e "arr[maior]" são trocados.

Em seguida, a função chama recursivamente "min_heapify" na subárvore cuja raiz agora está no índice "maior". Isso garante que a propriedade de min heap seja mantida em todos os níveis da árvore.

Algoritmo 24: heap_sort

```
Função heap_sort(arr, comprimento, chave ←
 lambda(x){x}):
    para i de comprimento//2 até 1 com
     passo de −1 faça
        min_heapify(arr, comprimento, i,
         chave)
    fim
    para i de comprimento até 2 com passo
     de −1 faça
        temp ← arr[i]
        arr[i] ← arr[0]
        arr[0] ← arr[temp]
        min_heapify(arr, i, 0, chave)
    fim
fim
```

Por fim, esta função descreve o algoritmo de ordenação "heap sort" em formato de pseudocódigo. O heap sort é um método de ordenação eficiente que utiliza estruturas de dados chamadas "heaps"(ou montes), que são árvores binárias especiais. A função "heap_sort" recebe como entrada uma sequência a ser ordenada, seu comprimento e uma função de chave opcional que define a base para a ordenação (por padrão, ela ordena os elementos com base em seus valores).

O algoritmo é dividido em duas partes principais. A primeira parte envolve a construção de um heap mínimo a partir da matriz original. Isso é feito percorrendo a matriz da metade para o início e aplicando a função "min_heapify" em cada elemento para garantir que a propriedade de heap mínimo seja mantida.

A segunda parte é responsável pela ordenação real. Ela extrai repetidamente o elemento mínimo do heap mínimo (que está no topo) e o coloca na posição correta na matriz ordenada. Isso é realizado percorrendo a matriz da direita para a esquerda e trocando o elemento mínimo com o último elemento não classificado. A função "min_heapify" é chamada novamente após cada troca para manter a propriedade do heap mínimo.

No final da execução do algoritmo, a matriz original estará completamente ordenada em ordem crescente.

4.6.1 Algoritmo da Otimização da Colônia de Formigas

Através das funções definidas anteriormente pode-se construir um algoritmo da otimização da colônia de formigas extremamente eficaz. O primeiro passo para isso é criar um método para percorrer o grafo.

Algoritmo 25: percorrer_grafo

```
Função percorrer_grafo(grafo, nodo_origem,
  matriz_feromonio, matriz_distancia, alfa,
  beta):
    vertices ← grafo.adiquirir_vertices()
    n_nodos ← |vertices|
    visitados ← [Falso] × n_nodos
    visitados[nodo_origem] = Verdadeiro
    ciclo ← [nodo_origem] × (n_nodos + 1)
    passos ← 0
    atual ← nodo_origem
    comprimento_total ← 0.0
    enquanto passos < n_nodos − 1 faça
        idx ← 1
        saltos_vizinhos ← [0] × (n_nodos −
          passos)
        saltos_valores ← [0] × (n_nodos −
          passos)
        para nodo de 1 até n_nodos faça
            se não visitados[nodo] então
                nivel_feromonio = obter_maior(
                  matriz_feromonio[vertices[
                  atual]][ vertices[nodo]], 1e − 5)
                nivel_distancia = obter_maior(
                  matriz_distancia[vertices[atual]]
                  [vertices[nodo]], 1e − 5)
                peso ← (nivel_feromonio^alfa) ×
                  (nivel_distancia^beta)
                saltos_vizinhos[idx] ← nodo
                saltos_valores[idx] ← peso
                idx ← idx + 1
            fim
            proximo_nodo ← escolher_aleatoriamente(saltosz_vizinhos,
              pesos ← saltos_valores)[0]
            visitados[proximo_nodo] ←
              Verdadeiro
            atual ← proximo_nodo
            ciclo[pasos + 1] ← atual
            pasos ← pasos + 1
        fim
    fim
    comprimento_total ←
      grafo.calcular_custo([vertices[i] para i em
      ciclo])
    retorna ciclo[:-1], comprimento_total
fim
```

Este código descreve um algoritmo para percorrer um grafo. O algoritmo é projetado para encontrar um ciclo em um grafo que maximize a influência de feromônios e a influência de distâncias, usando coeficientes alfa e beta como parâmetros de equilíbrio.

O algoritmo começa recebendo como entrada o grafo, um nó de origem, matrizes de feromônio e distância, bem como os valores alfa e beta que equilibram a influência desses fatores. Ele inicializa variáveis para rastrear os vértices visitados, o ciclo encontrado e outras informações.

A iteração principal do algoritmo ocorre até que todos os nós tenham sido visitados, exceto o nó de origem. Durante cada passo, o algoritmo avalia os vizinhos não visitados do nó atual. Para cada vizinho não visitado, ele calcula um "peso" com base na influência de feromônio (alfa) e na influência de distância (beta). Em seguida, armazena esses pesos e os vizinhos em listas correspondentes.

O próximo nó a ser visitado é escolhido aleatoriamente a partir dessas listas ponderadas, levando em consideração os pesos calculados. Após a escolha, o nó escolhido é marcado como visitado, e o ciclo e o comprimento total do caminho percorrido são atualizados.

O algoritmo continua até que todos os nós tenham sido visitados, exceto o nó de origem. Finalmente, ele calcula o custo total do ciclo encontrado e retorna tanto o ciclo quanto o comprimento total.

Agora se faz necessário uma função que reforce as melhores rotas encontradas.

Algoritmo 26: reforço

```
Função reforço(grafo, comprimento, rota,
  valor, matriz_feromonio):
    vertices ← grafo.adiquirir_vertices()
    se
      grafo.adiquirir_distancia(vertices[rota[1]],
      vertices[rota[2]]) <
      grafo.adiquirir_distancia(vertices[rota[-
      1]], vertices[rota[1]]) então
        para i de 2 até |rota| faça
            matriz_feromonio[vertices[rota[i −
              1]]][vertices[rota[i]]] ←
              matriz_feromonio[vertices[rota[i −
              1]]][vertices[rota[i]]] + valor
        fim
        matriz_feromonio[vertices[rota[−1]]][vertices[rota[1]]]
          ← ma-
          triz_feromonio[vertices[rota[−1]]][vertices[rota[1]]]
          + valor
    fim
    senão
        para i de 2 até |rota| faça
            matriz_feromonio[vertices[rota[i]]][vertices[rota[i−
              1]]] ← ma-
              triz_feromonio[vertices[rota[i]]][vertices[rotai−
              1]]] + valor
        fim
        matriz_feromonio[vertices[rota[1]]][vertices[rota[−1]]]
          ← ma-
          triz_feromonio[vertices[rota[1]]][vertices[rota[−1]]]
          + valor
    fim
fim
```

Este código descreve uma função chamada "reforço". A função é responsável por atualizar a quantidade de feromônio em um grafo, especificamente em uma determinada rota, após a formiga ter percorrido essa rota.

A função "reforço" recebe como entrada um grafo, o comprimento da rota percorrida, a rota em si, um valor de reforço e a matriz de feromônio. Ela começa obtendo a lista de vértices do grafo. Em seguida, verifica se o custo da rota entre o primeiro e o último vértice da rota é menor do que o custo entre o último e o primeiro vértice. Isso determina a direção da rota e como a matriz de feromônios será atualizada.

Essa função de reforço é essencial em algoritmos ACO, pois ajuda a atualizar as informações de feromônio nas arestas do grafo com base no desempenho das formigas. Isso influencia diretamente as decisões futuras de rota das formigas, uma vez que elas tendem a escolher caminhos com mais feromônio. Portanto, a função desempenha um papel crítico na melhoria da eficiência do algoritmo de otimização baseado em colônia de formigas.

Agora só resta definir a função principal do algoritmo.

Algoritmo 27: ACO

Entrada: Caminho do arquivo de input

```
grafo ← ler_arquivo_tsp(caminho)
ponto_de_retorno ← "1"
n_formigas ← 50
n_iteracoes ← 500 feromonio_inicial ← 0.0001
taxa_de_evaporacao ← 0.3
alfa ← 0.5
beta ← 2.9
k ← 1
q ← 0.4
atualizar_por ← "qualidade"
pior ← Falso
elitista ← Falso
vertices ← grafo.adquirir_vertices()
se não esta_contido(vertices,
 ponto_de_retorno) então
    levantar Excecao("Ponto de retorno não
     encontrado.")
fim
indice_ponto_de_retorno ← 0
matriz_distancia ← {}
matriz_feromonio ← {}
para i de 1 até |vertices| faça
    matriz_feromonio[vertices[i]] ← {}
    matriz_distancia[vertices[i]] ← {}
    para cada  j em vertices faça
        se j ≠ vertices[i] então
            matriz_feromonio[vertices[i]][j] ←
             feromonio_inicial
            dist ←
             grafo.adiquirir_distancia(vertices[i],
             j)
            se dist ≠ 0 então
                matriz_distancia[vertices[i]][j] ←
                 1/dist
            fim
            senão
                matriz_distancia[vertices[i]][j] ←
                 0
            fim
        fim
    fim
    vertices[i] = ponto_de_retorno
        indice_ponto_de_retorno ← i
    fim
fim
melhor_rota ← ∅
melhor_custo ← +∞
para i de 1 até n_iteracoes faça
    lote ← [([], 0)] × n_formigas
    para f de 1 até n_formigas faça
        trilha ← percorrer_grafo(grafo,
         indice_ponto_de_retorno,
         matriz_feromonio, matriz_distancia,
         alpha, beta, )
        lote[f] ← trilha
        trilha[2] < melhor_custo
            melhor_custo ← trilha[2]
            melhor_rota ← trilha[1]
        fim
    fim
    j em chaves de matriz_feromonio
        l em chaves de matriz_feromonio[j]
            matriz_feromonio[j][l] ←
             matriz_feromonio[j][l] × (1 −
             taxa_de_evaporacao)
        fim
    fim
    para j de 1 até n_formigas faça
        delta ← q/lote[j][2]
        para l de 1 até |lote[j][1]|−1 faça
            comeco ← vertices[lote[j][0][l]]
            fim ← vertices[lote[j][0][l + 1]]
            matriz_feromonio[comeco][fim] ←
             delta
        fim
        matriz_feromonio[vertices[lote[j][0][-
         1]]][vertices[lote[j][0][0]]] ← delta
    fim
    pior
        solucao, _ ← obter_máximo(lote,
         n_formigas, lambda(x){x[1]})
        reforço(grafo, solucao, -1,
         matriz_feromonio)
    fim
```

```
    se elitista então
        reforço(grafo, melhor_rota, 1,
         matriz_feromonio)
    fim
    se atualizar_por = "quality" então
        heap_sort(lote, n_formigas,
         lambda(x){x[1]})
        para cada solucao, _ em lote[:k] faça
            reforço(grafo, solucao, 1,
             matriz_feromonio)
        fim
    fim
    senão se atualizar_por = "rank" então
        heap_sort(lote, n_formigas,
         lambda(x){x[1]})
        delta ← k
        para cada solucao, _ em lote[:k] faça
            reforço(grafo, solucao, delta,
             matriz_feromonio)
            delta ← delta − 1
        fim
    fim
fim
melhor_rota_traduzida ← [vertices[i] para i em
 melhor_rota]
i de 1 até |melhor_rota_traduzida|
    Escrever(melhor_rota_traduzida[i])
fim
Escrever("\n")
```

O algoritmo ACO começa lendo um arquivo que representa um grafo. Ele define diversos parâmetros, como o ponto de retorno (ponto inicial), o número de formigas, o número de iterações, o valor inicial de feromônio, a taxa de evaporação do feromônio, os coeficientes alfa e beta que controlam a influência de feromônio e distância nas escolhas das formigas, bem como outros parâmetros relacionados ao processo de otimização. Os valores para cada um desses parâmetros foi encontrado através de uma busca cega, onde foram sorteadas diversas combinações possíveis para estes parâmetros, e a que aqui se encontra foi a que produziu o melhor resultado, apresentando não só extrema eficácia nos testes com 52 pontos, como também um tempo relativamente baixo de apénas 40 segundos em média.

Antes de iniciar a otimização, o código verifica se o ponto de retorno está contido nos vértices do grafo e atribui um índice a ele. Em seguida, inicializa as matrizes de distância e feromônio, onde a matriz de feromônio começa com valores iniciais e a matriz de distância é preenchida com as distâncias entre os vértices do grafo.

Para cada par de vértices no grafo, o algoritmo calcula as distâncias e atribui valores iniciais de feromônio às arestas correspondentes. O ponto de retorno recebe um valor especial na matriz de feromônio para indicar seu papel como ponto de partida.

Depois de inicializar as estruturas de dados, o algoritmo entra em um loop principal que executa um número específico de iterações. Durante cada iteração, um conjunto de formigas é enviada para percorrer o grafo. As formigas fazem escolhas de caminhos com base na quantidade de feromônio e nas distâncias entre os vértices, usando os coeficientes alfa e beta para equilibrar a influência desses fatores.

Após cada iteração, o feromônio é atualizado com base na qualidade do caminho percorrido pelas formigas. O algoritmo pode ser configurado para atualizar o feromônio com base na qualidade do caminho encontrado (melhor caminho) ou usando uma abordagem elitista.

Esse processo de otimização é repetido ao longo de várias iterações, e o resultado final é um caminho otimizado no grafo que representa uma solução para o problema em questão, como a rota mais curta para o PCV.

Após as configurações iniciais, o código entra em um loop principal que executa um número específico de iterações, representadas pela variável "n_iteracoes". Durante cada iteração, um lote de trilhas é criado para as formigas percorrerem. Cada formiga percorre o grafo usando a função "percorrer_grafo", que calcula uma rota com base nas influências de feromônio e distância, utilizando os coeficientes alfa e beta.

O código acompanha a melhor rota encontrada até o momento, atualizando-a se uma nova rota tiver um custo menor. Isso é feito comparando o custo da rota recém-encontrada (armazenado em "trilha[2]") com o melhor custo atual.

Após todas as formigas completarem suas trilhas, o código realiza a atualização do feromônio. Primeiro, ele aplica a evaporação do feromônio em todas as arestas do grafo, reduzindo o valor do feromônio de acordo com a taxa de evaporação.

Em seguida, o código aumenta a quantidade de feromônio nas arestas percorridas pelas formigas. Isso é feito para cada formiga e cada aresta percorrida, aumentando o feromônio na direção da trilha percorrida.

Existem opções adicionais de atualização do feromônio com base em configurações como "pior"(que atualiza o feromônio com base na pior rota encontrada), "elitista"(que atualiza o feromônio com base na melhor rota encontrada), "qualidade"ou "rank"(que atualizam o feromônio com base na qualidade das rotas).

Finalmente, o código traduz a melhor rota encontrada em termos de índices de vértices para os vértices reais do grafo e a imprime. Isso fornece a solução encontrada pelo algoritmo ACO para o PCV.

4.6.2 Exemplo de Funcionamento do Algoritmo ACO

Tome a mesma entrada da instância do AG. Serão geradas quatro soluções para este problema. O número de máximo de iterações neste exemplo será de duas iterações. Vale ressaltar que estas formigas do algoritmo têm memória, ou seja, não é permitido que elas

passem por um mesmo vértice duas vezes.

4.6.2.1 Primeira Iteração

Na primeira iteração, as quatro formigas iniciais são colocadas em um ponto aleatório. Suponha que os vértices iniciais são: A, B, B e C. Depois de sorteado o ponto de origem, as formigas se movem sendo influenciadas pela trilha de feromônios de outras formigas, que será chamado de fator alfa, e pela distância dos vértices, que será chamado de fator beta.

Cada fator tem uma constante multiplicativa prédefinida que é usada para ponderar as decisões das formigas. Suponha que a trilha de feromônios impacte três vezes mais do que a distância.

Na primeira iteração não há trilhas de feromônios, pois a trilha é deixada pela formiga mais eficiente de cada iteração. Então, o trajeto será influenciado fortemente pela distância dos vértices.

Por exemplo, a formiga que começa em A, pode ir para B, C e D que têm distâncias de 3, 4 e 4, respectivamente . É mais provável que a formiga escolha B, p ao fator beta, suponha que foi isso que ela fez. Como a formiga já visitou A e B, então ela só pode ir para C ou D de distâncias 3 e 5, é mais provável que ela escolha o vértice C, suponha que ela fez isso. Tendo visitado A, B e C, só resta a cidade D. Assim, o primeiro caminho formado é ABCD.

O mesmo processo se repete para cada formiga. Então, são formados , por acaso, os caminhos: ABCD, BACD, BADC, CADB, cujas distâncias já foram calculadas antes. Depois disso, a formiga que trilhou o menor caminho deixa uma trilha de feromônios que influenciará bastante as trilhas das formigas na próxima iteração.

4.6.2.2 Segunda Iteração

Novamente são sorteados quatro vértices iniciais para as formigas, suponha que foram escolhidos os vértices A, B, B, C. Nesta iteração, a solução mais eficiente, no caso ABCD, deixou uma trilha de feromônios aumentando a desejabilidade das arestas desta trilha.

Repete-se o processo da primeira iteração. Desta vez, para um indivíduo que começa em B. Esta formiga pode percorrer os vértices A, C e D com distâncias de 3, 3 e 5. Neste caso, as distâncias BA e BC são as mesmas, mas como a aresta BC está na trilha que deixou feromônios, ela tem uma desejabilidade três vezes maior. Repetindo este algoritmo para os outros vértices foi construído o caminho BCAD.

Cada formiga executa procedimento similar. Então foi construído os seguintes caminhos: ABDC,BCAD, BADC e CBAD de distâncias 14, 21, 16, 17. Como o número de gerações deste exemplo foi excedido, o caminho retornado pelo algoritmo será o caminho ABDC de custo 14. Nota-se que a solução ABDC é a solução ótima para este problema, no entanto, não é possível garantir a melhor solução para todos os problemas em que ACO é aplicado.

5 Experimentos

5.1 Tempo de execução do problema de permutação

Tabela 7: Tempos de execução do algoritmo de força bruta

Nº de Vértices	Tempo decorrido(s)
1	0.05832051
2	0.067533054
3	0.169657311
4	0.052155086
5	0.066800107
6	0.084402602
7	0.449491816
8	18.454111356
9	1932.339425095

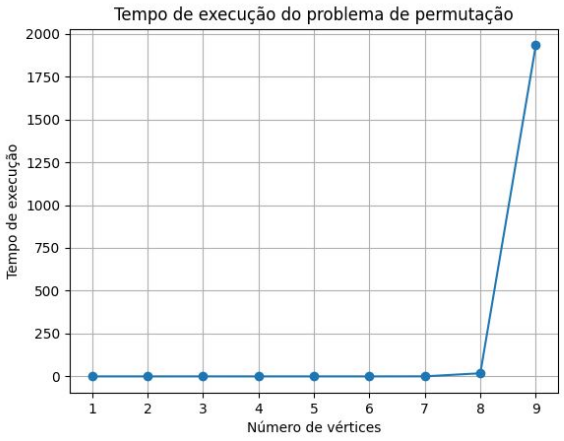

Figura 8: Representação gráfica dos tempos do Tempo de execução do problema de permutação

O experimento foi realizado em um computador Acer Aspire E5-574 equipado com um processador i5-6200U, 8 GB de memória RAM, SSD de 512GB e uma placa de vídeo Intel Graphics 520. O sistema operacional utilizado foi o windows 10. A linguagem utilizada foi o Python com a biblioteca MatPlotLib.

Neste gráfico é acentuado o fenômeno da explosão combinatória que é uma característica presente em certos tipos de problemas em que um pequeno aumento na entrada pode levar a um imenso aumento no tempo de execução. É notável que para até sete vértices o tempo de execução é menor que um segundo. No entanto, o tempo de execução para as permutações de nove vértices é de cerca de trinta e três minutos. Como mostrado na tabela 3.

5.2 Instâncias de execução do problema

5.2.1 Tempo de execução de três pontos de entrega

```
4 4
B 0 0 0
0 A 0 0
0 0 0 0
R 0 C 0
```

Figura 9: Entrada de dados

```
100% [■■■■■■■■■■] - Generating permutations
100% [■■■■■■■■■■] - Checking paths
Best path: ['B', 'A', 'C']
Shorter distance: (10)
Elapsed time: 0.59106884s
```

Figura 10: Saída produzida pelo algoritmo

Nesse cenário, a solução ótima é dada em centésimos de segundo. Logo o algoritmo é viável para no caso de roteirização de drones com três pontos de entrega.

5.2.2 Tempo de execução de seis pontos de entrega

```
4 5
E 0 0 0 D
0 A 0 F 0
0 0 0 0 C
R 0 B 0 0
```

Figura 11: Entrada de dados

```
100% [■■■■■■■■■■] - Generating permutations
100% [■■■■■■■■■■] - Checking paths
Best path: ['E', 'A', 'F', 'D', 'C', 'B']
Shorter distance: (16)
Elapsed time: 0.216779605s
```

Figura 12: Saída produzida pelo algoritmo

Para seis pontos de entrega, o tempo de execução é quase instantâneo. Logo é uma solução viável para até seis pontos de entrega também.

5.2.3 Tempo de execução de nove pontos de entrega

```
8 8
B 0 0 0 0 0 0 E
0 A 0 0 0 0 0 0
0 0 0 0 0 0 0 0
R 0 C 0 0 F 0 0
0 0 0 0 0 0 0 0
0 0 I 0 D 0 0 0
0 0 0 0 0 0 0 G
H 0 0 0 0 0 0 0
```

Figura 13: Entrada de dados

```
100% [■■■■■■■■■■] - Generating permutations
100% [■■■■■■■■■■] - Checking paths
Best path: ['B', 'A', 'C',
  →   →   → 'F', 'E', 'G',
  →   →   → 'D', 'I', 'H']
Shorter distance: (36)
Elapsed time: 1932.339425095s
```

Figura 14: Saída produzida pelo algoritmo

Para o cenário de nove pontos de entrega o algoritmo se torna inviável. Esta execução demorou trinta e três minutos.

5.3 Comparação entre algoritmo da força bruta, algoritmo da colônia de formigas e algoritmo genético para pequenas entradas

Nesta seção, serão comparados o algoritmo de força bruta, o algoritmo da colônia de formigas e o algoritmo genético. Esta comparação se dará em função do tempo de execução, das distâncias totais das rotas e do custo de cada rota.

Esta seção e a próxima seção usarão como entradas o formato berlin52. Este é um problema de otimização cujo objetivo é percorrer 52 pontos e retornar para o ponto de retorno da cidade de Berlin, na Alemanha, na menor distância possível. No caso, serão usadas entradas de 7, 8 e 9 vértices.

Os experimentos para grandes e pequenas entradas foram realizados em um computador Acer Nitro AN515-44 equipado com um processador AMD Ryzen 7 Mobile 4800H, 16GB (2x8GB) de memória RAM DDR4 de 3200MHz, 2 SSD's, um WDC PC SN530 SDBPNPZ-512G-1014 de 512GB (no qual o sistema operacional está instalado) e um KINGSTON SNV2S de 1TB, com placa de vídeo integrada AMD Radeon™ Graphics Vega 7nm e placa de vídeo dedicada NVIDIA GeForce GTX 1650 com VRAM GDDR6 de 4GB. O sistema operacional utilizado foi o Windows 11 Home Single Language versão 22H2. A linguagem utilizada foi o Python com a biblioteca MatPlotLib.

5.3.1 Tempo de execução para pequenas entradas

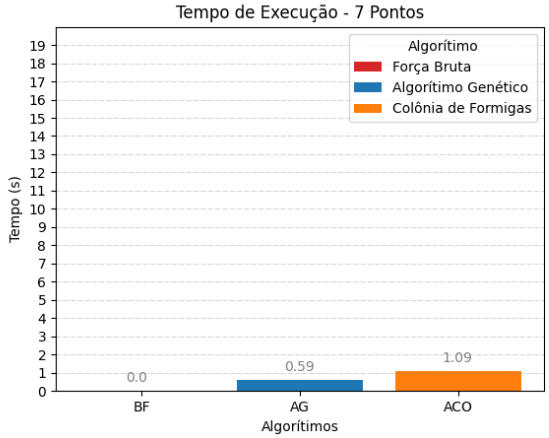

Figura 15: Tempos de execução para 7 pontos

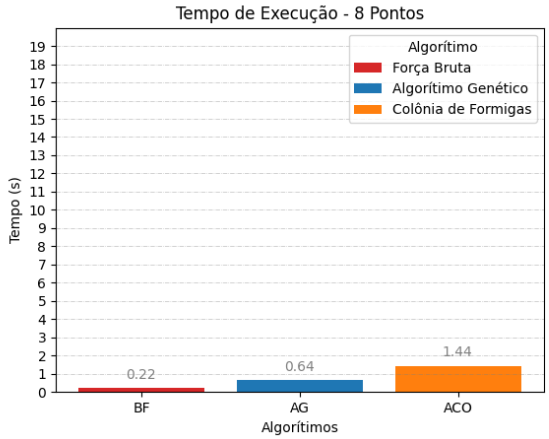

Figura 16: Tempos de execução para 8 pontos

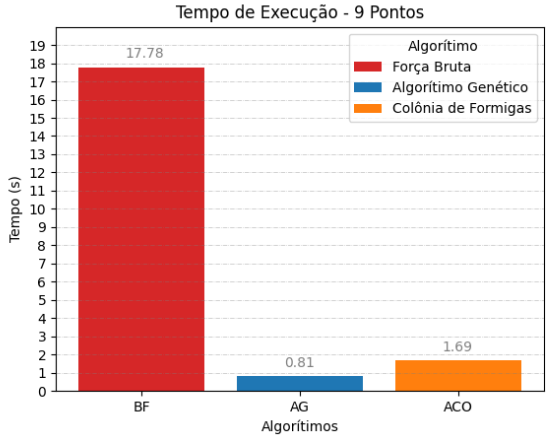

Figura 17: Tempos de execução para 9 pontos

Como é possível notar nas figuras 15, 16 e 17, o algoritmo de força bruta tem um desempenho melhor no quesito tempo do que o AG e do ACO no exemplo de sete vértices e oito vértices. Isto se dá devido aos parâmetros iniciais escolhidos.

Os parâmetros que maximizam a eficiência do ACO são o número de formigas que é 146 e o número de gerações que são 321, isso faz com que o espaço de busca do ACO seja igual a 46.866 possibilidades. Já no caso do AG, são 30 indivíduos iniciais e 1500 gerações, com um espaço de busca de 45000. O algoritmo de força bruta para 7 e 8 entradas tem o espaço de busca de 5040 e 40320 possibilidades.

Isso não acontece a partir de nove vértices. Pois, o espaço de busca do algoritmo de força bruta excede o dos demais algoritmos. Portanto, a partir de nove vértices as meta-heurísticas são mais eficientes em questão de tempo.

5.3.2 Rotas obtidas e custo das rotas para pequenas entradas

Para pequenas entradas acontece um fenômeno muito interessante. As rotas dos algoritmos meta-heurísticos coincidem com os algoritmos de força bruta. Isto se dá, pois os espaços de busca, como provado anteriormente, são bastante semelhantes.

No caso de entradas com mais vértices isto é mais difícil de acontecer. A figura 18 mostra a comparação entre as rotas obtidas pelo algoritmo de força bruta, algoritmo genético e colônia de formigas para 7, 8 e 9 vértices.

5.4 Comparação entre algoritmo da colônia de formigas e algoritmo genético para grandes entradas

A figura 19 trata da comparação entre as meta-heurísticas de algoritmo de colônia de formigas para todas as 52 entradas do Berlin52. Para esses experimentos, os algoritmos foram executados 30 vezes e seus resultados foram colocados no gráfico da figura.

Nota-se que o ACO é bem mais eficiente do que o AG. Tanto por apresentar um desempenho que varia consideravelmente menos, como também por ficar muito mais próximo da solução ótima. Isto se dá, pois o AG não tem métodos de busca local, isso faz com que boas soluções sejam perdidas durante o crossover e mutação. Os AGs que têm mecanismos de busca local são conhecidos como algoritmos genéticos híbridos.

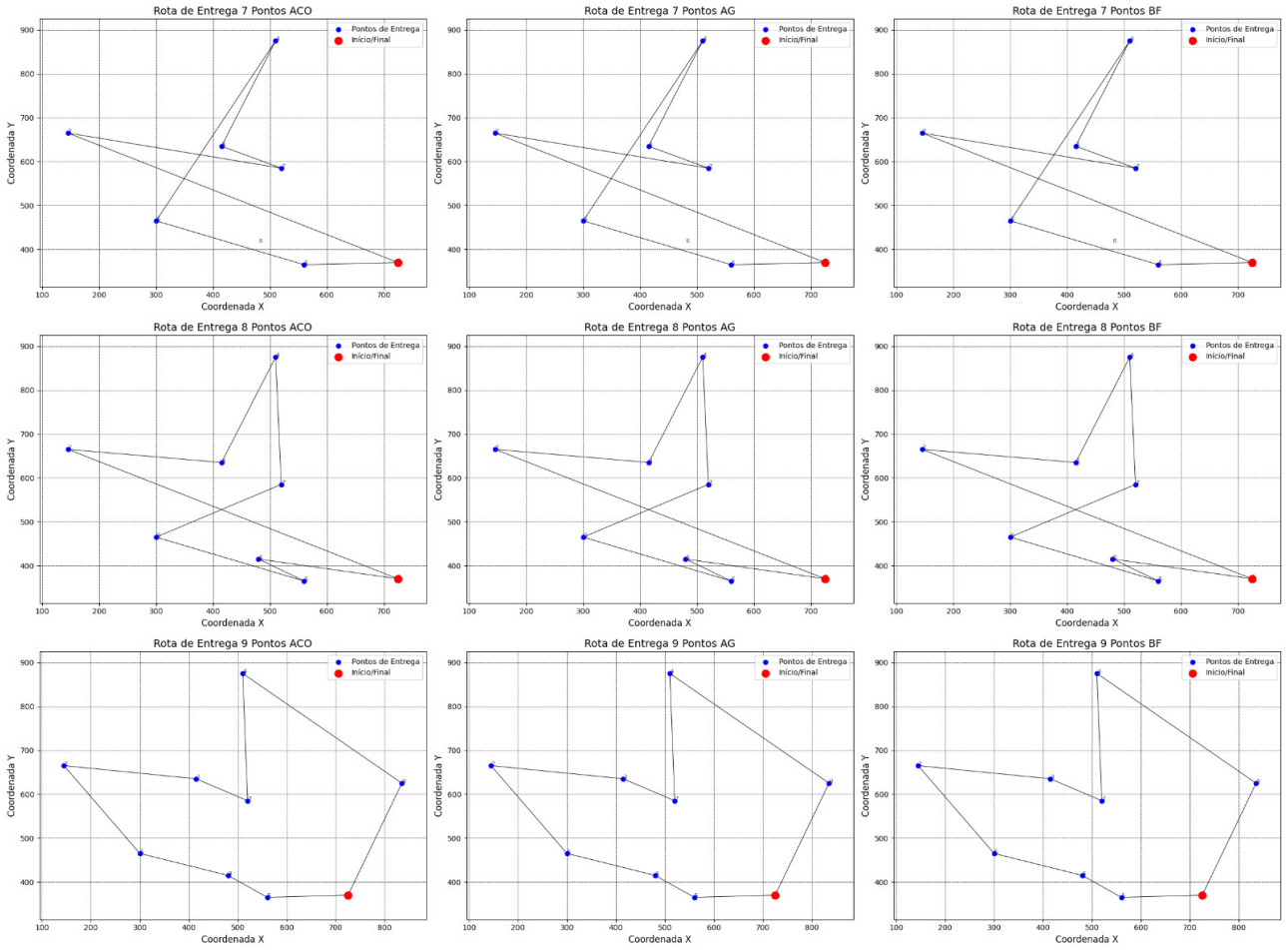

Figura 18: Rotas obtidas para pequenas entradas

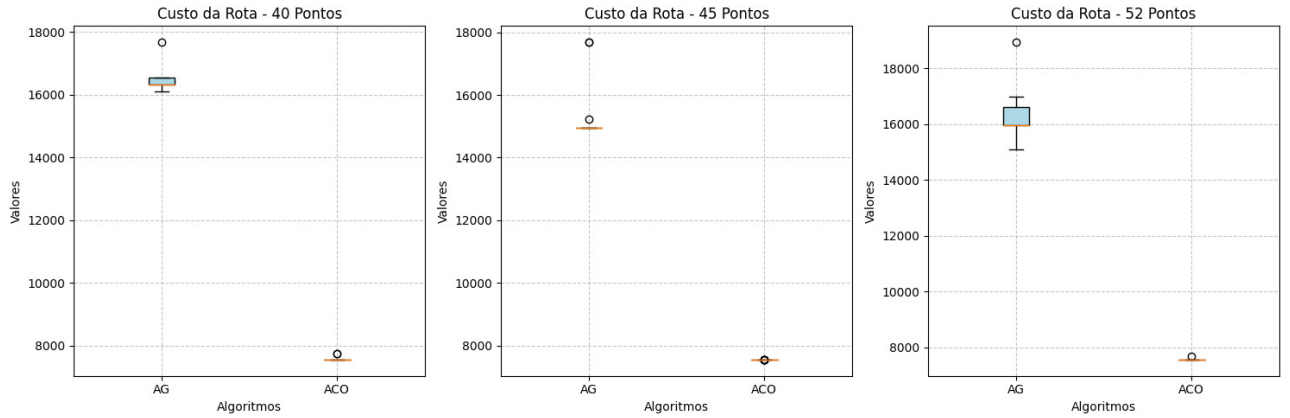

Figura 19: Custo das Rotas obtidas para grades entradas: AG e ACO

5.4.1 Variação no tempo de execução das meta-heurísticas

Neste caso, conforme a figura 20, o tempo de execução do ACO, ficou consideravelmente mais alto do que o AG. Isso pode se dar pelo motivo de o algoritmo de colônia de formigas ser mais focado na exploração do espaço de busca de vértice a vértice que aumenta o tempo de execução para problemas muito grandes como o do Berlin52.

Como mostrado na figura 21, a roteirização do AG e do ACO são bastante distintas. O ACO tem um desenho mais contíguo, enquanto a rota do AG possuí várias arestas concorrentes. Nota-se também que o deslocamento do ACO é significativamente menor em comparação com o AG.

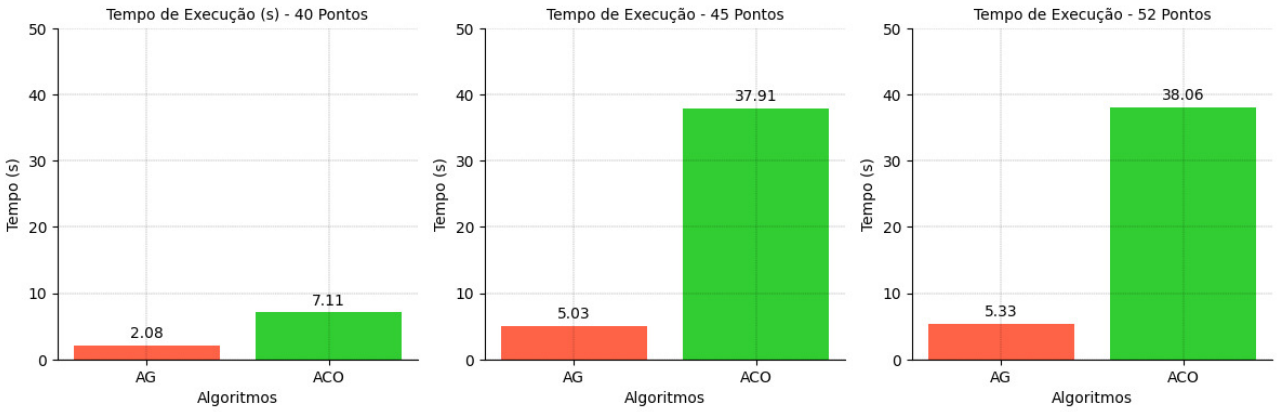

Figura 20: Tempos de execução obtidas de grades entradas: AG e ACO

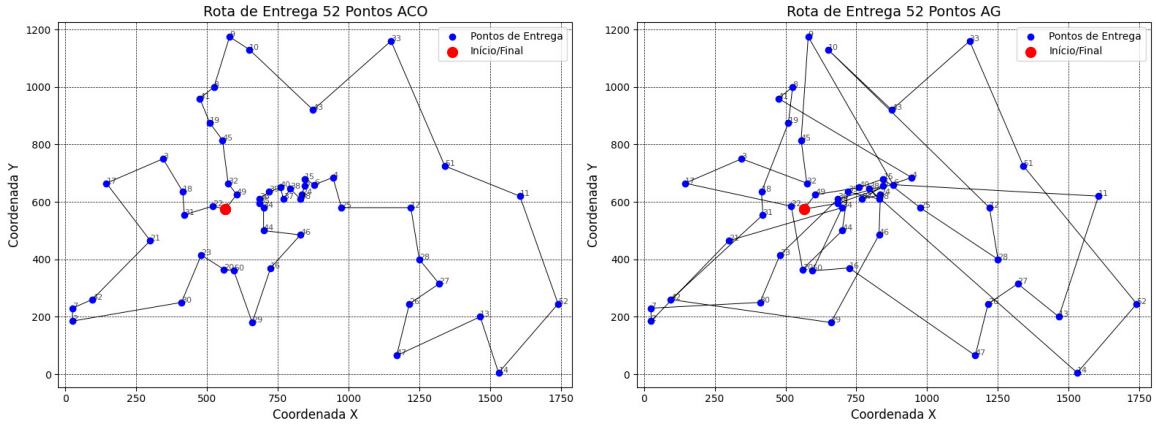

Figura 21: Rotas obtidas para grandes entradas: ACO e AG

6 Resultados

Portanto, os experimentos concluem que o algoritmo de força bruta aplicado ao PCV nos garante uma solução em tempo hábil para até oito vértices para o problema do FlyFood. Isso é suficiente para fins práticos, considerando que o trajeto de ida e volta do estabelecimento passando por oito pontos intermediários daria uma média de três minutos por entrega. No entanto, algoritmos que utilizam métodos meta-heurísticos dão soluções muito boas em tempo hábil para problemas que seriam intratáveis pelo algoritmo de força bruta. Sendo uma solução mais viável para rotas mais complexas.

7 Conclusão

Este trabalho teve como objetivo o desenvolvimento de soluções eficientes para o PCV usando o algoritmo de força bruta, o algoritmo de colônia de formigas e o algoritmo genético no contexto do Problema do FlyFood. O algoritmo de força bruta retorna soluções ótimas de forma quase instantânea até sete vértices. Enquanto os algoritmos que utilizam metaheurísticas dão respostas boas em tempo hábil para problemas intratáveis por meio do algoritmo de força bruta.

Para trabalhos futuros serão usadas técnicas de busca local para melhorar o tempo de execução do algoritmo de roteirização baseado em algoritmos genéticos. Estas técnicas devem tornar o algoritmo genético menos preso a soluções boas locais, deste modo, o AG explorará melhor o espaço de busca garantindo melhores soluções.

Referências

[1] CORMEN, Thomas H. Algoritmos: Teoria e Prática. Rio de Janeiro: Elsevier Editora Ltda, 2002.

[2] ScienceDirect. "Traveling Salesman Problem - an overview — ScienceDirect Topics", 2017.

[3] CORMEN, Thomas H. Desmistificando Algoritmos. Rio de Janeiro: Elsevier Editora Ltda, 2014.

[4] PICOLO, Ana Paula; DA SILVA, Roger Sá. Aplicação do Problema do Caixeiro Viajante para determinação de rotas turísticas. Rio Grande do Sul: Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS), 2022.

[5] RAIVI, Asif Mahmud et al. Drone Routing for Drone-Based Delivery Systems: A Review of Trajectory Planning, Charging, and Security. Coreia do Sul: Chosun university, 2023.

[6] BISPO, Rodolfo César. Planejador de roteiros turísticos: uma aplicação do problema do Caixeiro Viajante na cidade do Recife. Recife: Universidade Federal Rural de Pernambuco, 2018.

[7] CHOPARD, Bastien; TOMASSINI, Marco. An Introduction to Metaheuristics for Optimization. 2. ed. London: Wiley, 2018.