



**Universidade Federal Rural de Pernambuco**  
**Departamento de Estatística e Informática**  
**Bacharelado em Sistemas de Informação**

**ABORDAGENS HEURÍSTICAS PARA O PROBLEMA DO  
CAIXEIRO VIAJANTE**

João Pedro de Lima

Márcia Alves de Assis Lima

Mateus Nicolas Santos Lins

Wilka Vitória Granjeiro do Nascimento

Recife, 12 de março de 2025.

# LISTA DE ILUSTRAÇÕES

## LISTA DE FIGURAS

Figura 1 . Comparação entre os tempos de complexidade-funções polinomial x funções exponenciais. ....	9
Figura 2.Exemplo de entrada do algoritmo. ....	18
Figura 3. Resumo do fluxo do código-fonte.....	20
Figura 4. Algoritmo de força bruta - Leitura da matriz e permutações. ....	20
Figura 5. Algoritmo de força bruta: Cálculo das distâncias dos pontos e verificação do melhor .....	21
Figura 6. Algoritmo de força bruta: Saída do código .....	22
Figura 7. Rota do caso 1 .....	22
Figura 8. Rota do caso 2 .....	23
Figura 9. Rota do caso 3 .....	23
Figura 10. Rota do caso 4 .....	23
Figura 11.Algoritmo Genético - Função algoritmo genético.....	27
Figura 12.Algoritmo Genético - Função fitness .....	28
Figura 13. Algoritmo Genético - método da roleta.....	29
Figura 14. Algoritmo Genético – crossover .....	30
Figura 15.Algoritmo Genético - mutação.....	30
Figura 16. Algoritmo Genético - Removendo genes repetidos .....	31
Figura 17. Algoritmo Genético - ajuste populacional .....	31
Figura 18. Algoritmo Genético - seleção do melhor indivíduo .....	32
Figura 19. Gráfico comparativo das metaheurísticas com a solução ótima.....	41
Figura 20. Gráfico comparativo do tempo de execução com a solução ótima .....	41

## LISTA DE TABELAS

Tabela 1.Resultados para o algoritmo de Força Bruta.....	35
Tabela 2.Parâmetros utilizados no algoritmo genético.....	376
Tabela 3.Parâmetros utilizados no algoritmo colônia de formiga. ....	377

# ÍNDICE

1. Introdução .....	5
1.1. Apresentação e Motivação .....	5
1.2. Objetivos .....	6
1.2.1. Objetivo Geral .....	6
1.2.2. Objetivo Específico .....	7
1.3. Organização do projeto .....	7
2. Referencial Teórico .....	8
2.1. Problemas NP-Completo .....	8
2.2. Algoritmos de tempo polinomial e exponencial .....	8
2.3. Algoritmos determinísticos e não-determinísticos .....	9
2.4. Classes de complexidade .....	10
2.5. O problema do caixeiro viajante .....	10
2.6. Algoritmos de Força Bruta e o problema do caixeiro viajante .....	10
2.7. Algoritmos genéticos e o problema do caixeiro viajante .....	11
2.8. Algoritmos do vizinho mais próximo e o problema do caixeiro viajante .....	13
2.9. Algoritmos da otimização por colônia de formigas e o problema do caixeiro viajante .....	14
3. Trabalhos relacionados .....	15
4. Metodologia .....	16
4.1. Algoritmo de Força Bruta .....	16
4.2. Algoritmo Genético .....	17
4.3. Algoritmo do vizinho mais próximo .....	17
4.4. Algoritmo de colônia de formigas .....	17
5. Experimentos .....	17
5.1. Algoritmo de Força bruta .....	18

5.1.1.	Descrição do código .....	18
A)	Importação e inicialização.....	18
B)	Funções auxiliares.....	18
C)	Função principal (main).....	19
D)	Execução .....	20
5.2.	Algoritmo do vizinho mais próximo .....	24
5.2.1	Descrição do código.....	24
5.3.	Algoritmo genético .....	26
5.3.1.	Descrição do código.....	26
5.4.	Algoritmo colônia de formiga .....	32
5.4.1.	Descrição do código.....	32
6.	Resultados .....	34
6.1.	Algoritmo de Força Bruta .....	34
6.2.	Algoritmo Genético .....	35
6.3.	Algoritmo Colônia de Formiga.....	36
6.4.	Algoritmo do Vizinho Mais Próximo .....	38
7.	Conclusão.....	40
8.	Referências bibliográficas.....	42

## Resumo

Problemas de otimização são frequentemente encontrados no mundo real, mas muitos desses problemas são desafiadores de resolver na teoria, pois pertencem à classe dos problemas NP-completos. Um exemplo clássico dessa categoria é o Problema do Caixeiro Viajante. Nesta versão do projeto, foram abordados quatro tipos de algoritmos – Algoritmo de Força Bruta, o Genético (AG), a Otimização por Colônia de Formigas (ACO - *Ant Colony Optimization*) e o Algoritmo do Vizinho Mais Próximo (VMP) – para a resolução dessa classe de problemas, utilizando as instâncias clássicas Berlin52, ST70, Brazil58, PR107 e kroA100. O método foi utilizado visando a solução de problemas de otimização para os quais não se conhece um algoritmo comprovadamente eficiente.

**Palavras-chave:** caixeiro viajante, algoritmos heurísticos, tempo exponencial, otimização, NP-completos.

# 1. Introdução

## 1.1. Apresentação e Motivação

O FlyFood é um aplicativo inovador de entregas que utiliza drones para transportar múltiplos pedidos de maneira eficiente, atendendo a diferentes destinos e retornando à base de operação. A complexidade de gerenciar rotas para drones em tempo real, considerando limitações como distância, carga e tempo, exige a aplicação de conceitos avançados da computação e matemática, especialmente o Problema do Caixeiro Viajante (TSP - *Travelling Salesman Problem*).

No cerne do problema enfrentado pelo FlyFood está o TSP, que visa determinar o caminho mais eficiente para que um "viajante" (no caso, um drone) visite um conjunto de locais e retorne ao ponto de origem, minimizando o custo total, que pode ser medido por tempo de viagem ou consumo de energia.

Para resolver esse desafio, inicialmente o FlyFood foi modelado como um problema de grafos, no qual cada ponto de entrega, representado por  $P(x,y)$ , é comparado ao ponto de origem,  $R(x,y)$ . A partir desse ponto inicial, o algoritmo considera a coordenada como ponto atual e recalcula a distância entre os pontos vizinhos, determinando a menor distância até o próximo ponto, enquanto descarta os pontos já visitados. Esse processo é repetido até que todos os pontos de entrega sejam visitados, o que está diretamente relacionado ao conceito de Caminho Hamiltoniano, que descreve um percurso em um grafo no qual cada nó é visitado uma única vez [9].

O desafio de encontrar soluções para o TSP é classificado como NP-difícil, o que significa que, até o momento, não se conhece um algoritmo eficiente que consiga resolver todas as instâncias do problema em tempo polinomial. No contexto do FlyFood, isso se traduz na dificuldade de calcular rotas ótimas de forma rápida e em tempo real, especialmente à medida que o número de pontos de entrega aumenta, tornando o problema computacionalmente mais desafiador.

Para lidar com essa complexidade, o FlyFood recorre a algoritmos de otimização de rotas, combinando técnicas exatas e aproximadas para equilibrar eficiência e viabilidade prática. Neste trabalho, são exploradas quatro abordagens algorítmicas para resolver o TSP: o Algoritmo de Força Bruta, o Genético (AG), a Otimização por Colônia de Formigas (ACO - *Ant Colony Optimization*) e o Algoritmo do Vizinho Mais Próximo (VMP).

O Algoritmo Genético, inspirado na teoria da evolução natural, utiliza operadores como seleção, cruzamento e mutação para encontrar soluções aproximadas. Já a Colônia de Formigas, baseada no comportamento de formigas reais em busca de alimentos, emprega

feromônios para explorar e explorar caminhos promissores. Por fim, o Algoritmo do Vizinho Mais Próximo, uma heurística simples e gulosa, constrói soluções iterativamente, sempre escolhendo a cidade mais próxima como próximo passo.

Cada um desses algoritmos possui características distintas em termos de complexidade, eficiência e qualidade das soluções geradas. Enquanto o Vizinho Mais Próximo é rápido e fácil de implementar, ele tende a produzir soluções subótimas. Por outro lado, o Algoritmo Genético e a Colônia de Formigas, embora mais complexos e computacionalmente custosos, são capazes de explorar o espaço de busca de forma mais robusta, encontrando soluções de melhor qualidade.

Para avaliar e comparar o desempenho dos três algoritmos de otimização, o Vizinho Mais Próximo, o Algoritmo Genético e a Colônia de Formigas, as instâncias clássicas do Problema do Caixeiro Viajante (TSP) — Berlin52, ST70, Brazil58, PR107 e kroA100 — foram utilizadas como casos de teste [10]. Essas instâncias são importantes porque são referência comum, as quais permitem testar a eficiência, precisão e escalabilidade dos algoritmos em diferentes cenários.

No cotidiano, é comum enfrentar situações que exigem a maximização ou minimização de custos, como no empacotamento de objetos em contêineres, na localização de centros de distribuição ou no escalonamento e roteamento de veículos. Portanto, é fundamental buscar soluções eficientes para esses desafios. No caso do FlyFood, o objetivo é minimizar os custos operacionais, calculando a menor distância que um drone pode percorrer com base em coordenadas que indicam o ponto de origem e os locais de entrega.

Dessa forma, o FlyFood integra esses métodos em um sistema dinâmico de roteamento, visando maximizar a eficiência operacional e garantir um serviço ágil e confiável aos clientes. O uso de drones como agentes inteligentes de entrega permite a implementação de soluções que, apesar de não serem ótimas, são altamente eficientes e práticas para o contexto de entregas em tempo real.

## **1.2.Objetivos**

### **1.2.1. Objetivo Geral**

O objetivo deste trabalho é implementar e comparar diferentes abordagens algorítmicas para resolver o problema de roteamento de drones, visando calcular a menor distância percorrida ao visitar todos os pontos de entrega e avaliar o tempo de execução de cada algoritmo.

## **1.2.2. Objetivo Específico**

- Implementar o algoritmo de Força Bruta para calcular a menor distância que um drone pode percorrer ao visitar todos os pontos de entrega, avaliando sua viabilidade em problemas de pequena escala e analisando sua complexidade computacional;
- Implementar o algoritmo do Vizinho Mais Próximo como uma heurística simples e rápida para resolver o problema de roteamento, comparando sua eficiência e qualidade das soluções com os outros métodos;
- Implementar o Algoritmo Genético para explorar soluções aproximadas ao problema, utilizando operadores de seleção, cruzamento e mutação, e avaliar sua capacidade de encontrar rotas eficientes em um tempo computacional viável;
- Implementar o algoritmo de Colônia de Formigas para simular o comportamento de formigas na busca por caminhos curtos, analisando sua eficácia em explorar o espaço de soluções e encontrar rotas otimizadas;
- Comparar os resultados obtidos pelos três algoritmos (Vizinho Mais Próximo, Algoritmo Genético e Colônia de Formigas) em termos de:
  - Distância total da rota.
  - Tempo de execução.
  - Escalabilidade em instâncias de diferentes tamanhos, como Berlin52, ST70, Brazil58, PR107 e kroA100.

## **1.3. Organização do projeto**

No Capítulo 2, é apresentado o referencial teórico, com a exposição de conceitos fundamentais, incluindo algoritmos de tempo polinomial e exponencial, algoritmos determinísticos e não determinísticos, o problema do caixeiro viajante e a metodologia baseada nos algoritmos força bruta, VMP, AG e ACO. O Capítulo 3 descreve os métodos e as ferramentas empregadas no desenvolvimento dos algoritmos. No Capítulo 4, é apresentada a descrição detalhada do processo de criação dos códigos, com explicações e passo a passo. Em seguida, no Capítulo 5, são descritos e analisados os resultados obtidos. Por fim, o Capítulo 6 traz as conclusões, discutindo as implicações do estudo e as possibilidades de trabalhos futuros.



## 2. Referencial Teórico

### 2.1. Problemas NP-Completo

Problemas considerados sem solução eficiente são aqueles classificados como intratáveis, não se conhece um algoritmo determinístico capaz de resolvê-los em tempo polinomial. Esses problemas são de alta complexidade computacional e estão intimamente relacionados à classe de problemas conhecidos como NP-completos. Para compreender essa questão e a relação com os problemas NP-completos, é necessário explorar os seguintes conceitos fundamentais:

### 2.2. Algoritmos de tempo polinomial e exponencial

Em computação a importância do tempo de execução de um algoritmo é categórica. Desse modo, uma Função complexidade de tempo de um algoritmo expressa o tempo máximo necessário, dentre os possíveis comprimentos de entrada  $n$  que variam segundo os "tamanhos" dos casos do problema e segundo o "formato" fixado para medir o comprimento de entrada de cada caso.[\[2\]](#)

**Definição 1.1** Algoritmo de tempo polinomial é aquele cuja função complexidade de tempo é  $O(p(n))$  para alguma função polinomial  $p(n)$ , quando  $n$  é o comprimento da entrada.  
[\[2\]](#) Exemplos: pesquisa sequencial ( $O(n)$ ), ordenação por inserção ( $O(n^2)$ ), e multiplicação de matrizes ( $O(n^3)$ ).[\[1\]](#)

**Definição 1.2** Algoritmo de tempo exponencial é aquele cuja função complexidade de tempo não pode ser limitada por um polinômio.[\[2\]](#) Sua função de complexidade é  $O(c^n)$ ,  $c > 1$ .  
[\[1\]](#) Exemplo: Problema do Caixeiro Viajante (PCV) ( $O(n!)$ ).[\[1\]](#)

Com o crescimento do comprimento da entrada  $n$  de um problema a distinção entre o tempo de execução computacional utilizando algoritmos polinomiais e exponenciais se torna mais evidente. Problemas NP-completos demandam algoritmos de tempo exponencial. A figura 2 ilustra essa discrepância:

Função complexidade Tempo	Comprimento n					
	10	20	30	40	50	60
$n$	0,00001 seg	0,00002 seg	0,00003 seg	0,00004 seg	0,00005 seg	0,00006 seg
$n^2$	0,0001 seg	0,0004 seg	0,0009 seg	0,0016 seg	0,0025 seg	0,0036 seg
$n^3$	0,001 seg	0,008 seg	0,027 seg	0,064 seg	0,125 seg	0,216 seg
$n^5$	1 seg	3,2 seg	24,3 seg	1,7 min	5,2 min	13,0 min
$2^n$	0,001 seg	1,0 seg	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 seg	58 min	6,5 anos	3855 séc.	$2 \times 10^8$ séc.	$1,3 \times 10^{13}$ séc.

Figura 1 . Comparação entre os tempos de complexidade-funções polinomial x funções exponenciais.

Fonte: A teoria da complexidade computacional, 1987.

## 2.3. Algoritmos determinísticos e não-determinísticos

**Definição 1.3** Algoritmos determinísticos: pode-se prever o seu comportamento. Para as mesmas entradas de um problema, teremos sempre as mesmas saídas. Formalmente, um algoritmo determinístico computa uma função matemática; uma função tem um valor único para cada entrada dada, e o algoritmo é um processo que produz este valor em particular como saída.[5]

**Definição 1.4** Algoritmo não determinístico: é um algoritmo em que, dada uma certa entrada, pode apresentar comportamentos diferentes em diferentes execuções, ao contrário de um algoritmo determinístico. Em outras palavras, em complexidade computacional, algoritmos não determinísticos são aqueles que, a cada passo possível, podem permitir múltiplas continuações. [4]

O conceito de problemas sem uma solução eficiente refere-se àqueles problemas para os quais não existem algoritmos capazes de resolvê-los em tempo polinomial utilizando máquinas determinísticas. No entanto, inúmeros problemas intratáveis, encontrados na prática, são decidíveis. Ou seja, é possível resolvê-los, em tempo polinomial, com o auxílio de um computador não-determinístico. Estes problemas constituem-se na classe dos problemas NP.

[2]

## 2.4. Classes de complexidade

**Definição 1.5** Classe P: conjunto de todos os problemas que podem ser resolvidos por algoritmos determinísticos em tempo polinomial;

**Definição 1.6** Classe NP: (non deterministic polynomial time): consiste nos problemas que podem ser verificados em tempo polinomial;

**Definição 1.7** Classe NP-Completo: consiste nos problemas que demandam tempo exponencial para serem solucionados. Se um deles puder ser resolvido em tempo polinomial então todo problema NP-Completo terá uma solução em tempo polinomial. O problema do caixeiro viajante é um exemplo clássico desse tipo de classe;

**Definição 1.8** Classe NP-difícil: consiste quando todos os problemas em NP podem ser reduzidos a ele em tempo polinomial.

## 2.5. O problema do caixeiro viajante

O problema do caixeiro viajante (ou traveling salesman problem, TSP) pode ser associada a um viajante que pretende passar por um conjunto de cidades uma única vez e por fim voltar à cidade inicial, percorrendo a menor distância possível.[6] É um dos problemas mais estudados de otimização combinatória e tem uma grande variedade de aplicações. E foi formulado matematicamente pela primeira vez em 1932, por Karl Menger[7].

O problema diz-se simétrico (STSP) se  $c(a, b) = c(b, a)$ , para todo  $a$  e  $b$ , caso contrário, diz-se assimétrico (ATSP). Caso os custos/distâncias entre as cidades cumpram a norma euclidiana, o TSP diz-se euclidiano.[6] No contexto do problema presente neste artigo as distâncias serão dadas por métodos não-euclidianos, especificamente a geometria pombalina (fórmula:  $dT = |x_2 - x_1| + |y_2 - y_1|$ ). Por ser calculada em módulo, este problema pode ser considerado um TSP simétrico.

## 2.6. Algoritmos de Força Bruta e o problema do caixeiro viajante

Apesar de possuir um enunciado aparentemente simples, o problema do caixeiro viajante é classificado como NP-completo. A aplicação de um método de pesquisa exaustiva, ou força bruta, para resolvê-lo consiste em calcular todas as possíveis soluções (ou percursos) e selecionar, posteriormente, a solução que apresenta o menor custo total.

Entretanto, esse método torna-se rapidamente ineficiente à medida que o número de Cidades aumenta. Para um TSP em rede completa com  $n$  cidades, existem  $(n - 1)!$  possíveis soluções no caso geral ou  $\frac{(n-1)!}{2}$  no caso simétrico, em que a distância entre duas cidades é a

mesma independentemente da direção do percurso. Na seção de resultados, será apresentada uma tabela com informações específicas que ilustram o crescimento exponencial das possibilidades e o impacto na eficiência do algoritmo.

## **2.7. Algoritmos genéticos e o problema do caixeiro viajante**

Os algoritmos genéticos (GAs: Genetic Algorithms) são inspirados na teoria da evolução das espécies de Darwin e na genética. Se utilizam principalmente da probabilidade de um mecanismo de busca paralela, cuja característica é realizar diferentes cálculos no mesmo ou em diferentes conjuntos de dados e adaptativa, baseado no princípio de sobrevivência dos mais aptos e na reprodução. [9]

Esses princípios são replicados na criação de algoritmos computacionais que visam encontrar soluções mais eficazes para problemas específicos. No contexto do problema do caixeiro viajante, ao considerar um número significativo de cidades, localizar a melhor solução torna-se extremamente desafiador com um algoritmo de força bruta. Assim, algoritmos adaptativos, como os Algoritmos Genéticos (GAs), se mostram como a alternativa mais apropriada.

Os algoritmos genéticos englobam uma série de processos, incluindo representação, decodificação, avaliação, seleção, cruzamento e mutação, além de parâmetros fundamentais como o tamanho da população, taxa de reprodução, taxa de mutação e critério de parada. Um dos conceitos iniciais é a representação dos cromossomos, que são as soluções potenciais para o problema. Esses cromossomos podem ser apresentados em diversos formatos, como binário, números reais, permutações de símbolos, entre outros. Neste trabalho foi utilizado a permutação de símbolos. Por exemplo:

Indivíduo  $\rightarrow$  B D C A

A decodificação consiste na construção de uma solução para que o programa o avalie. Neste projeto, é utilizada uma matriz de pontos distintos e suas respectivas posições dentro da matriz. A partir dos pontos são criadas as permutações que serão utilizadas no processo posterior.

A avaliação se refere ao cálculo do fitness (aptidão) de cada indivíduo. No caso do caixeiro viajante, se trata da soma das distâncias dos caminhos de cada indivíduo. É a distância entre cada letra, calculada pela fórmula matemática:  $dT = |x_2 - x_1| + |y_2 - y_1|$ . Por exemplo:

Indivíduo  $\rightarrow$  B D C A  $\rightarrow$  14

O processo de seleção em algoritmos genéticos envolve a escolha de indivíduos para a

reprodução, baseando-se naqueles que apresentam o melhor desempenho, ou seja, os mais aptos. Neste contexto, o indivíduo mais apto é aquele que possui o menor caminho. Existem diversos métodos para realizar essa seleção, mas neste projeto optou-se pelo método da roleta. Nesse método, cada indivíduo é representado por uma fatia proporcional à sua aptidão relativa. O operador de seleção utilizado na roleta considera a soma do fitness total da população dividida pelo fitness de cada indivíduo, além de um piso e um valor fixo de 0,01. O piso é incremental: começa em zero e, a partir do segundo indivíduo, soma-se o valor do indivíduo anterior mais 0,01.

$$\text{piso} + \frac{\sum_{i=1}^n f_i \rightarrow \text{Soma total}}{f_i \rightarrow \text{indivíduo}} + 0,01$$

A ideia principal da roleta é a seleção aleatória de pares de cromossomos, que servirão de genitores de dois novos indivíduos criados a partir de seus genes no cruzamento. Quando um indivíduo é selecionado ele é retirado da roleta.

O operador de cruzamento (crossover) é considerado a característica fundamental dos GAs e ocorre logo após a seleção dos melhores indivíduos. Pares de genitores são escolhidos aleatoriamente da população, baseado na aptidão, e novos indivíduos são criados a partir da troca do material genético. Os descendentes serão diferentes de seus pais, mas com características genéticas de ambos os genitores [9]. Por exemplo:

*Pai 1* → *B D C | A E F*  
*Pai 2* → *A D B | E F C*  
  
*Filho 1* → *B D C E F C*  
*Filho 2* → *A D B A E F*

A abordagem mais simples consiste em selecionar um único ponto de corte aleatório (crossover de um ponto) e dividir os progenitores em duas partes. A criação dos filhos ocorre ao combinar as partes resultantes dos progenitores.

Os cromossomos criados a partir do operador de crossover são então submetidos a operação de mutação (essa podendo ou não ocorrer dependendo da taxa de probabilidade de mutação). Mutação é um operador exploratório que tem por objetivo aumentar a diversidade na população [8]. O operador de mutação neste projeto aconteceu selecionando dois pontos aleatórios do indivíduo filho e trocando suas posições de lugar.

Por fim, no algoritmo genético os parâmetros controlam o processo evolucionário:

- **Tamanho da População:** é o número de indivíduos que irão compor uma população;
- **Taxa de Crossover:** é a probabilidade de um indivíduo ser recombinado com outro. A taxa padrão utilizada neste projeto foi de 60%, ou seja, apenas sessenta por cento dos indivíduos da população poderão ser selecionados. Com uma taxa muito alta pode ocorrer perda de estruturas de alta aptidão, mas com um valor baixo, o algoritmo pode tornar-se muito lento; [10]
- **Taxa de Mutação:** é a probabilidade do conteúdo de uma posição/gene do cromossoma ser alterado. De modo geral, as taxas de mutação são muito baixas ( $<1\%$ ), pois taxas altas tornam as alterações essencialmente aleatórias [10]. A taxa padrão utilizada no projeto foi de 0,5%;
- **Número de Gerações:** total de ciclos de evolução de um GAs, ou o número de vezes que o algoritmo irá realizar todos os processos descritos anteriormente até retornar a melhor solução encontrada.

É por meio da evolução de gerações que encontrar uma melhor solução aproximada do problema do caixeiro viajante se torna viável, mesmo que o número de cidades presentes no problema seja extenso.

## 2.8. Algoritmos do vizinho mais próximo e o problema do caixeiro viajante

O algoritmo do Vizinho Mais Próximo (*Nearest Neighbor Algorithm*) é uma heurística simples e gulosa usada para resolver problemas de roteamento, como o Problema do Caixeiro Viajante (TSP). É chamado de "guloso" porque, a cada passo, toma a decisão que parece ser a melhor naquele momento, sem considerar o impacto global dessa decisão.

O algoritmo funciona construindo uma rota iterativamente, sempre escolhendo o próximo ponto mais próximo do ponto atual. Ele começa com a escolha de um ponto inicial, como a primeira cidade da lista. Em seguida, ele entra em um processo iterativo: a partir do ponto atual, calcula a distância para todos os pontos não visitados e escolhe o mais próximo como próximo destino. Esse ponto é adicionado à rota e marcado como visitado, e o ponto atual é atualizado para o novo ponto escolhido. Esse processo é repetido até que todos os pontos tenham sido visitados. Por fim, o algoritmo retorna ao ponto inicial para completar o ciclo. Essa abordagem heurística é simples e rápida, porém possui complexidade  $O(n^2)$ , as quais podem gerar soluções subótimas, especialmente em problemas maiores.

## 2.9. Algoritmos da otimização por colônia de formigas e o problema do caixeiro viajante

O Algoritmo Otimização por Colônia de Formigas (Ant Colony Optimization – ACO) é uma meta-heurística baseada em inteligência coletiva, inspirada no comportamento de formigas reais durante a busca por alimento. Inicialmente proposto por Marco Dorigo na década de 1990 [11], esse algoritmo tem sido amplamente empregado na resolução de problemas de otimização combinatória. Seu funcionamento se baseia na comunicação indireta entre as formigas por meio da deposição e evaporação de feromônios, um mecanismo biológico que permite que as colônias encontrem rotas eficientes entre o formigueiro e uma fonte de alimento. No contexto computacional, essa lógica é utilizada para determinar o melhor caminho em um problema de otimização.

A execução do ACO ocorre em etapas sucessivas. Inicialmente, uma população de formigas é criada e distribuída aleatoriamente no espaço de busca. Em seguida, cada formiga constrói uma solução, percorrendo um caminho definido com base em uma regra probabilística que leva em consideração a presença de feromônio e a heurística do problema. Após a construção das soluções, ocorre a avaliação, na qual a qualidade de cada trajeto é determinada com base no custo total da solução encontrada, como a distância percorrida em problemas de roteamento. Para aprimorar o processo de busca, os feromônios são atualizados ao final de cada iteração.

As formigas depositam feromônio nas rotas percorridas de forma proporcional à qualidade da solução encontrada, enquanto a evaporação do feromônio ocorre para evitar que soluções subótimas se perpetuem, promovendo a diversificação da busca. Esse ciclo de construção, avaliação e atualização dos feromônios é repetido por diversas iterações, até que um critério de parada seja atingido, como um número máximo de iterações, um tempo limite de execução ou a convergência para uma solução satisfatória.

O comportamento das formigas no ACO é modelado matematicamente por regras probabilísticas, as quais determinam o caminho que cada formiga seguirá durante a busca pela solução ótima. A escolha do próximo ponto  $y$  a partir do ponto  $x$  é determinada pela equação:

$$P_{xy}^k = \frac{[T_{xy}^{\square}(t)]^{\alpha} * [\eta_{xy}(t)]^{\beta}}{\sum_{t \in N_x^k} [T_{xy}^{\square}(t)]^{\alpha} * [\eta_{xy}(t)]^{\beta}}$$

Onde:

$P_{xy}^k$ : representa a probabilidade da formiga  $k$  escolher o ponto  $y$  partindo do ponto  $x$ ;

$T_{xy}^{\square}(t)$ : é o inverso da distância entre  $x$  e  $y$ ;

$\eta_{xy}(t)$ : representa a concentração de feromônio entre  $x$  e  $y$ ;

$\alpha$ : parâmetro que controla a influência da distância;

$\beta$ : parâmetro que controla a influência do feromônio;

$t$ : iteração.

Após todas as formigas completarem suas soluções, ocorre a atualização do feromônio, a qual é regida pela equação:

$$T_{xy}^k(t) = (1 - \sigma) * T_{xy}^k(t - 1) + \sum_{k=1}^m \Delta T_{xy}^k(t)$$

Onde:

$\sigma$ : representa a taxa de evaporação do feromônio, evitando que soluções subótimas se perpetuem ao longo das iterações.

A quantidade de feromônio depositado pela formiga  $k$  é dada por:

$$\Delta T_{xy}^k = \frac{Q}{dk}$$

sendo  $Q$  uma constante de reforço do feromônio e  $dk$  a distância total percorrida pela formiga  $k$ .

Esse modelo permite que as rotas mais curtas e eficientes acumulem maior quantidade de feromônio ao longo do tempo, atraindo mais formigas para esses trajetos e conduzindo o algoritmo à convergência para uma solução de alta qualidade.

### 3. Trabalhos relacionados

Rodrigues (2021) [12] propôs uma abordagem para otimizar a logística empresarial, com foco na redução dos custos de transporte. O autor utilizou programação linear em conjunto com o Solver para identificar a rota mais eficiente entre cinco cidades próximas. Embora essa técnica apresente bons resultados para pequenos conjuntos de dados, o Problema do Caixeiro Viajante (TSP - *Traveling Salesman Problem*) é classificado como NP-difícil. Isso significa que, conforme aumenta o número de cidades, o crescimento exponencial das variáveis e restrições torna inviável a aplicação exclusiva de programação linear, devido ao alto custo computacional.

No estudo conduzido por Silva et al. (2013) [13], foram analisadas diferentes heurísticas construtivas voltadas à otimização de rotas no transporte de cargas. Para avaliar o desempenho dessas abordagens, os pesquisadores testaram quatro métodos: vizinho mais



próximo, inserção do mais distante, inserção do mais rápido e inserção do mais próximo. Os testes foram realizados por meio de um aplicativo móvel que coletou coordenadas geográficas das rotas experimentadas. A análise comparativa revelou que a heurística de inserção do mais distante obteve os melhores resultados na definição de trajetos otimizados.

Calado e Ladeira (2011) [14] exploraram distintas estratégias para resolver o Problema do Caixeiro Viajante, comparando algoritmos genéticos, redes neurais e uma heurística específica. A pesquisa foi conduzida com três instâncias do problema, extraídas da base TSPLIB. Os resultados apontaram que o algoritmo genético teve melhor desempenho em relação às demais abordagens. No entanto, os autores destacam que a eficácia desse método depende diretamente da configuração dos operadores de seleção e mutação, fatores que influenciam significativamente a qualidade das soluções obtidas.

Já Silva (2022) [15] investigou a aplicação da meta-heurística de Colônia de Formigas (ACO) para resolver o Problema do Caixeiro Viajante em uma empresa distribuidora de laticínios localizada em Angicos/RN. O estudo avaliou a eficiência do método com base na qualidade das rotas geradas e no tempo de processamento do algoritmo. A análise, realizada com 48 cidades, demonstrou que a abordagem proposta se mostrou competitiva e eficiente quando comparada a outros métodos encontrados na literatura.

Além disso, observou-se que o desempenho do algoritmo depende do ajuste adequado dos parâmetros, fator essencial para aprimorar a busca por soluções otimizadas. Os pesquisadores sugerem que futuras investigações podem explorar ajustes adicionais e novas estratégias para aperfeiçoar ainda mais os resultados obtidos. Assim, o estudo atingiu seus objetivos ao desenvolver, aplicar e validar o modelo baseado em Colônia de Formigas, demonstrando seu potencial para otimizar rotas logísticas.

Diante dessas considerações, este trabalho propõe o uso de métodos heurísticos como alternativa eficiente para encontrar soluções aproximadas do ótimo em menos tempo, especialmente para instâncias mais complexas do problema.

## **4. Metodologia**

### **4.1. Algoritmo de Força Bruta**

Nesta versão do projeto, a linguagem de programação Python foi utilizada para implementar o algoritmo de força bruta. Durante a construção do código, foram aplicados conceitos fundamentais de programação, como estruturas condicionais, laços de repetição, matrizes, dicionários e funções. Além disso, testes foram conduzidos utilizando arquivos no formato .txt elaborados especificamente para o experimento.

## **4.2. Algoritmo Genético**

A elaboração deste algoritmo tem como objetivo desenvolver um método mais eficiente ao problema do caixeiro viajante, utilizando uma solução adaptativa.

Para a implementação do algoritmo genético foram utilizadas a linguagem de programação Python e a biblioteca random para a geração de números aleatórios necessários ao algoritmo.

## **4.3. Algoritmo do vizinho mais próximo**

O algoritmo do vizinho mais próximo é uma abordagem heurística para resolver o problema do caixeiro viajante de forma eficiente, priorizando a simplicidade e a rapidez computacional. Este método utiliza uma estratégia gulosa, onde a cada passo é selecionado o ponto mais próximo do local atual, construindo incrementalmente uma rota sem repetição de pontos. O método complementa as demais abordagens implementadas no projeto, como o algoritmo genético, permitindo uma análise comparativa das eficiências e limitações de cada técnica na resolução do problema proposto.

## **4.4. Algoritmo de colônia de formigas**

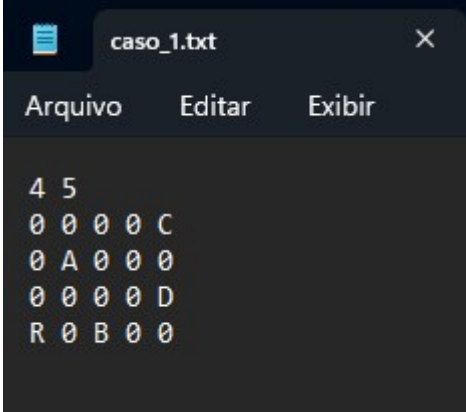
O Algoritmo de Colônia de Formigas (ACO) é uma abordagem meta-heurística inspirada no comportamento coletivo das formigas para resolver problemas de otimização combinatória, como o Problema do Caixeiro Viajante. Diferente de métodos puramente gulosos, o ACO utiliza um mecanismo de aprendizado baseado na deposição e evaporação de feromônios, permitindo que as formigas explorem múltiplas rotas e, gradativamente, favoreçam os caminhos mais eficientes. Esse processo possibilita um equilíbrio entre exploração e intensificação, permitindo a descoberta de soluções próximas ao ótimo global.

## **5. Experimentos**

Essa seção apresenta o passo a passo realizado para a construção dos algoritmos de força bruta, vizinho mais próximo, genético e colônia de formiga para a resolução do problema do Flyfood.

## 5.1. Algoritmo de Força bruta

Para a parte inicial do algoritmo de força bruta foi utilizada uma função para realizar a leitura de uma matriz a partir de um arquivo .txt.



```
caso_1.txt
Arquivo  Editar  Exibir

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

Figura 2.Exemplo de entrada do algoritmo.

O arquivo contém a descrição da matriz utilizada no problema, com informações sobre sua dimensão: 4 linhas e 5 colunas. Logo abaixo, está representada a matriz, incluindo o local de origem, identificado pela coordenada R, e os pontos de entrega, representados pelas coordenadas A, B, C e D. Essa configuração estabelece a base para a aplicação do algoritmo, permitindo o cálculo das distâncias entre os pontos e a determinação do percurso mais eficiente.

### 5.1.1. Descrição do código

O código apresenta uma solução por força bruta para o problema do "Caminho do Viajante" (ou Problema do Caixeiro-Viajante), adaptado para o contexto de entregas por drone. A implementação segue etapas bem definidas, que analisam todas as possíveis rotas para determinar o trajeto mais eficiente. Ao final, uma figura ilustra o funcionamento e os resultados descritos.

#### A) Importação e inicialização

O código importa bibliotecas essenciais, como as para manipulação de arquivos e *matplotlib.pyplot* para visualização de gráficos. É utilizado um diretório chamado *matrizes* que contém arquivos de texto representando matrizes de pontos de entrega.

#### B) Funções auxiliares

- **matrizes(*dir\_mat*):** Lista os arquivos no diretório fornecido (*dir\_mat*),

- organizando-os em um dicionário numerado e retorna um dicionário {número: nome\_arquivo};
- **sel\_arq\_matriz(dir\_mat):** Exibe os arquivos listados pela função anterior. Permite que o usuário escolha interativamente uma matriz para análise, retornando o nome do arquivo selecionado;
  - **geo\_taxi(x1, y1, x2, y2):** A geometria do táxi calcula a distância entre dois pontos (x1, y1) e (x2, y2):  $dT = |x2-x1| + |y2-y1|$ ;
  - **permutacoes(lista):** Calcula todas as permutações possíveis dos elementos da lista fornecida usando recursão. Essa etapa é necessária para testar todas as rotas possíveis para entrega;
  - **rota(pt\_a, pt\_b, visitados):** Gera a rota ponto a ponto entre dois locais (pt\_a) e (pt\_b), movendo-se primeiro verticalmente e depois horizontalmente. Evitando revisitar locais já percorridos;
  - **rota\_completa(caminho, coord\_pts):** Constroi a rota completa, incluindo a volta ao ponto inicial ('R'), com base em um caminho e nas coordenadas dos pontos;
  - **plotar(caminho, coord\_pts):** Gera um gráfico visual da rota calculada, indicando os pontos de entrega e as conexões entre eles.

### C) Função principal (main)

A) **seleção e leitura da matriz:** A matriz selecionada é lida de um arquivo de texto. A primeira linha especifica o número de linhas e colunas e as linhas subsequentes formam a matriz, onde:

- ✓ '0' representa espaço vazio;
- ✓ Letras representam pontos de entrega ('R' é o ponto inicial).
- **identificação dos pontos e coordenadas:** O código percorre a matriz, identificando pontos de entrega e suas respectivas coordenadas no plano cartesiano.
- **geração de permutações:** Usa a função *permutacoes(lista)* para criar todas as sequências possíveis dos pontos de entrega, permitindo analisar cada rota.
- **cálculo do melhor caminho:** Para cada permutação é calculada a distância total da rota, incluindo o retorno ao ponto inicial ('R'). Sendo atualizado o melhor caminho e sua distância se a rota for mais curta.

- **Resultados e visualização:** Exibe o melhor caminho (sequência de pontos de entrega), a distância total percorrida, o tempo de execução e plota o gráfico da rota.

#### D) Execução

- O código é executado ao chamar `main()` no bloco `if __name__ == '__main__':`.

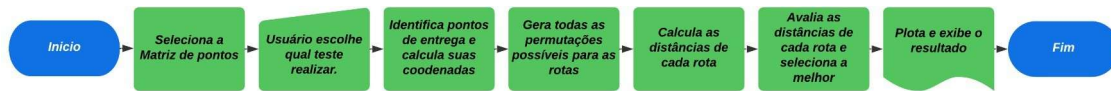


Figura 3. Resumo do fluxo do código-fonte

Na função principal do programa, denominada *main()*, é conduzido o fluxo principal das operações de otimização de rotas. Inicialmente, o programa solicita ao usuário que selecione uma matriz para análise. Para isso, utiliza a função *sel\_arq\_matriz()*, que lista os arquivos disponíveis no diretório especificado e valida a escolha do usuário. Após a seleção, o programa lê o conteúdo do arquivo correspondente e armazena as informações da matriz em uma lista estruturada.

Posteriormente, o programa realiza a identificação dos pontos de entrega e suas respectivas coordenadas. A matriz é percorrida linha a linha, e cada elemento distinto de zero é reconhecido como um ponto de entrega. Suas coordenadas são armazenadas em um dicionário chamado *coord\_pts*, no qual as chaves correspondem aos identificadores dos pontos (como letras) e os valores são suas coordenadas na matriz. Simultaneamente, os pontos de entrega são adicionados à lista *pts\_entrega*, enquanto o ponto de partida, identificado pela letra "R", é armazenado separadamente.

```

# Lendo o arquivo da matriz
with open(f'{dir_mat}/{arq_matriz}', 'r') as file:
    linhas = file.read().splitlines()

# Definindo o número de linhas e colunas da matriz
num_linhas_matriz, num_colunas_matriz = [int(x) for x in linhas[0].split()]
matriz = [linha.split() for linha in linhas[1 : num_linhas_matriz + 1]] # Lê a matriz a partir da segunda linha do arquivo

# Início da execução do cálculo de caminhos
import time
start_time = time.time()

# Busca na matriz e guarda as coordenadas dos pontos em um dicionário
pts_entrega = [] # Armazena letras diferentes de '0' (pontos de entrega)
coord_pts = {} # Armazena coordenadas dos pontos

# Itera sobre a matriz para encontrar os pontos de entrega e suas coordenadas
for l in range(num_linhas_matriz):
    for c in range(num_colunas_matriz):
        letra = matriz[l][c] # Obtem a letra na posição (x, y)
        if letra != '0': # Armazena a coordenada se for diferente de '0'
            x = c
            y = (num_linhas_matriz - 1) - l # O número da linha é o inverso da coordenada y
            coord_pts[letra] = (x, y) # Armazena a coordenada em forma de tupla
            if letra != 'R': # Adiciona as letras diferentes de 'R' na lista
                pts_entrega.append(letra)

# Calcula das permutações dos elementos de entrega
perm_pts = permutacoes(pts_entrega) # Gera todas as permutações das letras
  
```

Figura 4. Algoritmo de força bruta - Leitura da matriz e permutações.

Com os pontos de entrega identificados, o programa passa a calcular todas as possíveis ordens de visita dos pontos, utilizando a função *permutacoes()*. Este cálculo gera todas as permutações possíveis dos pontos de entrega, representando diferentes sequências de trajetos que o drone pode seguir. Cada permutação é avaliada individualmente para determinar a distância total associada ao trajeto.

A avaliação das permutações é realizada em um loop, no qual o programa calcula a distância total de cada rota utilizando a função *geo\_taxi()*. Este cálculo abrange a distância entre o ponto de partida e o primeiro ponto de entrega, as distâncias entre pontos consecutivos na permutação e, por fim, a distância de retorno ao ponto de partida. Durante este processo, o programa mantém o registro do menor trajeto encontrado e da menor distância associada, atualizando estas variáveis sempre que uma rota mais eficiente é identificada.

```
# Busca pelo melhor caminho
dist_menor_caminho = float('inf') # Inicia com infinito para garantir que qualquer distancia encontrada seja a menor
menor_caminho = []

# Coordenadas do ponto de partida R
pt_r_x, pt_r_y = 0, 0

# Calculo das distancias e busca do melhor caminho
for caminho in perm_pts:
    soma_dist = 0 # Soma das distancias para o caminho atual

    # Coordenadas do primeiro ponto do caminho
    pt_0 = caminho[0]
    pt_0_x, pt_0_y = coord_pts[pt_0][0], coord_pts[pt_0][1]

    # Distancia do ponto 'R' ao primeiro ponto do caminho
    soma_dist += geo_taxi(pt_r_x, pt_r_y, pt_0_x, pt_0_y)

    # Distancias entre os pontos na permutacao
    for x in range(len(caminho) - 1):
        pt_a = caminho[x] # Ponto atual
        pt_a_x, pt_a_y = coord_pts[pt_a][0], coord_pts[pt_a][1]

        pt_b = caminho[x + 1] # Proximo ponto
        pt_b_x, pt_b_y = coord_pts[pt_b][0], coord_pts[pt_b][1]

        soma_dist += geo_taxi(pt_a_x, pt_a_y, pt_b_x, pt_b_y)
```

**Figura 5. Algoritmo de força bruta: Cálculo das distâncias dos pontos e verificação do melhor**

Ao término da execução, o programa apresenta os resultados ao usuário. O menor trajeto encontrado é exibido em forma de uma sequência de pontos, acompanhado da distância total do percurso e do tempo de execução do algoritmo. Além disso, a função *plotar()* é chamada para gerar um gráfico que ilustra o caminho percorrido, destacando visualmente os pontos de entrega e as conexões entre eles. Essa abordagem permite ao usuário compreender de forma clara e visual a solução proposta pelo algoritmo para o problema de otimização de rotas.

```
def plotar(caminho, coord_pts):
    coordenadas = rota_completa(caminho, coord_pts) # Obtém todas as coordenadas do caminho
    coordenadas.append((0, 0)) # Retornando ao ponto 'R' # Adiciona o ponto de partida para fechar o caminho
    x_coords, y_coords = [], []
    for ponto in coordenadas:
        x_coords.append(ponto[0])
        y_coords.append(ponto[1])

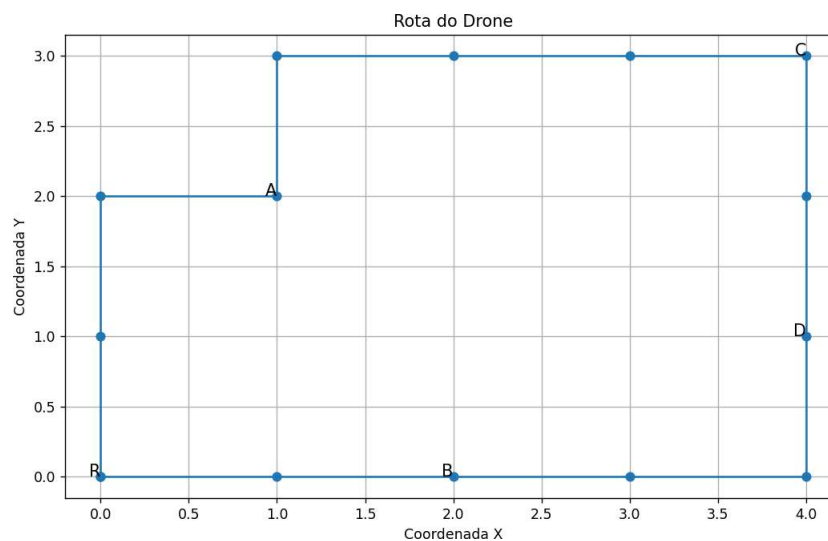
    # Criação do gráfico da rota
    import matplotlib.pyplot as plt
    plt.figure(figsize=(10, 6))
    plt.plot(x_coords, y_coords, marker='o', linestyle='--') # Plota a rota com marcadores nos pontos
    plt.title('Rota do Drone') # Título do gráfico
    plt.xlabel('Coordenada X') # Rotulo do eixo X
    plt.ylabel('Coordenada Y') # Rotulo do eixo Y

    # Adiciona os pontos de entrega ao gráfico
    for ponto, (x, y) in coord_pts.items():
        plt.text(x, y, ponto, fontsize=12, ha='right') # Adiciona o texto com o nome do ponto

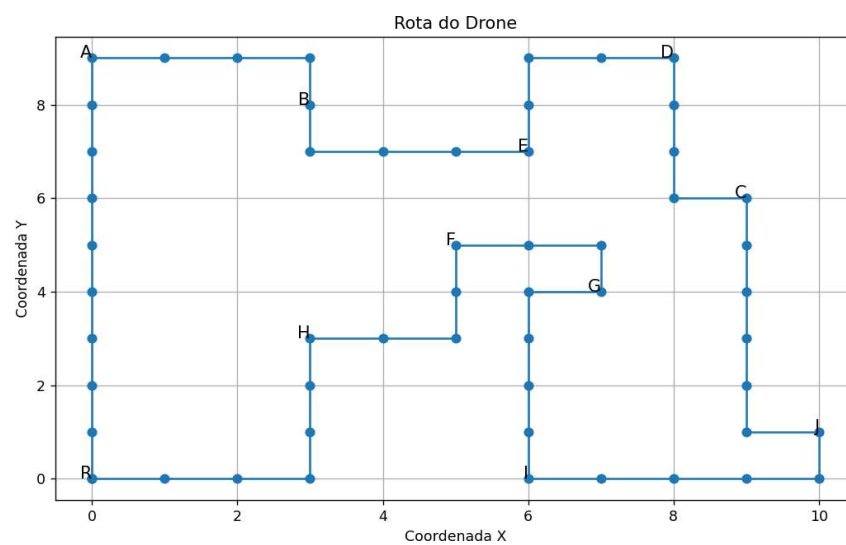
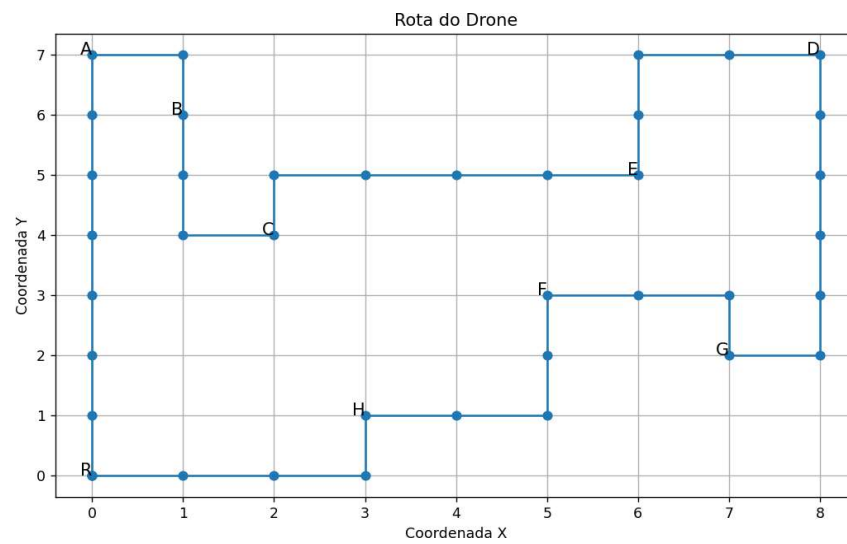
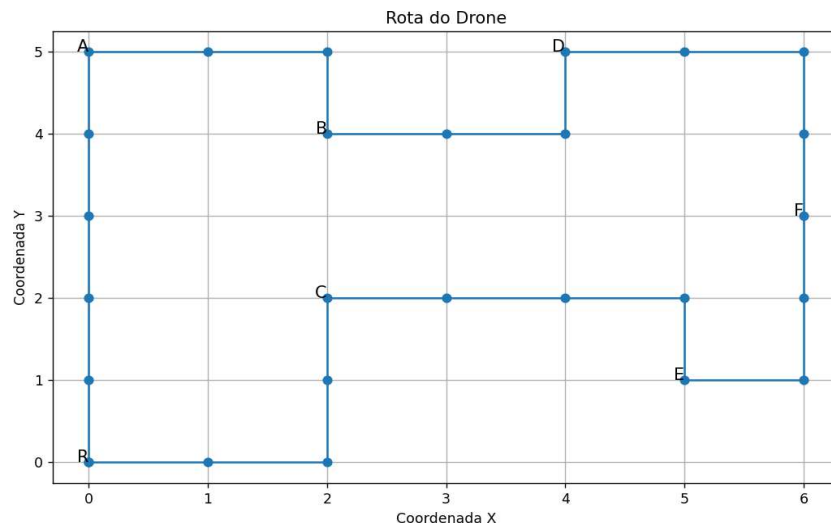
    plt.grid() # Adiciona uma grade ao gráfico
    plt.show() # Exibe o gráfico
```

**Figura 6. Algoritmo de força bruta: Saída do código**

Foram utilizados cinco arquivos de teste, cada um contendo uma matriz com um número específico de pontos de entrega. As matrizes foram configuradas com, respectivamente, quatro, seis, oito, nove e doze pontos de entrega. Essa variação permite avaliar o desempenho do algoritmo de força bruta em diferentes escalas, ilustrando o impacto do crescimento exponencial no número de possibilidades à medida que o número de pontos aumenta.



**Figura 7. Rota do caso 1**





## 5.2. Algoritmo do vizinho mais próximo

O algoritmo do Vizinho Mais Próximo é uma heurística gulosa, projetada para reduzir a complexidade computacional em comparação com abordagens exaustivas, como a força bruta. A implementação desenvolvida suporta arquivos no formato TSPLIB, compatível com instâncias baseadas em coordenadas geográficas ou matrizes de distâncias explícitas, garantindo flexibilidade na entrada de dados.

### 5.2.1 Descrição do código

#### A) Importação e inicialização

O código importa bibliotecas essenciais, como *os* para manipulação de arquivos e *time* para medição de tempo de execução. É utilizado um diretório chamado *tsp* que contém arquivos no formato TSPLIB, incluindo instâncias com coordenadas geográficas ou matrizes de distâncias explícitas.

#### B) Funções auxiliares

- **arquivos\_tsp(dir\_tsp):** Lista os arquivos no diretório *dir\_tsp*, organizando-os em um dicionário numerado e retorna um dicionário {número: nome\_arquivo}.
- **sel\_arq\_tsp(dir\_tsp):** Exibe os arquivos disponíveis e valida a entrada do usuário via prompt, garantindo que apenas números correspondentes a arquivos listados sejam aceitos.
- **distancia\_euclidiana(x1, y1, x2, y2):** Implementa a fórmula  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$  para calcular a distância euclidiana entre dois pontos, essencial para instâncias baseadas em coordenadas.
- **roteamento\_vizinho\_mais\_proximo(pontos, matriz):** Recebe como entrada um dicionário pontos (com identificadores de nós como chaves e tuplas de coordenadas geográficas x, y ou valores None para casos não geográficos) e uma matriz de distâncias pré-calculada matriz (opcional). O processamento inicia selecionando o primeiro nó como origem, definindo a rota inicial e a lista de nós não visitados. Em seguida, executa uma iteração gulosa: para cada nó atual, calcula as distâncias aos nós não visitados utilizando a função *distancia\_euclidiana* para coordenadas geográficas ou consultando diretamente a matriz fornecida em casos não geográficos, selecionando sempre o nó mais próximo para adicionar à rota e acumular a distância total. Após visitar todos os nós, o algoritmo fecha o ciclo retornando ao ponto de origem. A saída final consiste em uma lista representando a rota completa e o valor numérico da distância total percorrida.

### **C) Leitura de Arquivos TSP**

A leitura de arquivos TSP é adaptada para dois tipos de instâncias: com coordenadas geográficas ou matrizes de distâncias explícitas. Para instâncias com coordenadas, o algoritmo identifica a seção `NODE_COORD_SECTION` no arquivo, extraindo os dados e armazenando-os em um dicionário no formato `{'nó': (x, y)}`, permitindo cálculos de distância euclidiana posteriormente. Já para instâncias com matrizes pré-definidas, a seção `EDGE_WEIGHT_SECTION` é localizada, e os valores numéricos são processados para construir uma matriz simétrica  $n \times n$ , onde  $n$  corresponde à dimensão declarada no cabeçalho do arquivo. Essa matriz é armazenada como uma lista de listas, garantindo acesso rápido às distâncias via índices inteiros. Essa estruturação permite que o algoritmo trate ambos os formatos de entrada de forma eficiente, adaptando-se às especificações do problema TSP em questão.

### **D) Função principal**

- **Seleção e leitura do arquivo TSP:**

- A) O usuário escolhe um arquivo TSP do diretório `tsp` via interface interativa.

- B) O arquivo é processado conforme seu formato:

- Coordenadas geográficas: Extrai nós e suas coordenadas da seção `NODE_COORD_SECTION` (ex: nó '1': (2.5, 4.1)).
    - Matriz de distâncias: Constrói matriz simétrica a partir da seção `EDGE_WEIGHT_SECTION`.

- **Processamento dos dados:**

- A) Para coordenadas: Armazena nós em um dicionário pontos (chave = ID, valor = coordenadas).

- B) Para matrizes: Define nós como IDs sem coordenadas e preenche a matriz de distâncias.

- **Execução do algoritmo:**

- C) Define o primeiro nó como ponto inicial (ex: '1').

- D) Constrói a rota iterativamente, selecionando o vizinho mais próximo não visitado em cada etapa.

- E) Retorna à origem após incluir todos os nós, fechando o ciclo.

- **Cálculo e resultados:**

**F)** Mede o tempo de execução com `time.time()`.

**G)** Exibe o melhor caminho (sequência de pontos de entrega), a distância total percorrida e o tempo de execução em milissegundos.

**E) Execução**

Na função `principal()`, o algoritmo inicia com a seleção do arquivo TSP por meio da função `sel_arq_tsp('tsp')`, que lista os arquivos disponíveis no diretório especificado e valida a entrada do usuário. Após a escolha, o arquivo é lido e processado conforme seu formato. Para instâncias baseadas em coordenadas geográficas, a seção `NODE_COORD_SECTION` é identificada, e os dados são armazenados em um dicionário no formato `{'nó': (x, y)}`, onde cada chave representa um ponto e seu valor corresponde às coordenadas cartesianas. Em casos de matrizes de distâncias explícitas, a seção `EDGE_WEIGHT_SECTION` é parseada, e uma matriz simétrica é construída para representar as distâncias entre todos os pares de nós, garantindo acesso eficiente via índices inteiros.

Com os dados carregados, o algoritmo inicia a construção da rota. O primeiro nó é definido como ponto de partida, e uma lista de nós não visitados é criada. Em cada iteração, o nó atual é comparado com os nós restantes para identificar o mais próximo. Para instâncias geográficas, a distância é calculada usando a função `distancia_euclidiana()`, que aplica a fórmula matemática padrão da geometria euclidiana. Em matrizes explícitas, a distância é recuperada diretamente da estrutura pré-computada. O nó selecionado é adicionado à rota, removido da lista de não visitados e passa a ser o novo nó atual. Esse processo se repete até que todos os nós sejam incluídos na rota.

Após a inclusão do último nó, o algoritmo fecha o ciclo retornando ao ponto de origem, calculando a distância final entre o último nó visitado e a origem. A rota completa é representada como uma sequência de nós e a distância total é acumulada durante o processo. O tempo de execução é medido utilizando a biblioteca `time`, com início e fim da cronometragem encapsulando a chamada da função `roteamento_vizinho_mais_proximo()`.

## **5.3. Algoritmo genético**

### **5.3.1. Descrição do código**

Na figura 12 está apresentada a função principal do algoritmo genético, onde é criada a população inicial a partir da lista dos pontos ordenados, em seguida é calculado o fitness dessa população inicial. Posteriormente, é iniciada a parte do código que será repetida até atingir o critério de parada (iterações), em que ocorrerão as chamadas das funções.

```

1  import random
2  import os
3  import time
4
5  # Função que retorna um dicionário de arquivos TSP a partir de um diretório
6  def arquivos_tsp(dir_tsp):
7      # Lista os arquivos no diretório e cria um dicionário numerado
8      dic_arquivos = {str(i + 1): arquivo for i, arquivo in enumerate(sorted(os.listdir(dir_tsp)))}
9      return dic_arquivos
10
11 # Função para selecionar um arquivo TSP do diretório
12 def sel_arq_tsp(dir_tsp):
13     dic_arquivos = arquivos_tsp(dir_tsp)
14     print('\n Arquivos TSP localizados: \n')
15     for numero, arquivo in dic_arquivos.items():
16         print(f'{numero} - {arquivo}') # Exibe os arquivos disponíveis
17     print()
18     num_arquivo = 0
19     while num_arquivo not in dic_arquivos: # Escolha do arquivo que será testado
20         num_arquivo = input('Informe o número do arquivo TSP que deseja analisar: ')
21     return dic_arquivos[num_arquivo] # Retorna o arquivo escolhido
22
23 # Calcula a distância euclidiana entre dois pontos
24 def distancia_euclidiana(x1, y1, x2, y2):
25     return ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5
26

```

**Figura 11. Algoritmo Genético - Função algoritmo genético**

A soma total dos fitness é calculada e a população inicial ordenada do menor fitness para o maior. Em seguida, é calculada a porcentagem de cada indivíduo em relação à soma total, logo após, é calculado seu valor inverso. Então é iniciado o método da roleta, onde os indivíduos são escolhidos aleatoriamente e as chances de escolhas são ditas pela porcentagem inversa do fitness de cada indivíduo. Após retornar os pares dos pais segue para o crossover, e logo após o ajuste populacional - onde serão eliminados os piores indivíduos aleatoriamente por competição - e finalmente a escolha do melhor indivíduo.

Na figura 13, o processo do cálculo dos fitness é o mesmo descrito no algoritmo bruto, com a exceção de que cada fitness de um indivíduo calculado será armazenado em uma lista chamada `fitnessIndividuos` junto com o indivíduo em si. EX: [14, ['A', 'B', 'C', 'D']]. E em seguida armazenado em uma lista maior chamada `fitnessPopulacao`. No final do looping será retornada a lista com todos os indivíduos e seus respectivos fitness calculados. Dentro do cálculo do fitness é feita a chamada da função `GeoTaxi()`, descrita na figura 13.

```

# Cálculo do fitness (Aptidão)
def fitness(dic_pontos, matriz_distancias, tipo_distancia, populacao):
    fitness_populacao = []
    for individuo in populacao:
        soma_dist = 0.0
        for j in range(len(individuo)):
            pt_atual = individuo[j]
            if j == len(individuo) - 1:
                pt_proximo = individuo[0] # Fecha o ciclo
            else:
                pt_proximo = individuo[j + 1]
            if tipo_distancia == 'EUC_2D':
                x1, y1 = dic_pontos[pt_atual]
                x2, y2 = dic_pontos[pt_proximo]
                dist = distancia_euclidiana(x1, y1, x2, y2)
            elif tipo_distancia == 'EXPLICIT':
                i = int(pt_atual) - 1
                j_idx = int(pt_proximo) - 1
                dist = matriz_distancias[i][j_idx]
            soma_dist += dist
        fitness_individuo = [soma_dist, individuo]
        fitness_populacao.append(fitness_individuo)
    fitness_populacao.sort(key=lambda x: x[0]) # Ordena pelo menor fitness
    return fitness_populacao

```

Figura 12. Algoritmo Genético - Função fitness

A figura 14 representa o método da roleta, que recebe os parâmetros, população e taxa de reprodução. É criada uma cópia da população, pois é necessário a remoção dos indivíduos durante o processo, sem que altere a lista original da população. A taxa é calculada multiplicando a quantidade total de indivíduos na população pela taxa de reprodução dividido por dois e dividido por cem, pois como serão selecionados pares de pais, cem por cento da população é o mesmo que a metade dos indivíduos, ou seja, o para selecionar cem por cento dos indivíduos o while rodará 50 vezes. A lógica é a mesma para qualquer porcentagem. O primeiro for representa a quantidade de pais que serão selecionados a cada while, a variável limite tem o papel de limitar o espaço da roleta pela metade, com o objetivo de selecionar os melhores de indivíduos e o segundo for fará o papel de busca pelo indivíduo selecionado na roleta. Os pais selecionados serão armazenados em uma lista chamada pais e no fim do looping retornados.

```

# Método da roleta
def roleta(populacao, tx_de_reproducao):
    pop = populacao.copy()
    cont = 0
    pais = []
    taxa = int(len(populacao) * ((tx_de_reproducao/2) / 100))

    while cont < taxa:
        pai = []
        for _ in range(2):
            limite = pop[int(len(pop)/2)][3] if len(pop) > 0 else 0
            roleta_ponteiro = round(random.uniform(0, limite), 2)
            for j in range(len(pop)):
                if j == 0:
                    if roleta_ponteiro <= pop[j][3]:
                        pai.append(pop[j])
                        pop.pop(j)
                        break
                else:
                    if pop[j-1][3] < roleta_ponteiro <= pop[j][3]:
                        pai.append(pop[j])
                        pop.pop(j)
                        break
            pais.append(pai)
        cont += 1
    return pais

```

**Figura 13. Algoritmo Genético - método da roleta**

A figura 15 é representada pelo crossover. A função receberá os parâmetros pais e probabilidade de mutação. O parâmetro pais será a lista criada no método da roleta e a probabilidade de mutação é a mesma descrita no início do código, igual a 0,5%. A princípio é selecionado aleatoriamente um ponto de corte, este ponto servirá para dividir os pais. O primeiro filho1 receberá a primeira parte do pai1 e a segunda parte do pai2, já no filho2 ocorrerá o inverso. Em seguida são chamadas as funções mutação() e organizarFilho(). Os dois filhos então serão armazenados na lista novaPopulacao e por fim são ordenados do menor para o maior, após o cálculo dos seus fitness.

```

# Crossover
def crossover(dic_pontos, matriz_distancias, tipo_distancia, pais, prob_de_mutacao):
    nova_populacao = []
    ponto_corte = random.randint(1, len(pais[0][0][1])-1) if pais else 0
    for par in pais:
        if len(par) < 2:
            continue
        pai_1 = par[0][1]
        pai_2 = par[1][1]
        filho_1 = pai_1[:ponto_corte] + pai_2[ponto_corte:]
        filho_2 = pai_2[:ponto_corte] + pai_1[ponto_corte:]
        filho_1 = mutacao(filho_1, prob_de_mutacao)
        filho_2 = mutacao(filho_2, prob_de_mutacao)
        organizar_filho(pai_1, filho_1)
        organizar_filho(pai_2, filho_2)
        nova_populacao.append(filho_1)
        nova_populacao.append(filho_2)
    nova_populacao = fitness(dic_pontos, matriz_distancias, tipo_distancia, nova_populacao)
    return nova_populacao

```

**Figura 14. Algoritmo Genético – crossover**

Na figura 16 está representada a função `mutacao()`, que receberá os parâmetros `filho` e `taxa de mutação`. Seu propósito é gerar uma maior aleatoriedade de indivíduos dentro da nova população. É gerado um número aleatório entre 0 e 1. Se a taxa for menor que 0.5 então serão escolhidos dois pontos aleatórios desse indivíduo. Em seguida, é feita a troca da posição de um ponto pela outra. Caso contrário, o filho não será alterado.

```

# Mutação
def mutacao(filho, tx_de_mutacao):
    if random.uniform(0.0, 1.0) < tx_de_mutacao:
        id1, id2 = random.sample(range(len(filho)), 2)
        filho[id1], filho[id2] = filho[id2], filho[id1]
    return filho

```

**Figura 15. Algoritmo Genético - mutação**

A figura 17 apresenta a função `organizarFilho()`, cujos parâmetros são `pai` e `filho`. Seu propósito é organizar os filhos que possuem genes repetidos. Primeiro é feita a busca pelas letras repetidas no filho utilizando um laço `for`, em seguida, é encontrada as letras que estão faltando nele. Depois é feita a busca e armazenamento dos índices das letras repetidas do indivíduo filho. Logo em seguida, são realizadas as trocas das letras repetidas pelas que estão faltando.



```

# Organiza o filho para remover repetições
def organizar_filho(pai, filho):
    faltantes = [p for p in pai if p not in filho]
    duplicados = []
    visto = set()
    for i, p in enumerate(filho):
        if p in visto:
            duplicados.append(i)
        else:
            visto.add(p)
    for i in duplicados:
        if faltantes:
            filho[i] = faltantes.pop()
    return filho

```

**Figura 16. Algoritmo Genético - Removendo genes repetidos**

A função ajuste populacional é iniciada apenas quando houver a junção da nova população com a anterior e feita sua ordenação. Seu objetivo é eliminar os piores indivíduos aleatoriamente, de maneira que a nova população volte a seu tamanho inicial. Os parâmetros da função são: população e tamanho da população. É feita uma competição entre dois indivíduos aleatórios da lista população; o pior indivíduo é eliminado.

```

# Ajuste da população
def ajuste_populacional(populacao, tamanho_populacao):
    while len(populacao) > tamanho_populacao:
        i = random.randint(0, len(populacao)-1)
        populacao.pop(i)
    return populacao

```

**Figura 17. Algoritmo Genético - ajuste populacional**

Por fim, após a função do algoritmo genético atingir o critério de parada, é retornada a melhor solução.



```

# Algoritmo genético
def alg_genetico(dic_pontos, matriz_distancias, tipo_distancia, pontos,
tamanho_populacao, tx_de_reproducao, prob_de_mutacao, criterio_de_parada):
    # População inicial
    populacao = []
    for _ in range(tamanho_populacao):
        individuo = random.sample(pontos, len(pontos))
        populacao.append(individuo)
    populacao = fitness(dic_pontos, matriz_distancias, tipo_distancia, populacao)

    for geracao in range(criterio_de_parada):
        # Calcula fitness total e probabilidades
        fitness_total = sum(ind[0] for ind in populacao)
        for ind in populacao:
            ind.append(round(fitness_total / ind[0], 2))
            ind.append(round(ind[2] + (populacao.index(ind) * ind[2]), 2)) if
populacao.index(ind) != 0 else ind.append(ind[2])

```

Figura 18. Algoritmo Genético - seleção do melhor indivíduo

## 5.4. Algoritmo colônia de formiga

### 5.4.1. Descrição do código

#### • Importação

O código utiliza as seguintes bibliotecas:

**re:** Para manipulação de expressões regulares, usada na leitura de arquivos TSP.

**matplotlib.pyplot:** Para visualização gráfica.

**time:** Para medir o tempo de execução do algoritmo.

**os:** Para manipulação de arquivos e diretórios.

**random:** Para gerar números aleatórios, usados na seleção de nós durante a construção de rotas.

#### • Funções auxiliares

**listar\_arquivos\_tsp(diretorio)**

- Lista todos os arquivos TSP em um diretório e os organiza em um dicionário numerado.
- Retorna um dicionário onde as chaves são números e os valores são os nomes dos arquivos.

**selecionar\_arquivo\_tsp(diretorio)**

- Permite ao usuário selecionar um arquivo TSP da lista de arquivos disponíveis no diretório.
- Retorna o nome do arquivo selecionado.

### **ler\_coordenadas(arquivo)**

- Lê as coordenadas dos nós de um arquivo TSP no formato **EUC\_2D** (coordenadas euclidianas).
- Retorna um dicionário onde as chaves são os identificadores dos nós e os valores são tuplas com as coordenadas (x, y).

### **ler\_matriz\_explicita(arquivo, dimensao, formato)**

- Lê a matriz de distâncias de um arquivo TSP no formato **EXPLICIT** (matriz explícita de distâncias).
- Suporta o formato **UPPER\_ROW** (matriz triangular superior).
- Retorna a matriz de distâncias completa.

### **calcular\_distancia\_euclidiana(p1, p2)**

- Calcula a distância euclidiana entre dois pontos (x1, y1) e (x2, y2).

### **criar\_matriz\_distancias(coordenadas)**

- Cria uma matriz de distâncias a partir das coordenadas dos nós.
- Retorna a matriz de distâncias e a lista de nós ordenados.

### **inicializar\_feromonios(matriz, tau0)**

- Inicializa a matriz de feromônios com um valor inicial tau0.

### **calcular\_probabilidades(matriz\_dist, matriz\_fero, alfa, beta)**

- Calcula as probabilidades de transição entre nós com base nos feromônios e nas distâncias.
- Usa os parâmetros alfa (importância dos feromônios) e beta (importância da heurística).

### **selecionar\_proximo\_no(atual, probs, visitados)**

- Seleciona o próximo nó a ser visitado usando o método da roleta (probabilístico).
- Evita nós já visitados.

### **calcular\_custo\_rota(rota, matriz\_dist)**

- Calcula o custo total de uma rota com base na matriz de distâncias.

### **atualizar\_feromonios(matriz\_fero, rotas, custos, rho, Q)**

- Atualiza a matriz de feromônios usando o processo de evaporação e depósito.
- rho é a taxa de evaporação, e Q é uma constante de atualização.

## **• Algoritmo de Colônia de Formigas**

### **colonia\_de\_formigas(arquivo\_tsp, num\_formigas, max\_iter, alfa, beta, rho, Q)**

- Implementa o algoritmo de colônia de formigas para resolver o TSP.
- Lê o arquivo TSP e determina o tipo de dados (coordenadas ou matriz explícita).

- Inicializa a matriz de distâncias e a matriz de feromônios.
- Executa o algoritmo por um número máximo de iterações (`max_iter`), com um número específico de formigas (`num_formigas`).
- Em cada iteração, as formigas constroem rotas probabilísticas, e os feromônios são atualizados com base na qualidade das rotas.
- Retorna a melhor rota encontrada, seu custo e a lista de nós.

### • Função Principal – `main()`

Gerencia a execução do programa:

- Lista os arquivos TSP no diretório `tsp`.
- Permite ao usuário selecionar um arquivo.
- Configura os hiperparâmetros do algoritmo (número de formigas, iterações,  $\alpha$ ,  $\beta$ ,  $\rho$ ,  $Q$ ).
- Executa o algoritmo e exibe a melhor rota encontrada, seu custo e o tempo de execução.

## 6. Resultados

Os experimentos realizados neste projeto utilizaram arquivos de teste próprios, desenvolvidos especificamente para validar a implementação do algoritmo. Esses arquivos, juntamente com os códigos dos algoritmos, estão disponibilizados para consulta e análise em um repositório público no GitHub [8].

### 6.1. Algoritmo de Força Bruta

A tabela a seguir apresenta os resultados obtidos para cada arquivo de teste, incluindo a quantidade de pontos e o tempo necessário, em segundos, para o cálculo do melhor caminho. Além disso, é exibido o número de possibilidades  $n!$  que o algoritmo de força bruta precisa verificar até encontrar a solução ótima. Observa-se que, à medida que o número de pontos aumenta, o número de possibilidades cresce exponencialmente. Isso ocorre porque o problema do caixeiro viajante é de complexidade  $O(n!)$ .

Logo após, estão os números de possibilidades para um PCV simétrico, apenas para fins de demonstração. E por fim, as soluções ótimas encontradas para cada caso. Note que para o caso 5, com apenas 12 pontos distintos, não foi possível obter uma solução devido a sua complexidade e baixos recursos de hardware.

Arquivos	Quantidade de pontos	Tempo	Possibilidades $n!$	Possibilidades $(n-1)!/2$	Solução ótima
Caso 1.txt	4	0,13 mseg	24	3	14
Caso 2.txt	6	3,89 mseg	720	60	26
Caso 3.txt	8	0,15 seg	40.320	2.520	38
Caso 4.txt	10	19,48 s	3.628.800	181.440	54
Caso 5.txt	12	-	479.001.600	19.958.400	

**Tabela 1. Resultados para o algoritmo de Força Bruta.**

## 6.2. Algoritmo Genético

Os experimentos realizados com o Algoritmo Genético foram conduzidos com base em uma seleção criteriosa de parâmetros, definida após a aplicação de uma abordagem inicial de busca cega. A busca cega, também conhecida como busca exaustiva, consiste em um método que avalia sistematicamente diversas combinações de parâmetros sem qualquer heurística prévia, permitindo identificar padrões iniciais de desempenho e restringir o espaço de busca para ajustes mais refinados. A partir dessa análise preliminar, foram escolhidas cinco configurações distintas, combinando taxas de reprodução (60, 70, 80, 90 e 100) com taxas de mutação correspondentes (0,01; 0,05; 0,1; 0,15 e 0,2), sendo todas testadas sob um critério de parada fixo em 900 gerações. Essas configurações foram aplicadas às instâncias berlin52, ST70, brazil58, PR107 e kroa100, da TSPLIB, permitindo uma avaliação comparativa do desempenho do algoritmo em diferentes cenários.

Na instância berlin52, a configuração com taxa de reprodução de 70 e taxa de mutação de 0,05 atingiu a menor distância total, evidenciando uma convergência eficiente. A combinação 80/0,1 apresentou desempenho ligeiramente inferior, enquanto ajustes extremos – como 60/0,01 e 90/0,15 – resultaram em trajetos menos otimizados. Esses resultados indicam que parâmetros moderados favorecem um equilíbrio entre exploração e intensificação da busca, evitando tanto a estagnação em mínimos locais quanto variações excessivas.

Na instância ST70, que possui um número reduzido de nós, a convergência ocorreu de forma mais rápida, mas a tendência observada em berlin52 se manteve: a configuração 70/0,05 apresentou a melhor solução, seguida de perto por 80/0,1. O uso de parâmetros extremos geraram maior variabilidade nos resultados, reforçando que mesmo em problemas de menor escala, ajustes equilibrados são fundamentais para evitar trajetos mais longos. Um padrão semelhante foi identificado na instância brazil58, onde a dispersão dos pontos tornou a exploração do espaço de soluções um fator crítico. Aqui, novamente, taxas moderadas garantiram soluções mais eficazes, enquanto mutações muito baixas limitaram a diversidade da população e taxas elevadas comprometeram a estabilidade do algoritmo.

Na instância PR107, que apresenta maior complexidade devido ao aumento do número de nós, a parametrização 70/0,05 não apenas gerou a menor distância total, como também garantiu maior consistência ao longo das gerações. Em espaços de busca amplos, taxas elevadas de mutação dispersaram os resultados, enquanto taxas reduzidas prejudicaram a exploração de novas soluções. Esse comportamento foi ainda mais evidente na instância kroa100, a mais desafiadora do conjunto, onde a configuração moderada manteve-se como a melhor opção, garantindo trajetos mais curtos sem comprometer significativamente o tempo de processamento.

Essas análises ressaltam a importância de uma parametrização equilibrada no Algoritmo Genético para problemas combinatórios complexos como o TSP (problema do caixeiro-viajante). Independentemente da instância analisada, a configuração com taxa de reprodução de 70 e taxa de mutação de 0,05 demonstrou-se a mais eficaz, proporcionando soluções de melhor qualidade e otimizando a busca pelo ótimo global.

Parâmetros (P)	População	Taxa de Reprodução (%)	Taxa de Mutação	Critério de parada
P1	100	60	0,01	900
P2	100	70	0,05	900
P3	100	80	0,1	900
P4	100	90	0,15	900
P5	100	100	0,2	900

**Tabela 2.**Parâmetros utilizados no algoritmo genético.

### **6.3. Algoritmo Colônia de Formiga**

O algoritmo de colônia de formigas foi avaliado em cinco instâncias do TSPLIB: berlin52, brasil58, st70, kroA100 e pr107. Para cada uma dessas instâncias, foram utilizados cinco conjuntos distintos de parâmetros, representados pela letra "P" seguida de um número de 1 a 5. A configuração de cada conjunto de parâmetros é apresentada na seguinte tabela:

Parâmetros (P)	Taxa de influência do feromônio ( $\alpha$ )	Taxa de influência da distância ( $\beta$ )	Taxa de evaporação ( $\rho$ )	Constante de atualização (Q)
P1	1.0	1.0	0.05	1.0
P2	2.0	2.0	0.1	10.0
P3	3.0	3.0	0.2	20.0
P4	5.0	5.0	0.3	50.0
P5	10.0	10.0	0.4	100.0

**Tabela 3. Parâmetros utilizados no algoritmo colônia de formiga.**

Em todas as simulações, o número de formigas foi mantido constante em 100, e o algoritmo foi executado por até 100 iterações para alcançar um resultado satisfatório. Cada conjunto de parâmetros foi testado 100 vezes em cada instância, garantindo robustez na análise dos resultados.

Na instância berlin52, o melhor resultado foi obtido com os parâmetros P3, alcançando uma distância de 7458,99 unidades. No entanto, a análise dos gráficos de caixa revela que, apesar de alcançar a menor distância, os resultados associados a P3 apresentaram uma grande amplitude, indicando uma elevada variação entre as distâncias obtidas, tanto superiores quanto inferiores à mediana. Além disso, a menor distância registrada encontra-se fora da região esperada de valores, sugerindo que esse conjunto de parâmetros pode gerar soluções inconsistentes.

Os parâmetros P4 e P5 demonstraram o pior desempenho, atingindo as maiores distâncias médias, com P5 apresentando os piores resultados. Além disso, ambos exibiram uma grande amplitude, reforçando a instabilidade das soluções geradas. Por outro lado, os parâmetros P1 e P2 alcançaram o mesmo resultado ótimo, com uma distância de 7544 unidades. Entretanto, a menor variação observada nos resultados obtidos com P1 indica que

este foi o melhor conjunto de parâmetros para essa instância, pois garantiu um desempenho mais estável e consistente na maioria das execuções.

Na instância brasil58, o melhor resultado foi obtido com os parâmetros P2, alcançando uma distância de 25.874 unidades. No entanto, assim como observado anteriormente, os resultados apresentaram uma grande amplitude, indicando uma alta variação entre as soluções encontradas e, consequentemente, uma falta de consistência. Os parâmetros P1, P3 e P4 seguiram um padrão semelhante, produzindo boas soluções, mas com grande dispersão nos

resultados, o que compromete a confiabilidade das respostas geradas.

Por outro lado, os parâmetros P5 apresentaram a pior solução, com uma distância de 26.257 unidades. No entanto, essa configuração demonstrou a menor variação nos resultados, tornando-se um conjunto de parâmetros mais consistente para essa instância. Embora tenham sido observados alguns resultados discrepantes, estes foram insignificantes em quantidade e não comprometeram o desempenho geral desse conjunto de parâmetros.

A instância st70 apresentou um comportamento peculiar. Todos os conjuntos de parâmetros produziram resultados bastante próximos em relação à menor distância encontrada: P1: 706, P2: 704, P3: 712,5, P4: 706,2 e P5: 716,8. No entanto, todos esses valores ficaram fora do intervalo esperado, considerando a mediana das distâncias obtidas para cada conjunto. Apenas os conjuntos P4 e P5 demonstraram uma grande variação nos resultados, o que compromete sua consistência. Por outro lado, os conjuntos P1, P2 e P3 apresentaram um bom desempenho para essa instância, com P2 se destacando como o mais estável, pois obteve a menor variação nos resultados e uma mediana mais próxima da menor distância encontrada.

Na instância kroA100, o melhor resultado foi obtido com o conjunto P3, registrando 22.170,61 unidades de distância. Além de alcançar a menor distância, P3 demonstrou um bom desempenho, com uma pequena variação nos resultados, garantindo maior consistência. Os conjuntos P1 e P4 também apresentaram bons resultados, com variações relativamente pequenas em relação à mediana. Por outro lado, P2, apesar de alcançar o mesmo resultado ótimo que P1, mostrou uma grande variação, indicando menor estabilidade. O conjunto P5 obteve o pior desempenho, tanto em termos de distância quanto de variação, sugerindo que seus parâmetros não foram eficazes para essa instância.

Na instância pr107, o pior resultado foi obtido com o conjunto P1, enquanto o melhor desempenho foi registrado com P2, alcançando 46.185 unidades de distância. Esse resultado foi bastante próximo dos encontrados por P3, P4 e P5, indicando que esses conjuntos também foram eficazes para essa instância. Entretanto, ao analisar a variação dos resultados, P2, P3 e P5 apresentaram uma grande dispersão em relação à mediana, o que compromete a consistência das soluções encontradas. P4, por outro lado, demonstrou a menor variação entre os resultados, tornando-se o conjunto de parâmetros mais estável e confiável para essa instância.

## **6.4. Algoritmo do Vizinho Mais Próximo**

Os experimentos com o Algoritmo do Vizinho Mais Próximo foram conduzidos em cinco instâncias do TSPLIB: berlin52, ST70, brazil58, PR107 e kroa100, com o objetivo de

avaliar a eficiência computacional e a qualidade das soluções obtidas. Ao utilizar uma abordagem gulosa, o método constrói a rota de forma determinística, iniciando a partir de um ponto inicial e, a cada etapa, selecionando o nó não visitado mais próximo. Para mitigar a influência da escolha inicial, o algoritmo foi executado múltiplas vezes, variando o ponto de partida, o que permitiu uma análise comparativa robusta dos resultados.

Na instância berlin52, as execuções demonstraram uma variação considerável nas distâncias totais, com valores oscilando aproximadamente entre 15.000 e 20.000 unidades, e um tempo médio de processamento em torno de 1,7 milissegundos por execução. Essa amplitude nos resultados evidencia a sensibilidade do método à escolha do nó inicial, característica inerente à estratégia gulosa, que pode conduzir a soluções significativamente diferentes mesmo sob condições de execução semelhantes.

Nas instâncias ST70 e brazil58, a menor quantidade de nós contribuiu para tempos de execução ainda mais reduzidos, mantendo-se na ordem dos milissegundos. Contudo, a qualidade das soluções mostrou-se limitada: embora o algoritmo construa rapidamente a rota, os trajetos encontrados apresentaram distâncias totais superiores, indicando a dificuldade do método em escapar de mínimos locais.

Na instância PR107, com sua complexidade aumentada, a variação entre as soluções tornou-se mais evidente, reforçando a dependência da heurística na escolha do ponto inicial. Já na instância kroa100, a mais desafiadora do conjunto, os tempos de processamento continuaram competitivos, mas as distâncias dos percursos foram significativamente maiores, ressaltando as limitações da abordagem para problemas de alta escala.

Esses resultados destacam que, embora o Algoritmo do Vizinho Mais Próximo seja extremamente rápido e de fácil implementação, sua capacidade de gerar trajetos otimizados é comprometida pela dependência da escolha inicial e pela tendência a convergir para mínimos locais. Assim, enquanto essa abordagem pode ser vantajosa em aplicações que priorizam a rapidez na execução, sua utilização pode ser inadequada quando a minimização da distância percorrida é um critério essencial para a eficiência do trajeto.



## 7. Conclusão

Os experimentos realizados permitiram avaliar a eficácia de diferentes abordagens heurísticas na resolução do Problema do Caixeiro Viajante (PCV), destacando suas particularidades em termos de complexidade, tempo de execução e qualidade das soluções. O algoritmo de Força Bruta, embora garanta a solução ótima, se torna impraticável para instâncias com mais de 10 nós devido ao crescimento exponencial de seu custo computacional  $O(n!)$ . Essa limitação reforça a classificação do PCV como um problema NP-difícil, para o qual métodos exatos são inviáveis em grande escala.

Por outro lado, as heurísticas que analisamos mostraram que é possível equilibrar eficiência e qualidade. O algoritmo do Vizinho Mais Próximo, que tem uma complexidade de  $O(n^2)$ , se destacou por ter tempos de execução muito rápidos, na casa dos milissegundos, mesmo quando lidávamos com casos mais complexos, como o kroA100. No entanto, ele gerou soluções subótimas, evidenciando sua dependência da escolha inicial e sua tendência a ficar preso em mínimos locais. Em contrapartida, os métodos meta-heurísticos, como o Algoritmo Genético e a otimização por Colônia de Formigas, se destacaram pela capacidade de explorar o espaço de soluções de forma robusta. O Algoritmo Genético, com uma taxa de reprodução de 70% e uma mutação de 0,05%, conseguiu resultados que se aproximaram do ótimo em casos como berlin52 e pr107. Já a otimização por Colônia de Formigas, com os ajustes certos de feromônio e exploração, se mostrou útil para evitar que o algoritmo ficasse preso em soluções ruins, especialmente em problemas de maior escala.

A comparação entre os métodos reforça que, embora heurísticas mais simples funcionem bem quando a velocidade é prioridade, abordagens mais adaptativas são fundamentais para garantir soluções de alta qualidade em problemas complexos. Essas estratégias não só contornam as limitações dos métodos exatos, como também mostram como a inteligência computacional pode transformar problemas difíceis em oportunidades de otimização prática. Portanto, este trabalho corrobora a relevância das heurísticas como ferramentas indispensáveis no enfrentamento de problemas NP-difíceis, onde a busca pelo equilíbrio entre precisão e viabilidade computacional define o sucesso na resolução de desafios combinatórios.

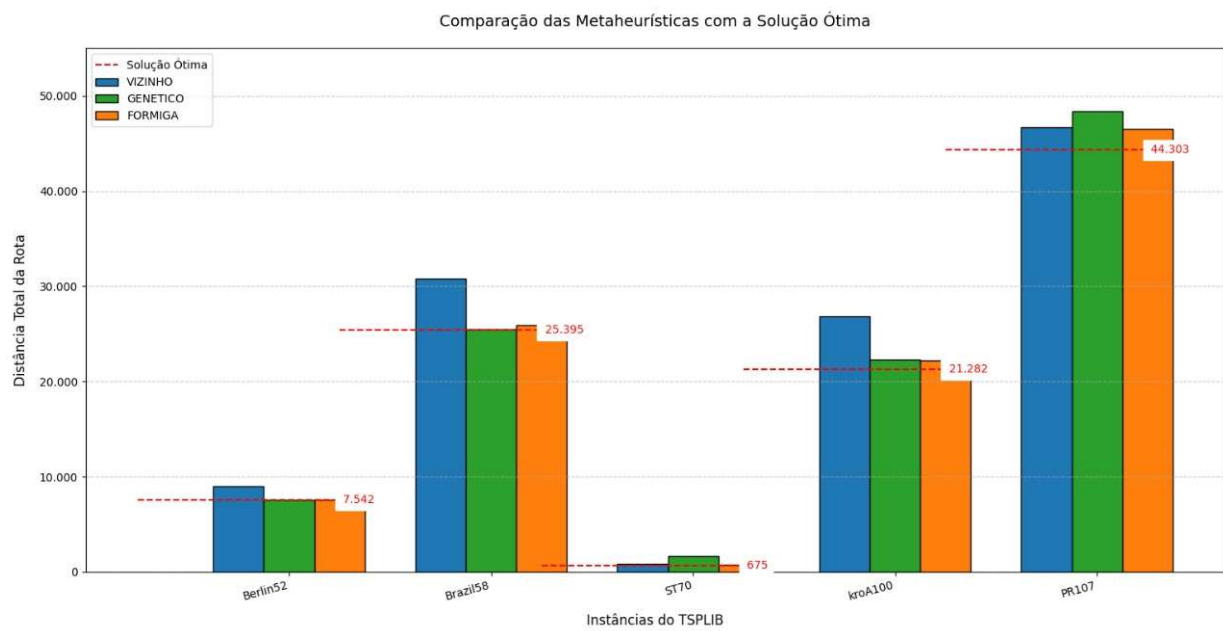


Figura 19. Gráfico comparativo das metaheurísticas com a solução ótima

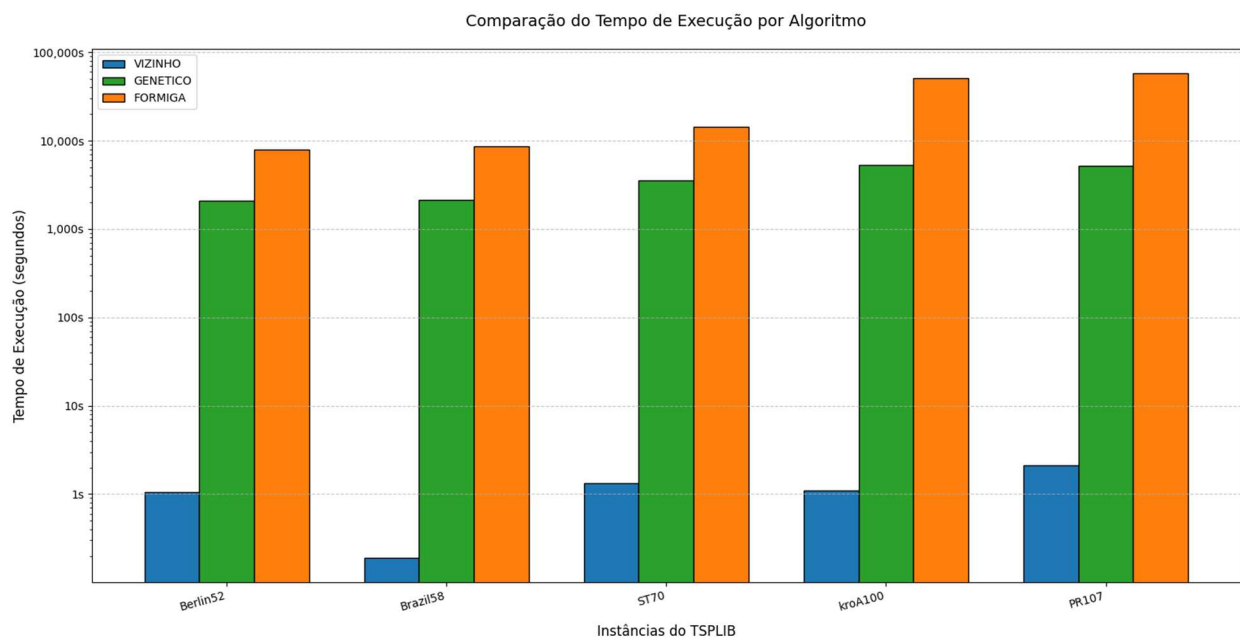


Figura 20. Gráfico comparativo do tempo de execução com a solução ótima

## 8. Referências bibliográficas

1. LOUREIRO, Antonio Alfredo Ferreira. Teoria de Complexidade. **UFMG**, 2008. Disponível em: <http://www.decom.ufop.br/menotti/paa111/slides/aula-Complexidade-imp.pdf>  
Acesso em 01 de dez. de 2024.
2. DE ABREU, Nair Maria Maia. A Teoria da Complexidade Computacional. **R. mil. Cio e Tecno/.**, Rio de Janeiro, 4(1):90-95, jan./mar. 1987. Disponível em: [http://rmct.ime.eb.br/arquivos/RMCT\\_1\\_tri\\_1987/teoria\\_complex\\_comput.pdf](http://rmct.ime.eb.br/arquivos/RMCT_1_tri_1987/teoria_complex_comput.pdf)  
Acesso em: 01 de dez. de 2024.
3. ANÁLISE de Complexidade. **UFMG**. Disponível em: <https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/complexity.pdf>  
Acesso em 01 de dez. de 2024.
4. ALGORITMO não determinístico. **Wikipedia**, 2017. Disponível em: [https://pt.wikipedia.org/wiki/Algoritmo\\_n%C3%A3o\\_determin%C3%ADstico](https://pt.wikipedia.org/wiki/Algoritmo_n%C3%A3o_determin%C3%ADstico)  
Acesso em 01 de dez. de 2024.
5. ALGORITMO determinístico. **Wikipedia**, 2017. Disponível em: [https://pt.wikipedia.org/wiki/Algoritmo\\_determin%C3%ADstico](https://pt.wikipedia.org/wiki/Algoritmo_determin%C3%ADstico)  
Acesso em 01 de dez. de 2024.
6. OLIVEIRA, André Filipe Maurício de Araújo. Extensões do Problema do Caixeiro Viajante. **UC**, 2015. Disponível em: <https://estudogeral.sib.uc.pt/bitstream/10316/31684/1/>  
Acesso em 03 de dez. de 2024.
7. ROBERTSON, J J O'Connor. Karl Menger. **MacTutor**, 2014. Disponível em: <https://mathshistory.st-andrews.ac.uk/Biographies/Menger/>  
Acesso em 03 de dez. de 2024.
8. BARBOSA, A. et al., Flyfood1VA, (2024), repositório GitHub, <https://github.com/JPlimajots/Flyfood1VA>
9. Rosen, K. H. Matemática discreta e suas aplicações. 6ª Ed. New York: McGraw-Hill, 2009.
10. TSPLIB95 - A Library of Sample Instances for the TSP (and Related Problems) from Various Sources and of Various Types. 1995. Disponível em: <http://comopt.ifl.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/>. Acesso em: 03 fevereiro de 2025.
11. Dorigo, M. Optimization, Learning and Natural Algorithms. Tese de Doutorado – Politecnico di Milano, Milan, Italy, 1992.
12. RODRIGUES, M. O.; *Problema do Caixeiro Viajante aplicado em rota de entrega aérea e solucionado por meio da Programação Linear*. Trabalho de Conclusão de Curso (Graduação em Matemática) – Universidade Estadual de Goiás, Goiás, 2021.
13. SILVA, G. A. N. et al. Algoritmos heurísticos construtivos aplicados ao problema do caixeiro viajante para a definição de rotas otimizadas. *Colloquium Exactarum*. 2013.
14. CALADO, M. F.; LADEIRA, P. A. Problema do caixeiro viajante: Um estudo comparativo de técnicas de inteligência artificial. E-xacta, 2011.
15. SILVA, Abdiel Jônatas Alves da. Algoritmo de colônia de formigas aplicado ao problema do caixeiro viajante: estudo de caso. 2022. 13 f. TCC (Graduação em Ciência e Tecnologia) - Universidade Federal Rural do Semi-Árido, Mossoró, 2022.

