
**Universidade Federal Rural de Pernambuco**

**Departamento de Estatística e Informática**

**Bacharelado em Sistemas de Informação**

# PROJETO FLYFLOOD

Antony Albuquerque Silva

**Recife**

Fevereiro de 2023

Dedido esse trabalho a meus queridos pais, que sempre acreditaram em mim e me deram o apoio incondicional em todas as minhas jornadas. Aos meus amigos, que foram uma fonte constante de motivação e apoio em todos os momentos. Obrigado por tudo.

## RESUMO

O crescimento desordenado das cidades prejudicou a mobilidade da população, resultando em trânsito ruim e impactando tanto as pessoas quanto as empresas que dependem de locomoção, incluindo serviços de entrega que sofrem com eventuais atrasos. Pensando nisso a empresa FlyFlood tenta solucionar essa situação utilizando drones para fazer entregas, no entanto existe uma limitação nas baterias desses drones. Para abordar este problema, o objetivo deste projeto é contornar a limitação das baterias dos drones de entrega de alimentos, buscando a melhor rota para o drone realizar as entregas em seu ciclo de vida da bateria. Para alcançar isso, foi elaborado um algoritmo de força bruta, utilizando o python 3, com intuito de encontrar a rota mais curta para o drone percorrer e contornar o problema da bateria. Os resultados obtidos a partir dos experimentos realizados apontam que a solução por força bruta é viável para entradas com até oito pontos. É importante destacar que a solução desenvolvida neste projeto alcançou os objetivos estabelecidos e pode ser aplicada com sucesso em outros sistemas de entrega de comida, bem como em qualquer outro sistema de roteamento.



**Palavras-chave:** Problema, Mobilidade, Sistemas de roteamento, Delivery de comida, Solução computacional, Drones, Força Bruta, Menor caminho.

# 1. INTRODUÇÃO

## 1.1 Apresentação e Motivação

Frequentemente, encontramos situações que requerem a minimização de custos e rotas na vida real. O crescimento na utilização no serviço de delivery tornou ainda mais importante encontrar as melhores rotas e menores custos. Como resultado, há a necessidade de encontrar soluções viáveis para lidar com esses desafios de deslocamento que ocasionam atraso nas entregas .

A empresa FlyFood busca oferecer uma solução para contornar o desafio de deslocamento de entregas, com a utilização de drones para fazer entregas, capaz de transportar vários pedidos em seu compartimento, otimizando o tempo de entrega. No entanto, a bateria dos drones é o maior obstáculo na implementação desta solução. É necessário otimizar o trajeto do drone para que possa realizar todas as entregas dentro do ciclo de bateria. Por esse motivo, há a necessidade de desenvolver um algoritmo que determine a rota mais curta para realizar as entregas com o drone, considerando o ponto de origem, pontos de entrega e ponto de retorno do drone.

## 1.2 Formulação do problema

Formulação matemática do problema. Definir os conceitos (matriz de entrada, coordenadas, pontos de entrega, ponto de partida) e equações (distância, custo de uma solução, equação do problema) que devem ser resolvidas para a construção de um software prático, real.

- Matriz de Entrada: Esta é a matriz que será recebida pelo programa e que conterá o ponto de origem (denominado como Ponto R) e os pontos de entrega.

- Ponto de Partida e Pontos de Entrega: É o local de onde o drone deverá partir e retornar (Ponto de Partida ou Ponto R) e os pontos que deverão ser visitados pelo drone para conclusão das entregas (Pontos de Entrega).

- Coordenadas: As coordenadas de uma matriz são as posições específicas de cada elemento na matriz. Em uma matriz bidimensional, as coordenadas são representadas por duas dimensões, linha e coluna. Por exemplo, se um elemento estiver na linha 2 e coluna 3, suas coordenadas seriam (2, 3).

| | COLUNA 1 | COLUNA 2 | COLUNA 3 | COLUNA 4 |
|---|---|---|---|---|
| LINHA 1 | 1,1 | 1,2 | 1,3 | 1,4 |
| LINHA 2 | 2,1 | 2,2 | 2,3 | 2,4 |
| LINHA 3 | 3,1 | 3,2 | 3,3 | 3,4 |

**Figura 1 . Coordenadas de uma matriz bidimensional.**

**Fonte: De autoria própria.**

- Danômetro: O termo "Danômetro" será utilizado como uma unidade de medida de distância para avaliar o custo do percurso.

- Distância de Manhattan: A Distância de Manhattan é uma medida de distância comumente utilizada em planos bidimensionais. É conhecida também como Distância de Táxi ou Geometria Pombalina, e é calculada como a soma das diferenças absolutas dos componentes das coordenadas dos pontos. Esta medida é amplamente utilizada em diversas áreas, como inteligência artificial, teoria da informação e outras, por sua simplicidade e precisão na representação de distâncias em planos 2D.

Agora que entendemos alguns conceitos fundamentais, pode-se pensar na formulação do problema matematicamente, Inicialmente, para calcularmos a distância entre dois pontos utilizando a fórmula: d = |x1 - x2| + |y1 - y2|.

A fórmula: “d = |x1 - x2| + |y1 - y2|”, representa a distância Manhattan entre dois pontos em um plano 2-dimensional com coordenadas (x1, y1) e (x2, y2). As barras de valor absoluto garantem que a diferença entre os dois pontos sempre seja positiva, e as duas diferenças são então somadas para encontrar a distância total entre os pontos. Por exemplo, digamos que temos os pontos A (2,3) e B (4,5), para calcular a distância Manhattan entre esses pontos, basta substituir as coordenadas na fórmula:

$$d = |x1 - x2| + |y1 - y2|$$

$$d = |2 - 4| + |3 - 5|$$

$$d = |-2| + |-2|$$

$$d = 2 + 2 = 4$$

Assim, a distância entre os pontos A e B é 4 “danômetros”. Uma vez que a solução seja escalável, é necessário calcular a distância diversas vezes. Diante disso,o somatório da distância de Manhattan (também conhecido como distância L1):

$$f(d) = \sum_i d(n_i , n_{i+1})$$

O somatório pode ser referido como ***f(d)*** e, com o objetivo de encontrar o menor percurso para realizar as entregas, precisamos calcular o argumento mínimo da função ***f(d)***, ou seja, ***argmin(f(d))***. Esse argumento representa o caminho que minimiza ***f(d)***, resultando no menor percurso possível.

## 1.3 Objetivos

Este relatório tem como objetivo elaborar um algoritmo que vai ler uma matriz, a partir de um arquivo, com os pontos de entrega e o ponto de origem e retorno. Ele deverá retornar a ordem em que o drone deve percorrer os pontos de entrega. Essa ordem deve ser a de menor custo, ou seja, a que o drone vá percorrer a menor distância em ‘’dronômetros’’.

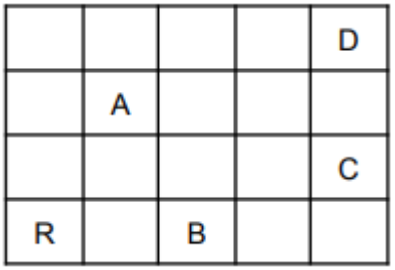


**Figura 2. Representação de uma matriz dos pontos.**
**Fonte: PISI2 - Descrição do projeto - Flyfood.**

Utilizando o python 3, a saída deverá ser a sequência de pontos (em forma de *string*) que produz o menor circuito possível a ser percorrido pelo drone entre os pontos de entrega, partindo e retornando ao ponto.

## 1.4 Organização do trabalho

Este trabalho está organizado em: o capítulo 2 é apresentado o referencial teórico, abordando conceito do caixeiro viajante, NP-Completos, classes de problemas. No capítulo 3 fornece detalhes sobre trabalhos correlatos onde é abordado três trabalhos da literatura: Kuroswiski, André Rossi et al. (2021) apresenta abordagens Exata e Heurística na Otimização do Problema do Caixeiro Viajante com Drone, é uma abordagem para otimizar o uso de caminhões e drones na logística de entregas, CB da Cunha, U de Oliveira Bonasser (2002) apresentam, experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante e DA SILVA, Gabriel Altafini Neves(2013) apresenta Algoritmos heurísticos construtivos aplicados ao problema do caixeiro viajante para a definição

de rotas otimizadas. Já o capítulo 4 apresenta a metodologia de descrição do passo a passo realizado para criação dos códigos, onde apresenta conceitos de estruturas de controle (condicionais e repetições), matrizes, dicionários e funções. Capítulo 5 descreve os experimentos com base em arquivos de teste específicos que apresentavam matrizes variando de 4 a 10 pontos. O capítulo 6 apresenta os resultados obtidos a partir dos experimentos conduzidos neste projeto. Capítulo 7 apresenta as conclusões.

# 2. REFERENCIAL TEÓRICO

## 2.1 Problema do caixeiro viajante

O problema do caixeiro viajante (PVC) pode se referir a um caixeiro-viajante que pretende viajar por um conjunto de cidades por vez. Apenas uma vez e, finalmente, retorne à cidade inicial, percorrendo a distância mais curta. Este é um dos problemas de otimização combinatória mais estudados e possui uma ampla gama de aplicações no mundo real. Foi expressa pela primeira vez matematicamente por Carl Menger entre 1931 a 1923.

Para percorrer uma determinada sequência de cidades em uma passagem e retornar à cidade inicial, todas as combinações possíveis de caminhos são dadas e podem ser calculadas usando (N-1)!/2, onde N é o número de cidades. Algoritmos que procuram percorrer todos os caminhos possíveis e fornecer uma solução ótima para o problema têm um fator de tempo de computação de O(n!),[1] e são considerados computacionalmente intratáveis.

O problema do caixeiro viajante é considerado NP-difícil, indicando que não há uma solução ótima conhecida para problemas com um grande número de cidades e o tempo de computação aumenta exponencialmente com o tamanho do problema. Isso se deve ao grande número de possíveis combinações de rotas que

[1] A notação O(f(n)) é uma forma de representar a complexidade temporal de um algoritmo, que indica o crescimento máximo do tempo de computação em relação ao tamanho da entrada n. É uma forma de descrever a escala de tempo de um algoritmo, permitindo a comparação entre diferentes algoritmos. Quanto menor a notação, mais eficiente é o algoritmo.

precisam ser avaliadas para encontrar a melhor solução. Algoritmos heurísticos, como algoritmos genéticos e de busca tabu, são utilizados para encontrar soluções aproximadas.

## 2.2 Problemas NP Completo

Problemas NP completos são considerados difíceis de resolver. Não tendo algoritmo determinístico que possa solucioná-los no tempo polinomial, esses problemas são caracterizados por terem soluções verificáveis, mas onde o tempo de computação aumenta exponencialmente com o tamanho do problema, mesmo que você tenha uma máquina super poderosa, ainda pode levar muito tempo para encontrar a resposta certa. NP completo está contido no subclasse de problemas NP,  e são considerados os mais difíceis no conjunto de NP. Veja alguns conceitos básicos para o melhor entendimento dos problemas NP completos:

### 2.2.1 Classes de complexidade

São categorias que ajudam a descrever quanto tempo e quantos recursos computacionais são necessários para resolver um determinado problema.

**Definição 1.1 Classe P:** é composta por problemas que podem ser resolvidos em tempo polinomial, ou seja, em um tempo computacional razoável, ou seja, o tempo de processamento cresce proporcionalmente ao tamanho dos dados.

**Definição 1.2 Classe NP:** E a classe de problemas para os quais a verificação de uma solução pode ser feita em um tempo polinomial. Isso significa que, embora encontrar a solução possa ser difícil ou impossível, é possível verificar se uma solução dada é correta em um tempo razoável.

**Definição 1.3 Classe NP Completo:** é uma subclasse de NP, incluindo os problemas mais difíceis dentro desse conjunto. Eles são chamados de NP-completos porque, se qualquer um deles pudesse ser resolvido em tempo

polinomial, então todos os outros problemas NP também poderiam ser resolvidos em tempo polinomial.

**Definição 1.4 Classe NP-Difícil:** É a classe de problemas que são tão difíceis quanto o mais difícil dos problemas em NP, e provavelmente exigem tempo exponencial para serem resolvidos. Exemplos de problemas NP-difíceis incluem o problema do caixeiro-viajante e o problema da mochila, que são problemas clássicos de otimização combinatória.

## 3. TRABALHOS RELACIONADOS

Kuroswiski, André Rossi et al. (2021) apresenta abordagens Exata e Heurística na Otimização do Problema do Caixeiro Viajante com Drone. É uma abordagem para otimizar o uso de caminhões e drones na logística de entregas, este trabalho científico propõe uma abordagem para resolver a variação do Problema do Caixeiro Viajante (TSP) chamada Flying Sidekick TSP (FSTSP), que inclui a logística de entregas utilizando caminhões e drones. A solução proposta inclui uma formulação matemática utilizando Programação Linear Inteira Mista e um Algoritmo Genético Híbrido (HGenFS). A vantagem do HGenFS é que ele é capaz de encontrar soluções ótimas para o FSTSP em poucos segundos. Ele também incorpora heurísticas específicas e uma fase de busca local, o que aumenta a eficiência na solução do problema. Além disso, é adequado para solucionar problemas de até dez clientes, o que é uma boa escala para aplicações em logística de entregas. No entanto, há também desvantagens no HGenFS. Ele pode não ser a solução mais eficiente para problemas de grande escala. Além disso, ele pode ser afetado por limitações computacionais e restrições de tempo de execução. Além disso, depende de uma boa escolha de parâmetros para garantir resultados satisfatórios, o que pode requerer muita experimentação. Em resumo, este trabalho científico propõe uma solução para o FSTSP que tem vantagens na eficiência de solução para problemas de pequena escala, mas também tem desvantagens relacionadas à eficiência em problemas de grande escala e na escolha de parâmetros.

CB da Cunha, U de Oliveira Bonasser (2002) apresenta experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. O artigo descreve as heurísticas utilizadas na solução do Problema do Caixeiro Viajante (PCV) que se baseiam em procedimentos de melhoria de soluções do tipo 2-opt e 3-opt. O método do vizinho mais próximo é frequentemente usado para obter a solução inicial, e a investigação visa verificar a influência desta solução inicial e dos procedimentos de melhorias na qualidade das soluções finais e no desempenho computacional. O artigo também examina o potencial da utilização conjunta dos procedimentos 2-opt e 3-opt, bem como o impacto do roteiro inicial na qualidade das soluções. Vantagens: O método k-opt é conhecido por ser eficiente em termos de tempo computacional, além de ser capaz de encontrar soluções de alta qualidade para o PCV.

Desvantagens: No entanto, o processo pode ser complexo de implementar e pode exigir muita memória computacional, dependendo da escala do problema a ser resolvido. Além disso, a qualidade da solução final também pode ser influenciada pela escolha da solução inicial e pelos procedimentos de melhoria.

DA SILVA, Gabriel Altafini Neves (2013) apresenta algoritmos heurísticos construtivos aplicados ao problema do caixeiro viajante para a definição de rotas otimizadas. Este artigo examina e compara diferentes técnicas de otimização de rotas de distribuição de produtos, incluindo heurísticas de construção de rotas e um algoritmo genético. O objetivo é encontrar o caminho mais eficiente e econômico para transportar produtos de diferentes locais. Os resultados foram obtidos a partir de dados reais de uma distribuidora de produtos em Curitiba, Brasil, e foram comparados com as rotas atualmente utilizadas pela empresa. O algoritmo 2-opt também foi aplicado para melhorar as rotas geradas.

Este trabalho avaliou a eficiência de diferentes heurísticas de construção e melhoria de rotas, incluindo a heurística "Inserção do Mais Distante" com melhoria 2-opt, que apresentou os melhores resultados, e o uso de Algoritmos Genéticos para o Problema do Caixeiro Viajante (PCV). Os resultados mostraram que o uso de

Algoritmos Genéticos permite encontrar soluções de boa qualidade com tempo computacional de aproximadamente 7 segundos. No entanto, há uma desvantagem de não haver garantia de que a solução encontrada é a melhor.

## 4. METODOLOGIA

Iniciaremos este projeto com a implementação do algoritmo de força bruta,[2] utilizando Python 3. Ao construir o código, aplicamos conceitos de estruturas de controle (condicionais e repetições), matrizes, dicionários e funções. Oferecemos uma solução para o problema do projeto ‘’FlyFood’’, que consiste em encontrar o caminho mínimo ***(argmin(s(p)))***. Guiaremos o processo de forma clara e precisa, explicando cada etapa detalhadamente.

[2] Algoritmo de força bruta é um método de resolução de problemas que consiste em tentar todas as combinações possíveis até encontrar a solução ótima. É um método simples e direto, mas geralmente não é o mais eficiente em termos de tempo de execução.

**4.1 Pseudocódigo da solução ótima utilizando força bruta.**

| | **Pseudocódigo do algoritmo com força bruta** |
|---|---|
| 1 | Começo da função para realizar a permutação |
| 2 | |
| 3 | var  saida : list |
| 4 | |
| 5 | função **permutacao**(l, inicio=0) |
| 6 |  se inicio é igual a comprimento de l - 1 então |
| 7 |  retornar [cópia de l] |
| 8 |  senão |
| 9 |  saída <- lista vazia |
| 10 |  para i <- inicio até comprimento de l |
| 11 |  troque l[inicio] com l[i] |
| 12 |  saída <- saída + permutacao(l, inicio + 1) |
| 13 |  troque l[inicio] com l[i] |
| 14 |  retornar saída |
| 15 | fim |
| 16 | |
| 17 | Começo da função para calcular melhor rota |
| 18 | |
| 19 | var  custo_minimo, custo_atual, x1, x2, y1, y2 : inteiro |
| 20 |  melhor_percuso : list |
| 21 | |
| 22 | algoritmo **melhor_rota**(p, c) |
| 23 |  custo_minimo <- infinito |
| 24 |  melhor_percuso <- lista vazia |
| 25 |  para rotas em p |
| 26 |  custo_atual <- 0 |
| 27 |  rotas <- ['R'] + rotas + ['R'] |
| 28 |  para indice_somas <- 0 até comprimento de rotas - 1 |
| 29 |  x1, y1 <- c[rotas[indice_somas]] |
| 30 |  x2, y2 <- c[rotas[indice_somas + 1]] |
| 31 |  custo_atual <- (custo_atual + valor absoluto de x1 - x2 + |
| 32 | valor absoluto de y1 - y2 ) |
| 33 |  se custo_atual < custo_minimo então |
| 34 |  custo_minimo <- custo_atual |
| 35 |  melhor_percuso <- rotas |
| 36 |  retornar melhor_percuso[1:-1], custo_minimo |
| 37 | fim |
| 38 | |
| 39 | |
| 40 | |
| 41 | |

```
42  algoritmo main()
43
44  var
45      arquivo, m, n, : caracter
46      coordenadas : dict
47      pontos_entrega, percurso : list
48
49    arquivo <- abrir arquivo ".txt" com modo de leitura "r"
50    m, n <- ler uma linha do arquivo e dividir por espaço
51    linhas <- ler todo o arquivo e dividir por linhas
52    coordenadas <- criar dicionário vazio
53    pontos_entrega <- criar lista vazia
54    para i <- 0 até m como inteiro
55      m <- linhas[i] dividido por espaço
56      para j em m
57        se j != '0' então
58          coordenadas[j] <- (i, posição de j em m)
59          adicionar j em pontos_entrega
60    remover 'R' de pontos_entrega
70    percurso <- chamar função melhor_rota (com entrada
71  pontos_entrega permutados e coordenadas)
72
73    imprimir " ".join(percurso[0]), percurso[1] com formatação
    fim
```

**4.2 Funcionamento da solução ótima utilizando força bruta.**

**Linha 5 a 14**: Este trecho de código é uma implementação do algoritmo de *backtracking* que gera todas as permutações possíveis de uma lista de elementos (l). A função permutacao() usa recursão para ir trocando os elementos na lista, gerando todas as possíveis permutações. A cada chamada recursiva, o índice inicial é incrementado para evitar repetição de elementos. As permutações geradas são adicionadas à lista "saida" e retornadas quando a recursão chegar ao final da lista.

A escalabilidade de uma permutação depende do tamanho do conjunto de elementos que está sendo permutado. Quanto maior o tamanho do conjunto, maior será o número de permutações possíveis.

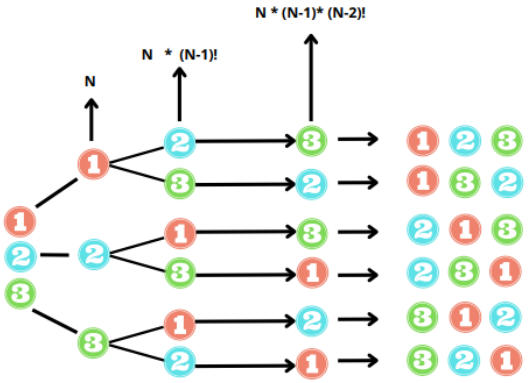


**Figura 3: Exemplo de como funciona a permutação.**

**Fonte: Elaboração própria.**

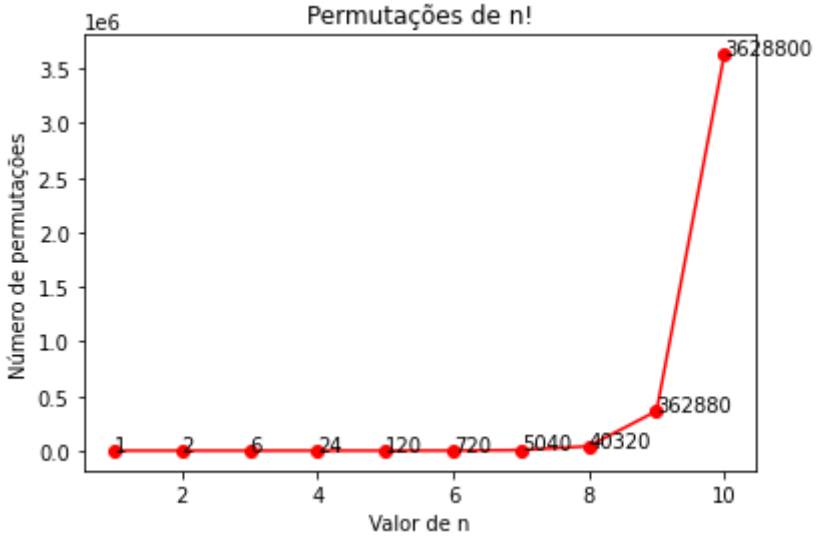


**Gráfico 1: Gráfico do número de permutações do valor n! (fatorial)**

**Fonte: Biblioteca Matplotlib do Python**

**Linha 22 a 36:** A função "melhor_rota" recebe uma lista de rotas possíveis e um dicionário de coordenadas de cidades. Ela calcula o custo mínimo de percorrer cada uma das rotas (onde o custo é a distância de Manhattan entre as cidades), e retorna a rota com o menor custo e o seu custo mínimo. A função itera sobre todas as rotas possíveis, calculando o custo de cada uma, e retorna a rota com o menor

custo. A rota "R" é adicionada no início e no final de cada rota, para que a função comece e termine em "R". A função usa a fórmula da distância de Manhattan para calcular o custo de cada rota.

**Linha 49 a 73:** Abre um arquivo '.txt' que contém os dados da matriz. A primeira linha do arquivo contém o número de linhas da matriz, e as linhas subsequentes contêm os valores da matriz.

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

**Figura 4. Arquivo .TXT do projeto Flyfood.**

**Fonte: PISI2 - Descrição do projeto Flyfood.**

O laço processa a informação da matriz, adiciona os pontos de entrega a lista "pontos_entregas", e armazena as coordenadas de cada ponto de entrega na lista "coordenadas". Em seguida, a função remove 'R' da lista "pontos_entregas" e chama a função "melhor_rota" com a lista de permutações gerada pela função "permutacao" e as coordenadas dos pontos de entrega. Finalmente, a função imprime a melhor rota e seu custo, partindo do ponto inicial ***"R"*** e retornando, melhor percurso como **"R - B - C - D - A - R"** , com 14 danômetros.

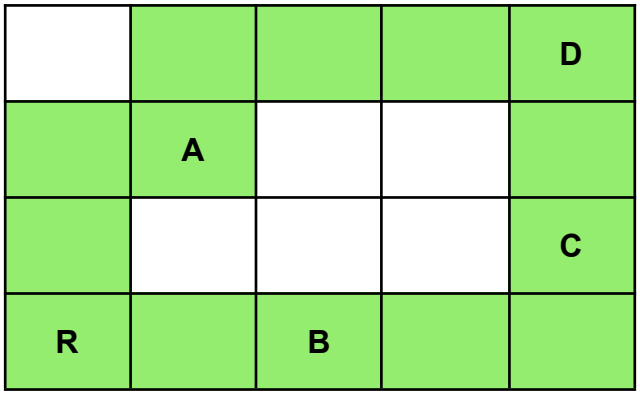


**Figura 4. Melhor percurso da matriz de entrada 4x5.**

**Fonte: De autoria própria.**

## 5. EXPERIMENTOS

No presente projeto, foram conduzidos experimentos com base em arquivos de teste específicos que apresentavam matrizes variando de 4 a 10 pontos. Para tal, utilizamos a biblioteca "time" do Python e empregamos a biblioteca "matplotlib" para a construção de gráficos. A máquina empregada para os testes foi em máquinas virtuais fornecidas pela plataforma Google Colab, de 12 GB de memória de GPU.

## 6. RESULTADOS

A seguir, apresentamos os resultados obtidos a partir dos experimentos conduzidos neste projeto. Esses resultados foram alcançados através da utilização de arquivos de teste específicos. A análise dos resultados permitiu a obtenção de insights valiosos sobre o comportamento de diferentes matrizes. Além disso, monitoramos o tempo de crescimento entre as entradas, o que é crucial para avaliar o desempenho da solução proposta e verificar se ela atende às expectativas em termos de velocidade de processamento.

A tabela apresenta os resultados dos testes dos arquivos. Ela mostra a quantidade de pontos e a solução ótima para o melhor caminho. A tabela também indica o número de possibilidades (n!) que o algoritmo de força bruta precisa verificar para encontrar a solução ideal. Observe que com o aumento do número de pontos, o número de possibilidades aumenta exponencialmente, resultando em uma complexidade de tempo O(n!)

| Arquivos usados | Quantidade de pontos | Possibilidades n! | Solução ótima | Tempo (segundos) |
|---|---|---|---|---|
| teste 1 | 4 | 24 | 14 | tempo < 1 |
| teste 2 | 6 | 720 | 22 | tempo < 1 |
| teste 3 | 8 | 40.320 | 30 | tempo < 1 |
| teste 4 | 10 | 3.628.800 | 38 | 30 |
| teste 5 | 12 | 479.001.600 | - | - |

**Tabela 1. Tabela de resultado dos testes.**

**Fonte: De autoria própria.**

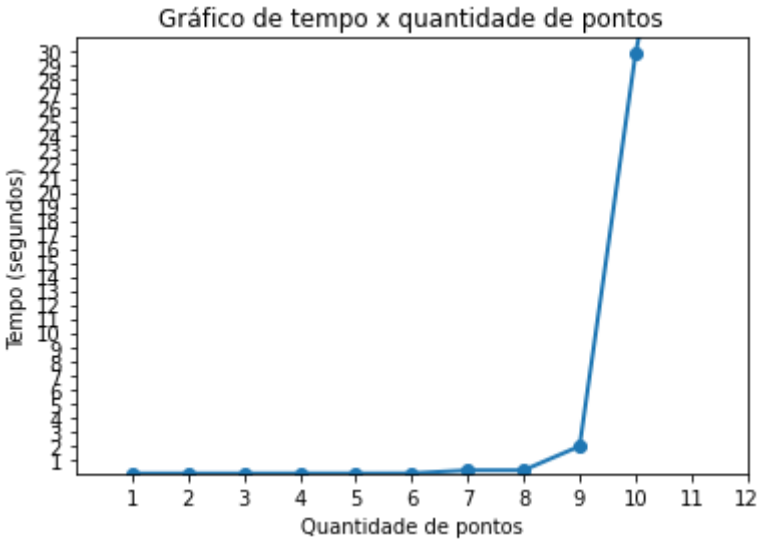


**Gráfico 2. Tempo x quantidade de pontos.**

**Fonte: Matplotlib do Python.**

Por fim, apresentamos as soluções ótimas encontradas para cada caso estudado. Cabe ressaltar que, no caso 5, mesmo com apenas 12 pontos distintos, infelizmente não foi possível alcançar uma solução devido à complexidade do problema e às restrições de recursos de hardware.

# 7. CONCLUSÃO

Em resumo, este projeto teve como objetivo desenvolver um algoritmo que encontre a rota mais eficiente para um sistema de delivery com drone. O algoritmo seria capaz de determinar a ordem de visita dos pontos de entrega, buscando minimizar a distância percorrida pelo drone. Este trabalho contribuirá para a otimização de processos de entrega, tornando-os mais eficientes e reduzindo os custos envolvidos.

Após um período dedicado às atividades, era importante avaliar os resultados alcançados. Neste momento, destacou-se o sucesso na resolução do projeto FlyFlood, que buscava encontrar o menor trajeto para as entregas do drone dentro do limite da bateria. Além disso, foram identificadas conquistas e avanços significativos, que serviram para a identificação de pontos de melhoria para futuras ações.

Adicionalmente, verificou-se o êxito do algoritmo de força bruta em entradas com tamanho inferior a 12 pontos. Contudo, tornou-se evidente que sua eficiência seria insuficiente em entradas de maior porte. Desta forma, torna-se necessária a busca por alternativas eficazes para lidar com entradas superiores às previamente estabelecidas nos casos de teste.

Em conclusão, é evidente que há amplo potencial para melhorar a solução atual, tanto no que diz respeito à sua eficiência quanto à sua facilidade de uso. O uso de heurísticas é uma abordagem promissora para solucionar problemas de otimização, como o problema do caixeiro viajante. Essas técnicas permitem encontrar soluções aproximadas de maneira rápida e eficiente, em vez de tentar encontrar a solução perfeita através de métodos exaustivos. Além disso, as heurísticas podem ser ajustadas e adaptadas para atender às necessidades específicas de cada caso, tornando-as uma ferramenta versátil e eficaz para solucionar problemas de otimização.

# REFERÊNCIAS BIBLIOGRÁFICAS

Wikipedia. Geometria do táxi. Wikipedia, 2021. Disponível em: https://pt.wikipedia.org/wiki/Geometria_do_t%C3%A1xi. Acesso em: 20 de dez. de 2022.

Wikipedia. Problema do caixeiro viajante. Wikipedia, 2023. Disponível em: https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante. Acesso em: 23 de fev. de 2023.

Ferreira, P. NP-Completo. Análise de Algoritmos, Instituto de Matemática e Estatística da Universidade de São Paulo, 2012. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html. Acesso em: 27 de dez. de 2022.

Lima, H. Diferença entre algoritmos determinísticos e não determinísticos. Acervo Lima, 2021. Disponível em: https://acervolima.com/diferenca-entre-algoritmos-deterministicos-e-nao-deterministicos/. Acesso em: 27 de dez. de 2022.

KUROSWISKI, André Rossi et al. Abordagens Exata e Heurística na Otimização do Problema do Caixeiro Viajante com Drone. Disponível em: https://www.sige.ita.br/edicoes-anteriores/2021/st/218394_1.pdf. Acesso em: 19 de jan. de 2023.

CB da Cunha, U de Oliveira Bonasser. Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. Disponível em: https://www.researchgate.net/profile/Claudio-Cunha-3/publication/228434832_Experimentos_computacionais_com_heuristicas_de_melhorias_para_o_problema_do_caixeiro_viajante/links/54803ccb0cf2ccc7f8bb2c18/Experimentos-computacionais-com-heuristicas-de-melhorias-para-o-problema-do-caixeiro-viajante.pdf. Acesso em: 25 de jan. de 2023.

DA SILVA, Gabriel Altafini Neves et al. apresenta Algoritmos heurísticos construtivos aplicados ao problema do caixeiro viajante para a definição de rotas otimizadas. Disponível em: https://journal.unoeste.br/index.php/ce/article/view/939. Acesso em: 29 de jan. de 2023.

SILVA, Antony Albuquerque. Repositório do trabalho. GitHub. Recife, PE. 2023. Disponível em: https://github.com/antonyalbuquerque/FlyFlood.git. Acesso: 15 de fev. de 2023.