Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

PROJETO FLYFOOD

Felipe Cavalcante Santos

Gustavo Costa Santos

Pedro Ailton Alves da Cunha

Recife

Outubro de 2025

Este projeto é dedicado à equipe que o idealizou e construiu. Foi a nossa união que transformou debates em planos, e linhas de código em uma solução funcional. Este trabalho é o registro fiel do nosso comprometimento e da nossa capacidade de execução como um time.

Resumo

Este trabalho apresenta o desenvolvimento de uma ferramenta em Python, no contexto da empresa Flyfood, para resolver o problema de otimização de rotas de entrega. O objetivo principal foi criar uma solução computacional capaz de determinar a sequência ótima de visitas a um conjunto de pontos, minimizando a distância total percorrida. A metodologia adotada foi a implementação de um algoritmo de força bruta que resolve uma variação do Problema do Caixeiro Viajante (PCV), explorando todas as permutações de rotas possíveis. O custo de cada rota foi calculado com base na distância de Manhattan, uma métrica apropriada para o cenário proposto. A ferramenta implementada na função otimizarRotaPlus não apenas identifica a rota de menor custo (ótima), mas também a de maior custo (pior), oferecendo uma medida clara da eficácia da otimização. Os resultados experimentais validam a corretude do algoritmo, enquanto a análise de performance confirma seu crescimento de tempo em ordem fatorial (O(n!)), limitando sua aplicação prática a cenários com um número reduzido de pontos de entrega. O software cumpre seu objetivo e serve como uma base de referência precisa (benchmark) para o futuro desenvolvimento de soluções heurísticas mais escaláveis.

Palavras-chave: Drones de Entrega. Otimização de Rotas. Algoritmo de Força Bruta.

1. Introdução

1.1 Apresentação e Motivação

O trabalho está inserido em um cenário futurista, no ano de 2030, onde o trânsito nas cidades se tornou caótico. Neste contexto, uma empresa chamada FlyFood foi criada para realizar entregas por meio de drones. Para simplificar o problema de localização e coordenadas, o ambiente da cidade é abstraído como uma matriz de pontos.

Os principais problemas encontrados nesse contexto são:

- As empresas de delivery tradicionais não conseguem mais realizar entregas em um tempo considerado aceitável;
- O custo com entregadores humanos tornou-se muito alto;
- Apesar do uso dos drones, surge uma nova limitação: a capacidade de trabalho por carga da bateria dos drones.

A motivação do trabalho é a necessidade de superar a limitação das baterias dos drones. Para que o sistema de entregas da FlyFood seja viável e eficiente, é crucial otimizar ao máximo o trajeto percorrido. Portanto, a justificativa para o desenvolvimento do algoritmo é a capacidade de definir o menor trajeto possível para a realização de todas as entregas, garantindo que a rota seja concluída dentro de um único ciclo de bateria, no mínimo. Isso otimiza o tempo de entrega dos pedidos e viabiliza a solução proposta.

1.2 Formulação do problema

O problema consiste em encontrar a rota de menor custo, dentre todas as possíveis rotas, para um drone, partindo de um ponto de origem, visitando um conjunto de pontos de entrega exatamente uma vez, e retornando ao ponto de origem.

1.2.1 Definições de Conceitos

- Matriz de Entrada: Uma matriz A de dimensões M×N que representa a área de entrega. Cada elemento Aij (com i representando a linha e j a coluna) contém um identificador. O valor "0" indica uma posição vazia, enquanto letras (como 'R', 'A', 'B', etc.) identificam pontos de interesse.
- Coordenadas: Cada ponto de interesse P é definido por uma tupla de coordenadas (i,j), onde i é o índice da linha ($0 \leq i < M$) e j é o índice da coluna ($0 \leq j < N$).
- Ponto de Partida: É um ponto único, denominado “R”, com coordenadas (iR , jR). Por convenção da proposta do trabalho, é o ponto de origem e retorno para todas as rotas.
- Pontos de Entrega: É o conjunto de todos os pontos de interesse E={e1 , e2, ..., ek} que não são o ponto de partida R. O drone deve visitar cada um desses k pontos.

1.2.2 Equações

- **Equação de Distância (Métrica de Manhattan):**

A distância entre dois pontos quaisquer, P1=(i1,j1) e P2=(i2,j2), é calculada utilizando a distância de Manhattan, uma vez que o drone não se move na diagonal, mas em linhas retas horizontais ou verticais.

A imagem a seguir representa duas possíveis métricas, a Euclidiana e a de Manhattan, deixando clara a utilidade da segunda para a aplicação:

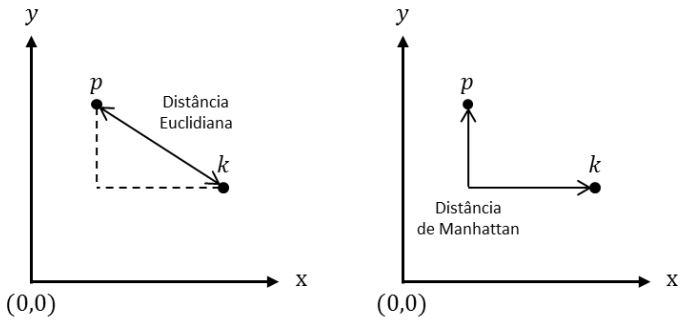

figura 1. Comparação da distância Euclidiana e de Manhattan em um plano cartesiano (VIEIRA, 2016)

A função de distância d(P1,P2) é definida como:

$$d(P1, P2) = |i2 - i1| + |j2 - j1|$$

Esta equação está implementada na função calcularDistancia(), no arquivo otimizador.py.

- **Equação do Custo de uma Solução:**

Uma solução é uma rota, que pode ser representada como uma permutação específica dos pontos de entrega.

Seja S uma rota definida por uma sequência ordenada de pontos de entrega S = (e1, e2, ..., ek). O custo total C(S) dessa rota é a soma das distâncias de todos os segmentos do percurso, começando em R e terminando em R. A equação formulada representativa é a seguinte:

$$\mathbf{C(S) = d(R, e1) + \left(\sum_{n=1}^{k-1} d(en , en + 1) \right) + d(ek, R)}$$

Esta lógica está implementada na função calcularCustoTotalDaRota() em otimizador.py.

- **Equação do Problema (Otimização):**

O problema central é encontrar a rota ótima, que minimiza a função de custo total. Seja Π(E) o conjunto de todas as permutações possíveis do conjunto de pontos de entrega E. O objetivo é encontrar a rota S que satisfaça a seguinte equação:

$$Sótima = argmin C(S) \mid S \in P(E)$$

Onde a função de custo total de rota (C(S)) apresentada no tópico anterior deve retornar o argumento mínimo (a entrada para o menor valor resultante da função, *argmin*), para toda rota (S) pertencente ao conjunto de todas as permutações dos pontos de interesse (P(E)).

Em outras palavras, o software deve encontrar a permutação de pontos de entrega (melhor_rota) que resulta no menor valor possível da função de custo total (menor_custo), conforme implementado na função otimizarRota() em otimizador.py.

1.3 Objetivos

1.3.1. Objetivo Geral

O objetivo geral deste trabalho é desenvolver um algoritmo de roteamento capaz de otimizar a rota de entrega de um drone. A partir de uma matriz que representa um mapa com múltiplos pontos de entrega, o algoritmo deve determinar o menor trajeto possível para que o drone visite todos os locais designados e retorne ao seu ponto de origem, minimizando o custo de percurso e, consequentemente, o consumo da bateria.

1.3.2. Objetivos Específicos

Funcionalidades específicas do programa:

- Analisar o arquivo de entrada: Desenvolver uma função para ler um arquivo de texto, interpretar a matriz que representa o mapa e extrair as coordenadas de todos os pontos de interesse (o ponto de origem 'R' e os pontos de entrega)33.
- Implementar o cálculo de distância: Criar uma função para calcular a distância entre dois pontos quaisquer na matriz. O cálculo deve seguir a métrica da Distância de Manhattan, considerando que o drone se move apenas na horizontal e na vertical.
- Desenvolver o algoritmo de otimização: Implementar a lógica principal que encontra a rota de menor custo. Isso envolve gerar todas as permutações possíveis de rotas entre os pontos de entrega e calcular o custo total de cada uma para identificar a ideal.
- Apresentar o resultado: Exibir a rota ótima encontrada como uma sequência de pontos formatada em uma string, indicando a ordem em que as entregas devem ser realizadas, sem considerar o ponto R.

1.4 Organização do trabalho

O presente trabalho está estruturado em sete seções principais, precedidas por um Resumo e seguidas das Referências Bibliográficas e de um Apêndice, que possui o material de suporte, scripts utilizados nos testes desenvolvidos.

- A **Seção 1**, "Introdução", apresenta o contexto do projeto, a motivação e a formulação matemática do problema , além de definir os objetivos geral e específicos a serem alcançados.
- A **Seção 2**, "Referencial Teórico", aborda os conceitos que fundamentam o trabalho , discutindo a classe de complexidade computacional do problema , a definição do Problema do Caixeiro Viajante e os métodos de solução aplicados.
- A **Seção 3**, "Trabalhos Relacionados", contextualiza o projeto em relação a outras abordagens , detalhando soluções algorítmicas concorrentes e aplicações comerciais existentes.
- A **Seção 4**, "Metodologia", descreve os aspectos práticos do desenvolvimento , incluindo as tecnologias , bibliotecas e a estrutura dos dados utilizados.
- As **Seções 5 e 6**, "Experimentos" e "Resultados", são designadas para detalhar o passo a passo dos testes realizados, dados, critérios de avaliação e os resultados obtidos.
- A **Seção 7**, "Conclusão", finaliza o documento recapitulando os objetivos e resultados, e propondo trabalhos futuros.

2. Referencial Teórico

Este trabalho se baseia em conceitos fundamentais da teoria da computação e da otimização combinatória para resolver um problema prático de roteirização.

2.1 Classes de Problemas de Computação e Complexidade

O problema central do projeto FlyFood pertence a uma classe de problemas de otimização conhecidos por sua alta complexidade computacional. Especificamente, ele é uma instância do **Problema do Caixeiro Viajante (PCV)**, que é classificado como **NP-difícil** (*Nondeterministic Polynomial time hard*) (ROSEN, 2019).

Isso implica que não se conhece nenhum algoritmo que possa encontrar a solução ótima em tempo polinomial em relação ao número de pontos de entrega. Em outras palavras, à medida que o número de pontos de entrega aumenta, o tempo

necessário para encontrar a rota perfeita cresce exponencialmente. Essa característica torna a busca por soluções eficientes um desafio significativo e justifica o estudo de diferentes abordagens algorítmicas. A solução por força bruta, adotada neste projeto, embora garanta a otimalidade, exemplifica essa complexidade, tornando-se impraticável para um número elevado de pontos.

2.2 O Problema do Caixeiro Viajante (PCV)

O Problema do Caixeiro Viajante (do inglês *Traveling Salesperson Problem* - TSP) é um dos problemas mais estudados em otimização. Sua definição clássica é: dado um conjunto de cidades e as distâncias entre cada par delas, qual é o menor caminho possível que visita cada cidade exatamente uma vez e retorna à cidade original?

No contexto do FlyFood, a analogia é direta:

- As "cidades" são os **pontos de entrega** e o ponto de partida **R**.
- A "distância" é o custo do percurso em **dronômetros**, calculado com base na movimentação horizontal e vertical na matriz.
- O objetivo é encontrar a sequência de visitas que minimiza a distância total percorrida pelo drone, partindo e retornando a **R**.

2.3 Métodos de Solução

2.3.1. Abordagem de Força Bruta (Permutação Completa)

O método implementado neste trabalho é o de **força bruta**. Ele se baseia na geração e teste de todas as rotas possíveis. Para um conjunto de *n* pontos de entrega, o número total de rotas (permutações) é *n!* (n fatorial). O algoritmo funciona da seguinte maneira:

1. Gerar todas as sequências possíveis (permutações) dos pontos de entrega.
2. Para cada permutação, calcular o custo total da rota, incluindo a ida do ponto 'R' ao primeiro ponto da sequência e a volta do último ponto para 'R'.
3. Comparar o custo de cada rota e armazenar aquela com o menor custo encontrado até o momento.

Essa técnica garante a descoberta da solução ótima, mas, devido ao crescimento fatorial, seu uso é viável apenas para um número pequeno de pontos de entrega.

Segue a tabela de valores da função fatorial, para demonstrar a inviabilidade do uso dessa técnica:

n	n! (Fatorial de n)
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5.040
8	40.320
9	362.880
10	3.628.800
11	39.916.800
12	479.001.600
13	6.227.020.800
14	87.178.291.200
15	1.307.674.368.0
16	20.922.789.888.
17	355.687.428.096
18	6.402.373.705.7
19	121.645.100.408
20	2.432.902.008.1

figura 2. Tabela de valores da função fatorial. Fonte: autor.

2.3.2. Distância de Manhattan

Como o drone está restrito a se mover apenas na horizontal ou vertical, a métrica de distância utilizada é a **Distância de Manhattan**, também conhecida como "distância do quarteirão" ou "geometria do táxi". Para dois pontos, P1=(i1,j1) e P2=(i2,j2), a distância é calculada pela soma das diferenças absolutas de suas coordenadas:

$$d(P1,P2) = |i2 - i1| + |j2 - j1|$$

Esta métrica é perfeitamente adequada para representar o custo de deslocamento em uma grade, como a matriz que representa a cidade no projeto; como apresentado na seção 1.2.2.

3.Trabalhos relacionados

O problema de otimização de rotas, central para o projeto FlyFood, é uma instância de um dos problemas mais clássicos e estudados na ciência da computação: o Problema do Caixeiro Viajante (PCV), ou Traveling Salesperson Problem (TSP). Nesta seção, contextualizamos o projeto em relação a este problema fundamental, discutimos abordagens algorítmicas alternativas e o comparamos com sistemas comerciais existentes.

3.1 O Problema do Caixeiro Viajante (PCV)

O PCV busca determinar a menor rota possível para um "caixeiro viajante" que precisa visitar um conjunto de cidades, passando por cada uma exatamente uma vez, e retornando à sua cidade de origem. A relação com o projeto FlyFood é direta:

- O drone é o "caixeiro viajante".
- Os pontos de entrega são as "cidades" a serem visitadas.
- O ponto R é a "cidade de origem".
- O custo a ser minimizado é a distância total percorrida em "dronômetros".

O PCV é classificado como um problema NP-difícil, o que significa que o tempo necessário para encontrar a solução ótima cresce exponencialmente com o número de pontos de entrega. A abordagem de força bruta, que testa todas as permutações possíveis e foi a implementada no projeto FlyFood, garante a solução ótima, mas se torna computacionalmente inviável para um número maior de pontos (geralmente acima de 10-12 pontos).

3.2 Abordagens Algorítmicas Concorrentes

Em contraste com a abordagem de força bruta, existem outros métodos para resolver o PCV que são mais escaláveis, divididos principalmente em duas categorias:

- Algoritmos heurísticos: Não garantem a solução ótima, mas buscam encontrar uma solução "muito boa" em um tempo computacionalmente viável. São os mais utilizados em sistemas reais com muitas paradas. Exemplos incluem:
    - Vizinho Mais Próximo (Nearest Neighbor): Um algoritmo simples e rápido onde, a cada passo, o drone escolheria o ponto de entrega mais próximo ainda não visitado. Geralmente não produz a melhor rota, mas serve como uma boa aproximação inicial.
    - Algoritmos Genéticos e Simulated Annealing: Técnicas mais avançadas de meta-heurística que imitam processos naturais (evolução e recozimento de metais, respectivamente) para explorar o espaço de soluções e encontrar rotas de alta qualidade, mesmo para centenas de pontos.
- Algoritmos exatos: Buscam a solução ótima de forma mais inteligente que a força bruta. Um exemplo é o Branch and Bound, que constrói a rota passo a passo e "poda" ramos da árvore de busca que não podem levar a uma solução melhor que a já encontrada, economizando processamento.

3.3 Sistemas e Aplicações Comerciais

No âmbito comercial, o problema de roteirização é resolvido por diversas empresas em larga escala:

1. Sistemas de Logística (FedEx, Amazon Logistics): Essas empresas resolvem uma versão mais complexa do PCV, conhecida como Problema de Roteamento de Veículos (VRP), que envolve múltiplos veículos, capacidades de carga, janelas de tempo para entrega e outras restrições. Elas utilizam algoritmos heurísticos sofisticados para planejar as rotas de suas frotas diariamente.
2. Aplicativos de Delivery (iFood, Rappi, Uber Eats): Estes serviços resolvem uma versão dinâmica do problema em tempo real, onde os pontos de entrega e os veículos (entregadores) mudam constantemente. Seus algoritmos precisam ser extremamente rápidos para atribuir pedidos e otimizar rotas de forma contínua.

3. Projetos de Entrega por Drones (Amazon Prime Air, Wing): São os concorrentes conceituais diretos do FlyFood. Além da otimização de rota, eles lidam com desafios adicionais como gerenciamento de tráfego aéreo, restrições de voo, condições climáticas e a própria gestão da vida útil da bateria em condições reais, um problema central que motivou o projeto FlyFood.

4. Metodologia

4.1 Tecnologias

Python 3 — Utilizado como a principal linguagem de programação para desenvolver toda a lógica do projeto. Isso inclui a leitura e interpretação do arquivo de entrada, o cálculo da distância entre os pontos, e a implementação do algoritmo de otimização de rotas por força bruta, que testa todas as permutações possíveis dos pontos de entrega.

Visual Studio Code (VSCode) — Utilizado como ambiente de desenvolvimento integrado (IDE) para escrever, editar, depurar e gerenciar os arquivos de código-fonte do projeto.

Git — Utilizado como sistema de controle de versão para registrar o histórico de alterações no código-fonte, permitindo o gerenciamento das diferentes etapas do desenvolvimento do projeto.

Github — Utilizado como plataforma para hospedar o repositório Git do projeto na nuvem. Facilita o armazenamento remoto, o controle de versão e o compartilhamento do código-fonte.

4.2 Bibliotecas

Para o desenvolvimento da solução, foram utilizadas as seguintes bibliotecas padrão do Python:

- **itertools** — Fornece a função *permutations*, usada para gerar, de forma iterável, todas as ordens possíveis dos pontos de entrega. No modo FAST, iteramos

diretamente sobre *permutations*(...) (sem materializar em lista), reduzindo o uso de memória; no modo PLUS, coletamos todas as rotas para ordená-las e exibi-las.

- **time** — Utilizada como ferramenta opcional para análise de desempenho. A função perf_counter permite cronometrar o tempo de execução do algoritmo, desde a leitura do arquivo até a apresentação do resultado final.
- **csv** — O programa utiliza esta biblioteca para ler o arquivo em formato .csv, caso o usuário opte por utilizá-lo, ao invés do arquivo em formato .txt.

4.3 Dados

O conjunto de dados para o problema é fornecido por meio de um .txt (arquivo de texto), ou .csv, que estrutura o mapa da cidade em uma matriz. O formato do arquivo é o seguinte:

- Primeira Linha: Contém dois números inteiros separados por espaço, M e N, que definem as dimensões da matriz (M = número de linhas, N = número de colunas).
- Linhas Subsequentes: As M linhas seguintes contêm os dados da matriz, onde cada caractere (N caracteres por linha) representa um ponto no mapa.
  - O caractere 'R' designa o ponto de origem e retorno do drone;
  - As demais letras (A, B, C, etc.) representam os pontos de entrega;
  - O caractere '0' representa uma posição vazia na matriz.

4.4. Algoritmo

Abaixo segue um pseudocódigo de cada função a fim de demonstrar-se o algoritmo construído e utilizado:

4.4.1. Função main(), presente no arquivo main.py, função principal do programa

Passo	Descrição / Comando
1	INÍCIO_PROGRAMA *main(cronometro=False)*

2	EXIBIR mensagem de boas-vindas.
3	SOLICITAR ao usuário o caminho do arquivo de mapa.
4	ARMAZENAR o caminho na variável *caminho*.
5	SOLICITAR modo de execução (*fast / plus*)
6	NORMALIZAR *(strip().lower())* e ARMAZENAR em *modo*
7	TENTAR FAZER:
8	SE *cronometro == True:*
	tempo_inicial ← perf_counter()
	dados_do_mapa = CHAMAR a função *parseArquivo* com *caminho*.
9	EXIBIR para o usuário as informações do arquivo (dimensões, matriz, pontos).
9	SE *modo == "plus":*
	Modo detalhado - *(melhor_rota, melhor_custo, pior_rota, pior_custo) ← otimizarRotaPlus(pontos_mapeados["pontos"], mostrar_todas=True)*
10	SENÃO:
	Modo rápido - *(melhor_rota, melhor_custo) ← otimizarRota(pontos_mapeados["pontos"], mostrar_atualizacoes=False)*
	pior_rota, pior_custo ← None
11	SE *cronometro == True:*
	duracao ← perf_counter() − tempo_inicial
12	SENÃO:
	duracao ← 0
13	*EXIBIR resultado final* (*melhor_rota* e *melhor_custo*)
14	SE *pior_rota* ≠ None:
	EXIBIR pior_rota e *pior_custo*
15	SE *cronometro == True:*
	EXIBIR duracao
16	SE OCORRER UM ERRO:
	EXIBIR uma mensagem de erro apropriada.
17	FIM_PROGRAMA

4.4.2. Função parseArquivo(), presente no arquivo parser.py, utilizada para leitura do arquivo de entrada (matriz)

Passo	Descrição / Comando
1	FUNÇÃO *parseArquivo(caminho_do_arquivo)*
2	CHAMAR uma função auxiliar *lerArquivo* para obter uma lista de linhas do arquivo.

3	EXTRAIR *numero_de_linhas* e *numero_de_colunas* da primeira linha.
4	INICIAR uma *matriz* vazia.
5	PARA CADA linha restante na lista de linhas:
6	Adicionar a linha (dividida em células) para a *matriz*.
7	INICIAR um dicionário *pontos* vazio.
8	PARA CADA linha (índice i) E coluna (índice j) na *matriz*:
9	SE o valor na *matriz* [i] [j] for um ponto de interesse (diferente de "0"):
10	ARMAZENAR em *pontos* o nome do ponto e suas coordenadas (i, j).
11	RETORNAR um objeto contendo: *numero_de_linhas*, *numero_de_colunas*, *matriz* e *pontos*.
12	FIM_FUNÇÃO

4.4.3. Função otimizarRota(), presente no arquivo otimizador.py, função chave para o principal requisito funcional do projeto: retornar a melhor rota dentre todas as possibilidades

Passo	Descrição / Comando
1	FUNÇÃO *otimizarRota(pontos)*
2	CRIAR uma lista *pontos_de_entrega* contendo todos os pontos exceto 'R'.
3	GERAR todas as permutações possíveis de *pontos_de_entrega*.
4	ARMAZENAR em uma lista *rotas_possiveis*.
5	INICIAR *melhor_rota* como nula.
6	INICIAR *menor_custo* com um valor infinito.
7	PARA CADA *rota_atual* em rotas_possiveis:
8	*custo_atual* = CHAMAR a função *calcularCustoTotalDaRota(rota_atual, pontos)*.
9	SE *custo_atual* for menor que *menor_custo*:
10	ATUALIZAR *menor_custo* para o valor de *custo_atual*.
11	ATUALIZAR *melhor_rota* para ser a *rota_atual*.
12	FORMATAR a *melhor_rota* em uma string com espaços.
13	RETORNAR a string da *melhor_rota*, então formatada.
14	FIM_FUNÇÃO

4.4.4. Função otimizarRotaPlus(), presente no arquivo otimizador.py, versão detalhada que lista todas as rotas avaliadas, ordena por custo e retorna também a pior rota

Passo	Descrição / Comando
1	FUNÇÃO *otimizarRotaPlus(pontos, mostrar_todas=False)*
2	CRIAR uma lista *pontos_de_entrega* contendo todos os pontos exceto 'R'.
3	PARA CADA *ponto* em *pontos*:
4	SE *ponto* ≠ 'R': ADICIONAR *ponto* em pontos_de_entrega
5	SE pontos_de_entrega estiver vazia:
6	RETORNAR *mensagem "Nenhum ponto de entrega foi especificado."*
7	GERAR todas as permutações possíveis de *pontos_de_entrega*.
8	ARMAZENAR em uma lista *rotas_possiveis*.
9	CRIAR lista vazia *resultados*
10	PARA CADA *rota_atual* em rotas_possiveis
11	*custo = calcularCustoTotalDaRota(rota_atual, pontos)*
12	ADICIONAR (*rota_atual*, *custo*) em *resultados*.
13	ORDENAR resultados em ordem crescente de custo
14	SE *mostrar_todas* for *True*
15	IMPRIMIR cabeçalho "Rotas avaliadas e seus custos:"
16	PARA CADA (*rota*, *custo*) enumerada em resultados:
17	IMPRIMIR índice, rota formatada com ' -> ' e *"Custo total: {custo}"*.
18	OBTER a melhor e pior rota e custo
19	RETORNAR tupla (*melhor_formatada, melhor_custo, pior_formatada, pior_custo*)
20	FIM_FUNÇÃO

4.4.5. Função calcularCustoTotalDaRota(), presente no arquivo otimizador.py, utilizada para calcular o custo de uma das rotas possíveis

Passo	Descrição / Comando
1	FUNÇÃO *calcularCustoTotalDaRota(rota, pontos)*
2	INICIAR *custo_total* como 0.

3	ADICIONAR ao *custo_total* a *calcularDistancia* de 'R' ao primeiro ponto da rota.
4	PARA CADA ponto na rota (até o penúltimo):
5	ADICIONAR ao *custo_total* a *calcularDistancia* do ponto atual ao próximo ponto.
6	ADICIONAR ao *custo_total* a *calcularDistancia* do último ponto da rota de volta para 'R'.
7	RETORNAR *custo_total*.
8	FIM_FUNÇÃO

4.4.6. Função calcularDistancia( ), presente no arquivo otimizador.py, baseada na distância manhattan:

Passo	Descrição / Comando
1	FUNÇÃO *calcularDistancia(ponto_A, ponto_B)*
2	*distancia_manhattan* = |*linha_A* - *linha_B*| + |*coluna_A* - *coluna_B*|.
3	RETORNAR *distancia_manhattan*.
4	FIM_FUNÇÃO

4.5. Geração de rotas com *itertools.permutations*

A geração sistemática das rotas é feita com *itertools.permutations*, que produz, de forma iterável, todas as ordenações possíveis dos pontos de entrega (excluindo R). Para um conjunto com *n* pontos, são geradas *n!* rotas distintas, o que explica o crescimento fatorial observado nos tempos de execução (ver Seção 5).

- **Modo FAST (otimizarRota)** – Itera diretamente sobre permutations(*pontos_de_entrega*) sem armazenar todas as rotas. A cada rota, calcula-se o custo e atualizam-se apenas os melhores (e, opcionalmente, as "atualizações"). Essa abordagem economiza memória e evita a ordenação de uma lista com *n!* elementos.

- **Modo PLUS (otimizarRotaPlus)** – Armazena cada par *(rota, custo)* em uma lista para, ao final, **ordenar por custo** e exibir todas as rotas, além de reportar explicitamente a pior. Essa opção é mais didática para relatórios, porém mais custosa em tempo e memória.

4.6. Exemplo de Execução

Com o objetivo de ilustrar o funcionamento do algoritmo desenvolvido, será apresentado um exemplo prático de execução do programa FlyFood a seguir. O sistema implementado em Python, foi estruturado em três módulos principais:

- **main.py:** responsável pela interação com o usuário e controle da execução;
- **parser.py:** encarregado de ler e interpretar o arquivo de entrada;
- **otimizador.py:** funções de cálculo (distância/custo) e otimização (modos fast e plus).

O programa solicita ao usuário o caminho de um arquivo .txt (ou .csv) contendo uma matriz que representa o mapa da cidade. Nessa matriz, o caractere “R” indica o ponto de partida e retorno do drone, as letras A, B, C, D, ... representam os pontos de entrega e o número 0 indica uma célula vazia.

Como citado anteriormente, a primeira linha do arquivo é sempre composta por dois números inteiros que indicam as dimensões da matriz. A partir dessa informação, o programa sabe quantas linhas deve ler e como estruturar a grade que representa o ambiente de entrega. A seguir, apresenta-se o conteúdo de um arquivo de entrada utilizado como exemplo de execução:

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

Nesse exemplo, a primeira linha (*4 5*) indica uma matriz de 4 linhas por 5 colunas.
Assim, o caractere “R” define o ponto de partida e retorno, as letras “A”, “B” , “C” e “D” indicam os locais de entrega, e o caractere “0” representa células vazias do grid.

Em seguida, será questionado ao usuário o modo de execução desejado, sendo eles:

- FAST — otimizarRota(): percorre todas as permutações sem armazená-las; mantém apenas a melhor solução encontrada e, opcionalmente, imprime as atualizações de melhor rota durante a busca).

É o modo recomendado para entradas maiores por reduzir overhead de memória e evitar ordenações.

- PLUS — otimizarRotaPlus(): armazena e ordena todas as rotas por custo; pode imprimir a lista completa e ordenada de rotas com seus "dronômetros", e informa, não só a melhor, como também a pior rota. Indicado para relatórios e demonstrações.

O custo é calculado pela função *calcularCustoTotalDaRota()*, que utiliza a *Distância de Manhattan* como métrica de deslocamento. Após a leitura (via parseArquivo()), o programa exibe dimensões da matriz, conteúdo do mapa e coordenadas dos pontos.

```
Número de linhas: 4
Número de colunas: 5

Matriz:
0 0 0 0 D
0 A 0 0 0
0 0 0 0 C
R 0 B 0 0

Pontos mapeados:
- A: (1, 1)
- B: (3, 2)
- C: (2, 4)
- D: (0, 4)
- R: (3, 0)
```

Ao final, o sistema exibe apenas o tempo total de execução e a melhor rota identificada com seu custo, na versão fast , como no exemplo abaixo:

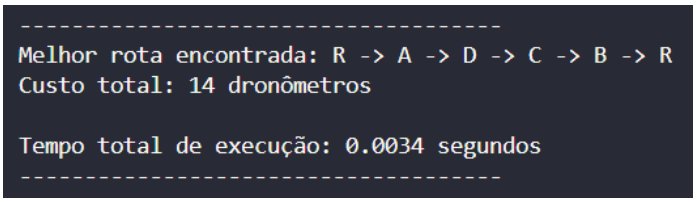

Figura 3. Saída de terminal modo *FAST*

Sendo opção Plus, teremos todas as rotas encontradas, de maneira ordenada pelo custo total, a melhor e pior rota encontrada, além do tempo de execução, conforme exemplo:

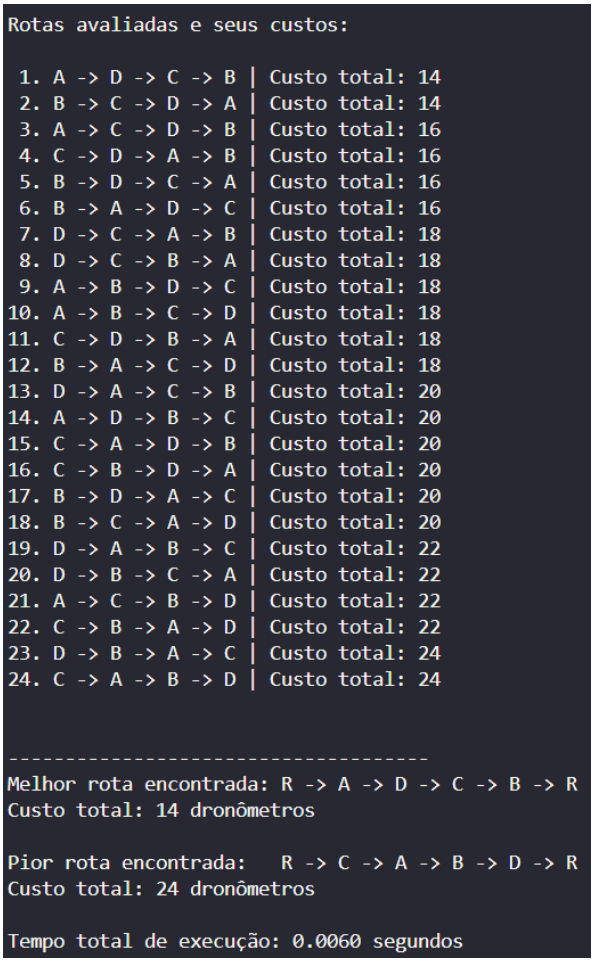

Figura 4. Saída de terminal modo *PLUS*

4.7. Repositório

O repositório do projeto e código-fonte completo encontra-se publicamente hospedado no site do *GitHub* (uma plataforma de hospedagem gratuita de repositórios *Git* e colaboração no desenvolvimento de programas entre usuários) através do seguinte link: https://github.com/pedroailton/flyfood-pisi-2.

5. Experimentos

Os experimentos desenvolvidos envolveram os seguintes parâmetros:

1. Tempo de execução em função da quantidade de pontos de interesse (pontos de entrega) na entrada;

2. Uma rota simples (não otimizada) em comparação a uma Rota ótima calculada pelo algoritmo da Fly Food.

Todas as execuções do programa foram realizadas em um mesmo computador, máquina com as seguintes especificações:

- Sistema operacional Windows 11;
- 20GB de memória RAM;
- Processador Intel Core i5 1334U;
- 512 GB de Armazenamento SSD;
- Executado na IDE Visual Studio Code em sua versão 1.104.3 (user setup);
- Ambiente Python na versão 3.14.0.

5.1. Rota Calculada Em Comparação A Uma Rota Não Otimizada

Para este experimento, foi utilizada a seguinte matriz exemplo:

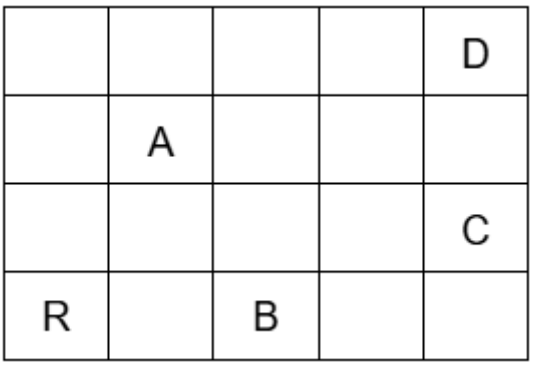

Figura 5. Matriz exemplo para o experimento da comparação de rotas. Fonte: autor.

Foi traçada a seguinte rota, seguindo a lógica de formação de uma rota circular (um algoritmo simples e aleatório), solução que poderia ser proposta sem a análise e uso do sistema de otimização de rotas:

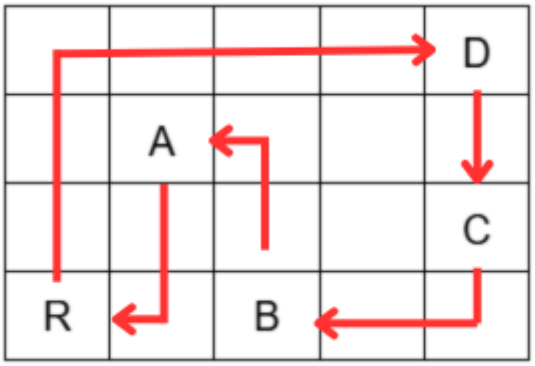

Figura 6. Rota traçada pelo algoritmo aleatório, no experimento de comparação de rotas. Fonte: autor.

Essa rota possui custo total **18**.

E também foi traçada a rota ótima calculada pelo programa Fly Food com a mesma matriz exemplo de entrada:

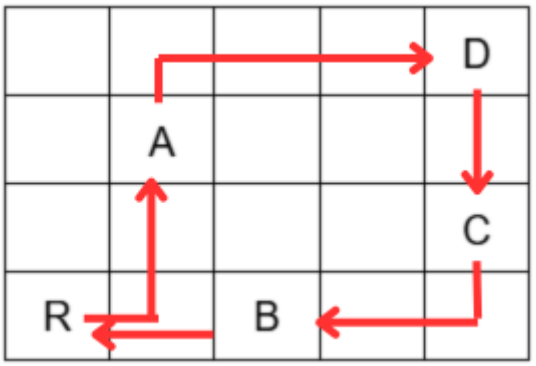

Figura 7. Rota traçada pelo algoritmo de otimização desenvolvido, no experimento de comparação de rotas. Fonte: autor.

Essa rota tem custo total **14**.

5.2. Tempo De Execução Em Função Da Quantidade De Pontos De Interesse Na Entrada

O programa foi executado usando como entrada arquivos com pontos de entrega definidos (entrada considerada nos processos principais do programa) que variaram de 1

até 11, presentes na subpasta “entradas” presente na pasta “flyfood” do nosso repositório.

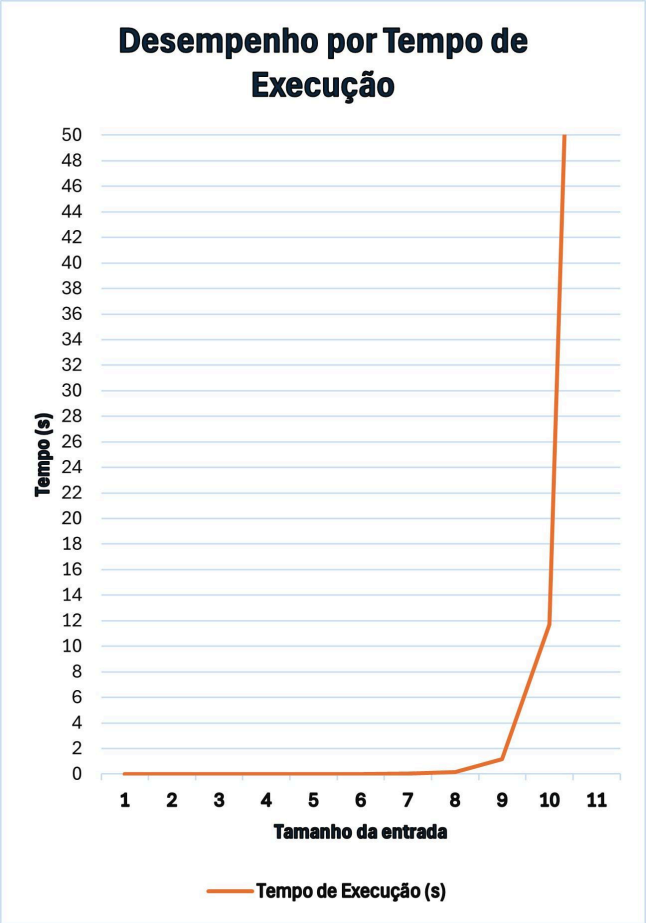

Figura 8. Gráfico dos resultados obtidos no experimento de desempenho por tempo de execução

Tamanho da entrada	Tempo de Execução (s)
0	0,0034
1,00	0,0048
2,00	0,0082
3,00	0,0155
4,00	0,0062
5,00	0,0062
6,00	0,0093
7,00	0,0374
8,00	0,1407
9,00	1,1652
10,00	11,7064
11,00	131,8047

Figura 9. Tabela de dados obtidos no experimento de tempo de execução.

6. Resultados

Nesta seção, são apresentados e analisados os resultados obtidos nos experimentos, tanto em relação à eficácia da otimização quanto à eficiência computacional do algoritmo.

6.1 Validação da Otimização da Rota

O primeiro resultado valida que o algoritmo cumpre seu objetivo principal de encontrar uma rota ótima. Em um experimento de comparação, a rota calculada pelo programa demonstrou ser **4 unidades de distância mais curta** do que uma rota traçada aleatoriamente, comprovando a eficácia da solução de força bruta. A metodologia e os dados detalhados deste experimento são apresentados na **Seção 5.1**.

6.2 Análise de Performance e Complexidade Computacional

A análise do tempo de execução revelou que, embora eficaz, o algoritmo possui um custo computacional que cresce de forma explosiva. Conforme os dados, uma entrada com 11 pontos de entrega resultou em um tempo de execução superior a dois minutos, o que seria impraticável em aplicações comerciais reais como a da empresa Flyfood.

Para comprovar que este crescimento é de ordem fatorial, foi aplicado o Critério da Razão de Crescimento Fatorial. Este critério baseia-se na propriedade recursiva da função fatorial:

$$n! = n \times (n-1)! \Rightarrow \frac{n!}{(n-1)!} = n$$

Analogamente, se o tempo de execução t(n) possui complexidade O(n!), a razão entre duas medições consecutivas deve se aproximar de n:

$$\frac{t(n)}{t(n-1)} \approx n$$

A tabela a seguir aplica este critério aos dados coletados:

Tamanho da Entrada (n)	Tempo de Execução t(n) [s]	Razão de Crescimento $[\frac{t(n)}{t(n-1)}]$	Valor Teórico Esperado (n)	Observação
0	0,0034	-	-	Tempo de inicialização do programa.
1	0,0048	1,41	1	Ruído domina a medição.
2	0,0082	1,71	2	Ruído domina a medição.
3	0,0155	1,89	3	Ruído domina a medição.
4	0,0062	0,4	4	Anomalia na medição (tempo menor que o anterior).
5	0,0062	1	5	Anomalia na medição.
6	0,0093	1,5	6	Anomalia na medição.
7	0,0374	4,02	7	O padrão de crescimento começa a emergir.
8	0,1407	3,76	8	O padrão se torna mais consistente.
9	1,1652	8,28	9	Forte evidência de crescimento fatorial.
10	11,7064	10,05	10	Forte evidência de crescimento fatorial.
11	131,8047	11,26	11	Forte evidência de crescimento fatorial.

6.3. Discussão e Implicações

Os resultados demonstram que a solução proposta é correta e aplicável, mas eficiente apenas em cenários com um número reduzido de entregas. Para aplicações em escala real, com dezenas de pontos, o tempo de execução fatorial torna o algoritmo inviável.

No entanto, esta validação é fundamental. A implementação de força bruta, por garantir a rota verdadeiramente ótima, serve agora como uma referência (benchmark) precisa para o desenvolvimento e avaliação de algoritmos heurísticos. Tais heurísticas, que serão desenvolvidas na segunda etapa da disciplina, podem não encontrar a solução ótima em todos os casos, mas visam fornecer soluções "boas o suficiente" em um tempo computacionalmente viável, e sua eficácia poderá ser medida em comparação com os resultados ótimos aqui estabelecidos.

7. Conclusão

O presente trabalho teve como objetivo o desenvolvimento e a validação de uma solução computacional para o problema de otimização de rotas de entrega, no contexto do projeto Flyfood. Ao final do projeto, é possível concluir que os objetivos propostos foram plenamente alcançados.

A ferramenta desenvolvida em Python demonstrou ser capaz de encontrar a rota ótima garantida para um dado conjunto de pontos de entrega. Isso foi realizado através da implementação de um algoritmo de força bruta que avalia todas as permutações possíveis, conforme implementado nas funções *otimizarRota* e *otimizarRotaPlus*. Adicionalmente, a funcionalidade de identificar a pior rota também foi implementada com sucesso, oferecendo um contraste que quantifica o ganho obtido pela otimização e contextualiza o valor da solução encontrada.

A principal limitação da solução, conforme previsto na análise teórica e confirmado pelos experimentos de performance, é a sua complexidade de tempo fatorial (O(n!)). Essa complexidade é uma consequência direta da abordagem de força bruta que, embora garanta a otimalidade, torna o programa inviável para problemas com um número de entregas superior a uma dezena, devido ao tempo de execução exponencialmente crescente.

A constatação dessa limitação abre um caminho claro para trabalhos futuros. A próxima etapa natural seria a implementação de algoritmos heurísticos (como Algoritmos Genéticos ou Simulated Annealing) ou de aproximação (como o Vizinho Mais Próximo) para encontrar soluções "boas o suficiente" em um tempo computacionalmente viável. A solução de força bruta desenvolvida neste trabalho servirá como um benchmark essencial e confiável para medir a precisão e a eficiência dessas novas abordagens.

Portanto, o projeto Flyfood cumpre seu escopo inicial com sucesso, entregando uma solução funcional e correta, e estabelecendo uma base sólida e validada para futuras otimizações que visem a escalabilidade e a aplicação em cenários comerciais mais complexos.

Referências Bibliográficas

INÁCIO, Diego. **Métricas de distância e dissimilaridade.** Medium, 2021. Disponível em: https://diegoinacio.medium.com/metricas-de-distancia-e-dissimilaridade-94f9d8d962d4. Acesso em: 4 out. 2025.

PYTHON SOFTWARE FOUNDATION. **Built-in Functions.** [S. l.]: Python Software Foundation, 2025. Disponível em: https://docs.python.org/3/library/functions.html. Acesso em: 4 out. 2025.

ROSEN, Kenneth H. **Matemática Discreta e Suas Aplicações**. 8. ed. Porto Alegre: AMGH Editora, 2019. Cap. 3: Algoritmos, Números Inteiros e Matrizes.

VIEIRA, Luiz. **Geometria Computacional: Determinar o vizinho mais próximo.** Stack Overflow em Português, 2016. Disponível em: https://pt.stackoverflow.com/questions/163899/geometria-computacional-determinar-vizinho-mais-pr%C3%B3ximo. Acesso em 11 out. 2025.

Apêndice A

Durante o desenvolvimento do projeto Flyfood, foram criados alguns scripts e funções com o objetivo de facilitar a validação dos algoritmos, a geração de casos de teste e a visualização dos dados. Embora não façam parte do fluxo principal da aplicação final, eles foram fundamentais para garantir a corretude e a qualidade do código. Essas funções auxiliares estão presentes apenas no modo “plus”, selecionado pelo usuário na execução do programa, que aciona a função *otimizarRotaPlus()*, que possui as mesma funcionalidades da função otimizarRota(), e as auxiliares. Esta seção documenta essas ferramentas auxiliares.

A.1. Print de Todas as Atualizações de Rota

Durante a execução, o programa apresenta visualmente a atualização de cada melhor rota encontrada até o momento, indicando também o custo total associado a essa rota.

```
1ª atualização da melhor rota: A C B D; Custo total: 18
2ª atualização da melhor rota: A C D B; Custo total: 16
3ª atualização da melhor rota: B A C D; Custo total: 14
```

Figura 9. Print das atualizações da melhor rota, no terminal.

Essa funcionalidade permite que o usuário observe o aperfeiçoamento progressivo da solução em tempo real.

A.2. Cálculo da Pior Rota

A pior rota é calculada com base em sua posição na lista de resultados possíveis, ordenada pelo método .sort() (essa função padrão do Python funciona com o Timsort, uma junção de Merge Sort e Insertion Sort, aplicada de maneira híbrida utilizando o que há de melhor em cada um desses dois algoritmos de ordenação). Para que a ordenação funcione corretamente em uma lista de tuplas (rota, custo), o método é chamado com o parâmetro key=lambda x: x[1], que instrui a função a ordenar a lista utilizando apenas o segundo elemento de cada tupla — o custo.

Como a ordenação padrão é ascendente (do menor para o maior), a pior rota, que possui o maior custo, é garantidamente o último elemento da lista ordenada (resultados[-1]).

Assim, a pior rota é apresentada no terminal da seguinte forma:

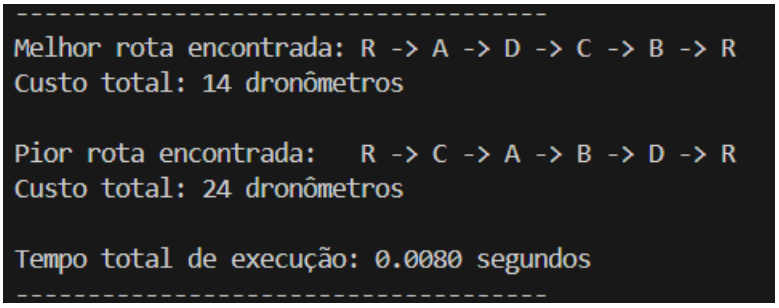

Figura 10. Recorte do terminal, demonstrando funcionamento do cálculo da pior rota.

A exibição da pior rota é uma funcionalidade crucial que enriquece a análise de resultados, pois permite ao usuário compreender a real dimensão da eficiência do algoritmo. Isso ocorre de três formas principais:

- Quantifica o Ganho da Otimização: A diferença entre o custo da melhor e da pior rota representa o ganho tangível (em unidades de distância) que o algoritmo proporciona;
- Contextualiza o Resultado Ótimo: A rota ótima, vista isoladamente, pode não transmitir seu valor completo. Ao contrastá-la com o pior cenário possível, o programa delimita o espectro completo de soluções, dando ao usuário uma noção clara de onde a solução ótima se encontra em relação a todas as outras possibilidades;
- Estabelece um Benchmark de Ineficiência: A pior rota serve como um "anti-objetivo", um benchmark claro do que se deve evitar.