Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

ALGORITMO DE ROTEAMENTO PARA ENTREGAS POR DRONE COM BASE NO PROBLEMA DO CAIXEIRO VIAJANTE

Isis Maria Oliveira Nilo de Souza

Recife

Fevereiro de 2023

Dedico a todos aqueles que estiveram presentes nessa jornada e me apoiaram direta, indiretamente e incondicionalmente.

Resumo

O setor de delivery tem enfrentado desafios em sua operação devido ao aumento da mobilidade urbana e aos elevados custos de mão de obra humana. A entrega por drone surgiu como uma solução viável e eficiente, porém, a limitação da bateria dos drones é um obstáculo que precisa ser superado. O Problema do Caixeiro Viajante (TSP) é uma possibilidade de otimizar o trajeto do drone de forma a maximizar o número de entregas que podem ser realizadas dentro do limite de tempo da bateria. O estudo apresentado utilizou a distância de Manhattan como método para calcular a distância entre dois pontos e um algoritmo de força bruta de roteamento implementado na linguagem de programação Python com base no TSP para maximizar a eficiência de delivery por drone. A aplicação desse algoritmo permite otimizar o planejamento das rotas de entrega, aumentando a eficiência e reduzindo os custos envolvidos.

Palavras-chave: Roteamento, Entrega por drone, Problema do caixeiro viajante, Algoritmo de força bruta, Python.

1. Introdução

Nos países emergentes, a urbanização crescente tem causado uma série de desafios e pressões no que diz respeito ao transporte e mobilidade, o que tem se tornado cada vez mais evidente desde a metade do século XX. Esse fenômeno é observado em todo o mundo. Com isso, o setor de delivery, que depende da mobilidade urbana, tem sofrido com atrasos, erros, devoluções e, em alguns casos, até acidentes de trabalho. Além disso, a mão-de-obra humana se tornou mais cara, tornando inviável para pequenas empresas manterem funcionários para realizar entregas. A tecnologia, no entanto, vem revolucionando diversas áreas da economia mundial, e o setor de delivery não foi exceção.

Com a automatização de processos, a entrega por drone se tornou uma solução viável e eficiente, permitindo o transporte de vários pedidos em diversos endereços e otimizando o tempo de entrega. O software de controle de voo do drone é projetado para planejar rotas predefinidas, desviar de obstáculos e atender aos dispositivos legais. Mas, ainda assim, a limitação de bateria dos drones é um obstáculo que precisa ser superado. É necessário otimizar o trajeto do drone de forma a maximizar o número de entregas que podem ser realizadas dentro do limite de tempo da bateria.

Neste contexto, esse trabalho investiga soluções para o Problema do Caixeiro Viajante (TSP) em busca de roteamento otimizado para realizar as entregas no tempo disponível de bateria dos drones. O TSP tem como objetivo encontrar o caminho mais curto que passe por todos os pontos uma única vez, começando e terminando no mesmo ponto, ou seja, a viagem de ida e volta mais curta.

Assim, com um conjunto de cidades e uma matriz de distâncias entre elas, é possível encontrar uma rota que: comece na cidade origem, passe por todas as outras cidades apenas uma vez, retorne à cidade origem ao final e percorra a rota de menor distância. Essa rota é chamada de ciclo fechado Hamiltoniano. O TSP é considerado um problema NP-difícil, o que significa que sua complexidade aumenta de forma exponencial com o aumento do número de cidades. Embora pareça simples, a solução ideal para o TSP pode requerer um grande custo computacional. No entanto, existem vários métodos aproximados e heurísticas que podem ser usados para resolver o problema, oferecendo resultados satisfatórios, mesmo que não sejam a solução perfeita.

O objetivo deste estudo é maximizar a eficiência de delivery por drone, por meio de cálculo do melhor caminho a partir da inserção de coordenadas em uma matriz que mostra os pontos de partida e os de entrega.

				D
	A			

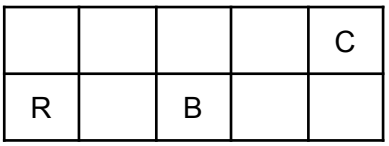

Figura 1. Representação de uma matriz de pontos

Para calcular a distância entre dois pontos, será utilizada a distância de Manhattan, também conhecida como distância "táxi". Essa medida é calculada como a soma das diferenças absolutas das coordenadas (x e y) de dois pontos, sem considerar a direção ou o ângulo entre eles. Assim, dado dois pontos A e B, com coordenadas (x_1,y_1) e (x_2,y_2), respectivamente, a distância entre eles pode ser obtida pela seguinte fórmula:

$$d\Gamma = | x_2 - x_1 | + | y_2 - y_1 |$$

Com isso, será utilizado um algoritmo de roteamento implementado na linguagem de programação Python com base no Problema do Caixeiro Viajante (TSP). A aplicação desse algoritmo permitirá otimizar o planejamento das rotas de entrega, aumentando a eficiência e reduzindo os custos envolvidos, garantindo a realização de todas as entregas dentro do limite de tempo da bateria do drone.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho apresenta uma solução para o desafio de roteamento de entregas por drone, visando encontrar o caminho mais curto para realizar todas as entregas dentro do limite de tempo da bateria do drone.

1.1.2 Objetivos Específicos

Para alcançar o objetivo geral, será preciso realizar os seguintes objetivos específicos:

- Efetuar pesquisas sobre o problema do caixeiro viajante e suas soluções existentes na literatura;
- Identificar implementações disponíveis para a solução do problema do caixeiro viajante na linguagem de programação Python;
- Construir um algoritmo de roteamento que seja capaz de definir o menor trajeto para o estudo de caso desejado;
- Analisar seus resultados e a eficácia do algoritmo.

1.2 Organização do trabalho

Além dessa breve introdução, o documento apresenta a seguinte estrutura:
No capítulo 2, é abordado brevemente o Problema do Caixeiro Viajante (TSP), incluindo sua formulação, complexidade e abordagem escolhida para resolver o

problema proposto. No capítulo 3, são descritos alguns trabalhos relacionados e a relação entre eles e o trabalho proposto. No capítulo 4, são apresentados os métodos, ferramentas e dados utilizados no trabalho. No capítulo 5, são relatados os experimentos realizados e o resultado dos mesmos. E por fim, no Capítulo 6, são apresentadas as conclusões finais deste estudo.

2. Referencial Teórico

2.1 Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante ou TSP (do inglês *Traveling Salesman Problem*) é um clássico problema de otimização combinatória que busca encontrar o menor caminho que percorra todos os pontos de um determinado conjunto uma única vez, começando e terminando em um mesmo ponto. Uma de suas aplicações são os problemas de roteamento que consistem em um conjunto de cidades ou localidades e uma matriz de distâncias entre elas, e o objetivo é encontrar uma rota que parta da cidade de origem, passe por todas as demais cidades uma única vez, retorne à cidade de origem ao final do percurso e percorra uma rota que dê a menor distância possível.

O TSP pode ser modelado utilizando uma representação baseada em grafos. Para uma melhor compreensão sobre o assunto, segue algumas definições:

- Um grafo é uma estrutura matemática composta por vértices (também conhecidos como nós) e arestas que conectam esses vértices. Os vértices representam entidades ou objetos e as arestas representam relações ou conexões entre eles. Os grafos podem ser direcionados ou não direcionados, dependendo de se as arestas têm direção ou não. Além disso, os grafos podem ser ponderados, o que significa que as arestas possuem peso associado a elas.
- Um ciclo hamiltoniano é um caminho em um grafo que visita cada vértice exatamente uma vez e termina no mesmo vértice de onde começou. Em outras palavras, é uma rota que passa por todos os vértices do grafo sem repetir nenhum deles.

O Problema do Caixeiro Viajante consiste em encontrar um ciclo hamiltoniano mínimo de um grafo que representa uma série de cidades e suas distâncias com o menor custo.

Sendo um grafo G = (V, E), onde V = {1, ..., n} é o conjunto de nós e E = {1, ..., m} é o conjunto de arestas de G. O custo, Cij, está relacionado a grandeza de cada aresta ligando os nós i e j. O somatório dos custos Cij resulta no custo total do caminho. O objetivo é encontrar o caminho com o menor custo possível.

A solução mais óbvia para o problema do caixeiro viajante é o método de enumeração, no qual é realizada a busca por todas as rotas possíveis. Essa abordagem consiste em

calcular o comprimento de cada rota através de um computador e, posteriormente, determinar qual é a rota mais curta, ou seja, com menor custo, escolhendo-a como a solução final.

2.1 Formulação do problema

A formulação do Problema do Caixeiro Viajante (TSP) pode ser representada como uma modelagem de programação linear inteira (PLI), onde se busca a rota mais curta para visitar todas as cidades em uma determinada ordem e retornar à origem. Uma das formulações mais conhecidas é a de Miller-Tucker-Zemlin (MTZ):

Dado um número *n* de cidades rotuladas de 1,…, *n*, sendo a cidade 1 a origem, as variáveis de decisão são definidas da seguinte forma:

$$x_{ij} = \begin{cases} 1 & \text{se a rota incluir um link direto entre cidades } i \text{ e } j, \\ 0 & \text{caso contrário.} \end{cases}$$

Para cada cidade $i = 1, 2, \ldots, n$, existe uma variável auxiliar $u_i \in \mathbb{R}^+$ e a distância entre essas cidades é definida como c_{ij}. Com isso, a formulação de MTZ para o TSP é:

$$\min \quad \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} c_{ij} x_{ij},$$

sujeito a

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1, \quad j = 1, 2, \ldots, n, \tag{1}$$

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1, \quad i = 1, 2, \ldots, n, \tag{2}$$

$$\begin{aligned} & u_i - u_j + n x_{ij} \leq n - 1, \quad 2 \leq i \neq j \leq n, \\ & x_{ij} \in \{0, 1\} \quad i, j = 1, 2, \ldots, n, \quad i \neq j, \\ & u_i \in \mathbb{R}^+ \quad i = 1, 2, \ldots, n. \end{aligned} \tag{3}$$

A formulação MTZ do Problema do Caixeiro Viajante (TSP) restringe as rotas a serem representadas por variáveis binárias e usa uma função objetiva que minimiza o comprimento total da rota. As restrições em (1) e (2) são impostas para garantir que cada cidade seja visitada apenas uma vez. Em (3), as restrições são necessárias para impedir soluções com subrotas (ou ciclos), já que as restrições anteriores não asseguram a formação de uma única rota que abranja todas as cidades.

A resolução do modelo PLI é geralmente realizada por meio de técnicas de otimização matemática, como o algoritmo branch and bound ou o simplex. É importante destacar

que essa abordagem pode ser computacionalmente intensiva e, como resultado, é menos utilizada em problemas de grande escala.

2.2 Complexidade e Problemas Computacionais

Os algoritmos podem ser classificados como tratáveis ou intratáveis com base em sua complexidade computacional. A complexidade de um algoritmo é a medida de sua eficiência em relação ao tempo ou ao espaço necessário para sua solução. Algoritmos tratáveis possuem complexidade polinomial e são capazes de solucionar problemas em um tempo razoável, enquanto algoritmos intratáveis apresentam complexidade exponencial, tornando-se inviáveis para soluções em problemas de grande escala devido ao tempo computacional exorbitante necessário para sua verificação.

As classes de problemas computacionais são agrupadas em problemas que possuem propriedades semelhantes, como a dificuldade de solução, a estrutura de dados necessária e a complexidade de tempo de processamento. Estas classes permitem comparar soluções distintas para o mesmo problema e escolher a mais adequada.

2.2.1 Algoritmos de tempo polinomial e exponencial

As principais classes de problemas computacionais de algoritmo de tempo polinomial e tempo exponencial são:

- Classe P (*Polynomial Time*): Classe de problemas computacionais cuja solução pode ser **encontrada** em tempo polinomial, ou seja, o tempo de resolução é proporcional a um polinômio do tamanho da entrada. Problemas desta classe são considerados teoricamente resolvíveis.

- Classe NP (*Nondeterministic Polynomial time*): Classe de problemas cuja solução pode ser **verificada** em tempo polinomial, mas cuja solução completa pode levar tempo exponencial. Problemas desta classe são considerados teoricamente não resolvíveis, mas são úteis para aplicações práticas.

- Classe NP-difícil: Classe de problemas computacionais cuja solução é considerada difícil, mas que ainda podem ser verificados em tempo polinomial. Problemas desta classe são considerados teoricamente não resolvíveis.

- Classe NP-completo: Classe de problemas computacionais que são considerados os mais difíceis dentro da classe NP. Problemas desta classe são NP-difíceis e podem ser usados como uma medida para comparar a dificuldade de outros problemas computacionais.

2.2.2 Complexidade do TSP

O Problema do Caixeiro Viajante (TSP) é considerado NP-difícil porque, ao contrário de problemas como o cálculo da soma de uma série, que podem ser resolvidos com uma solução ótima em tempo linear, o TSP é considerado de difícil solução pois a complexidade cresce exponencialmente à medida que o número de cidades é aumentado. Isso significa que o tempo de processamento para encontrar uma solução ótima para o TSP aumenta rapidamente à medida que o número de cidades é aumentado, tornando a solução inviável para grandes conjuntos de dados.

Por exemplo, para acharmos o número R(n) de rotas para um caso com n cidades, sendo a primeira delas fixa, utiliza-se o cálculo combinatório: R(n) = (n-1) x (n-2) x ... x 2 x 1. Reescrevendo de forma fatorial, temos: R(n) = (n-1)!. Assim, o método consiste em gerar cada uma dessas R(n) = (n - 1)! rotas, calcular o comprimento total das viagens de cada rota e ver qual delas tem o menor comprimento total. Ou seja, embora seja eficiente para conjuntos pequenos de pontos, esse método pode se tornar inviável quando o número de cidades aumenta, devido à essa complexidade computacional.

Além disso, o TSP também é considerado NP-difícil porque o problema é de difícil verificação, ou seja, é difícil determinar se uma solução dada é ótima. Por exemplo, se uma solução propõe visitar todas as cidades e retornar à cidade de origem em uma determinada ordem, seria difícil determinar se esta é realmente a menor rota sem comparar com todas as outras rotas possíveis.

Em resumo, o TSP é NP-difícil devido à combinação da dificuldade para encontrar uma solução ótima e a dificuldade para verificar a validade de uma solução proposta.

3. Trabalhos Relacionados

3.1 A Greedy Algorithm for the Drone Delivery Problem

O artigo "A Greedy Algorithm for the Drone Delivery Problem" de M. A. Ahmed et al. (2017) apresenta uma abordagem baseada em algoritmo guloso para solucionar o problema de roteamento de entregas por drone. O método proposto consiste na escolha da rota mais curta a cada etapa da entrega, priorizando o ponto de entrega mais próximo. O algoritmo apresentado é de fácil implementação e compreensão, tornando-o uma opção atrativa para soluções de baixo custo. Além de ser eficiente em termos de tempo de execução, especialmente em conjuntos de dados de tamanho moderado. Entretanto, a solução apresentada não garante a otimização em termos de tempo ou distância percorrida pelo drone. O algoritmo pode ser afetado por problemas de escolhas

prejudiciais, onde a escolha inicial de rota prejudica as escolhas subsequentes, resultando em soluções subótimas.

3.2 Drone Delivery Path Optimization Based on a Heuristic Algorithm

O artigo "Drone Delivery Path Optimization Based on a Heuristic Algorithm" por F. Y. Wang et al. (2019) apresenta uma solução para o desafio de roteamento de entregas por drone, com o objetivo de encontrar o caminho mais curto para realizar todas as entregas dentro do limite de tempo da bateria do drone. A solução proposta é um algoritmo heurístico baseado em duas técnicas: algoritmo genético e o algoritmo de vizinhança de swap. O algoritmo proposto é capaz de gerar soluções ótimas em um tempo razoável, mesmo para entregas em grande escala. Entretanto, pode ser considerado complexo, já que utiliza duas técnicas de otimização, o que pode dificultar sua implementação e manutenção. Além disso, a solução proposta depende da qualidade e precisão dos dados de entrada, como as coordenadas geográficas dos pontos de entrega e o tempo máximo de voo do drone. Se esses dados não forem precisos, o algoritmo pode gerar soluções imprecisas.

3.3 An Exact Algorithm for the Drone Delivery Problem

O artigo "An Exact Algorithm for the Drone Delivery Problem" de Y. Chen et al. (2019) apresenta uma solução exata para o desafio de roteamento de entregas por drone. A solução utiliza um algoritmo baseado em programação linear para encontrar o caminho ótimo de entrega. Suas vantagens de solução incluem a capacidade de encontrar a solução ótima de maneira eficiente, além de levar em consideração todos os aspectos do problema, incluindo restrições de tempo e capacidade de carga. Por outro lado, o uso de programação linear pode não ser viável para todos os casos. Isso acontece porque o tempo de processamento pode ser muito longo em casos de grande escala ou com muitas restrições, pois aumenta sua complexidade matemática, tornando difícil o modelamento do algoritmo.

4. Metodologia

4.1 Algoritmo de Força Bruta

A abordagem utilizada neste trabalho para solucionar o problema com base no TSP será o algoritmo de força bruta. Ele é um método de resolução de problemas que consiste em testar todas as possibilidades até encontrar a solução. Esse algoritmo funciona comparando todas as possíveis soluções, verificando se cada uma delas atende aos

critérios de otimização ou satisfação de restrições. É importante notar que ele não utiliza nenhum tipo de heurística ou otimização, como é o caso de algoritmos mais avançados. Em vez disso, ele testa todas as possibilidades até encontrar a solução. É uma abordagem simples e direta, mas pode ser extremamente demorada e ineficiente se o número de possibilidades for muito grande.

4.1.1 Notação Big-O

Big-O é uma notação matemática usada para descrever a complexidade de um algoritmo em termos de tempo e espaço. É usada para avaliar a complexidade do tempo de um algoritmo e é uma forma de se estimar o número máximo de operações que um algoritmo executará dado um tamanho específico de entrada – para o problema em questão, o tamanho de entrada é caracterizado pelo número de pontos presentes na matriz de coordenadas. A utilização da notação Big-O é uma ferramenta importante para avaliar o desempenho de um algoritmo e prevenir problemas relacionados à performance insuficiente da máquina.

O	Complexity	Rate of growth
O(1)	constant	fast
O(log n)	logarithmic	
O(n)	linear	
O(n * log n)	log linear	
O(n^2)	quadratic	
O(n^3)	cubic	
O(2^n)	exponential	
O(n!)	factorial	slow

Figura 2. Tabela de complexidade referente a notação Big-O

Fonte: https://jarednielsen.com/big-o-factorial-time-complexity/

A complexidade de tempo deste projeto, com base na quantidade de pontos de entrega (n) a serem visitados pelo drone, é O(n!). Isso se deve ao fato de que é necessário gerar todas as possíveis rotas que visitam cada ponto uma única vez e escolher a rota mais curta. Ou seja, é necessário gerar n! (fatorial de n) permutações dos pontos de entrega, o que leva a um tempo de processamento que cresce rapidamente à medida que o número de cidades aumenta. O fatorial de um número n é calculado como o produto de todos os números inteiros de 1 a n. Por exemplo, 5! = 5 x 4 x 3 x 2 x 1 = 120. É uma função que cresce muito rapidamente e é comumente usada como uma referência para a complexidade de tempo.

O cálculo da complexidade de tempo de um algoritmo fatorial pode ser feito a partir da análise do número de operações realizadas por ele. A complexidade é obtida a partir da expressão T(n) = n * T(n-1), onde n é o número a ser fatorado e T(n) representa o

número de operações realizadas para calcular o número fatorial de n. O cálculo pode ser feito através de uma série de multiplicações, onde cada multiplicação representa uma operação. O resultado final é T(n) = n * (n-1) * (n-2) * ... * 1, que é O(n!), como é mostrado na Figura 3 a seguir:

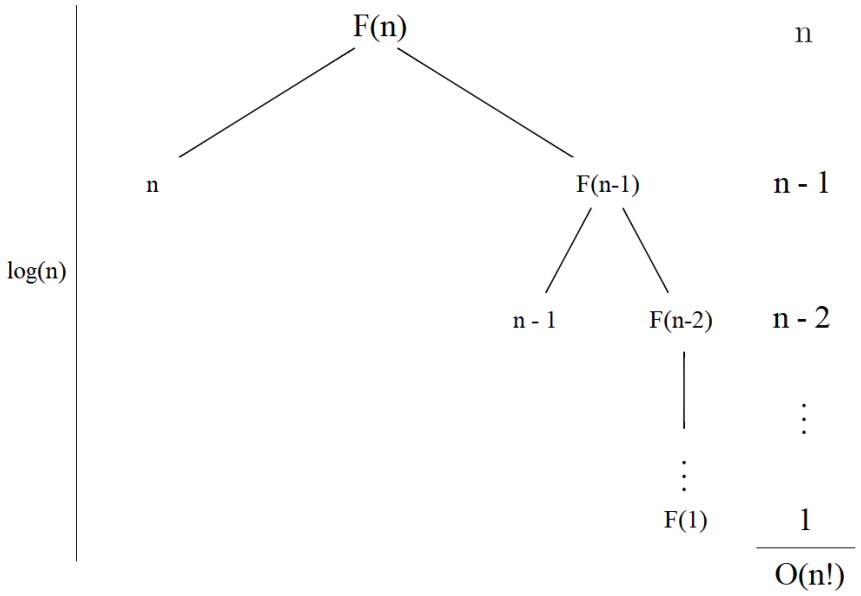

Figura 3. Árvore Fatorial O(n!)

A análise de complexidade usando a notação O é importante porque ajuda a identificar quais algoritmos são adequados para soluções eficientes para diferentes tipos de problemas. Isso permite que os desenvolvedores escolham a abordagem mais adequada para resolver um problema em questão, garantindo uma solução otimizada e escalável.

O algoritmo de força bruta pode ser inadequado para problemas com uma grande quantidade de possibilidades, como é o caso do problema do caixeiro viajante. Mas, para este projeto, não será necessário lidar com um número tão elevado de pontos de entrega a ponto de tornar o algoritmo ineficiente, o que não afetará os resultados esperados.

4.2 Construção do Algoritmo

Para se obter as coordenadas dos pontos de entrega e o ponto de origem e retorno, o algoritmo realiza a leitura de uma matriz (Figura 4) a partir de um arquivo.

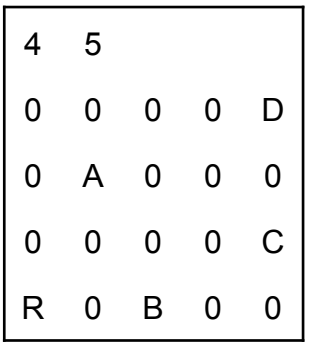

Figura 4. Matriz de exemplo de entrada do algoritmo.

Neste exemplo, foram utilizados pontos de entrega representados por caracteres, como 'A', 'B', 'C' e 'D', onde 'R' representa o ponto de origem e retorno. Resumidamente, o algoritmo aplicado sobre esses pontos retorna a ordem mais eficiente para que o drone percorrer, ou seja, o menor percurso, partindo do ponto de origem e retornando ao mesmo, passando por cada ponto de entrega uma única vez.

Para solucionar o desafio de roteamento de entregas por drone, foi desenvolvido um algoritmo de força bruta em Python 3 que calcula o caminho mais curto dentro do limite de tempo da bateria do drone para realizar todas as entregas. Ele faz isso lendo os dados de um arquivo de texto e armazenando as coordenadas de um ponto de origem e pontos de entrega em um dicionário. Em seguida, usa uma função para calcular a distância total da rota, que é a distância do ponto de origem a cada ponto de entrega e de volta ao ponto de origem. Após isso, usa duas funções para permutar todas as possíveis combinações de pontos de entrega e encontrar a combinação que resulta na rota mais curta. E finalmente, imprime a rota mais curta encontrada.

Para implementar a leitura do arquivo que contém a matriz de coordenadas dos pontos de entrega e o de origem/retorno em Python, aplicamos a lógica descrita a seguir em pseudocódigo:

```
programa algoritmo de roteamento
início

    abrir arquivo = “nome-do-arquivo”

    # Criação de lista para o ponto de origem e dicionário para as coordenadas
    origem <- []
    coordenadas <- {}

    # Armazenamento das coordenadas
    contador_linhas <- 0
    para cada linha no arquivo faça:
       contador_colunas <- 0

       para cada caractere na linha faça:
          se caractere = '0’ então:
             contador_colunas <- contador_linhas + 1

          se senão caractere = 'R' então:
             origem <- [contador_linhas, contador_colunas]
             contador_colunas <- contador_linhas + 1
```

```
            se senão caractere ≠ '\n' e caractere ≠ ' ' então:
                coordenadas[caractere] <- [contador_linhas, contador_colunas]
                contador_colunas <- contador_linhas + 1

            contador_linhas <- contador_linhas + 1
```

Assim, foram utilizados laços de repetições para identificar as coordenadas de cada ponto de entrega dentro da matriz no arquivo. Para auxiliar o algoritmo a percorrer todas as linhas e colunas da matriz, foram criadas duas variáveis que fazem a contagem de cada linha e coluna percorrida para garantir que todas elas sejam analisadas. Caso o caractere encontrado seja igual à 'R', suas coordenadas são armazenadas em uma lista chamada **origin** (origem). Mas caso o caractere seja diferente de '0', espaços ' ' e quebras de linha '\n', ele é armazenado em um dicionário **coordinates** (coordenadas) com chave igual ao caractere e valores iguais a coordenada encontrada.
Por exemplo, utilizando a Figura 4 como base, teremos:

origin = [3, 0]
coordinates = {'A': [1, 1], 'B': [3, 2], 'C': [2, 4], 'D': [0, 4]}

```python
file = open('file-name', 'r')
line = file.readline()

origin = []
coordinates = {}

row_counter = 0
for line in file:
  column_counter = 0
  for char in line:
    if char == '0':
      column_counter += 1
    elif char == 'R':
      origin = [row_counter, column_counter]
      column_counter += 1
    elif char != '\n' and char != ' ':
      coordinates[char] = [row_counter, column_counter]
      column_counter += 1
  row_counter += 1

file.close()
```

A seguir, é criada uma função **manhattan_distance** para calcular a distância entre os pontos através da Fórmula de Manhattan. Como citado no Capítulo 1, ela é definida como $dT = | x_2 - x_1 | + | y_2 - y_1 |$, dado dois pontos A e B, com coordenadas (x_1, y_1) e (x_2, y_2), respectivamente. As coordenadas são tidas como parâmetros nessa função.

```python
def manhattan_distance(x1, y1, x2, y2):
  return abs(x1 - x2) + abs(y1 - y2)
```

A função de Manhattan é utilizada para o cálculo da distância total das possibilidades de percurso em outra função chamada **route_distance** através da seguinte lógica:

```
função route_distance(lista):
        rota <- distância_manhattan (x-origem, y-origem, x-ponto1, y-ponto1)
        para i de 0 até (comprimento da lista - 1) faça:
                y1 <- coordenadas[lista[i]][0]
                x1 <- coordenadas[lista[i]][1]
                y2 <- coordenadas[lista[i + 1]][0]
                x2 <- coordenadas[lista[i + 1]][1]
                rota <- rota + distância_manhattan (x1, y1, x2, y2)
        rota <- rota + distância_manhattan (x-origem, y-origem,
                                            x-pontoN, y-pontoN)
retorne rota
```

Para realizar o cálculo de custo total de um percurso, a função **route_distance** utiliza como parâmetro um dicionário com os pontos de entrega, como por exemplo dic = { 'B': [3, 2], 'D': [0, 4], 'A': [1, 1], 'C': [2, 4] }. Essa lista serve para calcular a distância entre os pontos da função a partir de uma estrutura de repetição "for", funcionando da seguinte maneira:

1. A função **route_distance** realiza o cálculo das distâncias entre o ponto de origem e a primeira coordenada do dicionário obtida a partir da função **manhattan_distance** e atribui o valor obtido a uma variável chamada “route” (referente à rota).
2. Em seguida, o código calcula a distância entre todos os pontos dessa lista um através de um laço **for**.
   Para cada iteração, a função **manhattan_distance** é utilizada para calcular a distância entre dois pontos consecutivos e o resultado é acumulado na variável “route”. Por exemplo, se a lista tem comprimento 3, a estrutura será executada 3 vezes, de 0 a 2, calculando a distância entre os pontos B e D, D e A e A e C e somando o valor obtido em "route".
   No exemplo, supondo que a distância entre o ponto de origem e B seja 2 e a distância entre B e D seja 5, a variável "route" armazenará o valor 7 ao final da primeira execução. Em seguida, a distância entre D e A será calculada e adicionada ao valor armazenado em "route" até que todos os pontos da lista tenham sido percorridos.
3. Após isso, o algoritmo também leva em consideração o retorno do drone ao ponto de origem, calculando também a distância entre o último ponto da lista e a origem utilizando a função **manhattan_distance** e adicionando seu resultado à “route”.
   Exemplificando, se a soma da distância entre todos os pontos da lista, obtida por meio da variável "route", for 12 e a distância entre o último ponto da lista (ponto C) e a origem (ponto R) for 5, a distância total a ser percorrida será 17.
   Desta forma, a solução abrange a viagem completa do drone, desde o ponto inicial até o destino final, e o seu retorno.

```python
def route_distance(lst):
  route = manhattan_distance(origin[1], origin[0],
                             coordinates[lst[0]][1], coordinates[lst[0]][0])
  for i in range(0, len(lst) - 1):
    y1 = coordinates[lst[i]][0]
    x1 = coordinates[lst[i]][1]
    y2 = coordinates[lst[i + 1]][0]
```

```
        x2 = coordinates[lst[i + 1]][1]
        route += manhattan_distance(x1, y1, x2, y2)
    route += manhattan_distance(origin[1], origin[0],
                                coordinates[lst[len(lst) - 1]][1],
                                coordinates[lst[len(lst) - 1]][0])
    return route
```

Prosseguindo, são criadas as funções **permute** e **exchange** que juntas geram todas as permutações possíveis de uma lista e atualizam a rota mínima encontrada com base na função **route_distance**.

```
função permute(str, l):
        se l = comprimento de str então:
                custo_mínimo = route_distance(str)
                se custo_mínimo < menor_rota[0] então:
                        menor_rota[0] = custo_minimo
                        menor_rota[1] = str
                senão:
                        para j de l até comprimento de str faça:
                                função troca(str, l, j)
                                função permutação(str, l + 1)
                                função troca(str, j, 1)

função troca(str, n, m):
        aux = str[n]
        str[n] = str[m]
        str[m] = aux
```

A função **permute** é uma função de permutação recursiva. A função é chamada com dois argumentos: "str" e "l". "str" é uma lista de elementos e "l" é um índice. A função funciona recursivamente, gerando todas as permutações possíveis da lista "str".

Se o índice "l" for igual ao comprimento da lista "str", a função **route_distance** é chamada com a lista "str" como argumento. A função **route_distance** retorna o custo da rota. Se o custo retornado for menor que o custo mínimo armazenado na lista min_route (menor_rota), a lista min_route é atualizada com o novo custo mínimo e a lista "str".

Se "l" não for igual ao comprimento da lista "str", o laço "for" é executado para cada elemento posterior ao índice "l". Para cada elemento, a função **exchange** é chamada para trocar o elemento com o elemento em "l". Em seguida, a função **permute** é chamada com "str" e "l + 1" como argumentos. Finalmente, a função **exchange** é chamada novamente para trocar os elementos de volta para sua posição original.

A função **exchange** é uma função simples que troca dois elementos em uma lista "str". A função é chamada com três argumentos: "str", "n" e "m". A variável auxiliar "aux"

armazena o elemento em "n", o elemento em "n" é substituído pelo elemento em "m" e o elemento em "m" é substituído pelo elemento armazenado em "aux".

```
def permute(str, l):
  if l == len(str):
     min_cost = route_distance(str)
     if min_cost < min_route[0]:
        min_route[0] = min_cost
        min_route[1] = str.copy()
  else:
     for j in range(l, len(str)):
        exchange(str, l, j)
        permute(str, l + 1)
        exchange(str, j, l)


def exchange(str, n, m):
  aux = str[n]
  str[n] = str[m]
  str[m] = aux
```

Assim, a fim de obter a menor rota a ser percorrida pelo drone, os seguintes comandos são chamados no algoritmo:

```
delivery_points = list(coordinates.keys())
min_route = [route_distance(delivery_points), ' ']

permute(delivery_points, 0)

print(' '.join(min_route[1]))
```

Onde, com as funções **route_distance** e **permute** sendo definidas previamente no código e o dicionário coordinates estando preenchido com as coordenadas dos pontos de entrega a partir da matriz do arquivo de entrada, teremos:

1. A lista delivery_points é inicializada com as chaves do dicionário coordinates. Isso significa que delivery_points conterá uma lista de pontos de entrega.
2. A variável min_route é inicializada como uma lista com o resultado da chamada à função **route_distance** com delivery_points como argumento e uma string vazia como segundo elemento. Isso significa que min_route conterá o custo mínimo da rota e a própria rota mínima.
3. A função **permute** é chamada com delivery_points e 0 como argumentos. Isso gera todas as permutações possíveis da lista delivery_points e atualiza a variável min_route com a rota mínima encontrada.
4. O resultado final é a impressão da rota mínima, com as entregas separadas por espaços. Isso é feito usando o método join de strings com min_route[1] como argumento.

5. Experimentos

5.1 Entradas

Para avaliar a eficiência do código, foram criados 5 arquivos no formato '.txt' para realização de testes. Cada arquivo possui uma matriz de pontos de entrega de tamanho e quantidade de pontos diferentes, permitindo avaliar o desempenho do código em diferentes situações.

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

Entrada 1 (input-01)

6	7					
0	0	0	A	0	0	0
R	0	F	0	0	0	0
0	0	0	0	C	0	0
0	0	B	0	0	0	0
0	0	0	0	0	0	E
0	0	0	0	D	0	0

Entrada 2 (input-02)

8	9							
C	0	0	0	0	A	0	0	0
0	0	D	0	0	0	0	0	0
0	0	0	0	0	0	0	R	0
0	0	B	0	0	0	0	0	0

0	0	0	0	E	0	0	0	0
F	0	0	0	0	0	0	0	0
0	0	G	0	0	0	0	0	0
0	0	0	0	0	0	0	0	H

Entrada 3 (input-03)

10	11									
0	0	0	0	B	0	0	0	0	0	0
0	C	0	0	0	0	0	0	0	0	0
0	0	0	F	0	0	0	0	R	0	0
A	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	D	0
0	0	0	0	0	0	E	0	0	0	0
0	0	G	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	H
J	0	0	0	0	0	0	0	0	0	0
0	0	0	I	0	0	0	0	0	0	0

Entrada 4 (input-04)

12	13											
0	0	0	0	0	0	0	0	0	C	0	0	0
0	B	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	E	0	0
0	0	0	0	A	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	G	0	0	0	0	D
0	0	0	H	0	0	0	0	0	0	R	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	I	0	0	0	0	0	0	0
0	0	0	0	0	0	J	0	0	0	0	0	0
0	0	K	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	L	0	0

Entrada 5 (input-05)

Nas entradas utilizadas, a primeira linha da matriz identifica o número de colunas e linhas presentes na mesma, enquanto R representa o ponto de origem/retorno e os outros caracteres, como A, B, C e assim por diante, os pontos de entrega pelo drone.

5.2 Resultados

A tabela a seguir representa os resultados obtidos, tendo como dados de entrada as matrizes apresentadas no tópico 5.1, utilizando como solucionador o algoritmo baseado no problema do caixeiro viajante. Para cada arquivo de matriz avaliado, são apresentados a quantidade de pontos de entrega, possibilidades de percurso e o tempo de processamento em segundos. Além disso, a tabela evidencia quantas soluções ótimas o algoritmo de força bruta é capaz de mapear e a qual delas é impressa por ele após a sua execução.

Matriz de entrada	Pontos de entrega	Possibilidades de rotas (n!)	Tempo de execução (s)	Soluções ótimas	Saída do algoritmo (menor percurso)
input-01	4	24	0,016 s	2	A D C B
input-02	6	720	0,062 s	8	A C E D B F
input-03	8	40.320	2,703 s	10	A C D B E F G H
input-04	10	3.628.800	47,08 s	4	B F C A J I G E H D
input-05	12	479.001.600	-	-	-

Tabela 1. Resultados do experimento utilizando Algoritmo de Força Bruta

5.3 Análise de Resultados

Após a análise dos resultados, é constatado que o algoritmo de força bruta tem uma eficiência limitada na resolução de problemas de TSP. Embora ele seja capaz de mapear todos os percursos possíveis e encontrar soluções ótimas, a performance do algoritmo degrada-se mediante o aumento no número de nós ou pontos de entrega. Além disso, o tempo de execução aumenta significativamente conforme maior for o número de nós.

Figura 5. Gráfico de Tempo x Pontos de entrega

Isto é particularmente evidente na entrada 05, que, mesmo com apenas 12 pontos, exigiu recursos significativos de hardware e tempo de processamento, tornando-se uma dificuldade para obter uma solução ótima.

6. Conclusão

Este trabalho apresenta uma estratégia para resolver o desafio do roteamento de entregas por drone, visando encontrar a rota mais curta que permita realizar todas as entregas dentro do limite de tempo da bateria do drone. O algoritmo de força bruta com base no problema do caixeiro viajante foi utilizado para encontrar soluções ótimas. No entanto, após a análise dos resultados, ficou claro que o algoritmo de força bruta apresenta limitações na resolução de problemas de TSP. Embora seja capaz de explorar todas as rotas possíveis e encontrar soluções ideais, a eficiência do algoritmo diminui à medida que o número de pontos de entrega aumenta. Além disso, o tempo de processamento aumenta significativamente, o que requer recursos significativos de hardware e tempo de processamento. Isso torna difícil obter uma solução ideal, especialmente quando o número de pontos é alto.

Referências Bibliográficas

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: MIT Press.

BRASIL. MINISTÉRIO DO MEIO AMBIENTE. Sustentabilidade urbana: impactos do desenvolvimento econômico e suas consequências sobre o processo de urbanização em

países emergentes. Brasília: MMA, 2015. Textos para as discussões da Rio+20, volume 1.

FERREIRA, Rafael Fernandes; MARTINS, Ricardo Silveira. Entregas do e-commerce e mobilidade urbana: análise das relações entre fretes, prazos e locais de entrega. Revista ADM.MADE, vol. 25, n. 2, 2021.

GOMES, Lohany Letro Melo. Roteirização de serviços de manutenção bancária através de um estudo comparativo de bibliotecas para o problema do caixeiro viajante. 2022. 50 f. Monografia (Graduação em Engenharia de Controle e Automação) - Instituto Federal do Espírito Santo, Serra, 2022.

Gambini Santos, H.; A. M. Toffolo, T. TUTORIAL DE DESENVOLVIMENTO DE MÉTODOS DE PROGRAMAÇÃO LINEAR INTEIRA MISTA EM PYTHON USANDO O PACOTE PYTHON-MIP. Pesquisa Operacional para o Desenvolvimento, [S. l.], v. 11, n. 3, p. 127–138, 2019. DOI: 10.4322/PODes.2019.009. Disponível em: https://www.podesenvolvimento.org.br/podesenvolvimento/article/view/629. Acesso em: 1 fev. 2023.

Miyazawa, F.K., Viccari, J.P.B. Algoritmos e Heurísticas para o Problema do Caixeiro Viajante com Minimização de Energia. Relatório Técnico - Universidade Estadual De Campinas Instituto De Computação, 2021.

Machado, Dartagnan Scalon. Geração automática de trajetórias a partir da detecção de obstáculos utilizando visão computacional. TCC (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Engenharia de Controle e Automação. 2022.

Chen, Y., Wang, H., Zhang, Y., & Shao, J. (2019). An exact algorithm for the drone delivery problem. Journal of Ambient Intelligence and Humanized Computing, 10(7), 8181-8195. https://doi.org/10.1007/s12652-018-0702-y

AHMED, M. A. et al. A greedy algorithm for the drone delivery problem. Journal of Systems Science and Complexity, v. 30, n. 6, p. 1593-1610, 2017.

WANG, F. Y. et al. Drone delivery path optimization based on a heuristic algorithm. Journal of Ambient Intelligence and Humanized Computing, v. 10, n. 3, p. 787-808, 2019.

NIELSEN, Jared. Big O Factorial Time Complexity. [s.d.]. Disponível em: https://jarednielsen.com/big-o-factorial-time-complexity/. Acesso em: 9 fev. 2023.