



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

FlyFood

José Roberto Oliveira de Araújo Filho

Recife
Dezembro de 2022

Resumo

Dado o grande fluxo de automóveis nas cidades e o alto custo com mão-de-obra, há uma oportunidade para que empresas de delivery utilizem drones como veículos de entrega, de forma que realizem suas entregas mais rapidamente e tenham um custo menor. A partir disso, a FlyFood surgiu com o objetivo de desenvolver um algoritmo que calcule as distâncias dos pontos de entrega e encontre a rota de menor custo, a fim de otimizar os trajetos dos drones. Para tal, foi implementado um algoritmo de força-bruta, que gera todos os caminhos possíveis e encontra a menor rota. Contudo, por se tratar de um problema NP-Difícil semelhante ao Problema do Caixeiro-Viajante, este algoritmo mostrou-se eficiente apenas para um pequeno número de pontos de entrega.

Palavras-chave: Algoritmo; Otimizar; Força-bruta; NP-Difícil; Problema do Caixeiro-Viajante.

1. Introdução

1.1. Apresentação e motivação

Ao longo dos anos, as pessoas têm preferido cada vez mais fazer suas compras de forma online, evitando perder tempo indo até a loja física e esperando na fila para realizar suas compras, além de economizar o custo com o transporte. Com isso, as empresas de e-commerce vêm investindo gradativamente em caminhões, carros e vans para realizarem suas entregas mais rapidamente. Contudo, devido ao aumento do fluxo de automóveis nas cidades, as empresas de delivery vem enfrentando dificuldades para realizarem suas entregas, tendo um custo mais alto com entregadores por causa da mão-de-obra e não conseguindo fazer suas entregas em um tempo adequado.

Visto que entregas por vias terrestres têm sido um problema, observa-se uma oportunidade de solucionar tal questão utilizando drones para realizarem as entregas por via aérea. Apesar do drone não ter a capacidade de transporte de um automóvel - como um caminhão ou uma van, por exemplo -, ele mostra-se como uma maneira da empresa prestar seus serviços de forma mais eficiente - diminuindo o custo com entregadores, combustível, e o tempo gasto no trânsito.

Dada a alta demanda da sociedade por serviços de delivery e a dificuldade cada vez maior de locomoção por via terrestre, é necessária a implementação de uma alternativa por meio da tecnologia, a fim de: contribuir para a modernização dos serviços prestados pelas empresas; para desobstrução do tráfego nas cidades; e para que os clientes recebam suas mercadorias mais rapidamente; permitindo assim o aumento de produtividade de toda a sociedade e a evolução da tecnologia em busca de algoritmos e métodos mais eficientes.

1.2. Formulação do problema

Este problema tem como entrada uma matriz, na qual $\mathbf{P} = \{p_1, p_2, \dots, p_n\}$ representa o conjunto com n pontos p , incluindo o ponto de origem e retorno ('R'). As coordenadas de cada ponto são dadas pela sua posição ij na matriz; e a distância - medida em dronômetros -

entre cada ponto é dada por (d_{xy}) , onde $d_{xy} = d(p_x, p_y)$, que é a distância de p_x para p_y .

O objetivo aqui é otimizar a rota dos drones, ou seja, queremos minimizar o somatório das distâncias d_{xy} , de forma que consigamos encontrar o menor caminho. Logo, o nosso problema pode ser representado matematicamente por:

$$\min \sum d_{xy}$$

1.3. Objetivo

Desenvolver um algoritmo que calcule as distâncias dos pontos de entrega em uma matriz e encontre a rota de menor custo, de forma que otimize os trajetos dos drones para que eles realizem entregas mais rápidas e o máximo de entregas possíveis dentro da sua capacidade de bateria.

1.4. Organização do trabalho

Daqui em diante, o trabalho está organizado da seguinte forma: A Seção 2 apresenta o embasamento teórico para entender o problema e a solução desenvolvida. Três trabalhos relacionados a este são apresentados na Seção 3, com uma breve descrição de como eles funcionam e suas limitações. A metodologia utilizada para solucionar o problema deste trabalho está presente na Seção 4. As Seções 5 e 6 apresentam os experimentos realizados com o algoritmo desenvolvido e os resultados obtidos neste trabalho, respectivamente. As conclusões do trabalho estão na Seção 7.

2. Referencial teórico

Esta seção apresenta a base teórica necessária para compreender o tipo de problema que estamos lidando e como é feita a verificação de eficiência de um algoritmo. Sendo assim, aqui contém explicações da análise de algoritmos e das classes de problemas computacionais - sendo eles pilares fundamentais para o entendimento da solução desenvolvida, da complexidade e limitações do problema.

2.1. Análise de algoritmos

A análise de algoritmos é utilizada para pressupor os recursos necessários para que um determinado algoritmo seja executado - dentre esses recursos, os que mais buscamos mensurar são a memória e o tempo. O algoritmo mais eficiente é encontrado após fazer esta análise com vários outros algoritmos que resolvem o mesmo problema, e, por vezes, é possível até encontrar mais de um algoritmo eficiente (CORMEN, 2012).

De acordo com Cormen (2012), tendo em vista que o tempo que o algoritmo gasta para resolver um problema cresce de acordo com o tamanho da entrada, e, além disso, depende da máquina em que ele está sendo executado e da linguagem de programação utilizada, a eficiência de um algoritmo se dá pela quantidade de passos que ele executa para processar uma entrada de tamanho n - em que a entrada e n dependem do problema. Portanto, com a análise de algoritmos nós podemos saber quanto tempo um algoritmo leva para processar uma entrada de tamanho n (FEOFILOFF, 2015).

Quando vamos medir a eficiência de um algoritmo pelo tempo que ele despende para resolver um problema com uma entrada de tamanho n , nós utilizamos a análise assintótica. Essa análise nos diz qual a ordem de crescimento do tempo de execução de um algoritmo quando a entrada é muito grande (BRUNET, 2019).

A principal notação assintótica usada é a Notação O (Big O notation), pois ela define o limite superior do crescimento do tempo de execução de um algoritmo no pior caso. Ou seja, a Big O notation nos fornece o tempo de execução máximo de um algoritmo quando o valor de n da entrada é muito grande.

Por exemplo: Merge Sort é um algoritmo que ordena os elementos de uma estrutura de dados, geralmente chamada de array. A notação O desse algoritmo é $O(n \log_2 n)$, ou seja, o tempo de execução máximo que o Merge Sort leva para ordenar, no pior caso, um array com n elementos, é igual a $n \log_2 n$.

Sendo assim, o algoritmo mais eficiente é aquele que tem o menor Big O notation em comparação com outros algoritmos que buscam resolver o mesmo problema. Contudo, vale ressaltar que:

Devido a fatores constantes e termos de ordem mais baixa, um algoritmo cujo tempo de execução tenha uma ordem de crescimento mais alta pode demorar menos tempo para pequenas entradas do que um algoritmo cuja ordem de crescimento seja mais baixa. Porém, para entradas suficientemente grandes, um algoritmo $\theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\theta(n^3)$. (CORMEN, 2012).

2.2. Classes de problemas computacionais

Os problemas computacionais são divididos em classes de acordo com a sua complexidade, ou seja, de acordo com o tempo de execução do algoritmo no pior caso. Além disso, os problemas são divididos em *tratáveis* e *intratáveis*. Os problemas tratáveis, ou fáceis, são aqueles que um algoritmo resolve em tempo polinomial; já os problemas intratáveis, ou difíceis, são problemas que o algoritmo não consegue resolver em tempo polinomial. As classes de problemas que serão descritas a seguir serão a P, NP, NP-Completo e NP-Difícil, mas vale ressaltar que elas não são as únicas existentes.

A **classe P** compreende os problemas considerados como fáceis ou tratáveis, ou seja, problemas que podem ser resolvidos em tempo polinomial. Exemplos: problema do caminho mínimo e problema de ordenação.

A **classe NP** consiste nos problemas que são verificáveis em tempo polinomial, isto é, dada uma solução, conseguimos verificar em tempo polinomial se esta solução é válida ou não (OLIVETE, 2020?). Exemplo: problema do caminho Hamiltoniano.

O problema pertence à **classe NP-Completo** se ele também pertence à classe NP, destacando que, se existir um algoritmo que resolva um problema NP-Completo em tempo polinomial, todos os problemas NP também serão resolvidos em tempo polinomial. Exemplo: problema do circuito Hamiltoniano em grafo orientado.

Um problema *A* pertence à **classe NP-Difícil** se existir um problema NP-Completo *B*, de forma que *B* seja redutível à *A* em tempo polinomial (ACERVO LIMA, c2022). Assim sendo, um problema é NP-Difícil se ele for tão difícil quanto um problema NP-Completo. Exemplo: problema do caixeiro viajante. A diferença entre um problema NP-Completo e NP-Difícil é que, o problema é NP-Completo se ele fizer parte da classe NP e NP-Difícil; e para o problema ser

NP-Difícil ele tem que ser tão difícil quanto um problema NP, mas não necessariamente pertencer à classe NP (ACERVO LIMA, c2022).

2.3. Problema do Caixeiro Viajante

O problema do caixeiro viajante (PCV) consiste em encontrar a rota de menor distância em um grafo, partindo de uma cidade (vértice) de origem, passando por várias outras cidades uma única vez e retornando à cidade inicial. O custo da rota é calculado somando a distância (peso) entre cada vértice adjacente de um determinado caminho.

O problema deste trabalho é NP-Difícil, visto que ele é semelhante ao PCV. No entanto, é necessário elucidar que, até o momento em que este trabalho foi realizado, não foi descoberto nenhum algoritmo capaz de resolver problemas NP-Completo e NP-Difícil em tempo polinomial - mas também não foi provado que nenhum algoritmo desse tipo exista.

3. Trabalhos relacionados

Aqui são apresentados três trabalhos relacionados a este, com uma breve descrição de como eles funcionam e suas limitações.

3.1. Uma heurística aplicada ao Problema do Caixeiro Viajante

Este trabalho traz uma abordagem chamada Heurística Permutacional (HP*) para a solução do PCV. Este procedimento consiste em dividir o conjunto de soluções viáveis S em n vizinhanças $N(s_i)$ distintas entre si, onde para cada $1 \leq i \leq n$ s_i é uma permutação que inicia com o elemento i , e cada uma dessas vizinhanças será particionada em quatro sub vizinhanças [6].

Um dos objetivos do trabalho era obter uma solução mais próxima possível da solução ótima, com um baixo percentual de desvio e um tempo computacional aceitável. Porém os resultados obtidos mostram que a heurística utilizada não obteve sucesso nas soluções, apresentando uma média de desvio muito alta em relação à solução ótima.

3.2. Solving the Parallel Drone Scheduling Traveling Salesman Problem via Constraint Programming

Este trabalho propõe um modelo de programação por restrições com o objetivo de minimizar o tempo necessário que um caminhão e uma frota de drones levam para fazerem entregas a um conjunto de clientes [7].

Os resultados obtidos a partir dos experimentos realizados apontam que o método utilizado conseguiu encontrar a solução ótima para todas as instâncias normalmente consideradas na literatura [7].

3.3. Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante

Este trabalho visa utilizar heurísticas do tipo k -opt, que consiste em remover k arcos de um roteiro definido inicialmente e substituí-los por outros k arcos, buscando diminuir a distância total percorrida [8].

Os resultados deste trabalho mostram que a solução inicial tem grande importância na qualidade das soluções obtidas com o método k -opt. Observou-se também que quando os métodos de melhoria partem de uma solução ruim, as soluções obtidas com esses métodos também são ruins.

4. Metodologia

Esta seção apresenta os pseudo-códigos do algoritmo implementado para gerar as permutações dos pontos de entrega e calcular o custo de cada uma dessas permutações (caminhos).

4.1. Introdução

O problema deste trabalho consiste em encontrar a rota de pontos de entrega que tenha o menor custo de distância total. Para isso, precisamos seguir os seguintes passos:

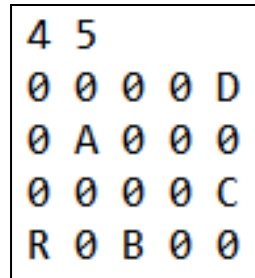
1. Ler a matriz de entrada.
2. Identificar quais são os pontos de entrega e suas posições na matriz.
3. Gerar todas as permutações possíveis com os pontos de entrega identificados.
4. Calcular a distância total de cada um desses caminhos e encontrar aquele de menor custo.

O algoritmo foi desenvolvido utilizando o método de força bruta, que consiste em gerar todos os possíveis caminhos e calcular o custo de cada um

deles. As subseções a seguir trarão o pseudo-código das soluções implementadas em cada passo.

4.2. Passo 1: Ler matriz de entrada

A matriz de entrada deve estar em um arquivo '.txt', com a primeira linha informando a quantidade de linhas e colunas da matriz. Veja, na imagem abaixo, um exemplo de como a matriz deve ser informada no arquivo:



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

O ponto 'R' representa o ponto de origem e retorno da rota.

Essa função:

- Identifica o número de linhas e colunas da matriz;
- Transforma cada linha em uma lista, e as adiciona em uma lista principal que representa a matriz;
- Elimina as quebras de linhas ('\n') de cada linha lida;
- Retorna o número de linhas e colunas e a lista que representa a matriz.

Função 1: Ler matriz

função

função_1 () : inteiro, inteiro, lista

var

quantidade_linhas, quantidade_colunas : inteiro

matriz_lida : lista

inicio

```
1: abrir 'arquivo.txt' no modo leitura
2:   ler primeira linha
3:     quantidade_linhas ← primeiro elemento da linha lida
4:     quantidade_colunas ← terceiro elemento da linha lida
5:   fim-lerprimeiralinha
6:   matriz_lida ← listavazia
7:   para cada_linha em ler linhas restantes faça
8:     matriz_lida.append(lista(cada_linha.split()))
9:   fimpara
10:  para cada_elemento em matriz_lida faça
11:    se cada_elemento[-1] == '\n' então
```

```

12:                cada_elemento ← cada_elemento[1:-1]
13:            fimse
14:    fimpara
15:    retorne quantidade_linhas, quantidade_colunas, matriz_lida
16: fechar-aquivo
    fimfunção

```

4.3. Passo 2: Identificar os pontos de entrega e suas posições

Essa função percorre toda a matriz para identificar quais são os pontos de entrega e suas coordenadas, baseadas na posição *ij* de cada ponto, além de remover o ponto de origem e retorno da lista de pontos de entrega. Ao final, a função retornará uma lista com todos os pontos de entrega e um dicionário com as coordenadas de cada elemento.

A **função 2** recebe como parâmetros o número de linhas e colunas e a matriz lida na **função 1**.

Função 2: Identificar os pontos de entrega

função

função_2 (linhas : inteiro, colunas : inteiro, matriz : lista) : lista, dicionário

var

pontos: lista

pontos_e_coordenadas : dicionário

inicio

```

1: para i de 0 até linhas faça
2:     para j de 0 até colunas faça
3:         se matriz[i][j] é letra então
4:             pontos.append(matriz[i][j])
5:             pontos_e_coordenadas[matriz[i][j]] = (i, j)
6:         fimse
7:     fimpara
8: fimpara
10: pontos.remove('R')
11: retorne pontos, pontos_e_coordenadas
    fimfunção

```

4.4. Passo 3: Gerar todos os caminhos possíveis

A terceira função é responsável por gerar todos os caminhos possíveis, e recebe como parâmetro a lista de pontos de entrega retornada na **função 2**.

Nesta função, nós usamos um loop para ‘travar’ cada ponto de entrega para gerar os caminhos que começam por esse ponto. A cada iteração do loop, chamamos a função recursivamente para que ela gere as permutações com cada um dos outros pontos restantes. No final, a função retornará uma lista com todos os caminhos possíveis.

Função 3: Gerar as permutações

função

função_3 (pontos : lista) : lista

var

combinações, pontos_não_visitados : lista

inicio

1: **se** len(pontos) == 1 **então**

2: **retorne** [pontos]

3: combinações ← listavazia

4: **para** i **de** 0 **até** len(pontos) **faça**

5: pontos_não_visitados ← pontos[0:i] + pontos[i+1:]

6: **para** cada_ponto **em** função_3 (pontos_não_visitados) **faça**

7: combinações.**append**([pontos[i]] + ponto)

8: **fimpara**

9: **fimpara**

10: **retorne** combinações

fimfunção

4.5. Passo 4: Encontrar a menor distância total

Antes de calcular a distância total de cada caminho, precisamos criar uma função que calcule a distância entre dois pontos. Para isso, usaremos as coordenadas de cada ponto de entrega identificadas na **função 2**. Como as coordenadas são dadas pelo par (i, j), calcularemos a distância da seguinte forma:

$$\text{abs}(i_a - i_b) + \text{abs}(j_a - j_b)$$

O ‘abs’ serve para garantirmos que o resultado da subtração não seja um número negativo.

Função 4: Calcular distância entre dois pontos

função

função_4 (ponto_a : inteiro, ponto_b : inteiro) : inteiro

var

distância : inteiro

inicio

1: distância = abs(ponto_a[0] - ponto_b[1]) + abs(ponto_a[0] - ponto_b[1])

2: **retorne** distância

fimfunção

Agora que temos a função que calcula a distância entre dois pontos, criamos a função que calcula a distância de cada ponto em cada caminho.

Primeiramente, essa função inicia em 9999 (ou qualquer outro número grande que você prefira) a variável que armazena o custo da menor rota, e atribuímos uma lista vazia para a variável que armazena o menor caminho.

Então, criamos um loop para verificar cada sublista da lista de rotas retornada na **função 3**, inicializando com 0 a variável que armazena o custo da menor rota e adicionamos o ponto de origem e retorno na sublista atual.

Logo após, fazemos um loop para iterar sobre a sublista que está sendo verificada no momento, de forma que calculemos a distância do ponto atual para o próximo, pegando suas posições no dicionário retornado na **função 2** e passando-as como parâmetro para a **função 4**.

Depois de calcularmos a distância total do caminho, verificamos se esse custo é o menor já calculado. Se for, atribuímos esse valor à variável que armazena o menor custo, e atribuímos esse caminho para a variável que armazena qual é a menor rota.

No final, quando já calculamos todas as distâncias, imprimimos qual foi a menor rota e o seu custo total.

Função 5: Calcular distância total de cada caminho

função

função_5 (caminhos : lista, coords : dicionário)

var

custo_menor_caminho : inteiro

menor_caminho : lista

custo_caminho_atual : inteiro

inicio

1: custo_menor_caminho ← 999

2: menor_caminho ← listavazia

3: **para** cada_caminho **em** caminhos **faça**

4: custo_caminho_atual ← 0

5: inserir ponto de origem no início do caminho

6: inserir ponto de retorno no final do caminho

7: **para** i **de** 0 **até** len(caminho) - 1 **faça**

8: coord_ponto_a ← coords[cada_caminho[i]]

9: coord_ponto_b ← coords[cada_caminho[i + 1]]

10: custo_caminho_atual ← função_4(coord_ponto_a,
coord_ponto_b)

11: **fimpara**

12: **se** custo_caminho_atual < custo_menor_caminho **então**

13: custo_menor_caminho ← custo_caminho_atual

14: menor_caminho ← caminho

15: **fimse**

16: **fimpara**

17: escreva(menor_caminho)

18: escreva(custo_menor_caminho)

fimfunção

4.6. Função principal

Esta função chama as funções citadas acima e armazena em variáveis os valores retornados de cada uma das funções.

Função main: Chamar as demais funções

função

função_main ()

var

linhas, colunas, matriz, pontos_de_entrega, rotas : lista
coordenadas : dicionário

inicio

1: linhas, colunas, matriz \leftarrow função_1 ()

2: pontos_de_entrega, coordenadas \leftarrow função_2 (linhas, colunas, matriz)

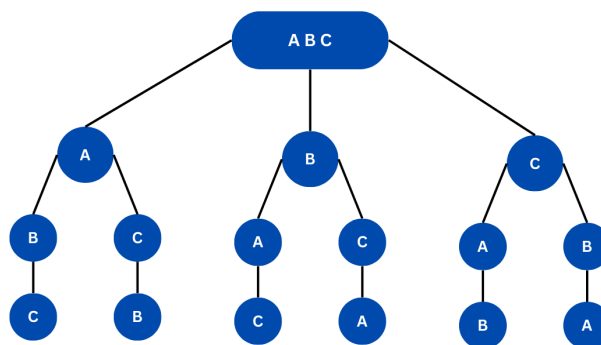
3: rotas \leftarrow função_3 (pontos_de_entrega)

4: função_5 (rotas)

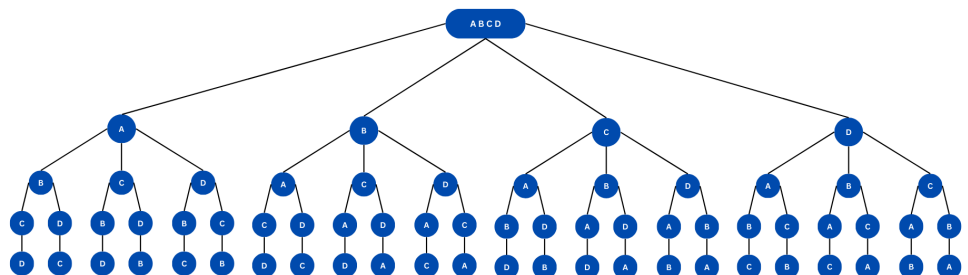
fimfunção

4.7. Complexidade

Para entender a complexidade deste algoritmo, será usado dois exemplos. Nesse primeiro exemplo (ver imagem abaixo) temos 3 pontos de entrega: 'A B C'. Para cada um dos pontos temos dois possíveis caminhos, resultando em 6 caminhos possíveis no total ($3 * 2 = 6$).



No exemplo seguinte (ver imagem abaixo), temos 4 pontos de entrega: 'A B C D'. Cada ponto de entrega inicial tem 3 possíveis pontos como secundário, e cada ponto secundário tem 2 possíveis caminhos, resultando em 24 rotas possíveis ($4 * 3 * 2 = 24$).



Percebe-se que a quantidade de caminhos cresce fatorialmente à medida que o número de pontos de entrega aumenta. Sendo assim, o algoritmo implementado é $O(n!)$, em que n é a quantidade de pontos de entrega de entrega.

5. Experimentos

Esta seção apresenta quatro experimentos realizados com o algoritmo desenvolvido, detalhando quais foram as entradas, os resultados e fazendo uma análise dos resultados obtidos.

5.1. Entradas

Foram realizados quatro experimentos, cada um deles com uma quantidade diferente de pontos de entregas. As entradas foram duas matrizes de ordem 5, com 4 e 6 pontos de entrega, e duas matrizes de ordem 6, com 8 e 10 pontos de entrega.

Veja as matrizes de entrada nas imagens abaixo:

Matriz 1:

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

Matriz 2:

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

Matriz 3:

6	6						
A	0	0	0	0	0		
0	0	0	0	B	0		
0	C	0	D	0	0		
0	0	0	0	G	0		
0	E	0	0	0	H		
R	0	0	F	0	0		

Matriz 4:

6	6						
A	0	0	0	0	I		
0	0	J	0	B	0		
C	0	0	D	0	0		
0	0	0	0	G	0		
0	E	0	0	0	H		
R	0	0	F	0	0		

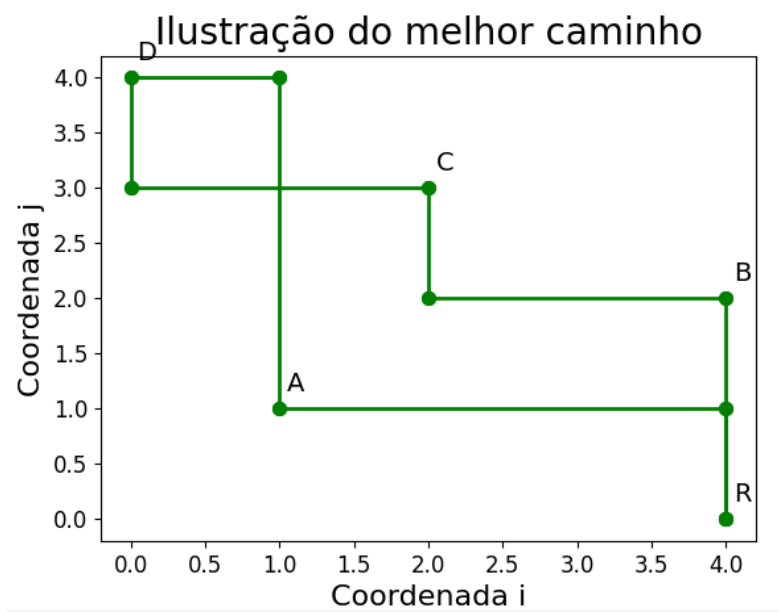
5.2. Resultados

Os resultados obtidos nos mostram quanto tempo foi necessário para calcular todos os caminhos possíveis e o custo de cada um deles à medida que o número de pontos de entrega foi aumentando.

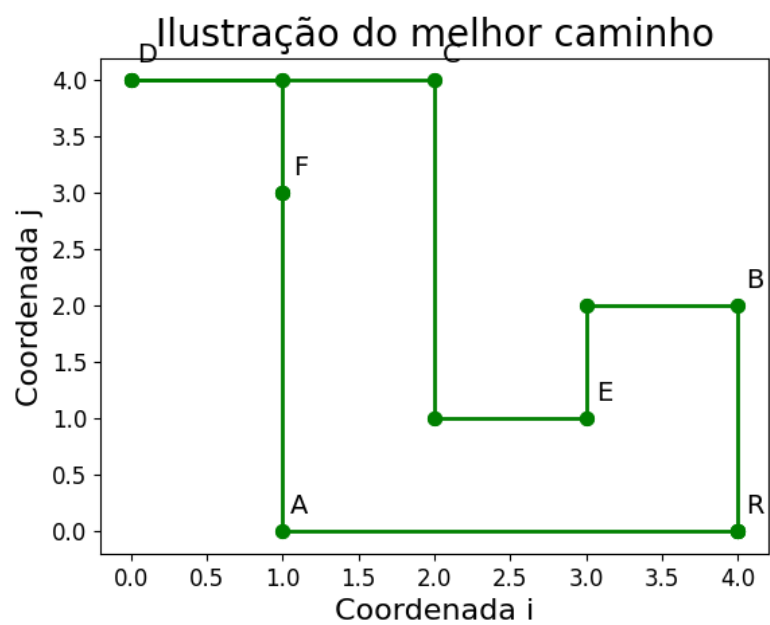
Ordem da matriz	Pontos de entrega	Custo do caminho	Tempo (segundos)
5x5	4	16 dronômetros	0.008 segundos
5x5	6	18 dronômetros	0.01 segundos
6x6	8	24 dronômetros	0.5 segundos
6x6	10	28 dronômetros	53.24 segundos

Veja abaixo as imagens dos menores caminhos calculados em cada experimento:

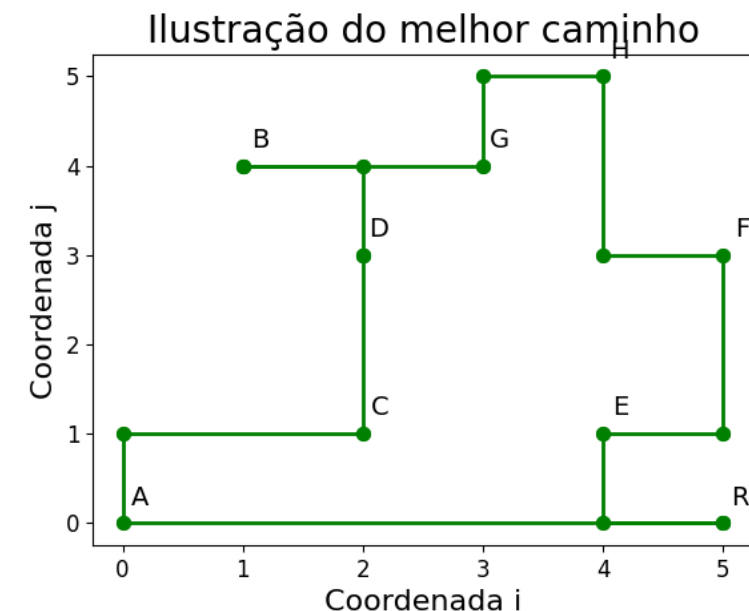
Pontos de entrega: 4; Rota: 'R A D C B R'; Custo: 16



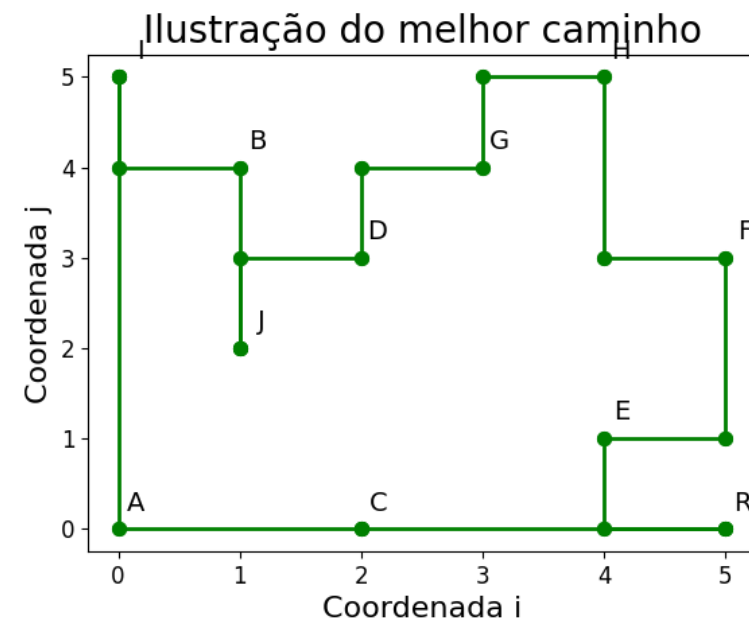
Pontos de entrega: 6; Rota: 'R A F D C E B R'; Custo: 18



Pontos de entrega: 8; Rota: 'R A C D B G H F E R'; Custo: 24



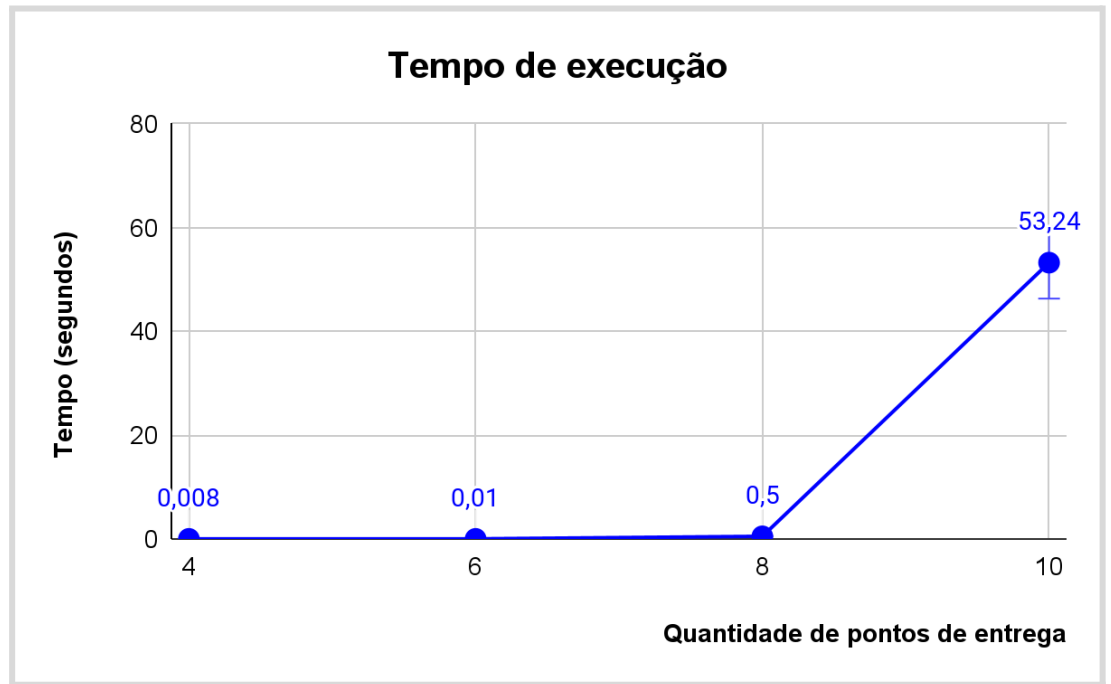
Pontos de entrega: 10; Rota: 'R C A I B J D G H F E R'; Custo: 28



5.3. Análise de resultados

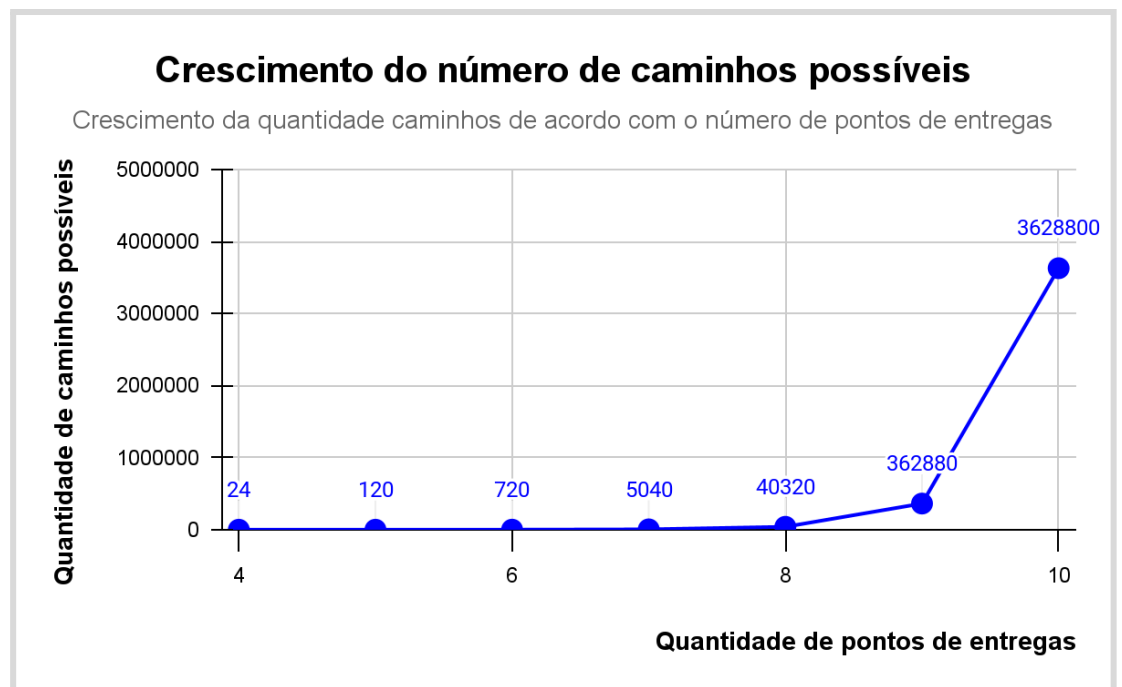
Com base nos resultados dos experimentos feitos, podemos observar que o tempo que o algoritmo leva para gerar as rotas e calcular seus custos cresce cada vez mais rápido conforme a elevação da quantidade de pontos.

Veja abaixo um gráfico demonstrando o crescimento do tempo de execução de acordo com o aumento do número de pontos de entrega:



6. Resultados

Como o número de caminhos possíveis cresce fatorialmente de acordo com o número de pontos de entrega (ver imagem abaixo), a complexidade de tempo do algoritmo para gerar as permutações e encontrar o melhor caminho será enorme a partir de uma certa quantidade de pontos - com 15 pontos existem mais de **1 trilhão** de possíveis rotas, por exemplo -, fazendo com que o algoritmo seja muito lento ou até mesmo inviável a partir de um número de pontos relativamente grande.



7. Conclusão

O objetivo deste trabalho era desenvolver um algoritmo capaz de calcular a rota de menor custo, de forma que otimize os trajetos dos drones para que eles realizem suas entregas mais eficientemente. Conseguimos implementar uma solução que gera todos os caminhos possíveis e encontra aquele de menor distância total. Entretanto, como nosso programa é $O(n!)$, ele mostrou-se eficiente apenas para um número pequeno de pontos de entrega, visto que à medida que o número de pontos aumenta, a quantidade de rotas possíveis cresce fatorialmente.

Portanto, a partir do conhecimento acerca dos problemas NP-Difíceis e dos experimentos realizados neste trabalho, concluimos que o método de força-bruta não é o mais adequado para esse tipo de problema. Isso se dá porque, apesar desse método nos dar o caminho de menor custo, ele precisa gerar todas as permutações possíveis, que, como citado anteriormente, cresce fatorialmente, fazendo com que o algoritmo seja muito lento ou até mesmo inviável para uma quantidade de pontos de entrega muito grande.

Referências Bibliográficas

- [1] CORMEN, Thomas. **Algoritmos - Teoria e Prática**. 3.ed. Barueri: Grupo GEN, 2012.
- [2] FEOFILOFF, Paulo. Análise de Algoritmos. **IME-USP**. São Paulo, 2015. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/. Acesso em 11 jan. 2023.
- [3] BRUNET, João. Análise Assintótica. **Joaoarthurbm**. Campina Grande, 2019. Disponível em: <https://joaoarthurbm.github.io/eda/posts/analise-assintotica/>. Acesso em 12 jan. 2023.
- [4] OLIVETE, Celso. Complexidade computacional - classes de problemas. FCT Unesp. Presidente Prudente, [2020?]. Disponível em: <http://www2.fct.unesp.br/docentes/dmec/olivete/tc/arquivos/Aula11.pdf>. Acesso em 13 jan. 2023.
- [5] DIFERENÇA entre NP Difícil e NP Completo. **Acervo Lima**. c2022. Disponível em: <https://acervolima.com/diferenca-entre-np-difcil-e-np-completo/>. Acesso em: 13 jan. 2023.

- [6] SILVA, J. L. C. et al. Uma Heurística aplicada ao problema do caixeiro viajante. In: **SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL**, 37., 2005, Granmado-RS. Anais... Granmado-RS: 2005. Disponível em: <http://www.repositorio.ufc.br/handle/riufc/13121>. Acesso em: 11 fev. 2023.
- [7] Montemanni R, Dell'Amico M. Solving the Parallel Drone Scheduling Traveling Salesman Problem via Constraint Programming. **Algorithms**, 2023; 16(1):40. Disponível em: <https://www.mdpi.com/1999-4893/16/1/40>. Acesso em: 13 fev. 2023.
- [8] DA CUNHA, Claudio Barbieri; DE OLIVEIRA BONASSER, Ulisses; ABRAHÃO, Fernando Teixeira Mendes. Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante. In: **XVI Congresso da Anpet**. 2002. Disponível em: [\(PDF\) Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante](#). Acesso em 13 fev. 2023.