Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

Fly Food & algoritmos

José Lucas Cabral Braga

Recife

Fevereiro de 2022

Dedico esse trabalho a todas as que me acompanharam e ajudaram diretamente e indiretamente e ao meu instrutor durante essa jornada.

Resumo

O projeto Fly food veio com a necessidade de encontrar um novo caminho para as entregas no nosso atual mundo em que temos uma grande demanda e o trabalho humano se valorizou, seu principal objetivo é alcançar uma forma onde que drones consigam fazer entregas de maneira totalmente autônoma, otimizando suas rotas e economizando bateria para completar seu percurso, contudo, se viu um desafio quando a maneira de encontrar a melhor rota se mostrou, foi usado o problema do caixeiro viajante para trabalhar por cima, visto que de inúmeras alternativas, uma deveria ser escolhida e a melhor possível e para isso foi usado o algoritmo de força bruta, contudo por sua baixa velocidade, também foi usado metaheurística (é um método para resolver de maneira genérica problemas de otimização, em problemas que não conhecemos um algoritmo eficiente) no algoritmo genético. Com as comparações finais, percebemos que o algoritmo de força bruta se torna lento com o aumento de pontos e o algoritmo genético se torna mais efetivo com maior número de pontos.

Palavras-chave: Algoritmo determinístico, algoritmo não-determinístico, algoritmo de força bruta, algoritmo genético, problema do caixeiro viajante, problemas de otimização, tempo polinomial, tempo exponencial.

1. Introdução

1.1 Apresentação e Motivação

Em nossa atual realidade, podemos ver que as entregas de delivery estão cada vez mais demoradas e com problemas com sua entrega, seja demorando bastante ou então chegando muitas vezes fria ou o entregador se perdendo por falta de conhecimento do local, além do alto custo para manter essas entregas. Então como proposta para suprir essa demanda e dar mais um passo afrente, uma nova ideia surgiu de criar um segmento de empresa de entregas com drones, na qual sejam programados para seguir uma rota e conseguir entregar com facilidade evitando o trânsito e reduzindo o custo dessa atividade, assim como facilitando para os usuários e os consumidores.

1.2 Formulação do problema

Em nossa atual realidade temos toda a visão de que o projeto Fly food irá trazer uma melhora na qualidade das entregas em questão de tempo e velocidade, contudo para isso acontecer, é preciso saber que terá complicações para ser implementada, ou seja seu problema.

De certa forma inicialmente temos que associar o mapa e as residências, a maneira que foi encontrando é transformar o mapa em uma matriz e suas residências em pontos, o estabelecimento comercial também foi considerado um ponto, onde deve ir até todas as residências que solicitaram os serviços dos drones e depois retornar ao ponto do estabelecimento.

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

Dessa forma temos um problema de otimização, onde devemos minimizar os custos e para isso é necessário que encontremos primeiro o melhor caminho, que é o caminho que representa a menor distância que um drone pode percorrer.

Para realizar o cálculo das distâncias, vamos calcular um ponto do outro assim como no plano cartesiano em que dado dois pontos A e B, com as coordenadas (x1,y1) e (x2,y2), respectivamente, veremos a sua distância entre pontos fazendo uso da seguinte função: $D = |x2-x1| + |y2-y1|$.

Por fim, esse problema se assemelha ao Problema do Caixeiro Viajante (PCV), sendo uma otimização combinatória, onde o número de rotas n é dado através de uma

equação baseada me permutações, sendo $[(n-1)!]/2$, sendo simétrico pois a distância das rotas é a mesma e com os experimentos durante esse trabalho, a quantidade de permutações que podem ser geradas dentro desse problema podem ir muito além do que se pode calcular em um tempo ótimo, se tornando um problema de NP-Completo.

1.3 Objetivos

Melhorar e revolucionar o método do delivery evitando trânsito e problemas como trocas de pedidos e entregadores perdidos e gastos desnecessários, será criar e implementar uma solução em drones aéreos para conseguirem trabalhar de maneira otimizada para evitar trânsito congestionado e acelerando a entrega pela melhor rota possível.

1.4 Organização do trabalho

Inicialmente veremos a localização principal da ida e da volta, assim de certa forma precisamos também ver os locais e também a capacidade da bateria do drone para assim conseguir calcular a melhor rota possível e nesse trabalho mostraremos, assim depois podemos se situar, primeiro fazendo toda o tráfego referencial, mostrando em que podemos se localizar a partir de um mapa de matrizes, assim como retificar o código de atuação do drone, colocando sobre a linguagem Python, demonstrar como vai ser as possibilidades dos experimentos e as rotas alcançadas e por fim ver se é viável conseguir fazer entregas com o drone. Durante a primeira seção teremos a introdução inteira do trabalho, mostrando suas partes como motivação e objetivos, após isso indo para sua segunda seção teremos os conceitos nos quais trabalhamos e as técnicas usadas, assim como gastos. Na terceira estarão trabalhos que possam complementar ou os demais que tiveram ideias semelhantes ou parecidas para integrar informações. Na quarta seção teremos toda a metodologia, assim como linguagem trabalhada, formas que foram feitas elaboradas os códigos baseando-se nas matrizes apresentadas e a sua atuação dentro do drone. Na quinta seção será visto como foi a natureza que o código foi feito, em quanto tempo ele foi capaz de atuar, por quanto tempo e o seu peso, assim como logo após o seu uso e a sua forma que agiu dentro do drone. Na sexta, todos os resultados foram vistos dentro dos experimentos e as suas modificações aparentes, de certa forma que consiga evoluir com o projeto. E na última e sétima seção teremos todos os resultados finalizados até agora, como tempo de duração com determinado código e seu desempenho.

2. Referencial Teórico

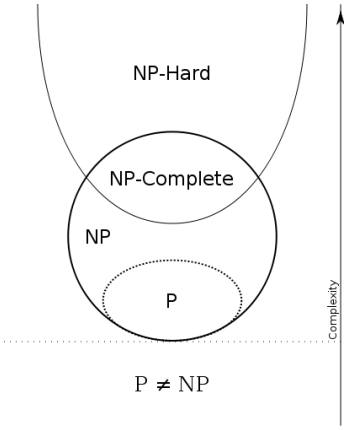
Nessa seção serão apresentadas as motivações de criação do Fly food, algumas noções de complexidade e sobre Big O e o pior cenário.

2.1 Fly food e suas motivações

"*Veículos não tripulados estão cada vez mais presentes no cotidiano das empresas e das pessoas, pois esse tipo de veículo está de forma crescente desempenhando atividades que anteriormente eram apenas executadas por seres humanos.*" [1] assim como visto, drones estão cada vez mais capacitados para fazerem trabalhos humanos, consequentemente durante o tempo e as modificações do trânsito se tornaram visíveis, fazendo com que todo um complexo de trabalho começasse a ter seus problemas, contudo: "*Com o avanço da tecnologia, novas formas de transporte vêm se tornando cada vez mais eficazes para os processos logísticos das empresas, com isso, elas vêm estudando uma forma de implementar os drones, de forma tornar suas entregas mais rápidas e seguras para seus clientes finais.*" [2] Com isso aí entra o projeto Flyfood uma forma autônoma de entrega com a livre movimentação dos drones pelo espaço aéreo, evitando problemas da sociedade atual. "*No entanto, para que tais serviços avançados para cidades inteligentes se tornem realidade, é necessário garantir a segurança nas entregas.*" [3] Após anos, várias modificações na nossa sociedade foram feitas, como o aumento de valores no trabalho humano, assim como a redução dos custos para produção dos drones.

2.2 Complexidade de problemas de classes

"*A complexidade de um problema computacional é o consumo de tempo do melhor algoritmo possível para o problema. É preciso lembrar que para muitos problemas o melhor algoritmo conhecido está longe de ser o melhor algoritmo possível. Podemos dizer, informalmente, que a complexidade de um problema é o tempo (como função do tamanho das instâncias) absolutamente indispensável para resolver o problema.*" [6] O problema de otimização de dados é justamente o tratamento que ele deve receber e o modo em qual seu algoritmo foi ajustado, sendo assim os seus modelos influenciam diretamente no resultado, levando em consideração o tempo e o consumo de dados. Portanto é visto que as classes de problemas podem ser diferentes que classificam os problemas de maneiras ao seu tempo de resposta e pela resolução de seu algoritmo e como um polinômio e assimilando com os dados e o tempo do programa. De modelos de classes temos a: P, NP e NP-Completo.



- **Algoritmo determinístico** é o que sempre produz o mesmo resultado dadas determinadas entradas de dados.
- **Algoritmo não determinístico** é aquele que pode produzir resultados diferentes mesmo com os mesmos dados

2.2.1 Classe P

A classe P é o conjunto de todos os problemas que podem ser resolvidos por algoritmos determinísticos em tempo polinomial.

2.2.2 Classe NP

A classe NP é o conjunto que consiste nos problemas que podem ser verificados em tempo polinomial.

2.2.3 Classe NP-Completo

A classe dos problemas NP, possuem uma derivação dos problemas NP-Completos, sendo sua subclasse de modo que todo problema NP se pode reduzir, com uma redução do tempo. Pode-se dizer que os problemas NP-Completos são problemas mais difíceis de solucionar do que os problemas NP. Nessa classe os problemas são intratáveis, ou seja, são normalmente resolvidos em tempo exponencial. Se um deles for resolvido em tempo polinomial, ele conseguirá uma resolução em tempo polinomial. O PCV é um exemplo que faz parte dessa classe.

2.3 BIG O

A notação Big O é de importância universal e origina-se do fato de descrever a eficiência do algoritmo escrito independente da linguagem de programação. Tendo diferentes tipos de complexidades do Big O, como $O(n)$, $O(2n)$, $O(n^2)$, $O(\log n)$, entre outras. Essa notação pode descrever a complexidade de uma seção pelo seu tempo de execução, sendo assim o Big O descreve o tempo em execução em seu pior caso julgando pela sua entrada de dados..

2.3.1 Pior caso

Pior caso é quando a entrada do algoritmo normalmente excede até o esperando do consumo de dados para sua saída. Por exemplo a memória necessária para funcionar um loop que gera infinitas matrizes e busque uma específica, enquanto outro só irá gerar uma matriz específica, ou até mesmo no problema em que atuamos, onde trabalhe com fatorial, porém existem números fatoriais que excedem os limites e acabe gerando problemas de dados, se tornando algoritmos intratáveis.

2.4 Problema do Caixeiro Viajante

O problema do Fly Food está relacionado com a geração de caminhos, assim como a teoria dos grafos, de certa forma está ligado a PCV de maneira que sua compreensão e seus caminhos sigam um padrão, contudo existindo diversos caminhos e apenas um que consiga tirar o máximo de vantagem por desempenho, contudo a forma que ele seja encontrado é a maneira mais bruta para ter certeza que o melhor caminho entre todos foi alcançado. Esse problema se torna de maneira um problema de otimização combinatória de forma que vemos a maneira específica para achar o melhor método para se guiar entre as heurísticas, durante o problema do Flyfood se torna um problema tratável por seu baixo número de locais, porém se aumentar logo ele se tornará um problema intratável e se tornando uma classe de problema chamada NP-Difícil.

2.5 Algoritmo de tempo polinomial e exponencial

Na computação a importância do uso do tempo na execução de algoritmos pode ser entendida como categórica de modo que uma função complexidade de tempo de um algoritmo expressa o tempo máximo preciso, isso é dentre as possíveis entradas que podem variar de acordo com o tamanho e dos casos dos problemas.

- **O algoritmo de tempo polinomial** é aquela cujo função de complexidade é $O(p(n))$ para alguma função polinomial $p(n)$, com n sendo o comprimento da entrada. [8]
- **O Algoritmo de Tempo Exponencial** é aquele cuja função complexidade de tempo não pode ser limitada por um polinômio. Por exemplo do caixeiro viajante que estamos usando. [8]

2.6 Algoritmo Genético

Os algoritmos genéticos ou genetic Algorithms (GAs) são inspirados no princípio darwinista da evolução das espécies e na genética, onde um ser sobrevive por suas adaptações ao meio em que vive através das possibilidades em qual se adapta melhor para seguir em frente, desenvolvido por John Holland (1975). Os AGs são algoritmos probabilísticos que fornecem um método de busca paralela e adaptativa baseado na teoria da evolução Darwiniana, dos mais aptos e na reprodução[10].

Os mesmos ensinamentos de Darwin são imitados dentro da construção desses códigos, que procuram uma forma melhor para a solução do problema. No caso do problema do Fly food, o caixeiro viajante como visto no algoritmo de força bruta após um número de pontos ele se torna extremamente difícil de se resolver, portanto o algoritmo genético se torna mais efetivo.

Os AGs possuem processos para serem realizados (representação, decodificação, avaliação, seleção, cruzamento, mutação, entre outros) e parâmetros para serem determinados (tamanho da população, taxa de reprodução, taxa de mutação e critério de parada). Seu primeiro conceito que podemos ver ser similar a teoria de Darwin é a representação de cromossomos, que são as soluções dentro do problema e podem ser diversos tipos [11].

A decodificação é a construção de uma solução para que o programa avalie-o. No fly food é usado uma matriz de pontos distintos e suas posições dentro desta matriz, a partir dos pontos são criadas permutações para serem utilizadas e comparadas [11].

Enquanto isso, a avaliação refere-se ao cálculo do fitness (aptidão) de cada indivíduo. Em que no problema do caixeiro viajante, se trata das somas dos caminhos de cada indivíduo. O método escolhido neste projeto foi torneio, onde se sorteia dois indivíduos ao acaso e comparam-se suas aptidões e o mais apto desses dois é selecionado. Este procedimento vai sendo repetido com todos os indivíduos, sendo esse procedimento muito mais simples e rápido [11].

Seguindo com o operador de cruzamento (Crossover) é considerado uma característica fundamental dos GAs que logo acontece após a seleção dos melhores indivíduos, nesse momento é onde a genética se encontra pelo menos no assim dito, onde no torneio aqueles que passaram da seleção, ambos vão se agregar e ter um filho e esse filho vai ser diferente dos dois, mas vai portar suas características. A maneira mais fácil de encontrar um ponto de corte aleatório é (one-point crossover) é dividir os progenitores em duas partes e a geração dos filhos acontece juntando as partes divididas dos pais [11].

Os cromossomos criados a partir da operação são então submetidos a uma possível mutação, podendo ou não ocorrer dependendo da sua taxa e probabilidade, essa mutação tem a função de aumentar a diversidade daquela população [11].

Outro operador que podemos ver também é o de elitismo, onde de certa forma ele pode quebrar a melhor solução, onde uma cópia ou um ser é passado para a próxima geração diretamente, tendo um certo privilégio durante a passagem do torneio, sendo ausentado da vista do fitness.

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

3. Trabalhos relacionados

Durante esta seção, teremos referenciais e trabalhos relacionados ou semelhantes a temática, ou que deram início a procuração do fly food como uma maneira de implementar suporte de entregas e logística.

3.1 Entrega por drones nos dias atuais

Podemos ver que desde antes, tínhamos o anseio por uma entrega com grande mobilidade e baixo custo, assim como usando drones não tripulados com maior facilidade para alcançar os seus objetivos, sendo mais seguro e também econômico, visando uma forma de efetivar o alcance e as formas que podem ser melhoradas, fazem comparações com programas de entrega por drones já existentes, mostrando que apenas um pequeno percentual mantém esse tipo de trabalho. E visto que os maiores problemas são com a burocracia brasileira para implementar os drones no nosso espaço aéreo se torna algo mais custoso pelas nossas taxas, contudo ainda tem alguns que defendem que poderão ser o futuro de nossas entregas, facilitando e acelerando processos. [2]

3.2 Otimização da entrega de encomendas por drones

Visto do ramo logístico, os drones poderiam ser uma solução para problemas de engarrafamentos e do modal rodoviário, é perceptível que eles seriam uma evolução de entregas pequenas, junto com eles e sua pequena taxa de bateria acabavam contribuindo com o meio ambiente por serem duráveis e serem elétricos, reduzindo a emissão de gases derivados da queima de combustível, contudo foi visto que sua maior desvantagem relacionado aos caminhões é a sua capacidade de não comportar grandes cargas. [4]

3.3 Desenvolvimento De Protótipo De Veículo Aéreo Não Tripulado De Baixo Custo E Integração Com Ferramenta Educacional Para Treinamento De Vôo

Os drones aéreos ganharam grande espaço durante a modernização, como em agricultura, construções civis, meio ambiente, mineração entre outros e com o aumento dessa tecnologia percebeu que deveriam diminuir os gastos para sua produção e ter maior facilidade para configurar ele para exercer sua função, com isso foi projetado interfaces para auxiliar aqueles que estavam moldando um novo drone quadri hélices e começar a treinar vôos em um simulador. [5]

Visto que inúmeros trabalhos estão no ramo de logística para conseguir entregar assim como modelagem e criação, porém é visto que em nossa atual realidade temos que aproveitar os drones e a sensibilização da situação para baixar os custos das taxas aéreas Brasileiras e evitar o trânsito caótico das grandes cidades, assim como auxiliar o meio ambiente com a redução de Co^2, fazendo entregas da melhor forma possível e com mais eficiência sem ter problemas com a manutenção dos drones.

3.4 EXPERIMENTOS COMPUTACIONAIS COM HEURÍSTICAS DE MELHORIAS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

"O Problema do Caixeiro Viajante – PCV (do inglês Traveling Salesman Problem - TSP) é um dos problemas mais estudados em otimização combinatória (Laporte, 1992). Apesar da sua definição singela, o PCV representa, até hoje, um dos desafios da Pesquisa Operacional" [7].Usado no fly food atualmente para representar o grande número de caminhos que podem ser tomados, contudo apenas um é o melhor e menor caminho que pode solucionar esse problema.

4. Metodologia

Durante essa sessão, teremos os pseudocódigos desenvolvidos como base para implementação da solução do Fly food sendo por base do problema do caixeiro viajante assim como a formação e implementação, juntamente com ele a solução desenvolvida em Python 3

4.1 Algoritmo de força bruta

A primeira solução foi desenvolvida visando o melhor resultado, porém usando o método do algoritmo de força bruta, sendo ele um algoritmo exaustivo já que ele vai tentar coletar todas as informações, ou seja permutar várias rotas diferentes e dentre elas escolher a melhor rota, ou seja a menor rota possível que transite entre todos os pontos e retorne para o ponto inicial.

```
    Entrada.split() #Onde deve conter as dimensões de uma matriz
    #Todas as listas que serão ocupadas durante a execução do
programa
```

O programa irá pegar as dimensões da matriz que seu modelo é linhas colunas que foi estabelecida anteriormente e assim trabalhar com ela, deixando você encaixar as coordenadas que serão programadas logo após, assim como o programa todo funciona em torno de listas de dados para conseguir demonstrar as matrizes, onde todas as listas devem ser determinadas para o seu funcionamento eficaz.

```
    #função recursiva para definir o melhor caminho
Função Recursiva melhor_caminho(combinações, coordenadas,
listaVazia)
```

```
     # a lista vazia irá armazenar a melhor solução
     menor_rota = 0
     i = 0
     j = 0
     temp = 0
     loop_for c in alcance(0,len(combinações)):
          while (coordenadas[i][0] != combinações[c][d]):
               i += 1
               #Garantir o funcionamento das coordenadas
          while (coordenadas[j][0] != combinações[c][d+1]):
               j += 1
          temp = temp + (abs(a-b) + abs(x-y))
          i = j = 0
     se(menor rota  == 0):
          lista Vazia = combinações[c]
          menor rota = temp
     senão(temp < menor rota ):
          menor rota = temp
          lista Vazia = combinações[c]
     temp = 0
imprima(Lista   Vazia,   combinações   e   quantidade   de
combinações, menor rota)
```

Após listar as listas, começamos a fazer funções dentro do programa, dentre elas estabelecemos uma função recursiva dentro de laços de repetição e condicionais, onde elas comparam as informações, para buscar a sua menor rota, ou otimizando sua velocidade de execução de sua rota, definindo então a melhor rota possível, contudo esse programa aceita uma matriz que inicialmente está vazia e dentro dessa função ordenamos os pontos que devem ser localizados dentro da matriz por seu posicionamento, por fim fazendo a comparação das combinações geradas para localizar suas melhores rotas.

```
#Função recursiva para gerar as permutações
Função recursiva permutações(lista, r = None):
     Montar tuplas para
     n = len(pool)
     r = n se r is None senão r
     se r > n:
          retorne
     tabela = list(alcance(n))
     rotação = list(alcance(n, n-r, -1))

     yield tuple(pool[i] for i in tabela [:r])
     while n:
          for i in reversed(alcance(r)):
               rotação [i] -= 1
               se rotação [i] == 0:
               tabela [i:] = tabela [i+1:] + tabela [i:i+1]
               rotação [i] = n - i
```

```
          else:
               j = rotação [i]
          tabela [i], tabela [-j] = tabela [-j], tabela [i]
               yield tuple(pool[i] for i in tabela [:r])

                    quebre
               else:
                    retorne
for c in alcance(0, inteiro(dimensoes_matriz[0])):
     matriz.adicione([])
     for d in alcance(0,inteiro(dimensoes_matriz[1])):
          matriz[c].adicione(0)

while True:
     pontos = entradaModeloPlanoCartesiano.split
# Ponto Eixo X e Eixo Y
     if pontos:
          nome = pontos[0].upper()
          locais.adicione(nome)
          x = inteiro(pontos[1])
          y = inteiro(pontos[2])
          matriz[x-1][y-1] = nome
     else:
          quebre
```

Dentro de uma nova função recursiva, podemos alinhar tanto as informações sobre o caixeiro viajante, quando as permutações, anteriormente nós determinamos o tamanho da matriz seja ela de Linhas X Colunas, onde ficará vazia e fará uma lista de listas, com o determinado tamanho de itens dentro para serem modificados, sendo substituídos por uma entrada onde determine Ponto determinado por uma letra, linha e coluna, sendo semelhante: D 1 0. Dessa forma seguindo o método, também precisamos determinar a quantidade de permutações que existiram, o que determina isso é a quantidade de pontos, já que ele deve sair do ponto inicial e retornar, assim terminando o ciclo, nessa função é usado para fazer gerações o yield para criar a lista de dados, para determinar a forma de geração, enquanto as permutações são feitas com o cálculo fatorial, assim como o ponto inicial não deve ser contado já que ele é o começo e também o fim do ciclo, sendo removido futuramente.

```
pontoDeInicio = entradaDeString.upper
removerDe.locais(pontoDeInicio)
```

Dessa forma o ponto inicial é removido do conjunto de forma que as matrizes fiquem mantendo ele como início e fim, podendo ir e retornar não fazendo parte da conta.

```
arranjo = list(permutacoes(locais, len(locais)))
for item in arranjo:
     melhor_rota.adicione(list(item))
for item in melhor_rota :
     item.adicione(start)
     item.insira(0,start)
```

```
for c in alcance(0,len(matriz)):
     for d in alcance(0,len(matriz[c])):
          for e in alcance(0,len(melhor_rota )):
               for f in alcance(0,len(melhor_rota [e])):
                    se len(rotas) == 0:
                         se  matriz[c][d]  ==  melhor_rota
[e][f]:
                    rotas.adicione([matriz[c][d],c+1,d+1])
               else:
                    se (matriz[c][d] == melhor rota [e][f])
and ([matriz[c][d],c+1,d+1] not in rotas):
                    rotas.adicione([matriz[c][d],c+1,d+1])
#Por fim, você chama as funções recursivas
melhor caminho(melhor rota , rotas, comp)
```

Por fim, todas as funções são chamadas para conseguir ter êxito, fazendo um loop fazer ele rodar por todas as combinações geradas e também pelas sequências dadas pelas permutações, sendo observadas e depois calculadas de certa forma procurando a melhor rota dentre todas as rotas geradas, exibindo as melhores rotas.

4.2 Algoritmo genético

Agora refazendo de maneira que o algoritmo genético seja empregado para se testar o que deve ser alterado, de certa forma se adaptando e resultando em vários pontos e métodos novos para serem agregados. Os parâmetros usados no algoritmo foram: tamanho da população = 100, elitismo = 2, taxa de mutação = 1% e número de gerações = 10. Todos os parâmetros podem ser alterados para mudança de resultados.

```
Classe fitness
     def inicio (self, rota)
          self.rota = rota
          self.distância = 0
          self.fitness = 0.0
     def distância das rotas
          se distancia = 0
          distância das rotas = 0
               Dentro do loop (0, até len(self.rota))
               cidade Inicial = self.rota[i]
               Próxima Cidade = ?
          se i + 1 < len(self.rota)
               Próxima cidade = self.rota[i + 1]
          senão
               Próxima cidade = self.rota[0]
               d rota += Cidade ini distância(Prox.cidade)
               retorne self distância
```

```
def rota fitness(self)
     se self fitness = 0
          self.fitness     =     1     /     math.cos
(self.rotadistancia())
     retorne self fitness
```

Podemos assim calcular o fitness e como no código anterior devemos calcular ambas as distâncias para podermos usá-las para calcular a melhor rota, se iniciando com uma classe para tentar manter o código organizado e manter loops de verificação para as rotas caso aumentem durante sua execução, dentro da função recursiva fitness, temos um porém usando math.cos, porém também pode-se ser usado float diretamente, mas o recomendo é usar cos.

```
Classe cidade
     def inicio (self, nome, x, y)
          self nome = nome
          self x = x
          self y = y
     def distancia(self, cidade)
          distância  x = distância x - cidade
          distancia  y = distancia y - cidade
          distância = x + y
          Retorne distância
     def retorno
          Retorne a função com a resposta do ponto
```

A classe cidade, ela é para ser composta assim como o primeiro código, mas também funciona usando pontos que são gerados pela entrada a partir dos pontos cartesianos dados por coordenadas, onde é armazenado.

```
     def criar Rota(Lista De Cidades)
          rota  =  sample(Lista  De  Cidades,  len(Lista  De
Cidades))
          Retorne as rotas
```

Dessa forma, nós buscamos criar as primeiras rotas do programa, usando o random.sample que retorna o comprimento específico dos itens da sequência trazendo amostragem aleatória sem reposição.

```
     def popu Inicial(tam popu inicial, lista de cidades)
          populacao = []
          Em um loop de 0 até o tamanho da popu inicial
          adicionar a popu na lista das cidades
          Retornar popu
```

```
def ordenar rotas(populacao)
     fitness resultados = {}
Dentro de um loop de 0 até pessoas dentro de popu no fitness
Rotas ordenadas dentro do dicionário com key=operator.itemgetter
```

Dentre essas duas funções recursivas, uma irá gerar a população inicial e adicionar a população dentro da lista da cidade, enquanto a outra irá organizar o fitness das cidades dentro de um dicionário.

```
def média distância rotas(rotas ordenadas)
     soma = 0
     loop de 0 até len(rotas ordenadas)
        soma += Fitness(rotas ordenadas[i]).rota distância()
return soma/len(rotas ordenadas)
```

Nesta função recursiva, buscamos ver o tamanho da rota e calcular sua média das distâncias em cada geração.

```
def torneio(popu ordenada, tam elitismo)
     resultado da seleção = []
     a = popu ordenada[random.randint(0, len(popu ordenada) - 1)]
     b = popu ordenada[random.randint(0, len(popu ordenada) - 1)]
Loop dentro de 0 indo até tam elitismo
     resultado da selecao.adicone(popu ordenada[i][0])
Loop de popu remova - elitismo
a = popu_ordenada[random.randint(0, len(popu_ordenada) - 1)][0]
b = popu ordenada[random.randint(0, len(popu ordenada) - 1)]
Caso A = B
b = popu ordenada[random.randint(0, len(popu ordenada)
Se a ⩾ b
     resultado da seleção.adicione(a)
senão
     resultado da selecao.adicione(b)
retorne seleção
```

Após começar o torneio, onde entram dois e dentro de uma competição usando o fitness e sua aptidão, ambos devem chegar lá, ser sorteados pela função randint, onde irá

tirar eles da lista da população e mexer com eles, contudo, existe o fator que podem ser iguais, ou seja, para evitar colocamos uma regra onde não os deixe ser ambos iguais A e B devem ser pessoas diferentes para continuar o progresso do algoritmo, por fim terminando e colocando eles dentro de uma nova lista de pessoas selecionadas para ir para a fase da procriação.

```
     def crossOver1(pai1, pai2)
          filho = []
          filhoP1 = []
          filhoP2 = []
          geneA = int(sorteie entre() * len(pai1))
          geneA = int(sorteie entre() * len(pai2))
          início = min(geneA, geneB)
          final = max(geneA, geneB)
     Loop do inicio para o final
          filhoP1.adicione (pai1[i])
     filhoP2 = [item for item in pai2 if item not in
filhoP1]
     filho = filhoP1 + filhoP2
     return filho

     def crossOver2(pai1, pai2)
          filho = []
          filhoP1 = []
          filhoP2 = []
          meio = len(pai1)//2
          Loop de 0 até o meio
               filhoP1.adicione(pai1[i])
           filhoP2 = [item for item in pai2 if item not in
filhoP1]
          filho = filhoP1 + filhoP2
     return filho
```

Após usar os crossover para juntar aqueles que depois do torneio se mantiveram na fila e entraram para a fila de cruzamento, de forma que conseguisse tentar manter diversidade, também foi usado o método one-point crossover onde existe a partição de metade da população e depois a junção novamente. Após juntar odos agora começaram a se unir para gerar novos filhos para continuarem para as próximas gerações, levando em conta o fitness para melhorá-lo.

```
     def crossOver População(cruzamento, tam elitismo)
          filhos = []
          tamanho = len(Cruzamento) - tam elitismo
```

```
          amostra        =        random.sample(Cruzamento,
len(Cruzamento))

     loop de 0 até tam elitismo
          filhos.adicione(Cruzamento[i])
     loop de 0 até tamanho
          filho       =       crossOver       1(amostra[i],
amostra[len(cruzamento) - i - 1])
          filhos.adicione(filhos)
          Retorne filhos)
```

Após continuar, eles vão agora se juntar com a população e continuar no crossOver dentro da cidade, até que retorne os filhos para a próxima geração.

```
     def mutar (indivíduo, taxa mutação)
          Loop em trocado até (len(indivíduo))
               se (sorteio() < taxa mutação)
                    trocar    com    =    int(sorteio()    *
len(indivíduo))
               Cidade1 = indivíduo[trocado]
               Cidade2 = indivíduo[trocar Com]
               indivíduo[trocado] = Cidade2
               individuo[trocarCom] = Cidade1
          return indivíduo
```

Dentro de um loop comece a procurar se a mutação mesmo por baixa vai ser tirada no sorteio em busca de saber se ela vai ser mutada, após isso ela vai mutar um gene de uma daquela rota.

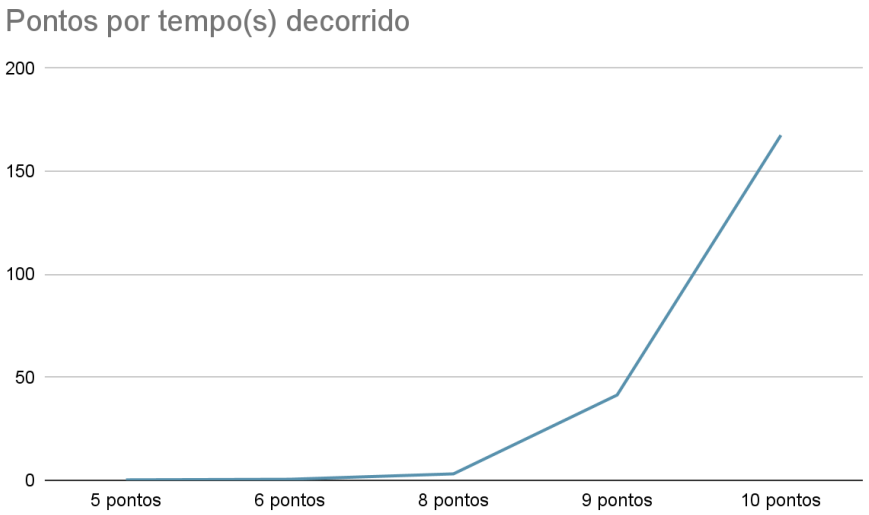
```
     def muta populacao(populacao, taxa mutação)
          população mutante = []
          Loop de 0 até len(população))
          mutardInd = mutar(população[ind], taxa mutação)
          população mutante.append(mutardInd)
     return população mutante
```

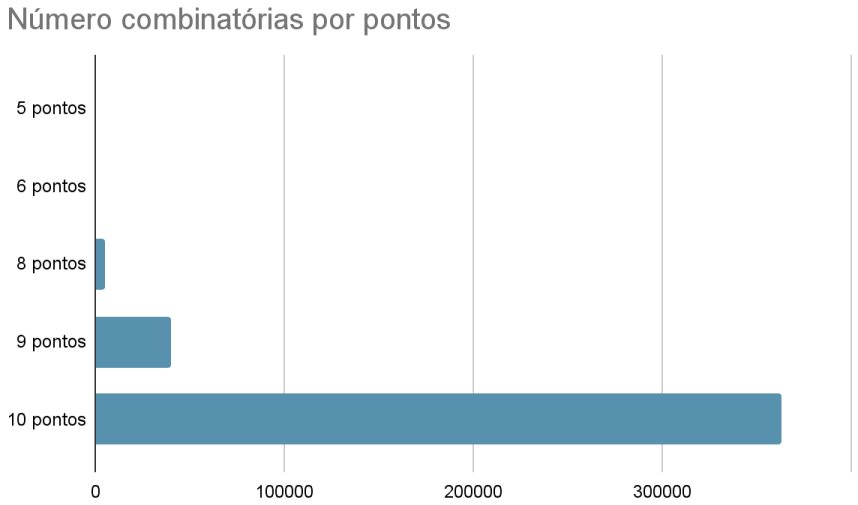
Descobrindo até isso, podendo finalmente ter indivíduos mutantes, após isso, começa a juntar todas as funções, para por fim começar a trabalhar com todo o algoritmo, unindo o gene atual, subtraindo o elitismo e unindo a taxa de mutação para saber se alguma pessoa irá realmente haver mutação, após unir todas as funções, agora finalmente é a hora de descobrir a melhor rota.

5. Experimentos

5.1 Algoritmo de força bruta

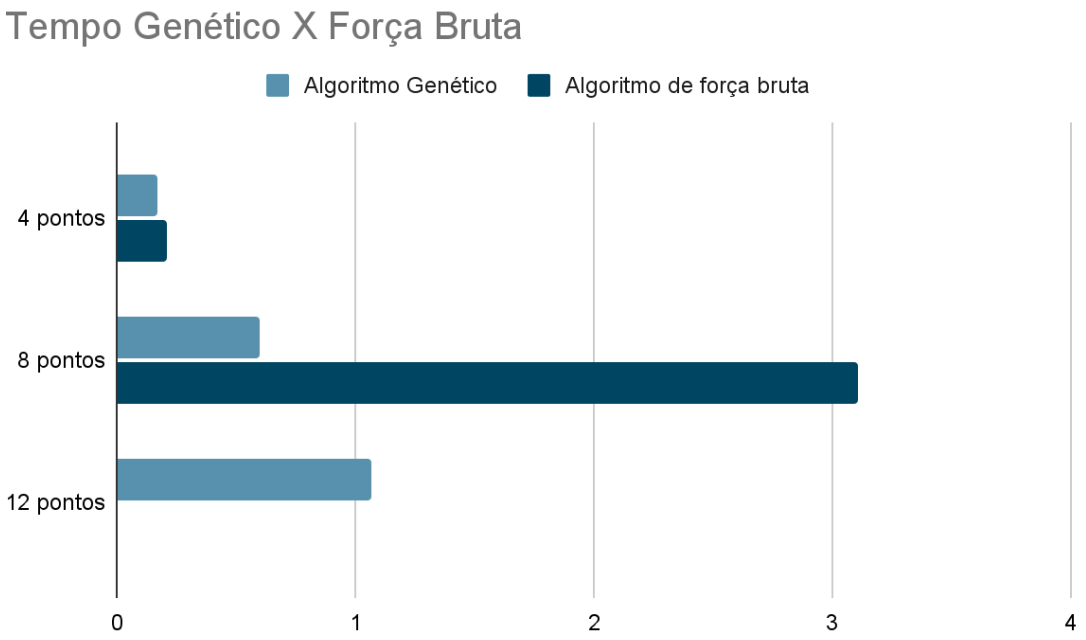
Durante os experimentos, foi capaz de perceber a mudança a cada ponto adicionado, seja de segundos até mais tempo, além da quantidade de combinações crescendo. De certa forma quando a entrada era uma matriz com cinco pontos, seu tempo era relativamente baixo conseguindo trabalhar da melhor forma possível mas ao decorrer dos pontos serem adicionados, o seu tempo de execução do algoritmo crescia junto com a quantidade de combinações.





5.2 Algoritmo Genético

Durante os experimentos, percebeu-se que o modo que o tempo ainda se mantém mesmo com o aumento dos pontos, gerando de certa forma ainda em tempo estável e mesmo com 30 pontos, ainda se mantém com tempo bom.



Podendo ver claramente que ao chegar nos 12 pontos não existe uma resposta em tempo correto para o algoritmo de força bruta, dessa forma fazendo que o algoritmo genético seja mais eficiente nos casos de mais pontos.

6. Resultados

6.1 Algoritmo de força bruta

Dentre o termo usado, assim como o algoritmo de força bruta para relacionar o problema do Fly food de entregas por drones, foi visto que também se torna usável quando vai até 5 pontos, passando disso já se torna inviável começando a apresentar inúmeros problemas, entre eles, fazendo o tempo ser bem mais do que deveria ser, começando a ter problemas em sua otimização, visto que isso tirando a velocidade de programa que deveria entregar isso a pronto momento.

Tamanho da matriz	Número de pontos	Tempo (Segundos)	Combinações geradas	Melhor percurso
4 x 5	4 pontos	0.21s	24	14
4 x 5	6 pontos	0.50s	720	22
5 x 6	8 pontos	3.11s	40.320	38
6 x 7	10 pontos	167.5s	362.880	50
6 x 7	12 pontos	Undefined	Undefined	Undefined

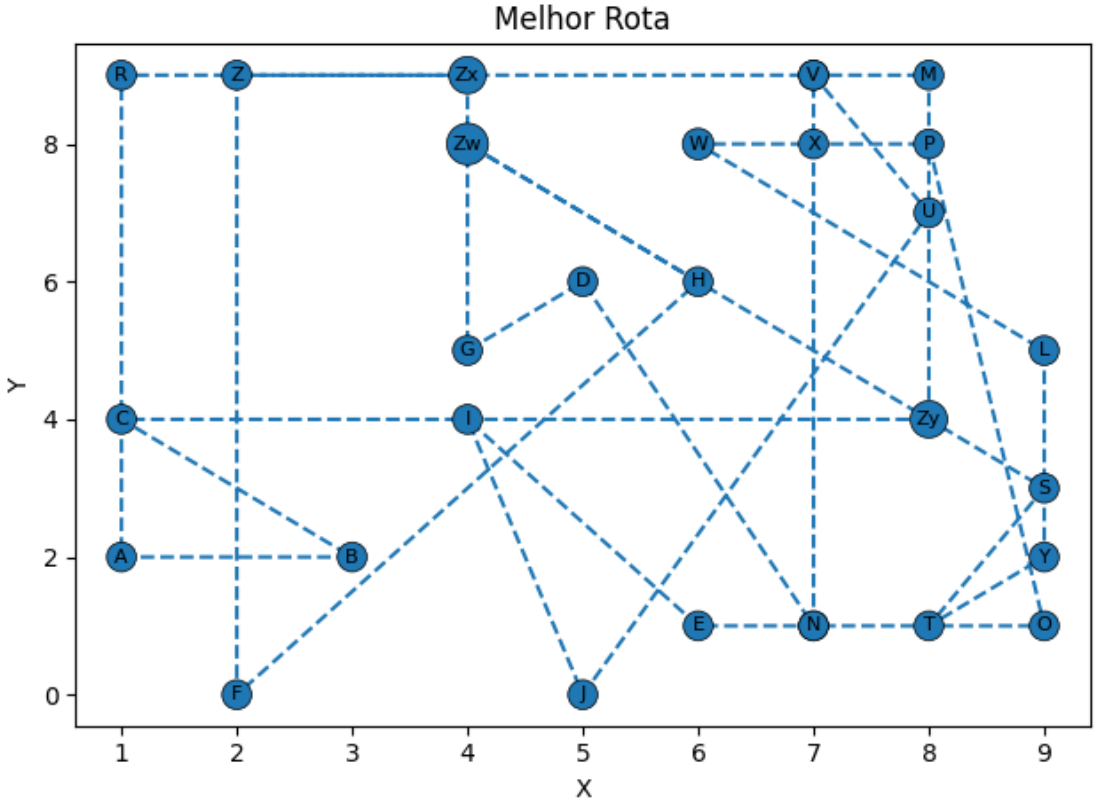
Dentro da tabela podemos ver que com o aumento dos pontos, o tempo vai começando a aumentar, junto com a quantidade de combinações, se aproximar de de 10 pontos, seu tempo já está muito acima do esperado para ter uma ótima solução e enquanto vai aumentando, seu número se torna inviável pelo tempo demorado, ou seja por otimização esse problema se torna um NP-Difícil.

6.2 Algoritmo Genético

O algoritmo genético é visivelmente mais rápido do que o algoritmo de força bruta, visto que ele mesmo com 30 pontos ele não demora tanto para processar a informação.

Número de pontos	Tempo (Segundos)	População	Combinações geradas
4 pontos	0.17s	100	24
6 pontos	0.51s	100	720
8 pontos	0.60s	100	2.520
10 pontos	0.84s	100	362.880
12 pontos	1.07s	100	479001600
29 pontos	2.33s	100	2.6525286e+32

Segue abaixo a melhor rota de 29 pontos encontrada.



(Zy), (C), (B), (A), (R), (Zx), (G), (D), (K), (X), (Q), (U), (J), (I), (E), (N), (O), (P), (W), (L), (Y), (T), (S), (Zw), (H), (F), (Z), (V), (M)

7. Conclusão

Com o decorrer do trabalho, foi perceptível que nossa empresa Fly food atende uma pequena área de normalmente um bairro com uma pequena quantidade de clientes, onde todos são atendidos efetivamente no menor tempo possível de entrega. Ao achar a lógica do caixeiro viajante temos que ver as inúmeras rotas permutadas, são várias possibilidades de caminhos e rotas a serem tomadas, essa perspectiva leva em consideração que dentre todas sempre vai haver um caminho. Dessa forma o método de brute force sendo um algoritmo exaustivo acaba chegando ao seu limite quando o número de pontos de entrega começa a crescer, enquanto quanto ao uso do algoritmo genético, ele se mantém mesmo com uma grande quantidade de rotas, foi usado 30 pontos de entrega e ele não teve um aumento de tempo tão significante, mostrando ser efetivo ter encontrado uma rota boa, porém nem sempre uma ótima solução, mas uma rota que consiga cumprir seu objetivo.

Referências Bibliográficas

1. MEDEIROS NETO, Manoel Pedro de. Veículos aéreos não tripulados e sistema de entrega: estudo, desenvolvimento e testes. 2016. 102f. Dissertação (Mestrado em Sistemas e Computação) - Centro de Ciências Exatas e da Terra, Universidade Federal do Rio Grande do Norte, Natal, 2016.
2. DAS GRAÇAS CARVALHO, Edmara et al. ENTREGA POR MEIO DE DRONES NOS DIAS ATUAIS, 2021.
3. DE OLIVEIRA, Fabíola MC; BITTENCOURT, Luiz F.; KAMIENSKI, Carlos A. Prevenção de Colisões em Serviços de Entregas por Drones em Cidades Inteligentes. In: Anais do V Workshop de Computação Urbana. SBC, 2021. p. 182-195.
4. BARBOSA, João Pedro Moutinho Alves. Otimização da entrega de encomendas por drones. 2020. Tese de Doutorado.
5. SHIBATA YAHAGUIBASHI, Henrique. Desenvolvimento De Protótipo De Veículo Aéreo Não Tripulado De Baixo Custo E Integração Com Ferramenta Educacional Para Treinamento De Voo. 2019.
6. FEOFILOFF, Paulo, atualizado em 2021, acessado em março de 2022 www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html
7. 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.
8. LOUREIRO, Antonio Alfredo Ferreira. Teoria de Complexidade. UFMG, 2008. acessado em abril de 2022. https://homepages.dcc.ufmg.br/~loureiro/alg/091/paa_Complexidade.pdf
9. Cabral Braga, José Lucas. Flyfood e Otimização. Github, 2022. Disponível em: https://github.com/zeth-l/Flyfood-e-Otimizacao . Acesso em abril de 2022.
10. CARVALHO, André Ponce de Leon F. de. Algoritmos Genéticos. Department of Computer Science. Disponível em: https://sites.icmc.usp.br/andre/research/genetic/ Acessado em maio de 2022.
11. PACHECO, Marco Aurélio Calvalcanti. Algoritmos Genéticos: Princípios E Aplicações. ICA: Laboratório de Inteligência Computacional Aplicada, Rio de Janeiro. Disponível em: http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf Acessado em: maio de 2022.