Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

FLYFOOD

José Carlos Eliodoro de Santana

Recife

12 de 2022

Resumo

Este trabalho tem como objetivo resolver o problema do trânsito caótico em grandes cidades, especialmente em relação às entregas de comida. O tráfego de veículos nas cidades é um obstáculo para as empresas de delivery, afetando a velocidade das entregas. Para enfrentar esse problema, este estudo propõe a utilização de drones para entregas, juntamente com algoritmos de roteamento para encontrar a rota mais curta e eficiente para o drone percorrer, levando em conta as restrições de voo dos drones e a menor distância entre os pontos da cidade.

Neste trabalho, foram desenvolvidos e comparados dois algoritmos de roteamento: o Brute Force e o Algoritmo Genético. Ambos algoritmos têm como objetivo encontrar a menor rota em um percurso. A complexidade dos algoritmos foi discutida em termos de tempo de processamento e a viabilidade de cada um para diferentes tamanhos de rotas.

Além disso, a utilização de drones para entregas traz diversas vantagens, como redução dos custos de transporte e agilidade nas entregas, já que os drones não dependem do trânsito de veículos. Portanto, a implementação de algoritmos de roteamento em sistemas de delivery, como o proposto neste estudo, pode trazer benefícios significativos para as empresas e clientes envolvidos no processo, tornando-o mais ágil, econômico e sustentável. O desenvolvimento contínuo dessas soluções pode acompanhar as necessidades em constante evolução do mercado de delivery.

1. Introdução

1.1 Apresentação e Motivação

Com o crescimento exacerbado das cidades por inúmeros fatores, observa-se que os sistemas de distribuição inseridos em tal ambiente, estão em maioria decadentes, o que fez com que os fornecedores de serviços tendessem a enfrentar diversos desafios no seu dia-a-dia, como incertezas relacionadas com a entrega de mercadorias, principalmente com o alto custo da mão-de-obra humana, o que os levou a explorar implementações do uso de meios alternativos para solucionar o problema.

Tendo em vista esses fatos a empresa do ramo alimentício Flyfood, desenvolveu uma maneira de realizar entregas com a utilização de drones, o que seria uma solução viável contanto que os drones possuíssem um algoritmo de roteamento eficiente no qual os mesmos pudessem operar utilizando a melhor rota possível, para evitar desperdícios de cargas da bateria, enfim reduzindo o tempo gasto para entregas e portanto aumentando a eficiência das entregas.

1.2 Formulação do problema

O problema pode ser formalizado matematicamente como um problema de otimização de rotas.

- **Entrada:** A matriz de entrada será a informada com os pontos de origem e os de entrega.
- **Ponto vazio:** Representado por “0” é um ponto no qual nada será constado nele.
- **Ponto de partida:** Representado por “R” é um ponto no qual o drone deve começar e terminar o trajeto.
- **Pontos de entrega:** Os pontos de entrega serão onde o drone irá realizar as entregas dos pedidos representado por qualquer elemento diferente de “0” e “R”.
- **Distância entre pontos:** Será a distância medida em dronômetro.
- **Dronômetro:** Será a unidade de medida utilizada para percorrer a menor distância

de um ponto a para um ponto b em uma matriz $M_{x,y,}$ sem percorrer pela diagonal. Tem-se a distância representado por $d(a, b)=|x_b – x_a|+|y_b –y_a|$.

Com todos os conceitos iniciais podemos formular a resolução do problema, inicialmente calculando a distância entre todos os pontos de uma rota, porém a função d(a, b)=|xb – xa|+|yb –ya|, soma apenas a distância entre um único ponto, porém, é necessário uma solução que some toda a rota. Frente a isso sendo temos a função f(d) =∑ d. como buscamos o menor caminho dentre esses para realizar todas as entregas, temos que calcular o valor mínimo que retorna por essa função, e temos a função argmin(f(d)).

1.3 Objetivos

Este trabalho tinha como objetivo principal desenvolver um algoritmo que pudesse calcular a menor distância entre os pontos de uma matriz, representando pontos em uma cidade, levando em consideração os limites impostos pelo movimento dos drones. O objetivo era encontrar uma forma de otimizar o transporte de mercadorias, reduzindo os custos envolvidos.

Além disso, serão propostas possíveis melhorias e aplicações futuras do algoritmo desenvolvido. Espera-se que este trabalho possa contribuir para a otimização do transporte de mercadorias, proporcionando benefícios econômicos e ambientais.

2. Referencial Teórico

Nesta seção, serão apresentadas as referências utilizadas para a elaboração do trabalho, a fim de proporcionar uma melhor compreensão dos temas abordados.

2.1 Complexidade de Algoritmo

O custo computacional é definido como a quantidade de recursos, tais como o tempo de processamento e espaço de armazenamento da máquina, o que faz com que um mesmo algoritmo tenha um desempenho diferente em máquinas com hardwares

diferentes.

2.2 Classes de problemas

As classes de problemas computacionais são usadas para classificar os problemas de acordo com o tempo e a quantidade de recursos necessários para resolvê-los em um computador. Algumas das classes de problemas mais comuns são:

Classe P (Problemas polinomiais): são aqueles que podem ser resolvidos em tempo polinomial em relação ao tamanho da entrada. Em outras palavras, o tempo necessário para resolver o problema cresce no máximo com uma potência do tamanho da entrada. Exemplos de problemas em P incluem a ordenação de uma lista, busca em uma tabela hash e cálculo de um produto escalar.

Classe NP (Problemas não determinísticos polinomiais): são aqueles para os quais existe um algoritmo que pode verificar uma solução proposta em tempo polinomial, mas não há um algoritmo conhecido que possa encontrar uma solução em tempo polinomial. Um exemplo de problema NP é o problema da coloração de grafos.

Classe NP-Completo (Problemas NP-completos): são aqueles que são pelo menos tão difíceis quanto qualquer problema em NP, mas não se sabe se são ou não problemas em P. Em outras palavras, se um algoritmo eficiente para resolver um problema NP-completo fosse encontrado, isso significaria que P = NP. Exemplos de problemas em NP-completo incluem o problema do caixeiro viajante e o problema da mochila.

NP-Difícil (Problemas NP-difíceis): são problemas que são pelo menos tão difíceis quanto os problemas NP, mas podem não estar em NP. Em outras palavras, um problema NP-difícil pode ser mais difícil do que qualquer problema em NP. Exemplos de problemas NP-difíceis incluem o problema do caminho hamiltoniano e o problema do conjunto dominante mínimo.

Essas classes de problemas são usadas para entender a complexidade dos problemas computacionais e ajudar na classificação e na criação de algoritmos para resolvê-los de maneira eficiente.

2.3 O Problema do Caixeiro Viajante

O problema do caixeiro viajante (PCV) pode se referir a um caixeiro-viajante que pretende viajar por um conjunto de cidades por vez. Apenas uma vez e, finalmente, retorne à cidade inicial, percorrendo a distância mais curta. Este é um dos problemas de otimização combinatória mais estudados e possui uma ampla gama de aplicações no mundo real. Foi expressa pela primeira vez matematicamente por Carl Menger entre 1931 a 1932.

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

O problema do caixeiro viajante é considerado NP-difícil, indicando que não há uma solução ótima conhecida para problemas com um grande número de cidades e o tempo de computação aumenta exponencialmente com o tamanho do problema. Isso se deve ao grande número de possíveis combinações de rotas que precisam ser avaliadas para encontrar a melhor solução. Algoritmos heurísticos, como algoritmos genéticos e de busca tabu, são utilizados para encontrar soluções aproximadas.

2.4 Algoritmos Genéticos

O algoritmo genético foi criado em meados da década de 1960 por John Henry Holland, um cientista da computação americano. A ideia por trás do algoritmo genético foi inspirada pelas teorias da evolução e seleção natural propostas por Charles Darwin e pela genética mendeliana proposta por Gregor Mendel. Usando um método de busca que imitasse o processo de seleção natural, Holland desenvolveu um algoritmo que usava uma população de soluções potenciais e um processo de seleção, reprodução e mutação para gerar novas soluções ao longo do tempo, no qual indivíduos bem adaptados ao ambiente eram selecionados para sobreviver e se reproduzir, transmitindo suas

características genéticas para a próxima geração.

Um algoritmo genético é uma técnica de otimização inspirada na evolução biológica que simula o processo de seleção natural para encontrar soluções para problemas complexos. Ele é um método de busca heurística que usa uma população de indivíduos (soluções potenciais) e uma função de avaliação para determinar quais indivíduos são mais adaptados ao problema. O qual começa criando uma população inicial de indivíduos aleatórios, cada indivíduo é representado por um conjunto de cromossomos, que são as unidades básicas de informação genética. Em seguida, o algoritmo avalia cada indivíduo de acordo com uma função de aptidão, que mede o quão bem ele resolve o problema em questão. Os indivíduos mais aptos são selecionados para a próxima geração, enquanto os menos aptos são descartados.

Os indivíduos selecionados são então submetidos a operadores genéticos, como a reprodução (cruzamento) e a mutação, para criar novos indivíduos. Na reprodução, os cromossomos de dois indivíduos selecionados são combinados para criar um novo indivíduo, na mutação, um ou mais cromossomos de um indivíduo são modificados aleatoriamente. Esses operadores genéticos são utilizados para criar novas soluções a partir da população atual, melhorando assim a população de indivíduos.

O processo de seleção, reprodução e mutação é repetido por várias gerações até que a solução ideal seja encontrada ou até que um limite de tempo ou número de gerações seja alcançado. O algoritmo genético pode ser aplicado a uma ampla variedade de problemas, desde otimização de funções matemáticas até design de sistemas complexos, como redes neurais e sistemas de controle.

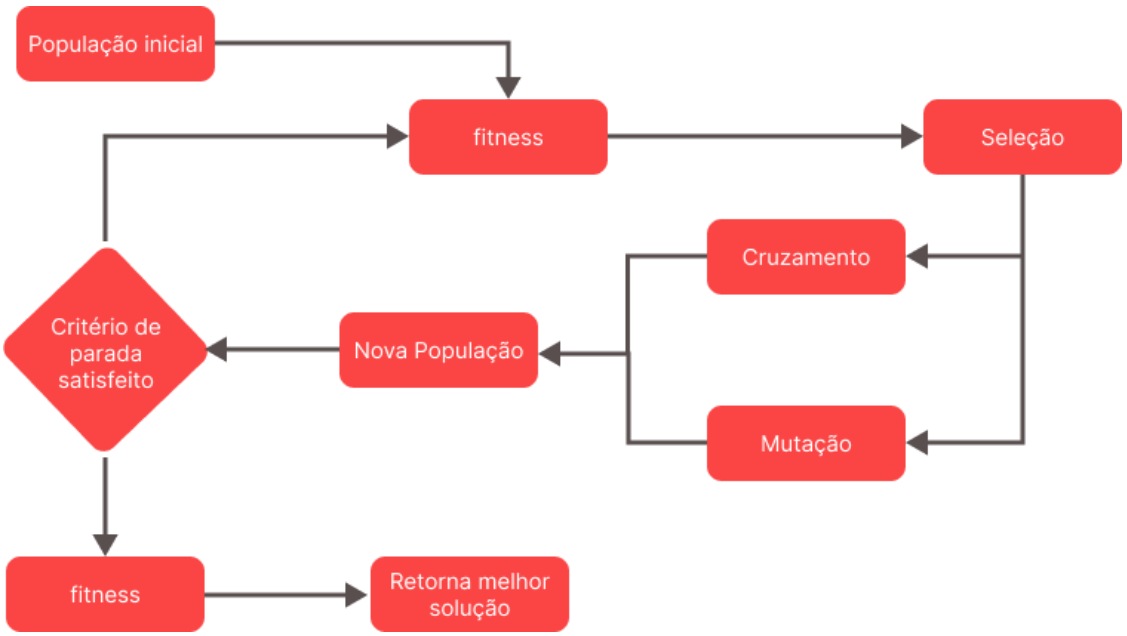

Figura 1. Representação gráfica do funcionamento de um Algoritmo Genético.

2.5 Operadores de crossover

Sabendo-se que busca sempre a formação de melhores indivíduos que sejam baseados em seus pais, existem vários tipos de operadores de crossover usados em algoritmos genéticos. Aqui estão alguns exemplos de tipos de crossover:

Partially Mapped Crossover (PMX): O PMX é um operador de crossover que usa uma abordagem baseada em mapeamento parcial para criar novas soluções. Nesse operador, dois pontos de corte são escolhidos aleatoriamente nas cadeias de genes dos pais selecionados. Em seguida, uma seção da primeira solução é mapeada para a mesma seção da segunda solução, com base em um esquema de mapeamento aleatório. O PMX preserva a ordem dos genes e evita a duplicação de genes na nova solução.

Cycle Crossover (CX): O CX é um operador de crossover que usa um ciclo para criar novas soluções. Nesse operador, um ciclo é criado a partir de um gene da primeira solução é o mesmo gene da segunda solução. Em seguida, os genes restantes são adicionados ao ciclo seguindo a mesma lógica. Depois que um ciclo é criado, os genes que estão fora do ciclo são trocados entre as soluções pais. O CX preserva a ordem dos genes e garante que todas as posições dos genes sejam trocadas.

Position-based Crossover (PPX): O PPX é um operador de crossover que usa uma abordagem baseada em posição para criar novas soluções. Nesse operador, as posições dos genes nas cadeias de genes dos pais selecionados são usadas para determinar quais genes serão incluídos na nova solução. Os genes são escolhidos aleatoriamente das posições correspondentes nas cadeias de genes dos pais. O PPX preserva a ordem dos genes e pode criar novas soluções com características intermediárias entre as soluções pais.

Cada um desses tipos de crossover tem suas próprias vantagens e desvantagens, dependendo do problema específico que está sendo resolvido. A escolha do tipo de crossover a ser usado depende do conhecimento do usuário sobre o problema e das características das soluções candidatas.

3. Trabalhos Relacionados

"Otimização de Rotas Utilizando Abordagens Heurísticas em Um Ambiente Georreferenciado". Este artigo discute a aplicação de algoritmos heurísticos na otimização de rotas de entrega de merenda em escolas, incluindo técnicas como Algoritmos Genéticos, Simulated Annealing e Algoritmos de Busca Tabu. Os autores apresentam uma análise comparativa dos métodos e mostram como cada um se aplica a diferentes cenários de entrega. Algumas desvantagens dos métodos heurísticos apresentados por eles são a não garantia de uma solução ótima, e a instabilidade de suas respostas, pois, podem resultar em soluções diferentes a cada execução. Já o método de força bruta possui a garantia de encontrar a solução ótima, pois, testa todas as possibilidades e garante encontrar a solução ótima.

"APLICAÇÃO DA METAHEURÍSTICA ALGORITMO GENÉTICO NA OTIMIZAÇÃO DAS ROTAS DE ENTREGAS DA DISTRIBUIÇÃO FÍSICA DE PRODUTOS NO MUNICÍPIO DE FORTALEZA"
Este artigo aborda a utilização de Algoritmos Genéticos para a otimização de rotas de entrega em uma cidade. Os autores apresentam um estudo de caso com a implementação do algoritmo em uma empresa de entrega e comparam os resultados com outros métodos heurísticos. Porém, apesar de conseguirem um bom resultado em comparação aos algoritmos existentes, ainda não possuem a garantia de uma resposta ótima, além da complexidade de implementação do método heurístico utilizado.

No artigo de Larranaga et al. (1999), é abordada a codificação de soluções para o problema do caixeiro viajante (TSP) por meio de diferentes métodos, como representação binária, matriz de adjacência, representação ordinal e representação por percurso. Além disso, são revisados diversos operadores de cruzamento e mutação que foram desenvolvidos especificamente para o TSP. O artigo também apresenta um algoritmo genético que utiliza o conceito de GENITOR, onde apenas um cromossomo é gerado por geração e inserido na população se a sua aptidão for melhor do que a do pior cromossomo da população atual.

O estudo também compara empiricamente um conjunto variado de operadores de cruzamento e mutação que empregam a codificação baseada em representação por percurso em experimentos com três instâncias diferentes do TSP, contendo 24, 47 e 48 cidades. Os operadores Genetic Edge Recombination Crossover (ER), Order Crossover 1 (OX1), Order Based Crossover (OX2), Position Based Crossover (POS), Partially-Mapped Crossover (PMX) e Cycle Crossover (CX) apresentaram os melhores resultados, sendo que o ER, OX1, OX2 e POS demonstraram maior superioridade em qualidade em relação aos demai.

O estudo de Samuel Wanberg, intitulado "Análise de operadores de cruzamento genético aplicados ao problema do Caixeiro Viajante", tem como objetivo avaliar o desempenho de diferentes operadores de cruzamento em algoritmos genéticos para solucionar o problema do Caixeiro Viajante (TSP). O autor compara diversos métodos de cruzamento, dentre os quais se destacam o Partially-Mapped Crossover (PMX) e o Cycle Crossover (CX). Estes operadores foram submetidos a experimentos utilizando diferentes instâncias do TSP e seus resultados foram comparados em termos de qualidade das soluções encontradas e tempo de execução.

Os resultados obtidos por Wanberg mostraram que o operador PMX apresentou um desempenho superior em relação ao operador CX em termos de qualidade das soluções encontradas. Por outro lado, o operador CX apresentou um desempenho melhor em termos de tempo de execução. Portanto, este estudo pode ser considerado um trabalho relacionado relevante para aqueles que desejam comparar o desempenho de diferentes operadores de cruzamento em algoritmos genéticos para solucionar o problema do TSP, em especial aqueles que desejam comparar os operadores de cruzamento.

4. Metodologia

Nesta seção, apresentamos uma maneira de resolver o problema do projeto FlyFood, utilizando um algoritmo de Brute Force(Força Bruta) para encontrar o menor caminho (argmin(s(p))). Explicamos de forma didática e clara todo o passo a passo da solução.

4.1 Solução ótima utilizando Brute Force

Linhas	**Algoritmo 1** Classe que representa um local de entrega
1	`classe House():`
2	`  metodo_construtor(x, y:Inteiro, name :Caractere):`
3	`  inicio`
4	`    parametro.x <- x`
5	`    parametro.y <- y`
6	`    parametro.name <- name`
7	`  fimmetodo`
8	`  metodo distant(house :Objeto):`
9	`  inicio`
10	`    retorne valor_absoluto(parametro.x - house.x) + valor_absoluto(parametro.y - house.y)`
11	`  fimmetodo`
12	`fimclasse`

Linhas	**Algoritmo 2** Classe de entrada de dados
1	`classe DateInput():`
2	`  metodo_construtor(url :Caractere)`
3	`  inicio`
4	`    parametro.url <- url`
5	`  fimmetodo`

```
6   metodo read_file(url:Caractere):
7   var
8     line :Vetor
9   inicio
10    abrir(parametro.url, 'r') chame obj:
11      line <- obj.readlines()
12    retorne line
13  fimmetodo
14
15  metodo create_router(text :Caractere):
16  var
17    x, y :Inteiro
18    position, line, ln :Vetor
19  inicio
20    x <- 0
21    para (line cada item text) faca
22      ln <- line.replace('\n', '').split(' ')
23      para (y de 0 ate sizeof(ln) passo 1) faca
24        se ln[y] != '0' entao
25          position.acrescentar(House(x, y, ln[y]))
26        fimse
27      fimpara
28      x <- x + 1
29    fimpara
30    return position
31  fimmetodo
32 fimclasse
```

Linhas	**Algoritmo 3** Função para permutar elementos de um array
1	função permut(lista)
2	var
3	lista_aux :Vetor

4	inicio
5	se comprimento(lista) <= 1
6	retorne [lista]
7	fimse
8	para i, atual em enumerar(lista)
9	elementos_restantes <- lista[:i] + lista[i+1:]
10	para p em permut(elementos_restantes)
11	lista_aux.extend([atual] + p)
12	fimpara
13	fimpara
14	retorne lista_aux
15	fimfuncao

Linhas	**Algoritmo 4** Função Intermediária que retorna menor rota
1	funcao (routes :Vetor, r: Objeto)
2	var
3	minim, x, y :inteiro
4	rt :caractere
5	route, distanci :Vetor
6	inicio
7	para (x de 0 ate sizeof(routes) passo 1) faca
8	route <- ['R']
9	distanci <- [r.distant(routes[x][0])]
10	para (y de 0 ate sizeof(routes[x]) passo 1) faca
11	route.acrescentar(routes[x][y].name)
12	se (y < sizeof(routes[x])-1) entao
13	distanci.acrescentar(
14	routes[x][y].distant(routes[x][y+1])
15	)
16	fimse
17	fimpara
18	route.acrescentar('R')
19	distanci.acrescentar(r.distant(routes[x][-1]))

```
20      se (minim = 0 or minim > sum(distanci)) entao
21        minim <- sum(distanci)
22        rt <- route
23      fimse
24    fimpara
25    retorne 'A menor rota é %s com exatos %d dronômetros'
26              %(rt, minim)
27  fimfuncao
```

```
Linhas  Algoritmo 5 Função para ler o arquivo de entrada
1   funcao menor_distancia_file(url)
2   var
3     datas, r :Objeto
4     routes :Vetor
5   inicio
6     datas = DateEntry(url)
7     routes = datas.create_router(datas.read_file())
8     r = list(filtro el: el.name esta em 'R', routes))[0]
9     routes = permut(list(filtro el: el.name esta em 'R',
10  routes)))
11    retorne menor(routes, r)
12  fimfuncao
```

4.2 Funcionamento do algoritmo de Brute Force

Para compreendermos melhor a solução que encontramos, primeiramente, necessitamos entender o funcionamento de um algoritmo de Brute Force. Um algoritmo de Força Bruta é um método para resolver um problema testando todas as possibilidades até encontrar a solução mais adequada. Isso é feito por meio da verificação sistemática de todas as combinações possíveis até que a solução ótima seja encontrada, a qual pode ser encontrada com a utilização de uma permutação dos elementos desejados.

Na função permut(lista) é onde está o cerne da solução, ela que executa essa permutação o parâmetro de entrada e um dado do tipo vetor como descrito na 1, na linha 5 até a linha 7 está o caso base da nossa recursão, em seguida temos o início de um laço de repetição na linha 8 do tamanho da lista original, que reservara à primeira posição para o primeiro elemento e assim por diante e criará uma outra lista que será passada adiante para também serem permutados. Visto isso criamos um outro laço de repetição na linha 11 o qual utilizará da mesma função para fazer uma chamada recursiva e gerar todas as permutações, essas permutações serão adicionadas na lista auxiliar e por fim passadas como resultado no fim da função. Essa é a parte com maior complexidade em nosso algoritmo, pois é ela quem faz o algoritmo de Brute Force. Criando uma árvore de permutações como vemos abaixo.

A complexidade desse algoritmo é de fatorial, ou seja, O(n!). Isso ocorre porque o número de permutações que serão geradas é igual a n! (fatorial de n), onde n é o comprimento da lista de entrada. Para cada elemento da lista, a função permut(lista) é chamada recursivamente, gerando uma nova lista com elementos restantes. Isso significa que para cada elemento da lista original, haverá n-1 elementos restantes, levando a uma quantidade de chamadas recursivas igual a (n-1) × (n-2) × (n-3) × ... × 2
× 1. O resultado é que a função permut(lista) é chamada n! vezes, o que resulta em uma complexidade de tempo O(n!).

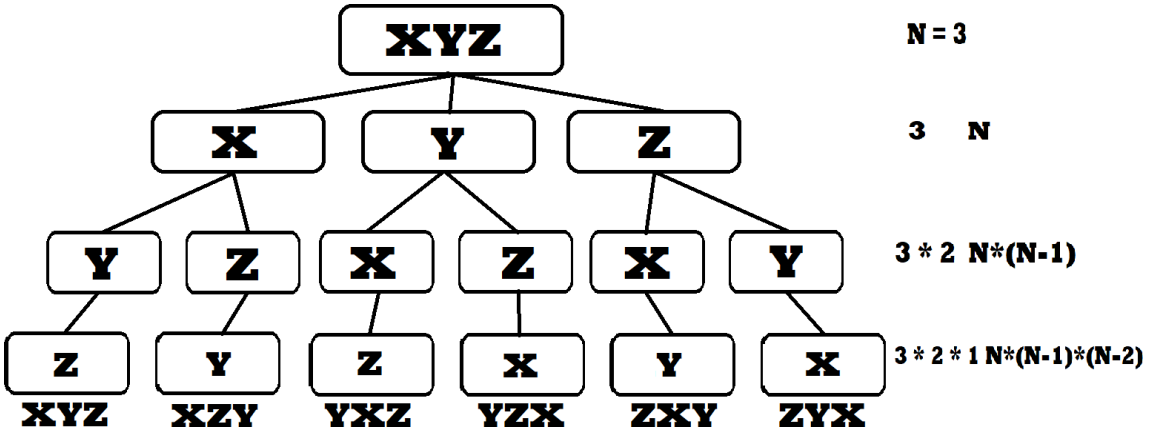

Figura 2. Representação da árvore de permutações.

A função menor_dist(url) é utilizada para calcular a menor distância possível entre uma série de pontos dados. A entrada desta função é uma URL que aponta para um arquivo de dados contendo informações sobre esses pontos. A ideia principal por trás dessa função é utilizar o algoritmo de permutação para gerar todas as combinações possíveis de trajetos entre os pontos e, em seguida, calcular a distância total de cada uma dessas rotas. A menor dessas distâncias é então retornada como a menor distância possível.

A classe DateInput é uma classe que lê dados a partir de um arquivo de texto, ela tem 2 métodos principais: read_file e create_router. O método read_file abre o arquivo na URL especificada e lê as linhas do arquivo, retornando-as como uma lista. O método create_router() percorre sobre cada linha na lista de strings e cria um objeto House para cada elemento não-nulo (não igual a 0) encontrado. As coordenadas x e y do objeto House são definidas como o índice da linha atual e da coluna atual, espectivamente. O valor do objeto House é definido como o valor encontrado na lista de strings. A função retorna uma lista de objetos House.

As duas últimas funções "menor_distancia_file(url)" e "menor(routes, r)" são funções auxiliares que ajudam a encontrar a menor distância possível entre as casas de uma determinada região. A primeira função "menor_distancia_file(url)" usa a classe "DateInput" para ler um arquivo de entrada e criar uma lista de objetos "House", filtrando apenas as casas que possuem uma rota. A segunda função "menor(routes, r)" é responsável por calcular a menor distância entre todas as possíveis rotas, retornando a rota mais curta com a distância total. Essas duas funções são usadas juntas para encontrar a solução final.

Embora o algoritmo retorne a solução ótima em 100% dos casos, o tempo gasto utilizando o método de brute force é dado em função de O(n!), tempo fatorial, devido a parte das permutações que é muito custosa para qualquer computador, uma maneira de conseguir uma solução menos custosa seria com a utilização de heurísticas, como algumas apresentadas na seção dos trabalhos relacionados, mas teria que aceitar o fato de que nem sempre se conseguiria obter o menor caminho de forma exata.

4.3 Solução Utilizando Algoritmo Genético

Linhas	**Algoritmo 1** Roleta
1	função roleta(lista: vetor de reais) -> inteiro
2	rand = aleatório() * somar_elementos(lista)
3	soma = 0
4	para i de 0 até tamanho_do_vetor(lista) - 1 faça
5	soma = soma + lista[i]
6	se soma >= rand então
7	retorne i
8	fim_se
9	fim_para
10	fim_função

Linhas	**Algoritmo 2** Gera a população inicial
1	função criar_população_inicial(arr: vetor de objetos, n_individual: inteiro) -> vetor de vetores
2	n_individual = abs(n_individual)
3	se (n_individual > fatorial(tamanho_do_vetor(arr))) então
4	n_individual = fatorial(tamanho_do_vetor(arr))
5	fim_se
6	permuts = [Nulo] * n_individual
7	count = 0
8	enquanto (count < n_individual) faça
9	permut_generacto = ordenar(arr, lambda x: aleatório())[:tamanho_do_vetor(arr)]
10	se não permut_generacto está em permuts então
11	permuts[count] = permut_generacto
12	count = count + 1
13	fim_se
14	fim_enquanto
15	retorne permuts
16	fim_função

Repositório no Github do trabalho: https://github.com/JoseEliodoro/Flyfood

Linhas	**Algoritmo 3** Calcula a distância de uma rota
1	função calcular_distancias(router: vetor de objetos, r: objeto) -> real
2	distancia = r.distant(router[0])
3	para index de 0 até tamanho_do_vetor(router) - 1 faça
4	se index < tamanho_do_vetor(router)-1 então
5	distancia = distancia + el.distant(router[index+1])
6	fim_se
7	fim_para
8	distancia = distancia + router[tamanho_do_vetor(router)-1].distant(r)
9	retorne distancia
10	fim_função

Linhas	**Algoritmo 4** Calcula a distância de todas as rotas
1	função calcular_todas_as_distancias(lista_cidades: vetor de objetos) -> vetor de reais
2	rotas = [Nulo] * tamanho_do_vetor(lista_cidades)
3	para i de 0 até tamanho_do_vetor(lista_cidades) - 1 faça
4	rotas[i] = calcular_distancias(lista_cidades[i][1:], lista_cidades[i][0])
5	fim_para
6	retorne rotas
7	fim_função

Linhas	**Algoritmo 5** escalona uma lista de valores
1	função escalonar_lista(lst: vetor de reais) -> vetor de reais
2	min_val = mínimo(lst)
3	max_val = máximo(lst)
4	lst_escalada = [0] * tamanho_do_vetor(lst)
5	para i de 0 até tamanho_do_vetor(lst) - 1 faça
6	lst_escalada[i] = (lst[i] - min_val + 1) / (max_val - min_val + 1)
7	fim_para

8	retorne lst_escalada
9	fim_função

Linhas	**Algoritmo 6** fitness(apitidão) dos individuos
1	função fitness(pop_list: vetor de reais) -> vetor de reais
2	fitness_list = [Nulo] * tamanho_do_vetor(pop_list)
3	para i de 0 até tamanho_do_vetor(pop_list) - 1 faça
4	fitness_list[i] = cos(pop_list[i] * radians(90))
5	fim_para
6	retorne fitness_list
7	fim_função

Linhas	**Algoritmo 7** Cruzamento por CX
1	função CX(parente1: vetor, parente2: vetor) -> vetor
2	tamanho = tamanho_do_vetor(parente1)
3	indice = aleatório(0, tamanho - 1)
4	filho = [Nulo] * tamanho
5	temp1, temp2 = parente1[indice:], parente2[indice:] + parente2[:indice]
6	para i de 0 até tamanho - 1 faça
7	se parente1[i] está em temp2 então
8	filho[i] = parente1[i]
9	senão
10	filho[i] = temp1[0]
11	temp1 = remover_primeiro(temp1)
12	fim_se
13	fim_para
14	retorne filho
15	fim_função

Linhas	**Algoritmo 8** Cruzamento por PMX
1	função PMX(pai1: vetor, pai2: vetor) -> vetor
2	ponto_de_corte1 = aleatório(0, tamanho_do_vetor(pai1) - 1)

3	ponto_de_corte2 = aleatório(0, tamanho_do_vetor(pai1) - 1)
4	se ponto_de_corte1 > ponto_de_corte2 então
5	ponto_de_corte1, ponto_de_corte2 = ponto_de_corte2, ponto_de_corte1
6	fim_se
7	filhos = copiar_vetor(pai1)
8	para i de ponto_de_corte1 até ponto_de_corte2 - 1 faça
9	se pai2[i] não está em filhos[ponto_de_corte1:ponto_de_corte2] então
10	j = encontrar_indice(pai1, pai2[i])
11	filhos[i], filhos[j] = filhos[j], filhos[i]
12	fim_se
13	fim_para
14	para i de ponto_de_corte1 até ponto_de_corte2 - 1 faça
15	se pai2[i] não está em filhos[ponto_de_corte1:ponto_de_corte2] então
16	j = encontrar_indice(pai1, pai2[i])
17	filhos[j], filhos[i] = filhos[i], filhos[j]
18	fim_se
19	fim_para
20	retorne filhos
21	fim_função

Linhas	**Algoritmo 9** Cruzamento de dois pais
1	função crossover_fathers(father1: lista de objetos, father2: lista de objetos, crossover_rate: float) -> tupla de lista de objetos :
2	se random() < crossover_rate então
3	retornar PMX(father1, father2), PMX(father2, father1)
4	senão
5	retornar father1, father2
6	fim_se
7	fim_função

Linhas	**Algoritmo 10** Cruzamento de todos os pais

1	função crossover(fathers_list: lista de lista de objetos, crossover_rate: float) -> lista de lista de objetos:
2	children_list = [None] * tamanho(fathers_list)
3	para i de 0 até tamanho(fathers_list) passo 2 faça
4	son1, son2 = crossover_fathers(fathers_list[i], fathers_list[i + 1], crossover_rate)
5	children_list[i], children_list[i + 1] = son1, son2
6	fim_para
7	retornar children_list
8	fim_função

Linhas	**Algoritmo 11** Mutação
1	Função mutation(pop_list: Lista de Listas de Objetos, mutation_rate: Float) -> Lista de Listas de Objetos:
2	Para cada indivíduo i na população pop_list:
3	Se um número aleatório entre 0 e 1 for menor ou igual ao mutation_rate:
4	Seleciona aleatoriamente duas posições a e b no indivíduo i
5	Troca os elementos nas posições a e b do indivíduo i
6	fim_se
7	fim_para
8	Retorna a população pop_list mutada
9	fim_função

Linhas	**Algoritmo 12** Escreve a sequência de cidades a ser visitada
1	Função Name_route(route: Lista de listas de objetos) -> Lista de strings
2	name = Lista vazia do tamanho da lista de roteiros
3	Para i de 0 até o tamanho da lista de rotas:
4	house = Lista vazia
5	Para z em rota[i]:
6	house.adicionar(z.nome)
7	fim_para
8	name[i] = união('-'.join(house))
9	fim_para
10	retornar nome
11	Fim da Função

Linhas	**Algoritmo 13** Imprime o melhor indivíduo da geração
1	função print_pop(pop_list: lista de listas de objetos, distance_list: lista de floats, generation: inteiro) -> vazio:
2	route_name = Name_route(pop_list)
3	melhor_indice = índice da menor distância em distance_list
4	imprime "Melhor solução da " + generation + "° geração é " + route_name[melhor_indice] + "-" + route_name[melhor_indice][0] + " e sua distância foi " + int(distance_list[melhor_indice]) + "."
5	fim_função

Linhas	**Algoritmo 14** Seleção de pais
1	Função select_fathers(pop_list, sel_func):
2	distancias_escaladas = scale_list(calc_all_distances(pop_list))
3	fitness_list = fitness(distancias_escaladas)
4	fathers_list = lista vazia do tamanho da pop_list
5	Para count variando de 0 até o tamanho de fathers_list, de 2 em 2:
6	father1 = sel_func(fitness_list)
7	father2 = sel_func(fitness_list[:father1]+fitness_list[father1+1:])

8	fathers_list[count] = pop_list[father1]
9	fathers_list[count + 1] = pop_list[father2]
10	fim_para
11	Retorne fathers_list
12	fim_função

Linhas	**Algoritmo 15** Evolução
1	função evolução(
2	dados: Lista[objetos], n_individuals: inteiro, n_generations: inteiro, crossover_rate: flutuante, mutation_rate: flutuante, sel_func: função
3	) -> Tupla[Lista[Lista[objetos]], Lista[flutuantes]]:
4	população_inicial = criar_população_inicial(dados, n_individuals)
5	para geracao em faixa(n_generations):
6	distancias = calcular_todas_distancias(população_inicial)
7	imprimir_população(população_inicial, distancias, geracao)
8	pais = selecionar_pais(população_inicial, sel_func)
9	filhos = crossover(pais, crossover_rate)
10	filhos_mutados = mutação(filhos, mutation_rate)
11	população_inicial = filhos_mutados[:]
12	fim_para
13	retornar população_inicial, distancias
14	fim_função

Linhas	**Algoritmo 16** Entrada de dados
1	Abrir arquivo contendo os pontos de entrada
2	Colocar os pontos de entrada como Objeto House()
3	Evolucao(dadosEntrada, n° individuos, n° gerações,
4	taxa_cruzamento, taxa_mutação, função de seleção)

4.4 Funcionamento da solução por Algoritmo Genético

Inicialmente precisamos definir uma forma de selecionar indivíduos com base em suas aptidões, para isso definimos o (algoritmo 1) que implementa o método de seleção por roleta. Agora, sabendo que algoritmos genéticos são baseados na teoria evolutiva de Darwin é necessário gerar uma população inicial (algoritmo 2) que é inicializada pegando pontos de entrega de uma rota inicial, colocando em uma lista de objetos House definida no (algoritmo 1, de Brute force), e posteriormente gerando uma ordem aleatória, sem que se repitam (como é visto no algoritmo 2, na linha 10), ou seja os genes serão os pontos de entrega representados pela classe House e o cromossomo é o caminho.

Agora que definimos uma população inicial temos que calcular a aptidão dos indivíduos, para calcular essa aptidão é necessário possuir a distância das rotas de forma escalonada, para calcular essa função temos a função calcular_todas_distancias (algoritmo 4), que faz a chamada de calcula_distancias (algoritmo 3) para calcular a distância de cada rota, depois escalonar (algoritmo 5) essas distâncias e por fim a função de aptidão (algoritmo 6), calcular a aptidão de cada os indivíduos utilizamos a fórmula $cos(d(r) * \pi/2)$ definida no (algoritmo 6, linha 4), d(r) distância escalonada num limite de 0 a 1, e $\pi/2$ faz os resultados permanecerem no primeiro quadrante da circunferência. Tendo todas as aptidões calculadas, a função de seleção de pais (algoritmo 14), pode fazer uma seleção por roleta (algoritmo 1) para ocorrer o crossover entre os melhores indivíduos, são escolhidos 2 candidatos aleatórios (linha 8 e 9), e com todos os melhores selecionados é realizado um crossover entre eles.

No crossover (algoritmo 10) é feita um cruzamento de todos os pais, gerando uma uma lista de filhos, para gerar esses filhos é chamada a função de crossover_fathers (algoritmo 9) no qual são gerados dois filhos, utilizando a técnica de cruzamento (definida na linha 3). Com os a novos cromossomos (rotas) gerados, podemos então passar eles pela função de mutação (algoritmo 11) com uma chance de acontecimentos definida, são escolhidos 2 genes diferentes aleatórios e é realizada a troca dos mesmos. por fim a função print_pop (algoritmo 13) imprime o melhor indivíduo e sua distância.

No algoritmo 7 e 8, são dois dos operadores de crossover descritos na seção 2.5 deste relatório, o algoritmo 7 está o operador CX começa selecionando um ponto de corte aleatório na cadeia de bits que representa o cromossomo dos pais. Em seguida, ele copia os bits de um pai para o filho correspondente até chegar ao ponto de corte. Em seguida, ele verifica o valor do bit no mesmo índice do outro pai e, se esse bit ainda não estiver presente na cadeia de bits do filho, ele copia este bit para o filho. Esse processo continua

até que todos os bits tenham sido copiados do primeiro pai para o filho, usando o segundo pai para preencher quaisquer lacunas na cadeia de bits do filho. Exemplo na figura 3. O operador PMX começa selecionando dois pontos de corte aleatórios na cadeia de bits que representa o cromossomo dos pais. Em seguida, ele copia a seção entre esses pontos de corte de um pai para o filho correspondente. Em seguida, ele mapeia os valores dos genes na seção do outro pai para o filho correspondente, verificando se o valor já não está presente na sessão do primeiro pai. Se o valor estiver presente, o operador PMX tenta encontrar um gene correspondente no segundo pai que possa ser mapeado para o filho. Esse processo continua até que todos os genes sejam mapeados para o filho. Exemplo na figura 4.

Figura 3. Exemplo de cruzamento com CX. Fonte: Elaborado pelo autor.

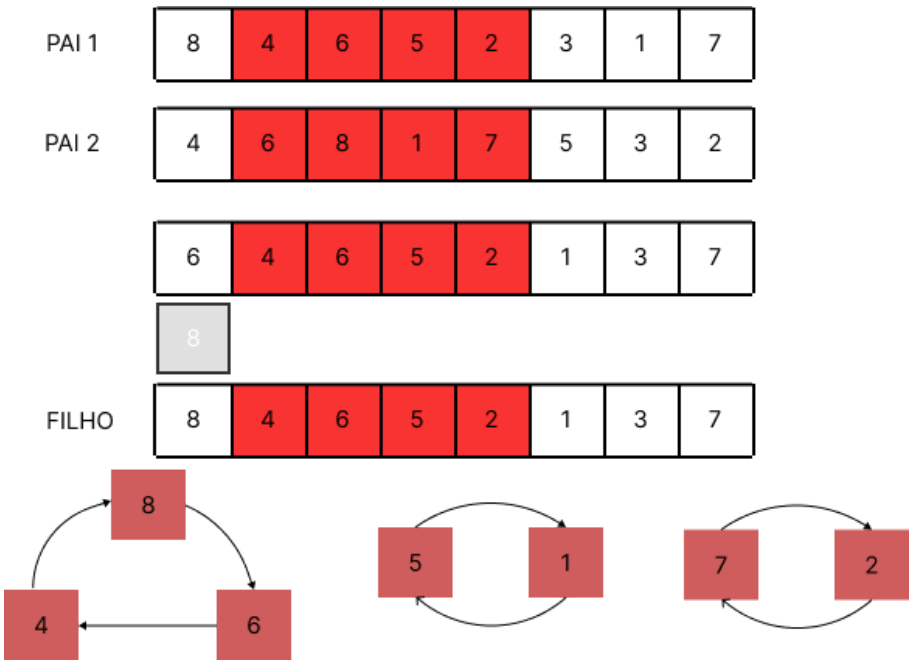

Figura 4. Exemplo de cruzamento com PMX. Fonte: Elaborado pelo autor.

Por fim do algoritmo genético, a função evolução (algoritmo 15) garante a evolução das gerações, nela são utilizadas todas as funções anteriormente citadas. Iniciando criando a matriz de distâncias, gerando a primeira população, calculando os fitness e realizando a seleção dos mesmos, e o primeiro indivíduo é selecionado como menor caminho. Com isso, agora pode-se fazer a evolução da população para uma certa quantidade de gerações (critério de parada), assim é gerada uma nova população e todo o processo é repetido, para cada geração é realizada um cálculo para pegar o melhor indivíduo e comparar com o da geração anterior, caso o da nova geração seja menor, ele será definido como menor caminho. Finalmente é exibido o menor caminho gerado por todas gerações, e ao sair do loop de gerações é exibida a rota do menor caminho.

E como fim de todo o algoritmo, no algoritmo 16, encontramos a leitura do arquivo que contém os pontos de entrada e as coordenadas (que deve seguir o padrão mostrado abaixo), e é iniciado o algoritmo genético chamando a função evolução (algoritmo 15) e passando como parâmetros a matriz de Entrada e a quantidade de gerações que serão utilizadas para tentar buscar a melhor solução.

5. Experimentos

A realização de experimentos é uma etapa fundamental para a validação de resultados e para o avanço da pesquisa em diversas áreas, incluindo a otimização combinatória. Nesse contexto, o TSPLIB é uma importante fonte de dados para a realização de experimentos em problemas de otimização do problema do caixeiro viajante (TSP).

No presente trabalho, utilizamos os dados dos arquivos burma14, berlin52 e d198 disponibilizados no TSPLIB para solucionar o problema de pesquisa do FlyFood utilizando algoritmo genético. O FlyFood é um problema de otimização combinatória que busca encontrar a rota mais eficiente para um caminhão de comida em um determinado conjunto de cidades

.

Utilizamos os dados dos arquivos burma14, berlin52 e d198 para criar instâncias do problema do FlyFood e avaliar a eficiência do algoritmo genético na sua solução. Realizamos experimentos variando os parâmetros do algoritmo, como o tamanho da população, a taxa de cruzamento e a taxa de mutação, e avaliamos o desempenho do algoritmo em termos do tempo de execução e da qualidade da solução encontrada.

Além da variação dos parâmetros do algoritmo genético, neste trabalho também foi realizada uma comparação entre dois tipos de operadores de crossover: o PMX e o CX. O crossover é um dos principais operadores do algoritmo genético, que é responsável por combinar informações genéticas de duas soluções candidatas para gerar novas soluções.

Por fim, serão utilizadas matrizes de 1 a 10 pontos, para que seja possível fazer uma comparação entre as soluções encontradas por algoritmo genético e Brute Force, para podermos entender até qual ponto um é superior ao outro, calculando o tempo de processamento de ambos os algoritmos, utilizando o Google Colab como campo de testes.

6. Resultados

Ao realizar experimentos, devemos ter em mente que ao utilizar algoritmos genéticos abdicamos a obtenção de uma solução ótima quando falamos de uma grande quantidade de pontos de entrega, então o que buscamos é uma solução satisfatória, nos experimentos utilizamos três instâncias diferentes: Burma14, Berlin52 e D198. Para cada uma dessas instâncias, foram utilizados 20 indivíduos na população inicial, 1000 gerações, taxa de cruzamento de 80% e taxa de mutação de 10%, os resultados obtidos foram pertinentes visto que, foi uma solução em tempo polinomial para um problema np-completo.

- Para o teste de burma14 a solução conhecida é de 3323, e o menor percurso obtido foi 4243, gastando aproximadamente 0.89 segundos do tempo de processamento.
- Para o teste de berlin52 a solução conhecida é de 7542, e o menor percurso obtido foi 14951, gastando aproximadamente 3.525 segundos do tempo de processamento.
- Para o teste de d198 a solução conhecida é de 15780, e o menor percurso obtido foi 22420, gastando aproximadamente 28.08 segundos do tempo de processamento.

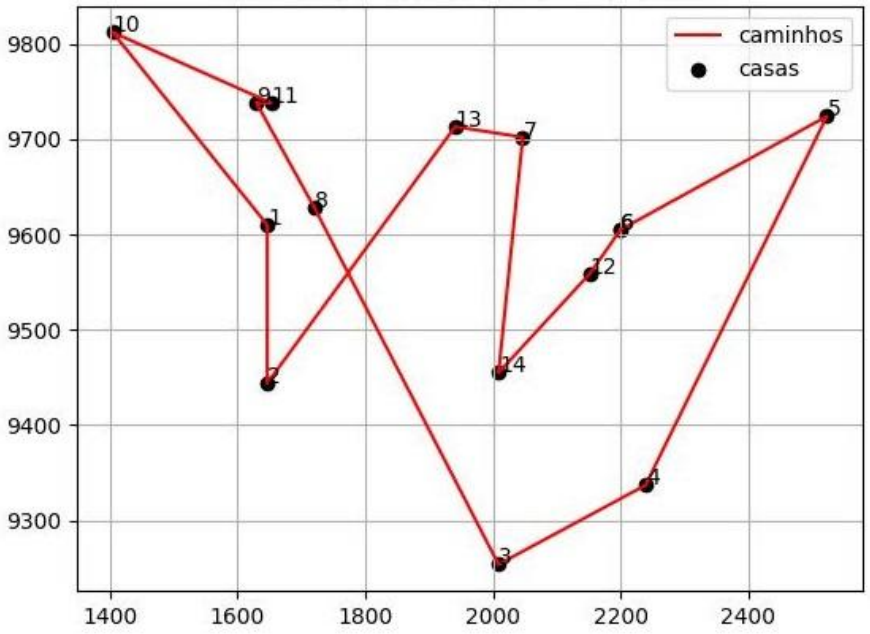

Figura 5: Exemplo do melhor caminho encontrado pelo operador PMX.

Executando o Algoritmo genético com o operador PMX e possui uma consistência de melhoramento de indivíduo com o passar das gerações, enquanto a curva de melhoramento de indivíduo acontece muito mais acentuada no operador CX, como podemos observar nas figuras a seguir:

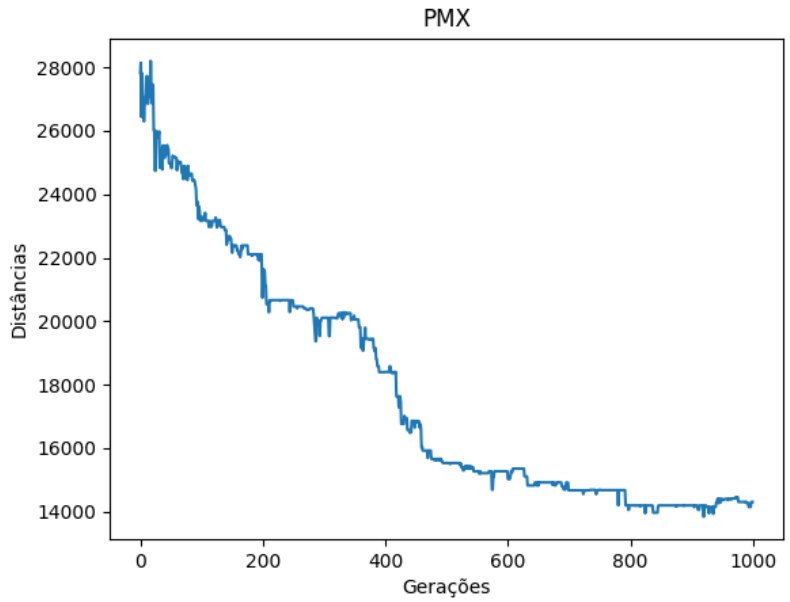

Figura 6: Exemplo de resultado obtido com o operador PMX

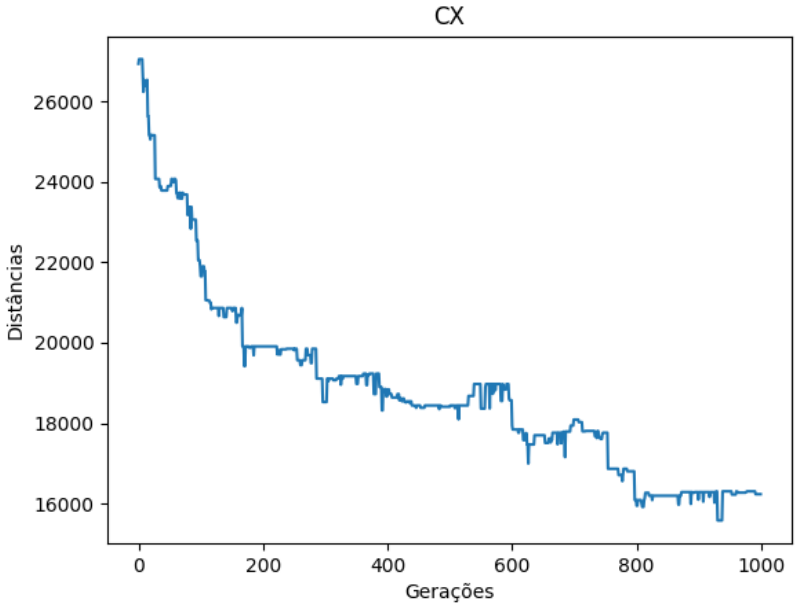

Figura 7: Exemplo de resultado obtido com o operador CX

Constatou-se, por fim, que a comparação entre os algoritmos Brute Force e Algoritmo Genético revelou que, para matrizes pequenas com até 9 pontos, o uso do algoritmo Brute Force é mais vantajoso, uma vez que sempre produz uma solução ótima em aproximadamente 1 segundo. Contudo, em matrizes maiores, obter uma solução ótima exigiria um tempo impraticável no mundo atual. Nesse sentido, a solução adotada seria abrir mão da solução ótima e aceitar uma solução satisfatória, considerando a diferença evidente entre os tempos de solução fatorial e polinomial, conforme demonstram os gráficos abaixo:

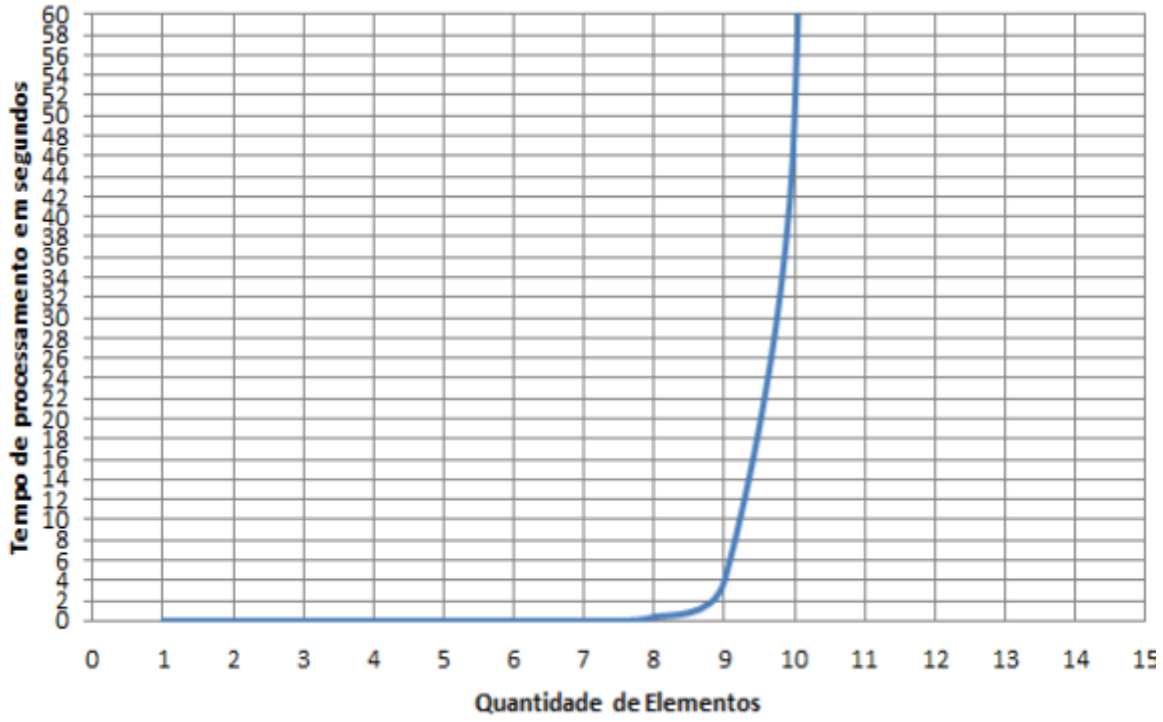

Figura 8. Gráfico do № elementos com o tempo de processamento do algoritmo de brute force.

7. Conclusão

Este trabalho teve como objetivo encontrar a menor rota no sistema de delivery com drones da empresa Flyfood, para o qual foram desenvolvidos e apresentados dois algoritmos distintos: Brute Force e Algoritmo Genético. Observou-se que, embora o algoritmo Brute Force proporcione uma solução ótima para 100% dos casos, ele se torna inviável em rotas maiores, devido ao seu alto custo computacional de ordem fatorial. Por outro lado, o Algoritmo Genético fornece soluções satisfatórias e com menor tempo de processamento em rotas maiores.

Em relação ao tempo de processamento, a diferença entre os algoritmos Brute Force e Algoritmo Genético em termos de complexidade foi clara. O algoritmo Brute Force tem complexidade fatorial, o que significa que o tempo de processamento aumenta exponencialmente à medida que a quantidade de pontos a serem percorridos aumenta. Em contrapartida, o Algoritmo Genético tem complexidade polinomial, o que significa que o tempo de processamento aumenta de forma mais suave e previsível à medida que a quantidade de pontos aumenta. Assim, o Algoritmo Genético se mostrou uma opção mais viável para rotas maiores, uma vez que permite encontrar soluções satisfatórias em um tempo de processamento menor em comparação com o Brute Force.

É importante destacar a relevância dos algoritmos de roteamento para a computação, especialmente em sistemas de delivery como o Flyfood. A possibilidade de encontrar rotas mais eficientes e rápidas para a entrega de produtos pode trazer benefícios significativos para as empresas e clientes envolvidos no processo, tornando-o mais ágil e econômico. Nesse sentido, é fundamental continuar aprimorando a solução desenvolvida para atender às necessidades do mercado de delivery em constante evolução.

Referências Bibliográficas

RODRIGUES, R. A. N.; CARVALHAES, M. F. A. Análise e Complexidade de Algoritmos: Backtracking e Força Bruta. **Revista Ada Lovelace**, *[S. l.]*, v. 1, p. 45–48, 2017. Disponível em: http://anais.unievangelica.edu.br/index.php/adalovelace/article/view/4117. Acesso em: 10 jan. 2023.

SILVEIRA, J.f. Porto da. Problema do Caixeiro Viajante. 2000. Disponível em: http://www.mat.ufrgs.br/~portosil/caixeiro.html.

Godoy Dotta, Alexandre (2021): USO DE PROGRAMAÇÃO EM „R" APLICADO AO PROBLEMA CAIXEIRO VIAJANTE COM OFICINA PARA DEMONSTRAR DEMAIS APLICAÇÕES. figshare. Presentation. https://doi.org/10.6084/m9.figshare.13857299.v1

TOVÁ, A. C.; MENDES, O. L. OTIMIZAÇÃO DE ROTAS PARA ENTREGA DOS INGREDIENTES DA MERENDA EM ESCOLAS DA CIDADE DE BARRETOS – SP. Revista Interface Tecnológica, [S. l.], v. 17, n. 2, p. 825–837, 2020. DOI: 10.31510/infa.v17i2.1028. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1028

BARBOSA, R. C. Aplicação da metaheurística algoritmo genético na otimização das rotas de entregas da distribuição física de produtos no município de Fortaleza. 90 f. 2014. Dissertação (Mestrado em Logística e Pesquisa Operacional) – Pró-Reitoria de Pesquisa e Pós-Graduação, Universidade Federal do Ceará, Fortaleza, 2014. Disponível em: https://repositorio.ufc.br/handle/riufc/15038

NERY, Samuel Wanberg Lourenço et al. Análise de operadores de cruzamento genético aplicados ao problema do Caixeiro Viajante. Monografia (Bacharelado em Ciência da Computação) - Universidade Federal de Goiás, Goiânia, 2016. Disponível em: https://files.cercomp.ufg.br/weby/up/498/o/SamuelWanbergLourencoNery2016.pdf

Acesso em 15 mar. 2023.

LARRANAGA, P.; MURGA, C. M. H. K. R.; INZA, I.; DIZDAREVIC, S. Genetic algorithms

for the travelling salesman problem: A review of representations and operators. Artificial Intelligence Review, v. 13, p. 129–170, 1999.

TSPLIB. (s.d.). The Traveling Salesman Problem Library. Recuperado em 17 de março de 2023, de http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/