Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

SOLUÇÃO ALGORÍTMICA PARA O FLYFOOD: FORÇA BRUTA E ALGORITMO GENÉTICO

Carla Maria Carolyne Marques da Silva

Recife

Março de 2023

Resumo

O desenvolvimento acelerado das cidades nas últimas décadas trouxe uma série de desafios para a mobilidade urbana, em particular para o trânsito de veículos. As ruas cada vez mais lotadas e congestionadas têm prejudicado a vida das pessoas, bem como o funcionamento das empresas que dependem de transporte para realizar suas atividades, como os serviços de entrega.

Nesse contexto, a FlyFlood tem buscado soluções inovadoras para melhorar a eficiência das entregas, utilizando drones para realizar o transporte de alimentos e outros produtos. No entanto, um dos principais obstáculos enfrentados pela empresa é a limitação da vida útil das baterias dos drones, que restringe a área de cobertura e a quantidade de entregas que podem ser feitas em um determinado ciclo.

Para superar esse desafio, a equipe da FlyFlood decidiu investir em um projeto de otimização de rotas para drones de entrega. A ideia é desenvolver um algoritmo que possa encontrar a rota mais eficiente para o drone percorrer, levando em conta não apenas a distância a ser percorrida, mas também a duração da bateria e outros fatores relevantes.

Inicialmente, utilizou-se o algoritmo de força bruta para otimizar as rotas de seus drones de entrega. No entanto, após analisar os resultados e considerar as possibilidades de aprimoramento do método, a equipe decidiu adotar um novo modelo: o algoritmo genético.

Palavras-chave: algoritmo; força bruta; algoritmo genético, problema do caixeiro viajante; classe de problemas; análise de algoritmos; grande O; big O; otimização; NP completo; NP hard; análise assintótica; Heurística e Meta-heurística

1. Introdução

1.1 Apresentação e Motivação

A otimização de rotas tem se tornado cada vez mais importante na vida moderna, especialmente no setor de delivery. Com o aumento do número de entregas realizadas diariamente, torna-se fundamental encontrar soluções que permitam reduzir custos e tempo de deslocamento, garantindo que as entregas sejam realizadas de forma eficiente e sem atrasos.

A FlyFood é uma empresa que busca oferecer soluções inovadoras para otimização de entregas, utilizando drones capazes de transportar vários pedidos em seu compartimento. No entanto, a limitação das baterias dos drones é um grande desafio que precisa ser superado para tornar essa solução viável.

Para enfrentar esse desafio, a empresa está desenvolvendo um algoritmo capaz de determinar a melhor rota para o drone realizar as entregas, levando em consideração a duração da bateria, o ponto de origem, os pontos de entrega e o ponto de retorno do drone. O objetivo é garantir que todas as entregas possam ser realizadas dentro do ciclo de bateria do drone, maximizando a eficiência do processo.

Com essa solução, a FlyFood espera oferecer um serviço de delivery mais rápido, eficiente e sustentável, reduzindo o tempo de deslocamento e minimizando o impacto ambiental causado pelos veículos convencionais. A otimização de rotas é um importante passo em direção a um futuro mais inteligente e sustentável para o setor de entregas.

1.2 Formulação do problema

Para criar um software prático e real, é necessário definir os conceitos e equações que serão utilizados para resolver o problema. O programa terá como entrada uma matriz que contém o ponto de origem (chamado de Ponto R) e os pontos de entrega.

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

O drone que fará as entregas deve partir de um Ponto de Partida e retornar a ele depois de completar todas as entregas nos Pontos de Entrega. As coordenadas de uma matriz são as posições específicas de cada elemento na matriz, e são representadas por duas dimensões, linha e coluna.

Para avaliar o custo do percurso, utilizaremos o termo "Danômetro" como unidade de medida de distância. A Distância de Manhattan é uma medida de distância comumente utilizada em planos bidimensionais, também conhecida como Distância de Táxi ou Geometria Pombalina. Ela é calculada como a soma das diferenças absolutas dos componentes das coordenadas dos pontos. A distância de Manhattan é amplamente utilizada em várias áreas por sua simplicidade e precisão na representação de distâncias em planos 2D.

Para calcular a distância entre dois pontos utilizando a fórmula:

$$d = |x1 - x2| + |y1 - y2|$$

A fórmula representa a distância Manhattan entre dois pontos em um plano 2-dimensional com coordenadas (x1, y1) e (x2, y2). As barras de valor absoluto garantem que a diferença entre os dois pontos sempre seja positiva, e as duas diferenças são então somadas para encontrar a distância total entre os pont

Para encontrar o menor percurso para realizar as entregas, precisamos calcular o somatório da distância de Manhattan (também conhecido como distância L1):

$$f(d) = \sum i \; d(n\boldsymbol{i} \; , \; n\boldsymbol{i}+\mathbf{1})$$

O somatório pode ser referido como f(d), e o objetivo é encontrar o argumento mínimo da função f(d), ou seja, argmin(f(d)). Esse argumento representa o caminho que minimiza f(d) e resulta no menor percurso possível. Com essa solução escalável, a distância de Manhattan será calculada diversas vezes para encontrar o menor percurso possível.

1.3 Objetivos

O objetivo geral deste trabalho é construir uma solução computacional que seja capaz de encontrar o percurso de menor distância total entre os destinos de entrega e o ponto inicial, de forma mais rápida e otimizada, além de conduzir experimentos para verificar a eficiência do algoritmo.

Objetivos específicos:

1. O estudo da literatura e da análise de algoritmos.
2. Codificação do algoritmo
3. Fazer experimentos para provar a eficácia da solução

1.4 Organização do trabalho

Na seção 1 do relatório encontra-se a Introdução onde é possível encontrar sucintamente o contexto do trabalho e o que foi feito, como por exemplo a formulação do algoritmo para resolver o problema. Já na seção 2 encontra-se o Referencial Teórico onde é possível encontrar conceitos que foram usados ao fazer esse trabalho. Além disso, na seção 3 é possível encontrar trabalhos relacionados a esse relatório e na seção 4 será encontrada a metodologia utilizada no trabalho. Por fim, nas seções seguintes podemos visualizar os experimentos realizados, bem como os resultados, conclusão e as referências bibliográficas do relatório.

2. Referencial Teórico

No referencial teórico, trataremos dos conceitos importantes para o entendimento total do trabalho, como a análise de algoritmos, notação assintótica, classe de problemas.

2.1 Análise de Algoritmos

Segundo Cormen (2012, p. 16), “analisar um algoritmo significa prever os recursos de que o algoritmo necessita”. Tal análise se faz necessária uma vez que, ao solucionar um problema computacionalmente, haverão recursos limitados, como a memória. Um algoritmo precisa encontrar a solução, mas também é preciso que seja escrito da maneira mais eficaz possível, evitando custos maiores.

O tipo de análise estudada neste trabalho foi o da ordenação por inserção, que se baseia no tamanho da entrada e no tempo de execução. O tamanho de entrada pode ser o número de itens na entrada ou o número total de bits necessários para representar a entrada. No entanto, o tempo de execução representa o número de passos feitos, ou seja, a soma dos tempos para cada instrução executada. Em tal análise, determinamos o tempo de execução do pior caso, para que tenhamos um limite e garantia sobre o maior tempo possível. Com o tempo de execução do pior caso calculado, atribuímos tal função a *t(n)*. Tal função é a responsável por calcular o crescimento da função, de acordo com a entrada (*n*).

Um algoritmo pode ser eficiente de duas formas: a eficiência temporal e a eficiência espacial. A eficiência temporal se refere ao tempo de execução, o quão rápido o algoritmo em questão é executado. A eficiência espacial está relacionada ao espaço extra que o algoritmo necessita durante a execução. Devido a grande quantidade extra de memória que temos nos dias atuais – quando comparado aos computadores iniciais – será a coadjuvante na análise de algoritmos neste trabalho.

Com a análise do algoritmo, também se torna possível prever a quantidade de recursos, como memória, tempo de execução que seriam alocados para funcionamento perfeito de tal algoritmo. O tempo de execução, ou seja, o custo é expresso na função anteriormente comentada, *t(n)*. A eficiência do algoritmo torna-se muito importante quando o problema a ser resolvido é de grande dimensão.

2.2 Algoritmos de força bruta para o problema do caixeiro viajante

A abordagem de força bruta para resolver o problema do caixeiro viajante testa todas as combinações possíveis de caminhos para encontrar a solução ideal. Apesar de garantir a solução ótima, pode ser impraticável para problemas maiores devido ao alto consumo de recursos. Por exemplo, para 10 cidades, existem mais de 3 milhões de possíveis combinações de caminhos. Para ilustrar, se considerarmos 5 cidades, o número de possíveis combinações pode ser calculado por meio da fórmula (5-1)! = 4! = 24, o que implica em 24 rotas que podem ser percorridas pelo caixeiro viajante.

2.3 Algoritmos genético para o problema do caixeiro viajante

O algoritmo genético é uma técnica de otimização que se baseia na evolução biológica para resolver problemas complexos em diversas áreas, como engenharia, ciência da computação e biologia. Para o problema do caixeiro viajante, por exemplo, o algoritmo genético gera uma população inicial de soluções aleatórias, representando sequências de cidades, e aplica seleção, cruzamento e mutação para gerar novas soluções mais eficientes.

A seleção é feita escolhendo as soluções mais aptas, aquelas com menor custo (ou distância percorrida), enquanto o cruzamento mistura características de duas soluções selecionadas e a mutação adiciona pequenas perturbações às soluções. Esse processo é repetido por várias gerações, gerando soluções cada vez melhores, até que uma solução ótima seja encontrada ou um critério de parada seja atingido.

Assim, o algoritmo genético é uma técnica poderosa e flexível para encontrar soluções aproximadas para problemas complexos como o do caixeiro viajante.

Etapas do Algoritmo genético que pode ser usado para resolver o problema do caixeiro viajante:

1. Inicialização: Criação de uma população aleatória de rotas, onde cada rota é representada por uma sequência aleatória das cidades que devem ser visitadas.
2. Avaliação: Cálculo do valor de aptidão (fitness) de cada rota da população, com base na distância total percorrida (menor distância).
3. Seleção: Escolha as rotas mais aptas para reprodução, geralmente as rotas com os maiores valores de aptidão (para o problema do caixeiro viajante significa menor percurso).
4. Cruzamento: Combinar partes das rotas selecionadas para gerar novas rotas. Para realizar o cruzamento, primeiro é necessário selecionar um ponto de corte na rota, que pode ser escolhido aleatoriamente. A partir desse ponto de corte, os filhos são gerados combinando as partes de cada pai.
5. Mutação: Introduzir pequenas alterações aleatórias nas novas rotas geradas.
6. Substituição: Substituição da população anterior pela nova população de rotas geradas.
7. Condição de parada: Verificar se a solução ótima foi encontrada ou um critério de parada foi atingido, caso contrário, retornar à etapa de seleção.

2.4 Notação Assintótica

Ao analisarmos um algoritmo com tamanhos de entrada suficientemente grandes que fazem a ordem de crescimento do tempo de execução se tornar relevante, estamos fazendo a análise assintótica de um algoritmo. Um algoritmo assintoticamente eficiente provavelmente será a melhor escolha para todas as entradas.

A notação assintótica é, no entanto, uma notação matemática usada para melhor analisarmos o comportamento das funções para grandes valores de n, podendo ignorar as constantes dadas na função *t(n)*. E o comportamento da velocidade que as funções crescem, é chamado de comportamento assintótico. Os algoritmos que têm funções assintoticamente menores, que crescem mais devagar, são as melhores opções para tratar entradas grandes. Antes de entender os tipos de notação, é importante que entendamos os tipos de análise possíveis.

Ao fazermos a análise do algoritmo, não estamos interessados somente no tempo de execução geral, mas nos extremos. Existem o pior caso, o caso médio e o melhor caso. O pior caso significa o tempo máximo de execução, ou seja, consiste basicamente em assumir o pior dos casos que podem acontecer, sendo muito usado e sendo normalmente o mais fácil de determinar. O melhor caso assume que o melhor irá acontecer, ou seja, o resultado do menor tempo possível. Não é muito utilizado e tem aplicação em poucos casos. O caso médio que retorna o tempo médio de execução para todo tipo de entrada possível, é o mais difícil de determinar, pois necessita de análise estatística e, portanto, de muitos testes. Porém, é muito utilizado.

Na análise assintótica, as constantes multiplicativas e os termos de menor ordem dados na função *t(n)* são ignorados e é adotada a notação matemática específica para representar a complexidade do algoritmo. É válido reforçar que existem outras abordagens técnicas para a análise de algoritmos, porém a assintótica é a abordagem mais utilizada encontrada na literatura.

Existem tipos de notação, como o Big O, Ω e Θ. O Big O ou Grande O, é uma notação dada ao pior caso do algoritmo. Será o limite assintótico superior. A notação Ω será o limite assintótico inferior, pode ser usado para o melhor caso. A notação Θ representa o limite assintótico firme (figura 2).

Com o Big O de *t(n)*, é possível observar o limite superior de cada algoritmo e ver como ele se comporta, quando o número de entradas tende ao infinito. Os algoritmos mais eficientes – quanto ao tempo – terão crescimento devagar. Ou seja, os que são O(n!) e $O(2^n)$ não serão capazes de tratar entradas grandes tão bem como os que são O(n) ou O(n log n).

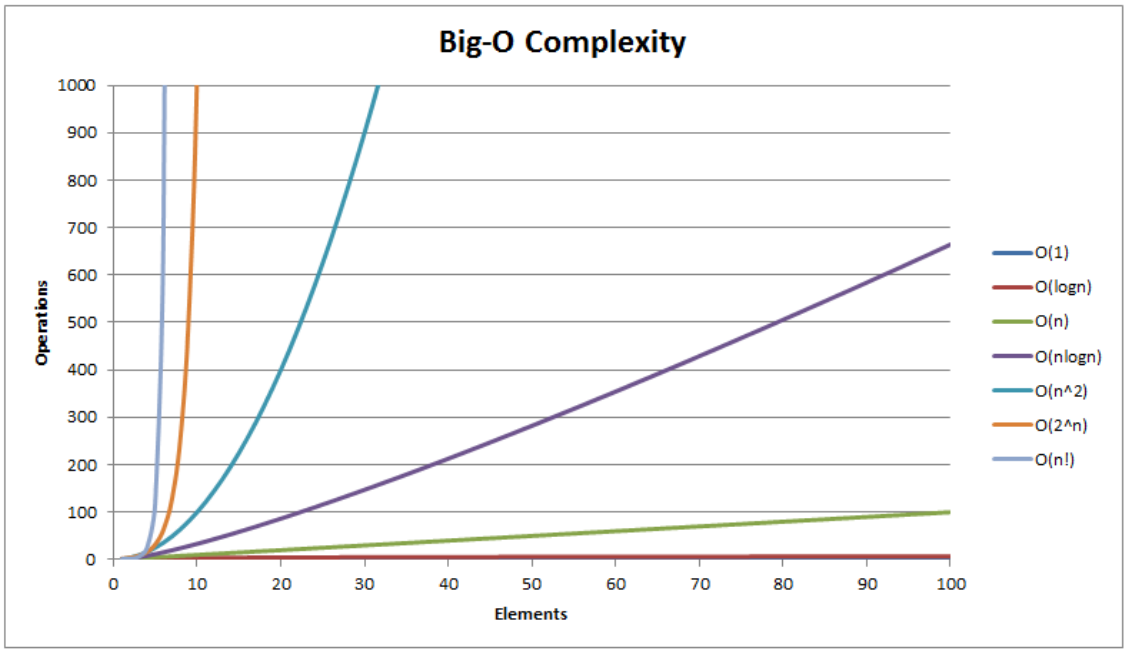

Figura 1. Gráfico da notação BIG-O (O grande)

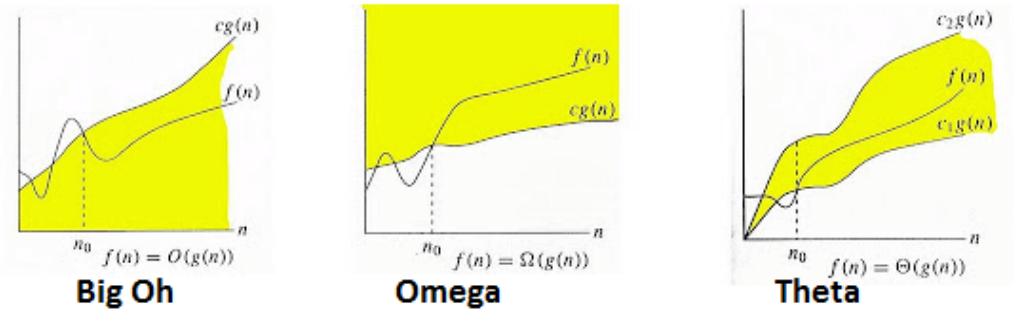

Figura 2. Gráfico comparativo das notações O, Omega e Theta

2.5 Classe de problemas

Existem, então, classes de problemas nas quais se encaixam os algoritmos. Como por exemplo os algoritmos polinomiais, que são definidos polinomiais se seu consumo de tempo no pior caso é limitado por uma função polinomial do tamanho das instâncias do problema. São considerados tratáveis e razoavelmente rápidos. Sobre algoritmos polinomiais, podemos afirmar que:

> A classe P consiste nos problemas que podem ser resolvidos em tempo polinomial. Mais especificamente, são problemas que podem ser resolvidos no tempo O(nk) para alguma constante k, onde n é o tamanho da entrada para o problema. (CORMEN, Thomas. **Algoritmos - Teoria e Prática**. Grupo GEN, 2012. p. 765)

Também existem os algoritmos não polinomiais, que são não polinomial se não existir algoritmo polinomial que resolva o problema. São considerados inaceitavelmente lentos. A classe NP consiste nos problemas que podem ser verificados em tempo polinomial e são considerados intratáveis. Um problema é NP-completo se pertence a NP e todo outro problema de NP se reduz polinomialmente ao problema dado. Um problema *Q* é NP-difícil se todo outro problema de NP se reduz polinomialmente a *Q*.

O circuito hamiltoniano e o problema do caixeiro viajante são dois problemas que são considerados NP-completos. O circuito hamiltoniano descreve o seguinte problema: dado um grafo, encontrar um circuito simples que passe por todos os vértices do grafo (ou constatar que tal circuito não existe). Tal problema pode ser verificado em tempo polinomial, mas não há solução polinomial. O Problema do Caixeiro Viajante é um problema que tenta determinar a menor rota para percorrer uma série de cidades, retornando à cidade de origem. Os dois problemas têm grande semelhança com o problema que este trabalho descreve, portanto o FlyFood também será considerado um problema NP-completo, já que consegue se reduzir a esses dois problemas.

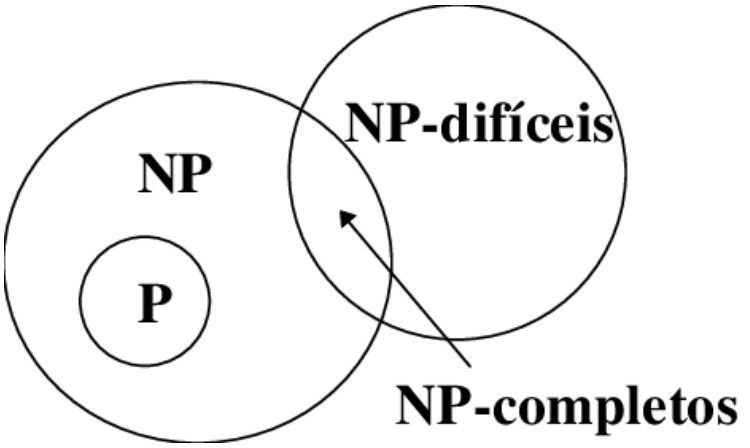

Figura 3. Diagrama de Venn com os tipos de problemas.

2.6 Heurística e Meta-heurística

Segundo Elizabeth Goldbarg, em “Otimização Combinatória e Meta-heurísticas - Algoritmos e Aplicações”, uma heurística é “ uma técnica computacional aproximativa que visa alcançar uma solução avaliada como aceitável para um dado problema que pode ser representado em um computador, utilizando um esforço computacional considerado razoável, sendo capaz de garantir, em determinadas condições, a viabilidade ou a otimalidade da solução encontrada.”. Em muitos casos, é esperado a solução ótima por heurísticas de problemas NP-difíceis ou NP-completos. No entanto, tal solução ótima não pode ser garantida, então, ao usar uma heurística, tomamos a decisão do que é mais importante: algoritmos computacionalmente eficientes ou a solução ótima.

Neste presente trabalho, o algoritmo de força bruta desenvolvido trará a resposta ótima, no entanto, não será eficiente computacionalmente. Para termos um algoritmo que resolve o problema em tempo razoável, desenvolvemos o algoritmo genético aplicado ao nosso problema de roteamento. O algoritmo genético é uma meta-heurística, que pode ser explicada por Goldbarg:

> Trata-se de uma arquitetura geral de regras que, formada a partir de um tema em comum, pode servir de base para o projeto de uma ampla gama de heurísticas computacionais. (GOLDBARG, Elizabeth. **Otimização Combinatória e Meta-heurísticas - Algoritmos e Aplicações.** Grupo GEN, 2015. E-book. ISBN 9788595154667. p. 72)

Gendreau e Potvin, em seu livro "Handbook of Metaheuristics" também trazem uma definição para Meta-heurística, na qual defendem que "Meta-heurísticas, em sua definição original, são métodos de solução que orquestram uma interação entre procedimentos de melhoria local e estratégias de alto nível que criam um processo capaz de escapar o ótimo local e performar uma busca robusta entre o espaço de solução." (tradução literal).

2.7 Computação Evolucionária

A computação evolucionária é uma área da computação bioinspirada. Segundo Goldberg, é caracterizada pelas seguintes condições: É realizada através de um processo iterativo, baseia-se em uma população, possui intrinsecamente uma arquitetura de processamento paralelizável, corresponde a um processo de busca estocástica com viés – busca guiada, emprega o princípio darwiniano da seleção natural de acúmulos de variações genéticas. Eles operam seguindo o seguinte esquema geral:

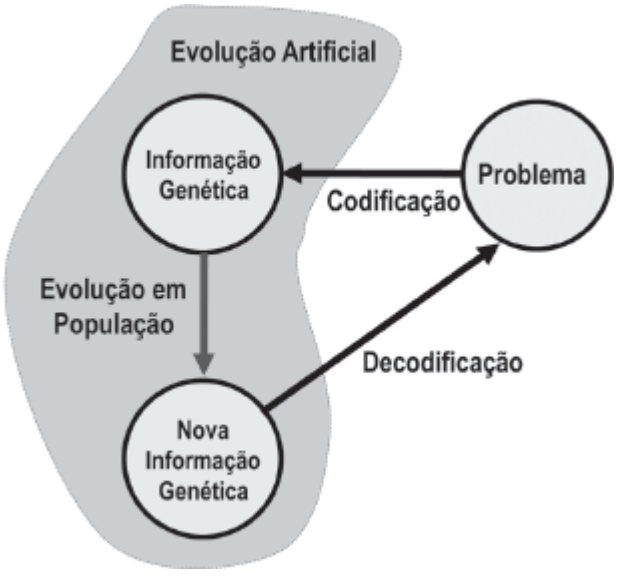

Figura 4: Evolução da informação genética. Retirada do livro Otimização Combinatória.

Tal informação genética, chamada também de cromossomos, será transportada por uma população de indivíduos que evolui de acordo com a seleção. Tais indivíduos da população representam propostas de solução para o problema em pauta, sejam elas viáveis ou não. É então feita a avaliação dos indivíduos, e o processo de reprodução é direcionado a favorecer os indivíduos mais aptos, seguindo a teoria da seleção natural de Darwin. A modificação do material genético de cada indivíduo foi descrita por Goldberg como: “No contexto computacional, recombinações e mutações são procedimentos heurísticos que visam utilizar as possibilidades combinatórias codificadas nos indivíduos da população trabalhada pelo algoritmo.”.

Os algoritmos evolucionários mimetizam o processo biológico, no entanto, ainda admitem grande grau de afastamento do modelo natural, segundo Burke *et al* em 1998. No entanto, tal afastamento pode ser justificável, já que há grande diferença entre o processo biológico e o computacional. Além disso, aprofundar a mimetização e deixar o processo computacional muito semelhante ao biológico não preserva a simplicidade de uma heurística, que é um dos aspectos mais importantes para o sucesso da Computação Evolucionária (Goldberg, 2015).

2.8 Algoritmo Genético

O processo de mimetização biológica que suporta a metáfora dos Algoritmos Genéticos está associado à reprodução multicelular sexuada. Ou seja, o processo de variação genética é realizado por meio da reprodução e da mutação. Tal reprodução segue o paradigma sexual, que reúne pais para a produção de um ou mais filhos.

Em tal algoritmo, “o indivíduo é representado por um cromossomo. A reprodução é um processo de mistura do material genético de dois (ou mais) indivíduos.” (GOLDBERG, 2015). Tal modelo se concentra na evolução biológica empregando os três princípios de Darwin: variação genética, hereditariedade e seleção natural. A variedade genética pode ser resultado de mutações ou de recombinação de genes. A hereditariedade é a preservação dos genes pertencentes aos pais. A seleção natural pode ser feita de três formas mais conhecidas: genética, de organismos e de grupos.

O algoritmo genético, segue então, alguns princípios. O primeiro é de codificação da informação em um conjunto de cromossomos denominado de população. O segundo é a reprodução da população por meio da combinação genética dos cromossomos. O terceiro é o emprego de mutações aleatórias ou guiadas para introduzir diversidade na população. E, por fim, fizemos a seleção dos cromossomos de melhor adequação (aptidão). Esses foram os princípios utilizados para o desenvolvimento desse algoritmo. Alguns termos muito comuns para descrever o

processo de mimetização genética estão presentes na seguinte tabela, retirada do livro Otimização Combinatória de Goldberg:

Natureza	Algoritmo Genético
Indivíduo – cromossomo	Solução viável do problema
População	Conjunto de soluções
Adequação (*fitness*)	Valor (ou custo) da solução
Gene	Parte da solução
Cruzamento	Operador de busca
Mutação	Operador de busca
Seleção natural	Seleção de soluções

Tabela 1. Analogias da Mimetização

3. Trabalhos relacionados

Esta seção tem como objetivo mostrar alguns trabalhos relacionados ao Flyfood, que abordam os tópicos descritos no referencial teórico.

Jacqueline Fonseca, Elisangela Martine, Fabrício Molica e Paulo Sanches, apresentaram em seu trabalho “Otimização de rotas de entregas de materiais em uma rede hospitalar por meio do algoritmo do problema do caixeiro viajante” propostas de roteirização para distribuição de materiais médico-hospitalares. Nessas propostas, valorizaram minimizar os custos de transporte, por meio da redução das distâncias percorridas. Foram testados, então, modelos de otimização linear, baseados no algoritmo do Problema do Caixeiro Viajante. Em tal trabalho, foram utilizados cinco veículos nos quais dividiam rotas, diferentemente do FlyFood, que levava em conta apenas um drone por rota. Além disso, também eram considerados o peso máximo transportado por cada veículo e as restrições impostas pelas normas de circulação da

cidade de Belo Horizonte. Tal trabalho, apesar de muito bem construído e oferecer uma solução exata para o problema, foca bastante na distribuição dos veículos, e de cargas, o que não é o foco do FlyFood.

Gabriel Alfatini *et al* apresentam em seu trabalho "Algoritmos heurísticos construtivos aplicados ao Problema do Caixeiro Viajante para a definição de rotas otimizadas." a análise de tais algoritmos utilizados na otimização de rotas para resolver o Problema do Caixeiro Viajante. Em seu trabalho, eles estudam o algoritmo do vizinho mais próximo, inserção do mais distante, inserção do mais rápido e inserção do mais próximo, sendo o de inserção do mais distante com maior vantagem. No entanto, tais algoritmos heurísticos nem sempre correspondiam à rota mais rápida. Os experimentos foram feitos através de um aplicativo móvel e do Google Geocoding API, com as coordenadas geográficas para os vértices das rotas. Tal trabalho, apesar de interessante, utiliza apenas de heurísticas simples, como a do vizinho mais próximo. Neste trabalho, desenvolvemos dois algoritmos, um de solução exata e uma metaheurística mais sofisticada.

Ferreira Ribeiro, José Francisco *et al,* em "Logística para o recolhimento de frutas: um estudo de caso", apresentam um método e um programa computacional que determinava a ordem de passagem dos veículos para recolher as frutas colhidas nas propriedades rurais do Brasil e de Gana. Utilizaram do modelo matemático do problema do caixeiro viajante e do *Evolutionary*, disponível no *Microsoft-Excel-Solver*. Os testes realizados mostraram bom desempenho e resultados melhores ou iguais aos métodos que eram utilizados anteriormente para tomada de decisão da rota. No entanto, tal trabalho apenas oferece a solução exata com o apoio de um software que já está tem o algoritmo necessário para resolver o problema do caixeiro viajante.

4. Metodologia

Para desenvolver a solução computacional, utilizamos a linguagem Python e implementamos dois tipos de algoritmos: um baseado em força bruta e outro em metaheurística (algoritmo genético). O algoritmo de força bruta utiliza chamadas recursivas e buscas sequenciais para encontrar a solução ótima. Já o algoritmo genético é uma técnica de otimização baseada na evolução biológica, que busca encontrar uma solução satisfatória através de uma população de soluções e operadores genéticos. Com esses dois tipos de solução algorítmica, buscamos oferecer ao usuário diferentes opções para resolver o problema proposto de forma eficiente.

4.1 Construção do algoritmo de força bruta

O algoritmo desenvolvido neste trabalho recebe como entrada um arquivo de texto, no qual contém o tamanho da matriz, especificado na primeira linha, e a matriz nas linhas posteriores. O ponto inicial é representado pela letra “R” e os pontos de entrega são representados por letras A, B, C e D.

Para ler um arquivo que contém uma matriz e separar suas coordenadas em Python, foram necessário os seguintes passos:

1. Usar a função open() para abrir o arquivo e armazenar o objeto arquivo em uma variável.
2. Usar o método read() para ler o conteúdo do arquivo como uma string.
3. Usar a função split() para separar a string em uma lista de linhas.
4. Criação de listas vazias
5. Usar loops para iterar sobre as linhas da lista e separar cada linha em uma lista de elementos.
6. Usar a condição “se” para adicionar a lista vazia todos os pontos que sejam diferentes de 0.
7. Usar o método remove() para remover da lista o ponto R e coordenada (0,3)

Dessa forma, o programa abre o arquivo "matriz" para leitura e armazena em "file".
Lê a primeira linha de "file" e armazena os valores separados por espaço em "n" e "m".
Lê todas as linhas restantes de "file" e armazena em "lines" e cria uma lista vazia chamada "lista_coordenadas" e outra "lista_pontos".
Encontra o índice de "R" em "lista_pontos" e armazena em "indice".
Cria uma variável "ponto_R" com o valor na posição "indice" em "lista_coordenadas"
Remove a coordenada R em "lista_coordenadas[indice]" e remove a string "R" de "lista_pontos".

```
# Recebendo como entrada a matriz e separando seus pontos e
coordenadas.

file <- open("matriz", "r")
n, m <- file.readline().split()
lines <- file.read().splitlines()
lista_coordenadas <- []
lista_pontos <- []
para i no intervalo de 0 até (n - 1) faça
    line <- lines[i].split()
    para j em line então
        se j != "0" então
            coordenada <- (i, line.index(j))
            lista_coordenadas.append(coordenada)
```

```
            lista_pontos.append(j)
        fim-se
    fim-para
fim-para
indice <- lista_pontos.index("R")
ponto_R <- lista_coordenadas[indice]
lista_coordenadas.remove(lista_coordenadas[indice])
lista_pontos.remove("R")
```

Figura 5. Pseudocódigo que extrai da matriz os pontos e coordenadas

Após isso, com a lista que contém todos os pontos de entrega, definiremos todas as permutações de rotas possíveis para que seja possível o cálculo da distância através da função *permute()*. A função utiliza o algoritmo de backtracking para gerar todas as permutações de uma lista de forma recursiva e possui três argumentos: lista, k e tamanho_lista, onde lista é a lista que será permutada, k é a posição inicial da permutação atual e tamanho_lista é o tamanho total da lista.

LISTA_PERMUTADA é uma lista global vazia que irá armazenar todas as permutações geradas pela função permute.

Sendo assim, o corpo da função permute é executado recursivamente. A cada chamada recursiva, o valor de k é incrementado em 1. Se k é igual a tamanho_lista, significa que todos os elementos da lista foram permutados e uma nova permutação é adicionada à LISTA_PERMUTADA. Caso contrário, o algoritmo itera sobre o intervalo [k, tamanho_lista) e permuta cada elemento da lista com o elemento na posição k. Em seguida, a função permute é chamada recursivamente com k + 1 como novo valor de k. Ao final da iteração, os valores são trocados de volta para sua posição original, permitindo que outra permutação seja gerada.

Este algoritmo garante que todas as permutações possíveis sejam geradas sem duplicidade, e todas as permutações geradas são armazenadas na lista LISTA_PERMUTADA.

```
# Função que permuta todas minhas rotas possíveis
LISTA_PERMUTADA <- []
funcao permute(lista, k, tamanho_lista):
    se k == tamanho_lista então
        LISTA_PERMUTADA.append(tuple(lista))
    senao faça
        para i no intervalo de k até (tamanho_lista - 1) faça
            lista[k], lista[i] <- lista[i], lista[k]
            permute(lista, k + 1, tamanho_lista)
            lista[k], lista[i] <- lista[i], lista[k]
    fim-se
```

```
fim-funcao
```

Figura 6. Pseudocódigo da função permutação das rotas de entrega

Atrelado a isso, foi definido a função "calcular_distancias" onde tem como objetivo calcular a distância total percorrida a partir da posição inicial "ponto_R".

Ela realiza isso de duas formas: primeiro, a função "permute" é chamada para gerar todas as permutações possíveis das coordenadas das rotas, armazenando-as na lista "LISTA_PERMUTADA". Em seguida, é realizado um loop através das rotas geradas, calculando a distância total entre cada ponto da rota e armazenando-a na lista "distancias". E finalmente, a função retorna a lista "distancias".

```
funcao calcular_distancias(lista_coordenadas, ponto_R):
    distancias <- []
    permute(lista_coordenadas, 0, len(lista_coordenadas))
    coordenadas <- LISTA_PERMUTADA
    para cada pontos na lista coordenadas faça
        pontos <- list(pontos)
        pontos.append(ponto_R)
        posicao_atual <- ponto_R
        distancia_total <- 0
        para cada ponto na lista pontos faça
            x <- posicao_atual[0] - ponto[0]
            y <- posicao_atual[1] - ponto[1]
            dist_percorrida <- abs(x) + abs(y) #soma do módulo de
x e y.
            distancia_total += dist_percorrida
            posicao_atual <- ponto
        fim-para
        distancias.append(distancia total)
    fim-para
    retorne distancias
fim-função
```

Figura 7. Pseudocódigo da função que calcula a distância das rotas de entrega

Finalmente, o código busca o caminho mínimo entre as rotas para efetuar a entrega. A função "calcular_distancias" é usada para calcular a distância entre o ponto de partida (ponto_R) e todos os outros pontos da lista_coordenadas. O resultado é armazenado na variável "distancias".

Em seguida, há um loop que percorre todas as distâncias calculadas e verifica qual delas é a menor. Se a distância atual é igual à distância mínima, o código usa a função "permute" para gerar todas as permutações possíveis das coordenadas na lista_coordenadas e armazena a permutação corrente em "pontos_permutados".

Por fim, o código usa uma list comprehension para criar uma lista "nomes_pontos" que contém os nomes dos pontos ao invés das coordenadas. A lista é criada a partir de uma busca pelo índice do ponto corrente na lista_coordenadas e usando esse índice para encontrar o nome correspondente na lista_pontos. Finalmente, o código imprime os pontos da rota e a sua distância total.

```
# Caminho mínimo dentre as rotas para efetuar a entrega
distancias <- calcular_distancias(lista_coordenadas, ponto_R)
escreva("\n*** Caminho mínimo ***")
para cada distancia na lista distancias com indice i, faça
    se distancia == min(distâncias)
        nomes_pontos <- []
        pontos_permutados <- permute(lista_coordenadas, 0,
len(lista_coordenadas))[i]
        para cada ponto em pontos_permutados faça
           nomes_pontos.append(lista_pontos[lista_coordenadas.index
(ponto)])
        fim-para
        print(f"Rota {i+1}:{nomes_pontos} Distância:{distancia}")
    fim-se
fim-para
```

Figura 8. Pseudocódigo que busca o caminho mínimo das rotas de entrega

```
*** Caminho mínimo ***
Rota 7: ['A', 'D', 'C', 'B']  Distância: 14
Rota 22: ['B', 'C', 'D', 'A']  Distância: 14
```

Figura 9. Saída do programa FlyFood

4.2 Análise do algoritmo de força bruta

A análise de algoritmo é importante para estipularmos o gráfico de crescimento, de acordo com a entrada. No desenvolvimento do algoritmo, utilizamos da recursão e

busca sequencial onde foi verificado que um número grande na entrada resulta em processamento maior dos dados.

A função *permute* é a implementação de uma rotina de permutação. Ela gera todas as permutações possíveis de uma lista de coordenadas (representadas como tuplas com coordenadas x e y). Ela faz isso usando recursão com uma técnica de permutação por troca, também conhecida como backtracking.

Uma desvantagem da recursão no código é que ele pode ser propenso ao uso excessivo da pilha de chamadas, resultando em um consumo excessivo de memória e, possivelmente, em um erro de pilha cheia se o tamanho da lista for muito grande. Além disso, o tempo de execução da função *permute* pode ser longo para listas muito grandes, pois o número de chamadas recursivas é proporcional ao fatorial do tamanho da lista.

Uma maneira de melhorar o desempenho seria usar uma abordagem iterativa em vez de recursiva para gerar as permutações. Por exemplo, o algoritmo de Heap para gerar todas as permutações de uma lista. Esse algoritmo é mais eficiente do que a abordagem recursiva usada no código, pois ele não requer a utilização da pilha de chamadas.

A busca sequencial é uma técnica de busca onde os elementos são percorridos em ordem sequencial, verificando cada um até encontrar a solução desejada. Acaba não sendo uma solução ideal em alguns casos, como é o caso do nosso programa, pois, a complexidade temporal do algoritmo de força bruta é de O(n!), dessa forma existem $n!$ maneiras de visitar n pontos. Isso significa que, o tempo de execução do algoritmo aumenta exponencialmente com o número de pontos de entrega, tornando inviável para problemas com um grande número de cidades.

Tal busca sequencial foi utilizada na função *calcular_distancias,* onde recebe uma lista de coordenadas e um ponto "R" e calcula a distância total entre todos os pontos, incluindo o ponto R. Ela primeiro usa a função permute para gerar todas as permutações de coordenadas, e então calcula a distância total de cada uma dessas rotas. A distância é calculada como a soma das distâncias absolutas entre dois pontos consecutivos.

Existem outras alternativas como a busca binária, que costuma ser melhor, porém nesse caso, não fazemos muitas pesquisas e o custo de ordenação da lista para fazer tal busca seria maior que o uso da busca sequencial. Além disso, existe o algoritmo de programação linear e o algoritmo genético, que têm complexidade temporal menor e são mais adequados para problemas de grande escala. Esses algoritmos são baseados em técnicas de aprendizado de máquina e busca heurística, o que os torna mais flexíveis e capazes de lidar com problemas mais complexos.

4.3 Construção do Algoritmo Genético

O código começa abrindo e lendo um arquivo txt, que contém a matriz do flyfood. A primeira linha é ignorada e as outras linhas são lidas e armazenadas em uma lista de listas chamada "matrix".

```
1   import random, time
2
3   # Inicializando meu programa e obtendo a minha matriz:
4
5   with open("matriz", "r") as f:
6       linhas = f.readlines()[1:]
7   matrix = []
8   for linha in linhas:
9       linha = linha.strip().split()
10      matrix.append(linha)
```

Figura 10. Entrada da matriz

A lista "pontos" é criada para armazenar os valores diferentes de zero na matriz "matrix". Isso é feito usando dois loops for para percorrer cada elemento da matriz e, se o valor for diferente de zero, adiciona o valor à lista "pontos".

```
13  # Cria a lista com os pontos diferentes de zero:
14  pontos = []
15  for i in range(0, len(matrix)):
16      for j in range(0, len(matrix[0])):
17          if str(matrix[i][j]) != "0":
18              pontos.append(matrix[i][j])
19  pontos.remove("R")
```

Figura 11. Criação da lista que recebe os pontos de entrega

Os parâmetros do algoritmo genético são definidos, incluindo o tamanho da população, o tamanho da elite, a taxa de mutação e o número de gerações.

```
21  # Define os parâmetros do algoritmo genético
22  tamanho_populacao = 100
23  tamanho_elite = 10
24  taxa_mutacao = 0.1
25  geracoes = 200
```

Figura 12. Definição dos parâmetros

A função "fitness" recebe um indivíduo (uma lista de índices de "pontos") e calcula a distância percorrida ao percorrer todos os pontos na ordem dada pelo

indivíduo, adicionando a distância entre cada par de pontos adjacentes. A função retorna a distância total percorrida.

```python
def fitness(individuo):
    lista = ["R"] + [pontos[i] for i in individuo] + ["R"]
    distancia = 0
    for i in range(0, len(lista) - 1):
        ponto1 = lista[i]  # primeira coordenada
        ponto2 = lista[i + 1]  # segunda coordenada
        x1, y1 = None, None
        x2, y2 = None, None
        for indice, linha_matriz in enumerate(
            matrix
        ):  # vai iterar sob cada linha da matriz com seu indice
            if ponto1 in linha_matriz:
                x1 = indice
                y1 = linha_matriz.index(ponto1)
            if ponto2 in linha_matriz:
                x2 = indice
                y2 = linha_matriz.index(ponto2)
            if x1 is not None and y2 is not None:
                break
        distancia += abs(x1 - x2) + abs(y1 - y2)  # distancia de manhattan
    return distancia
```

Figura 13. Função que calcula a distância percorrida em um indivíduo.

A função "selection" recebe a população atual e retorna uma nova população selecionada com base no método de seleção de torneio. Os primeiros "tamanho_elite" indivíduos da população atual são selecionados diretamente para a nova população. Para os demais indivíduos, são realizados torneios entre "tamanho_torneio" indivíduos selecionados aleatoriamente da população atual, e o melhor indivíduo é selecionado para a nova população.

```python
def selection(populacao):
    tamanho_torneio = 5
    selecionados = []
    for i in range(tamanho_elite):
        selecionados.append(populacao[i])
    for i in range(tamanho_populacao - tamanho_elite):
        torneio = random.sample(populacao, tamanho_torneio)
        melhor = min(torneio, key=lambda x: fitness(x))
        selecionados.append(melhor)
    return selecionados
```

Figura 14. Função que seleciona indivíduos por torneio.

A função "crossover" recebe dois pais (indivíduos) e realiza o crossover para gerar um filho. O crossover é realizado escolhendo dois pontos de corte aleatórios nos pais e trocando as informações entre esses pontos. Em seguida, as informações restantes são adicionadas ao filho na ordem em que aparecem no segundo pai.

```
75  def crossover(pai1, pai2):
76      ponto_corte1 = random.randint(0, len(pai1) - 1)
77      ponto_corte2 = random.randint(0, len(pai1) - 1)
78      if ponto_corte1 > ponto_corte2:
79          ponto_corte1, ponto_corte2 = ponto_corte2, ponto_corte1
80      filho = pai1[ponto_corte1:ponto_corte2]
81      for i in pai2:
82          if i not in filho:
83              filho.append(i)
84      return filho
```

Figura 15. Função que realiza o crossover entre dois pais para gerar um filho

A função "mutation" recebe um indivíduo e realiza a mutação com uma taxa de mutação. Para cada elemento do indivíduo, há uma chance igual a "taxa_mutacao" de ser trocado com outro elemento aleatório.

```
88  def mutation(individuo):
89      for i in range(len(individuo)):
90          if random.random() < taxa_mutacao:
91              j = random.randint(0, len(individuo) - 1)
92              individuo[i], individuo[j] = individuo[j], individuo[i]
93      return individuo
```

Figura 16. Função que realiza a mutação em um indivíduo.

A população inicial é criada como uma lista de "populacao" indivíduos. Cada indivíduo é uma lista de índices de "pontos", que é embaralhada aleatoriamente.

```
97  def population(tamanho_populacao, pontos):
98      populacao = []
99      for i in range(tamanho_populacao):
100         individuo = list(range(len(pontos)))
101         random.shuffle(individuo)  # embaralha ale
102         populacao.append(individuo)
103     return populacao
```

Figura 17. Função que cria população inicial aleatoriamente

A função “algoritmo_genetico” necessita de 3 parâmetros onde o valor de “gerações” é a quantidade de vezes que o algoritmo será executado. A cada geração, a função "selection" é usada para selecionar uma nova população. Em seguida, a nova população é criada por meio de crossover e mutação dos indivíduos selecionados. A nova população é então usada na próxima iteração do algoritmo.

```
108  def algoritmo_genetico(geracoes, tamanho_populacao, pontos):
109      for i in range(geracoes):
110          populacao = selection(population(tamanho_populacao, pontos))
111          nova_populacao = []
112          while len(nova_populacao) < tamanho_populacao:
113              pai1 = random.choice(populacao)
114              pai2 = random.choice(populacao)
115              filho = crossover(pai1, pai2)
116              filho = mutation(filho)
117              nova_populacao.append(filho)
118          populacao = nova_populacao
119      return populacao
```

Figura 18. Execução do algoritmo genético

Após a última geração, o melhor indivíduo é selecionado da população final usando a função "min"

```
124 melhor_individuo = min(
125     algoritmo_genetico(geracoes, tamanho_populacao, pontos), key=lambda x: fitness(x)
126 )
```

Figura 19. Seleção do melhor indivíduo da população final

Por fim, imprime o resultado da melhor rota com sua respectiva distância e tempo de execução.

```
127  # Imprime o resultado
128  rota = [pontos[i] for i in melhor_individuo]
129  distancia = fitness(melhor_individuo)
130  end = time.time()
131  print("Melhor caminho: ", "-".join(rota))
132  print("Distância total: ", distancia)
133  print("Tempo de execução: ", end - start)
```

Figura 20. Comando para imprimir a rota com menor distância

```
Melhor caminho:  B-C-D-A
Distância total:  14
Tempo de execução:  1.283177137374878
```

Figura 21. Saída do programa FlyFood

4.4 Análise do Algoritmo Genético

O algoritmo implementado é um algoritmo genético para resolver um problema de caminho mínimo em uma matriz com pontos marcados. É uma técnica de busca baseada em evolução que simula o processo de seleção natural. Ele utiliza uma população de soluções candidatas, que evoluem ao longo de várias gerações, e usa operadores de seleção, cruzamento e mutação para gerar novas soluções.

Uma das vantagens do algoritmo genético é que ele é capaz de encontrar soluções ótimas ou próximas do ótimo em problemas complexos. Além disso, o algoritmo é fácil de implementar e pode lidar com uma grande variedade de problemas de otimização.

No entanto, o algoritmo também apresenta algumas desvantagens. Uma delas é que o processo de evolução pode levar muito tempo, dependendo do tamanho da população e do número de gerações. Além disso, os resultados obtidos pelo algoritmo genético não são garantidos, ou seja, ele pode não encontrar a solução ótima em todos os casos.

O algoritmo implementado utiliza as seguintes funções:

- A função fitness calcula o valor de fitness de um indivíduo da população, que é a distância total percorrida pelo caminho que ele representa. Essa função é uma função de avaliação adequada para o problema em questão, pois mede o desempenho do indivíduo de acordo com o objetivo do problema.

- A função selection é responsável por selecionar os indivíduos da população que irão gerar os descendentes na próxima geração. Ela utiliza o método de torneio para selecionar os indivíduos, o que pode levar a uma boa diversidade genética na população.

- A função crossover é responsável por criar um novo indivíduo a partir de dois indivíduos selecionados da população atual. Ela utiliza o método de crossover de pontos aleatórios, que é um método simples, mas eficaz para criar novos indivíduos com características de ambos os pais.

- A função mutation é responsável por introduzir variação genética na população, permitindo que os indivíduos evoluam para soluções melhores. Ela utiliza o método de mutação de permutação, que é um método simples, mas eficaz para criar variação genética na população.

- A função selection, crossover e mutation são combinadas para formar o loop principal do algoritmo genético, que executa o processo de evolução da população.

Uma das vantagens dessas funções é que elas são simples e fáceis de entender. Além disso, elas são altamente modulares, o que significa que podem ser facilmente adaptadas para outros problemas de otimização. Porém, também apresentam algumas limitações como por exemplo, o método de seleção de torneio pode não selecionar sempre os melhores indivíduos, o método de crossover de pontos aleatórios pode levar a soluções subótimas e o método de mutação de permutação pode introduzir mutações que não levam a melhorias.

Em resumo, o algoritmo genético é uma técnica poderosa para resolver problemas de otimização combinatória. No entanto, o sucesso dele depende da escolha cuidadosa dos parâmetros de entrada, o que pode levar muito tempo para encontrar a solução ideal.

5. Experimentos

Para testar a eficiência do algoritmo construído, conduzimos experimentos para verificarmos o tempo levado para processar a matriz e calcular a rota de distância mínima, levando em conta o número de pontos de entrega e as dimensões da matriz.

Número de pontos de entrega	Tamanho da matriz	Menor distância (em dronômetros)	Distância do melhor indivíduo - AG	Tempo de execução força bruta	Tempo de execução algoritmo genético
4	4x5	14	14	0.000971	0.81996
5	4x5	16	16	0.009004	0.882999
7	6x7	24	24	0.429994	1.519031
8	7x8	28	28	20.49354	1.97299
10	10x12	-	52	>10 min	3.25859
14	16x22	-	102	>10 min	14.59041

Tabela 2. Comparações de experimentos com o algoritmo de força bruta

Primeiramente, a tabela 2 exemplifica o número de pontos de entrada, as matrizes de entrada, a distância percorrida no menor caminho e o tempo de execução

tanto do algoritmo de força bruta como do genético. Sendo assim, com esses dados percebemos que o algoritmo de força bruta só é eficiente com poucos pontos de entrega, que no caso a máquina utilizada (Notebook com processador Intel Core i5-10500 H, de 10ª geração, com 12GB de memória RAM e com armazenamento SSD.) conseguiu rodar de forma eficiente com no máximo 8 pontos de entrega. Logo, para casos de solução com poucos pontos de entrega, o algoritmo de força bruta encontra uma solução ótima, porém, para mais pontos de entrega o tempo de execução ultrapassa os 10 minutos. É válido ressaltar que seu desempenho pode variar, de acordo com as configurações da máquina utilizada.

Já o algoritmo genético apresentou uma boa performance para todos os casos testados, encontrando soluções ótimas ou muito próximas do ótimo em um tempo consideravelmente menor do que o algoritmo de força bruta. Mesmo com um aumento no número de pontos de entrega, o algoritmo genético conseguiu manter um tempo de execução razoável, com no máximo 14 segundos para o exemplo com 14 pontos de entrega.

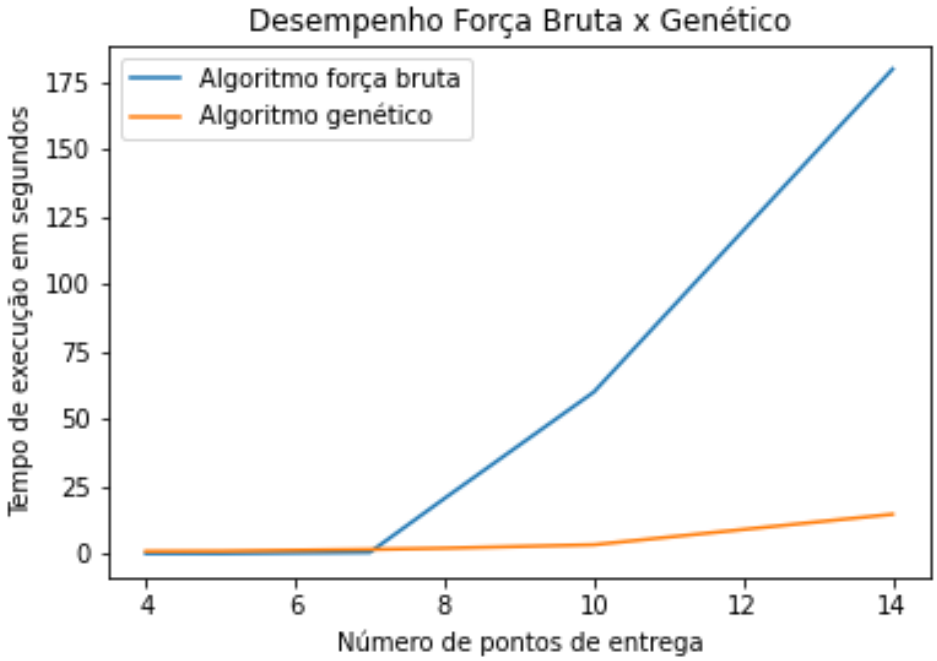

Figura 22. Tempo de execução força bruta x algoritmo genético

Ao analisarmos o gráfico do tempo de execução (figura 22) feito no colab, leva-se em conta a quantidade de pontos e o tempo levado para calcular a rota. Com tais informações, é importante perceber a diferença de tempo de execução do algoritmo de força bruta de uma matriz com 10 pontos de entrega, tendo em consideração um tempo aproximado, pois ultrapassou os 10 minutos e o algoritmo não obteve retorno, pois utilizou-se mais memória do que o equipamento poderia oferecer e o algoritmo

genético levava apenas 3 segundos para retornar a resposta. Logo, quando analisamos a eficiência do algoritmo de força bruta levando em conta o espaço e o tempo gasto, 10 seria o ponto que ele se tornaria infrutífero.

6. Conclusão

O objetivo principal deste trabalho foi desenvolver uma solução computacional que fosse capaz de encontrar o percurso com menor distância total que, saindo do ponto inicial, passe por todos os pontos de entrega e retorne ao ponto inicial. Então, foi elaborado um algoritmo de força bruta que calcule todas as rotas possíveis de entrega e retorne o caminho que obtivesse a menor distância.

Com o algoritmo implementado, foram feitos testes para calcular o tempo de execução para um valor x de pontos de entrega. Tais experimentos revelaram que o algoritmo conseguiu calcular rotas com até oito pontos de entrega, entretanto, para pontos maiores que 8 levava mais de 10 minutos, que foi considerado longo demais para ser produtivo. Logo, para um número mais razoável de pontos de entrega, consegue retornar a rota de menor distância e reduzir custos, visto que percursos maiores significam mais bateria e combustível gastos.

Entretanto, trouxemos também uma solução de caso onde o problema exige muitos pontos de entrega e o algoritmo genético apresentou uma boa performance para todos os casos testados, encontrando soluções próximas do ótimo em um tempo razoável,o que não é possível com o algoritmo de força bruta.

Analisou-se também que o algoritmo genético apresenta algumas desvantagens. A primeira é a dificuldade de se escolher os parâmetros ideais, como o tamanho da população e a taxa de mutação, o que pode impactar a qualidade da solução encontrada. Além disso, pode ficar preso em mínimos locais, o que pode levar a soluções sub-ótimas. Por fim, o algoritmo genético não garante a solução ótima, embora apresente soluções próximas do ótimo em um tempo razoável.

Em resumo, a solução proposta neste trabalho foi eficiente e conseguiu resolver o problema de encontrar a rota mais curta que passasse por todos os pontos de entrega. A combinação de algoritmos de força bruta e genética permitiu encontrar soluções para problemas de diferentes tamanhos e complexidades, tornando o processo mais eficiente e produtivo.

Referências Bibliográficas

Fonseca, J. D. O., Sá, E. M., Mendonça, F. M., & Sanches Junior, P. F. (2020). Otimização de rotas de entregas de materiais em uma rede hospitalar por meio do algoritmo do problema do caixeiro viajante. Rev. Gest. Sist. Saúde, São Paulo, 9(2), 283-302. https://doi.org/10.5585/rgss.v9i2.16570

CORMEN, Thomas. Algoritmos - Teoria e Prática. [Brasil]: Grupo GEN, 2012. 9788595158092. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595158092/. Acesso em: 27 jan. 2023.

FILEOFF, Paulo. Problemas NP-completos. Instituto de Matemática e Estatística da USP, São Paulo. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html. Acesso em 27 jan. 2023.

ANÁLISE de Complexidade. UFMG. Disponível em: https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/complexity.pdf Acesso em 27 de jan. de 2023.

DE ABREU, Nair Maria Maia. A Teoria da Complexidade Computacional. R. mil. Cio e Tecno/., Rio de Janeiro, 4(1):90-95, jan./mar. 1987. Disponível em: http://rmct.ime.eb.br/arquivos/RMCT_1_tri_1987/teoria_complex_comput.pdf Acesso em: 27 de jan. de 2023.

Gabriel Altafini Neves da Silva, et al. “ALGORITMOS HEURÍSTICOS CONSTRUTIVOS APLICADOS AO PROBLEMA DO CAIXEIRO VIAJANTE PARA A DEFINIÇÃO DE ROTAS OTIMIZADAS.” Colloquium Exactarum (Online), vol. 5, no. 2, 2013, pp. Colloquium Exactarum (Online), 2013, Vol.5 (2).

GOLDBARG, Elizabeth. Otimização Combinatória e Meta-heurísticas - Algoritmos e Aplicações. Grupo GEN, 2015. E-book. ISBN 9788595154667. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595154667/. Acesso em: 15 mar. 2023.

GENDREAU, Michel., POTVIN, Jean-Yves. Handbook of Metaheuristics. Springer, 2019. ISBN 978-3-319-91085-7. Acesso em: 15 mar. 2023.