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

Projeto FlyFood

Gustavo de Melo Moreira
João Vitor Soares da Silva
Lucas Henrique Cavalcanti de Melo
Luís Henrique Gonçalves da Silva
Maycon Romario dos Santos Pereira

Recife
Setembro de 2023

Resumo

Problemas de otimização são comuns no mundo real, contudo boa parte destes problemas são difíceis de solucionar na teoria, pois se classificam como problemas NP-Completos: um exemplo é o clássico “the traveling salesman”, ou Caixeiro Viajante. Nesse sentido, neste trabalho foram abordados três tipos de algoritmos para a resolução dessa classe de problemas: algoritmo de força bruta, algoritmo genético e algoritmo de colônia de formiga. Foram utilizados métodos convencionais para a elaboração do algoritmo de força bruta e metaheurísticas. Os resultados da comparação entre os três tipos de algoritmos foi satisfatório para as instâncias utilizadas, ressaltando que o uso de metaheurísticas é o meio mais viável nestes casos.

Palavras-chave: caixeiro viajante, força bruta, algoritmo genético, algoritmo colônia de formiga, tempo polinomial, tempo exponencial, algoritmos determinísticos, algoritmos não-determinísticos.

1. Introdução

1.1 Apresentação e Motivação

O projeto está focado no setor de delivery e logística, onde a empresa busca resolver os desafios enfrentados pelo trânsito caótico e o tempo excessivo necessário para realizar as entregas. Para contornar essas dificuldades, a empresa FlyFlood propõe o uso de drones para realizar as entregas, oferecendo uma alternativa eficiente e rápida.
Uma das principais preocupações do projeto é a limitação das baterias dos drones, que impõe restrições de tempo às operações de entrega. Isso significa que o projeto precisa levar em consideração a duração da bateria de cada drone ao calcular as rotas, garantindo que as entregas possam ser realizadas dentro do tempo de voo disponível. Para atingir esse objetivo, é necessário desenvolver o melhor algoritmo para calcular as rotas que os drones devem seguir.

Ao calcular as rotas de forma eficiente, o projeto visa otimizar o tempo de voo dos drones, minimizando o tempo gasto em viagens desnecessárias ou trajetos ineficientes. Isso resultará em um aumento da eficiência das entregas, redução dos atrasos e uma melhoria geral na qualidade do atendimento prestado aos clientes.

1.2 Formulação do problema

A formulação do problema consiste em encontrar o menor percurso em um grid retangular, visitando uma série de pontos de entrega a partir de um ponto de partida. Para isso, consideramos um conjunto de pontos P = {P1, P2, P3, ..., Pn}, onde cada ponto representa um ponto de entrega no grid. O objetivo é encontrar a ordem de visita aos pontos de entrega que resulta no menor custo total do percurso:

$$Pn = (P1,\ P2,\ P3,\ \dots,\ Pn)$$

O grid é composto por células que podem conter blocos ou estar vazias. A matriz de entrada descreve o estado dos blocos no grid, onde cada célula da matriz representa uma célula do grid. O valor de uma célula na matriz é 0 se contém um bloco, indicando que não pode ser percorrida, e 1 caso contrário, indicando que está livre para visita.

A distância entre dois pontos, calculada utilizando a fórmula da distância Euclidiana, determina a distância em linha reta entre dois pontos em um espaço bidimensional. A fórmula é aplicada para calcular a distância entre os pontos no grid, considerando suas coordenadas x e y. A equação da distância Euclidiana é:

$$distanciaeuclidiana = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

O custo de uma solução é determinado pela soma das distâncias percorridas ao longo do percurso. O custo entre dois pontos consecutivos é calculado utilizando a fórmula da distância Euclidiana. Para uma sequência de n pontos de entrega na ordem de visita P = {P1, P2, P3, ..., Pn}, o custo total é calculado somando as distâncias Euclidianas entre os pontos consecutivos:

$$custo = distanciaeuclidiana(P_1, P_2) + \dots + distanciaeuclidiana(P_n - 1,\ P_n)$$

A equação do problema envolve encontrar a ordem de visita aos pontos de entrega que resulta no menor custo total do percurso. Podemos representar essa busca pelo menor custo como:

$$resposta1 = argmin\ f(retorna\ o\ argumento\ x\ que\ gera\ o\ menor\ f) = 0$$

Essa equação retorna o argumento P que gera o menor custo. Além disso, podemos determinar o menor custo diretamente utilizando a equação:

$$resposta2 = min\ f(retornaomenorf) = 1$$

Isso nos permite encontrar tanto a ordem de visita que resulta no menor custo quanto o valor mínimo do custo alcançado. Em resumo, a “resposta1" indica a sequência de pontos que gera o menor custo, enquanto a "resposta2" indica o valor mínimo do custo encontrado.

Por fim, o objetivo é determinar a sequência de visitas aos pontos de entrega que minimize a distância total percorrida no grid, levando em consideração a fórmula da distância Euclidiana. Isso nos permite encontrar a rota mais eficiente para entregar todos os pontos no menor percurso possível.

1.3 Objetivos

Os objetivos gerais do trabalho são a otimização das entregas, a redução dos custos operacionais, a melhoria da qualidade do serviço, a minimização do impacto no trânsito, a busca por inovação tecnológica e a viabilização de entregas em áreas de difícil acesso. Através do uso de drones e do algoritmo de roteamento, o projeto almeja alcançar essas metas e promover uma revolução no setor de entregas, tornando-o mais eficiente, ágil e sustentável..

Os objetivos específicos são, realizar uma pesquisa abrangente sobre o problema do caixeiro viajante, estudando as abordagens existentes, os algoritmos clássicos e as técnicas de otimização combinatória utilizadas para solucioná-lo. Serão conduzidos experimentos e simulações para avaliar a eficiência e a precisão do algoritmo de roteamento proposto, comparando-o com outras abordagens existentes, desenvolvemos um modelo matemático que represente o problema de roteamento de drones, será detalhada a etapa de desenvolvimento do algoritmo de roteamento, explicando as decisões tomadas, os critérios considerados e os passos seguidos para a criação do algoritmo, de maneira a tornar o processo compreensível para o público.

1.4 Organização do trabalho

A organização deste trabalho é dividida em quatro capítulos, para uma melhor apresentação e compreensão do conteúdo. Introdução, onde ocorre a contextualização e motivação, formulação dos objetivos e alocação dos problemas. Referências teóricas, com exploração dos estudos e trabalhos com relação ao tema, em conjunto com análises de abordagem e soluções. Metodologia e Implementação, dando ênfase à descrição da metodologia para resolução, explicação, técnica e análise gráfica. Resultado e conclusão, apresentando resultados e analisando os mesmos, discutindo possíveis melhorias e conclusões gerais. Essa estrutura permite uma abordagem completa do trabalho, desde a introdução até a conclusão, abrangendo o referencial teórico, a metodologia utilizada e a apresentação dos resultados obtidos.

2. Referencial Teórico

2.1 P vs NP

As classes de problemas oferecem uma maneira organizada de compreender a complexidade e a dificuldade dos problemas computacionais. A classe P (Problems/Polly) engloba problemas que podem ser resolvidos em tempo polinomial por algoritmos determinísticos eficientes, como ordenação e busca em estruturas de dados.

Por outro lado, a classe NP (Nondeterministic Polynomial) abrange problemas cujas soluções podem ser verificadas em tempo polinomial, mas ainda não se sabe se podem ser resolvidos eficientemente. Um exemplo notável é o Problema do Subconjunto, que busca determinar se existe um subconjunto que some um determinado valor em um conjunto de números.

Dentro da classe NP, existe uma subclasse particularmente desafiadora: a classe NP-Completo (Nondeterministic Polynomial Complete). Essa classe contém problemas para os quais ainda não se encontrou uma solução polinomial eficiente. Um exemplo clássico de problema NP-completo é o Problema da Mochila, onde se busca encontrar a combinação mais valiosa de itens que possa ser colocada em uma mochila com uma capacidade limitada.

Além disso, existem problemas que são tão difíceis quanto os NP-completos, mas podem não estar em NP. Essa classe é conhecida como NP-Difícil (Nondeterministic Polynomial Hard). Problemas NP-difíceis compartilham da complexidade dos NP-completos, mas podem não ter a propriedade de verificação em tempo polinomial. Um exemplo é o Problema do Halting, que busca determinar se um programa termina ou entra em um loop infinito.

2.1.1 Diferença entre NP-Completo e NP-Difícil

Problema NP-Difícil:

Um problema X é NP-Difícil se existir um problema NP-Completo Y, tal que Y é redutível a X em tempo polinomial. Os problemas NP-Difíceis são tão difíceis como os problemas NP-Completos. Um problema NP-Difícil não precisa de pertencer à classe NP. Se todos os problemas de NP puderem ser reduzidos em tempo polinomial, chama-se NP Difícil. Muitas vezes é preciso resolver um problema específico e reduzir problemas diferentes.

Exemplos:
Ciclo Hamiltoniano.
Problema de otimização.
Problema do caminho mais curto.

Problema NP-Completo:

Um problema X é NP-Completo se existir um problema NP Y, tal que Y é redutível a X em tempo polinomial. Os problemas NP-Completos são tão difíceis como os problemas NP. Um problema é NP-Completo se fizer parte tanto de um problema NP como de um problema NP-Difícil. Uma máquina de Turing não-determinística pode resolver um problema NP-Completo em tempo polinomial. Um problema é np-completo quando é simultaneamente np e np-difícil.Isto significa que os problemas np-completos podem ser verificados em tempo polinomial.

Exemplos:
Problemas de decisão.
Grafos regulares.

Diferença entre NP-Difícil e NP-Completo:

NP-Difícil	NP-Completo
Os problemas NP-Difíceis (digamos X) podem ser resolvidos se e só se existir um problema NP-Completo (digamos Y) que possa ser redutível a X em tempo polinomial.	Os problemas NP-Completos podem ser resolvidos por um Algoritmo não determinístico/Máquina de Turing em tempo polinomial.
Para resolver este problema, não é necessário que ele esteja em NP.	Para resolver este problema, é necessário que se trate de um problema NP e NP-Difícil.
O tempo é desconhecido em NP-Difícil.	O tempo é conhecido porque é fixo em NP-Difícil.
NP-difícil não é um problema de decisão.	NP-Completo é exclusivamente um problema de decisão.
Nem todos os problemas NP-difíceis são NP-completos.	Todos os problemas NP-completos são NP-difíceis
Não tem que ser um problema de decisão.	Trata-se exclusivamente de um problema de decisão.
É um problema de otimização utilizada.	É um problema da Decisão utilizada.
Exemplos: Problema de paragem, problema de cobertura de vértices, etc.	Exemplos: Determinar se um grafo tem um ciclo hamiltoniano, Determinar se uma fórmula booleana é ou não satisfazível, Problema de satisfatibilidade de circuitos, etc.

2.1.2 Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV) é desafiador e envolve encontrar a rota mais curta que visita todos os destinos uma vez e retorna ao ponto de partida. Algoritmos exatos baseados em Programação Inteira podem resolver o PCV, mas são computacionalmente custosos e consomem muito tempo. Por isso, são utilizados métodos heurísticos, que oferecem soluções boas em menos tempo, embora não garantam a otimalidade.

Os métodos exatos possuem vantagens, como a capacidade de provar a otimalidade e obter informações sobre limites inferiores e superiores. No entanto, eles têm desvantagens, como alto custo computacional e consumo de memória, limitando o tamanho das instâncias que podem ser solucionadas.

Os métodos heurísticos são mais rápidos e podem produzir soluções boas o suficiente. Existem diferentes tipos de heurísticas, como construtivas, de melhoria iterativa e metaheurísticas. A escolha da heurística depende do tempo de execução desejado e da flexibilidade em relação à solução ótima.

Em resumo, o PCV é solucionado por métodos exatos, mas eles são demorados e caros. Métodos heurísticos fornecem soluções boas em menos tempo. A escolha da heurística depende do tempo de execução desejado e da flexibilidade em relação à solução ótima.

2.1.3 Prova de que o problema do caixeiro viajante é NP-Difícil

Dado um conjunto de cidades e a distância entre cada par de cidades, o problema do caixeiro-viajante encontra o caminho entre essas cidades tal que é o caminho mais curto e atravessa cada cidade uma vez, regressando ao ponto de partida.

Problema - Dado um grafo ***G(V, E)***, o problema é determinar se o grafo tem um PCV de custo no máximo ***K***.

Explicação - Para provar que o Problema do Caixeiro Viajante é NP-Difícil, teremos de reduzir um problema NP-Difícil conhecido a este problema. Vamos efetuar uma redução do problema do Ciclo Hamiltoniano para o problema do Caixeiro Viajante.

Cada instância do problema do Ciclo Hamiltoniano que consiste num grafo ***G =(V, E)*** como entrada pode ser convertida num problema do Caixeiro Viajante que consiste no grafo ***G' = (V', E')*** e no custo máximo, ***K***. Vamos construir o grafo G' da seguinte forma:

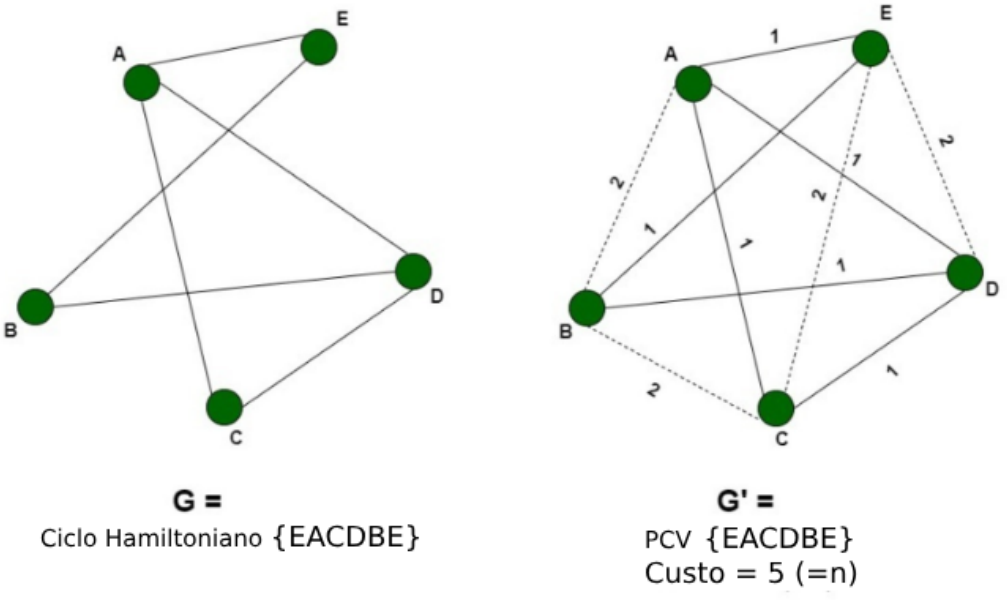
Para todas as arestas ***e*** pertencentes a ***E***, adicione o custo da aresta ***c(e)=1***. Ligue as restantes arestas, ***e'*** pertencentes a ***E'***, que não estão presentes no grafo original ***G***, cada uma com um custo ***c(e')= 2***.

E, por fim, defina ***K = N***.

O novo grafo ***G'*** pode ser construído em tempo polinomial, bastando converter ***G*** num grafo completo ***G'*** e adicionar os custos correspondentes. Esta redução pode ser provada pelas duas afirmações seguintes:

- Suponhamos que o grafo ***G*** contém um Ciclo Hamiltoniano, percorrendo todos os vértices ***V*** do grafo. Ora, estes vértices formam um PCV com ***custo = N***, pois utiliza todas as arestas do grafo original com custo ***c(e)=1***. E, como se trata de um ciclo, portanto, ele retorna ao vértice original.
- Assumimos que o grafo ***G'*** contém um PCV com custo, ***K = N***. O PCV percorre todos os vértices do grafo retornando ao vértice original. Ora, como nenhum dos vértices é excluído do grafo e o custo é igual a **n**, então, necessariamente, ele usa todas as arestas do grafo presentes em ***E***, com custo 1, formando assim um ciclo hamiltoniano com o grafo ***G***.

Figura 1. Ilustração do Ciclo Hamiltoniano e do Caixeiro-viajante.



Assim, podemos dizer que o grafo ***G'*** contém um PCV se o grafo ***G*** contém um ciclo hamiltoniano. Por conseguinte, qualquer instância do problema do caixeiro-viajante pode ser reduzida a uma instância do problema do ciclo hamiltoniano. Assim, o PCV é NP-Difícil.

2.2 Métodos exatos

Os algoritmos exatos são projetados para encontrar a solução ideal para o Problema do Caixeiro Viajante (PCV), que é o caminho com o menor comprimento. No entanto, esses algoritmos são computacionalmente caros, pois devem considerar todas as soluções possíveis para identificar o caminho ótimo. Resolver o PCV em um computador poderoso por um longo período de tempo pode não ser econômico, considerando que é possível encontrar uma solução próxima do ideal mais rapidamente em um computador menos potente. Por isso, os algoritmos heurísticos ou aproximados são frequentemente preferidos para resolver os grandes PCVs que ocorrem na prática.

Dentre os métodos exatos para solução do PCV encontram-se o Método de Força Bruta (FB), Método Branch & Bound e Branch & Cut. Na Tabela 1 é possível verificar que no caso de poucos pontos (vértices) o problema do caixeiro viajante não é tão complicado, porém, quando o problema tem mais pontos, a contagem dos caminhos possíveis já é um número enorme. No caso de 20 pontos é uma quantidade tão grande de cálculos que é impossível resolvê-lo em tempo real. Houve um grande número de estudiosos na história que tentaram encontrar um algoritmo de otimização universal, porém sem sucesso.

Contagem de pontos	Contagem de maneiras possíveis
4	24
8	40 320
12	479 001 600
16	20 922 789 888 000
20	2 432 902 008 176 640 000
25	15 511 210 043 330 985 984 000 000

Tabela 1. Contagem de permutações para resolver PCV com algoritmo exato. Adaptado de Štencek (2013)

Problemas que dependam da análise das permutações dos elementos de um conjunto podem exigir algoritmos exponenciais, onde o número de permutações de um conjunto com n elementos é n! (fatorial). Por outro lado, algoritmos que examinam os sucessores de cada vértice tomado como nova raiz da iteração podem ter uma complexidade quadrática. Portanto, a complexidade do algoritmo do problema do caminho mínimo a partir de um vértice é, no pior caso, da ordem de n^2, ou abreviadamente $O(n^2)$ [Boaventura Netto e Jurkiewicz 2009].

A Tabela 2 mostra as implicações da complexidade exponencial apresentando estimativas do tempo de computação para algoritmos de diversas complexidades, considerando-se uma máquina que consegue solucionar exatamente um algoritmo O(n) de um grafo com 10 vértices em 10^-6 segundos [Boaventura Netto e Jurkiewicz 2009].

n	O(n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	10^{-6} seg	10^{-4} seg	10^{-3} seg	10^{-3} seg
50	5.10^{-6} seg	$2,5.10^{-3}$ seg	0,125 seg	35,7 anos
100	10^{-5} seg	10^{-2} seg	1 seg	4×10^{14} séculos
500	5.10^{-5} seg	0,25 seg	2 min 25 seg	$1,03.10^{135}$ séculos

Tabela 2. Tempo de computação para algoritmos de diversas complexidades. [Boaventura Netto e Jurkiewicz 2009]

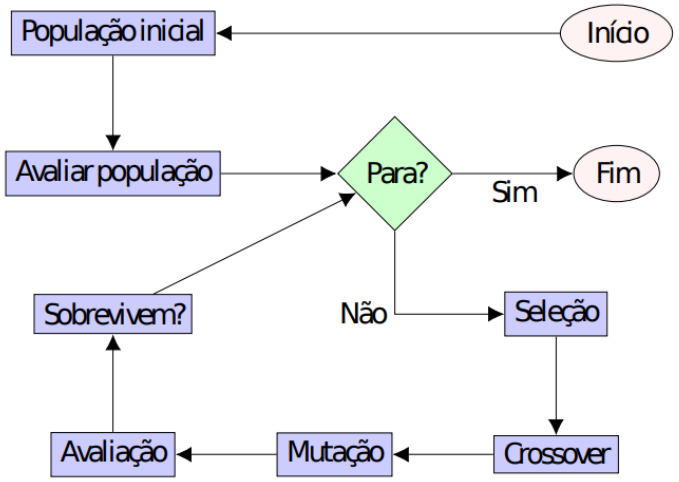
2.3 Algoritmos genéticos

Os **algoritmos genéticos (GAs: Genetic Algorithms)** são inspirados na teoria da evolução das espécies de Darwin e na genética. Se utilizam principalmente da probabilidade de um mecanismo de busca paralela, cuja característica é realizar diferentes cálculos no mesmo ou em diferentes conjuntos de dados e adaptá-los, baseado no princípio de sobrevivência dos mais aptos e na reprodução.

Estes princípios são imitados na construção de algoritmos computacionais que buscam uma melhor solução para um determinado problema. No caso do *problema do caixeiro viajante*, a partir de um certo número de cidades, a melhor solução se torna extremamente difícil de encontrar utilizando um algoritmo de *força bruta*, portanto, algoritmos adaptativos como os *GAs* é a opção mais adequada.

Os algoritmos genéticos possuem processos (representação, decodificação, avaliação, seleção, cruzamento, mutação, entre outros) e parâmetros fundamentais (tamanho da população, taxa de reprodução, taxa de mutação e critério de parada). O primeiro conceito é o de representação dos cromossomos, que são as possíveis soluções do problema e podem ser dos tipos: binária, números reais, permutação de símbolos e etc.

Figura 2. Fluxograma ilustrando o funcionamento básico do AG.



A ***decodificação*** consiste na construção de uma solução para que o programa o avalie. Neste projeto, é utilizado pontos distintos e suas respectivas posições. A partir dos pontos são criadas as permutações que serão utilizadas no processo posterior.

A ***avaliação*** se refere ao cálculo do fitness (aptidão) de cada indivíduo. No caso do caixeiro viajante, se trata da soma das distâncias dos caminhos de cada indivíduo. É a distância entre cada letra, calculada pela o formula matemática: dT = |x2-x1| + |y2-y1|. No exemplo:

$$Indivíduo \rightarrow B\ D\ C\ A \rightarrow 14$$

O processo de ***seleção*** em algoritmos genéticos seleciona indivíduos para a reprodução. E se baseia na escolha dos indivíduos que possuem os melhores fitness ou os indivíduos mais aptos, neste caso o indivíduo mais apto é o que possui o menor caminho.

Para a realização da seleção existem diferentes métodos de seleção: ***seleção uniforme (SU)***, ***seleção por torneio (ST)*** e ***seleção por roleta giratória (SRG)*** proposto no trabalho de *Timóteo (2002)*. Segundo *Simão (2013)*, na estratégia de seleção uniforme os indivíduos que participam da etapa de cruzamento são selecionados de forma arbitrária, sem levar em consideração o grau de aptidão. Já no método de torneio ocorre uma seleção aleatória de dois indivíduos e aquele que possuir maior grau de aptidão é selecionado. Por fim, segundo *Barcellos (2010)*, na estratégia de seleção por roleta giratória todo indivíduo possui uma chance de ser selecionado equivalente ao seu índice fitness.

O operador de ***cruzamento (crossover)*** é considerado a característica fundamental dos GAs e ocorre logo após a seleção dos melhores indivíduos. Pares de genitores são escolhidos aleatoriamente da população, baseado na aptidão, e novos indivíduos são criados a partir da troca do material genético. Os descendentes serão diferentes de seus pais, mas com características genéticas de ambos os genitores . Por exemplo:

$$Pai\ 1 \rightarrow B\ D\ C \mid A\ E\ F$$
$$Pai\ 2 \rightarrow \boldsymbol{A\ D\ B \mid E\ F\ C}$$

$$Filho\ 1 \rightarrow B\ D\ C\ \boldsymbol{E\ F\ C}$$
$$Filho\ 2 \rightarrow \boldsymbol{A\ D\ B}\ A\ E\ F$$

A maneira mais simples é escolher apenas um ponto de corte aleatório (one-point crossover) e dividir os progenitores em duas partes. A geração dos filhos acontece juntando as partes divididas dos progenitores.

Os cromossomos criados a partir do operador de crossover são então submetidos a operação de ***mutação*** (essa podendo ou não ocorrer dependendo da taxa de probabilidade de mutação). Mutação é um operador exploratório que tem por objetivo aumentar a diversidade na população. O operador de mutação neste projeto aconteceu selecionando dois pontos aleatórios do indivíduo filho e trocando suas posições de lugar.

Por fim, no algoritmo genético os parâmetros controlam o processo evolucionário:

- **Tamanho da População:** é o número de indivíduos que irão compor uma população.

- **Taxa de Crossover:** é a probabilidade de um indivíduo ser recombinado com outro. A taxa utilizada neste projeto foi de 60%, ou seja, apenas sessenta por cento dos indivíduos da população poderão ser selecionados. Com uma taxa muito alta pode ocorrer perda de estruturas de alta aptidão, mas com um valor baixo, o algoritmo pode tornar-se muito lento.
- **Taxa de Mutação:** probabilidade do conteúdo de uma posição/gene do cromossoma alterado. De modo geral, as taxas de mutação são muito baixas (<1%), pois taxas altas tornam as alterações essencialmente aleatórias.
- **Número de Gerações:** total de ciclos de evolução de um GAs, ou o número de vezes que o algoritmo irá realizar todos os processos descritos anteriormente até retornar a melhor solução encontrada.

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

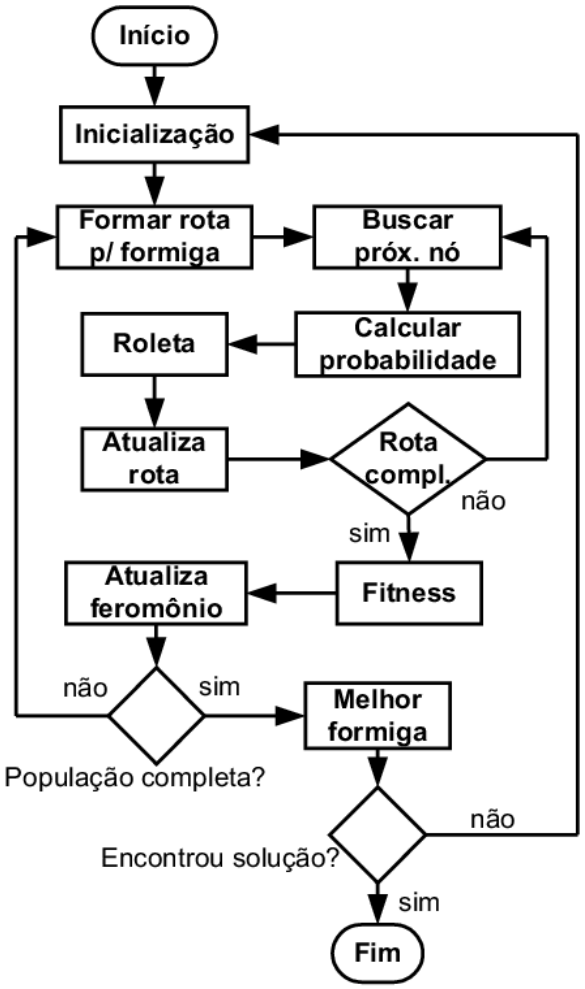
2.4 Algoritmo de Otimização por Colônia de Formigas

As formigas quando em busca de alimento exploram aleatoriamente o ambiente em torno da colônia de uma forma aparentemente desordenada. Ao percorrerem esse trajeto, as formigas liberam uma substância química chamada feromônio. Influenciadas pela presença de feromônios no caminho, as formigas tendem a seguir na direção em que a concentração de feromônios é mais forte.

Os experimentos com formigas reais demonstraram que essa coordenação entre formigas via trilhas de feromônio produz um comportamento coletivo de auto-organização, onde os caminhos mais curtos entre os seus ninhos e as fontes de alimento são 44 progressivamente seguidos pelas demais formigas ao reforçarem as trilhas de feromônio nas melhores rotas e, eventualmente, encontrarem o caminho mais curto. A Figura 4 mostra a tendência das formigas em seguir o caminho mais curto entre o ninho e a fonte de alimentos, em função da concentração de feromônios.

O Algoritmo Colônia de Formigas (ACO, do inglês Ant Colony Optimization) é uma meta-heurística bioinspirada, na qual formigas artificiais cooperam entre si, de forma a encontrar soluções ótimas para problemas difíceis de otimização combinatória.

Figura 3. Fluxograma ilustrando o funcionamento básico do ACO.

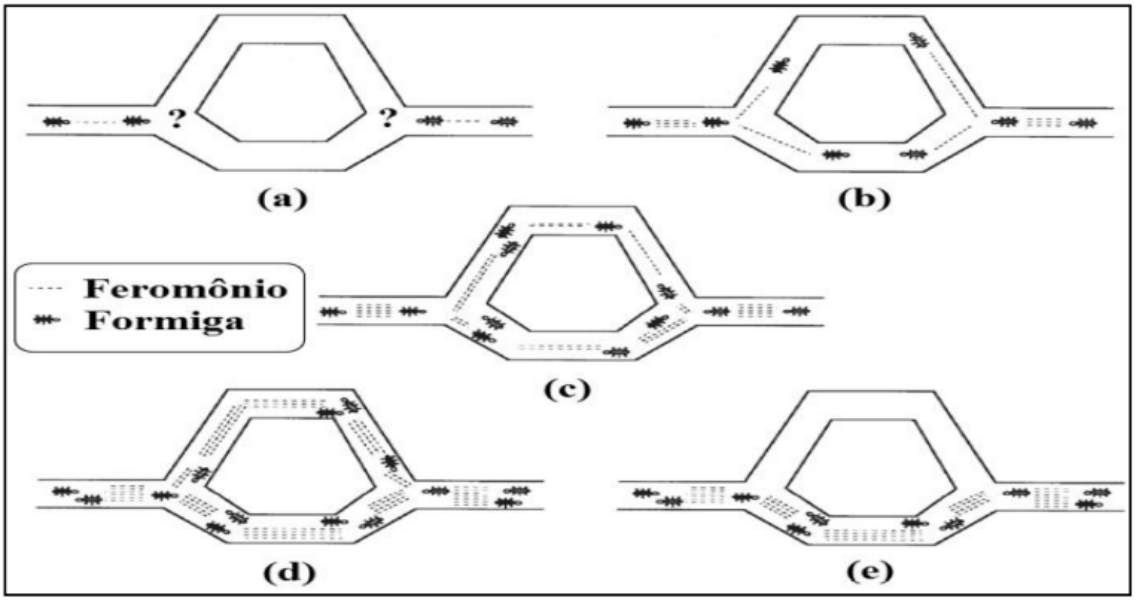


O primeiro Algoritmo baseado no comportamento das colônias de formigas é conhecido como Ant System (AS) e foi apresentado pela primeira vez por Colorni et al. (1991) aplicado na resolução de um problema do Caixeiro Viajante.

Além do problema do Caixeiro Viajante, o ACO tem sido aplicado na resolução de problemas diversos, tais como o processamento de imagens (HOSEINI; SHAYESTEH, 2010), problemas de carregamento (ZHANG; DU, 2011), problemas de corte (LEVINNE; DUCATELLE, 2003), dentre outros.

Como ilustra a Figura 3 abaixo, as formigas começam a percorrer os caminhos entre o ninho e a fonte de alimento aleatoriamente (a). À medida que percorrem os caminhos, liberam feromônios (b). A concentração de feromônios é maior nos caminhos por onde passam mais formigas (c). Assim, as formigas tendem a seguir o trajeto mais curto, ou seja, com a maior concentração de feromônios (d) e (e).

Figura 4. Tendência das formigas em seguir o trajeto mais curto em função da concentração de feromônios.



Fonte: Adaptado de Dorigo e Gambardella (1997).

O ACO pode ser representado como a ação combinada de três procedimentos (DORIGO; STÜTZLE, 2004, p.37):

- Construção das soluções: uma colônia de formigas é gerenciada de forma concorrente e assíncrona, construindo soluções incrementais para um determinado problema de otimização, a partir do uso das trilhas de feromônio e da informação heurística (que pode ser, por exemplo, a distância entre cidades adjacentes, no caso do problema do caixeiro viajante).

Cada formiga que constrói uma solução, ou enquanto uma solução é construída, avalia a solução (parcial) de forma a possibilitar que o procedimento seguinte decida a quantidade de feromônios a depositar.

- Atualização de Feromônios: é o procedimento a partir do qual as trilhas de feromônio são modificadas de forma incremental (quando as formigas depositam feromônio nas conexões usadas) ou decremental (devido à evaporação dos feromônios nas conexões menos utilizadas).

Do ponto de vista prático, o depósito de novos feromônios aumenta a probabilidade de uma conexão ser utilizada por um número maior de formigas, ou que seja utilizada por pelo menos uma formiga que produzirá uma boa solução, a qual será usada novamente por futuras formigas. Já a evaporação é uma forma bastante útil de esquecimento, evitando uma convergência prematura do algoritmo para uma região sub-ótima, favorecendo, assim, a exploração de novas áreas no espaço de busca.

- Processamento complementar: é utilizado para implementar ações centralizadas que não podem ser realizadas pelas formigas normais, como, por exemplo, a ativação de um procedimento de busca local.

A forma como estes três procedimentos irão interagir na implementação do ACO fica a cargo da natureza do problema ao qual ele será aplicado. Além do problema do caixeiro viajante, o ACO tem sido aplicado em diversos outros problemas de otimização combinatória, tais como roteamento de veículos (REIMANN et al., 2002), colorização de grafos (COSTA; HERTZ, 1997) e problemas de corte (LEVINNE; DUCATELLE, 2003).

3. Trabalhos relacionados

Na busca por um método de solução do PCV que pudesse ser utilizado neste projeto, foi realizada uma pesquisa de trabalhos que apresentassem comparações entre as técnicas para possibilitar o embasamento desta escolha.

No estudo realizado por Bispo (2018), foi abordado o problema da roteirização do turismo em Recife, utilizando o Problema do Caixeiro Viajante (PCV) em um aplicativo móvel. O autor testou três métodos para resolver esse problema: o algoritmo de Força Bruta (FB), o Vizinho Mais Próximo (VMP) e uma combinação do Vizinho Mais Próximo com a heurística 2-OPT. O algoritmo de Força Bruta mostrou uma tendência de aumento exponencial no tempo de execução à medida que o número de pontos aumentava. Ele se mostrou viável para até 8 pontos, com um tempo de espera razoável de 5 segundos. No entanto, quando testado com 11 pontos, o algoritmo se tornou impraticável devido ao alto consumo de memória e CPU, o que levou à interrupção da aplicação. Por outro lado, o método do Vizinho Mais Próximo foi capaz de encontrar rapidamente uma solução, mas à medida que novos pontos eram adicionados, o roteiro gerado se afastava cada vez mais da solução ótima. No entanto, o custo computacional em termos de memória e CPU foi menor em comparação com o algoritmo de Força Bruta. A heurística 2-OPT, quando combinada com o Vizinho Mais Próximo, apresentou uma melhoria significativa. Ela conseguiu reduzir o tempo da rota em até 50,4 minutos e melhorar a rota gerada pelo Vizinho Mais Próximo em até 20%. Os testes de CPU e memória foram realizados com 20 pontos turísticos cadastrados no aplicativo. Após a geração do roteiro, houve um aumento de 1,6 MB na memória. Apesar disso, o processo foi concluído em pouco mais de 1 segundo, tornando-se o método mais rápido entre os testados. Essas informações indicam que a combinação do Vizinho Mais Próximo com a heurística 2-OPT obteve um bom desempenho em termos de tempo de execução e melhoria da rota em comparação com os outros métodos testados.

O estudo de Hui (2012) realiza uma comparação entre o Algoritmo Genético (AG), a Rede Neural de Hopfield (RNH) e o Algoritmo de Colônia de Formigas (OCF) para abordar o Problema do Caixeiro Viajante (PCV). De acordo com os resultados encontrados, o algoritmo genético apresenta a maior complexidade de tempo, enquanto o algoritmo de colônia de formigas demonstra a menor complexidade temporal. No que diz respeito à complexidade espacial, o algoritmo genético é o que demanda maior espaço, diferentemente das redes de Hopfield e do algoritmo de colônia de formigas, que possuem complexidades espaciais semelhantes. Uma das vantagens do algoritmo genético é a possibilidade de escapar de ótimos locais devido aos operadores de crossover e mutação, permitindo a busca pelo ótimo global. Já os resultados da RNH e do OCF podem variar dependendo dos parâmetros utilizados, uma vez que eles afetam

diretamente a convergência do algoritmo e os prós e contras dos resultados obtidos. Esses dois algoritmos também são mais suscetíveis a ficarem presos em ótimos locais. Em termos de implementação, os três algoritmos foram codificados no MATLAB. O algoritmo genético possui 186 linhas de código, enquanto a RNH e o OCF possuem 93 e 96 linhas, respectivamente, evidenciando que o algoritmo genético é mais complexo de ser implementado. Quanto aos resultados obtidos, o algoritmo de colônia de formigas se aproxima mais da solução ótima, enquanto os resultados do algoritmo genético e da rede de Hopfield diferem significativamente do valor ótimo. Por fim, as pontuações consolidadas dos algoritmos genéticos, redes de Hopfield e algoritmos de colônia de formigas foram de 0,29, 0,18 e 0,53, respectivamente. Isso indica que o algoritmo de colônia de formigas obteve o melhor desempenho, seguido pelo algoritmo genético e, por último, a rede de Hopfield. Vale ressaltar que, além do estudo do Problema do Caixeiro Viajante (PCV), a aplicação dos algoritmos genéticos, redes neurais de Hopfield e algoritmos de colônia de formigas também pode ser explorada em outros desafios interessantes, como o intrigante problema do 'FlyFood'. Nesse contexto, busca-se otimizar as rotas de entrega de alimentos para as moscas da fruta em um laboratório de pesquisa, explorando as capacidades adaptativas dessas abordagens para obter soluções eficientes e sutilmente adaptadas a esse cenário peculiar.

No estudo conduzido por Halim e Ismail (2019), foram empregados 15 problemas de PCV para comparar o tempo de computação, o valor estatístico e a convergência de diferentes algoritmos, nomeadamente o Vizinho Mais Próximo (VMP), Algoritmo Genético (AG), Recozimento Simulado (RS), Busca Tabu (BT), Otimização de Colônia de Formigas (ACO) e Otimização de Fisiologia de Árvores (TPO). Com relação ao tempo de computação, observou-se que os métodos Busca Tabu e Otimização de Colônia de Formigas requerem, respectivamente, seis e três vezes mais tempo em comparação aos demais algoritmos. Em termos de velocidade de execução, os algoritmos Vizinho Mais Próximo, Otimização de Fisiologia de Árvores e Algoritmo Genético foram considerados os mais rápidos. No que se refere à análise estatística, os algoritmos Busca Tabu, Otimização de Fisiologia de Árvores e Algoritmo Genético aproximaram-se mais da rota ótima. Contudo, constatou-se que a precisão do Busca Tabu decai em problemas de maior dimensão. Em relação à convergência dinâmica, os pesquisadores observaram que os algoritmos que convergem mais rapidamente em 300 iterações são o Vizinho Mais Próximo, Busca Tabu e Otimização de Colônia de Formigas, ao passo que a Otimização de Fisiologia de Árvores apresenta melhor desempenho em problemas de menor escala. Além disso, foi relatado que o Recozimento Simulado possui uma taxa de convergência mais lenta quando comparado aos demais algoritmos. Além disso, é importante ressaltar que o estudo de Halim e Ismail (2019) focou na análise do Problema do Caixeiro Viajante (PCV) e em comparações entre diferentes algoritmos de otimização. No entanto, é válido mencionar que existem outros problemas relevantes no campo da otimização, como o desafiador "Problema do FlyFood", que envolve a otimização de rotas para entregas de alimentos em tempo real, considerando restrições logísticas e preferências dos clientes.

Esse problema apresenta um conjunto único de desafios que demandam abordagens específicas de otimização para encontrar soluções eficientes e satisfatórias.

Essas são apenas algumas das variantes do PCV que podem ser comparadas ao problema do Projeto FlyFood. Cada uma delas apresenta desafios e características específicas, e as abordagens de resolução podem ser adaptadas de acordo com as particularidades de cada problema.

4. Metodologia

A metodologia do Projeto FlyFood é fundamentada em uma abordagem algorítmica voltada para a solução de um desafio logístico crítico: encontrar o menor percurso em um mapa que apresenta diversos pontos de entrega e um ponto de origem. Buscando otimizar a rota de entrega percorrendo a menor distância possível, passando por todos os pontos de entrega e retornando ao ponto de partida.

Dessa forma, a metodologia do Projeto FlyFood pode ser organizada em etapas específicas, cada uma com um papel crucial no processo de resolução do desafio logístico. Essas etapas são cuidadosamente combinadas para garantir que o algoritmo encontre a melhor rota possível, maximizando a eficiência das entregas e reduzindo os custos operacionais. A seguir vejamos as etapas:

1. Etapa: Leitura do arquivo e obtenção da matriz de entrada:

```
Função ler_arquivo(nome_arquivo):
    matriz <- Lista vazia
    origem <- Nenhum
    pontos_de_entrega <- Lista vazia

    Tentar abrir o arquivo nome_arquivo em modo de leitura como arquivo:
        Para cada linha_numero, linha no arquivo (enumerado):
            linha <- Remover espaços em branco do início e fim de linha, e dividir em elementos
            Adicionar linha à lista matriz
            Para cada coluna, celula em enumerar(linha):
                Se celula é igual a 'R':
                    origem <- (linha_numero, coluna)
                Senão, se celula é uma letra do alfabeto:
                    Adicionar (linha_numero, coluna) à lista pontos_de_entrega

        Retornar matriz, origem e pontos_de_entrega
```

O pseudocódigo representa a função ler_arquivo que é responsável por ler um arquivo TXT contendo a matriz de entrada para o Projeto FlyFood. A função lê o arquivo linha por linha e cria a matriz de entrada a partir dos elementos de cada linha. Também identifica o ponto de origem (R) e os pontos de entrega (representados por letras do alfabeto) nas coordenadas corretas (i, j) na matriz. Em caso de sucesso na leitura do arquivo, a função retorna a matriz, o ponto de origem e a lista de pontos de entrega.

2. Etapa: Geração de todas as permutações possíveis dos pontos de entrega:

```
Função gerar_permutacoes(pontos):
    Se tamanho de pontos é menor ou igual a 1:
        Retornar lista contendo pontos

    permutacoes <- Lista vazia
    Para cada i de 0 até tamanho de pontos:
        ponto_atual <- pontos[i]
        pontos_restantes <- Copiar os elementos de pontos exceto o ponto_atual
        permutacoes_restantes <- Chamar a função gerar_permutacoes(pontos_restantes)
        Para cada permutacao_restante em permutacoes_restantes:
            Adicionar à lista permutacoes a lista [ponto_atual] concatenada com permutacao_restante

    Retornar permutacoes
```

O pseudocódigo representa a função gerar_permutacoes, que é utilizada para gerar todas as permutações possíveis dos pontos de entrega. A função recebe uma lista chamada pontos como entrada.

Primeiramente, a função verifica se o tamanho da lista pontos é menor ou igual a 1. Se sim, isso significa que não há pontos suficientes para gerar permutações, portanto, ela simplesmente retorna uma lista contendo os pontos existentes. Caso contrário, a função inicializa uma lista vazia chamada permutacoes. Em seguida, ela itera sobre cada elemento da lista_pontos utilizando a variável i como índice. Para cada elemento da lista, ela armazena o ponto_atual correspondente. A seguir, cria uma cópia dos pontos restantes, exceto o ponto_atual, e chama recursivamente a função gerar_permutacoes passando como argumento a lista pontos_restantes. A função retornará uma lista contendo todas as permutações dos pontos restantes.

Após obter as permutações restantes, a função realiza outro loop for, onde para cada permutação restante obtida, ela concatena o ponto_atual com cada permutação e adiciona o resultado à lista permutacoes. Após percorrer todos os elementos da lista pontos, a função retorna a lista permutacoes, que conterá todas as permutações possíveis dos pontos de entrega.

3. Etapa: Identificação do menor percurso e apresentação do resultado final:

```
Função encontrar_menor_percurso(matriz, origem, pontos_de_entrega):
    pontos <- cópia de pontos_de_entrega
    menor_percurso <- nulo
    menor_custo <- infinito

    permutacoes <- gerar_permutacoes(pontos)

    Para cada permutacao em permutacoes:
        percurso <- [origem] + permutacao + [origem]
        custo <- calcular_custo_percurso(matriz, percurso)
        Se custo < menor_custo:
            menor_custo <- custo
            menor_percurso <- percurso

    Retornar menor_percurso, menor_custo

Função encontrar_ordem_visita(matriz, origem, pontos_de_entrega):
    menor_percurso, menor_custo <- encontrar_menor_percurso(matriz, origem, pontos_de_entrega)
    ordem_de_entrega <- criar lista com os elementos de menor_percurso, excluindo o primeiro e o último elemento
    Retornar ordem_de_entrega, menor_custo
```

O pseudocódigo apresenta duas funções, encontrar_menor_percurso e encontrar_ordem_visita.

A função encontrar_menor_percurso recebe como entrada uma matriz, um ponto de origem e uma lista de pontos de entrega pontos_de_entrega. Ela inicializa uma lista pontos com uma cópia dos pontos de entrega, e duas variáveis menor_percurso e menor_custo com valores nulos e infinito, respectivamente. A seguir, a função chama a função gerar_permutacoes para obter todas as permutações possíveis dos pontos de entrega.

Então, para cada permutação em permutacoes, a função monta um percurso adicionando o ponto de origem no início e no final, criando a lista percurso. Em seguida, chama a função calcular_custo_percurso para calcular o custo do percurso e compara com o menor custo atual. Se o custo calculado for menor que o menor custo encontrado até o momento, atualiza as variáveis menor_custo e menor_percurso com os novos valores. Ao final do loop, a função retorna o menor_percurso e o menor_custo.

A função encontrar_ordem_visita recebe uma matriz matriz, um ponto de origem origem e uma lista de pontos de entrega pontos_de_entrega. Ela chama a função encontrar_menor_percurso para obter o menor percurso e o menor custo. A seguir, cria uma lista ordem_de_entrega com os elementos do menor_percurso, excluindo o primeiro e o último elemento. Por fim, a função retorna a ordem_de_entrega e o menor_custo em Dronômetros.

Dessa forma, através dessa metodologia, é possível resolver o problema do FlyFood e encontrar o menor percurso em Dronômetros para visitar todos os pontos de entrega, partindo de um ponto de partida específico. Isso permite otimizar a distância total percorrida, garantindo uma solução eficiente e prática para a logística do FlyFood.

.Fluxograma da ordem das etapas de solução apresentadas na metodologia.

4.1.Análise Algorítmica

A Análise Algorítmica do Problema do Projeto FlyFood revela importantes aspectos matemáticos a serem considerados. Ao desvendar os componentes-chave, como a complexidade de tempo, a complexidade de espaço e a análise probabilística, podemos compreender melhor o funcionamento do algoritmo e suas implicações práticas.

A complexidade de tempo é um aspecto crucial, medindo o número de operações realizadas pelo algoritmo em relação ao tamanho da entrada, ou seja, o número de pontos de entrega. A função T(n) é uma forma de representar a complexidade temporal de um algoritmo em termos do tamanho da entrada "n". Ela nos permite analisar e entender o comportamento geral do algoritmo e estimar sua eficiência para diferentes tamanhos de entrada. Para determinar a função T(n) específica para o projeto FlyFood, analisaremos a complexidade de cada etapa do algoritmo e as combinaremos adequadamente.

O algoritmo do projeto FlyFood possui várias etapas que contribuem para a sua complexidade geral:

1. Leitura do arquivo TXT e criação da matriz de entrada: A leitura do arquivo é linear. O algoritmo do projeto FlyFood possui várias etapas que contribuem para a sua complexidade geral: vez que cada linha é processada uma vez, e, para cada linha, cada elemento é lido uma vez. Portanto, essa etapa tem uma complexidade O(L*C), onde "L" é o número de linhas da matriz e "C" é o número de colunas da matriz.

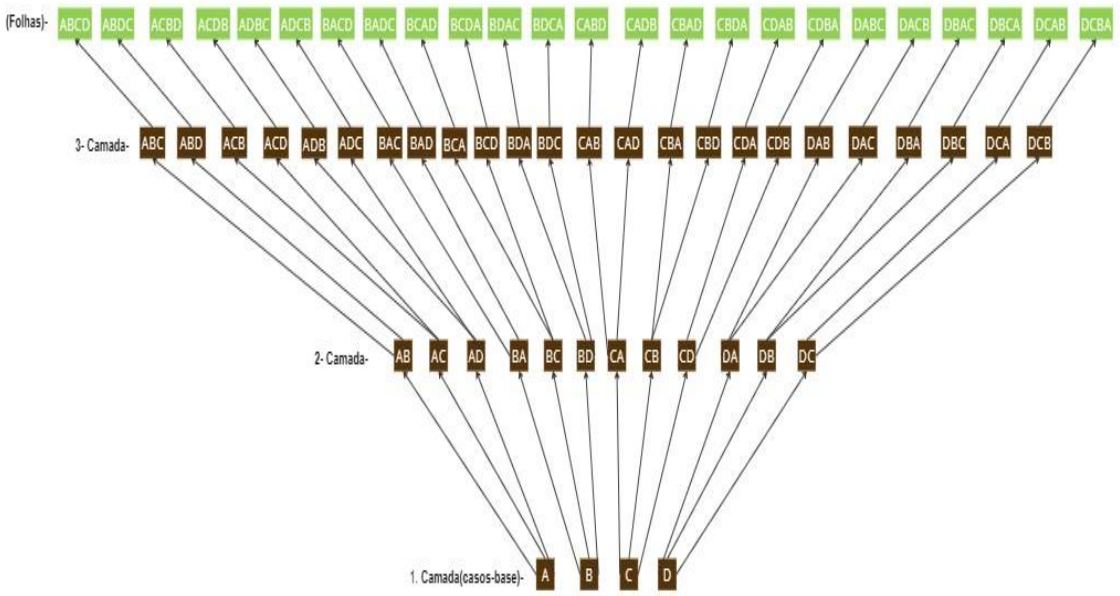
2. Identificação dos pontos de entrega e ponto de origem: A busca pelos pontos de entrega e pelo ponto de origem é realizada ao percorrer a matriz criada na etapa anterior. Nessa etapa, são feitas iterações sobre cada elemento da matriz, portanto, também possui uma complexidade O(n*m).

3. Geração de todas as permutações dos pontos de entrega: Essa etapa é realizada pela função "gerar_permutacoes". O número de permutações possíveis dos pontos de entrega é representado por "k!", onde "k" é o número de pontos de entrega. Portanto, a complexidade desta etapa é O(k!).

4. Cálculo do custo para cada permutação: Para cada permutação gerada, o algoritmo calcula o custo total percorrendo os pontos na ordem da permutação e somando as distâncias entre eles. Essa etapa é realizada pela função "calcular_custo_percurso", e sua complexidade é O(k), pois são realizadas "k" iterações para calcular o custo, onde "k" é o número de pontos de entrega na permutação.

5. Comparação e seleção do menor percurso: Após calcular o custo de cada permutação, o algoritmo compara os resultados e seleciona o percurso de menor custo. Essa etapa é realizada em tempo constante, com complexidade O(1), uma vez que apenas algumas operações de comparação são executadas.

Considerando todas as etapas acima, a complexidade final do algoritmo é dominada pela geração de permutações dos pontos de entrega (etapa 3), ou seja, O(k!). Portanto, a função T(n) específica para o projeto FlyFood é T(n) = O(k!), onde "k" representa o número de pontos de entrega na matriz.



.Representação da função T(n) do FlyFood em formato de árvore.

A complexidade de espaço refere-se à quantidade de memória necessária pelo algoritmo. Nesse caso, a quantidade de memória depende do número de pontos de entrega e da representação utilizada para armazenar as sequências de pontos. Considerando que o armazenamento das sequências é proporcional ao número de pontos de entrega, a complexidade de espaço pode ser aproximada como O(n), ou seja, uma complexidade linear.

A análise probabilística é outro aspecto relevante. No caso do Projeto FlyFood, como o algoritmo explora todas as possíveis sequências de pontos, a probabilidade de encontrar a solução ótima é garantida e igual a 1.

4.2. FlyFood NP-difícil

Um problema NP-difícil é uma classe especial de problemas computacionais que apresenta alta complexidade em sua solução. Esses problemas estão relacionados à classe NP (Nondeterministic Polynomial time), que engloba todos os problemas que podem ser verificados em tempo polinomial(medida de complexidade de algoritmos que se relaciona com a eficiência de sua execução em função do tamanho da entrada), ou seja, uma solução proposta pode ser verificada em tempo polinomial.

No contexto da Teoria dos Grafos, o "Circuito Hamiltoniano" é um dos problemas mais estudados. Ele consiste em encontrar um caminho em um grafo que passe por todos os seus vértices exatamente uma vez e retorne ao vértice inicial, formando um ciclo. O problema recebe esse nome em homenagem ao matemático Sir William Rowan Hamilton, que contribuiu para sua formulação.

O Problema do Circuito Hamiltoniano é considerado NP-Difícil porque não se sabe até o momento se ele pertence à classe NP (PoliNomial Não-determinístico). Embora haja algoritmos que possam verificar se um dado circuito é hamiltoniano em tempo polinomial, ainda não foi encontrado um algoritmo eficiente para encontrar um circuito hamiltoniano em grafos não triviais.

A dificuldade do Circuito Hamiltoniano reside na busca por um algoritmo eficiente que encontre tal caminho em todos os grafos possíveis. Até o momento, não conhecemos um algoritmo de tempo polinomial capaz de resolvê-lo para todos os casos, o que o classifica como um problema NP-difícil.

Ao compararmos o Circuito Hamiltoniano com o problema do "Projeto FlyFood", observamos que o último pode ser mapeado para o primeiro. No Projeto FlyFood, temos uma matriz de entrada que representa um grid retangular com células contendo blocos vazios ou pontos de entrega identificados pelas letras A, B, C, D, e o ponto de partida e chegada identificado pela letra R. Para encontrar a melhor solução para o Projeto FlyFood, procuramos um caminho que percorra todos os pontos de entrega exatamente uma vez e retorne ao ponto de partida, minimizando a distância total percorrida. Essa descrição é essencialmente um Circuito Hamiltoniano, uma vez que o objetivo é encontrar um ciclo que visite todos os pontos de entrega sem repetições e retorne ao ponto de partida.

Portanto, o Projeto FlyFood é considerado NP-difícil porque, assim como o Circuito Hamiltoniano, não conhecemos um algoritmo eficiente para resolvê-lo em tempo polinomial para todas as instâncias possíveis. Isso implica que, caso seja possível encontrar uma solução rápida para o Projeto FlyFood, poderíamos usá-la para resolver o Circuito Hamiltoniano de forma eficiente, o que até o momento é considerado improvável. Tendo isso em mente, ambos os problemas são desafios computacionais complexos e relevantes para a área de otimização e teoria da complexidade.

4.3.Algoritmo Genético flyfood.

O algoritmo genético pode ser usado para resolver o problema do caixeiro viajante no FlyFood. O algoritmo genético é uma meta-heurística que pode ser usada para encontrar soluções aproximadas para problemas de otimização. O algoritmo genético funciona da seguinte forma:

agora vamos entender como funciona a etapa do ciclo dentro do problema do flyfood:

1.Etapa: importar bibliotecas usadas:

As bibliotecas `ramdom` e `math` são bibliotecas padrão do Python que são usadas para gerar números aleatórios e calcular funções matemáticas. A biblioteca `heapq` é uma biblioteca que é usada para trabalhar com heap (pilha de prioridade). A biblioteca `time` é uma biblioteca que é usada para medir o tempo de execução de um programa. A biblioteca `statistics` é uma biblioteca que é usada para calcular estatísticas sobre um conjunto de dados.

2. Etapa: definir semente e inicializar:
A linha de código `seed = 37` define a semente para o gerador de números aleatórios. A semente é um valor que é usado para inicializar o gerador de números aleatórios. Isso garante que os números aleatórios gerados sejam os mesmos sempre que o programa for executado com a mesma semente. A linha de código `random.seed(seed)` inicializa o gerador de números aleatórios com o valor da semente.

3. Etapa: receber coordenadas das cidades:

```
*Definição da classe Vertice*

* Cada vértice é definido por um número, uma coordenada x e uma coordenada y.
* O construtor da classe recebe esses três valores como parâmetros.

*Definição da lista de cidades*

* A lista de cidades é um array de vértices.
* O array é inicializado com cinco vértices.
* Cada vértice é inicializado com um número, uma coordenada x e uma coordenada y.
* O tamanho da lista é armazenado na variável `tamanhoDaLista`.

*Algoritmo*

* *Início do algoritmo*
* *Inicializar o contador `i` com 0*
* *Enquanto o contador `i` for menor que o tamanho da lista*
    * *Carregar o vértice `v` da lista*
    * *Imprimir o número, a coordenada x e a coordenada y do vértice `v`*
    * *Aumentar o contador `i` em 1*
* *Fim do algoritmo*
```

Esta parte do código define as coordenadas de entrada para o problema do Flyfood. É criada uma classe chamada `Vertice` para representar um vértice em um grafo. Cada vértice tem um número e um par de coordenadas (x, y). Por fim, é criada uma lista chamada `listaCidade` que contém cinco vértices. Cada vértice representa uma cidade.

4. Etapa: definir parâmetro do algoritmo genético:

O tamanhoDaPopulação é o número de indivíduos que serão considerados em cada geração do algoritmo genético. A qtdGeracao é o número de vezes que o algoritmo genético será executado. tamanhoCorte é a porcentagem da população que será selecionada para reprodução em cada geração. numeroMaximoDeTentativas é o número máximo de vezes que o algoritmo genético será executado, Se o algoritmo não encontrar uma solução melhor após o número máximo de tentativas, ele será encerrado.

5. Etapa: definir cálculo aptidão:

```
*Algoritmo para o método `calculaFitness()`*

* *Início do algoritmo*
* *Se o valorFitness for -1*
    * *Inicializar a variável soma com 0*
    * *Para cada índice x de 0 a tamanhoDaLista - 2*
        * *Carregar o vértice um da vez da lista de cidades*
        * *Carregar o vértice dois da vez da lista de cidades*
        * *Calcular a distância entre os dois vértices*
        * *Adicionar a distância à variável soma*
    * *Carregar o vértice um da vez da lista de cidades*
    * *Carregar o vértice dois da vez da lista de cidades*
    * *Calcular a distância entre os dois vértices*
    * *Adicionar a distância à variável soma*
    * *Definir o valorFitness como a soma*
* *Retornar o valorFitness*
* *Fim do algoritmo*
```

A função __it__ é um método usado para comparar dois cromossomos. O cromossomo que tem o menor valor de aptidão é considerado menor. Fazendo isso com o resultado da outra função abaixo, calcularFitness: Este método calcula o valor de aptidão do cromossomo. O valor de aptidão é uma medida da qualidade da solução representada pelo cromossomo. A distância entre as duas cidades é calculada usando a fórmula da distância euclidiana.

6.Etapa: Armazenar caminho e adicionar índices:

O código inicializadorDaPopulacao = [] é uma lista vazia que, recebe pelo loop for e itera de 1 ao tamanho da lista, e adiciona índice ao inicioDaPopulação.

7.Etapa: Inicializar população:

```
*Algoritmo para inicializar a população*

* *Início do algoritmo*
* *Inicializar a lista `inicializadorDaPopulacao` com os números de 1 a tamanhoDaLista*
* *Para cada índice x de 0 a tamanhoDaLista - 1*
    * *Adicionar o número x à lista `inicializadorDaPopulacao`*
* *Fim do algoritmo*
```

O pseudocódigo `procedimento geraPopulacao()` cria uma população inicial de `tamanhoDaPopulacao` indivíduos para o problema de roteamento de veículos. Cada indivíduo é uma lista de índices das cidades. Os índices são embaralhados para criar uma solução aleatória para o problema.

8.Etapa: Seleção indivíduos:

```
def selecaoTorneio(parametroPopulacao, tamanhoTorneio):
    vencedores = []
    for x in range(tamanhoTorneio):
        vencedor = selecaoIndividuo(parametroPopulacao)
        vencedores.append(vencedor)
    return vencedores[random.randrange(tamanhoTorneio)]
```

Função chamada `selecaoIndividuo()` que seleciona dois indivíduos da população e retorna o indivíduo com o menor fitness. O fitness de um indivíduo é uma medida de quão bom ele é. Neste caso, o fitness de um indivíduo é a distância total que ele percorre. A seleção de indivíduos com o menor fitness é uma estratégia comum em algoritmos genéticos. Isso porque os indivíduos com o menor fitness são mais propensos a gerar descendentes com melhor fitness.

9.Etapa: Cruzamento dos indivíduos:

```
# Criar uma lista vazia para armazenar os novos cromossomos
listaNovosIndividuos = []

# Criar uma variável para controlar o número de gerações
contador = 0

# Enquanto o contador for menor que o número de gerações
while contador < qtdGeracao:

    # Criar uma variável para controlar o número de genes adicionados ao novo cromossomo
    qtdGenesAdicionados = 0

    # Criar uma lista com os primeiros tamanhoCorte genes do cromossomo individuoUm
    geneDosPaisParaFilho = individuoUm.caminho[:tamanhoCorte]

    # Para cada gene x no cromossomo individuoDois
    for x in individuoDois.caminho:

        # Se qtdGenesAdicionados for igual ao tamanho da lista de genes do cromossomo individuoUm menos o tamanho do corte
        if qtdGenesAdicionados == (tamanhoDaLista - tamanhoCorte):

            # Sair do loop
            break

        # Se x não estiver na lista geneDosPaisParaFilho
        if x not in geneDosPaisParaFilho:
```

```
            # Adicionar o gene x à lista geneDosPaisParaFilho
            geneDosPaisParaFilho.append(x)

            # Atualizar a variável qtdGenesAdicionados
            qtdGenesAdicionados += 1

    # Aplicar a mutação ao cromossomo geneDosPaisParaFilho
    geneDosPaisParaFilho = mutacaoIndividuo(geneDosPaisParaFilho)

    # Criar um novo cromossomo com o caminho geneDosPaisParaFilho
    novoIndividuo = Cromossomo()
    novoIndividuo.caminho = geneDosPaisParaFilho

    # Adicionar o novo cromossomo à lista listaNovosIndividuos
    listaNovosIndividuos.append(novoIndividuo)

    # Atualizar a variável contador
    contador += 1

# Retornar a lista listaNovosIndividuos
return listaNovosIndividuos
```

A função `crossoverIndividuo()` recebe dois indivíduos como argumentos. O primeiro indivíduo é o pai e o segundo indivíduo é a mãe. A função retorna uma lista com os dois indivíduos filhos.

10.Etapa: Mutação indivíduos:

```
def mutacaoIndividuo(caminhoDoNovoFilhoGerado):
    # Início do algoritmo

    # Selecionar dois genes aleatórios do cromossomo
    geneUm = random.randrange(tamanhoDaLista)
    geneDois = random.randrange(tamanhoDaLista)

    # Trocar a posição dos dois genes
    caminhoDoNovoFilhoGerado[geneUm], caminhoDoNovoFilhoGerado[geneDois] = caminhoDoNovoFilhoGerado[geneDois], caminhoDoNovoFilhoGerado[geneUm]

    # Retornar o cromossomo com os genes trocados
    return caminhoDoNovoFilhoGerado
```

A função `mutacaoIndividuo()` recebe um caminho como argumento. O caminho é uma lista de números, que representam as cidades a serem visitadas. A função retorna o caminho mutado. O caminho mutado é um novo caminho, com dois genes trocados de posição.

11.Etapa: Atualizar população

```
def atualizaPopulacao(parametroPopulacao, parametroListaDeNovosIndividuos):

    # Início do algoritmo

    # Para cada novo indivíduo x na lista de novos indivíduos
    for x in parametroListaDeNovosIndividuos:

        # Obter o indivíduo com o melhor fitness na população
        individuoMaiorFitness = heapq.nlargest(1, parametroPopulacao)[0]

        # Obter o indivíduo com o menor fitness na população
        menor = parametroPopulacao[0]

        # Obter o fitness do novo indivíduo
        xFitness = x.calculaFitness()

        # Se o fitness do novo indivíduo for menor que o fitness do indivíduo com o melhor fitness
        if xFitness < individuoMaiorFitness.calculaFitness():

            # Remover o indivíduo com o melhor fitness da população
            parametroPopulacao.remove(individuoMaiorFitness)

            # Adicionar o novo indivíduo à população
            heapq.heappush(parametroPopulacao, x)
            heapq.heapify(parametroPopulacao)
```

```
            # Se o fitness do novo indivíduo for menor que o fitness do indivíduo com o menor fitness
            if xFitness < menor.calculaFitness():

                # Reiniciar o número de tentativas
                numeroTentativas = 0

            else:

                # Incrementar o número de tentativas
                numeroTentativas += 1

        else:

            # Incrementar o número de tentativas
            numeroTentativas += 1

        # Se o número de tentativas for igual ao número máximo de tentativas
        if numeroTentativas == numeroMaximoDeTentativas:

            # Definir a melhor solução como False
            melhorSolucao = False

    # Retornar a população atualizada
    return parametroPopulacao
```

a função `atualizaPopulacao()` atualiza a população com os novos indivíduos. Para isso, ela compara o fitness de cada novo indivíduo com o fitness do indivíduo com melhor fitness na população atual. Se o fitness do novo indivíduo for menor, o novo indivíduo é adicionado à população atual e o processo de atualização começa do início. Se o fitness do novo indivíduo não for menor, o número de tentativas é incrementado. Se o número de tentativas for igual ao número máximo de tentativas, a solução não é melhorável.

12.Etapa: melhor individuo da população:

```
def obtemMelhor(parametroPopulacao):

    # Início do algoritmo
    # Obter o indivíduo com o melhor fitness na população
    melhorIndividuo = parametroPopulacao[0]

    # Retornar o indivíduo com o melhor fitness
    return melhorIndividuo

def geraPopulacao():

    # Início do algoritmo
    # Criar uma lista vazia para armazenar a população
    populacao = []

    # Para cada indivíduo na população
    for _ in range(tamanhoDaPopulacao):

        # Gerar um indivíduo aleatório
        individuo = Cromossomo()
        individuo.caminho = random.sample(listaDeCidades, tamanhoDaLista)

        # Adicionar o indivíduo à população
        populacao.append(individuo)

    # Retornar a população
    return populacao
```

A função `obtemMelhor()` retorna o indivíduo com melhor fitness na população. A função `geraPopulacao()` inicializa a população com indivíduos aleatórios. A variável `fitness_por_geracao` armazena o fitness do melhor indivíduo da população em cada geração. A variável `tempo_inicio` armazena o tempo de início do algoritmo. A variável `melhor_fitness_geral` armazena o melhor fitness geral encontrado pelo algoritmo. A variável `geracao_melhor_fitness_geral` armazena a geração em que o melhor fitness geral foi encontrado.

13.Etapa: impressão dos resultados:

```
# Início do algoritmo

geracao = 1

# Enquanto a geração for menor ou igual ao número de gerações
while geracao <= qtdGeracao:

    # Selecionar dois indivíduos da população
    individuoUm = selecaoIndividuo(populacao)
    individuoDois = selecaoIndividuo(populacao)

    # Aplicar o operador de crossover nos dois indivíduos
    novosIndividuosGerados = crossoverIndividuo(individuoUm, individuoDois)

    # Atualizar a população com os novos indivíduos
    populacao = atualizaPopulacao(populacao, novosIndividuosGerados)

    # Calcular e armazenar o fitness do melhor indivíduo da geração atual
    melhor_fitness = obtemMelhor(populacao).calculaFitness()
    fitness_por_geracao.append(melhor_fitness)

    # Atualizar o melhor fitness geral
    if melhor_fitness < melhor_fitness_geral:
        melhor_fitness_geral = melhor_fitness
        geracao_melhor_fitness_geral = geracao
```

```
    # Imprimir os resultados da geração
    print(f'Geração {geracao}: Melhor Fitness = {melhor_fitness}')
    melhorIndividuo = obtemMelhor(populacao)
    print(f"Melhor indivíduo (Menor distância): {melhorIndividuo.caminho}")
    print(f"Fitness: {melhorIndividuo.calculaFitness()}")
    print()  # Pula uma linha

    # Incrementar a geração
    geracao += 1

# Para o temporizador
tempo_fim = time.time()

# Imprimir os resultados gerais
melhorIndividuoGeral = obtemMelhor(populacao)
tempo_total = tempo_fim - tempo_inicio
desvio_padrao_fitness = statistics.stdev(fitness_por_geracao)

print(f"Melhor indivíduo geral (Menor distância) na Geração {geracao_melhor_fitness_geral}: {melhorIndividuoGeral.caminho}")
print(f"Fitness: {melhorIndividuoGeral.calculaFitness()}")
print(f"Tempo de execução total: {tempo_total} segundos")
print(f"Desvio Padrão do Fitness das Gerações: {desvio_padrao_fitness}")
```

O pseudocódigo acima implementa um algoritmo genético básico para resolver o problema do caixeiro viajante. O algoritmo funciona da seguinte maneira: A população é inicializada com indivíduos aleatórios. Para cada geração: 1-Dois indivíduos são selecionados da população por meio de seleção natural. 2-Os dois indivíduos selecionados são cruzados para produzir dois novos indivíduos. 3-Os novos indivíduos são adicionados à população. 4-O fitness do melhor indivíduo da geração atual é calculado e armazenado. 5-Se o fitness do melhor indivíduo da geração atual for menordo que o melhor fitness geral, o melhor fitness geral é atualizado. 6-Os resultados da geração atual são impressos. Os passos 2 e 3 são repetidos até que uma solução satisfatória seja encontrada ou até que um limite de tempo seja atingido. No final do algoritmo, são impressos os resultados do algoritmo, incluindo o melhor indivíduo geral, o tempo de execução total e o desvio padrão do fitness das gerações.

4.4 Colônia de Formigas

A seguir está a explicação detalhada do código que utiliza o algoritmo da Colônia de Formigas para resolver o problema do FLyFlood.

1. Função ler_coordenadas_cidades

```
Função ler_coordenadas_cidades(caminho_arquivo):
    Inicializa uma lista vazia de cidades
    Inicializa a variável primeira como nula

    Abra o arquivo especificado em modo de leitura
    Para cada linha no arquivo:
        Leia os valores z, x e y da linha como números de ponto flutuante
        Se z não for igual a 1:
            Adicione a tupla (x, y) à lista de cidades
        Caso contrário, atribua a tupla (x, y) à variável primeira

    Retorne a lista de cidades e a variável primeira
```

Esta função abre o arquivo contendo as coordenadas dos pontos. Uma lista vazia é iniciada para armazenar essas coordenadas, então cada linha do arquivo é percorrida. Divide a linha em três valores (z, x e y) usando o espaço em branco como separador, em seguida os espaços em branco são removidos e adiciona as coordenadas (x, y) à lista de cidades.

Entrada: O caminho do arquivo de entrada contendo as coordenadas das cidades.
Saída: Uma lista de coordenadas das cidades.

2. Classe ColoniaFormigasPCV

Essa classe implementa o algoritmo da Colônia de Formigas para resolver o Problema do FlyFood.

3. Método __init__ da classe ColoniaFormigasPCV

```
Classe ColoniaFormigasPCV:
    Método __init__(self, num_formigas, num_iteracoes, evaporacao_feromonio, alfa, beta):
        Inicializa os parâmetros da colônia de formigas
```

Essa função inicializa os parâmetros da classe. Esses parâmetros são: número de formigas, número de interações, evaporação de feromônios, alfa e beta.

Entrada: Número de formigas, número de iterações, taxa de evaporação do feromônio, parâmetros alfa e beta.

4. Método resolver_pcv

```
Método resolver_tsp(self, matriz_distancias):
    Inicializa a matriz de feromônios com valores iniciais
    Inicializa as variáveis melhor_caminho e melhor_distancia

    Para cada iteração no número de iterações:
        Gere caminhos para todas as formigas
        Atualize os feromônios com base nos caminhos gerados
        Encontre o melhor caminho atual e sua distância

        Se o melhor caminho atual for melhor do que o melhor caminho encontrado até agora:
            Atualize o melhor caminho e a melhor distância

        Evapore uma parte dos feromônios em todas as arestas

    Retorne o melhor caminho encontrado e sua distância
```

Inicializa a matriz de feromônios e as variáveis de melhor caminho e de melhor distância. Executa o algoritmo da Colônia de Formigas por um número especificado de iterações. Gera caminhos para cada formiga, atualiza os feromônios com base nos caminhos gerados e verifica se o novo caminho é melhor que o melhor caminho atual, por último evapora os feromônios existentes e retorna o melhor caminho e sua distância.

Entrada: Matriz de distâncias entre as cidades.
Saída: O melhor caminho encontrado e sua distância.

5. Método gerar_caminhos_formigas

```
Método gerar_caminhos_formigas(self, matriz_distancias, matriz_feromonios):
    Inicializa uma lista vazia de caminhos_formigas

    Para cada formiga no número de formigas:
        Inicializa um caminho da formiga com a primeira cidade
        Inicializa um conjunto de cidades visitadas com a primeira cidade

        Enquanto houver cidades não visitadas:
            Escolha a próxima cidade com base nos feromônios e distâncias
            Adicione a cidade escolhida ao caminho da formiga
            Marque a cidade como visitada no conjunto de cidades visitadas

        Adicione o caminho da formiga à lista de caminhos_formigas

    Retorne a lista de caminhos_formigas
```

Para cada formiga: Inicializa um caminho com a cidade inicial 0, então para cada cidade restante é escolhida a próxima cidade com base nas regras da função escolher_proxima_cidade. A próxima cidade é adicionada a lista caminho_formiga e ao conjunto cidades_visitadas. Adiciona o caminho da formiga à lista de caminhos.

Entrada: Matriz de distâncias e matriz de feromônios.
Saída: Uma lista de caminhos para as formigas.

6. Método escolher_proxima_cidade

```
Método escolher_proxima_cidade(self, cidade_atual, cidades_visitadas, matriz_feromonios, matriz_distancias):
    Calcule as probabilidades para todas as cidades não visitadas com base nos feromônios e distâncias
    Escolha a próxima cidade com base nas probabilidades calculadas
    Retorne a cidade escolhida
```

Escolher_proxima_cidade é responsável por determinar qual será a próxima cidade visitada por uma formiga, com base em critérios como o nível de feromônio entre as cidades e a distância entre elas. Um cálculo é feito para que se tenha a probabilidade de cada cidade ser escolhida. A fórmula desse cálculo é : feromônio ** alfa * (1.0 / distância) ** beta. O feromônio é depositado nas trilhas enquanto as formigas se movem, alfa é um parâmetro que controla a importância do feromônio e beta é outro parâmetro, que controla a importância da distância.

Saída: Retorna a cidade escolhida como a próxima cidade a ser visitada pela formiga

7. Método atualizar_feromonios

```
Método atualizar_feromonios(self, matriz_feromonios, caminhos_formigas, matriz_distancias):
    Para cada caminho da formiga em caminhos_formigas:
        Calcule a distância percorrida pelo caminho da formiga
        Para cada par de cidades consecutivas no caminho da formiga:
            Atualize os feromônios na aresta com base na qualidade do caminho
```

O método atualizar_feromonios é responsável por atualizar os níveis de feromônio nas trilhas após cada iteração do algoritmo. Para cada formiga que completa um passeio (solução), a quantidade de feromônio depositada em cada aresta é calculada.

Entradas: A matriz que armazena os níveis de feromônio entre as cidades, uma lista de caminhos percorridos por todas as formigas em uma iteração e a matriz que armazena as distâncias entre todas as cidades.

Saída: A matriz de feromônio atualizada com os novos valores após todas as formigas terem percorrido seus caminhos na iteração atual.

8. Método calcular_distancia_caminho

```
Método calcular_distancia_caminho(self, caminho, matriz_distancias):
    Inicializa a variável de distância como zero
    Para cada par de cidades consecutivas no caminho:
        Adicione a distância entre as cidades à distância total
    Adicione a distância da última cidade à primeira cidade (fechando o ciclo)
    Retorne a distância total
```

Essa função calcula a distância total de um caminho dado.

Entrada: Um caminho e a matriz de distâncias.
Saída: A distância total do caminho.

9. Parte Principal do Programa

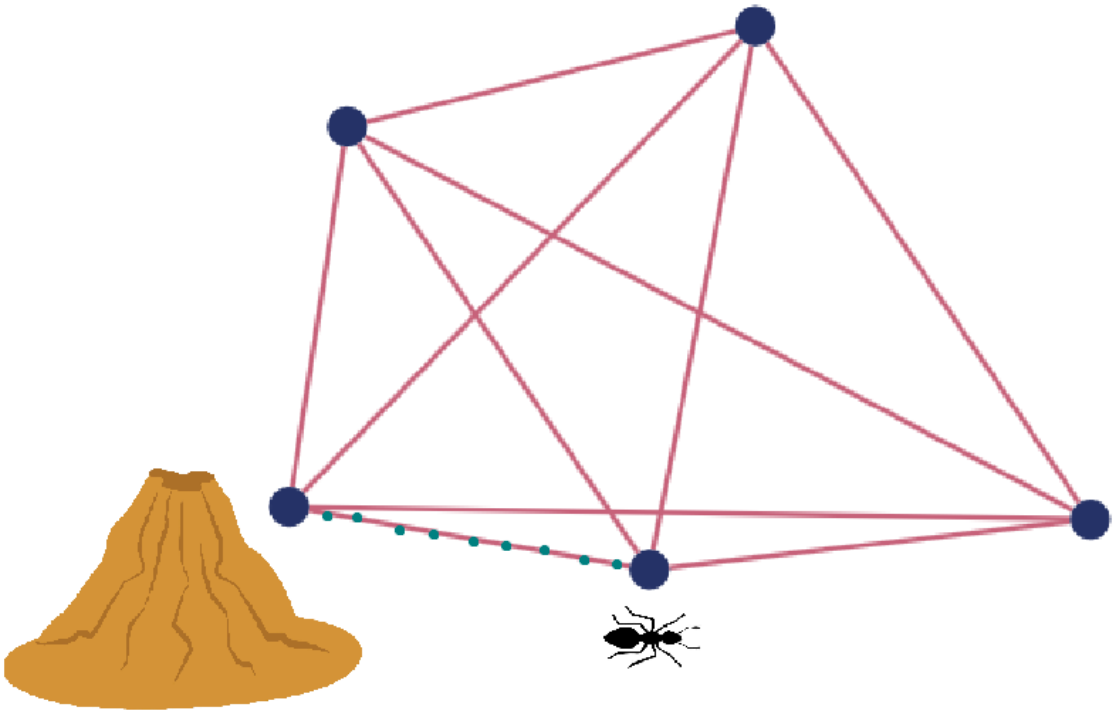
```
Se o programa for executado como o programa principal:
    Configure a semente aleatória para reprodutibilidade
    Leia as coordenadas das cidades a partir de um arquivo
    Calcule a matriz de distâncias com base nas coordenadas
    Configure os parâmetros do algoritmo de colônia de formigas

    Crie uma instância da classe ColoniaFormigasTSP com os parâmetros especificados
    Execute o algoritmo para resolver o Problema do Caixeiro Viajante
    Adicione a cidade inicial (0) ao melhor caminho encontrado
    Imprima o melhor caminho e a melhor distância encontrados
```

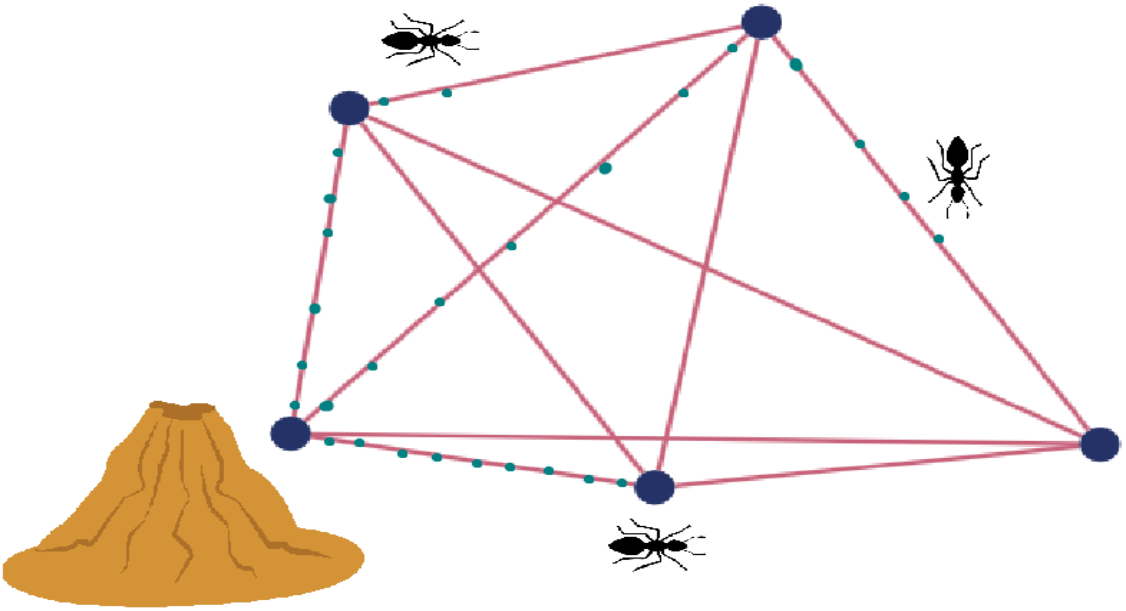
Configuração e execução do algoritmo. Define a semente para geração de números aleatórios, lê as coordenadas das cidades a partir de um arquivo, calcula a matriz de distâncias entre as cidades, configura os parâmetros do algoritmo (número de formigas, iterações, taxa de evaporação, alfa e beta), cria uma instância da classe ColoniaFormigasPCV com os parâmetros especificados, executa o algoritmo para encontrar o melhor caminho e então imprime o melhor caminho e a melhor distância encontrados.

Demonstração do algoritmo

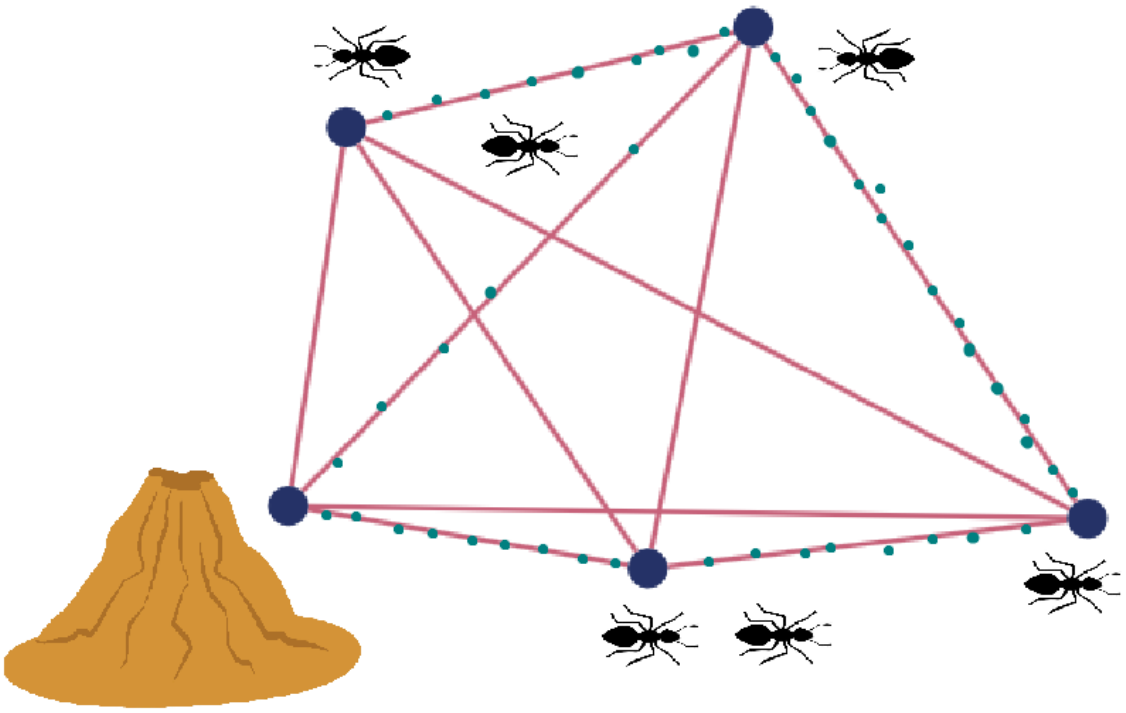
Depois de montado o plano cartesiano, as formigas iniciam sua jornada por esse espaço, deixando um rastro de feromônios à medida que avançam.



No início as formigas saem de forma desordenada, já que os feromônios não foram depositados as distância tem o papel de definir as rotas. Após a primeira interação os feromônios são atualizados.



Com a atualização dos feromônios, as formigas começam a priorizar as melhores rotas à medida que o algoritmo progride. Rotas que levam a soluções de menor custo acumulam mais feromônios, tornando-as mais atraentes para as formigas.



5. Experimentos

5.1 Testes Realizados no algoritmo de Força Bruta

Para realizar os testes, foi utilizada uma máquina virtual com uma capacidade modesta, contendo uma CPU de 0,5 GHz e 512 MB de RAM.

Os testes foram conduzidos no site: replit.com. Os resultados dos testes foram registrados na tabela abaixo:

Pontos	Menor Distância em Dronômetros	Tempo de Execução em Segundos
5	14	0,0003
6	18	0,0009
7	22	0,0074
8	26	0,2971
9	30	6,4127
10	34	15,1592
11	38	157,7150

Ao analisar a tabela, fica evidente que a partir de 7 pontos, o algoritmo começa a se comportar como uma função fatorial (O(n!)), resultando em um crescimento absurdo no tempo de execução em segundos. Um teste com 12 pontos foi tentado, porém o limite de memória reservado foi excedido, levantando o erro "repl process died unexpectedly: signal: killed".

A abordagem de Força Bruta oferece resultados mais precisos em termos de distância mínima percorrida, como indicado pela menor distância em Dronômetros. Isso ocorre porque o algoritmo de Força Bruta explora todas as combinações possíveis, garantindo que a solução encontrada seja a melhor possível.

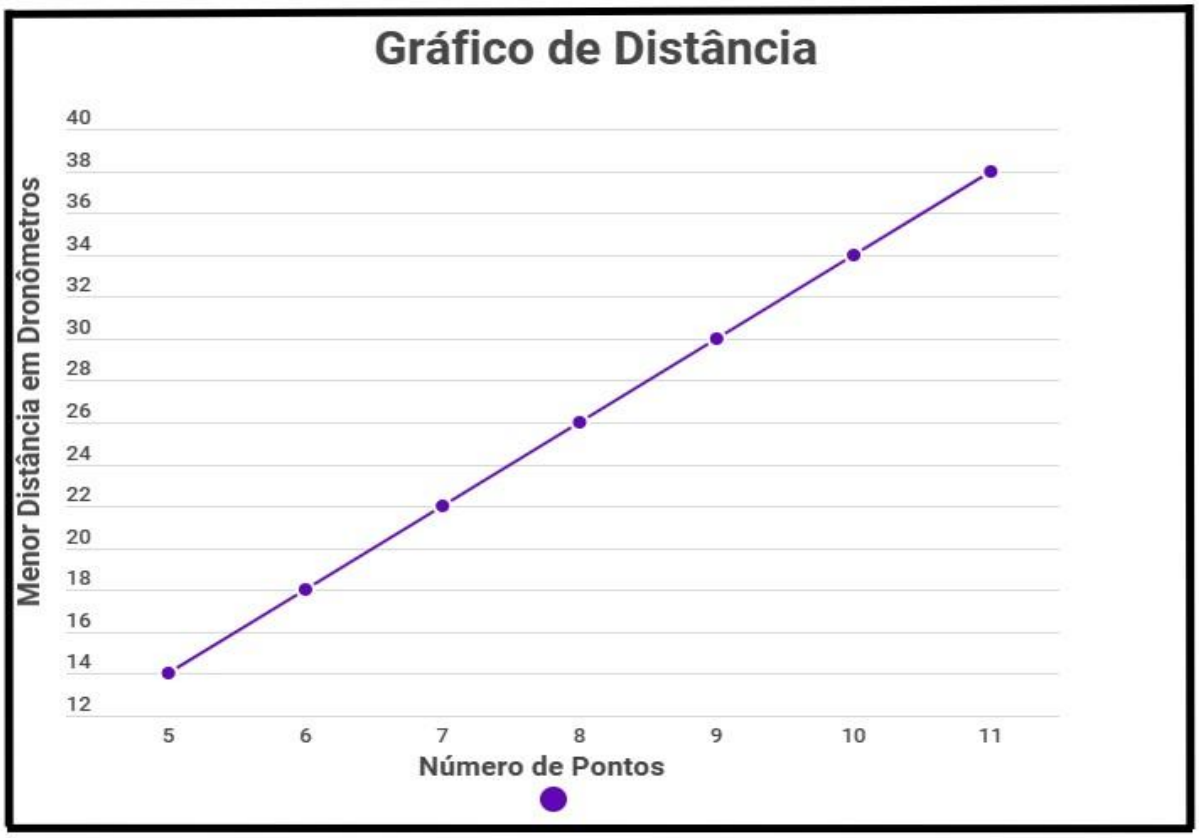


Gráfico de Distância do Algoritmo de Força Bruta

Uma observação notável é o crescimento exponencial no tempo de execução à medida que o número de pontos de entrega aumenta. Enquanto o tempo necessário para resolver o problema com 5 pontos é extremamente curto (0,0003 segundos), ele aumenta significativamente à medida que mais pontos são adicionados.

O tempo de execução torna-se impraticável para cenários com mais de 10 pontos, conforme observado no teste com 11 pontos, onde o tempo de execução ultrapassa 2 minutos. Para problemas de roteirização com um número reduzido de pontos de entrega, o algoritmo de Força Bruta pode ser uma escolha viável, proporcionando soluções precisas em um tempo aceitável.

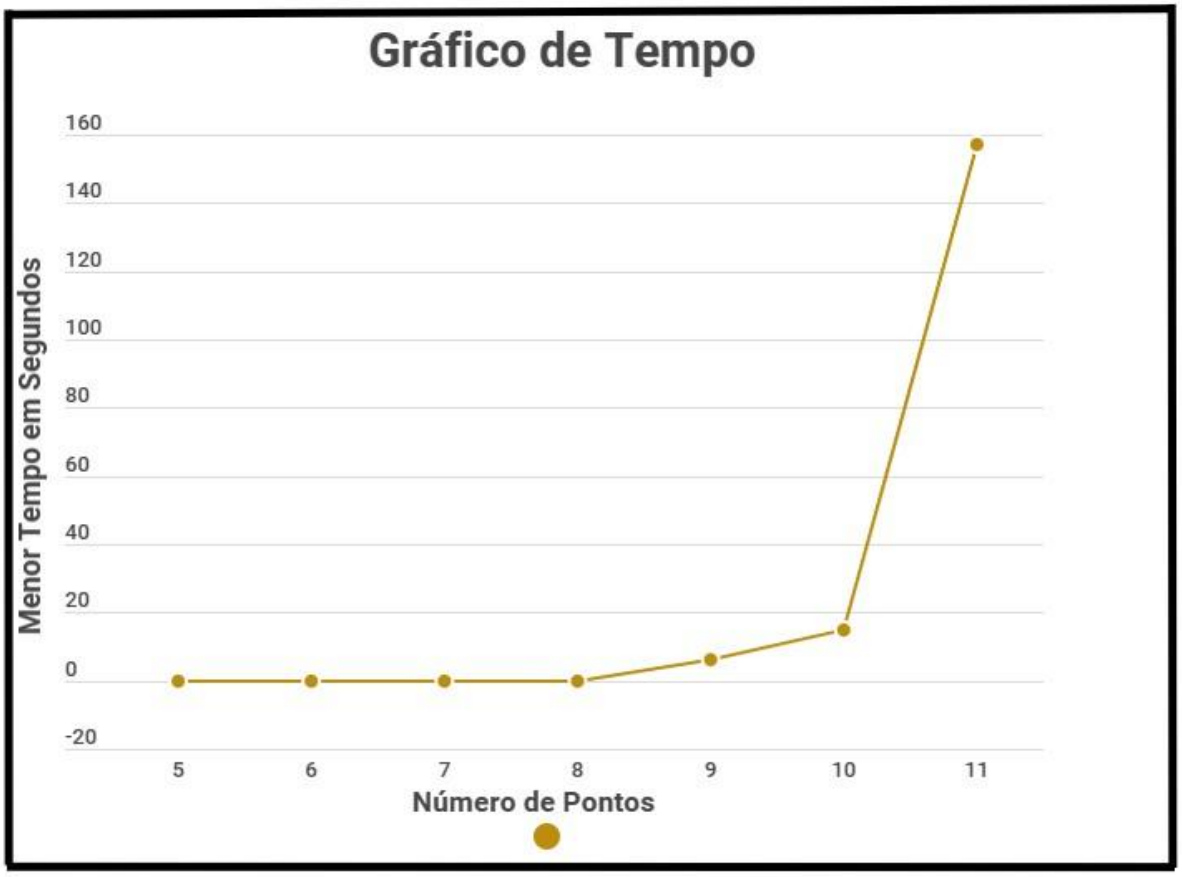
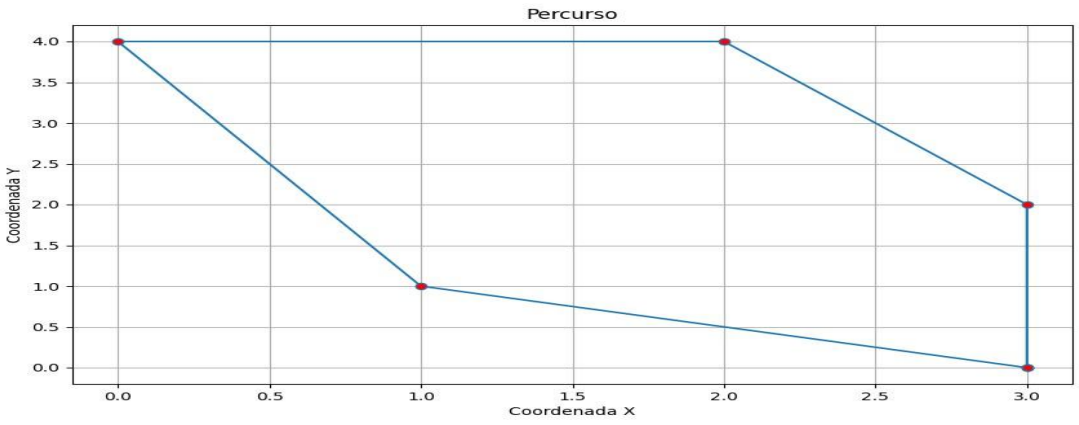


Gráfico de Tempo do Algoritmo de Força Bruta

Em resumo, os testes do algoritmo de Força Bruta destacam sua capacidade de encontrar soluções precisas, mas também revelam sua limitação em termos de escalabilidade. Para problemas com um número moderado de pontos de entrega, essa abordagem pode ser útil. No entanto, para cenários mais complexos, o tempo de execução excessivo a torna impraticável, destacando a importância de explorar alternativas, como algoritmos genéticos, para otimizar a roteirização em grande escala.

Também foi feito o Plot do melhor percurso para o FlyFood segundo o algoritmo de força bruta, o qual segue a ordem de entrega [R, A, D, C, B, R].



Assim, conclui-se que esse algoritmo pode ser uma solução razoável para problemas com percursos um número reduzido de pontos, mas para roteiros maiores, torna-se impraticável devido ao rápido crescimento fatorial do tempo de execução em segundos.

.

5.2 Testes Realizados no algoritmo Genético

Para realizar os testes, foi utilizada uma máquina virtual com uma capacidade modesta, contendo uma CPU de 0,5 GHz e 512 MB de RAM.

Os testes foram executados seguindo os seguintes parâmetros:

Semente	Tamanho Da População	Gerações	N. Tentativas	Tamanho Corte
11	50	200	1000	0.90

Os testes foram conduzidos no site: replit.com. Os resultados dos testes foram registrados na tabela abaixo:

Pontos	Menor Distância em Dronômetros	Tempo de Execução em segundos
5	11,6344	6,4495
6	14,1065	6,7375
7	17,2688	6,9177
8	20,4311	7,0745
9	23,4667	7,1812
10	26,2727	7,1885
11	29,9526	7,3118
12	31,2278	7,4115

Os resultados obtidos nos testes podem variar dependendo da configuração específica do dos parâmetros utilizados. É possível ajustar esses parâmetros, como o tamanho da população, o número de gerações, a taxa de mutação para modificar o desempenho em problemas específicos.

Os resultados mostram que o algoritmo genético conseguiu encontrar soluções para problemas com até 12 pontos de entrega. Isso demonstra que o algoritmo é capaz de lidar com problemas de tamanho moderado, o que é uma característica desejável em muitas aplicações do nosso mundo real.

O aumento da menor distância à medida que o número de pontos aumenta é uma tendência natural. Isso ocorre porque, à medida que mais pontos são adicionados ao problema de roteirização, há um aumento nas possíveis combinações de sequências de visitas. O algoritmo genético busca encontrar a sequência que minimiza a distância total, mas encontrar a solução ótima se torna mais difícil à medida que o espaço de busca se expande com mais pontos. Portanto, é esperado que a menor distância aumente à medida que o problema se torna mais complexo.

O aumento no tempo de execução à medida que o número de pontos aumenta é uma característica comum de algoritmos de otimização. À medida que o espaço de busca se torna mais amplo e complexo, o algoritmo precisa realizar mais iterações e avaliações para encontrar uma solução aceitável. O aumento gradual no tempo de execução demonstra que o algoritmo está encontrando soluções razoáveis em um tempo aceitável, mesmo para problemas mais desafiadores.

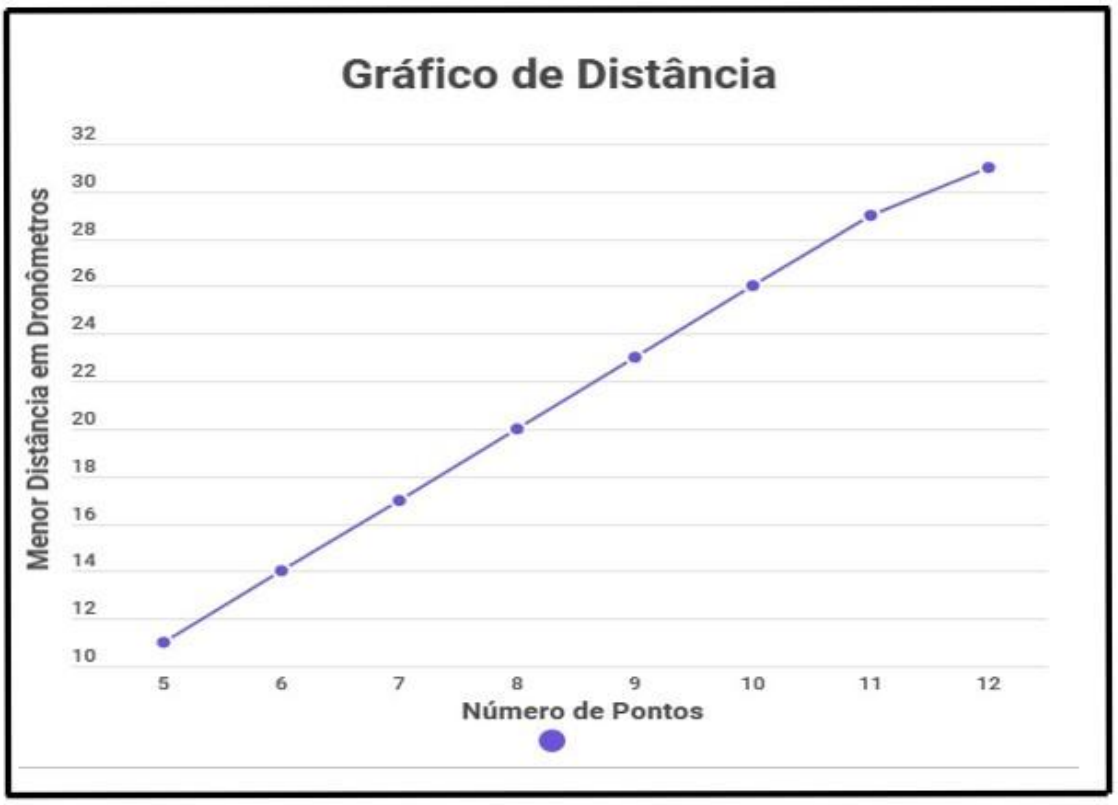
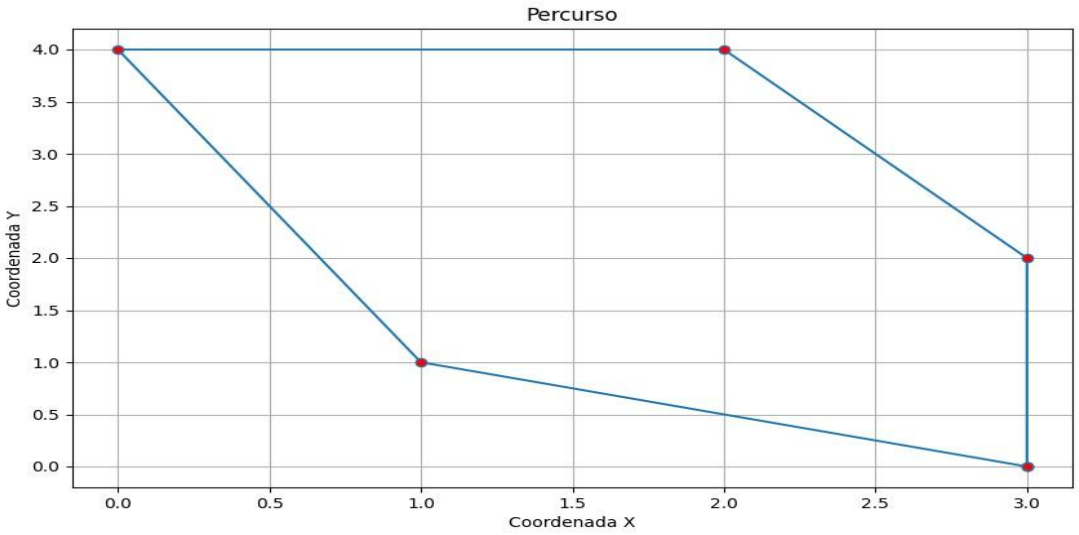


Gráfico de Distância

Gráfico de Tempo

Também foi feito o Plot do melhor percurso para o FlyFood segundo o algoritmo genético, o qual segue a ordem de entrega [R, A, D, C, B, R].



Em resumo, os resultados dos testes deixam claro que o algoritmo genético é uma opção viável para resolver problemas de roteirização com vários pontos de entrega. Ele consegue equilibrar a busca por boas soluções com um tempo de execução bastante razoável. Sendo assim uma ferramenta que mostra sua relevância em cenários do dia a dia, podendo ser muito útil para melhorar processos logísticos e planejar rotas com eficácia, beneficiando diversas aplicações práticas.

5.3 Testes Realizados no algoritmo Genético para a entrada Berlim52.

A entrada “Berlim52” consiste em 52 pontos de visita na cidade de Berlim, Alemanha, cada um com coordenadas geográficas representadas em um espaço bidimensional (X e Y). O objetivo do é encontrar a rota mais curta que visita cada um desses pontos de visita uma única vez e retorna ao ponto de partida.

Os testes foram executados seguindo os seguintes parâmetros para o algoritmo:

Tamanho Da População	Gerações	N. Tentativas	Tamanho Corte
50	500	1000	0,95

Foram executados 30 testes para a entrada “Berlim52”, a seguir estão os testes cujo o número da semente é um múltiplo de 5:

Semente	Menor Distância em quilômetros	Tempo de Execução em segundos	Desvio Padrão
5	10188,2769	35,0733	2497,5460
10	11877,5326	34,5568	1945,0252
15	11607,8621	35,3267	1529,1996
20	9472,8952	38,1755	2500,9399
25	10997,2582	36,3269	2116,7178
30	11049,3045	35,8173	2150,8149

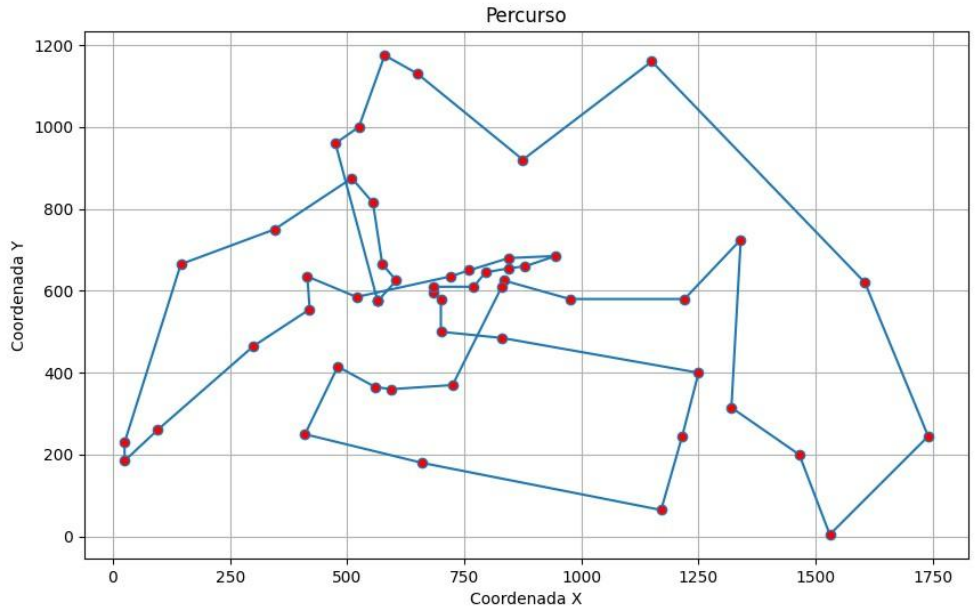
Em seguida foram calculadas a média da distância, tempo de execução e desvio padrão:

Média da distância em quilômetros	Média do tempo de execução em segundos	Média do desvio Padrão
10625,1869	37,1034	2208,9284

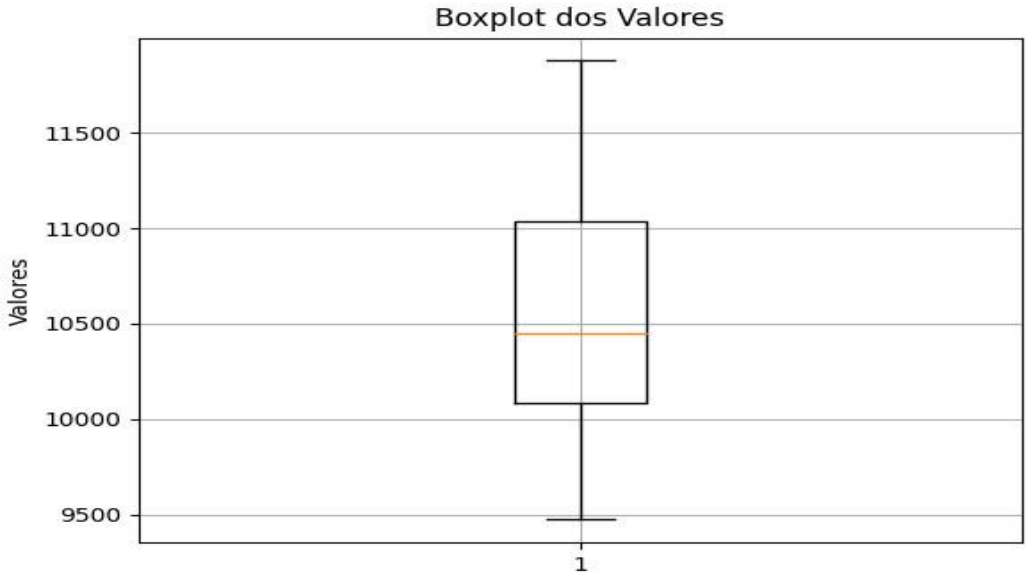
Ao final dos testes foi selecionado o melhor percurso:

Semente	Melhor distância em quilômetros	Tempo de execução em segundos	Desvio Padrão
20	9472,8952	38,1755	2500,9399

Plot do melhor percurso:



E em seguida foi feito o Boxplot que engloba todos os testes:



5.4 Testes Realizados no algoritmo Colônia de formigas

O Algoritmo de Colônia de Formigas (ACO) é uma meta-heurística inspirada no comportamento das formigas reais para resolver problemas de otimização (como já visto na seção 2.4). Ele é amplamente utilizado em uma variedade de domínios, incluindo o Problema do Caixeiro Viajante (PCV), roteamento de veículos e colorização de grafos. Neste relatório, exploraremos os resultados obtidos ao aplicar o ACO em diferentes instâncias do PCV com um conjunto de parâmetros específicos.

Para os experimentos realizados, foram utilizados os seguintes parâmetros:

t	α	β	$\eta(x,y)(t)$	Random Seed	Número de Formigas	Número de Interações
200	1.0	2.0	0.5	11	50	200

- Número de Formigas: 50 formigas foram utilizadas para cada execução do algoritmo.
- Número de Interações: O algoritmo foi executado por 200 iterações.
- Parâmetros α e β: Foram definidos α = 1.0 e β = 2.0, que controlam a influência relativa da trilha de feromônio e da informação heurística na seleção de caminhos pelas formigas.
- Informação Heurística: A informação heurística (η(x,y)(t)) foi definida como 0.5, indicando uma influência moderada das informações locais no processo de tomada de decisão das formigas.
- Semente Aleatória: A semente aleatória foi definida como 11 para garantir a reprodutibilidade dos resultados.

A seguir, apresentamos os resultados obtidos para diferentes instâncias do PCV com um número variável de pontos (de 5 a 12 pontos):

Pontos	Menor Distancia em Dronômetros	Tempo de Execução (segundos)
5	11.6344	4.3109
6	14.1065	8.0242
7	17.2688	8.9581
8	20.4311	10.3707
9	23.4667	10.4967
10	26.2727	11.6860
11	29.9525	11.8066
12	31.2277	13.1085

Os resultados dos testes podem variar consideravelmente de acordo com a configuração específica dos parâmetros empregados. É possível ajustar esses parâmetros, como o número de formigas, o número de iterações, os valores de α e β, para personalizar o desempenho do ACO em problemas específicos.

Os resultados do teste mostram que, à medida que o número de pontos aumenta, a menor distância encontrada também aumenta, o que é uma tendência natural em problemas de roteirização, onde o espaço de busca se torna mais complexo. O tempo de execução também aumenta à medida que o problema se torna mais complexo, o que é uma característica comum em algoritmos de otimização. No entanto, mesmo para problemas com 12 pontos, o tempo de execução permanece razoável, não ultrapassando os 13.1085 segundos. Isso demonstra a capacidade do ACO de encontrar soluções eficazes em um tempo aceitável.

Os experimentos realizados com diferentes números de pontos mostram que o ACO é capaz de lidar com problemas de roteirização de diferentes tamanhos, convergindo gradualmente para soluções de alta qualidade. A combinação de trilhas de feromônio, informação heurística e atualizações adequadas de feromônio torna o ACO uma abordagem versátil e eficaz para uma ampla variedade de problemas do mundo real.

5.5 Testes Realizados no algoritmo ACO para a entrada Berlim52.

Os testes foram executados seguindo os seguintes parâmetros para o algoritmo:

t	α	β	η(x,y)(t)	Random Seed	Número de Formigas	Número de Interações
200	1.0	2.0	0.5	11	50	200

Foram executados 20 testes para a entrada "Berlim52", a seguir estão os testes realizados:

Semente Aleatória	Melhor distância em quilômetros	Tempo de execução em segundos	desvio padrão
90	7549.2890	118.4160	703
60	7891.8254	129.6118	667
50	7988.1042	141.2717	313
45	7751.5180	147.3880	345
40	7871.3342	152.5703	370

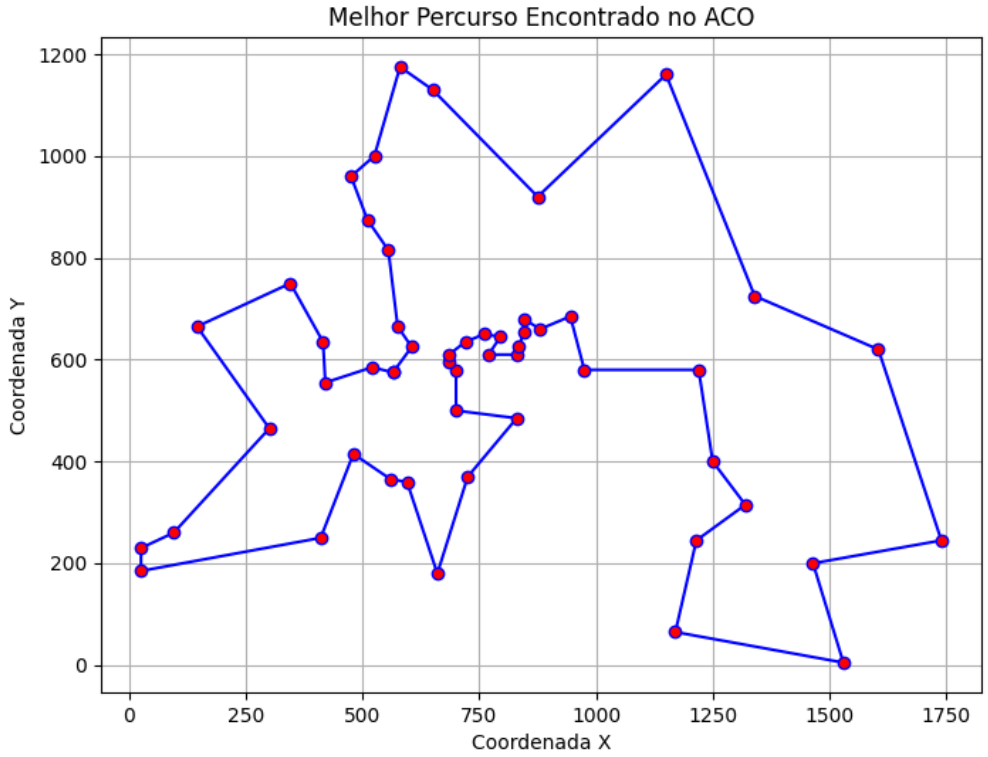
Em seguida foram calculadas a média da distância, tempo de execução e desvio padrão:

Semente Aleatória	Melhor distância em quilômetros	Tempo de execução em segundos	Média do desvio padrão
90	7549.2890	152.5703	1.219

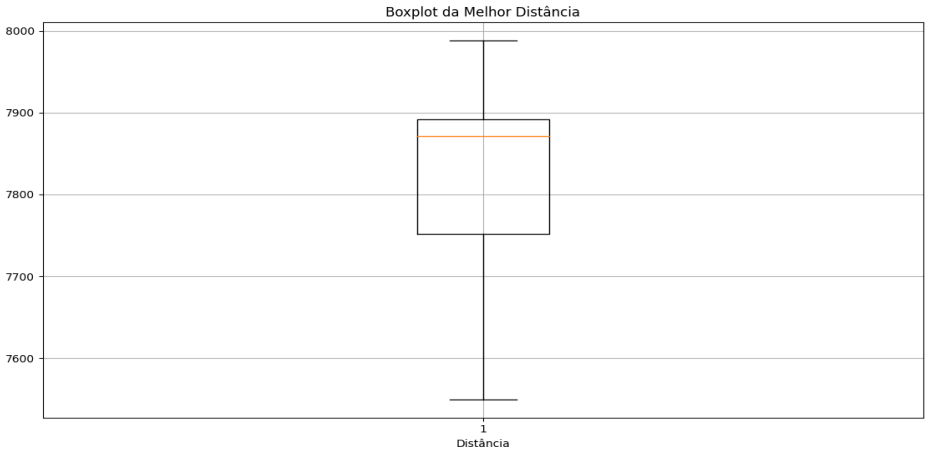
Ao final dos testes foi selecionado o melhor percurso:

Semente Aleatória	Melhor distância em quilômetros	Tempo de execução em segundos	desvio padrão
90	7549.2890	118.4160	703

Plot do melhor percurso:



E em seguida foi feito o Boxplot que engloba todos os testes:



5.6 Comparação entre o Algoritmo De Força Bruta e o Algoritmo Genético

Os testes foram conduzidos no site: replit.com. Os resultados dos testes foram registrados na tabela abaixo:

Algoritmo de Força Bruta		//	Algoritmo Genético	
Menor Distância em Dronômetros	Tempo de Execução em segundos	Pontos	Menor Distância em Dronômetros	Tempo de Execução em segundos
14	0,0003	5	11,6344	6,4495
18	0,0009	6	14,1065	6,7375
22	0,0074	7	17,2688	6,9177
26	0,2971	8	20,4311	7,0745
30	6,4127	9	23,4667	7,1812
34	15,1592	10	26,2727	7,1885
38	157,7150	11	29,9526	7,3118
--	----	12	31,2278	7,4115

O algoritmo de Força Bruta apresentou resultados precisos para encontrar a menor distância entre os pontos de entrega. Ele garantiu resultados ótimos, indicados pelas distâncias registradas.

No entanto, o tempo de execução aumenta de forma significativa à medida que o número de pontos de entrega cresce. A partir de 8 pontos, o tempo de execução começa a crescer exponencialmente, tornando-se inviável para roteiros maiores. O teste com 12 pontos não pôde ser concluído devido a limitações de memória.

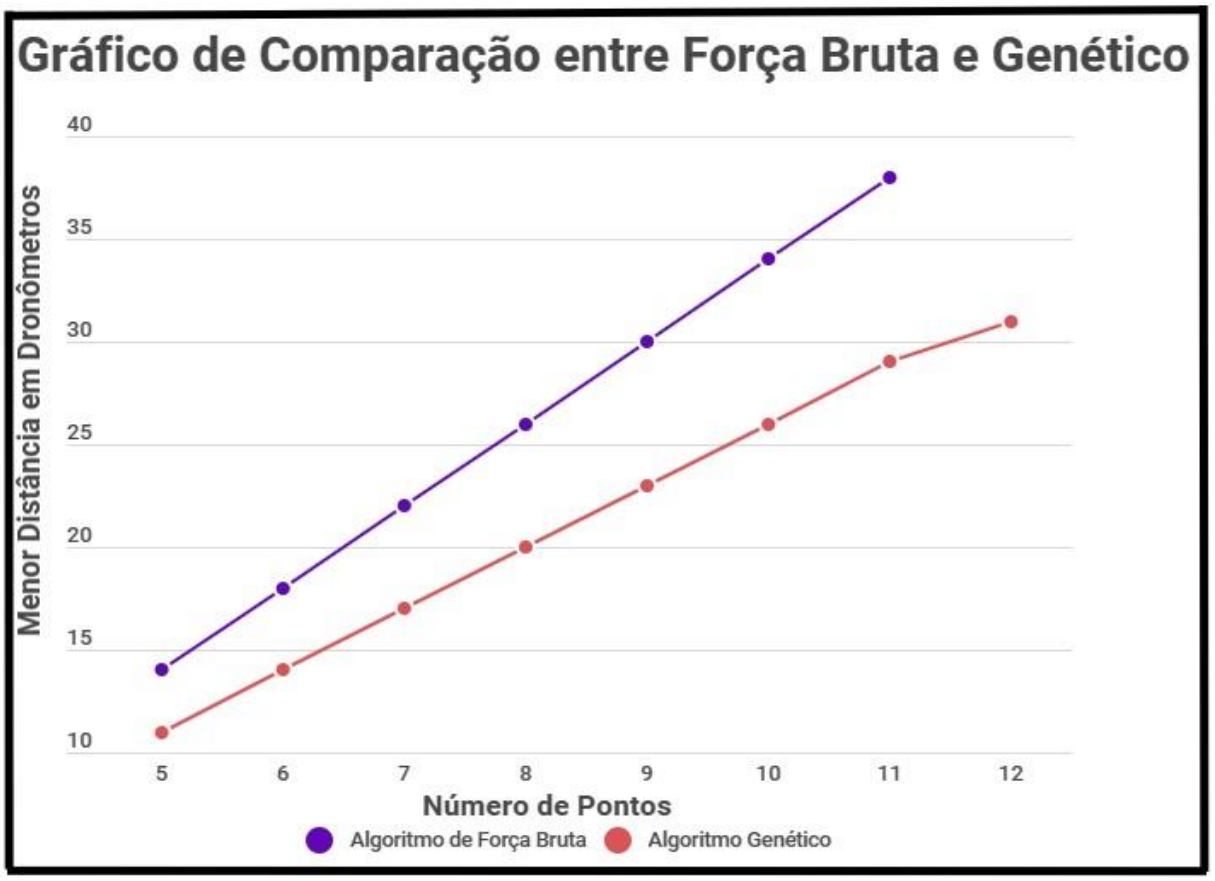


Gráfico de Comparação de Distância entre o Algoritmo de Força Bruta e o Algoritmo Genético

O algoritmo genético demonstrou uma notável capacidade de encontrar soluções com distâncias consideravelmente menores quando comparado ao algoritmo de Força Bruta. Essa superioridade se destacou em roteiros de tamanho moderado, abrangendo um espectro que varia até um determinado limite em termos do número de pontos de entrega.

O tempo de execução do algoritmo genético é razoável e não cresce exponencialmente com o aumento do número de pontos. Isso torna o algoritmo genético uma alternativa viável para roteirização com um número razoável de pontos de entrega.

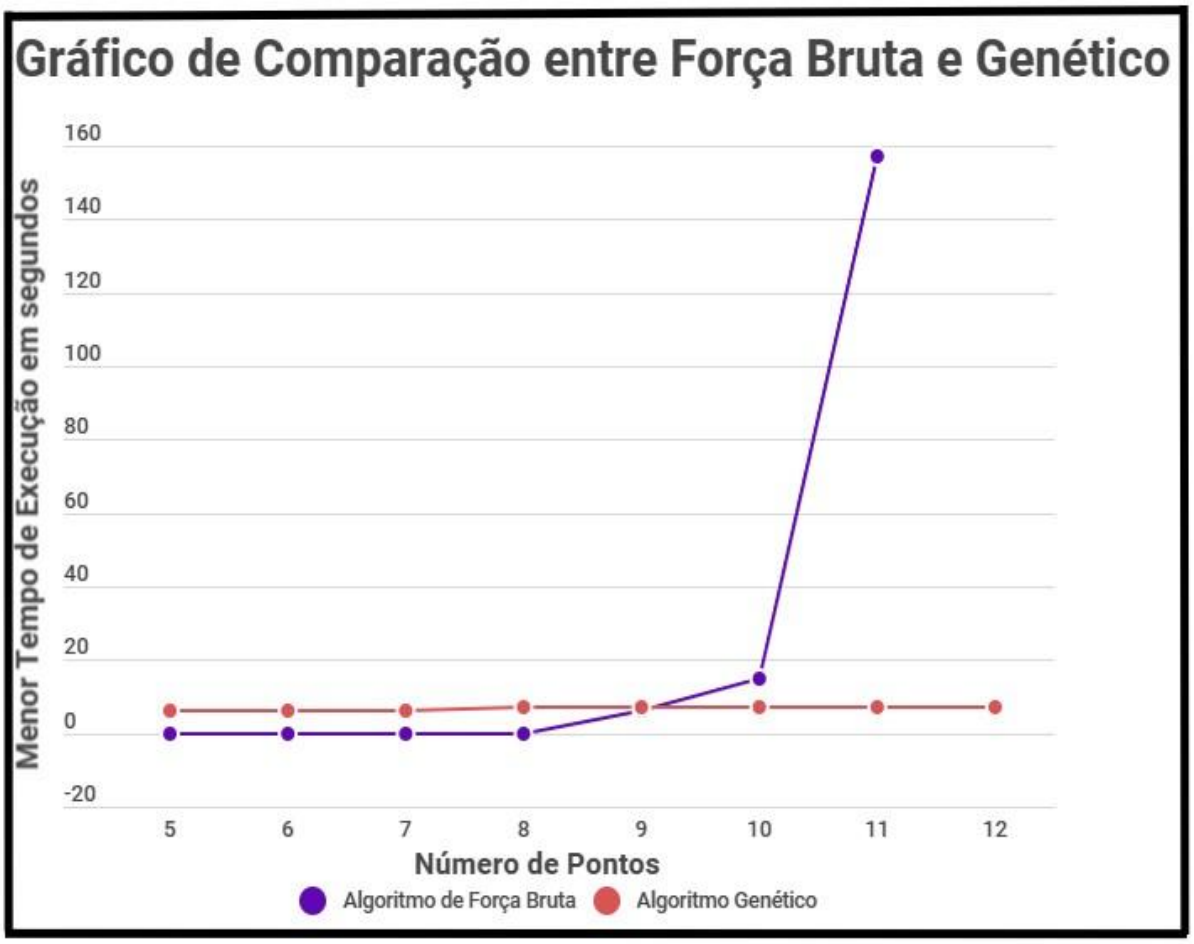


Gráfico de Comparação de Tempo entre o Algoritmo de Força Bruta e o Algoritmo Genético

Ambos os algoritmos apresentam limitações em cenários com um grande número de pontos de entrega. O algoritmo de Força Bruta oferece resultados ótimos, mas seu tempo de execução torna-se impraticável rapidamente. O algoritmo genético, embora não seja ótimo, encontra soluções aceitáveis com um tempo de execução razoável até um certo limite.

Portanto, a escolha entre esses algoritmos dependerá do tamanho do problema e das restrições de tempo disponíveis. Em resumo, o algoritmo genético é uma abordagem viável para problemas de roteirização com um número razoável de pontos de entrega, oferecendo um equilíbrio entre encontrar soluções de boa qualidade e tempo de execução razoável.

5.7 Comparação entre o Algoritmo De Força Bruta e o Algoritmo Colônia de Formigas

Os testes foram conduzidos no site: replit.com. Os resultados dos testes foram registrados na tabela abaixo:

Algoritmo de Força Bruta	//	Algoritmo Colônia De Formigas

Menor Distância em Dronômetros	Tempo de Execução em segundos	Pontos	Menor Distância em Dronômetros	Tempo de Execução em segundos
14	0,0003	5	11,6344	4.3109
18	0,0009	6	14,1065	8.0242
22	0,0074	7	17,2688	8.9581
26	0,2971	8	20,4311	10.3707
30	6,4127	9	23,4667	10.4967
34	15,1592	10	26,2727	11.6860
38	157,7150	11	29,9525	11.8066
--	----	12	31,2277	13.1085

O algoritmo de Força Bruta garante resultados precisos e ótimos, identificando a menor distância em dronômetros entre os pontos de entrega. Seu principal ponto fraco é o tempo de execução, que cresce de forma fatorial à medida que o número de pontos aumenta.O algoritmo de Força Bruta não conseguiu concluir o teste para 12 pontos de entrega devido a limitações de memória.

A Colônia de Formigas demonstra uma notável capacidade de encontrar soluções com distâncias significativamente menores em dronômetros em comparação ao algoritmo de Força Bruta, especialmente em cenários com 8 pontos ou mais.
Seu tempo de execução é razoável e cresce com o aumento do número de pontos de entrega, tornando-a adequada para problemas com um número razoável de pontos.
A Colônia de Formigas conseguiu concluir o teste para 12 pontos de entrega em um tempo de execução de 13,1085 segundos.

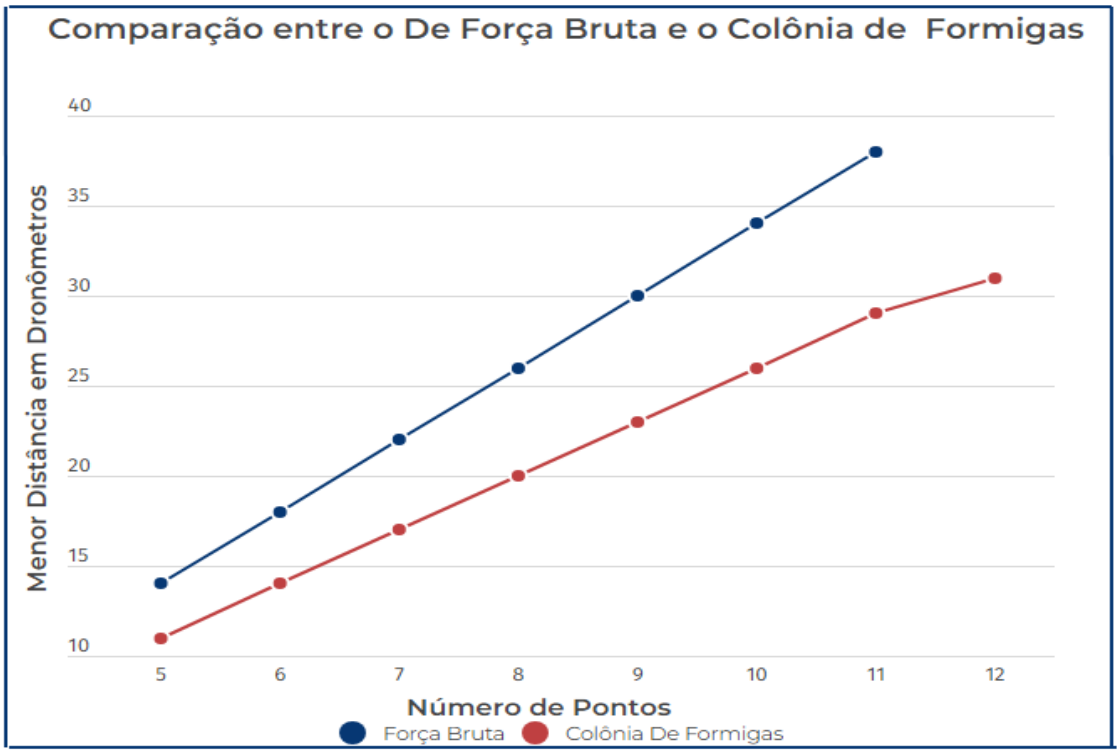


Gráfico de Comparação de Distância entre o Algoritmo de Força Bruta e o Algoritmo Colônia de Formigas

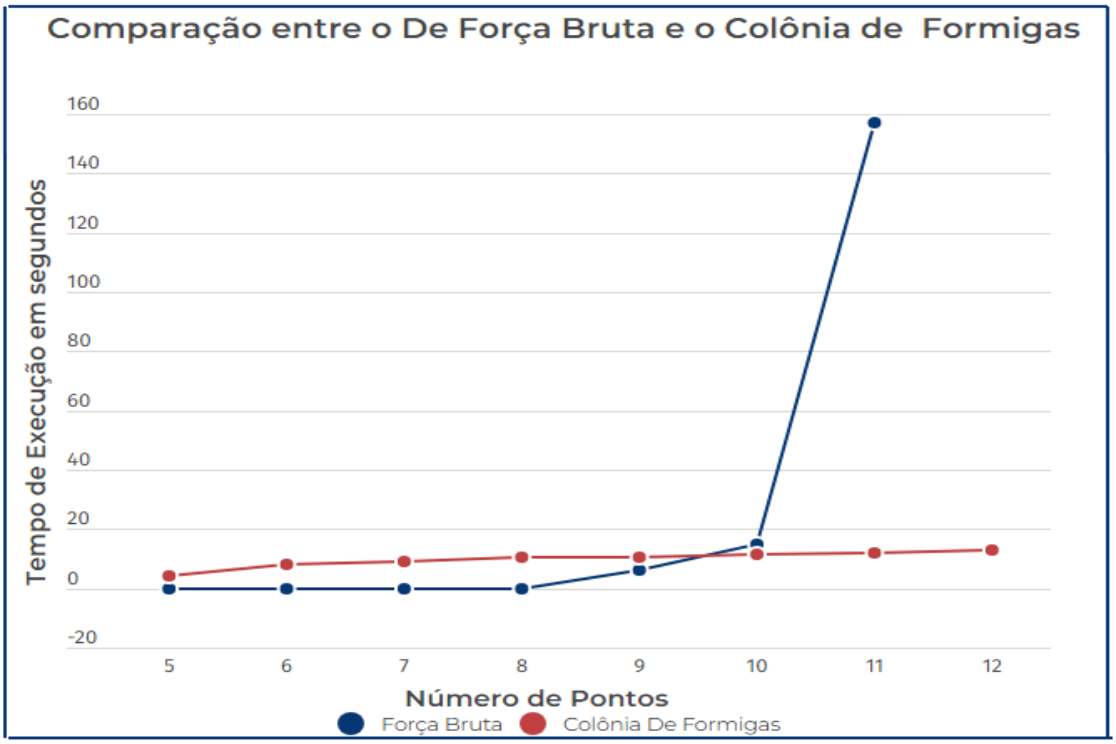


Gráfico de Comparação de Tempo entre o Algoritmo de Força Bruta e o Algoritmo Colônia de Formigas

Em resumo, o algoritmo de Colônia de Formigas supera o algoritmo de Força Bruta em termos de tempo de execução e ainda é capaz de encontrar soluções com distâncias competitivas em dronômetros. O Força Bruta é a escolha ideal para problemas pequenos e críticos em termos de precisão.Portanto, a escolha entre esses algoritmos dependerá principalmente do tamanho do problema e das restrições de tempo disponíveis

5.8 Comparação entre os Algoritmo Genético e o Algoritmo Da Colônia De Formigas

Os testes foram conduzidos no site: replit.com. Os resultados dos testes foram registrados na tabela abaixo:

Algoritmo Genético	//	Algoritmo Colônia De Formigas

Menor Distância em Dronômetros	Tempo de Execução em segundos	Pontos	Menor Distância em Dronômetros	Tempo de Execução em segundos
11,6344	6,4495	5	11,6344	4.3109
14,1065	6,7375	6	14,1065	8.0242
17,2688	6,9177	7	17,2688	8.9581
20,4311	7,0745	8	20,4311	10.3707
23,4667	7,1812	9	23,4667	10.4967
26,2727	7,1885	10	26,2727	11.6860
29,9526	7,3118	11	29,9525	11.8066
31,2278	7,4115	12	31,2277	13.1085

O Algoritmo Genético e a Colônia de Formigas mostram um desempenho muito semelhante em termos de menor distância para os casos de 5 e 10 pontos. No entanto, à medida que o número de pontos aumenta, a Colônia de Formigas se mostrou levemente superior ao Algoritmo Genético nos casos 11 e 12 pontos.

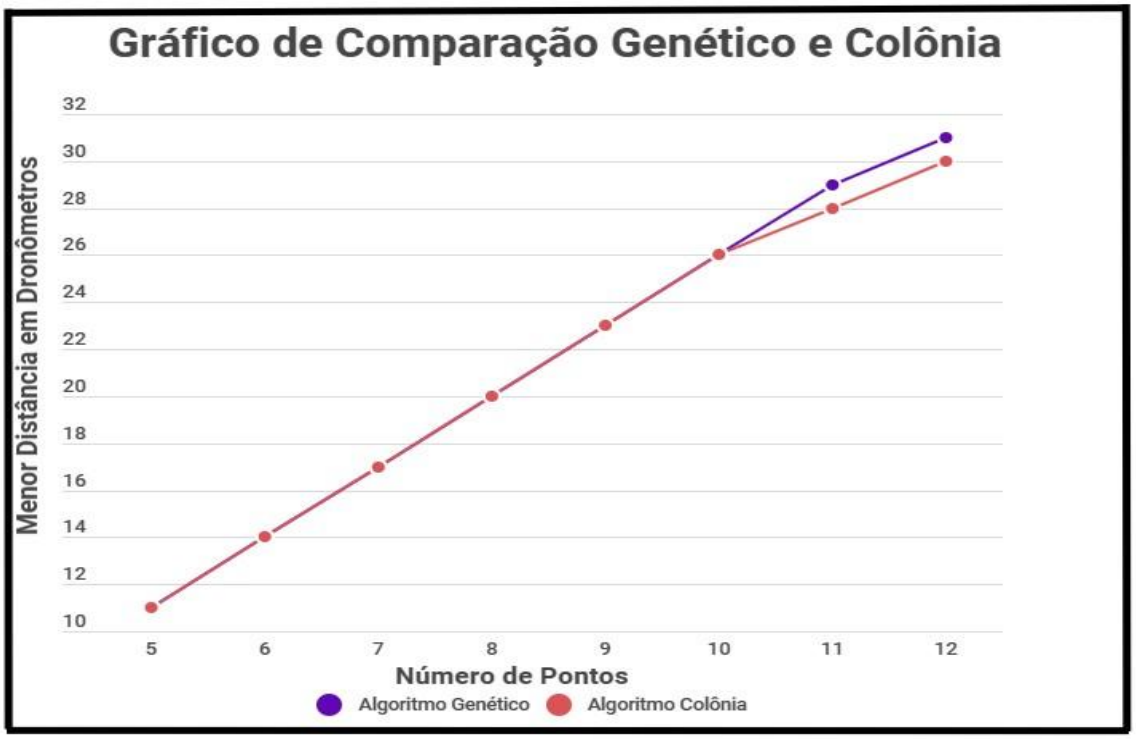


Gráfico De Comparação da Distância entre o Algoritmo Genético e o Algoritmo da Colônia de Formigas

O Algoritmo Genético é mais rápido na execução, tornando-o uma escolha apropriada quando o tempo é um fator crítico e as soluções de qualidade semelhantes são aceitáveis.

A Colônia de Formigas tende a ter tempos de execução mais longos, mas ainda assim razoáveis. É uma escolha adequada quando se busca otimizar a qualidade da solução e o tempo não é tão restrito.

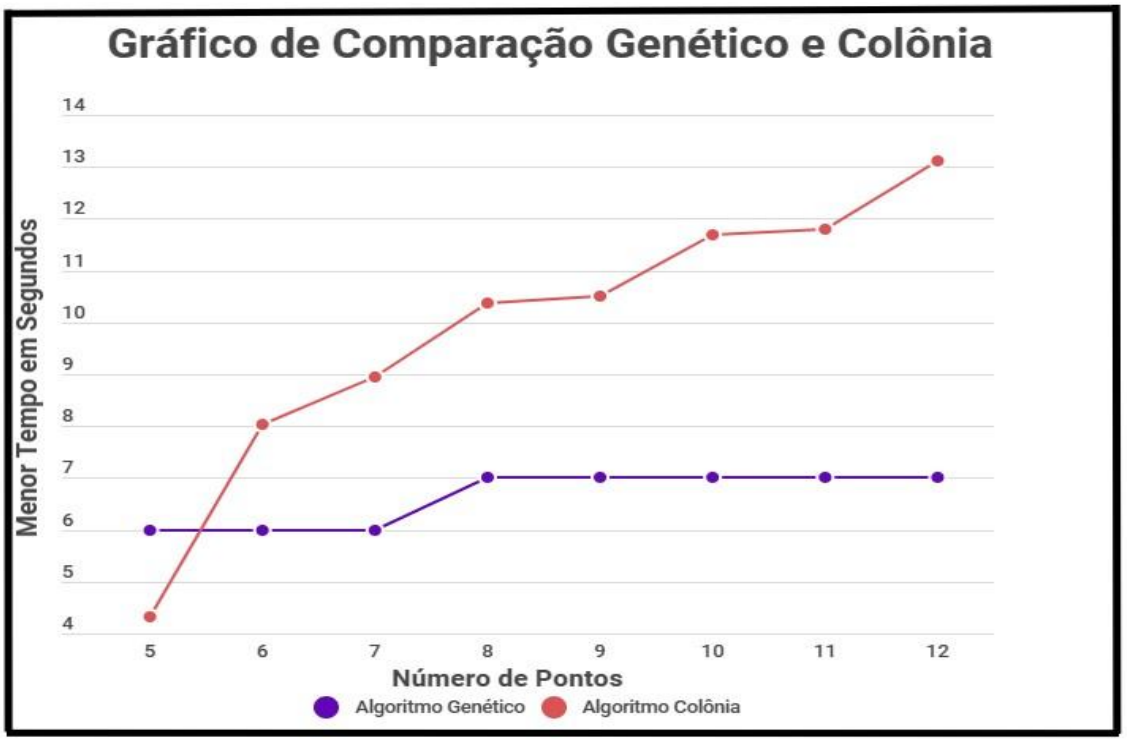


Gráfico De Comparação do Tempo entre o Algoritmo Genético e o Algoritmo da Colônia de Formigas

Em última , a escolha entre esses algoritmos depende dos requisitos específicos do problema. Se a prioridade for encontrar a melhor solução possível, a Colônia de Formigas pode ser a escolha preferida. No entanto, se o tempo de execução for crítico e soluções próximas da ótima forem aceitáveis, o Algoritmo Genético pode ser mais apropriado. Em última análise, a seleção do algoritmo depende de um equilíbrio entre qualidade da solução e restrições de tempo.

5.9 Comparação entre os Algoritmo Genético e o Algoritmo Da Colônia De Formigas para a entrada Berlin52

A eficiência desses algoritmos depende fortemente da escolha adequada de parâmetros, como o tamanho da população, número de gerações, taxas de evaporação, pesos alfa e beta, entre outros. Portanto, é fundamental ajustar esses parâmetros com base nas características específicas do problema e na computação disponível.

Os testes do algoritmo genético foram executados seguindo os seguintes parâmetros para o algoritmo Genético:

Semente	Tamanho Da População	Gerações	N. Tentativas	Tamanho Corte
20	50	500	1000	0.95

Os testes do algoritmo da colônia foram executados seguindo os seguintes parâmetros para o algoritmo Colônia:

Semente	Tamanho Da População	Iterações	Taxa evaporação do feromônio	Alfa	Beta
40	200	400	0.5	1.0	2.0

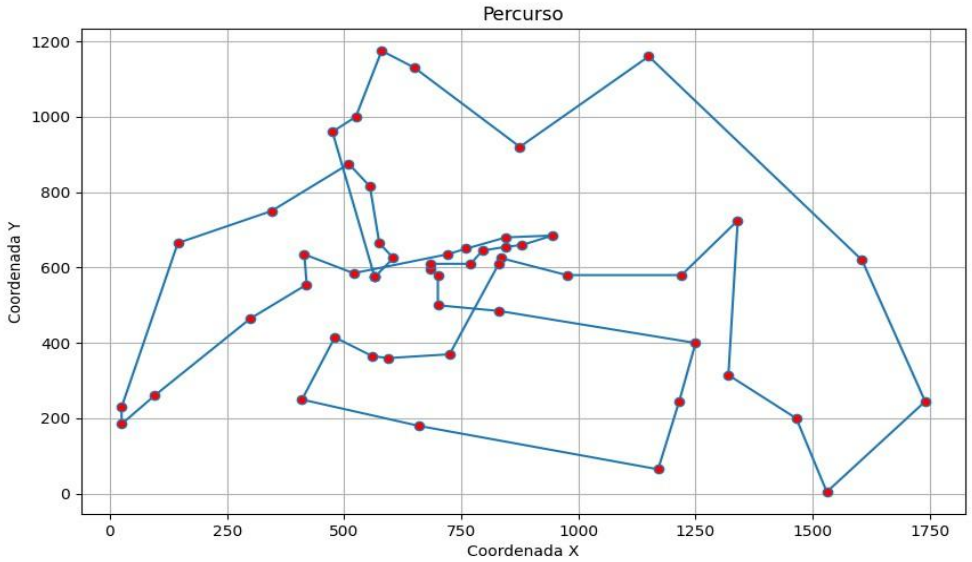
Os testes foram conduzidos em computadores pessoais. Os resultados dos testes foram registrados na tabela abaixo:

Algoritmo Genético		//	Algoritmo Colônia De Formigas	
Menor Distância em Dronômetros	Tempo de Execução em segundos	Pontos	Menor Distância em Dronômetros	Tempo de Execução em segundos
9472,8952	38,1755	52	7871,3342	22,6707

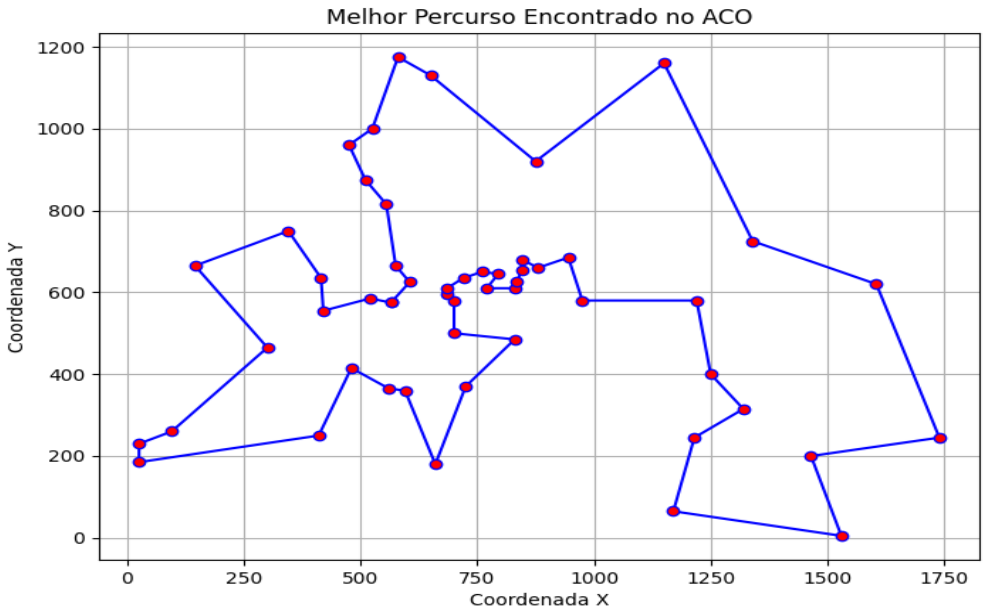
O Algoritmo da Colônia de Formigas obteve um desempenho notável ao encontrar uma solução com uma distância significativamente menor (7871,3342 unidades de distância) em comparação com o Algoritmo Genético (9472,8952 unidades de distância). Isso sugere que, para a instância "Berlim52" deste problema, o algoritmo de colônia de formigas foi capaz de encontrar uma rota mais curta em um espaço de tempo muito mais curto.

Embora o Algoritmo Genético tenha alcançado um resultado razoável, ele levou consideravelmente mais tempo (38,1735 segundos) para chegar a essa solução em comparação com o Algoritmo da Colônia de Formigas (22,6707 segundos). Isso indica que o Algoritmo da Colônia de Formigas é mais eficiente em termos de tempo de execução para esta entrada específica do problema.

Melhor distância do ACO no Berlin52



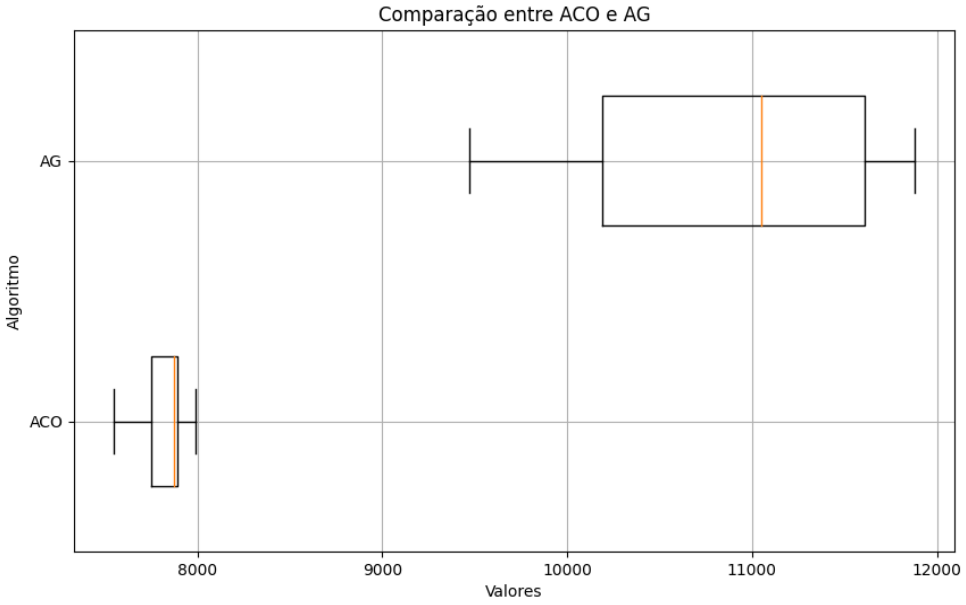
Melhor distância algoritmo genético



Melhor distância Colônia de formigas

Em suma, os resultados destacam a capacidade do Algoritmo da Colônia de Formigas em encontrar soluções de alta qualidade para a instância "Berlin52", com uma distância menor e um tempo de execução mais curto em comparação com o Algoritmo Genético. Portanto, para esse problema específico, o Algoritmo da Colônia de Formigas é a escolha preferida, proporcionando uma solução de alta qualidade de forma eficiente em termos de tempo.

Boxplot que engloba todos os testes:



6. Resultados

Analisando os resultados, fica evidente que o algoritmo de Força Bruta é eficaz para resolver o problema até um certo limite de pontos de entrega. Nos três primeiros casos de teste, o algoritmo conseguiu encontrar o menor percurso em tempo razoável, apresentando a resposta rapidamente. No entanto, ao atingir o sexto caso de teste, o tempo de execução aumentou drasticamente, saindo de milissegundos para três segundos e logo no próximo caso já atingindo quase um minuto.

O Algoritmo Genético (AG) demonstrou eficácia na busca por soluções com distâncias inferiores ao algoritmo de força bruta, pelo menos até um certo limite. Esse desempenho reflete sua capacidade de explorar diferentes soluções e identificar ótimos locais de maneira eficiente. No entanto, é importante notar que o tempo de execução do AG aumentou gradualmente à medida que o número de pontos de entrega cresceu, sugerindo sua adequação para problemas de tamanho moderado, mas perdendo eficiência em instâncias mais complexas. Embora o AG, como abordagem probabilística, não garanta soluções ótimas, oferece um equilíbrio satisfatório entre qualidade da solução e tempo de execução, tornando-se uma escolha sólida para problemas PCV com um número moderado de pontos de entrega.

O Algoritmo Colônia de Formigas (ACO) se destacou ao encontrar soluções com distâncias consideravelmente menores em comparação com o algoritmo de Força Bruta, especialmente em cenários com 8 pontos ou mais. Embora tenha tempos de execução mais longos em comparação com o Algoritmo Genético, eles ainda são razoáveis. A ACO conseguiu concluir o teste para 12 pontos de entrega em um tempo de execução de 13,1085 segundos. Isso o torna uma escolha adequada para o problema específico do Flyfood.

Portanto os três algoritmos, Força Bruta, Algoritmo Genético e Colônia de Formigas, abordam o Problema do FlyFood de maneiras distintas. O Força Bruta garante soluções ótimas, mas seu tempo de execução fatorial o limita a conjuntos de dados pequenos. O Algoritmo Genético oferece um equilíbrio entre qualidade da solução e tempo de execução, sendo eficaz para conjuntos de dados moderados. Já a Colônia de Formigas, busca soluções de alta qualidade de forma eficiente. A escolha entre eles depende das necessidades do problema e dos recursos disponíveis. Finalizando, cada algoritmo tem seu lugar em diferentes cenários, proporcionando soluções viáveis, dependendo do tamanho e complexidade do conjunto de dados, bem como dos recursos computacionais disponíveis.

7. Conclusão

7.1 Objetivos do Trabalho

O objetivo do trabalho é desenvolver um algoritmo que seja capaz de definir o menor trajeto para a realização de todas as entregas do drone.O algoritmo deve ser capaz de ler uma matriz, a partir de um arquivo, com os pontos de entrega e o ponto de origem e retorno. Ele deverá retornar a ordem em que o drone deve percorrer os pontos de entrega. Essa ordem deve ser a de menor custo, ou seja, a que o drone vá percorrer a menor distância em dronômetros.

7.2 Resultados alcançados

Os resultados alcançados foram os seguintes: O algoritmo foi capaz de encontrar o menor percurso para os três primeiros casos de teste em tempo razoável.

- O tempo de execução aumentou drasticamente para o sétimo caso de teste, demorando 79.24 segundos para encontrar a solução.
- O limite de pontos para o funcionamento eficiente do algoritmo é em torno de 11 pontos de entrega.
- O algoritmo utiliza uma abordagem de força bruta, que é ineficiente para um grande número de pontos.
- Dependendo das restrições e exigências do problema, algoritmos heurísticos podem oferecer soluções aceitáveis em tempo viável.

Já para o Algoritmo Genético e a Colônia de Formigas foram capazes de suportar uma quantidade maior de pontos comparada com o algoritmo de força bruta, embora não entreguem resultados ótimos.

- Tempo de execução melhor com grandes quantidades de pontos
- Suporta uma grande quantidade de pontos como por exemplo a do Berlin52

7.3 Conclusões Obtidas

Os resultados obtidos indicam que o algoritmo de força bruta é eficaz para resolver o problema de roteamento de drones para um pequeno número de pontos de entrega, mas não é viável para um grande número de pontos. Em cenários da vida real com um grande número de pontos de entrega, é importante considerar alternativas mais eficientes, como algoritmos heurísticos.

conclusões que podem ser tiradas dos resultados obtidos:

- O algoritmo de força bruta é eficaz para resolver o problema de roteamento de drones para um pequeno número de pontos de entrega.
- O tempo de execução do algoritmo de força bruta aumenta de forma exponencial conforme o número de pontos de entrega aumenta.
- O limite de pontos para o funcionamento eficiente do algoritmo de força bruta é em torno de 11 pontos de entrega.
- Algoritmos heurísticos podem oferecer soluções aceitáveis em tempo viável para um grande número de pontos de entrega.

Com Meta-heurísticas, embora não consigamos obter uma resposta ótima, vamos obter uma resposta satisfatória, assim, para entradas pequenas é recomendado o uso da força bruta já que nos trará uma resposta ótima, para entradas que sejam muito grande o recomendado é o uso de metaheurísticas globais.

- **Eficiência em Problemas Complexos:** Algoritmos genéticos e colônia de formigas são mais eficientes do que a força bruta em lidar com problemas complexos, nos quais o espaço de busca é muito grande ou mal estruturado. Isso ocorre porque eles exploram múltiplas soluções simultaneamente e podem convergir para soluções melhores ao longo do tempo.
- **Exploração mais Inteligente:** Ambas as abordagens, algoritmos genéticos e colônia de formigas, incorporam mecanismos de exploração inteligente. Os algoritmos genéticos usam seleção, recombinação e mutação para explorar o espaço de busca de maneira mais eficiente, enquanto a colônia de formigas utiliza feromônios para comunicar informações sobre soluções promissoras.
- **Convergência para Soluções Ótimas:** Tanto os algoritmos genéticos quanto a colônia de formigas têm a capacidade de convergir para soluções ótimas à medida que o algoritmo evolui ou as formigas exploram o ambiente. Isso permite encontrar soluções melhores à medida que o tempo passa.
- **Robustez:** Essas abordagens são geralmente mais robustas em relação a variações nos dados de entrada e podem lidar com problemas que podem não ser abordados de maneira eficaz pela força bruta, que testa todas as soluções.

- **Paralelismo Natural:** Tanto os algoritmos genéticos quanto a colônia de formigas podem ser paralelizados de forma eficiente, o que permite acelerar o processo de otimização em hardware moderno com várias CPUs ou GPUs.
- **Aplicabilidade a Diversos Domínios:** Essas abordagens são amplamente aplicáveis a uma variedade de domínios, desde problemas de roteamento até otimização de parâmetros em aprendizado de máquina, o que as torna versáteis em muitas situações.

7.4 Trabalhos Futuros

É importante notar que esses são apenas alguns exemplos de possíveis trabalhos futuros. Existem muitas outras maneiras de melhorar a eficiência e a eficácia do processo de roteamento de drones.

- Melhorar a eficiência do algoritmo de força bruta para lidar com um grande número de pontos de entrega.
- Implementar algoritmos de aprendizado de máquina para aprender a criar rotas de drones eficientes.
- Considerar a influência do clima e de outros fatores no processo de roteamento de drones.
- Desenvolver ferramentas para ajudar os usuários a criar rotas de drones eficientes.
- Usar metaheurísticas diferentes para que uma complemente a outra.
- Investigar como os sistemas de drones podem ser integrados de forma eficiente com cadeias de suprimentos e operações logísticas de empresas para melhorar a entrega de mercadorias.
- Estudar e implementar redes de comunicação em malha para drones, permitindo uma comunicação eficaz entre eles e com centros de controle.
- Focar em estratégias de roteamento específicas para ambientes urbanos, onde os desafios de navegação são mais complexos devido à presença de edifícios, tráfego e restrições de espaço.
- Desenvolver algoritmos que otimizem múltiplos objetivos, como minimizar o tempo de entrega, reduzir o consumo de energia dos drones e maximizar a satisfação do cliente.
- Pesquisar maneiras de tornar o roteamento de drones mais ecologicamente correto, considerando a pegada de carbono e propondo soluções para reduzir o impacto ambiental.

Referências Bibliográficas

BARCELLOS, João Carlos Holland de. Algoritmos Genéticos Adaptativos: Um estudo Comparativo. 2000. 143 f. Dissertação (Mestrado em Engenharia) –Universidade Federal de Lavras, Belo Horizonte, 2002.

BARRICHELLO, Leonardo; RODRIGUES, Claudina Izepe; COSTA, Sueli I. Geometria do Táxi: Distâncias. Matemática multimídia, Campinas. Disponível em .

Bispo, R. C. (2018), "Planejador de Roteiros Turísticos: Uma Aplicação do Problema do Caixeiro Viajante na Cidade do Recife". Universidade Federal Rural de Pernambuco UFRPE, Curso de Bacharelado em Sistemas de Informação, Recife. (Trabalho de conclusão de curso)

Boaventura Netto, P. O. e Jurkiewicz, S. (2009) Grafos: Introdução e Prática. São Paulo, Editora Blucher, 2ª ed.

CARVALHO, André Ponce de Leon F. de. Algoritmos Genéticos. Department of Computer Science. Disponível em: https://sites.icmc.usp.br/andre/research/genetic/

Cormen, T. H.; C. E. Leiserson, R. L. Rivest e C. Stein (2009). Introduction to Algorithms. 3rd. The MIT Press.

COSTA, D.; HERTZ, A. Ants can color graphs. Journal of the Operational Research Society, v. 48, p. 295-305, 1997.

Dekkers A. and Aarts E. (1991) "Global optimization and simulated annealing".

DORIGO, M.; GAMBARDELLA, L. M. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. IEEE Transactions on Evolutionary Computation, v. 1, n. 1, p. 53-66, 1997.

DORIGO, M., STÜTZLE, T. Ant Colony Optimization. 1. ed. Massachusetts: The MIT Press, 2004. 305p.

FAVA NETO, Irineu. Um novo conceito de distância: a distância do táxi e aplicações. 2013. 46 f. Dissertação (mestrado) - Universidade Estadual Paulista

Gelfand, S. B. and Mitter, S. K. (1985) "Analysis Of Simulated Annealing For Optimization"

Halim, A.H., Ismail, I. (2019) "Tree physiology optimization on SISO and MIMO PID control tuning

HOSEINI, P.; SHAYESTEH, M.G. Hybrid Ant Colony Optimization, Genetic Algorithm, and Simulated Annealing for Image Contrast Enhancement. Evolutionary Computation (CEC), 2010.

Hui, Wang (2012) "Comparison of several intelligent algorithms for solving TSP problems in industrial engineering".

KRAUSE, Eugene F. Taxicab Geometry. New York: Dover, 1986.

LEVINE, J.; DUCATELLE, F. Ant Colony Optimisation and Local Search for Bin Packing and Cutting Stock Problems. Journal of the Operational Research Society, p. 93, 2003.

PACHECO, Marco Aurélio Cavalcanti. Algoritmos Genéticos: Princípios E Aplicações. ICA: Laboratório de Inteligência Computacional Aplicada, Rio de Janeiro. Disponível em: http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf

REIMANN, M.; STUMMER, M.; DOERNER, K. A savings based Ant System for the vehicle routing problem.Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002). Morgan Kaufman, p. 1317-1325, 2002.

Štencek, J. (2013), "Traveling salesman problem Degree Programme in Software engineering". JAMK University of Applied sciences, Degree Programme in Software engineering (Bachelor's Thesis).

SIMÃO, Thiago Dias. Utilização de Algoritmos Genéticos para Otimização de Soluções para o Timetabling escolar. 2013. 73 f. Monografia (Graduação em Ciência da Computação) – Universidade de São Paulo, São Paulo, 2000.

TIMÓTEO, Guilherme Tadeu Silva. Desenvolvimento de um Algoritmo Genético para a resolução do Timetabling. 2002. 144 f. Monografia (Graduação em Ciência da Computação) – Universidade Federal de Lavras, Belo Horizonte, 2002.

ZHANG, D.; DU, L. Hybrid Ant Colony Optimization Based on Genetic Algorithm for Container Loading Problem. International Conference of Soft Computing and Pattern Recognition (SoCPaR). p. 10-14, 2011.