



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

Otimização de Rotas: Uma Abordagem Eficiente Utilizando Metaheurísticas

Ellen Caroliny Tavares

Evny Vitória Pereira

Recife

Setembro de 2024

Resumo

O problema da otimização de rotas de entrega é um desafio crucial em logística, especialmente em áreas urbanas densamente povoadas. Neste trabalho, é apresentado o Flyfood, um sistema de entrega por drones que busca revolucionar o setor, oferecendo entregas mais rápidas e eficientes. Inspirado no *Traveling Salesman Problem* (Problema do Caixeiro Viajante), um clássico da otimização combinatória, que busca encontrar a rota mais curta que visite cada cidade exatamente uma vez e retorne ao ponto de partida. Nesse sentido, devido à sua natureza NP-completa, a resolução por força bruta torna-se inviável para instâncias de grande porte. Dessa maneira, foi explorado o uso de metaheurísticas, como o Algoritmo Genético e a Busca Tabu, para abordar essa complexidade. Essas técnicas de otimização heurística oferecem uma alternativa eficiente para encontrar soluções de boa qualidade em tempo razoável. Os resultados desta pesquisa demonstram o potencial das metaheurísticas como ferramentas poderosas para resolver problemas de otimização complexos, demonstrando serem mais eficazes nestes casos do que o algoritmo de força bruta, ainda que não apresentem uma solução ótima.

Palavras-chave: Problema do Caixeiro Viajante; Entrega por drones; Algoritmo Genético; Busca Tabu; Metaheurísticas; Otimização de rotas.

1. Introdução

1.1 Apresentação e Motivação

Diante do cenário atual, marcado pela crescente demanda por serviços de entregas rápidas e eficientes, propõe-se uma solução inovadora que visa reduzir o tempo das entregas e os custos associados, otimizando todas as etapas do processo de delivery. Essa proposta baseia-se na utilização de drones como meio de transporte aéreo, com o objetivo de diminuir o impacto ambiental, reduzir a dependência de veículos terrestres e mitigar problemas como congestionamentos.

No entanto, existem lacunas significativas nesse contexto. A eficiência logística atual é limitada, devido a congestionamentos urbanos, restrições de trânsito e altos custos operacionais. Além disso, a dependência de veículos terrestres contribui para o aumento das emissões de carbono, exacerbando os impactos ambientais. Embora o uso de drones para entregas ofereça uma alternativa promissora, ainda enfrentamos desafios tecnológicos relacionados à autonomia, capacidade de carga e navegação em ambientes complexos.

Logo, nesse contexto, insere-se o desenvolvimento do projeto Flyfood, inspirado no problema do caixeiro viajante. O projeto busca implementar um algoritmo de roteamento capaz de determinar as rotas de entrega mais eficientes, a partir do mapeamento detalhado das regiões. As entregas, realizadas por drones, têm como finalidade otimizar e transformar o mercado de delivery, oferecendo uma alternativa sustentável e tecnologicamente avançada.

1.2 Formulação do problema

Objetivo: Desenvolver um software para encontrar a rota ótima para entregas por drones no projeto FlyFood, minimizando a distância total percorrida entre os pontos de entrega.

Conceitos

1. Matriz de Entrada (Matriz de Distâncias):

- o **Arquivo Berlin52:** O arquivo berlin52 da TSPLib contém dados de coordenadas para 52 cidades na cidade de Berlim. Cada linha do arquivo fornece as coordenadas (x_i, y_i) de cada cidade.
- o **Matriz de Distâncias (D):** Uma matriz $n \times n$ ($n = 52$ para o arquivo berlin52) onde o elemento D_{ij} representa a distância entre a cidade i e a cidade j .

2. Coordenadas:

○ Cada cidade i é representada por um par de coordenadas (x_i, y_i) fornecido no arquivo berlin52.

3. **Pontos de Entrega:**

○ São os locais que precisam ser visitados, no caso em questão, cada ponto consiste em uma cidade, e cada ponto de entrega tem uma posição específica representada por suas coordenadas.

4. **Ponto de Partida:**

○ No PCV, o ponto de partida é o mesmo que o ponto de retorno, ou seja, a rota começa e termina no mesmo ponto.

Equações

1. **Distância entre Duas Cidades:**

○ A distância euclidiana D_{ij} entre duas cidades i e j com coordenadas (X_i, Y_i) , (X_j, Y_j) é calculada por:

$$D_{ij} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}$$

Onde:

- D_{ij} é a distância entre as cidades i e j ,
- X_i, Y_i são as coordenadas da cidade i ,
- X_j, Y_j são as coordenadas da cidade j .

○

2. **Custo de uma Solução:**

○ Uma solução é uma permutação π das cidades. O custo total $C(\pi)$ da rota que visita todas as cidades na ordem dada pela permutação π e retorna ao ponto inicial é calculado por:

$$C(\pi) = \sum_{i=1}^{n-1} D(\pi_i, \pi_{i+1}) + D(\pi_n, \pi_1)$$

Onde:

- $C(\pi)$ é o custo total da rota,
- π representa a permutação (rota) das cidades,
- π_i é a i -ésima cidade na permutação,
- $D(\pi_i, \pi_{i+1})$ é a distância entre as cidades consecutivas i e $i + 1$,
- $D(\pi_n, \pi_1)$ fecha o ciclo, conectando a última cidade n à primeira cidade na rota.

○

3. Equação do Problema:

- O objetivo é minimizar o custo total $C(\pi)$, que é a soma das distâncias entre cidades consecutivas na permutação e a distância de volta ao ponto de partida:

$$\min_{\pi} C(\pi) = \sum_{i=1}^{n-1} D(\pi_i, \pi_{i+1}) + D(\pi_n, \pi_1)$$

Onde:

- $C(\pi)$ é o custo total da rota,
- π é uma permutação das cidades $\{1, 2, \dots, n\}$,
- $D(\pi_i, \pi_{i+1})$ é a distância entre as cidades i e $i + 1$,
- $D(\pi_n, \pi_1)$ é a distância entre a última cidade n e a primeira cidade, fechando o ciclo.

1.3 Objetivos

1.3.1 Objetivo geral

O objetivo geral deste trabalho é otimizar as rotas de entrega por drones para o projeto FlyFood, utilizando o Problema do Caixeiro Viajante como base para a otimização. O trabalho visa comparar a eficácia de dois métodos de solução, o algoritmo genético e o algoritmo de busca tabu, para identificar a abordagem que melhor resolve o problema de forma eficiente. A meta é determinar a melhor estratégia para minimizar a distância total percorrida entre os pontos de entrega, melhorando a eficiência operacional e reduzindo os custos de operação do sistema de entrega por drones.

1. Objetivos específicos

1.1. Implementar e Avaliar o Algoritmo Genético:

- Desenvolver e implementar um algoritmo genético adaptado para resolver o Problema do Caixeiro Viajante, considerando as particularidades do projeto FlyFood.
- Avaliar o desempenho do algoritmo genético em termos de qualidade das soluções encontradas, tempo de execução e capacidade de encontrar soluções próximas ao ótimo global.
- Comparar os resultados obtidos com os resultados do algoritmo de busca tabu.

1.2. Implementar e Avaliar o Algoritmo de Busca Tabu:

- Desenvolver e implementar um algoritmo de busca tabu para o Problema do Caixeiro Viajante, ajustando-o para as necessidades específicas do projeto FlyFood.

- Avaliar a eficácia do algoritmo de busca tabu em termos de qualidade das soluções, tempo de computação e capacidade de evitar mínimos locais.
- Comparar os resultados obtidos com os resultados do algoritmo genético.

1.3. Comparar Desempenho dos Algoritmos:

- Comparar a eficácia do algoritmo genético e do algoritmo de busca tabu com base na qualidade das soluções encontradas, tempo de execução e eficiência geral.
- Identificar as vantagens e limitações de cada algoritmo na otimização das rotas de entrega por drones.

1.4. Aplicar as Soluções no Contexto do FlyFood:

- Implementar a solução otimizada no contexto prático do projeto FlyFood, avaliando a viabilidade e impacto das rotas otimizadas nas operações reais de entrega por drones.
- Avaliar o impacto das rotas otimizadas na eficiência operacional e na redução dos custos do projeto.

1.4 Organização do trabalho

O trabalho está organizado em partes fundamentais, cada uma abordando um aspecto crucial do desenvolvimento e avaliação dos algoritmos de otimização para o problema do Caixeiro Viajante (TSP).

1.4.1 Desenvolvimento e Implementação dos Algoritmos

1. Algoritmo Genético:

-Desenvolvimento: O algoritmo Genético foi desenvolvido para encontrar soluções aproximadas para o TSP através da simulação de processos evolutivos naturais, como seleção, cruzamento e mutação.

-Implementação: A implementação inclui a codificação das operações genéticas, a definição da função de aptidão, e a configuração dos parâmetros do algoritmo, como taxa de mutação e cruzamento.

2. Algoritmo de Busca Tabu:

-Desenvolvimento: O algoritmo de Busca Tabu foi projetado para otimizar a rota de entregas utilizando técnicas de busca local e uma estrutura de lista tabu para evitar ciclos e soluções já visitadas.

-Implementação: A implementação inclui a definição das operações de vizinhança (como inversão de subrotas), a configuração dos parâmetros da busca tabu, e a

integração do algoritmo com um arquivo de entrada contendo as coordenadas das cidades.

1.4.2 Fase de Testes

- **Testes Unitários:** Cada componente dos algoritmos foi testado individualmente para garantir que as funções estejam operando conforme o esperado.
- **Testes de Integração:** Foram realizados testes para verificar a integração entre os diferentes componentes dos algoritmos e a correta leitura dos dados do arquivo de coordenadas.
- **Validação dos Algoritmos:** Testes foram executados para validar a eficácia dos algoritmos, incluindo a comparação das soluções geradas com as soluções conhecidas ou esperadas.

1.4.3 Comparação dos Resultados

- **Execução Múltipla:** O algoritmo de Busca Tabu foi executado várias vezes com diferentes seeds para garantir a robustez dos resultados. As execuções foram analisadas para verificar a consistência e a qualidade das soluções encontradas.
- **Análise Comparativa:** Os resultados dos algoritmos Genético e Busca Tabu foram comparados em termos de eficiência, qualidade das soluções e tempo de execução. Gráficos foram gerados para visualizar as médias do tempo e do percurso mínimo obtido.

1.4.4 Análise de Cada Passo

- **Análise do Algoritmo Genético:** Inclui a revisão do desempenho do algoritmo Genético, discussão sobre a escolha dos parâmetros e a análise dos resultados obtidos.
- **Análise do Algoritmo de Busca Tabu:** Examina o comportamento do algoritmo de Busca Tabu, incluindo a eficiência da lista tabu e a capacidade do algoritmo em evitar ciclos e encontrar soluções ótimas.
- **Relatório de Resultados:** A análise dos resultados inclui a documentação dos testes realizados, a discussão sobre as diferenças observadas entre os algoritmos e a conclusão sobre a melhor abordagem para o problema do Caixeiro Viajante.

2. Referencial Teórico

2.1 NP-completude

Problemas NP-completos se caracterizam por uma explosão combinatória, ou seja, o número de possíveis soluções cresce exponencialmente com o tamanho da

entrada. Essa característica torna a busca por soluções ótimas computacionalmente inviável para instâncias de grande porte.

2.1.1 Algoritmos Exponenciais e Polinomiais

A diferença entre problemas polinomiais e NP-completos está diretamente relacionada ao tempo de execução dos algoritmos utilizados para resolvê-los.

Algoritmos polinomiais têm um tempo de execução que cresce polinomialmente com o tamanho da entrada. Já algoritmos exponenciais têm um tempo de execução que cresce exponencialmente com o tamanho da entrada.

Definição 1.1 *Algoritmo de tempo polinomial é aquele cuja função complexidade de tempo é $O(p(n))$ para alguma função polinomial $p(n)$, quando n é o comprimento da entrada.[4] Exemplos: pesquisa binária ($O(\log n)$), pesquisa sequencial ($O(n)$), ordenação por inserção ($O(n^2)$), e multiplicação de matrizes ($O(n^3)$).[1]*

Definição 1.2 *Algoritmo de tempo exponencial é aquele cuja função complexidade de tempo não pode ser limitada por um polinômio.[4] Sua função de complexidade é $O(c^n)$, $c > 1$. [3] Exemplo: Problema do Caixeiro Viajante (PCV) ($O(n!)$).[3]*

Para problemas NP-completos, os melhores algoritmos conhecidos são exponenciais, o que limita sua aplicabilidade a instâncias pequenas. A distinção entre algoritmos polinomiais e exponenciais tem profundas implicações práticas. Problemas que podem ser resolvidos em tempo polinomial são considerados viáveis, enquanto problemas NP-completos são geralmente considerados intratáveis para grandes instâncias.

Portanto, essa diferença justifica a busca por soluções aproximadas e metaheurísticas para problemas NP-completos.

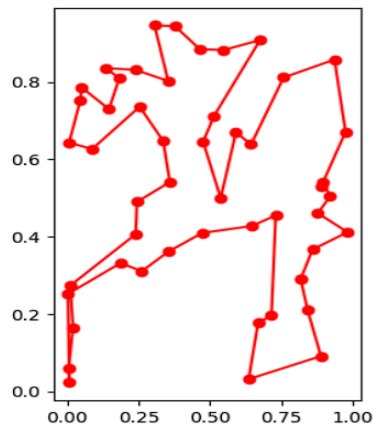
2.2 Meta-heurísticas

Meta Heurísticas são algoritmos de alto nível que coordenam procedimentos de busca local com estratégias de mais alto nível, visando escapar de mínimos locais e encontrar soluções próximas ao ótimo global. Essas técnicas inspiram-se em fenômenos naturais, como a evolução biológica, no caso de algoritmos genéticos, e processos de memória a curto prazo na busca tabu. Ao contrário de algoritmos exatos, as metaheurísticas não garantem que a solução encontrada seja ótima, mas oferecem um bom equilíbrio entre qualidade da solução e tempo de execução sendo uma ótima alternativa para resolver o Problema do Caixeiro Viajante.

2.3 Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV) é um clássico da otimização combinatória que consiste em determinar a rota mais curta que um caixeiro deve percorrer para visitar um conjunto de cidades exatamente uma vez, retornando ao ponto

de partida. Devido à sua natureza combinatória, o PCV é classificado como um problema NP-difícil, o que significa que encontrar a solução ótima para instâncias de grande porte é computacionalmente inviável.



2.3.1 Algoritmo de força bruta e o problema do caixeiro viajante

2.3.2 Algoritmo genético e o problema do caixeiro viajante

Ao longo de várias gerações, o algoritmo busca melhorar as soluções através da combinação de boas características das rotas anteriores, enquanto evita soluções de baixo desempenho. A ideia é que, após diversas iterações, o algoritmo convirja para uma rota próxima da ótima, sem testar exaustivamente todas as combinações possíveis.

Essa abordagem é vantajosa por explorar simultaneamente várias regiões do espaço de soluções e se adaptar bem a problemas complexos, como o TSP, onde o número de rotas possíveis aumenta exponencialmente com o número de cidades.

2.3.3 Busca tabu e o problema do caixeiro viajante

A Busca Tabu aplicada ao Problema do Caixeiro Viajante é uma técnica de otimização que busca encontrar uma rota de custo mínimo sem testar todas as possíveis combinações de rotas, como ocorre no algoritmo de força bruta. A Busca Tabu utiliza uma solução inicial e, a partir dela, realiza pequenas alterações na rota (vizinhança), procurando uma solução melhor. Para evitar revisitar soluções já exploradas e escapar de ótimos locais, a Busca Tabu mantém uma lista de movimentos proibidos (Lista Tabu), que limita as transições entre soluções já analisadas recentemente. Essa abordagem permite encontrar soluções de alta qualidade de forma mais eficiente, especialmente em instâncias com muitas cidades, onde o número de combinações possíveis é muito grande para ser calculado diretamente.

3. Trabalhos relacionados

3.1 Greedy Permuting Method for Genetic Algorithm on Traveling Salesman Problem

O Greedy Permuting Method[2], trata-se de um algoritmo memético, pois é um algoritmo genético que utiliza uma heurística gulosa para inicializar a população. Por utilizar o operador 2-point crossover, ele pode limitar a diversidade da população e a capacidade de explorar diferentes regiões do espaço de busca. Essa abordagem, embora eficiente para problemas menores, pode não ser adequada para o PCV, que é um problema NP-difícil.

Em contrapartida, o algoritmo genético que foi desenvolvido neste trabalho, com sua abordagem mais exploratória e a utilização do operador Order-crossover, que preserva a ordem relativa de um grande número de cidades, permite a criação de 'atalhos' e a exploração de novas regiões do espaço de busca.

Além disso, a otimização multiobjetivo, que considera tanto a distância total da rota quanto o tempo de viagem, torna o algoritmo citado mais adequado para aplicações reais, em que múltiplos objetivos precisam ser balanceados

3.2 TSP Ejection Chains

O TSP Ejection Chains propõe uma abordagem inovadora para o Problema do Caixeiro Viajante (PCV) por meio de cadeias de ejeção, que geram soluções de maneira eficiente ao realizar trocas controladas de arestas em uma estrutura chamada "stem-and-cycle". Essa estrutura permite explorar novas regiões do espaço de busca de forma

organizada. No entanto, a diversidade de soluções pode ser limitada se o método não for combinado com técnicas adicionais de diversificação.

Ao comparar com o algoritmo de Busca Tabu utilizado no projeto FlyFood, ambos compartilham o objetivo de explorar o espaço de soluções de forma eficiente. Contudo, o método de cadeias de ejeção oferece menos flexibilidade em termos de memória adaptativa quando comparado à Busca Tabu, que utiliza listas tabu dinâmicas para evitar repetições e ampliar a exploração. Além disso, enquanto as cadeias de ejeção otimizam localmente com base em heurísticas específicas, a Busca Tabu no Fly Food adota uma abordagem mais equilibrada, gerenciando tanto a exploração quanto a intensificação de soluções de forma mais robusta.

Embora o método de *Ejection Chains* seja eficaz para escapar de ótimos locais, ele pode não garantir a mesma diversidade a longo prazo que a Busca Tabu, especialmente no que diz respeito ao controle dinâmico da lista tabu e ao balanceamento entre a exploração de novas soluções e a melhoria das atuais.

3.3 Solving the Free Clustered TSP Using a Memetic Algorithm

O artigo "Solving the Free Clustered TSP Using a Memetic Algorithm" propõe um algoritmo memético (MA) que combina a capacidade de busca global de um algoritmo genético (GA) com a busca local refinada usando o *Guided Local Search* (GLS) para resolver o Problema do Caixeiro Viajante com Clusters Livres (FCTSP). A combinação de GA e busca local proporciona ao algoritmo memético uma abordagem mais eficiente na exploração do espaço de soluções, especialmente ao refinar soluções através do 2-Opt. Essa combinação equilibra a exploração global do espaço de busca com a intensificação local.

Ao comparar esse método com o algoritmo de Busca Tabu do projeto FlyFood, ambos compartilham a característica de evitar ótimos locais. O algoritmo de Busca Tabu utiliza listas tabu para controlar a repetição de soluções, promovendo uma exploração contínua, enquanto o algoritmo memético combina a busca local com a exploração global por meio do cruzamento e mutação típicos de GA. A Busca Tabu é mais flexível em sua estratégia de exploração devido à lista tabu, que é ajustada dinamicamente. Por outro lado, o algoritmo memético foca mais na diversificação da população por meio do crossover, embora dependa fortemente do desempenho da heurística local (GLS) para refinamento.

Comparando também com o algoritmo genético, o MA se destaca ao incluir a otimização local (GLS) para escapar de ótimos locais e melhorar a qualidade das soluções. Já o GA puro tende a sofrer com a falta de refinamento local, o que pode limitar sua capacidade de encontrar soluções de alta qualidade rapidamente. No entanto, o GA puro ainda pode ser vantajoso para problemas maiores devido à sua abordagem de busca mais ampla, enquanto o MA e a Busca Tabu tendem a focar mais na intensificação e diversificação controlada.

4. Metodologia

4.1 Algoritmo de força bruta

Na primeira parte deste trabalho, foi implementado o algoritmo de força bruta em Python 3 para resolver o Problema do Caixeiro Viajante. Utilizando a biblioteca *itertools*, mais especificamente o iterador *permutations*, todas as possíveis rotas entre as cidades foram geradas. A distância total de cada rota foi calculada a partir da matriz de distâncias, previamente construída com base em algumas das coordenadas dos pontos extraídos do TSPLib (instância *52berlin*). O algoritmo, embora exaustivo, não sendo eficiente para entradas maiores, garante a obtenção da solução ótima para o problema, servindo como um ponto de referência para a avaliação de outras heurísticas.

4.2 Algoritmo genético

Visando encontrar soluções mais eficientes para o PCV, foi implementado um algoritmo genético multiobjetivo. Em que cada indivíduo representa uma possível ordem de visita das cidades e a partir das coordenadas das cidades extraídas do TSPLib (instância *52berlin*), foram gerados tempos de viagem aleatórios para cada rota, simulando diferentes condições de tráfego ou restrições de tempo.

Ademais, a função de fitness considera tanto a distância total percorrida quanto o tempo de viagem estimado, sendo ambos os objetivos otimizados simultaneamente, a seleção dos indivíduos foi realizada utilizando o método do torneio, e os operadores genéticos empregados foram o order-crossover e a mutação por inversão.

Por fim, a fronteira de Pareto foi utilizada para identificar as soluções não dominadas, ou seja, aquelas que não podem ser melhoradas em um objetivo sem piorar outro.

4.3 Busca Tabu

Com o objetivo de encontrar uma solução de alta qualidade para o Problema do Caixeiro Viajante, optou-se por implementar também a metaheurística da Busca Tabu. Essa escolha se justifica pela sua eficiência em problemas de otimização combinatória, como o PCV, e pela capacidade de explorar o espaço de soluções de forma intensiva.

Sendo assim, cada solução representa uma ordem de visita das cidades, com base nas coordenadas da instância *52berlin* do TSPLib, a função de avaliação considera apenas a distância total percorrida. A busca tabu explora soluções vizinhas e utiliza uma lista tabu para evitar revisitar soluções recentes, prevenindo ciclos e promovendo uma exploração mais eficaz do espaço de soluções. O processo de busca é interrompido quando um critério de parada pré-definido é atendido, foi utilizada uma semente para garantir a reprodutibilidade dos resultados.

5. Experimentos

Essa seção apresenta o passo a passo realizado para a construção dos algoritmos de força bruta, genético e busca tabu para a resolução do problema do caixeiro viajante. O código fonte utilizado para a implementação dos algoritmos está disponível em [7].

5.1 Algoritmo de força bruta

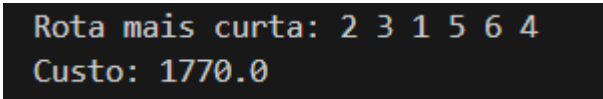
Para resolver o problema do caixeiro viajante, foi implementado um algoritmo de força bruta. Inicialmente, algumas cidades da instância berlin52 do TSPLib foram utilizadas. As coordenadas dessas cidades foram lidas de um arquivo .txt e armazenadas em um dicionário Python, onde cada chave representa uma cidade e o valor associado é uma tupla com suas coordenadas (x, y), por exemplo: dicionário = {C = (x,y)}.

Em seguida, o algoritmo gerou todas as possíveis permutações das cidades utilizando a função *permutations* da biblioteca *itertools*. Para cada permutação, o custo total da rota foi calculado, considerando a distância euclidiana entre cada par de cidades consecutivas. A distância euclidiana é calculada pela fórmula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

```
Função LerArquivo(nome_arquivo)
  Criar um dicionário "coordenadas" para armazenar as coordenadas de cada ponto
  Criar uma lista "pontos_de_entrega" para armazenar os identificadores dos pontos
  Abrir o arquivo "nome_arquivo" para leitura
  Para cada linha no arquivo:
    Dividir a linha em partes (id, x, y)
    Adicionar a coordenada (x, y) ao dicionário "coordenadas" com a chave "id"
    Adicionar o "id" à lista "pontos_de_entrega"
  Retornar coordenadas, pontos_de_entrega
Função CalcularRota(pontos_de_entrega, coordenadas)
  Definir "menor_custo" como infinito
  Criar uma lista vazia "melhor_rota"
  Para cada permutação dos "pontos_de_entrega":
    Inicializar "custo_atual" como zero
    Para cada par consecutivo de cidades na permutação:
      Calcular a distância entre as duas cidades
      Adicionar a distância ao "custo_atual"
    Se "custo_atual" for menor que "menor_custo":
      Atualizar "menor_custo" e "melhor_rota"
  Retornar "melhor_rota" e "menor_custo"
Função ImprimirRota(rota, custo)
  Imprimir "Rota mais curta:" seguido da rota
  Imprimir "Custo:" seguido do custo
Função Principal
  Chamar a função "LerArquivo" para obter as coordenadas e os pontos de entrega
  Chamar a função "CalcularRota" para encontrar a melhor rota e o custo
  Chamar a função "ImprimirRota" para exibir os resultados
```

Figura 2. Pseudocódigo para o algoritmo de força bruta do PCV.

A rota com a menor distância total é considerada a solução ótima e é armazenada e finalmente, a melhor rota e o seu custo total são impressos na tela.



```
Rota mais curta: 2 3 1 5 6 4
Custo: 1770.0
```

Figura 3. Algoritmo de força bruta - Saída do código com seis cidades.

Foram utilizados cinco arquivos testes. Cada um deles contendo, respectivamente, quatro, seis, oito, dez e doze pontos de entrega.

Observação: O algoritmo de força bruta possui uma complexidade computacional de $O(n!)$, tornando-o inviável para problemas de grande porte. A escolha da instância berlin52 com apenas 6 cidades permitiu demonstrar o funcionamento do algoritmo sem um custo computacional excessivo.

5.2 Algoritmo genético multiobjetivo

Para resolver o problema do caixeiro viajante 52-Berlin, foi empregado um algoritmo genético. Adicionalmente, para simular condições reais, foi introduzido um componente de tempo aleatório nas rotas armazenado em um arquivo JSON contendo dicionários aninhados. A otimização visou minimizar tanto a distância total da rota quanto o tempo de percurso.

Os parâmetros do algoritmo foram ajustados através de uma busca exaustiva em um espaço pré-definido. Após 30 execuções, a configuração que apresentou o melhor desempenho foi adotada: tamanho da população de 94 indivíduos, probabilidade de mutação de 0,43% e critério de parada de 4200 iterações. Essa combinação de parâmetros demonstrou um bom equilíbrio favorecendo a exploração do espaço de busca, resultando em rotas mais curtas e eficientes em termos de tempo.

A Figura 4 ilustra a função principal do algoritmo genético, responsável por guiar o processo evolutivo da população em busca de soluções que otimizem tanto a distância total percorrida quanto o tempo total de viagem. Inicialmente, uma população aleatória de rotas é gerada, onde cada rota representa um possível caminho a ser percorrido pelo caixeiro viajante. Em seguida, o fitness de cada indivíduo é calculado com base em uma função multiobjetivo que considera tanto a distância quanto o tempo.

```
Função ClassificarPopulacao(população, avaliações):
// Inicializar as frentes de Pareto
frentes <- uma lista vazia
// Inicializar a distância crowding para cada indivíduo
distancia_crowding <- um vetor de zeros com tamanho igual à população
// Classificar os indivíduos em frentes de Pareto
frentes <- ClassificarPorDominancia(população, avaliações)
// Calcular a distância crowding para cada frente
Para cada frente em frentes:
    Calcular a distância crowding para cada indivíduo na frente
    // (implementação detalhada da distância crowding)
    // Atribuir o fitness aos indivíduos
    Para cada indivíduo i na população:
        Para cada frente j:
            Se o indivíduo i pertence à frente j:
                fitness[i] <- -j + distancia_crowding[i]
Retornar frentes, distancia_crowding
```

Figura 4. Função fitness do algoritmo genético

A utilização de uma função multiobjetivo permite encontrar soluções que representam um bom compromisso entre as duas métricas, formando a chamada

fronteira de Pareto. Soluções nesta fronteira são consideradas não dominadas, ou seja, não existe outra solução que seja melhor em ambas as métricas [1]. Para auxiliar na seleção de soluções dentro a fronteira de Pareto, é utilizada a técnica de distância crowding, que atribui um valor de densidade a cada solução, privilegiando soluções em regiões menos densamente povoadas da fronteira.

Por conseguinte, para a seleção dos pais, foi utilizado um torneio binário determinístico. A cada geração, um grupo de 3 indivíduos era selecionado aleatoriamente pela população e o indivíduo com o melhor fitness era escolhido para compor o par de pais. Este tipo de torneio favorece a exploração do espaço de busca, uma vez que indivíduos com alta aptidão têm maior probabilidade de serem selecionados.

```

Função SelecaoTorneio(populacao, tamanho_populacao, tamanho_torneio, fitness):
// Inicializa uma lista para armazenar os pais selecionados
pais <- lista vazia
// Repete até que todos os pais sejam selecionados
Enquanto tamanho(pais) < tamanho_populacao:
    // Seleciona aleatoriamente 'tamanho_torneio' indivíduos
    participantes <- SelecionarAleatoriamente(populacao, tamanho_torneio)
    // Encontra o indivíduo com o melhor fitness
    melhor_individuo <- EncontrarMelhorIndividuo(participantes, fitness)
    // Adiciona o melhor indivíduo à lista de pais pais
    adicionar(melhor_individuo)
Retornar pais

```

Figura 5. Seleção por torneio

A fim de promover a exploração do espaço de busca e evitar a convergência prematura, foi adotada a mutação por inversão. Essa operação consiste em inverter a ordem dos genes em um segmento aleatório do cromossomo, preservando a informação genética, mas alterando a sua estrutura. A escolha da mutação por inversão se justifica pela sua eficácia em problemas de otimização combinatória, onde a ordem dos elementos é crucial, como no caso do Problema do Caixeiro Viajante.

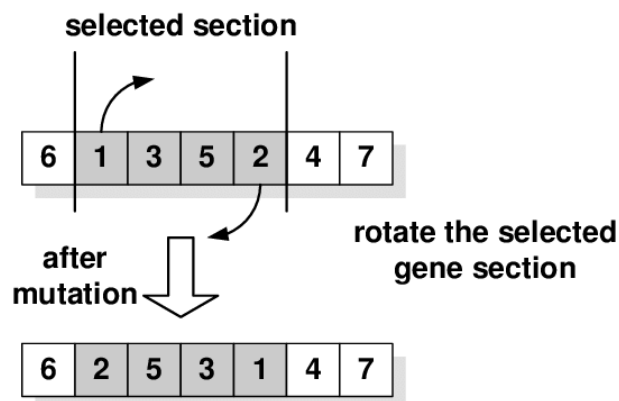


Figura 6. Exemplo da mutação por inversão

O operador de cruzamento utilizado foi o *order crossover*. Neste operador, cada indivíduo é representado por uma permutação de cidades. Dois pontos de corte são selecionados aleatoriamente no cromossomo de cada pai. A seção do cromossomo entre os pontos de corte é copiada diretamente para o filho correspondente. Em seguida, os genes restantes do outro pai são copiados para o filho, preservando a ordem relativa em que aparecem no pai. Por exemplo, se o pai 1 for [1, 2, 3, 4, 5] e o pai 2 for [5, 4, 3, 2, 1], e os pontos de corte forem 2 e 4, o filho 1 será [1, 2, 5, 4, 3] e o filho 2 será [5, 4, 1, 2, 3]. O *order crossover* é particularmente adequado para o problema do caixeiro viajante, pois preserva a ordem relativa das cidades, que é uma característica importante para a construção de rotas viáveis.

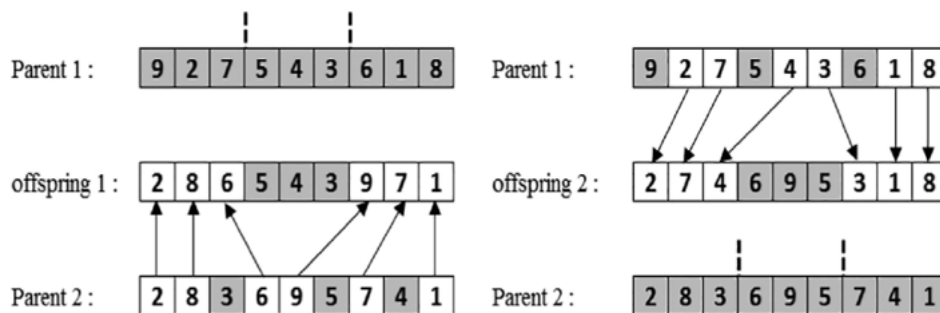


Figura 7. Exemplo do Order Crossover

O processo evolutivo é executado por um número máximo de gerações predefinido, garantindo que o algoritmo seja encerrado após um número específico de iterações. Este critério foi escolhido por permitir um controle preciso sobre o tempo de execução e facilitar a comparação com outros trabalhos. A cada geração, os indivíduos são avaliados, selecionados, cruzados e mutados, dando origem a uma nova população. O processo é repetido até que o critério de parada seja atingido.

A seleção dos melhores indivíduos é uma etapa crucial no algoritmo genético multiobjetivo. O objetivo é garantir que as soluções mais promissoras sejam propagadas para as próximas gerações, preservando a diversidade da população. Utilizando um algoritmo de classificação por dominância não dominada para identificar as frentes de Pareto. Em seguida, a seleção é realizada de forma a manter a diversidade da população e priorizar soluções de alta qualidade. A distância de crowding é utilizada como um critério de desempate quando múltiplos indivíduos pertencem à mesma frente de Pareto. Essa métrica mede a densidade de soluções em uma determinada região do espaço de busca, garantindo que a diversidade da população seja mantida.


```

Função SelecionarMelhoresIndividuos(população, avaliações,
tamanho_população):
    // Classificar a população em frentes de Pareto
    frentes <- ClassificarPorDominânciaNãoDominada(população,
avaliações)

    // Inicializar a nova população
    nova_população <- lista vazia

    // Iterar sobre as frentes
    para cada frente em frentes:
        se o tamanho da nova população + o tamanho da frente exceder
o tamanho_população:
            // Ordenar a frente pela distância de crowding
            ordenar frente de acordo com a distância de crowding
(decrecente)
            // Adicionar indivíduos da frente até completar a nova
população
            adicionar os primeiros (tamanho_população -
tamanho_da_nova_população) indivíduos da frente ordenada à nova
população
            // Interromper o loop
            sair do loop
        senão:
            // Adicionar todos os indivíduos da frente à nova
população
            adicionar todos os indivíduos da frente à nova população

    // Retornar a nova população
    retornar nova_população

```

Figura 8. Pseudocódigo da seleção dos melhores indivíduos

A escolha da classificação por dominância não dominada e da distância de crowding se justifica pela necessidade de lidar com problemas multiobjetivos, onde não existe uma única solução ótima, mas sim um conjunto de soluções não dominadas. A distância de crowding permite explorar diferentes regiões do espaço de busca, aumentando a probabilidade de encontrar soluções de alta qualidade e bem distribuídas.

5.3 Busca Tabu

Para resolver o Problema do Caixeiro Viajante (TSP) com 52 cidades, foi aplicado um Algoritmo de Busca Tabu, projetado para encontrar a rota de menor custo explorando iterativamente vizinhanças de soluções e evitando ciclos de retrocesso por meio de uma lista tabu. Essa lista armazena soluções recentemente visitadas, impedindo sua revisitação para evitar a estagnação em ótimos locais.

O algoritmo inicia com a geração de uma solução inicial aleatória (figura 7), que é uma rota das cidades obtida por uma permutação aleatória das coordenadas das cidades. O custo dessa rota é calculado com base na soma das distâncias entre as cidades consecutivas, incluindo o retorno à cidade inicial.

Função GerarSolucaoInicialAleatoria(coord, seed):

//Se seed não for Nulo, então:

//Definir a semente aleatória

Criar uma lista com as chaves de coord

Embaralhar a lista

Adicionar a primeira cidade ao final da lista

Retornar a lista como a solução inicial

Figura 9. função para gerar solução aleatória

Durante o processo de busca, vizinhos da solução atual são gerados através da inversão de segmentos da rota. A função de gerar vizinhos (figura 8) cria essas variações da rota atual, e a função buscar melhor vizinho avalia esses vizinhos para identificar a melhor solução possível, respeitando as restrições impostas pela lista tabu.

Função BuscarMelhorVizinho(vizinhos, listaTabu, melhorCustoAtual, coord):

Inicializar melhorVizinho como Nulo

Inicializar melhorCusto como infinito

Para cada vizinho na lista de vizinhos faça:

Calcular o custo do vizinho

**Se o vizinho não estiver na lista tabu ou o custo do vizinho for menor que o melhor custo atual
então:**

Se o custo do vizinho for menor que o melhor custo então:

Atualizar melhorVizinho com o vizinho

Atualizar melhorCusto com o custo do vizinho

Retornar melhorVizinho

Figura 10. Função de buscar melhor vizinho

A lista tabu é mantida com um tamanho fixo, onde entradas antigas são removidas à medida que novas soluções são adicionadas. A estratégia é manter a lista

tabu suficientemente pequena para evitar a convergência prematura, mas grande o suficiente para garantir a diversidade das soluções.

Abaixo na figura 9 está um exemplo da função principal da lista tabu. Esta função realiza a busca tabu para encontrar a melhor solução para o problema. Começa com uma solução inicial aleatória e itera por um número definido de iterações. Em cada iteração, gera vizinhos da solução atual, encontra o melhor vizinho respeitando a lista tabu e atualiza a solução se um vizinho melhor for encontrado. A lista tabu é atualizada para evitar ciclos e estagnação. A função retorna a melhor solução encontrada e seu custo.

```
Função BuscaTabu(coord, tamLista, maxIte, seed):  
  solucaoInicial = GerarSolucaoInicialAleatoria(coord, seed)  
  melhorSolucao = cópia da solucaoInicial  
  melhorCusto = CalcCusto(melhorSolucao, coord)  
  Inicializar listaTabu como uma lista vazia  
  Para i de 1 até maxIte faça:  
    vizinhos = GerarVizinhos(melhorSolucao)  
    melhorVizinho = BuscarMelhorVizinho(vizinhos, listaTabu, melhorCusto, coord)  
    Se melhorVizinho não for Nulo então:  
      custoVizinho = CalcCusto(melhorVizinho, coord)  
      Se custoVizinho for menor que melhorCusto então:  
        Atualizar melhorSolucao com melhorVizinho  
        Atualizar melhorCusto com custoVizinho  
        Adicionar melhorVizinho à listaTabu  
      Se o tamanho da listaTabu exceder tamLista então:  
        Remover a solução mais antiga da listaTabu  
  Retornar melhorSolucao e melhorCusto
```

Figura 11. Função principal

6. Resultados

6.1 Algoritmo de força bruta

Os resultados evidenciam que o tempo de execução cresce exponencialmente de acordo com o tamanho da entrada. Dessa maneira, nota-se que o algoritmo de força bruta não é eficiente para otimizar as rotas de matrizes maiores com muitos pontos de entrega, por se tratar de um problema NP-Completo com algoritmo de complexidade $O(n!)$.

Os testes foram realizados com 5 matrizes de tamanhos diferentes :

Quantidade de pontos	Solução ótima	Tempo (segundos)
6	1770.0	0.01
10	2790.0	7.61
9	2650.0	0.67
8	2270.0	0.6
7	1770.0	0.1

Figura 12. Tabela de resultados usando força bruta

6.2 Algoritmo Genético

Nesta seção, são apresentados os resultados obtidos pela aplicação do algoritmo genético proposto para o problema de otimização de rotas de drones. Foram realizadas 30 execuções independentes do algoritmo, variando a semente aleatória a cada execução, para avaliar a robustez e a reprodutibilidade dos resultados. A tabela na figura 12 apresenta uma média dos resultados para cada parâmetro e seus valores de máximo e de mínimo.

Métrica	Valor Mínimo	Valor Máximo	Média
Distância	7.954.124	10.115.140	8.803.994
Tempo	3906	6009	4774,5

Figura 13. Tabela com média, valores máximo e mínimo dos parâmetros

A Figura 14 apresenta a melhor rota encontrada pelo algoritmo genético para o problema em questão. A rota representa a sequência ótima de visitas aos pontos, minimizando a distância total percorrida. Observa-se que a rota encontrada apresenta um padrão de distribuição espacial característico, indicando que o algoritmo foi capaz de encontrar uma solução eficiente.

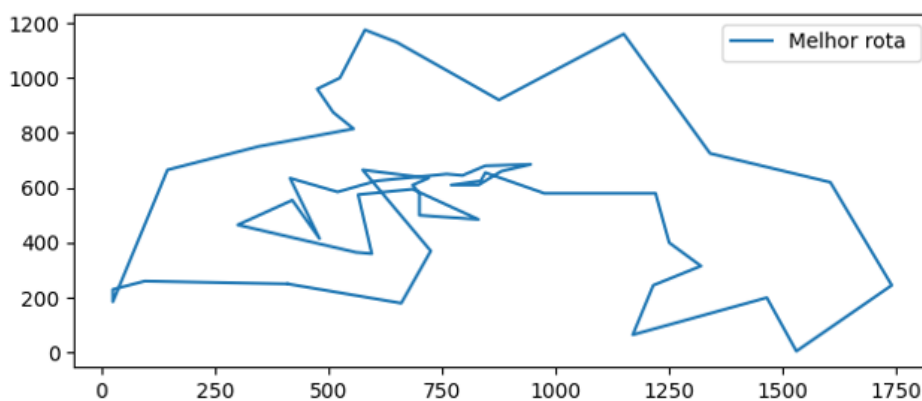


Figura 14. Representação gráfica da melhor rota

A Figura 15 apresenta a curva de convergência do algoritmo genético ao longo das gerações, evidenciando a busca pela melhor solução. Observa-se uma rápida diminuição na distância total percorrida nas primeiras gerações, indicando uma eficiente exploração do espaço de busca. À medida que o número de gerações aumenta, a taxa de convergência diminui, sugerindo que o algoritmo se aproxima de um ótimo local. A estabilização da curva nas últimas gerações indica que o algoritmo convergiu para uma solução de alta qualidade, embora não se possa garantir que seja a solução ótima global. A análise da curva de convergência permite avaliar o desempenho do algoritmo e ajustar os parâmetros, caso necessário, para melhorar a eficiência da busca.

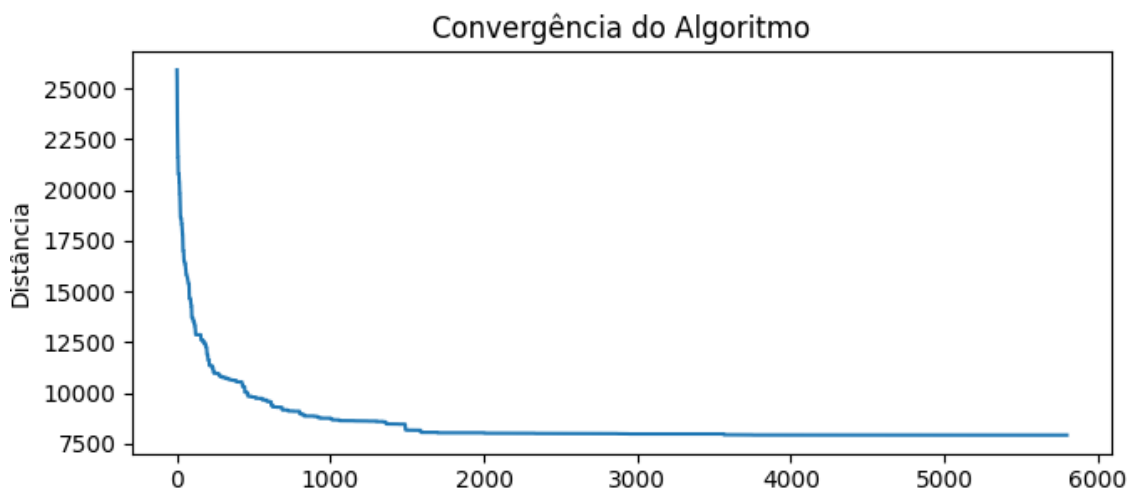


Figura 15. Representação gráfica da convergência do algoritmo

A Figura 16 ilustra a fronteira de Pareto obtida a partir da aplicação do algoritmo genético multiobjetivo. Cada ponto na fronteira representa uma solução que otimiza simultaneamente a distância total percorrida e o tempo de voo. Observa-se que a fronteira de Pareto apresenta uma forma convexa característica, indicando a existência de um trade-off entre os dois objetivos: ao se buscar minimizar a distância, o tempo de voo tende a aumentar, e vice-versa. A diversidade de soluções na fronteira de Pareto demonstra a capacidade do algoritmo genético em explorar o espaço de busca de forma eficiente, oferecendo um conjunto de soluções não dominadas que podem ser utilizadas para tomada de decisão, de acordo com as prioridades específicas de cada cenário.

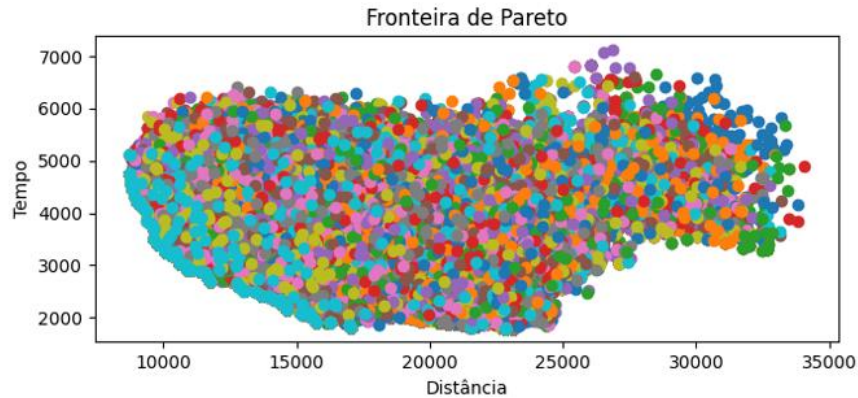


Figura 16. Representação gráfica da Fronteira de pareto

Portanto, os resultados obtidos demonstram que o algoritmo genético proposto é capaz de encontrar soluções de alta qualidade para o problema de otimização de rotas de drones. A convergência rápida do algoritmo indica sua eficiência, enquanto a análise da fronteira de Pareto mostra que o algoritmo é capaz de encontrar um bom equilíbrio entre os objetivos de minimizar a distância e o tempo de voo.

6.2 Busca Tabu

Nesta seção, são apresentados os resultados obtidos pela aplicação do algoritmo de busca tabu propostos para o problema de otimização de rotas de drones. Foram realizadas 30 execuções independentes do algoritmo, variando a semente aleatória a cada execução, para avaliar a robustez e a reprodutibilidade dos resultados.

A figura 17 ilustra graficamente a comparação entre os custos obtidos a cada execução do algoritmo de busca tabu. O gráfico destaca a variação nos custos finais e permite identificar as execuções com os maiores e menores custos

A figura 18 é um gráfico que mostra a relação entre o custo obtido e o tempo que cada execução levou para retornar um resultado. A figura ajuda a visualizar como o tempo gasto para a execução impacta no resultado final.

A figura 19 representa a melhor solução encontrada pelo algoritmo para o problema em questão (TSP-berlin52). As cidades estão conectadas de acordo com a ordem em que devem ser visitadas, para evidenciar o melhor caminho a ser percorrido.

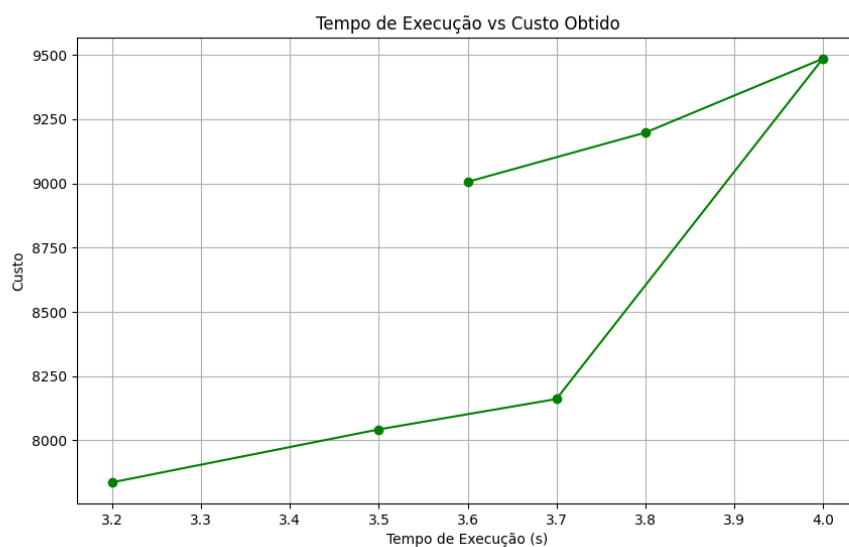
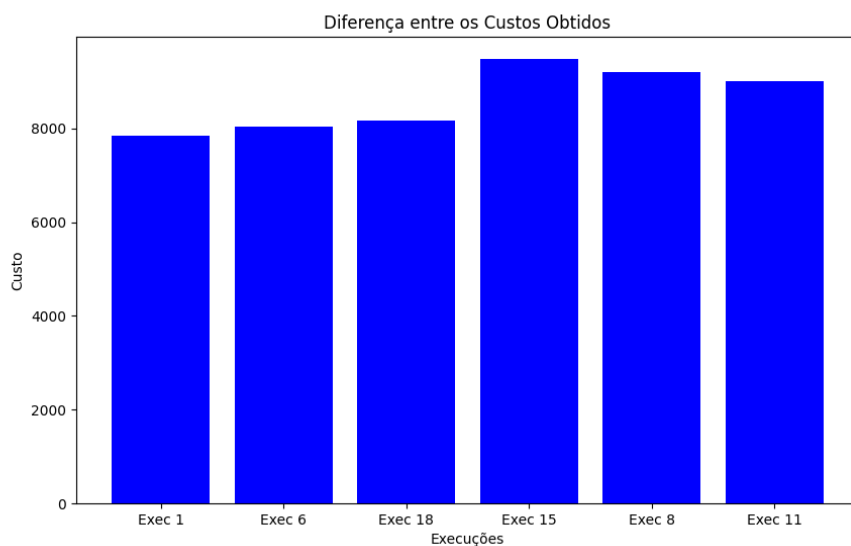


Figura 17. Gráfico diferença entre os custos

Figura 18. Gráfico da variação do tempo de execução

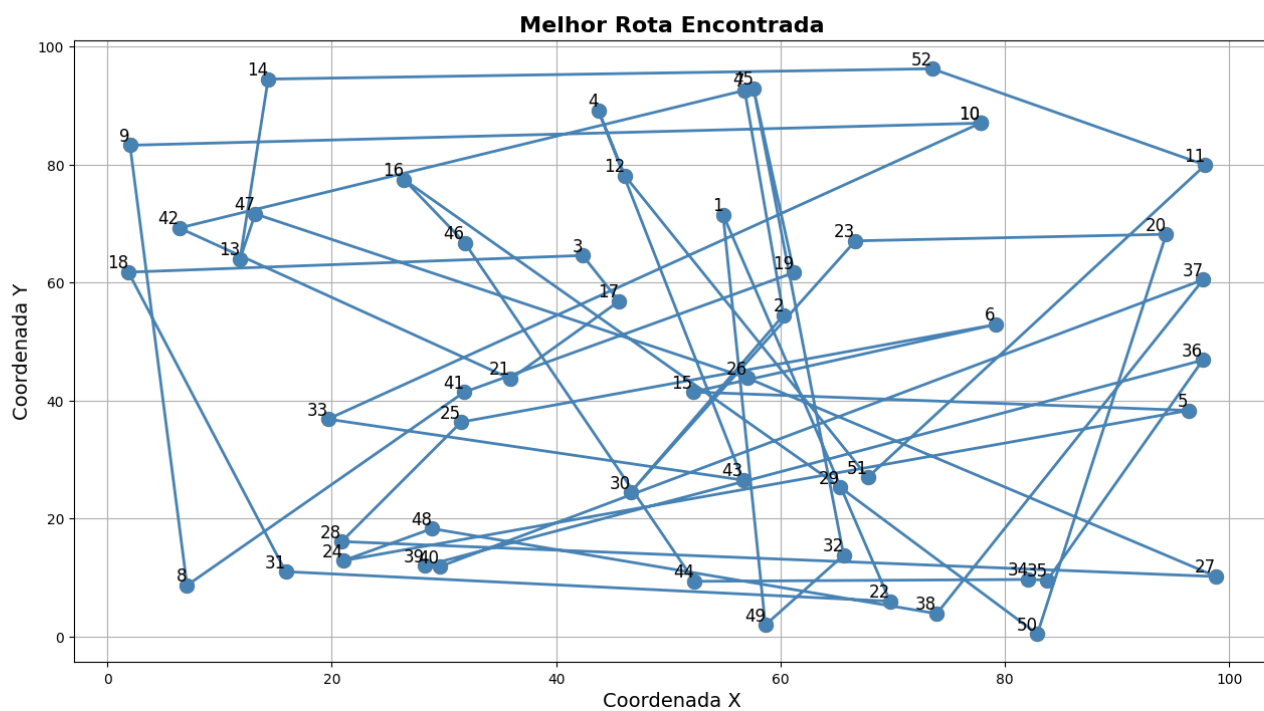


Figura 19. Gráfico da melhor rota

Os parâmetros utilizados nas 30 execuções foram:

- Tamanho da lista: 200
- Número máximo de iterações: 5000
- Solução inicial gerada de forma aleatória

Essa combinação de parâmetros demonstrou um bom equilíbrio entre a diversificação e a intensificação da busca, resultando em rotas mais curtas e eficientes em termos de tempo. Abaixo está uma tabela que ilustra as principais informações obtidas após a implementação do algoritmo.

Rota	Seed	Custo	Tempo execução
Melhor rota	721296153	7837.1323	2.5 minutos
Pior rota	2913372783	9484.7352	2.4 minutos
Rota média		8492.6277	2.45 minutos

7. Conclusão

Este trabalho teve como objetivo principal desenvolver e comparar algoritmos genéticos e busca tabu para otimizar rotas de entrega por drones, com foco no problema do caixeiro viajante. Os resultados obtidos demonstram que ambos os algoritmos são eficazes em encontrar soluções de alta qualidade, superando significativamente a abordagem de força bruta. O algoritmo genético se destacou na exploração do espaço de busca, enquanto a busca tabu apresentou um desempenho superior em termos de tempo de execução para instâncias menores.

Como trabalhos futuros, pretende-se evoluir o algoritmo genético proposto através da hibridização com outras heurísticas, como algoritmos de colônias de formigas ou algoritmos gulosos. Adicionalmente, por meio de testes de hipóteses, será comparado o desempenho do algoritmo genético híbrido com o algoritmo genético original e com a busca tabu em um conjunto diversificado de instâncias do problema de roteamento de drones.

Portanto, conclui-se que este trabalho contribui para o avanço do campo da otimização de rotas e demonstra o potencial dos algoritmos genéticos e busca tabu para resolver problemas complexos como o problema do caixeiro viajante em aplicações reais, como sistemas de entrega por drones.

8- Referências Bibliográficas

- 1- PANT, M.; VERMA, S. A Comprehensive Review on NSGA-II for Multi-Objective Combinatorial Optimization Problems. 17 mar. 2021.
- 2- Liu, J., & Li, W. (2018). Greedy Permuting Method for Genetic Algorithm on Traveling Salesman Problem. 2018 8th International Conference on Electronics Information and Emergency Communication (ICEIEC).
- 3- LOUREIRO, Antonio Alfredo Ferreira. Teoria de Complexidade. UFMG, 2008. Disponível em: <http://www.decom.ufop.br/menotti/paall1/slides/aula-Complexidade-imp.pdf> Acesso em 06 de set. de 2024.
- 4- DE ABREU, Nair Maria Maia. A Teoria da Complexidade Computacional. R. mil. Cio e Tecno/., Rio de Janeiro, 4(1):90-95, jan./mar. 1987. Disponível em: http://rmct.ime.eb.br/arquivos/RMCT_1_tri_1987/teoria_complex_comput.pdf Acesso em 06 de set. de 2024.
- 5- Pesch, E., & Glover, F. (1997). TSP Ejection Chains. Springer.
- 6-Alsheddy, A. (2017). Solving the Free Clustered TSP Using a Memetic Algorithm. International Journal of Advanced Computer Science and Applications, 8(8), 404–408.
- 7-TAVARES, Ellen Caroliny, VITÓRIA, Evny. GitHub - ellencaroliny/PISI2-2VA. Disponível em: <<https://github.com/ellencaroliny/PISI2-2VA>>. Acesso em: 10 set. 2024.