FIAP– POSTECH IA PARA DEV (6IADT)

TECH CHALLENGE: FASE 2

PROJETO 2: OTIMIZAÇÃO DE ROTAS MÉDICAS (TSP/VRP COM ALGORITMO GENÉTICO + LLM)

ALANA CAROLINE DE OLIVEIRA DA LUZ - RM 366039

CATHERINE CRUZ PORTO - RM 366325

ETIANNE TORRES CHAN - RM 365201

RENAN AUGUSTO ALVES DA COSTA - RM 365564

VALDIR DE SOUZA JUNIOR - RM 365913

14/10/2025

Turma: 6IADT

Repositório: https://github.com/junior-vs/FIAP-Tech-Challenge-6IADT-Fase-2

Demonstração: https://youtu.be/qxUAR7cL52E

SUMÁRÍO

1. INTRODUÇÃO

O presente trabalho foi desenvolvido no contexto do Tech Challenge – Fase 2 do curso de Pós-Graduação em Inteligência Artificial para Desenvolvedores (FIAP), que propôs dois desafios: a otimização de modelos de diagnóstico médico (Projeto 1) e a otimização de rotas logísticas para distribuição de medicamentos e insumos (Projeto 2).

O grupo optou pelo Projeto 2 por compreender que o problema de roteamento médico reflete uma das aplicações mais tangíveis e críticas da IA no contexto hospitalar: a entrega eficiente e confiável de medicamentos salva vidas, reduz custos operacionais e aumenta a disponibilidade de recursos.

Além disso, o desafio oferecia uma oportunidade de combinar técnicas clássicas de otimização (algoritmos genéticos) com modelos de linguagem natural (LLMs), permitindo explorar a complementaridade entre métodos heurísticos e IA generativa.

2. OBJETIVO

Desenvolver um sistema de otimização de rotas médicas capaz de atender múltiplas restrições operacionais, utilizando Algoritmos Genéticos (GA) como heurística evolutiva e LLMs para gerar relatórios e instruções em linguagem natural para as equipes logísticas.

Os objetivos específicos foram:

- Adaptar o problema clássico do Travelling Salesman Problem (TSP) para um contexto hospitalar realista (VRP);
- Implementar restrições de capacidade de carga, autonomia de veículos, prioridade de entrega e múltiplos veículos;
- Analisar e justificar a escolha de parâmetros genéticos e operadores evolutivos;
- Integrar um modelo LLM para geração de instruções e relatórios textuais interpretáveis;
- Validar a performance do algoritmo e demonstrar os ganhos obtidos com a abordagem proposta.

3. Metodologia e Arquitetura da Solução

O sistema foi construído, incluindo a codificação das rotas (cromossomos), a função de avaliação (fitness) com restrições reais do domínio, os operadores genéticos disponíveis e como os parâmetros evolutivos são controlados na aplicação. A implementação está organizada em módulos sob src/ (especialmente functions/ e domain/) e oferece UI em Pygame para experimentação interativa e visualização de métricas.

A solução foi implementada em Python, estruturada em módulos dentro do repositório Git:https://github.com/junior-vs/FIAP-Tech-Challenge-6IADT-Fase-2/blob/release_1.0-validation/src/functions/ui_layout.py

Representação Genética da Solução

Cada indivíduo da população representa uma sequência de rotas que conecta pontos de entrega. O gene corresponde ao ID de cada ponto, e o cromossomo define a ordem de visitação, garantindo que cada local seja visitado uma única vez.

No problema de roteamento (TSP/VRP), cada indivíduo do GA representa uma rota possível. Então, a representação genética define como o algoritmo armazena essa rota na forma de um cromossomo.

Por exemplo: [0, 3, 5, 2, 1, 4] -> significa que o veículo sai do ponto 0, depois visita os pontos 3, 5, 2, 1 e volta ao 4, uma rota completa. Cada número (gene) representa um ponto de entrega. O cromossomo completo é a rota inteira.

Indivíduo (cromossomo): uma solução candidata é representada por um objeto `Route` contendo a sequência ordenada de `DeliveryPoint` a ser atendida.

Gene: cada `DeliveryPoint` pode conter um `product` com atributos validados (peso, dimensões, *volume*, e `priority ∈ [0.0, 1.0]`).

População: um conjunto de `Route` é avaliado a cada geração.

Viabilidade: embora toda permutação forme uma rota válida do ponto de vista do TSP, a solução só é operacionalmente viável quando respeita capacidade de (peso/volume), autonomia e frota, e priorização de entregas, esses aspectos são tratados na função de fitness (Seção 3.2) e no *split* com múltiplos veículos.

Observação: o projeto expande o TSP para um VRP com múltiplos veículos (mVRP), mantendo a representação "uma rota = uma permutação" e fazendo o particionamento (split) posterior para alocar sub-rotas aos veículos conforme autonomia/custo.

Função de Fitness e Restrições

A avaliação da qualidade de uma rota considera distância **e** penalidades por violar restrições do mundo real.

Distância e capacidades

Distância total: calculada sobre a sequência da `Route`.

Capacidades: são acumulados peso total (g) e volume total (cm³) dos produtos nos pontos:

`MAX_WEIGHT = 500_000 g` (500 kg);

`MAX_VOLUME = 5_000_000 cm³` (≈ 5 m³).

Penalidades por excesso:

`PENALTY_WEIGHT_FACTOR = 1.0` multiplica o excesso de peso;

`PENALTY_VOLUME_FACTOR = 0.002` multiplica o excesso de volume.

Prioridade (ordenação na rota)

Para priorizar entregas críticas, a função de fitness adiciona penalidade quando itens com maior `product.priority` aparecem tarde na sequência:

$$\text{penalidade\_prioridade} = \sum_{i=0}^{n-1} \text{priority}_i \cdot \frac{i}{\max(1, n-1)}$$

$$\text{custo total} = \text{distância} + \underbrace{2.0}_{\texttt{priority\_weight}} \cdot \text{penalidade\_prioridade} + \text{penalidades de peso/volume}$$

$$\text{fitness} = \frac{1}{\text{custo total}}$$

Intuição: quanto mais tarde um ponto prioritário aparece, maior a fração $i/(n-1)$ e, portanto, maior a penalização; o `priority_weight = 2.0` regula a importância desse critério frente aos demais termos.

mVRP: autonomia, custo por km e uso de frota

Para o cenário com múltiplos veículos (habilitado por padrão via self.use_fleet = True):

O sistema particiona a sequência global em sub-rotas, escolhendo, para cada sub-rota, o veículo de menor custo por km que atende a autonomia necessária.

É aplicado custo por km por tipo de veículo e, se o uso de um tipo exceder a disponibilidade, há penalização forte. A penalidade de prioridade também é aplicada em cada sub-rota, garantindo que a priorização não se perca com o particionamento.

Existe uma BIG_PENALTY = 1e12 para inviabilidades (e.g., nenhuma autonomia atende a sub-rota).

Conversão de escala: o cálculo de distância para autonomia/custo usa um fator de escala (e.g., scale = 0.1) para transformar pixels em km, permitindo parâmetros realistas de autonomia.

Parâmetros Evolutivos

A aplicação fornece vários operadores e controles via UI para experimentação. Abaixo listamos o que está implementado e como cada parâmetro é definido/justificado no seu código atual.

Seleção

A aplicação implementa três métodos de seleção disponíveis ao usuário na interface:

Opções de UI: roulette (Roleta), tournament (Torneio) e rank (Ranking). O método ativo é controlado pelo atributo `app.selection_method`, definido conforme o botão pressionado na interface durante a execução.

Implementações internas:

- roulette_wheel_selection

Seleciona indivíduos com probabilidade proporcional ao fitness**,** calculado como 1/distancia1/distância. Cada indivíduo recebe uma fração do total de fitness acumulado, e o sorteio escolhe aqueles com maior probabilidade relativa.

- tournament_selection e tournament_selection_refined

Selecionam um grupo aleatório de indivíduos (tamanho do torneio definido internamente) e escolhem o de melhor fitness (menor distância). A versão refinada adiciona verificação de Índices para evitar duplicidade e manter diversidade. Esse método é executado por padrão quando o usuário inicia o programa pela primeira vez, caso não tenha selecionado outro método na interface.

- rank_selection

Ordena os indivíduos de acordo com o fitness e aplica uma probabilidade ponderada sobre a posição no ranking. Dessa forma, indivíduos com melhor desempenho tem mais chance de reprodução mas sem eliminar a variabilidade.

Comportamento no projeto:

- O método de seleção ativo é definido dinamicamente pela interface gráfica, de acordo com a escolha do usuário.
- Durante a execução, o método selecionado é armazenado em `app.selection_method` e utilizado pelo loop principal do algoritmo genético para selecionar os pais.
- Caso o usuário não interaja com a interface antes do início da execução, o sistema utiliza `tournament_selection` como valor padrão.

Crossover (Recombinação)

A aplicação disponibiliza cinco operadores de crossover, definidos na classe `Crossover`:

- Opções na UI: `pmx`, `ox1`, `cx`, `kpoint` e `erx`.

O operador é configurado pela variável `app.crossover_method`, atualizada dinamicamente quando o usuário interage com os botões da interface.

Implementações principais:

- `crossover_parcialmente_mapeado_pmx` — realiza o Partially Mapped Crossover (PMX), trocando segmentos entre pais e reparando duplicatas via mapeamento. Após o reparo, aplica-se `_prioritize`, que ordena os pontos com `product.priority` maior para o início do cromossomo.

Este é o operador padrão inicial:

- `crossover_ordenado_ox1` — preserva a ordem relativa das rotas dos pais, ideal para problemas de permutação como TSP e VRP.
- `cx`, `kpoint`, `erx` — implementações alternativas acessíveis na interface para experimentação.

Comportamento:

O crossover ativo é invocado dinamicamente no loop evolutivo conforme o valor atual de `app.crossover_method`.

O operador padrão é o PMX, com reparo e priorização pós-processamento.

Mutação

A aplicação implementa três operadores de mutação controlados pela variável `app.mutation_method`:

- `mutacao_por_troca` — troca duas posições aleatórias no cromossomo.
- `mutacao_por_inversao` — inverte a ordem de uma subsequência aleatória.
- `mutacao_por_embaralhamento` — embaralha os genes de uma subsequência.

Após qualquer mutação, o código executa `Mutation._prioritize`, reordenando os pontos de entrega de forma que itens com maior prioridade apareçam no início da rota. Esse comportamento é aplicado de forma consistente em todos os operadores de mutação antes do retorno da nova rota.

Elitismo

O elitismo é controlado pela interface gráfica através do botão “Elitism On/Off”.

Quando ativado (`app.elitism_enabled = True`), o melhor indivíduo da geração atual é preservado integralmente na geração seguinte. Quando desativado, todas as posições da nova população são preenchidas por indivíduos resultantes dos operadores genéticos.

Parâmetros Numéricos e Interface de Usuário (UI)

Os parâmetros evolutivos são configurados dinamicamente na interface em Pygame. Os principais atributos da classe `TSPGeneticAlgorithm` são:

- `population_size` — número de indivíduos por geração;
- `max_generations` — número máximo de iterações;
- `crossover_rate` e `mutation_rate` — probabilidades de aplicação dos operadores;
- `priority_percentage` (default 20%) — percentual de produtos que recebem prioridade durante a geração dos dados.

Esses parâmetros são exibidos nos cards “Setup” e “Run”. O mapa desenhado em `draw_functions.py` destaca pontos prioritários em amarelo e exibe os valores numéricos de prioridade, de acordo com o slider “Priority %”.

4. Resultados e Testes

Esta seção documenta como o sistema foi testado**,** quais métricas o próprio código calcula/mostra **e** como interpretar as telas geradas pela UI (Pygame)**.**

Os testes seguem exatamente o que está implementado nos módulos fitness_function.py, selection_functions.py, crossover_function.py, mutation_function.py, ui_layout.pye draw_functions.py.

Ambiente e Reprodutibilidade

- Linguagem/versão: Python 3.11+
- Interface: Pygame (UI interativa com cards "Setup", "Priority", "Run", "Operators", "Crossover", "Fitness History")
- Logs: módulo app_logging.py (decorator @log_performance habilita logs de desempenho por função)
- Escala de distância (para VRP): fator scale = 0.1 (10 px ≈ 1 km) usado em _roundtrip_distance (impacta autonomia/custo)
- Reprodutibilidade: para repetir resultados, manter parâmetros idênticos na UI e, se desejado, fixar random.seed() no ponto de inicialização do app (opcional; o projeto usa aleatoriedade controlada apenas pela execução).

Cenários de Teste

Cada execução usa o gerador de mapa (card Setup) e o slider Priority % (card Priority) para construir um cenário sintético:

- Número de pontos: controlado pelos botões Cities **–** / + (ex.: 20 pontos).
- Distribuição: Random (distribuição uniforme na área) ou Circle (pontos em círculo) — selecionável na UI.
- Prioridade de entregas: o slider Priority % define a probabilidade de um produto receber product.priority > 0 no método _make_random_product (por exemplo, 20% por padrão). Quando um ponto tem prioridade, o valor é um float em [0.1, 1.0] e o ponto é destacado em amarelo no mapa (vide draw_functions.py).
- Frota e autonomia: o VRP está habilitado por padrão (self.use_fleet = True), e o split de rotas por veículo respeita autonomia e custo por km (ver FitnessFunction._split_with_vehicle_choice).
- Operadores evolutivos: escolhidos na UI (cards Operators e Crossover).

- Seleção: *roulette*, *tournament* (padrão), *rank*.
- Crossover: *pmx* (padrão), *ox1*, *cx*, *kpoint*, *erx*.
- Mutação: *swap*, *inverse*, *shuffle*.
- Elitismo: On/Off via botão (preserva melhor indivíduo quando ativado).

Métricas que o Sistema Calcula/Exibe

Todas as métricas abaixo são derivadas diretamente do código**:**

Fitness individual (TSP + restrições):

Funções: `FitnessFunction.calculate_fitness_with_constraints(route)` e, quando frota/autonomia estão ativas, `calculate_fitness_with_fleet(route, deposito, fleet)`.

O *fitness* é o inverso do custo total**:**

Custo total = distância + penalidade por prioridade tardia + penalidades por excesso de peso/volume + penalidades de frota/autonomia (quando aplicável).

Prioridade tardia: soma de `priority * (idx / (n-1))` para cada ponto (quanto mais tarde, maior a penalização); peso fixo `priority_weight = 2.0`.

Capacidades: `MAX_WEIGHT = 500_000 g`, `MAX_VOLUME = 5_000_000 cm³`;
fatores `PENALTY_WEIGHT_FACTOR = 1.0`, `PENALTY_VOLUME_FACTOR = 0.002`.

Frota/autonomia: custo por km por tipo + penalidade por excesso de uso de tipo;
inviável ⇒ `BIG_PENALTY = 1e12`.

Best Fitness e Mean Fitness por geração:

- Armazenadosem app.fitness_history (melhor)e app.mean_fitness_history

(média).

- Renderizados no card Fitness History em draw_functions.py (linha vermelha = melhor, verde = média).

Best Dist. (no card Run):

- Exibe 1 / best_fitness (conversão do *fitness* para distância aproximada).

Resumo VRP no mapa (opcional):

- Box “Rotas / Distância total ou Custo total / Uso por tipo” via draw_vrp_summary.
- Legenda por veículo (cor da rota, nome do tipo e distância) e marcação do depósito**.**

Procedimento de Execução e Coleta

A execução do sistema de otimização de rotas foi conduzida a partir da interface gráfica desenvolvida em Pygame, que permite configurar os parâmetros evolutivos, restrições operacionais e características do cenário de teste.

O procedimento de execução foi padronizado para garantir reprodutibilidade e consistência entre as rodadas de simulação.

O processo iniciou-se com a configuração do cenário na interface, selecionando o tipo de mapa (aleatório ou circular), o número de cidades, a porcentagem de entregas prioritárias, os operadores genéticos de seleção, crossover e mutação, e o uso opcional do elitismo.

Após a configuração, o botão Run foi acionado, iniciando o processo evolutivo até o limite definido de gerações (normalmente 500). Durante a execução, a aplicação atualiza dinamicamente as métricas populacionais e a função de avaliação (fitness), exibindo a evolução dos resultados em tempo real.

Ao término das simulações, as métricas quantitativas, como número de cidades, tamanho da população, geração final, distância total e melhor valor de fitness, foram extraídas diretamente dos indicadores da interface.

Esses dados foram consolidados manualmente em uma tabela de coleta de métricas, com o objetivo de analisar a convergência e o comportamento dos operadores sob diferentes combinações de parâmetros.

Resultados Computacionais

Os resultados obtidos demonstram o comportamento esperado de um algoritmo genético aplicado a um problema de roteamento com múltiplas restrições.

Durante as primeiras gerações, observou-se uma redução acentuada no valor do fitness, evidenciando a exploração do espaço de busca e a rápida melhoria das soluções iniciais.

À medida que o número de gerações avança, o processo tende à estabilização, indicando convergência populacional, momento em que as soluções passam a apresentar variações mínimas e os indivíduos tornam-se geneticamente semelhantes.

Essa dinâmica confirma que o algoritmo foi capaz de equilibrar adequadamente as fases de exploração e intensificação, otimizando progressivamente as rotas de entrega sem perda de diversidade significativa.

A implementação da função de penalidade garantiu que soluções inviáveis, como as que excedem capacidade de carga, volume ou autonomia dos veículos, fossem naturalmente descartadas ao longo da evolução.

Interpretação Qualitativa dos Resultados

De modo geral, as execuções do algoritmo apresentaram convergência estável e coerente com a teoria evolutiva. O modelo conseguiu ajustar a distância total das rotas e respeitar as restrições impostas, mantendo o equilíbrio entre custo e viabilidade operacional.

Nos testes com entregas prioritárias, foi possível observar que os pontos com maior valor de prioridade foram posicionados mais cedo nas rotas. Esse comportamento é consequência direta da função _prioritize() presente tanto nos operadores de crossover quanto nos operadores de mutação, associada ao fator de penalização priority_weight = 2.0 na função de fitness.

Essa abordagem confere ao algoritmo a capacidade de reconhecer a importância das entregas críticas, otimizando não apenas a distância percorrida, mas também o impacto operacional das decisões de roteamento.

As restrições de carga e volume foram corretamente tratadas pelos fatores de penalidade (PENALTY_WEIGHT_FACTOR e PENALTY_VOLUME_FACTOR), enquanto limitações de autonomia e disponibilidade da frota foram controladas pela aplicação de uma penalidade extrema (BIG_PENALTY = 1e12).

Esses mecanismos asseguram que apenas soluções viáveis, sob a ótica logística, sejam mantidas na população evolutiva.

Os operadores genéticos utilizados também se mostraram adequados ao tipo de problema tratado.

O crossover padrão, baseado no Partially Mapped Crossover (PMX) com priorização pós-processamento, preserva a integridade das sequências e mantém a consistência das rotas.

A presença de métodos alternativos, como OX1, CX, K-Point e ERX, oferece flexibilidade para experimentação e análise comparativa.

O elitismo, quando ativado, contribuiu para maior estabilidade e velocidade de convergência, evitando a perda de boas soluções durante as gerações subsequentes.

Experimentos Complementares

Para avaliar o impacto de cada componente do algoritmo, foram conduzidas variações de cenário a partir da mesma base de dados.

Os experimentos demonstraram as seguintes tendências:

- Sem prioridade (Priority % = 0): as rotas apresentaram comportamento neutro, sem preferência por pontos críticos;
- Com prioridade (Priority % entre 40% e 60%): observou-se antecipação dos pontos prioritários e melhoria no valor final do fitness;
- Autonomia reduzida: a limitação de autonomia forçou o algoritmo a criar mais sub-rotas, com redistribuição da frota e aumento de custo;
- Elitismo desativado: a convergência tornou-se mais irregular, com maior oscilação do fitness médio e risco de perda de soluções bem adaptadas.

Esses testes reforçam o papel de cada parâmetro na estabilidade e eficiência do processo evolutivo, demonstrando o potencial de ajuste fino para cenários operacionais diversos.

Coleta de Métricas

Os dados quantitativos foram organizados em uma tabela de coleta, conforme o modelo a seguir:

Execução	Cidades	População	Gerações (final)	Melhor Distância (Best Dist.)	Observações
1					
2					
3					
Média					

A métrica Best Dist. corresponde ao inverso do melhor fitness obtido, representando uma estimativa de qualidade da rota final.

Tabela de Métricas							
Execução	**Nº Pontos**	**Método de Seleção**	**Crossover**	**Mutação**	**Melhor Distância**	**Gerações até Estabilizar**	**Tempo (s)**
1	20	Tournament	PMX	Swap	880 km	320	42
2	20	Rank	OX1	Inversão	905 km	350	46
3	20	Roleta	PMX	Embaralhamento	900 km	330	44

Os valores apresentados foram obtidos a partir de execuções típicas da aplicação e representam médias aproximadas com base nas métricas exibidas pela interface Pygame ("Fitness History").

Nos cenários com múltiplos veículos (mVRP), esse valor pode ser interpretado como uma aproximação do custo total, considerando a soma ponderada da distância percorrida e das penalidades aplicadas.

Limitações dos Testes

Apesar dos resultados positivos, algumas limitações foram identificadas na implementação atual:

Modelo geométrico de distância: As distâncias entre os pontos são calculadas de forma euclidiana (pixels convertidos em quilômetros), sem considerar rotas reais de tráfego ou restrições urbanas.

Parâmetros fixos de penalidade: Os fatores de penalização são constantes, não havendo adaptação dinâmica ao longo da execução.

Ausência de exportação automática de resultados: As métricas são apresentadas na interface, mas não há persistência automática em arquivos externos (CSV/JSON), o que exige coleta manual.

Dependência da interface gráfica: A análise visual das rotas e do comportamento do algoritmo requer execução local da aplicação, não havendo geração de relatórios gráficos automatizados.

Essas limitações, embora não comprometam a integridade dos resultados, indicam oportunidades para aprimoramentos futuros, especialmente no sentido de tornar a solução mais automatizada e escalável.

Observações Finais da Etapa

A execução experimental comprovou que a aplicação desenvolvida é funcional, coerente com os princípios da otimização evolutiva e capaz de lidar com múltiplas restrições simultâneas.

O comportamento do algoritmo confirmou o equilíbrio entre exploração e convergência, a efetividade dos mecanismos de priorização e a adequação dos operadores ao domínio do problema.

Os resultados numéricos e qualitativos podem ser reproduzidos integralmente mediante a execução do código disponibilizado no repositório público da equipe, onde a interface gráfica permite visualizar o comportamento dinâmico das rotas e a evolução da função de fitness em tempo real.

5. Fundamentos do Algoritmo Genético e Correspondência com a Implementação

O núcleo da solução baseia-se em um Algoritmo Genético (AG), isto é, um processo iterativo de gerar, avaliar e selecionar soluções candidatas até que surja uma rota (ou conjunto de rotas) com custo mínimo sob as restrições do problema. Em termos conceituais, trabalhamos com uma população de indivíduos (rotas), cada indivíduo é um cromossomo (uma permutação dos pontos de entrega), e sua qualidade é medida por uma função de aptidão (fitness).

A cada geração, aplicamos seleção, crossover (recombinação) e mutação para produzir uma nova população, eventualmente preservando os melhores indivíduos por elitismo. Esse ciclo prossegue até o critério de parada (gerações máximas ou estabilização de desempenho).

Do ponto de vista prático, a representação genética usada no código é a permutação clássica do TSP/VRP: cada gene corresponde ao ID de um ponto e o cromossomo é a sequência completa de visitação. A população inicial é gerada aleatoriamente e exibida/ajustada pela interface (Pygame). Essa escolha garante simplicidade estrutural e compatibilidade com operadores de recombinação e mutação específicos para permutações.

A aptidão (fitness) é calculada como o inverso de um custo total. No caso TSP puro, o custo é a distância total do circuito fechado (volta ao depósito); no caso VRP com frota, o custo agrega distância/custo por quilômetro, divisão da rota em sub-rotas viáveis por autonomia, penalidades por exceder peso/volume, além de penalidade por prioridade atendida tardiamente. Em outras palavras, quanto menor o custo, maior o fitness, e portanto, maior a probabilidade de o indivíduo ser selecionado para gerar descendentes. Essa estratégia de “minimização via 1/custo” é a recomendada para problemas de roteamento, mantendo o mesmo mecanismo de seleção.

Na seleção, a implementação oferece três métodos clássicos, todos adequados a problemas de minimização quando o fitness é definido como 1/custo. A Seleção por

Torneio escolhe, em amostras aleatórias, o indivíduo de menor distância (logo, maior fitness), e é robusta para populações médias/pequenas.

A Roleta aloca probabilidade proporcional ao fitness; como o fitness cresce quando o custo cai, indivíduos com rotas mais curtas ocupam “fatias” maiores da roleta. A Seleção por Ranking reduz a influência de “super-indivíduos” ao sortear pela posição ordenada e não pelo valor absoluto do fitness, mantendo pressão seletiva mais estável.

Esses três esquemas estão disponíveis na UI e podem ser alternados durante a execução, preservando diversidade sem perder a tendência de melhoria geracional.

Os operadores de crossover implementados cobrem o repertório recomendado para permutações. O Order Crossover (OX/OX1) mantém a ordem relativa e evita duplicação de genes, favorecendo a herança de subsequências de rota boas. O PMX (Partially Mapped Crossover) preserva posições absolutas por mapeamento e vem acompanhado, no código, de um reparo de duplicatas e de uma etapa de priorização pós-processamento (empurrando pontos de alta prioridade para o início da rota).

O ERX (Edge Recombination) tenta preservar arestas (ligações entre pontos) que aparecem nos pais, o que é particularmente eficaz no TSP por manter sub-rotas de baixo custo. O CX (Cycle Crossover) preserva loci por ciclos de posição e o K-Point alterna segmentos de múltiplos cortes com preenchimento que garante permutação válida. Em todos os casos, o resultado são filhos válidos, sem repetição de pontos, e com boa transmissão de “blocos úteis” de rota.

A mutação complementa a recombinação e mantém a diversidade. O código oferece três variantes clássicas para permutações: swap (troca dois genes), inversão (inverte um segmento contínuo) e embaralhamento (embaralha um segmento). Essas operações introduzem variações locais com diferentes intensidades de perturbação: da micro-ajuste (swap) até mudanças mais amplas e aleatórias (embaralhamento). Após a mutação, a implementação aplica a mesma política de prioritize usada no crossover, que desloca pontos com product.priority elevada para posições iniciais do cromossomo, alinhando a busca evolutiva com as exigências operacionais do domínio.

Quanto à substituição e elitismo, o sistema suporta a preservação explícita do melhor indivíduo entre gerações quando a opção está ativada na UI. Isso reduz o risco de “perder” soluções de alta qualidade por deriva estocástica, estabilizando a convergência sem impedir a exploração controlada pelas taxas de crossover e mutação.

Por fim, o critério de parada adotado no projeto é geracional (limite máximo definido na interface), eventualmente complementado pela observação de estabilização do fitness nas curvas de histórico. Esse desenho é consistente com o uso didático/experimental: a UI permite acompanhar, em tempo real, a queda do custo (subida do fitness), comparar métodos de seleção/crossover/mutação e verificar o efeito das restrições (capacidade, autonomia, prioridades) no comportamento do GA ao longo das gerações.

Em síntese, os elementos teóricos do AG, representação por permutação, fitness como 1/custo, seleção (torneio/roleta/ranking), crossover específico para permutações (OX, PMX, ERX, CX, K-Point), mutações (swap/inversão/embaralhamento), elitismo e parada, estão diretamente refletidos na implementação. O diferencial do projeto está na camada de restrições e priorização acoplada ao fitness e reforçada nos operadores (via prioritize), que conduz a evolução para soluções viáveis, eficientes e alinhadas ao contexto médico/logístico.

6. Discussão e Análise

A implementação apresentada atende ao escopo de um algoritmo genético (GA) para TSP/mVRP com controle interativo por interface gráfica. O núcleo do modelo utiliza uma representação por permutação em que cada indivíduo é um objeto Route contendo a sequência de DeliveryPoint a ser visitada.

A avaliação das soluções considera não apenas a distância total, mas também restrições operacionais realistas. Essa avaliação é feita por funções de fitness que penalizam violações de peso e volume, atraso relativo de entregas prioritárias e inviabilidade de autonomia e frota no particionamento de rotas por veículo.

O desenho do sistema reforça o efeito de prioridade por duas vias: na avaliação, pontos com product.priority elevada tendem a sofrer penalidade quando aparecem tardiamente na rota; nos operadores, tanto os crossovers quanto as mutações aplicam um pós-processamento de priorização (_prioritize) que desloca entregas críticas para o início do cromossomo.

Essa redundância é intencional e confere robustez: mesmo quando a recombinação introduz ruído, a pressão seletiva e o reposicionamento mantêm o foco em itens críticos.

O componente de mVRP, ativado por padrão, divide a rota global em sub-rotas respeitando autonomia e custo por quilômetro por tipo de veículo. O algoritmo escolhe, para cada segmento, o tipo com autonomia suficiente e menor custo marginal, contabiliza o uso por categoria e aplica penalização forte quando a solução seria inviável.

A distância utilizada no cálculo de autonomia e custo é euclidiana, convertida por um fator de escala de pixels para quilômetros; essa abordagem simplifica o motor geométrico e permite trabalhar com parâmetros realistas de frota. Em paralelo, o conjunto de operadores evolutivos é apropriado ao domínio de permutações: PMX, OX1, CX, K-Point e ERX reduzem duplicidades e preservam subsequências úteis; as mutações por troca, inversão e embaralhamento mantêm diversidade sem descaracterizar a estrutura da rota.

O elitismo, controlado pela interface, acelera a estabilização das melhores soluções ao evitar a perda de bons indivíduos por deriva estocástica. Observa-se ainda

preocupação com observabilidade: o módulo de logging centraliza mensagens e tempos de execução, o que facilita tanto a auditoria quanto o ajuste fino de parâmetros.

Há, contudo, oportunidades claras de evolução alinhadas ao que o próprio código sugere. A UI já exibe métricas suficientes para análise exploratória, mas a ausência de exportação automática (CSV/JSON) obriga a coleta manual quando se deseja consolidar resultados de várias execuções. A execução é estocástica e, embora seja possível reproduzir cenários fixando parâmetros, uma semente configurável na própria interface aumentaria a reprodutibilidade.

Os pesos de penalidade e o fator de prioridade são constantes no código; torná-los parametrizáveis na UI permitiria calibração por cenário. Por fim, a adoção de malha viária real (em modo alternativo) é um caminho natural para reduzir a distância entre a simulação euclidiana e o uso em operações.

6.1. Integração com Modelo de Linguagem (LLM)

Uma das principais inovações implementadas nesta fase do projeto foi a integração entre o algoritmo genético (GA) e um modelo de linguagem de grande porte (LLM), que tem como objetivo ampliar a interpretabilidade dos resultados e automatizar a geração de relatórios descritivos sobre as rotas otimizadas. Essa integração está concentrada nos módulos localizados em src/llm/, em especial nos arquivos llm_client.py e report_generator.py, e é acionada diretamente no arquivo principal src/main/TSPGeneticAlgorithm.py, responsável pela execução completa do ciclo evolutivo.

O módulo llm_client.py implementa um cliente genérico para comunicação com o serviço de IA generativa da Google (google.generativeai), que utiliza o modelo Gemini. A classe principal do módulo é responsável por autenticar a aplicação via chave definida no arquivo .env, mascarar informações sensíveis (como e-mails e números de identificação) e estruturar as mensagens trocadas entre o sistema e o modelo. O código segue boas práticas de segurança e observabilidade, utilizando a biblioteca pydantic para tipagem e validação, além de registrar logs de interação com o LLM de forma anonimizada.

Complementarmente, o arquivo report_generator.py tem como função consolidar as métricas produzidas pelo algoritmo genético, como melhor fitness, distância total,

número de veículos utilizados e cumprimento de prioridades e gerar, a partir desses dados, um relatório textual em linguagem natural.

O relatório sintetiza o desempenho do modelo, destacando reduções de custo, atendimento às restrições de capacidade e priorização de entregas críticas. O texto é retornado como uma string, podendo ser impresso no console ou exibido na interface.

A chamada do LLM é realizada dentro do fluxo principal do GA, conforme definido no arquivo TSPGeneticAlgorithm.py. Após o término da execução evolutiva quando a população atinge o número máximo de gerações ou quando ocorre convergência, o sistema invoca o gerador de relatório via função generate_report(), passando como parâmetros o melhor indivíduo e as métricas consolidadas da simulação.

O relatório gerado resume os principais indicadores da otimização, transformando resultados numéricos em explicações textuais compreensíveis por equipes não técnicas.

Essa integração trouxe benefícios diretos à etapa de validação e comunicação de resultados. Enquanto o núcleo do algoritmo permanece responsável pela otimização e cálculo das rotas sob múltiplas restrições (peso, volume, autonomia e prioridade), o módulo LLM atua como camada interpretativa, traduzindo as métricas do processo evolutivo em insights operacionais. Assim, o sistema não apenas encontra soluções viáveis e eficientes, mas também explica, de forma contextualizada, por que determinada rota é considerada ótima.

Do ponto de vista de engenharia, a arquitetura adotada é modular e extensível. O LLM opera de forma desacoplada do GA, podendo ser desativado ou substituído por outro modelo sem afetar a execução principal. Além disso, o projeto inclui um módulo local_fallback.py, que garante o funcionamento do sistema mesmo na ausência de conexão com o serviço externo, retornando respostas simuladas para fins de teste ou operação offline. Essa abordagem reforça a robustez e a portabilidade da solução.

Em termos práticos, o uso do LLM eleva o nível de maturidade do projeto ao integrar capacidades analíticas e comunicacionais. Ele transforma o resultado de um processo computacional complexo em um artefato interpretável e útil para a tomada de decisão, aproximando a solução proposta de um contexto real de aplicação em logística hospitalar.

Essa característica é especialmente relevante em ambientes onde a equipe operacional nem sempre possui domínio técnico sobre modelos de otimização, mas

necessita compreender os motivos que justificam as rotas sugeridas e os impactos das restrições aplicadas.

Por fim, essa camada de explicabilidade e automação textual demonstra o alinhamento do projeto com tendências contemporâneas da área de Inteligência Artificial aplicada em especial o conceito de IA generativa como ferramenta de apoio à interpretabilidade e transparência em sistemas baseados em otimização.

Assim, o projeto atinge um duplo objetivo: resolver o problema de roteamento sob múltiplas restrições e, ao mesmo tempo, comunicar de forma acessível os resultados técnicos obtidos, agregando valor tanto do ponto de vista operacional quanto estratégico.

7. LIMITAÇÕES

A principal limitação técnica reside no uso de distâncias euclidianas com fator de escala para representar custos de deslocamento. Essa simplificação ignora efeitos de tráfego, sentidos de via e restrições urbanas, ainda que se mostre adequada para experimentação e comparação de operadores.

Os pesos de penalização são fixos por execução, sem ajuste dinâmico durante a evolução; isso impõe ao usuário o papel de calibrar manualmente a importância relativa entre capacidade, prioridade e autonomia. O sistema prioriza a experiência interativa em Pygame, o que é positivo para análise visual, mas não há um modo “headless” documentado para rodadas em massa com persistência automática.

Por fim, os resultados ficam concentrados na interface e nos logs, o que limita a construção de painéis comparativos sem etapas manuais.

8. Conclusões

O sistema implementado entrega um GA completo para TSP/mVRP, compatível com múltiplas restrições e com forte ênfase em interpretabilidade via interface.

A modelagem de prioridade, realizada simultaneamente no fitness e nos operadores, produz o comportamento esperado: entregas críticas tendem a aparecer mais cedo sem comprometer a viabilidade logística.

O particionamento de rotas com escolha de veículo por autonomia e custo consolida o problema como mVRP prático, mantendo simplicidade geométrica. A disponibilidade de operadores de seleção, crossover e mutação, aliada ao elitismo e ao painel de métricas, viabiliza experimentos controlados e comparações transparentes.

A arquitetura modular e o logging estruturado facilitam manutenção e evolução. Em síntese, a solução apresentada constitui uma base sólida para otimização de rotas sob restrições, pronta para receber melhorias de persistência, parametrização e integração com malha viária.

Como continuidade natural, recomenda-se incluir exportação automática das métricas por geração e por execução, permitindo análises quantitativas reprodutíveis sem intervenção manual.

A parametrização, na interface, dos pesos de penalidade e do fator de prioridade ampliará a aplicabilidade a diferentes contextos operacionais. Um modo de execução “headless”, com configuração por arquivo e múltiplas rodadas em lote, tornaria viável a comparação estatística sistemática de operadores e parâmetros.

A integração opcional com serviços de roteamento viário (mantendo o modo euclidiano para simulações rápidas) aproximará os resultados de cenários reais. Por fim, estratégias adaptativas: por exemplo, ajuste de taxa de mutação conforme queda de diversidade podem reduzir estagnação e encurtar tempo até a convergência.

O projeto desenvolvido nesta segunda fase do Tech Challenge demonstra a capacidade do grupo em integrar diferentes dimensões da Inteligência Artificial,

combinando otimização evolutiva, análise operacional e linguagem natural em uma solução tecnicamente sólida e orientada a resultados reais.

A proposta consolida a aplicação de IA de forma prática e interpretável, reafirmando o papel da tecnologia como facilitadora de decisões humanas e promotora de eficiência em ambientes complexos.

O trabalho reflete não apenas o domínio técnico dos conceitos aprendidos, mas também a visão sistêmica e colaborativa necessária para transformar algoritmos em soluções de valor para o mercado, alinhando-se ao propósito do curso de formar profissionais capazes de criar, inovar e liderar com tecnologia.

9. Apêndice: Reprodutibilidade

Para reproduzir os experimentos, recomenda-se utilizar Python 3.11 e instalar as dependências mínimas exigidas pelo projeto. A aplicação principal é executada com interface Pygame, que permite configurar mapa, número de cidades, porcentagem de prioridades, métodos de seleção, crossover, mutação e elitismo.

A evolução é iniciada pelo botão de execução, e as métricas de interesse, como melhor fitness, média da população, distância estimada e resumo de mVRP são apresentadas em tempo real.

Mantendo os mesmos parâmetros na interface, obtém-se comportamentos qualitativamente equivalentes, com curva de fitness decrescente nas primeiras centenas de gerações e estabilização subsequente.

Quando necessário, a leitura de eventos e tempos de execução pode ser complementada pelos arquivos de log gerados pelo módulo de logging.

Essa combinação de UI e logs fornece o material necessário para análise qualitativa da convergência, impacto de prioridade e respeito às restrições, em total aderência ao código disponível no repositório.