FIAP - Faculdade de Informática e Administração Paulista

Pós-Graduação em Inteligência Artificial para Devs

OTIMIZADOR DE ROTAS

COM INTELIGÊNCIA ARTIFICIAL

Algoritmo Genético para Solução do Problema de Roteamento de Veículos (VRP)

TECH CHALLENGE - FASE 2

Grupo "Sala 14"

Adriana Martins de Souza - RM 368050
Diego Oliveira da Silva RM 367964
Eduardo Nicola F. Zagari - RM 368021
Renan de Assis Torres - RM 368513

São Paulo

Janeiro de 2026

Grupo "Sala 14"

OTIMIZADOR DE ROTAS COM INTELIGÊNCIA ARTIFICIAL

Documentação Técnica do Algoritmo Genético para Solução do TSP e VRP em Entregas Hospitalares

Relatório técnico apresentado como parte dos requisitos para aprovação no Tech Challenge da Fase 2 do curso de Pós-Graduação em Inteligência Artificial para Devs da FIAP.

Orientador: Prof. Sérgio Polimante

São Paulo

Janeiro de 2026

Disponibilização do Código e Materiais

Todo o código desenvolvido neste projeto do algoritmo genético para resolver o problema de roteamento e da integração com IA Generativa, bem como esta documentação e o link para o vídeo demonstrativo estão disponíveis publicamente em:

GitHub: https://github.com/Zagari/routing-optimizer-ai

Conteúdo Disponível:

- Conjuntos de endereços de teste (CSV) utilizados no projeto
- Código-fonte completo do Algoritmo Genético em Python
- Código-fonte da integração com IA Generativa (OpenAI API)
- Aplicação Streamlit para visualização interativa do Algoritmo Genético e da IA Generativa
- Este Relatório Técnico completo em PDF

Documentação:

- README completo com instruções de uso
- Este Relatório Técnico em formato PDF

Vídeo demonstrativo: https://www.youtube.com/watch?v=ZXJC8kMe_Sc

Chave OpenAI: necessário usar chave própria para executar a integração com IA Generativa ou solicitá-la via Discord aos integrantes do grupo:

BÔNUS – Deploy em Nuvem:
A aplicação está disponível online para demonstração (somente http://, sem https://):

http://100.30.130.165:8501

Infraestrutura como Código (Terraform): O diretório `infra/` contém código Terraform para deploy da aplicação em sua própria conta AWS.

Licença: MIT License (código aberto para fins educacionais e de pesquisa)

Resumo

Este documento apresenta a documentação técnica completa do Algoritmo Genético implementado para resolver o Problema do Caixeiro Viajante (TSP) e o Problema de Roteamento de Veículos (VRP). O projeto foi desenvolvido como parte do Tech Challenge da Fase 2 do curso de Pós-Graduação em Inteligência Artificial para Devs da FIAP.

O Algoritmo Genético (AG) é uma metaheurística inspirada na evolução natural, capaz de encontrar boas soluções para problemas de otimização combinatória em tempo razoável. A implementação inclui melhorias em relação ao algoritmo genético tradicional: (1) **inicialização híbrida** com 10% da população usando heurística Nearest Neighbor; (2) **Route-Based Crossover** que preserva rotas inteiras em vez de destruí-las; (3) **múltiplas mutações** (1-3) por indivíduo para maior exploração; (4) **busca local 2-opt** aplicada aos elites para refinamento; (5) **deep copy dos elites** para evitar corrupção de dados; e (6) **convergência antecipada** que para automaticamente após 20% de max_epochs sem melhoria.

O sistema foi projetado para otimizar rotas de entregas hospitalares, considerando múltiplas restrições práticas: capacidade de carga dos veículos (200 kg), autonomia de deslocamento (1500 px), e prioridades de entrega (crítico, urgente, normal). A função de fitness combina distância total com penalidades ponderadas para violações de restrições: 1000x para excesso de capacidade, 50x/20x para prioridades incorretas, e 100x por veículo utilizado.

Os parâmetros padrão do algoritmo incluem: população de 200 indivíduos (10% híbrida), probabilidade de mutação de 60% com 1-3 mutações por indivíduo, máximo de 1000 gerações (com parada antecipada por estagnação), torneio de tamanho 5, e preservação dos 2 melhores indivíduos com busca local 2-opt. A implementação está disponível em duas versões: uma enhanced com visualização Pygame para fins didáticos, e uma refatorada com interface OOP (classe VRPSolver) para integração em sistemas de produção.

Adicionalmente, o sistema integra capacidades de IA Generativa através da API da OpenAI, utilizando o modelo `gpt-4o-mini`. A classe `RouteAssistant` oferece quatro funcionalidades principais: geração de instruções detalhadas para motoristas, relatórios profissionais de eficiência, chat interativo sobre as rotas otimizadas, e correção inteligente de endereços com falha de geocodificação. Esta integração demonstra a complementaridade entre técnicas de otimização clássicas e Large Language Model (LLM) modernos.

Palavras-chave: Algoritmo Genético; TSP; VRP; Otimização Combinatória; Metaheurística; Roteamento de Veículos; 2-opt; Convergência Antecipada; IA Generativa; LLM.

Abstract

This document presents the complete technical documentation of the Genetic Algorithm implemented to solve the Traveling Salesman Problem (TSP) and the Vehicle Routing Problem (VRP). The project was developed as part of Phase 2 Tech Challenge of the Postgraduate course in Artificial Intelligence for Developers at FIAP.

The Genetic Algorithm (GA) is a metaheuristic inspired by natural evolution, capable of finding good solutions for combinatorial optimization problems in reasonable time. The implementation includes improvements over the traditional genetic algorithm: (1) **hybrid initialization** with 10% of the population using Nearest Neighbor heuristic; (2) **Route-Based Crossover** that preserves entire routes instead of destroying them; (3) **multiple mutations** (1-3) per individual for greater exploration; (4) **2-opt local search** applied to elites for refinement; (5) **deep copy of elites** to prevent data corruption; and (6) **early convergence** that automatically stops after 20% of max_epochs without improvement.

The system was designed to optimize hospital delivery routes, considering multiple practical constraints: vehicle load capacity (200 kg), travel autonomy (1500 px), and delivery priorities (critical, urgent, normal). The fitness function combines total distance with weighted penalties for constraint violations: 1000x for capacity excess, 50x/20x for incorrect priorities, and 100x per vehicle used.

Default algorithm parameters include: population of 200 individuals (10% hybrid), 60% mutation probability with 1-3 mutations per individual, maximum of 1000 generations (with early stopping on stagnation), tournament size of 5, and preservation of the top 2 individuals with 2-opt local search. The implementation is available in two versions: an enhanced one with Pygame visualization for educational purposes, and a refactored one with OOP interface (VRPSolver class) for production system integration.

Additionally, the system integrates Generative AI capabilities through the OpenAI API, using the `gpt-4o-mini` model. The `RouteAssistant` class provides four main functionalities: detailed driver instruction generation, professional efficiency reports, interactive chat about optimized routes, and intelligent address correction for geocoding failures. This integration demonstrates the complementarity between classical optimization techniques and modern Large Language Models (LLMs).

Keywords: Genetic Algorithm; TSP; VRP; Combinatorial Optimization; Metaheuristics; Vehicle Routing; 2-opt; Early Convergence; Generative AI; LLM.

Sumário

Lista de Abreviaturas e Siglas

AG Algoritmo Genético. i

API Application Programming Interface. i, 38

GPT Generative Pre-trained Transformer. 38

LLM Large Language Model. i, 4, 38

OX Order Crossover. i

PMX Partially Mapped Crossover. i

TSP Traveling Salesman Problem. i, 1

VRP Vehicle Routing Problem. i, 1

1

Introdução

Este capítulo apresenta uma visão geral do projeto, contextualizando os problemas de otimização abordados e a arquitetura do sistema implementado.

1.1 Contexto e Motivação

O projeto implementa um **Algoritmo Genético (AG)** para resolver dois problemas clássicos de otimização combinatória:

1. **Traveling Salesman Problem (TSP) (Problema do Caixeiro Viajante):** Encontrar a rota mais curta que visita todas as cidades exatamente uma vez e retorna à origem.
2. **Vehicle Routing Problem (VRP) (Problema de Roteamento de Veículos):** Extensão do TSP para múltiplos veículos com restrições de capacidade, autonomia e prioridades de entrega.

1.2 O Problema do Caixeiro Viajante (TSP)

Imagine que você é um vendedor que precisa visitar 10 cidades. Você quer encontrar a rota mais curta que:

- Visite **todas** as cidades exatamente uma vez
- Retorne ao ponto de partida

Complexidade Computacional

Com 10 cidades, existem **3.628.800** rotas possíveis (10!). Com 20 cidades, são mais de **2 quintilhões**! É impossível testar todas as combinações em tempo razoável.

1.3 O Problema de Roteamento de Veículos (VRP)

O VRP é uma extensão do TSP para cenários reais com múltiplas restrições:

Múltiplos veículos: Vários caminhões fazendo entregas simultaneamente

Capacidade: Cada veículo carrega no máximo X kg

Autonomia: Cada veículo percorre no máximo Y km

Prioridades: Algumas entregas são mais urgentes que outras

1.4 Por que usar Algoritmo Genético?

Algoritmos Genéticos encontram **boas soluções** (não necessariamente a melhor) em **tempo razoável**, inspirando-se na evolução natural:

Metaheurística

Algoritmos Genéticos são **metaheurísticas** — encontram soluções boas, mas não garantem a ótima. Para problemas grandes, a solução ótima é praticamente impossível de encontrar de qualquer forma.

1.5 Arquitetura do Sistema

A Figura 1.1 apresenta a arquitetura geral do Algoritmo Genético implementado.

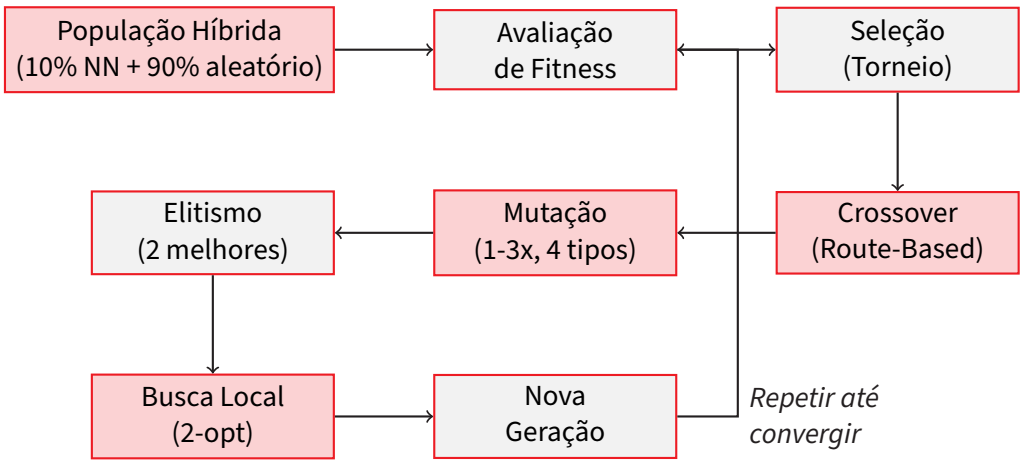

Figura 1.1: Arquitetura do Algoritmo Genético para TSP/VRP com as melhorias implementadas (destacadas em vermelho claro).

1.6 Estrutura de Arquivos

O projeto está organizado em dois módulos principais:

Tabela 1.1: Estrutura de arquivos do projeto

Arquivo	Descrição
`genetic_algorithm_tsp_enhanced/`	
`genetic_algorithm_enhanced.py`	Implementação enhanced do GA
`tsp_enhanced.py`	Sistema principal de entregas hospitalares
`draw_functions_enhanced.py`	Funções de visualização Pygame
`benchmark_att48.py`	Dataset benchmark com 48 cidades
`src/routing_optimizer/genetic_algorithm/`	
`config.py`	Configuração do GA (GAConfig)
`core.py`	Funções core do GA
`vrp.py`	Classe VRPSolver
`src/routing_optimizer/llm/`	
`openai_client.py`	Cliente OpenAI (RouteAssistant)

1.7 Organização do Documento

Este relatório está organizado da seguinte forma:

Capítulo 2 – Conceitos Fundamentais: Apresenta a terminologia do Algoritmo Genético e a estrutura de dados utilizada.

Capítulo 3 – Distâncias e População: Descreve as funções de cálculo de distância e geração da população inicial.

Capítulo 4 – Fitness e Seleção: Detalha a função de fitness com penalidades e o operador de seleção por torneio.

Capítulo 5 – Crossover e Mutação: Apresenta os operadores genéticos de recombinação e mutação.

Capítulo 6 – Algoritmo Principal: Descreve o loop principal e o fluxo de execução completo.

Capítulo 7 – Configuração: Detalha os parâmetros configuráveis e a classe VRPSolver.

Capítulo 8 – Integração com IA Generativa: Descreve a integração com a API da OpenAI para adicionar capacidades de LLM ao sistema, incluindo geração de instruções para motoristas, relatórios de eficiência, chat interativo e correção de endereços.

Capítulo 9 – BÔNUS: Deploy em Nuvem: Apresenta a infraestrutura como código (Terraform) para deploy da aplicação em ambiente AWS, incluindo EC2, Secrets Manager e configuração automatizada.

Capítulo 10 – FAQ e Glossário: Responde perguntas frequentes e apresenta a terminologia utilizada.

2

Conceitos Fundamentais

Este capítulo apresenta a terminologia do Algoritmo Genético e a estrutura de dados utilizada na implementação.

2.1 Terminologia do Algoritmo Genético

Um Algoritmo Genético é uma metaheurística inspirada no processo de seleção natural. A Tabela 2.1 apresenta os principais conceitos utilizados.

Tabela 2.1: Terminologia do Algoritmo Genético no contexto TSP/VRP

Termo	**Definição no Contexto do TSP/VRP**
Indivíduo	Uma solução completa (rota ou conjunto de rotas)
Gene	Uma cidade/entrega na rota
Cromossomo	A sequência de genes que forma uma rota
População	Conjunto de soluções candidatas
Fitness	Qualidade da solução (distância total + penalidades)
Geração	Uma iteração do algoritmo
Crossover	Combinação de duas soluções para criar uma nova
Mutação	Pequena alteração aleatória em uma solução
Seleção	Escolha de indivíduos para reprodução
Elitismo	Preservação dos melhores indivíduos

2.2 Estrutura dos Dados

2.2.1 Representação de uma Cidade/Entrega

Uma localização é representada por coordenadas (x, y):

```
# Cidade simples (TSP)
cidade = (500, 300)  # posição x=500, y=300

# Entrega completa (VRP)
entrega = {
    'id': 1,
    'location': (500, 300),
    'priority': 1,       # 1=Crítico, 2=Urgente, 3=Normal
    'weight': 15.0,      # kg
    'item_type': "Medicamento Oncológico"
}
```

Listing 2.1: Representação de cidade e entrega

2.2.2 Representação de uma Solução

Solução TSP

Uma lista ordenada de cidades representando a ordem de visitação:

```
rota_tsp = [(100, 200), (300, 400), (500, 100), (200, 300)]
# Significado: Visite nesta ordem e volte ao início
```

Listing 2.2: Representação de solução TSP

Solução VRP

Lista de rotas, uma para cada veículo:

```
rotas_vrp = [
    [entrega1, entrega4, entrega7],  # Veículo 1
    [entrega2, entrega5],            # Veículo 2
    [entrega3, entrega6, entrega8],  # Veículo 3
]
```

Listing 2.3: Representação de solução VRP

2.2.3 Estrutura da População

Uma população é um conjunto de soluções candidatas:

```
populacao = [
    solucao_1,  # Uma rota possível
    solucao_2,  # Outra rota possível
    solucao_3,  # Mais uma...
    # ... (tipicamente 100-500 soluções)
]
```

Listing 2.4: Estrutura da população

A Figura 2.1 ilustra a estrutura hierárquica de uma população VRP.

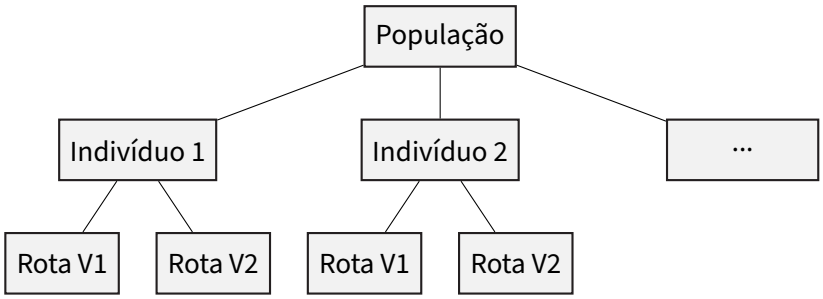

Figura 2.1: Estrutura hierárquica da população VRP.

2.3 Prioridades de Entrega

O sistema implementa três níveis de prioridade para as entregas, conforme a Tabela 2.2.

Tabela 2.2: Níveis de prioridade das entregas

Prioridade	Nível	Itens Exemplo	Peso Típico
1	Crítico	Medicamento Oncológico, Insulina Especial, Anticoagulante	2-15 kg
2	Urgente	Antibióticos, Analgésicos, Soro Fisiológico	5-30 kg
3	Normal	Materiais Cirúrgicos, EPIs, Curativos	10-50 kg

A distribuição padrão das entregas é:

- 20% entregas críticas
- 33% entregas urgentes
- 47% entregas normais

3

Distâncias e População Inicial

Este capítulo descreve as funções de cálculo de distância e a geração da população inicial do algoritmo genético.

3.1 Cálculo de Distâncias

3.1.1 Distância Euclidiana

Arquivo: `genetic_algorithm_enhanced.py:6-8` e `core.py:18-28`

```
1 def calculate_distance(city1: Tuple[int, int], city2:
    ↪ Tuple[int, int]) -> float:
2     """Calcula distância euclidiana entre duas cidades"""
3     return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
        ↪ city2[1])**2)
```

Listing 3.1: Função calculate_distance

Propósito: Calcula a distância euclidiana entre dois pontos no plano 2D.
Fórmula matemática:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.1}$$

Parâmetros:

- `point1`: Tupla (x, y) do primeiro ponto
- `point2`: Tupla (x, y) do segundo ponto

Exemplo prático:

```python
ponto_a = (0, 0)
ponto_b = (3, 4)

# Cálculo manual:
# dx = 3 - 0 = 3
# dy = 4 - 0 = 4
# distância = sqrt(3^2 + 4^2) = sqrt(9 + 16) = sqrt(25) = 5

distancia = calculate_distance(ponto_a, ponto_b)
print(distancia)  # 5.0
```

Listing 3.2: Exemplo de cálculo de distância

Por que distância euclidiana?

- Simples de calcular
- Aproximação razoável para rotas urbanas
- Em aplicações reais, pode-se usar distâncias de ruas (Google Maps API)

3.1.2 Distância Total de uma Rota

Arquivo: `genetic_algorithm_enhanced.py:10-19` e `core.py:31-60`

```python
def calculate_route_distance(
    route: List[int],
    distance_matrix: np.ndarray,
    depot_index: int = 0,
) -> float:
    """Calcula distância total de uma rota usando matriz de
       ↪ distâncias"""
    if not route:
        return 0.0

    total = 0.0
    # Depósito até primeira parada
    total += distance_matrix[depot_index, route[0]]

    # Entre paradas consecutivas
    for i in range(len(route) - 1):
        total += distance_matrix[route[i], route[i + 1]]
```

```
17
18      # Última parada de volta ao depósito
19      total += distance_matrix[route[-1], depot_index]
20
21      return total
```

Listing 3.3: Função calculate_route_distance

Propósito: Calcula a distância total de uma rota usando uma matriz de distâncias pré-computada.

Funcionamento:

1. Soma a distância do depósito até a primeira parada
2. Soma as distâncias entre paradas consecutivas
3. Soma a distância da última parada de volta ao depósito

Exemplo prático:

```
1  rota = [(0, 0), (3, 0), (3, 4), (0, 4)]
2
3  # Cálculo:
4  # (0,0) -> (3,0) = 3
5  # (3,0) -> (3,4) = 4
6  # (3,4) -> (0,4) = 3
7  # (0,4) -> (0,0) = 4  (retorno)
8  # Total = 3 + 4 + 3 + 4 = 14
9
10 distancia_total = calculate_route_distance(rota)
11 print(distancia_total)  # 14.0
```

Listing 3.4: Exemplo de cálculo de rota

3.2 Geração da População Inicial

3.2.1 População para TSP Simples

Arquivo: `genetic_algorithm_enhanced.py:98-106`

```
1  def generate_random_population(cities: List[Tuple[int, int]],
2                                 population_size: int) ->
                                     ↪ List[List[Tuple[int,
                                     ↪ int]]]:
3      """Gera população inicial aleatória para TSP simples"""
```

```
    population = []
    for _ in range(population_size):
        individual = cities.copy()
        random.shuffle(individual)
        population.append(individual)
    return population
```

Listing 3.5: Função generate_random_population para TSP

Propósito: Cria uma população inicial de soluções aleatórias para o TSP.
Funcionamento:

1. Para cada indivíduo na população
2. Copia a lista de cidades
3. Embaralha aleatoriamente (cada embaralhamento é uma rota diferente)

Por que aleatório?

- Garante **diversidade** inicial
- O algoritmo vai "evoluir" a partir dessas soluções
- Soluções ruins serão descartadas nas próximas gerações

3.2.2 População para VRP

Arquivo: `genetic_algorithm_enhanced.py:108-137`

Para VRP, além de embaralhar, precisamos **distribuir as entregas entre os veículos**:

```
def generate_random_population_vrp(deliveries: List[Dict],
                                   num_vehicles: int,
                                   population_size: int) ->
                                      ↪ List[List[List[Dict]]]:
    """Gera população inicial para VRP"""
    population = []

    for _ in range(population_size):
        # Embaralha entregas
        shuffled = deliveries.copy()
        random.shuffle(shuffled)

        # Divide entre veículos (round-robin)
        routes = [[] for _ in range(num_vehicles)]
```

```
14          for i, delivery in enumerate(shuffled):
15              vehicle_idx = i % num_vehicles  # Distribui
                    ↪ ciclicamente
16              routes[vehicle_idx].append(delivery)
17
18          population.append(routes)
19
20      return population
```

Listing 3.6: Função generate_random_population_vrp

Exemplo de distribuição round-robin:

Tabela 3.1: Exemplo de distribuição round-robin com 6 entregas e 2 veículos

Índice	Entrega	Veículo (i % 2)
0	E3	0
1	E1	1
2	E6	0
3	E4	1
4	E2	0
5	E5	1
Resultado: Veículo 0: [E3, E6, E2] Veículo 1: [E1, E4, E5]		

3.2.3 Inicialização Híbrida (Melhoria #1)

Arquivo: `core.py:424-485`

Problema da Inicialização 100% Aleatória

Com população inicial totalmente aleatória, o algoritmo começa com soluções muito ruins e precisa de muitas gerações para convergir. Para datasets grandes (500+ entregas), isso pode ser muito lento.

Solução: Usar **inicialização híbrida** com 10% da população usando a heurística **Nearest Neighbor** (vizinho mais próximo) e 90% aleatória.

Heurística Nearest Neighbor

A heurística "sempre visita o vizinho mais próximo":

```python
def nearest_neighbor_solution(distance_matrix, num_vehicles):
    """Gera uma solução usando heurística do vizinho mais
       ↪ próximo."""
    num_locations = distance_matrix.shape[0] - 1  # Excluindo
       ↪ depósito
    unvisited = set(range(1, num_locations + 1))
    routes = [[] for _ in range(num_vehicles)]

    for vehicle_idx in range(num_vehicles):
        if not unvisited:
            break
        current = 0  # Começa no depósito
        while unvisited:
            # Encontra o vizinho mais próximo
            nearest = min(unvisited,
                          key=lambda x:
                             ↪ distance_matrix[current, x])
            routes[vehicle_idx].append(nearest)
            unvisited.remove(nearest)
            current = nearest

    return routes
```

Listing 3.7: Função nearest_neighbor_solution

População Híbrida

```python
def generate_hybrid_population(
    num_locations: int,
    num_vehicles: int,
    population_size: int,
    distance_matrix: np.ndarray,
    heuristic_ratio: float = 0.1,  # 10% heurística
) -> List[List[List[int]]]:
    """Gera população híbrida: parte heurística + parte aleatória."""

    num_heuristic = max(1, int(population_size * heuristic_ratio))
    num_random = population_size - num_heuristic

    population = []

    # Parte heurística (Nearest Neighbor)
    base_solution = nearest_neighbor_solution(distance_matrix, num_vehicles)
    for _ in range(num_heuristic):
        # Adiciona variação para diversidade
        solution = [route.copy() for route in base_solution]
        population.append(solution)

    # Parte aleatória
    for _ in range(num_random):
        solution = generate_random_vrp_solution(num_locations, num_vehicles)
        population.append(solution)

    return population
```

Listing 3.8: Função generate_hybrid_population

Benefícios da Inicialização Híbrida

- **Convergência mais rápida:** Começa com soluções melhores
- **Mantém diversidade:** 90% ainda é aleatório para evitar ótimos locais
- **Configurável:** Parâmetro `heuristic_ratio` (padrão: 0.1)

4

Fitness e Seleção

Este capítulo descreve a função de fitness que avalia a qualidade das soluções e o operador de seleção por torneio.

4.1 Função de Fitness

A função de fitness **avalia a qualidade** de uma solução. No contexto de otimização de rotas, **menor valor = melhor solução**.

4.1.1 Fitness para TSP Simples

Arquivo: `genetic_algorithm_enhanced.py:21-23`

```
1 def calculate_fitness(individual: List[Tuple[int, int]]) ->
    ↪ float:
2     """Calcula fitness básico (distância total)"""
3     return calculate_route_distance(individual)
```

Listing 4.1: Função de fitness para TSP

Propósito: Avalia a qualidade de uma solução TSP simples. O fitness é apenas a distância total da rota.

4.1.2 Fitness para VRP (com Penalidades)

Arquivo: `genetic_algorithm_enhanced.py:25-96` e `core.py:81-128`

O VRP é mais complexo porque precisamos considerar múltiplas restrições. A Tabela 4.1 apresenta os componentes do fitness.

Tabela 4.1: Componentes do fitness VRP

Componente	**Descrição**	**Peso**
Distância Total	Soma de todas as rotas	1x
Penalidade Capacidade	Excesso de peso por veículo	1000x por kg
Penalidade Autonomia	Excesso de distância por veículo	1000x por pixel
Penalidade Prioridade	Entregas críticas não feitas primeiro	50x/20x
Penalidade Veículos	Número de veículos usados	100x por veículo

Fórmula do Fitness:

$$\text{fitness} = \text{distância\_total} + \text{penalidade\_capacidade} + \text{penalidade\_prioridade} + \text{penalidade\_veículos} \quad (4.1)$$

```python
def calculate_fitness_vrp(routes, depot, priorities,
                          vehicle_capacity, vehicle_autonomy,
                          penalty_weight=1000) -> float:

    total_distance = 0
    total_penalty = 0
    priority_penalty = 0

    for route in routes:
        # 1. Calcular distância da rota
        route_distance = calcular_distancia_completa(route,
            ↪ depot)
        total_distance += route_distance

        # 2. Calcular carga total
        route_load = sum(d['weight'] for d in route)

        # 3. PENALIDADE: Excesso de capacidade
        if route_load > vehicle_capacity:
            total_penalty += (route_load - vehicle_capacity)
                ↪ * penalty_weight

        # 4. PENALIDADE: Excesso de autonomia
        if route_distance > vehicle_autonomy:
            total_penalty += (route_distance -
                ↪ vehicle_autonomy) * penalty_weight

        # 5. PENALIDADE: Prioridade errada
```

```
26          for posicao, delivery in enumerate(route):
27              if delivery['priority'] == 1:  # Crítico
28                  priority_penalty += posicao * 50
29              elif delivery['priority'] == 2:  # Urgente
30                  priority_penalty += posicao * 20
31
32      # 6. PENALIDADE: Número de veículos
33      num_vehicles_penalty = len([r for r in routes if r]) * 100
34
35      return total_distance + total_penalty + priority_penalty
           ↪ + num_vehicles_penalty
```

Listing 4.2: Função de fitness para VRP

Exemplo de Penalidade por Capacidade

```
Capacidade máxima: 100 kg
Rota do Veículo 1: carga = 120 kg

Excesso: 120 - 100 = 20 kg
Penalidade: 20 * 1000 = 20.000 pontos adicionados ao fitness

Isso faz a solução parecer MUITO pior, forçando o algoritmo
a encontrar soluções que respeitam as restrições.
```

4.1.3 Penalidade de Prioridade

Entregas críticas devem ser feitas **primeiro**. A penalidade aumenta com a **posição** na rota:

$$\text{penalidade} = \text{posição} \times \text{peso\_prioridade} \tag{4.2}$$

Exemplo:

```
Rota: [Normal, Urgente, CRÍTICO, Normal]
                        ^ posição 2

Penalidade = 2 * 50 = 100

Se o crítico estivesse primeiro:
Rota: [CRÍTICO, Normal, Urgente, Normal]
       ^ posição 0
Penalidade = 0 * 50 = 0  (nenhuma!)
```

4.2 Seleção por Torneio

A seleção escolhe quais indivíduos vão "reproduzir" (gerar filhos).

Arquivo: `genetic_algorithm_enhanced.py:276-283` e `core.py:174-192`

```python
def tournament_selection(population: List,
                         fitness_values: List[float],
                         tournament_size: int = 3) -> List:
    """Seleção por torneio"""
    # 1. Escolhe aleatoriamente 'tournament_size' indivíduos
    tournament_indices =
        ↪ random.sample(range(len(population)),
        ↪ tournament_size)

    # 2. Pega o fitness de cada um
    tournament_fitness = [fitness_values[i] for i in
        ↪ tournament_indices]

    # 3. Retorna o MELHOR (menor fitness)
    winner_idx =
        ↪ tournament_indices[tournament_fitness.index(min(tournament_fi
    return population[winner_idx]
```

Listing 4.3: Função tournament_selection

Propósito: Seleciona um indivíduo para reprodução usando competição.

Funcionamento:

1. Seleciona aleatoriamente `tournament_size` indivíduos
2. Compara seus valores de fitness
3. Retorna o indivíduo com menor fitness (melhor solução)

Analogia com a natureza

É como uma competição onde alguns animais competem e o mais adaptado vence e pode se reproduzir.

4.2.1 Exemplo de Torneio

Tabela 4.2: Exemplo de seleção por torneio

Indivíduo	Fitness	Selecionado?
rota_A	500	
rota_B	300	✓ (vencedor)
rota_C	800	
rota_D	450	
rota_E	600	
Torneio de tamanho 3 seleciona índices [1, 2, 4]		
Fitness do torneio: [300, 800, 600]		
Vencedor: rota_B (menor fitness = 300)		

4.2.2 Vantagens da Seleção por Torneio

- Pressão seletiva controlável (ajustando `tournament_size`)
- Não requer ordenação completa da população
- Preserva diversidade genética
- Simples de implementar e eficiente

Parâmetro `tournament_size`:

Maior (ex: 10) Mais pressão seletiva — converge mais rápido, risco de ótimo local

Menor (ex: 2) Mais diversidade — converge mais lento, explora mais

Padrão 3 a 5 (bom equilíbrio)

5

Crossover e Mutação

Este capítulo apresenta os operadores genéticos de recombinação (crossover) e mutação, fundamentais para a exploração do espaço de soluções.

5.1 Crossover (Recombinação)

Crossover combina dois "pais" para criar um "filho" que herda características de ambos.

5.1.1 Desafio Especial do TSP

No TSP, não podemos simplesmente cortar e juntar rotas:

```
Pai 1: [A, B, C, D, E]
Pai 2: [E, D, C, B, A]

Crossover ingênuo (corte no meio):
Filho: [A, B, C] + [B, A] = [A, B, C, B, A]

ERRO! B e A aparecem 2x, D e E faltam!
```

Restrição de Permutação

Em problemas de permutação como TSP, cada cidade deve aparecer **exatamente uma vez**. Crossovers tradicionais não funcionam — precisamos de operadores especializados.

5.1.2 Order Crossover (OX)

Arquivo: `genetic_algorithm_enhanced.py:139-155`

O OX resolve o problema preservando a **ordem relativa** dos genes:

```python
def order_crossover(parent1: List, parent2: List) -> List:
    """Order Crossover (OX) para TSP"""
    size = len(parent1)

    # 1. Escolhe dois pontos de corte aleatórios
    start, end = sorted(random.sample(range(size), 2))

    # 2. Copia segmento do pai 1
    child = [None] * size
    child[start:end] = parent1[start:end]

    # 3. Preenche resto com pai 2 (sem duplicar)
    pointer = end
    for city in parent2[end:] + parent2[:end]:
        if city not in child:
            if pointer >= size:
                pointer = 0
            child[pointer] = city
            pointer += 1

    return child
```

Listing 5.1: Função order_crossover (OX)

Funcionamento Visual:

```
Parent 1: [A, B, C, D, E, F, G, H]
Parent 2: [C, G, A, H, B, D, F, E]

Seleção aleatória: start=3, end=6

Child após copiar segmento do Parent 1:
Child:    [_, _, _, D, E, F, _, _]

Preencher com Parent 2 (ordem: H, B, D, F, E, C, G, A)
- H não está no child -> adiciona
- B não está no child -> adiciona
- D já está -> pula
- F já está -> pula
- E já está -> pula
- C não está -> adiciona
```

```
- G não está -> adiciona
- A não está -> adiciona

Child Final: [C, G, A, D, E, F, H, B]
```

Características:

- Preserva a ordem relativa dos elementos do Parent 2
- Mantém a subsequência do Parent 1
- Garante que todas as cidades aparecem exatamente uma vez

5.1.3 PMX Crossover

Arquivo: `genetic_algorithm_enhanced.py:157-176`

O PMX (Partially Mapped Crossover) preserva **posições absolutas**:

```python
def pmx_crossover(parent1: List, parent2: List) -> List:
    """Partially Mapped Crossover (PMX) para TSP"""
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))

    child = [None] * size
    child[start:end] = parent1[start:end]

    for i in range(start, end):
        if parent2[i] not in child:
            pos = i
            while start <= pos < end:
                pos = parent2.index(parent1[pos])
            child[pos] = parent2[i]

    for i in range(size):
        if child[i] is None:
            child[i] = parent2[i]

    return child
```

Listing 5.2: Função pmx_crossover

Tabela 5.1: Diferença entre OX e PMX

Aspecto	OX (Order Crossover)	PMX (Partially Mapped)
Preserva	Ordem relativa	Posições absolutas
Funcionamento	Preenche sequencialmente	Usa mapeamento
Complexidade	Mais simples	Mais complexo
Uso no projeto	Principal	Alternativo

5.1.4 VRP Crossover — Route-Based (Melhoria #2)

Arquivo: `core.py:195-258`

Problema do Crossover Tradicional

O crossover tradicional (OX/PMX) "achata" as rotas dos pais e redistribui aleatoriamente. Se o Pai 1 tem uma rota ótima `[A, B, C]` para um cluster geográfico, ela é **destruída**! Crossover tradicional = praticamente reinicialização aleatória!

Solução: O **Route-Based Crossover** preserva **rotas inteiras** de um pai:

1. Seleciona k rotas completas do Pai 1 para preservar
2. Copia essas rotas inteiras para o filho
3. Preenche o restante com locais do Pai 2 (sem duplicar)

```python
def vrp_crossover(parent1, parent2):
    """Route-Based Crossover que preserva rotas inteiras."""
    num_vehicles = len(parent1)
    child = [[] for _ in range(num_vehicles)]

    # Encontra rotas não-vazias do Pai 1
    non_empty_p1 = [i for i, r in enumerate(parent1) if r]

    if non_empty_p1:
        # Seleciona rotas aleatórias para preservar
        num_to_keep = max(1, len(non_empty_p1) // 2)
        routes_to_keep = random.sample(non_empty_p1,
            ↪ num_to_keep)

        # Copia rotas selecionadas inteiras
        used_locations = set()
        for idx in routes_to_keep:
```

```
17              child[idx] = parent1[idx].copy()
18              used_locations.update(parent1[idx])
19
20          # Preenche restante com Pai 2 (sem duplicar)
21          remaining = []
22          for route in parent2:
23              for loc in route:
24                  if loc not in used_locations:
25                      remaining.append(loc)
26
27          # Distribui restantes nas rotas vazias
28          empty_routes = [i for i in range(num_vehicles) if not
                ↪ child[i]]
29          for i, loc in enumerate(remaining):
30              route_idx = empty_routes[i % len(empty_routes)]
                  ↪ if empty_routes else i % num_vehicles
31              child[route_idx].append(loc)
32
33      return child
```

Listing 5.3: Função vrp_crossover (Route-Based)

Benefícios do Route-Based Crossover

- **Preserva clusters geográficos:** Rotas boas não são destruídas
- **Mantém boas atribuições:** Entregas ficam nos veículos certos
- **Convergência mais rápida:** Herda estruturas de qualidade dos pais

5.2 Mutação

A mutação introduz **variação aleatória** para explorar novas soluções.

5.2.1 Mutação Simples (TSP)

Arquivo: `genetic_algorithm_enhanced.py:210-215`

```
1  def mutate(individual: List, mutation_probability: float) ->
       ↪ List:
2      """Mutação por troca (swap)"""
3      if random.random() < mutation_probability:
4          idx1, idx2 = random.sample(range(len(individual)), 2)
```

```
            individual[idx1], individual[idx2] =
                ↪ individual[idx2], individual[idx1]
        return individual
```

Listing 5.4: Função mutate para TSP

Exemplo:

```
Antes:  [A, B, C, D, E]
Índices selecionados: 1 e 4
Depois: [A, E, C, D, B]  (trocou B e E)
```

5.2.2 Mutação para VRP (4 Tipos)

Arquivo: `genetic_algorithm_enhanced.py:217-267` e `core.py:243-306`

O VRP usa **4 tipos diferentes** de mutação, escolhidos aleatoriamente:

Tipo 1: swap_within (troca dentro da mesma rota)

```
ANTES:
  Veículo 1: [E1, E2, E3, E4]

DEPOIS (trocou E2 e E4):
  Veículo 1: [E1, E4, E3, E2]
```

Quando é útil: Otimiza a ordem dentro de uma rota.

Tipo 2: swap_between (troca entre rotas)

```
ANTES:
  Veículo 1: [E1, E2, E3]
  Veículo 2: [E4, E5]

DEPOIS (trocou E2 do V1 com E5 do V2):
  Veículo 1: [E1, E5, E3]
  Veículo 2: [E4, E2]
```

Quando é útil: Rebalanceia carga entre veículos.

Tipo 3: reverse (inverte subsequência)

```
ANTES:
  Veículo 1: [E1, E2, E3, E4, E5]
```

```
DEPOIS (inverteu E2, E3, E4):
  Veículo 1: [E1, E4, E3, E2, E5]
```

Operador 2-opt

A mutação `reverse` implementa implicitamente o movimento **2-opt**, um dos operadores mais eficazes para otimização de rotas!

Tipo 4: relocate (move para outra rota)

```
ANTES:
  Veículo 1: [E1, E2, E3]
  Veículo 2: [E4, E5]

DEPOIS (moveu E2 para V2):
  Veículo 1: [E1, E3]
  Veículo 2: [E4, E5, E2]
```

Quando é útil: Esvazia rotas desnecessárias.

5.2.3 Múltiplas Mutações por Indivíduo (Melhoria #3)

Arquivo: `core.py:261-343`

Problema de Uma Única Mutação

Para problemas grandes (500+ entregas), uma única troca tem efeito mínimo no fitness. O indivíduo mutado é praticamente igual ao original.

Solução: Quando a mutação é ativada, aplicar **1 a 3 mutações** aleatórias em vez de apenas uma:

```python
def mutate_vrp(
    individual: List[List[int]],
    mutation_probability: float,
    max_mutations: int = 3,  # Melhoria: 1 a 3 mutações
) -> List[List[int]]:
    """Aplica mutação com 1 a max_mutations operações."""

    if random.random() > mutation_probability:
        return individual  # Não muta

    # Número de mutações: 1 a max_mutations
    num_mutations = random.randint(1, max(1, max_mutations))

    for _ in range(num_mutations):
        _apply_single_mutation(individual)

    return individual

def _apply_single_mutation(individual):
    """Aplica uma única mutação de um dos 4 tipos."""
    mutation_type = random.choice([
        "swap_within", "swap_between", "reverse", "relocate"
    ])
    # ... aplica a mutação escolhida ...
```

Listing 5.5: Função mutate_vrp com múltiplas mutações

Benefícios das Múltiplas Mutações

- **Maior exploração:** Cada indivíduo pode fazer mudanças mais significativas
- **Especialmente útil para datasets grandes:** Onde uma troca tem efeito mínimo
- **Configurável:** Parâmetro `max_mutations_per_individual` (padrão: 3)

5.2.4 Resumo das Mutações

Tabela 5.2: Tipos de mutação e seus efeitos

Tipo	Operação	Efeito
swap_within	Troca dentro da rota	Otimiza ordem local
swap_between	Troca entre rotas	Rebalanceia carga
reverse	Inverte segmento	Movimento 2-opt
relocate	Move para outra rota	Consolida rotas

Nota: Com a melhoria #3, cada indivíduo pode receber de 1 a 3 dessas mutações quando selecionado para mutação (probabilidade de 60%).

6

Algoritmo Principal

Este capítulo descreve o loop principal do algoritmo genético, incluindo elitismo e o fluxo completo de execução.

6.1 Elitismo com Deep Copy (Melhoria #5)

Elitismo preserva os melhores indivíduos de uma geração para a próxima, garantindo que nunca "perdemos" uma boa solução.

Problema do Shallow Copy

O código original fazia `new_population = population[:elite_count]`, que é um **shallow copy**. Se algo modificasse as rotas dos elites (ex: mutação acidental), os elites originais seriam **corrompidos**!

Solução: Fazer **deep copy** segura dos elites:

```
# Ordenar população por fitness (melhor primeiro)
population, fitness = sort_population(population, fitness)

# MELHORIA #5: Deep copy dos 2 melhores (evita corrupção)
new_population = [
    [route.copy() for route in individual]
    for individual in population[:elitism_count]  # padrão: 2
]

# Preencher o resto com filhos
while len(new_population) < POPULATION_SIZE:
    parent1 = tournament_selection(...)
    parent2 = tournament_selection(...)
    child = vrp_crossover(parent1, parent2)  # Route-Based
```

```
15      child = mutate_vrp(child, MUTATION_PROBABILITY,
            ↪ max_mutations=3)
16      new_population.append(child)
```

Listing 6.1: Implementação do elitismo com deep copy

Por que Deep Copy?

Cada rota é uma lista de inteiros. `route.copy()` cria uma nova lista, garantindo que modificações nos filhos não afetam os elites preservados.

6.1.1 Importância do Elitismo

Tabela 6.1: Comparação: com e sem elitismo

Geração	Sem Elitismo	Com Elitismo
1	Fitness = 500	Fitness = 500 (preservado)
2	Fitness = 600 (piorou!)	Fitness = 480 (melhorou)

Sem elitismo

Sem elitismo, boas soluções podem ser perdidas entre gerações, causando regressão na qualidade da população.

6.1.2 Ordenação da População

Arquivo: `genetic_algorithm_enhanced.py:269-274` e `core.py:309-325`

```
1  def sort_population(
2      population: List,
3      fitness_values: List[float]
4  ) -> Tuple[List, List[float]]:
5      """Ordena população por fitness (menor é melhor)"""
6      sorted_pairs = sorted(zip(population, fitness_values),
           ↪ key=lambda x: x[1])
7      sorted_population = [ind for ind, _ in sorted_pairs]
8      sorted_fitness = [fit for _, fit in sorted_pairs]
9      return sorted_population, sorted_fitness
```

Listing 6.2: Função sort_population

Propósito: Ordena a população em ordem crescente de fitness (melhores primeiro).

6.2 Busca Local 2-opt (Melhoria #4)

Arquivo: `core.py:346-422`

O algoritmo genético é bom para **exploração global**, mas ruim para **refinamento local**. Rotas podem ter "cruzamentos" que são facilmente corrigíveis.

O que é 2-opt?

O 2-opt inverte segmentos da rota até não haver mais melhoria. Remove arestas que se cruzam:

```
Antes:  A → B → C → D  (com cruzamento)
Depois: A → C → B → D  (sem cruzamento, menor distância)
```

```python
def two_opt(route, distance_matrix, depot_index=0):
    """Inverte segmentos até não haver mais melhoria."""
    if len(route) < 2:
        return route

    best_route = route.copy()
    best_distance = calculate_route_distance(best_route,
        ↪ distance_matrix, depot_index)
    improved = True

    while improved:
        improved = False
        for i in range(len(best_route) - 1):
            for j in range(i + 2, len(best_route)):
                # Cria rota com segmento invertido
                new_route = best_route[:i+1] +
                    ↪ best_route[i+1:j+1][::-1] +
                    ↪ best_route[j+1:]
                new_distance =
                    ↪ calculate_route_distance(new_route,
                    ↪ distance_matrix, depot_index)

                if new_distance < best_distance:
                    best_route = new_route
                    best_distance = new_distance
                    improved = True
                    break
            if improved:
```

```
                break

    return best_route
```

Listing 6.3: Função two_opt

6.2.1 Aplicação nos Elites

O 2-opt é aplicado **apenas nos elites** para não aumentar muito o tempo de execução:

```
def apply_local_search(individual, distance_matrix,
    ↪ depot_index=0):
    """Aplica 2-opt em todas as rotas de um indivíduo."""
    return [
        two_opt(route, distance_matrix, depot_index)
        for route in individual
    ]

# No loop principal, após elitismo:
if config.local_search_elites_only:
    for i in range(elitism_count):
        new_population[i] =
            ↪ apply_local_search(new_population[i],
            ↪ distance_matrix)
```

Listing 6.4: Função apply_local_search

Benefícios do 2-opt

- **Remove cruzamentos:** Rotas mais curtas e lógicas
- **Refinamento local:** Complementa a exploração global do AG
- **Eficiente:** Aplicado apenas nos 2 melhores (elites)

6.3 Loop Principal

Arquivo: `tsp_enhanced.py:204-249`

```
def evolve_generation(self):
    """Executa UMA geração do algoritmo genético"""

    # ========== 1. AVALIAÇÃO ==========
```

```
    fitness_values = [
        calculate_fitness_vrp(individual, self.depot, ...)
        for individual in self.population
    ]

    # ========== 2. ORDENAÇÃO ==========
    self.population, fitness_values = sort_population(
        self.population, fitness_values
    )

    # ========== 3. REGISTRO ==========
    best_fitness = fitness_values[0]
    best_solution = self.population[0]
    self.best_fitness_values.append(best_fitness)

    # ========== 4. ELITISMO ==========
    elite_size = max(2, POPULATION_SIZE // 10)
    new_population = self.population[:elite_size]

    # ========== 5. REPRODUÇÃO ==========
    while len(new_population) < POPULATION_SIZE:
        parent1 = tournament_selection(self.population,
            ↪ fitness_values, TOURNAMENT_SIZE)
        parent2 = tournament_selection(self.population,
            ↪ fitness_values, TOURNAMENT_SIZE)
        child = vrp_crossover(parent1, parent2)
        child = mutate_vrp(child, MUTATION_PROBABILITY)
        new_population.append(child)

    # ========== 6. SUBSTITUIÇÃO ==========
    self.population = new_population

    return best_fitness, best_solution
```

Listing 6.5: Função evolve_generation

6.4 Diagrama de Fluxo

A Figura 6.1 apresenta o diagrama de fluxo detalhado do algoritmo.

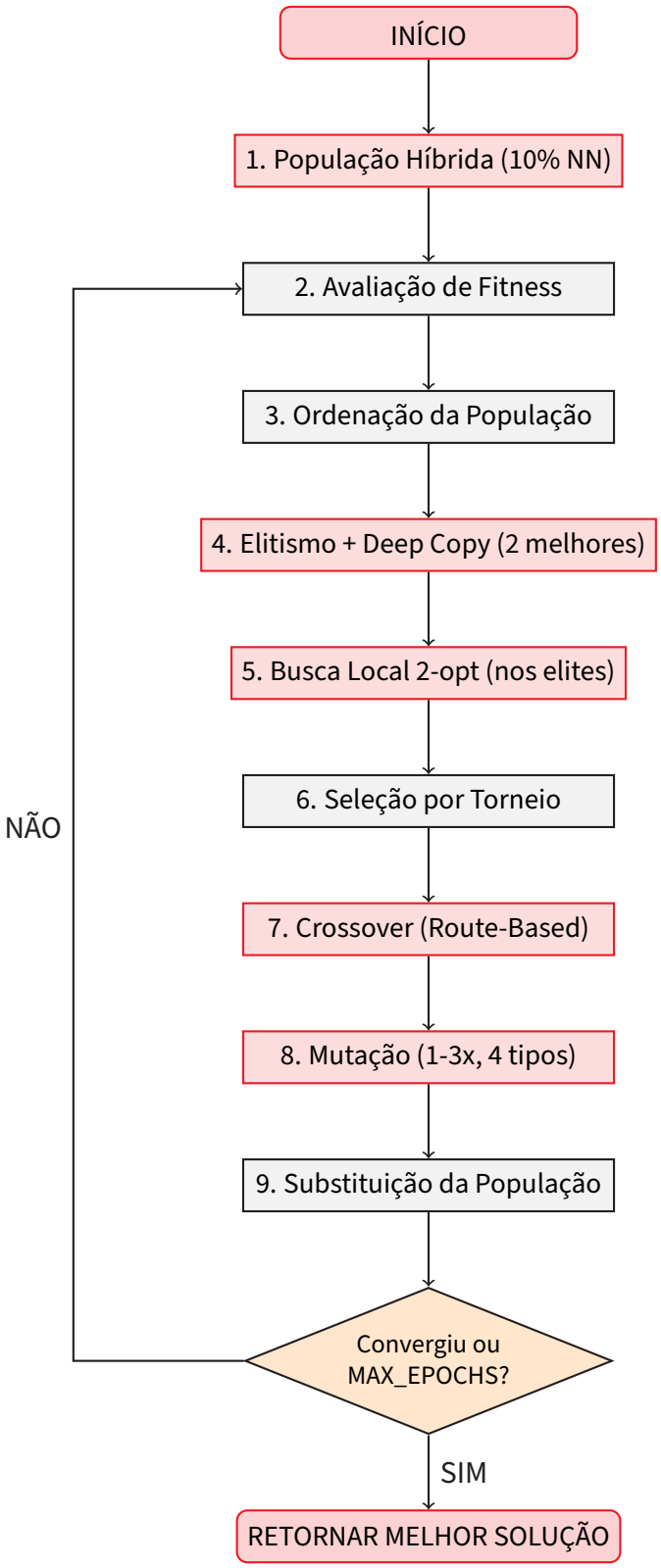

Figura 6.1: Diagrama de fluxo do Algoritmo Genético com as 6 melhorias (destacadas em vermelho claro).

6.5 Execução Completa

```python
# INICIALIZAÇÃO
population = generate_random_population_vrp(deliveries,
    ↪ NUM_VEHICLES, POPULATION_SIZE)

# LOOP PRINCIPAL
for generation in range(MAX_EPOCHS):  # Ex: 1000 gerações
    best_fitness, best_solution = evolve_generation()
    print(f"Geração {generation}: Fitness = {best_fitness}")

# RESULTADO FINAL
print("Melhor solução encontrada:", best_solution)
```

Listing 6.6: Execução completa do algoritmo

6.6 Convergência

A Figura 6.2 ilustra um gráfico típico de convergência.

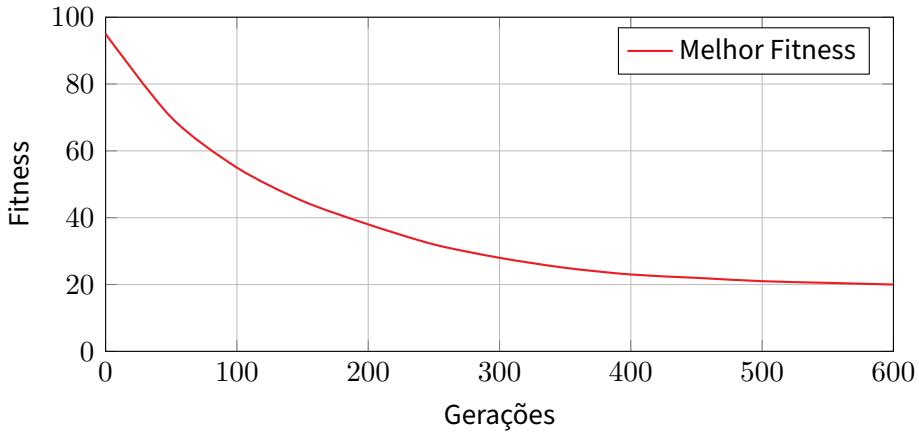

Figura 6.2: Gráfico típico de convergência do Algoritmo Genético.

Fases da convergência:

Início Melhoria rápida (muita "gordura" para cortar)

Meio Melhoria mais lenta (otimizações finas)

Fim Estabilização (próximo do ótimo)

6.7 Convergência Antecipada (Melhoria #6)

Arquivo: `vrp.py:134-165`

Problema de Executar Todas as Gerações

O algoritmo sempre executava todas as 1000 gerações, mesmo quando já tinha convergido na geração 200. Para datasets grandes, isso desperdiça 80% do tempo de execução!

Solução: Detectar **estagnação** — se passar de **20% de max_epochs** (padrão: 200 gerações para max_epochs=1000) sem melhoria no fitness, parar automaticamente:

```
stagnation_counter = 0
previous_best = float('inf')
self.converged = False
self.final_epoch = 0

for epoch in range(self.config.max_epochs):
    # ... avaliação de fitness ...

    current_best = fitness_values[0]

    # Verificar estagnação
    if abs(current_best - previous_best) < 1e-6:
        stagnation_counter += 1
        if stagnation_counter >=
            ↪ self.config.stagnation_threshold:  # 20% de
            ↪ max_epochs
            self.converged = True
            self.final_epoch = epoch
            break  # Para antecipadamente!
    else:
        stagnation_counter = 0
        previous_best = current_best

    self.final_epoch = epoch

    # ... resto do loop ...
```

Listing 6.7: Implementação da convergência antecipada

6.7.1 Atributos Úteis

Após a execução, o solver fornece informações sobre a convergência:

Tabela 6.2: Atributos de convergência do VRPSolver

Atributo	Descrição
`solver.converged`	`True` se parou por estagnação, `False` se executou todas as gerações
`solver.final_epoch`	Número da última geração executada

6.7.2 Exemplo de Economia

Tabela 6.3: Comparação de tempo com e sem convergência antecipada

Cenário	Gerações	Tempo Relativo
Sem parada antecipada	1000	100%
Com parada antecipada	∼200	∼20%

Benefícios da Convergência Antecipada

- **Economia de tempo:** Pode reduzir execução em 50-80%
- **Sem perda de qualidade:** Só para quando não há mais melhoria
- **Configurável:** Parâmetro `stagnation_threshold` (padrão: 20% de max_epochs)

7

Parâmetros e Configuração

Este capítulo descreve os parâmetros do algoritmo genético e a classe VRPSolver que encapsula a implementação.

7.1 Parâmetros do Algoritmo Genético

Arquivo: `config.py:10-61` e `tsp_enhanced.py:24-33`

A Tabela 7.1 apresenta os parâmetros do algoritmo genético.

Tabela 7.1: Parâmetros do Algoritmo Genético

Parâmetro	**Descrição**	**Valor Padrão**
Parâmetros Básicos		
`population_size`	Número de indivíduos na população	200
`mutation_probability`	Probabilidade de mutação por indivíduo	0.6 (60%)
`max_epochs`	Número máximo de gerações (até 10.000)	1000
`tournament_size`	Indivíduos por torneio de seleção	5
`elitism_count`	Indivíduos preservados por elitismo	2
Parâmetros das 6 Melhorias		
`hybrid_initialization`	Usar inicialização híbrida (Melhoria #1)	True
`heuristic_ratio`	Fração da população com Nearest Neighbor	0.1 (10%)
`max_mutations_per_individual`	Máximo de mutações por indivíduo (Melhoria #3)	3
`local_search_rate`	Taxa de aplicação de busca local	0.1
`local_search_elites_only`	Aplicar 2-opt apenas nos elites (Melhoria #4)	True
`stagnation_threshold`	Gerações sem melhoria para parar (Melhoria #6)	20% de max_epochs

7.1.1 Parâmetros do Problema VRP

A Tabela 7.2 apresenta os parâmetros específicos do problema VRP.

Tabela 7.2: Parâmetros do Problema VRP

Parâmetro	Descrição	Valor
`NUM_VEHICLES`	Número de veículos disponíveis	4
`VEHICLE_CAPACITY`	Capacidade máxima de carga	200 kg
`VEHICLE_AUTONOMY`	Distância máxima por viagem	1500 px

7.2 Classe GAConfig

Arquivo: `config.py:10-61`

A classe `GAConfig` encapsula a configuração do algoritmo usando dataclass, incluindo os parâmetros das 6 melhorias:

```python
@dataclass
class GAConfig:
    # Parâmetros básicos
    population_size: int = 200
    mutation_probability: float = 0.6
    max_epochs: int = 1000  # Máximo: 10000
    tournament_size: int = 5
    elitism_count: int = 2

    # Parâmetros de busca local (Melhoria #4)
    local_search_rate: float = 0.1
    local_search_elites_only: bool = True

    # Convergência antecipada (Melhoria #6)
    stagnation_threshold: Optional[int] = None  # None = 20%
        ↪ de max_epochs

    # Múltiplas mutações (Melhoria #3)
    max_mutations_per_individual: int = 3

    # Inicialização híbrida (Melhoria #1)
    hybrid_initialization: bool = True
    heuristic_ratio: float = 0.1

    def __post_init__(self):
```

```
        """Validação dos parâmetros e cálculo de defaults."""
        if self.population_size < 2:
            raise ValueError("population_size must be at
                ↪ least 2")
        if not 0.0 <= self.mutation_probability <= 1.0:
            raise ValueError("mutation_probability must be
                ↪ between 0 and 1")
        if self.max_epochs > 10000:
            raise ValueError("max_epochs must be at most
                ↪ 10000")

        # Calcula stagnation_threshold como 20% de max_epochs
            ↪ se não definido
        if self.stagnation_threshold is None:
            self.stagnation_threshold = max(1,
                ↪ int(self.max_epochs * 0.2))

        if self.stagnation_threshold < 1:
            raise ValueError("stagnation_threshold must be at
                ↪ least 1")
```

Listing 7.1: Classe GAConfig com todas as melhorias

7.2.1 Validações

A classe GAConfig implementa validações para todos os parâmetros:

- `population_size >= 2`
- `0.0 <= mutation_probability <= 1.0`
- `1 <= max_epochs <= 10000`
- `tournament_size >= 2`
- `elitism_count < population_size`
- `stagnation_threshold >= 1` ou `None` para usar 20% de `max_epochs` (Melhoria #6)
- `max_mutations_per_individual >= 1` (Melhoria #3)
- `0.0 <= heuristic_ratio <= 1.0` (Melhoria #1)

Valores Inválidos

A classe lança `ValueError` caso algum parâmetro esteja fora dos limites permitidos.

7.3 Classe VRPSolver

Arquivo: `vrp.py:25-239`

A classe `VRPSolver` encapsula o algoritmo genético para VRP em uma interface orientada a objetos.

7.3.1 Atributos

```
class VRPSolver:
    config: GAConfig                # Configuração do algoritmo
    fitness_history: List[float]    # Histórico de fitness por
        ↪ geração
    best_solution: List[List[int]]  # Melhor solução
        ↪ encontrada
    best_fitness: float             # Fitness da melhor solução
    _distance_matrix: np.ndarray    # Matriz de distâncias
```

Listing 7.2: Atributos da classe VRPSolver

7.3.2 Método solve()

Arquivo: `vrp.py:50-81`

```
def solve(
    self,
    locations: List[Tuple[float, float]],
    num_vehicles: int,
    capacity: float,
    demands: Optional[List[float]] = None,
) -> List[List[int]]:
    """Resolve VRP calculando matriz de distâncias
        ↪ automaticamente"""
    ...
```

Listing 7.3: Método solve da classe VRPSolver

Propósito: Resolve VRP calculando matriz de distâncias automaticamente.

Parâmetros:

- `locations`: Lista de coordenadas (primeira é o depósito)
- `num_vehicles`: Veículos disponíveis
- `capacity`: Capacidade máxima por veículo
- `demands`: Demanda por localização (padrão: 1.0)

Retorno: Lista de rotas (índices de localização)

7.3.3 Método solve_with_distance_matrix()

Arquivo: `vrp.py:83-163`

Propósito: Resolve VRP usando matriz de distâncias pré-computada.

Uso: Quando você já tem distâncias calculadas (ex: distância real de estradas via Google Maps API).

7.3.4 Métodos Auxiliares

Tabela 7.3: Métodos auxiliares da classe VRPSolver

Método	Descrição	Linhas
`get_fitness_history()`	Retorna histórico de fitness para plotagem	165-171
`get_total_distance()`	Calcula distância total de um conjunto de rotas	173-194
`get_route_details()`	Retorna detalhes de cada rota	196-238

7.3.5 Exemplo de Uso

```
# Criar solver com configuração padrão
solver = VRPSolver()

# Definir localizações (primeiro é o depósito)
locations = [(0, 0), (10, 5), (15, 10), (20, 5), (25, 15)]

# Resolver o problema
routes = solver.solve(
    locations=locations,
    num_vehicles=2,
    capacity=100,
    demands=[0, 30, 40, 35, 25]  # Depósito tem demanda 0
```

```
)

# Obter detalhes das rotas
details = solver.get_route_details(routes)
for route in details:
    print(f"Veículo {route['vehicle']}: {route['stops']}
        ↪ paradas, "
          f"distância: {route['distance']:.1f}")

# Obter histórico de convergência
history = solver.get_fitness_history()
```

Listing 7.4: Exemplo de uso da classe VRPSolver

Exemplo de saída do método `get_route_details()`:

```
[
    {
        "vehicle": 1,
        "stops": 5,
        "locations": [3, 7, 2, 9, 1],
        "distance": 245.5
    },
    {
        "vehicle": 2,
        "stops": 3,
        "locations": [4, 6, 8],
        "distance": 180.2
    }
]
```

Listing 7.5: Saída de get_route_details

8

Integração com IA Generativa

Este capítulo descreve a integração do sistema com a API da OpenAI para adicionar capacidades de Large Language Model (LLM) ao otimizador de rotas.

8.1 Visão Geral

O projeto utiliza a **API da OpenAI** para adicionar capacidades de IA Generativa ao sistema. A integração é implementada através de um módulo dedicado (`llm/`) que encapsula todas as interações com o modelo GPT.

Tabela 8.1: Configuração da integração com OpenAI

Parâmetro	Valor
Biblioteca	`openai >= 1.0.0`
Modelo	`gpt-4o-mini`
Método de API	`chat.completions.create()`
Variável de ambiente	`OPENAI_API_KEY`

8.2 Arquitetura da Integração

A Figura 8.1 apresenta a arquitetura da integração com IA Generativa.

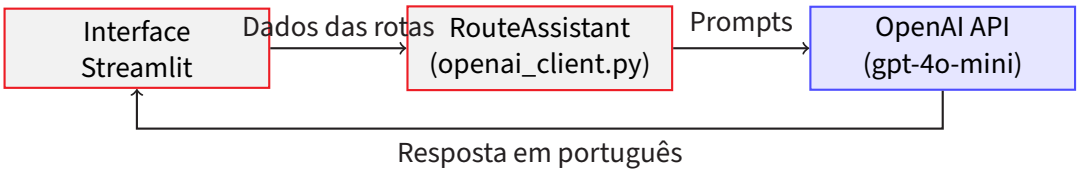

Figura 8.1: Arquitetura da integração com IA Generativa.

8.2.1 Estrutura de Arquivos

```
src/routing_optimizer/
├── llm/
│   ├── __init__.py ..................................Exporta RouteAssistant
│   └── openai_client.py ..................Implementação completa (281 linhas)
└── app/pages/
    ├── 1_upload.py .........................Usa LLM para correção de endereços
    └── 4_instructions.py ..............................Interface principal LLM
```

8.3 Classe RouteAssistant

Arquivo: `src/routing_optimizer/llm/openai_client.py`

A classe `RouteAssistant` encapsula todas as interações com a API da OpenAI.

```python
class RouteAssistant:
    DEFAULT_MODEL = "gpt-4o-mini"

    def __init__(
        self,
        api_key: Optional[str] = None,
        model: Optional[str] = None,
    ):
        self.api_key = api_key or os.getenv("OPENAI_API_KEY")
        self._client: Optional[OpenAI] = None
        self.model = model or self.DEFAULT_MODEL

    @property
    def client(self) -> OpenAI:
        """Lazy initialization of OpenAI client."""
        if self._client is None:
            self._client = OpenAI(api_key=self.api_key)
        return self._client

    def is_configured(self) -> bool:
        """Check if API key is set."""
        return bool(self.api_key)
```

Listing 8.1: Classe RouteAssistant

8.3.1 Métodos Disponíveis

A Tabela 8.2 apresenta os métodos da classe RouteAssistant.

Tabela 8.2: Métodos da classe RouteAssistant

Método	Propósito	max_tokens	temperature
`generate_driver_instructions()`	Gera instruções de navegação	1000	0.7
`generate_efficiency_report()`	Gera relatório de eficiência	1500	0.7
`chat_about_routes()`	Responde perguntas sobre rotas	500	0.7
`suggest_address_corrections()`	Sugere correções de endereços	300	0.3
`is_configured()`	Verifica configuração da API	–	–

8.4 Funcionalidades Implementadas

8.4.1 Geração de Instruções para Motoristas

Arquivo: `openai_client.py:52-99`

Esta funcionalidade gera instruções detalhadas de navegação para motoristas de entrega de medicamentos.

```
system_prompt = """Voce e um assistente de logistica especializado em entregas
de medicamentos. Gere instrucoes claras e objetivas para motoristas. Seja
conciso mas inclua dicas uteis sobre o trajeto. Responda sempre em portugues
brasileiro."""
```

Listing 8.2: System prompt para instruções de motorista

O user prompt solicita:

- Resumo da rota (origem, destino, número de paradas)
- Tempo estimado total considerando trânsito médio
- Dicas de trânsito para horários de pico em São Paulo
- Observações para entrega de medicamentos
- Sugestão de horário ideal para iniciar

8.4.2 Relatório de Eficiência

Arquivo: `openai_client.py:101-160`

Gera relatórios profissionais sobre a eficiência da otimização de rotas.

```
system_prompt = """Voce e um analista de logistica especializado em otimizacao
de rotas. Gere relatorios profissionais e detalhados sobre eficiencia de
entregas. Use dados concretos e metricas relevantes. Responda sempre em
portugues brasileiro."""
```

Listing 8.3: System prompt para relatório de eficiência

O relatório inclui:

1. Resumo Executivo
2. Métricas de Eficiência (distância média por veículo, paradas por veículo)
3. Análise de Balanceamento de Carga
4. Recomendações de Melhoria
5. Comparação com Benchmarks do Setor

8.4.3 Chat Interativo sobre Rotas

Arquivo: `openai_client.py:162-200`

Permite ao usuário fazer perguntas interativas sobre as rotas otimizadas.

```
system_prompt = f"""Voce e um assistente especializado em rotas de entrega de
medicamentos em Sao Paulo. Responda perguntas de forma util, precisa e em
portugues.

Contexto das rotas atuais:
{routes_context}

Se a pergunta for sobre algo que nao esta no contexto, responda com base no
seu conhecimento sobre logistica e distribuicao farmaceutica."""
```

Listing 8.4: System prompt para chat sobre rotas

Contexto incluído automaticamente:

- Número de veículos utilizados
- Total de paradas
- Distância total
- Tempo de otimização
- Detalhes por veículo (paradas e distância)

8.4.4 Correção de Endereços

Arquivo: `openai_client.py:210-281`

Sugere correções para endereços que falharam na geocodificação.

```
system_prompt = """Voce e um assistente especializado em
    ↪ enderecos brasileiros.
Sua tarefa e corrigir enderecos que nao foram encontrados
    ↪ pelo geocodificador.

Regras:
1. Analise o endereco e identifique possiveis erros de
    ↪ digitacao ou abreviacoes
2. Retorne APENAS o endereco corrigido no formato:
    "Nome do Logradouro, Numero, Bairro, Cidade"
3. NAO inclua CEP, estado, pais ou outras informacoes
4. Se nao tiver certeza, sugira as 2-3 opcoes mais provaveis
5. Se o endereco parecer completamente invalido, retorne
    ↪ "IGNORAR"
"""
```

Listing 8.5: System prompt para correção de endereços

Temperature baixa para consistência

Este método usa `temperature=0.3` (mais baixa que os outros) para obter resultados mais consistentes e previsíveis na correção de endereços.

8.5 Interface de Usuário

8.5.1 Página de Instruções (4_instructions.py)

Arquivo: `src/routing_optimizer/app/pages/4_instructions.py`

Esta página oferece três funcionalidades LLM:

Seção 1 – Instruções para Motoristas: Seleção de veículo, visualização das paradas, botão para gerar instruções e download em TXT.

Seção 2 – Relatório de Eficiência: Exibe métricas da otimização e gera relatório profissional com download.

Seção 3 – Chat sobre Rotas: Input de texto para perguntas, histórico das últimas 5 perguntas e respostas.

8.5.2 Página de Upload (1_upload.py)

Arquivo: `src/routing_optimizer/app/pages/1_upload.py:154-219`

Usa LLM para correção de endereços com falha de geocodificação:

1. Detecta endereços que falharam na geocodificação
2. Oferece opção "Usar Assistente para corrigir"
3. Para cada endereço com erro:
    - Consulta o LLM para sugestões de correção
    - Exibe as opções em radio buttons
    - Permite escolher uma sugestão ou ignorar
4. Re-geocodifica os endereços corrigidos

8.6 Configuração

8.6.1 Variável de Ambiente

A API key deve ser configurada via variável de ambiente:

```
export OPENAI_API_KEY="sk-your-key-here"
```

Ou através do arquivo `.env` na raiz do projeto:

```
# OpenAI API Key (obrigatório para integração LLM)
OPENAI_API_KEY=sk-your-key-here
```

8.6.2 Dependência Opcional

A integração com OpenAI é uma dependência opcional do projeto:

```
# pyproject.toml
[project.optional-dependencies]
llm = [
    "openai>=1.0.0",
]
```

Para instalar:

```
pip install -e ".[llm]"
```

8.6.3 Verificação de Configuração

As páginas Streamlit verificam se a API está configurada antes de usar:

```
assistant = RouteAssistant()
if not assistant.is_configured():
    st.error("OPENAI_API_KEY nao configurada!")
    st.stop()
```

Listing 8.6: Verificação de configuração da API

8.7 Parâmetros de Geração

A Tabela 8.3 resume os parâmetros utilizados em cada tipo de geração.

Tabela 8.3: Parâmetros de geração por funcionalidade

Funcionalidade	Tipo de Saída	Tokens	Temp.	Criatividade
Instruções	Guia de navegação detalhado	1000	0.7	Moderada
Relatório	Análise profissional	1500	0.7	Moderada
Chat	Respostas contextualizadas	500	0.7	Moderada
Correção	Sugestões de endereço	300	0.3	Baixa

Custo de API

A integração com OpenAI tem custo por token. O modelo `gpt-4o-mini` foi escolhido por oferecer boa qualidade com custo reduzido.

9

BÔNUS: Deploy em Nuvem

Este capítulo apresenta a infraestrutura como código (IaC) desenvolvida para deploy da aplicação em ambiente de nuvem AWS, utilizando Terraform para provisionamento automatizado.

9.1 Visão Geral da Infraestrutura

A aplicação Routing Optimizer foi projetada para execução tanto local quanto em nuvem. Para o ambiente de produção, foi desenvolvida uma infraestrutura na AWS (Amazon Web Services) que provisiona automaticamente todos os recursos necessários.

Tabela 9.1: Componentes da infraestrutura AWS

Componente	Descrição
EC2 (t3.small)	Instância de computação com Amazon Linux 2023
Elastic IP	Endereço IP público fixo para acesso à aplicação
Security Group	Firewall virtual com regras para HTTP, HTTPS e SSH
IAM Role	Permissões para EC2 acessar o Secrets Manager
Secrets Manager	Cofre seguro para armazenamento da API Key OpenAI

Custo Estimado

O custo mensal estimado da infraestrutura é de aproximadamente **US$ 16-20/mês**, incluindo EC2 t3.small (24/7), Elastic IP e Secrets Manager.

9.2 Arquitetura

A Figura 9.1 apresenta a arquitetura da solução em nuvem.

O fluxo de funcionamento é:

1. O usuário acessa a aplicação via navegador (HTTP na porta 8501)

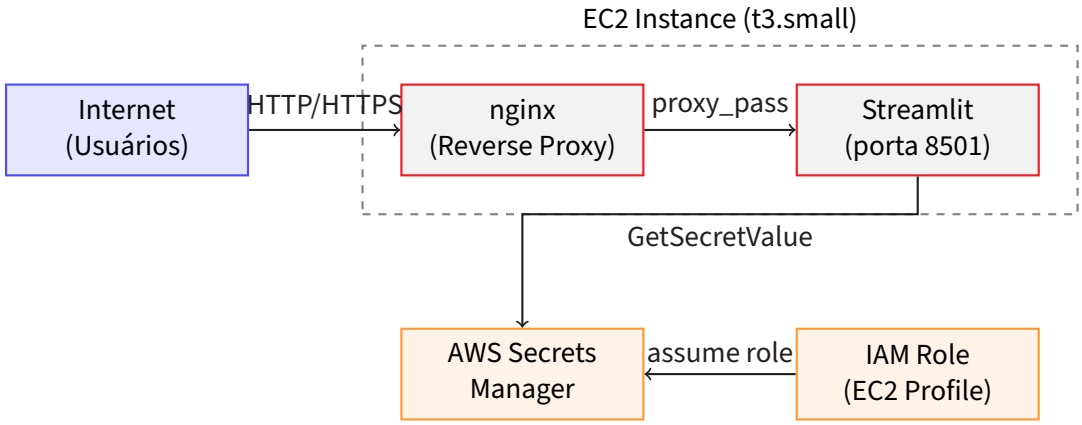

Figura 9.1: Arquitetura da infraestrutura AWS para o Routing Optimizer.

2. O nginx atua como reverse proxy, encaminhando requisições para o Streamlit
3. A aplicação Streamlit busca a API Key do OpenAI no AWS Secrets Manager
4. A IAM Role associada à EC2 autoriza o acesso ao segredo

9.3 Estrutura de Arquivos Terraform

O código Terraform está organizado no diretório `infra/` do projeto:

```
infra/
├── main.tf ..... Recursos principais (EC2, VPC, Key Pair, EIP)
├── variables.tf ..... Variáveis de entrada configuráveis
├── outputs.tf ..... Saídas (IP público, URL, comando SSH)
├── secrets.tf ..... AWS Secrets Manager para API Key
├── iam.tf ..... IAM Role e Policy para EC2
├── security.tf ..... Security Group (firewall)
├── userdata.sh ..... Script de inicialização da EC2
├── terraform.tfvars.example ..... Exemplo de configuração
└── .gitignore ..... Ignora arquivos sensíveis
```

Segurança

O arquivo `terraform.tfvars` contém a API Key da OpenAI e **nunca** deve ser commitado no repositório. Ele está listado no `.gitignore`.

9.4 Provisionamento Automatizado

Quando o Terraform cria a instância EC2, o script `userdata.sh` é executado automaticamente para configurar todo o ambiente. A Tabela 9.2 descreve as etapas de provisionamento.

Tabela 9.2: Etapas de provisionamento automatizado (userdata.sh)

#	Etapa	Descrição
1	Atualização do sistema	`dnf update -y`
2	Instalação de dependências	Python 3.11, pip, git, nginx
3	Criação de usuário	Usuário `streamlit` sem privilégios
4	Clone do repositório	`git clone` do GitHub
5	Ambiente virtual Python	`python3.11 -m venv venv`
6	Instalação do projeto	`pip install -e ".[all]"` e boto3
7	Configuração Streamlit	`config.toml` para modo headless
8	Variáveis de ambiente	`AWS_REGION` e `SECRET_NAME`
9	Serviço systemd	Configuração para auto-restart
10	Configuração nginx	Reverse proxy para Streamlit

9.4.1 Serviço systemd

A aplicação é gerenciada como um serviço do sistema operacional:

```
[Unit]
Description=Streamlit Routing Optimizer
After=network-online.target

[Service]
Type=simple
User=streamlit
WorkingDirectory=/opt/app
EnvironmentFile=/opt/app/.env.aws
ExecStart=/opt/app/venv/bin/streamlit run \
    src/routing_optimizer/app/main.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Listing 9.1: Configuração do serviço systemd

Isso garante que a aplicação:

- Inicie automaticamente após o boot
- Reinicie automaticamente em caso de falha
- Execute com usuário sem privilégios (segurança)

9.5 Gerenciamento de Segredos

A API Key da OpenAI é armazenada de forma segura no AWS Secrets Manager, um serviço de cofre gerenciado pela AWS.

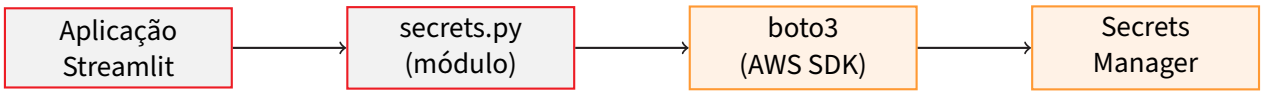

Figura 9.2: Fluxo de obtenção da API Key do Secrets Manager.

9.5.1 Módulo secrets.py

O módulo `src/routing_optimizer/utils/secrets.py` implementa a integração com AWS Secrets Manager:

```python
def get_secret_from_aws(
    secret_name: str,
    region_name: Optional[str] = None
) -> Optional[str]:
    """Retrieve a secret from AWS Secrets Manager."""
    if not BOTO3_AVAILABLE:
        return None

    region = region_name or os.getenv("AWS_REGION",
        ↪ "us-east-1")

    try:
        client = boto3.client("secretsmanager",
            ↪ region_name=region)
        response =
            ↪ client.get_secret_value(SecretId=secret_name)
        return response.get("SecretString")
    except ClientError:
        return None
```

Listing 9.2: Função para obter segredo do AWS Secrets Manager

9.5.2 Prioridade de Fontes

A aplicação busca a API Key em múltiplas fontes, com a seguinte prioridade:

1. **Session State** – Chave fornecida pelo usuário na interface

2. **AWS Secrets Manager** – Chave configurada no cofre (produção)
3. **Variável de ambiente** – `OPENAI_API_KEY` (desenvolvimento)

Isso permite que usuários utilizem suas próprias chaves, enquanto mantém uma chave padrão configurada no servidor.

9.6 Como Executar o Terraform

9.6.1 Pré-requisitos

1. **AWS CLI** instalado e configurado com credenciais
2. **Terraform** versão 1.5 ou superior
3. **Chave SSH** em `~/.ssh/id_rsa.pub`
4. **API Key** da OpenAI

9.6.2 Configuração de Credenciais AWS

```
# Configurar credenciais
aws configure
# AWS Access Key ID: [sua-access-key]
# AWS Secret Access Key: [sua-secret-key]
# Default region name: us-east-1
# Default output format: json

# Verificar configuração
aws sts get-caller-identity
```

Listing 9.3: Configuração das credenciais AWS

9.6.3 Deploy da Infraestrutura

```
# Navegar para o diretório de infraestrutura
cd infra

# Copiar e editar arquivo de variáveis
cp terraform.tfvars.example terraform.tfvars
# Editar terraform.tfvars com sua OPENAI_API_KEY e IP

# Inicializar Terraform (download de providers)
```

```
terraform init

# Visualizar plano de execução
terraform plan

# Aplicar infraestrutura (criar recursos)
terraform apply

# Ver outputs (IP, URL, comando SSH)
terraform output
```

Listing 9.4: Comandos para deploy com Terraform

9.6.4 Arquivo de Variáveis

Exemplo de configuração do arquivo `terraform.tfvars`:

```
aws_region        = "us-east-1"
project_name      = "routing-optimizer"
instance_type     = "t3.small"
openai_api_key    = "sk-your-key-here"
domain_name       = ""
ssh_allowed_cidr  = "SEU_IP/32"
github_repo       =
    ↪ "https://github.com/Zagari/routing-optimizer-ai.git"
```

Listing 9.5: Exemplo de terraform.tfvars

Restrição de SSH

O parâmetro `ssh_allowed_cidr` deve ser configurado com seu IP público seguido de `/32` para restringir o acesso SSH apenas à sua máquina. Nunca use `0.0.0.0/0` em produção.

9.6.5 Destruição da Infraestrutura

Para remover todos os recursos criados e evitar custos:

```
cd infra
terraform destroy
```

Listing 9.6: Comando para destruir infraestrutura

9.7 Demonstração Online

Uma instância da aplicação está disponível publicamente para demonstração:

`http://100.30.130.165:8501`

Disponibilidade

O servidor de demonstração pode estar temporariamente indisponível fora do período de avaliação do Tech Challenge, devido aos custos de manutenção da infraestrutura AWS.

10

FAQ e Glossário

Este capítulo apresenta perguntas frequentes sobre o algoritmo e um glossário de termos técnicos.

10.1 Perguntas Frequentes (FAQ)

10.1.1 Sobre os Parâmetros

P1: Por que a probabilidade de mutação é tão alta (60%)?

Para VRP, 60% é alto mas necessário porque:

- A mutação é "suave" (pequenas mudanças)
- Existem 4 tipos diferentes (diversidade)
- Ajuda a escapar de ótimos locais

Para TSP clássico, valores de 5-20% são mais comuns.

P2: O que acontece se eu aumentar o tamanho da população?

Vantagem: Mais diversidade, melhor exploração.
Desvantagem: Mais lento (mais fitness para calcular).
Recomendação: 100-500 para a maioria dos problemas.

P3: Quantas gerações são necessárias?

Depende do problema:

- Pequeno (10-20 cidades): 100-500 gerações
- Médio (20-50 cidades): 500-2000 gerações
- Grande (50+ cidades): 1000-10000+ gerações

Use o gráfico de convergência para ver quando estabiliza.

P4: O que é o tournament_size e como afeta o resultado?

- **Maior (ex: 10):** Mais "pressão seletiva" — converge mais rápido, mas pode ficar preso em ótimos locais
- **Menor (ex: 2):** Menos pressão — mais diversidade, converge mais devagar
- **Valor típico:** 3-5 (bom equilíbrio)

10.1.2 Sobre o Algoritmo

P5: O algoritmo sempre encontra a solução ótima?

Não. Algoritmos genéticos são **heurísticas** — encontram soluções **boas**, mas não garantem a **ótima**. Para problemas grandes, a solução ótima é praticamente impossível de encontrar de qualquer forma.

P6: Como funciona a penalidade de prioridade?

Entregas críticas devem ser feitas **primeiro**. A penalidade aumenta com a **posição** na rota:

```
Rota: [Normal, Urgente, CRÍTICO, Normal]
                         ^ posição 2

Penalidade = posição * peso_prioridade
           = 2 * 50 = 100

Se o crítico estivesse primeiro:
Rota: [CRÍTICO, Normal, Urgente, Normal]
       ^ posição 0
Penalidade = 0 * 50 = 0  (nenhuma!)
```

P7: Por que usar seed fixa para entregas?

No arquivo `tsp_enhanced.py:121`:

```
1 rng = random.Random(42)  # Seed fixa
```

Isso garante **reprodutibilidade**: toda vez que executar, as mesmas entregas serão geradas, permitindo comparar resultados de forma justa.

P8: Qual a diferença entre OX/PMX e Route-Based Crossover?

- **OX/PMX:** Crossovers tradicionais que "achatam" rotas e redistribuem — **destroem clusters geográficos**

- **Route-Based (usado neste projeto): Preserva rotas inteiras** dos pais, mantendo agrupamentos geográficos

O Route-Based Crossover é uma das 6 melhorias implementadas (Melhoria #2).

P9: O que é convergência antecipada?

O algoritmo **para automaticamente** após **20% de max_epochs** (por padrão, 200 gerações para max_epochs=1000) sem melhoria no fitness. Isso pode economizar **50-80%** do tempo de execução quando o problema converge rapidamente.

- `solver.converged` — `True` se parou por estagnação
- `solver.final_epoch` — Número da última geração executada

P10: Por que usar inicialização híbrida?

A inicialização híbrida usa **10% da população** com a heurística Nearest Neighbor (vizinho mais próximo) e **90% aleatória**. Isso faz o algoritmo começar com soluções melhores, convergindo mais rápido, enquanto mantém diversidade genética.

10.2 Glossário

A Tabela 10.1 apresenta os termos técnicos utilizados neste documento.

Tabela 10.1: Glossário de termos técnicos

Termo	Definição
AG / GA	Algoritmo Genético / Genetic Algorithm
TSP	Traveling Salesman Problem (Problema do Caixeiro Viajante)
VRP	Vehicle Routing Problem (Problema de Roteamento de Veículos)
Fitness	Medida de qualidade de uma solução (menor = melhor)
Crossover	Operador que combina dois pais para criar filhos
Route-Based Crossover	Crossover que preserva rotas inteiras (Melhoria #2)
Mutação	Operador que introduz variação aleatória
Elitismo	Estratégia de preservar os melhores indivíduos
Deep Copy	Cópia completa que não compartilha referências (Melhoria #5)
Torneio	Método de seleção por competição
2-opt	Movimento de otimização local que inverte segmentos de rota
Busca Local	Refinamento de solução por pequenas modificações (Melhoria #4)
Nearest Neighbor	Heurística que sempre visita o vizinho mais próximo
Inicialização Híbrida	População inicial com parte heurística e parte aleatória (Melhoria #1)
Early Stopping	Parada antecipada quando não há mais melhoria (Melhoria #6)
Estagnação	Período sem melhoria no fitness
Depósito	Ponto de partida e chegada dos veículos
Heurística	Método que encontra soluções aproximadas em tempo razoável
Metaheurística	Estratégia de alto nível para guiar heurísticas
Ótimo local	Solução que é melhor que seus vizinhos, mas não a melhor global
Convergência	Processo de estabilização do algoritmo em uma solução

10.3 Conclusão

O Algoritmo Genético para TSP/VRP implementado inclui **6 melhorias** em relação ao algoritmo tradicional:

1. **Gera população híbrida** (10% Nearest Neighbor + 90% aleatória) — Melhoria #1
2. **Avalia** cada rota (fitness = distância + penalidades)
3. **Seleciona** os melhores por torneio
4. **Combina** pais via **Route-Based Crossover** (preserva rotas) — Melhoria #2
5. **Muta** filhos com **1-3 mutações** de 4 tipos — Melhoria #3
6. **Preserva** os 2 melhores com **deep copy** — Melhoria #5
7. **Aplica 2-opt** nos elites para refinamento local — Melhoria #4
8. **Para automaticamente** se não melhorar por 20% de max_epochs — Melhoria #6

Tabela 10.2: Resumo das 6 melhorias implementadas

#	Melhoria	Impacto	Parâmetro
1	Inicialização Híbrida	Convergência mais rápida	`heuristic_ratio`
2	Route-Based Crossover	Qualidade das soluções	—
3	Múltiplas Mutações	Exploração do espaço	`max_mutations`
4	Busca Local 2-opt	Refinamento das rotas	`local_search_*`
5	Deep Copy Elites	Estabilidade do código	`elitism_count`
6	Convergência Antecipada	Economia de tempo (50-80%)	`stagnation_threshold`

Resultado Final

O resultado é uma **boa solução** (não necessariamente ótima) encontrada em **tempo razoável**, adequada para aplicações práticas de logística e otimização de rotas. As 6 melhorias implementadas aumentam a **qualidade** das soluções e reduzem o **tempo de execução**.

A

Referências de Código

Este apêndice apresenta um índice completo das funções implementadas e suas localizações nos arquivos fonte.

A.1 Arquivos Principais

A Tabela A.1 lista os arquivos principais do projeto e suas descrições.

Tabela A.1: Arquivos principais do projeto

Arquivo	Descrição	Linhas
`genetic_algorithm_enhanced.py`	Implementação completa do GA	1-286
`core.py`	Funções core do GA (versão refatorada)	1-326
`config.py`	Configuração do algoritmo	1-61
`vrp.py`	Classe VRPSolver	1-239
`tsp_enhanced.py`	Sistema de entregas hospitalares	1-547
`draw_functions_enhanced.py`	Visualização Pygame	1-157
`benchmark_att48.py`	Dataset de benchmark	1-107
Integração LLM (OpenAI)		
`llm/openai_client.py`	Cliente OpenAI para instruções e relatórios	1-281
`llm/__init__.py`	Exportações do módulo LLM	1-8
Páginas da Aplicação Web (Streamlit)		
`app/pages/0_home.py`	Página inicial com visão geral	1-43
`app/pages/1_upload.py`	Upload, geocodificação e correção de endereços	1-668
`app/pages/2_optimize.py`	Configuração e execução do AG	1-285
`app/pages/3_results.py`	Visualização de rotas no mapa	1-248
`app/pages/4_instructions.py`	Geração de instruções com LLM	1-252
`app/pages/5_experiments.py`	Experimentos comparativos	1-374

A.2 Funções por Categoria

A.2.1 Funções de Distância

Tabela A.2: Funções de cálculo de distância

Função	Localização
`calculate_distance()`	`genetic_algorithm_enhanced.py:6-8`, `core.py:18-28`
`calculate_route_distance()`	`genetic_algorithm_enhanced.py:10-19`, `core.py:31-60`

A.2.2 Funções de Fitness

Tabela A.3: Funções de cálculo de fitness

Função	Localização
`calculate_fitness()`	`genetic_algorithm_enhanced.py:21-23`
`calculate_fitness_vrp()`	`genetic_algorithm_enhanced.py:25-96`, `core.py:81-128`

A.2.3 Funções de População

Tabela A.4: Funções de geração de população

Função	Localização
`generate_random_population()`	`genetic_algorithm_enhanced.py:98-106`, `core.py:156-171`
`generate_random_population_vrp()`	`genetic_algorithm_enhanced.py:108-137`
`nearest_neighbor_solution()`	`core.py:424-450`
`generate_hybrid_population()`	`core.py:452-485`

Melhoria #1: Inicialização Híbrida

As funções `nearest_neighbor_solution()` e `generate_hybrid_population()` implementam a inicialização híbrida, onde 10% da população é gerada usando a heurística do vizinho mais próximo.

A.2.4 Funções de Seleção

Tabela A.5: Funções de seleção

Função	Localização
`tournament_selection()`	`genetic_algorithm_enhanced.py:276-283`, `core.py:174-192`

A.2.5 Funções de Crossover

Tabela A.6: Funções de crossover

Função	Localização
`order_crossover()`	`genetic_algorithm_enhanced.py:139-155`
`pmx_crossover()`	`genetic_algorithm_enhanced.py:157-176`
`vrp_crossover()` (Route-Based)	`core.py:195-258`

Melhoria #2: Route-Based Crossover

A função `vrp_crossover()` implementa o Route-Based Crossover que preserva rotas inteiras dos pais, diferente dos crossovers tradicionais (OX/PMX) que destroem clusters geográficos.

A.2.6 Funções de Mutação

Tabela A.7: Funções de mutação

Função	Localização
`mutate()`	`genetic_algorithm_enhanced.py:210-215`
`mutate_vrp()`	`genetic_algorithm_enhanced.py:217-267`, `core.py:261-343`
`_apply_single_mutation()`	`core.py:267-306`

Melhoria #3: Múltiplas Mutações

A função `mutate_vrp()` foi aprimorada para aplicar 1 a 3 mutações por indivíduo (parâmetro `max_mutations`). A função interna `_apply_single_mutation()` executa uma das 4 operações: swap_within, swap_between, reverse ou relocate.

A.2.7 Funções de Busca Local

Tabela A.8: Funções de busca local (Melhoria #4)

Função	Localização
`two_opt()`	`core.py:346-391`
`apply_local_search()`	`core.py:393-422`

Melhoria #4: Busca Local 2-opt

A função `two_opt()` implementa a otimização local que inverte segmentos da rota até não haver mais melhoria. A função `apply_local_search()` aplica o 2-opt em todas as rotas de um indivíduo, sendo usada apenas nos elites para eficiência.

A.2.8 Funções de Ordenação

Tabela A.9: Funções de ordenação

Função	Localização
`sort_population()`	`genetic_algorithm_enhanced.py:269-274`, `core.py:309-325`

A.3 Funções de Integração LLM (OpenAI)

A.3.1 Classe RouteAssistant

A Tabela A.10 apresenta os métodos da classe `RouteAssistant`, responsável pela integração com a API da OpenAI para geração de conteúdo relacionado às rotas.

Tabela A.10: Métodos da classe RouteAssistant

Método	Descrição	Linhas
`__init__()`	Inicializa o assistente com API key e modelo	30-43
`client` (property)	Inicialização lazy do cliente OpenAI	45-50
`generate_driver_instructions()`	Gera instruções detalhadas para motoristas	52-99
`generate_efficiency_report()`	Gera relatório de eficiência da otimização	101-160
`chat_about_routes()`	Responde perguntas sobre as rotas	162-200
`is_configured()`	Verifica se a API key está configurada	202-208
`suggest_address_corrections()`	Sugere correções para endereços com erro	210-280

A.3.2 Funções de Geração de Conteúdo

Tabela A.11: Funções de geração de conteúdo com LLM

Função	Localização
`generate_driver_instructions()`	`llm/openai_client.py:52-99`
`generate_efficiency_report()`	`llm/openai_client.py:101-160`
`chat_about_routes()`	`llm/openai_client.py:162-200`
`suggest_address_corrections()`	`llm/openai_client.py:210-280`

A.4 Funções Auxiliares da Interface Web

A.4.1 Funções de Upload e Geocodificação

Tabela A.12: Funções auxiliares da página de upload

Função	Descrição	Linhas
`go_to_tab()`	Navega para uma aba específica	20-23
`clear_temp_upload_state()`	Limpa estado temporário de upload	26-39
`_save_and_continue()`	Salva dataset e continua para mapa	42-94
`_regeocode_corrected()`	Re-geocodifica endereços corrigidos	97-151
`_render_correction_interface()`	Renderiza interface de correção com LLM	154-219

A.4.2 Integração LLM na Interface

Tabela A.13: Pontos de integração LLM nas páginas

Página	Uso da LLM
`1_upload.py`	Correção de endereços via `suggest_address_corrections()`
`4_instructions.py`	Geração de instruções via `generate_driver_instructions()`
`4_instructions.py`	Relatório de eficiência via `generate_efficiency_report()`
`4_instructions.py`	Chat sobre rotas via `chat_about_routes()`

A.5 Estrutura de Diretórios

```
genetic_algorithm_tsp_enhanced/
├── genetic_algorithm_enhanced.py ..... Implementação enhanced do GA
├── tsp_enhanced.py ..... Sistema principal de entregas
├── draw_functions_enhanced.py ..... Funções de visualização
├── benchmark_att48.py ..... Dataset benchmark com 48 cidades
└── backup/
    └── genetic_algorithm.py ..... Versão original básica do GA
src/routing_optimizer/
├── genetic_algorithm/
│   ├── config.py ..... Configuração do GA (GAConfig)
│   ├── core.py ..... Funções core do GA
│   └── vrp.py ..... Classe VRPSolver
├── llm/ ..... Integração com LLM (OpenAI)
│   ├── __init__.py ..... Exportações do módulo
│   └── openai_client.py ..... Classe RouteAssistant
└── app/ ..... Aplicação Web Streamlit
    └── pages/
        ├── 0_home.py ..... Página inicial
        ├── 1_upload.py ..... Upload e geocodificação
        ├── 2_optimize.py ..... Configuração e execução do AG
        ├── 3_results.py ..... Visualização de rotas
        ├── 4_instructions.py ..... Instruções com LLM
        └── 5_experiments.py ..... Experimentos comparativos
```

A.6 Índice de Listagens

As listagens de código neste documento estão organizadas conforme a Tabela A.14.

Tabela A.14: Índice de listagens de código

Capítulo	Listagem	Descrição
3	3.1	Função calculate_distance
3	3.3	Função calculate_route_distance
3	3.5	Geração de população TSP
3	3.6	Geração de população VRP
3	3.7	Heurística Nearest Neighbor (Melhoria #1)
3	3.8	População Híbrida (Melhoria #1)
4	4.1	Fitness para TSP
4	4.2	Fitness para VRP
4	4.3	Seleção por torneio
5	5.1	Order Crossover (OX)
5	5.2	PMX Crossover
5	5.3	Route-Based Crossover (Melhoria #2)
5	5.4	Mutação para TSP
5	5.5	Múltiplas Mutações (Melhoria #3)
6	6.1	Elitismo com Deep Copy (Melhoria #5)
6	6.2	Função sort_population
6	6.3	Busca Local 2-opt (Melhoria #4)
6	6.4	Aplicação de Busca Local
6	6.5	Função evolve_generation
6	6.6	Execução completa
6	6.7	Convergência Antecipada (Melhoria #6)
7	7.1	Classe GAConfig (todas as melhorias)
7	7.3	Método solve do VRPSolver

A.7 Resumo das 6 Melhorias Implementadas

A Tabela A.15 apresenta um resumo das 6 melhorias implementadas no algoritmo genético e suas localizações no código.

Tabela A.15: Resumo das 6 melhorias e localizações no código

#	Melhoria	Função Principal	Localização
1	Inicialização Híbrida	`generate_hybrid_population()`	`core.py:452-485`
2	Route-Based Crossover	`vrp_crossover()`	`core.py:195-258`
3	Múltiplas Mutações	`mutate_vrp()`	`core.py:261-343`
4	Busca Local 2-opt	`two_opt()`, `apply_local_search()`	`core.py:346-422`
5	Deep Copy Elites	Elitismo no loop principal	`vrp.py:166-174`
6	Convergência Antecipada	Detecção de estagnação	`vrp.py:134-165`

Parâmetros de Configuração

Todas as melhorias são configuráveis através da classe `GAConfig` em `config.py:10-61`. Os principais parâmetros são:

- `hybrid_initialization`: Ativa inicialização híbrida (Melhoria #1)
- `heuristic_ratio`: Fração da população com Nearest Neighbor (padrão: 0.1)
- `max_mutations_per_individual`: Máximo de mutações por indivíduo (padrão: 3)
- `local_search_elites_only`: Aplica 2-opt apenas nos elites (padrão: True)
- `stagnation_threshold`: Gerações sem melhoria para parar (padrão: 20% de max_epochs)