Algoritmos Genéticos: Otimização de Rotas para o Problema do Caixeiro-Viajante com Visualização Aprimorada.

Igor Mariano Alencar e Silva, Jhony Wictor do Nascimento Santos, Karleandro Santos da Silva, Lucas Rosendo de Farias

Universidade Federal de Alagoas - UFAL, Campus Arapiraca — SEDE.
Bacharelado em Ciência da Computação.

{igor.alencar,jhony.santos,karleandro.silva,lucas.farias}@arapiraca.ufal.br

ABSTRATCT. *This work presents the development of a genetic algorithm for route optimization, applied to the Traveling Salesman Problem, with enhanced visualization capabilities. The solution implements selection, crossover, and mutation operators, combined with elitism, to maximize the performance of generated routes. The graphical interface allows real-time monitoring of the solutions' evolution, displaying performance metrics and intermediate routes. Requested by Professor Rômulo Nunes de Oliveira, in the subject of Artificial Intelligence, in the Bachelor's degree in Computer Science, class CPTA126-01 2025.1.*

Keywords: *Genetic Algorithm; Route Optimization; Traveling Salesman Problem; Visualization; Computational Intelligence.*

RESUMO. *Este trabalho apresenta o desenvolvimento de um algoritmo genético para a otimização de rotas, aplicado ao Problema do Caixeiro Viajante, com recurso de visualização aprimorada. A solução implementa operadores de seleção, cruzamento e mutação, aliados a elitismo, para maximizar o desempenho das rotas geradas. A interface gráfica permite acompanhar em tempo real a evolução das soluções, exibindo métricas de desempenho e rotas intermediárias. Solicitado pelo Professor Rômulo Nunes de Oliveira, na disciplina de Inteligência Artificial, no curso de Bacharelado em Ciência da Computação, turma CPTA126-01 2025.1.*

Palavras-chave: *Algoritmo Genético; Otimização de Rotas; Problema do Caixeiro Viajante; Visualização; Inteligência Computacional.*

1. Introdução ao tema e seus desafios

Os algoritmos genéticos (AGs) são técnicas de busca e otimização inspiradas nos processos de evolução natural propostos por Charles Darwin. Baseiam-se nos princípios de seleção natural, cruzamento e mutação, buscando encontrar soluções satisfatórias para problemas complexos, muitas vezes de difícil resolução por métodos determinísticos. A robustez dos AGs advém da sua capacidade de explorar e explorar simultaneamente o espaço de busca, adaptando-se gradualmente a soluções cada vez mais adequadas.

Um dos problemas clássicos que desafiam pesquisadores e engenheiros de computação é o Problema do Caixeiro Viajante (PCV). Nele, um agente deve visitar um conjunto de cidades uma única vez e retornar à cidade de origem, minimizando a distância percorrida. Esse problema é classificado como NP-difícil, o que significa que o tempo necessário para encontrar a solução ótima cresce exponencialmente com o número de cidades, inviabilizando soluções exatas em instâncias de grande porte.

No contexto atual, a otimização de rotas não se limita a aplicações teóricas: ela é fundamental em setores como logística, transporte urbano, entregas automatizadas e roteamento de redes. Tais cenários exigem soluções eficientes e, muitas vezes, em tempo real. A proposta aqui abordada consiste na utilização de um algoritmo genético com visualização aprimorada, permitindo não apenas a execução da otimização, mas também a interpretação gráfica da evolução das soluções ao longo das gerações.

2. Métodos Utilizados

A implementação desenvolvida é fundamentada nos princípios clássicos dos algoritmos genéticos, adaptando-os ao contexto específico de otimização de rotas. O método é composto por cinco componentes principais:

1. Codificação da solução – Cada indivíduo da população representa uma rota completa, onde a ordem dos genes corresponde à sequência das cidades visitadas.
2. Avaliação (Fitness) – O desempenho de cada indivíduo é calculado com base na distância total percorrida, utilizando a fórmula da distância euclidiana entre pares de cidades consecutivas. A função de avaliação atribui maior valor a rotas mais curtas.
3. Seleção – É adotada a estratégia de seleção por roleta, onde indivíduos com melhor fitness possuem maior probabilidade de serem escolhidos como pais para a próxima geração.
4. Crossover – Foi implementado um cruzamento do tipo “ordem preservada” (Order Crossover – OX), garantindo que as cidades não se repitam e mantendo segmentos herdados dos pais.
5. Mutação – Uma taxa fixa de mutação (20%) é aplicada de forma aleatória, trocando a posição de duas cidades na rota, preservando a diversidade populacional e evitando a estagnação precoce.

O processo é complementado pelo uso de elitismo, no qual um conjunto fixo dos melhores indivíduos de cada geração é mantido inalterado, garantindo que as melhores soluções não sejam perdidas.

3. Funcionamento e Processamento do Código

A estrutura do código é modular, dividindo-se em três blocos centrais: configuração inicial, funções do algoritmo genético e rotinas de visualização. Na configuração inicial, parâmetros como o número de indivíduos, taxa de mutação, número mínimo e máximo de cidades, e limites para estagnação são definidos. Um conjunto aleatório de cidades é gerado dentro de um plano bidimensional, cada uma com coordenadas distintas.

A execução do algoritmo segue o ciclo evolutivo clássico:

1. Inicialização da população – Uma população inicial é gerada aleatoriamente e avaliada.
2. Pré-seleção – Um painel interativo apresenta visualmente os indivíduos iniciais, permitindo a compreensão da diversidade de soluções antes da evolução.
   Ciclo de evolução – Em cada geração:

   a. O fitness médio e o melhor fitness são calculados e armazenados para posterior análise.
   b. A seleção por roleta e o crossover geram novos indivíduos.
   c. A mutação é aplicada de forma probabilística.
   d. O elitismo preserva os melhores indivíduos.

3. Critério de parada – O algoritmo encerra quando não há melhora significativa após um número pré-definido de gerações consecutivas, ou quando um padrão ótimo é encontrado.

O código também gerencia estados como pré-seleção, transição, pausa e encerramento, permitindo que a execução seja acompanhada em tempo real e interrompida ou reiniciada pelo usuário.

3.1. Exibição do Código

Vale ressaltar que, para a utilização e funcionamento deste código foi utilizada a ferramenta Processing, cujo manipulação foi realizada na linguagem Python.

```
# Simulacao de Algoritmo Genetico - Otimizacao de Rotas com
Visualizacao Aprimorada
# Processing (Python Mode)
import random
import math

# ----------------------------------------
# Configuracoes
# ----------------------------------------
NUM_INDIVIDUOS = 8          # Numero de individuos na populacao em
cada geracao
PRE_SELECTION_POOL_SIZE = 12 # Pool inicial para pre-selecao
ELITE_SIZE = 2
TAXA_MUTACAO = 0.2
GERACOES_ESTAGNADAS_MAX = 10
```

```
FRAMES_TRANSICAO = 30
PAUSE_FRAMES = 120

FITNESS_SCALE_FACTOR = 10000

# Intervalo de cidades aleatorio
NUM_CIDADES_MIN = 5
NUM_CIDADES_MAX = 10

# Variavel global para o numero de cidades (definida em setup)
NUM_CIDADES = 0

# ----------------------------------------
# Estruturas de dados
# ----------------------------------------
class Individuo:
    def __init__(self, rota=None):
        if rota:
            self.rota = rota[:]
        else:
            self.rota = random.sample(range(NUM_CIDADES),
NUM_CIDADES)
        self.fitness = self.calcular_fitness()
        self.elite = False
        self.mutado = False
        self.pais = None

    def calcular_fitness(self):
        dist = 0
        for i in range(len(self.rota)):
            c1 = cidades[self.rota[i]]
            c2 = cidades[self.rota[(i+1) % len(self.rota)]]
            dist += dist_euclidiana(c1, c2)

        if dist == 0: return float('inf')

        return (1 / dist) * FITNESS_SCALE_FACTOR

# ----------------------------------------
# Funcoes auxiliares do AG
# ----------------------------------------
def dist_euclidiana(c1, c2):
    return ((c1[0]-c2[0])**2 + (c1[1]-c2[1])**2) ** 0.5

def criar_populacao_inicial_pool():
    return [Individuo() for _ in range(PRE_SELECTION_POOL_SIZE)]

def criar_populacao():
    return [Individuo() for _ in range(NUM_INDIVIDUOS)]

def selecao_elite(pop):
    pop_ordenada = sorted(pop, key=lambda ind: ind.fitness,
reverse=True)
    for i, ind in enumerate(pop_ordenada):
        ind.elite = (i < ELITE_SIZE)
    return pop_ordenada
```

```
def crossover(pai, mae):
    start, end = sorted(random.sample(range(NUM_CIDADES), 2))
    filho_rota = [None] * NUM_CIDADES
    filho_rota[start:end] = pai.rota[start:end]

    pos = end
    for cidade in mae.rota:
        if cidade not in filho_rota:
            while filho_rota[pos % NUM_CIDADES] is not None:
                pos += 1
            filho_rota[pos % NUM_CIDADES] = cidade
            pos += 1

    filho = Individuo(filho_rota)
    filho.pais = (pai, mae)
    return filho

def mutacao(ind):
    if random.random() < TAXA_MUTACAO:
        i, j = random.sample(range(NUM_CIDADES), 2)
        ind.rota[i], ind.rota[j] = ind.rota[j], ind.rota[i]
        ind.mutado = True
        ind.fitness = ind.calcular_fitness()

def proxima_geracao(pop):
    nova_pop = []
    elite = selecao_elite(pop)[:ELITE_SIZE]
    nova_pop.extend(elite)

    while len(nova_pop) < NUM_INDIVIDUOS:
        total_fitness = sum(ind.fitness for ind in pop)

        pais = []
        for _ in range(2):
            roleta = random.uniform(0, total_fitness)
            acumulado = 0
            for ind in pop:
                acumulado += ind.fitness
                if acumulado >= roleta:
                    pais.append(ind)
                    break

        pai, mae = pais[0], pais[1]

        filho = crossover(pai, mae)
        mutacao(filho)
        nova_pop.append(filho)

    return nova_pop

# ----------------------------------------
# Setup e Draw
# ----------------------------------------
cidades = []
populacao = []
```

```
geracao = 0
prev_routes = []
anim_frame = 0
fitness_medias = []
fitness_melhor = []
global_state = "PAUSE"
best_individual_final = None
geracoes_sem_melhora = 0
melhor_fitness_geral = 0
geracao_final_encontrada = 0
initial_pool = []

def setup():
    global cidades, populacao, geracao, prev_routes, anim_frame,
global_state
    global fitness_medias, fitness_melhor, best_individual_final
    global geracoes_sem_melhora, melhor_fitness_geral,
geracao_final_encontrada, NUM_CIDADES
    global initial_pool

    size(900, 600)
    textFont(createFont("Arial", 12))
    random.seed(random.randint(0, 10000))

    NUM_CIDADES = random.randint(NUM_CIDADES_MIN, NUM_CIDADES_MAX)

    cidades = [(random.randint(550, 850), random.randint(100,
550)) for _ in range(NUM_CIDADES)]
    initial_pool = criar_populacao_inicial_pool()
    populacao = []
    geracao = 0
    prev_routes = []
    anim_frame = 0

    fitness_medias = []
    fitness_melhor = []

    global_state = "PRE_SELECTION"
    best_individual_final = None
    geracoes_sem_melhora = 0
    melhor_fitness_geral = 0
    geracao_final_encontrada = 0

def draw():
    global populacao, geracao, prev_routes, anim_frame,
global_state
    global fitness_medias, fitness_melhor, best_individual_final
    global geracoes_sem_melhora, melhor_fitness_geral,
geracao_final_encontrada

    background(240)

    if global_state == "PRE_SELECTION":
        draw_pre_selection_panel()
        return
```

```
    draw_ui_panel()

    if global_state == "END":
        draw_best_route(best_individual_final.rota)
        return

    draw_map_and_routes()

    if global_state == "TRANSITION":
        anim_frame += 1
        if anim_frame > FRAMES_TRANSICAO:
            global_state = "PAUSE"
            anim_frame = 0
    elif global_state == "PAUSE":
        anim_frame += 1
        if anim_frame > PAUSE_FRAMES:
            if geracoes_sem_melhora >= GERACOES_ESTAGNADAS_MAX:
                best_individual_final =
selecao_elite(populacao)[0]
                geracao_final_encontrada = geracao
                global_state = "END"
            else:
                prev_routes = [ind.rota[:] for ind in populacao]
                populacao = proxima_geracao(populacao)
                geracao += 1

                novo_melhor_fitness =
selecao_elite(populacao)[0].fitness
                if novo_melhor_fitness > melhor_fitness_geral:
                    melhor_fitness_geral = novo_melhor_fitness
                    geracoes_sem_melhora = 0
                else:
                    geracoes_sem_melhora += 1

                fitness_medias.append(sum(ind.fitness for ind in
populacao) / NUM_INDIVIDUOS)
                fitness_melhor.append(novo_melhor_fitness)

                global_state = "TRANSITION"
                anim_frame = 0

def draw_pre_selection_panel():
    fill(255)
    stroke(200)
    rect(10, 10, 500, 580, 8)

    fill(0)
    textSize(18)
    text("Pre-selecao da Populacao Inicial", 20, 30)

    textSize(14)
    text("Pressione 'espaco' para iniciar a evolucao", 20, 55)

    pop_draw = sorted(initial_pool, key=lambda ind: ind.fitness,
reverse=True)
```

```
    box_w = 230
    box_h = 70
    margin_x = 10
    margin_y = 10

    for i, ind in enumerate(pop_draw):
        x_box = 20 + (i % 2) * (box_w + margin_x)
        y_box = 80 + (i // 2) * (box_h + margin_y)

        fill(255)
        stroke(200)

        if i < NUM_INDIVIDUOS:
            stroke(255, 215, 0)
            strokeWeight(3)
        else:
            stroke(200)
            strokeWeight(1)

        rect(x_box, y_box, box_w, box_h, 8)

        fill(200, 200, 255, 100)
        noStroke()
        rect(x_box + 5, y_box + 5, box_w-10, box_h-10, 5)

        fill(0)
        textSize(12)
        text("Fitness: {:.2f}".format(ind.fitness), x_box + 15,
y_box + 25)

        textSize(10)
        rota_txt = "->".join([chr(65+c) for c in ind.rota])
        text("Rota: {}".format(rota_txt), x_box + 15, y_box + 45)

    draw_pre_selection_map()

def draw_pre_selection_map():
    map_x_offset = 550
    map_y_offset = 100
    map_width = width - map_x_offset - 50
    map_height = height - map_y_offset - 50

    min_x = min(cidade_x for cidade_x, cidade_y in cidades)
    max_x = max(cidade_x for cidade_x, cidade_y in cidades)
    min_y = min(cidade_y for cidade_x, cidade_y in cidades)
    max_y = max(cidade_y for cidade_x, cidade_y in cidades)

    range_x = max_x - min_x
    range_y = max_y - min_y

    scale_x = map_width / range_x if range_x != 0 else 1
    scale_y = map_height / range_y if range_y != 0 else 1
    scale = min(scale_x, scale_y)

    center_x = map_x_offset + map_width / 2
    center_y = map_y_offset + map_height / 2
```

```
    cities_center_x = min_x + range_x / 2
    cities_center_y = min_y + range_y / 2

    # Desenha todas as rotas do pool inicial
    stroke(0, 100, 255, 60)
    strokeWeight(1)
    for ind in initial_pool:
        for j in range(len(ind.rota)):
            x1, y1 = cidades[ind.rota[j]]
            x2, y2 = cidades[ind.rota[(j+1) % len(ind.rota)]]

            scaled_x1 = (x1 - cities_center_x) * scale + center_x
            scaled_y1 = (y1 - cities_center_y) * scale + center_y
            scaled_x2 = (x2 - cities_center_x) * scale + center_x
            scaled_y2 = (y2 - cities_center_y) * scale + center_y

            line(scaled_x1, scaled_y1, scaled_x2, scaled_y2)

    # Desenha as cidades por cima
    for idx_cidade, (x, y) in enumerate(cidades):
        scaled_x = (x - cities_center_x) * scale + center_x
        scaled_y = (y - cities_center_y) * scale + center_y
        fill(0, 150, 0)
        ellipse(scaled_x, scaled_y, 12, 12)
        fill(255)
        textSize(10)
        textAlign(CENTER, CENTER)
        text(chr(65+idx_cidade), int(scaled_x), int(scaled_y))
        textAlign(LEFT, BASELINE)

def draw_ui_panel():
    panel_w = 500
    panel_h = 580
    rect(10, 10, panel_w, panel_h, 8)

    fill(0)
    textSize(18)
    text("Algoritmo Genetico - Otimizacao de Rotas", 20, 30)

    if global_state != "END":
        textSize(14)
        text("Geracao: {}".format(geracao + 1), 20, 55)
        draw_legend(20, 80)
        draw_individuals(20, 150)
        draw_stats_graph(20, 455, 480, 120)
    else:
        draw_final_info(20, 55)

def draw_final_info(x, y):
    textSize(14)
    fill(0, 100, 0)
    text("SIMULACAO CONCLUIDA!", x, y)

    fill(0)
    text("Melhor Rota Encontrada:", x, y + 30)
```

```
    if best_individual_final:
        rota_txt = "->".join([chr(65+c) for c in
best_individual_final.rota])
        text("Geracao: {}".format(geracao_final_encontrada), x +
20, y + 60)
        text("Rota: {}".format(rota_txt), x + 20, y + 80)
        text("Fitness:
{:.2f}".format(best_individual_final.fitness), x + 20, y + 100)

def draw_legend(x, y):
    noStroke()

    fill(255, 215, 0)
    ellipse(x + 10, y, 15, 15)
    fill(0)
    text("Elite", x + 30, y + 5)

    fill(255, 100, 100)
    ellipse(x + 10, y + 20, 15, 15)
    fill(0)
    text("Mutado", x + 30, y + 25)

    fill(180)
    ellipse(x + 10, y + 40, 15, 15)
    fill(0)
    text("Normal", x + 30, y + 45)

def draw_individuals(x_base, y_base):
    box_w = 230
    box_h = 70
    margin_x = 10
    margin_y = 10

    pop_draw = sorted(populacao, key=lambda ind: ind.fitness,
reverse=True)

    for i, ind in enumerate(pop_draw):
        x_box = x_base + (i % 2) * (box_w + margin_x)
        y_box = y_base + (i // 2) * (box_h + margin_y)

        fill(255)
        stroke(200)
        rect(x_box, y_box, box_w, box_h, 8)

        if ind.elite:
            stroke(255, 215, 0)
            strokeWeight(3)
        elif ind.mutado:
            stroke(255, 0, 0)
            strokeWeight(2)
        else:
            stroke(100)
            strokeWeight(1)

        fill(200, 200, 255, 100)
```

```
        rect(x_box + 5, y_box + 5, box_w-10, box_h-10, 5)

        fill(0)
        textSize(12)
        text("Fitness: {:.2f}".format(ind.fitness), x_box + 15,
y_box + 25)

        textSize(10)
        rota_txt = "->".join([chr(65+c) for c in ind.rota])
        text("Rota: {}".format(rota_txt), x_box + 15, y_box + 45)

        if ind.pais:
            fill(0, 150, 0)
            text("Filho da G{}.".format(geracao), x_box + 15,
y_box + 60)

def draw_map_and_routes():
    if best_individual_final: return

    map_x_offset = 550
    map_y_offset = 100
    map_width = width - map_x_offset - 50
    map_height = height - map_y_offset - 50

    min_x = min(cidade_x for cidade_x, cidade_y in cidades)
    max_x = max(cidade_x for cidade_x, cidade_y in cidades)
    min_y = min(cidade_y for cidade_x, cidade_y in cidades)
    max_y = max(cidade_y for cidade_x, cidade_y in cidades)

    range_x = max_x - min_x
    range_y = max_y - min_y

    scale_x = map_width / range_x if range_x != 0 else 1
    scale_y = map_height / range_y if range_y != 0 else 1
    scale = min(scale_x, scale_y)

    center_x = map_x_offset + map_width / 2
    center_y = map_y_offset + map_height / 2

    cities_center_x = min_x + range_x / 2
    cities_center_y = min_y + range_y / 2

    for idx_cidade, (x, y) in enumerate(cidades):
        scaled_x = (x - cities_center_x) * scale + center_x
        scaled_y = (y - cities_center_y) * scale + center_y
        fill(0, 150, 0)
        ellipse(scaled_x, scaled_y, 12, 12)
        fill(255)
        textSize(10)
        textAlign(CENTER, CENTER)
        text(chr(65+idx_cidade), int(scaled_x), int(scaled_y))
        textAlign(LEFT, BASELINE)

    t = anim_frame / float(FRAMES_TRANSICAO)
    if t > 1: t = 1
```

```
    for ind_idx, ind in enumerate(populacao):
        stroke(0, 100, 255, 120)
        strokeWeight(1)

        if ind_idx < len(prev_routes):
            prev_rota = prev_routes[ind_idx]
        else:
            prev_rota = ind.rota

        for j in range(len(ind.rota)):
            cidade_prev_idx = prev_rota[j % len(prev_rota)]
            cidade_new_idx = ind.rota[j % len(ind.rota)]

            x1_prev, y1_prev = cidades[cidade_prev_idx %
len(cidades)]
            x1_new, y1_new = cidades[cidade_new_idx %
len(cidades)]

            x2_prev, y2_prev = cidades[prev_rota[(j+1) %
len(prev_rota)] % len(cidades)]
            x2_new, y2_new = cidades[ind.rota[(j+1) %
len(ind.rota)] % len(cidades)]

            scaled_x1_prev = (x1_prev - cities_center_x) * scale +
center_x
            scaled_y1_prev = (y1_prev - cities_center_y) * scale +
center_y
            scaled_x1_new = (x1_new - cities_center_x) * scale +
center_x
            scaled_y1_new = (y1_new - cities_center_y) * scale +
center_y

            scaled_x2_prev = (x2_prev - cities_center_x) * scale +
center_x
            scaled_y2_prev = (y2_prev - cities_center_y) * scale +
center_y
            scaled_x2_new = (x2_new - cities_center_x) * scale +
center_x
            scaled_y2_new = (y2_new - cities_center_y) * scale +
center_y

            x1 = lerp(scaled_x1_prev, scaled_x1_new, t)
            y1 = lerp(scaled_y1_prev, scaled_y1_new, t)

            x2 = lerp(scaled_x2_prev, scaled_x2_new, t)
            y2 = lerp(scaled_y2_prev, scaled_y2_new, t)

            line(x1, y1, x2, y2)

def draw_best_route(rota):
    map_x_offset = 550
    map_y_offset = 100
    map_width = width - map_x_offset - 50
    map_height = height - map_y_offset - 50

    min_x = min(cidade_x for cidade_x, cidade_y in cidades)
```

```
    max_x = max(cidade_x for cidade_x, cidade_y in cidades)
    min_y = min(cidade_y for cidade_x, cidade_y in cidades)
    max_y = max(cidade_y for cidade_x, cidade_y in cidades)

    range_x = max_x - min_x
    range_y = max_y - min_y

    scale_x = map_width / range_x if range_x != 0 else 1
    scale_y = map_height / range_y if range_y != 0 else 1
    scale = min(scale_x, scale_y)

    center_x = map_x_offset + map_width / 2
    center_y = map_y_offset + map_height / 2

    cities_center_x = min_x + range_x / 2
    cities_center_y = min_y + range_y / 2

    for idx_cidade, (x, y) in enumerate(cidades):
        scaled_x = (x - cities_center_x) * scale + center_x
        scaled_y = (y - cities_center_y) * scale + center_y
        fill(0, 150, 0)
        ellipse(scaled_x, scaled_y, 12, 12)
        fill(255)
        textSize(10)
        textAlign(CENTER, CENTER)
        text(chr(65+idx_cidade), int(scaled_x), int(scaled_y))
        textAlign(LEFT, BASELINE)

    stroke(255, 0, 0)
    strokeWeight(3)
    for j in range(len(rota)):
        x1, y1 = cidades[rota[j % len(rota)] % len(cidades)]
        x2, y2 = cidades[rota[(j+1) % len(rota)] % len(cidades)]

        scaled_x1 = (x1 - cities_center_x) * scale + center_x
        scaled_y1 = (y1 - cities_center_y) * scale + center_y
        scaled_x2 = (x2 - cities_center_x) * scale + center_x
        scaled_y2 = (y2 - cities_center_y) * scale + center_y

        line(scaled_x1, scaled_y1, scaled_x2, scaled_y2)

def draw_stats_graph(x, y, w, h):
    graph_h = 80

    fill(0)
    textSize(14)
    text("Evolucao da Pontuacao", x, y)

    stroke(100); noFill(); rect(x, y + 10, w, graph_h); noStroke()

    if len(fitness_medias) > 1:
        max_fitness = max(max(fitness_medias),
max(fitness_melhor))
        min_fitness = 0
        denom = max_fitness - min_fitness if max_fitness >
min_fitness else 1.0
```

```
        stroke(0, 100, 255)
        strokeWeight(2)
        noFill()
        beginShape()
        for i, m in enumerate(fitness_medias):
            px = map(i, 0, max(1, len(fitness_medias)-1), x, x+w)
            py = map(m, min_fitness, max_fitness, y + graph_h +
10, y + 10)
            vertex(px, py)
        endShape()

        stroke(255, 215, 0)
        strokeWeight(2)
        noFill()
        beginShape()
        for i, m in enumerate(fitness_melhor):
            px = map(i, 0, max(1, len(fitness_melhor)-1), x, x+w)
            py = map(m, min_fitness, max_fitness, y + graph_h +
10, y + 10)
            vertex(px, py)
        endShape()

    if len(fitness_medias) > 0:
        fill(0)
        textSize(12)
        text("Media Atual: {:.2f}".format(fitness_medias[-1]), x,
y + graph_h + 30)
        text("Melhor Atual: {:.2f}".format(fitness_melhor[-1]), x
+ w/2, y + graph_h + 30)

    fill(0)
    textSize(12)
    text("Geracoes sem melhora:
{}/{}".format(geracoes_sem_melhora, GERACOES_ESTAGNADAS_MAX), x, y
+ graph_h + 50)

def keyPressed():
    global global_state, anim_frame, populacao, initial_pool
    if key == ' ':
        if global_state == "PRE_SELECTION":
            # Realiza a selecao e inicializa a primeira geracao
            populacao = sorted(initial_pool, key=lambda ind:
ind.fitness, reverse=True)[:NUM_INDIVIDUOS]

            global fitness_medias, fitness_melhor,
melhor_fitness_geral
            fitness_medias.append(sum(ind.fitness for ind in
populacao) / NUM_INDIVIDUOS)
            melhor_fitness = selecao_elite(populacao)[0].fitness
            fitness_melhor.append(melhor_fitness)
            melhor_fitness_geral = melhor_fitness

            global_state = "PAUSE"
            anim_frame = 0
```

```
        elif global_state == "PAUSE":
            anim_frame = PAUSE_FRAMES
        elif global_state == "TRANSITION":
            anim_frame = FRAMES_TRANSICAO

    if key == 'r':
        setup()

def mousePressed():
    setup()
```

4. Visualização

O diferencial desta implementação está na visualização aprimorada do processo evolutivo. A interface gráfica apresenta:

a. Mapa das cidades – Representadas como pontos verdes numerados, facilitando a identificação da ordem de visita.
b. Rotas de cada indivíduo – Desenhadas como linhas, com cores e espessuras diferentes para distinguir elites, mutados e indivíduos comuns.
c. Evolução do fitness – Um gráfico exibe a curva do fitness médio e do melhor indivíduo ao longo das gerações, permitindo avaliar a convergência do algoritmo.
d. Animação de transição – Durante a mudança de geração, as rotas se interpolam suavemente entre as configurações anteriores e as novas, facilitando a percepção das alterações introduzidas por cruzamento e mutação.
e. Exibição da melhor rota final – Ao término da execução, a rota ótima encontrada é destacada em vermelho, junto com seu valor fitness e a geração em que foi obtida.

4.1 Representação Ilustrativa

O experimento realizado demonstra a aplicação de um Algoritmo Genético para resolver o problema de Otimização de Rotas, com o objetivo de determinar o caminho mais eficiente que conecta todos os pontos de interesse do mapa.

4.1.1 Descrição da Evolução das Gerações

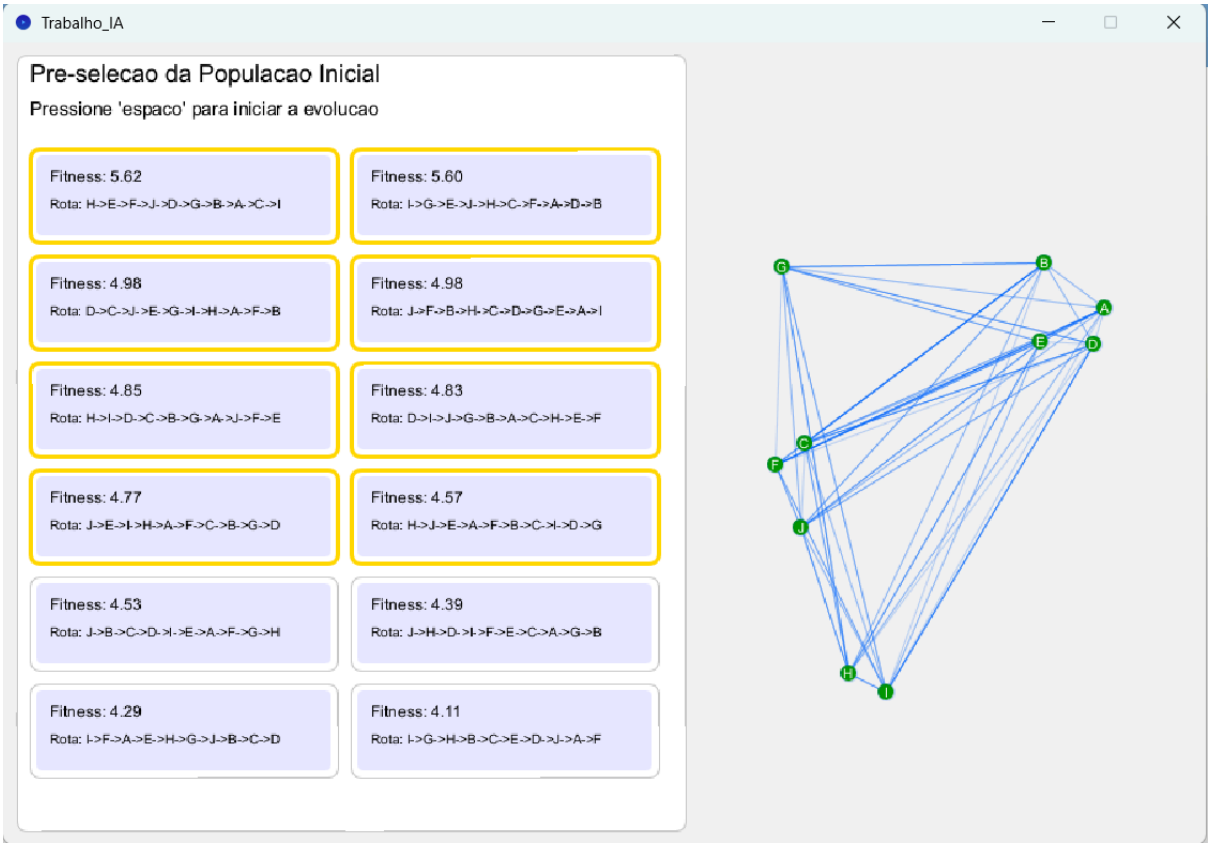

Figura 1. Representação Ilustrativa: Pré-Seleção de Indivíduos

O programa exibe a pré-seleção da população inicial do algoritmo genético aplicado ao problema do caixeiro viajante (TSP). No painel esquerdo, cada caixa representa um indivíduo, ou seja, uma rota possível para visitar todas as cidades. O valor de fitness indica a qualidade da solução, sendo maior para rotas com menor distância total. A sequência das cidades é apresentada por letras de A a J, e as bordas amarelas destacam os indivíduos mais aptos, que terão maior chance de participar do cruzamento e gerar novas soluções. No painel direito, vê-se o mapa das cidades, onde nós verdes representam as cidades e linhas azuis indicam todas as conexões possíveis entre elas. Nesta fase, nenhuma rota específica está destacada, pois o sistema está apenas avaliando as soluções iniciais.

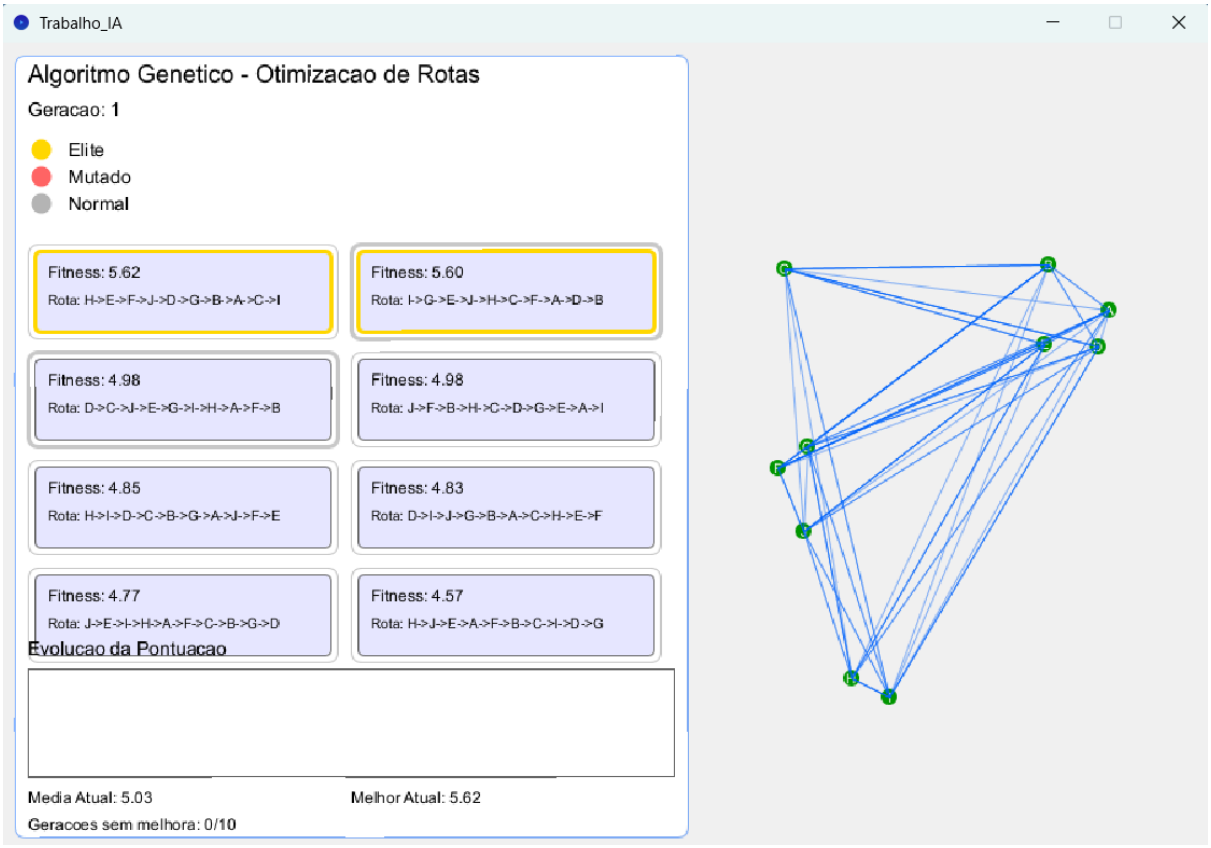

Figura 2. Representação Ilustrativa: 1ª Geração de Indivíduos

Na primeira execução, o sistema inicia com uma população de rotas geradas de forma aleatória. O valor de fitness inicial registrado é 5,62, indicando que o trajeto contém deslocamentos longos e pouco otimizados. Nesta fase, o algoritmo ainda não realizou adaptações significativas e as conexões entre os pontos não seguem um padrão de minimização de distância.

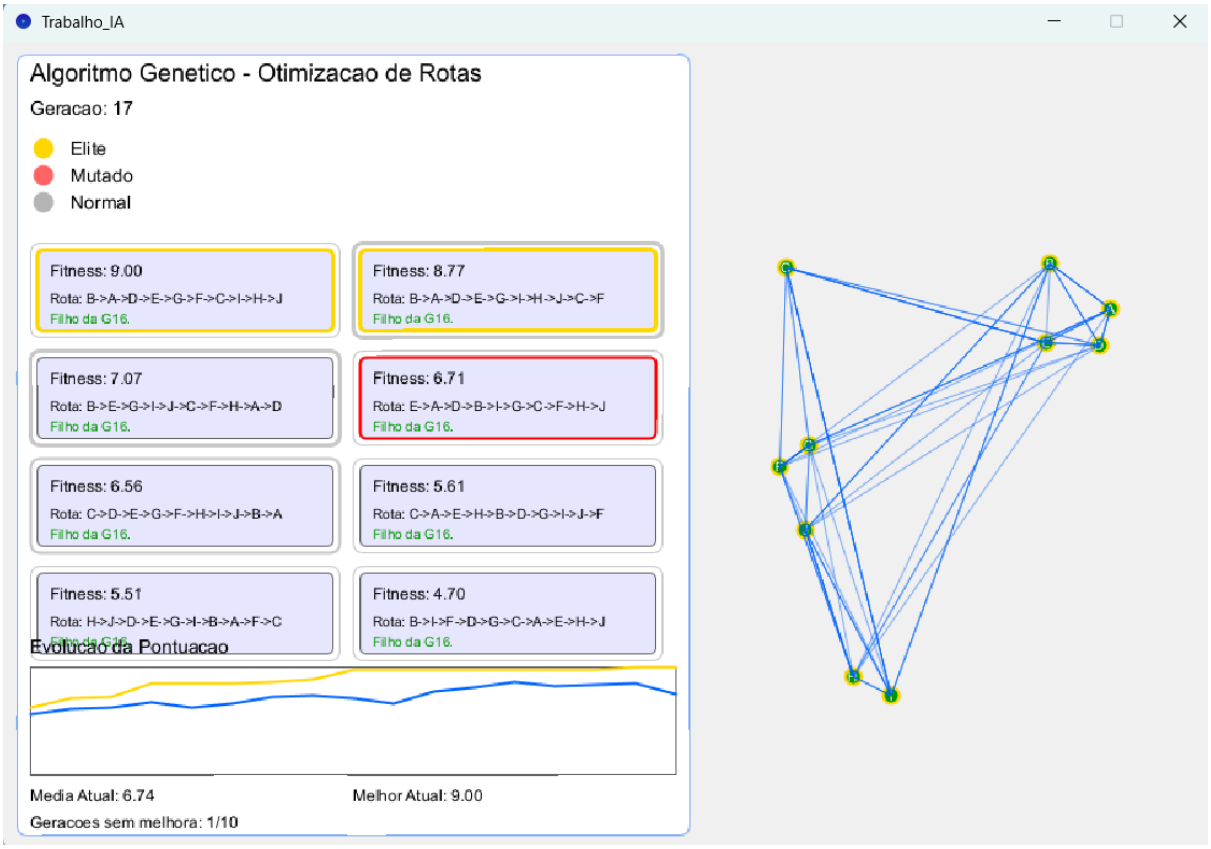

Figura 3. Representação Ilustrativa: 17ª Geração de Indivíduos

Após 17 gerações, o algoritmo apresenta melhorias consistentes na rota, alcançando um fitness de aproximadamente 9,00. Esse avanço se deve aos operadores genéticos de seleção, cruzamento e mutação, que, ao longo das iterações, preservam características vantajosas e reduzem distâncias desnecessárias no trajeto.

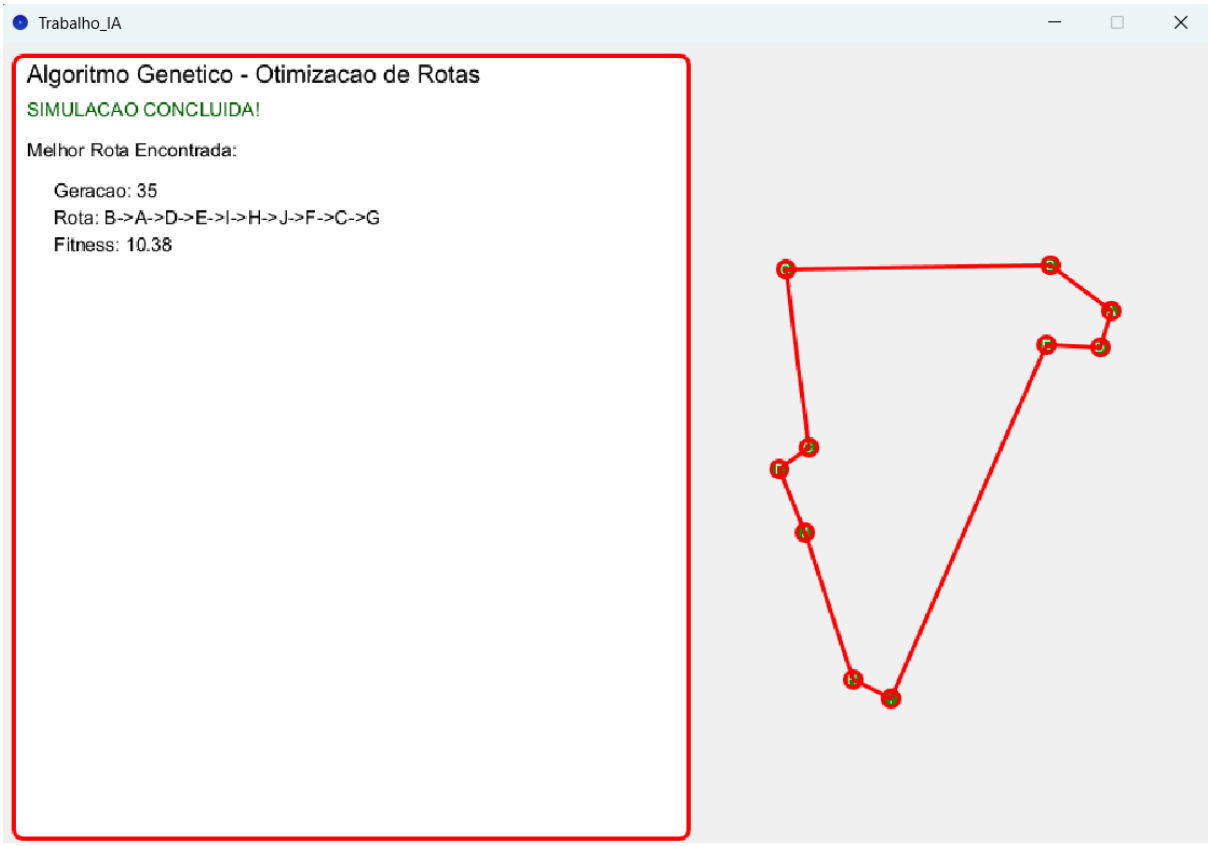

Figura 4. Representação Ilustrativa: Finalização da Simulação

A simulação é concluída na geração 35, momento em que o algoritmo encontra a melhor rota global registrada: B → A → D → E → I → H → J → F → C → G
O valor de fitness obtido foi 10,38, representando a configuração mais eficiente identificada durante toda a execução. O gráfico à direita da interface exibe visualmente o trajeto final, com as conexões entre os pontos destacadas em vermelho, evidenciando a rota otimizada.

4.1.2 Conclusão da Simulação

O resultado confirma a eficácia do algoritmo genético na solução de problemas complexos de otimização. A partir de soluções iniciais aleatórias, o método foi capaz de, em poucas gerações, alcançar um trajeto significativamente mais eficiente, reduzindo distâncias percorridas e melhorando o desempenho global.
Essa abordagem de visualização não apenas torna o processo mais intuitivo para o observador, como também auxilia na depuração do código e na análise do comportamento do algoritmo frente a diferentes configurações e parâmetros.

5. Considerações Finais

A aplicação de algoritmos genéticos para a otimização de rotas, particularmente no contexto do Problema do Caixeiro Viajante, demonstra a capacidade desta técnica de fornecer soluções eficientes para problemas complexos e de grande relevância prática. A implementação apresentada, além de atender à função de otimizar rotas, incorpora recursos visuais que potencializam a compreensão do processo evolutivo, tornando-a adequada para fins educacionais, experimentais e demonstrativos.

Apesar dos bons resultados obtidos, é importante destacar que o desempenho de um algoritmo genético depende fortemente da configuração de seus parâmetros. Fatores como tamanho da população, taxa de mutação, método de seleção e pressão seletiva influenciam diretamente a qualidade das soluções e a velocidade de convergência.

Como perspectivas futuras, pode-se considerar a integração de heurísticas híbridas, como inserção de soluções obtidas por algoritmos gulosos na população inicial, ou a aplicação de métodos de ajuste dinâmico da taxa de mutação. Essas abordagens podem ampliar a eficiência do algoritmo e reduzir a probabilidade de estagnação em ótimos locais.

Assim, a proposta aqui desenvolvida não apenas reforça a importância dos algoritmos genéticos no campo da otimização, mas também evidencia o papel fundamental de uma visualização bem projetada como ferramenta de análise e compreensão em problemas complexos de computação.

REFERÊNCIAS

1. SOARES, Gabriel; BULHÕES, Teobaldo; BRUCK, Bruno. *Um algoritmo genético híbrido para o problema do caixeiro viajante com tempos de liberação*. In: ENCONTRO DE TEORIA DA COMPUTAÇÃO (ETC), 8., 2023, João Pessoa/PB. Anais . Porto Alegre: Sociedade Brasileira de Computação, 2023. p. 160-164. ISSN 2595-6116. DOI: https://doi.org/10.5753/etc.2023.230515. Acesso em: 09 ago. 2025.

2. GONÇALVES, Wellington; OLIVEIRA, Matheus Sales; ROCHA, Alessandro Roberto. *Algoritmo genético aplicado ao problema de roteamento de veículos: problema do caixeiro viajante no setor varejista*. Cadernos UniFOA, Volta Redonda, v. 15, n. 43, 2020. DOI: 10.47385/cadunifoa.v15.n43.3273. Disponível em: https://revistas.unifoa.edu.br/cadernos/article/view/3273. Acesso em: 10 ago. 2025 Portal de Revistas UniFOA.

3. BRAGA, Edgar Augusto Silva; DROGUETT, Enrique Andrés López. *Modelagem e otimização do problema do caixeiro viajante com restrições de tempo, distância e confiabilidade via algoritmos genéticos*. 2007. Dissertação (Mestrado em Engenharia de Produção) – Universidade Federal de Pernambuco, Recife, 2007. Acesso em: 11 ago. 2025 Repositório UFPE.

4. RAMOS, Iloneide Carlos de Oliveira. *Metodologia estatística na solução do problema do caixeiro viajante e na avaliação de algoritmos: um estudo aplicado à transgenética computacional*. 2005. Tese (Doutorado em Automação e Sistemas; Engenharia de Computação; Telecomunicações) – Universidade Federal do Rio Grande do Norte, Natal, 2005. Acesso em: 11 ago. 2025 Repositório UFRN.

5. APPLEGATE, David L.; et al. *The travelling salesman problem: a computational Study*. Princeton: Princeton University Press, 2006. ISBN 978-0-691-12993-8. Acesso em: 12 ago. 2025 Wikipédia.

6. BÄCK, Thomas. *Evolutionary algorithms in theory and practice*. New York: Oxford University Press, 1996. ISBN 978-0-195-09971-3. Acesso em: 12 ago. 2025 Wikipédia.

7. COSTA, Fredson Vieira; VIDAL, Fábio Silveira; ANDRÉ, Claudomiro Moura Gomes. *SLAG – Resolvendo o problema do caixeiro viajante utilizando algoritmos genéticos* [em linha]. Tocantins, Brasil: Sistemas e Computação – Pós-Graduação, 2003. Acesso em: 12 ago. 2025 Wikipédia.

8. CUNHA, Claudio Barbieri da; BONASSER, Ulisses de Oliveira; ABRAHÃO, Fernando Teixeira Mendes. *Experimentos computacionais com heurísticas de melhorias para o problema do caixeiro viajante* [em linha]. São Paulo: ANPET, 2002. Acesso em: 12 ago. 2025 Wikipédia.