Documentação do Projeto 4: Algoritmo Genético

1. Visão Geral do Projeto

Este projeto é uma implementação clássica de um **Algoritmo Genético (AG)** para resolver um problema de otimização. O objetivo é "evoluir" uma população de sequências de caracteres aleatórias até que uma delas corresponda perfeitamente a uma frase-alvo (neste caso, "HELLO WORLD").

A aplicação é executada via console e imprime o "melhor indivíduo" (a frase mais próxima do alvo) de cada geração, permitindo ao usuário observar o processo de evolução em tempo real até que a solução perfeita seja encontrada.

2. Como Foi Desenvolvido

O projeto foi desenvolvido inteiramente em **Python**, focando nos mecanismos centrais do algoritmo genético.

- **Linguagem:** Python.
- **Bibliotecas:** random (para todas as operações estocásticas: população inicial, seleção, cruzamento e mutação) e string (para definir o "alfabeto" de genes permitidos).
- **Interface:** Baseada em terminal. O progresso é exibido através de print(), mostrando o número da geração, o melhor fitness e a melhor frase encontrada até o momento.
- **Estrutura de Dados:** Uma @dataclass chamada Individual é usada para representar cada solução. Ela armazena os genes (a lista de caracteres) e o fitness (a pontuação de quão boa ela é).

3. Algoritmo Utilizado

Algoritmo Genético (AG)

Um Algoritmo Genético é uma técnica de busca e otimização inspirada na teoria da **evolução natural** de Charles Darwin. Ele opera sobre uma "população" de soluções candidatas, aplicando os seguintes mecanismos iterativamente:

1. **População Inicial:** O programa começa criando uma grande população (POP_SIZE) de indivíduos, cada um com uma sequência de genes (caracteres) completamente aleatória.
2. **Avaliação (Fitness):** Cada indivíduo é avaliado por uma **função de fitness**. Neste código (evaluate_fitness), o fitness é simplesmente o número de caracteres que estão na posição correta, em comparação com a frase-alvo.

3. **Seleção:** Indivíduos com maior fitness são selecionados para se tornarem "pais" da próxima geração. Este projeto utiliza a **Seleção por Torneio (tournament_selection)**, onde um pequeno subconjunto de indivíduos é escolhido aleatoriamente, e o melhor desse subconjunto é selecionado como pai.
4. **Cruzamento (Crossover):** Dois pais são combinados para criar "filhos", misturando seu material genético. O código usa o **Crossover de Ponto Único (single_point_crossover)**: um ponto de corte aleatório é escolhido, e os pais trocam os genes após esse ponto.
5. **Mutação (mutate):** Uma pequena chance (MUTATION_RATE) é aplicada a cada gene de um filho. Se um gene sofrer mutação, ele é trocado por outro caractere aleatório do alfabeto. Isso introduz nova diversidade genética e evita que o algoritmo fique preso em um ótimo local.
6. **Elitismo:** Para garantir que a melhor solução encontrada nunca seja perdida, a técnica de **Elitismo (ELITISM)** é usada: o melhor indivíduo da geração atual é copiado diretamente para a próxima geração.

Esse ciclo se repete por um número máximo de gerações (MAX_GENERATIONS) ou até que um indivíduo com fitness perfeito (igual ao tamanho da frase-alvo) seja encontrado.

```python
def genetic_algorithm():
    """Algoritmo Genético simples para evoluir uma string alvo a partir de uma população aleatória.
    O algoritmo utiliza seleção por torneio, crossover de ponto único e mutação ponto-a-ponto
    em uma população de indivíduos representados como listas de caracteres."""
    target_len = len(TARGET)
    population = make_initial_population(POP_SIZE, target_len)
    # Avalia população inicial
    for ind in population:
        evaluate_fitness(ind, TARGET)

    best = max(population, key=lambda i: i.fitness)

    generation = 0
    while generation < MAX_GENERATIONS and best.fitness < target_len:
        new_population = []

        # Elitismo: carrega o melhor indivíduo para a próxima geração
        if ELITISM:
            new_population.append(Individual(best.genes.copy(), best.fitness))

        # Gera novos indivíduos até completar a população
        while len(new_population) < POP_SIZE:
            # Seleção: escolhe dois pais via torneio
            parent1 = tournament_selection(population, TOURNAMENT_K)
            parent2 = tournament_selection(population, TOURNAMENT_K)

            # Cruzamento
            child1, child2 = single_point_crossover(parent1, parent2)

            # Mutação
            mutate(child1, MUTATION_RATE)
            mutate(child2, MUTATION_RATE)

            # Avalia os filhos
            evaluate_fitness(child1, TARGET)
            evaluate_fitness(child2, TARGET)

            new_population.append(child1)
            if len(new_population) < POP_SIZE:
                new_population.append(child2)

        population = new_population
        best = max(population, key=lambda i: i.fitness)
        generation += 1

        # Imprime progresso a cada 5 gerações ou se encontrar a solução
        if generation % 5 == 0 or best.fitness == target_len:
            print(
                f"Geração {generation:4d} | Melhor fitness: {best.fitness:2d} | '{best.as_string()}'")
```

Figura 01: Visão geral do código

```
❯ /home/gabriel/inteligencia-artificial/env/bin/python
Geração    5 | Melhor fitness:  7 | 'HROLOKWORQD'
Geração   10 | Melhor fitness: 10 | 'HELLOCWORLD'
Geração   12 | Melhor fitness: 11 | 'HELLO WORLD'

=== RESULTADO ===
Gerações: 12
Melhor indivíduo: 'HELLO WORLD' (fitness 11/11)
```

Figura 02: Visão geral da saída do algoritmo