UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE COMPUTAÇÃO

DISCIPLINA: Inteligência Artificial

DOCENTE: Hendrik

AVALIAÇÃO 3 - Caixeiro Viajante com Algoritmo Genético

PARTICIPANTES:

LUAN ALMEIDA VALENÇA

LUIZ FELIPE TOJAL GOMES CORUMBA

RAFAEL MACHADO COSTA MENESES

RAFAEL SANTOS SILVA

SÃO CRISTÓVÃO – SE

02/2026

1. Descrição do Problema

O problema escolhido para o projeto é o problema do caixeiro viajante. Nesse cenário, trata-se de um desafio de otimização onde o objetivo é encontrar a rota mais curta que visite um conjunto de cidades exatamente uma vez e retorne à cidade de origem.

Dessa forma, à medida que o número de cidades aumenta, o número de rotas possíveis cresce de forma fatorial, tornando inviável por meio de força bruta. Por isso, utilizamos o algoritmo genético, para encontrar soluções próximas do ideal em um tempo aceitável.

2. Algoritmo Genético

O algoritmo genético é uma meta-heurística inspirada na teoria da evolução de Darwin. De forma que se encaixa no que o problema necessita, pois trabalha com uma população de rotas que evoluem ao longo de gerações através de:

1. Seleção: escolha dos indivíduos mais aptos (quanto menor, melhor).
2. *Crossover*: troca de informações genéticas entre pais para gerar filhos.
3. Mutação: introdução de variações aleatórias para evitar a convergência prematura em locais mínimos.

3. AG para o TSP

Para entender como o Algoritmo Genético encontra a rota ideal para o Caixeiro Viajante, nós observamos o ciclo de vida da população de rotas. O AG não constrói uma rota cidade por cidade de forma lógica, assim como fariam algoritmos gulosos, em vez disso, ele cria um conjunto de rotas inteiras e as melhora iterativamente. O processo ocorre nas seguintes etapas:

- **Chute Inicial:** Cria aleatoriamente várias rotas completas.
- **Avaliação e Seleção:** Mede a distância de todas elas e prioriza as mais curtas.
- **Cruzamento:** Mistura os trechos dessas rotas boas para criar novas opções de caminhos, combinando o que cada uma tinha de melhor.
- **Mutação:** De forma rara e aleatória, inverte a ordem de algumas cidades na rota para tentar desfazer nós e descobrir atalhos inéditos.
- **Evolução:** Guarda as rotas excelentes, descarta as ruins e repete esse processo de mistura e melhoria milhares de vezes. A cada geração, o caminho vai ficando cada vez mais curto e limpo.

4. Pseudocódigo e Implementação

Abaixo, detalhamos onde cada etapa do pseudocódigo se manifesta no código:

A. Genetic-Algorithm

Pseudocódigo	**Implementação**
repeat ... until enough time has elapsed	for generation in range(self.num_generations):
weights ← WEIGHTED-BY(population, fitness)	weights, fit_vals = self.weighted_by(pop)

parent1, parent2 ← WEIGHTED-RANDOM-CHOICES	parent1, parent2 = random.choices(pop, weights=weights, k=2)
child ← REPRODUCE(parent1, parent2)	filho = self.reproduce(parent1, parent2)
if (small random prob) then child ← MUTATE	filho = self.mutate(filho)
return the best individual	return melhor_geral, melhor_fit_geral, historico

B. Reproduce

No código, o método *reproduce* adapta a lógica de corte do pseudocódigo para o contexto de permutações, onde não pode haver cidades repetidas.

Embora o pseudocódigo possua uma concatenação de *substrings* (*APPEND*), para o nosso problema, isso resultaria em cidades repetidas. De modo que, a implementação foi alterada para realizar *Order Crossover*, onde a *substring* do primeiro pai é preservada, e os genes restantes são extraídos do segundo pai, mas filtrando aqueles já presentes. Logo, isso mantém a herança genética de ambos os pais sem quebrar a restrição de permutação única.

Pseudocódigo	**Implementação**
c ← random number 1 to n	c = random.randint(1, n - 1)
SUBSTRING(parent1, 1, c)	parent1[:c]
APPEND(..., SUBSTRING(parent2...))	O loop for cidade in parent2 com verificação if not in set_filho.

C. Mutate

A mutação implementada segue o conceito de aleatorização do pseudocódigo, utilizando a inversão de segmento:

```
if random.random() < self.prob_mutacao:
            # seleciona dois pontos e inverte o trecho entre eles
            i, j = sorted(random.sample(range(len(filho)), 2))
            # inverte o segmento
            filho[i:j+1] = filho[i:j+1][::-1]
        return filho
```

5. Detalhes

1. Fitness: calculado no método fitness, onde a aptidão é a soma das distâncias na matriz_distancias. Nesse panorama, como no TSP o objetivo é minimizar, o peso para seleção é o inverso do fitness (1.0 / (f + 1e-10)).
2. Elitismo: foi adicionado o parâmetro elite_size, que garante que as melhores rotas de uma geração passem direto para a próxima, acelerando a convergência.
3. Representação: cada indivíduo é uma lista de inteiros representando os índices das cidades.

6. Conclusão

No geral, o Algoritmo Genético demonstrou ser eficiente para resolver problemas de rota, demonstrando que pode muitas vezes ser mais viável do que uma simples força bruta. Somado a ajustes finos como o elitismo e o cálculo de *fitness* inversamente proporcional à distância, podemos consolidar o AG como uma ferramenta prática e robusta, capaz de fugir de ótimos locais e convergir de forma consistente para rotas altamente otimizadas em um tempo de processamento aceitável.