Disciplina: Inteligência Artificial

Professor: Gustavo Siqueira Vinhal Ano: 2024/2

Aluno: Felipe Camilo Alves Matrícula: 2019.1.0028.0017-8

Trabalho 2 – Algoritmos Genéticos

Contexto:

O Problema do Caixeiro Viajante (Traveling Salesman Problem - TSP) é um dos problemas clássicos da computação e da teoria dos grafos. Consiste em encontrar o menor percurso possível que permita a um caixeiro viajante passar por um conjunto de cidades exatamente uma vez cada e retornar à cidade de origem. Este problema é conhecido por ser NP-difícil, o que significa que não existe um algoritmo eficiente conhecido para resolvê-lo exatamente em tempo polinomial para um grande número de cidades.

Algoritmos genéticos são métodos de busca heurísticos inspirados no processo de seleção natural da evolução biológica. Eles são particularmente úteis para resolver problemas de otimização complexos, como o TSP, onde métodos exatos são inviáveis devido ao alto custo computacional.

Objetivo:

Desenvolver um algoritmo genético para encontrar uma solução aproximada para o Problema do Caixeiro Viajante, utilizando uma matriz de distâncias entre 100 cidades fornecida em um arquivo CSV. O algoritmo deve implementar os principais componentes de um algoritmo genético:

- Representação dos cromossomos (percurso do caixeiro).
- Função de avaliação (fitness), calculando o custo total do percurso.
- Operadores genéticos de seleção, crossover e mutação.
- Mecanismo de criação de novas gerações.
- Opcionalmente, implementar elitismo para preservar os melhores indivíduos.

Método

A implementação foi realizada em Python, utilizando bibliotecas como **numpy**, **random**, **matplotlib** e **pandas**. O algoritmo genético segue os passos padrão:

1. **Inicialização**: Geração de uma população inicial de cromossomos (permutação de cidades).
2. **Avaliação**: Cálculo do fitness de cada indivíduo (custo total do percurso).
3. **Seleção**: Escolha de indivíduos para reprodução com base no fitness.
4. **Crossover**: Combinação de pares de indivíduos para gerar descendentes.
5. **Mutação**: Aplicação de pequenas alterações nos descendentes.
6. **Substituição**: Criação de uma nova geração a partir dos descendentes.
7. **Iteração**: Repetição dos passos 2 a 6 por um número definido de gerações.

Detalhamento do Código

A seguir, apresentamos cada parte do código, explicando sua função e como contribui para a solução do problema.

Importação das Bibliotecas

```
import numpy as np
import random
import matplotlib.pyplot as plt
import pandas as pd
```

- **numpy**: Biblioteca para operações numéricas eficientes com arrays.
- **random**: Módulo para geração de números aleatórios.
- **matplotlib.pyplot:** Biblioteca para geração de gráficos.
- **pandas**: Biblioteca para manipulação de dados tabulares.

Função para Carregar a Matriz de Distâncias

```
def gera_matriz_distancias(filepath):
    df = pd.read_csv(filepath, header=None, dtype=float)
    distancias = df.to_numpy()
    np.fill_diagonal(distancias, np.inf)
    return distancias
```

Objetivo: Carregar a matriz de distâncias entre as cidades a partir de um arquivo CSV.

Descrição:

- Lê o arquivo CSV usando pandas.read_csv.
- Converte o DataFrame em um array NumPy.

- Preenche a diagonal principal com infinito (np.inf) para evitar que o algoritmo considere deslocamentos de uma cidade para ela mesma.

Uso: Fornece a matriz de distâncias que será utilizada em todo o algoritmo.

Função para Gerar a População Inicial

```
def gera_populacao_inicial(tamanho_pop, num_cidades):
    return [list(np.random.permutation(num_cidades)) for _ in range(tamanho_pop)]
```

Objetivo: Criar a população inicial de cromossomos.

Descrição:

- Para cada indivíduo na população, gera uma permutação aleatória das cidades usando np.random.permutation.
- Converte cada permutação em uma lista.

Uso: Fornece uma diversidade inicial de soluções para o algoritmo.

Função para Calcular o Fitness

```
def calcula_fitness(cromossomo, distancias):
    return sum(distancias[cromossomo[i], cromossomo[(i + 1) % len(cromossomo)]] for i in range(len(cromossomo)))
```

Objetivo: Avaliar a qualidade de um cromossomo.

Descrição:

- Calcula o custo total do percurso representado pelo cromossomo, somando as distâncias entre cidades adjacentes e retornando à cidade inicial.
- Utiliza o operador módulo % para garantir que o percurso seja fechado (cíclico).

Uso: Orienta o processo evolutivo, favorecendo indivíduos com menor custo total.

Funções de Seleção de Pais

Seleção por Roleta

```python
def seleciona_pais_roleta(populacao, fitnesses):
    total_fitness = sum(fitnesses)
    probabilidades = [f / total_fitness for f in fitnesses]
    pais = random.choices(populacao, weights=probabilidades, k=2)
    return pais[0], pais[1]
```

Objetivo: Selecionar pais para a reprodução com base no fitness.

Descrição:

- Calcula a probabilidade de seleção de cada indivíduo proporcional ao seu fitness.
- Usa random.choices para selecionar dois pais com base nas probabilidades.

Uso: Implementa o método de seleção por roleta, favorecendo indivíduos com menor custo (já que o fitness é o custo total).

Seleção por Torneio

```python
def seleciona_pais_torneio(populacao, fitnesses, tamanho_torneio):
    pais = []
    for _ in range(2):
        competidores = random.sample(list(zip(populacao, fitnesses)), tamanho_torneio)
        vencedor = min(competidores, key=lambda x: x[1])
        pais.append(vencedor[0])
    return pais[0], pais[1]
```

Objetivo: Selecionar pais usando torneios.

Descrição:

- Realiza um torneio entre um subconjunto aleatório de indivíduos.
- Seleciona o indivíduo com menor custo (melhor fitness) como vencedor.

Uso: Oferece uma alternativa à seleção por roleta, controlando a pressão seletiva através do tamanho do torneio.

Função Genérica de Seleção

```python
def seleciona_pais(populacao, fitnesses, metodo_selecao, **kwargs):
    return metodo_selecao(populacao, fitnesses, **kwargs)
```

Objetivo: Permitir a escolha do método de seleção.

Descrição:

- Recebe como parâmetro o método de seleção a ser utilizado (roleta ou torneio).
- Passa os argumentos necessários para o método selecionado.

Uso: Flexibiliza o algoritmo, permitindo experimentar diferentes estratégias de seleção.

Função de Crossover Ordenado

```python
def crossover_ordenado(pai1, pai2):
    tamanho = len(pai1)
    filho1, filho2 = [None]*tamanho, [None]*tamanho

    ponto1, ponto2 = sorted(random.sample(range(tamanho), 2))

    meio_pai1 = pai1[ponto1:ponto2 + 1]
    filho1[ponto1:ponto2 + 1] = meio_pai1

    pos_filho = (ponto2 + 1) % tamanho
    for elemento in pai2:
        if elemento not in meio_pai1:
            filho1[pos_filho] = elemento
            pos_filho = (pos_filho + 1) % tamanho

    meio_pai2 = pai2[ponto1:ponto2 + 1]
    filho2[ponto1:ponto2 + 1] = meio_pai2

    pos_filho = (ponto2 + 1) % tamanho
    for elemento in pai1:
        if elemento not in meio_pai2:
            filho2[pos_filho] = elemento
            pos_filho = (pos_filho + 1) % tamanho

    return filho1, filho2
```

Objetivo: Combinar pais para gerar filhos, preservando a ordem das cidades.

Descrição:

- Seleciona dois pontos de corte aleatórios.
- Copia o segmento entre os pontos do pai para o filho.
- Preenche as posições restantes com as cidades do outro pai, respeitando a ordem e evitando duplicatas.

Uso: Garante que os filhos sejam percursos válidos, sem cidades repetidas.

Função de Crossover

```python
def crossover(pai1, pai2, distancias):
    if pai1 == pai2:
        return pai1.copy()
    filho1, filho2 = crossover_ordenado(pai1, pai2)
    fit_filho1 = calcula_fitness(filho1, distancias)
    fit_filho2 = calcula_fitness(filho2, distancias)
    return filho1 if fit_filho1 < fit_filho2 else filho2
```

Objetivo: Decidir qual dos filhos gerados será inserido na nova geração.

Descrição:

- Evita cruzar pais idênticos.
- Calcula o fitness de ambos os filhos.
- Retorna o filho com melhor fitness.

Uso: Introduz novos indivíduos na população, potencialmente com melhor desempenho.

Função de Mutação

```python
def mutacao(cromossomo, probabilidade_mutacao):
    if random.random() < probabilidade_mutacao:
        tipo_mutacao = random.choice(['swap', 'inversion', 'insertion'])
        if tipo_mutacao == 'swap':
            idx1, idx2 = random.sample(range(len(cromossomo)), 2)
            cromossomo[idx1], cromossomo[idx2] = cromossomo[idx2], cromossomo[idx1]
        elif tipo_mutacao == 'inversion':
            idx1, idx2 = sorted(random.sample(range(len(cromossomo)), 2))
            cromossomo[idx1:idx2+1] = cromossomo[idx1:idx2+1][::-1]
        elif tipo_mutacao == 'insertion':
            idx1, idx2 = random.sample(range(len(cromossomo)), 2)
            elemento = cromossomo.pop(idx1)
            cromossomo.insert(idx2, elemento)
    return cromossomo
```

Objetivo: Introduzir variação na população para evitar convergência prematura.

Descrição:

- Aplica mutação com uma certa probabilidade.
- Implementa três tipos de mutação:
- Swap: Troca a posição de duas cidades.
- Inversion: Inverte a ordem de um segmento do cromossomo.
- Insertion: Remove uma cidade e a insere em outra posição.

Uso: Mantém a diversidade genética na população.

Função para Criar Nova Geração

```python
def cria_nova_geracao(populacao, fitnesses, probabilidade_mutacao, distancias, elitismo=False):
    nova_populacao = []
    tamanho_pop = len(populacao)
    if elitismo:
        melhor_idx = np.argmin(fitnesses)
        melhor_individuo = populacao[melhor_idx].copy()
        nova_populacao.append(melhor_individuo)
    while len(nova_populacao) < tamanho_pop:
        pai1, pai2 = seleciona_pais(populacao, fitnesses, seleciona_pais_torneio, tamanho_torneio=3)
        filho = crossover(pai1, pai2, distancias)
        filho = mutacao(filho, probabilidade_mutacao)
        nova_populacao.append(filho)
    return nova_populacao
```

Objetivo: Gerar uma nova população de indivíduos.

Descrição:

- Se o elitismo estiver ativado, mantém o melhor indivíduo da geração atual.
- Continua gerando novos indivíduos até atingir o tamanho da população.
- Utiliza seleção, crossover e mutação para criar novos indivíduos.

Uso: Evolui a população, potencialmente melhorando a qualidade média das soluções.

Função Principal do Algoritmo Genético

```python
def algoritmo_genetico(tamanho_pop, num_geracoes, probabilidade_mutacao, distancias, num_cidades, elitismo=False):
    populacao = gera_populacao_inicial(tamanho_pop, num_cidades)
    melhor_fitness_por_geracao = []
    melhor_cromossomo = None
    melhor_fitness = float('inf')

    for geracao in range(num_geracoes):
        fitnesses = [calcula_fitness(ind, distancias) for ind in populacao]
        min_fitness = min(fitnesses)
        melhor_fitness_por_geracao.append(min_fitness)

        if min_fitness < melhor_fitness:
            melhor_fitness = min_fitness
            melhor_cromossomo = populacao[fitnesses.index(min_fitness)].copy()

        populacao = cria_nova_geracao(populacao, fitnesses, probabilidade_mutacao, distancias, elitismo)

    melhor_cromossomo = [int(cidade) for cidade in melhor_cromossomo]

    plt.figure(figsize=(10, 5))
    plt.plot(melhor_fitness_por_geracao, label='Melhor Fitness por Geração')
    plt.xlabel('Geração')
    plt.ylabel('Fitness')
    plt.title('Progresso do Fitness ao Longo das Gerações')
    plt.legend()
    plt.grid(True)
    plt.show()

    return melhor_cromossomo, melhor_fitness
```

Objetivo: Coordenar a execução do algoritmo genético.

Descrição:

- Inicializa a população e variáveis para rastrear o melhor indivíduo.
- Em cada geração:
    - Calcula o fitness de cada indivíduo.
    - Atualiza o melhor fitness e cromossomo se necessário.
    - Gera uma nova população.
- Após as gerações, converte o melhor cromossomo para inteiros padrão.
- Plota o progresso do fitness ao longo das gerações.

Uso: Controla o fluxo do algoritmo, gerenciando a evolução da população.

Bloco Principal (Main)

```python
if __name__ == "__main__":
    filepath = r'distancias_entre_100_cidades.csv'
    distancias = gera_matriz_distancias(filepath)
    num_cidades = len(distancias)
    tamanho_pop = 100
    num_geracoes = 1000
    probabilidade_mutacao = 0.01
    elitismo = False

    # Opcional: definir sementes para reprodução de resultados
    # np.random.seed(42)
    # random.seed(42)

    melhor_cromossomo, melhor_fitness = algoritmo_genetico(
        tamanho_pop, num_geracoes, probabilidade_mutacao, distancias, num_cidades, elitismo
    )
    print("Melhor percurso:", melhor_cromossomo)
    print("Custo do melhor percurso:", melhor_fitness)
```

Objetivo: Definir os parâmetros do algoritmo e executar a função principal.

Descrição:

- Especifica o caminho para o arquivo de distâncias.
- Define os parâmetros do algoritmo (tamanho da população, número de gerações, probabilidade de mutação e elitismo).
- Chama a função algoritmo_genetico e captura o melhor percurso e seu custo.
- Imprime os resultados.

Uso: Ponto de entrada do programa, permitindo a execução autônoma do script.

Resultados

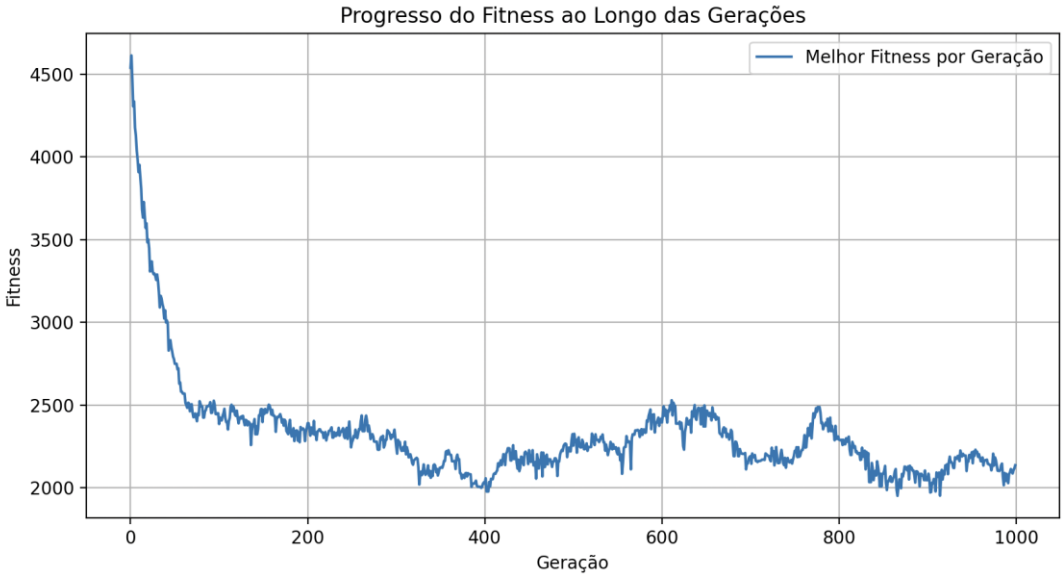
Melhor Percurso e Custo

Após a execução do algoritmo genético, o melhor percurso encontrado e seu custo total são exibidos:

```
Melhor percurso: [18, 37, 99, 2, 33, 7, 48, 52, 54, 11, 81, 76, 87, 22, 26, 64, 13, 55, 3, 15, 38, 57, 63, 83, 28, 46, 60, 53, 96, 43, 29, 50, 9, 41, 45, 19,
 5, 40, 91, 49, 44, 74, 39, 32, 65, 61, 6, 92, 31, 25, 14, 24, 94, 30, 80, 20, 78, 16, 67, 35, 71, 90, 77, 51, 84, 12, 66, 34, 69, 62, 56, 10, 27, 82, 85, 58
, 95, 23, 72, 93, 4, 79, 59, 36, 75, 98, 97, 70, 17, 42, 8, 86, 47, 0, 68, 1, 88, 89, 73, 21]
Custo do melhor percurso: 1952.0
```

Gráfico do Progresso do Fitness

O algoritmo gera um gráfico mostrando a evolução do melhor fitness ao longo das gerações, permitindo visualizar a convergência do algoritmo.



Anexos

Em anexo se encontra a base de dados e todo o código usado para rodar esse exemplo acima.