Instituto Politécnico Nacional
Escuela Superior de Cómputo
Materia: GENETIC ALGORITHMS
Lab Session 5. GA's for Combinatorial Optimization
Alumno: Torres Abonce Luis Miguel
Profesor: Jorge Luis Rosas Trigueros
Fecha de Realización: 20 Octubre 2024
Fecha de Entrega: 22 Octubre 2024

## Marco Teórico.

El Problema del Viajante (TSP, por sus siglas en inglés: Travelling Salesman Problem) es uno de los problemas clásicos de optimización combinatoria y tiene aplicaciones en diferentes áreas de la ingeniería y la investigación operativa. El objetivo del TSP es encontrar la ruta más corta que permita a un viajante visitar cada ciudad de una lista una sola vez y regresar al punto de partida. La dificultad radica en la enorme cantidad de posibles rutas, que crece exponencialmente con el número de ciudades involucradas. En consecuencia, resolver este problema de manera eficiente es muy desafiante, especialmente cuando se trata de un número grande de ciudades.

Dado el carácter complejo del TSP, se suelen emplear técnicas metaheurísticas, como los Algoritmos Genéticos (GA), para obtener una solución lo más cercana posible al óptimo en un tiempo razonable. Los Algoritmos Genéticos son una técnica de búsqueda inspirada en el proceso de selección natural propuesto por Charles Darwin, donde una población de posibles soluciones evoluciona hacia mejores soluciones mediante operadores como selección, cruce y mutación. En este contexto, la solución se representa como un conjunto de individuos que representan diferentes rutas posibles.

En un GA, la selección es un paso clave donde se eligen los individuos más aptos para generar una nueva generación. En el código proporcionado, se utiliza la selección por torneo, donde dos individuos son seleccionados al azar y el mejor de los dos es el que pasa a la siguiente fase. Este método asegura que las mejores soluciones tengan más probabilidades de sobrevivir y pasar sus características a la siguiente generación.

Otro operador fundamental es el cruce (o crossover), que permite combinar dos soluciones existentes para crear una nueva. En este caso se utiliza el "Order Crossover" (OX), una técnica que permite mantener una porción de las ciudades del padre y asegurar que el hijo resultante sea una permutación válida de las ciudades. Además, la mutación se emplea para evitar la convergencia prematura a una solución subóptima al introducir variabilidad en la población, lo cual se logra intercambiando dos ciudades al azar en una ruta.

La implementación de este algoritmo genético busca minimizar la distancia total de la ruta recorrida. La función de evaluación (fitness) calcula esta distancia, lo cual permite comparar distintas soluciones e identificar cuáles son las mejores. A lo largo de varias generaciones, el objetivo es reducir el valor de la función de evaluación, acercándose cada vez más a la solución óptima del problema.

Cabe destacar que los Algoritmos Genéticos no garantizan encontrar la solución óptima en todos los casos, pero sí pueden proporcionar una solución lo suficientemente buena en un tiempo limitado, lo cual los hace muy útiles en problemas como el TSP donde la búsqueda exhaustiva es computacionalmente inviable. Estos algoritmos son ampliamente utilizados en diversas áreas como logística, planificación de rutas, diseño de circuitos y problemas de asignación.

## Material y Equipo

- **Hardware:** Computadora con capacidad para ejecutar Python.
- **Software:**
  - Python 3.x
  - Google Colab.

## Desarrollo de la practica

En el desarrollo de este laboratorio se implementó un Algoritmo Genético (GA) para resolver el Problema del Viajante (TSP) para 10 ciudades. A continuación, se describen paso a paso las actividades realizadas, las dificultades encontradas y los resultados obtenidos:

1. **Definición del Problema y Preparación del Entorno**

En primer lugar, se definió el problema del TSP con 10 ciudades. Para representar las distancias entre las ciudades, se utilizó una matriz que asigna un valor de distancia entre cada par de ciudades. La matriz de distancias se definió de tal forma que ciertas conexiones tienen una distancia más corta (valor de 1) mientras que la mayoría tiene una distancia predeterminada de 2. Esto se logró mediante el uso de la librería numpy en Python.

2. **Inicialización de la Población**

Se generó una población inicial de posibles soluciones, donde cada solución representa un recorrido diferente por las 10 ciudades. Se utilizó una función para generar recorridos aleatorios (tours), asegurándose de que cada ciudad aparezca una sola vez en cada recorrido. La dificultad aquí fue garantizar la aleatoriedad sin generar rutas repetidas en la población. Para resolverlo, se implementaron listas de verificación para validar la unicidad de cada recorrido generado.

3. **Evaluación de la Población**

La función de evaluación (fitness) fue implementada para calcular la distancia total de cada recorrido en la población. Se recorrieron las ciudades de cada tour y se sumaron las distancias entre cada par de ciudades consecutivas. Al principio, se encontraron errores en el cálculo de la distancia total debido a un mal manejo de los índices en la matriz de distancias. Estos errores se resolvieron revisando cuidadosamente la indexación y añadiendo pruebas para verificar el comportamiento esperado.

4. **Selección, Cruce y Mutación**

Para la selección de padres, se utilizó el método de selección por torneo. Dos individuos de la población fueron seleccionados al azar y el mejor de los dos (el de menor distancia total) fue elegido como padre. Luego, se realizó el cruce entre padres utilizando el operador Order Crossover (OX). Este paso presentó dificultades en cuanto a la implementación correcta del operador de cruce, ya que era necesario asegurarse de que las ciudades no se repitieran en el hijo resultante. Tras varios intentos y revisiones del algoritmo, se logró implementar el cruce de manera exitosa.

La mutación fue implementada mediante el intercambio aleatorio de dos ciudades en un recorrido, lo cual permitió mantener la diversidad en la población y evitar la convergencia prematura. Se encontró que una tasa de mutación demasiado alta generaba recorridos subóptimos, mientras que una tasa demasiado baja provocaba estancamiento en la mejora de las soluciones. Finalmente, se determinó que una tasa de mutación del 10% era adecuada para este problema.

5. **Evolución de la Población**

A lo largo de 500 generaciones, se fue reemplazando la población con nuevas soluciones

generadas mediante selección, cruce y mutación. Cada 100 generaciones se imprimía la mejor solución encontrada hasta el momento, lo cual permitió monitorear el progreso del algoritmo. Inicialmente, las distancias eran bastante elevadas, pero con el tiempo se observaron mejoras significativas. La principal dificultad en este paso fue la convergencia prematura, la cual se mitigó mediante ajustes en la tasa de mutación y aumentando la diversidad de la población inicial.

6. **Resultados Obtenidos**

Al finalizar las 500 generaciones, el algoritmo encontró una solución con una distancia total de 13, lo cual es cercano al óptimo teórico para el problema planteado. El mejor recorrido obtenido fue uno que aprovechaba las conexiones de menor distancia definidas en la matriz. Este resultado demuestra la efectividad de los Algoritmos Genéticos para aproximarse a la solución de problemas complejos como el TSP, aunque se debe tener en cuenta que el resultado puede variar debido a la naturaleza estocástica del algoritmo.

## Diagramas, gráficas y pantallas.

**Figura 1**. Se puede observar que la mejor distancia se estabilizó en 11.0 a partir de la generación 200, manteniéndose constante hasta el final del proceso. Acercándose a la solución optima

```
Generación 0: Mejor distancia = 16.0, Mejor tour = [3, 5, 7, 4, 6, 9, 8, 10, 1, 2]
Generación 100: Mejor distancia = 13.0, Mejor tour = [9, 2, 4, 1, 6, 8, 10, 3, 5, 7]
Generación 200: Mejor distancia = 11.0, Mejor tour = [9, 2, 4, 6, 8, 10, 1, 3, 5, 7]
Generación 300: Mejor distancia = 11.0, Mejor tour = [9, 2, 4, 6, 8, 10, 1, 3, 5, 7]
Generación 400: Mejor distancia = 11.0, Mejor tour = [9, 2, 4, 6, 8, 10, 1, 3, 5, 7]

Mejor solución encontrada: Distancia = 11.0, Tour = [9, 2, 4, 6, 8, 10, 1, 3, 5, 7]
```

**Figura 1.**

## Conclusiones

El uso de GA para resolver el Problema del Viajante es una herramienta efectiva para encontrar soluciones cercanas al óptimo en problemas de gran complejidad. A lo largo del desarrollo de esta práctica, observe cómo la selección, el cruce y la mutación contribuyeron a mejorar la calidad de las soluciones generación tras generación. Aunque el algoritmo no garantiza siempre encontrar la mejor solución posible, logró encontrar una ruta con una distancia total de 11.0, lo cual es una aproximación muy buena al problema planteado. Las dificultades encontradas, como la convergencia prematura y la correcta implementación de los operadores genéticos, fueron resueltas mediante ajustes en los parámetros.

**Gutin, G., & Punnen, A. P. (2002). The Traveling Salesman Problem and Its Variations.**

- **Autor:** Springer Science & Business Media.
- **Fuente:** https://www.cs.us.es/~fsancho/Blog/posts/ACO_TSP.md

## Anexo:

Código:

```
import numpy as np
import random

# Matriz de distancias entre ciudades
M = 2 * np.ones([11, 11])
M[1][3] = 1; M[3][5] = 1; M[5][7] = 1; M[7][9] = 1
M[9][2] = 1; M[2][4] = 1; M[4][6] = 1; M[6][8] = 1
M[8][10] = 1

# Definir parámetros del algoritmo genético
NUM_CITIES = 10
POPULATION_SIZE = 100
NUM_GENERATIONS = 500
MUTATION_RATE = 0.1

# Función de evaluación: Calcula la distancia total de una ruta
def fitness(tour):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += M[tour[i]][tour[i + 1]]
    total_distance += M[tour[-1]][tour[0]]  # Volver a la ciudad inicial
    return total_distance

# Genera una población inicial aleatoria
def initialize_population():
    population = []
    for _ in range(POPULATION_SIZE):
        tour = list(range(1, NUM_CITIES + 1))
        random.shuffle(tour)
        population.append(tour)
    return population

# Selección por torneo: Selecciona el mejor entre dos soluciones aleatorias
def tournament_selection(population):
    tournament = random.sample(population, 2)
    fittest = min(tournament, key=fitness)
    return fittest

# Cruce por orden (Order Crossover - OX)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM_CITIES), 2))
```

```
    child = [None] * NUM_CITIES
    child[start:end] = parent1[start:end]

    ptr = 0
    for city in parent2:
        if city not in child:
            while child[ptr] is not None:
                ptr += 1
            child[ptr] = city

    return child

# Mutación: Intercambia dos ciudades al azar
def mutate(tour):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
        tour[i], tour[j] = tour[j], tour[i]
    return tour

# Reemplazo: Reemplaza la población actual con la nueva generación
def replace_population(population, new_population):
    population[:] = new_population

# Función principal para el algoritmo genético
def genetic_algorithm():
    population = initialize_population()

    for generation in range(NUM_GENERATIONS):
        new_population = []
        for _ in range(POPULATION_SIZE):
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        # Reemplazar la población con la nueva generación
        replace_population(population, new_population)

        # Mostrar la mejor solución cada 100 generaciones
        if generation % 100 == 0:
            best_tour = min(population, key=fitness)
            best_fitness = fitness(best_tour)
```

```
            print(f"Generación {generation}: Mejor distancia = {best_fitness}, Mejor tour =
{best_tour}")

    # Resultado final
    best_tour = min(population, key=fitness)
    best_fitness = fitness(best_tour)
    print(f"\nMejor solución encontrada: Distancia = {best_fitness}, Tour = {best_tour}")
    return best_tour, best_fitness

# Ejecutar el algoritmo genético
best_tour, best_fitness = genetic_algorithm()
```