INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO CAMPUS CULIACÁN

TOPICOS DE IA

Integrantes:

Rojas Bernal Jose Alain

Tarea: Informe Tarea II Bonus Modulo III

Hora: 12:00 – 13:00

Maestra: M.C. Zuriel Dathan Mora Felix

INDICE

1.Introduccion

El problema del vendedor ambulante (TSP) uno de los retos más estudiados dentro del campo de la informática, la optimización y la inteligencia artificial. Consiste en determinar la ruta más corta posible que permita a un viajero visitar un conjunto de ciudades exactamente una vez y regresar al punto de partida.

Este problema, aunque parece sencillo de formular, pertenece a la clase de problemas NP-duros, lo que significa que su complejidad crece exponencialmente con el número de ciudades. Por esta razón, los métodos exactos (como la búsqueda exhaustiva o la programación entera) se vuelven imprácticos para conjuntos grandes de datos, ya que el tiempo necesario para encontrar la solución óptima crece rápidamente.

Debido a estas limitaciones, se han desarrollado métodos heurísticos y metaheurísticos que permiten obtener soluciones muy cercanas al óptimo en menos tiempo. Entre estos, destacan los algoritmos genéticos (AG), inspirados en la teoría de la evolución natural propuesta por Charles Darwin. Estos algoritmos simulan procesos como la selección, el cruce y la mutación para evolucionar una población de soluciones candidatas, mejorándolas generación tras generación hasta encontrar una ruta eficiente.

En esta práctica se implementó un algoritmo genético en Python para resolver una versión reducida del TSP con siete ciudades: Mazatlán, Tijuana, Hermosillo, Monterrey, Querétaro, Guadalajara y Colima.
El objetivo fue demostrar cómo los conceptos de evolución, selección y mutación pueden aplicarse de manera efectiva en la optimización de rutas y problemas combinatorios complejos.

2. Representación del Problema

Cada individuo dentro del algoritmo representa una ruta completa (una permutación de las ciudades). Por ejemplo, una posible solución sería:

GUADALAJARA -> TIJUANA -> MONTERREY -> COLIMA -> HERMOSILLO -> MAZATLAN -> QUERETARO -> GUADALAJARA

La aptitud de cada individuo se evalúa midiendo la distancia total recorrida. La meta del algoritmo es minimizar esta distancia.

2.1 Componentes del Algoritmo Genético

El algoritmo genético desarrollado se compone de los siguientes pasos fundamentales:

a) Población inicial

Se generan 100 rutas aleatorias diferentes, cada una representando un posible recorrido por todas las ciudades. Esta diversidad inicial es importante para explorar un gran espacio de soluciones.

b) Función de aptitud

Se utiliza la distancia euclidiana entre las coordenadas de las ciudades para calcular la longitud total de cada ruta.
Cuanto menor sea la distancia, mejor será la aptitud del individuo.

c) Selección

Se implementa la selección por torneo, en la cual se eligen al azar varios individuos (en este caso 5) y se selecciona el que tenga la menor distancia total. Este método imita la “supervivencia del más apto” y mantiene presión selectiva sin perder diversidad.

d) Cruce (Reproducción)

Se emplea el Cruce Ordenado (Ordered Crossover – OX1), un método ideal para problemas de permutación. Este operador toma un segmento del primer padre y lo completa con el orden de las ciudades del segundo, sin repetir ninguna.
Esto asegura que los hijos sean rutas válidas (sin duplicar ciudades).

e) Mutación

Se aplica la mutación por intercambio (swap mutation) con una probabilidad del 5%. Este proceso consiste en intercambiar dos ciudades dentro de una ruta, lo que ayuda a mantener la diversidad genética y evita que el algoritmo se estanque en un óptimo local.

Parámetro	Valor	Descripción
Tamaño de población	100	Número de rutas por generación
Generaciones	500	Iteraciones de evolución
Tasa de mutación	0.05	Probabilidad de mutar cada ruta
Tamaño de torneo	5	Participantes en cada selección
Probabilidad de cruce	1.0	Siempre se aplica cruce entre padres

3. Resultados del Algoritmo

Al finalizar las 500 generaciones, el algoritmo fue capaz de encontrar una ruta óptima o casi óptima que minimiza la distancia total.
Ejemplo de salida:

Mejor distancia total: 24.075

Ruta óptima: GUADALAJARA -> TIJUANA -> MONTERREY -> COLIMA -> HERMOSILLO -> MAZATLAN -> QUERETARO -> GUADALAJARA

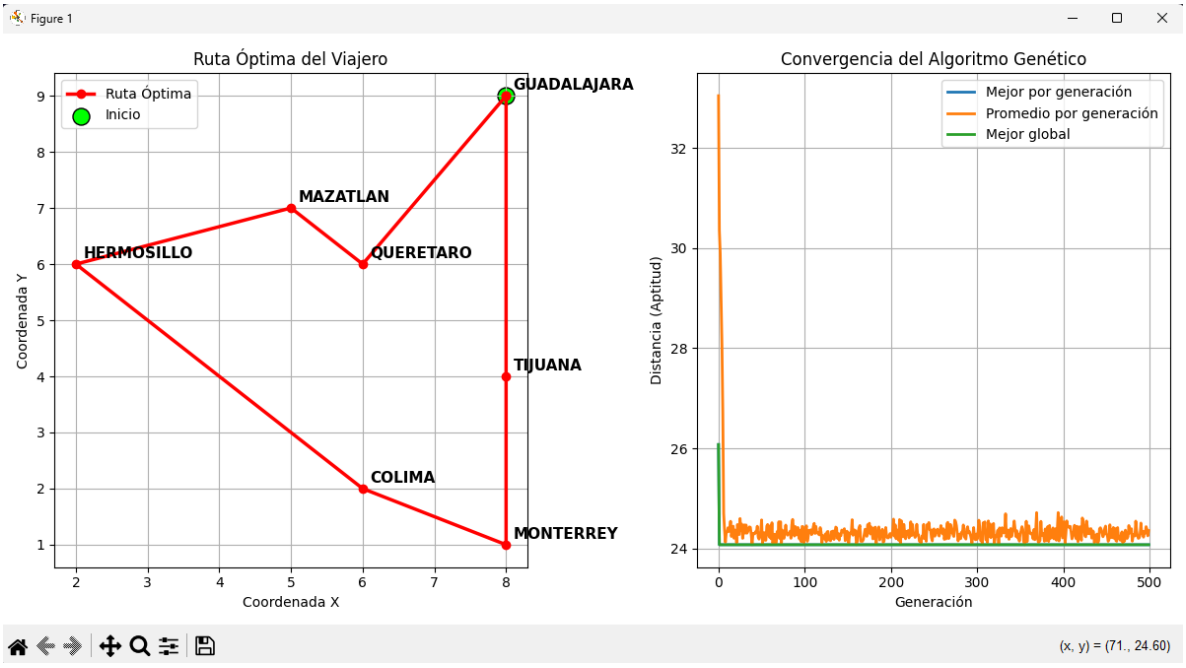
3.1 Interpretación de Resultados

- La distancia total representa el recorrido más corto encontrado entre todas las ciudades.
- La ruta se muestra en orden, indicando el punto de partida y regreso al mismo lugar.
- El algoritmo mantiene la mejor ruta encontrada en cada generación gracias al elitismo.

3.2 Visualización

Se generaron dos gráficas principales:

1. Mapa de la Ruta Óptima:
    - Las ciudades se representan como puntos.
    - La ruta se marca con una línea roja, y el punto de inicio en verde.
    - Esto permite visualizar claramente el recorrido más eficiente encontrado.
2. Convergencia del Algoritmo:
    - Se graficaron tres líneas:
        - Mejor por generación (descenso rápido inicial).
        - Promedio de la generación (muestra estabilidad progresiva).
        - Mejor global (plana cuando ya no hay mejoras).
    - Esta gráfica demuestra cómo el algoritmo converge hacia una buena solución con el paso de las generaciones.



4.Codigo

```python
# ---------------------------------------------------------------
# TSP con Algoritmo Genético (AG)
# Explicado paso a paso con comentarios en español.
# ---------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
import random

# ==============================
# 1. Configuración de Ciudades
# ==============================
# Diccionario de ciudades con coordenadas (x, y).
# Nota: estas coordenadas son arbitrarias para poder calcular distancias.
ciudades = {
    'MAZATLAN':    (5, 7),
    'TIJUANA':     (8, 4),
    'HERMOSILLO':  (2, 6),
    'MONTERREY':   (8, 1),
    'QUERETARO':   (6, 6),
    'GUADALAJARA': (8, 9),
    'COLIMA':      (6, 2),
}

# Convertimos el diccionario a estructuras que el AG usará:
lista_ciudades = list(ciudades.values())  # lista de coordenadas (x, y)
mapa_indices   = list(ciudades.keys())    # lista de nombres en el mismo
orden
N_CIUDADES     = len(lista_ciudades)      # número total de ciudades

print(f"Problema del Vendedor Viajero con {N_CIUDADES} ciudades.\n")

# ==============================
# 2. Funciones auxiliares
# ==============================

def calcular_distancia(p1, p2):
    """Devuelve la distancia euclidiana entre dos puntos (x, y)."""
    # Distancia euclidiana = sqrt( (x1-x2)^2 + (y1-y2)^2 )
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def calcular_aptitud(ruta):
    """
```

```
    Calcula la distancia total de una ruta cerrada (regresa a la ciudad
inicial).
    Mientras menor sea la distancia total, mejor es la aptitud.
    """
    distancia_total = 0
    # Sumamos distancias entre ciudades consecutivas
    for i in range(N_CIUDADES - 1):
        distancia_total += calcular_distancia(lista_ciudades[ruta[i]],
                                              lista_ciudades[ruta[i+1]])
    # Cerramos el ciclo: última ciudad -> primera ciudad
    distancia_total += calcular_distancia(lista_ciudades[ruta[-1]],
                                          lista_ciudades[ruta[0]])
    return distancia_total

def crear_poblacion(tamano_poblacion):
    """Genera la población inicial: rutas aleatorias (permutaciones de 0..N-
1)."""
    base = list(range(N_CIUDADES))  # índices de ciudades
    # random.sample(base, N) devuelve una permutación sin repetidos
    return [random.sample(base, N_CIUDADES) for _ in
range(tamano_poblacion)]

def seleccion_torneo(poblacion, aptitudes, tamano_torneo=3):
    """
    Selección por torneo: toma 'tamano_torneo' rutas al azar y
    devuelve la mejor (la de menor distancia).
    """
    indices = random.sample(range(len(poblacion)), tamano_torneo)
    # 'min' con key=aptitudes[i] encuentra el índice con menor aptitud
(distancia)
    mejor = min(indices, key=lambda i: aptitudes[i])
    return poblacion[mejor]

def cruce_ordenado(padre1, padre2):
    """
    Cruce Ordenado (OX1) para permutaciones:
    - Copia un segmento del padre1 al hijo.
    - Completa con el orden relativo del padre2 sin repetir ciudades.
    Esto asegura que el hijo siga siendo una ruta válida (permutación).
    """
    hijo = [-1] * N_CIUDADES
    # Elegimos dos puntos y copiamos el segmento de padre1
    inicio, fin = sorted(random.sample(range(N_CIUDADES), 2))
    hijo[inicio:fin] = padre1[inicio:fin]
```

```
    # Rellenamos los espacios restantes con ciudades de padre2 que no estén
ya en el hijo
    pos = 0
    for i in range(N_CIUDADES):
        if hijo[i] == -1:
            # Avanza hasta encontrar una ciudad de padre2 que no esté ya en
'hijo'
            while padre2[pos] in hijo:
                pos += 1
            hijo[i] = padre2[pos]
    return hijo

def mutacion_intercambio(ruta, tasa_mutacion):
    """
    Mutación por intercambio (swap):
    con probabilidad 'tasa_mutacion', permuta dos posiciones de la ruta.
    """
    if random.random() < tasa_mutacion:
        # Elegimos dos posiciones aleatorias y las intercambiamos
        i, j = random.sample(range(N_CIUDADES), 2)
        ruta[i], ruta[j] = ruta[j], ruta[i]
    return ruta

# ==============================
# 3. Parámetros del Algoritmo Genético
# ==============================
# Estos valores funcionan bien para 5-10 ciudades. Se pueden ajustar.
TAMANO_POBLACION = 100   # cuántas rutas por generación
GENERACIONES     = 500   # cuántas iteraciones evolutivas
TASA_MUTACION    = 0.05  # probabilidad de mutación por hijo
TAMANO_TORNEO    = 5     # presión selectiva (k del torneo)
PROB_CRUCE       = 1.0   # probabilidad de aplicar cruce (1.0 = siempre)

print("Iniciando Algoritmo Genético...\n")

# ==============================
# 4. Proceso Evolutivo (AG)
# ==============================
# 4.1 Población inicial
poblacion = crear_poblacion(TAMANO_POBLACION)

# 4.2 Variables para guardar el mejor resultado global
mejor_ruta_global   = None
mejor_aptitud_global = float('inf')
```

```python
# 4.3 Estructuras para graficar la convergencia
historial_mejor_gen    = []  # mejor distancia de cada generación
historial_promedio_gen = []  # distancia promedio por generación
historial_mejor_global = []  # mejor distancia observada hasta ese punto

for gen in range(GENERACIONES):
    # 4.4 Evaluación: calculamos la aptitud (distancia total) de cada ruta
    aptitudes = [calcular_aptitud(r) for r in poblacion]

    # Guardamos estadísticas de la generación
    mejor_aptitud_gen    = min(aptitudes)
    promedio_aptitud_gen = np.mean(aptitudes)
    mejor_ruta_gen       = poblacion[np.argmin(aptitudes)]

    # 4.5 Actualizamos el mejor global si la gen. actual mejora el récord
    if mejor_aptitud_gen < mejor_aptitud_global:
        mejor_aptitud_global = mejor_aptitud_gen
        mejor_ruta_global    = mejor_ruta_gen

    # Guardamos datos para las gráficas
    historial_mejor_gen.append(mejor_aptitud_gen)
    historial_promedio_gen.append(promedio_aptitud_gen)
    historial_mejor_global.append(mejor_aptitud_global)

    # 4.6 Reproducción: elitismo + selección + cruce + mutación
    nueva_poblacion = [mejor_ruta_global]  # Elitismo: conservamos el mejor

    # Llenamos el resto de la nueva población
    while len(nueva_poblacion) < TAMANO_POBLACION:
        # Seleccionamos dos padres por torneo
        padre1 = seleccion_torneo(poblacion, aptitudes, TAMANO_TORNEO)
        padre2 = seleccion_torneo(poblacion, aptitudes, TAMANO_TORNEO)

        # Aplicamos cruce con probabilidad PROB_CRUCE; si no, clonamos
padre1
        if random.random() < PROB_CRUCE:
            hijo = cruce_ordenado(padre1, padre2)
        else:
            hijo = padre1[:]

        # Mutamos el hijo (posible intercambio de dos ciudades)
        hijo = mutacion_intercambio(hijo, TASA_MUTACION)

        # Añadimos el hijo a la nueva población
        nueva_poblacion.append(hijo)
```

```
    # 4.7 Reemplazo generacional: la nueva población sustituye a la anterior
    poblacion = nueva_poblacion

    # Mensaje de progreso cada 50 generaciones (opcional, informativo)
    if (gen + 1) % 50 == 0:
        print(f"Generación {gen+1}/{GENERACIONES} -> Mejor distancia:
{mejor_aptitud_global:.2f}")

# ==============================
# 5. Resultados Finales
# ==============================
# Convertimos los índices de la mejor ruta a nombres de ciudades y cerramos
el ciclo
ruta_optima_nombres = [mapa_indices[i] for i in mejor_ruta_global] +
[mapa_indices[mejor_ruta_global[0]]]

print("\n--- RESULTADOS ---")
print(f"Mejor distancia total: {mejor_aptitud_global:.3f}")
print(f"Ruta óptima: {' -> '.join(ruta_optima_nombres)}")

# ==============================
# 6. Visualización de Resultados
# ==============================
plt.figure(figsize=(12, 6))

# --- Gráfica de la ruta óptima (izquierda) ---
plt.subplot(1, 2, 1)
# Ordenamos las coordenadas de acuerdo con la mejor ruta e incluimos el
regreso al inicio
coords = [lista_ciudades[i] for i in mejor_ruta_global] +
[lista_ciudades[mejor_ruta_global[0]]]
x, y = zip(*coords)

# Dibujamos la ruta en rojo con puntos
plt.plot(x, y, 'ro-', linewidth=2.5, label='Ruta Óptima')
# Marcamos el punto de inicio en verde para distinguirlo
plt.scatter(x[0], y[0], color='lime', s=150, edgecolor='black',
label='Inicio')

# Escribimos el nombre de cada ciudad al lado de su punto
for i, nombre in enumerate(mapa_indices):
    plt.text(lista_ciudades[i][0] + 0.1, lista_ciudades[i][1] + 0.1,
             nombre, fontsize=11, weight='bold')
```

```
plt.title("Ruta Óptima del Viajero")
plt.xlabel("Coordenada X")
plt.ylabel("Coordenada Y")
plt.legend()
plt.grid(True)

# --- Gráfica de convergencia (derecha) ---
plt.subplot(1, 2, 2)
# Tres curvas: cómo mejora el mejor, cómo se comporta el promedio,
# y cuál es el mejor histórico alcanzado hasta cada generación.
plt.plot(historial_mejor_gen,    label="Mejor por generación", linewidth=2)
plt.plot(historial_promedio_gen, label="Promedio por generación",
linewidth=2)
plt.plot(historial_mejor_global, label="Mejor global", linewidth=2)
plt.title("Convergencia del Algoritmo Genético")
plt.xlabel("Generación")
plt.ylabel("Distancia (Aptitud)")
plt.legend()
plt.grid(True)

# Ajuste de espacios y despliegue de las gráficas
plt.tight_layout()
plt.show()
```

5. Conclusión

El algoritmo genético resultó ser una herramienta muy útil y práctica para resolver el problema del vendedor ambulante. Aunque no siempre encuentra la ruta perfecta, logra soluciones muy buenas en poco tiempo y con un proceso sencillo de entender.

Esta tarea me ayudó a comprender cómo funcionan los algoritmos genéticos y cómo pueden aprender a mejorar con cada generación. También demuestra que este tipo de métodos pueden aplicarse fácilmente a situaciones reales, como planificar rutas, optimizar recorridos o resolver otros problemas donde hay muchas posibles combinaciones.

Github:

https://github.com/RojasBernalJose/Topicos_De_Ia