Instituto Tecnológico Nacional de México, campus Culiacán

Ingeniería en Sistemas Computacionales

Trabajo Bonus Evolución Diferencial.

Protocolo de investigación para Tópicos de Inteligencia Artificial

Integrantes del proyecto:
Payan Urquidez Rafael Alberto
Quiñonez Ramirez Nestor de Jesus

Docente: Mora Felix Zuriel Dathan

Repositorio:
https://github.com/Norkat/Topicos-de-Inteligencia-Artificial

Culiacán, Sinaloa a 4 Noviembre del 2025

Diseño del Algoritmo

A continuación se presenta la descripción detallada del diseño e implementación del algoritmo genético desarrollado para estimar los coeficientes **A**, **B** y **C** de una ecuación cuadrática **Ax^2 + Bx + C** dado un valor de **x**. El propósito es documentar las decisiones de representación, operadores evolutivos, parámetros y consideraciones prácticas del código provisto.

1. Objetivo

Encontrar (o aproximar) un triplete de coeficientes **(A,B,C)** tal que la evaluación del polinomio **f(x) = Ax2 + Bx + C** cumpla un criterio definido por la función de aptitud (fitness). En la implementación actual el fitness se calcula como:

$$\text{fitness}(A, B, C; x) = \text{round}\left(\frac{1}{1 + (Ax^2 + Bx + C)}, 2\right)$$

Este valor se utiliza para ordenar y comparar individuos; valores más altos indican mejor ajuste según la función implementada.

2. Representación de individuos

- Cada individuo (cromosoma) es una tupla de tres valores reales: **(A,B,C).**
- Los genes son números de punto flotante, generados y manipulados directamente (codificación real).

- Rango inicial: la población inicial se genera con coeficientes muestreados uniformemente en **[−100,100]** con precisión hasta 2 decimales.

Ejemplo de individuo: **(12.34, -7.50, 0.25).**

3. Inicialización

- Tamaño de población: **poblation_size = 50.**
- La población inicial se crea con **initial_poblation(x, poblation_size)** donde cada individuo viene emparejado con su valor de fitness: **((A,B,C), fitness)**.
- Cada coeficiente se muestrea con **random.uniform(-100, 100)** y se redondea a dos decimales.

Esto asegura diversidad inicial amplia, útil para explorar el espacio de coeficientes.

4. Función de aptitud (fitness)

- Implementada como **1 / (1 + f(x))** y redondeada a 2 decimales.

- Propiedades y observaciones importantes:
    - Si **f(x)** es grande y positivo, el fitness se aproxima a 0.
      Si **f(x)** está cerca de 0, el fitness tiende a 1.
    - Si **f(x)** es negativo, el denominador **1 + f(x)** puede acercarse a 0 o volverse negativo, produciendo fitness muy grande o negativo, esto puede ser indeseado y llevar a comportamientos inestables.
    - Redondear a dos decimales reduce la resolución entre individuos cercanos.

5. Selección

La selección usada combina dos ideas:

1. Elitismo parcial por “mejor absoluto”
    - **best = min(poblation, key=lambda x: abs(x[1] - 1))**
      Aquí se busca el individuo cuyo fitness esté más cerca de 1 (es decir, la menor distancia absoluta a 1). Se devuelve su vector de coeficientes.
    - Esto actúa como una forma de elite/retención del mejor individuo respecto al criterio de estar cerca de **fitness = 1.**

2. Torneo aleatorio
    - Se eligen 5 participantes al azar: participants = random.sample(poblation, 5).
    - El ganador del torneo se selecciona con **tournament_winner = max(participants, key=lambda x: x[1])**, es decir, aquel con mayor valor de fitness dentro de los 5.
    - Esto introduce presión selectiva y diversidad.

La función selection devuelve dos padres: el “mejor” (según cercanía a 1) y el ganador del torneo (según valor de fitness).

6. Cruce (crossover)

- Operador: blend lineal entre coeficientes (simétrico).
- Se genera un alpha aleatorio en **[0,1]**. Para cada coeficiente:
    - **coef_hijo1 = alpha * padre1 + (1 - alpha) * padre2**
    - **coef_hijo2 = (1 - alpha) * padre1 + alpha * padre2**
- Los resultados se redondean a dos decimales.
- Este operador produce hijos intermedios en el segmento entre los padres (convex combination), promoviendo exploración local en torno a los padres.

7. Mutación

- Mutación por gen, con probabilidad **mutation_rate = 1 / poblation_size * 3** (en el código actual eso es **3 / poblation_size**, con **poblation_size=50 → 0.06**).

- Para cada gen del hijo, si **random.uniform(0,1) <= mutation_rate**, se suma un delta aleatorio en **[-0.5, 0.5]** (redondeado a 2 decimales).
- La mutación opera después del cruce y permite escapar de óptimos locales añadiendo ruido pequeño a los coeficientes.

8. Reemplazo (estrategia de supervivencia)

- En cada generación se generan dos hijos (**son1, son2**) y se colocan directamente:
  - **poblation[0] = son1** (reemplaza primer elemento)
  - **poblation[-1] = son2** (reemplaza último elemento)
- La población no se reordenó explícitamente tras el reemplazo dentro del bucle, salvo que al inicio se ordena la población inicial.

9. Criterio de parada

- Número máximo de generaciones: **max_generations = 100.**
- No hay una condición de convergencia adicional (por ejemplo, tolerancia sobre mejora del fitness). Podría añadirse una condición temprana si el fitness alcanza un umbral deseado.

10. Flujo principal (resumen / pseudocódigo)

```
Leer x_value
poblacion = initial_poblation(x_value, N)          # cada elemento = ((A,B,C), fitness)
ordenar poblacion por fitness descendente          # (se hace inicialmente)

para gen in 0 .. max_generations-1:
   padre_elite, padre_torneo = selection(poblacion)
   hijo1_coef, hijo2_coef = crossover(padre_elite, padre_torneo)
   hijo1_coef = mutation(hijo1_coef)
   hijo2_coef = mutation(hijo2_coef)
   hijo1 = (hijo1_coef, fitness(hijo1_coef, x_value))
   hijo2 = (hijo2_coef, fitness(hijo2_coef, x_value))
   poblacion[0] = hijo1
   poblacion[-1] = hijo2
   best = max(poblacion, key=lambda x: x[1])
   imprimir estado
fin para
```

11. Parámetros clave y valores usados

- poblation_size = 50
- mutation_rate = 3 / poblation_size ≈ 0.06
- max_generations = 100
- Rango inicial coeficientes: [-100, 100]
- Delta de mutación por gen: [-0.5, 0.5]
- Tamaño del torneo: 5

12. Complejidad computacional

- Evaluación del fitness: **O(1)** por individuo.
- Cada generación realiza: selección (muestreo y comparaciones constantes), cruce y mutación (constante), y un reemplazo constante → coste por generación **O(1)** por operación multiplicado por tamaño de población cuando se reordena o se recorre para encontrar el mejor **(max)** → **O(N)**.
- Total aproximado: **O(generaciones×N)**.

13. Conclusión

El desarrollo de un algoritmo genético como el presentado permite comprender de manera práctica cómo los principios de la evolución natural (como la selección, el cruce y la mutación) pueden aplicarse a la resolución de problemas de optimización y búsqueda de soluciones. A través de este ejercicio se aprende a diseñar representaciones adecuadas de los individuos, definir funciones de aptitud que guíen la búsqueda, y ajustar parámetros que influyen directamente en la convergencia del algoritmo, como el tamaño de la población, la tasa de mutación y el número de generaciones.

Además, la implementación facilita observar cómo la población evoluciona generación tras generación, mejorando progresivamente las soluciones propuestas y mostrando el equilibrio entre exploración (diversidad) y explotación (mejora de soluciones). En términos prácticos, este tipo de algoritmos demuestra su capacidad para aproximar resultados en problemas donde no existe una solución analítica simple, permitiendo experimentar con estrategias adaptativas, análisis de desempeño y ajustes empíricos.

En conjunto, este proyecto refuerza la comprensión de los conceptos fundamentales de la computación evolutiva y evidencia la utilidad de los algoritmos genéticos como herramientas versátiles en la inteligencia artificial, la ingeniería y la optimización de funciones complejas.