Materia:

DISEÑO ELECTRÓNICO BASADO EN SISTEMAS EMBEBIDOS

Alumno:

Posadas Pérez Isaac Sayeg

Paniagua Rico Juan Julian

García Azzúa Jorge Roberto

Grado y grupo:

8°G

Profesor:

Garcia Ruiz Alejandro Humberto

Unidad 3 - Práctica 5:

Genético

Objetivo de la práctica

Implementar un **Algoritmo Genético**, técnica inspirada en la evolución biológica, para resolver problemas de optimización. A través de selección, cruce (reproducción) y mutación, este algoritmo permite encontrar soluciones eficientes en espacios de búsqueda complejos.

Descripción del funcionamiento del programa

```
from os import error
from optimization_manager import OptimizationManager
import copy
from params import Param
from typing import List, Tuple
import random as rd
from results_manager import ResultManager,ResultsManager
class Genetico:
    def __init__(self,tam_poblacion,optimizer_inicial:OptimizationManager):
        self.tam_poblacion = tam_poblacion
        self.opt_inicial = optimizer_inicial

    def generar_poblacion(self):
        poblacion = [self.opt_inicial.generar_optimizer_vecino(rd.random())
for _ in range(self.tam_poblacion)]
        return poblacion

    def calc_fitness_poblacion(self,poblacion:List[OptimizationManager]):
        mejores = []
        for p in poblacion:
            mejores.append((p,p.funcion_objetivo()))
        return mejores

    def
seleccion(self,num_padres:int,lista_fitness:List[Tuple[OptimizationManager,
float]]):
        if num_padres > self.tam_poblacion:
            raise ValueError("El numero de seleccion de padres no puede ser
mayor a la poblacion")
        else:
            mejores = sorted(lista_fitness, key=lambda x: x[1],
reverse=True)[:num_padres]
            return mejores

    def
cruza(self,padres:List[Tuple[OptimizationManager,float]],prob_cruza:float):
        descendientes = []
        optimizer_padres = [p[0] for p in padres]

        while len(descendientes) < self.tam_poblacion:
            padre1,padre2 = rd.sample(optimizer_padres,2)
```

```
            if rd.random() < prob_cruza:
                hijo1 = padre1.cruzar(padre2)
                hijo2 = padre2.cruzar(padre1)
                descendientes.append(hijo1)
                if len(descendientes) < self.tam_poblacion:
                    descendientes.append(hijo2)
            else:
                descendientes.append(copy.deepcopy(padre1))
                if len(descendientes) < self.tam_poblacion:
                    descendientes.append(copy.deepcopy(padre2))

        return descendientes

    def
mutar_poblacion(self,poblacion:List[OptimizationManager],prob_muta:float,po
tencia_muta:float):
        poblacion_mutada = []
        for p in poblacion:
            if rd.random() < prob_muta:

poblacion_mutada.append(p.generar_optimizer_vecino(potencia_muta))
            else:
                poblacion_mutada.append(p)
        return poblacion_mutada

    def
run(self,num_generaciones,prob_cruza,prob_muta,potencia_muta,num_padres):
        poblacion = self.generar_poblacion()
        mejor_global = None
        fitness_mejor_global = float('-inf')
        resultados = ResultsManager([])


        for generacion in range(num_generaciones):
            lista_fitness = self.calc_fitness_poblacion(poblacion)

            mejor_actual = max(lista_fitness,key=lambda x: x[1])
            if mejor_actual[1] > fitness_mejor_global:
                mejor_global = mejor_actual[0]
                fitness_mejor_global = mejor_actual[1]

            padres =
self.seleccion(num_padres=num_padres,lista_fitness=lista_fitness)

            #Cruza
            descendientes = self.cruza(padres=padres,prob_cruza=prob_cruza)

            poblacion =
self.mutar_poblacion(descendientes,prob_muta,potencia_muta)

            #print(f"Generación {generacion}: Mejor fitness =
{mejor_actual[1]}")
            r =
ResultManager(va=mejor_actual[1],vo=fitness_mejor_global,iteracion=generaci
on,modelo="Genetico")
            resultados.guardar_dato(r)

        return mejor_global,resultados
```

El programa define una clase `Genetico`, que implementa un optimizador basado en evolución. A partir de una población inicial de soluciones, se simula la reproducción y evolución durante un número definido de generaciones, seleccionando las mejores soluciones en cada ciclo.

Componentes clave

- **`tam_poblacion`**: Número de soluciones (individuos) por generación.
- **`opt_inicial`**: Instancia base del optimizador, que se usará como punto de partida para crear variaciones iniciales.

Métodos principales

- **`generar_poblacion()`**
  Crea la población inicial generando soluciones vecinas aleatorias del optimizador base.
- **`calc_fitness_poblacion(poblacion)`**
  Evalúa cada individuo de la población utilizando la función objetivo. Retorna una lista de tuplas `(solución, fitness)`.
- **`seleccion(num_padres, lista_fitness)`**
  Selecciona los individuos con mejor fitness para ser padres en el cruce. Se asegura de que no se seleccionen más padres que la población disponible.
- **`cruza(padres, prob_cruza)`**
  Realiza cruces entre pares de padres seleccionados con una cierta probabilidad. Genera descendientes combinando características de los padres. Si no se realiza cruce, los padres se copian tal cual.
- **`mutar_poblacion(poblacion, prob_muta, potencia_muta)`**
  Aplica mutaciones a los descendientes con una probabilidad determinada. Las mutaciones consisten en generar soluciones vecinas, modificando ligeramente los individuos.

- **`run(num_generaciones, prob_cruza, prob_muta, potencia_muta, num_padres)`**
  Ejecuta el ciclo completo de evolución genética durante `num_generaciones`. En cada generación:

  1. Se evalúa la población.
  2. Se seleccionan los padres.
  3. Se cruzan para formar nuevos individuos.
  4. Se mutan los descendientes.
  5. Se guarda el mejor resultado de la generación.

¿Para qué sirve este algoritmo?

El algoritmo genético es útil en:

- **Optimización global** de funciones complejas con múltiples variables y restricciones.
- **Diseño de sistemas inteligentes**, redes, rutas y configuraciones.
- **Sistemas embebidos** que requieren soluciones eficientes sin sobrecargar recursos.
- **Ajuste automático de parámetros** en modelos de aprendizaje automático o simulaciones.

Ejemplo de ejecución (corrida final)

Durante la ejecución del algoritmo, el historial de mejoras por generación podría mostrar resultados como:

```
Generación 0: Mejor fitness = 0.475
```

```
Generación 1: Mejor fitness = 0.489

Generación 3: Mejor fitness = 0.514

Generación 6: Mejor fitness = 0.535

Generación 10: Mejor fitness = 0.551

...
```

Al finalizar:

```
# Resultado del método run()

mejor_global: <instancia de OptimizationManager con mejor
solución>

resultados: <ResultsManager con datos por generación>
```

Requisitos previos

Para que este script funcione correctamente, requiere los anteriores codigos ya utilizados los cuales son:

- `OptimizationManager`: Define individuos y proporciona métodos como `funcion_objetivo()` y `cruzar()`.
- `Param`: Clase para gestionar los parámetros del individuo.
- `ResultManager` y `ResultsManager`: Administran los resultados registrados por generación.

Genetico Main

```
from params import Param
from optimization_manager import OptimizationManager
from genetico import Genetico  # Cambiado a importar Genetico
import polars as pl
import matplotlib.pyplot as plt


def main():
    # Creación de parámetros (igual que antes)
    param1 = Param(name="Temperatura", min=0, max=40, v_actual=30,
weight=0.4, costo_cambio=12, optim_mode="min")
    param2 = Param(name="Humedad", min=0, max=100, v_actual=30, weight=0.4,
costo_cambio=12, optim_mode="min")
    param3 = Param(name="Presion", min=900, max=1100, v_actual=1000,
weight=0.2, costo_cambio=5, optim_mode="max")

    lista_params = [param1, param2, param3]
    opt = OptimizationManager(lista_params)

    # Configuración del Algoritmo Genético
    ga = Genetico(
        tam_poblacion=50,  # Tamaño de la población
        optimizer_inicial=opt  # Solución inicial
    )

    # Ejecución del algoritmo genético
    best_solution, resultados = ga.run(
        num_generaciones=100,  # Número de generaciones
        prob_cruza=0.8,  # Probabilidad de cruce
        prob_muta=0.2,  # Probabilidad de mutación
        potencia_muta=0.1,  # Intensidad de la mutación
        num_padres=15  # Número de padres para selección
    )

    print("\n------ Mejor solución encontrada ------")
    best_solution.show_params()
    print(f"Valor objetivo: {best_solution.funcion_objetivo()}")

    # Conversión de resultados a DataFrame
    data = []
    for r in resultados.resultados:
        data.append({
            "valor_actual": r.va,
            "valor_optimo": r.vo,
            "iteracion": r.iteracion,
            "modelo": r.modelo
        })

    df = pl.DataFrame(data)

    # Exportación a CSV
    df.write_csv("ga_results.csv")

    # Visualización de resultados
    plt.figure(figsize=(10, 6))
    plt.plot(df["iteracion"], df["valor_actual"], 'b-', label='Mejor
actual')
    plt.plot(df["iteracion"], df["valor_optimo"], 'r-', label='Mejor
global')
    plt.xlabel("Generación")
```

```
    plt.ylabel("Valor de Fitness")
    plt.title("Progreso del Algoritmo Genético")
    plt.legend()
    plt.grid(True)
    plt.savefig("ga_progress.png")
    plt.show()


if __name__ == "__main__":
    main()
```

1. Estructura General

El archivo `Main_Genetico.py` implementa una solución completa de optimización mediante **algoritmos genéticos**, diseñado para:

- Configurar parámetros de optimización.
- Ejecutar el **algoritmo genético**.
- Visualizar y almacenar resultados.
- Proporcionar retroalimentación del proceso.

2. Componentes Principales

2.1 Importaciones

```
from params import Param

from optimization_manager import OptimizationManager

from genetico import Genetico

import polars as pl

import matplotlib.pyplot as plt
```

- **Param**: Define parámetros individuales con sus restricciones.
- **OptimizationManager**: Gestiona la función objetivo y soluciones.
- **Genetico**: Implementación del **algoritmo genético**.

- **Polars**: Manejo eficiente de datos de resultados.
- **Matplotlib**: Visualización gráfica del progreso.

2.2 Configuración de Parámetros

```
param1 = Param(name="Temperatura", min=0, max=40, v_actual=30,
        weight=0.4, costo_cambio=12, optim_mode="min")
param2 = Param(name="Humedad", min=0, max=100, v_actual=30,
        weight=0.4, costo_cambio=12, optim_mode="min")
param3 = Param(name="Presion", min=900, max=1100, v_actual=1000,
        weight=0.2, costo_cambio=5, optim_mode="max")
```

Cada parámetro incluye:

- **Rangos operativos** (min/max).
- **Valor inicial** (`v_actual`).
- **Peso** en la función objetivo.
- **Costo de modificación** (`costo_cambio`).
- **Dirección de optimización** (`min` para minimizar, `max` para maximizar).

2.3 Inicialización del Optimizador

```
lista_params = [param1, param2, param3]
opt = OptimizationManager(lista_params)
```

El **gestor de optimización** agrupa los parámetros y prepara:

- Cálculo de la función objetivo.
- Generación de soluciones vecinas.
- Gestión de restricciones.

3. Configuración del Algoritmo Genético

```
ga = Genetico(
    tam_poblacion=50,          # Tamaño de cada generación
    optimizer_inicial=opt      # Punto de partida
)
```

3.1 Ejecución del Algoritmo

```
best_solution, resultados = ga.run(
    num_generaciones=100,    # Ciclos evolutivos
    prob_cruza=0.8,          # Probabilidad de reproducción
    prob_muta=0.2,           # Probabilidad de mutación
    potencia_muta=0.1,       # Intensidad de las mutaciones
    num_padres=15            # Individuos seleccionados
)
```

El proceso interno sigue estos pasos:

- **Selección**: Elitismo de los mejores individuos.
- **Cruce**: Recombinación de soluciones prometedoras.
- **Mutación**: Introducción de diversidad genética.
- **Reemplazo**: Generación de nueva población.

4. Manejo de Resultados

4.1 Visualización por Consola

```
print("\n------ Mejor solución encontrada ------")

best_solution.show_params()

print(f"Valor objetivo: {best_solution.funcion_objetivo()}")
```

Se muestran:

- **Configuración óptima** de parámetros.
- **Valor alcanzado** de la función objetivo.

4.2 Procesamiento de Datos

```
data = [{

    "valor_actual": r.va,      # Mejor fitness de la generación

    "valor_optimo": r.vo,      # Mejor histórico

    "iteracion": r.iteracion,  # Número de generación

    "modelo": r.modelo         # "Genetico"

} for r in resultados.resultados]
```

```
df = pl.DataFrame(data)
```

Estructura del DataFrame:

- **Valor actual** (`valor_actual`).
- **Mejor valor histórico** (`valor_optimo`).
- **Número de iteración** (`iteracion`).
- **Modelo utilizado** (`Genetico`).

4.3 Exportación y Visualización

```
df.write_csv("ga_results.csv")  # Datos completos en CSV


plt.figure(figsize=(10, 6))
plt.plot(df["iteracion"], df["valor_actual"], 'b-', label='Mejor actual')
plt.plot(df["iteracion"], df["valor_optimo"], 'r-', label='Mejor global')
plt.xlabel("Generación")
plt.ylabel("Valor de Fitness")
plt.title("Progreso del Algoritmo Genético")
plt.legend()
plt.grid(True)
plt.savefig("ga_progress.png")  # Gráfico de convergencia
plt.show()
```

Genera un **gráfico de convergencia**, mostrando:

- **Evolución del fitness** en cada generación.
- **Comparación entre el mejor valor global y el mejor de cada iteración**.

5. Parámetros Recomendados

Parámetro	Valores Típicos	Descripción
`tam_poblacion`	30-100	Balance entre diversidad y costo computacional.
`num_generaciones`	50-200	Depende de la complejidad del problema.
`prob_cruza`	0.7-0.95	Controla la recombinación genética.
`prob_muta`	0.01-0.2	Evita convergencia prematura.
`potencia_muta`	0.05-0.3	Magnitud de los cambios aleatorios.
`num_padres`	20-50% población	Número de elites seleccionadas.

6. Flujo de Ejecución

Inicialización de parámetros y población

1. Se configuran los parámetros del problema.
2. Se inicializa una población inicial con soluciones aleatorias.

Loop generacional

1. **Evaluación del fitness inicial**.

2. **Selección de individuos prometedores**.
3. **Operadores genéticos (cruce y mutación)**.
4. **Evaluación de la nueva población**.
5. **Registro de resultados en cada iteración**.

Post-procesamiento

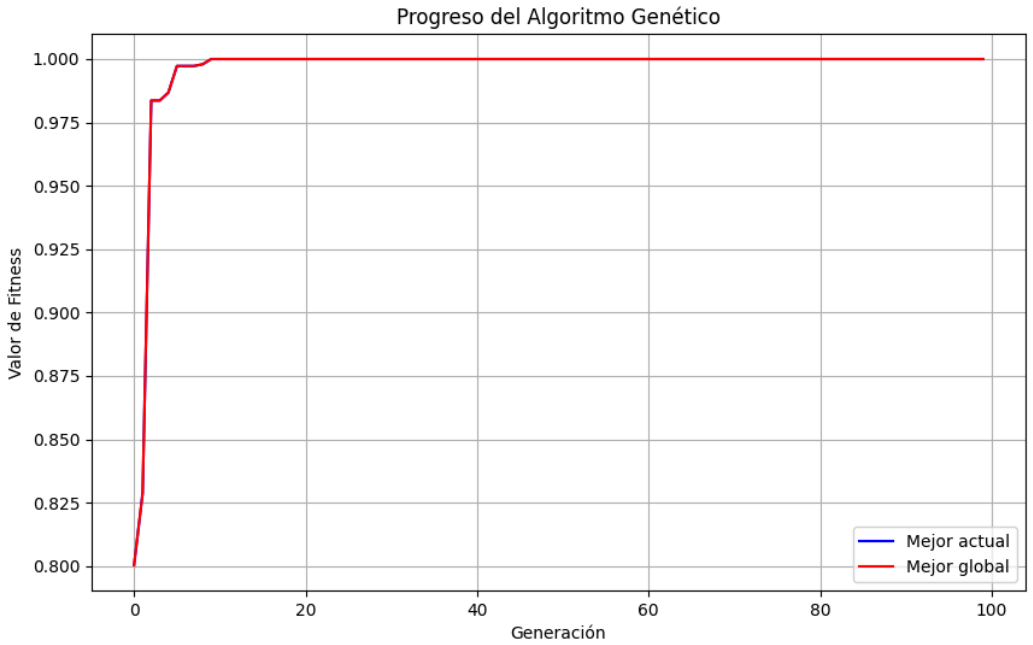
1. **Análisis de convergencia**.
2. **Exportación de datos** en formato CSV.
3. **Visualización gráfica del progreso**.

7. Archivos Generados

Archivo	Formato	Contenido
`ga_results.csv`	CSV	Datos numéricos completos de la ejecución.
`ga_progress.png`	PNG	Gráfico de convergencia del algoritmo.

Corrida del programa

```
------ Mejor solución encontrada ------
Parametro: Temperatura
  Min: 0
  Max: 40
  Valor Actual: 0
  Peso: 0.4
  Costo de Cambio: 12
  Modo de Optimización: min
  ---------------------
Parametro: Humedad
  Min: 0
  Max: 100
  Valor Actual: 0
  Peso: 0.4
  Costo de Cambio: 12
  Modo de Optimización: min
  ---------------------
Parametro: Presion
  Min: 900
  Max: 1100
  Valor Actual: 1100
  Peso: 0.2
  Costo de Cambio: 5
  Modo de Optimización: max
  ---------------------
Valor objetivo: 1.0
```


Progreso del Algoritmo Genético
Valor de Fitness
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
20
40
60
80
100
Generación
Mejor actual
Mejor global