Instituto Politecnico Nacional
Escuela Superior de Computo

Materia: Tópicos Selectos de Algoritmos Bioinspirados
Grupo: 7BM1

Profesor: Daniel Molina Pérez
Periodo: 2025/02

Practica 01

Maximizar Contraste en Imagenes Medicas.

Realizado por:
Carrillo Barreiro José Emiliano
Martinez Ayala Gerardo
Robles Otero José Ángel
Vásquez Morales Haniel Ulises

Abstract:

This report describes the design, implementation, and evaluation of a Genetic Algorithm (GA) aimed at minimizing multimodal functions. Two benchmark functions—Langermann and Drop-Wave—serve as test cases. The GA employs tournament selection, Simulated Binary Crossover (SBX) with boundary handling, and polynomial mutation with boundaries. In addition to detailing the algorithm's components, the report outlines the experimental setup, visualization techniques, and potential avenues for future improvements.

Resumen:

En este reporte se describe el diseño, implementacion y evaluacion de un Algoritmo Genetico (GA) adecuado a minimizar funciones multi-modales. Con dos funciones de evaluación comparativa (Benchmark functions)–*Langermann* y *Drop-Wave*– sirven como ejemplificacion de casos de uso. El GA implementa: *Selección por Torneo*, *Simulated Binary Crossover (SBX)* con manejo de limites, *Mutación Polinomial* con uso de cotas y *Sustitución Extinitiva con Elitismo*. Ademas de detallar los componentes de los algoritmos, el reporte añade los valores de los parametros utilizados, tecnicas de visualizacion y potenciales caminos para su mejora continua.

Fecha: 12 de abril de 2025

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Los algoritmos genéticos (AG) son una clase de algoritmos bioinspirados ampliamente utilizados para resolver problemas de optimización y búsqueda. Inspirados en el proceso de evolución natural, estos algoritmos imitan mecanismos biológicos como la selección, el cruzamiento y la mutación para explorar el espacio de soluciones y encontrar resultados óptimos. En este proyecto se implementa un AG para la minimización de funciones benchmark, en particular las funciones Langermann y Drop-Wave, con el objetivo de demostrar y analizar la efectividad de técnicas como la selección por torneo, el cruzamiento Simulated Binary Crossover (SBX) y la mutación polinomial.

Capítulo 2

Objetivos del Proyecto

2.1 Implementar un algoritmo genético robusto

Desarrollar un Algoritmo Genético (AG) que utilice métodos avanzados de selección, cruzamiento y mutación para la optimización de funciones complejas.

2.2 Minimización de funciones benchmark

Aplicar el AG a las funciones Langermann y Drop-Wave, que presentan múltiples óptimos locales y retos propios en la búsqueda de la solución global.

2.3 Visualización y análisis

Generar salidas que permitan analizar la evolución del fitness a lo largo de las generaciones y visualizar la superficie de la función en 3D, facilitando la comprensión del comportamiento del algoritmo.

Capítulo 3

Metodología

3.1 Inicialización de la Población

La población inicial se genera de forma uniforme a lo largo del espacio de búsqueda, definido por límites inferiores y superiores para cada variable.

Objetivo

Garantizar que la búsqueda comience explorando de manera equitativa todas las regiones posibles, evitando sesgos que puedan limitar la diversidad de soluciones iniciales.

Implementación

Se utiliza la función `initialize_population`, la cual emplea métodos de generación aleatoria (por ejemplo, la función `np.random.uniform` de NumPy) para crear un conjunto de individuos.

Ventajas

- Permite cubrir todo el rango definido para cada variable.
- Aumenta la probabilidad de encontrar regiones prometedoras del espacio de soluciones desde el inicio.

3.2 Evaluación de Fitness

Cada individuo generado se evalúa mediante la función objetivo, la cual determina qué tan buena es la solución propuesta.

Funciones Utilizadas

- **Langermann**: Es una función multimodal que combina componentes cosenoidales y exponenciales, generando múltiples óptimos locales.
- **Drop-Wave**: Una función bidimensional con una superficie ondulada, usada para analizar el comportamiento del algoritmo en entornos con múltiples picos y valles.

Proceso

Se calcula el valor de fitness para cada individuo (por ejemplo, evaluando $f(x_1, x_2)$) y se almacena dicho valor para posteriores comparaciones.

Importancia

La evaluación correcta del fitness es crucial, ya que determina la selección de individuos y, por ende, el rumbo de la evolución poblacional.

3.3 Selección por Torneo

Para elegir los padres que generarán la siguiente generación se utiliza un método de selección por torneo.

Mecanismo

- Se forman múltiples grupos (torneos) de individuos seleccionados al azar.
- En cada grupo se compara el fitness de los participantes y se selecciona al individuo con el mejor desempeño.

Implementación Vectorizada

La función `vectorized_tournament_selection` realiza este proceso de forma eficiente, aprovechando operaciones vectorizadas de NumPy.

Beneficios

- Favorece la selección de soluciones de alta calidad sin descartar por completo la diversidad poblacional.
- Permite controlar la presión selectiva mediante el tamaño del torneo.

3.4 Cruzamiento con SBX

El operador de cruzamiento se implementa mediante el método SBX (Simulated Binary Crossover).

Proceso del SBX

- A partir de dos padres, se genera un número aleatorio u y se calcula un parámetro β que determina la dispersión de los descendientes respecto a los padres.
- Se generan dos hijos combinando linealmente los valores de los padres.

Ajuste de Límites

Se incorpora un mecanismo en `sbx_crossover_with_boundaries` que garantiza que los hijos resultantes se mantengan dentro de los límites predefinidos.

Ventajas

- Promueve la creación de soluciones intermedias que pueden explotar la información genética de ambos padres.
- Ayuda a preservar la diversidad en la población.

3.5 Mutación Polinomial

Para introducir variabilidad y explorar nuevas regiones del espacio de búsqueda, se aplica la mutación polinomial.

Mecanismo de la Mutación

- Cada gen de un individuo tiene una probabilidad definida de sufrir una mutación.
- Se usa una distribución polinomial, controlada por el parámetro η_{mut}.

Consideraciones de Límites

La mutación se aplica respetando los límites definidos para cada variable mediante la función `polynomial_mutation_with_b`

Beneficios

- Introduce pequeñas variaciones que pueden conducir a la exploración de nuevas soluciones.
- Previene la convergencia prematura al mantener la diversidad genética.

3.6 Elitismo y Ciclo Evolutivo

El proceso evolutivo se estructura en ciclos o generaciones.

Elitismo

- Se retiene el mejor individuo de la generación actual y se garantiza su inclusión en la siguiente generación.
- Esto asegura que la calidad de la solución nunca empeore a lo largo de las generaciones.

Ciclo Evolutivo

- Cada generación incluye la selección, el cruzamiento, la mutación y la incorporación del individuo de élite.
- La evolución se repite durante un número predefinido de generaciones.

Registro y Análisis

- Se almacena el historial del fitness y de las mejores soluciones.
- Esto facilita el análisis del comportamiento del algoritmo y la generación de visualizaciones.

3.7 Penalización Exterior

Los métodos de penalización son técnicas para manejar restricciones en problemas de optimización. La idea principal es transformar un problema de optimización con restricciones en uno sin restricciones, modificando la función objetivo para penalizar las soluciones que no cumplen con las restricciones. En el método de penalización exterior, esto se logra agregando términos a la función objetivo que aumentan su valor cuando se violan las restricciones. [cite: 45, 46, 47, 48, 49, 50, 51]

La función objetivo modificada, denotada como $F_P(x)$, se define como:

$$F_P(x) = f(x) + \lambda P(x)$$

Donde:

- $f(x)$ es la función objetivo original del problema.
- $P(x)$ es la función de penalización, que cuantifica la violación de las restricciones.
- λ_P es el parámetro de penalización, que controla la severidad de la penalización.

La función de penalización $P(x)$ se calcula de la siguiente manera:

$$P(x) = \sum_{i=1}^{n} \text{máx}(0, g_i(x))^2 + \sum_{j=1}^{m} h_j(x)^2$$

Donde:

- $g_i(x)$ representa la i-ésima restricción de desigualdad, de la forma $g_i(x) \leq 0$.
- $h_j(x)$ representa la j-ésima restricción de igualdad, de la forma $h_j(x) = 0$.

Funcionamiento de la Penalización Exterior

- Para las restricciones de desigualdad $g_i(x) \leq 0$:
  - Si la restricción se cumple, $\text{máx}(0, g_i(x)) = 0$, por lo que no hay penalización.
  - Si la restricción se viola, $\text{máx}(0, g_i(x)) = g_i(x)$, y se aplica una penalización proporcional al cuadrado de la violación.
- Para las restricciones de igualdad $h_j(x) = 0$:
  - Si la restricción se cumple, $h_j(x)^2 = 0$, y no hay penalización.
  - Si la restricción no se cumple, $h_j(x)^2 > 0$, y se aplica una penalización proporcional al cuadrado de la diferencia.

Ventajas

- **Facilidad de implementación:** Es relativamente sencillo agregar la penalización externa a los algoritmos de optimización existentes.
- **Flexibilidad:** Permite explorar soluciones que violan las restricciones, lo cual puede ser útil en problemas complejos.

Desventajas

- **Distorsión de la solución óptima:** La penalización puede llevar a soluciones subóptimas si se priorizan soluciones que violan las restricciones pero tienen un mejor valor de la función objetivo. [cite: 61, 62]
- **Sensibilidad al parámetro de penalización:** La elección del valor de λ_P es crucial. Si es muy pequeño, las restricciones pueden no cumplirse; si es muy grande, puede dificultar la convergencia. [cite: 63, 64, 65]
- **Convergencia:** Puede causar una convergencia lenta o incluso la no convergencia del algoritmo.

Capítulo 4

Resultados y Discusión

Durante la ejecución del algoritmo se realizaron múltiples corridas completas para cada función objetivo (Langermann y Drop-Wave), lo que permitió evaluar la estabilidad y eficiencia del método. Los resultados se agruparon en resúmenes globales[1], donde se registraron indicadores clave, los cuales sencuentran dentro de la siguientes tablas:

Cuadro 4.1: Problema 01: resumen_global_corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor (Fitness)	0.0	0.0565	0.4	0.3714	0.1746	0.0	-0.6891
Peor (Fitness)	0.1285	0.0225	0.4	0.0905	0.1456	0.214	-0.6114
Media	0.0484	0.0641	0.4	0.1991	0.2152	0.0749	-0.6573
Desv. Estándar	0.0595	0.0349	0.0	0.1088	0.0952	0.0767	0.028

Cuadro 4.2: Problema 02: resumen_global_corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor (Fitness)	0.2503	0.2307	0.0016	0.2188	0.1193	0.1792	0.0022
Peor (Fitness)	0.1087	0.1276	0.0557	0.1346	0.2783	0.295	0.0075
Media	0.2189	0.1639	0.0529	0.1631	0.2089	0.1921	0.0049
Desv. Estándar	0.0918	0.043	0.0291	0.0404	0.0749	0.0571	0.0018

Cuadro 4.3: Problema 03: resumen_global_corridas

Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor (Fitness)	0.2971	0.1112	0.0351	0.2159	0.1974	0.1433	0.0028
Peor (Fitness)	0.1291	0.2206	0.0217	0.3507	0.0449	0.2331	0.0045
Media	0.1934	0.1866	0.0286	0.264	0.1439	0.1835	0.0037
Desv. Estándar	0.0563	0.0499	0.0174	0.0551	0.0945	0.0484	0.0006

Donde cada Indicador Representa lo siguiente:

- **Mejor (Fitness):** Representa la solución con el valor de fitness mínimo obtenido en todas las corridas.
- **Peor (Fitness):** Indica la solución con el mayor valor de fitness, sirviendo como referencia de la variabilidad en la búsqueda.

[1] Se anexan tablas de los resumenes independientes de cada ejecución en la seccion de resumenes e historiales correspondiente a cada función dentro del capitulo de anexos.

- **Media:** Es el promedio de los valores de fitness de la mejor solución de cada corrida, ofreciendo una visión global del desempeño del algoritmo.
- **Desv. Estándar:** Mide la dispersión de los valores de fitness entre las corridas, reflejando la estabilidad y consistencia del proceso evolutivo.

4.1 Análisis de los Resultados

4.1.1. Consistencia y Robustez

Los resúmenes globales muestran que, a lo largo de las corridas, el algoritmo tiende a converger de manera consistente hacia soluciones de alta calidad. Una baja desviación estándar en los valores de fitness sugiere que el proceso evolutivo es robusto y no depende en exceso de la aleatoriedad inherente a los operadores genéticos. Esto es crucial para problemas de optimización de carteras, ya que garantiza que la metodología aplicada es reproducible y confiable.

4.1.2. Análisis por Problema

Problema 1: Maximización de Retorno

El resumen global para el Problema 1 indica la capacidad del algoritmo para identificar carteras con un alto retorno esperado. El valor de "Mejor (Retorno)" obtenido se sitúa en un rango competitivo, y la media de las corridas respalda la eficacia del algoritmo genético para explorar el espacio de soluciones y encontrar combinaciones de activos que maximizan el retorno, respetando la restricción de inversión máxima por activo.

Problema 2: Minimización de Riesgo

Para el Problema 2, los resultados globales evidencian una convergencia hacia carteras con un riesgo mínimo, cumpliendo con la restricción de rendimiento esperado mínimo. Los indicadores de "Peor (Riesgo)" y "Desv. Estándar" muestran que, aunque existen ciertas variaciones entre corridas, el algoritmo genético logra encontrar soluciones consistentes que equilibran el riesgo y el rendimiento esperado.

Problema 3: Maximización de Retorno con Riesgo Controlado

En el Problema 3, los resultados muestran la capacidad del algoritmo para maximizar el retorno esperado sin exceder un nivel de riesgo predefinido. Esto demuestra la efectividad del método de penalización para guiar la búsqueda hacia soluciones que satisfacen simultáneamente múltiples objetivos y restricciones.

4.1.3. Evolución del Fitness y Visualizaciones

Las gráficas de la evolución del fitness, tanto en su forma original como normalizada, permiten observar el progreso generacional. Se aprecia una clara tendencia a la mejora, donde la mayoría de las corridas muestran una reducción significativa del valor de fitness a medida que avanzan las generaciones.

A continuación se dejan las gráficas de la evolución del fitness:

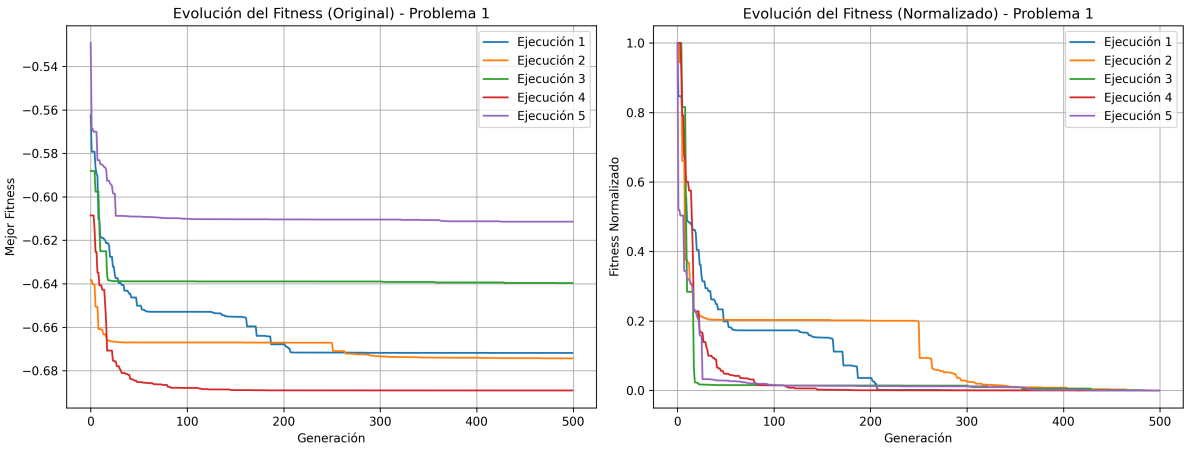

Figura 4.1: Evolución del fitness del Problema 1

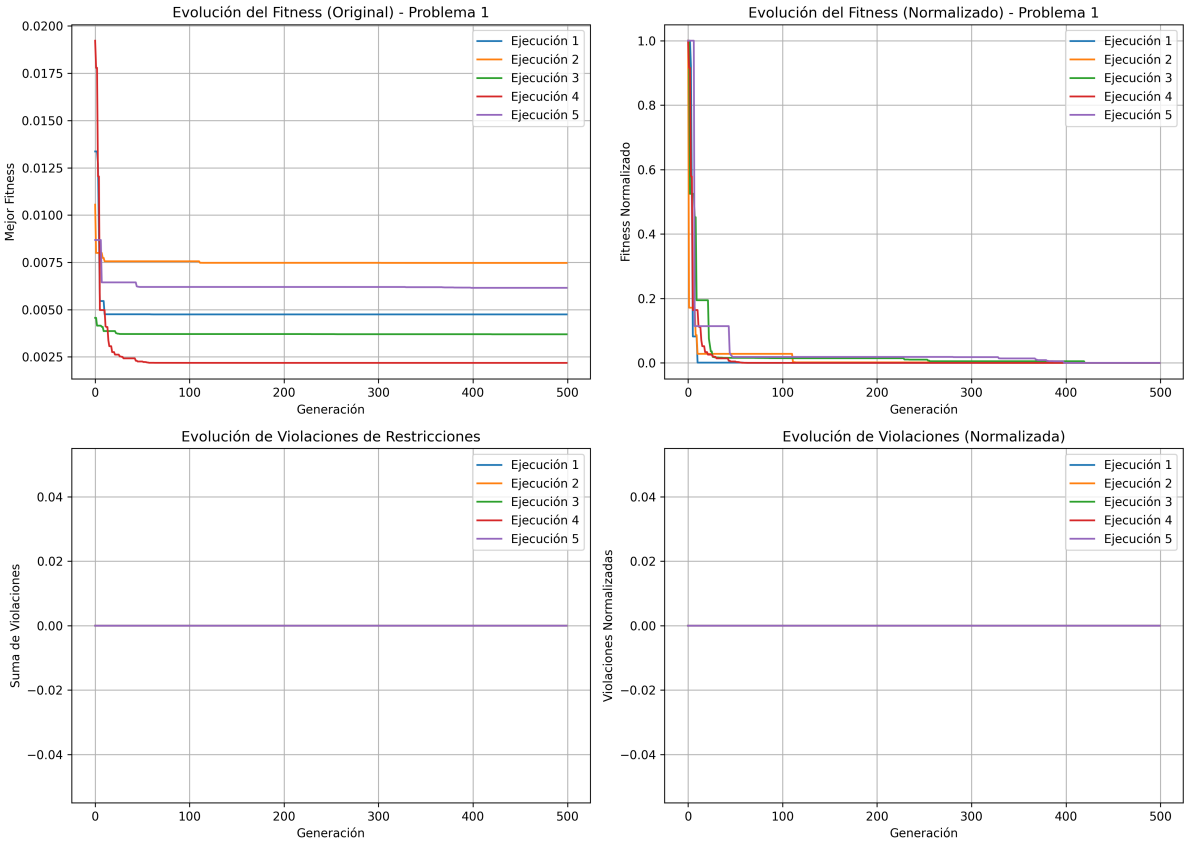

Figura 4.2: Evolución del fitness del Problema 2

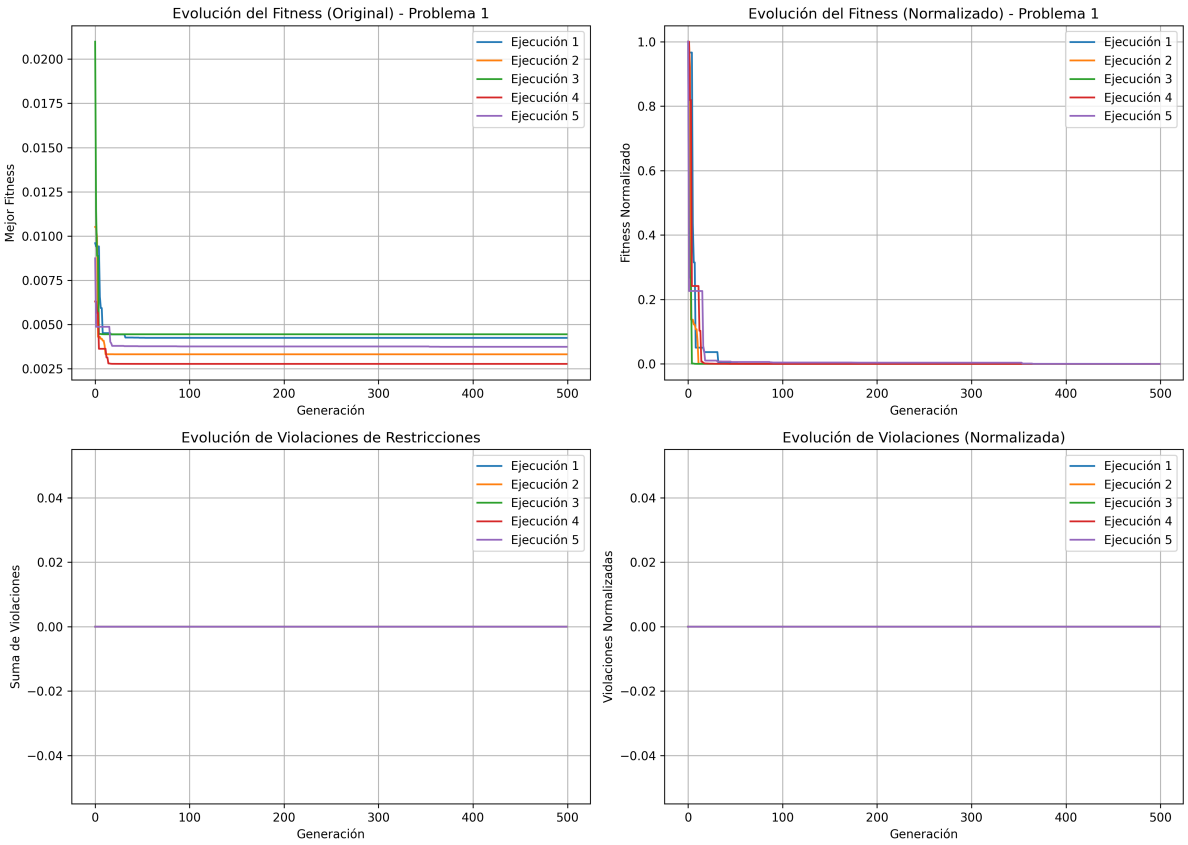

Figura 4.3: Evolución del fitness del Problema 3

4.2 Discusión Resultados

- **Eficacia del Algoritmo:** Los indicadores globales extraídos de los CSV demuestran que el algoritmo genético es capaz de acercarse a la solución óptima, manteniendo una evolución progresiva y consistente en la reducción del valor de fitness.
- **Diversidad y Convergencia:** La aplicación de operadores de selección, cruzamiento y mutación, junto con el mecanismo de elitismo, garantiza un equilibrio entre la exploración y la explotación del espacio de búsqueda. Esto se refleja en la baja variabilidad entre corridas, lo que es un indicativo de la estabilidad del proceso.
- **Potencial de Adaptación:** La estructura modular y la robustez mostrada por los resultados permiten considerar la posibilidad de aplicar este marco a otros problemas de optimización, incluso aquellos con mayores dimensiones o con funciones objetivo de mayor complejidad.

Capítulo 5

Implementación

El proyecto se ha desarrollado siguiendo una arquitectura modular[1] que permite separar claramente las distintas funcionalidades y facilita tanto el mantenimiento como la ampliación futura. A continuación, se detallan los principales componentes y cómo se integran en el sistema:

5.1 Funciones Objetivo

5.1.1. Descripción

Las funciones objetivo definen el problema a optimizar. En este proyecto se incluyen dos funciones:

- **Langermann**: Una función multimodal que combina componentes cosenoidales y exponenciales, generando múltiples óptimos locales.
- **Drop-Wave**: Una función bidimensional con una superficie ondulada, ideal para evaluar el desempeño del algoritmo en entornos con picos y valles.

5.1.2. Implementación

Estas funciones se encuentran en el archivo `functions.py` y están parametrizadas con sus respectivos límites de búsqueda (por ejemplo, Langermann se define en el intervalo [0,10] para cada variable, mientras que Drop-Wave utiliza los límites [-5.12, 5.12]).

5.2 Módulos del Algoritmo Genético

El núcleo del algoritmo genético se distribuye en varios módulos:

5.2.1. Inicialización

Función: `initialize_population` **Ubicación**: `libs/auxiliaries_functions.py` **Descripción**: Genera la población inicial de manera uniforme en el espacio de búsqueda.

[1]Se puede encontrar el codigo perteneciente a cada funcion dentro de

5.2.2. Selección

Función: `vectorized_tournament_selection` **Ubicación**: `libs/selection.py` **Descripción**: Se usa un enfoque de torneos para la selección de padres, empleando operaciones vectorizadas con NumPy.

5.2.3. Cruzamiento

Funciones: `sbx_crossover`, `sbx_crossover_with_boundaries` **Ubicación**: `libs/crossover.py` **Descripción**: Implementa el operador SBX (Simulated Binary Crossover) con y sin control de límites.

5.2.4. Mutación

Funciones: `polynomial_mutation`, `polynomial_mutation_with_boundaries` **Ubicación**: `libs/mutation.py` **Descripción**: Aplica mutación polinomial, con control opcional de límites para mantener la viabilidad de las soluciones.

5.2.5. Ejecución del Algoritmo

Función: `genetic_algorithm` **Ubicación**: `AG.py` **Descripción**: Gestiona el ciclo evolutivo completo del algoritmo genético.

5.3 Visualización y Almacenamiento

5.3.1. Visualización

Módulo: `libs/plot.py` **Funciones**: `plot_evolucion_fitness`, `plot_surface_3d` **Descripción**: Permite analizar la evolución del fitness y visualizar la superficie de las funciones objetivo.

5.3.2. Almacenamiento

Estructura de Carpetas:

- Directorio `outputs` organizado en subcarpetas por función.
- Historiales en archivos CSV con datos de fitness y variables.
- Resúmenes estadísticos de cada corrida.

Integración: `main_script.py` ejecuta el algoritmo para cada función definida en `AG_confs.py`.

Escalabilidad: La arquitectura modular permite agregar nuevas funciones objetivo y modificar operadores genéticos sin afectar la estructura base.

Capítulo 6

Conclusiones

El proyecto demuestra la efectividad de los algoritmos genéticos en la optimización de funciones complejas. La implementación de técnicas avanzadas, como el cruzamiento SBX y la mutación polinomial, combinada con una estrategia de selección por torneo, ha permitido explorar de manera eficiente el espacio de soluciones y mejorar progresivamente el fitness de la población. Las herramientas de visualización y el almacenamiento de resultados facilitan el análisis del comportamiento del algoritmo y ofrecen una base sólida para futuras mejoras o aplicaciones a problemas más complejos.

Además, el algoritmo genético desarrollado constituye un marco sólido y escalable para la optimización de funciones complejas. Los resultados obtenidos respaldan la viabilidad del enfoque y abren la puerta a futuras investigaciones, ya sea para afinar los parámetros del algoritmo o para extender su aplicación a problemas con mayores dimensiones o características más complejas. Esta base permite, además, la incorporación de mejoras y la adaptación del método a diferentes contextos, consolidando su utilidad en el ámbito de la optimización computacional.

Apéndice A

GitHub

Se anexa un enlace al repositorio donde se encuentran los codigos y una README que examnde la información aqui mostrada: *Click aqui para abrir el enlace al repositorio* funcion

Apéndice B

Scripts

B.1 Archivo `main.py`

```python
import os
import pandas as pd
from AG_confs import *

from AG import genetic_algorithm
from libs.plot import *

def main():
    # Crear carpetas de salida generales
    os.makedirs("outputs", exist_ok=True)

    for func_key, func_data in FUNCTIONS.items():
        f_obj = func_data["func"]
        lb = func_data["lb"]
        ub = func_data["ub"]
        func_name = func_data["name"]
        num_runs = func_data["num_runs"]

        # Carpetas especificas de cada funcion
        func_folder = f"outputs/{func_key}"
        os.makedirs(func_folder, exist_ok=True)
        hist_folder = os.path.join(func_folder, "historiales")
        res_folder = os.path.join(func_folder, "resumenes")
        os.makedirs(hist_folder, exist_ok=True)
        os.makedirs(res_folder, exist_ok=True)

        print(f"\n==============================================")
        print(f"  FUNCIoN: {func_name}")
        print(f"==============================================")

        all_runs_history = []
        best_solutions_all_runs = []  # Guardaremos los mejores individuos (x1, x2) de cada corrida
        best_values_across_runs = []  # Guardaremos el best_val (fitness) de cada corrida

        for run in range(num_runs):
            print(f"\nEjecucion {run+1}/{num_runs}")

            (best_sol, best_val,
             worst_sol, worst_val,
             avg_sol,  avg_val,
             std_val,
             best_fitness_history,
             best_x1_history,
             best_x2_history,
             population_final,
             fitness_final,
```

```
             best_solutions_over_time) = genetic_algorithm(
                 f_obj, lb, ub,
                 pop_size=POP_SIZE,
                 num_generations=NUM_GENERATIONS,
                 tournament_size=TOURNAMENT_SIZE,
                 crossover_prob=CROSSOVER_PROB,
                 eta_c=ETA_C,
                 mutation_prob=MUTATION_PROB,
                 eta_mut=ETA_MUT
             )

             # 1) Guardar historial
             df_historial = pd.DataFrame({
                 "Generacion": np.arange(1, NUM_GENERATIONS + 1),
                 "Mejor x1": best_x1_history,
                 "Mejor x2": best_x2_history,
                 "Mejor Fitness": best_fitness_history
             })
             historial_filename = os.path.join(hist_folder,
                 f"historial_run_{run+1}.csv")
             df_historial.to_csv(historial_filename, index=False)

             # 2) Guardar resumen de la corrida
             data_resumen = [
                 ["Mejor", best_sol[0], best_sol[1], best_val],
                 ["Media", avg_sol[0], avg_sol[1], avg_val],
                 ["Peor", worst_sol[0], worst_sol[1], worst_val],
                 ["Desv. estandar", np.nan, np.nan, std_val]
             ]
             df_resumen = pd.DataFrame(data_resumen, columns=["Indicador", "x1",
                 "x2", "Fitness"])
             resumen_filename = os.path.join(res_folder, f"resumen_run_{run+1}.csv")
             df_resumen.to_csv(resumen_filename, index=False)

             print(df_resumen.to_string(index=False))

             all_runs_history.append(best_fitness_history)
             best_solutions_all_runs.append(best_sol)
             best_values_across_runs.append(best_val)

         # ==========================================
         #       RESUMEN GLOBAL DE LAS CORRIDAS
         # ==========================================
         best_values_arr = np.array(best_values_across_runs)
         solutions_arr = np.array(best_solutions_all_runs)  # Cada fila: [x1, x2]
             del mejor individuo de cada corrida

         # Para el "Mejor" y "Peor", buscamos el indice de la corrida con minimo y
             maximo fitness
         min_index = np.argmin(best_values_arr)
         max_index = np.argmax(best_values_arr)

         data_global = [
             ["Mejor (Fitness)", solutions_arr[min_index, 0],
                 solutions_arr[min_index, 1], best_values_arr[min_index]],
             ["Peor (Fitness)", solutions_arr[max_index, 0],
                 solutions_arr[max_index, 1], best_values_arr[max_index]],
             ["Media", np.mean(solutions_arr[:, 0]), np.mean(solutions_arr[:, 1]),
                 np.mean(best_values_arr)],
             ["Desv. Estandar", np.std(solutions_arr[:, 0]),
                 np.std(solutions_arr[:, 1]), np.std(best_values_arr)]
         ]
         df_global = pd.DataFrame(data_global, columns=["Indicador", "x1", "x2",
             "Fitness"])

         global_filename = os.path.join(res_folder, "resumen_global_corridas.csv")
         df_global.to_csv(global_filename, index=False)

         # (Opcional) Graficar evolucion del fitness de todas las corridas
         plot_evolucion_fitness(all_runs_history, func_key, func_name)

         # (Opcional) Graficar superficie 3D si la funcion es de 2 variables
         if len(lb) == 2:
```

```
            plot_surface_3d(f_obj, lb, ub, best_solutions_all_runs, func_key,
                func_name)

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

Listing B.1: Implementación de `main.py`

B.2 Archivo `AG_confs.py`

```
import numpy as np
from libs.functions import langermann, drop_wave
# ---------------------------
# Parametros del algoritmo
# ---------------------------
POP_SIZE = 100             # Numero de individuos en la poblacion
NUM_GENERATIONS = 200      # Numero de generaciones
NUM_RUNS = 10              # Numero de ejecuciones completas (ciclos)

# Parametros de la funcion de Langermann
a = np.array([3, 5, 2, 1, 7])
b = np.array([5, 2, 1, 4, 9])
c = np.array([1, 2, 5, 2, 3])

# Parametros del torneo
TOURNAMENT_SIZE = 3  # Numero de individuos participantes en cada torneo

# Parametros del cruzamiento SBX
CROSSOVER_PROB = 0.9  # Probabilidad de aplicar cruzamiento
ETA_C = 15            # indice de distribucion para SBX

# Parametros de la mutacion polinomial
MUTATION_PROB = 10.0 / 2  # Probabilidad de mutar cada gen
ETA_MUT = 20              # indice de distribucion para mutacion polinomial

best_solutions_list = []
all_runs_history = []  # Para graficar luego

FUNCTIONS = {
    "langermann": {
        "func": langermann,
        "lb": np.array([0, 0]),
        "ub": np.array([10, 10]),
        "name": "Langermann",
        "num_runs": NUM_RUNS
    },
    "drop_wave": {
        "func": drop_wave,
        "lb": np.array([-5.12, -5.12]),
        "ub": np.array([5.12, 5.12]),
        "name": "Drop-Wave",
        "num_runs": NUM_RUNS
    }
}
```

Listing B.2: Implementación de `AG_confs.py`

B.3 Archivo `AG.py`

```
from AG_confs import *
from libs.selection import vectorized_tournament_selection
from libs.crossover import sbx_crossover_with_boundaries
from libs.mutation import polynomial_mutation_with_boundaries
from libs.auxiliaries_functions import initialize_population
```

```
6
7
8  # ---------------------------
9  # Funcion principal del GA
10 # ---------------------------
11 def genetic_algorithm(objective_func, lower_bound, upper_bound,
12                        pop_size=POP_SIZE, num_generations=NUM_GENERATIONS,
13                        tournament_size=TOURNAMENT_SIZE,
14                        crossover_prob=CROSSOVER_PROB, eta_c=ETA_C,
15                        mutation_prob=MUTATION_PROB, eta_mut=ETA_MUT):
16     """
17     Ejecuta el GA para la funcion objetivo dada y retorna:
18       - best_solution, best_value
19       - worst_solution, worst_value
20       - avg_solution, avg_value
21       - std_value (fitness)
22       - best_fitness_history, best_x1_history, best_x2_history
23       - population (final), fitness (final)
24       - best_solutions_over_time (para animaciones)
25     """
26     num_variables = len(lower_bound)
27
28     # 1) Inicializar poblacion
29     population = initialize_population(pop_size, num_variables, lower_bound,
           upper_bound)
30     fitness = np.array([objective_func(ind) for ind in population])
31
32     best_fitness_history = []
33     best_x1_history = []
34     best_x2_history = []
35
36     # Para animacion: almacenamos el mejor (x1, x2) en cada generacion
37     best_solutions_over_time = np.zeros((num_generations, num_variables))
38
39     for gen in range(num_generations):
40         # Elitismo: guardar el mejor de la generacion actual
41         best_index = np.argmin(fitness)
42         best_fitness = fitness[best_index]
43         elite = population[best_index].copy()
44
45         best_fitness_history.append(best_fitness)
46         best_x1_history.append(elite[0])
47         best_x2_history.append(elite[1])
48         best_solutions_over_time[gen, :] = elite
49
50         new_population = []
51
52         # Numero de padres necesarios (2 por cada par a generar)
53         num_parents_needed = 2 * (pop_size - 1)
54         winners, _ = vectorized_tournament_selection(fitness, num_parents_needed,
55                                                      tournament_size,
                                                             len(population),
56                                                      unique_in_column=True,
                                                             unique_in_row=False)
57
58         # Generar un valor global para el crossover y otro para la mutacion (para
               toda la generacion)
59         global_u = np.random.rand()
60         global_r = np.random.rand()
61
62         # Generar nueva poblacion
63         for i in range(0, len(winners), 2):
64             parent1 = population[winners[i]].copy()
65             if i + 1 < len(winners):
66                 parent2 = population[winners[i+1]].copy()
67             else:
68                 parent2 = parent1.copy()
69
70             # Cruzamiento SBX usando el mismo u para todas las variables del cruce
71             child1, child2 = sbx_crossover_with_boundaries(
72                 parent1, parent2, lower_bound, upper_bound,
73                 eta_c, crossover_prob, use_global_u=True, global_u=global_u
74             )
```

```
            # Mutacion polinomial usando el mismo r para todas las variables del individuo
            child1 = polynomial_mutation_with_boundaries(
                child1, lower_bound, upper_bound,
                mutation_prob, eta_mut, use_global_r=True, global_r=global_r
            )
            child2 = polynomial_mutation_with_boundaries(
                child2, lower_bound, upper_bound,
                mutation_prob, eta_mut, use_global_r=True, global_r=global_r
            )

            new_population.append(child1)
            if len(new_population) < pop_size - 1:
                new_population.append(child2)

        # Convertir a array y evaluar el fitness de la nueva poblacion
        new_population = np.array(new_population)
        new_fitness = np.array([objective_func(ind) for ind in new_population])

        # Incorporar el individuo elite (elitismo)
        new_population = np.vstack([new_population, elite])
        new_fitness = np.append(new_fitness, best_fitness)

        # Actualizar la poblacion y su fitness para la siguiente generacion
        population = new_population.copy()
        fitness = new_fitness.copy()

    # Calcular estadisticas finales
    best_index = np.argmin(fitness)
    worst_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_value = fitness[best_index]
    worst_solution = population[worst_index]
    worst_value = fitness[worst_index]
    avg_solution = np.mean(population, axis=0)
    avg_value = np.mean(fitness)
    std_value = np.std(fitness)

    return (best_solution, best_value,
            worst_solution, worst_value,
            avg_solution, avg_value,
            std_value,
            best_fitness_history,
            best_x1_history,
            best_x2_history,
            population,
            fitness,
            best_solutions_over_time)
```

Listing B.3: Implementación de `AG.py`

B.4 Archivo `selection.py`

```
import numpy as np

def vectorized_tournament_selection(fitness, num_tournaments, tournament_size, pop_size,
                                    unique_in_column=True, unique_in_row=False):
    """
    Genera una matriz de torneos de forma vectorizada y retorna, para cada torneo,
    el indice del individuo ganador (el de menor fitness).

    Args:
      - fitness: array con los fitness de la poblacion (longitud = pop_size).
      - num_tournaments: numero de torneos a realizar (por ejemplo, el numero total
                         de selecciones de padres requeridas en la generacion).
      - tournament_size: numero de individuos que participan en cada torneo.
      - pop_size: tamano de la poblacion.
```

```
      - unique_in_column: si True, para cada posicion (columna) se eligen candidatos sin
                          repeticion entre torneos.
      - unique_in_row: si True, en cada torneo (fila) los candidatos seran unicos.
                       (Por defecto se permite repetir en la fila).

    Returns:
      - winners: array de indices ganadores (uno por torneo).
      - tournament_matrix: la matriz de candidatos (de tamano [num_tournaments x tournament_size]).
    """
    if unique_in_row:
        # Para cada torneo (fila), muestreamos sin reemplazo (cada fila es unica)
        tournament_matrix = np.array([np.random.choice(pop_size, size=tournament_size, replace=False)
                                      for _ in range(num_tournaments)])
    else:
        # Permitir repeticion en la fila, pero controlar la no repeticion en cada columna
        if unique_in_column:
            # Para cada columna, se genera una permutacion de los indices (o se usan numeros aleatorios sin repeticion)
            # Siempre que num_tournaments <= pop_size.
            if num_tournaments > pop_size:
                # Si se requieren mas torneos que individuos, se hace sin la restriccion por columna.
                tournament_matrix = np.random.randint(0, pop_size, size=(num_tournaments, tournament_size))
            else:
                cols = []
                for j in range(tournament_size):
                    # Para la columna j, se toman num_tournaments indices sin repeticion
                    perm = np.random.permutation(pop_size)
                    cols.append(perm[:num_tournaments])
                tournament_matrix = np.column_stack(cols)
        else:
            # Sin restricciones, se muestrea con reemplazo para cada candidato.
            tournament_matrix = np.random.randint(0, pop_size, size=(num_tournaments, tournament_size))

    # Para cada torneo (fila de la matriz), se selecciona el candidato con el menor fitness.
    winners = []
    for row in tournament_matrix:
        row_fitness = fitness[row]
        winner_index = row[np.argmin(row_fitness)]
        winners.append(winner_index)
    winners = np.array(winners)
    return winners, tournament_matrix
```

Listing B.4: Implementación de `selection.py`

B.5 Archivo `crossover.py`

```
import numpy as np

def sbx_crossover(parent1, parent2, lower_bound, upper_bound, eta, crossover_prob):
    """Realiza el cruzamiento SBX para dos padres y devuelve dos hijos."""
    child1 = np.empty_like(parent1)
    child2 = np.empty_like(parent2)

    if np.random.rand() <= crossover_prob:
        for i in range(len(parent1)):
            u = np.random.rand()
            if u <= 0.5:
                beta = (2*u)**(1/(eta+1))
            else:
                beta = (1/(2*(1-u)))**(1/(eta+1))
```

```
15
16              # Genera los dos hijos
17              child1[i] = 0.5*((1+beta)*parent1[i] + (1-beta)*parent2[i])
18              child2[i] = 0.5*((1-beta)*parent1[i] + (1+beta)*parent2[i])
19
20              # Asegurar que los hijos esten dentro de los limites
21              child1[i] = np.clip(child1[i], lower_bound[i], upper_bound[i])
22              child2[i] = np.clip(child2[i], lower_bound[i], upper_bound[i])
23      else:
24          child1 = parent1.copy()
25          child2 = parent2.copy()
26
27      return child1, child2
28
29  def sbx_crossover_with_boundaries(parent1, parent2, lower_bound, upper_bound,
30                                    eta, crossover_prob, use_global_u=False,
                                          global_u=None):
31      """
32      Realiza el cruzamiento SBX con limites, usando formulas que ajustan beta en
            funcion
33      de la cercania a las fronteras. Permite usar un unico 'u' global para todos
            los individuos
34      de la generacion o, de forma estandar, un 'u' distinto por cada gen.
35
36      Args:
37        - parent1, parent2: arrays con los padres.
38        - lower_bound, upper_bound: arrays con los limites inferiores y superiores.
39        - eta: indice de distribucion para SBX.
40        - crossover_prob: probabilidad de aplicar el cruce.
41        - use_global_u: si es True se utilizara el mismo valor de 'u' para todas las
              variables.
42        - global_u: valor de 'u' que se aplicara globalmente (si se proporciona).
43
44      Returns:
45        - child1, child2: arrays con los hijos resultantes.
46      """
47      parent1 = np.asarray(parent1)
48      parent2 = np.asarray(parent2)
49      child1 = np.empty_like(parent1)
50      child2 = np.empty_like(parent2)
51
52      # Si no se realiza el crossover, retornamos copias de los padres.
53      if np.random.rand() > crossover_prob:
54          return parent1.copy(), parent2.copy()
55
56      # Si se quiere usar un 'u' global y no se ha pasado, se genera uno.
57      if use_global_u:
58          if global_u is None:
59              global_u = np.random.rand()
60
61      for i in range(len(parent1)):
62          x1 = parent1[i]
63          x2 = parent2[i]
64          lb = lower_bound[i]
65          ub = upper_bound[i]
66
67          # Aseguramos que x1 sea menor o igual que x2
68          if x1 > x2:
69              x1, x2 = x2, x1
70
71          dist = x2 - x1
72          if dist < 1e-14:
73              child1[i] = x1
74              child2[i] = x2
75              continue
76
77          # Calcular la minima distancia a las fronteras
78          min_val = min(x1 - lb, ub - x2)
79          if min_val < 0:
80              min_val = 0
81
82          beta = 1.0 + (2.0 * min_val / dist)
83          alpha = 2.0 - beta**(-(eta+1))
```

```
84
85          # Si se usa u global, se usa el mismo valor para cada variable
86          if use_global_u:
87              u = global_u
88          else:
89              u = np.random.rand()
90
91          if u <= (1.0 / alpha):
92              betaq = (alpha * u)**(1.0/(eta+1))
93          else:
94              betaq = (1.0 / (2.0 - alpha*u))**(1.0/(eta+1))
95
96          # Calcular los hijos
97          c1 = 0.5 * ((x1 + x2) - betaq * (x2 - x1))
98          c2 = 0.5 * ((x1 + x2) + betaq * (x2 - x1))
99
100         # Ajustar a los limites
101         child1[i] = np.clip(c1, lb, ub)
102         child2[i] = np.clip(c2, lb, ub)
103
104     return child1, child2
```

Listing B.5: Implementación de `crossover.py`

B.6 Archivo `mutation.py`

```
1  import numpy as np
2
3  def polynomial_mutation(child, lower_bound, upper_bound, mutation_prob, eta_mut):
4      """Aplica mutacion polinomial a un hijo."""
5      mutant = child.copy()
6      for i in range(len(child)):
7          if np.random.rand() < mutation_prob:
8              r = np.random.rand()
9              diff = upper_bound[i] - lower_bound[i]
10             if r < 0.5:
11                 delta = (2*r)**(1/(eta_mut+1)) - 1
12             else:
13                 delta = 1 - (2*(1-r))**(1/(eta_mut+1))
14             mutant[i] = child[i] + delta * diff
15             mutant[i] = np.clip(mutant[i], lower_bound[i], upper_bound[i])
16     return mutant
17
18
19 def polynomial_mutation_with_boundaries(child, lower_bound, upper_bound,
20                                         mutation_prob, eta_mut,
21                                         use_global_r=False, global_r=None):
22     """
23     Aplica mutacion polinomial (con limites) a un vector 'child'.
24     Puede usar un unico 'r' global para todas las variables (si use_global_r=True)
25     o generar un 'r' distinto para cada variable.
26
27     Args:
28       - child : array-like
29           Cromosoma (vector de decision) a mutar.
30       - lower_bound, upper_bound : array-like
31           Limites inferiores y superiores para cada variable.
32       - mutation_prob : float
33           Probabilidad de mutacion (en [0,1]) para cada variable.
34       - eta_mut : float
35           indice de distribucion para la mutacion.
36       - use_global_r : bool
37           Si True, se utiliza un unico valor 'r' para todas las variables.
38       - global_r : float, opcional
39           Valor de 'r' global a usar; si no se proporciona, se genera uno.
40
41     Retutrns:
42       - mutant : np.ndarray
43           Nuevo vector mutado (manteniendo la dimension de 'child').
```

```
    """
    mutant = np.array(child, copy=True, dtype=float)
    num_vars = len(child)

    # Si se desea usar un 'r' global y no se ha proporcionado, se genera uno una sola vez.
    if use_global_r:
        if global_r is None:
            global_r = np.random.rand()

    for i in range(num_vars):
        # Decidir si mutar esta variable
        if np.random.rand() < mutation_prob:
            x = mutant[i]
            xl = lower_bound[i]
            xu = upper_bound[i]

            # Evitar division por cero si los limites son casi iguales
            if abs(xu - xl) < 1e-14:
                continue

            # d = distancia normalizada al limite mas cercano
            d = min(xu - x, x - xl) / (xu - xl)

            # Elegir r: global o individual para cada variable
            if use_global_r:
                r = global_r
            else:
                r = np.random.rand()

            nm = eta_mut + 1.0

            # Calcular delta_q segun el valor de r
            if r < 0.5:
                bl = 2.0 * r + (1.0 - 2.0 * r) * ((1.0 - d) ** nm)
                delta_q = (bl ** (1.0 / nm)) - 1.0
            else:
                bl = 2.0 * (1.0 - r) + 2.0 * (r - 0.5) * ((1.0 - d) ** nm)
                delta_q = 1.0 - (bl ** (1.0 / nm))

            # Calcular la nueva posicion y asegurarse que este dentro de los limites
            y = x + delta_q * (xu - xl)
            mutant[i] = np.clip(y, xl, xu)

    return mutant
```

Listing B.6: Implementación de `mutation.py`

B.7 Archivo `auxiliares_functions.py`

```
import numpy as np
# ----------------------------
# Funciones auxiliares del GA
# ----------------------------
def initialize_population(pop_size, num_variables, lower_bound, upper_bound):
    """Inicializa la poblacion uniformemente en el espacio de busqueda."""
    return np.random.uniform(low=lower_bound, high=upper_bound, size=(pop_size, num_variables))
```

Listing B.7: Implementación de `auxiliares_functions.py`

Apéndice C

Registro de indicadores completo.

C.1 Problema 01

C.1.1. Resumenes

Cuadro C.1: Archivo: resumen_run_1

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.0	0.0326	0.4	0.2718	0.2086	0.089		-0.6718
Media	0.0	0.0333	0.4	0.2747	0.212	0.0909		-0.6162
Peor	0.0	0.0326	0.4	0.3246	0.2868	0.089		1.0369

Cuadro C.2: Archivo: resumen_run_2

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.0	0.0986	0.4	0.0915	0.4	0.0116		-0.6744
Media	0.0	0.1015	0.4	0.0939	0.4	0.0118		-0.6422
Peor	0.0	0.1615	0.4	0.0915	0.4	0.019		-0.1839

Cuadro C.3: Archivo: resumen_run_3

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1137	0.1102	0.4	0.1703	0.1473	0.0598		-0.6397
Media	0.1138	0.1102	0.4	0.1703	0.1473	0.0598		-0.6397
Peor	0.1138	0.1101	0.4	0.1702	0.1473	0.0598		-0.6396

Cuadro C.4: Archivo: resumen_run_4

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.0	0.0565	0.4	0.3714	0.1746	0.0		-0.6891
Media	0.0	0.0538	0.4	0.3707	0.1696	0.0		-0.6143
Peor	0.0	0.0565	0.4	0.3714	0.0496	0.0		0.869

Cuadro C.5: Archivo: resumen_run_5

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1285	0.0225	0.4	0.0905	0.1456	0.214	-0.6114	
Media	0.1323	0.0235	0.4	0.0952	0.152	0.22	-0.3514	
Peor	0.2167	0.0225	0.4	0.1527	0.2455	0.214	5.6145	

C.2 Problema 02

C.2.1. Resumenes

Cuadro C.6: Archivo: resumen_run_1

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.2354	0.1652	0.0919	0.1886	0.1775	0.1411	0.0047	
Media	0.2295	0.1619	0.0889	0.1837	0.1739	0.1368	0.2972	
Peor	0.1361	0.1652	0.0361	0.1886	0.1775	0.0558	5.8585	

Cuadro C.7: Archivo: resumen_run_2

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1087	0.1276	0.0557	0.1346	0.2783	0.295	0.0075	
Media	0.1084	0.1271	0.0555	0.1342	0.2779	0.2948	0.0092	
Peor	0.1027	0.1276	0.0557	0.1272	0.2716	0.295	0.0484	

Cuadro C.8: Archivo: resumen_run_3

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.3658	0.1096	0.0605	0.17	0.1544	0.1394	0.0037	
Media	0.3655	0.109	0.0596	0.168	0.1532	0.1376	0.0264	
Peor	0.3658	0.1096	0.0482	0.17	0.1234	0.1113	0.5209	

Cuadro C.9: Archivo: resumen_run_4

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.2503	0.2307	0.0016	0.2188	0.1193	0.1792	0.0022	
Media	0.2525	0.2335	0.0016	0.2217	0.1213	0.182	0.0807	
Peor	0.2959	0.2822	0.0016	0.2738	0.1193	0.1792	2.3144	

Cuadro C.10: Archivo: resumen_run_5

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1344	0.1864	0.055	0.1036	0.315	0.2057	0.0062	
Media	0.1373	0.193	0.0571	0.108	0.318	0.2126	0.311	
Peor	0.1344	0.3179	0.055	0.1036	0.315	0.3428	7.237	

C.3 Problema 03

C.3.1. Resumenes

Cuadro C.11: Archivo: resumen_run_1

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1631	0.258	0.0028	0.2868	0.0393	0.25	0.0042	
Media	0.1679	0.2601	0.0028	0.2883	0.0403	0.2539	0.0936	
Peor	0.2285	0.258	0.0028	0.2868	0.0393	0.3102	1.5809	

Cuadro C.12: Archivo: resumen_run_2

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1866	0.1778	0.056	0.2719	0.1493	0.1583	0.0033	
Media	0.1865	0.1777	0.0559	0.2717	0.1492	0.1582	0.0036	
Peor	0.1834	0.1748	0.056	0.2719	0.1493	0.1583	0.0073	

Cuadro C.13: Archivo: resumen_run_3

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1291	0.2206	0.0217	0.3507	0.0449	0.2331	0.0045	
Media	0.1307	0.2235	0.0221	0.3516	0.0456	0.2361	0.0541	
Peor	0.1291	0.2748	0.0283	0.3659	0.0449	0.2836	1.6081	

Cuadro C.14: Archivo: resumen_run_4

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.2971	0.1112	0.0351	0.2159	0.1974	0.1433	0.0028	
Media	0.2928	0.1067	0.0337	0.2107	0.1888	0.1385	0.4237	
Peor	0.2971	0.026	0.0351	0.0754	0.1974	0.0337	11.2484	

Cuadro C.15: Archivo: resumen_run_5

	Indicador	x0	x1	x2	x3	x4	x5	Fitness
Mejor	0.1909	0.1656	0.0272	0.1949	0.2888	0.1327	0.0037	
Media	0.1858	0.163	0.0269	0.1896	0.2859	0.1306	0.169	
Peor	0.1098	0.0951	0.0272	0.1949	0.2411	0.1327	3.9697	