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 - Practica 3:

Recocido Simulado

Documentación de la práctica: Recocido Simulado

Objetivo de la práctica

Implementar el algoritmo de **Recocido Simulado (Simulated Annealing)** para resolver problemas de optimización mediante una técnica inspirada en los procesos físicos de enfriamiento de metales. Este algoritmo permite escapar de óptimos locales para buscar soluciones más cercanas al óptimo global, lo cual es especialmente útil en espacios de búsqueda complejos o ruidosos.

Código Utilizado

Descripción del funcionamiento del programa

El programa define una clase llamada `Annealing`, que representa un optimizador basado en la técnica de recocido simulado. El objetivo es encontrar una solución óptima dentro de un espacio de búsqueda evaluando iterativamente soluciones vecinas y utilizando una **temperatura** para controlar la aceptación de soluciones peores en ciertas etapas del proceso.

Componentes clave

- **`temp_inicial`**: Temperatura inicial del sistema; determina la probabilidad de aceptar soluciones peores al inicio.
- **`temp_minima`**: Temperatura mínima en la que el proceso de búsqueda se detiene.
- **`optimizer`**: Instancia del gestor de optimización (`OptimizationManager`) que representa la solución actual.
- **`max_iter`**: Número máximo de iteraciones que se ejecutarán durante el proceso.
- **`tasa_enfriamiento`**: Factor por el cual se multiplica la temperatura en cada iteración para reducirla gradualmente.

Método `run()`

Este método implementa la lógica principal del algoritmo:

1. **Inicialización**:
    - Se copia el optimizador actual y se evalúa su función objetivo.
    - Se establece la temperatura inicial.
2. **Iteraciones** (bucle principal):
    - Mientras la temperatura no sea menor que la mínima y no se haya llegado al número máximo de iteraciones:
        - Se genera una solución vecina.
        - Se calcula la diferencia (`delta`) entre el valor actual y el del vecino.
        - Si el vecino es mejor (`delta > 0`), se acepta.
        - Si el vecino es peor, se acepta con una **probabilidad decreciente** basada en la temperatura.
        - Si la nueva solución es la mejor encontrada hasta el momento, se actualiza como la mejor solución global.
        - La temperatura se reduce multiplicándola por la tasa de enfriamiento.
        - Se registran los resultados de la iteración.
3. **Finalización**:
    - Se devuelve la mejor solución encontrada, su valor de función objetivo y el historial de resultados.

¿Para qué sirve este algoritmo?

El algoritmo de recocido simulado se utiliza en:

- **Optimización global** de funciones con muchos mínimos locales.
- **Sistemas embebidos**, donde se requiere eficiencia y adaptabilidad.
- **Optimización combinatoria**, como el problema del viajante, diseño de redes, o asignación de tareas.
- Problemas donde una solución ligeramente peor puede llevar eventualmente a una mucho mejor.

Ejemplo de ejecución (corrida final)

Imaginemos que se está optimizando una función de desempeño para un sistema embebido. Una salida típica podría lucir así:

```
Iteración 0: resultado actual = 0.492, mejor = 0.492
Iteración 1: resultado actual = 0.505, mejor = 0.505
Iteración 2: resultado actual = 0.500, mejor = 0.505
Iteración 5: resultado actual = 0.524, mejor = 0.524
Iteración 9: resultado actual = 0.543, mejor = 0.543
...
Iteración 49: resultado actual = 0.584, mejor = 0.584
```

Resultado final:

```
# Variables de retorno del método run()
mejor_opt: <instancia de OptimizationManager>
mejor_resultado: 0.584
resultados: <ResultsManager con datos de cada iteración>
```

Requisitos previos

Este programa requiere el uso de las siguientes clases auxiliares:

- `OptimizationManager`: Define el optimizador y su función objetivo.

- `Param`: Clase que representa parámetros a ajustar.
- `ResultManager` y `ResultsManager`: Almacenan el historial de resultados en cada iteración.

Clase OptimizationManager:

```
import random
import math
import copy
from params import Param
from results_manager import ResultsManager

class OptimizationManager:
    """
    Clase encargada de manejar una lista de parámetros y calcular su
estado óptimo
    usando distintos métodos de optimización (búsqueda local, SA, Tabu,
etc.).
    """

    def __init__(self, params_list: list[Param]):
        """
        Inicializa el administrador con una lista de parámetros a
optimizar.

        Args:
            params_list (list[params]): Lista de objetos de tipo `params`.
        """
        self.params_list = params_list
        self.best_solution = None
        self.best_fitnes = float("-inf")

    def show_params(self):
        """
        Muestra en consola todos los parámetros actuales.
        """
        for param in self.params_list:
            param.show()

    def funcion_objetivo(self, detailed=False):
        """
        Calcula la función objetivo (fitness total) de la solución actual.

        Args:
            detailed (bool, optional): Si True, devuelve un resumen
detallado por parámetro.

        Returns:
            float o dict: Fitness total o un diccionario con detalles de
cada parámetro.
        """
        fitness_componentes = []
        fitness_total = 0
```

```
        for param in self.params_list:
            satisfaction = param.calc_satisfaction()
            satisfaccion_ponderada = satisfaction * param.weight

            penalty = 0
            if not param.is_in_range(param.v_actual):
                penalty = param.calc_satisfaction_con_penalty()

            param_fitness = satisfaccion_ponderada - penalty

            fitness_componentes.append(
                {
                    "name": param.name,
                    "satisfaction": satisfaction,
                    "peso": param.weight,
                    "satisfaccion ponderada": satisfaccion_ponderada,
                    "penalty": penalty,
                    "fitness": param_fitness,
                }
            )

            fitness_total += param_fitness

        if detailed:
            return {"fitness_total": fitness_total, "componentes":
fitness_componentes}
        return fitness_total

    def generar_vecinos(self, step_percentage: float = 0.1) -> list[Param]:
        """
        Genera una nueva lista de parámetros vecinos con pequeños cambios
aplicados.

        Args:
            step_percentage (float): Porcentaje del rango a utilizar como
paso de modificación.

        Returns:
            list[params]: Lista de parámetros modificados (vecinos).
        """
        neighbor = copy.deepcopy(self)
        for param in neighbor.params_list:
            valor_vecino = param.generate_neighbor_v_actual(step_percentage)
            param.v_actual = valor_vecino
        return neighbor.params_list

    def generar_optimizer_vecino(self, step_percentage: float):
        """
        Genera una nueva instancia de OptimizationManager con valores
vecinos.

        Args:
            step_percentage (float): Porcentaje del rango a utilizar como
paso de modificación.

        Returns:
            OptimizationManager: Nuevo optimizador con parámetros vecinos.
        """
        neighbor = copy.deepcopy(self)
```

```
        for param in neighbor.params_list:
            valor_vecino = param.generate_neighbor_v_actual(step_percentage)
            param.v_actual = valor_vecino
        return neighbor

    def
cruzar(self,otro_optimizer:'OptimizationManager')->'OptimizationManager':

        hijo = copy.deepcopy(self)
        for i,param in enumerate(hijo.params_list):
            if random.random() < 0.5:
                param.v_actual = self.params_list[i].v_actual
            else:
                param.v_actual = otro_optimizer.params_list[i].v_actual
        return hijo
```

La clase `OptimizationManager` permite gestionar una lista de parámetros (`params_list`) y optimizarlos mediante diferentes métodos de búsqueda de soluciones óptimas. Se enfoca en mejorar la configuración de los parámetros utilizando estrategias como:

- Cálculo de la función objetivo (fitness total).
- Generación de soluciones vecinas para explorar mejores configuraciones.
- Cruce de soluciones para aplicar operadores evolutivos.

2. Atributos de la Clase

python
self.params_list = params_list
self.best_solution = None
self.best_fitnes = float("-inf")

- `params_list`: Almacena la lista de parámetros que serán optimizados.
- `best_solution`: Guarda la mejor solución encontrada durante el proceso de optimización.
- `best_fitnes`: Registra la mejor función objetivo alcanzada.

3. Métodos de la Clase

a) Mostrar parámetros

python
def show_params(self):
    for param in self.params_list:
        param.show()

Imprime en consola los valores actuales de los parámetros en la lista.

b) Cálculo de la función objetivo (fitness)

python
def funcion_objetivo(self, detailed=False):

- Evalúa la calidad de la configuración actual de parámetros.
- Considera penalizaciones si algún parámetro está fuera de su rango permitido.
- Devuelve el valor total de fitness o un detalle por parámetro si `detailed=True`.

c) Generación de soluciones vecinas

python
def generar_vecinos(self, step_percentage: float = 0.1) -> list[Param]:

- Genera pequeñas modificaciones en los parámetros actuales.
- Usa `copy.deepcopy(self)` para duplicar la instancia sin modificar los valores originales.
- Devuelve una lista de parámetros con valores modificados.

d) Generación de un nuevo optimizador vecino

python
def generar_optimizer_vecino(self, step_percentage: float):

- Similar a `generar_vecinos()`, pero en lugar de devolver una lista de parámetros, devuelve una nueva instancia de `OptimizationManager` con parámetros ajustados.

e) Cruce de soluciones

python
def cruzar(self,otro_optimizer:'OptimizationManager')->'OptimizationManager':

- Opera como un algoritmo genético, combinando valores de dos soluciones (`self` y `otro_optimizer`).
- Para cada parámetro, elige aleatoriamente si toma el valor de `self` o `otro_optimizer`.
- Devuelve una nueva instancia `hijo` que combina ambas soluciones.

Params:

```python
import random


class Param:
    """
    Clase que representa un parámetro a optimizar, incluyendo su rango válido, valor actual,
    peso de importancia, costo de cambio y el modo de optimización ("max" o "min").
    """

    def __init__(self, name: str, min: int, max: int, v_actual: float,
                 weight: float, costo_cambio: float, optim_mode: str):
        """
        Inicializa un nuevo parámetro.

        Args:
            name (str): Nombre del parámetro.
            min (int): Valor mínimo permitido.
            max (int): Valor máximo permitido.
            v_actual (float): Valor actual del parámetro.
            weight (float): Peso o importancia del parámetro.
            costo_cambio (float): Costo asociado a modificar este parámetro.
            optim_mode (str): Modo de optimización: "max" o "min".
        """
        self.name = name
        self.min = min
        self.max = max
        self.v_actual = v_actual
        self.weight = weight
        self.costo_cambio = costo_cambio
        self.optim_mode = optim_mode

    def show(self):
        """Imprime por pantalla la información del parámetro."""
        print(f"Parametro: {self.name}")
        print(f"  Min: {self.min}")
        print(f"  Max: {self.max}")
        print(f"  Valor Actual: {self.v_actual}")
        print(f"  Peso: {self.weight}")
        print(f"  Costo de Cambio: {self.costo_cambio}")
        print(f"  Modo de Optimización: {self.optim_mode}")
        print("  ----------------------")

    def calc_satisfaction(self) -> float:
        """
        Calcula el nivel de satisfacción del valor actual según el modo de optimización.

        Returns:
            float: Satisfacción normalizada entre 0 y 1.
        """
        if self.optim_mode == "max":
            return (self.v_actual - self.min) / (self.max - self.min)
```

```
        elif self.optim_mode == "min":
            return (self.max - self.v_actual) / (self.max - self.min)
        else:
            raise ValueError("Modo de optimización no válido. Debe ser 'max'
o 'min'.")

    def calc_distancia_x_costo(self) -> float:
        """
        Calcula el impacto del costo de cambiar este parámetro.

        Returns:
            float: Valor penalizado por el costo de cambio.
        """
        if self.optim_mode == "max":
            return self.costo_cambio * abs((self.v_actual - self.min) /
(self.max - self.min))
        elif self.optim_mode == "min":
            return self.costo_cambio * abs((self.max - self.v_actual) /
(self.max - self.min))
        else:
            raise ValueError("Modo de optimización no válido. Debe ser 'max'
o 'min'.")

    def calc_quadratic_penalty(self) -> float:
        """
        Calcula una penalización cuadrática si el valor actual está fuera
del rango permitido.

        Returns:
            float: Penalización por violar el rango permitido.
        """
        if self.v_actual < self.min:
            return self.costo_cambio * ((self.min - self.v_actual) /
(self.max - self.min)) ** 2
        elif self.v_actual > self.max:
            return self.costo_cambio * ((self.v_actual - self.max) /
(self.max - self.min)) ** 2
        return 0

    def calc_satisfaction_con_penalty(self) -> float:
        """
        Calcula la satisfacción penalizada si el valor actual está fuera
del rango.

        Returns:
            float: Satisfacción después de aplicar penalización.
        """
        base_satisfaction = self.calc_satisfaction()
        penalty = 0
        if self.v_actual < self.min or self.v_actual > self.max:
            penalty = self.calc_quadratic_penalty()
        return max(0, base_satisfaction - penalty)

    def generate_neighbor_v_actual(self, percentage_step: float) -> float:
        """
        Genera un nuevo valor vecino para el parámetro en base a un
porcentaje del rango.

        Args:
            percentage_step (float): Tamaño del paso en porcentaje del
```

```
rango total.

        Returns:
            float: Nuevo valor vecino dentro de los límites permitidos.
        """
        step_size = (self.max - self.min) * percentage_step
        direction = random.choice([-1, 1])
        valor_vecino = self.v_actual + direction * random.uniform(0,
step_size)
        return max(self.min, min(self.max, valor_vecino))

    def is_in_range(self, value=None) -> bool:
        """
        Verifica si un valor (o el actual) está dentro del rango
permitido.

        Args:
            value (float, optional): Valor a verificar. Por defecto se usa
v_actual.

        Returns:
            bool: True si está dentro del rango, False en caso contrario.
        """
        check_value = value if value is not None else self.v_actual
        return self.min <= check_value <= self.max
```

La clase `Param` tiene como objetivo representar un parámetro configurable dentro de un **proceso de optimización**. Su diseño permite:

- Establecer un rango válido de valores.
- Evaluar el nivel de satisfacción de un valor dentro de su rango.
- Penalizar valores fuera del rango permitido.
- Generar valores vecinos para exploración en algoritmos de optimización.

2. Atributos de la Clase

```
self.name = name
self.min = min
self.max = max
self.v_actual = v_actual
self.weight = weight
self.costo_cambio = costo_cambio
self.optim_mode = optim_mode
```

- `name`: Nombre del parámetro.

- `min`, `max`: Límite inferior y superior del rango permitido.
- `v_actual`: Valor actual del parámetro.
- `weight`: Peso o importancia del parámetro en la función objetivo.
- `costo_cambio`: Coste asociado a modificar el valor del parámetro.
- `optim_mode`: Modo de optimización, que puede ser **"max"** o **"min"**.

3. Métodos de la Clase

a) Mostrar información del parámetro

```
def show(self):
```

Este método imprime los valores actuales del parámetro, mostrando información relevante como su rango y su modo de optimización.

b) Cálculo de satisfacción del parámetro

python
def calc_satisfaction(self) -> float:

- Determina cuán favorable es el valor actual respecto al **modo de optimización**.
- Si el parámetro debe **maximizarse**, la satisfacción se mide en relación con el valor máximo.
- Si debe **minimizarse**, la satisfacción se mide en relación con el valor mínimo.

c) Cálculo de impacto del costo de cambio

```
def calc_distancia_x_costo(self) -> float:
```

Este método evalúa el **costo de modificar el parámetro**, penalizando cambios significativos en el valor actual respecto a su rango.

d) Penalización por valores fuera del rango

```
def calc_quadratic_penalty(self) -> float:
```

- Si el valor **excede los límites**, se aplica una penalización cuadrática proporcional a la magnitud del desajuste.
- La penalización es mayor cuanto más alejado esté el valor de su rango permitido.

e) Cálculo de satisfacción considerando penalización

```
def calc_satisfaction_con_penalty(self) -> float:
```

- Evalúa la satisfacción del parámetro, pero considerando una reducción si el valor está fuera de su rango.
- Usa el método `calc_quadratic_penalty()` para aplicar una corrección cuando el valor es inválido.

f) Generación de valores vecinos

```
def generate_neighbor_v_actual(self, percentage_step: float) -> float:
```

- Crea un nuevo valor dentro del rango permitido, aplicando un **pequeño ajuste aleatorio**.
- El cambio se basa en un porcentaje definido del rango total del parámetro.

g) Validación de valores dentro del rango

```
def is_in_range(self, value=None) -> bool:
```

- Determina si el valor actual (o uno proporcionado) está dentro del rango permitido.
- Devuelve **True** si está dentro de los límites, **False** en caso contrario.

Clase `ResultManager`

Esta clase representa un **resultado individual** de una iteración en el proceso de optimización.

```
import polars as pl
class ResultManager:
    def __init__(self,va:float,vo:float,iteracion:int,modelo:str) -> None:
        self.va = va
        self.vo = vo
        self.iteracion = iteracion
        self.modelo = modelo

    def __str__(self):
        return f"Valor Actual: {self.va}, Valor Objetivo: {self.vo},
Iteracion: {self.iteracion}, Modelo: {self.modelo}"



class ResultsManager:
    def __init__(self,resultados:list[ResultManager]):
        self.resultados = resultados

    def guardar_dato(self,resultado:ResultManager):
        self.resultados.append(resultado)

    def show(self):
        for r in self.resultados:
            print(f"{r}")

    def to_polars(self):
        import polars as pl
        data = []

        for r in self.resultados:
            data.append({
                "Valor Actual":r.va,
                "Valor Objetivo":r.vo,
                "Iteracion":r.iteracion,
                "Modelo":r.modelo
            })

        return pl.DataFrame(data)

    def to_csv(self,filepath:str):
        import polars as pl
        data = []

        for r in self.resultados:
            data.append({
                "Valor Actual":r.va,
                "Valor Objetivo":r.vo,
```

```
                "Iteracion":r.iteracion,
                "Modelo":r.modelo
            })

        df = pl.DataFrame(data)
        try:
            df.write_csv(file=filepath)
            print("Archivo escrito")
        except ValueError:
            print("No se pudo escribir")
```

Atributos

- `va`: Valor actual en la iteración.
- `vo`: Valor objetivo alcanzado.
- `iteracion`: Número de iteración.
- `modelo`: Nombre del modelo de optimización utilizado.

Métodos

- `__str__()`: Devuelve una cadena de texto con los valores del resultado, permitiendo que pueda imprimirse fácilmente.

Clase `ResultsManager`

Esta clase gestiona **una colección de resultados** generados por `ResultManager`.

Atributos

- `resultados`: Lista de objetos `ResultManager`, que almacena los resultados de múltiples iteraciones.

Métodos

- `guardar_dato(resultado)`: Añade un nuevo resultado a la lista.
- `show()`: Imprime en consola todos los resultados almacenados.
- `to_polars()`: Convierte la lista de resultados en un `DataFrame` de `polars`.
- `to_csv(filepath)`: Exporta los resultados a un archivo CSV.

En la función `to_csv()`, se gestiona el intento de escritura con `try-except`, manejando posibles errores al escribir el archivo.

Main Explicación:

```
import polars as pl
class ResultManager:
    def __init__(self,va:float,vo:float,iteracion:int,modelo:str) -> None:
        self.va = va
        self.vo = vo
        self.iteracion = iteracion
        self.modelo = modelo

    def __str__(self):
        return f"Valor Actual: {self.va}, Valor Objetivo: {self.vo},
Iteracion: {self.iteracion}, Modelo: {self.modelo}"



class ResultsManager:
    def __init__(self,resultados:list[ResultManager]):
        self.resultados = resultados

    def guardar_dato(self,resultado:ResultManager):
        self.resultados.append(resultado)

    def show(self):
        for r in self.resultados:
            print(f"{r}")

    def to_polars(self):
        import polars as pl
        data = []

        for r in self.resultados:
            data.append({
                "Valor Actual":r.va,
                "Valor Objetivo":r.vo,
                "Iteracion":r.iteracion,
                "Modelo":r.modelo
            })

        return pl.DataFrame(data)

    def to_csv(self,filepath:str):
        import polars as pl
        data = []

        for r in self.resultados:
            data.append({
                "Valor Actual":r.va,
```

```
                "Valor Objetivo":r.vo,
                "Iteracion":r.iteracion,
                "Modelo":r.modelo
            })

        df = pl.DataFrame(data)
        try:
            df.write_csv(file=filepath)
            print("Archivo escrito")
        except ValueError:
            print("No se pudo escribir")
```

Componentes Importados:

- **Param:** Clase para definir parámetros de optimización.
- **OptimizationManager:** Gestor de soluciones y función objetivo.
- **IteratedLocalSearch:** Implementación del algoritmo ILS.
- **Polars:** Librería para manejo eficiente de datos.
- **Matplotlib:** Para visualización gráfica de resultados.

2. Función Principal (`main()`)

2.1 Configuración de Parámetros del Sistema

```
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")
```

Parámetros Configurados:

Temperatura:

- Rango: 0 a 40
- Valor inicial: 30
- Peso: 0.4
- Costo de cambio: 12
- Modo: Minimización

Humedad:

- Rango: 0 a 100
- Valor inicial: 30
- Peso: 0.4
- Costo de cambio: 12
- Modo: Minimización

Presión:

- Rango: 900 a 1100
- Valor inicial: 1000
- Peso: 0.2
- Costo de cambio: 5
- Modo: Maximización

2.2 Inicialización del Gestor de Optimización

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

Se crea el optimizador principal con los parámetros definidos.

2.3 Configuración del Algoritmo ILS

```
ils = IteratedLocalSearch(
    max_ils_iter=20,        # Número de iteraciones globales
    max_ls_iter=50,         # Iteraciones por búsqueda local
    optimizer=opt,
    ls_vecindad=0.1,        # Radio de vecindad para búsqueda local
    perturbation_strength=0.5  # Intensidad de perturbación
)
```

Parámetros Clave:

- **20 iteraciones globales** del proceso ILS.
- **50 iteraciones por cada búsqueda local**.
- **Vecindad de 0.1** para exploración local precisa.
- **Perturbación de 0.5** para escapes efectivos de óptimos locales.

2.4 Ejecución del Algoritmo

```
best_solution, best_value, resultados = ils.run()
```

Resultados obtenidos:

- `best_solution`: Objeto con la mejor configuración de parámetros.
- `best_value`: Valor óptimo de la función objetivo.
- `resultados`: Historial completo de todas las iteraciones.

2.5 Visualización de Resultados

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

Muestra por consola:

- Parámetros optimizados.
- Valor objetivo alcanzado.

3. Procesamiento y Visualización de Datos

3.1 Conversión 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)
```

Estructura del DataFrame:

- `valor_actual`: Valor en cada iteración.
- `valor_optimo`: Mejor valor histórico.
- `iteracion`: Número de iteración.
- `modelo`: Tipo de algoritmo ("ILS").

3.2 Exportación a CSV

```
df.write_csv("ils_results.csv")
```

Guarda todos los datos en formato CSV para análisis posterior.

3.3 Visualización Gráfica

```
plt.figure(figsize=(10, 6))
plt.plot(df["iteracion"], df["valor_actual"], 'b-', label='Valor Actual')
plt.plot(df["iteracion"], df["valor_optimo"], 'r-', label='Mejor Valor')
plt.xlabel("Iteración")
plt.ylabel("Valor Objetivo")
plt.title("Progreso de Búsqueda Local Iterada")
plt.legend()
plt.grid(True)
plt.savefig("ils_progress.png")
plt.show()
```

Genera gráfico con:

- **Línea azul:** Valor actual por iteración.
- **Línea roja:** Mejor valor histórico.
- **Ejes etiquetados y cuadrícula**.
- **Guardado como imagen PNG**.

4. Ejecución del Programa

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

Garantiza que el script solo se ejecute cuando es invocado directamente.

5. Flujo Completo de Ejecución

Configuración Inicial:

- Define parámetros a optimizar.
- Crea el gestor de optimización.

Optimización:

- Configura algoritmo ILS.
- Ejecuta el proceso iterativo:
    - Búsquedas locales intensivas.
    - Perturbaciones estratégicas.
    - Selección de mejores soluciones.

Post-Procesamiento:

- Muestra resultados por consola.
- Exporta datos a CSV.
- Genera visualizaciones gráficas.

6. Archivos Generados

- **ils_results.csv**: Datos numéricos completos.
- **ils_progress.png**: Gráfico de evolución del algoritmo.

7. Personalización

Para adaptar a otros problemas:

- **Modificar parámetros** (rangos, pesos, modos).
- **Ajustar configuración ILS**:
    - Aumentar iteraciones para problemas complejos.
    - Ajustar tamaño de vecindad según precisión requerida.
    - Modificar fuerza de perturbación según espacio de búsqueda.

Corrida final:

```
------ Mejor solución encontrada ------
Parametro: Temperatura
  Min: 0
  Max: 40
  Valor Actual: 1.4346487803512389
  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: 981.3132886768364
  Peso: 0.2
  Costo de Cambio: 5
  Modo de Optimización: max
  ---------------------
Valor objetivo: 0.866966800873324
```