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 2:

Búsqueda Local Iterada

Documentación del Código: Búsqueda Local Iterada (Iterated Local Search)

1. Objetivo del Algoritmo

El **algoritmo de Búsqueda Local Iterada (Iterated Local Search - ILS)** es una técnica de optimización metaheurística diseñada para encontrar soluciones de alta calidad en espacios de búsqueda complejos. Su propósito principal es:

- Explorar eficientemente el espacio de soluciones mediante búsquedas locales.
- Escapar de óptimos locales mediante perturbaciones controladas.
- Combinar intensificación (exploración profunda de regiones prometedoras) y diversificación (exploración de nuevas regiones).

Este enfoque es particularmente útil en problemas donde:

- El espacio de soluciones es **grande y complejo**.
- Existen **múltiples óptimos locales**.
- La evaluación de la función objetivo es **computacionalmente costosa**.

2. Estructura del Código

```
import copy
import random
from optimization_manager import OptimizationManager
from results_manager import ResultManager, ResultsManager
from params import Param

class IteratedLocalSearch:
   def __init__(self,
                max_ils_iter: int,
                max_ls_iter: int,
                optimizer: OptimizationManager,
                ls_vecindad: float,
                perturbation_strength: float) -> None:
       """
       Inicializa el algoritmo de búsqueda local iterada.

       Args:
           max_ils_iter: Número máximo de iteraciones ILS
```

```
            max_ls_iter: Número máximo de iteraciones de búsqueda local en
cada reinicio
            optimizer: Gestor de optimización con la solución inicial
            ls_vecindad: Tamaño de vecindad para búsqueda local
            perturbation_strength: Fuerza de la perturbación (mayor que
ls_vecindad)
        """
        self.max_ils_iter = max_ils_iter
        self.max_ls_iter = max_ls_iter
        self.optimizer = optimizer
        self.ls_vecindad = ls_vecindad
        self.perturbation_strength = perturbation_strength

    def local_search(self, initial_solution: OptimizationManager):
        """Búsqueda local hasta alcanzar un óptimo local"""
        solucion_actual = copy.deepcopy(initial_solution)
        mejor_solucion = copy.deepcopy(solucion_actual)
        mejor_valor = solucion_actual.funcion_objetivo()

        for i in range(self.max_ls_iter):
            # Generate neighboring solution
            solucion_vecina =
solucion_actual.generar_optimizer_vecino(self.ls_vecindad)
            v_actual = solucion_actual.funcion_objetivo()
            v_vecino = solucion_vecina.funcion_objetivo()

            # If neighbor is better, move to that solution
            if v_vecino > v_actual:
                solucion_actual = solucion_vecina

                # Update best solution if needed
                if v_vecino > mejor_valor:
                    mejor_solucion = copy.deepcopy(solucion_vecina)
                    mejor_valor = v_vecino
            else:
                # If no improvement, we've reached a local optimum
                break

        return mejor_solucion, mejor_valor

    def perturb_solution(self, solution: OptimizationManager):
        """
        Perturba fuertemente la solución para escapar del óptimo local.
        Usa una perturbación más fuerte que la vecindad de búsqueda local.
        """
        perturbed = copy.deepcopy(solution)

        # Apply stronger perturbation to multiple parameters
        for param in perturbed.params_list:
            # Randomly decide if we perturb this parameter (50% chance)
            if random.random() < 0.5:
                continue

            # Perturb with higher strength to escape local optima
            valor_perturbado =
param.generate_neighbor_v_actual(self.perturbation_strength)
            param.v_actual = valor_perturbado

        return perturbed
```

```
    def run(self):
        """Ejecuta el algoritmo de búsqueda local iterada"""
        resultados = ResultsManager([])

        # Initial solution and first local search
        current_solution = copy.deepcopy(self.optimizer)
        best_solution, best_value = self.local_search(current_solution)

        # Record initial results
        r = ResultManager(va=current_solution.funcion_objetivo(),
vo=best_value,
                          iteracion=0, modelo="ILS")
        resultados.guardar_dato(r)

        for i in range(1, self.max_ils_iter + 1):
            # Perturb the current solution
            perturbed_solution = self.perturb_solution(best_solution)

            # Apply local search from the perturbed solution
            candidate_solution, candidate_value =
self.local_search(perturbed_solution)

            # Record results for this iteration
            r = ResultManager(va=candidate_value, vo=best_value,
                              iteracion=i, modelo="ILS")
            resultados.guardar_dato(r)

            # Accept if better (or implement other acceptance criteria)
            if candidate_value > best_value:
                best_solution = copy.deepcopy(candidate_solution)
                best_value = candidate_value
                print(f"ILS: Nueva mejor solución en iteración {i}, valor:
{best_value}")

        return best_solution, best_value, resultados
```

Clase IteratedLocalSearch

Atributos

- `max_ils_iter`: Número máximo de iteraciones del algoritmo ILS.
- `max_ls_iter`: Número máximo de iteraciones para cada búsqueda local.
- `optimizer`: Instancia de **OptimizationManager** que gestiona la solución actual.
- `ls_vecindad`: Tamaño de vecindad para la búsqueda local (**exploración fina**).
- `perturbation_strength`: Fuerza de perturbación para escapar de óptimos locales (**exploración amplia**).

Métodos principales

1. **local_search(self, initial_solution: OptimizationManager)**
    - Realiza una **búsqueda local** desde una solución inicial.
    - Genera soluciones vecinas dentro del rango `ls_vecindad`.
    - Evalúa cada solución vecina y actualiza la mejor solución encontrada.
    - **Retorna** el óptimo local encontrado y su valor.
2. **perturb_solution(self, solution: OptimizationManager)**
    - Aplica una **perturbación fuerte** a la solución actual.
    - Modifica aleatoriamente los parámetros (**50% de probabilidad para cada uno**).
    - Usa `perturbation_strength` para garantizar que la perturbación sea significativa.
    - **Retorna** la solución perturbada lista para una nueva exploración local.
3. **run(self)**
    - **Método principal** que ejecuta el algoritmo completo.
    - Realiza:
        - Una **búsqueda local inicial**.
        - Iteraciones de **perturbación + búsqueda local**.
        - **Registro** de resultados en cada iteración.
    - **Retorna**:
        - La **mejor solución encontrada**.
        - Su **valor objetivo**.
        - Objeto **ResultsManager** con el historial de iteraciones.

3. Flujo del Algoritmo

Inicialización

1. Se parte de una solución inicial.
2. Se realiza una primera **búsqueda local** para encontrar un óptimo local.

Ciclo principal (repite `max_ils_iter` veces)

a. **Perturbación:** Modifica significativamente la mejor solución actual. b. **Búsqueda local:** Explora el vecindario de la solución perturbada. c. **Criterio de aceptación:** Si la nueva solución es mejor, se actualiza la mejor solución. d. **Registro:** Se guardan métricas de la iteración.

Finalización

1. Retorna **la mejor solución encontrada** durante todo el proceso.
2. Proporciona **datos para análisis posteriores**.

4. Ejemplo de Uso

```
# Creación 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")

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

# Configuración del 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,    # Tamaño de vecindad para búsqueda local
    perturbation_strength=0.5  # Fuerza de perturbación
)

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

5. Visualización de Resultados

El código incluye funcionalidad para: **Exportar resultados a CSV** (`ils_results.csv`).

Generar gráficas de progreso (`ils_progress.png`) que muestran:

- **Valor actual** en cada iteración.
- **Mejor valor** encontrado hasta el momento.

6. Requisitos Previos

Para utilizar este código se necesitan las siguientes clases:

- **OptimizationManager**: Gestiona la solución actual y la función objetivo.
- **Param**: Define los parámetros a optimizar.
- **ResultManager y ResultsManager**: Gestionan el almacenamiento de resultados.

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:

```
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
```

```python
    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.

7. Ventajas del Enfoque ILS

Balance entre exploración y explotación: Combina búsqueda local intensiva con perturbaciones para explorar nuevas regiones.

Eficiencia: Reutiliza los óptimos locales encontrados como puntos de partida para nuevas exploraciones.

Flexibilidad: Puede adaptarse a diversos problemas de optimización.

Escalabilidad: Adecuado para problemas con **espacios de búsqueda grandes**.

8. Casos de Aplicación

Este algoritmo es especialmente útil para:

Optimización de parámetros en sistemas embebidos.

Ajuste fino de modelos predictivos.

Problemas de diseño electrónico.

Optimización de recursos en sistemas complejos.

Problemas NP-difíciles donde soluciones aproximadas son aceptables.

Main Explicación:

Codigo Main

```
# Example usage in main.py
from params import Param
from optimization_manager import OptimizationManager
from Unidad3.iterated_ls import IteratedLocalSearch
import polars as pl
import matplotlib.pyplot as plt

def main():
    # Create parameters
    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)

    # Run Iterated Local Search
    ils = IteratedLocalSearch(
        max_ils_iter=20,         # Number of ILS iterations
        max_ls_iter=50,          # Max iterations for each local search
        optimizer=opt,
        ls_vecindad=0.1,         # Small neighborhood for local search
        perturbation_strength=0.5  # Stronger perturbation to escape local
optima
    )

    best_solution, best_value, resultados = ils.run()

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

    # Convert results to Polars 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)

    # Export to CSV
    df.write_csv("ils_results.csv")

    # Plot results
```

```
    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()

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

1. Creación 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")
```

Aquí se definen **tres parámetros** clave para la optimización:

- **Temperatura**, **Humedad** y **Presión** con sus respectivos rangos (`min`, `max`).
- Cada uno tiene un peso (`weight`) en la función objetivo.
- `costo_cambio` representa el costo de modificar el parámetro.
- `optim_mode`: Define si se busca **minimizar** o **maximizar** el valor.

2. Creación del gestor de optimización

```
opt = OptimizationManager(lista_params)
```

Se inicializa un **gestor de optimización** con la lista de parámetros para buscar la mejor configuración posible.

3. Configuración del algoritmo de búsqueda local iterada

```
ils = IteratedLocalSearch(
    max_ils_iter=20,        # Número de iteraciones globales (ILS)
    max_ls_iter=50,         # Iteraciones máximas por búsqueda local
    optimizer=opt,
    ls_vecindad=0.1,        # Pequeña vecindad para exploración local
    perturbation_strength=0.5  # Fuerza de perturbación para salir de óptimos locales
)
```

Este bloque inicializa **Iterated Local Search**, que:

- **Realiza 20 iteraciones** de optimización a nivel global.
- **Cada iteración tiene 50 exploraciones locales**.
- Usa **vecindades pequeñas** (`ls_vecindad=0.1`) para refinar la búsqueda.
- **Perturbación fuerte** (`perturbation_strength=0.5`) para evitar estancarse en óptimos locales.

4. Ejecución del algoritmo y obtención de resultados

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

Se ejecuta la búsqueda local iterada, obteniendo:

- **Mejor solución encontrada**.
- **Valor objetivo óptimo**.
- **Historial de resultados**.

Luego se imprimen los **parámetros optimizados**:

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

best_solution.show_params()

print(f"Valor objetivo: {best_value}")
```

5. Conversión de resultados a DataFrame (Polars)

```
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)
```

Aquí, se transforma la información de cada iteración en un **DataFrame con Polars**, una librería optimizada para manejo de datos en Python.

6. Exportación de resultados a CSV

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

Los resultados son guardados en **un archivo CSV** para análisis posterior.

7. Generación de gráficas con Matplotlib

```
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()
```

Este bloque genera una **gráfica de evolución** del algoritmo, mostrando:

- La variación de los valores en cada iteración.
- Comparación entre los valores actuales y los óptimos.
- Se guarda la imagen (`ils_progress.png`) y se muestra en pantalla.

8. Ejecución del script

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

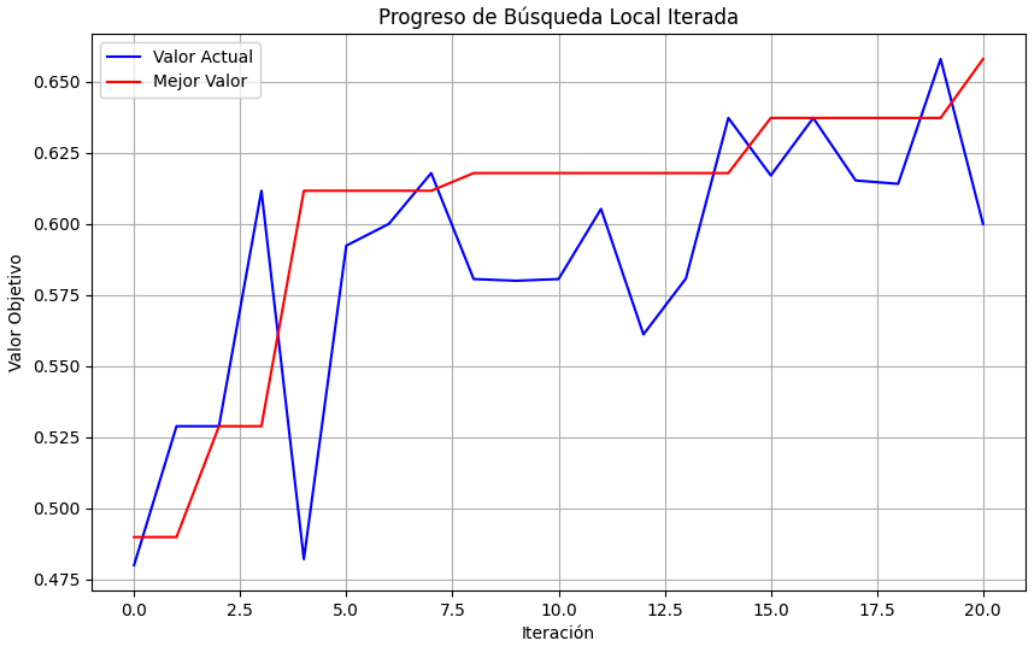
Este fragmento garantiza que la función `main()` se ejecute solo si el script es ejecutado directamente.

Resumen del proceso

1. Se definen **parámetros de optimización** (Temperatura, Humedad, Presión).
2. Se inicializa un **gestor de optimización**.
3. Se configura **Iterated Local Search** con **perturbaciones y exploración local**.
4. Se ejecuta la optimización y se obtiene la **mejor configuración posible**.
5. Se **guardan los resultados en CSV** y se **grafican los progresos**.

Corrida final en python

```
ILS: Nueva mejor solución en iteración 1, valor: 0.534177024280729
ILS: Nueva mejor solución en iteración 2, valor: 0.5615777838239588
ILS: Nueva mejor solución en iteración 4, valor: 0.5818304993582614
ILS: Nueva mejor solución en iteración 6, valor: 0.9110067626642577
ILS: Nueva mejor solución en iteración 8, valor: 0.9986941566230609
ILS: Nueva mejor solución en iteración 19, valor: 1.0
```


Progreso de Búsqueda Local Iterada
Valor Actual
Mejor Valor
Valor Objetivo
Iteración
0.650
0.625
0.600
0.575
0.550
0.525
0.500
0.475
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0