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 1:

Búsqueda Local

Documentación de la práctica: Búsqueda Local:

Objetivo de la práctica

Implementar un algoritmo de **búsqueda local (Local Search)** como técnica de optimización, cuyo propósito es encontrar la mejor solución posible dentro del espacio de soluciones, evaluando iterativamente soluciones vecinas. Este enfoque es útil en contextos donde el espacio de búsqueda es complejo o no completamente conocido, como en problemas de diseño, ajuste de parámetros o sistemas embebidos.

Codigo en python utilizado:

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

class LocalSearch:
   def __init__(self, max_iter: int, optimizer: OptimizationManager,
rango_vecindad: float) -> None:
       self.max_iter = max_iter
       self.optimizer = optimizer
       self.rango_vecindad = rango_vecindad

   def run(self):
       resultados = ResultsManager([])
       solucion_actual = self.optimizer
       mejor_solucion = solucion_actual
       mejor_valor = solucion_actual.funcion_objetivo()

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

           # Record results
           r = ResultManager(va=v_actual, vo=mejor_valor, iteracion=i,
modelo="Busqueda Local")
           resultados.guardar_dato(r)

           # 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 = solucion_vecina
                    mejor_valor = v_vecino
                    print(f"Solucion mejorada en iteracion {i}, valor:
{mejor_valor}")

        return mejor_solucion, mejor_valor, resultados
```

Descripción del funcionamiento del programa

El programa define una clase llamada `LocalSearch`, la cual representa un optimizador que utiliza la técnica de búsqueda local. El objetivo es explorar soluciones cercanas (vecinas) a una solución actual e ir avanzando hacia mejores resultados en función de una **función objetivo**.

Componentes clave

- **`max_iter`**: Número máximo de iteraciones que el algoritmo ejecutará. Define cuánto tiempo buscará una mejor solución.
- **`optimizer`**: Instancia inicial del gestor de optimización (`OptimizationManager`), que representa la solución actual.
- **`rango_vecindad`**: Determina qué tan cerca o lejos están las soluciones vecinas generadas.

Método `run()`

Este método realiza el procedimiento de optimización:

1. **Inicialización**:
    - Se establece la solución inicial como la mejor solución actual.
    - Se evalúa la función objetivo sobre esa solución inicial.
2. **Iteraciones**:

   - En cada iteración, se genera una **solución vecina** a partir de la solución actual.
   - Se evalúan ambas soluciones (actual y vecina).
   - Se registra la iteración con el valor actual y el mejor valor hasta el momento.
   - Si la solución vecina es mejor que la actual, se actualiza la solución.
   - Si además supera a la mejor solución encontrada hasta ahora, se actualiza el mejor valor.

3. **Resultados**:
   - Al final del proceso, se devuelve la mejor solución, su valor y el registro de resultados.

¿Para qué sirve este algoritmo?

El algoritmo de búsqueda local es especialmente útil para:

- **Optimización de parámetros** en modelos donde la evaluación es costosa.
- **Sistemas embebidos**, donde se busca minimizar recursos como consumo energético o latencia.
- **Diseño asistido por computadora**, en ajustes finos de parámetros.
- **Problemas NP-difíciles**, donde encontrar la solución óptima exacta no es factible computacionalmente.

Ejemplo de ejecución (corrida final)

Supongamos que el optimizador está ajustando parámetros para mejorar el rendimiento de un modelo predictivo. Una ejecución típica del algoritmo podría mostrar algo como esto en la consola:

```
Solucion mejorada en iteracion 3, valor: 0.512

Solucion mejorada en iteracion 7, valor: 0.538

Solucion mejorada en iteracion 15, valor: 0.558

Solucion mejorada en iteracion 24, valor: 0.574

Solucion mejorada en iteracion 37, valor: 0.584
```

Y al finalizar:

```
# Resultado esperado (variables de retorno del método run)

mejor_solucion: <instancia de OptimizationManager>

mejor_valor: 0.584

resultados: <ResultsManager con datos de cada iteración>
```

Requisitos previos

Para que este script funcione, es necesario contar con las siguientes clases e implementaciones:

- `OptimizationManager`: clase que contiene la solución actual y la función objetivo.
- `Param`: clase que define los parámetros que se están optimizando.
- `ResultManager` y `ResultsManager`: gestionan el almacenamiento y seguimiento de resultados durante las iteraciones.

Codigo Main

```python
# Example usage in main.py
from params import Param
from optimization_manager import OptimizationManager
```

```python
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 p:

```
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
```