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 4:

Búsqueda Tabú

Búsqueda Tabú (Tabu Search)

Objetivo de la práctica

Implementar un algoritmo de **Búsqueda Tabú**, una técnica de optimización metaheurística que mejora soluciones evitando caer en ciclos o repetir soluciones ya exploradas, mediante el uso de una **lista tabú**. Es especialmente eficaz para escapar de óptimos locales y explorar de forma más eficiente el espacio de búsqueda.

Codigo utilizado:

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

class Tabu:
    def __init__(
        self, tam_tabu: float, optimizer: OptimizationManager, iterations:
int
    ):
        self.lista_tabu = []
        self.tam_tabu = tam_tabu
        self.iterations = iterations
        self.optimizer = optimizer
        self.mejor_fitness_global = float("-inf")
        self.mejor_solucion_global = None

    def isTabu(self, valor: float):
        return valor in self.lista_tabu

    def run(self):
        opt_actual = self.optimizer
        mejor_opt = opt_actual
        f_objetivo_actual = opt_actual.funcion_objetivo(True)
        fitness_actual = f_objetivo_actual.get("fitness_total")
        resultados = ResultsManager([])

        for i in range(self.iterations):
            opt_vecino = opt_actual.generar_optimizer_vecino(0.1)
            f_objetivo_vecino = opt_vecino.funcion_objetivo(True)
            fitness_vecino = f_objetivo_vecino.get("fitness_total")
            r =
ResultManager(va=fitness_actual,vo=self.mejor_fitness_global,modelo="Tabu",
iteracion=i)

            # Update global best solution if needed
            if fitness_vecino > self.mejor_fitness_global:
                self.mejor_fitness_global = fitness_vecino
                self.mejor_solucion_global = opt_vecino
```

```
            if fitness_vecino > fitness_actual:
                if fitness_vecino not in self.lista_tabu:
                    mejor_opt = opt_vecino
                    opt_actual = mejor_opt
                    fitness_actual = fitness_vecino

                    #print(f"\nMejor solución encontrada:
{fitness_vecino}")
                     #print(f"Previa: {fitness_actual}")

                    self.lista_tabu.append(fitness_vecino)
                    if len(self.lista_tabu) > self.tam_tabu:
                        self.lista_tabu.pop(0)


            elif fitness_vecino not in self.lista_tabu:
                mejor_opt = opt_vecino
                opt_actual = mejor_opt
                fitness_actual = fitness_vecino

                self.lista_tabu.append(fitness_vecino)
                if len(self.lista_tabu) > self.tam_tabu:
                    self.lista_tabu.pop(0)
            resultados.guardar_dato(r)


        return self.mejor_solucion_global,resultados
```

Descripción del funcionamiento del programa

El programa define una clase llamada Tabu, que representa un optimizador basado en la estrategia de búsqueda tabú. Este enfoque consiste en moverse iterativamente hacia la mejor solución vecina, pero evitando aquellas que están en la lista tabú (soluciones ya visitadas recientemente).

Componentes clave

- **`tam_tabu`**: Tamaño máximo de la lista tabú. Controla cuántas soluciones recientes se deben evitar.
- **`optimizer`**: Instancia del gestor de optimización (`OptimizationManager`), que representa la solución actual.
- **`iterations`**: Número de iteraciones que ejecutará el algoritmo.

Estructura de datos adicional

- **`lista_tabu`**: Lista que guarda los valores de soluciones (fitness) que no deben considerarse temporalmente, para evitar ciclos.

- **`mejor_fitness_global`**: Guarda el mejor valor de fitness encontrado durante toda la ejecución.

- **`mejor_solucion_global`**: Guarda la mejor solución correspondiente a ese valor de fitness.

Método `run()`

Este método ejecuta el procedimiento principal del algoritmo:

1. **Inicialización**:
    - Se evalúa la función objetivo de la solución inicial.
    - Se almacena su fitness y se inicializa la lista tabú vacía.
2. **Iteraciones**:
    - Se genera una solución vecina.
    - Se evalúa su valor de fitness.
    - Se registra el estado actual en los resultados.
    - Si el vecino tiene mejor fitness que el mejor global, se actualiza la mejor solución global.
    - Si el vecino mejora la solución actual y no está en la lista tabú:
        - Se acepta como nueva solución actual.
        - Se agrega a la lista tabú.
    - Si el vecino no mejora pero tampoco está en la lista tabú:
        - También se acepta como nueva solución, permitiendo cierta exploración del espacio.

- Si la lista tabú excede el tamaño permitido, se elimina el elemento más antiguo.

3. **Finalización**:
    - Se devuelve la mejor solución global encontrada y el historial de resultados.

¿Para qué sirve este algoritmo?

La búsqueda tabú es ideal para:

- **Problemas de optimización combinatoria** como rutas, asignación de recursos, horarios, etc.
- **Sistemas donde se presentan múltiples óptimos locales**, y se desea evitar converger prematuramente.
- **Aplicaciones en inteligencia artificial y sistemas embebidos**, donde se requiere eficiencia y capacidad de adaptación sin repetir patrones de solución.

Ejemplo de ejecución (corrida final)

Una ejecución típica de este algoritmo podría mostrar internamente resultados como los siguientes (si se imprimieran):

java

CopiarEditar

```
Iteración 0: fitness actual = 0.430, mejor global = 0.430

Iteración 1: fitness actual = 0.452, mejor global = 0.452

Iteración 2: fitness actual = 0.445, mejor global = 0.452

Iteración 5: fitness actual = 0.463, mejor global = 0.463

Iteración 8: fitness actual = 0.476, mejor global = 0.476

...
```

Al finalizar el método `run()`:

```
# Variables de retorno del método run()

mejor_solucion_global: <instancia de OptimizationManager>

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

Requisitos previos

El correcto funcionamiento del programa depende de tener implementadas estas clases:

- `OptimizationManager`: Define la solución actual y la función objetivo (debe devolver un diccionario con la clave `"fitness_total"`).
- `Param`: Clase para manejar parámetros del optimizador.
- `ResultManager` y `ResultsManager`: Se encargan de almacenar y manejar el historial de resultados por iteración.

Clase OptimizationManager:

```python
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.).
    """
```

```python
    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

    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
```

```python
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 Tabu:

```
from params import Param
from optimization_manager import OptimizationManager
from tabu import Tabu  # Cambiado a importar Tabu en lugar de
IteratedLocalSearch
import polars as pl
import matplotlib.pyplot as plt


def main():
    # 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")

    lista_params = [param1, param2, param3]
```

```
    opt = OptimizationManager(lista_params)

    # Configuración del algoritmo Tabú
    tabu_search = Tabu(
        tam_tabu=10,  # Tamaño de la lista Tabú
        optimizer=opt,  # Solución inicial
        iterations=100  # Número de iteraciones
    )

    # Ejecución del algoritmo
    best_solution, resultados = tabu_search.run()

    # Obtención del mejor valor objetivo
    best_value = best_solution.funcion_objetivo(True).get("fitness_total")

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

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

    df = pl.DataFrame(data)

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

    # Visualización de resultados
    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 de Fitness")
    plt.title("Progreso de la Búsqueda Tabú")
    plt.legend()
    plt.grid(True)
    plt.savefig("tabu_progress.png")
    plt.show()


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

Estructura del Código

1. Importaciones

```
from params import Param

from optimization_manager import OptimizationManager

from tabu import Tabu

import polars as pl

import matplotlib.pyplot as plt
```

- **Param**: Clase para definir parámetros individuales a optimizar.
- **OptimizationManager**: Gestor principal de la solución y función objetivo.
- **Tabu**: Implementación del algoritmo de Búsqueda Tabú.
- **Polars**: Librería para manejo eficiente de datos.
- **Matplotlib**: Para visualización gráfica de resultados.

2. Función Principal

2.1 Configuración de Parámetros

python

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

Cada parámetro contiene:

- **name**: Identificador descriptivo.
- **min/max**: Rango de valores permitidos.

- **v_actual**: Valor inicial.
- **weight**: Importancia relativa en la función objetivo.
- **costo_cambio**: Penalización por modificación.
- **optim_mode**: Dirección de optimización ("min" o "max").

2.2 Inicialización del Optimizador

python

```
lista_params = [param1, param2, param3]

opt = OptimizationManager(lista_params)
```

Agrupa los parámetros en un gestor de optimización que:

- Evalúa la función objetivo.
- Genera soluciones vecinas.
- Maneja las restricciones.

2.3 Configuración del Algoritmo Tabú

python

```
tabu_search = Tabu(
    tam_tabu=10,
    optimizer=opt,
    iterations=100
)
```

Parámetros clave:

- **tam_tabu=10**: Tamaño máximo de la memoria tabú (evita ciclos).
- **optimizer=opt**: Configuración inicial del problema.
- **iterations=100**: Número máximo de iteraciones.

2.4 Ejecución del Algoritmo

```
best_solution, resultados = tabu_search.run()

best_value = best_solution.funcion_objetivo(True).get("fitness_total")
```

El método `run()` implementa:

- Generación de vecinos.
- Evaluación de soluciones.
- Actualización de la lista tabú.
- Selección de la mejor solución global.

3. Procesamiento de Resultados

3.1 Visualización por Consola

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

best_solution.show_params()

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

Muestra:

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

3.2 Conversión a DataFrame

```
data = [{

    "valor_actual": r.va,
```

```
    "valor_optimo": r.vo,
    "iteracion": r.iteracion,
    "modelo": r.modelo
} for r in resultados.resultados]

df = pl.DataFrame(data)
```

Estructura del dataset:

- **valor_actual**: Fitness en cada iteración.
- **valor_optimo**: Mejor fitness histórico.
- **iteracion**: Paso del algoritmo.
- **modelo**: Identificador ("Tabu").

3.3 Exportación a CSV

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

Guarda el progreso completo para análisis posterior.

3.4 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 de Fitness")
```

```
plt.title("Progreso de la Búsqueda Tabú")

plt.legend()

plt.grid(True)

plt.savefig("tabu_progress.png")

plt.show()
```

Genera un gráfico que muestra:

- Progreso de la búsqueda (línea azul).
- Convergencia al óptimo (línea roja).
- Ejes etiquetados y cuadrícula.

Flujo del Algoritmo

Inicialización

- Crear solución inicial.
- Inicializar lista tabú vacía.

Iteración Principal

- Generar solución vecina.
- Evaluar función objetivo.
- Actualizar mejor solución global.
- Aplicar criterios tabú.
- Actualizar memoria a corto plazo.

Criterio de Terminación

- Máximo de iteraciones alcanzado.
- Convergencia satisfactoria.

Post-procesamiento

- Exportar resultados.
- Generar visualizaciones.

Personalización

Ajuste de Parámetros

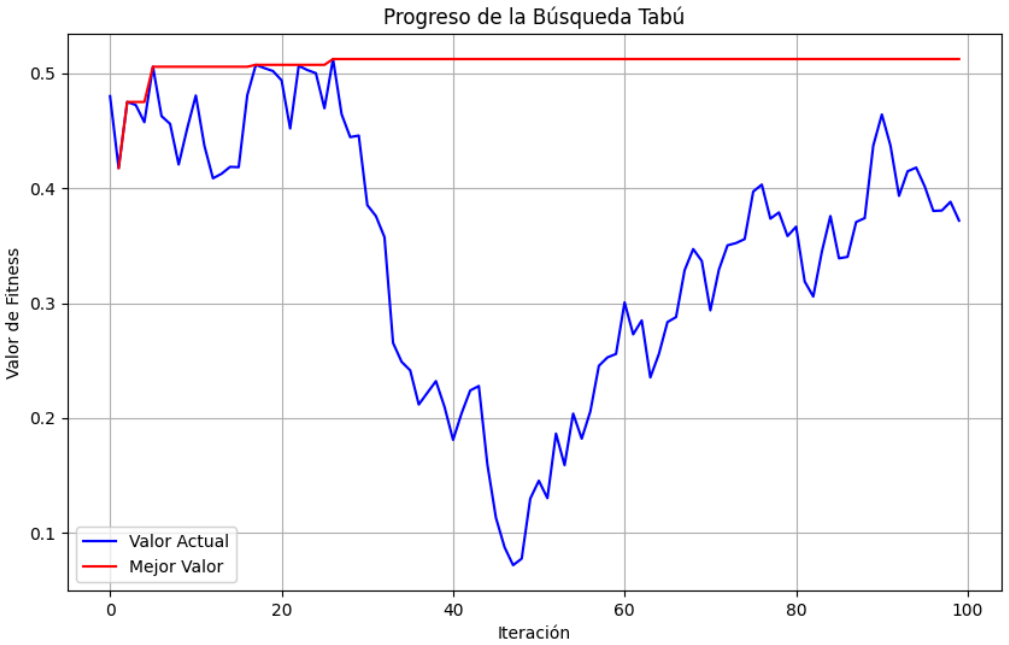
- **Lista Tabú**:
  - Tamaños pequeños (5-10): Búsqueda más agresiva.
  - Tamaños grandes (15-20): Evita ciclos más efectivamente.
- **Iteraciones**:
  - Problemas simples: 50-100.
  - Problemas complejos: 200-500+.
- **Parámetros del Problema**:
  - Modificar rangos según dominio del problema.
  - Ajustar pesos según importancia relativa.

Archivos Generados

- **tabu_results.csv**: Datos numéricos completos.
  - Formato CSV compatible con Excel/Python/R.
  - Contiene el histórico completo de la búsqueda.
- **tabu_progress.png**: Gráfico de convergencia.
  - Imagen PNG de alta calidad.
  - Visualización inmediata del progreso.

Corrida del código

```
------ Mejor solución encontrada ------
Parametro: Temperatura
  Min: 0
  Max: 40
  Valor Actual: 28.632047367584036
  Peso: 0.4
  Costo de Cambio: 12
  Modo de Optimización: min
  ---------------------
Parametro: Humedad
  Min: 0
  Max: 100
  Valor Actual: 21.29200940299173
  Peso: 0.4
  Costo de Cambio: 12
  Modo de Optimización: min
  ---------------------
Parametro: Presion
  Min: 900
  Max: 1100
  Valor Actual: 983.7752782566481
  Peso: 0.2
  Costo de Cambio: 5
  Modo de Optimización: max
  ---------------------
Valor objetivo: 0.5122867669688408
```


Progreso de la Búsqueda Tabú
Valor de Fitness
Iteración
Valor Actual
Mejor Valor
0.1
0.2
0.3
0.4
0.5
0
20
40
60
80
100