Modulo de optimizacion Documentacion

Versión 1.0.5

Kleiver Carrasco

26 de mayo de 2019

Contenido:

Documentacion completa del modulo de optimizacion, aca se describen los modos de uso, la estructura interna, modos de agregar otros algoritmos y problemas, y los docstring del codigo generados de forma automatica con autodoc de sphinx.

CAPÍTULO 1

Requerimientos

- Python >= 3.6
- numpy
- scipy
- matplotlib
- tqdm
- Descargar el repositorio correspondiente al módulo de optimización: https://github.com/ezalorpro/Optimizacion

Los programas a crear deben estar en la misma carpeta del módulo de optimización, esto es importante para poder realizar las importaciones de manera correcta.

CAPÍTULO 2

Inicio

2.1 Ejecución simple

Para poder resolver cualquiera de los problemas se debe empezar por importar el modulo Optimizar el cual posee todas las funciones necesarias para poder establecer el problema, el algoritmo y un post procesado de los resultados:

```
import Optimizar
```

Luego se debe instanciar un optimizador, el cual es un objeto de la clase Optimizar dentro del módulo Optimizar, después, se escoge el problema a resolver y el algoritmo a utilizar, además, se deben cargar los parámetros del algoritmo, esto se puede hacer con la función set_parameters o creando un diccionario con los parámetros correspondientes:

```
optimizador = Optimizar.Optimizar()
Problema = optimizador.set_problem('TSP', vector_len=20, size_space=100, dimension=2)
instanciador, a_name = optimizador.set_algorithm(class_object='AFSA')
Parametros = optimizador.set_parameters('standar_AFSA.txt')
```

Finalmente, inicializamos el algoritmo usando como parámetros a Parametros y Problema, ambos son diccionarios, así que podemos realizar un unpack de ambos, para empezar el proceso de optimización, hacemos un llamado a la función del algoritmo empezar(), a esta función se le puede enviar el parámetro show_results, el cual por defecto es True y habilita la impresión del resultado, para procesar los resultados obtenidos podemos utilizar la función procesar() del optimizador:

```
Algoritmo = instanciador(**Parametros, **Problema)
fitness_evolution, objeto, metrica = Algoritmo.empezar(show_results=True)
optimizador.procesar(fitness_data=fitness_evolution, best_finded=objeto,␣
↪metrica=metrica, algoritmo=a_name)
```

Para el problema del agente viajero los resultados son los siguientes:

```
Tiempo transcurrido: 1.508
Problema: TSP
Ciudades base:[(32, 71), (83, 58), (57, 49), (48, 41), (95, 27), (69, 27), (58, 63),␣
↪(68, 78), (84, 18), (15, 64), (97, 65), (41, 50), (93, 51), (49, 7), (26, 69), (46,␣
↪49), (15, 7), (58, 42), (78, 1), (13, 99)]
```

(continué en la próxima página)

(proviene de la página anterior)

```
Recorrido:[(32, 71), (41, 50), (46, 49), (48, 41), (69, 27), (58, 42), (57, 49), (58,␣
↪63), (68, 78), (83, 58), (97, 65), (93, 51), (95, 27), (84, 18), (78, 1), (49, 7),␣
↪(15, 7), (15, 64), (13, 99), (26, 69), (32, 71)]
Fitness: 425.39469168944396
```

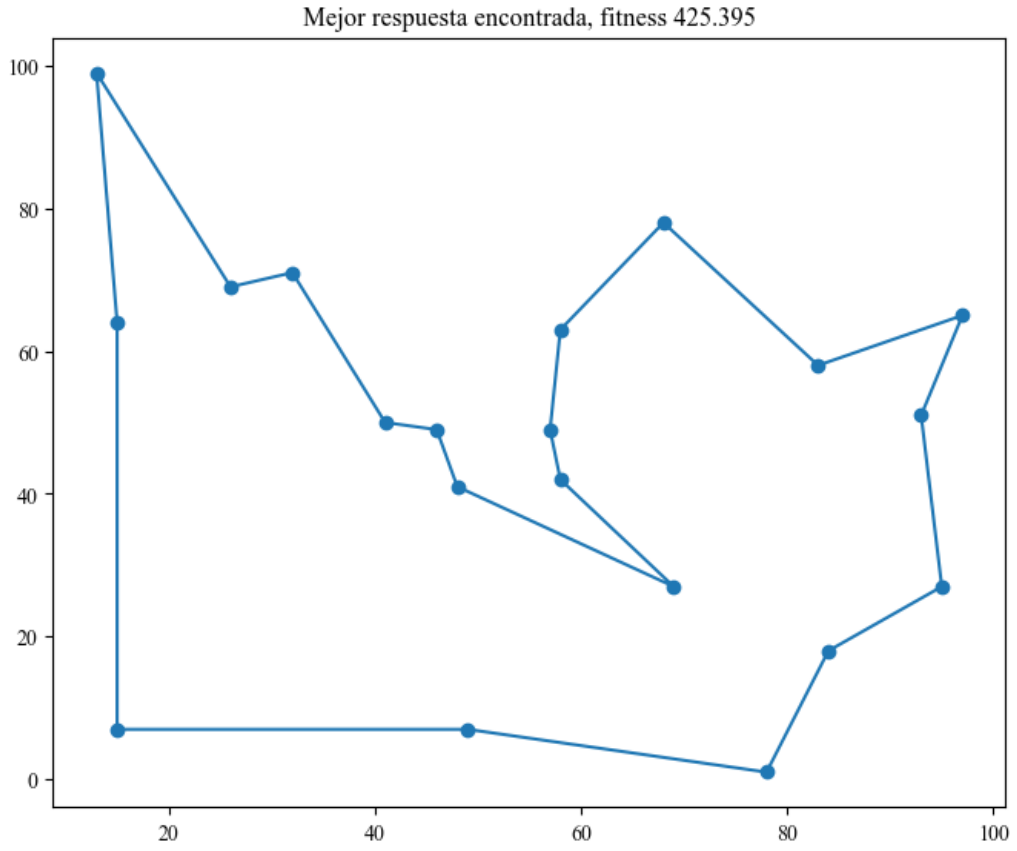

Figura 1: Recorrido encontrado.

Código completo:

```
import matplotlib.pyplot as plt
import Optimizar

optimizador = Optimizar.Optimizar()
Problema = optimizador.set_problem('TSP', vector_len=20, size_space=100, dimension=2)
instanciador, a_name = optimizador.set_algorithm(class_object='AFSA')
Parametros = optimizador.set_parameters('standar_AFSA.txt')

Algoritmo = instanciador(**Parametros, **Problema)
fitness_evolution, objeto, metrica = Algoritmo.empezar(show_results=True)
optimizador.procesar(fitness_data=fitness_evolution, best_finded=objeto,␣
↪metrica=metrica, algoritmo=a_name)

plt.xlabel('iteraciones * poblacion')
plt.ylabel('fitness')
plt.title('fitness vs iteraciones * poblacion')
```

(continué en la próxima página)

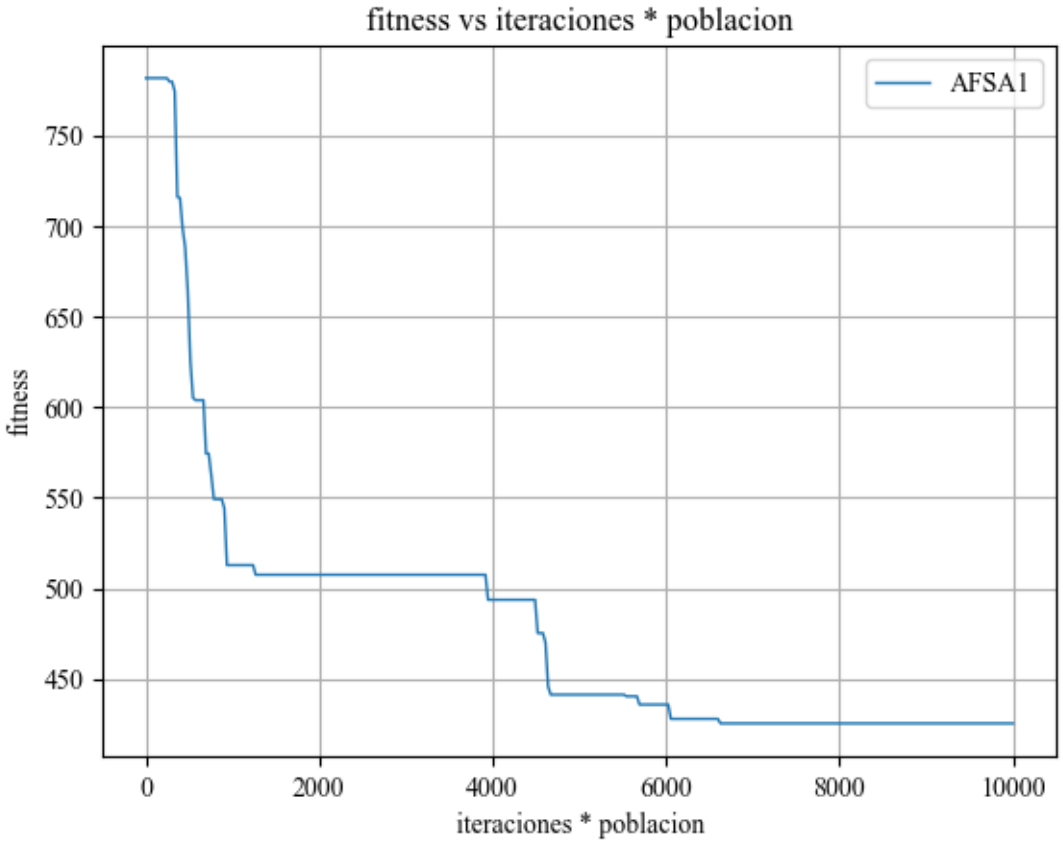

Figura 2: Grafica del fitness.

(proviene de la página anterior)

```
plt.legend()
plt.grid()
plt.show()
```

2.2 Ejecución alterna

Se puede reducir la verbosidad del código si conoce el funcionamiento exacto de los módulos, por tanto, se muestra una forma alterna de realizar el mismo ejemplo anterior:

```
Resultado = instanciador(**Parametros, **Problema).empezar(show_results=True)
optimizador.procesar(*Resultado, a_name)
```

El código final es más compacto a costa de explicites, se deja a manos del lector la elección de cual forma utilizar.

Código completo:

```
import matplotlib.pyplot as plt
import Optimizar

optimizador = Optimizar.Optimizar()
Problema = optimizador.set_problem('TSP', vector_len=20, size_space=100, dimension=2)
instanciador, a_name = optimizador.set_algorithm(class_object='AFSA')
Parametros = optimizador.set_parameters('standar_AFSA.txt')

Resultado = instanciador(**Parametros, **Problema).empezar(show_results=True)
optimizador.procesar(*Resultado, a_name)

plt.xlabel('iteraciones * poblacion')
plt.ylabel('fitness')
plt.title('fitness vs iteraciones * poblacion')
plt.legend()
plt.grid()
plt.show()
```

2.3 Ejecución múltiple en secuencia

Podemos realizar múltiples ejecuciones del mismo algoritmo, bien sea para un mismo problema o para distintos, con el siguiente código se puede ejecutar n veces el algoritmo escogido para un mismo problema, en este caso, 5 veces:

```
best_solution = []
for _ in range(5):    # Numero de ejecuciones secuenciales
        Resultado = instanciador(**Parametros, **Problema).empezar(show_results=True)
        optimizador.procesar(*Resultado, a_name)
        best_solution.append(Resultado[1])

best_solution = General.getbestsolution(best_solution)
print(f'Mejor solucion: {best_solution.position}\ncon fitness de : {best_solution.
↪fitness}')
```

Código completo:

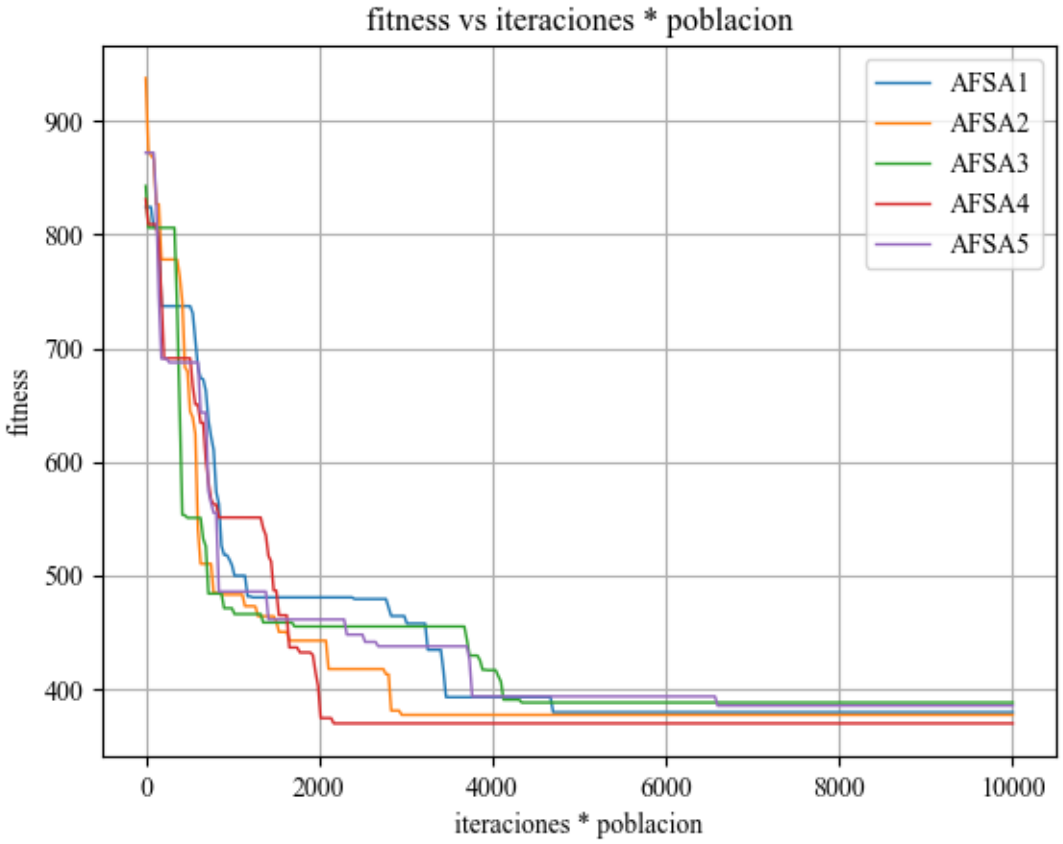

Figura 3: Grafica del fitness para las distintas pruebas.

```
import matplotlib.pyplot as plt
from funciones_generales import General
import Optimizar

optimizador = Optimizar.Optimizar()
Problema = optimizador.set_problem('TSP', vector_len=20, size_space=100, dimension=2)
instanciador, a_name = optimizador.set_algorithm(class_object='AFSA')
Parametros = optimizador.set_parameters('standar_AFSA.txt')

best_solution = []
for _ in range(5):   # Numero de ejecuciones secuenciales
        Resultado = instanciador(**Parametros, **Problema).empezar(show_results=True)
        optimizador.procesar(*Resultado, a_name)
        best_solution.append(Resultado[1])

best_solution = General.getbestsolution(best_solution)
print(f'Mejor solucion: {best_solution.position}\ncon fitness de : {best_solution.
↪fitness}')

plt.xlabel('iteraciones * poblacion')
plt.ylabel('fitness')
plt.title('fitness vs iteraciones * poblacion')
plt.legend()
plt.grid()
plt.show()
```

2.4 Ejecución múltiple en paralelo

Haciendo uso de la librería multiprocessing podemos correr dos algoritmos al mismo tiempo para resolver el mismo problema o uno distinto a gusto, la clave se encuentra en instanciar dos algoritmos:

```
# Se establecen los algoritmos a usar
instanciador1, a_name1 = optimizador.set_algorithm(class_object='AFSA')
instanciador2, a_name2 = optimizador.set_algorithm(class_object='GA')

# Primera opcion para cargar parametros al algoritmo: usando un archivo
Parametros1 = optimizador.set_parameters('standar_AFSA.txt')
Parametros2 = optimizador.set_parameters('standar_GA.txt')
```

Luego se haciendo uso de la librería multiprocessing podemos crear dos colas para retornar la información de los resultados en cada thread, además, se establecen los procesos a correr:

```
# Colas
q1 = Queue()
q2 = Queue()

# Se instancian ambos algoritmos enviando los parametros y el problema haciendo un␣
↪unpack de los diccionarios
ejecutar1 = instanciador1(**Parametros1, **Problema1)
ejecutar2 = instanciador2(**Parametros2, **Problema1)

# Se define la ejecucion de los procesos para ambos algoritmos enviando la cola para␣
↪retornar informacion
p1 = Process(target=ejecutar1.empezar, kwargs=dict(queue=q1, show_results=True,␣
↪position=0))
```

(continué en la próxima página)

(proviene de la página anterior)

```
p2 = Process(target=ejecutar2.empezar, kwargs=dict(queue=q2, show_results=True,␣
↪position=1))
```

Finalmente se ejecutan los procesos y se procesan cuando hayan terminado para luego continuar con el proceso principal utilizando las funciones join():

```
# Se inician los procesos
p1.start()
p2.start()

# Cuando cada algoritmo termine, se obtienen los datos de la cola y se procesan
optimizador.procesar(*q1.get(), a_name1)
optimizador.procesar(*q2.get(), a_name2)

p1.join()
p2.join()
```

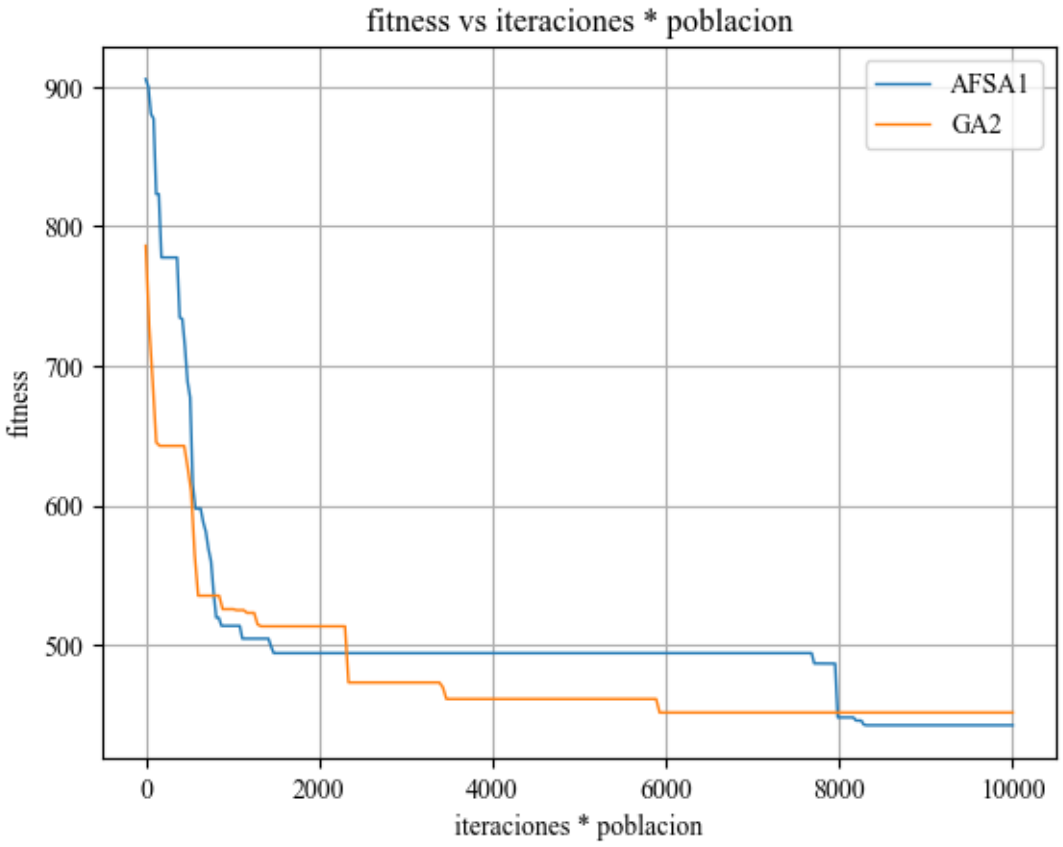

Figura 4: Grafica del fitness de ambos algoritmos.

Código completo:

```
from multiprocessing import Queue, Process
from matplotlib import pyplot as plt
from Optimizar import Optimizar
```

(continué en la próxima página)

(proviene de la página anterior)

```
if __name__ == '__main__':

        # Instanciacion de un Optimizador
        optimizador = Optimizar()

        # Definicion del problema
        Problema1 = optimizador.set_problem('TSP', vector_len=20, size_space=100,␣
↪dimension=2)

        # Se establecen los algoritmos a usar
        instanciador1, a_name1 = optimizador.set_algorithm(class_object='AFSA')
        instanciador2, a_name2 = optimizador.set_algorithm(class_object='GA')

        # Primera opcion para cargar parametros al algoritmo: usando un archivo
        Parametros1 = optimizador.set_parameters('standar_AFSA.txt')
        Parametros2 = optimizador.set_parameters('standar_GA.txt')

        # Colas
        q1 = Queue()
        q2 = Queue()

        # Se instancian ambos algoritmos enviando los parametros y el problema␣
↪haciendo un unpack de los diccionarios
        ejecutar1 = instanciador1(**Parametros1, **Problema1)
        ejecutar2 = instanciador2(**Parametros2, **Problema1)

        # Se define la ejecucion de los procesos para ambos algoritmos enviando la␣
↪cola para retornar informacion
        p1 = Process(target=ejecutar1.empezar, kwargs=dict(queue=q1, show_
↪results=True, position=0))
        p2 = Process(target=ejecutar2.empezar, kwargs=dict(queue=q2, show_
↪results=True, position=1))

        # Se inician los procesos
        p1.start()
        p2.start()

        # Cuando cada algoritmo termine, se obtienen los datos de la cola y se␣
↪procesan
        optimizador.procesar(*q1.get(), a_name1)
        optimizador.procesar(*q2.get(), a_name2)

        p1.join()
        p2.join()

        # Opciones para la grafica creada con optimizador.procesar()
        plt.xlabel('iteraciones * poblacion')
        plt.ylabel('fitness')
        plt.title('fitness vs iteraciones * poblacion')
        plt.legend()
        plt.grid()
        plt.savefig("imagenes\\fitnessgraph.png", bbox_inches='tight', pad_inches=0.1,
↪ format='png')
        plt.show()
```

CAPÍTULO 3

Estructura del modulo

3.1 Optimizar

Es el núcleo del módulo, se encarga de gestionar a los algoritmos y a los problemas, es la clase con la que el usuario interactuara para resolver los problemas, posee 3 funciones básicas:

- set_problem
- set_algorithm
- set_parameters

Los nombres de las funciones son un indicativo de su uso, set_problem se encargará de instanciar un objeto de la clase del problema y retornar las funciones necesarias del mismo, set_algorithm, instanciara un objeto de la clase del algoritmo y retornara el objeto para ser inicializado con sus respectivos parámetros, estos parámetros se pueden leer de un archivo de texto usando la función set_parameters, se presenta como ejemplo el formato para el archivo de configuración del AFSA:

```
maxiteration=1000
visual=4
crowfactor=0.83
n_fish=10
step=1
try_numbers=10
```

Adicionalmente se pretende agregar funciones para el post procesado de los resultados, actualmente la única función de este estilo es procesar( ), la cual grafica el fitness vs métrica utilizando los valores que retorna la función empezar( ) de cada algoritmo o aquellos que se guardan en una cola.

3.2 Algoritmos

Los algoritmos actuales (AFSA, GA) fueron codificados con la idea principal de resolver problemas combinatorios, no obstante, también son capaces de resolver problemas continuos, en el caso del algoritmo genético todo dependerá de

los operadores genéticos que se carguen como parámetros, por otro lado, AFSA puede manejar problemas continuos sin ningún tipo de condición, cabe aclarar que este último no está optimizado para problemas continuos.

Cada algoritmo está compuesto por una clase del mimo nombre, a su vez, cada clase requiere al menos la definición de un constructor (__ini__( )) y de una función llamada Empezar( ) quien iniciara el proceso de optimización, en cada algoritmo también existe una segunda clase básica que hace de objeto solución, sus únicos atributos son el fitness de la solución y la solución, se presenta como ejemplo la clase secundaria de AFSA:

```
class Fish:

        """Clase para definir los objetos tipo pez(Fish)"""

        __slots__ = ('position', 'fitness')

        def __init__(self, p: List[Tuple[float, float]], f: float) -> None:
                self.position = p
                self.fitness = f
```

En orden de agregar más algoritmos al módulo de optimización se recomienda estudiar los ya implementados, sobre todo la función empezar( ) y las funciones de los problemas. La función empezar( ) se puede estructurar de manera básica del siguiente modo:

```
def empezar(self, queue: None = None, show_results: bool = True, position: int = 0) ->
↪ Tuple[List[float], object, int]:
        # variables necesarias
        :
        :
        ciclo for principal:
                ciclo  for secundario:
                        # funciones del algoritmo
                # imprimir % de ejecucion
                # guardado de las mejores soluciones y de la evolucion del fitness
        :
        :
        # mostrar resultados

        if queue is not None:
                paquete = [fitness_evolution, copy.deepcopy(self.getbestsolution(best_
↪solutions)), metrica]
                queue.put(paquete)

        return fitness_evolution, copy.deepcopy(self.getbestsolution(best_solutions)),
↪ metrica
```

Por otro lado, cuando se defina el constructor se debe tomar en cuenta que este deberá poder aceptar todos los parámetros necesarios del algoritmo, adicionalmente deberá aceptar las funciones del problema y el problema base, en este constructor se inicializarán las variables de la clase que deberían corresponder con los parámetros recibidos, todos los parámetros recibidos estarán en formato string, por tanto, se deben transformar al tipo de variable que se requiera, es decir, int, float, list, etc. Una estructura básica para el constructor es la siguiente:

```
def __init__(self,
                parametros,
                problema: List[Tuple[float]] = None,
                funciones: List[Callable] = None) -> None:

        # Repetir para cada parametro
        # self.parametro = necessary_type(parametro)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        self.GroupObject_secundary_class = []

        self.crear_vector, self.calcular_fitness, self.imprimir_respuesta = funciones

        self.problema_base = problema

        for num in range(self.num_objects):
                vector = self.crear_vector(self.problema_base)
                fit = self.calcular_fitness(vector, len(vector))
                Object_ini = Object_secundary_class(copy.copy(vector), copy.copy(fit))
                self.GroupObject_secundary_class.append(Object_ini)

        self.vector_len = len(self.problema_base)
```

El primer segmento del código debería crear los atributos de la clase usando los parámetros recibidos, se debe crear una lista que almacenara los objetos de la clase secundaria i.g., objetos de la clase Fish para AFSA, se deben asignar las funciones del problema a atributos de la clase, con esto, se hará el llamado a las funciones de manera general, lo siguiente es la creación de la población inicial, este paso es requerido por todos los algoritmos (hasta donde conozco), por último, debemos crear el atributo self.vector_len quien nos indicara el tamaño del problema a resolver.

3.3 Problemas

Los problemas pueden ser continuos o combinatorios, cada problema posee dos clases, una para definir al problema cuyo nombre es igual al del archivo, y una segunda clase que complemente a la del problema, esta segunda se encarga de crear un método para obtener las funciones de la clase principal de manera ordenada, esto con el fin de ser utilizado por el modulo Optimizar:

```
class OrderedClassMembers(type):
        @classmethod
        def __prepare__(self, name, bases):
                return collections.OrderedDict()

        def __new__(self, name, bases, classdict):
                classdict['__ordered__'] = [key for key in classdict.keys()
                                                    if key not in ('__module__',
↪'__qualname__')]
                return type.__new__(self, name, bases, classdict)
```

La clase principal del problema debe poseer sin excepción las siguientes funciones:

- __init__(self)
- ini_class_name(self, numero_posiciones, size_space: int, dimension: int, problema: str)
- crear_xxxx(problema_base: List[any]) -> List[any]
- calcular_fitness(solucion: List[any], numero_posiciones: int) -> float
- imprimir_respuesta(problema_base: List[any], mejor_posicion: object, cola: None = None) -> None

El orden de las funciones debe ser el mismo que acá se presenta, cualquiera otra función que se requiera para el problema deberá ser definida después de las ya mencionadas. Los nombres de las funciones no son importantes pero se recomienda utilizar la misma convención para mantener la consistencia entre los problemas.

3.3.1 __init__

Se utiliza para crear atributos de la clase, por ahora este constructor no acepta ningún parámetro, pero se tiene en mente expandir todos los constructores de todos los problemas para aceptar *args and **kwargs con el fin de agregar flexibilidad a la definición de problemas.

3.3.2 ini_class_name

Esta función creará el problema base a resolver, class_name se debe reemplazar con el nombre del problema deberá retornar una lista conteniendo una representación del problema, normalmente esta lista contiene una serie tuples, este es el caso incluso para problemas continuos, donde el retorno es una lista con un tuple de largo uno i.e., [(float,)]. Requiere de 4 parámetros, incluso si no se van a usar, en caso de no ser necesarios se pueden definir usando _, __ , ___ y _____ para dejar claro que no son necesarios.

3.3.3 crear_xxxx

Función crear la población inicial o un nuevo candidato, la idea es que devuelva una solución generada de manera random del problema base, xxxx se debe sustituir con un nombre representativo del problema.

3.3.4 calcular_fitness

Función para calcular el fitness correspondiente al problema, es posiblemente la función principal de la clase, se recomienda optimizar lo mejor posible esta función, pues en muchos casos se requieren múltiples cálculos del fitness, bien sea por requerimientos del algoritmo o de forma general para ir guardando las mejores soluciones, al igual que con el constructor, se tiene la intensión de expandir sus parámetros con el uso de *args y **kwargs para mayor flexibilidad.

3.3.5 imprimir_respuesta

Una función simple para mostrar los resultados obtenidos, en caso de generar algunas grafica con matplotlib, se recomienda usar plt.draw, se invita al lector a revisar los problemas ya definidos para observar como condicionar a esta función para que el problema pueda ser resuelto en múltiples ejecuciones tanto en secuencia como en paralelo con multiprocessing.

CAPÍTULO 4

Documentación automática

4.1 Módulos generales

4.1.1 Optimizar

Clase para utilizar como un modulo de optimizacion, los problemas se deben definir en la carpeta Problemas, cada problema debera tener definida dos clases, una generica para poder extraer las funciones del problema en orden y una clase que define al problema, el nombre del archivo debera ser igual al nombre de la clase que define al problema, esta segunda condicion tambien es necesaria para el caso de los algoritmos, los cuales se deben definir en la carpeta Algoritmos y mantener una estructura aproximada a los algoritmos ya definidos.

Algoritmos:

- "AFSA"
- "GA"

*Nota 1: En la carpeta config_files se pueden modificar los parametros para cada algoritmo, en la misma carpeta se pueden encontrar ejemplos para cada algoritmo.

*Nota 2: A la hora de cargar los parametros del GA, se deben escoger los operadores geneticos adecuadamente segun el tipo de problema, i.e., si el problema es combinatorio o si es continuo.

Ejemplos:

Para problemas combinatorios: ga_operators=("proportional_selection","ordered_crossover","swap_mutation","elitism")

Para problemas continuos: ga_operators=("proportional_selection","heuristic_crossover","continuous_mutation","elitism

Problemas:

Combinatorios:

- "TSP" -> Travelling salesman problem
- "Nqns" -> N queens

*Nota 3: vector_len debe ser > 1, pues indica el numero de ciudades o reinas.

Continuos:

- "Ackley" -> Funcion Ackley
- "Rosenbrock" -> Funcion Rosenbrock
- "Rastrigin" -> Funcion Rastrigin

*Nota 4: Para dimension=3, se graficá la superficie en 3D, se recomienda un size_space bajo no mayor de 30 para apreciar mejor la grafica.

class Optimizar.**Optimizar**

Clase para definir al "modulo" Optimizar

procesar(*fitness_data: List[float], best_finded: object, metrica: int, algoritmo: str*) → None

Funcion de uso opcional, se utiliza para graficar la evolucion del fitness, especialmente util para comparar varios algoritmos, ver ejemplo en Ejecucion_multiple.py

Parámetros

- **fitness_data** – Lista con la evolucion del fitness.
- **best_finded** – Objeto con la mejor solucion encontrada, actualmente sin uso.
- **metrica** – Unidad metrica para el eje de las X, actualmente iteraciones * poblacion.
- **algoritmo** – Nombre del algoritmo para la leyenda.

Devuelve None

static set_algorithm(*class_object: str*) → Tuple[Callable, str]

Funcion para establecer el algoritmo a utilizar, esta funcion puede ser llamada varias veces para instanciar varios algoritmos.

Parámetros **class_object** – Nombre de la clase que define al algoritmo.

Devuelve Un Callable del algoritmo escogido, i.g., AFSA.AFSA, ademas, retorna le nombre del algoritmo.

static set_parameters(*archivo: str = None*) → Dict[str, float]

Funcion extraer los parametros de un archivo de texto.

Formato:

param1=value1 param2=value2 param3=value3

Parámetros **archivo** – Nombre del archivo que contiene los parametros del algoritmo

Devuelve Diccionario con los parametros del algoritmo

static set_problem(*elproblema: str, vector_len: int, size_space: int, dimension: int*) → Any

Funcion para establecer el problema base y las funciones asociadas a la clase de dicho problema.

Parámetros

- **elproblema** – Nombre de la clase que define al problema.
- **vector_len** – Tamaño del problema combinatorio ig. numero de ciudades para TSP.
- **size_space** – Espacio del problema, para ciertos problemas puede que no se use.
- **dimension** – Dimension del problema, i.g., 2 para TSP

Devuelve Diccionario con el problema base y las funciones de la clase del problema

4.1.2 Clase con métodos de uso general

Funciones de uso general, se puede heredar para los algoritmos o usar como modulo de funciones

class funciones_generales.**General**
Clase de funciones de uso general

static calcular_centroide(*grupo: object, problema_base: List[Tuple[int, int]], vector_len: int, vecinos: List[Tuple[int, int]]*) → List[Tuple[float, float]]
Metodo para calcular el centroide

Parámetros **grupo** – Grupo de candidatos

Devuelve xc, el centroide

static calcular_vecinos(*index: int, candidato: List[Tuple[float]], visual: int, grupo: List[object]*) → List[object]
Metodo para calcular los vecinos dado el rango visual

Parámetros

- **index** – Indice del pez actual
- **candidato** – candidato actual
- **visual** – Rango visual a utilizar para el calculo de los vecinos
- **grupo** – Grupo de candidatos

Devuelve vecinos, una lista con los vecinos obtenidos

static getbestsolution(*soluciones: List[object], cantidad: int = 1*) → object
Metodo para obtener la mejor solucion del grupo de soluciones actuales.

Parámetros

- **soluciones** – Grupo de soluciones al que se le obtendra la mejor respuesta basado en el fitness
- **cantidad** – Numero de soluciones a retornar

Devuelve La mejor o mejores soluciones, ie. la solucion o soluciones con menor fitness

4.2 Algoritmos

4.2.1 Algoritmo de la escuela de peces artificiales

Algoritmo de escuela de peces artificiales (AFSA). Se puede utilizar por medio de la clase Optimizar, para mas detalles estudiar el ejemplo Ejecucion_simple.py o Ejecucion_multiple.py

*Nota 1: El AFSA se modifico ligeramente para el caso de los problemas continuos dado que esta planteado para problemas combinatorios, si se quiere un mejor desempeño para estos casos, se recomienda plantear el AFSA para problemas continuos.

class AFSA.**AFSA**(*maxiteration: str = 5000, visual: str = 4, crowfactor: str = 0.83, n_fish: str = 10, step: str = 1, try_numbers: str = 10, problema: List[Tuple[int, int]] = None, funciones: List[Callable] = None*)
Clase base de la escuela de peces artificiales, incluye los movimientos necesarios para el algoritmo.

empezar (*queue: None = None*, *show_results: bool = True*, *position: int = 0*) → Tuple[List[float], AFSA.Fish, int]

Empieza el proceso de optimizacion, hace el llamado para guardar el mejor pez despues de cada iteracion y llama a la funcion de imprimir respuesta.

Parámetros

- **queue** – Variable de multiproceso (cola), se utiliza para devolver informacion al thread principal.
- **show_results** – Bandera para escoger el llamado a imprimir_respuesta().

Devuelve Lista con la evolucion del fitness, objeto Fish con la mejor solucion encontrada, numero de

operaciones realizadas (iteraciones * n_fish).

follow_behavior (*index: int*, *fish: AFSA.Fish*) → int

Se obtiene el mejor pez dentro de los vecinos del pez actual, se asigna su posicion en caso de que el fitness sea mejor y que no haya mucha congestion.

Parámetros

- **index** – Indice del pez actual
- **fish** – Pez actual

Devuelve 0, con fines de acabar la funcion

move_behavior_modificado (*index: int*, *fish: AFSA.Fish*) → int

Se modifico el move_behavior() original propuesto por Yun Cai (2010) el cual consistia en obtener una posicion random y sustituir la posicion del pez actual si dicha posicion generaba un mejor fitness.

Dado que en las pruebas este metodo no genero buenos resultados se opto por realizar un numero step de swaps entre posiciones random, el mejor valor conseguido para step fue de 1 y 2, para mayores valores de step, se tiende a generar una respuesta parecida al move_behavior() original, dado que se termina obteniendo una posicion random

Por otro lado, se agrego una version alterna en caso de que el problema a resolver sea de tipo continuo.

Parámetros

- **index** – Indice del pez actual
- **fish** – Pez actual

Devuelve 0, con fines de acabar la funcion

prey_behavior (*index: int*, *fish: AFSA.Fish*, *visual: int*) → int

Se escoge de manera random un pez vecino, si su fitness es mejor, la posicion del pez actual se reemplaza con la de dicho pez, este proceso se repite try_numbers veces.

Parámetros

- **index** – Indice del pez actual
- **fish** – Pez actual
- **visual** – Rango visual del pez actual

Devuelve 0, con fines de acabar la funcion

swarm_behavior (*index: int*, *fish: AFSA.Fish*) → int

Se calcula el centroide entre los vecinos, al pez actual se le asigna la posicion dada por el centroide si el fitness en dicha posicion es mejor y si la posicion no esta muy congestionada.

Parámetros

- **index** – Indice del pez actual
- **fish** – Pez actual

Devuelve 0, con fines de acabar la funcion

class AFSA.**Fish**(*p: List[Tuple[float, float]], f: float*)
Clase para definir los objetos tipo pez(Fish)

4.2.2 Algoritmo Genético

Algoritmo Genetico. Se puede utilizar por medio de la clase Optimizar, para mas detalles estudiar el ejemplo Ejecucion_simple.py o Ejecucion_multiple.py.

*Nota 1: A la hora de cargar los parametros del GA, se deben escoger los operadores geneticos adecuadamente segun el tipo de problema, i.e., si el problema es combinatorio o si es continuo.

Ejemplos:

Para problemas combinatorios: ga_operators=("proportional_selection","ordered_crossover","swap_mutation","elitism")

Para problemas continuos: ga_operators=("proportional_selection","heuristic_crossover","continuous_mutation","elitism

class GA.**Chromosome**(*p: List[Tuple[int, int]], f: float*)
Clase para definir los objetos tipo cromosoma(Chromosome)

class GA.**GA**(*maxgenerations: str = 1000*, *poppulation_size: str = 100*, *mutation_rate: str = 0.01*, *selection_rate: str = 0.5*, *ga_operators: str = ('proportional_selection', 'ordered_crossover', 'swap_mutation', 'elitism')*, *problema: List[Tuple[int, int]] = None*, *funciones: List[Callable] = None*)
Clase base del Algoritmo Genetico, incluye los operadores geneticos necesarios para el algoritmo.

continuous_mutation(*individuo: GA.Chromosome*, *mutation_rate: float = 0.01*) → GA.Chromosome
Funcion para realizar mutacion agregando un valor random entre -1 y 1 si se cumple la probabilidad de que suceda dicha mutacion dado por el mutation_rate. Esta mutacion solo puede suceder a una posicion del genoma.

Parámetros

- **individuo** – Cromosoma que posiblemente mutará.
- **mutation_rate** – Porcentaje relativo de mutaciones a realizar para una posicion en el genoma del cromosoma.

Devuelve Cromosoma mutado.

elitism(*old_generation: List[GA.Chromosome]*) → List[GA.Chromosome]
Funcion para reducir la poblacion basado en el principio de elitismo, donde solo los cromosomas con mejor fitness sobreviven. EL tamaño de la poblacion luego del crossover se reduce a al tamaño de la poblacion antes del crossover.

Parámetros **old_generation** – Generation anterior + hijos generados del crossover

Devuelve Nueva generacion de un tamaño igual a population_size

empezar(*queue: None = None*, *show_results: bool = True*, *position: int = 0*) → Tuple[List[float], GA.Chromosome, int]
Empieza el proceso de optimizacion, hace el llamado para guardar el mejor cromosoma despues de cada iteracion y llama a la funcion de imprimir respuesta.

Parámetros

- **queue** – Variable de multiproceso (cola), se utiliza para devolver informacion al thread principal.
- **show_results** – Bandera para escoger el llamado a imprimir_respuesta().

Devuelve Lista con la evolucion del fitness, objeto Chromosome con la mejor solucion encontrada, numero de

operaciones realizadas (iteraciones * population_size).

heuristic_crossover(*parent1: GA.Chromosome*, *parent2: GA.Chromosome*) → Tuple[GA.Chromosome, GA.Chromosome]

Funcion para calcular el crossover en el caso de problemas continuos. Se selecciona una posicion random dentro de la dimension del problema, luego, se modifica dicha posicion para el padre y para la madre tomando informacion de ambos y utilizando un factor beta que tomara valores entre 0 y 1.

Formula:

Padre Madre

P(N1, N2, N3,. . . , Nn) y M(N1, N2, N3,. . . , Nn)

Ni_pnew = Ni_p - beta*(Ni_p - Ni_m) Ni_mnew = Ni_m + beta*(Ni_p - Ni_m)

Con Ni como la posicion a modificar con i € Naturales y 0 <= i < dimension

Parámetros

- **parent1** – Cromosoma correspondiente al padre.
- **parent2** – Cromosoma correspondiente a la madre.

Devuelve Cromosomas correspondientes a los hijos generados.

ordered_crossover(*parent1: GA.Chromosome*, *parent2: GA.Chromosome*) → Tuple[GA.Chromosome, GA.Chromosome]

Funcion para generar los hijos, quienes seran los candidatos a formar la siguiente generacion. El proceso consta de tomar dos puntos de cortes en el padre y la madre, la informacion rodeada de dichos cortes se preserva para para los hijos, y el resto se rellena de manera secuencial con la data faltante, si el corte es tomado del padre, el resto de la data se rellena con informacion de la madre, y viceversa.

Ejemplo: cuts = [2, 5) padre 1 2 |3 4 5| 6 7 madre 5 6 |7 4 3| 2 1 ———————

x x |3 4 5| x x x x |7 4 3| x x

child1 6 7 |3 4 5| 2 1 child2 1 2 |7 4 3| 5 6

Parámetros

- **parent1** – Cromosoma correspondiente al padre.
- **parent2** – Cromosoma correspondiente a la madre.

Devuelve Cromosomas correspondientes a los hijos generados.

static proportional_selection(*chromosomes: List[GA.Chromosome], max_value: float*) → GA.Chromosome

Funcion para seleccionar los padres, los padres con un mejor fitness tienen una probabilidad mayor de ser escogidos.

Parámetros

- **chromosomes** – Lista de cromosomas correspondiente a la generacion actual.
- **max_value** – sumatoria total de fitness de la generacion actual.

Devuelve Padre seleccionado.

swap_mutation(*individuo: GA.Chromosome*, *mutation_rate: float = 0.01*) → GA.Chromosome
Funcion para realizar mutacion por medio de un swap, dicho swap se ejecuta para cada posicion del genoma, si se cumple la probabilidad de que suceda dada por el mutation_rate.

Parámetros

- **individuo** – Cromosoma que posiblemente mutará.
- **mutation_rate** – Porcentaje relativo de mutaciones a realizar por cada posicion en el genoma del cromosoma.

Devuelve Cromosoma mutado.

4.3 Problemas

4.3.1 Problema del agente viajero

Problema del agente viajero

class TSP.**OrderedClassMembers**
Meta clase para poder extraer funciones de manera ordenada

class TSP.**TSP**
Clase para definir el problema del agente viajero.

calcular_fitness(*recorrido: List[Tuple[int, int]]*, _) → float
Metodo para calcular el fitness, para este problema de TSP se plantea como funcion fitness la suma de distancias euclidianas entre cada ciudad de manera secuencial partiendo de la ciudad fija y finalizando en la misma.

Parámetros

- **recorrido** – Recorrido a seguir entre las ciudades.
- _ – None

Devuelve fitness

static crear_recorrido(*cities: List[Tuple[int, int]]*) → List[Tuple[int, int]]
Genera el recorrido a realizar

Parámetros **cities** – Ciudades base menos la ciudad inicial, el recorrido de crea a partir de esta.

Devuelve recorrido random

distancia_euclidean
Metodo para el calculo de la distancia euclidiana

Parámetros

- **punto1** – se explica solo
- **punto2** – se explica solo

Devuelve distancia entre el punto 1 y el punto 2

imprimir_respuesta(*problema_base: List[Tuple[int, int]]*, *bestfitnes: object*, *cola: None = None*) → None
Metodo para imprimir una representacion de los resultados obtenidos

Parámetros

- **problema_base** – Ciudades base y por tanto, el problema a resolver
- **bestfitnes** – objeto con el mejor fitness encontrado junto con la posicion para dicho fitness
- **cola** – Para determinar el mostrado de las figuras.

Devuelve None

ini_tsp(*n_ciudades: int*, *size_space: int*, *dimension: int*, *_*) → List[Tuple[int, int]]

Inicializa las ciudades base a resolver. La creacion de las ciudades posee dos formas, uno de manera random, una con forma de circunferencia + ruido, ademas, se agrega en comentarios un set de ciudades fijo para probar el algoritmo de manera mas consistente.

Parámetros

- **n_ciudades** – Numero de ciudades a recorrer
- **size_space** – Tamaño del espacio en el que se ubican las ciudades ig. 100x100
- **dimension** – dimension del problema (x1, x2, x3,. . . ,x_dimension)
- **_** – None

Devuelve Las ciudades generadas

static points_on_circumference(*center=(50*, *50)*, *r=40*, *n=20*)

Metodo para general las ciudades en un patron circular

Parámetros

- **center** – Centro de la circunferencia
- **r** – Radio de la circunferencia
- **n** – Numero de puntos, o sea, numero de ciudades

Devuelve ciudades generadas

4.3.2 Problema de las N reinas

Problema de las N reinas

class Nqns.**Nqns**

Clase para definir el problema de las N reinas.

static calcular_fitness(*reinas_position: List[int], n_reinas: int*) → int

Calculo del fitness para el problema de las N reinas.

Parámetros

- **reinas_position** – Posicion actual en el tablero.
- **n_reinas** – Numero de reinas.

Devuelve fitness dado por el numero total de colisiones.

static crear_tablero(*reinas: List[int]*) → List[int]

Genera la respuesta inicial random.

Parámetros **reinas** – Tablero base.

Devuelve Respuesta inicial random.

static imprimir_respuesta(*tablero: List[int], mejor_posicion: object, cola: None = None*) → None
Metodo para impirmir la mejor posicion en funcion del mejor fitness, ademas, grafica un tablero con la posicion de las reinas.

Parámetros

- **tablero** – Problema base.
- **mejor_posicion** – Mejor posicion encontrada.
- **cola** – Para determinar el mostrado de las figuras.

Devuelve None

static ini_nqns(*n_reinas: int*, _, __, ___) → List[int]
Inicializa el tablero tomando el numero de reinas.

Parámetros

- **n_reinas** – Numero de reinas.
- _ – None
- __ – None
- ___ – None

Devuelve lista desde 0 hasta n_reinas.

class Nqns.**OrderedClassMembers**
Meta clase para poder extraer funciones de manera ordenada

4.3.3 Función Ackley

Funcion Ackley

class Ackley.**Ackley**
Clase para definir el problema de la funcion Ackley.

static calcular_fitness_ackley(*x: List[Tuple[float, float,...,dimension]]*, _) → float
Genera el valor f(x) correspondiente a la funcion Ackley.

Parámetros

- **x** – Punto x que contiene (x1, x2, x3,. . . ,x_n) con n = self.dimension
- _ – None

Devuelve Ackley(x)

crear_x(_) → List[Tuple[float, float,...,dimension]]
Genera un valor real aleatorio en el rango +/- size_space/2 de dimension igual a dimension.

Parámetros _ – None

Devuelve valor real aleatorio.

imprimir_respuesta(_, *mejor_posicion: object*, *cola: None = None*) → None
Imprime la mejor respuesta encontrada, ademas, en caso de dimension=3 (self.dimension = 2), grafica la superficie en 3D y el punto encontrado.

Parámetros

- _ – None

- **mejor_posicion** – Mejor valor encontrado para la funcion Ackley.
- **cola** – Para determinar el mostrado de las figuras.

Devuelve None

ini_funcionesmath(_, *size_space: int*, *dimension: int*, *problema: str*) → List[int]
Para guardar las variables necesarias para el resto de funciones.

Parámetros

- _ – None
- **dimension** – Numero de demensiones a resolver.
- **size_space** – Tamaño del espacio.
- **problema** – Nombre del problema.

Devuelve Lista con len = 1 para establecer el problema como continuo.

class Ackley.**OrderedClassMembers**
Meta clase para poder extraer funciones de manera ordenada

4.3.4 Función Rosenbrock

Funcion Rosenbrock

class Rosenbrock.**OrderedClassMembers**
Meta clase para poder extraer funciones de manera ordenada

class Rosenbrock.**Rosenbrock**
Clase para definir el problema de la funcion Rosenbrock.

static calcular_fitness_rosen(*x: List[Tuple[float, float,...,dimension]]*, _) → float
Genera el valor f(x) correspondiente a la funcion Rosenbrock.

Parámetros

- **x** – Punto x que contiene (x1, x2, x3,. . . ,x_n) con n = self.dimension
- _ – None

Devuelve Rosenbrock(x)

crear_x(_) → List[Tuple[float, float,...,dimension]]
Genera un valor real aleatorio en el rango +/- size_space/2 de dimension igual a dimension.

Parámetros _ – None

Devuelve valor real aleatorio.

imprimir_respuesta(_, *mejor_posicion: object*, *cola: None = None*) → None
Imprime la mejor respuesta encontrada, ademas, en caso de dimension=3 (self.dimension = 2), grafica la superficie en 3D y el punto encontrado.

Parámetros

- _ – None
- **mejor_posicion** – Mejor valor encontrado para la funcion Rosenbrock.
- **cola** – Para determinar el mostrado de las figuras.

Devuelve None

ini_funcionesmath(_, *size_space: int*, *dimension: int*, *problema: str*) → List[int]
Para guardar las variables necesarias para el resto de funciones.

Parámetros

- _ – None
- **dimension** – Numero de demensiones a resolver.
- **size_space** – Tamaño del espacio.
- **problema** – Nombre del problema.

Devuelve Lista con len = 1 para establecer el problema como continuo.

4.3.5 Función Rastrigin

Funcion Rastrigin

class Rastrigin.**OrderedClassMembers**
Meta clase para poder extraer funciones de manera ordenada

class Rastrigin.**Rastrigin**
Clase para definir el problema de la funcion Rastrigin.

static calcular_fitness_rastrigin(*x: List[Tuple[float, float,...,dimension]]*, _) → float
Genera el valor f(x) correspondiente a la funcion Rastrigin.

Parámetros

- **x** – Punto x que contiene (x1, x2, x3,. . . ,x_n) con n = self.dimension
- _ – None

Devuelve Rastrigin(x)

crear_x(_) → List[Tuple[float, float,...,dimension]]
Genera un valor real aleatorio en el rango +/- size_space/2 de dimension igual a dimension.

Parámetros _ – None

Devuelve valor real aleatorio.

imprimir_respuesta(_, *mejor_posicion: object*, *cola: None = None*) → None
Imprime la mejor respuesta encontrada, ademas, en caso de dimension=3 (self.dimension = 2), grafica la superficie en 3D y el punto encontrado.

Parámetros

- _ – None
- **mejor_posicion** – Mejor valor encontrado para la funcion Rastrigin.
- **cola** – Para determinar el mostrado de las figuras.

Devuelve None

ini_funcionesmath(_, *size_space: int*, *dimension: int*, *problema: str*) → List[int]
Para guardar las variables necesarias para el resto de funciones.

Parámetros

- _ – None
- **dimension** – Numero de demensiones a resolver.
- **size_space** – Tamaño del espacio.

- **problema** – Nombre del problema.

Devuelve Lista con len = 1 para establecer el problema como continuo.

CAPÍTULO 5

Indices and tables

- genindex
- modindex

Índice de Módulos Python

Índice

O

P

R

S

T