24 DE OCTUBRE DE 2022

COMPETICIÓN. ALGORITMO EVOLUTIVO MULTIOBJETIVO BASADO EN AGREGACIÓN

EJERCICIO OBLIGATORIO APLICACIONES DE SOFT COMPUTING

IVÁN MORENO GRANADO

UNIVERSIDAD DE SEVILLA

IVAMORGRA@ALUM.US.ES

CONTENIDO

Tabla de contenidos

Versión	Descripción	Fecha
V1.0	Realización de la introducción y detalles de implementación de ZDT3	18/10/22
V1.1	Detalles de implementación de ZDT3 y CF6	19/10/22

Introducción

En este documento se detalla una comparativa entre el algoritmo conocido como NSGAII y el algoritmo basado en agregación. Para ello, se ha implementado este último para posteriormente comparar mediante métricas y estadísticas el algoritmo dado (NSGAII) y el desarrollado (agregación).

Por otro lado, se tienen en cuenta dos problemas multiobjetivo: ZDT3 y CF6. El primer problema vendrá dado por 30 dimensiones, mientras que CF6 se tratará con 4 y 16 dimensiones. Además, ZDT3 no cuenta con restricciones, mientras que CF6 sí tiene restricciones (en concreto dos), por lo que debemos aplicar un mecanismo que se encargue de gestionar estas restricciones.

A continuación, se detallará cada problema:

- ZDT3

  Cuenta con las siguientes funciones:

$$f_1(\boldsymbol{x}) = x_1$$

$$f_2(\boldsymbol{x}) = g(\boldsymbol{x}) \cdot h(f_1(\boldsymbol{x}), g(\boldsymbol{x}))$$

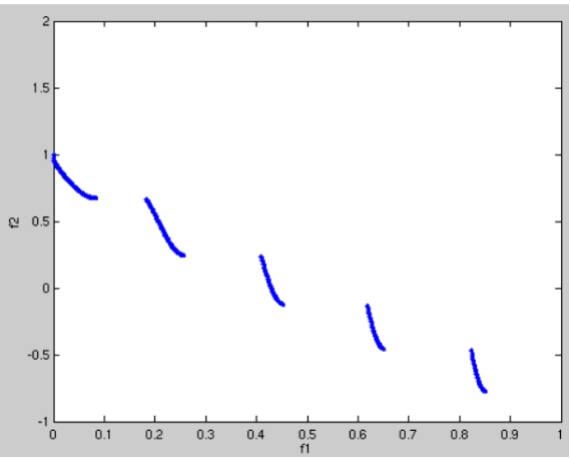
$$h(f_1(\boldsymbol{x}), g(\boldsymbol{x})) = 1 - \sqrt{f_1(\boldsymbol{x})/g(\boldsymbol{x})} - (f_1(\boldsymbol{x})/g(\boldsymbol{x}))\sin(10\pi f_1(\boldsymbol{x}))$$

$$g(\boldsymbol{x}) = 1 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$$

  Con un espacio de búsqueda definido:

$$0 \le x_i \le 1$$

  Además, el problema cuenta con un frente de Pareto óptimo conocido:



- CF6

Cuenta con las siguientes funciones:

$$f_1(\boldsymbol{x}) = x_1 + \sum_{j \in J_1} y_j^2$$
$$f_2(\boldsymbol{x}) = (1 - x_1)^2 + \sum_{j \in J_2} y_j^2$$

donde:

$$J_1 = \{ j \mid j \text{ es impar y } 2 \le j \le n \}$$
$$J_2 = \{ j \mid j \text{ es par y } 2 \le j \le n \}$$

y

$$y_j = \begin{cases} x_j - 0.8 x_1 \cos(6\pi x_1 + \frac{j\pi}{n}) & \text{si } j \in J_1 \\ x_j - 0.8 x_1 \sin(6\pi x_1 + \frac{j\pi}{n}) & \text{si } j \in J_2 \end{cases}$$

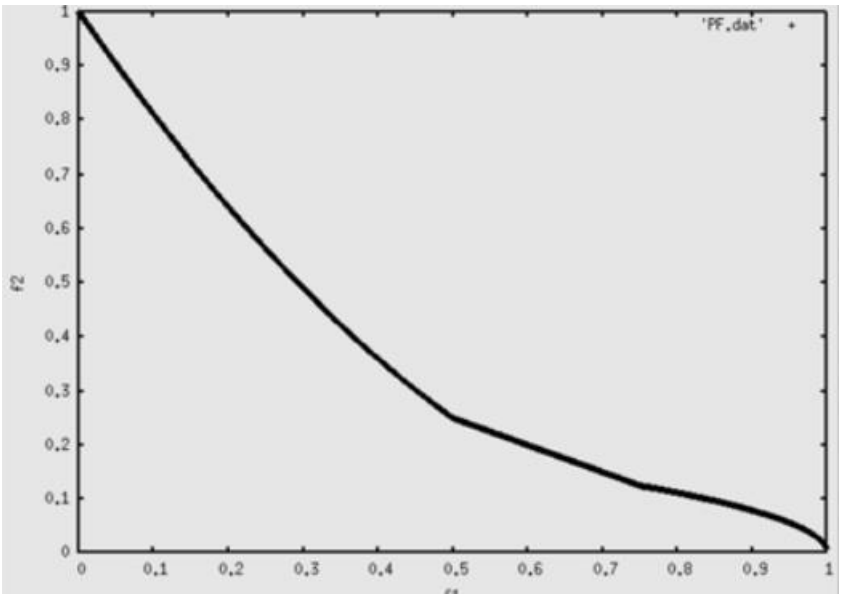
Las restricciones son:

$$x_2 - 0.8 x_1 \sin(6\pi x_1 + \frac{2\pi}{n}) - \text{sgn}(0.5(1 - x_1) - (1 - x_1)^2) \sqrt{|0.5(1 - x_1) - (1 - x_1)^2|} \ge 0$$
$$x_4 - 0.8 x_1 \sin(6\pi x_1 + \frac{4\pi}{n}) - \text{sgn}(0.25\sqrt{(1 - x_1)} - 0.5(1 - x_1)) \sqrt{|0.25\sqrt{1 - x_1} - 0.5(1 - x_1)|} \ge 0$$

El espacio de búsqueda es

$$0 \le x_1 \le 1$$
$$-2 \le x_i \le 2 \quad i > 1$$

Este problema, al igual que ZDT3, cuenta con un frente de Pareto óptimo conocido:



Contenido

1.- Detalles de implementación del algoritmo basado en agregación

1.1.- ZDT3 (30 dimensiones)

Para el desarrollo del algoritmo se ha tenido en cuenta la guía publicada en Enseñanza Virtual vista en clase de teoría, junto con anotaciones de apuntes. El algoritmo se encuentra en el archivo "alg.py". En este archivo se encuentra todo lo relacionado con los métodos auxiliares y las funciones desarrolladas para el desarrollo del algoritmo. Por otro lado, en el archivo "main.py" se encuentra la llamada a dicho algoritmo, además de la creación y escritura de los ficheros de salida de las ejecuciones. En este archivo se ha automatizado la creación de ficheros ".out" para el posterior uso de métricas y la creación y guardado de gráficas resultantes.

Fichero *alg.py:*

Se ha llevado a cabo varias decisiones en alguna de las funciones auxiliares del algoritmo:

- En la función ***normal_distribution*** (encargada de la creación de los vectores lambda) se ha redondeado a cuatro decimales cada uno de los valores que se obtienen al aumentar y disminuir el rango. Esto es debido a que si no se redondea lambda1 no llega al valor 1 (en concreto, se queda a 0.99998…) y lambda2 no llega a 0 (en concreto, llega a 0.0001…), ya que al restar y sumar el ratio con las lambdas, al ser ambos de tipo *float* se pierde parte de la información. Por otro lado, la resta de lambda2 con el ratio se hace en valor absoluto, ya que mediante pruebas se ha observado que lambda2 llega a -0.0, aunque realmente no se ha visto un impacto en el algoritmo debido a este hecho.
- En la función ***distance*** se ha hecho uso del método dist, capaz de calcular la distancia euclídea de dos puntos dados por parámetros. Se ha decidido usar esta función para ahorrarnos líneas de código y así evitar errores no deseados en la implementación.
- Se ha creado una función auxiliar ***get_index*** con el fin de facilitar el acceso al índice (representación del vecino en la población) de un valor dado procedente de una lista. Este método es fundamental para calcular los vectores (vecinos) más cercanos a cada subproblema. Por otro lado, se ha decidido pasar por parámetros una copia de la lista de estas distancias a la función **t_nearest**, encargada de encontrar los vecinos más cercanos de un subproblema. Esta decisión se ha llevado a cabo para evitar modificar los datos de estas distancias y que pueda afectar al algoritmo posteriormente, ya que cada vez que se encuentra el valor mínimo modificamos ese valor a uno muy alto para asegurarnos de que no se va a volver a escoger ese vecino.
- El parámetro t viene dado por el 20% de la población. En la guía viene especificado que dicho porcentaje ronda entre el 10% y 30%, así que se ha optado por escoger un valor intermedio.
- Para la generación de la población aleatoria se ha llevado a cabo la función de Python ***random.uniform***, ya que es una función conocida y probada en otros proyectos.
- En la fase iterativa, para elegir los 3 vectores más cercanos se ha hecho uso de la función **random.choice.** Esta función se encarga de elegir un valor aleatoria de una lista que se le pasa por parámetros.
- Para la evaluación de la función se ha tenido en cuenta que en la inicialización evaluamos directamente a una población entera, mientras que en la iteración vamos

evaluando conforme calculamos un nuevo individuo. Por tanto, existen dos métodos llamados evaluate_population y evaluate_individual, que se encargan de evaluar la población y al individuo, respectivamente.

- Como operadores evolutivos se han utilizado los que vienen en la guía, incluido el operador de mutación de Gauss. Se ha usado como probabilidad de cruce (cr=0.5) ya que es el recomendado, al igual que la probabilidad de mutación (f=0.5). Por otro lado, para obtener un valor aleatorio de la distribución de Gauss en la mutación se ha hecho uso de la función **random.gauss.** El rango que puede tomar este valor va desde 0 hasta la desviación de los cromosomas (teniendo en cuenta el máximo y mínimo valor) del individuo. Además, tal como indica la guía, el valor que toma SIG es 20 (recomendado).
- Por cada cromosoma que vamos obteniendo comprobamos que no se salga del espacio de búsqueda. Para ello comprobamos que no se pasa por abajo. Si se pasa asignamos directamente el valor mínimo de la dimensión del espacio de búsqueda en la que estamos. Lo mismo hacemos si el valor del cromosoma supera el límite alto.
- Para la actualización de vecinos, actualización de z y la función de tchebycheff se ha tenido que comparar toda la configuración de parámetros con respecto a la configuración inicial. En un primer momento, en cada iteración cuando habíamos aplicado los operadores a un individuo en concreto, el algoritmo ni evaluaba ni se actualizaba la z ni los vecinos en ese momento, sino que lo hacía al final cuando ya se aplicaran los operadores a todos los individuos. Por tanto, mediante pruebas y revisión de la guía, se tuvo que cambiar los parámetros teniendo en cuenta que a las funciones nombradas anteriormente no se les pasa la población, sino un individuo en concreto. Este individuo (mutado y cruzado) es el que debemos evaluar.

Fichero *main.py:*

Este fichero recoge todas las funciones que llaman al algoritmo, además de otras auxiliares que ayudan a automatizar el proceso de generación de ficheros y guardado de gráficas.

La jerarquía de estas funciones es la siguiente:

1. La función inicializadora, (se encarga de hacer la primera llamada) es la función **out.** Además de hacer la primera llamada comprueba si el usuario quiere ejecutar el algoritmo una vez de forma aleatoria o diez veces con semillas diferentes. Termina llamando a la función **main**.
2. La función que llama al algoritmo para que se ejecute, **main**. Hay dos diferentes, dependiendo de la ejecución que quiera el usuario hacer, explicado en el punto anterior.
3. Funciones auxiliares de creación de ficheros y creación de gráficas.

Para ello, se ha tomado las siguientes decisiones:

- Se ha añadido dos parámetros nuevos a la función de **main**. Además, estos parámetros se han añadidos a la función de inicialización del fichero alg.py. Estos parámetros reciben el nombre de *unique* y *seed*, de tipo booleano y float respectivamente. *Unique* indica si se quiere que la ejecución sea única y aleatoria. En ese caso, no se tendrá en cuenta la semilla (no existe en los parámetros de la función inicializadora llamada **out**). Si se quiere que no sea única, entonces llamará a la función **main** encargada de ejecutar el algoritmo completo diez veces cada una con una semilla diferente. De este

modo, tendremos siempre diez ejecuciones diferentes cada una con su gráfica final guardada y su fichero *.out* almacenado en la carpeta **outputfiles**.

1.2.- CF6

Para la realización del algoritmo dedicado al problema CF6 se ha tomado como base el algoritmo ZDT3, cambiando algunos parámetros y teniendo en cuenta que hay dos dimensionalidades diferentes (4 y 16 dimensiones) por los que muchos métodos han sido duplicados y configurados cada uno dependiendo de las características del problema. La estructura del código es la misma que la explicada anteriormente para ZDT3. Se cuenta con un archivo **maincf6.py** y **algcf6.py**.Como método de manejo de restricciones se ha optado por función penalti. A continuación, se da detalles de cómo se ha implementado esta función penalti:

- Para evaluar al individuo se hace uso de la función CF6 creada. Una vez calculado los valores f1 y f2 (las dos funciones a minimizar) se comprueba que cumpla o no las restricciones que vienen impuestas en el problema. En el caso en el que las cumpla no se le aplicará ninguna restricción. En cambio, si no se cumplen las restricciones entra en juego las penalizaciones. Es importante saber que esto se comprueba antes de haber obtenido el valor de las dos funciones con el individuo correspondiente. El manejo de restricciones se puede ver en la función **restrictions** encargada de cuantificar la penalización que se le aplicará a cada solución. Este método calcula la penalización teniendo en cuenta el peso que el usuario le pasa, que se lo suma al error que se comete en cada restricción al cuadrado. En nuestro caso, las restricciones imponen como valores mínimos cero, por lo que la resta al cuadrado se hace del valor que toma la restricción menos cero. De esta manera, podemos graduar el error que se comete, premiando antes errores que son leves frente a errores que suponen un impacto mayor en el problema. Finalmente, como lo que pretendemos es minimizar el resultado de las funciones, sumamos esta penalización a la solución de evaluar el individuo.