ALGORITMOS DE COMPUTACIÓN EVOLUTIVA MULTIOBJETIVO

Índice

Introducción

El algoritmo ha sido implementado en Python con motivo de la sencillez del lenguaje y por permitir manejar complejas estructuras de datos con gran facilidad. Además tengo mayor soltura con este lenguaje que con otros. Como consecuencia de esto para poder ejecutar el algoritmo es necesario tener instalado Python en su equipo.

En contraposición a lo anterior he tenido problemas al desarrollar funciones recursivas que he tenido que adaptar y convertir a iterativas ya que Python no soporta bien la recursividad.

El código se divide en dos ficheros: “ZDT3_main.py” y “CF6_main.py”. Estos tienen una estructura prácticamente idéntica cambiando únicamente la función de evaluación “gte” y la función de la que se extraen los objetivos del algoritmo: ZDT3 y CF6.

Al ejecutar el código se solicitan ciertos parámetros para controlar: la proporción de mutaciones, crossover, tamaño de la población, número de generaciones y proporción de vecinos.

Los resultados se muestran en una gráfica de la librería mathplotlib de Python la cual es necesario tener instalada para poder visualizar los resultados.

Evolución

El proceso de evolución se realiza sobre una población. Sobre la población se realiza un proceso de mutación donde todos los elementos pasan por la función de mutación. Posteriormente se desordenan aleatoriamente y por parejas se pasan por una función de crossover. Para terminar se repite el proceso de someter la población a las mutaciones. Tras esto se obtiene una población nueva.

Mutación

La mutación se realiza con un sencillo proceso en el que recorremos el individuo que se está mutando y por cada elemento del individuo se genera un número aleatorio entre 0 y 1. Si el número es menor que la probabilidad de mutación mutamos el individuo generando un valor aleatorio en esa posición. En caso contrario el elemento se mantiene en esa posición.

La recomendación sería una probabilidad de mutación 0,3 para que estadísticamente se produzca una mutación en cada gen.

Crossover

El crossover utilizado sigue la misma idea que la mutación respecto a recorrer cada elemento de cada pareja de individuos a la vez. Cuando al generar el número aleatorio, este sea menor que la probabilidad de realizar el crossover empieza el corte del crossover. A partir de este punto los elementos de ambos individuos se intercambiaran. Continuando con la generación de números aleatorios si el número generado vuelve a ser menor que la probabilidad de crossover se para el crossover dejando intacto a los individuos a partir de ese punto (corte final). Este proceso puede repetirse tantas veces como la longitud del individuo lo permita, es decir, en el crossover pueden producirse intercambios de información en distintos puntos con distintas longitudes y múltiples veces. Esto variará según el azar y la probabilidad de crossover.

La recomendación sería una probabilidad superior a 0.3 para producir al menos un corte.

Solución para la implementación general

A continuación se explicará el flujo del algoritmo y su funcionamiento.

En primer lugar se pide la entrada de parámetros para el algoritmo en el siguiente orden: probabilidad de mutación (valor entre 0.0 y 1.0), probabilidad de crossover (valor entre 0.0 y 1.0), número de subproblemas (número entero), número de generaciones (número entero) y proporción de vecinos (valor entre 0.0 y 1.0). En el caso de CF6 además de estos valores es necesario indicar la dimensión (4 o 16).

En segundo lugar inicializamos los pesos de forma que estén distribuidos de manera uniforme. Esto se consigue haciendo el uso de la función ‘arange’ de ‘numpy’ para iterar desde 0.0 a 1.0. El resultado es una lista de vectores de 2 elementos (x,y) en los cuales el primer valor (x) va decreciendo y el segundo valor (y) va creciendo de manera uniforme.

Tras la inicialización de los pesos inicializamos la población. Para esto haciendo uso de la función ‘random()’ de la librería ‘random’ y redondeamos al número de decimales predefinidos en el código, en nuestro caso 6. La población contará con N individuos (tantos como subproblemas) de longitud p. La p representa la dimensión del problema. En el caso de ZDT3 la dimensión será 30 y en CF6 será 4 o 16.

Continuando tras la inicialización de la población creamos subproblemas asignándole a cada subproblema el peso, una lista con los pesos de los vecinos más cercanos, el mejor individuo posible y su valor fitness (de la función gte).

Una vez preparados los subproblemas empezamos el bucle principal. Realizaremos el número de generaciones predefinidas en la entrada al programa -1 iteración ya que ya realizamos una evaluación previa en la inicialización y tenemos límite en el número de evaluaciones realizadas.

El bucle itera sobre todos los subproblemas (N) y para cada subproblema generaremos un número (G) de hijos usando la función de evolución descrita anteriormente y aprovechará el nuevo individuo para comprobar si es mejor que la mejor solución actual de todos los vecinos del subproblema (incluyéndose a sí mismo).

La evaluación se realiza dependiendo de la función y de si tiene o no restricciones.

Para terminar se realiza una gráfica con la población usando la librería 'mathplotlib'. Para comparar con mayor comodidad el resultado se aprovechan los archivos PF para pintar el frente ideal y posteriormente se pintan los objetivos conseguidos por cada subproblema.

Función ZDT3

En el caso de la función ZDT3 la evaluación es la más simple y usando la adaptación que he hecho de la función que se nos proporciona para las prácticas.

Tras la evaluación que se realiza con uno de los hijos generados anteriormente durante la evolución (con el individuo del subproblema y 3 vecinos elegidos aleatoriamente) se comprueba para todos los subproblemas de la vecindad si el nuevo individuo es mejor que alguno de cualquiera de los subproblemas. Esto se comprueba usando la función gte y quedándonos con el individuo con el mínimo fitness.

Función CF6

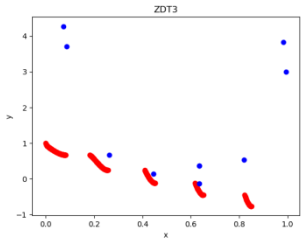
En el caso de la función CF6 la evaluación es similar pero además de los dos objetivos devuelve dos restricciones. Siguiendo exactamente el mismo esquema pero la función gte(…) del algoritmo de la función CF6 recibe además las dos restricciones, en caso de incumplir las restricciones se les suma a ambos objetivos disminuyendo la calidad de la solución.

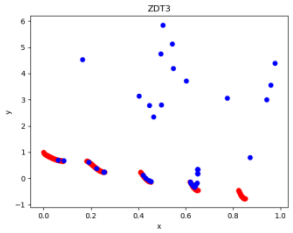
Pruebas

He ido mejorando el algoritmo y cambiando distintos elementos hasta que funcionara como lo hace actualmente.

Las anotaciones principales que he podido hacer sobre el algoritmo han sido las siguientes:

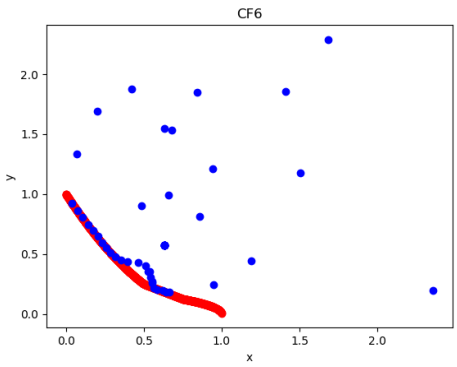
- Trabajar con un mayor número de subproblemas (N mayor) supone un tiempo de computo mayor. Sin embargo los resultados son algo menos exactos.
  En este ejemplo se muestra la diferencia en ZDT3 con mutación 0.1, crossover 0.2, 50N, 80G y 0.3 en el primer ejemplo. En el segundo ejemplo igual pero cambiando a 10N y 400G.

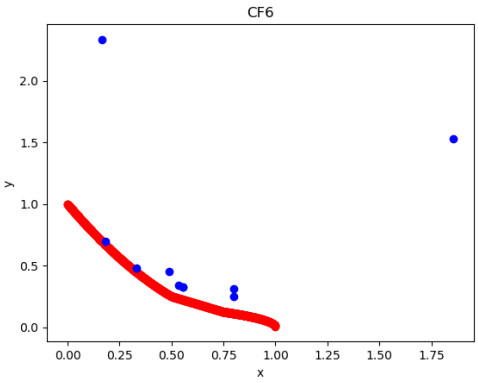




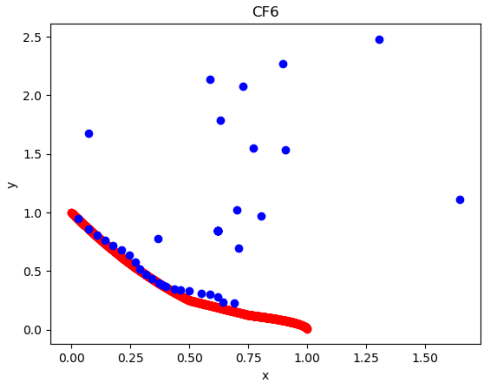
- Estos son dos de las múltiples pruebas que he hecho. La mayoría con resultados similares.

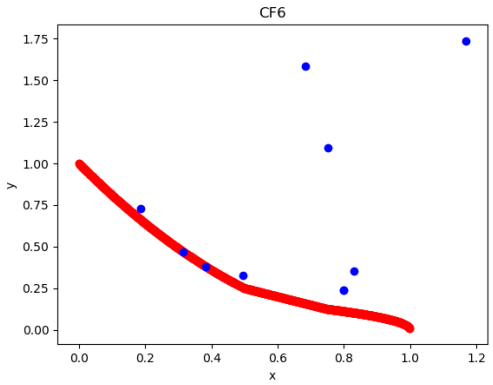
  Con CF6 pasa exactamente lo mismo. En este ejemplo usamos los mismos datos pero con dimensión = 4:





- En este ejemplo usamos los mismos datos pero con dimensión = 16:

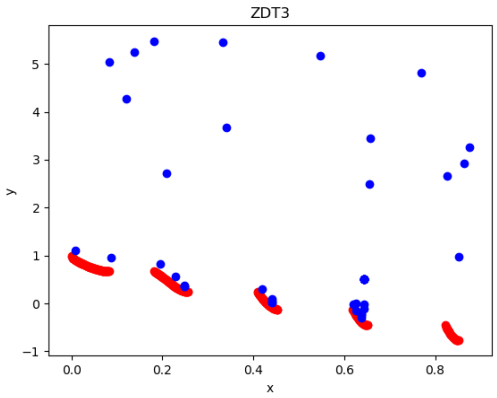




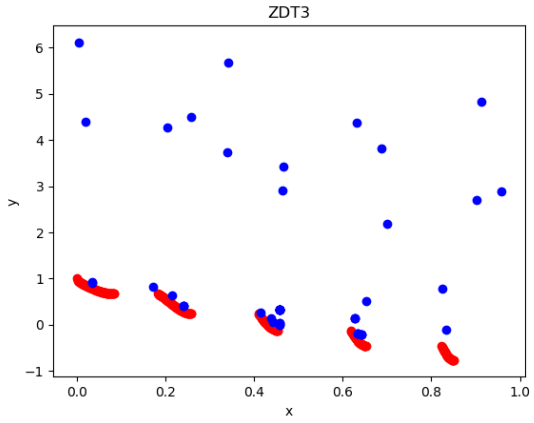
Pruebas con la probabilidad de mutación

Las siguientes pruebas las he hecho con valores de mutación ascendentes desde 0.1 a 0.9 (10% a 90%). El resto de parámetros son 0.1 para crossover, 50N, 80G y 0.3 vecinos con ZDT3:

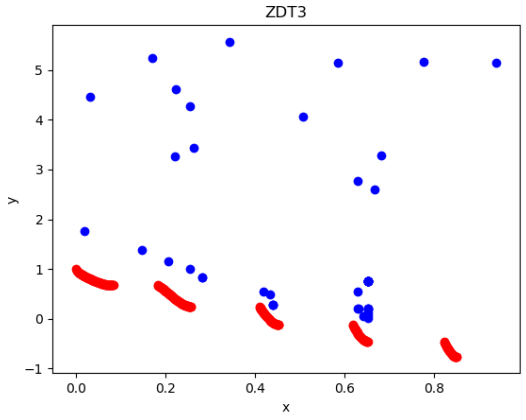
- 10%:



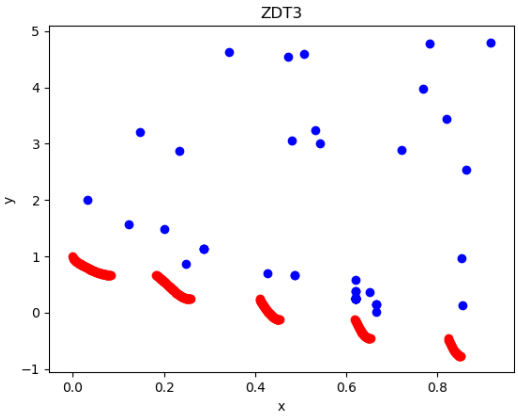
- 30%:



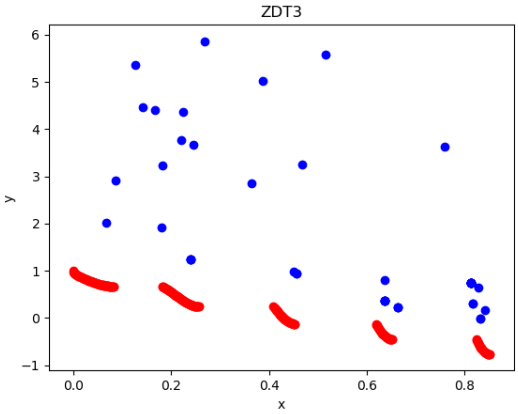
- 50%:



- 70%:



- 90%:

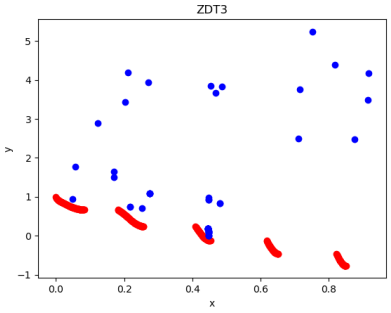


Como podemos observar la mutación es importante en los algoritmos evolutivos porque añade diversidad pero cuando supera cierta probabilidad se vuelve un problema. Esto debe ocurrir ya que los individuos mutan tanto que por muy bueno que sea un individuo su hijo tras la mutación conservará muy pocos valores del padre.

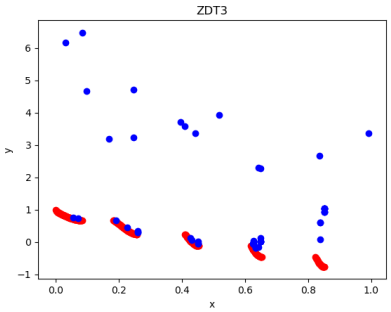
Pruebas con la probabilidad de crossover

Continuando con el siguiente parámetro he ido comprobando los efectos de ir modificando la probabilidad de que se produzca el crossover desde 0.1 a 0.9 (desde 10% a 90%) y la probabilidad de mutación siendo 0.3, 50N, 80G y 0.3 porcentaje del tamaño de la vecindad con ZDT3:

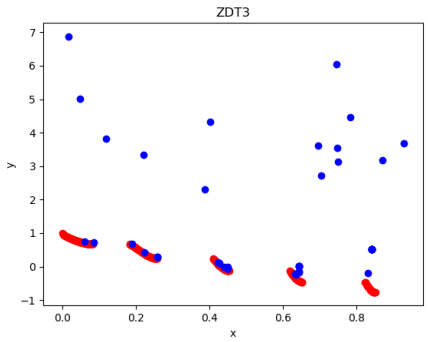
- 10%:



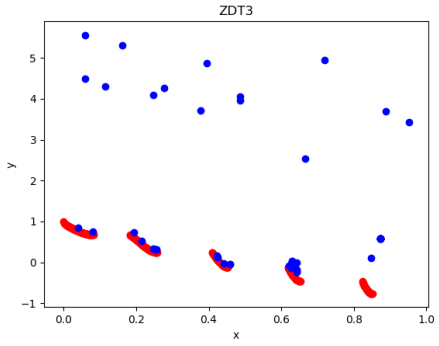
- 30%:



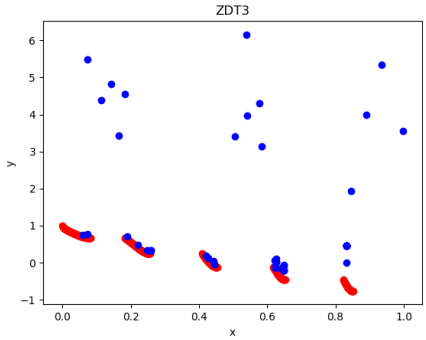
- 50%:



- 70%:



- 90%:



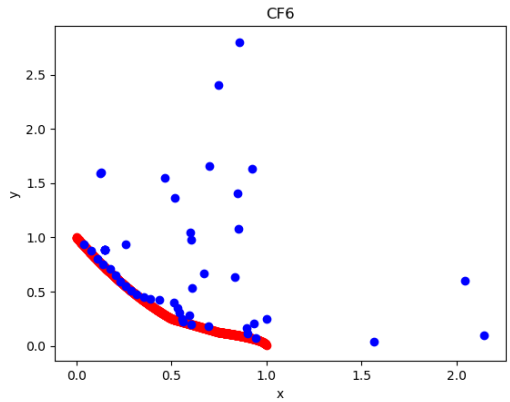
En este caso el crossover parece favorecer en casi todos los casos el que se encuentre la solución óptima. A partir del 10% las soluciones mejoran ligeramente. Esto debe ocurrir ya que con un 30% nos aseguramos de que se produzca (estadísticamente) un crossover mínimo por cada dos individuos.

No afecta que la probabilidad de crossover sea alta porque en caso de que ocurra esto lo que pasará es que se realizarán múltiples crossover pero con tramos pequeños de los individuos.

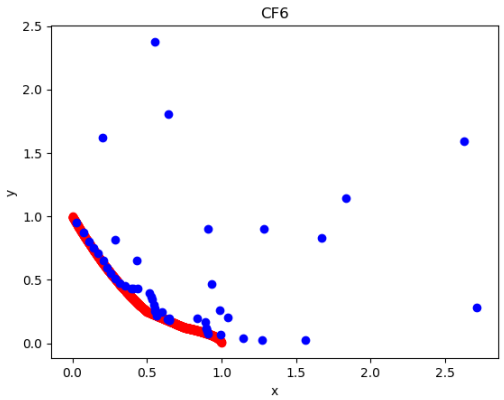
Pruebas con la cantidad de vecinos por cada subproblema

Ahora toca el valor del número de vecinos. Como los anteriores iremos variando los valores desde 0.1 hasta 0.9 (tamaño de la vecindad desde el 10% de la población al 90%) con probabilidad de mutar 0.3, probabilidad de crossover 0.1, 50N y 800G:

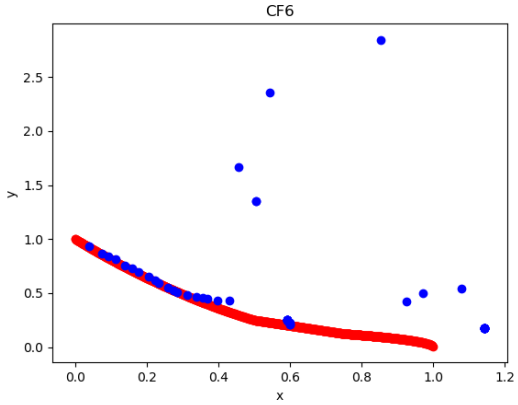
- 10%:



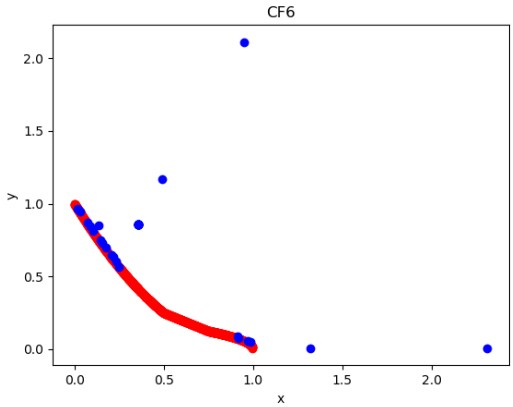
- 30%:


CF6
2.5
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
2.5
x

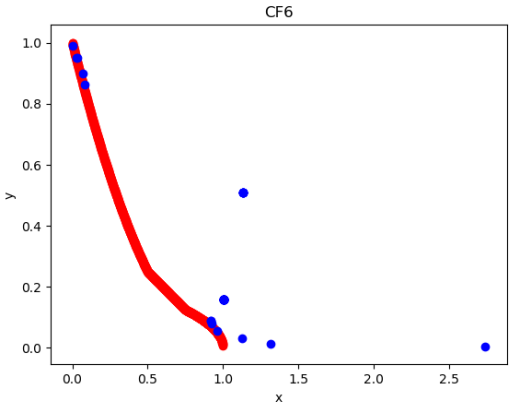
- 50%:


CF6
2.5
2.0
1.5
1.0
0.5
0.0
y
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x

- 70%:


CF6
2.0
1.5
1.0
0.5
0.0
y
0.0
0.5
1.0
1.5
2.0
x

- 90%:



Como podemos observar según vamos aumentando el tamaño de la vecindad se van formando clústeres mayores. Esto hace que se concentren todas las posibles soluciones en puntos masificados. Por ello no deberíamos aumentar el tamaño de la vecindad por encima del 30% de la población según mis resultados.

Otras pruebas

Durante el desarrollo del algoritmo tuve ciertas dificultades y tuve que hacer algunos cambios:

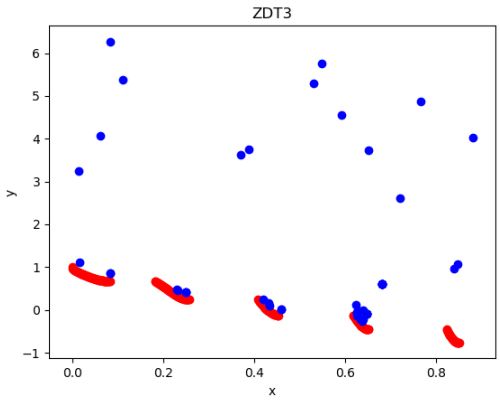
- La idea principal era que iterara sobre la población y fuera generando un nuevo individuo y probándolo para cada subproblema. Por alguna razón el rendimiento era menor que si iteraba sobre los subproblemas y para cada subproblema iba probando toda la población y generando los nuevos individuos. Esto hace que el frente en lugar de avanzar uniformemente vaya avanzando acercándose primero por un lado al frente pareto y luego vaya cerrando el frente.

Al necesitar 4 vecinos para realizar la evolución el algoritmo podía fallar y decidí añadir un valor mínimo (4) para el tamaño de la vecindad.

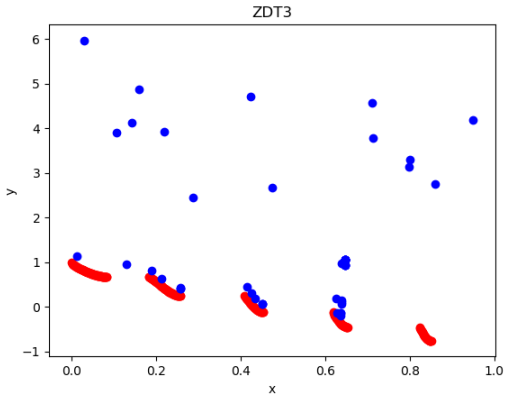
Las pruebas mostradas son aproximadamente un tercio de las hechas en mi ordenador pero he decidido omitirlas en el documento.

Los resultados con probabilidad de mutación 0.3, de crossover 0.1 y 0.3 de tamaño de la vecindad es:

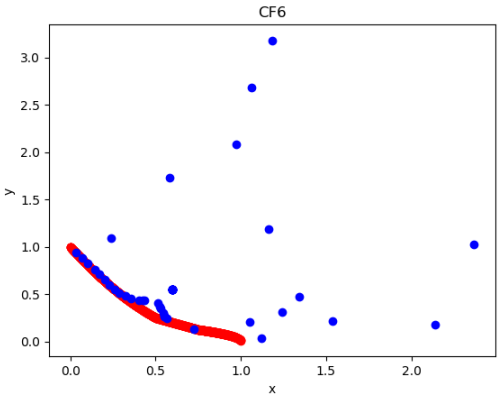
- 4000 evaluaciones ZDT3:



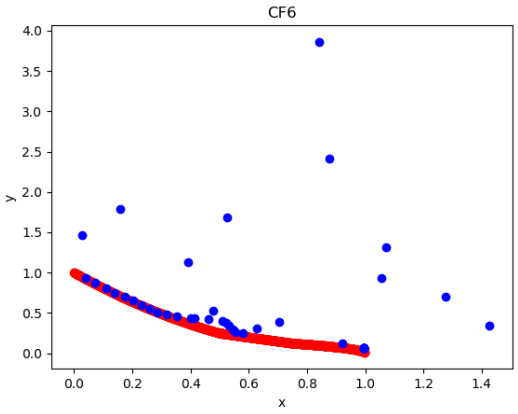
- 10000 evaluaciones ZDT3:



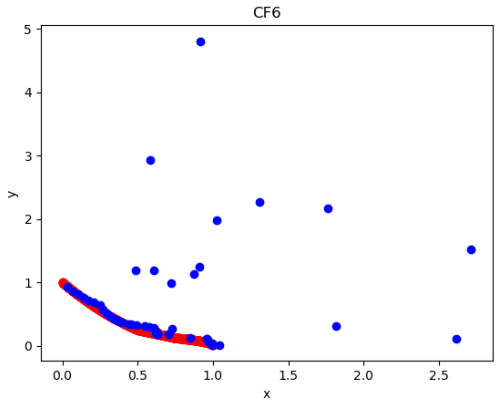
- 4000 evaluaciones CF6 de dimensión 4:



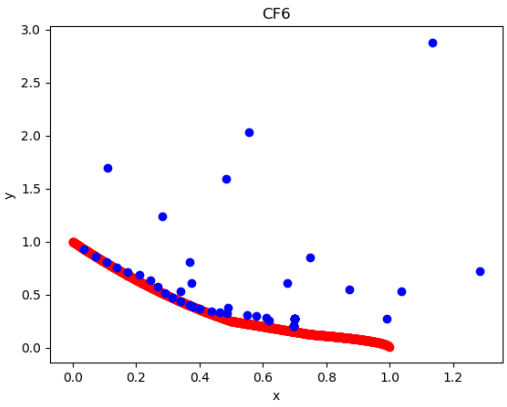
- 10000 evaluaciones CF6 de dimensión 4:



- 4000 evaluaciones CF6 de dimensión 16:



- 10000 evaluaciones CF6 de dimensión 16:



En resumen los resultados no han podido superar los del algoritmo propuesto por el profesor.

Conclusiones y comentarios

No he conseguido superar el algoritmo que se nos ha dado. He intentado implementar el algoritmo pero he tenido dificultades a la hora de entender el documento pero a pesar de ello con la ayuda de las tutorías del profesor he podido resolver las dudas y acabar el algoritmo.

Considero que podría mejorar el algoritmo pero tras muchos intentos no sé exactamente que falla. He realizado múltiples pruebas para mejorar el funcionamiento y además de ello tuve que tomar una decisión durante el desarrollo del proyecto.

Aproximadamente he realizado 3 pruebas por cada captura que se muestra en este documento tomando la más representativa.