

24 de OCTUBRE de 2022

iván moreno granado

universidad de sevilla

IVAMORGRA@ALUM.US.ES

competición. algoritmo evolutivo multiobjetivo basado en agregación

ejercicio obligatorio aplicaciones de soft computing

Contenido

[***1.- Ejecución de algoritmos con población de 30 individuos y 300 generaciones (iteraciones)*** 4](#_Toc114943354)

[***1.1.- Algoritmo genético*** 4](#_Toc114943355)

[***1.2.- Algoritmo evolución diferencial*** 5](#_Toc114943356)

[***1.3.- Algoritmo PSO*** 5](#_Toc114943357)

[***1.4.- Conclusión*** 6](#_Toc114943358)

[***2.- Ejecución de algoritmos con población de 90 individuos y 300 generaciones (iteraciones)*** 6](#_Toc114943359)

[***2.1.- Algoritmo Genético*** 6](#_Toc114943360)

[***2.2.- Algoritmo de Evolución Diferencial*** 7](#_Toc114943361)

[***2.3.- Algoritmo PSO*** 7](#_Toc114943362)

[***2.4.- Conclusión*** 8](#_Toc114943363)

[***3.- Ejecución de algoritmos con población de 30 individuos y 900 generaciones (iteraciones)*** 8](#_Toc114943364)

[***3.1.- Algoritmo Genético*** 8](#_Toc114943365)

[***3.2.- Algoritmo Evolución Diferencial*** 9](#_Toc114943366)

[***3.3.- Algoritmo PSO*** 10](#_Toc114943367)

[***3.4.- Conclusión*** 10](#_Toc114943368)

[***4.- Ejecución de algoritmos con población de 90 individuos y 900 generaciones (iteraciones)*** 11](#_Toc114943369)

[***4.1.- Algoritmo Genético*** 11](#_Toc114943370)

[***4.2.- Algoritmo Evolución Diferencial*** 11](#_Toc114943371)

[***4.3.- Algoritmo PSO*** 12](#_Toc114943372)

[***4.4.- Conclusión*** 12](#_Toc114943373)

[***5.- Ejecución de algoritmos con población de 30 individuos y 2700 generaciones (iteraciones)*** 13](#_Toc114943374)

[***5.1.- Algoritmo Genético*** 13](#_Toc114943375)

[***5.2.- Algoritmo Evolución Diferencial*** 13](#_Toc114943376)

[***5.3.- Algoritmo PSO*** 14](#_Toc114943377)

[***5.4.- Conclusión*** 14](#_Toc114943378)

[***6.- Ejecución de algoritmos con población de 90 individuos y 2700 generaciones (iteraciones)*** 14](#_Toc114943379)

[***6.1.- Algoritmo Evolución Diferencial*** 15](#_Toc114943380)

[***6.2.- Algoritmo PSO*** 15](#_Toc114943381)

[***6.3.- Conclusión*** 16](#_Toc114943382)

[***7.- Ejecución de algoritmos con población de 90 individuos y 24300 generaciones (iteraciones)*** 16](#_Toc114943383)

[***7.1.- Algoritmo Evolución Diferencial*** 16](#_Toc114943384)

[***7.2.- Algoritmo PSO*** 17](#_Toc114943385)

[***7.3.- Conclusión*** 17](#_Toc114943386)

[***8.- Conclusión final*** 18](#_Toc114943387)

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.

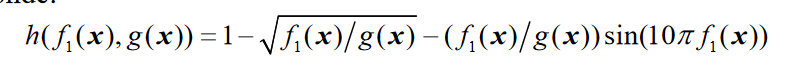
A continuación, se detallará cada problema:

* ZDT3

Cuenta con las siguientes funciones:

Texto, Carta

Descripción generada automáticamente



Un reloj con números romanos

Descripción generada automáticamente con confianza media

Con un espacio de búsqueda definido:



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

Interfaz de usuario gráfica, Gráfico

Descripción generada automáticamente

* CF6

Cuenta con las siguientes funciones:

Texto, Carta

Descripción generada automáticamente

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

Gráfico, Gráfico de líneas

Descripción generada automáticamente

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.
* 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:

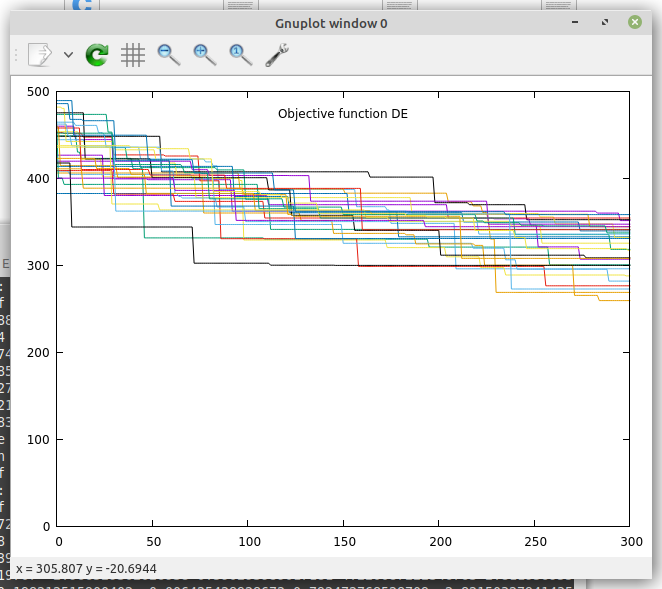
1. La función inicializadora, (se encarga de hacer la primera llamada) es la función **out.** Además de hacer la primera llamada comprueba si el usuario quiere ejecutar el algoritmo una vez de forma aleatoria o diez veces con semillas diferentes. Termina llamando a la función **main**.
2. La función 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.- Algoritmo evolución diferencial

**Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

En este caso se puede ver que el algoritmo de evolución diferencial para esta configuración al menos no presenta una gran resolución, puesto que al llegar a la última generación la función fitness se queda bastante alejada del 0. Además, se puede ver como la función no decrece nada rápido.

### 1.3.- Algoritmo PSO

**Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamenteInterfaz de usuario gráfica

Descripción generada automáticamente**

El algoritmo PSO presenta un buen decremento de la función fitness, llegando incluso a acercarse al 0 el valor de la función fitness en algunas ejecuciones. El principal problema que presenta es la variabilidad final en cada uno de los resultados finales de la función fitness. Presenta un rango amplio de soluciones, algo que hace que dependa mucho lo bueno que es este algoritmo en esta configuración dependiendo de la ejecución. Por tanto, podemos afirmar que la calidad de este algoritmo con esta configuración depende bastante de la ejecución.

### 1.4.- Conclusión

Podemos concluir con dicha configuración que el algoritmo de evolución diferencial **DE** es el peor de los tres con diferencia y el algoritmo **PSO** es capaz de obtener los valores mínimos de la función de fitness (más cercanos a 0). Sin embargo, presenta mayor variabilidad que el algoritmo genético **AG**. Esto puede causar un gran problema, puesto que dicho algoritmo puede tomar valores objetivos muy buenos, no tan buenos o bastante alejados de los óptimos. En este caso, es más interesante tomar un algoritmo que me dé la confianza que en cada una de las ejecuciones me va a presentar valores cercanos dentro de un rango bastante más estrecho y, por tanto, su variabilidad va a ser baja.

## 2.- Ejecución de algoritmos con población de 90 individuos y 300 generaciones (iteraciones)

### 2.1.- Algoritmo Genético

**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Aumentando la población para el algoritmo genético (triple de población respecto la configuración inicial) da lugar a una banda de soluciones más estrecha) para las treinta ejecuciones realizadas. Además, se mejora brevemente el valor de la función fitness. Por tanto, podemos decir que esta configuración para algoritmo genético da mejores resultados en cuanto a calidad se refiere.

### 2.2.- Algoritmo de Evolución Diferencial

**Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Para algoritmo DE vemos todo lo contrario, la función fitness es peor. No decrementa la función fitness apenas y tampoco podemos intuir que lo haga, observando un estancamiento claro del algoritmo con esta configuración. Aunque con la configuración inicial dicho algoritmo no presentaba un buen rendimiento, con esta configuración se muestra que es aún peor.

### 2.3.- Algoritmo PSO

**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

El algoritmo PSO por otro lado, presenta un rendimiento bastante parecido a la configuración anterior. Sin embargo, el rango de soluciones de la función *fitness* se reduce brevemente (como en el caso de algoritmos genéticos).

### 2.4.- Conclusión

Sin lugar a duda, el peor algoritmo para esta configuración sigue siendo el **DE**, ya que su función presenta unos valores mínimos de fitness muy alejados respecto a los dos restantes. Por tanto, dicho algoritmo queda totalmente descartado.

Entre PSO y AG podemos comparar lo rápido que decrece la función de fitness. Para ello, el algoritmo AG es mucho mejor, aunque alcanza valores algo más altos que los valores de PSO.

Por tanto, podemos concluir que el algoritmo AG alcanza con mayor facilidad (menor número de generaciones) menor valor de la función de calidad y además presenta menos variabilidad en dichas soluciones entre cada una de sus ejecuciones. Por otro lado, el algoritmo PSO puede dar el mejor valor solución de los tres algoritmos, puesto que llega a los valores más bajos de la función de calidad. Sin embargo, necesita más ejecuciones y presenta mayor variabilidad, concepto explicado anteriormente en el punto 1.4.

## 3.- Ejecución de algoritmos con población de 30 individuos y 900 generaciones (iteraciones)

### 3.1.- Algoritmo Genético

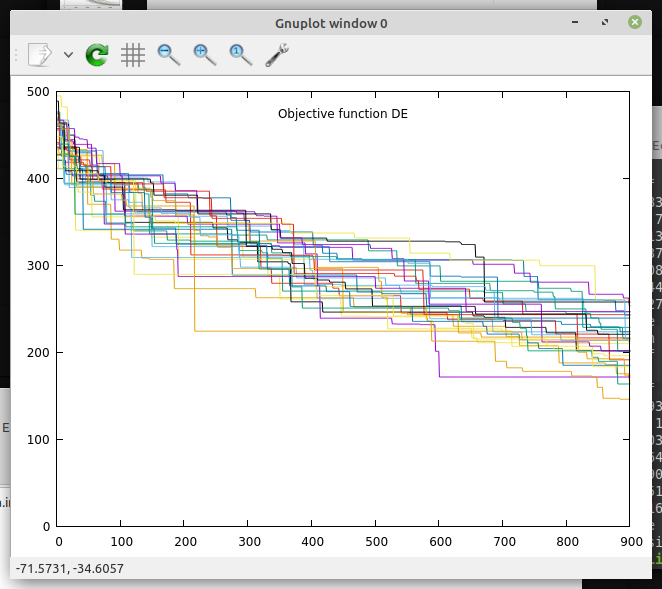
**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Al igual que en las anteriores pruebas con este algoritmo, se puede ver como a partir de ciertas ejecuciones el algoritmo se estanca y no da lugar a un decrecimiento de la función de fitness. Podemos intuir que este algoritmo tiene poco margen de mejora. Sin embargo, seguiremos probándolo en futuras configuraciones.

### 3.2.- Algoritmo Evolución Diferencial

**Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Claramente se puede ver que de todas las pruebas que llevamos con este algoritmo, esta configuración es la mejor. Se ha conseguido valores de calidad más bajos (llegando a ser menores a 200), algo que no había pasado antes. Sin embargo, sigue siendo muy variable y se sigue alejando de valores alcanzables por los dos anteriores algoritmos.

Por otro lado, este algoritmo tiene bastante margen de mejora y podemos intuir que aumentando el número de generaciones aumenta el rendimiento de **DE**.

### 3.3.- Algoritmo PSO

**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Se puede ver como este algoritmo mejora sobre todo en la variabilidad, que baja de manera considerable. Además, con dicha configuración alcanza valores muy bajos con respecto a las configuraciones anteriores.

### 3.4.- Conclusión

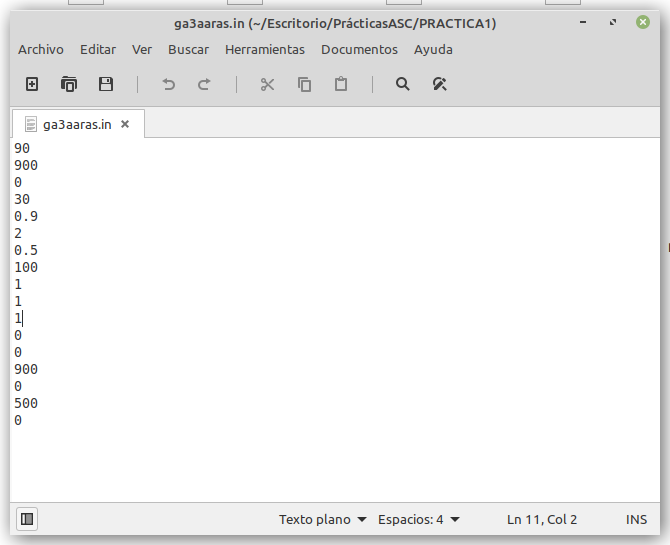
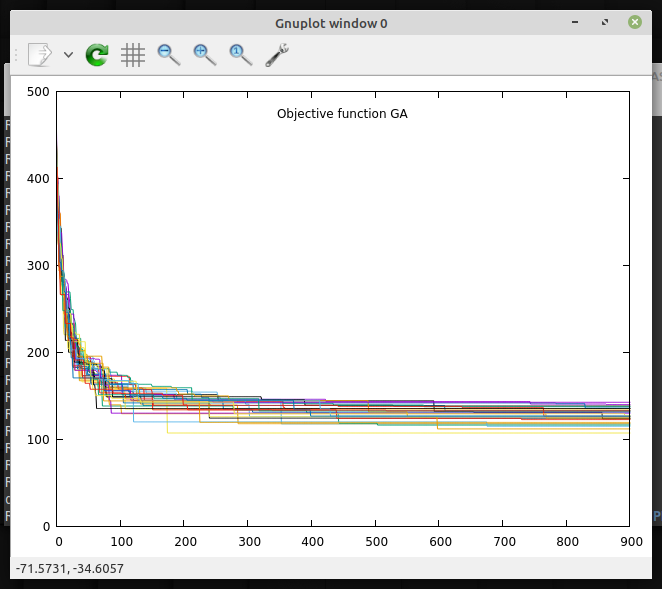
Aunque el algoritmo de evolución diferencial (DE) sigue sin ser competitivo con los dos algoritmos restantes, podemos ver una notable mejoría respecto a las configuraciones vistas anteriormente. Por tanto, se puede intuir que este algoritmo mejora cuando aumentamos el número de iteraciones.

Por otro lado, el algoritmo **PSO** compite ya más con el algoritmo genético y se puede plantear que para esta configuración sea bastante mejor, ya que de los tres es el que alcanza valores más bajos y su variabilidad de soluciones *fitness*, aunque sea mayor que el **AG**, no es alta.

Por otro lado, se intuye que este algoritmo no mejora mucho más, ya que la mayoría de las ejecuciones se estancan en un determinado valor a partir de un número de ejecuciones.

## 4.- Ejecución de algoritmos con población de 90 individuos y 900 generaciones (iteraciones)

### 4.1.- Algoritmo Genético

****

No presenta mejora notable con respecto a las anteriores configuraciones. No hay caracteristicas a destacar en su conjunto de ejecuciones. Se puede ver en cada una de las pruebas realizadas que el algoritmo no tiene apenas margen de mejora.

### 4.2.- Algoritmo Evolución Diferencial

**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

Con esta configuración de prueba podemos afirmar que para este algoritmo lo interesante es aumentar la relación individuos-generaciones, aumentando el segundo parámetro. Aumentando el tamaño de población no va a hacer que el algoritmo se comporte de mejor forma, conseguimos todo lo contrario.

### 4.3.- Algoritmo PSO

**Interfaz de usuario gráfica

Descripción generada automáticamenteInterfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente**

También pasa con este algoritmo, aunque visiblemente no se ve con tanta claridad. Sin embargo, el rendimiento es peor cuando aumentamos el número de individuos.

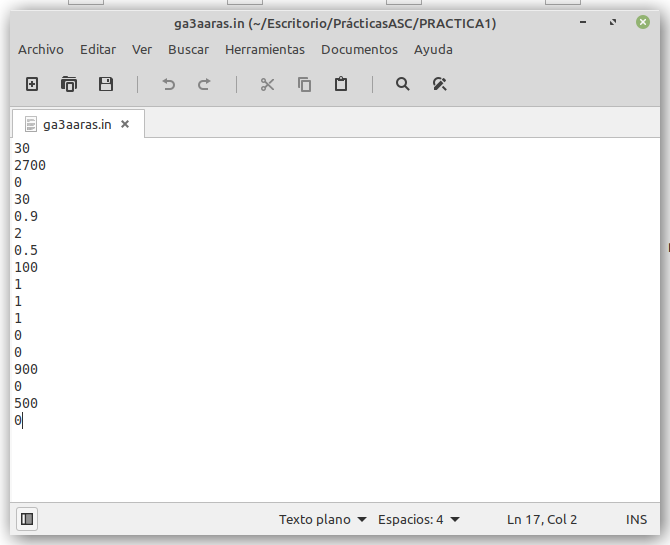
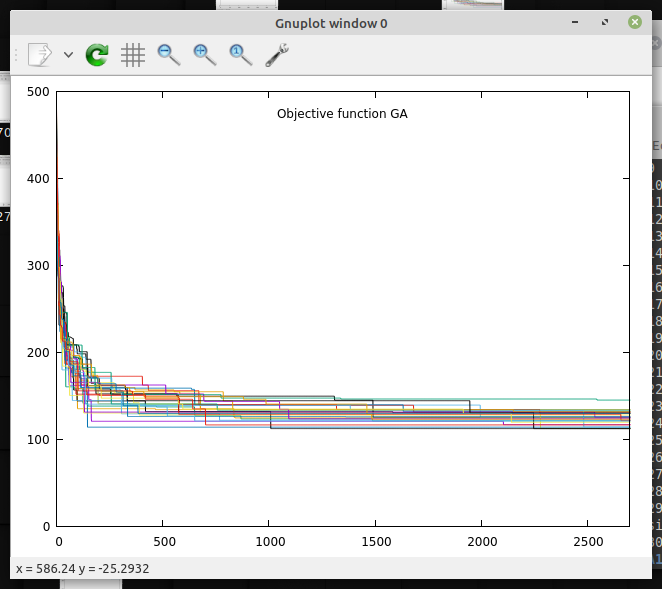
### 4.4.- Conclusión

Los algoritmos **PSO** y **DE** se comportan de peor forma cuando aumentamos el número de individuos. El algoritmo DE con un tamaño de población alto no compite con el resto de los algoritmos. El algoritmo AG con parámetros de configuración más bajos presenta un mayor comportamiento, aunque no llega a ser el óptimo en cuanto a valores mínimos de la función de calidad. Esto quiere decir que en general, el algoritmo **AG** no te da como valor objetivo el valor más cercano al valor real como puede dártelo el algoritmo **PSO**.

Las siguientes pruebas, por tanto, se harán aumentando considerablemente el número de generaciones.

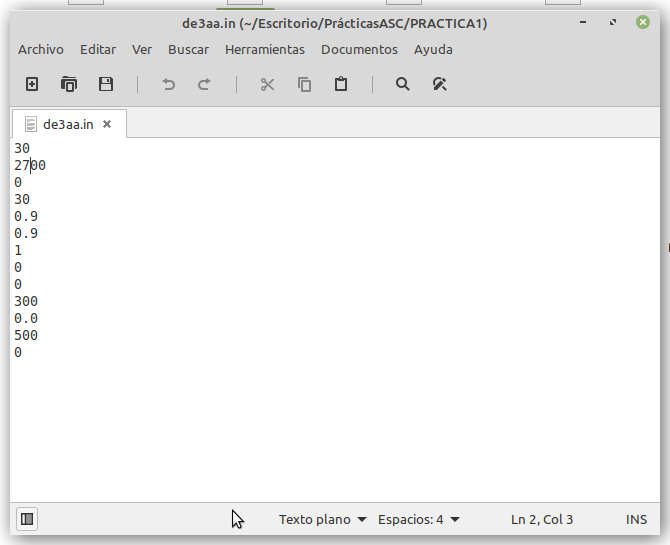
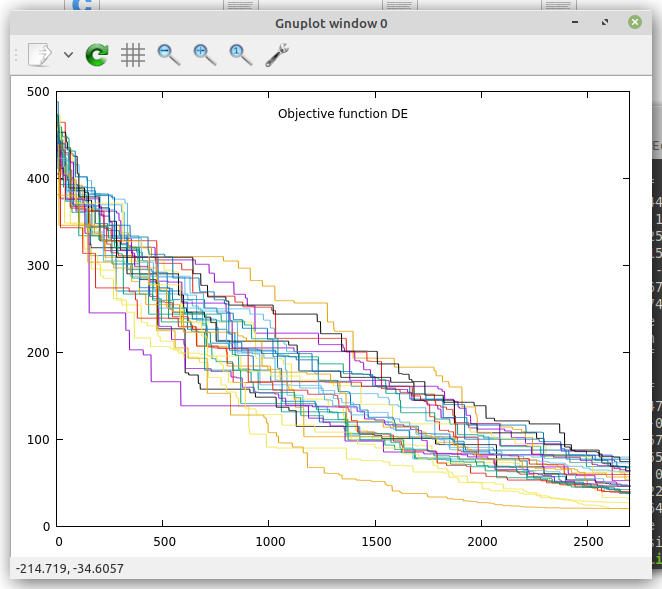
## 5.- Ejecución de algoritmos con población de 30 individuos y 2700 generaciones (iteraciones)

### 5.1.- Algoritmo Genético

****

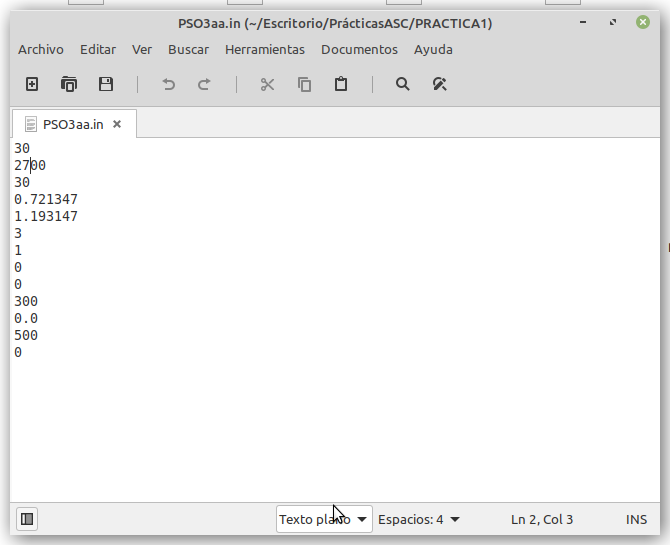
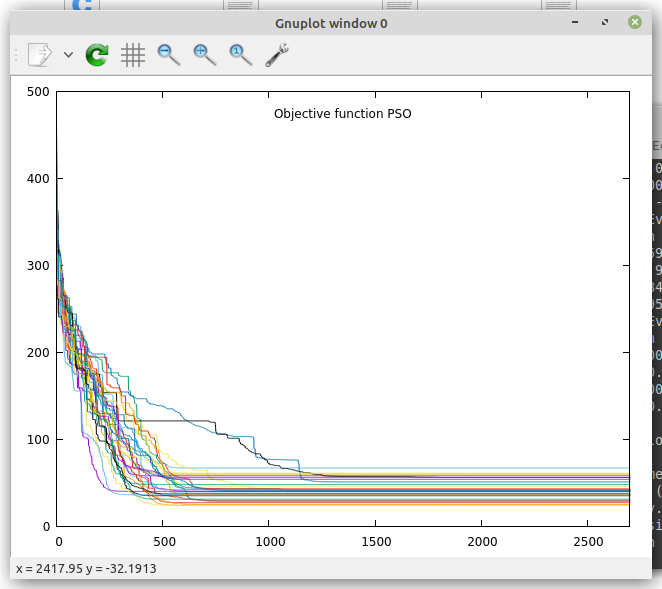
No se destaca ninguna característica especial respecto a las anteriores pruebas.

### 5.2.- Algoritmo Evolución Diferencial

****

Se puede observar un cambio notorio a mejor del rendimiento de este algoritmo. Sin duda, aumentando el número de **generaciones conseguimos unos valores de calidad muy bajos dentro de un rango de valores aún algo grande.**

### 5.3.- Algoritmo PSO

****

Con esta configuración el algoritmo PSO presenta unos valores similares a las configuraciones anteriores. Sin embargo, no parece que este algoritmo mejore mucho más.

### 5.4.- Conclusión

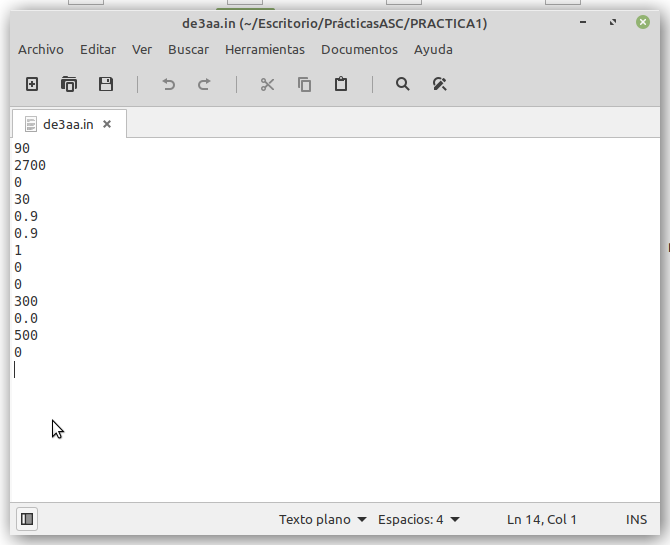
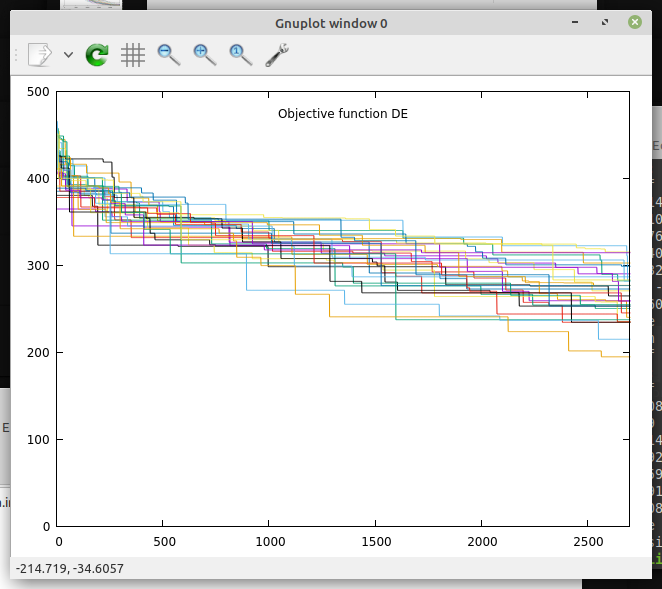
Con un alto número de iteraciones y una relación entre individuos y generaciones el algoritmo DE es el que presenta valores más bajos de la función de calidad. Con estos parámetros este algoritmo compite con el algoritmo PSO.

Por tanto, para esta configuración el algoritmo AG es el que descartamos ya que es el que peor rendimiento tiene.

## 6.- Ejecución de algoritmos con población de 90 individuos y 2700 generaciones (iteraciones)

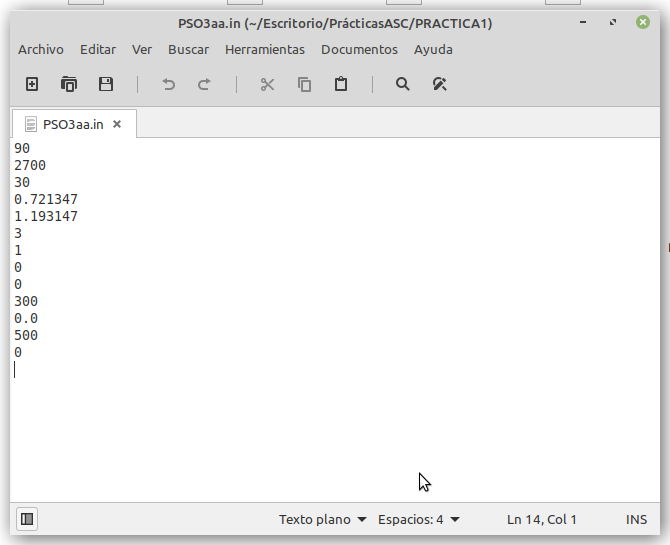
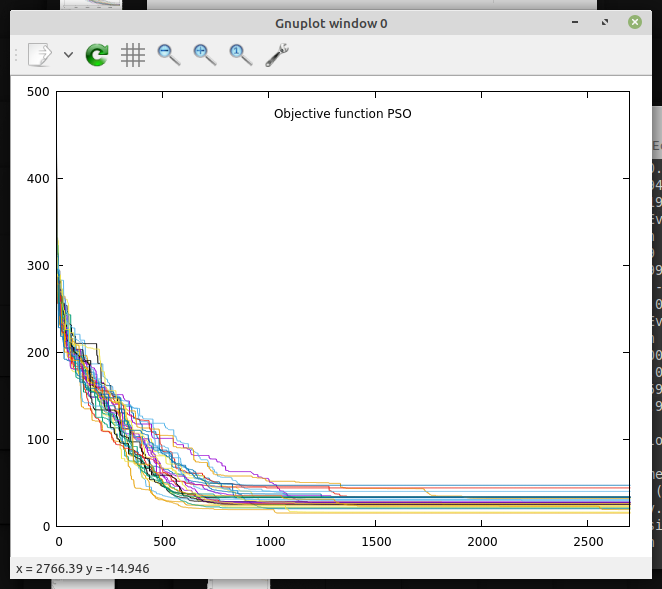
A partir de ahora no se ha tenido en cuenta el algoritmo genético **AG** y se ha sometido a las pruebas el algoritmo **PSO** y **DE**, ya que son los dos únicos que presentan algun margen de mejora (sobre todo el algoritmo **DE**).

### 6.1.- Algoritmo Evolución Diferencial

****

Como era de esperar si reducimos la relación entre individuos y número de generaciones el algoritmo DE presenta un comportamiento peor. Cabe destacar que a pesar de que el aumento de individuos no es tan alto como el aumento en el número de generaciones el comportamiento sigue siendo bastante alejado de lo que se quiere obtener.

### 6.2.- Algoritmo PSO

****

No presenta nada especial respecto a las anteriores pruebas.

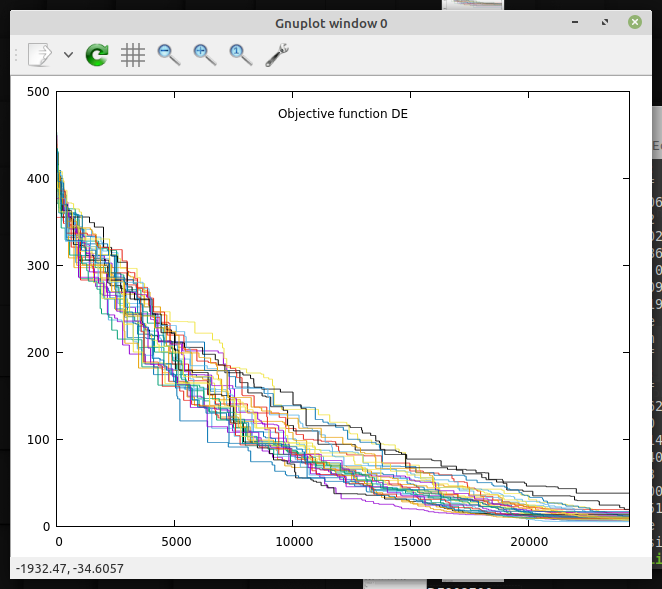
### 6.3.- Conclusión

Podemos concluir que sin ninguna duda el algoritmo **DE** no mejora si no aumentamos el número de generaciones y hacemos lo mayor posible la relación entre número de individuos y número de generaciones. Por tanto, vamos a someter en la siguiente prueba a los algoritmos DE y PSO a un valor muy alto (múltiplo de 3) de generaciones.

## 7.- Ejecución de algoritmos con población de 90 individuos y 24300 generaciones (iteraciones)

Para esta prueba se ha obviado la captura del archivo de configuración de cada uno de los algoritmos.

### 7.1.- Algoritmo Evolución Diferencial

****

Como se esperaba, el algoritmo presenta el mejor comportamiento de todas las pruebas realizadas anteriormente. Presenta valores muy bajos de la función de calidad y una variabilidad bastante baja.

### 7.2.- Algoritmo PSO

**Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente**

Como también era de esperar, el algoritmo **PSO** no presenta ninguna característica especial en esta prueba respecto a las anteriores.

### 7.3.- Conclusión

Para esta configuración el mejor algoritmo es evolución diferencial (**DE**), ya que es el que presenta los valores más cercanos al 0 en la función *fitness* que es lo que buscamos. De hecho, es el algoritmo que mejores resultados presenta en todas las pruebas realizadas. Sin embargo, para que este algoritmo empiece a ser considerablemente bueno necesita un tamaño (NxG) bastante alto, teniendo en cuenta que debe ser G (número de generaciones) el parámetro mayor y cuya relación debe ser lo suficientemente grande.

## 8.- Conclusión final

Tras realizar las diferentes pruebas sobre los tres algoritmos estudiados en la asignatura podemos sacar las siguientes conclusiones:

* Ningún algoritmo es mejor que otro en su totalidad, ya que dependemos del problema, que a su vez tiene un contexto con unos parámetros concretos. La elección de algoritmo mejor de estos tres depende esencialmente de estos parámetros que se le pasen como entrada.
* Para un tamaño de parametrización bajo el algoritmo **DE** se descarta y el algoritmo **AG** y **PSO** son los que mejor se comportan.
* Si el tamaño de parametrización es muy bajo, la variabilidad del algoritmo **AG** es menor que la de **PSO** y, por tanto, puede ser el mejor algoritmo de los tres.
* El algoritmo **AG** necesita menos iteraciones para llegar al valor mínimo que puede ofrecer como valor de calidad (*fitness*).
* Para una parametrización más alta, con más iteraciones, el algoritmo PSO es el mejor de los tres.
* El algoritmo que mejor rendimiento puede ofrecer es el algoritmo DE, mientras que los restantes tienden a estancarse antes, pero para ello es necesario un número de iteraciones muy alto