Práctica 2

x

UNIVERSIDAD DE JAÉN

Metaheurísticas

GRUPO 2

Jesús Morales Villegas --- @Jmv00037 --- 77770715X
Jesús Manzano Álvarez --- @Jma00068 --- 20887601J

1. Objetivo de la práctica

El objetivo de esta práctica es estudiar el funcionamiento de Algoritmos de los algoritmos evolutivos y Metaheurísticas basadas en poblaciones.

Para ello tendremos que implementar varios algoritmos como: Algoritmo Evolutivo y Algoritmo de Evolución Diferencial.

2. Descripción del problema

Disponemos de 10 funciones matemáticas, las cuales, son habitualmente utilizadas para la evaluación de algoritmos metaheurísticos.

Estas se caracterizan por recibir un vector, con una cierta longitud (dimensión). Este vector es construido con valores entre un rango determinado, el cual, viene definido en la descripción de dicha función. Finalmente, nos devolverá un valor dentro del dominio definido.

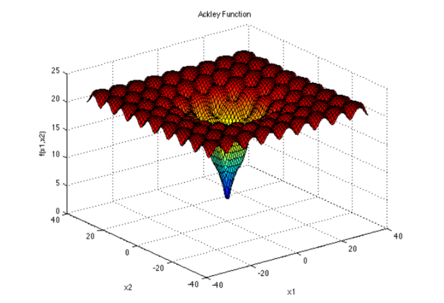
Todas estás funciones tienen un óptimo global y rango determinado, por tanto, vamos a exponer concretamente las principales características de cada una.

2.1 Función de Ackley

La función suele evaluarse sobre el hipercubo $x_i \in [-32.768, 32.768]$, para todo i = 1, …, d, aunque también puede estar restringida a un dominio más pequeño.

Su mínimo global se da cuando se cumple la siguiente condición:

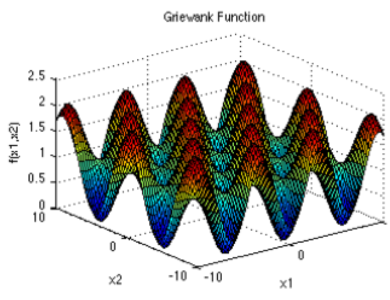
$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \dots, 0)$$



$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

2.2 Función de Griewank

La función suele evaluarse sobre el hipercubo $x_i \in [-600, 600]$, para todo i = 1, …, d.



Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \dots, 0)$$

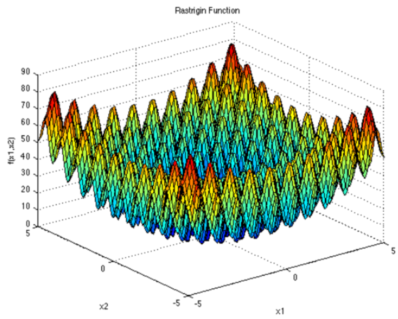
$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

2.3 Función de Rastrigin

La función suele evaluarse sobre el hipercubo $x_i \in$ [-5.12, 5.12], para todo i = 1, …, d.

Su mínimo global se da cuando se cumpe la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \dots, 0)$$



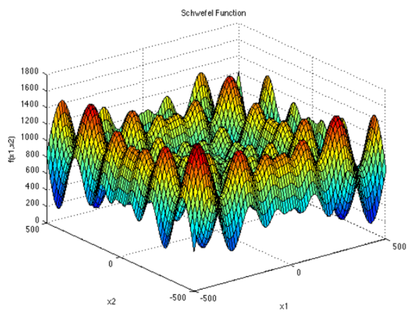
$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

2.4 Función de Schewefel

La función suele evaluarse sobre el hipercubo $x_i \in$ [-500, 500], para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

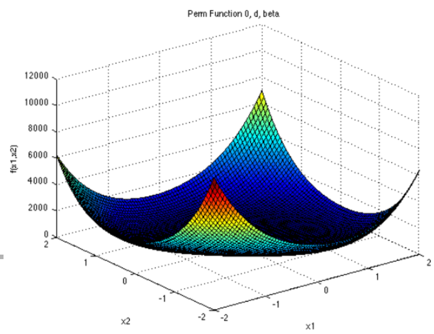
$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (420.9687, \dots, 420.9687)$$



$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

2.5 Función permanente 0, D, Beta

La función suele evaluarse sobre el hipercubo $x_i \in$ [-d, d], para todo i = 1, …, d.



$$f(\mathbf{x}) = \sum_{i=1}^{d} \left(\sum_{j=1}^{d} (j + \beta) \left(x_j^i - \frac{1}{j^i} \right) \right)^2$$

Su mínimo global se da cuando se cumpe la siguiente condición:

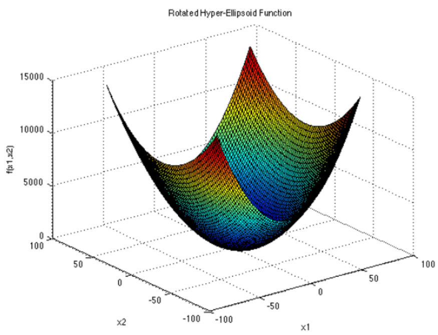
$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = \left(1, \tfrac{1}{2}, \ldots, \tfrac{1}{d}\right)$$

2.6 Función Rotated Hyper-Ellipsoid

La función suele evaluarse sobre el hipercubo $x_i \in [-65.536, 65.536]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$



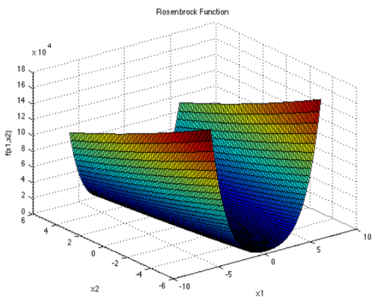
$$f(\mathbf{x}) = \sum_{i=1}^{d} \sum_{j=1}^{i} x_j^2$$

2.7 Función de Rosenbrock

La función suele evaluarse sobre el hipercubo $x_i \in [-5, 10]$, para todo i = 1, …, d, aunque puede restringirse al hipercubo $x_i \in [-2.048, 2.048]$, para todo i = 1, …, D.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (1, \ldots, 1)$$

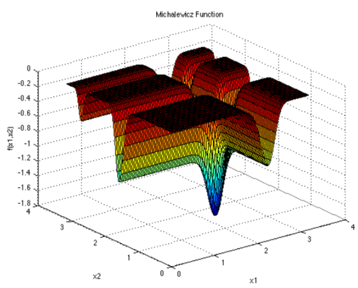


$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

2.8 Función de Michalewicz

La función suele evaluarse sobre el hipercubo $x_i \in [0, \pi]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumplen las siguientes condiciones, dependiendo del valor de la dimensión:



$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i)\sin^{2m}\left(\frac{i x_i^2}{\pi}\right)$$

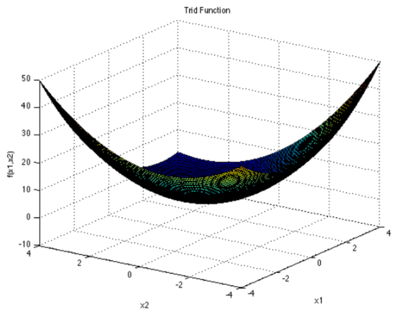
at $d = 2$: $f(\mathbf{x}^*) = -1.8013$, at $\mathbf{x}^* = (2.20, 1.57)$
at $d = 5$: $f(\mathbf{x}^*) = -4.687658$
at $d = 10$: $f(\mathbf{x}^*) = -9.66015$

2.9 Función de Trid

La función suele evaluarse sobre el hipercubo $x_i \in [-d^2, d^2]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = -d(d+4)(d-1)/6, \text{ at } x_i = i(d+1-i), \text{ for all } i = 1, 2, \ldots, d$$



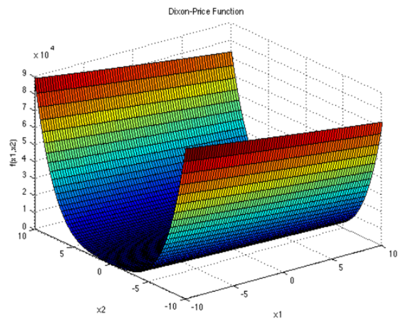
$$f(\mathbf{x}) = \sum_{i=1}^{d}(x_i - 1)^2 - \sum_{i=2}^{d} x_i x_{i-1}$$

2.10 Función de Dixon-price

La función suele evaluarse sobre el hipercubo $x_i \in [-10, 10]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } x_i = 2^{-\frac{2^i-2}{2^i}}, \text{ for } i = 1, \ldots, d$$

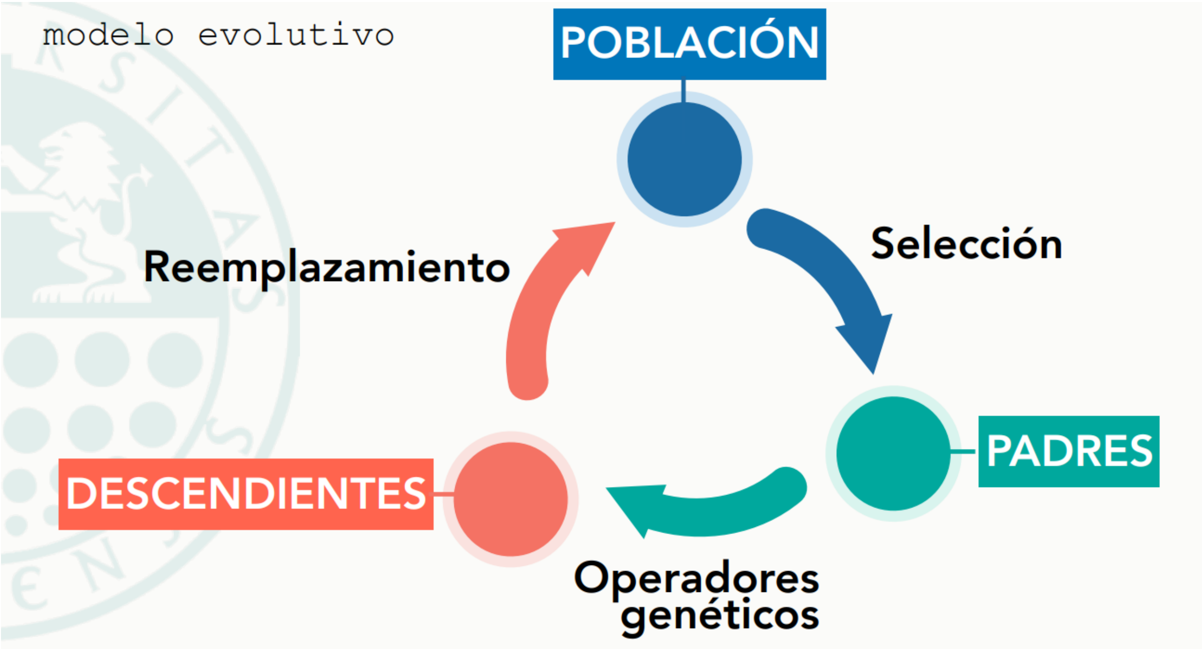


$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{i-1}\right)^2$$

3. Algoritmos basados en poblaciones

Los algoritmos basados en poblaciones son la base de muchos de los métodos utilizados actualmente en problemas de optimización.

Estos algoritmos se pueden ver como un proceso iterativo que parte de una población inicial (aleatoria o en base a un criterio) y la mejora con el tiempo, realizando modificaciones en los individuos a lo largo de las distintas generaciones.



Este tipo de algoritmos empiezan con una población inicial (cada individuo tiene su fitness, es decir, un valor que indica cuánto de “bueno” o “malo” es ese individuo), la cuál se puede generar de forma aleatoria o en base a algún criterio.

Una vez generada, aplicamos el operador de selección (existen diversos métodos de llevarlo a cabo) y obtenemos la población de padres.

Una vez tenemos la población de padres, se aplicarán los operadores genéticos, los cuales se encargarán de modificar estos individuos para generar una población descendiente.

Finalmente, se aplicará el operador de reemplazamiento el cuál se encargará de sustituir los individuos padres por sus hijos. Esto se puede aplicar de diversas maneras como sustituir toda la población padre por la hija.

Esquema general de los algoritmos basados en poblaciones:

```
INICIO
    t = 0
    GENERA(P(t))
    EVALUAR(P(t))
    REPETIR
        seleccionar P(t+1) de P(t)
        recombinar P(t+1)
        mutar P(t+1)
        EVALUAR P(t+1)
        reemplazar P(t) a partir de P(t+1)
        t++
    HASTA (CRITERIO DE PARADA);
FIN
```

3.1 Esquema general del algoritmo evolutivo

Hasta ahora hemos visto el esquema general de los algoritmos basados en poblaciones. Para el caso del algoritmo evolutivo vamos a encontrar varias formas de llevarlo a cabo (generacional y estacionario), pero en nuestro caso vamos a explicar la forma generacional, que es la que hemos implementado, además de aplicar ciertas restricciones que nos indicaba el enunciado del problema.

En primer lugar, vamos a exponer como es el pseudocódigo de este algoritmo, para posteriormente explicar con más detalle cada una de las variables que aparecen.

```
INICIO
    t = 0
    GENERA(P(t))
    EVALUAR(P(T))
    MIENTRAS (no se cumpla condición de parada)
        OBTENEMOS_ELITES(P(t))
        DESDE 1 HASTA individuos a generar
            DESDE 1 HASTA numero de padres
                padres <- TORNEO(k individuos)
            FIN DESDE

            hijo <- CRUCE(padres)
            MUTAR(hijo)
            P(t+1) <- hijo
        FIN DESDE
        PARA cada elite en P(t)
            IF (elite no está en P(t+1)
                ENTONCES P(t+1) <- elite (reemplazo por un aleatorio de los kpeores)
        FIN PARA
        P(t) <- P(t+1) (reemplazo por completo)
        t++
    FIN MIENTRAS
FIN
```

- **elites:** Lista que almacena los mejores individuos de cada generación. Esta implementa el concepto de elitismo (los mejores individuos sobreviven) y puede tener un o varios individuos. Por defecto tendrá tamaño 1.

- **padres:** Lista que almacena todos los individuos que se van generando, es decir, es la futura generación.

- **hijo:** Es el individuo que se crea tras hacer el cruce de los padres y después se le aplica el operador de mutación.

- **k individuos:** Indica el número de individuos con los que se va a realizar el torneo (se escogen k individuos aleatorios y te quedas con el mejor de ellos). Por defecto tendrá tamaño 2.

- **kpeores:** Indica el número de individuos peores que se escogen cuando se va a añadir un elite. Por defecto tendrá el tamaño 4.

- **t:** Indica la generación en la que nos encontramos.

En nuestro caso con respecto a este tipo de algoritmo vamos a implementar dos modificaciones muy parecidas.

Todo va a depender del tipo de operador de cruce que se implemente. En el primer caso será el operador de cruce de media aritmética y el operador de cruce BLX-Alfa.

3.2 Esquema general del algoritmo de evolución diferencial

Este tipo de algoritmos son similares en cuanto a la estrategia que utilizan. Al igual que anteriormente, se genera una población inicial de forma aleatoria o en base a un criterio.

En este caso para generar cada individuo se realizará una fusión entre los tres operadores de selección, cruce y mutación. Este se llama operador ternario, cuyo funcionamiento se basa en hacer una recombinación del padre con dos individuos escogidos aleatoriamente (este proceso se repite hasta generar todos los individuos de la siguiente generación).

$$recombinacion(P, a_1, a_2) = P + F(a_1 - a_2)$$

Esta es la función de recombinación, donde P es el padre, a1 y a2 son los dos individuos escogidos aleatoriamente y F es el factor de mutación (valor aleatorio entre 0 y 1).

```
INICIO
    t = 0
    GENERA(P(t))
    EVALUAR(P(T))
    MIENTRAS (no se cumpla condición de parada)
        DESDE 1 HASTA individuos a generar
            a1 <- aleatorio(P(t))
            a2 <- aleatorio(P(t)) // a1 != a2
            hijo <- OPERADOR_RECOM_TERNARIO(padre, a1, a2)
            IF hijo es mejor que padre
                ENTONCES P(t+1) <- hijo
            ELSE P(t+1) <- padre
        FIN DESDE
        P(t) <- P(t+1) (Pasamos a la siguiente generación)
        t++
    FIN MIENTRAS
FIN
```

- **padre:** Es el individuo por el que vamos recorriendo de forma secuencial para ir generando los hijos. Este individuo es de la población P(t)

En nuestro caso lo único que se va a modificar es el factor de mutación para así poder controlar si queremos una mayor exploración o explotación del espacio. En nuestro caso se ha establecido a un valor de 0.5

4. Operadores utilizados

Vamos a describir, los operadores que tienen cierta relevancia a la hora de ejecutar nuestros algoritmos. Para ello vamos a dividirlos en dos apartados.

4.1 Parámetros de configuración

En el archivo de configuración encontramos diferentes parámetros que usaremos en los diferentes algoritmos. Estos se extraerán del archivo “config.txt”, en el que añadiremos las diferentes constantes a usar en las diferentes ejecuciones.

- **ArrayList <String> archivos:** En él almacenamos los nombres de los ficheros donde se guardan los datos a utilizar en el problema.

- **ArrayList <String> algoritmos:** Almacena el nombre de todos los algoritmos que queremos ejecutar.

- **ArrayList <Long> semillas:** Almacena todas las semillas que vamos utilizar en las distintas ejecuciones. En nuestro caso, vamos a generar 5 semillas partiendo de una original 20887601(las siguientes cuatro se crean haciendo permutaciones de la original), con lo cúal los resultados con estas semillas siempre deberían de ser los mismos.

- **Dimension:** Almacena el valor del tamaño del vector (d, que por defecto es 10) que reciben nuestras funciones de evaluación.

- **maxIndividuos:** Almacena el valor de individuos que como máximo se van a evaluar. Establece la condición de parada para los algoritmos.

- **tamPoblacion:** Almacena el número de individuos que va tener cada una de las poblaciones que se van a generar.

- **ProbCruce:** Almacena el valor de la probabilidad de cruce para cada uno de los individuos a generar. Por defecto, este valor está establecido al 70 % => 0.7

- **ProbMutacion:** Almacena el valor de la probabilidad de mutación para cada uno de los individuos generados. Por defecto, este valor está establecido al 1 % => 0.01

- **ProbCambio:** Almacena el valor de la probabilidad de cambio para escoger el valor de esa posición pero de la función objetivo. Por defecto, este valor está establecido al 50 % => 0.5

- **k:** Indica el número de individuos con los que se va a realizar el torneo para llevar a cabo el operador de selección. Por defecto tiene un valor de 2.

- **numPadres:** Indica el número de padres que se van a escoger para realizar posteriormente el operador de cruce entre todos estos. Por defecto tiene un valor de 2.

- **numElites:** Indica el número de individuos que se van a escoger como elites. Estos pasarán a la población hija forzosamente. Por defecto tiene un valor de 1.

- **kIndObj:** Indica el número de individuos con los que realizar el tornero para escoger la función objetivo. Por defecto tiene un valor de 3.

- **NumDecimales:** Almacena el valor del número de decimales que como máximo van a tener todos los valores obtenidos (Podemos ser más exactos o menos). Tal y como lo hemos implementado nosotros, siempre redondeamos, es decir, no truncamos los valores.

4.2 Métodos utilizados

Vamos a exponer algunos de los métodos más utilizados para la ejecución de nuestro programa.

- **double evaluar (double[] vector):** Este método es el principal de todas las funciones matemáticas implementadas. Aquí es donde se calcula un valor dado un vector y posteriormente se devuelve dicho valor.

- **double getRango_min ():** Este método devuelve el valor del rango mínimo en el que se evalúa la función, este dependerá de cada función como hemos visto es la explicación de cada una de ellas.

- **double getRango_max ():** Este método devuelve el valor del rango máximo en el que se evalúa la función, este dependerá de cada función como hemos visto es la explicación de cada una de ellas.

- **double [] algoritmoEvolutivo (Evaluador exe, long semilla, TablaDatos datos):** Este método es principal de cada uno de los algorimos que se han implementado, ya que contiene el código propiamente dicho. Para ello recibe la función con la que queremos evaluar (exe), la semilla con la que se va a realizar toda la ejecución y finalmente la tabla de datos, donde se van a ir almacenando los datos para luego generar el fichero de salida.

- **ArrayList<ArrayList<double []>> getSoluciones():** Devuelve una lista con todas las soluciones por las que se han ido pasando a lo largo de la búsqueda.

- **limpiaAlgoritmo():** Limpia todos los atributos que tenía almacenados de ejecuciones anteriores, para que estas sean lo más independientes posible.

5. Análisis de los algoritmos

Una vez implementados los distintos algoritmos, vamos a pasar a la fase de análisis. Para ello, vamos a generar una tabla de datos donde para un algoritmo, semilla y función de evaluación concreta podamos ver cúal ha sido su desviación del error y el tiempo que ha tardado en ejecutar el algoritmo.

En primer lugar, por cada algoritmo que ejecutamos con una de las semillas y evaluamos con cada una de las funciones de evaluación (las 10 explicadas anteriormente), esto nos generará un conjunto de soluciones, desde la inicial hasta la final (aunque solo nos quedamos con la final/mejor). Todas estas soluciones se irán escribiendo en unos ficheros llamados **algoritmo_semilla_nombreFuncion.txt** (en la carpeta logs), los cuales nos permitirán revisar de forma más exhaustiva cuántas y cuáles son las soluciones que han ido generando cada una de las generaciones, dado un algoritmo y una semilla concreta.

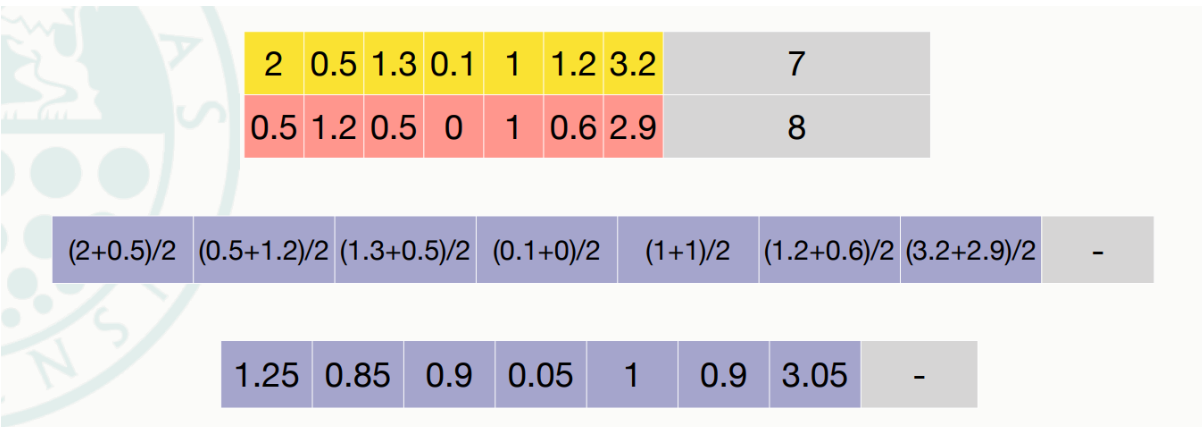
En segundo lugar, a diferencia del anterior, en este caso, vamos a generar un fichero **salida.txt**, el cual se encargue de almacenar los valores finales que irán en la tabla. Para ello, almacenamos todos estos valores en formato CSV (Comma Separated Values), que nos permitirá importar los datos en una hoja de cálculo de una forma mucho más sencilla.

Para que queden más claros los datos, primero vamos a mostrar los resultados de las distintas ejecuciones (una por cada semilla) con respecto a las **soluciones** y posteriormente con respecto al **tiempo de ejecución** (medido en milisegundos **ms**).

5.1 Algoritmo evolutivo (EvM)

Este algoritmo es uno de los algoritmos que hemos implementado y su funcionamiento se basa en la idea expuesta anteriormente.

La principal característica de este algoritmo es que el operador de cruce que se aplica, se basa en una media aritmética por cada gen del genotipo de los padres.

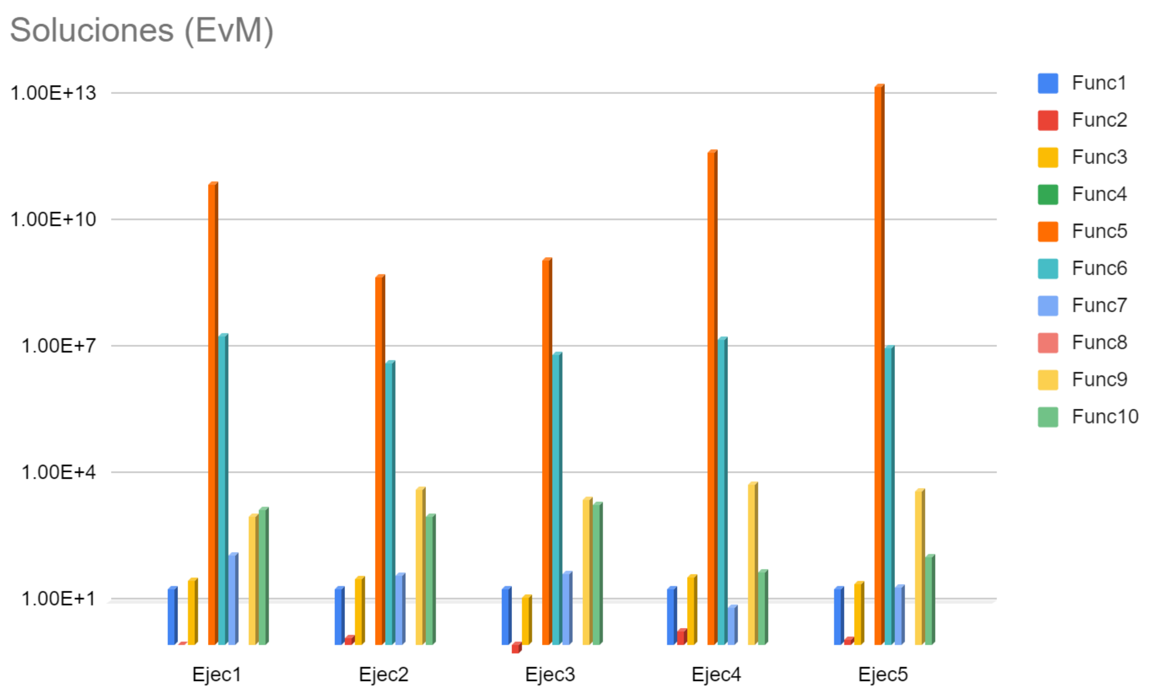


Como vemos cada uno de los alelos del individuo hijo se ha generado como la media aritmética de los padres.

Ahora vamos a exponer los diferentes resultados para cada una de la ejecuciones que se han realizado, para hacer un estudio de la calidad de nuestro algoritmo y finalmente compararlo con el resto de los que se han implementado.

Primero vamos a ver la tabla de los resultados con las diferentes soluciones.

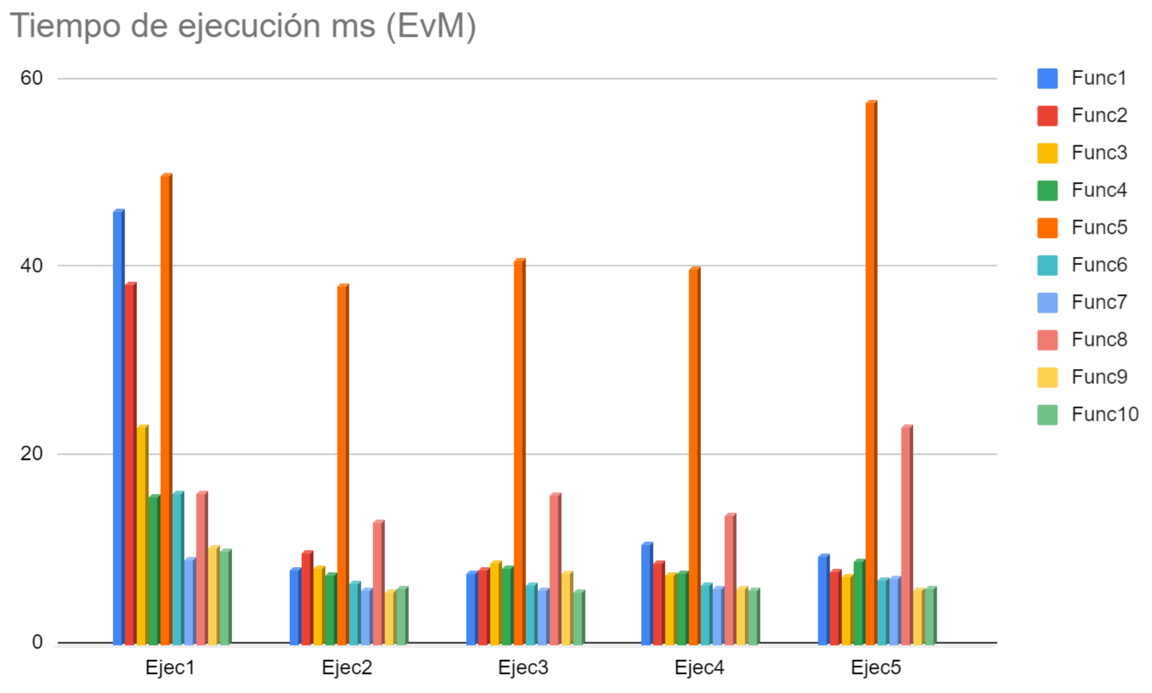
EvM	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.252	0.9633	33.8445	-2.81E+04	7.91E+10	2.06E+07	127.3069	-8.7483	1.09E+03	1.53E+03
Ejec2	20.2901	1.3975	37.2515	-1.88E+04	5.05E+08	4.70E+06	45.1173	-7.1678	4.68E+03	1.12E+03
Ejec3	20.4884	0.6293	12.8053	-2.39E+04	1.27E+09	7.26E+06	46.9721	-6.2073	2.69E+03	2.14E+03
Ejec4	20.2739	2.0255	37.8988	-2.33E+04	4.56E+11	1.68E+07	7.5205	-7.618	6.29E+03	5.08E+01
Ejec5	20.3692	1.3209	26.2846	-1.91E+04	1.75E+13	1.09E+07	21.6404	-7.2253	4.30E+03	1.15E+02
Media	20.3347	1.2673	29.6169	-2.26E+04	3.60E+12	1.21E+07	49.7114	2.2668	4.02E+03	9.90E+02
Desv.	0.0966	0.5230	10.4700	3.85E+03	7.75E+12	6.60E+06	46.4158	0.9182	1.99E+03	9.05E+02



Hay que añadir que esta tabla está en formato logarítmico, lo cuál muestra mejor los datos más pequeños (los datos más grandes se salen de la gráfica hacia arriba).

Ahora vamos a ver todos los tiempos de ejecución medidos en milisegundos, para posteriormente hacer un estudio de cuál ha sido la mejor ejecución.

EvM	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	46.1369	38.4533	23.1608	15.6813	50.0323	16.1825	9.1298	16.0764	10.2564	9.9903
Ejec2	8.0338	9.7083	8.1021	7.5121	38.2621	6.4705	5.8432	13.0879	5.6111	6.0018
Ejec3	7.6756	7.9744	8.7424	8.0858	40.9773	6.4463	5.8267	15.9598	7.5408	5.6886
Ejec4	10.6372	8.6861	7.4736	7.6813	40.0367	6.4059	6.0862	13.6954	5.9194	5.7366
Ejec5	9.4741	7.8831	7.2863	8.8127	57.646	6.8559	7.004	23.2325	5.8099	5.951
Media	16.3915	14.5410	10.9530	9.5546	45.3909	8.4722	6.7780	16.4104	7.0275	6.6737
Desv.	16.6701	13.3873	6.8483	3.4614	8.2264	4.3140	1.4000	4.0394	1.9625	1.8589



Como el tiempo de ejecución es inferior a los 60ms se puede considerar que es despreciable. Por tanto, la mejor ejecución será aquella que haya obtenido los valores más cercanos a los óptimos globales de cada una de las funciones.

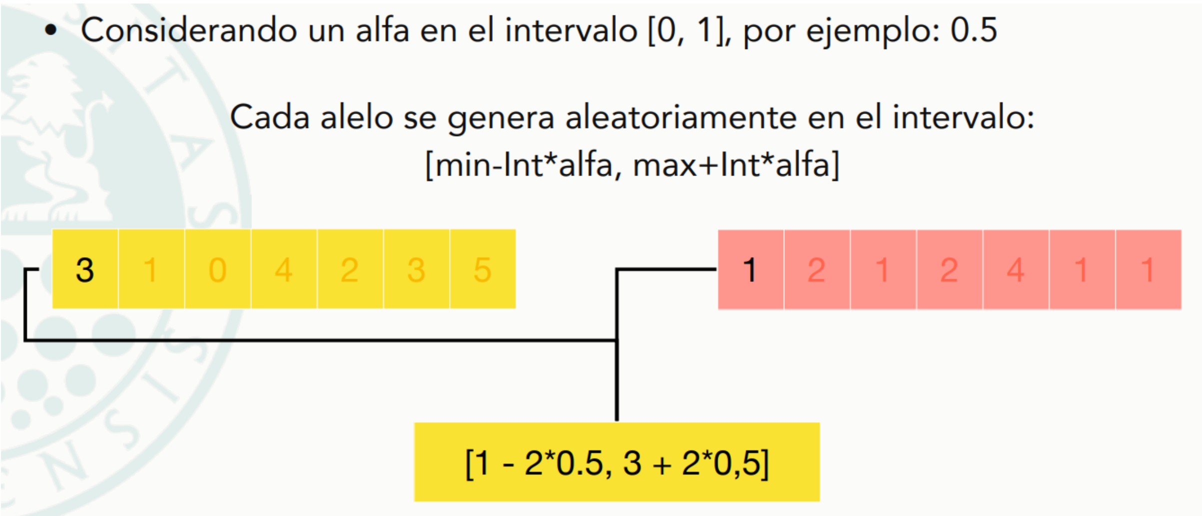
Por tanto en nuestro caso podemos afirmar que la mejor ejecución ha sido la número 2, que utiliza la semilla 88760120.

UNIVERSIDAD DE JAÉN

5.2 Algoritmo evolutivo (EvBLX)

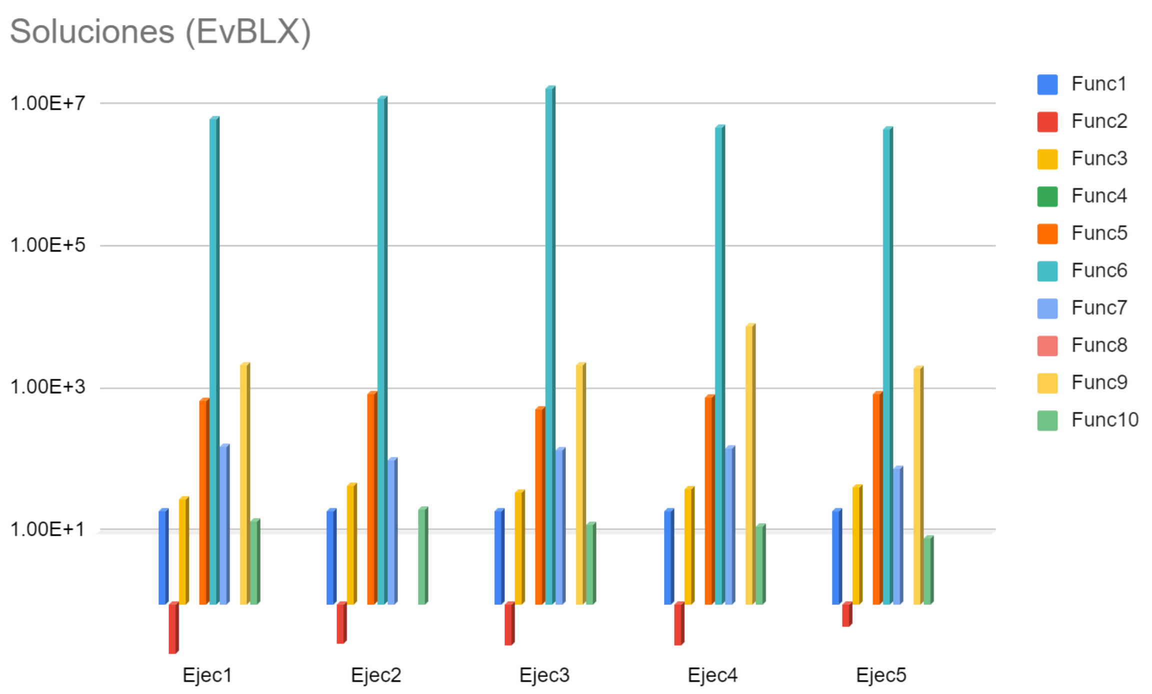
Este algoritmo funciona de la misma manera que el anterior. La única diferencia es el operador de cruce que utiliza para obtener los individuos hijos.

Para ello utiliza el llamado operador de cruce BLX-alfa, el cuál genera cada alelo de forma aleatoria entre un rango máximo y mínimo definido de la siguiente manera.



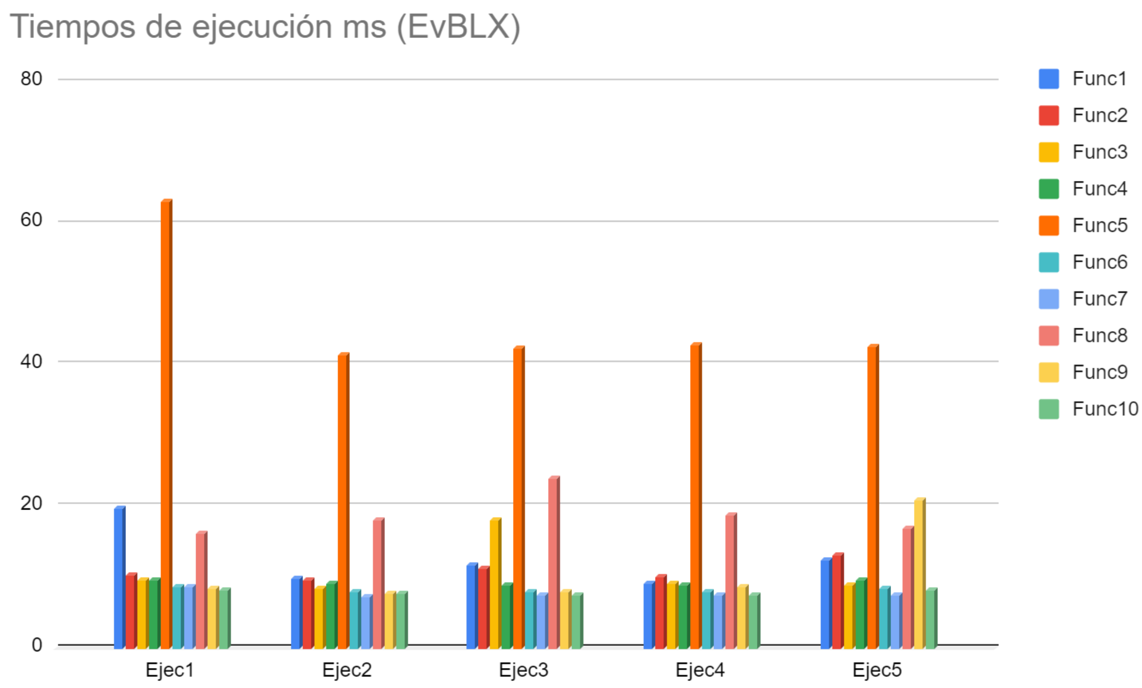
Una vez visto el funcionamiento de este algoritmo vamos a ver la tabla de soluciones que hemos obtenido después de hacer su implementación.

EvBLX	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.3924	0.1972	29.1254	-2.05E+04	718.9066	6.73E+06	166.2178	-4.6301	2.34E+03	14.3723
Ejec2	20.4595	0.277	46.7333	-2.82E+04	894.3038	1.31E+07	103.4686	-4.4637	-1.06E+02	21.1673
Ejec3	20.5916	0.2606	36.3267	-2.76E+04	537.6243	1.85E+07	148.8769	-4.4745	2.32E+03	12.8846
Ejec4	20.3208	0.2583	41.3988	-2.88E+04	827.327	5.15E+06	151.4947	-4.3627	8.25E+03	12.5711
Ejec5	20.4701	0.4778	42.4242	-2.61E+04	892.0551	4.84E+06	79.0525	-6.9136	2.03E+03	8.4066
Media	20.4469	0.2942	39.2017	-2.63E+04	7.74E+02	9.66E+06	129.8221	4.6912	3.18E+03	13.8804
Desv.	0.1006	0.1070	6.7404	3.36E+03	1.50E+02	5.96E+06	36.8425	1.0913	3.12E+03	4.6371



Ahora vamos a ver la tabla de tiempos que hemos obtenido.

EvBLX	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	19.7239	10.3096	9.7044	9.5727	63.0913	8.6347	8.703	16.106	8.402	8.0736
Ejec2	9.8533	9.5478	8.495	9.1804	41.4583	7.9615	7.2906	18.0334	7.6875	7.6113
Ejec3	11.7062	11.3503	17.9982	8.8664	42.2385	7.9563	7.472	24.0677	8.0528	7.4028
Ejec4	9.0627	9.953	9.0545	8.834	42.9061	7.8337	7.5164	18.8141	8.6129	7.5134
Ejec5	12.4923	13.0658	8.9438	9.6198	42.5986	8.3698	7.5806	16.8792	20.9197	8.0719
Media	12.5677	10.8453	10.8392	9.2147	46.4586	8.1512	7.7125	18.7801	10.7350	7.7346
Desv.	4.2312	1.4100	4.0253	0.3741	9.3137	0.3378	0.5641	3.1338	5.7043	0.3174



Para este algoritmo también podemos considerar que los tiempos de ejecución son insignificantes. Por tanto, se puede afirmar que la mejor ejecución ha sido la cuarta, que utiliza la semilla 76012088.

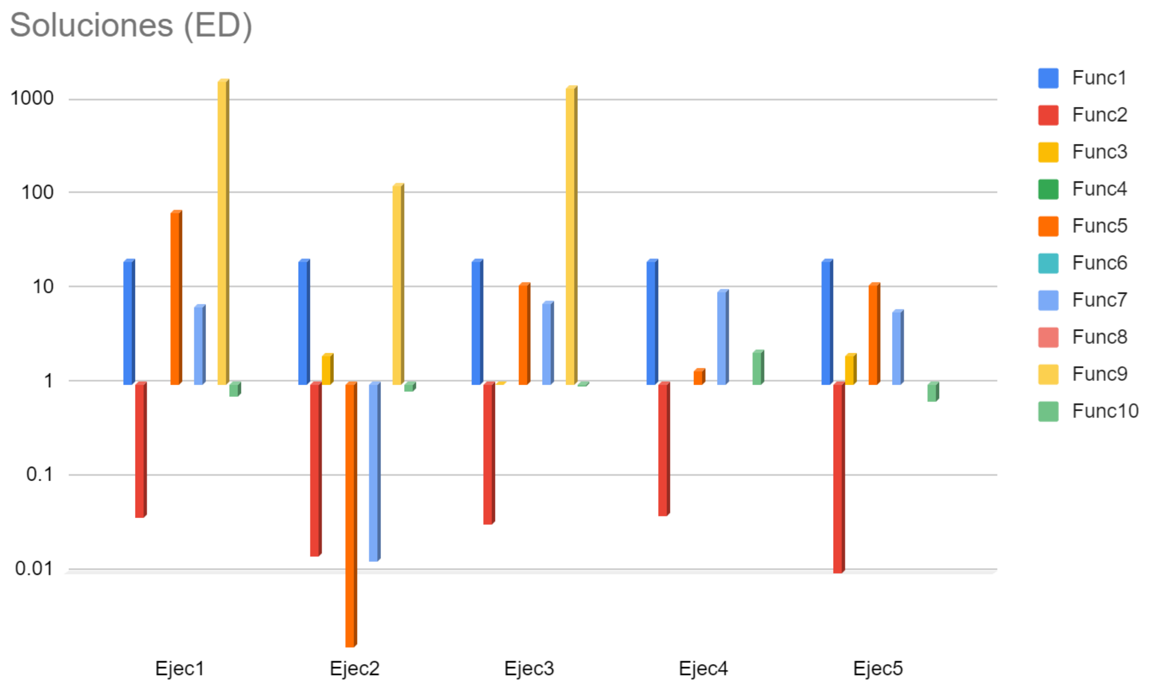
5.3 Algoritmo evolutivo diferencial (ED)

Este tipo de algoritmo explicado anteriormente, utiliza una técnica bastante diferente a los dos anteriores, aplicando un operador de recombinación ternario.

Esto permite realizar cruces y mutación sobre los individuos para generar la población hija. Este tipo de algoritmos están extremadamente enfocados en resolver este tipo de problemas.

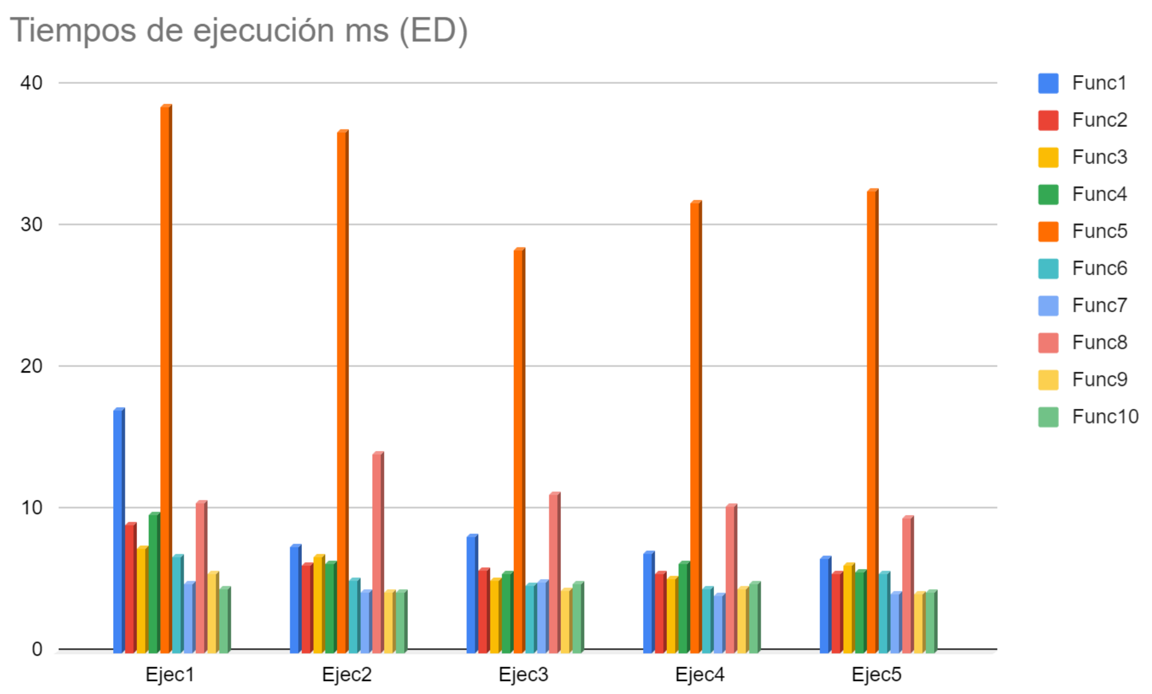
Ahora vamos a exponer los resultados que hemos obtenido tras ejecutar este algoritmo.

ED	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.0204	0.0382	0	-1.63E+08	65.6229	0	6.4749	-9.2487	1652.3078	0.7272
Ejec2	20.0072	0.0148	1.9899	-8.45E+08	0.0016	0	0.0133	-9.0028	129.1431	0.8215
Ejec3	20.0154	0.032	0.995	-1.43E+09	11.0993	0	6.9866	-9.582	1387.8741	0.9298
Ejec4	20.0154	0.0395	0	-2.25E+10	1.3495	0	9.4371	-9.389	-181.0892	2.1436
Ejec5	20.0082	0.0099	1.9899	-1.76E+10	11.0983	0	5.7527	-9.725	-196.4707	0.6667
Media	20.0133	0.0269	0.9950	-8.51E+09	17.8343	0	5.7329	0.2707	768.3530	1.0578
Desv.	0.0055	0.0137	0.9950	1.07E+10	27.2224	0	3.4842	0.2824	892.4121	0.6151



Ahora vamos a exponer la tabla de tiempos que hemos obtenido tras estas ejecuciones.

ED	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	17.1622	9.0773	7.4132	9.7049	38.4988	6.7686	4.8982	10.6314	5.531	4.5201
Ejec2	7.4881	6.1495	6.7682	6.3554	36.8102	5.056	4.2457	14.0723	4.2973	4.2522
Ejec3	8.2164	5.765	5.0662	5.5328	28.4765	4.6982	4.9337	11.1838	4.3452	4.8153
Ejec4	7.0521	5.5974	5.1881	6.3393	31.7632	4.5227	4.0699	10.3313	4.5194	4.8495
Ejec5	6.7044	5.5978	6.1379	5.642	32.5566	5.5836	4.1254	9.5488	4.1461	4.2453
Media	9.3246	6.4374	6.1147	6.7149	33.6211	5.3258	4.4546	11.1535	4.5678	4.5365
Desv.	4.4175	1.4929	1.0089	1.7146	4.0316	0.9027	0.4261	1.7352	0.5547	0.2922



Como vemos los tiempos de ejecución en este caso también son despreciables, ya que son inferiores a 40ms.

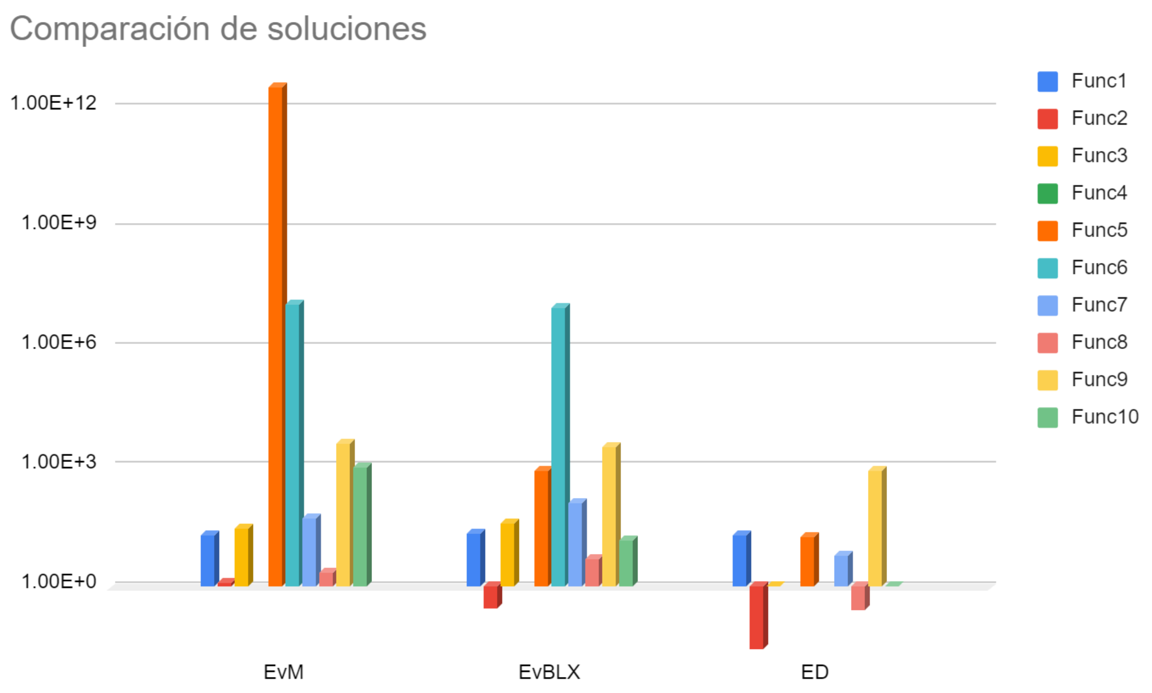
Por tanto, podemos decir que la mejor ejecución ha sido la ejecución 1, que utiliza la semilla 20887601.

5.4 Comparación entre algoritmos

Una vez hemos visto el análisis de cada uno de los algoritmos implementados, vamos a comparar los resultados de forma exhaustiva para poder tomar conclusiones y decidir finalmente cual es el algoritmo que obtiene mejores resultados.

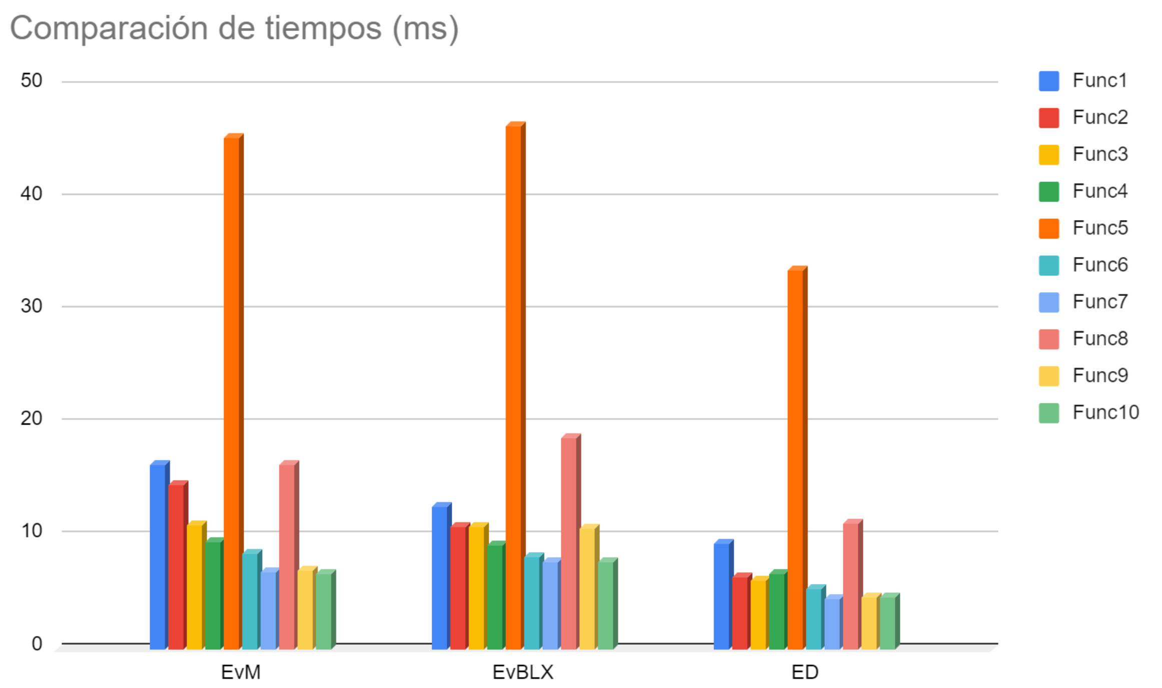
Para ello tal y como hemos hecho anteriormente, vamos a realizar dos tablas. Una con todos los datos de las medias de las desviaciones de error y otra con todas las medias de los tiempos de ejecución por cada uno de los algoritmos, semillas y funciones.

Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
EvM	20.33472	1.2673	29.61694	-22643.305	3.60E+12	1.21E+07	4.97E+01	2.26681	4.02E+03	9.90E+02
EvBLX	20.44688	0.29418	39.20168	-26254.87982	7.74E+02	9.66E+06	1.30E+02	4.69123	3.18E+03	1.39E+01
ED	20.01332	0.02688	0.99496	-8514268255	1.78E+01	0.00E+00	5.73E+00	0.27065	7.68E+02	1.06E+00



Ahora vamos a ver la comparación de tiempos entre los diferentes algoritmos.

Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
EvM	16.39152	14.54104	10.95304	9.55464	45.39088	8.47222	6.77798	16.4104	7.02752	6.67366
EvBLX	12.56768	10.8453	10.83918	9.21466	46.45856	8.1512	7.71252	18.78008	10.73498	7.7346
ED	9.32464	6.4374	6.11472	6.71488	33.62106	5.32582	4.45458	11.15352	4.5678	4.53648



Como vemos el algoritmo que ha tardado menos en ejecutarse es el “evolutivo diferencial”, además, este mismo algoritmo ha sido el que ha obtenido mejores resultados, ya que sus soluciones se han aproximado muchos a los óptimos globales de las funciones y en algunos casos a sido capaz de encontrarlos.

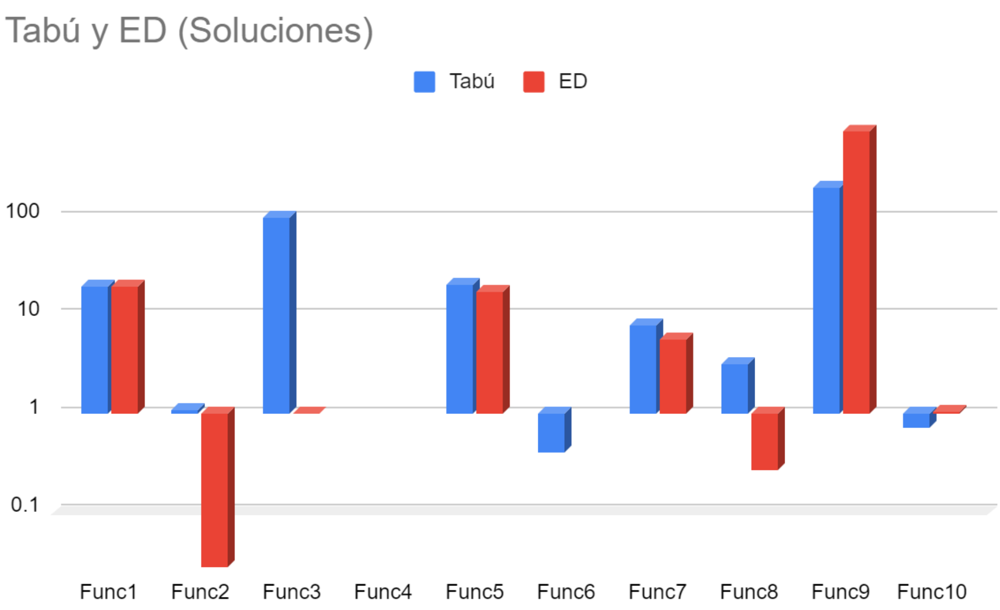
Por tanto, podemos afirmar que el algoritmo que obtiene mejores resultados en este tipo de problemas es el algoritmo evolutivo diferencial.

5.5 Comparación entre algoritmo trayectorias y poblaciones

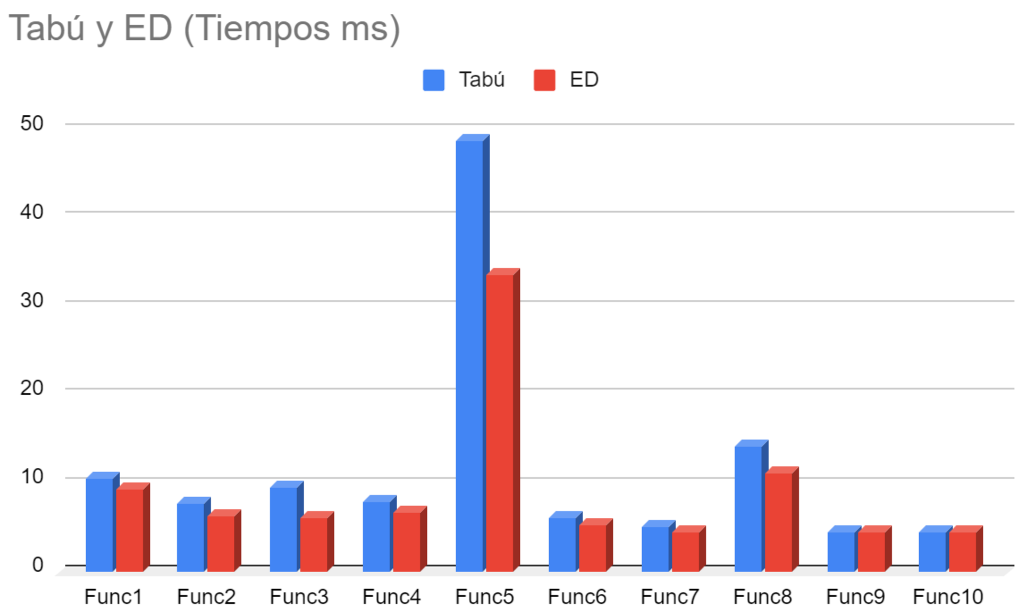
En la práctica anterior, implementamos varios algoritmos de trayectorias y finalmente pudimos definir cuál era el mejor de todos ellos para resolver este mismo problema (encontrar el óptimo global de las funciones comentadas anteriormente).

Por tanto, ahora vamos a hacer una comparativa entre el mejor algoritmo de trayectorias implementado (Búsqueda tabú) y el mejor algoritmo de poblaciones comentado en el apartado anterior (Algoritmo de evolución diferencial).

Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Tabú	20.20454	1.0924	101.75906	-20040.8786	2.11E+01	4.05E-01	8.23E+00	3.30673	2.07E+02	7.33E-01
ED	20.01332	0.02688	0.99496	-8514268255	1.78E+01	0.00E+00	5.73E+00	0.27065	7.68E+02	1.06E+00



Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Tabú	10.6168	7.71952	9.65324	8.0587	48.91158	6.19368	5.1603	14.29538	4.4895	4.47848
ED	9.32464	6.4374	6.11472	6.71488	33.62106	5.32582	4.45458	11.15352	4.5678	4.53648



Vemos que los resultados han sido ligeramente mejores en el caso del algoritmo de evolución diferencial. Esto se debe a que este algoritmo realiza una mayor exploración del espacio (puede encontrar soluciones más prometedoras). Además, los tiempos son son más reducidos en este también ya que solo aplica el operador de recombinación ternario que a fin de cuentas es una operación matemática (es relativamente rápido de aplicar).

6. Aplicación de los algoritmos en caso real

Una vez hemos visto el funcionamiento y hemos hecho un estudio sobre los algoritmos implementados, para saber la calidad de los mismos, vamos a aplicarlos en un caso real, es decir, sobre datos que se han tomado en una situación real.

Estos datos reales se tratan de una muestra sobre placas fotovoltaicas, ya que, la empresa que recogió estos datos está interesada en saber cuáles de todos estos valores son los que tienen más relevancia en la potencia generada por un módulo., teniendo en cuenta la irradiancia, temperatura, viento ambiental y el SMR.

$$P_m = DNI\left(a_1^a + a_2^a DNI + a_3^a T_A + a_4^a W_S + a_5^a SMR\right)$$

Por tanto, la idea en este caso es modificar los valores a1, a2, a3, a4 y a5 gracias a los algoritmos implementados, para intentar reducir al mínimo el error respecto al conjunto de observaciones que ha realizado la empresa previamente.

Para realizar la evaluación de un individuo y asignarle un fitness se hará con una nueva función que hemos llamado potencia, la cuál puede aplicar la fórmula MAPE o también RMSE. Así podremos comparar las soluciones que nos dan ambas.

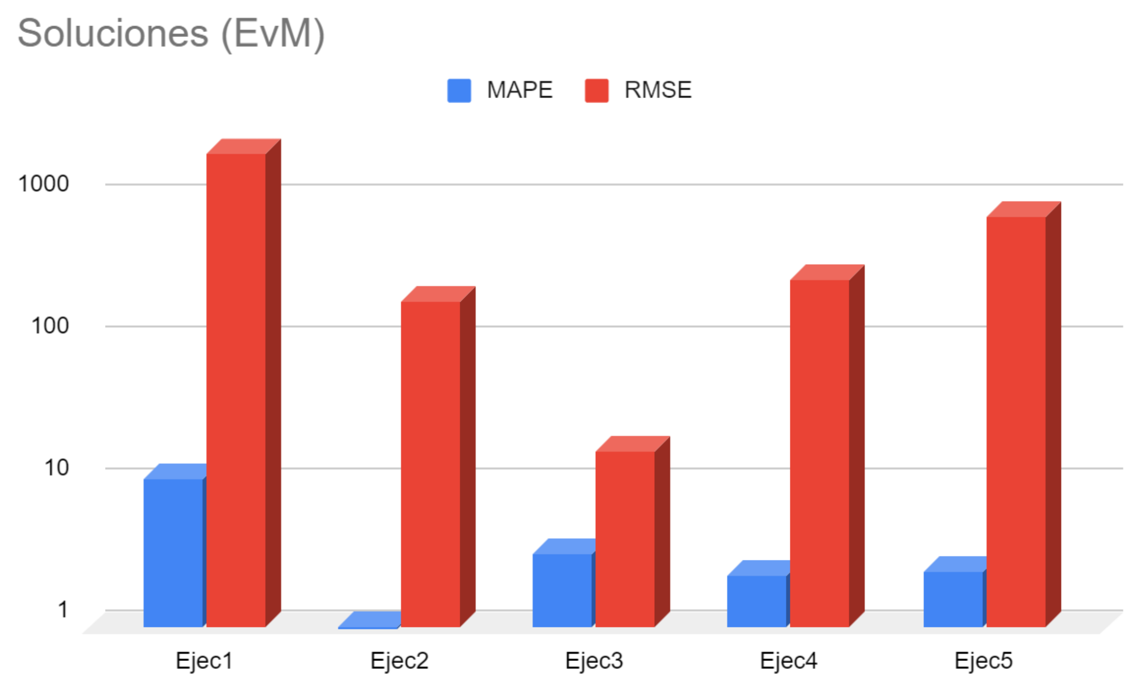
$$\mathrm{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \frac{|\hat{y}_i - y_i|}{y_i} \qquad RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

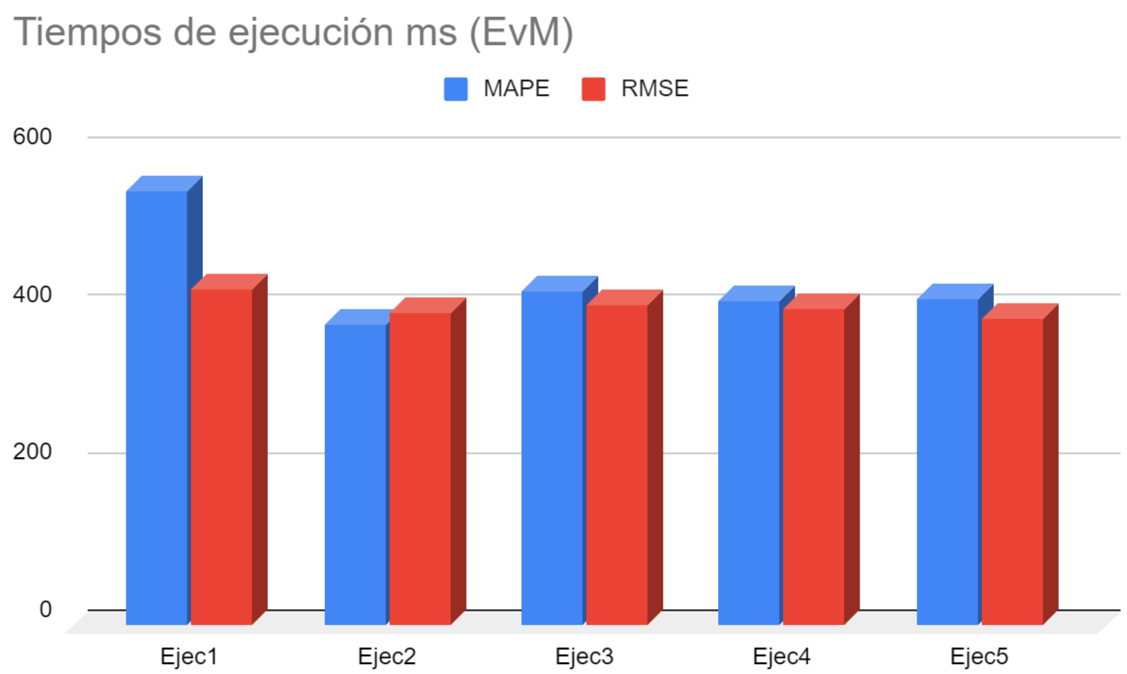
Ahora vamos a ver los resultados que hemos obtenido por cada uno de los algoritmos implementados.

6.1 Resultados con algoritmo evolutivo (EvM)

Esta vez en la misma tabla vamos a poner los valores de las soluciones y de tiempo, ya que solo hay que mostrar los resultados de los valores obtenidos con Potencia_MAPE y Potencia_RMSE.

EvM	**MAPE**	**MAPE**	**RMSE**	**RMSE**
	Solución	Tiempo	Solución	Tiempo
Ejec1	10.7917	551.8316	2104.7816	427.1911
Ejec2	0.9434	382.9508	190.2705	397.9022
Ejec3	3.2696	425.3008	16.8094	406.6242
Ejec4	2.2959	411.5359	276.6218	402.8936
Ejec5	2.4068	413.6754	764.4406	388.813
Media	3.9415	437.0589	670.5848	404.6848
Desv.	3.9188	66.0245	848.3897	14.2427



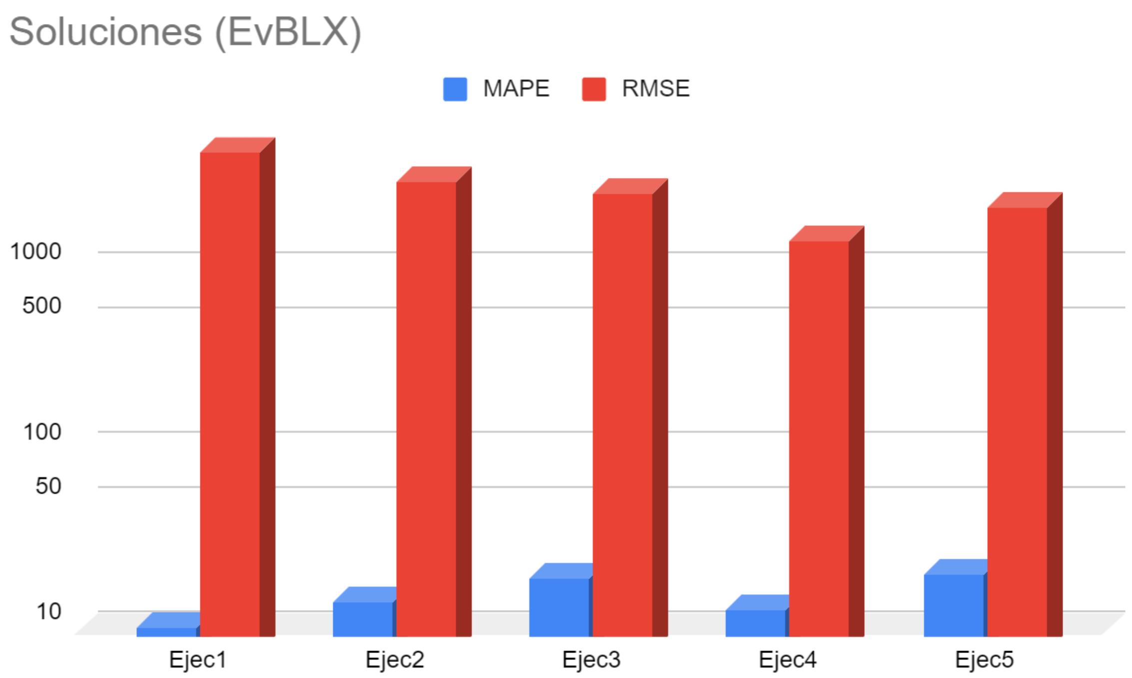


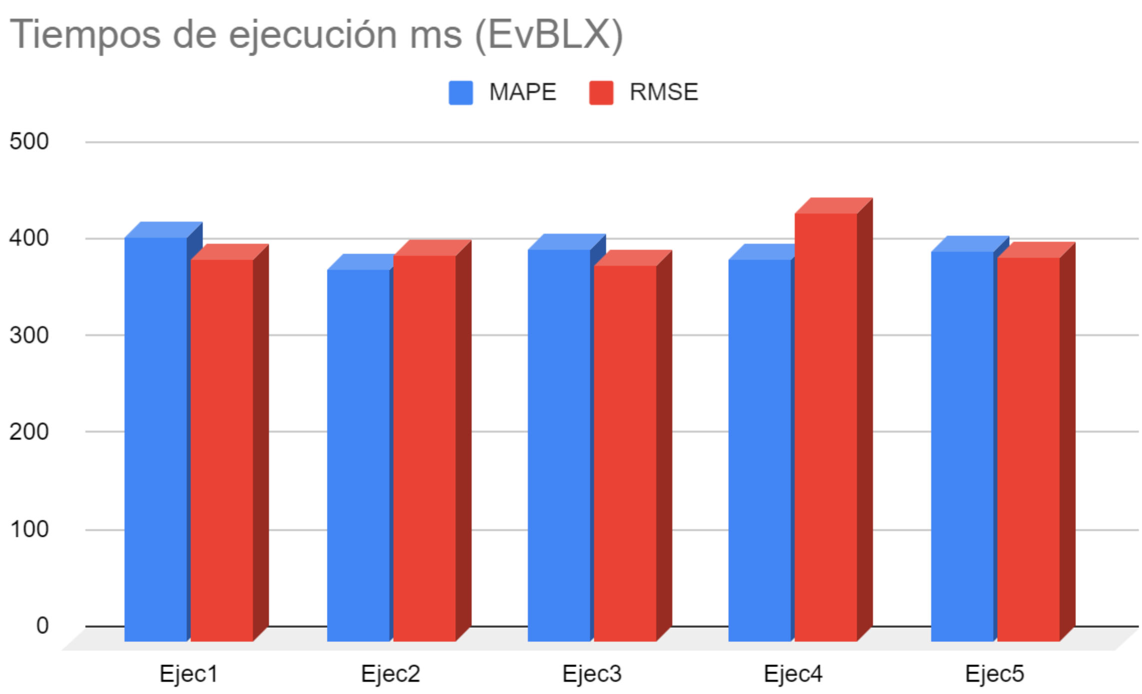
Los tiempos de ejecución son mucho mayores en este caso, ya que la evaluación de un individuo es relativamente costosa, ya que se tiene que hacer con respecto a toda la muestra dada.

Esto hace que el tiempo sea relativamente notable aunque ni siquiera llega a un segundo. Si nos fijamos en las soluciones arrojadas, vemos que la mejor ejecución para Potencia_MAPE ha sido la segunda con la semilla 8876012 y para el caso de Potencia_RMSE ha sido la tercera que utiliza la semilla 88760120.

6.2 Resultados con algoritmo evolutivo (EvBLX)

EvBLX	MAPE	MAPE	RMSE	RMSE
	Solución	Tiempo	Solución	Tiempo
Ejec1	9.8555	417.9906	4368.5606	394.92
Ejec2	13.9273	384.3014	3049.9706	398.1144
Ejec3	18.6111	405.645	2592.167	388.3718
Ejec4	12.554	395.1115	1428.9469	443.1475
Ejec5	19.9067	402.7954	2182.0633	396.2841
Media	14.9709	401.1688	2724.3417	404.1676
Desv.	4.2045	12.5185	1095.3394	22.0978



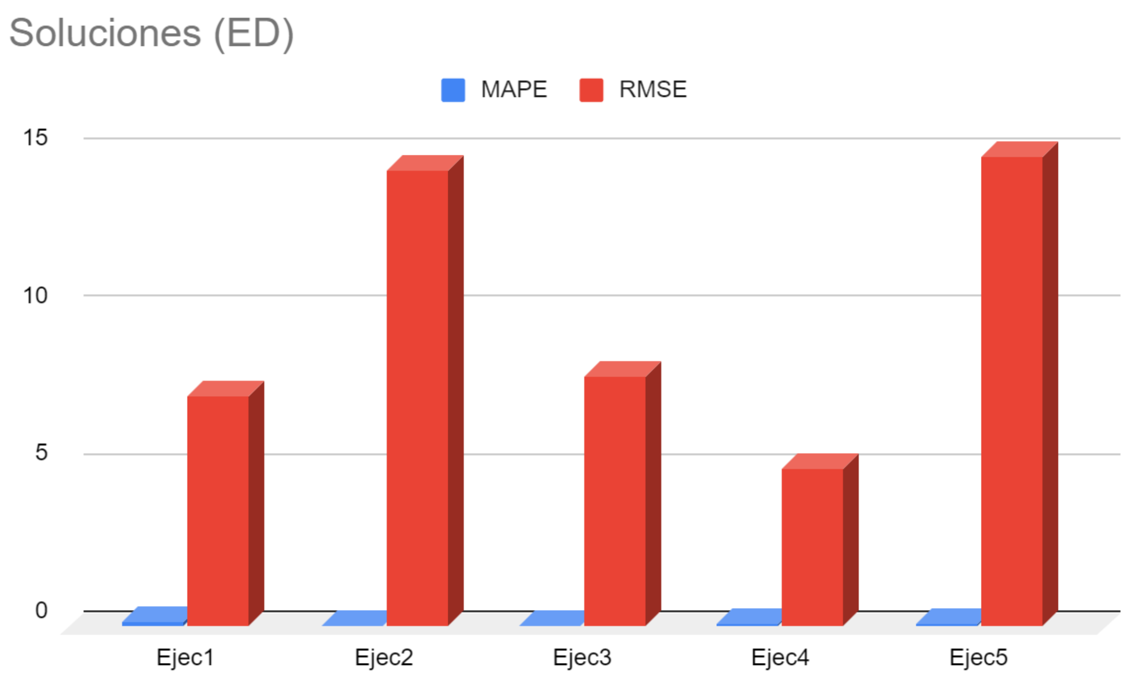


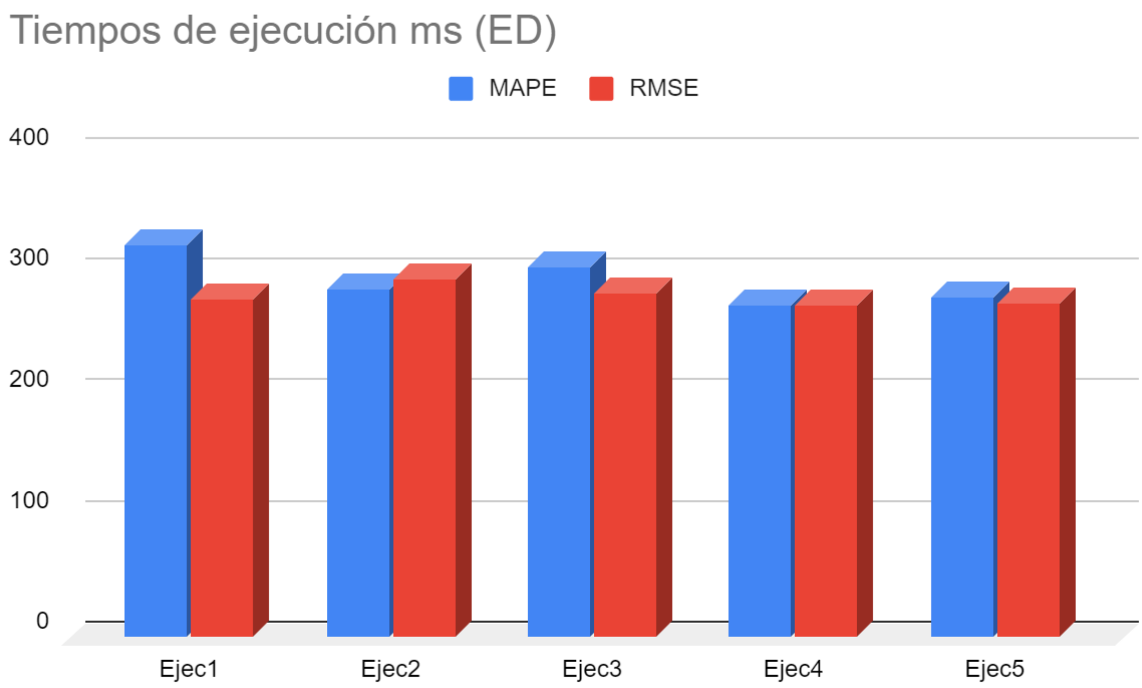
En este caso los tiempos de ejecución también son notables, pero aún así siguen siendo bastante reducidos, ya que están por debajo de medio segundo.

Por tanto, la mejor ejecución para la función de Potencia_MAPE ha sido la primera, que utiliza la semilla 20887601 y para el caso de Potencia_RMSE ha sido la cuarta que utiliza la semilla 76012088.

6.3 Resultados con algoritmo de evolución diferencial (ED)

ED	**MAPE**	**MAPE**	**RMSE**	**RMSE**
	Solución	Tiempo	Solución	Tiempo
Ejec1	0.1295	324.2921	7.287	279.0705
Ejec2	0.0349	287.0306	14.4487	295.3415
Ejec3	0.0349	306.0799	7.9432	283.9635
Ejec4	0.0773	275.0639	5.0091	275.1728
Ejec5	0.0993	280.2777	14.8796	276.7809
Media	0.0752	294.5488	9.9135	282.0658
Desv.	0.0412	20.3561	4.4739	8.1281


Soluciones (ED)
MAPE
RMSE
15
10
5
0
Ejec1
Ejec2
Ejec3
Ejec4
Ejec5

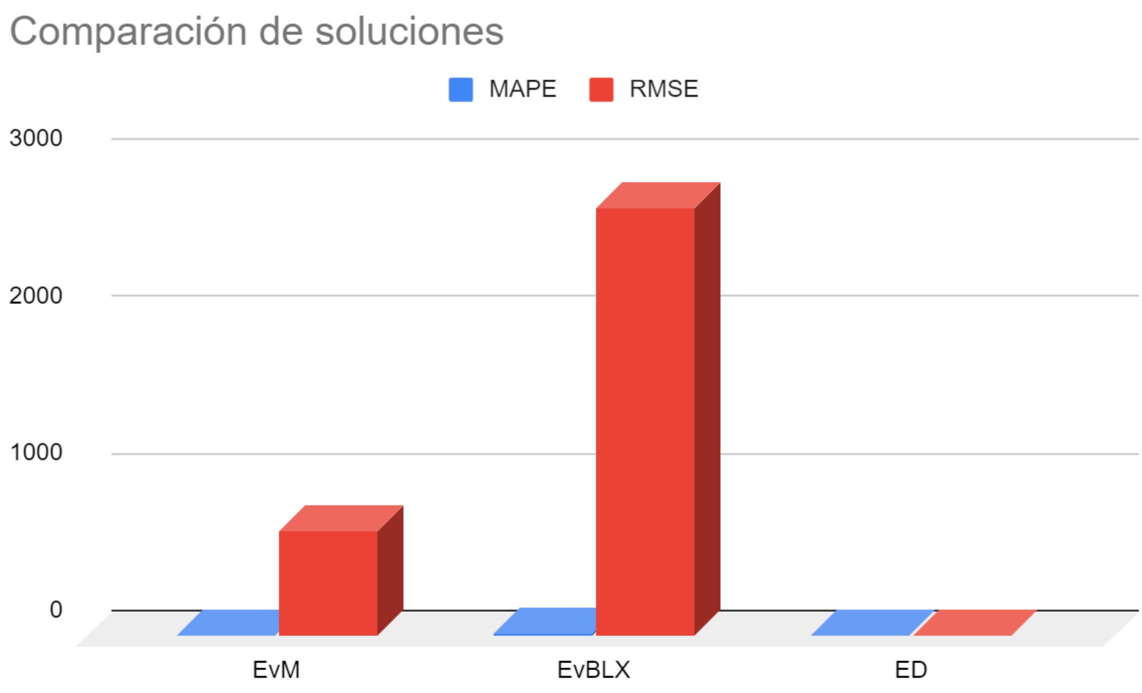

Tiempos de ejecución ms (ED)
MAPE
RMSE
400
300
200
100
0
Ejec1
Ejec2
Ejec3
Ejec4
Ejec5

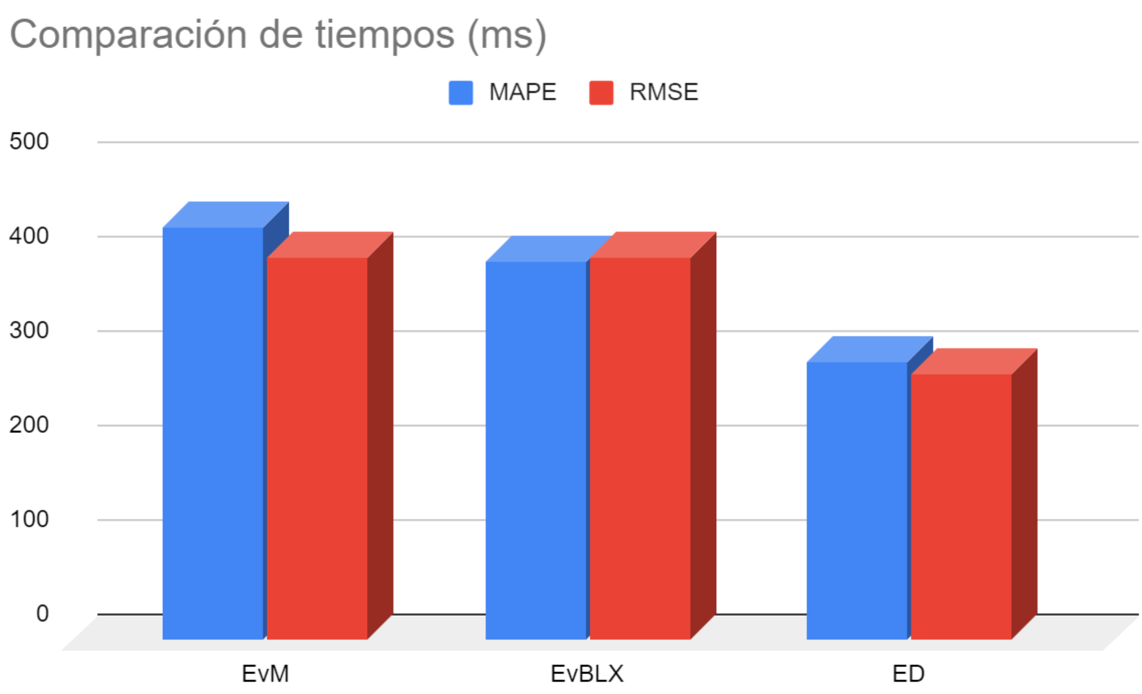
6.4 Comparación de los resultados

Ahora una vez que hemos expuesto los datos que hemos obtenido tras realizar las distintas ejecuciones, vamos a hacer una comparativa entre estos algoritmos en base a estos datos recogidos.

En el caso de RMSE los valores obtenidos siempre son mayores por la propia definición de la fórmula.

Alg.	**MAPE** Solución	**MAPE** Tiempo	**RMSE** Solución	**RMSE** Tiempo
EvM	3.94148	437.0589	670.58478	404.68482
EvBLX	14.97092	401.16878	2724.34168	404.16756
ED	0.07518	294.54884	9.91352	282.06584





Como vemos los mejores resultados han sido obtenidos por el algoritmo de evolución diferencial, ya que es el que ha obtenido soluciones con menor desviación de error en base a la muestra. Además, este lo hace incluso en menor tiempo que el resto de algoritmos.

Esto se debe a que este algoritmo simplemente aplica un único operador de recombinación ternario, por tanto, solo se hace un cálculo matemático que es mucho más rápido que hacer todo el resto de operadores que aplican el resto de algoritmos, además de tener en cuenta los élites.

6.5 Conclusión

Para saber cuáles son los datos más relevantes en la generación de potencia en los módulos fotovoltaicos, vamos a fijarnos en el algoritmo de evolución diferencial (ED), el cuál ha sido el que ha obtenido mejores resultados.

Los valores de la mejor solución obtenida han sido:

a1 (Indepen.)	a2 (DNI)	a3 (Ta)	a4 (Ws)	a5 (SMR)
0.1119	-9.433E-6	-7.366E-5	4.973E-4	0.01962

Como sabemos estos valores están comprendidos entre [-1, 1], por tanto cuanto más cerca del cero estén estos valores podemos decir que ese término tiene menor relevancia (al multiplicar en la fórmula se hace cero).

Por tanto si ordenamos los valores por orden de relevancia tendríamos:

1. **a1:** Término independiente
2. **a5:** SMR
3. **a4:** Velocidad del viento (Ws)
4. **a3:** Temperatura (Ta)
5. **a2:** Irradiancia (DNI)

7. Enlaces relacionados

- **Archivos de logs:** https://drive.google.com/drive/folders/1I_yNqRGThTxi8pbK2P3o3Fdo1PlElZgP?usp=sharing

- **Optimization Test Functions:** https://www.sfu.ca/~ssurjano/optimization.html

- **Algoritmos genéticos:** http://www.cs.us.es/~fsancho/?e=65