Algoritmos Bioinspirados: Evolución Diferencial

Alberto García y Diego Martínez

3 de mayo de 2020

En este trabajo hemos implementado la opción T3_DE, correspondiente al algoritmo de evolución diferencial desarrollado en el artículo de Tian, Gao y Dai [1]. Este informe contiene en sus secciones, la descripción del algoritmo, un resumen de los problemas a resolver, la estructura y los parámetros del experimento, los resultados y conclusiones, las referencias y un anexo con el código del programa.

1. Descripción del Algoritmo

Este algoritmo es de tipo evolutivo, luego su esquema general será el siguiente. Crearemos un conjunto de **mutaciones**, haremos un **crossover** entre la población de padres y las mutaciones para crear una nueva población de hijos, y realizaremos una **selección** para seleccionar los mejores mejores individuos entre la población de padres e hijos.

En concreto este algoritmo es de evolución diferencial, lo que quiere decir que la estrategia para las mutaciones está basada en diferencias entre individuos. Esto implica que la elección individuos más o menos "buenos" para calcular estas diferencias contribuye a que el algoritmo tenga un comportamiento más convergente o más exploratorio. En concreto, este algoritmo tiene un modelo híbrido con algunos parámetros autoadaptativos, de forma que especialmente en las en primeras generaciones y cuando el porcentaje de hijos que suceden a los padres es bajo, los parámetros se adaptan para ofrecer un mayor comportamiento exploratorio.

1.1. Mutación

La mutación para cada individuo en una generación se genera con una estrategia rand-to-guiding (exploratoria) o current-to-guiding (convergente), la cual se selecciona por sorteo con una umbral de probabilidad ξ_1. A estas estrategias se les añade una segunda componente extra que dependiente de un cierto $\vec{d}_{r2,G}$, el cual pretende añadir un elemento de aleatoriedad a la mutación. La expresión para la mutación queda de la siguiente manera:

$$\vec{v}_{i,G} = \begin{cases} \vec{x}_{\text{rand},G} + F_1(\vec{x}_{g,G} - \vec{x}_{\text{rand},G}) + F_2(\vec{x}_{r1,G} - \vec{d}_{r2,G}) & \text{si rand} < \xi_1, \\ \vec{x}_{\text{cur},G} + F_1(\vec{x}_{g,G} - \vec{x}_{\text{cur},G}) + F_2(\vec{x}_{r1,G} - \vec{d}_{r2,G}) & \text{caso contrario,} \end{cases} \tag{1}$$

donde las variables son:

- ξ_1: Umbral de probabilidad para la selección de estrategias. (Constante entre 0 y 1)
- rand: Variable aleatoria con probabilidad uniforme entre 0 y 1.
- G: Generación del individuo.
- $\vec{v}_{i,G}$: Mutación para el individuo i.
- $\vec{x}_{\text{cur},G}$: Individuo i para el que queremos crear la mutación. (Current individual)
- $\vec{x}_{\text{rand},G}$, $\vec{x}_{\text{r1},G}$: Individuos aleatorios. ($\vec{x}_{\text{rand},G} \neq \vec{x}_{\text{cur},G}$, $\vec{x}_{\text{r1},G} \neq \vec{x}_{\text{cur},G}$)
- $\vec{x}_{g,G}$: Individuo seleccionado entre la élite de la generación. (Guiding individual)
- F_1: Peso de la estrategia principal. (Parámetro entre 0 y 1)
- F_2: Peso de la componente puramente aleatoria. (Constante entre 0 y 1)
- $\vec{d}_{r2,G}$ Vector aleatorio dentro del espacio de búsqueda.

Hemos visto que la expresión anterior para la mutación requiere tres componentes que hay que determinar en cada generación. Estos son el guiding individual $\vec{x}_{g,G}$, el parámetro F_2 y el vector aleatorio $\vec{d}_{r2,G}$.

Guiding Individual $\vec{x}_{g,G}$: Se elegirá de manera equiprobable entre los elementos de Pop_s o bien Pop_g, donde la elección de un conjunto u otro se decidirá de si un parámetro llamado Success Ratio de la generación, SR_G sobrepasa un cierto umbral constante ξ_3 entre 0 y 1. El Success Ratio se define como $SR_0 = 1$ y para el resto $SR_G = NS_G/NP$, es el número de hijos que suceden a los padres en la generación entre el número total de la población. La expresión entonces es la siguiente:

$$\vec{x}_{g,G} \text{ seleccionado aleatoriamente de } = \begin{cases} \text{Pop}_s & \text{si } SR_{G-1} < \xi_3, \\ \text{Pop}_g & \text{caso contrario,} \end{cases} \tag{2}$$

donde Pop_s es el top $floor((1-t^3)\times 100)\,\%$ de la población, con $t = G/G_{\text{máx}}$ y $G_{\text{máx}}$ número máximo de generaciones; y Pop_g es el top $10\,\%$ de la población. Como ξ_3 acostumbra a ser pequeño (0.05), si ha habido poco reemplazo se seleccionará el guiding individual de Pop_s, el cual empieza siendo toda la población y a medida que avanzan las generaciones es cada vez más elitista. Luego si al principio el condicionamiento ha sido muy bueno y la tasa de reemplazos en las primeras generaciones es baja, la selección del guiding individual se hará sobre un conjunto muy grande de la población aumentando las posibilidades de seleccionar un "peor" individuo mejorando la exploración.

Parámetro F_1: Este parámetro corresponde al peso que se le da al vector de cambio $\vec{x}_{g,G} - \vec{x}_{\text{cur},G}$, por lo que su valor dependerá de dos escenarios.

- Si el fitness del guiding individual es mejor que el del current individual, entonces $\vec{x}_{g,G} - \vec{x}_{\text{cur},G}$ es una dirección prometedora. El peso F_1 será proporcional a cuan mejor es el fitness del guiding individual respecto su generación:

$$F_1 = \frac{1}{2}\left(1 + \frac{f_{\text{máx}} - f_g}{f_{\text{máx}} - f_{min}}\right), \tag{3}$$

  donde $f_{\text{máx}}$, f_{min}, f_g son el fitness del máximo, el mínimo y el guiding individual de la generación respectivamente.

- Si el fitness del guiding individual es peor que el del current individual, entonces $\vec{x}_{g,G} - \vec{x}_{\text{cur},G}$ no es una dirección prometedora. Tomaremos la dirección opuesta como dirección prometedora haciendo que F_1 sea:

$$F_1 = \begin{cases} -0.05 & \text{si } -randn > -0.5, \\ -0.95 & \text{si } -randn < -0.95, \end{cases} \tag{4}$$

  donde $randn$ es una variable aleatoria normal con $\mu = 0.5$ y $\sigma = 0.2$.

Vector aleatorio $\vec{d}_{r2,G}$: La estrategia para crear este vector aleatorio es elegir un individuo aleatorio de la población y sortear para cada una de sus componentes con un cierto umbral ξ_2, si se sustituye la componente por un elemento aleatorio dentro del espacio de búsqueda. Esto es:

$$\vec{d}^{\,j}_{r2,G} = \begin{cases} L^j + rand(0,1)\times(U^j - L^j) & \text{si } rand(0,1) < \xi_2, \\ \vec{x}_{\text{r2},G} & \text{caso contrario,} \end{cases} \tag{5}$$

- $\xi_2 = (1 + 9\times 10^{5(t-1)})/100$: Umbral de probabilidad para la selección aleatoria.
- rand(0,1): Muestreo aleatorio de una uniforme entre 0 y 1.
- $\vec{x}_{\text{r2},G}$: Individuo aleatorio. ($\vec{x}_{\text{r2},G} \neq \vec{x}_{\text{r1},G}$)
- L^j: Cota inferior de la componente j-ésima en el espacio de búsqueda.
- U^j: Cota superior de la componente j-ésima en el espacio de búsqueda.

Vemos que por la definición de ξ_2, y recordando la definición de $t = G/G_{\text{máx}}$ dada en el guiding individual, está comprendida entre 0 y 1. De hecho este parámetro es monótono decreciente, y en las primeras generaciones es cercano a 1 y en la última igual a 0. Eso hace que capacidad de obtener una componente puramente aleatoria y diferente a un individuo de la población muy alta al principio (explora) y muy baja al final (converge) que sería un comportamiento deseable.

1.2. Crossover

Mediante el crossover generamos nuevos individuos que comparen características entre la población inicial y difieren en alguna mutación. Lo primero que hacemos es definir la probabilidad CR de que un individuo de la siguiente generación herede características del individuo mutado.

$$\mathrm{CR} = 1 - \frac{R_g}{\mathrm{NP}}, \tag{6}$$

con

- NP: Número total de individuos.
- R_g: Ranking del guiding individual. En el conjunto ordenado de individuos por fitness, el puesto que ocupa el guiding individual respecto al mejor.

Si nos fijamos en la ec.(1) vemos que las mutaciones tienen tendencia a acercarse al guiding individual. El crossover explota esto otorgando una CR alta cuando el fitness del guiding individual es "mejor (R_g pequeño).

Una vez calculada la probabilidad de mutación CR generamos un individuo mutado o *trial* a partir de cada par individuo original - mutación. Hacemos que el *trial* $u_{i,G}$ herede componentes de la mutación en función del CR:

$$u^j_{i,G} = \begin{cases} v^j_{i,G} & \text{si rand} \leq \mathrm{CR} \text{ ó randn}(i) = j, \\ x^j_{i,G} & \text{otro caso,} \end{cases} \tag{7}$$

donde el superíndice j denota la j-ésima componente, y

- randn(i): Entero aleatorio entre 1 y D, siendo D el número de componentes de x, v, u.

La condición randn(i) = j asegura que al menos una componente (la componente número rand(i)) sea mutada. Nótese que para problemas de una única dimensión esta condición siempre se cumple y los *trials* son idénticos a la población mutada.

1.3. Selección

En esta parte del algoritmo se elige al individuo de la generación $G + 1$, eligiendo para ello entre el individuo de la generación G y el trial generado a partir de él.

Para efectuar esta selección se define previamente un fitness ponderado:

$$f_w(x_{i,G}) = \alpha \frac{f(x_{i,G}) - f_{\min}}{f_{\max} - f_{\min}} + (1 - \alpha) \frac{\mathrm{Dis}_{\max} - \mathrm{Dis}(x_{i,G}, x_{best,G})}{\mathrm{Dis}_{\max} + \mathrm{Dis}(x_{i,G}, x_{best,G})} \tag{8}$$

Con

- f_w: Función *fitness* ponderada.
- $f, f_{\min}, f_{\max}$: Función *fitness* del problema a resolver; el mínimo y el máximo valor de esta función para la población de la generación G.
- α: Variable aleatoria entre 0.8 y 1.
- Dis, $\mathrm{Dis}_{\max}$: Función que devuelve la distancia euclídea entre dos puntos; la máxima de estas distancias para la generación actual.
- $x_{\mathrm{best},G}$: Individuo con mejor *fitness* en la generación G.

La función fitness ponderada introduce una penalización por alejarse del individuo con mejor *fitness*.

El método de selección es esencialmente comparar ambos los *fitness* de el individuo original y el trial. Si el trial supera al original el individuo es sustituido por el trial.

$$\vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G} & \text{si } f(\vec{u}_{i,G}) < f(\vec{x}_{i,G}) \\ \vec{u}_{i,G} & \text{si } f(\vec{u}_{i,G}) \leq f(\vec{x}_{i,G}) \text{ y } x_{i,G} \neq x_{\mathrm{best},G} \\ \vec{x}_{i,G} & \text{otro caso} \end{cases} \tag{9}$$

Hacemos una excepción para el individuo con el mejor *fitness*, $x_{\mathrm{best},G}$, que siempre pasa a la siguiente generación.

2. Resumen de los Problemas a Resolver

Entre la selección de 10 problemas a resolver hemos tomado las funciones: Ackley, CrossInTray, Drop-Wave, Griewank, Bohachevsky3, Matyas, Zakharov, Rosenbrock, Michalewicz (2D) y Easom. La idea en esta selección es que haya una muestra representativa de diferentes comportamientos según el catálogo `http://www.sfu.ca/~ssurjano/optimization.html`.

En las siguientes secciones mostramos una tabla resumen de las funciones con información de su tipología y mínimos globales, así como la representación de las funciones y su topología comprender su comportamiento.

2.1. Tabla resumen de las Funciones

Nombre	Tipología	Mínimo Global
Ackley	Múltiples mínimos locales	$f(0,...,0) = 0$
CrossInTray	Múltiples mínimos locales	$f(\pm 1.3491, \pm 1.3491) = -2.06261$
Drop-Wave	Múltiples mínimos locales	$f(0,0) = -1$
Griewank	Múltiples mínimos locales	$f(0,...,0) = 0$
Bohachevsky3	Forma de Bol	$f(0,0) = 0$
Matyas	Forma de Lámina	$f(0,0) = 0$
Zakharov	Forma de Lámina	$f(0,...,0) = 0$
Rosenbrock	Forma de Valle	$f(1,...,1) = 0$
Michalewicz (2D)	Caidas Abruptas	$f(2.20, 1.57) = -1.8013$
Easom	Caidas Abruptas	$f(\pi, -\pi) = -1$

2.2. Topología de las Funciones

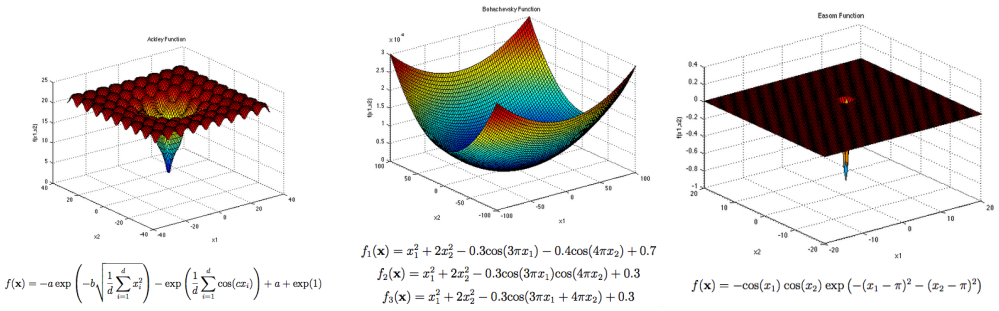


Figura 1: De izquierda a derecha las funciones de Ackley, Boachevsky3 y Easom

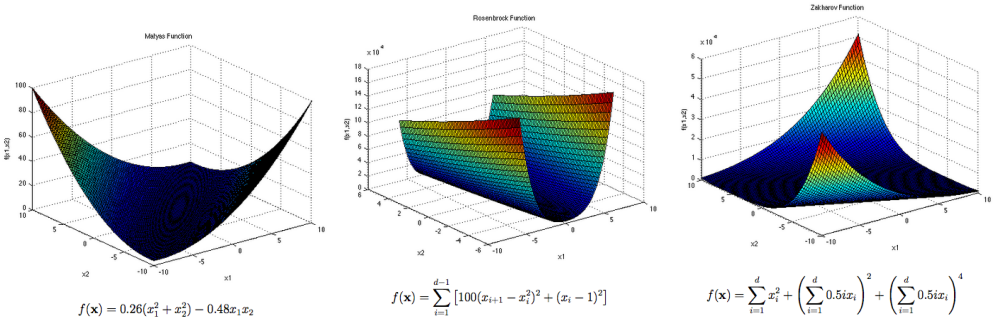
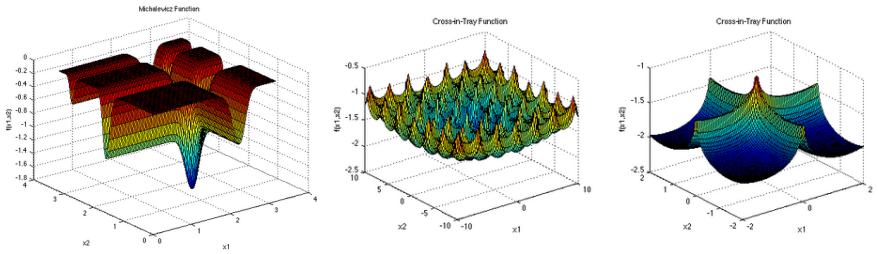


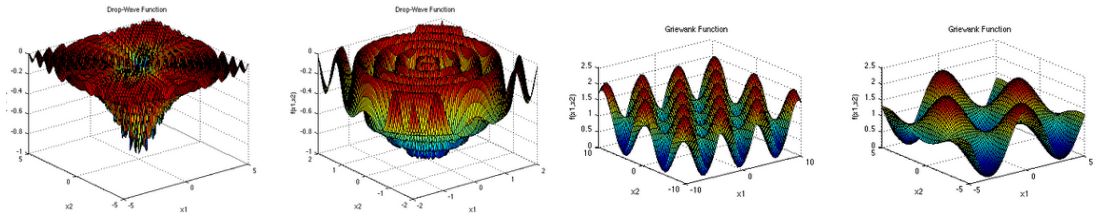
Figura 2: De izquierda a derecha las funciones de Matyas, Rosenbrock y Zakharov



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

$$f(\mathbf{x}) = -0.0001\left(\left|\sin(x_1)\sin(x_2)\exp\left(\left|100 - \frac{\sqrt{x_1^2+x_2^2}}{\pi}\right|\right)\right|+1\right)^{0.1}$$

Figura 3: De izquierda a derecha las funciones de Mixhalewicz y Crosstray



$$f(\mathbf{x}) = -\frac{1+\cos\left(12\sqrt{x_1^2+x_2^2}\right)}{0.5(x_1^2+x_2^2)+2}$$

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

Figura 4: De izquierda a derecha las funciones de Drop-Wave y Griewank

3. Desarrollo del Experimento

3.1. Estructura del Programa

El programa comienza con el `lanzador.R` que actúa como orquestrador. Desde aquí se llama al `inicializador.R`, que inicializa los datos de las funciones a resolver del directorio `funciones`; y se realiza un bucle que va llamando a la función en `evolutivo.R`, que contiene el algoritmo evolutivo; resolviendo todas las instancias de los problemas a resolver.

Dentro de `evolutivo.R` hay una función que va llamando a todas las subrrutinas que componen el algoritmo tal y como está descrito en el artículo [1] y la primera sección de este informe. Los scripts que contienen estas subrrutinas son:

1. `mutacion.R`: este script recibe como argumento una población de individuos y genera una lista de mutaciones, sin modificar la población original.

2. `crossover.R`: este script toma la población original y la lista de mutaciones de `mutacion.R` y las combina generando un nuevo individuo *trial*, en base a la variable *CR*, que determina la probabilidad de un individuo de mutar.

3. `seleccion.R`: este script selecciona los individuos eligiendo entre los individuos originales y los *trials*. Para ello tiene en cuenta el *fitness* de los individuos y el *fitness ponderado* (8) que se calcula desde `fitnessPonderado.R`. Devuelve una lista ordenada por *fitness* de la nueva población.

4. `fitnessPonderado.R`: Evalua la expresión (8) sobre la población y devuelve sus *fitness ponderado*.

3.2. Parámetros del Algoritmo

En este experimento existen tres parámetros que el usuario debe prefijar desde el principio y no se modifican por el algoritmo, que son $\xi_1 =$, ξ_3 y F_2. Aquí hemos utilizado los mismos que en el artículo [1], es decir, $\xi_1 = 0.05$, $\xi_3 = 0.05$ y $F_2 = 0.5$.

En el artículo se sugiere que el valor de ξ_1 se puede incrementar hasta 0.2 para problemas complicados. No obstante aquí hemos mantenido los parámetros del artículo, ya experimentando con pequeños ajustes no hemos obtenido cambios significativos en el comportamiento del algoritmo para los problemas seleccionados.

Además hemos para cada problema el algoritmo se ha repetido 25 veces y se ha tomado una población inicial aleatoria de 1000 individuos.

4. Resultados y Conclusiones

En la siguiente tabla se muestran los resultados obtenidos para los distintos problemas, incluyendo el tiempo de ejecución.

Problema	**Mejor F.Obj.**	**Error**	**Mediana F.Obj.**	**Tiempo 25 ejec.**
Ackley	0.0	0	0.020933	71.342
CrossInTray	-2.062612	0	-2.062612	60.682
Drop-Wave	-1.0	0	-0.936245	59.009
Griewank	0.0	0	$1e-0.6$	123.424
Bohachevsky3	0.0	0	2.11502	57.581
Matyas	0.0	0	0.000102	56.372
Zakharov	0.0	0	$5.9e-5$	70.644
Rosenbrock (1)	0.0	0	1.689961	70.063
Rosenbrock (2)	0.0	0	0.0	62.868
Michalewcz (2D)	-1.716269	0	-0.540872	73.296
Easom	-0.998611	0	-0.005659	62.741

Acabar Tabla

4.1. Mediana vs. media

En los resultados mostrados al comienzo de esta sección se utiliza la mediana en lugar de la media. Para explicar esta elección, veamos como ejemplo un extracto de las ejecuciones sobre la función Ackley:

Num. ejecución	Función objetivo
15	0.000000
16	0.000141
17	16.46224
18	0.095270
19	0.000000

De forma intuitiva descartamos la "explosión" de la ejecución número 17, ya que es dos órdenes de magnitud mayor que el resto. Este efecto es aún más notable cuando, de las 25 ejecuciones, solo hay dos con un fitness mayor a 0.1. Estas explosiones son comunes al optimizar las funciones benchmark, que están diseñadas con el objetivo de ofuscar algoritmos de optimización.

Las explosiones hacen de la media una herramienta inútil. Siguiendo con las 5 ejecuciones del ejemplo, al hacer la media obtenemos que es ≈ 5.3, esencialmente una quinta parte de la ejecución 17. La mediana, por otra parte, tiene un valor de 0.000141, mucho más acorde a los resultados. En resumen: la mediana nos permite filtrar las explosiones mientras que en la media contamina todas las medidas.

4.2. Conclusiones

Si miramos los valores de la mediana veremos que para la mayoría de los problemas está bastante cercana al mínimo de su problema, con varias excepciones como el Bohachevsky o el Michalewez.

Si atendemos a los valores de la mejor función objetivo, veremos que todos los valores están a menos de una décima de distancia y la mayoría tienen el valor de la función en el mínimo (precisión de 6 decimales). Esto es un muy buen resultado, ya que nos indica que el algoritmo es capaz de llegar al mínimo de la

función. Es muy probable que, adaptando el algoritmo (p.e. modificando las ξ_i), el algoritmo encontrase el mínimo absoluto, mejorando consecuentemente la mediana.

Referencias

[1] M. Tian, X. Gao, and C. Dai, "Differential evolution with improved individual-based parameter setting and selection strategy," *Applied Soft Computing*, vol. 56, pp. 286–297, 2017.

5. Anexo. Código del programa

5.1. lanzador.R

```
# Este fichero se encarga de lanzar la ejecucion de los algoritmos
# y de guardar la informacion correspondiente.

# Carga de funciones.
source("evolutivo.R")
source("inicializador.R")
source("mutacion.R")
source("fitnessPonderado.R")
source("crossover.R")
source("seleccion.R")


# Lista de problemas incorporados (se podria ampliar).
prob <- c("Ackley", "Bukin6", "CrossInTray", "DropWave" ,"Eggholder" ,"GramacyLee12","Griewank","HolderTable",
          "Langermann","Levy" ,"Levy13", "Rastrigin","Schaffer2","Schaffer4","Schwefel","Shubert","Bohachevsky1","Bohachevsky2",
          "Bohachevsky3","Perm0db","Rothyp","Sphere","Sphere2","Sumpow","Sumsqu","Trid","Booth","Matyas","Mccormick","Powersum",
          "Zakharov","Camel3","Camel6","DixonPrice","Rosenbrock1","Rosenbrock2","DeJong5","Easom","Michalewicz","Beale",
          "Branin1","Branin2","Branin3","Colville","Forrester","GoldsteinPrice","Hartmann3D","Hartmann4D","Hartmann6D","Permdb",
          "Powell","Shekel","StyblinskiTang")

# Conjunto de semillas para inicializar el generador de numeros aleatorios.
semilla <- c( 352668,  628434,  492990,  528643,  477348,  855426,  570702,  957864, 1019818,  849154,  982709,  991540,  776820,  302260,  509101,
             1104259,  778274,  937185, 1102620,  514412, 1026644,  288393,  848117, 1153861,  473884,  578922,  465690, 1092241,  538478,  764238,
             1005899,  434185,  681939, 1065173, 1177813,  178308, 1123423, 1159720,  280842,  563670,  694785,  918578,  854191, 1179079,  845770,
             1154990,  474168,  675549,  417239, 1007395)

#Parametros de ejecucion, que deberan fijarse de acuerdo a vuestro experimento.

numRepeticiones <- 25    # cuantas veces se resuelve cada problema.
sizePopulation <- 100   # tamano de la poblacion ##NP en el paper.
numIteraciones <- 1000  # numero de iteraciones del algoritmo ## en el paper G_max.

# Indicamos que problemas se van a resolver.
# Todos problemas <- 1:53
# Los diez primeros problemas <- 1:10
# Los diez ultimos problemas <- 44:53
# Unos cuantos problemas <- c(1,3,7,9,12,22,34)

problemas<-c(46:53) # Resolvemos problemas desde el 1 al 10

# Nombre del fichero en el que se almacenan los resultados.
# Si no existe lo crea y copia los resultados,
# si existe anade los nuevos resultados al contenido previo.

ficheroRes <- "resultados.txt"


# Loop principal, para cada problema ejecuta el algoritmo de optimizacion
# numRepeticiones veces, obteniendo una coleccion de valores de tiempo de
# ejecucion y valor alcanzado con el que elaborar las estadisticas.
#
for (i in problemas){
  for (j in 1:numRepeticiones){
    evolutivo(semilla[j],prob[i],sizePopulation,numIteraciones,ficheroRes)
  }
}
```

../lanzador.R

5.2. inicializador.R

```
# Esta funcion se encarga de establecer la informacion de los problemas
# y cargar el script correspondiente a la funcion
#
# Los argumentos que recibe son:
#
# problema:                    el nombre del problema
#
# deveuelve una lista con la informacion del problema, la descripcion la podeis
# consultar en :
# http://www.sfu.ca/ssurjano/stybtang.html
#
# Ademas en la parte final de la funcion se realiza un source() del fichero que contiene
# la definicien de la funcion a optimizar (maximizar), dicha fichero contiene siempre
# una funcion llamada evaluadora(), por lo que en genetico() no es necesario
# dar el nombre explicito de la funcion a resolver, basta con llamar a evaluadora()

inicializador <- function(problema) {
  info <- NULL
  switch(problema,
         Ackley =       {info$n <- 10;  info$l <- rep(-32.768,info$n); info$u <- -info$l
             ;},
         Bukin6 =       {info$n <-  2;  info$l <- c(-15,-3);           info$u <- c(-5,3)
             ;},
         CrossInTray = {info$n <-  2;  info$l <- c(-10,-10);          info$u <- -info$l
             ;},
         DropWave =     {info$n <-  2;  info$l <- rep(-5.12,info$n);   info$u <- -info$l
             ;},
         Eggholder =    {info$n <-  2;  info$l <- rep(-512,info$n);    info$u <- -info$l
             ;},
         GramacyLee12= {info$n <-  1;  info$l <- rep(0.5,info$n);     info$u <- rep(2.5,
             info$n);},
         Griewank     = {info$n <-  5;  info$l <- rep(-600,info$n);    info$u <- -info$l
             ;},
         HolderTable=  {info$n <-  2;  info$l <- rep(-10,info$n);     info$u <- -info$l
             ;},
         Langermann =  {info$n <-  2;  info$l <- rep(0,info$n);       info$u <- rep(10,
             info$n);},
         Levy =         {info$n <-  5;  info$l <- rep(-10,info$n);     info$u <- -info$l
             ;},
         Levy13 =       {info$n <-  2;  info$l <- rep(-10,info$n);     info$u <- -info$l
             ;},
         Rastrigin =    {info$n <- 10;  info$l <- rep(-5.12,info$n);   info$u <- -info$l
             ;},
         Schaffer2 =    {info$n <-  2;  info$l <- rep(-100,info$n);    info$u <- -info$l
             ;},
         Schaffer4 =    {info$n <-  2;  info$l <- rep(-100,info$n);    info$u <- -info$l
             ;},
         Schwefel =     {info$n <-  5;  info$l <- rep(-500,info$n);    info$u <- -info$l
             ;},
         Shubert =      {info$n <-  2;  info$l <- rep(-5.12,info$n);   info$u <- -info$l
             ;},
         Bohachevsky1=  {info$n <-  2;  info$l <- rep(-100,info$n);   info$u <- -info$l
             ;},
         Bohachevsky2=  {info$n <-  2;  info$l <- rep(-100,info$n);   info$u <- -info$l
             ;},
         Bohachevsky3=  {info$n <-  2;  info$l <- rep(-100,info$n);   info$u <- -info$l
             ;},
         Perm0db=       {info$n <-  5;  info$l <- rep(-info$n,info$n);info$u <- -info$l
             ;},
         Rothyp=        {info$n <-  5;  info$l <- rep(-65.536,info$n);info$u <- -info$l
             ;},
         Sphere=        {info$n <-  5;  info$l <- rep(-5.12,info$n);  info$u <- -info$l
             ;},
         Sphere2=       {info$n <-  6;  info$l <- rep(0,info$n);      info$u <- rep(1,
             info$n);},
         Sumpow=        {info$n <-  10; info$l <- rep(-1,info$n);     info$u <- -info$l
             ;},
         Sumsqu=        {info$n <-  10; info$l <- rep(-10,info$n);    info$u <- -info$l
             ;},
         Trid=          {info$n <-  6;  info$l <- rep(-info$n^2,info$n);   info$u <- -
             info$l;},
```

```
46          Booth=          {info$n <-  2;  info$l <- rep(-10,info$n);    info$u <- -info$l
                ;},
47          Matyas=         {info$n <-  2;  info$l <- rep(-10,info$n);    info$u <- -info$l
                ;},
48          Mccormick=      {info$n <-  2;  info$l <- c(-1.5,-3);         info$u <- c(4,4);},
49          Powersum=       {info$n <-  4;  info$l <- rep(0,info$n);      info$u <- rep(info$
                n,info$n);},
50          Zakharov =      {info$n <-  2;  info$l <- rep(-5,info$n);     info$u <-rep(10,
                info$n);},
51          Camel3 =        {info$n <-  2;  info$l <- rep(-5,info$n);     info$u <- -info$l
                ;},
52          Camel6 =        {info$n <-  2;  info$l <- c(-3,-2);           info$u <- -info$l
                ;},
53          DixonPrice =    {info$n <-  5;  info$l <- rep(-10,info$n);    info$u <- -info$l
                ;},
54          Rosenbrock1 =   {info$n <-  5; info$l <- rep(-5,info$n);      info$u <- rep(10,
                info$n);},
55          Rosenbrock2 =   {info$n <-   4; info$l <- rep(0,info$n);      info$u <- rep(1,
                info$n);},
56          DeJong5 =       {info$n <-   2; info$l <- rep(-65.536,info$n);info$u <- -info$l
                ;},
57          Easom =         {info$n <-   2; info$l <- rep(-100,info$n);   info$u <- -info$l
                ;},
58          Michalewicz =   {info$n <-   10; info$l <- rep(0,info$n);      info$u <- rep(pi,
                info$n);},
59          Beale =         {info$n <-    2; info$l <- rep(-4.5,info$n);   info$u <- rep(pi,
                info$n);},
60          Branin1 =       {info$n <-    2; info$l <- c(-5,0);            info$u <- c(10,15)
                ;},
61          Branin2 =       {info$n <-    2; info$l <- c(-5,0);            info$u <- c(10,15)
                ;},
62          Branin3 =       {info$n <-    2; info$l <- c(-5,0);            info$u <- c(10,15)
                ;},
63          Colville =      {info$n <-    4; info$l <- rep(-10,info$n);    info$u <- -info$l
                ;},
64          Forrester =     {info$n <-    1; info$l <- rep(0,info$n);      info$u <- rep(1,
                info$n);},
65          GoldsteinPrice={info$n <-    2; info$l <- rep(-2,info$n);      info$u <- -info$l
                ;},
66          Hartmann3D =    {info$n <-    3; info$l <- rep(0,info$n);      info$u <-  rep(1,
                info$n);},
67          Hartmann4D =    {info$n <-    4; info$l <- rep(0,info$n);      info$u <-  rep(1,
                info$n);},
68          Hartmann6D =    {info$n <-    6; info$l <- rep(0,info$n);      info$u <-  rep(1,
                info$n);},
69          Permdb=         {info$n <-    2; info$l <- rep(-info$n,info$n);info$u <- -info$l
                ;},
70          Powell=         {info$n <-    5; info$l <- rep(-4,info$n);     info$u <-  rep(5,
                info$n)},
71          Shekel=         {info$n <-    4; info$l <- rep(0,info$n);      info$u <- rep(10,
                info$n);},
72          StyblinskiTang={info$n <-   15; info$l <- rep(-5,info$n);      info$u <- -info$l
                ;}
73    )
74
75    source(paste("funciones/",problema,".R",sep=""))
76    return(info)
77  }
```

../inicializador.R

5.3. evolutivo.R

```
1  # Orquestra todas las componentes del algoritmo genetico: mutacion, crossover
2  # y seleccion de padres, ademas de la evaluacion de la fun obj y la ordenacion
3  # de la poblacion en base a fun obj para el ranking.
4  #
5  # El input es:
6  #     - seedE         Semilla para generar mismos parametros aleatorios por ejecucion.
7  #     - problem       Problema a resolver.
8  #     - sizepop       Tamano maximo de la poblacion.
9  #     - GMAX          Numero maximo de iteraciones.
```

```
#     - fileE          Fichero para escribir la salida de datos.
#
# El output es:
#     - NA
#
# Dependencias y Observaciones:
#     - Esta pensado para un problema de minimizacion.

evolutivo <- function(seedE, problem, sizepop, GMAX, fileE) {

  ## Inicializa el problema y la semilla.
  set.seed(seedE)                 # Inicializa la seed del rand para obtener los mismos
                                  # resultados en cada ejecucion.
  info = inicializador(problem)   # Inicializa los datos del problema.
  indlen = info$n                 # Longitud de los individuos. (Numero de variables).
  l = info$l;                     # Cotas inferiores de las variables
  u = info$u;                     # Cotas superiores de las variables
  rm(info)                        # Borrado de la lista info.

  ## Informacion general de la ejecucion, se muestra en la consola.
  cat(sprintf("Procesando problema: %s en dimension %d\n", problem, indlen))
  cat(sprintf("Semilla: %d\n", seedE))
  cat(sprintf("Numero maximo iteraciones: %d\n", GMAX))

  ti = proc.time()  #Guardamos comienzo de ejecucion

  ## Inicializamos la poblacion y parametros iniciales para la primera iteracion.
  NSG = sizepop                           # Inicializamos el num de indiv que pasan a
                                          # la siguiente iteracion a toda la poblacion.

   # Inicializamos la poblacion con individuos aleatorios entre las cotas.
  population = matrix(NA, sizepop, indlen)
  for(i in 1:indlen){
    population[,i] = runif(sizepop, l[i], u[i])
  }

  ## Iteramos hasta GMAX generaciones.
  for(G in 1:GMAX) {

   # Calculamos los parametros t y alpha.
   t = G/(GMAX+1)
   alpha = rnorm(1, 0.9, 0.05)
   if(alpha > 1) alpha = 1
   if(alpha < 0.8) alpha = 0.8

   # Calculamos fitness.
   # La funcion evaluadora devuelve la funcion objetivo de un problema de maximo.
   # El algoritmo es de un problema de minimos por lo que cambiamos el signo.
   fitness = -evaluadora(population)

   ## Ordenamos poblacion y fitness por fitness decreciente, ya que queremos
   ## la poblacion con fun obj mas baja en los primeros puestos para el ranking.
   aux = sort(fitness, decreasing = FALSE, index.return = TRUE)
   fitness = aux$x
   orden_fitness = aux$ix
   population = population[orden_fitness,] # Ordena la poblacion por fitness.
   rm(orden_fitness) # Eliminamos ya que la poblacion ya esta ordenada por fitness.

   # Calculamos la fitness ponderado.
   fitness_w <- fitnessPonderado(population, fitness, alpha)

   # Generamos una mutacion de todos los genes de todos los individuos.
   mutationOut = mutacion(population, fitness, l, u, t, NSG)
   Rg = mutationOut[["Rg"]] # Indice del guiding individual en el ranking de poblacion
       ordenada,
   mutationMat = mutationOut[["mutationMat"]] # Matriz con las mutaciones para los padres
       .

   # Calculamos el CR de cada individuo
   CR = 1-Rg/sizepop # CR es la prob que los padres tengan mutaciones.
   if(CR < 0.05) CR = 0.05
   if(CR > 0.95) CR = 0.95

```

```
81    # Generamos los trials (padres mutados) a partir de la poblacion de padres y la matriz
          de mutacion.
82    trialIndividuals = crossover(population, mutationMat, CR)
83
84    # Calculamos el fitness y fitness ponderado de los trials (nuevos padres mutados).
85    ufitness = -evaluadora(trialIndividuals)
86    ufitness_w = fitnessPonderado(trialIndividuals, ufitness, alpha)
87
88    # Seleccionamos los mejores individuos entre el padre y su trial para la nueva
          generacion.
89    seleccionOut = seleccion(population, trialIndividuals, fitness, ufitness, fitness_w,
          ufitness_w)
90    population = seleccionOut[["population"]]
91    NSG = seleccionOut[["NSG"]]
92
93    ## CRITERIO DE PARADA TBD
94  }
95
96
97  finEjecucion = proc.time()-ti
98  fileConn <- file(fileE,open="at")                          #apertura del fichero de
        resultados
99  writeLines(sprintf("%s\t%9d\t%3d\t%4d\t%f\t%f ",           #escribiendo informacion basica
100                     problem,seedE,sizepop, GMAX,max(fitness),finEjecucion[3]), fileConn)
101 close(fileConn)                                           #cierre de fichero de
        resultados
102
103 cat(sprintf("Minimo Fun Obj = %f\n\n",min(fitness)))      #vfo alcanzado
104 cat(sprintf("X(%d)= %8.6f ",1:length(1),population[which.min(fitness),]))
105 cat("\n\n")
106
107 return()
108 }
```

../evolutivo.R

5.4. fitnessPonderado.R

```
1  # Esta funcion se encarga de calcular el fitnes ponderado.
2  #
3  # El input es:
4  #     - population          Lista con la poblacion sobre la que crear las mutaciones.
5  #     - fitness             Lista ordenada con el fitness de cada individuo.
6  #     - alpha               Peso de los terminos.
7  #
8  # El output es:
9  #     - term1+term2    Poblacion mutada a evaluar.
10 #
11 # Dependencias y Observaciones:
12 #     - La poblacion que se parsea tiene que estar ordenada segun la funcion objetivo.
13 #     - Esta pensado para un problema de minimizacion.
14
15 fitnessPonderado <- function(population, fitness, alpha) {
16
17     ## Inicializa las dimensiones.
18     indlen = length(population[1,])  # Longitud de los individuos.
19     sizepop = length(population[,1]) # Numero de individuos en la poblacion.
20
21     ## Calculo del primer termino.
22     fmin = fitness[1]                # Min de la fun obj en la poblacion.
23     fmax = fitness[sizepop]          # Max de la fun obj en la poblacion.
24     if((fmax-fmin) > .Machine$double.eps) {
25         term1 = alpha * (fitness-fmin)/(fmax-fmin)
26     } else {
27         term1 = rep(0,sizepop)
28     }
29
30     ## Calculo del segundo termino.
31     xbest = population[1,]
32     xbestMat = matrix(rep(xbest, each=sizepop), nrow=sizepop)
33     dist = rowSums((xbestMat-population)*(xbestMat-population))
34     Dmax = max(dist)
```

```
    if((Dmax + min(dist)) > .Machine$double.eps) {
        term2 = (1 - alpha) * (Dmax - dist)/(Dmax + dist)
    } else {
        term2 = rep(0,sizepop)
    }

    return(term1+term2)
}
```

../fitnessPonderado.R

5.5. mutacion.R

```
# Esta funcion se encarga de crear un vector de mutacion para la poblacion
# de padres. La estrategia seguida es una mutacion de evolucion diferencial
# usando un guiding individual de una elite, el cual se combina linealmente
# con un hibrido de current-to-guiding y random-to-guiding.
#
# El input es:
#     - population          Lista con la poblacion sobre la que crear las mutaciones.
#     - fitness             Lista ordenada con el fitness de cada individuo.
#     - l                   Lower bound del problema.
#     - u                   Upper bounde del problema.
#     - t                   Cociente G/Gmax, iteracion actual/numero max de iteraciones.
#     - NSG                 Numero de indiv que fueron mutados en la iteracion anterior.
#
# El output es:
#     - iguide              El indice del guiding individual.
#     - v                   Matriz con los vectores de mutacion sobre la poblacion.
#
# Dependencias y Observaciones:
#     - La poblacion que se parsea tiene que estar ordenada segun la funcion objetivo.
#     - Esta pensado para un problema de minimizacion.

mutacion <- function(population, fitness, l, u, t, NSG) {

   ## Inicializacion de parametros y variables auxiliares.
    v = list()                          # Lista de mutaciones.
    indlen = length(population[1,])     # Longitud de los individuos.
    sizepop = length(population[,1])    # Numero de individuos en la poblacion.
    dr2 = rep(0,indlen)                 # Vector para la mutacion puramente aleatorio.
    vi = rep(0,indlen)                  # Vector de mutacion final.
    fmin = fitness[1]                   # Min de la fun obj en la poblacion.
    fmax = fitness[sizepop]             # Max de la fun obj en la poblacion.
    MAX_TRY = 10                        # Num maximo de intentos de los sorteos.

    ## Parametros especiales del problema.
    Pt = 1-t**3         # Expresion del maximo indice donde elegir el guiding indiv.
    SR = NSG/sizepop    # Numero de mutaciones que mejoraron su predecesor / tama?o pob.
    xi1 = 0.05 # 0.2 para problemas complicados.
    xi2 = (1+9*10^(5*(t-1)))/100
    xi3 = 0.05

    ## Seleccion del guiding individual.
    if(SR < xi3) {
        top10EliteLim = round(sizepop*0.1)  # Indice del ultimo elemento de POPs.
        iguide = sample(1:top10EliteLim, 1)
    } else {
        topPtEliteLim = max(round(sizepop*Pt),1)   # Indice del ultimo elemento de POPg.
        iguide = sample(1:topPtEliteLim, 1)
    }
    xguide = population[iguide,]
    fguide = fitness[iguide]
    ## Construccion del vector de mutacion.
    for(index in 1:sizepop) {
       # Determinacion del current individual y sus parametros de combinacion.
        xcur = population[index,]
        fcur = fitness[index]
        if(fcur > fguide) {
            F1 = (1+((fmax-fguide)/(fmax-fmin)))/2
        }
        else {
```

```
            F1 = -rnorm(n = 1, mean = 0.5, sd = 0.2)
            if(F1>-0.05) {
                F1 = -0.05
            }
            if(F1 < -0.95) {
                F1 = -0.95
            }
        }
        F2 = 0.5

        # Sortea los indices de r1 y r2 (si tras MAX_TRY sorteos no encuentra
        # indices diferentes entre si de xcur, xr1, xr2 devuelve un warning, no crea
        # la mutacion y pasa al siguiente individuo).
        k = 0
        while(k <= MAX_TRY) {
            r2 = sample(1:sizepop, 1)
            if(r2 != index){
                break
            }
            k = k+1
        }
        k = 0
        while(k <= MAX_TRY) {
            r1 = sample(1:sizepop, 1)
            if(r1 != index && r1 != r2) {
                break
            }
            k = k+1
        }
        if(k > MAX_TRY) {
            warning(paste0("Tras ", MAX_TRY, " intentos de muestreo,",
                           "no se han conseguido indices diferente de xcur, xr1, xr2."))
            v[[index]] <- xcur
            next
        }
        xr2 = population[r2,]
        xr1 = population[r1,]

        # Calcula el vector dr2 que interviene en la componente
        # fija de aleatoriedad de la mutacion.
        for(comp in indlen) {
            if(runif(1, 0, 1) < xi2) {
                dr2 = l[comp]+runif(1, 0, 1)*(u[comp]-l[comp])
            } else {
                dr2 = xr2[comp]
            }
        }

        # Encuentra el vector v de mutacion y lo anade a la lista.
        if(runif(1, 0, 1) < xi1) {
            xrand = population[sample(1:sizepop, 1),]
            vi = xrand+F1*(xguide-xrand)+F2*(xr1-dr2)
        } else {
            vi = xcur+F1*(xguide-xcur)+F2*(xr1-dr2)
        }

        v[[index]] = vi
    }

    v_matrix = matrix(as.numeric(unlist(vi)), nrow = sizepop, ncol = indlen)
    return(list("Rg" = iguide, "mutationMat" = v_matrix))
}
```

../mutacion.R

5.6. crossover.R

```
# Esta funcion se encarga de combinar los indivuos originales con los vectores
# de mutacion para generar una poblacion de indivuos nuevos.
#
# El input es:
#     - population          Lista con la poblacion sobre la que crear las mutaciones.
```

```
#     - mutationMat         Lista ordenada con el fitness de cada individuo.
#     - CR                  Coeficiente de recombinacion. Probabilidad de mutacion de
#                           las componentes de los individuos de mutar,
#
# El output es:
#     - trialIndividuals    Poblacion mutada a evaluar.
#
# Dependencias y Observaciones:
#     - La poblacion que se parsea tiene que estar ordenada segun la funcion objetivo.
#     - Esta pensado para un problema de minimizacion.

crossover <- function(population, mutationMat, CR) {

    ## Inicializa las dimensiones.
    indlen = length(population[1,])   # Longitud de los individuos.
    sizepop = length(population[,1])  # Numero de individuos en la poblacion.

    ## Inicializamos a los trials, (la poblacion a mutar), con la poblacion de padres.
    trialIndividuals = population
      # En el caso 1D "todas" las componentes mutan.
    if(indlen == 1) {
      return(mutationMat)
    }
      # Para el caso general usamos (ec. 10) para mutar a los padres.
    else {
        # Posicion aleatorias para cada padre en las que se garantiza una mutacion.
      randni = sample(1:indlen, sizepop, replace = TRUE)
      for(index in 1:sizepop) {
          compToMutate = runif(indlen, 0, 1) <= CR # Mutacion con cierta probabilidad.
          compToMutate[randni[index]] = TRUE
          trialIndividuals[index,compToMutate] = mutationMat[index,compToMutate]
       }
    }

    return(trialIndividuals)
 }
```

../crossover.R

5.7. seleccion.R

```
# Esta funcion se encarga de seleccionar los mejores individuos entre los padres
# y los individuos mutados.
#
# El input es:
#     - population          Lista con la poblacion de padres.
#     - trialIndividuals    Lista con la poblacion mutada.
#     - fitness             Lista ordenada con el fitness de los padres.
#     - ufitness            Lista ordenada con el fitness de los individuos mutados.
#     - fitness_w           Lista ordenada con el fitness ponderado de los padres.
#     - ufitness_w          Lista ordenada con el fitness ponderado de los indviduos
    mutados.
#
# El output es:
#     - iguide              Poblacion seleccionada para la siguiente generacion.
#     - NSG                 Numero de inviduos nuevos en la nueva generacion.
#
# Dependencias y Observaciones:
#     - La poblacion que se parsea tiene que estar ordenada segun la funcion objetivo.
#     - Esta pensado para un problema de minimizacion.

seleccion <- function(population, trialIndividuals, fitness, ufitness, fitness_w,
    ufitness_w) {

    # Condiciones para sustituir a los padres y renovacion.
    cond1 = ufitness < fitness
    cond2 = ufitness_w < fitness_w
    cond2[1] = FALSE

    changeFilter = cond1 | cond2
    population[changeFilter,] = trialIndividuals[changeFilter,]

```

```
30      # Conteo de los individuos mutados que han sustituido a los padres.
31      NSG = length(which(changeFilter))
32
33      return(list("population" = population, "NSG" = NSG))
34  }
```

../seleccion.R