Yojhan Steven García Peña

Programación Evolutiva

Práctica 1 – Convocatoria extraordinaria

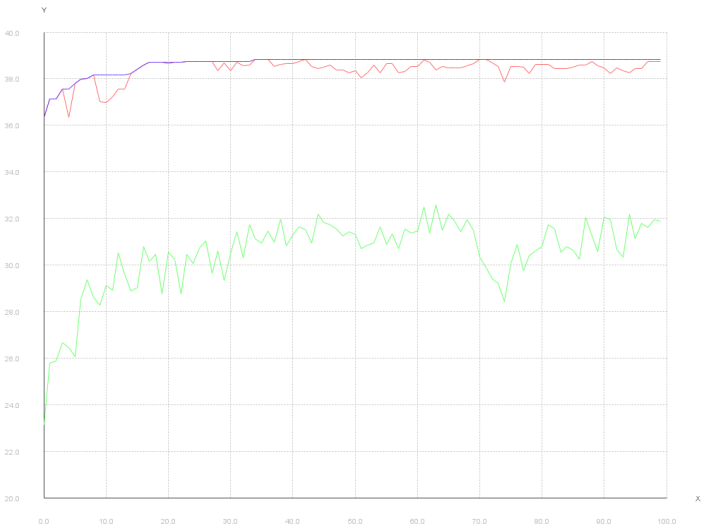
1. Gráficas

1.1 Calibración y prueba

Función:

$$f(x_1, x_2) = 21.5 + x_1.sen(4\pi x_1) + x_2.sen(20\pi x_2)$$

Resultados:



Resultado: 38.82493602974447

en x1=11.62622390428519, x2=5.723558162267839

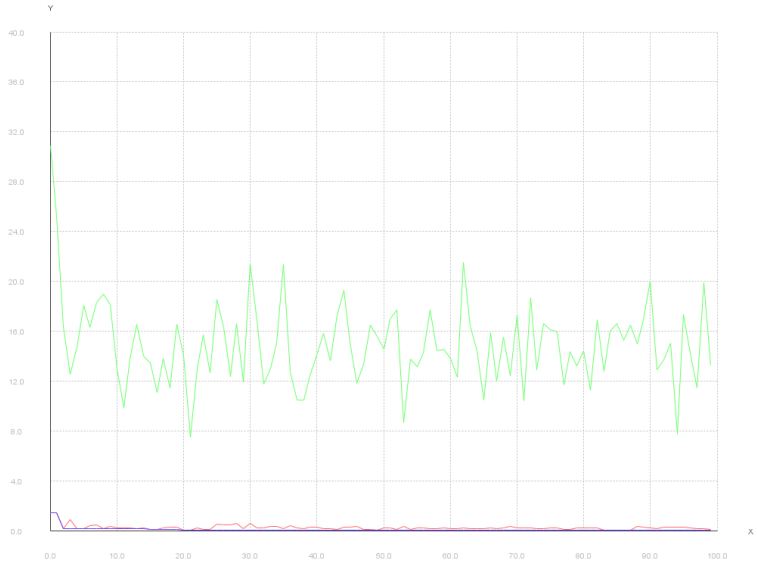
Error: 0.015

1.2 GrieWank

Función:

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

Resultados:



Resultado: 0.019023460867643505

en x1=2.98748301266005, x2=4.431728774765702

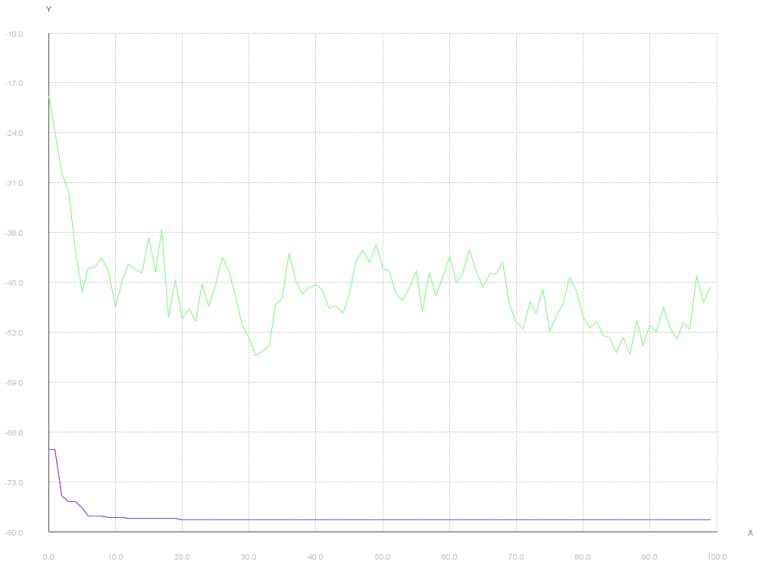
Error: 0.019

1.3 Styblinski-tang

Función:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{d}(x_i^4 - 16x_i^2 + 5x_i)$$

Resultados:



Resultado: -78.3323143475917

en x1=-2.9037968502014406, x2=-2.9025759980466366

Error: 0.00041 (Usando 2 dimensiones)

En este caso se ha usado elitismo, por ello la línea roja se solapa con la azul.

No he utilizado elitismo en el resto de las gráficas para que se pudieran ver los tres ejes por separado.

1.4Michalewicz

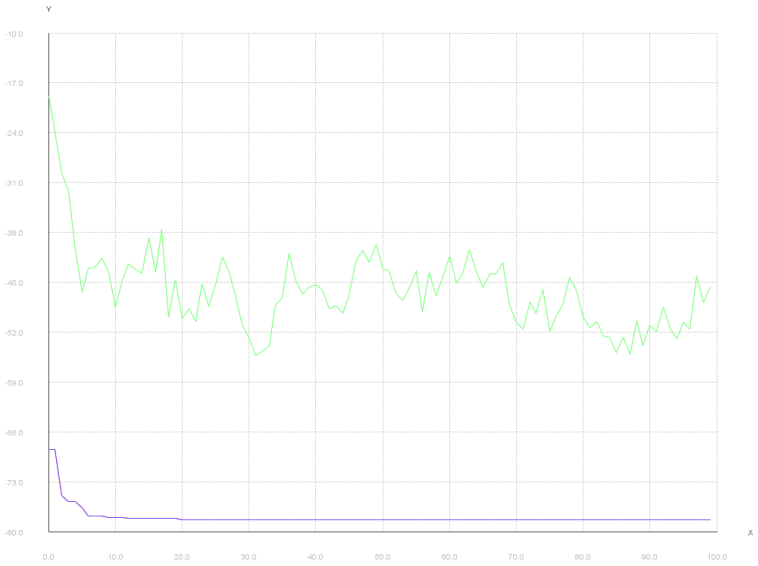
Función:

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

$$x_i \in [0, \pi] \quad m = 10$$

Resultados:

a) Genes binars

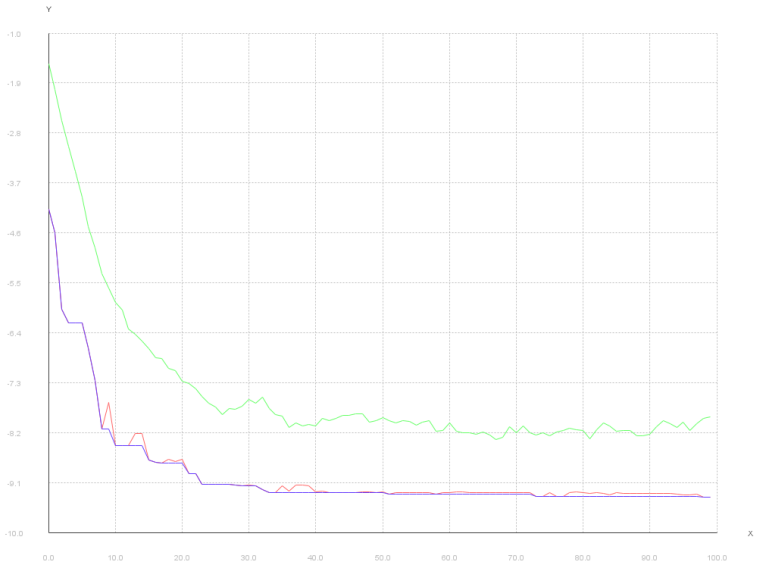


Resultado: -1.8006936047121758

en x1=2.2038725210331913, x2=1.5669595013752704

Error: 0.0013 (Usando 2 dimensiones)

b) Genes reales



Resultado: -9.364029555550115

en x1=2.2250596728559398, x2=1.5625262361810945, x3=1.295634276005909, x4=1.9131107118396034, x5=1.711630549045025, x6=1.5564301595873413, x7=1.4432334022423081, x8=1.3605837328727588, x9=1.292649577072403, x10=1.5753412770741992

Error: 0.30 (Usando 10 dimensiones)

2 Implementación

He hecho que las gráficas se solapen pues creo que eso es bastante útil a la hora de comparar distintos parámetros en la interfaz. Si se pulsa el botón 'Reiniciar' la gráfica y la casilla de resultado se borran para poder lanzar otra ejecución desde cero.

Otra cosa que quería comentar de la ejecución es que no lo he implementado como ponía en el anexo. En vez de tener un individuo abstracto al cual haya que implementar para cada algoritmo, tengo un individuo genérico el cual se puede parametrizar. De esta forma no hay que hacer una implementación nueva para cada función que se quiera hacer, y bastará con rellenar los valores desde fuera.

```
info = new InformacionAlgoritmo(2, true);

info.minimos[0] = -3.0;
info.maximos[0] = 12.1;

info.minimos[1] = 4.1;
info.maximos[1] = 5.8;

info.fitnessFunction = (input) -> {

    return 21.5 + input[0] * Math.sin(4 * Math.PI * input[0])
            + input[1] * Math.sin(20 * Math.PI * input[1]);
};
```

Por ejemplo, así es como se implementa la primera función 'Calibración y prueba'. Además, es compatible con bucles y código condicional para poder representar funciones más complejas.

```
info = new InformacionAlgoritmo(dim, false);

for (int i = 0; i < dim; i++) {
    info.minimos[i] = 0;
    info.maximos[i] = Math.PI;

}

info.fitnessFunction = (input) -> {


    double sum = 0;

    for(int i = 0; i < input.length; i++) {

        double x = input[i];
        sum += Math.sin(x) * Math.pow(Math.sin((i + 1) * x * x / Math.PI), 20);

    }

    return -sum;
};
```

En este caso, se usa un bucle para establecer todos los máximos y mínimos en función del parámetro dim.

Por último, otro cambio importante con respecto al ejemplo del anexo es que al no tener una clase abstracta Individuo, lo que tengo es una clase abstracta gen, de la cual tengo

dos implementaciones, una para un gen binario donde se representa mediante un array de booleanos, y otra para un gen real que se representa usando un 'double'.

3 Conclusiones y curiosidades

Por lo general creo haber conseguido resultados bastante buenos pues en la mayor parte de casos el error se encuentra en las centésimas.

El método de selección que a mi parecer mejores resultados da es el truncamiento, pues me parece que consigues mejores resultados además de que mejora la media considerablemente.

El cruce por otro lado, no he visto que haya mucha diferencia sobre utilizar el mono-punto o el uniforme, pero algo que me ha llamado la atención es que estos dos me parece que funcionan mejor que el aritmético y el BLX en la función real.

En cuanto a la mutación, solo he implementado la básica por lo que no he podido comparar distintos métodos, pero probando distintos porcentajes he visto que la mutación es una parte imprescindible para el desarrollo del algoritmo, pero no se debe abusar pues porcentajes muy altos generan mucho ruido al resultado final.

Y, por último, el elitismo es un cambio el cual es bastante sencillo pero mejora enormemente los resultados finales. No lo he usado en las gráficas de la memoria para que se pudieran ver los tres ejes ya que así las gráficas se ven más interesantes, pero la mejora que proporciona es considerable.

Uno de los problemas que hemos tenido y el motivo por el que suspendimos la primera entrega de la práctica fue el manejo de punteros en java. Esto nos ocurría en la selección a la hora de elegir a los mejores. Como puede ocurrir que se seleccione varias veces al mismo deberíamos duplicar al individuo. Esto no lo tuvimos en cuenta y por ello al final terminábamos como los 100 individuos apuntando al mismo objeto por lo que no había ningún tipo de diversidad y ahí nos quedábamos atascados.

4 Reparto de tareas

La práctica en la convocatoria ordinaria la hice junto a mi compañera Amparo, pero esta convocatoria extraordinaria la hemos hecho de forma individual.

He rehecho la práctica desde cero por mi cuenta, con la excepción de la interfaz, que cogí la que teníamos hecha de la entrega ordinaria, y algún que otro fragmento de código que he podido reutilizar (como los métodos de selección que hice yo).