# Framework para programación genética


Proyecto complementario al Doctorado del Maestro Alejandro Sosa
Supervisado por el Doctor Hugo Terashima
Patrocinado por CONACYT
Programado por el asistente Jesús Irais González Romero
ITESM Campus Monterrey
07/01/13

Índice

## Resumen

El framework descrito en este documento tiene como fin facilitar la definición de instancias programas evolutivos mediante algoritmos genéticos. El framework soporta 2 tipos de algoritmos genéticos: “Gene Expression Programming” y “Genetic Programming”.

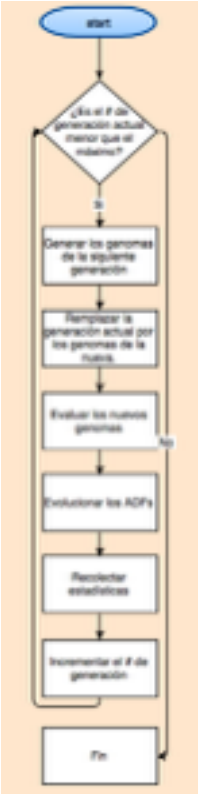
Características destacables del framework:

- Fácil traslación entre técnicas. (Gene Expression Programming <-> Genetic Programming.
- Soporte para Automatic Defined Functions (ADF) (Ferreira 2006).
- Utiliza un sistema de tipos (Strongly Typed Genetic Programming) (Montana 2002).
- Del sistema de tipos deriva la gramática; el framework genera de forma automática la gramática descrita por los tipos especificados y detecta errores antes de iniciar el algoritmo evolutivo.
- Posibilidad de crear grupos de pruebas que corren paralelamente y generan reportes gráficos al final.
- Fácil extensión de su funcionalidad.
- Escrito en Java utilizando únicamente librerías gratuitas.

## Introducción

El framework tiene como objetivo facilitar la investigación del uso de algoritmos evolutivos para la creación de hiperheurísticas, sin embargo, el framework se puede usar para cualquier programa evolutivo.

A continuación se presenta un diagrama representando a un algoritmo genético y el código que lo implementa:

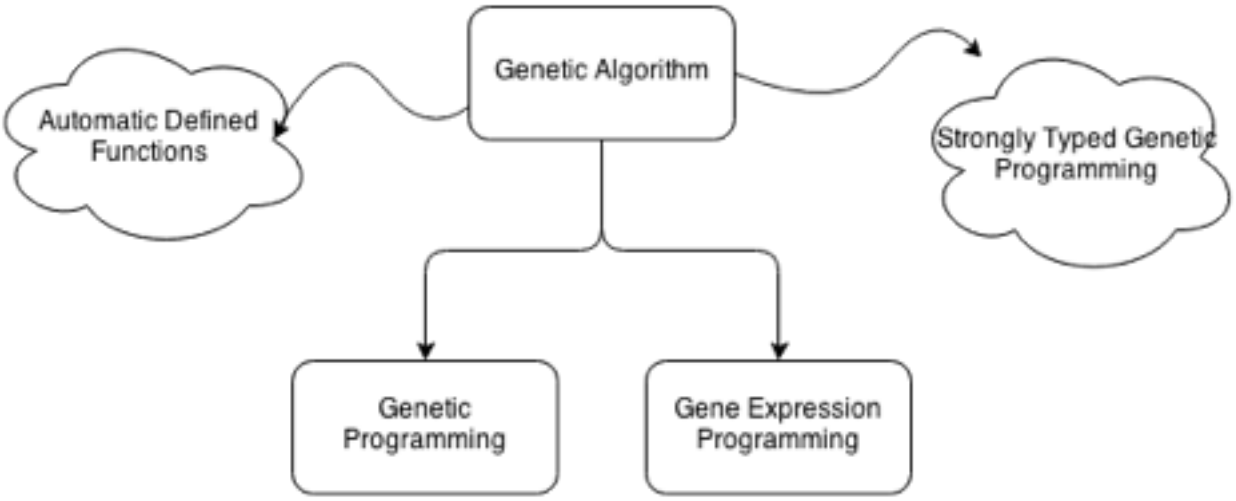



```
/**
 * Main loop of the algorithm.
 * @return The population generated in the last iteration.
 */
public Genome[] run(){
	while(currentGeneration < generationsToDo){
		Genome[] newGeneration = this.mainPopulation.generateNextGeneration();
		this.mainPopulation.replaceOldGenomes(newGeneration);
		this.evaluateNextGeneration(this.mainPopulation.completePopulation);
		this.mainPopulation.increaseGeneration();

		for(ADF adf : this.adfs){
			adf.evolve();
		}
		this.collectData();
		System.out.println("Current generation: " + currentGeneration);

		Genome bestGenomeInGeneration = this.mainPopulation.getBestGenome();
		if(bestGenomeInGeneration.compareTo(this.bestGenomeEver) == -1){
			this.bestGenomeEver = bestGenomeInGeneration;
		}

		currentGeneration ++;
	}
	return this.mainPopulation.getSortedFromBestToWorst();
}
```

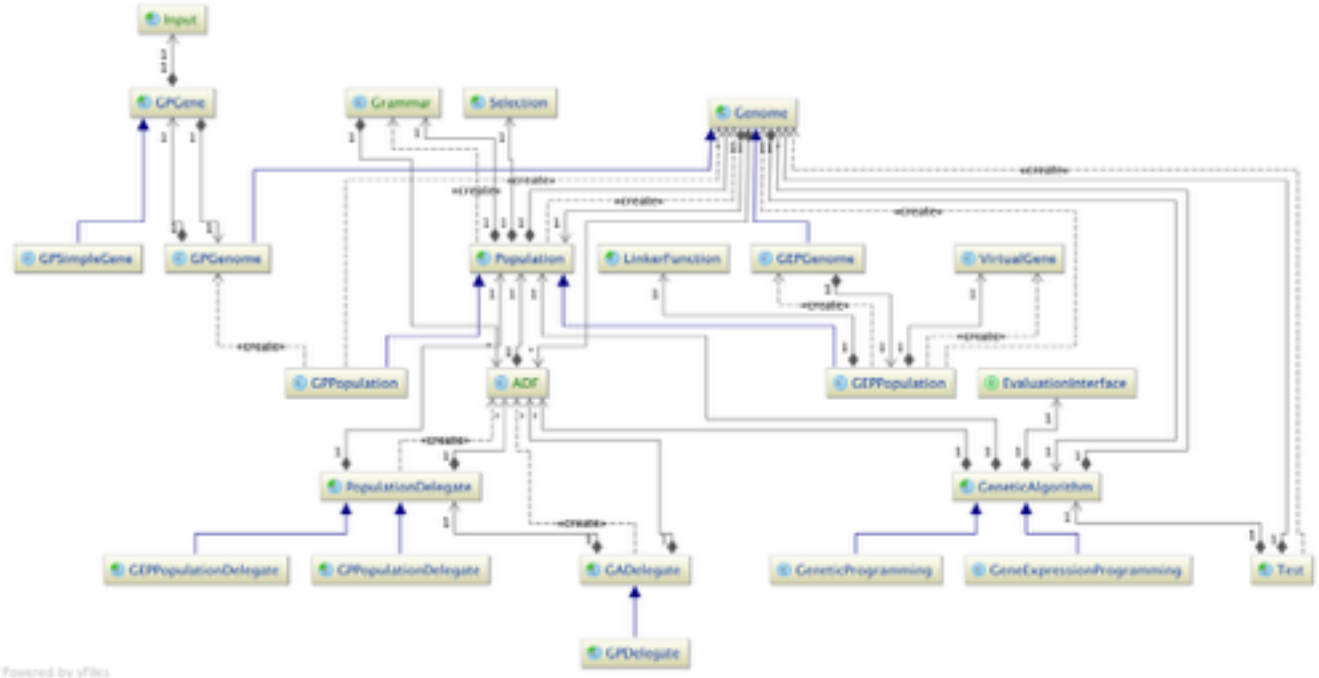
Como se puede ver en el diagrama, el algoritmo implementado es el algoritmo evolutivo simple con la adición de las ADFs; más adelante se hablará un poco sobre ellas.

## Conceptualización

El framework es una implementación de varios conceptos. He aquí un diagrama con todos los conceptos implementados por el framework:




Para una visión más completa, se muestra un diagrama de clases UML de los archivos .java más relevantes del framework:




## Breve demostración

### Requerimientos

El código del proyecto está ubicado en github: https://github.com/laygr/Framework-Algoritmos-Geneticos

El código depende de 2 librerías:

• jcommon, versión 1.0.17
• jchart, versión 1.0.14

Ambas librerías son gratuitas y se incluyen en el repositorio bajo el directorio llamado “lib” para conveniencia del usuario, sin embargo, ambas librerías también se encuentran en internet.

El repositorio incluye 3 pruebas, todas bajo el directorio llamado “tests”. Todas deben de poderse ejecutar sin ninguna dependencia adicional mas que las librerías incluidas en “lib”. Es importante no olvidar indicarle al IDE o al compilador que se estará usando dichas librerías para el proyecto.

Dado que el proyecto está desarrollado en Java, se requiere tener JDK 1.6 o superior instalado para ejecutar el código.

## Demostración

A continuación se mostrará un ejemplo sencillo para brindar una idea general del uso del del framework. Los comentarios del código incluidos debajo de las imágenes incluye información crucial sobre el uso del framework por lo que se recomienda leer detenidamente.

En este ejemplo se creará un programa evolutivo cuyo resultado se aproxime a cierta función. Para que el programa que se desea generar se comporte como la función que se intenta aproximar, este necesita poder recibir 1 parámetro, por lo que el framework se implementó de forma tal que los programas generados con el framework puedan recibir parámetros.

**Clase de configuración - tests/functionApproximation/FunctionAproximation1.java:**

```
10
11  public class FunctionAproximation1 extends Test {
12
13          @Override
14      public String getName() {
15          return "GP - Coevolutive";
16      }
17
18      @Override
19      public GADelegate configure() {
20          GPPopulationDelegate adfPopulationDelegate = new GPPopulationDelegate(new ADF[]{}, 100, false,
21              new TournamentSelection(75, 2), .05, .95,
22              new Class[]{Components.class}, Number.class,
23              ADF1Input.class, new HashMap(), 2);
24
25          adfPopulationDelegate.initiate();
26
27
28          ADFDelegate adfDelegate = new ADFDelegate(ADFDelegate.ADFType.COEVOLUTIVE, adfPopulationDelegate);
29          ADF adf = new ADF(0, adfDelegate);
30
31          GPPopulationDelegate mainPopulationDelegate =
32                  new GPPopulationDelegate(new ADF[]{adf}, 100, false, new TournamentSelection(99,2), .05, .95,
33                  new Class<?>[]{Components.class},
34              Number.class, X.class, new HashMap(), 5);
35
36          mainPopulationDelegate.initiate();
37
38          return new GADelegate(mainPopulationDelegate, 50, new ADF[]{adf}, new Evaluator());
39      }
40
41  }
```

**Línea 11:** La clase principal debe de extender la clase Test.

**Línea 19:** Configuración del test.
**Línea 20:** Se crea un delegado para la población que se usara como ADFs con las siguientes características:

- Del tipo definido por Genetic Programming.
- sin ADF's (caso base)
- 100 individuos
- cuya aptitud se intenta minimizar
- usando selección por torneo
- 5% de probabilidad de mutar
- 95% de probabilidad de cruce
- los componentes están definidos en la clase "Components.class"
- el resultado de evaluar un individuo debe de ser de tipo Number
- el tipo que recibe como parámetro esta definido por la clase ADF1Input.class
- sin probabilidades de generar un nodo if-then-else
- el árbol descrito por un individuo puede tener un nivel de hasta grado 2.

**Línea 25:** Se indica al delegado que ya debe de crear la población.
**Línea 28:** Se crea un delegado para el ADF. Nótese que en la **línea 20** se creó un delegado para una población; como una población no es lo mismo que una población de ADFs, un delegado especializado para ADFs debe de encargase de finalizar el proceso de crear lo que finalmente será el ADF.

El ADF tendrá una evolución coevolutiva y su población será provista por el delegado generado en la **línea 20**.
**Línea 31:** Finalmente se crea la población principal. Similar a la **línea 20**. Se hacen las siguientes observaciones:

La población principal usará el ADF recientemente creado.
El tipo de entrada está definido por la classe X.class.

**Línea 36:** Se inicia la población principal.

**Línea 38:** Finalmente, se termina la configuración creando el delegado para el algoritmo genético indicando:

El delegado con la población principal.
El número de corridas a utilizar.
Un arreglo con todos los ADFs que debe de mantener.
Una instancia de la clase encargada de evaluar la aptitud de los individuos de la población principal.

**Clase con los componentes - tests/functionApproximation/Components.java:**

```
4   public class Components {
5
6           public static Number sin(Number x){
7                   return new Double(Math.sin(x.doubleValue()));
8           }
9
10          public static Number multiplication(Number arg1, Number arg2){
11                  return new Double (arg1.doubleValue() * arg2.doubleValue());
12          }
13
14          public static Number addition(Number arg1, Number arg2){
15                  return new Double (arg1.doubleValue() + arg2.doubleValue());
16          }
17
18          public static Number substraction(Number arg1, Number arg2){
19                  return new Double (arg1.doubleValue() - arg2.doubleValue());
20          }
21
22          public static Number x(X x){
23                  return x.value;
24          }
25
26          public static Number one(){
27                  return 1;
28          }
29
30          public static Number getFirstParameterFromADF1(ADF1Input adfInput){
31                  return (Number) adfInput.getObjectAtIndex(0);
32          }
33  }
```

⋆ En este caso, todas las funciones son de tipo *Number*, pero no siempre es el caso.
⋆ **Línea 22:** Este método recibe como único parámetro un objeto de la clase X.java, indicado como el tipo de entrada para los individuos de la población.
⋆ **Línea 30:** Usualmente nos veremos en la necesidad de crear funciones como la definida en esta línea para extraer o calcular atributos de los parámetros de entrada.

La clase de define el tipo X que será usado como entrada para la población general - *X.java*

```
5   public class X extends Input {
6           public Number value;
7
8           public X(Number value){
9                   this.value = value;
10          }
11
12          public boolean equals(Input otherInput){
13                  X otherX = (X) otherInput;
14                  return this.value.doubleValue() == otherX.value.doubleValue();
15          }
16  }
```

Debe extender la clase Input.

**Clase con la función de evaluación - tests/functionApproximation/Evaluator.java:**

```
1   package tests.functionApproximation;
2
3   import ...
5
6   public class Evaluator implements EvaluationInterface {
7
8       @Override
9       public double evaluateGenome(Genome genome){
10          double correctValue;
11          double x;
12          double errorAcum = 0;
13          double genomeValue;
14          for(int i = 1; i < 10; i++){
15              x = Math.PI / i;
16
17              correctValue = Math.cos(2 * x);
18              Object result = genome.evaluate(new X(x));
19              genomeValue = ((Number) result).doubleValue();
20              errorAcum += Math.abs(correctValue - genomeValue);
21
22          }
23          return errorAcum;
24      }
25
26  }
```

Debe ser una implementación de la interface *EvaluationInterface*.
Como se especificó que la aptitud de la población debe de ser minimizada, un buen individuo es aquel que regrese el número más pequeño. Se puede ver en la **línea 20 y 23** como la variable que se regresará es la que mide el error.

## Automatic Defined Functions

Las ADFs (Funciones Definidas Automáticamente o Automatic Defined Functions) son subprogramas que evolucionan al mismo tiempo que evoluciona el programa principal. La teoría y justificación de dicha técnica está fuera del alcance de dicho reporte (consultar bibliografía), por lo que se limitará a describir que fue lo que se implementó y las posibles configuraciones.

Un algoritmo genético puede tener cero o mas ADFs evolucionando a la par del programa principal. Los ADFs pueden evolucionar conjuntamente de 3 formas: coevolutivamente, semicoevolutivamente o independientempente; para especificar que el tipo de evolución de un ADF, se debe de especificar en su PopulationDelegate como a continuación:

```java
package tests.functionApproximation;

import geneticAlgorithm.grammar.adf.ADFDelegate;

public class ADF1DelegateSemicoevolutive extends ADFDelegate {

    @Override
    public ADFType getADFType() { return ADFType.SEMICOEVOLUTIVE; }

}
```

Nótese como se especifica el tipo de evolución del ADF en la implementación del método abstracto *getADFType()*; básicamente es lo único que ocurre en la implementación de un *ADFDelegate*.

Y podemos ver a continuación como se ocupa dicha clase en la configuración del algoritmo genético:

```java
public class FunctionAproximation1 extends Test {

    @Override
    public GADelegate configure() {
        GPPopulationDelegate adf1PopulationDelegate = new ADF1PopulationDelegate();
        adf1PopulationDelegate.initiate();
        ADFDelegate adf1Delegate = new ADF1DelegateCoevolutive();
        adf1Delegate.setPopulationDelegate(adf1PopulationDelegate);

        ADF adf1 = new ADF(0, adf1Delegate);

        GPPopulationDelegate mainPopulationDelegate = new MainPopulationDelegate1_2_4();
        GPDelegate gpDelegate = new FunctionAproximationGPDelegate1_2();
        mainPopulationDelegate.addADF(adf1);
        mainPopulationDelegate.initiate();
        gpDelegate.setMainPopulationDelegate(mainPopulationDelegate);
        ADF[]adfs = {adf1};
        gpDelegate.setADFs(adfs);

        return gpDelegate;
    }
```

De la **Línea 13 a la 16** se configura un ADF. Se crea por separado el *ADFDelegate* y el *PopulationDelegate* de dicha población (la población de dicho ADF) y es hasta la **Línea 16** en la que se asigna el *PopulationDelegate* al *ADFDelegate*.
Teniendo ya construido y preparado el *ADFDelegate,* se puede crear el ADF, como en la **Línea 18**.
Finalmente, se asigna al *PopulationDelegate* del algoritmo principal cuales adfs debe usar, como en la **línea 26**.

Al igual que los programas principales, los ADFs igual tienen un *PupulationDelegate* y como cualquier *PopulationDelegate,* requiere que se especifique su input. Sin embargo el input de un *PopulationDelegate* es distinto al input de un programa principal
La clase que define el tipo de entrada para el ADF - ADF1Input.java

```
5   public class ADF1Input extends ADFInput {
6
7       public ADF1Input() {
8       }
9
10      @Override
11      public Class<?>[] getParameterTypes() {
12          Class<?>[] parameterTypes = {Number.class};
13          return parameterTypes;
14      }
15
16      @Override
17      public void init() {
18      }
19  }
```

Se puede apreciar la diferencia entre la entrada para la población general y la entrada para un ADF. Un ADF puede recibir una lista de parámetros, por lo que su implementación consiste en especificar el tipo de la lista y el número de elementos que debe contener dicha lista. Además, la clase otorga un método de inicialización de conveniencia, *Init,* para preprocesar los elementos de dicha lista.

## Traducción de Genetic Programming a Gene Expression Programming

La clase principal que se describió al principio usaba poblaciones del tipo GeneticProgramming (Genetic Programming). A continuación se muestra una adaptación para que use poblaciones del tipo GEP (Gene Expression Programming).

**Clase de configuración - tests/functionApproximation/FunctionAproximation4.java:**

```java
public class FunctionAproximation4 extends Test {

    @Override
    public String getName() {
        return "GEP-Simple";
    }

    @Override
    public GADelegate configure() {
        GEPPopulationDelegate adfPopulationDelegate =
                new GEPPopulationDelegate(new ADF[]{}, 500,
                false, new TournamentSelection(499, 2),
                new Class[]{Components.class}, Number.class,
                ADF1Input.class, new HashMap(), 4, new FunctionAproximationLinker(),
                .05, .1, .1, .1, .1, .1, .1);

        adfPopulationDelegate.initiate();


        ADFDelegate adfDelegate = new ADFDelegate(ADFDelegate.ADFType.SIMPLE,
                                                  adfPopulationDelegate);
        ADF adf = new ADF(0, adfDelegate);

        GPPopulationDelegate mainPopulationDelegate =
                new GPPopulationDelegate(new ADF[]{adf}, 100, false,
                new TournamentSelection(75, 2), .05, .95,
                new Class<?>[]{Components.class},
                Number.class, X.class, new HashMap(), 5);

        mainPopulationDelegate.initiate();

        return new GADelegate(mainPopulationDelegate, 1000, new ADF[]{adf}, new Evaluator());
    }
}
```

Se puede apreciar que es muy similar a la clase **FunctionAproximation1.java.** En este caso, la población principal sigue siendo de tipo Genetic Programming, sin embargo, podemos ver que la población que usara el ADF es de tipo Genetic Expression Programming en la **línea 21**. La principal diferencia entre declarar una población de tipo GEP o GP radica en que en GEP se tiene que especificar una linker function y parámetros para:

- probabilidad de mutar un gen.
- probabilidad de que ocurra IS Transposition.
- probabilidad de que ocurra RIS Transposition.
- probabilidad de que ocurra Gene Transposition.
- probabilidad de que ocurra un One Point Recombination.
- probabilidad de que ocurra un Two Point Recombination.
- probabilidad de que ocurra recombinación de genes.

En este caso, se especificó adicionalmente que el tipo de evolución que tendrá la población del ADF es de tipo simple.

**Por último, la linker function - tests/functionApproximation/ FunctionAproximationLinker.java:**

```java
public class FunctionAproximationLinker extends LinkerFunction{

    @Override
    public Class<?> getReturnType() {
        return Number.class;
    }

    @Override
    public Object evaluate(Object[] parameters) {
        double acum = 0;

        for(Object param : parameters){
            acum += ((Number) param).doubleValue();
        }

        return new Double(acum);
    }

    @Override
    public Class<?> getORFReturnType() {
        return Number.class;
    }

    @Override
    public int getNumberOfORFs() {
        return 1;
    }
}
```

La función de la función linker es darle significado a los diferentes ORFs.

### Virtual Gene

El framework utiliza Virtual Genes para la representación eficiente de los individuos en el caso de Gene Expression Programming. Todas las ideas de esta representación fueron tomadas de la implementación del Dr. Manuel Valenzuela de su framework para algoritmos evolutivos: vgGA. Desconozco el verdadero origen de dichas ideas.

A muy grosso modo, las ideas detrás del Virtual Gene son las de representar cualquier individuo en un arreglo de bits y así poderlo manipular mediante operaciones eficientes de bits.

## Definición de componentes y el sistema de tipos y gramática

Todos los programas están compuestos por componentes atómicos. La manera que un usuario del framework define dichos componentes es de la manera siguiente:

1. Enlistar los componentes en la forma de funciones estáticas dentro de una o más clases de java.

Nótese que la función x recibe un objeto llamado x de tipo X; ese tipo X está definido como la entrada del programa a generar en el archivo *X.java*. El framework se encarga de pasar el input al programa generada en la fase de evaluación

2. Indicar en el *PopulationDelegate* cuales son las clases que contienen los componentes como se muestra a continuación:

```java
package tests.functionApproximation;

import ...

public class MainPopulationDelegate3 extends GPPopulationDelegate {
    ArrayList<ADF> adfs = new ArrayList<ADF>();

    @Override
    public Class<?>[] getClassesForComponents() {
        Class<?>[] components = {Components.class};
        return components;
    }

    @Override
    public int getPopulationSize() {
        return 200;
    }
```

Dados los componentes, el framework se encargará de solamente crear, ya sea mediante mutación o cruce, individuos semánticamente válidos apoyándose del sistema de tipos de Java. Para ello, se crea automáticamente una gramática a partir de los componentes, es decir, el framework deduce la gramática que describe a todos los individuos validos a partir de los componentes.

La motivación de tener un sistema de tipos es reducir la posibilidad de que el algoritmo genere un programa inválido semánticamente, como lo sería sumar un entero con una cadena de caracteres.

El algoritmo evolutivo aprovecha el sistema de tipos para solo formar programas que sean semánticamente válidos.

Previo a iniciar el algoritmo evolutivo, se hace un chequeo para buscar que la gramática no contenga tipos inalcanzables. Un tipo inalcanzable es un tipo que se encuentra en la gramática pero que no hay forma de producirlo.
Supongamos que solo contamos con las siguientes funciones:

TipoX funcionF(){...}
TipoY funcionG(TipoY param1){...}

Se puede ver que no hay forma de producir un objeto de tipo TipoY, excepto si TipoY es el tipo indicado para la entrada. Si TipoY no es el tipo indicado como entrada, el framework indicará que hay un error en la gramática antes de iniciar la ejecución y terminará.

## Función IF-Then-Else

El sistema de tipos obliga a tener una función If-Then-Else para cada tipo. Para poder controlar la probabilidad que tiene cada tipo de If-Then-Else de ser añadido a un individuo, se permite especificar por separado la probabilidad de cada uno.
Regresando al ejemplo FunctionAproximation1.java recordemos como se veía:

```
@Override
public GADelegate configure() {
    GPPopulationDelegate adfPopulationDelegate = new GPPopulationDelegate(new ADF[]{}, 100, false,
        new TournamentSelection(75, 2), .05, .95,
        new Class[]{Components.class}, Number.class,
        ADF1Input.class, new HashMap(), 2);
```

Como se está pasando un HashMap vacío donde deberían de ir las probabilidades, el framework asume que se desea que cada *If-Then-Else* tenga la misma probabilidad que cualquier otro componente. Si se pasara un *null* en su lugar, no habría ningún *If-Then-Else*. Por último, si se quiere ser más especifico, se puede pasar un HashMap que contenga tuplas en la forma: (X, Y), donde X es un objeto de tipo Double que representa la probabilidad que una función If-Then-Else de tipo Y tiene de ser elegida. Ejemplo:

```
19  public GADelegate configure() {
20      HashMap<Class, Double>probabilitiesIfThenElse = new HashMap();
21      probabilitiesIfThenElse.put(Double.class, .5);
22      probabilitiesIfThenElse.put(String.class, .1);
23
24      GPPopulationDelegate adfPopulationDelegate = new GPPopulationDelegate(new ADF[]{}, 100, false,
25          new TournamentSelection(75, 2), .05, .95,
27          new Class[]{SomeComponents.class}, Number.class,
            ADF1Input.class, probabilitiesIfThenElse, 2);
```

En la **línea 20** se crea el HashMap. En la **21 y 22** se especifican las probabilidades para las Clases Double y String y en la **línea 27** se pasa el HashMap.

## Grupo de pruebas y generación de reportes

GroupOfTestsFunctionApproximation.java

```
1  import test.GroupOfTests;
2  import tests.functionApproximation.*;
3
4  public class GroupOfTestsFunctionApproximation {
5      public static void main(String[] args){
6          Class<?>[] testClasses =
7                  {
8                      FunctionAproximation1.class, FunctionAproximation2.class,
9                      FunctionAproximation3.class, FunctionAproximation4.class
10                 };
11         GroupOfTests groupOfTests = new GroupOfTests("Function Aproximation", testClasses, 30);
12         groupOfTests.graphAverageAverage();
13         groupOfTests.graphAverageBest();
14     }
15 }
```

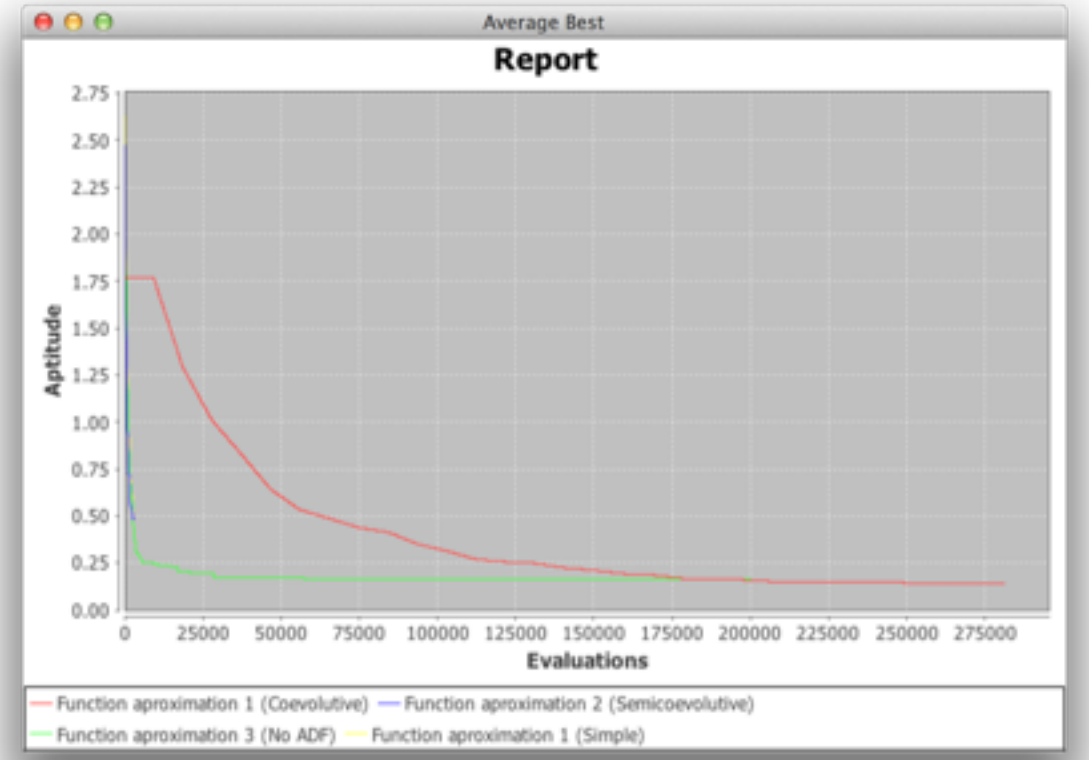
En vez de ejecutarse pruebas por separado, se puede preparar un grupo de tests para dejarlos corriendo; el framework paralelizará su ejecución para minimizar el tiempo que tomen en correr. Un grupo de tests se crea de la siguiente manera:
**Línea 6:** Se crea un arreglo de clases, cada clase debe de ser un Test (heredar de la clase Test)

**Línea 11:** Se crea un GroupOfTests pasando como argumento el nombre del grupo de tests, el arreglo de clases y el numero de corridas que cada test debe correr. Esta misma línea invoca la ejecución del grupo de tests.

Adicionalmente se pueden generar gráficas del promedio del promedio del total de corridas y del promedio de los mejores del total de corridas:

**Línea 12:** Se dibuja una gráfica *AveragesAverage* (Averages' average) y es el promedio de todas las corridas del promedio de la evaluación de los individuos por cada clase Test.
**Línea 13:** Se dibuja una gráfica *BestsAverage* (Bests' Average) y es el promedio del mejor individuo de todas las corridas por cada clase Test.

Así se ve la gráfica *BestsAverage* generada por ese archivo:




Se puede ver que incluye con distintos colores los 4 grupos de pruebas con su respectivo nombre. El eje x es el número de evaluaciones de la función objetivo y el eje y el promedio de la aptitud de los mejores individuos de todas las corridas.

## Conclusiones

Se concluyó el framework satisfactoriamente aunque quedaron oportunidades de mejora; en particular, sería muy interesante poder generar programas con ciclos, pero el problema es más complejo de lo que se podría pensar y se sale del alcance de este proyecto. La implementación de ciclos es particularmente difícil pues si no se idea un mecanismo para evitar ciclos infinitos, predominarían entre los programas generados programas que nunca terminan y esperar cierto tiempo como timeout sería particularmente indeseable para un algoritmo evolutivo, pues son muchos los programas que se tienen que evaluar y esperar a que ocurran tantos timeouts sería un gran overhead. Quizás sería mejor sustituir los ciclos por procesamientos de listas y sería muy interesante explorar una implementación de programación genética enfocado a programas funcionales.

## Bibliografía

Montana, D. (2002). "Strongly Typed Genetic Programming."

Ferreira, C. (2006). "AUTOMATICALLY DEFINED FUNCTIONS IN GENE EXPRESSION PROGRAMMING." Genetic Systems Programming: Theory and Experiences, Studies in Computational Intelligence 13: 21-56.