Capítulo 4

La Distribución Software lil-gp

Existen distintos paquetes software con los que se pueden desarrollar aplicaciones de programación genética. Para la realización de este proyecto, la herramienta elegida fue el sistema lil-gp. En este capítulo se intenta explicar brevemente su funcionamiento básico y sus características más importantes.

---

1. INTRODUCCIÓN

Una decisión importante a la hora de realizar este proyecto fin de carrera fue la de elegir un software de programación genética con el que desarrollar la aplicación. Pues bien, después de un proceso de búsqueda en la red y comparación de diferentes opciones, se optó por el paquete lil-gp.

Lil-gp es un sistema escrito en lenguaje C para desarrollar aplicaciones de programación genética, basado en el trabajo pionero de Koza [3] en la Universidad de Stanford. Los nodos que componen los árboles, son en realidad punteros a funciones C, de manera que la evaluación de un árbol se hace completamente con código compilado. Ésto disminuye el tiempo de computación y permite trabajar con problemas de gran envergadura (poblaciones de mayor tamaño y más generaciones). Además, es un sistema

portable en el sentido de que puede trabajar bajo diferentes sistemas operativos.

1.1. Características

Los principales puntos a destacar de lil-gp son:

- Facilidad de uso: las distintas opciones se especifican mediante ficheros de parámetros con un formato flexible. Los argumentos se manejan en la línea de comandos.
- Escrito en lenguaje C para facilitar su portabilidad a una amplia gama de plataformas como Unix/Linux, DOS o Mac.
- 7 métodos de selección denominados: fitness proportionate, greedy overselection, inverse fitness, variable size tournament, random, best y worst.
- 3 operadores genéticos: cruce, reproducción y mutación.
- Posibilidad de limitar el tamaño de los árboles, tanto en número de nodos como en profundidad.
- Soporte para generar constantes numéricas de forma aleatoria.
- Variedad muy extensa de información de salida, dada en forma de ficheros; incluyendo datos estadísticos de la ejecución.

1.2. El núcleo

En el núcleo de lil-gp, los árboles se representan de forma muy compacta (mediante punteros a funciones) y ello permite que su evaluación sea extremadamente rápida. Por otro lado, son almacenados como expresiones en pre-orden, con símbolos insertados que indican la existencia de constantes numéricas generadas aleatoriamente. Por ejemplo, (+ x1 2.3), que equivaldría a escribir $x1 + 2,3$.

La compactación de la que antes se hablaba, permite también trabajar con grandes poblaciones de individuos almacenados en memoria real, formando bloques contiguos. Ésto último, minimiza los efectos de la paginación de memoria.

También se dispone de un generador de números aleatorios que proporciona resultados muy consistentes en diferentes sistemas operativos.

1.3. Especificación de Problemas

El sistema lil-gp está estructurado de tal modo que se puede crear una nueva aplicación escribiendo una cantidad de código fuente mínima. El usuario debe proporcionar una función C por cada miembro integrante del conjunto de funciones primitivas (nodos no terminales) y del conjunto de nodos terminales. Además debe definir una función para evaluar el nivel de aptitud de los árboles.

Se dispone también de varias llamadas al sistema, que permiten al usuario realizar tareas de inicialización y reserva de memoria, diseño del formato de los datos de salida o establecer un criterio de terminación de la ejecución del programa.

1.4. Autor

Lil-gp fue desarrollado por Douglas Zongker bajo la dirección del Dr. Bill Punch, ambos pertenecientes al GARAGE (Genetic Algorithms Research and Applications Group) en la Universidad del Estado de Michigan.

Toda la información relacionada con lil-gp (manuales de usuario, nuevas versiones, etc.) se puede encontrar en internet, concretamente en la URL <http://isl.cps.msu.edu/GA/software/lil-gp> [6].

2. CONCEPTOS BÁSICOS

El término *Programación Genética* está asociado al trabajo de John Koza [3]. Un programa genético (o PG) no es más que un sistema de aprendizaje adaptativo, basado en muchos de los principios de los algoritmos genéticos (o AG) descritos por John Holland en su obra *Adaptation in Natural and Artificial Systems* [1] y por David Goldberg en *Genetic Algorithms* [2].

Existen varias similitudes entre un AG y un PG. Ambos mantienen una multitud de soluciones independientes, representadas como individuos en una población. Las poblaciones evolucionan a través de varias generaciones, en las cuáles los miembros de la misma son copiados, eliminados o modificados y luego insertados en la siguiente generación. Ambos usan un conjunto de operadores genéticos para modificar los individuos de la población. Cada uno de ellos hace uso de distintos métodos de selección de individuos, basados normalmente en la medida de la aptitud.

Sin embargo, hay diferencias importantes entre ambos como son:

1. La estructura de representación en un PG es un árbol, mientras que en un AG es una cadena de caracteres.
2. Los nodos de un árbol en un PG son normalmente funciones y terminales, haciendo que un árbol puede ser interpretado como un programa. Esto no es habitualmente cierto en aplicaciones de algoritmos genéticos.
3. Mientras que la longitud de una cadena en un AG es fija, los árboles en un PG son variables en tamaño y forma.

2.1. Definiendo una Aplicación

Para definir una aplicación o programa genético, uno debe proporcionar el conjunto de funciones y de nodos terminales con los cuáles se construirán los árboles que formen la población. Las funciones constituyen los nodos internos del árbol y representan funciones cuyos argumentos son los subárboles que cuelgan de dichos nodos. Los nodos terminales representarían funciones que no toman ningún argumento.

2.1.1. Condición de Cierre

Debido a que las estructuras que soportan a los individuos en un programa genético son programas, y debido también a que estos programas pueden ser construidos de formas no necesariamente previstas, las funciones elegidas deben ser capaces de tomar como argumentos cualquier valor que pudiera ser devuelto por un nodo terminal o que fuera resultado de la evaluación de otra función.

Un ejemplo clásico de esta situación sería el operador `división`. Este operador, que toma como argumentos dos números, debe ser diseñado de forma que cuando se encuentre con un divisor igual a cero, muestre un comportamiento por defecto que permita al programa continuar con su ejecución antes de que se muestre algún mensaje de error y se detenga. A esta propiedad de los programas genéticos se le denomina cierre.

2.1.2. Un Ejemplo

La distribución del paquete lil-gp incluye cinco programas ejemplo. Aquí se va mencionar sólo uno de ellos, que puede resultar intuitivo a la hora de desarrollar la solución de este proyecto. Se trata de una curva de regresión. Los individuos de la población serán funciones matemáticas que intentarán alcanzar una curva objetivo. Los casos de prueba serán una serie de pares (x,y) conocidos. Cada individuo es evaluado para un valor de x , y el valor devuelto se compara con el correspondiente valor de y . El nivel de aptitud será la diferencia entre estos dos valores, sumada para todos los casos posibles.

El conjunto de funciones definidas para este problema tiene 8 miembros que son: multiplicación, división protegida (devuelve un 1 cuando se divide entre 0), suma, resta, seno, coseno, exponenciación y logaritmo protegido (logaritmo de cero es 0 y en otro caso se devuelve el logaritmo del valor absoluto).

El conjunto de nodos terminales tiene dos tipos de miembros: los valores de entrada de x , y opcionalmente, constantes numéricas generadas de forma aleatoria o ERCs¹. Cuando se genera un terminal ERC, ya sea en la población inicial o más tarde por mutación, se asigna una constante numérica a dicho nodo y ya permanece inalterable en siguientes operaciones. En el ejemplo de la regresión se generaron valores en el rango $[-1, 1]$.

2.2. Tamaño de los Individuos

Los árboles sometidos al proceso evolutivo en un programa genético, pueden ir creciendo a lo largo de las generaciones hasta alcanzar tamaños muy grandes. Para evitar malgastar tiempo evaluando individuos demasiado grandes (que normalmente no representan una solución al problema), el usuario puede establecer límites en el número de nodos y/o la profundidad de los árboles.

John Koza en sus experimentos [3], establece un límite máximo en la profundidad de 17 (sin ninguna restricción en el número de nodos).

2.3. Aptitud

El grado de aptitud de cada individuo en la población se determina usando una función de evaluación. Basándose en los resultados de esta función, se toman decisiones relacionadas con la promoción o recombinación de un individuo en la siguiente generación. A la hora de definir un programa genético con lil-gp, se pueden utilizar distintos tipos de aptitud:

- **Aptitud bruta², f** Representa una medida directa, basada en el progreso que va teniendo la aplicación en sí misma a la hora de resolver el problema. A menudo, esta medida se hace comparando los distintos casos de prueba. Por ejemplo, en el caso de la curva de regresión, la aptitud bruta se mide como sigue. Se van a utilizar 20 casos de prueba, correspondientes a 20 pares (x, y) conocidos a priori. Para cada individuo, se evalúa en x y se resta el valor devuelto del y conocido. Si el valor absoluto de la diferencia es menor que un d (por ejemplo 0.01) entonces ese caso de prueba cuenta como un “hit”. La aptitud bruta se

---

¹ Del inglés Ephemeral Random Constants

² Raw fitness en inglés

define entonces como la suma de los valores absolutos de las diferencias, para los 20 casos de prueba.

- **Aptitud estándar³, f_s** Este tipo de aptitud simplemente traslada o mapea la aptitud bruta de manera que todos los valores de aptitud sean no negativos, siendo 0 el mejor caso de aptitud posible.
- **Aptitud ajustada⁴, f_a** Este valor se calcula a partir de la aptitud estándar y se define como sigue, para cada individuo i :

$$f_a(i) = \frac{1}{1 + f_s(i)}$$

f_a varía entre 0 y 1, siendo 1 el mejor caso posible. Este tipo de aptitud tiene la ventaja de que exagera o enfatiza pequeños incrementos en el grado de aptitud de los individuos. Por lo tanto, cuando se está próximo a una solución definitiva, se produce una realimentación adecuada que permite al programa genético perseguir la mejor solución.

2.4. Población Inicial

La población inicial de individuos en un programa genético se crea seleccionando aleatoriamente funciones y nodos terminales, para luego combinarlos de forma adecuada. Lil-gp dispone de tres métodos distintos para crear la población inicial:

- **Full** Este método genera sólo árboles “*completos*” (de ahí su denominación), es decir, árboles que tienen todos sus nodos terminales al mismo nivel de profundidad. Dicho de otro modo, la longitud del camino que hay que recorrer desde cualquier terminal hasta el nodo raíz, es siempre la misma.
- **Grow** El método *grow* toma cualquier nodo (función o terminal) como nodo raíz y luego, recursivamente se va llamando a sí mismo para generar subárboles hijo para aquellos nodos que así lo demanden. Este procedimiento está limitado por una profundidad máxima, es decir, si el árbol alcanza la profundidad máxima permitida, todos los nodos generados a partir de ese momento serán terminales cesando así su crecimiento.
- **Half-and-half** Este método simplemente es una combinación de los dos anteriores: el 50% de las veces se usa el método *full*, y el método *grow* en el 50% restante.

³ de standardized fitness en inglés

⁴ del inglés, adjusted fitness

Con todos los métodos mencionados, se puede especificar una “rampa” de valores de profundidad iniciales. Por ejemplo, si la rampa es 2-5, entonces el 25% de los árboles generados tendrán profundidad 2, el 25% con profundidad 3, y así.

Half-and-half con rampa de profundidad es, normalmente, el método más utilizado para la generar la población inicial ya que produce una amplia variedad de individuos tanto en forma como en tamaño. En otras palabras, genera diversidad genética.

Una vez que un individuo ha sido generado, lil-gp lo examina para comprobar que no supera los límites (si es que se han establecido) en el número de nodos y en la profundidad máxima. En caso afirmativo, dicho individuo no se inserta en la población. Además, lil-gp se asegura que no haya individuos duplicados en la población inicial.

2.5. Selección

Un método de selección no es más que una rutina utilizada para elegir a un individuo, de entre todos los que forman la población. Como ya se dijo en la introducción, lil-gp proporciona 7 métodos diferentes aunque los más utilizados son estos tres:

- **Fitness-Proportionate** este método selecciona un individuo basándose en la proporción que representa su *aptitud ajustada* frente a la *aptitud ajustada* total de la población. Esto es, cuando tenemos una población de n individuos, un individuo i será seleccionado con una probabilidad

$$p_i = \frac{f_a(i)}{\sum_{j=1}^n f_a(j)}$$

Ésto también se conoce como el método de la ruleta (ver figura 2.5). En él, cada individuo toma una porción de la ruleta, cuyo tamaño irá acorde a la fórmula de arriba (la rueda completa representará un 1). El individuo se selecciona haciendo girar la ruleta, de manera que aquellos con mayor aptitud f_a tendrán más posibilidades de ser elegidos (aunque todos tienen alguna oportunidad).

- **Overselection** Aunque el método anterior se considera bueno para la mayoría de las aplicaciones, a veces es necesario acelerar este proceso cuando se manejan grandes poblaciones (500 individuos o más, según Koza [3]). El procedimiento divide a la población en dos grupos distintos. El 80 % de las veces, el individuo resultará seleccionado del grupo I, y el 20% restante será elegido del grupo II. La partición se podría realizar de forma aleatoria pero Koza define una forma de hacerlo basada en el grado de aptitud. Para poblaciones de 1000 individuos, los primeros individuos (los de mayor *aptitud ajustada*) que

cuenten el 32% de la población total, van al primer grupo; el resto al segundo. Para poblaciones de 2000 miembros, el desdoblamiento se produce para el 16%; para poblaciones de 4000, el 8% y así.

- **Tournament** La selección por torneo (o tournament) se creó en los algoritmos genéticos para evitar que la presión que ejerce sobre la población el método *overselection*, pudiera conducir a la convergencia prematura. Como ya se ha visto en secciones anteriores, este método selecciona en primer lugar n individuos de forma totalmente aleatoria; luego, de entre esos n , se elige al mejor en lo que a aptitud se refiere. Se dice que n es el *tamaño del torneo* (o *size tournament*). Este es el método de selección utilizado en el desarrollo de este proyecto.

2.6. Operadores

Lil-gp permite operaciones de selección mixtas, es decir, se puede usar un método de selección para un determinado operador genético, por ejemplo el cruce, mientras que se utilizan otros distintos para el resto de operadores. Esto permite diseñar distintas estrategias a la hora de realizar experimentos.

Además, lil-gp permite cosas como usar un método de selección distinto para cada uno de los padres que entran en juego en la operación de cruce o jugar con la probabilidad de que el punto de cruce esté en nodos internos o más externos del árbol.

3. PARÁMETROS

Una gran cantidad de parámetros controlan el funcionamiento de lil-gp. Estos parámetros son introducidos a través de ficheros de datos y argumentos en la línea de comandos. Además, el usuario puede definir sus propios parámetros, específicos para una determinada tarea.

Se describirán en esta sección los parámetros más importantes para la ejecución de un programa desarrollado con el sistema lil-gp.

Antes de continuar, se muestran algunos convenios adoptados:

- La profundidad de los árboles se mide de manera que un árbol que tenga un solo nodo, se dice que tiene profundidad 0.
- Para parámetros de carácter binario, todas las cadenas de caracteres siguientes significarán “on”: `true`, `t`, `on`, `yes`, `y`, `1`. Del mismo modo, las siguientes expresiones se corresponderán con “off”: `false`, `f`, `off`, `no`, `n`, `0`.

3.1. Parámetros Generales

Estos parámetros gobiernan la ejecución del programa genético en su conjunto y se muestran en la tabla 4.1.

Nombre	Tipo	Descripción	Valor por defecto
<code>max_generations</code>	Entero	Máximo número de generaciones en la ejecución del programa	Ninguno
<code>pop_size</code>	Entero	Tamaño de la población	Ninguno
<code>random_seed</code>	Entero	Semilla para el generador de números aleatorios	1

Tabla 4.1 Parámetros de carácter general

3.2. Parámetros de Salida

Cada vez que se ejecuta un programa, lil-gp genera varios ficheros de salida con estadísticas sobre el tamaño de los árboles y su aptitud. El nombre base de estos ficheros (es el mismo para todos) va seguido de una extensión de tres caracteres. Éstas son:

- .sys** información general sobre la ejecución
- .gen** estadísticas sobre el tamaño y profundidad de los árboles
- .prg** estadísticas sobre el nivel de aptitud y los hits
- .bst** información sobre el(los) mejor(es) individuo(s) de la ejecución
- .his** historial del fichero .bst
- .stt** versión condensada de todas las estadísticas

El fichero **.bst** es reescrito después de cada generación y el resto de ficheros se guardan en disco al término de la ejecución del programa. Ésto permite al usuario consultar información de interés.

Cada mensaje impreso en un fichero de salida tiene una prioridad⁵ asociada, que variará entre 0 y 100. Un mensaje dado se mostrará en el fichero de salida sólo si su nivel de prioridad es menor o igual que el nivel especificado por un parámetro denominado **output.detail**.

Pues bien, la función de los parámetros de salida es controlar la escritura en los ficheros de salida, por parte del sistema lil-gp. En la tabla 4.2 se muestran los más significativos.

⁵ Expresada en forma de número entero.

Nombre	Tipo	Descripción	Valor por defecto
<code>Output.basename</code>	Cadena de caracteres	Nombre base para los ficheros de salida. Irán seguidos de una extensión de tres caracteres.	lilgp
<code>Output.detail</code>	Entero, 0-100	Nivel de detalle en los ficheros de salida. Con 100 se escribe todo y con 0 prácticamente nada.	50
<code>Output.stt_interval</code>	Entero positivo	Frecuencia (medida en número de generaciones) con la que se escribe en el fichero stt.	1
<code>Output.bestn</code>	Entero positivo	Indica cuantos individuos son impresos en el fichero bst. Por ejemplo, 5 significa que se imprimen los cinco mejores.	1
<code>Output.digits</code>	Entero	El número de cifras decimales con que se imprime la aptitud en los ficheros de salida.	4

Tabla 4.2 Parámetros de escritura en los ficheros de salida

3.3. Límites de Tamaño

Estos parámetros establecen los límites en el número de nodos y/o profundidad de los árboles que forman la población. Los más importantes se muestran en la tabla 4.3.

Nombre	Tipo	Descripción	Valor por defecto
<code>max_nodes</code>	Entero	Número máximo de nodos por individuo. Si no se especifica ninguno, no hay límites.	Ninguno
<code>max_depth</code>	Entero	Máxima profundidad permitida en un individuo. Si no se especifica ninguno, no hay límites.	Ninguno

Tabla 4.3 Parámetros limitadores del tamaño de los árboles

3.4. Inicialización

Los siguientes parámetros, reflejados en la tabla 4.4, controlan la generación aleatoria de la población inicial de individuos.

Nombre	Tipo	Descripción	Valor por defecto
<code>init.method</code>	half-and-half, grow o full	Método utilizado para la generación de la población inicial ⁶ .	half-and half
<code>init.depth</code>	Cadena de caracteres	Rampa de profundidad establecida para generación de la población inicial.	2-6
<code>Init.random_attempts</code>	Entero positivo	En la generación de la población inicial, se pueden crear individuos que violen los límites de tamaño o duplicados que el sistema rechaza sistemáticamente. Este parámetro establece el número de rechazos consecutivos que tolerará el programa antes de lanzar un mensaje de error y detener la ejecución.	100

Tabla 4.4 Parámetros de control de la generación de la población inicial

⁶ Véase sección 2.4. Población Inicial

3.5. Métodos de Selección

Un método de selección, como ya se vió, es un algoritmo para seleccionar a un individuo de la población. Se describen en este apartado los parámetros necesarios para definir dichos métodos.

Lil-gp dispone de 7 métodos de selección distintos; algunos de ellos tienen diferentes opciones que se pueden especificar mediante una lista de pares “opción = valor” separados por comas, que se escribe tras el nombre del método. Por ejemplo:

```
blabla.select = tournament, size = 7
```

que indica que se utilizará el método de torneo con grupos de 7 individuos.

Los métodos y sus distintas opciones se especifican con los siguientes nombres:

- **fitness** Método definido en la sección 2.5 como fitness-proportionate; los individuos son elegidos aleatoriamente con una probabilidad proporcional a su aptitud ajustada. No presenta ninguna opción.
- **fitness_overselect** En una población, los individuos con mejor aptitud que superen el corte (*cutoff*) son ubicados en el grupo I, el resto en el grupo II. Se elige un individuo del grupo I de forma aleatoria (proporcional al grado de aptitud) un tanto por ciento de las veces (*proportion*), y del grupo II el resto de las veces. Las opciones son:
  - cutoff** Para poblaciones menores de 1000, su valor por defecto es 0.32. Para poblaciones de mayor tamaño, su valor viene dado por la fórmula $320/\text{tamaño\_población}$ (16% para poblaciones de 2000, 8% para poblaciones de 4000, etc.).
  - proportion** Su valor por defecto es 0.80 (el 80% de las veces).
- **tournament** Se toma un número de individuos de forma aleatoria siguiendo una ley de probabilidad uniforme. Se selecciona el mejor de esos individuos. Tiene una opción:
  - size** Número de individuos seleccionados para competir.
- **inverse_fitness** Los individuos son seleccionados aleatoriamente con una probabilidad igual a $1/\text{aptitud ajustada}$.
- **random** La selección se hace de forma totalmente aleatoria, siguiendo una probabilidad uniforme.

- **best** La primera vez que selecciona un individuo, se elige al mejor de la población. En las siguientes elecciones se va tomando al 2º mejor, el 3º, el 4º, etc. No se debe usar este método cuando se necesiten seleccionar más individuos de los que hay en la población.
- **worst** Es igual que el método anterior pero siguiendo un orden ascendente del *aptitud ajustada*.

3.6. Breeding

Breeding es el término usado en programación genética para denominar al proceso mediante el cual se crea una nueva población para la siguiente generación. Este proceso se lleva a cabo en una serie de “fases”. Cada fase consiste en un operador genético (el cruce por ejemplo) y un porcentaje que indica con qué frecuencia va a actuar dicho operador.

Los parámetros que controlan el breeding son los siguientes:

- **breed_phases** Es un parámetro de tipo entero que indica el número de fases que componen el proceso del breeding. No tiene valor por defecto.
- **probabilistic_operators** Parámetro de tipo binario cuyo valor por defecto es “on”. En este caso, las distintas fases son seleccionadas al azar con una frecuencia proporcional a la tasa asociada a esa fase. En el caso de que su valor sea “off”, el número de individuos producidos en una fase dada es exactamente proporcional a su tasa asociada.

En los nombres de los parámetros siguientes, se debe sustituir el símbolo # por un entero perteneciente al intervalo [1, breed_phases].

- **breed[#].rate** Es una variable tipo float que muestra la tasa (frecuencia de ocurrencia) asociada a la fase indicada al comienzo. Si el parámetro probabilistic_operator toma el valor “on”, este número señala la probabilidad con la que la fase en cuestión será (aleatoriamente) elegida. En otro caso, especifica la proporción de individuos que serán creados en la nueva población con esta fase.
- **breed[#].operator** Parámetro tipo cadena de caracteres que especifica el operador genético asociado a la fase mostrada, y cualquier argumento asociado a éste.

A continuación se muestra un ejemplo de lo que sería un fichero de parámetros completo, utilizado en una aplicación real. Aquellos parámetros que no aparecen, toman su valor por defecto.

```

#
# User Defined Parameters
#
app.no_casos = 500
app.use_erces = on

#
# General Parameters
#
max_generations = 70
pop_size = 500
random_seed = 1779571

#
# Output Parameters
#
output.basename = liapunov
output.detail = 50
output.stt_interval = 1
output.bestn = 1
output.digits = 4

#
# Size Limit Parameters
#
max_depth = 7

```

```

#
# Initialization Parameters
#
init.method = half_and_half
init.depth = 2-6
init.random_attempts = 100

#
# Breed Phase Parameters
#
probabilistic_operators = on
breed_phases = 3

breed[1].operator = crossover,
select=(tournament, size=4)
breed[1].rate = 0.90

breed[2].operator = reproduction,
select=(tournament, size=4)
breed[2].rate = 0.08

breed[3].operator = mutation,
select=(tournament, size=4)
breed[3].rate = 0.02

```

4. IMPLEMENTACIÓN DE UN PROBLEMA

En este punto se describe cómo se implementa realmente una nueva aplicación en lil-gp. La distribución proporciona un fichero *esqueleto* que sirve al usuario como estructura básica sobre la que desarrollar el programa.

El código escrito por el usuario se divide en dos categorías: las funciones C que implementan las funciones (nodos no terminales) y los nodos terminales, y las llamadas de usuario. Las llamadas de usuario, normalmente ubicadas en el fichero `app.c`, realizan tareas específicas de la aplicación desarrollada por el usuario como la inicialización del conjunto de funciones que van a ser utilizadas, el cálculo del grado de aptitud, etc. El otro grupo de funciones C, situadas en el fichero `function.c`, representan el código llamado por el núcleo del sistema durante la evaluación de los árboles.

4.1. Definiciones Básicas

Existen dos constantes definidas en el fichero cabecera `appdef.h` que el núcleo del lil-gp necesita. Ellas son las que se muestran en la tabla 4.5.

Constante	Valor
MAXARGS	Máximo número de argumentos que puede tomar una función en el árbol.
DATATYPE	Tipo de datos C devuelto por todas las funciones y nodos terminales.

Tabla 4.5 Constantes definidas en el fichero `appdef.h`

Este fichero cabecera es también un buen lugar para que el usuario haga todas las definiciones (con la directiva `#define`) que necesite para su aplicación. Se sugiere que todas las constantes definidas lleven el prefijo `APP_` para no entrar en conflicto con otras definiciones hechas en el núcleo del sistema.

4.2. Funciones y Terminales

Por cada nodo terminal y cada función⁷ utilizada en el problema, el usuario deberá escribir una función C que implementará la acción de ese nodo. Estas funciones C se encuentran definidas en el fichero `function.c` y sus prototipos en el fichero `function.h`.

A cada función C se le pasan dos argumentos, uno de tipo `int` y otro un puntero de tipo (`farg *`). El tratamiento de estos argumentos dependerá del hecho de que la función C esté implementando un terminal o una función; además todas las funciones C deben devolver un valor del tipo definido en `DATATYPE`. Por ejemplo, una función C que defina al nodo no terminal “función suma” tendría un aspecto similar al que se muestra en la figura 4.1.

```

DATATYPE f_suma ( int tree, farg *args )
{
    return args[0].d + args[1].d;
}

```

Figura 4.1 Ejemplo de función C implementando un nodo no terminal.

⁷ en esta sección, el término “función” hace referencia a un nodo no terminal de un árbol, y el término “función C” a cualquier función desarrollada en lenguaje C.

Cuando se encuentra un nodo suma en la evaluación de un árbol, el núcleo de lil-gp evalúa primero a sus hijos almacenando sus valores en el array `args`, y luego se llama a la función en sí.

Por otro lado, las funciones C que implementan nodos terminales deben ignorar por completo los dos argumentos que se les pasa. Por ejemplo, supongamos una función de Lyapunov que tiene dos variables independientes `x1` y `x2` (correspondientes a las variables de estado del sistema bajo estudio). Pues bien, una función C que implemente el terminal asociado a una de estas dos variables podría ser la que se ilustra en la figura 4.2.

```
DATATYPE f_x1 ( int tree, farg *args )
{
    return g.x1;
}
```

Figura 4.2 Ejemplo de función C implementando un nodo terminal.

Esta función devuelve el valor de la variable independiente `x1` que forma parte de una variable global tipo `struct`, llamada `g`. Es global porque en ella se suelen almacenar los distintos casos de prueba que se van a probar. En el código fuente se tendría algo similar a lo que se muestra en la figura 4.3.

```
typedef struct
{
    double x1;
    double x2;
} globaldata;

extern globaldata g;
```

Figura 4.3 Variable global implementando las variables independientes

4.2.1. Constantes Aleatorias (o ERCs)

Para crear un terminal que actúe como una constante numérica generada de forma aleatoria, el usuario debe escribir dos funciones C. La primera generará una nueva constante y la otra imprimirá su valor en una cadena de caracteres. A la primera se le pasa un puntero a `DATATYPE`, que acabará almacenando el valor generado. Se ilustra un ejemplo en la figura 4.4.

```
void f_erc_gen ( DATATYPE *r )
{
    *r = random_double() * 10.0;
}
```

Figura 4.4 Función generadora de constantes numéricas

Esta función genera de forma aleatoria un número real perteneciente al intervalo [1, 10].

La segunda función se utiliza cuando se van a imprimir los individuos en un fichero de salida. Se le pasa como argumento un valor tipo `DATATYPE` y devuelve la dirección del buffer donde se ha escrito la cadena de caracteres que representa a la constante numérica. Tenemos un ejemplo de esta función en la figura 4.5.

```
char *f_erc_print ( DATATYPE d )
{
    static char buffer[20];
    sprintf ( buffer, "%.2f", d );
    return buffer;
}
```

Figura 4.5 Función para imprimir los valores de las ERCs generadas

4.3. Llamadas de Usuario

Sólo se necesitan escribir dos llamadas de usuario para definir por completo una aplicación determinada y poder ejecutarla posteriormente (`app_build_function_sets()` para definir el conjunto de funciones y terminales, y `app_eval_fitness()` para evaluar el grado de aptitud de los individuos). El resto de llamadas deben estar presentes en el código pero se pueden dejar tal cuál venían en el fichero esqueleto.

4.3.1. Definiendo el Conjunto de Funciones

La primera llamada de usuario que se tiene que definir es `app_build_function_sets()`. Esta función C crea una tabla donde cada uno de sus elementos representa una función (ya sea ordinaria o un terminal) y obedece a una estructura de tipo `function`. A continuación se muestra en varias tablas, cómo serían los ocho campos asociados al tipo de datos `struct function`, en los tres tipos de nodos que conocemos: funciones ordinarias, nodos terminales y constantes ERC.

Hay una serie de factores que se deben tener en cuenta:

- Los campos `code`, `ephem_gen` y `ephem_str` son punteros a funciones C, no cadenas de caracteres. Se debe escribir aquí el nombre de la función C a la que se está haciendo referencia.
- El campo `string` es el nombre de la función almacenado como cadena de caracteres. Será el nombre que aparezca en los ficheros de salida cuando se imprima un individuo.
- El campo `index` debe estar siempre a cero.

En la tabla 4.6 se muestran los campos en el caso de un nodo que actúa como una función ordinaria (suma, resta, etc.).

Campo	Descripción
<code>code</code>	Función C que implementa dicho nodo
<code>ephem_gen</code>	NULL
<code>ephem_str</code>	NULL
<code>arity</code>	Número de argumentos que toma dicha función (mayor que cero).
<code>string</code>	El nombre de la función
<code>type</code>	FUNC_DATA o FUNC_EXPR
<code>evaltree</code>	-1
<code>index</code>	0

Tabla 4.6 Campos en la estructura de un nodo que representa una función ordinaria

En la tabla 4.7 se hace lo propio cuando se trata de un nodo terminal ordinario.

Campo	Descripción
<code>code</code>	Función C que implementa el terminal
<code>ephem_gen</code>	NULL
<code>ephem_str</code>	NULL
<code>arity</code>	0
<code>string</code>	El nombre del terminal
<code>type</code>	TERM_NORM
<code>evaltree</code>	-1
<code>index</code>	0

Tabla 4.7 Nodo terminal ordinario

Por último, en la tabla 4.8 aparece el caso de un nodo terminal que represente una constante numérica, generada de forma aleatoria.

Campo	Descripción
<code>code</code>	NULL
<code>ephem_gen</code>	Función C para generar nuevas constantes aleatorias.
<code>ephem_str</code>	Función C para imprimir valores en una cadena de caracteres.
<code>arity</code>	0
<code>string</code>	El nombre genérico del terminal, aunque el valor que se imprime en los ficheros de salida será el valor numérico del mismo.
<code>type</code>	TERM_ERC
<code>evaltree</code>	-1
<code>index</code>	0

Tabla 4.8 Nodo terminal ERC

A continuación se muestra un ejemplo de definición del conjunto de funciones, extraído de la aplicación desarrollada en este proyecto. Se puede ver en la figura 4.6.

```

function sets[14] =
  {
    { f_mult, NULL, NULL, 2, "*", FUNC_DATA, -1, 0 },
    { f_div, NULL, NULL, 2, "/", FUNC_DATA, -1, 0 },
    { f_suma, NULL, NULL, 2, "+", FUNC_DATA, -1, 0 },
    { f_resta, NULL, NULL, 2, "-", FUNC_DATA, -1, 0 },
    { f_raiz, NULL, NULL, 1, "raiz", FUNC_DATA, -1, 0 },
    { f_cuad, NULL, NULL, 1, "^2", FUNC_DATA, -1, 0 },
    { f_sen, NULL, NULL, 1, "sen", FUNC_DATA, -1, 0 },
    { f_cos, NULL, NULL, 1, "cos", FUNC_DATA, -1, 0 },
    { f_atan, NULL, NULL, 1, "atan", FUNC_DATA, -1, 0 },
    { f_est1, NULL, NULL, 0, "f1", TERM_NORM, -1, 0 },
    { f_est2, NULL, NULL, 0, "f2", TERM_NORM, -1, 0 },
    { f_x1, NULL, NULL, 0, "X1", TERM_NORM, -1, 0 },
    { f_x2, NULL, NULL, 0, "X2", TERM_NORM, -1, 0 },
    { NULL, f_erc_gen, f_erc_print, 0, "R", TERM_ERC, -1, 0 }
  };

```

Figura 4.6 Tabla que define el conjunto de funciones que se utilizarán para formar los árboles

En este caso se tienen 9 funciones correspondientes a nodos no terminales (funciones ordinarias) que son la suma, resta, división, multiplicación, raíz cuadrada, potencia de dos, seno, coseno y arcotangente; 4 funciones correspondientes a nodos terminales ordinarios que son las dos variables de estado y las dos expresiones que representan su derivada con respecto al tiempo. Por último hay una función asociada a los nodos terminales que actúan como constantes numéricas aleatorias.

4.3.2. Función de Evaluación de la Aptitud

Esta función, denominada `app_eval_fitness()`, se llama cada vez que se necesita evaluar a un individuo. Se le pasa como argumento un puntero a una estructura tipo `individual`. Además se trabaja con una serie de variables que se muestran en la tabla 4.9.

Nombre	Descripción
<code>r_fitness</code>	La aptitud bruta.
<code>s_fitness</code>	Aptitud estándar (todos sus valores no negativos, el cero es un individuo perfecto).
<code>a_fitness</code>	Aptitud ajustada (valores en el intervalo [0, 1] donde 1 es el individuo perfecto).
<code>hits</code>	Medida auxiliar para contar los casos de prueba que representan un "éxito".
<code>evald</code>	Siempre debe estar igualado a <code>EVAL_CACHE_VALID</code> para indicar que el campo del fitness es válido.

Tabla 4.9 Variables necesarias para evaluar a un individuo

En el código de `app_eval_fitness()`, lo primero que se debe hacer es llamar a la función `set_current_individual()` pasándole como argumento el mismo puntero que se le pasó a la primera. Para calcular el valor devuelto por un árbol, hay que utilizar la función `evaluate_tree()`.

Normalmente, la función de evaluación del grado de aptitud contiene un bucle principal sobre el que se itera para todos los casos de prueba. La variable global `g`, a la que se hacía referencia en la figura 4.3, se utiliza para intercambiar información entre la función `app_eval_fitness()` y las funciones que implementan los nodos. Por ejemplo, en el caso de funciones de Lyapunov con dos variables independientes, en cada caso de prueba se le asignará un valor a `g.x1` y `g.x2` para que el árbol sea evaluado con esos valores. Cuando se está evaluando un individuo y el programa llega a un nodo terminal de este tipo, la función `C` que lo implementa simplemente lee su valor y lo devuelve (ver figura 4.2).

En la figura 4.7 se muestra la estructura básica que sigue normalmente la función de evaluación.

```

void app_eval_fitness ( individual *ind )
{
    set_current_individual ( ind );
    ...
    for ( < bucle para todos los casos de fitness > )
    {
        <código que asigna valores a la variable>
        <global g, para el actual caso de fitness>
        ...
        value = evaluate_tree( ind->tr[0].data,0);

        /* aquí se pueden evaluar todos los árboles
         * que sean necesarios para la asignación de
         * un fitness al individuo en cuestión. */
        ...
    }

    ind->hits = <lo que sea>;
    ind->r_fitness = <lo que sea>;
    ind->s_fitness = <lo que sea>;
    ind->a_fitness = <lo que sea>;

    /* indica que los campos de fitness son correctos */
    ind->evald = EVAL_CACHE_VALID;
}

```

Figura 4.7 Estructura básica de la función de evaluación de la aptitud

4.3.3. Otras Llamadas de Usuario

Después de la evaluación de cada generación, lil-gp llama a la función `app_end_of_evaluation()`. Se le pasa como argumentos la generación actual, un puntero a la población completa, estadísticas de la ejecución del programa y una bandera que indica si se ha encontrado un individuo mejor. La función devuelve un 1 si se alcanza la condición (o condiciones) impuesta por el criterio de terminación, o un 0 en otro caso. Esta función es un buen lugar para codificar el criterio de terminación de la ejecución, que normalmente estará determinado por el hecho de que aparezca un individuo “perfecto” que represente una solución adecuada al problema.

Lil-gp proporciona dos funciones para poder realizar tareas relacionadas con la inicialización, que resulten necesarias en la aplicación y que son: `app_initialize()` y `app_uninitialize()`. En la primera puede resultar útil realizar inicialización de variables o reserva dinámica de memoria. En la segunda, llamada al final de la ejecución del programa, se libera la memoria reservada.

4.4. Ejecución de lil-gp

Una vez que el usuario haya escrito todo el código necesario y se haya compilado con éxito, el sistema genera un fichero ejecutable llamado `gp`. En este momento, se está en condiciones de ejecutar la aplicación desde la línea de comandos. Existen varios argumentos que el usuario puede utilizar; éstos son algunos de ellos:

- f <fichero de parámetros>
Esta opción carga el fichero de parámetros especificado en la base de datos de lil-gp.
- p <parámetro = valor>
Esta opción pone el parámetro especificado a un valor determinado.
- q Provoa que lil-gp se ejecute en modo silencioso, sin mostrar ningún mensaje por pantalla. Sin esta opción, todos los mensajes se imprimen en el fichero de salida `.sys` y en la salida estándar.

Por ejemplo, la siguiente línea de comandos ejecutaría un programa en lil-gp cargando el fichero de parámetros `input.file` y haciendo que la semilla inicial valga 117:

```
gp -f input.file -p random_seed = 117
```

Se ha intentado en este capítulo, dar una breve (pero al mismo tiempo completa) descripción de los conceptos más importantes del software utilizado para desarrollar este proyecto. Se recomienda al lector que consulte

el manual de usuario proporcionado con la distribución, si desea profundizar más o aclarar algunas ideas. La distribución está disponible en [6].