Programación Evolutiva

Tema 7: Programación genética

Carlos Cervigón, Lourdes Araujo. 2009-2010

Programación genética

- Es un método de Programación Automática. El objetivo no es encontrar la solución a un problema sino el procedimiento para resolverlo.
- Las estructuras que evolucionan son programas que, al ser ejecutados, determinan dichas soluciones.
- Aplica un proceso evolutivo a una población de programas
- Los individuos son programas (formato LISP, normalmente)
- Los individuos se almacenan en un árbol
  - los nodos representan funciones
  - Las hojas representan símbolos terminales
- Todos los programas en la población inicial han de ser sintácticamente correctos (ejecutables) y los operadores genéticos (cruce, reproducción, mutación, …) también han de producir programas sintácticamente correctos

Algoritmo

- **Generar una población inicial de programas aleatorios, mediante expresiones LISP o árboles formados por funciones y símbolos terminales.**
- **Repetir hasta Fin:**
  - **Ejecutar cada programa y calificar su adaptación para resolver el problema.**
  - **Aplicar selección.**
  - **Aplicar cruce y/o mutación.**
- **Devolver el mejor programa como solución.**

Pasos a seguir

- Debido a que la representación más usual en programación genética es un árbol, hablaremos de funciones y terminales, que representan respectivamente los nodos internos y las hojas del árbol.
- Los pasos a seguir en un problema de programación genética son los siguientes:
  - Identificar el conjunto de terminales
  - Identificar el conjunto de funciones y su aridad
  - Identificar la función de adaptación
  - Identificar los parámetros del algoritmo
  - Identificar los criterios de terminación

Programación genética

- La función de adaptación normalmente consiste en evaluar la "calidad" de un programa o individuo respecto a la resolución del problema.
- Por ejemplo, si el problema consiste en recorrer el número máximo de casillas de un tablero, valoraríamos el número de casillas recorridas al ejecutar el programa en cuestión.
- Parámetros del algoritmo : tamaño de población (número de programas que evolucionan), las probabilidades de aplicación de los operadores genéticos, la profundidad de los árboles o longitud máxima de los programas, etc.

Programación genética

- Ejemplo: programa para calcular el periodo orbital de un planeta P, dada su distancia media al sol A
- La tercera ley de Kepler establece que

$$P^2 = cA^3$$

siendo c una constante.

- Suponiendo que P se expresa en años terrestres y A se expresa en unidades de la distancia media de la tierra al sol, c toma el valor 1.

$$P = sqrt(A*A*A)$$

Ejemplo: Cálculo periodo orbital

En Lisp:

```
(defun periodo_orbital ()
      (setf A 1.52)
      (sqrt (* A (* A A))))
```

- En Lisp los operadores preceden a sus argumentos.
- El operador setf asigna su segundo argumento (un valor) a su primer argumento (una variable).

Evolución de programas LISP

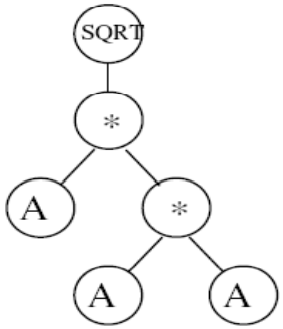
- El valor de la última expresión del programa se imprime automáticamente.
- Conociendo A, la instrucción relevante del programa es:

```
SQRT(*A(*AA)))
```

- Problema de programación automática: descubrir automáticamente esta expresión, a partir de datos medidos para P y A.
- Estas expresiones pueden representarse como árboles de análisis, compuestos de funciones y terminales.

Evolución de programas LISP

- SQRT es una función que toma un argumento, * una función que toma dos argumentos y *A* es un terminal.
- El argumento de una función puede ser el resultado de otra función.



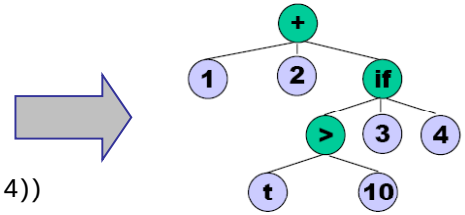
Ejemplo

PASCAL:

```
function ejemplo
   (tiempo:integer):integer;
var aux : integer;
begin
  if(tiempo >10) then aux:= 3
    else aux:= 4;
  ejemplo := 1 + 2 + aux;
End;
```

LISP :

(+ 1 2 (if(> tiempo 10) 3 4))



Evolución de programas LISP

- Se elige un conjunto de posibles funciones y terminales para el programa. No se sabe de antemano qué funciones y terminales se van a necesitar para que el programa tenga éxito.
- Para el problema del periodo orbital el conjunto de funciones puede ser {+,-,*,/,sqrt}, y el de terminales la variable {A}, (suponiendo que usuario sepa que la función será una función aritmética de A).
- Se genera un población inicial de programas aleatorios (árboles) usando el conjunto de funciones y terminales posibles.
- Estos árboles deben ser sintácticamente correctos.

Evolución de programas LISP

- Conviene establecer un límite a su profundidad de los árboles generados.
- Para generarlos en cada nodo se decide con cierta probabilidad si será una función o un terminal.
- Se calcula la adaptación de cada programa de la población, probándolo sobre los datos de entrada (un conjunto de medidas experimentales de P y A).
- Ejemplos de programas:

  (+A(*(SQRT A) A))

  (*A(-(*A A) (SQRT A)))

  (*A(-(* SQRT A A) (+ A A)))

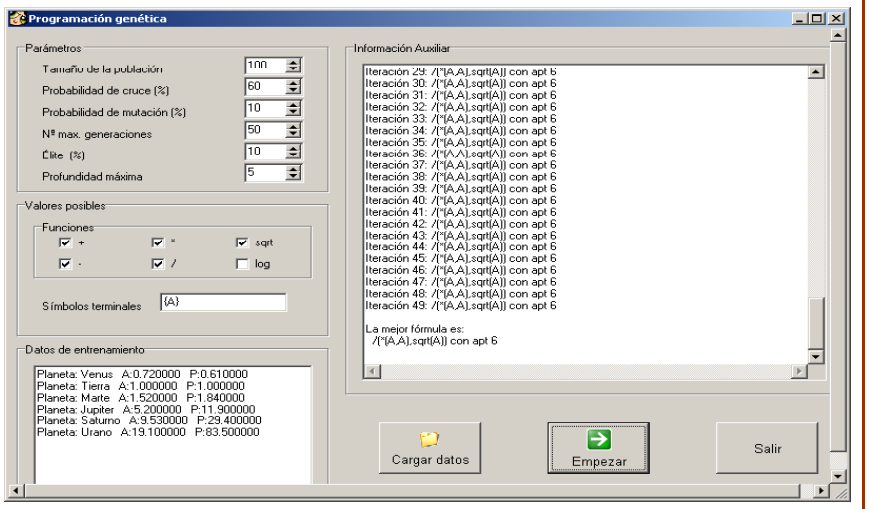
. . .

Evolución de programas LISP

- Para analizar la adaptación de cada uno de los programas generados calculamos su adaptación sobre un conjunto de datos de entrada de P y A como medidas experimentales;
- La adaptación de un programa puede ser el número de casos de prueba en los que produce un resultado correcto o con un error muy pequeño.
- La aptitud de un programa es una función del número de casos de prueba en los que produce el resultado correcto (o con un pequeño margen de error).

	A	P
Venus	0.72	0.61
Tierra	1.0	1.0
Marte	1.52	1.84
Jupiter	5.20	11.9
Saturno	9.53	29.4
Urano	19.1	83.5

Ejemplo



Evolución de programas LISP

- Se aplican los operados de selección, cruce y mutación para producir un cierto número de generaciones con una elección adecuada de parámetros para el algoritmo.
- Un cruce consiste en elegir un punto aleatorio de cada padre e intercambiar los subárboles bajo estos puntos para producir dos hijos.
- El cruce permite que el tamaño del programa se incremente o decremente.
- La mutación puede realizarse eligiendo un punto aleatorio del árbol y reemplazando el subárbol bajo este punto por otro generado aleatoriamente.
- También puede consistir en cambiar un operador por otro elegido aleatoriamente (respetando la aridad) o un terminal por otro.

Algoritmo en PG

- Crear al azar una población de P programas compuestos por los símbolos de los conjuntos de funciones y terminales.
- Hasta cumplir el criterio de terminación
  - Ejecutar cada programa de la población y obtener su aptitud
  - Seleccionar los individuos de la población por su aptitud
  - Crear nuevos individuos (programas) mediante la aplicación de los siguientes operadores genéticos con probabilidades específicas
    - **Cruce**
    - **Mutación**
- Devolver como resultado el mejor individuo de la población final

Esquema tradicional

```
  pob.evalua();

    for(int idx = 0; (idx < maxGen) && (!terminar); idx++ ){

        pob.selecciona();
        pob.reproduce();
        pob.muta();
        pob.evalua();
}

    return pob.getMejorIndividuo();
}
```


Otro ejemplo

- Objetivo: encontrar la expresión que nos permite encontrar una solución para ecuaciones del tipo

$$ax^2 + bx + c = 0$$

- Como ya sabemos, la siguiente expresión nos permite obtener dichas soluciones

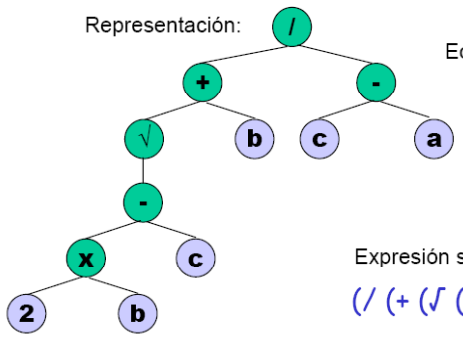
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Funciones: + - x / √

Terminales: a b c 2

Ejemplo : Representación

- Representamos cada individuo se representa mediante un árbol. Por ejemplo:



Ecuación: $x = \frac{\sqrt{2b - c} + b}{c - a}$

Expresión simbólica en LISP:

(/ (+ (√ (- (x 2 b) c)) b) (- c a))

Representación

```
public class Cromosoma {

 private double aptitud;
 private Arbol arbol;
 private int hMaxima;
 private final String[] cjtoTerms={"a","b","c","2","0"};
 private final String[] cjtoFuns={"+","-","*","/","sqrt"};
 . . .

}
```

Representación

```
public class Arbol {

 private Arbol hi;
 private Arbol hd;
 private Arbol padre;
 private int profundidad;
 private String valor;
 private boolean esRaiz;
 private boolean esHoja;
 private int numNodos;
 private boolean esHi;
 private int posSimbolo;
. . . .
```


Ejemplo: Evaluación

- Dado un conjunto de ecuaciones de segundo grado se calcula cómo resuelve el individuo dicho conjunto
- Cada ecuación del conjunto de ejemplos contendrá un valor para las variable a, b y c
- Se sustituyen dichos valores en la ecuación representada por el individuo a evaluar, y así se obtiene un valor para la x
- Se sustituye dicho valor en la ecuación de segundo grado, y se comprueba si el resultado es cero
- Si es así, el individuo resuelve la ecuación-ejemplo, si no es así, la diferencia se utilizará para el cálculo de la aptitud del individuo

Ejemplo: Evaluación

- Por ejemplo, dada la ecuación

$$-4x^2+4x-1=0$$

- Sustituyendo estos valores en el individuo ejemplo anterior se obtiene:

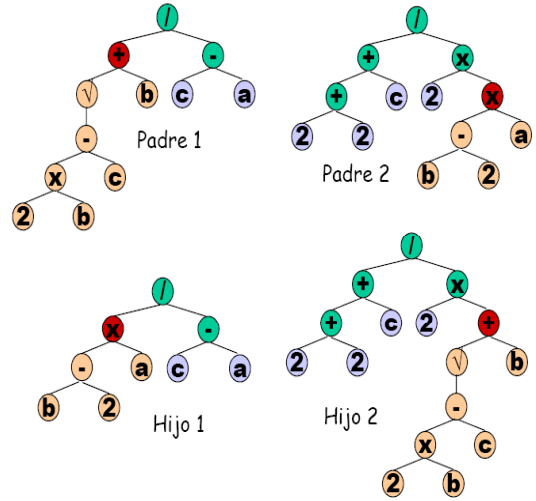
$$x=\frac{\sqrt{2b-c}+b}{c-a}=\frac{\sqrt{8-(-1)}+4}{(-1)-(-4)}=\frac{\sqrt{9}+4}{3}=\frac{7}{3}=2.33$$

- Sustituyendo x por su valor 2.33 en la ecuación de segundo grado se obtiene

$$-4x^2+4x-1=$$
$$-4(2.33)^2+4(2.33)-1=$$
$$-21.72+9.32-1=-13.4$$

Ejemplo: Cruce

- Los puntos de cruce del operador de cruce no tienen porque tener la misma probabilidad de selección
- Una estrategia frecuente consiste en seleccionar los nodos internos (funciones) con una probabilidad de, por ejemplo, 0.9 y cualquier nodo con el resto de la probabilidad



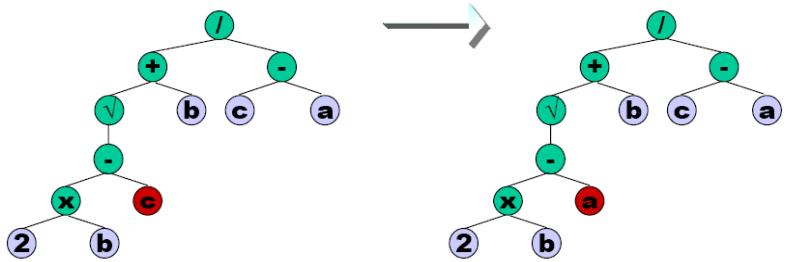
Cruce

```
Hijos TIndividuo::cruce(const TIndividuo* otroPadre) {
    Hijos hijos;
    hijos.hijo1= new TIndividuo(*this);
    hijos.hijo2=new TIndividuo(*otroPadre);
    Instruccion* instruccion1;
    Instruccion* instruccion2;
    int puntoCruce1;
    int puntoCruce2;
    puntoCruce1=2+random(hijos.hijo1->getArbol()->numInstrucc-1);
    puntoCruce2=2+random(hijos.hijo2->getArbol()->numInstrucc-1);
    corta(puntoCruce1, puntoCruce2, hijos.hijo1->getArbol(),
   hijos.hijo2->getArbol());
    //Calculamos el valor de los nuevos hijos.
    hijos.hijo1->adaptacion();
    hijos.hijo2->adaptacion();
    return hijos;
}
```


Ejemplo: mutación

La mutación consiste en generar un nuevo programa a partir de un único progenitor. Las técnicas de mutación más comunes son:

- Mutación terminal simple : Se selecciona al azar un símbolo terminal dentro del individuo, y se sustituye por otro diferente

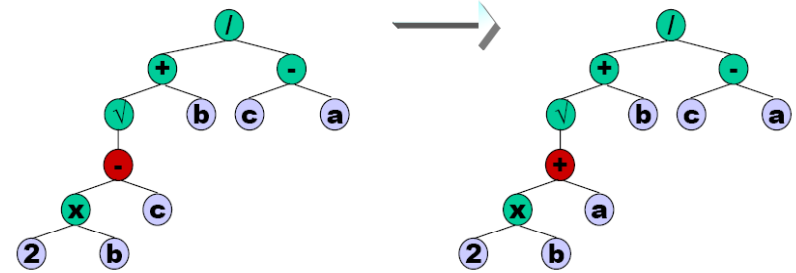


Mutación terminal simple

```
public void mutacionTerminalSimple(){
 char []terminales = new char[4];
 terminales[0] = 'a';terminales[1] = 'b';
 terminales[2] = 'c';terminales[3] = '2';

for (int i = 0; i < this.Poblacion; i++) {
 Cromosoma c = (Cromosoma) individuos.get(i);
 double numAle = Math.random();
 if(numAle<this.PrMut){
   int numAle2 = (int) (Math.random()*4);
   Simbolo s = getTerminalAleatorio(c);
   s.setTerminal(terminales[numAle2]);
  }
 }
}
```

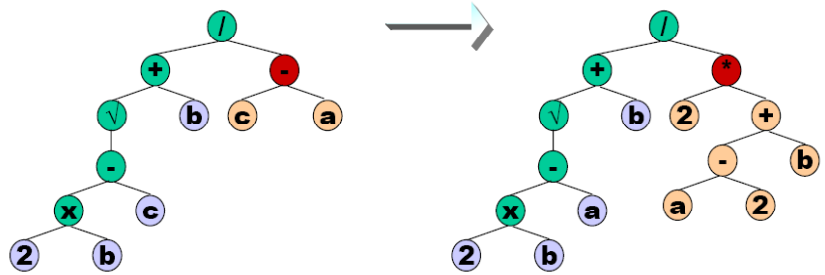

Ejemplo: mutación

- Mutación funcional simple : Se selecciona al azar una función dentro del individuo, y se sustituye por otra diferente del conjunto de funciones posibles con el mismo número de operandos



Ejemplo: mutación

- Mutación de árbol : Se selecciona un subárbol del individuo, igual que en el operador de recombinación, se elimina totalmente el subárbol seleccionado y en su lugar se incorpora un nuevo subárbol generado aleatoriamente



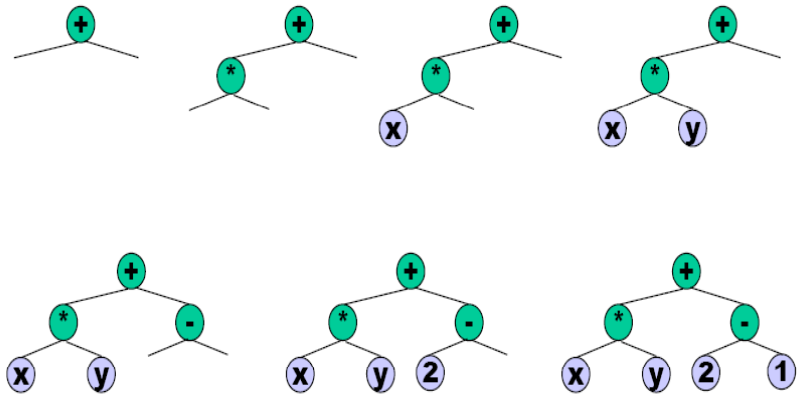
Ejemplo mutación de árbol

```
void TIndividuo::muta(float probMutacion){

    //Ejemplo de mutación en la que los individuos a mutar son
   directamente reconstruidos.

    //Comprobamos si vamos a mutar al individuo
    if(rand()/(RAND_MAX+1.0f) < probMutacion ){

        borraArbol(this->arbol);
        this->aptitud= 0.0f;
        this->ajustado=false;
        this->puntuacion = 0.0f;
        this->inicializa();

    }

}
```


Técnicas de inicialización

- Inicialización completa (*Full initialization*)
  - Vamos tomando nodos del conjunto de funciones hasta llegar a una máxima profundidad del árbol definida previamente
  - Una vez llegados a la profundidad máxima los símbolos sólo se toman del conjunto de símbolos terminales
- Inicialización creciente (*Grow initialization*)
  - Vamos tomando nodos del conjunto completo (funciones y terminales) hasta llegar al límite de profundidad especificado previamente
  - Una vez llegados a la profundidad máxima este método de inicialización se comporta igual que el método de inicialización completa

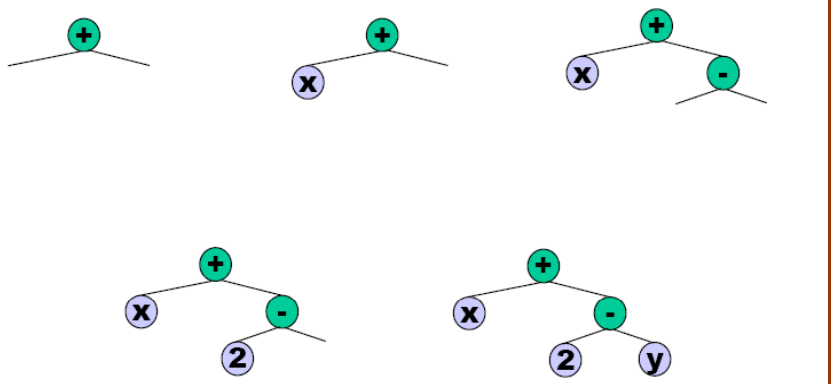
Inicialización completa



Inicialización completa

```
funcion inicializacionCompleta(profundidad) {
 si profundidad < maximaProdundidad entonces
   nodo ← aleatorio(conjFunciones)
   para i = 1 hasta número de hijos del nodo hacer
     Hijoi = inicializacionCompleta(profundidad+1 )
 eoc
   nodo ← aleatorio(conjTerminales)
 devolver nodo
}
```


Inicialización creciente

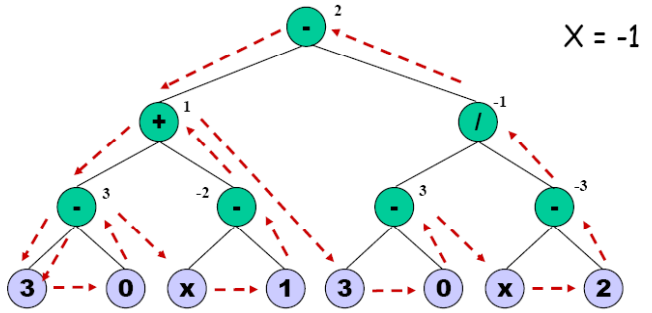


Inicialización creciente

```
función inicializacionCreciente(profundidad) {

 si profundidad < maximaProdundidad árbol entonces
   nodo ← aleatorio(conjFunciones ⊕ conjTerminales)
   para i = 1 hasta número de hijos del nodo hacer
     Hijoi = inicializacionCreciente(profundidad+1 )
 eoc
   nodo ← aleatorio(conjTerminales)
 devolver nodo
}
```


Interpretación de un programa en PG

- Interpretar un programa descrito por un árbol significa ejecutar los nodos del árbol en el orden que garantice que ningún nodo se ejecuta con anterioridad a la obtención de los valores de sus parámetros



Interpretación de un programa en PG

```
function evaluar ( expresión ) // Una expresión en notación prefija
{
  float valor;
  if ( expresión es una lista )   // Símbolo no terminal
    función = expresión [ 1 ];
    valor = función ( evaluar ( expresión [ 2 ] , expresión [ 3 ] , … ) );
  else // Símbolo terminal
    if ( expresión es una variable o una constante )
      valor = expresión;
    else // Función con aridad = 0
      valor = expresión ( );
  return (valor);
}
```


Aptitud de un programa

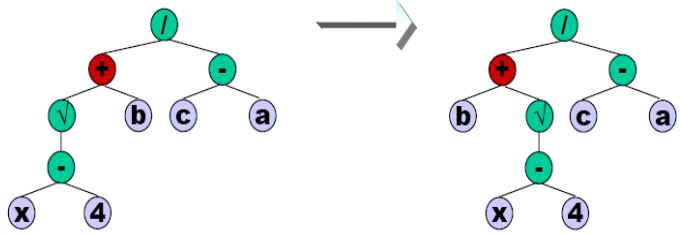
- El valor de aptitud asignado a un programa puede medirse de formas diferentes. Por ejemplo
  - La cantidad de error entre sus salidas y las salidas deseadas
  - La cantidad de tiempo (dinero, …) necesaria para llevar un sistema a un determinado estado objetivo
  - La precisión de un sistema de reconocimiento de patrones o sistema clasificador
  - La adecuación de una estructura compleja (antena, circuito, controlador, …)
- En muchos problemas, cada programa se ejecuta sobre una muestra representativa de diferentes casos (diferentes valores para las entradas, diferentes condiciones iniciales o diferentes entornos)

Cruce

- En PG, el cruce de dos individuos iguales no tiene porque generar dos descendientes iguales como ocurre en AG
- Por lo que el cruce de dos árboles parecidos con altos valores de adaptación (cuando estamos próximos a la convergencia del algoritmo) no tiene porque generar descendientes altamente adaptados
- Esto implica una buena exploración del espacio de búsqueda en detrimento de la explotación de dicho espacio
- Por ello, en PG, debemos emplear poblaciones mucho más grandes que en los AG

Permutación

- Intercambia el orden de la lista de argumentos de una función
- Operador equivalente al operador de inversión de los AG

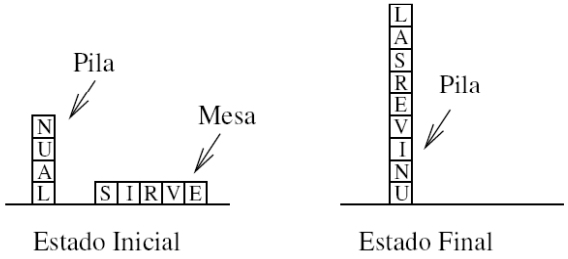


Problemas en PG

- Los principales problemas en PG son
  - La supervivencia de los individuos más grandes
  - El tratamiento de las constantes numéricas
- Para reducir el efecto del crecimiento de los árboles
  - Modificar los operadores de variación para que no construyan individuos muy grandes
  - Incluir algún término en la función de evaluación que penalice a los árboles según su tamaño
- Para mejorar el tratamiento de las constantes numéricas
  - Utilizar un algoritmo híbrido entre PG y AG

Ejemplo: Apilamiento de bloques (Koza)

- El objetivo es encontrar un programa que a partir de una configuración inicial de bloques (algunos en la mesa, algunos apilados) los coloque en el orden correcto en la pila.
- En el ejemplo, el orden correcto es el que corresponde al deletreo de la palabra UNIVERSAL.

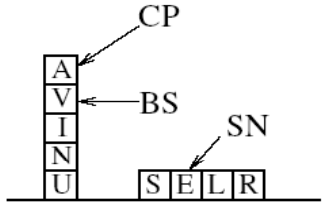


Ejemplo: Apilamiento de bloques (Koza)

- Los terminales para este problema son un conjunto de sensores T={CP,BS,SN} que devuelven:
  - CP (cima pila): nombre del bloque de la cima de la pila. Si la pila está vacía devuelve NIL.
  - BS (bloque superior correcto): nombre del bloque más alto en la pila tal que él y todos los bloques que tiene por debajo están en el orden correcto. Si no existe tal bloque BS devuelve NIL.

Ejemplo: Apilamiento de bloques (Koza)

- SN (siguiente necesario): nombre del bloque que se necesita poner inmediatamente sobre BS en el objetivo UNIVERSAL (independientemente de que haya o no bloques incorrectos en la pila). Si no se necesitan más bloques devuelve NIL.



- Estos terminales toman como valores las etiquetas de los bloques o NIL y representan las variables del programa.

Ejemplo: Apilamiento de bloques (Koza)

También se dispone de cinco funciones:

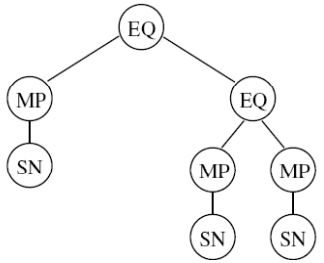
- MP <x>(mover a pila) pone el bloque x en la cima de la pila si x está sobre la mesa. (En Lisp cada función devuelve un valor, que a menudo se ignora).
- MM <x>(.Mover a mesa.) mueve el bloque de la cima de la pila a la mesa si el bloque x está en cualquier punto de la pila.
- DU(acción, predicado) ("do until") ejecuta la acción hasta que predicado se hace cierto (TRUE).
- NOT(expresión1) devuelve cierto (TRUE) si expresión1 es NIL; en otro caso devuelve NIL,
- EQ(expresión1, expresión2) devuelve cierto (TRUE) si expresión1 y expresión2 son iguales.

Ejemplo: Apilamiento de bloques (Koza)

- El algoritmo trabaja con árboles de análisis que representan programas construidos con los terminales y funciones descritos. El programa:

  (EQ(MP SN) (EQ (MP SN) (MP SN)))

- Mueve el siguiente bloque que se necesite a la pila tres veces y está representado por el árbol:



Algoritmo

- Se genera una población inicial de programas aleatorios (árboles sintácticamente correctos y con un límite de profundidad) usando las funciones y terminales definidos.
- La adaptación de cada programa se calcula como el resultado de su ejecución sobre distintas configuraciones iniciales.
- La adaptación de cada programa es una función del número de casos de prueba en los que se produce el resultado correcto.

```
* U N I V E R S A L *
U * N I V E R S A L *
U N * I V E R S A L *
U N I * V E R S A L *
U N I V * E R S A L *
U N I V E * R S A L *
U N I V E R * S A L *
        ...
```

Datos leidos de fichero

Hasta el asterisco son bloques sobre la mesa, desde el asterisco son bloques sobre la pila.

Ejemplo: Apilamiento de bloques (Koza)

- La aptitud de un programa es el número de casos favorables (configuraciones iniciales de bloques) para los que la pila era correcta después de ejecutar el programa.
- Koza usó 166 casos de prueba diferentes, construidos cuidadosamente para recoger las distintas clases de posibles configuraciones iniciales.
- La población inicial contiene entre 50 y 300 programas generados aleatoriamente.

Ejemplo: Apilamiento de bloques (Koza)

Ejemplos:

(EQ (MM CP) SN)

- Mover la cima actual de la pila a la mesa, y ver si es igual al siguiente necesario.
- No consigue ningún progreso: aptitud 0.

(MP BS)

- Mover a la pila el bloque más alto correcto de la pila.
- No hace nada, pero permite conseguir un caso de aptitud correcta: el caso en el que todos los bloques ya estaban en la pila en el orden correcto (aptitud = 1).

Ejemplo: Apilamiento de bloques (Koza)

Ejemplos:

(EQ(MP SN) (EQ (MP SN) (MP SN)))

- Mover el siguiente bloque que se necesite a la pila tres veces.
- Hace algunos progresos y consigue 4 casos correctos (aptitud=4).
- EQ sirve únicamente como una estructura de control.
- Lisp evalúa la primera expresión, después la segunda, y después compara sus valores. EQ ejecuta las dos expresiones en secuencia, y no importa si sus valores son iguales.

Ejemplo: Apilamiento de bloques (Koza)

- En la generación 5, la población contenía muchos programas de aptitudes relativamente buenas. El mejor:

(DU (MP SN) (NOT SN))

- Mover el siguiente bloque necesario a la pila hasta que no se necesiten más bloques.
- Funciona en todos aquellos casos en que los bloques de la pila ya están en el orden correcto.
- Hubo 10 de estos casos (aptitud=10).
- Este programa usa el bloque constructivo (MP SN) que se descubrió en la primera generación y que ha resultado útil aquí.

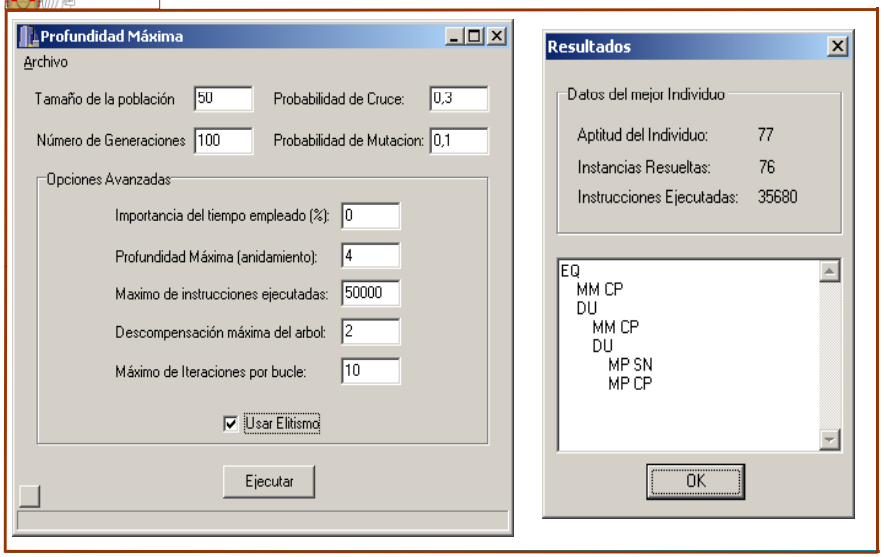
Ejemplo: Apilamiento de bloques (Koza)

- En la generación 10 se obtuvo un programa completamente correcto (aptitud 166):

(EQ (DU (MM CP) (NOT CP)) (DU (MP SN) (NOT SN)))

- Extensión del mejor programa de la generación 5.
- Vacía la pila sobre la mesa y después mueve el siguiente bloque que se necesita a la pila hasta que ya no se necesitan más bloques.
- La PG ha descubierto un plan que funciona en todos los casos, aunque no es muy eficiente.
- En una de sus publicaciones Koza presenta formas de modificar la función de aptitud para producir un programa más eficiente para esta tarea.

Ejemplo

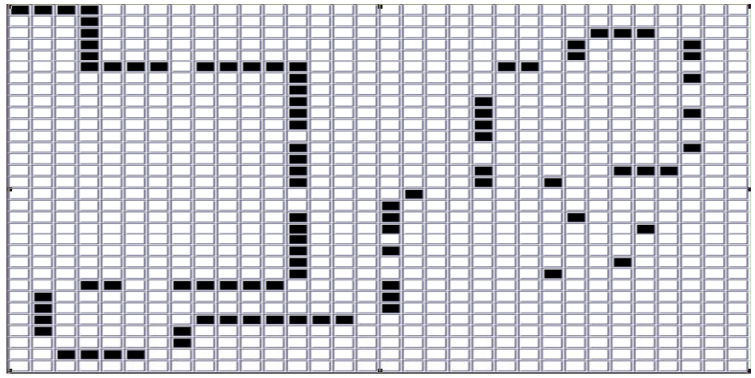


Ejemplo de la Hormiga artificial

- Se trata de diseñar una hormiga artificial capaz de encontrar toda la comida situada a lo largo de un rastro irregular. El objetivo es utilizar programación genética para obtener el programa que realiza esta tarea.
- La hormiga artificial se mueve en un tablero (toroidal) de 32 x 32. La hormiga comienza en la esquina superior izquierda del tablero, identificada por las coordenadas (0,0), y mirando en dirección este.
- Un modelo de rastro propuesto, con el que se han realizado diversos estudios, se conoce como "rastro de Santa Fe" , y tiene una forma irregular compuesta de 89 "bocados" de comida.

Ejemplo de la Hormiga

- El rastro no es recto y continuo, sino que presenta huecos de una o dos posiciones, que también pueden darse en los ángulos.



Ejemplo de la Hormiga

- Necesitamos operaciones que permitan a la hormiga moverse hacia delante, girar a uno u otro lado y detectar comida a lo largo de rastro irregular.

Terminales = {AVANZA, DERECHA, IZQUIERDA}

donde AVANZA mueve la hormiga hacia delante en la dirección a la que mira en ese momento, DERECHA gira la hormiga 90° a la derecha, e IZQUIERDA la gira 90° a la izquierda.

Funciones = { *SIC*(a,b), *PROGN2* (a,b), *PROGN3* (a,b) }

La información que queremos procesar es la que la hormiga obtiene del mundo exterior a través de un sencillo sensor que detecta si hay comida en frente. Por ello incluiremos en el conjunto de operadores uno de selección condicional dependiente de la información del sensor.

Ejemplo de la Hormiga

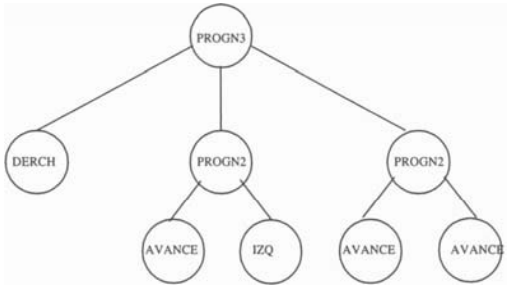
- En el conjunto de operadores incluiremos también conectivas que fuerzan la ejecución de sus argumentos en un determinado orden.
- Tomando el nombre de una conectiva de Lisp, PROGN, que causa este efecto, incluimos en el conjunto de operadores las conectivas PROGN2 y PROGN3, que toman dos y tres argumentos respectivamente. Por lo tanto disponemos de las siguientes funciones:
  - ***SIC (a,b)*** *:* el operador SI_Comidadelante toma dos argumentos y ejecuta *a* si se detecta comida delante y *b* en otro caso.
  - ***PROGN2(a,b)*** : evalúa *a*, luego *b*, y devuelve el valor de *b*.
  - ***PROGN3(a,b,c)*** : evalúa *a, b* luego *c*, devolviendo el valor de *c*

Ejemplos

(PROGN3 (DERECHA) (PROGN2 (PROGN2 (AVANZA) (IZQUIERDA)) (AVANZA) (AVANZA))

(PROGN3 (SIC) (SIC) (PROGN3 (SIC) (SIC (SIC AVANZA DERECHA) (SIC DERECHA DERECHA)) AVANZA))

- Las expresiones de este tipo pueden representarse como árboles de análisis, compuestos de funciones y terminales.



Ejemplos

- Una medida natural de la aptitud para este problema es la cantidad de alimento comido por la hormiga dentro de un espacio de tiempo razonable al ejecutar el programa a evaluar.
- Se considera que cada operación de movimiento o giro consume una unidad de tiempo. En nuestra versión del problema limitaremos el tiempo a 400 pasos. El tablero se va actualizando a medida que desaparece la comida.

Ejemplos

Falla AVANZA

(PROGN2 (DERECHA) (IZQUIERDA))

Ejemplos

- AVANZA sin mirar o girar

  (PROGN2 (AVANZA) (AVANZA))

- Contiene condicional pero no AVANZA

  (SIC (DERECHA) (IZQUIERDA))

. . . .

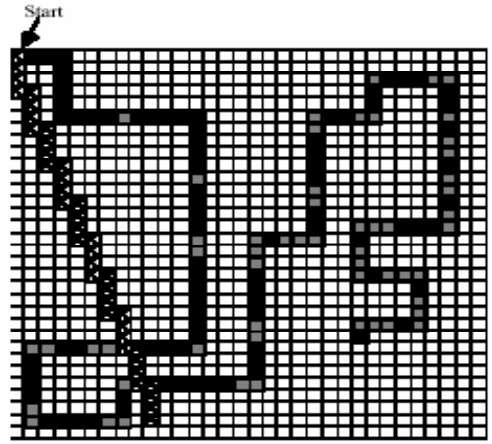
  (SIC (DERECHA) (DERECHA))

- Contiene condicional pero usa la información mal

  (SIC (PROGN2 (IZQUIERDA) (IZQUIERDA) (AVANZA))…)

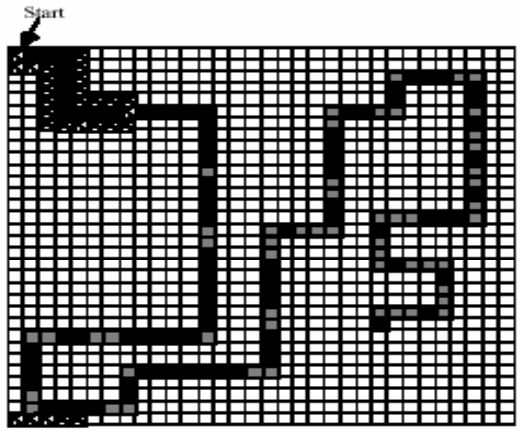
Ejemplos

- (PROGN3 (DERECHA) (PROGN3 (AVANZA) (AVANZA) (AVANZA)) (PROGN2 (IZQUIERDA) (AVANZA)))
  - Paso 1 :D,A,A,A
  - Paso 2 :I,A
  - Paso 3 : Goto Paso1



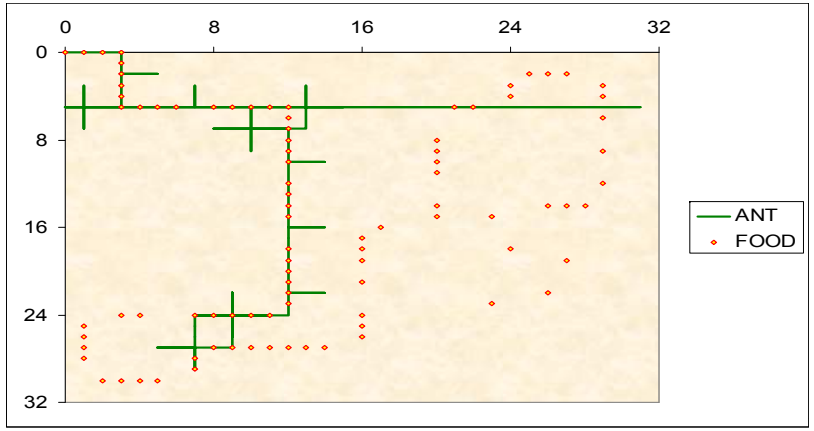
Ejemplos : estrategia esquivadora

- (SIC (DERECHA) (SIC (DERECHA) (PROGN2 (AVANZA) (IZQUIERDA))))



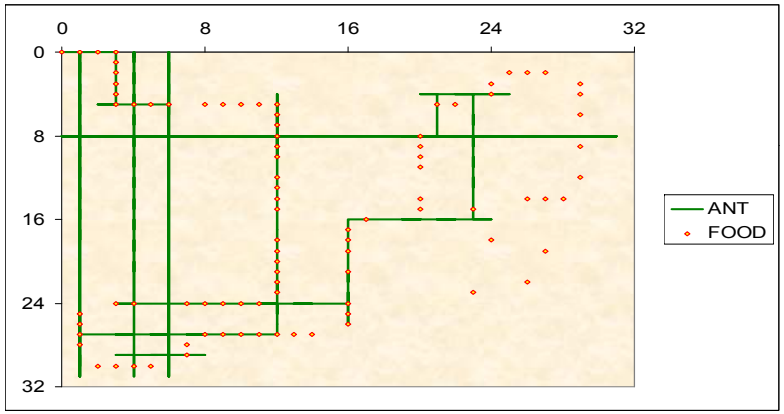
Mejor hormiga generación 1 :fitness 40

(progn3 (SIC avanza (SIC (SIC izquierda derecha) avanza)) (progn3 (progn2 avanza (SIC avanza derecha)) (SIC avanza (SIC izquierda derecha)) avanza) (progn2 avanza izquierda))



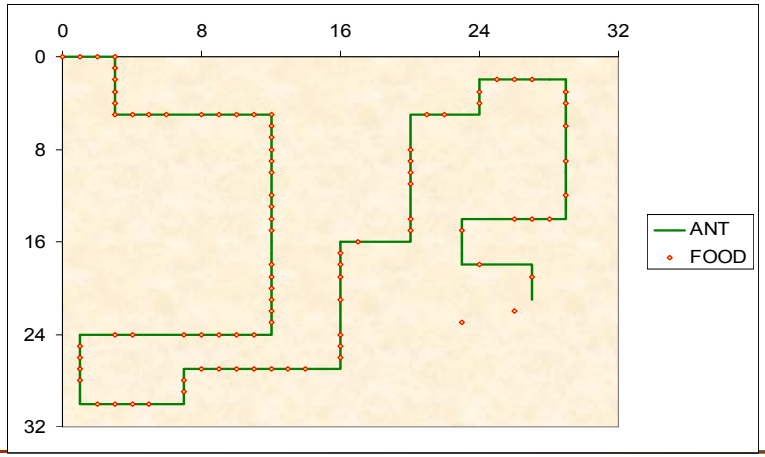
Mejor hormiga generación 10 :fitness 58

(progn3 (SIC avanza (SIC (SIC izquierda derecha) avanza)) (progn3 (progn2 avanza (SIC avanza derecha)) (SIC avanza (SIC izquierda derecha)) avanza) (progn3 (SIC izquierda derecha) (SIC (SIC avanza derecha) (SIC derecha derecha)) avanza))



Mejor hormiga generación 15 :fitness 88

(progn3 (SIC izquierda derecha) (SIC izquierda derecha) (progn3 (SIC izquierda derecha) (SIC (SIC avanza derecha) (SIC derecha derecha)) avanza))



Mejor hormiga generación 45 :fitness 90

(progn3 (SIC (progn3 (progn2 izquierda derecha) avanza derecha) derecha) (SIC izquierda derecha) (progn3 (SIC izquierda derecha) (SIC (SIC (SIC avanza derecha) (SIC derecha derecha)) (SIC derecha derecha)) avanza))