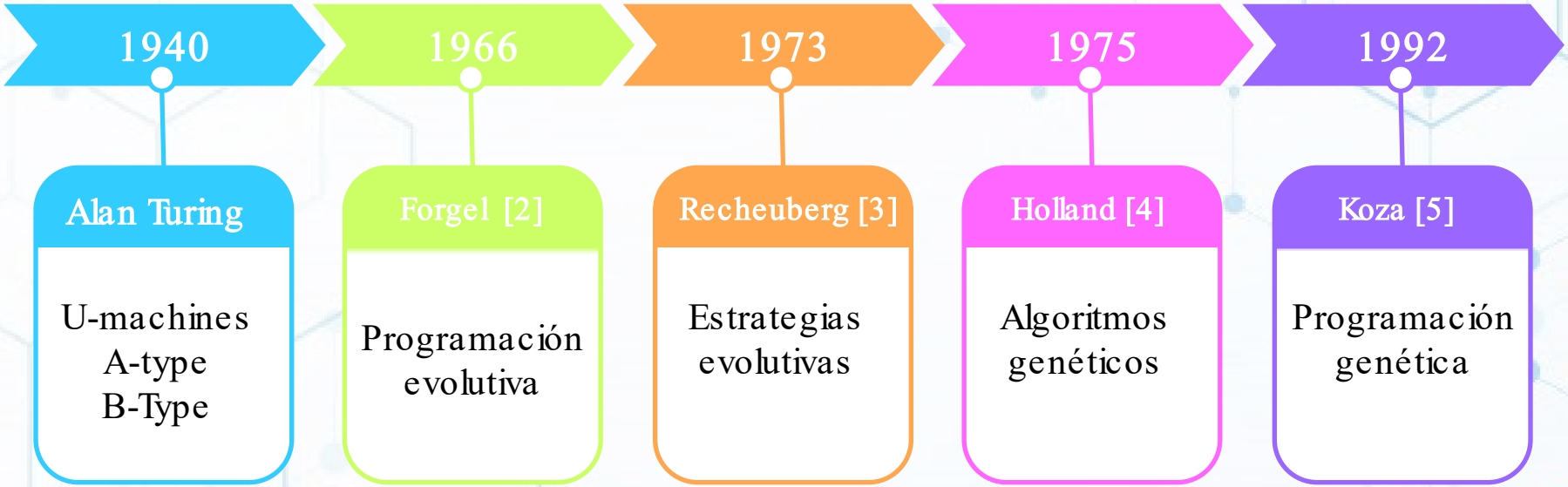
Algoritmos Bioinspirados

Unidad III : Cómputo Evolutivo – Programación Genética

Programa: Ingeniería en Inteligencia Artificial

Breve Reseña Histórica



Introducción

Uno de los desafíos centrales de las ciencias de la computación es lograr que una computadora haga lo que debe hacerse, sin decirle cómo hacerlo, es decir resolver problemas de forma automática.

En su plática de 1983 [2], Arthur Samuel dijo:

"to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence"

De hecho, lograr que las máquinas produzcan resultados similares a los humanos es la razón de la existencia de los campos de la inteligencia artificial y el aprendizaje automático.

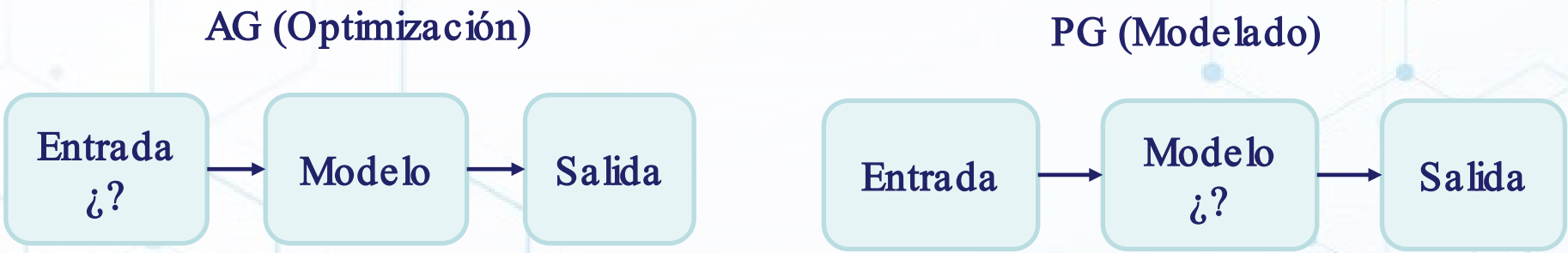
[2]. A.L.Samuel. (1983). AI: where it has been and where it is going. In: IJCAI, pp 1152–1157.

Programación Genética

La programación genética (PG) es una técnica de computación evolutiva que resuelve problemas automáticamente sin requerir que el usuario conozca o especifique la forma o estructura de la solución de antemano [3].

La programación genética logra este objetivo de programación automática (a veces también llamada síntesis de programa o inducción de programa) mediante la reproducción genética de una población de programas de computadora utilizando los principios de la selección natural darwiniana y operaciones inspiradas biológicamente [4].

La mayoría de los demás AG son para encontrar alguna entrada que obtenga la mayor ganancia, mientras que la PG se usa para buscar modelos con el máximo ajuste [4].



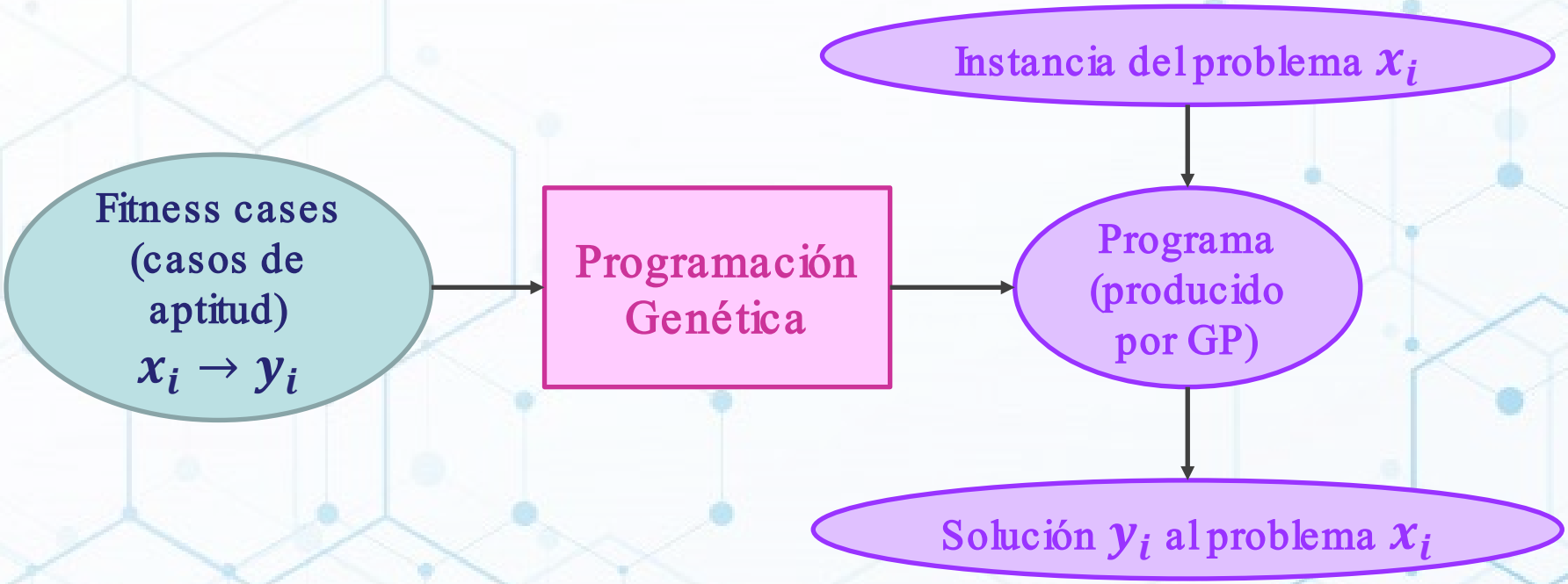
Los modelos, representados como árboles sintácticos, se tratan como individuos, y su aptitud es la calidad del modelo que se maximizará [5].

En el nivel más abstracto, GP es un método sistemático e independiente del área de conocimiento para hacer que las computadoras resuelvan problemas automáticamente a partir de una declaración de alto nivel de lo que se debe hacer [3].

En PG una población de "programas de computadoras" debe evolucionarse.

[3]. Poli, R., Langdon, W.B. & McPhee, N.F. (2008). A Field Guide to Genetic Programming. Lulu Enterprises UK Ltd.

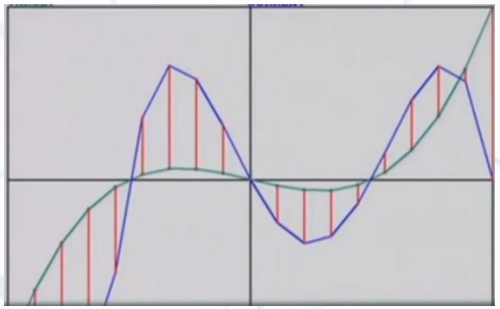
PG evoluciona un programa para resolver instancias de una clase de problema. La solución encontrada por PG es un programa que resuelve muchas instancias de problemas.

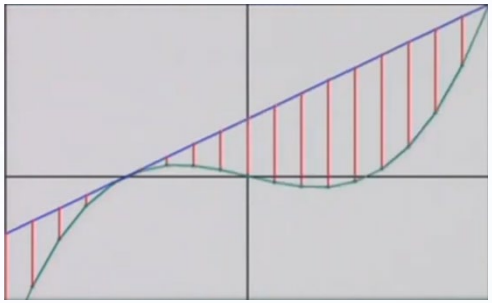


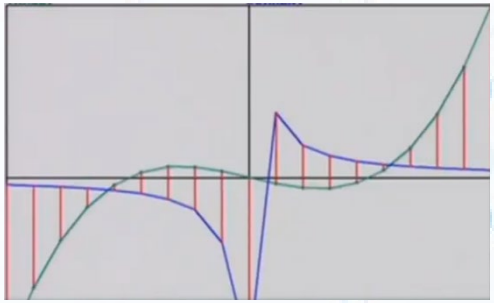
Ejemplo 1: Regresión simbólica. Dado un conjunto de valores de la forma (x, y), encontrar la función que los genera.

x	-7	-5	-3	0	1	3	5	7
y	-2	-0.2	0.2	0	-0.1	-0.3	0.3	1.5

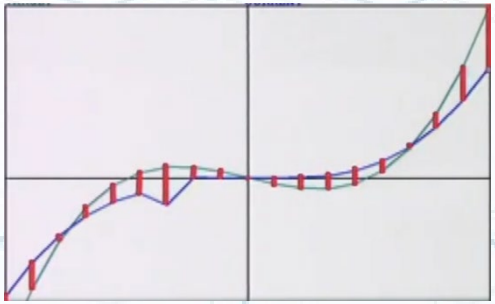
Población inicial



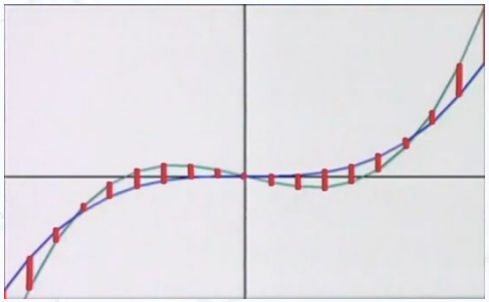




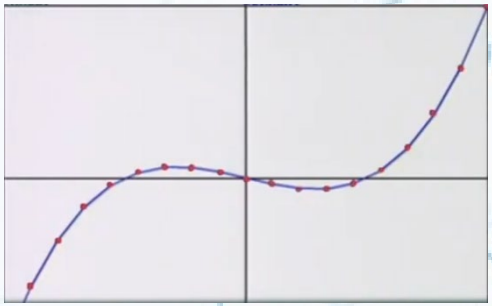
Mejor en Generación 3



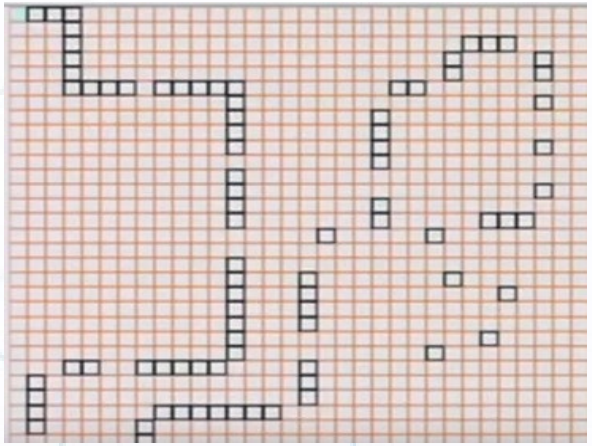
Mejor en Generación 5

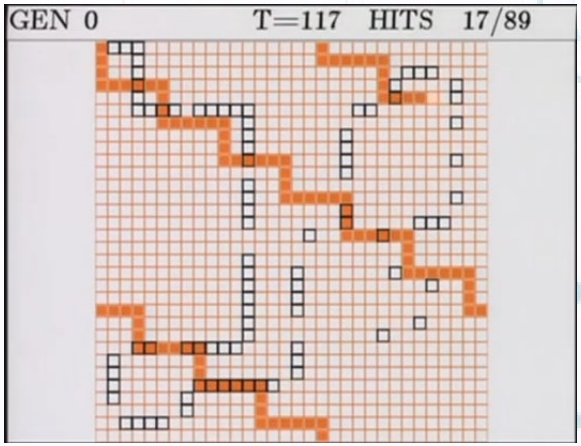


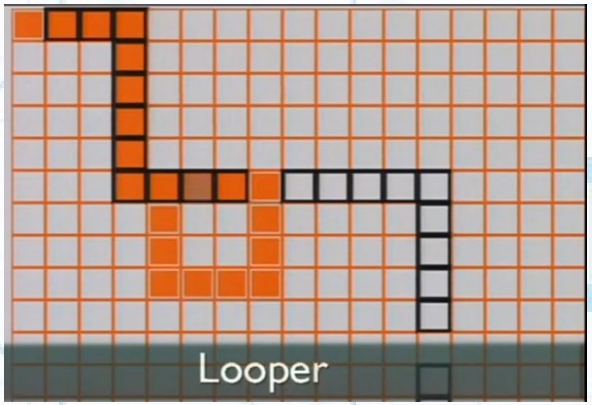
Solución Final

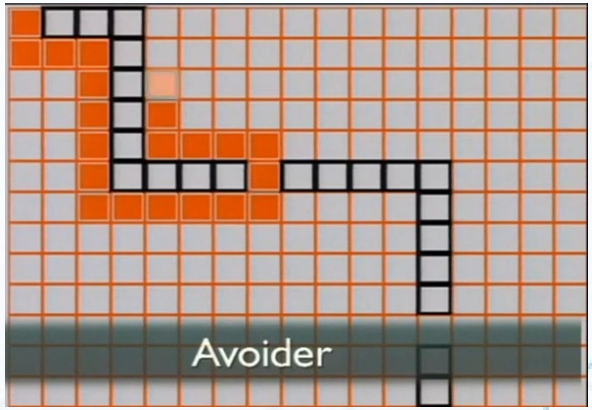


Ejemplo 2: Problema de la hormiga artificial. En este problema la comida se encuentra en un camino irregular. La aptitud es la cantidad de comida ingerida por la hormiga

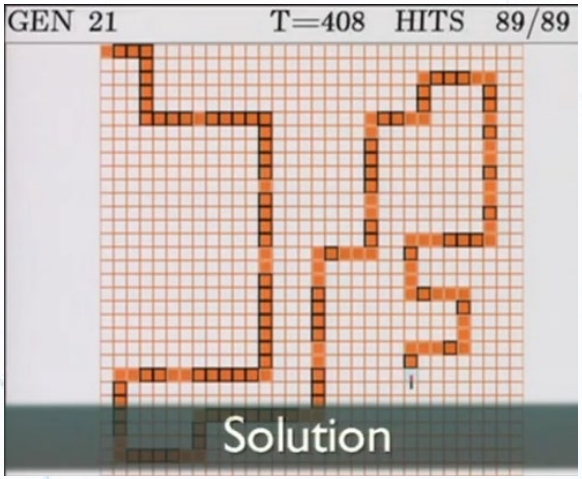


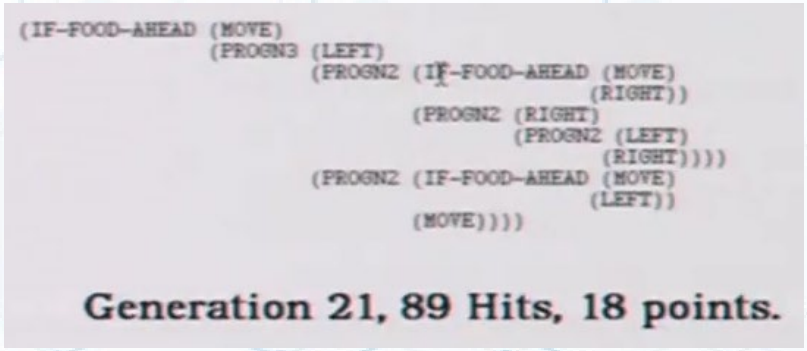


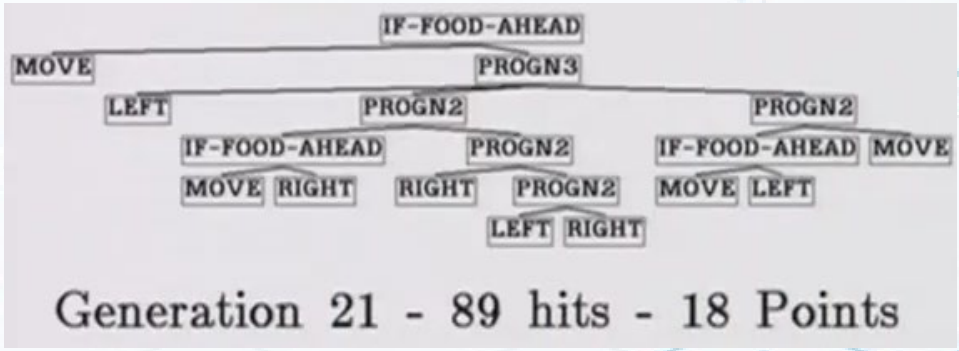




Ejemplo 2: Problema de la hormiga artificial. En este problema la comida se encuentra en un camino irregular. La aptitud es la cantidad de comida ingerida por la hormiga







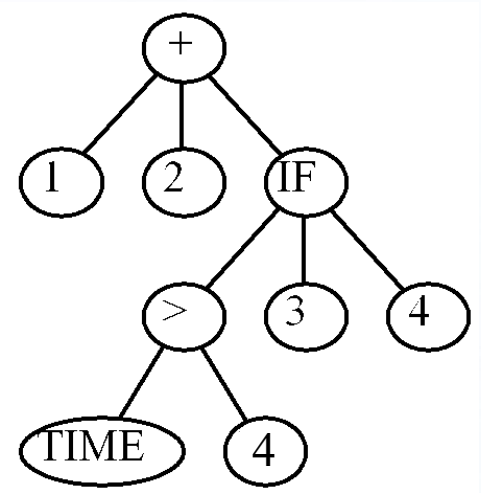
Ejemplo 3: Un programa en C.

```
int foo (int time)
{
  if (time > 4)
     temp1 = 3;
  else
     temp1 = 4;
  temp2 = temp1 + 1 + 2;
  return (temp2);
}
```

Entradas y salida del programa

Tiempo	0	1	2	3	4	5	6	7
Salida	7	7	7	7	7	6	6	6

Árbol sintáctico

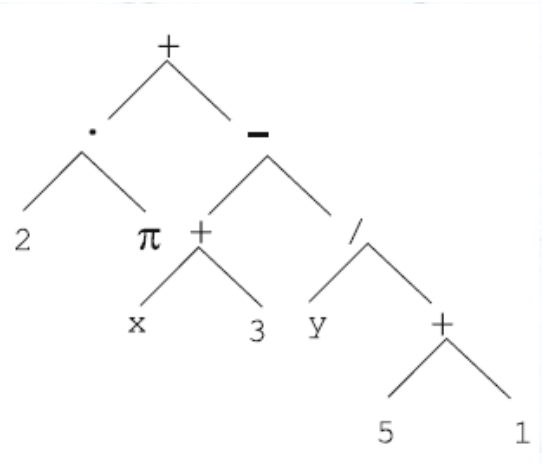


(+ 1 2 (IF (> TIME 10) 3 4))

Los árboles sintácticos usados en PG como cromosomas capturan expresiones en una sintaxis formal dada. Según el problema en cuestión, esto puede ser la sintaxis de expresiones aritméticas, expresiones lógicas o código escrito en un lenguaje de programación.

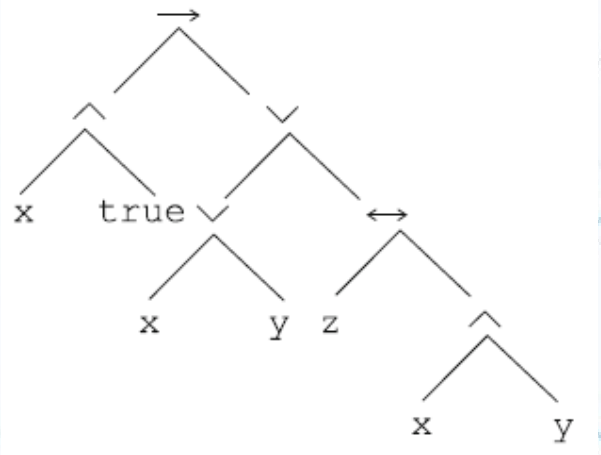
Expresión matemática

$$2 \cdot \pi + ((x + 3) - \frac{y}{5 + 1})$$



Expresión lógica

$$(x \wedge true) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$$



A pesar de que ambos paradigmas convergen en el concepto de evolución y supervivencia del más fuerte existen 3 importantes diferencias:

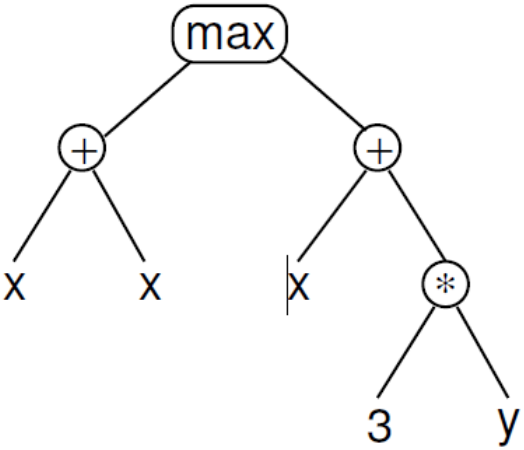
- Estructura
- Activo vs. Pasivo
- Tamaño fijo vs. variable

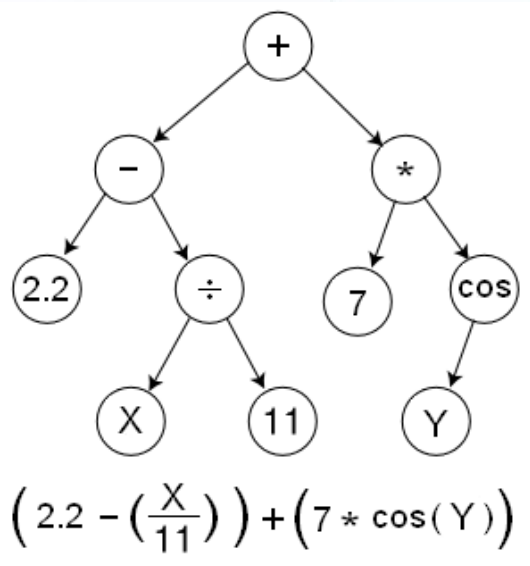
Características propuestas por el autor del paradigma

Representación	Estructura de árbol
Recombinación	Intercambios de subárboles
Mutación	Cambios aleatorios en árboles
Selección de padres	Proporcional
Selección de supervivientes	Reemplazo generacional

Estructura

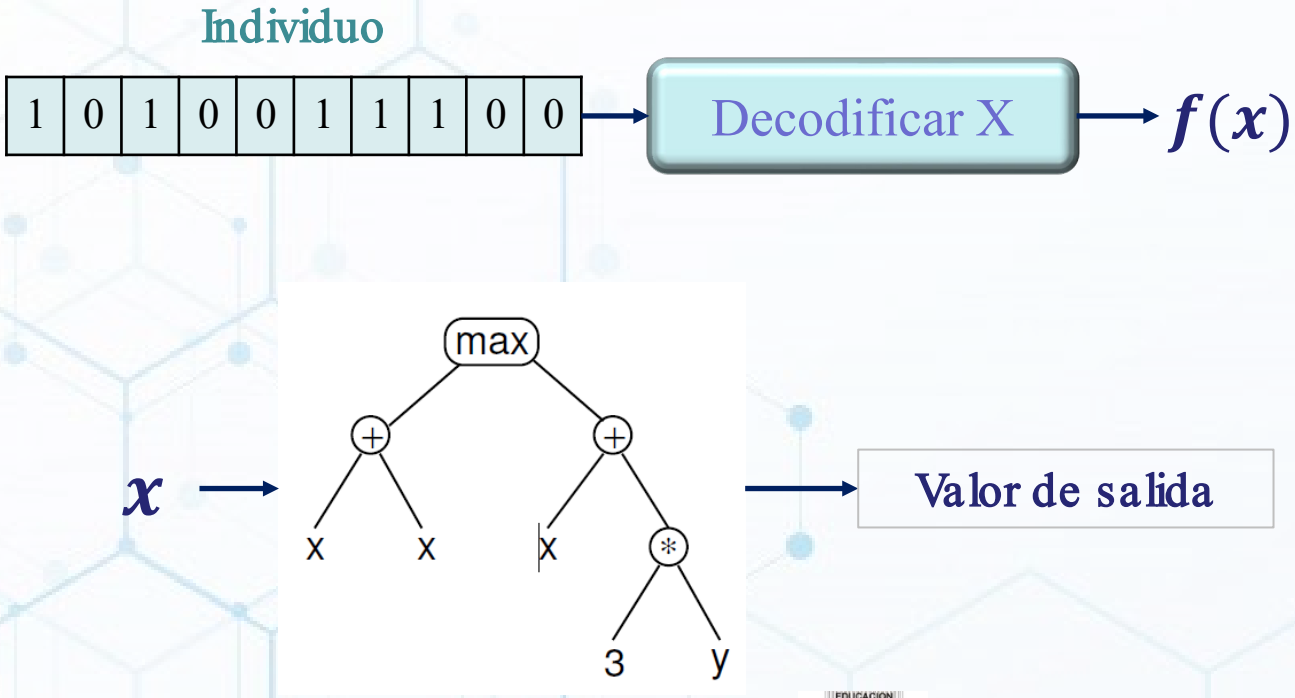
PG generalmente evoluciona estructuras de árbol mientras que AG evoluciona cadenas de números binarios o reales.





Activo vs. Pasivo

PG generalmente desarrolla programas, las soluciones se pueden ejecutar sin procesamiento posterior (estructuras activas), mientras que los GA generalmente operan en cadenas binarias codificadas (estructuras pasivas), que requieren procesamiento posterior.



Tamaño fijo vs. Tamaño variable

En los AG, la longitud de la cadena binaria y se fija antes de que comience el procedimiento de solución. Sin embargo, un árbol en PG puede variar en longitud a lo largo de la ejecución.

Individuo de tamaño fijo

1	0	1	0	0	1	1	1	0	0

Árboles de tamaño variable

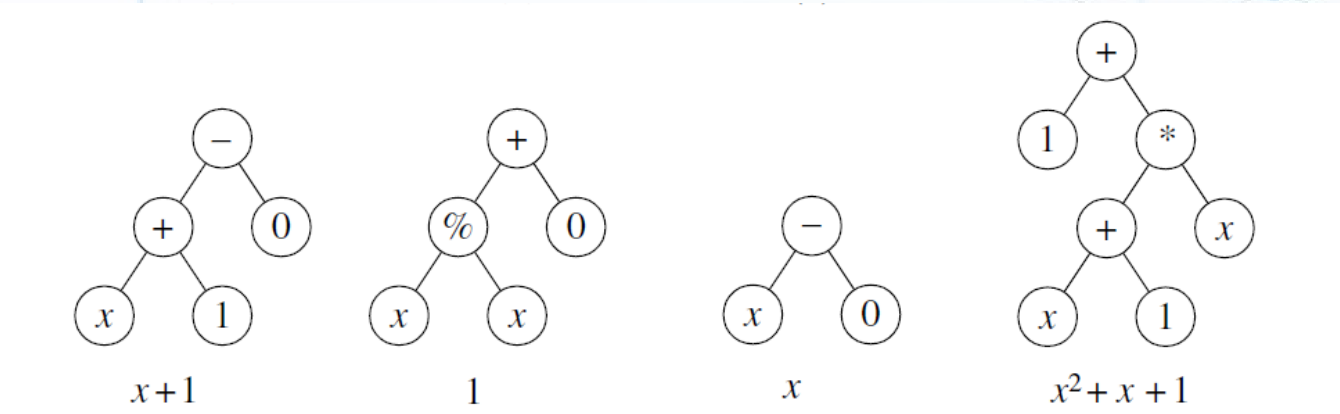
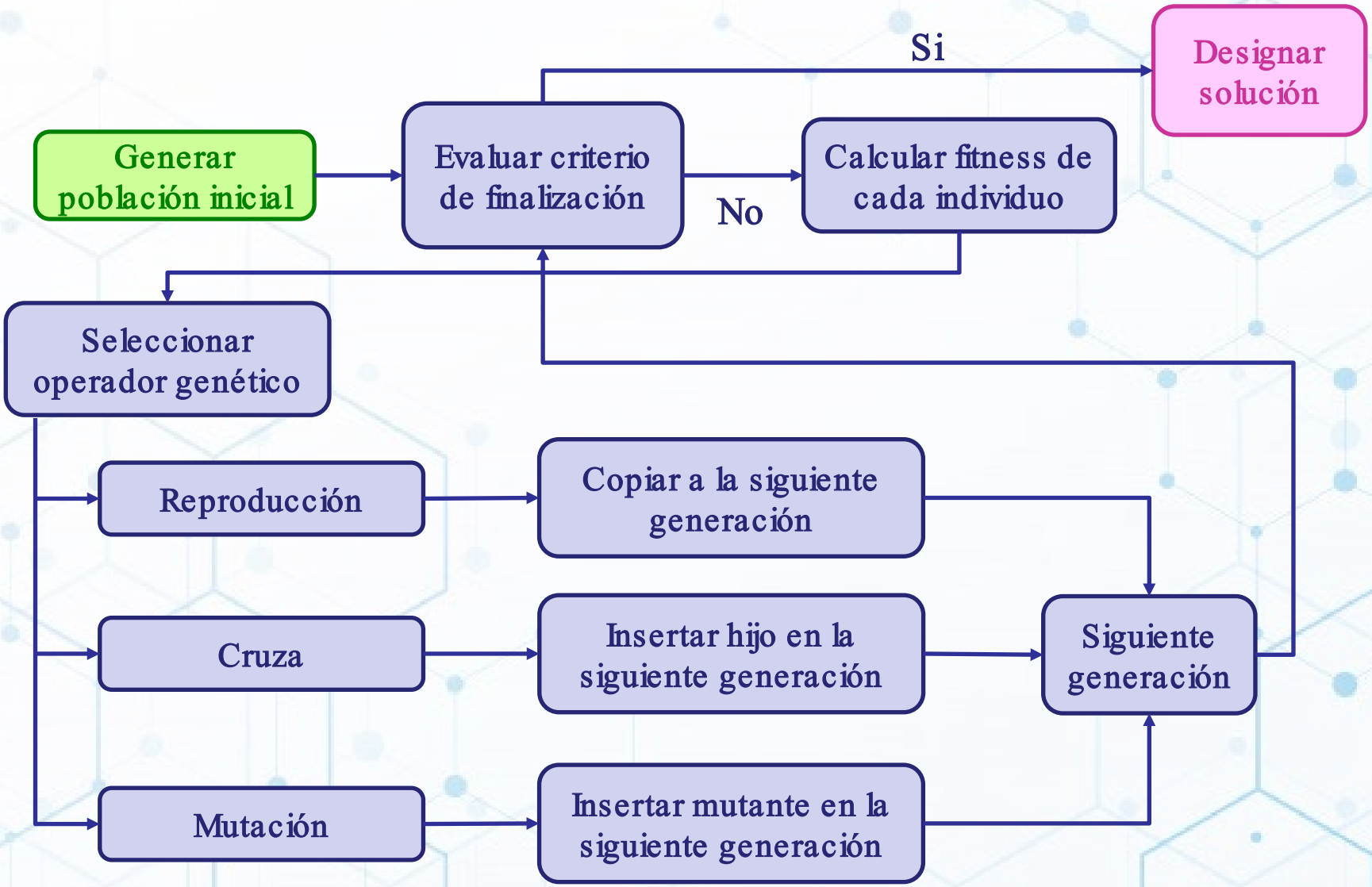


Diagrama de Flujo de la PG



Los pasos en PG se dividen en:

- Pasos preparatorios.
- Pasos de ejecución.

Pasos preparatorios

El usuario humano comunica la descripción de alto nivel del problema al sistema de programación genética realizando ciertos pasos preparatorios bien definidos. Estos 5 pasos preparatorios requieren que el humano defina:

1. Conjunto de nodos terminales
2. Conjunto de nodos función
3. Medida de aptitud
4. Parámetros de control
5. Criterio de terminación

Pasos de ejecución

Después de que el usuario haya realizado los pasos preparatorios para un problema, se puede iniciar la ejecución de la programación genética. Los pasos de ejecución son:

1. Cree aleatoriamente una población inicial de individuos (programas, árboles) compuestos por las funciones y terminales disponibles.
2. Repetir hasta que se cumpla el criterio de finalización:
   i. Ejecute cada programa en la población y determine su aptitud
   ii. Seleccione 1 o 2 individuos de la población con una probabilidad basada en la aptitud para participar en una operación genética
   iii. Crear nuevos individuos aplicando 1 operador genético: reproducción, cruza o mutación.
3. Una vez que se cumple el criterio de finalización, el mejor individuo puede ser una solución (solución aproximada) al problema.

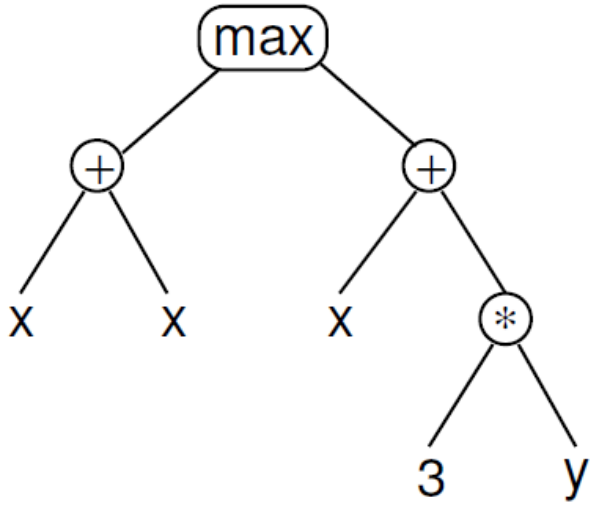
Representación

En PG, los programas generalmente se expresan como árboles de sintaxis en lugar de líneas de código.

Inicialización:

- Definir una profundidad máxima inicial D_{max}.
- Definir un conjunto de funciones y un conjunto de terminales válidos.

El conjunto de estructuras posibles en PG es el conjunto de todas las composiciones posibles de funciones que se pueden realizare recursivamente a partir del conjunto $F = \{f_1, f_2, \ldots, f_{N_{func}}\}$ de funciones y el conjunto $T = \{t_1, t_2, \ldots, t_{N_{term}}\}$ de terminales. Cada función particular f_i toma un número específico $z(f_i)$ de argumentos . Función f_i tiene aridad $z(f_i)$.



$$\max(x + x, x + 3 * y)$$

$$\max((+xx)(+x(* 3y)))$$

Es común la representación prefija.

Para este ejemplo:

$F = (+, *, max)$

$T = (x, y, 3)$

Estos 2 conjuntos forman el conjunto de primitivas de un sistema PG.

El conjunto de las funciones puede incluir:

- Operaciones aritméticas (+, -, *, etc.),
- Funciones matemáticas (como sin, cos, exp y log),
- Operaciones booleanas (como AND, OR, NOT),
- Operadores condicionales (como If-Then-Else),
- Funciones que provocan la iteración (como Do-Until),
- Funciones que causan recursividad
- Cualquier otra función específica del dominio que pueda definirse.

Los terminales suelen ser variables (que pueden representar las entradas, sensores, detectores o variables de estado de algún sistema) o constantes (un número o la constante booleana NIL).

Ocasionalmente, los terminales son funciones que no toman argumentos explícitos, la funcionalidad real de tales funciones radica en sus efectos secundarios sobre el estado del sistema (ejemplo de la hormiga artificial).

Delimitar nuestro espacio de búsqueda

El espacio de búsqueda para la programación genética es el espacio de todos los posibles programas que pueden crearse recursivamente mediante composiciones de las funciones y los terminales disponibles para el problema.

Clausura

La propiedad de cierre requiere que cada función en el conjunto de funciones pueda aceptar, como argumento, cualquier valor y tipo de dato que devuelva cualquier función en el conjunto de funciones y cualquier valor y tipo de dato que pueda tomar cualquier terminal en el conjunto de terminales.

```
def prot_div(num,den)
# función división protegida
    if(den = 0):
        return  -1  # undefined
    else:
        return num/den
```

```
def prot_sqrt(var)
# función raiz protegida
    return sqrt(abs(var))
```

La propiedad de cierre es deseable, pero no es absolutamente necesaria. Se puede optar por descartar individuos que no evalúen un resultado aceptable o asignar alguna penalización

Suficiencia

La propiedad de suficiencia requiere que el conjunto de terminales y el conjunto de funciones primitivas sean capaces de expresar una solución al problema.

$$\log(x) + 2^y$$

$$T = \{+, -, *, /\}$$

← El conjunto de terminales T no es suficiente para resolver el problema.

Dependiendo del problema, este paso puede ser obvio o puede requerir una comprensión considerable.

En algunos dominios, la tarea de identificar las variables para resolver el problema puede ser virtualmente imposible (resultados de una elección.)

Es el usuario quien debe proporcionar un conjunto de funciones apropiado para su problema.

Los dos primeros pasos preparatorios definen el conjunto de primitivas para un PG y, por lo tanto, el espacio de búsqueda. Esto incluye todos los programas que se pueden construir componiendo las primitivas de todas las formas posibles.

Sin embargo, en esta etapa, todavía no sabemos qué elementos o regiones de este espacio de búsqueda son buenos. Es decir, qué regiones del espacio de búsqueda incluyen programas que resuelven, o resuelven aproximadamente, el problema.

Función de aptitud

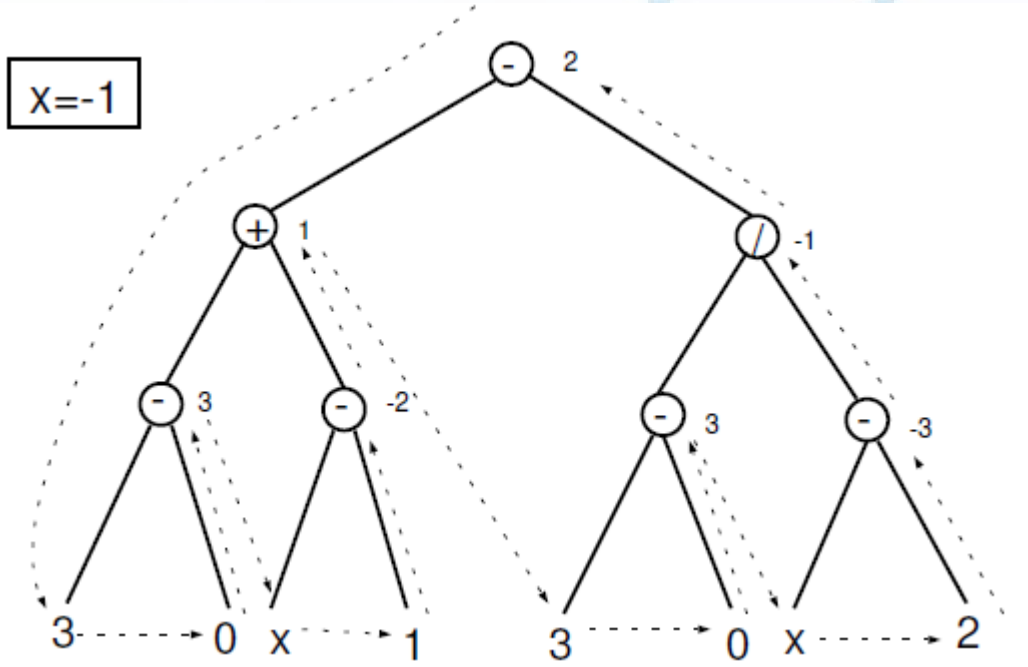
La tarea de medir que programas son buenos dentro de nuestro espacio de búsqueda es responsabilidad de la función de aptitud.

La función de aptitud en GP es diferente de la usada en otros enfoques genéticos, dado que los individuos que evolucionan son programas, por lo que para calcularla, generalmente, se requiere ejecutar todos los programas de la población.

Para esto se acostumbra usar intérpretes y no compiladores, dado el costo de construir un compilador

Interpretar un árbol (programa) significa ejecutar los nodos en el árbol en un orden que garantice que los nodos no se ejecuten antes de que se conozca el valor de sus argumentos (si los hay). Esto generalmente se hace recorriendo el árbol recursivamente comenzando desde el nodo raíz y posponiendo la evaluación de cada nodo hasta que se conozcan los valores de sus hijos (argumentos).

Este proceso recursivo de recorrido a profundidad (depth-first) se ilustra en la siguiente figura:



procedure: eval(expr)

```
1:  if expr is a list then
2:     proc = expr(1) {Non-terminal: extract root}
3:     if proc is a function then
4:        value = proc( eval(expr(2)), eval(expr(3)), ... ) {Function: evaluate
          arguments}
5:     else
6:        value = proc( expr(2), expr(3), ...) {Macro: don't evaluate argu-
          ments}
7:     end if
8:  else
9:     if expr is a variable or expr is a constant then
10:       value = expr {Terminal variable or constant: just read the value}
11:    else
12:       value = expr() {Terminal 0-arity function: execute}
13:    end if
14: end if
15: return value
```

Notes: expr is an expression in prefix notation, expr(1) represents the primitive at the root of the expression, expr(2) represents the first argument of that primitive, expr(3) represents the second argument, etc.

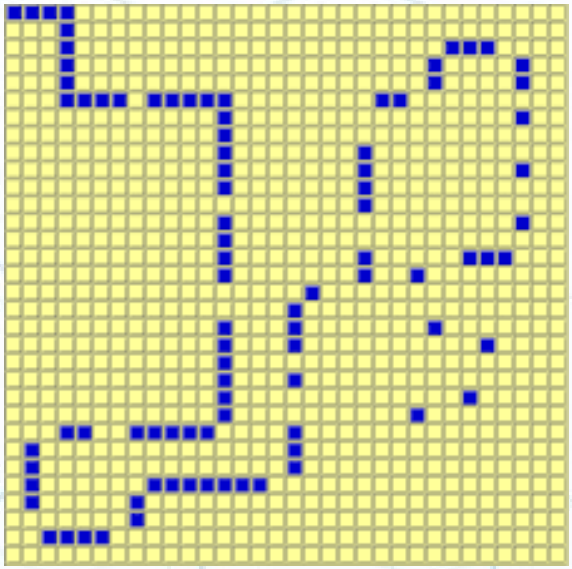
Algorithm 3.1: Interpreter for genetic programming

[3]. Poli, R., Langdon, W.B. & McPhee, N.F. (2008). A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd.

Función de aptitud

A cada programa de la población se le asigna un valor de aptitud, que representa que tan bien resuelve el problema de interés.

Aptitud bruta (*raw fitness*): es la medida de la aptitud que se establece en la terminología natural del problema mismo.



En el problema de la hormiga artificial (Santa Fe Trial) la medida de la aptitud cruda sería el número de piezas comidas por la hormiga. La aptitud va desde 0 hasta 89 en este caso.

A menudo, la aptitud bruta se calcula sobre un conjunto de casos de aptitud.

Un caso de aptitud corresponde a una situación representativa en la que se puede evaluar la capacidad de un programa para resolver un problema

Casos de aptitud

x	1	3	6	8	10
y	2	10	37	65	101

Función

$$f(x) = x^2 + 1$$

La aptitud bruta se calcula como la suma de las diferencias entre el valor esperado, especificado en los fitness cases, y los valores obtenidos por el individuo evaluado.

Función de aptitud cruda

$$f_R(i) = \sum_{j=1}^{N} |S(i,j) - y(j)|^k$$

$S(i,j)$ es el valor retornado por el individuo i sobre el caso de aptitud j.

$y(j)$ es la salida correcta del caso de aptitud j.

k es un número natural (generalmente 1 o 2)

- ✓ Los casos de aptitud suelen ser una pequeña muestra de todo el espacio del dominio.
- ✓ La elección de cuántos casos de aptitud (y cuáles) determina si una solución evolucionada generalizará o no sobre todo el dominio del problema.
- ✓ Balance entre del conocimiento sobre el problema y las consideraciones de rendimiento.

Aptitud estandarizada: es una reformulación de la aptitud bruta. Para problemas donde los valores más pequeños de aptitud son los mejores, la aptitud estandarizada es igual a la aptitud bruta, es decir $f_S = f_R$. Para problemas donde se buscan valores de aptitud más altos se calcula con la siguiente expresión:

$$f_S = f_R^{max} - f_R$$

f_R es el valor de aptitud bruta del individuo.

f_R^{max} es el máximo que puede tener la aptitud bruta (se asume que se conoce).

Aptitud ajustada: puede ser calculada a partir de la aptitud estandarizada y tiene la ventaja de poner mayor énfasis en pequeñas diferencias sobre los valores de la aptitud estandarizada.

$$f_A = \frac{1}{1 + f_S}$$

Aptitud normalizada: puede ser calculada a partir de la aptitud estandarizada y tiene la ventaja de poner mayor énfasis en pequeñas diferencias sobre los valores de la aptitud estandarizada.

$$f_N(i) = \frac{f_A(i)}{\sum_{i=1}^{M} f_A(i)}$$

Parámetros de control

El paradigma de programación genética está controlado parámetros numéricos y cualitativas que seleccionan entre varias formas alternativas de ejecutar el paradigma.

Parámetros principales

- Tamaño de la población
  Valor sugerido $M = 500$
- Número de generaciones
  Valor sugerido $G = 51$
  1 generación inicial llamada generación 0 y 50 generaciones subsecuentes.

Parámetros Secundarios

- Probabilidad de cruza
  Valor sugerido $P_c = 0.9$
- Probabilidad de reproducción
  Valor sugerido $P_r = 0.1$
- Probabilidad de mutación
  Valor sugerido $P_m = 0$
- Profundidad máxima
  Valor sugerido $D_{max} = 6$

Parámetros Secundarios

- Número de casos de aptitud
- Profundidad máxima creación
  Valor sugerido $D_{create} = 17$

En muchos casos, la principal limitación en el tamaño de la población es el tiempo necesario para evaluar las aptitudes, no el espacio requerido para almacenar los individuos.

El número de casos de aptitud está limitado por la cantidad de datos de entrenamiento disponibles .

Variables cualitativas

- Método de generación población inicial
  Ramped half-and-half.
- Medida para el valor de aptitud
  Medida de aptitud ajustada.
- Método de selección
  Proporcional al valor de aptitud.
- Sobreselección
  Solo en poblaciones mayores de 500.

Sobreselección

Es posible mejorar considerablemente el rendimiento de la programación genética al permitir que los individuos más aptos tenga mayor probabilidad de ser seleccionados.

	Aptitud Normalizada (f_N)
Individuo 1	15.3
⋮	⋮
Individuo i	8.9
⋮	⋮
Individuo 1000	0.5

Grupo I: individuos que acumulen el 32% del valor de la aptitud normalizada.

Grupo II: el resto de individuos.

80% de las veces los individuos son seleccionados del grupo I

Población	$f_{N_{acum}}$
2000	16%
4000	8%
8000	4%

Bloat

A veces llamado la "supervivencia del más gordo“, es un fenómeno observado en PG por el cual el tamaño promedio de los árboles tiende a crecer durante el transcurso de una corrida.

Si el tamaño o la profundidad de la descendencia está más allá de un límite:

- ✓ La descendencia es rechazada (y se vuelve a intentar el cruce).
- ✓ La decendencia es aceptada en la población, pero se le otorga una aptitud muy baja.

Criterio de Finalización

El criterio de finalización puede incluir un número máximo de generaciones a ejecutar, así como un predicado de éxito específico del problema.

El mejor individuo al final de la ejecución se designa como resultado (mejor individuo), aunque es posible que desee devolver individuos y datos adicionales según sea necesario o apropiado para el dominio del problema.

Bibliografía

1. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., & Lanza, G. (2006). Genetic programming IV: Routine human-competitive machine intelligence, vol. 5, Springer Science & Business Media.

2. A.L.Samuel. (1983). AI: where it has been and where it is going. In: IJCAI, pp 1152–1157.

3. Poli, R., Langdon, W.B. & McPhee, N.F. (2008). A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd.

4. S. N. Sivanandam & S. N. Deepa. 2008. Introduction to Genetic Algorithms (1st. ed.). Springer Publishing Company, Incorporated.

5. A. E. Eiben and James E. Smith. 2015. Introduction to Evolutionary Computing (2nd. ed.). Springer Publishing Company, Incorporated.

6. Koza, J.R. (1994). Genetic programming as a means for programming computers by natural selection. Statistics and computing, vol. 4(2), pp. 87-112.

¡ Gracias !

Thanks !

Obrigado

Xie xie ni

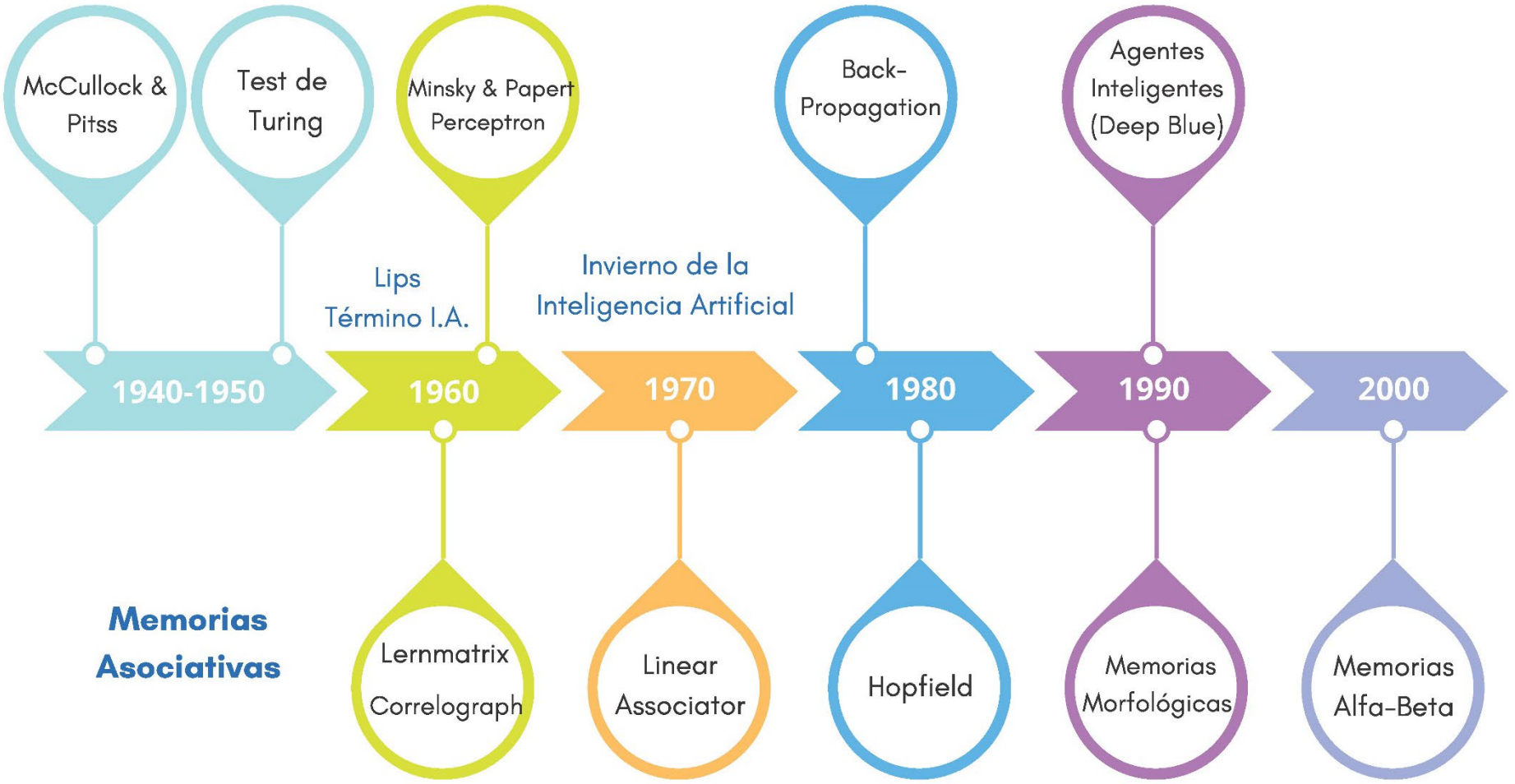
Domo arigatou

Спасибо

Merci

Grazie

Contexto Histórico



Introducción - Características

Genetic Programming Now Routinely Delivers High-Return Human-Competitive Machine Intelligence [1].

- Human-Competitive
- High-return
- Routinely
- Machine Intelligence

Koza afirma que el proceso de resolución de estos problemas puede reformularse como una búsqueda de un programa de computadora con aptitud alta en el espacio de todos los posibles programas de computadora [6].