Programación Genética

Programación Automática

- La Programación Genética es una técnica de Inteligencia Artificial que se enmarca dentro del Aprendizaje Automático y la Programación Automática
- PA = Generación automática de programas de ordenador
- Diciendo **QUÉ** se quiere que haga y no el **CÓMO**
- Es lo más **GENERAL** que un ordenador puede aprender (equivalente a una máquina de Turing).
  - Los algoritmos de aprendizaje automático pueden aprender redes de neuronas, árboles de decisión, etc., pero un programa de ordenador es más general
- Ese es el objetivo final, de momento se pueden resolver problemas limitados, pero de interés.

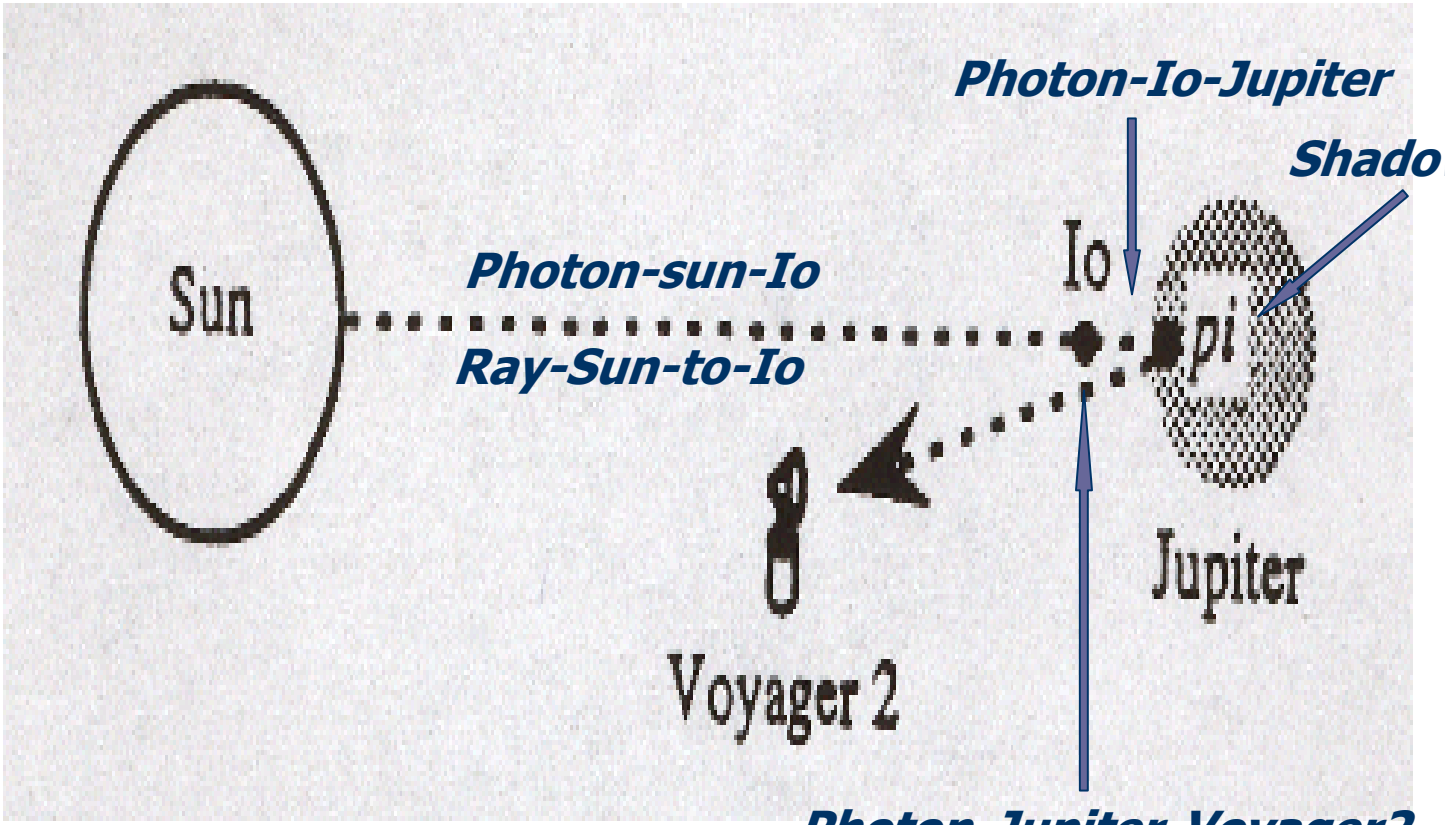
Tipos de Programación Automática

- **En Ingeniería del Software:** cualquier herramienta que facilite la generación u optimización de código (wizards paso a paso para generar interfaces gráficos, reutilización de código, clases genéricas, prototipos, templates, …)
- **En Inteligencia Artificial:**
  - **Deductiva:** Generar un programa a partir de una especificación a alto nivel (que sea más sencilla que el programa). Ej: PROLOG.
    - Ancestro(X,Y) <= padre(X,Z) y ancestro(Z,Y)
  - **Inductiva:** Generación de programas a partir de ejemplos de uso

Amphion. PA deductiva

- Ejemplo de especificación: “¿dónde está la sombra de la luna Io?”
- Se convierten a un teorema en lógica de predicados
- Que es probado por un demostrador automático de teoremas (SNARK)
- La demostración es utilizada para componer un programa FORTRAN-77 a partir de la librería de subrutinas SPICELIB
- Especificación -> teorema -> demostración -> programa

¿Dónde está la sombra de Io en Júpiter?



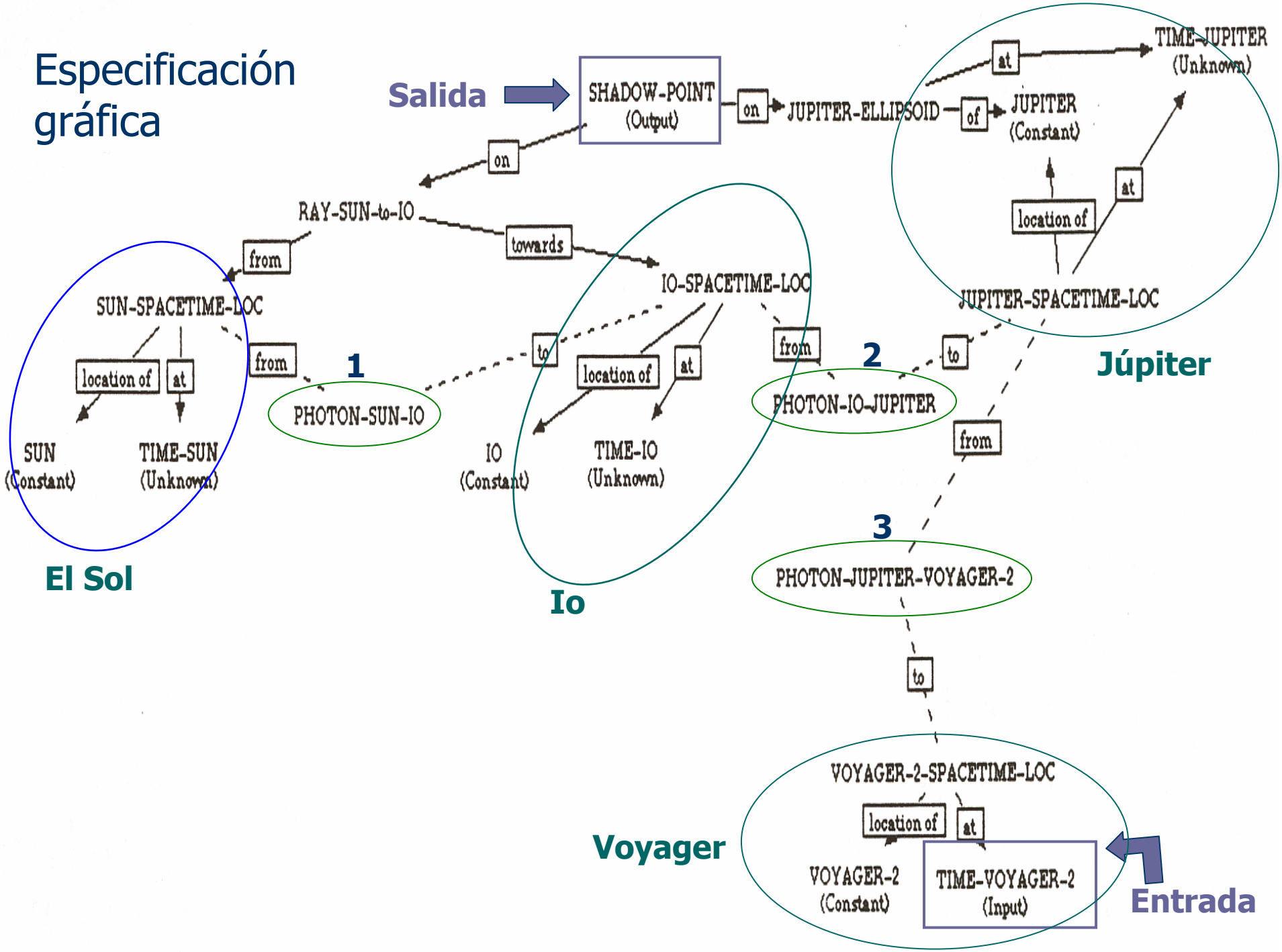
Amphion. Teorema sombra de Io

- Sea un fotón que sale del Sol, pasa por Io, rebota en el *Júpiter-Ellipsoid,* y llega a Voyager2, que está en un lugar y tiempo concretos (entrada al programa)
- ¿Existe algún *shadow-point*, que sea la intersección de *Ray-Sun-to-Io* y el *Júpiter-Ellipsoid ?*

```
(all (time-voyager-2-c)
     (find (shadow-point-c)
   (exists
   (time-sun sun-spacetime-loc time-io io-spacetime-loc
     time-jupiter jupiter-spacetime-loc time-voyager-2
     voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
     ray-sun-to-io)
   (and
    (= ray-sun-to-io
(two-points-to-ray
 (event-to-position sun-spacetime-loc)
 (event-to-position io-spacetime-loc)))
     (= jupiter-ellipsoid
(body-and-time-to-ellipsoid jupiter
    time-jupiter))
    (= shadow-point
(intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
     (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
     (lightlike? io-spacetime-loc jupiter-spacetime-loc)
     (lightlike? sun-spacetime-loc io-spacetime-loc)
     (= voyager-2-spacetime-loc
(ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
     (= jupiter-spacetime-loc
(ephemeris-object-and-time-to-event jupiter time-jupiter))
     (= io-spacetime-loc
(ephemeris-object-and-time-to-event io time-io))
     (= sun-spacetime-loc
(ephemeris-object-and-time-to-event sun time-sun))
     (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
     (= time-voyager-2
(abs ephemeris-time-to-time time-voyager-2-c)))))))
```

Teorema

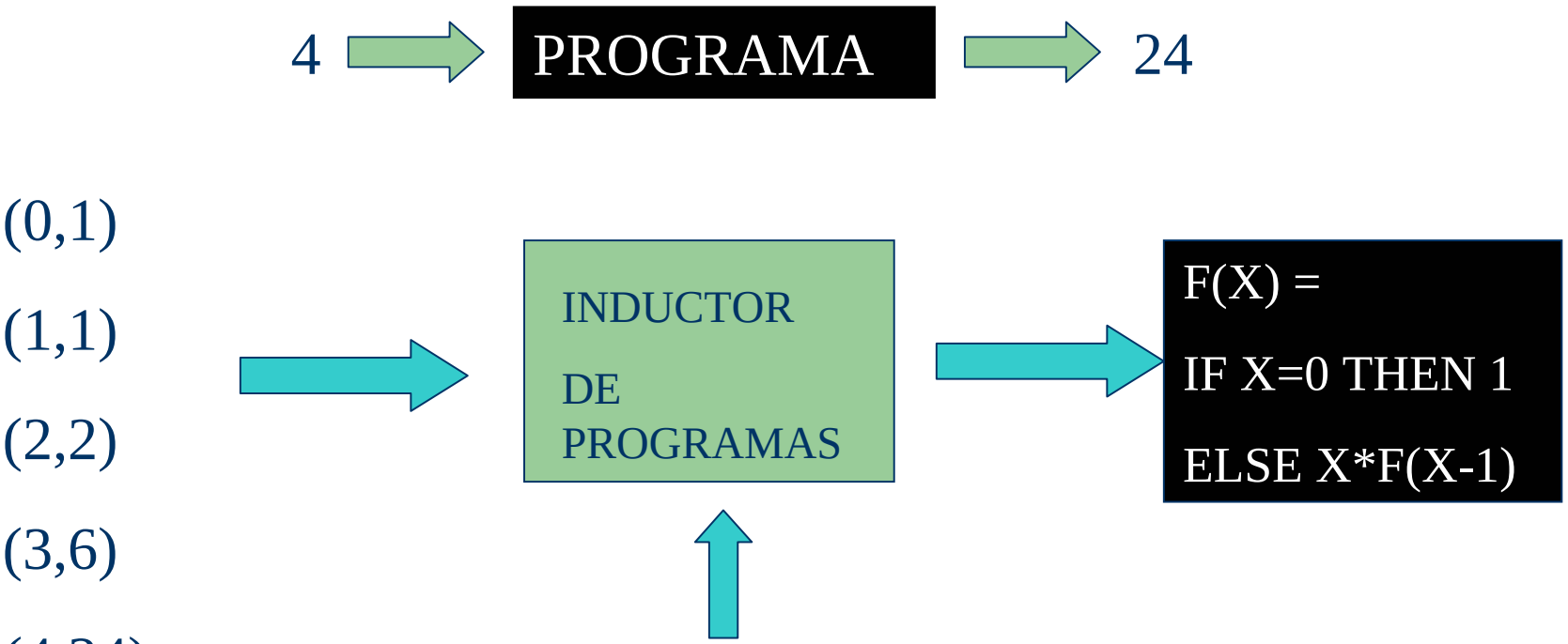
Especificación gráfica



Amphion. Programa FORTRAN

```
SUBROUTINE SHADOW ( TIMEVO, SHADOW )
DOUBLE PRECISION TIMEVO                 ...
INTEGER JUPITE
PARAMETER (JUPITE = 599)                ...
DOUBLE PRECISION RADJUP ( 3 )           ...
CALL BODVAR ( JUPITE, 'RADII', DMYO, RADJUP )
TJUPIT = SENT ( JUPITE, VOYGR2, TIMEVO )
CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY2O )
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
TIO = SENT ( IO, JUPITE, TJUPIT )
CALL FINDPV ( IO, TIO, PIO, DMY3O )
TSUN = SENT ( SUN, 10, TIO )
CALL FINDPV ( SUN, TSUN, PSUN, DMY4O )
CALL VSUB ( PIO, PSUN, DPSPI )
CALL VSUB ( PSUN, PJUPIT, DPJPS )
CALL MXV ( MJUPIT, DPSPI, XDPSPI )
CALL MXV ( MJUPIT, DPJPS, XDPJPS )
CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ), RADJUP
              RADJUP ( 3 ), P, DMY9O )
CALL VSUB ( P, PJUPIT, DPJUPP )
CALL MTXV ( MJUPIT, DPJUPP, SHADOW )
END
```

Inducción General de Programas



Representación de Programas

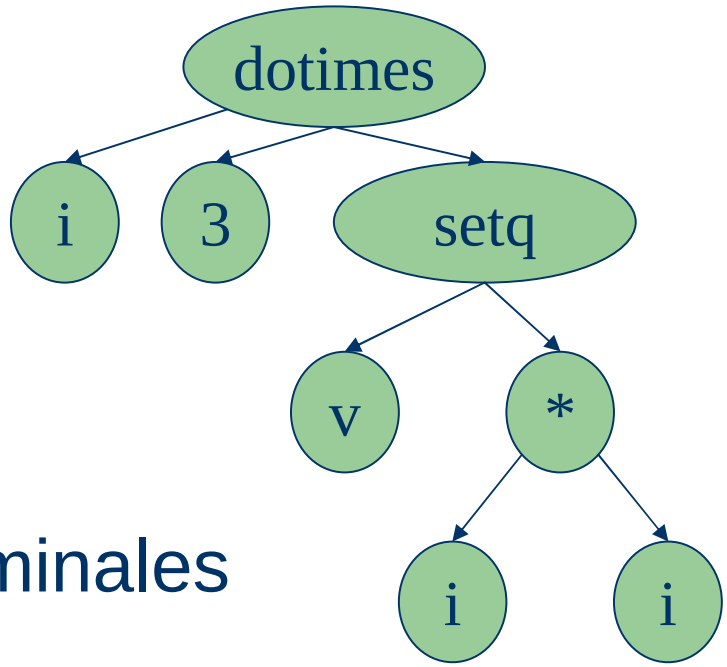
- Primera cuestión, ¿cómo representar programas?: C, Java, Prolog, lenguaje máquina, autómatas celulares, máquinas de Turing
- En Programación Genética se utilizó originalmente LISP (y todavía se usa)
- Aunque también se evolucionan programas en Java, bytecode, machinecode, PROLOG, …
- De hecho PROGEN es una librería escrita en JAVA que evoluciona programas en sintaxis LISP

LISP. Caracteristicas Fundamentales

- LISP = List Processor (lista interminable de insoportables paréntesis)
- Único tipo de datos: la lista (permite representar registros, árboles, grafos, …)
  - ‘(a b 1 2)
  - ‘(3 4 5 (a b c))
- Notación prefija: operador antes que operandos:
  - Infija: 3 + (4*2)
  - Prefija: (+ 3 (* 4 2))

LISP. Caracteristicas Fundamentales: UNIFORMIDAD

- Datos y programas se representan de la misma manera (listas o expresiones-S):
  - ‘(3 4 5 (a b c))
  - (dotimes i 3 (setq v (* i i)))
    - For(i=0;i<3;i++){v=i*i;}



- Lenguaje = Funciones + Terminales

Ejemplo: Paridad Par

- Generar un programa de ordenador que tome 10 bits y diga si el número de 1's es par:
  - Paridad-par(1,0,0,0,1,0,0,1,1) -> TRUE
- En términos de:
  - Funciones: AND, OR, NAND, NOR, NOT
  - Terminales: D0, D1, D2, ..., D9
- **QUÉ**: casos de prueba entrada/salida (2^10 = 1024 casos)

Casos de prueba paridad-par 10 bits (1024 casos)

D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	SALIDA
0	0	0	0	0	0	0	0	0	0	TRUE
0	0	0	0	0	0	0	0	0	1	FALSE
0	0	0	0	0	0	0	0	1	1	TRUE
...										

Ejemplo: Estrategia para Pacman

Ejemplo: Estrategia para Pacman

- **QUÉ**: maximizar el número de puntos comidos (hasta que los come todos o los fantasmas cazan al Pacman)
- Lenguaje:
  - Funciones:
    - si-obstaculo, si-punto, si-punto-gordo, si-fantasma, (son del tipo if-then-else)
    - secuencia2, secuencia3, secuencia4, ...
  - Terminales: avanzar, girar-izquierda, girar-derecha

Ejemplo de Programa en el Pacman

```
(si-fantasma
  (secuencia3 (girar-izquierda)
              (girar-izquierda)
              (avanzar))
  (si-punto-gordo
     (avanzar)
     (girar-derecha)))
```

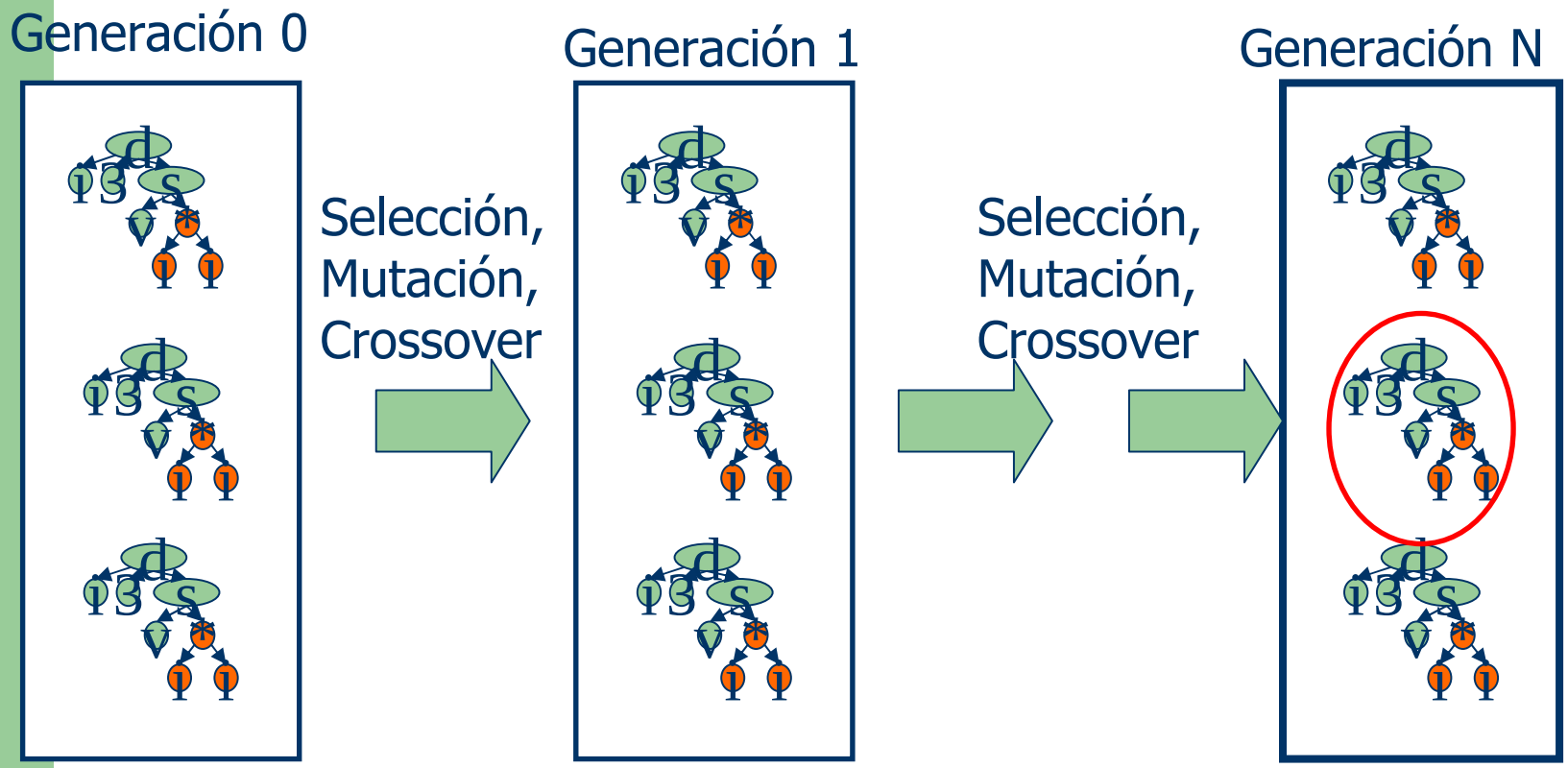
Evolución Darwiniana

- Se reproducen los individuos más capaces (**selección**)
- Los hijos no son idénticos a los padres. Cambios en el DNA debidos a mezcla, cruce, mutación (**variación**)
- El resultado es que se producen individuos adaptados a su medio
- Evolución = variación + selección
- Incremental: los cambios se van construyendo uno sobre otro

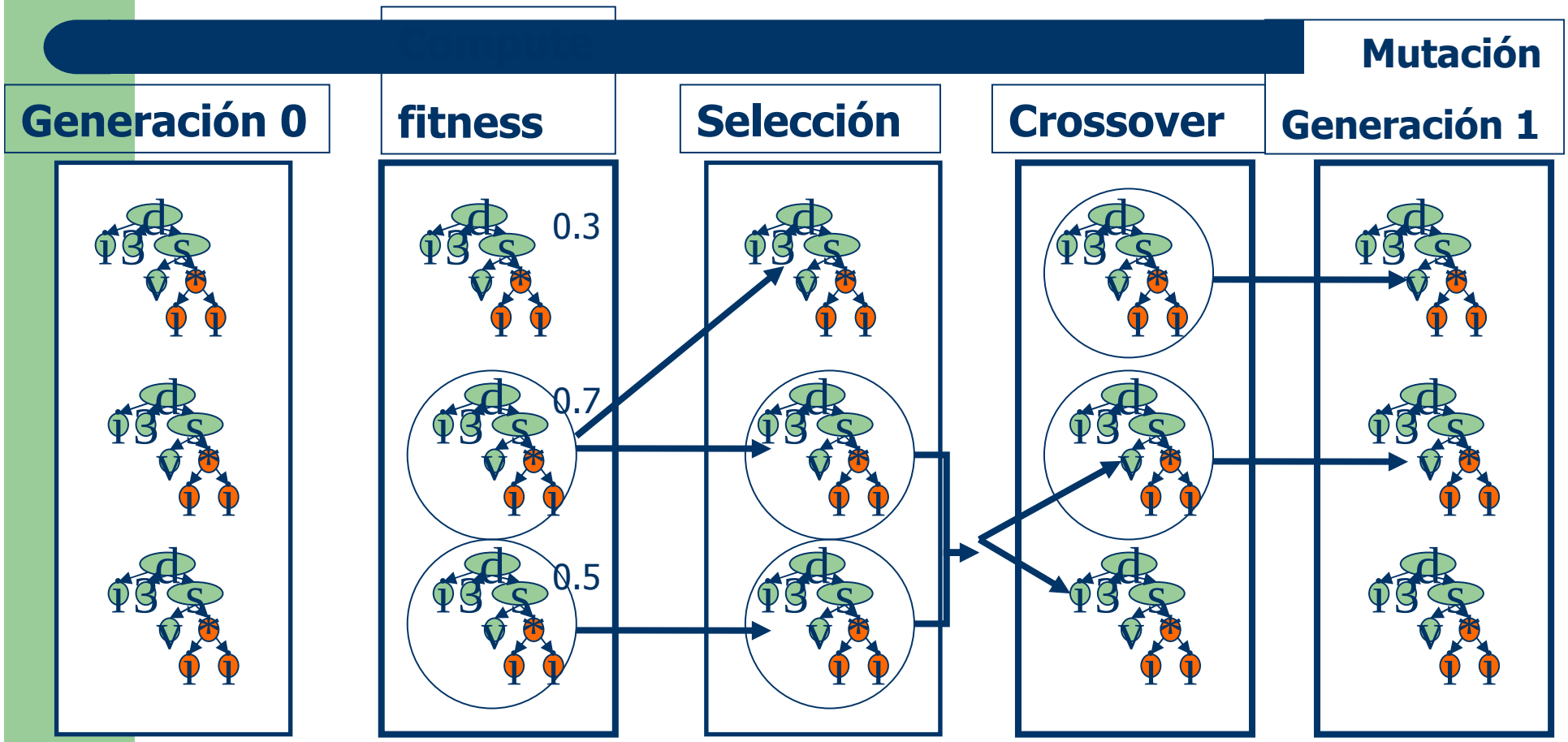
Algoritmo Programación Genética

1. Creación de una población de programas de ordenador aleatoria, utilizando funciones y terminales
2. Repetir:
   - Ejecución de todos los programas y evaluación de los mismos (función de *fitness*)
   - **Selección** de los mejores programas
   - Creación de una nueva población aplicando los operadores genéticos (**variación**) a los seleccionados
   - Volver a 2 hasta encontrar un “buen” programa

Programación Genética Generacional



De la Generación *i* a la *i+1*



Búsqueda Genética

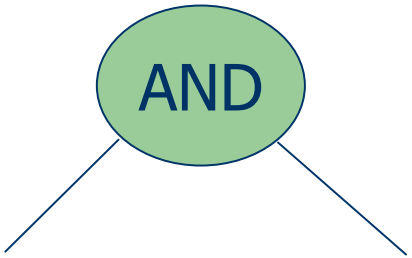
- La Programación Genética hace búsqueda heurística en el espacio de los programas de ordenador
- Es un tipo de Beam Search (con ciertas similaridades a “mejor primero” con lista ABIERTA de tamaño limitado – la población -)
- La función heurística es la función de *fitness*
- Los operadores de búsqueda son los llamados operadores genéticos (mutación y cruce)

Generación de población inicial

- Elegir una función para la raíz del árbol
- Ver la aridad de la función
- Para cada argumento de la función, generar:
  - Bien un terminal
  - Bien un subárbol
- En la práctica no se puede generar árboles más profundos que cierta constante

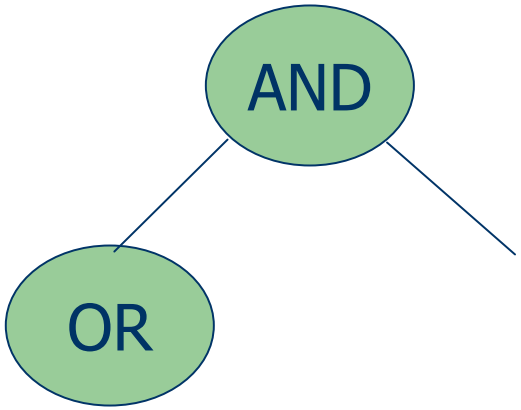
Creación Aleatoria de Individuos

- Selección aleatoria un símbolo para la raíz de:
  - {AND(1,2), OR(1,2), NAND(1,2), NOR(1,2), NOT(1)}
  - {D0, D1, D2, ..., D9}
- Crea tantas ramas como la aridad de la función



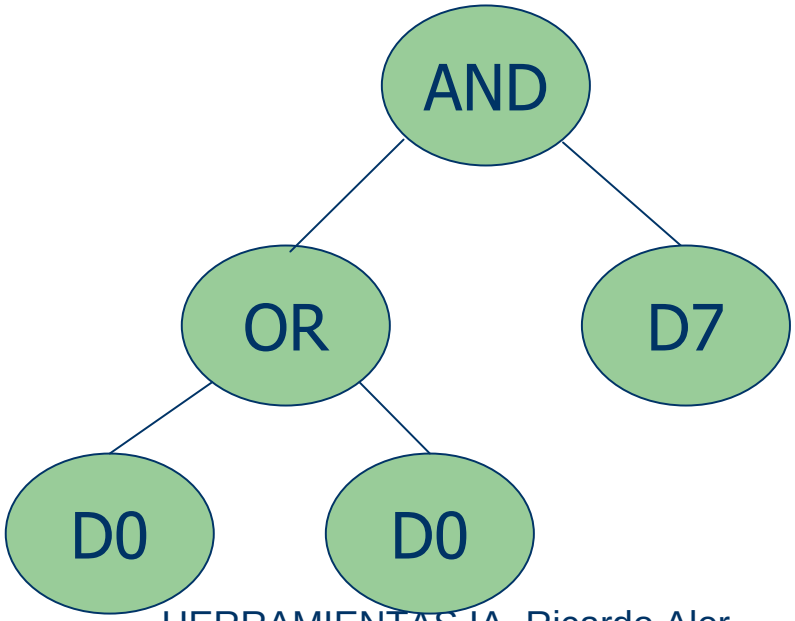
Creación Aleatoria de Individuos

- Crea tantos subárboles como ramas



Creación Aleatoria de Individuos

- Importante crear una población inicial tan diversa como sea posible
  - Diferentes estructuras de árboles
  - Diferentes profundidades de árboles



Métodos de generación de individuos

- “Full”: misma profundidad para todas las ramas del árbol
- “Grow”: profundidad variable para las ramas
- “Ramped half and half”:
  - Se generan árboles para cada posible profundidad
  - El 50% será full y el 50% grow
- Objetivo: maximizar la diversidad

Funciones y Terminales

- Funciones: aquellas funciones o macros que toman argumentos. Van en los nodos internos del árbol. Ej:

  (+ 3 4)

  (if-then-else c t e)

- Terminales (van en las hojas del árbol)
  - Funciones que no tienen argumentos. Ej: *(avanzar)*
  - Constantes: *3*, *a*, ...
  - “Ephemeral random constant” *R*: para problemas numéricos y de regresión simbólica
  - Variables de entrada: *D0*, *D1*, ...

Funciones y Terminales I

- Funciones/terminales suficientes para expresar la solución (ej: para funciones booleanas basta con *and*, *or* y *not*)
- Conviene que no haya funciones (o terminales) de sobra
- En ocasiones conviene incluir funciones potentes (para que el sistema no tenga que redescubrirlas). Ej: *sin(x)*

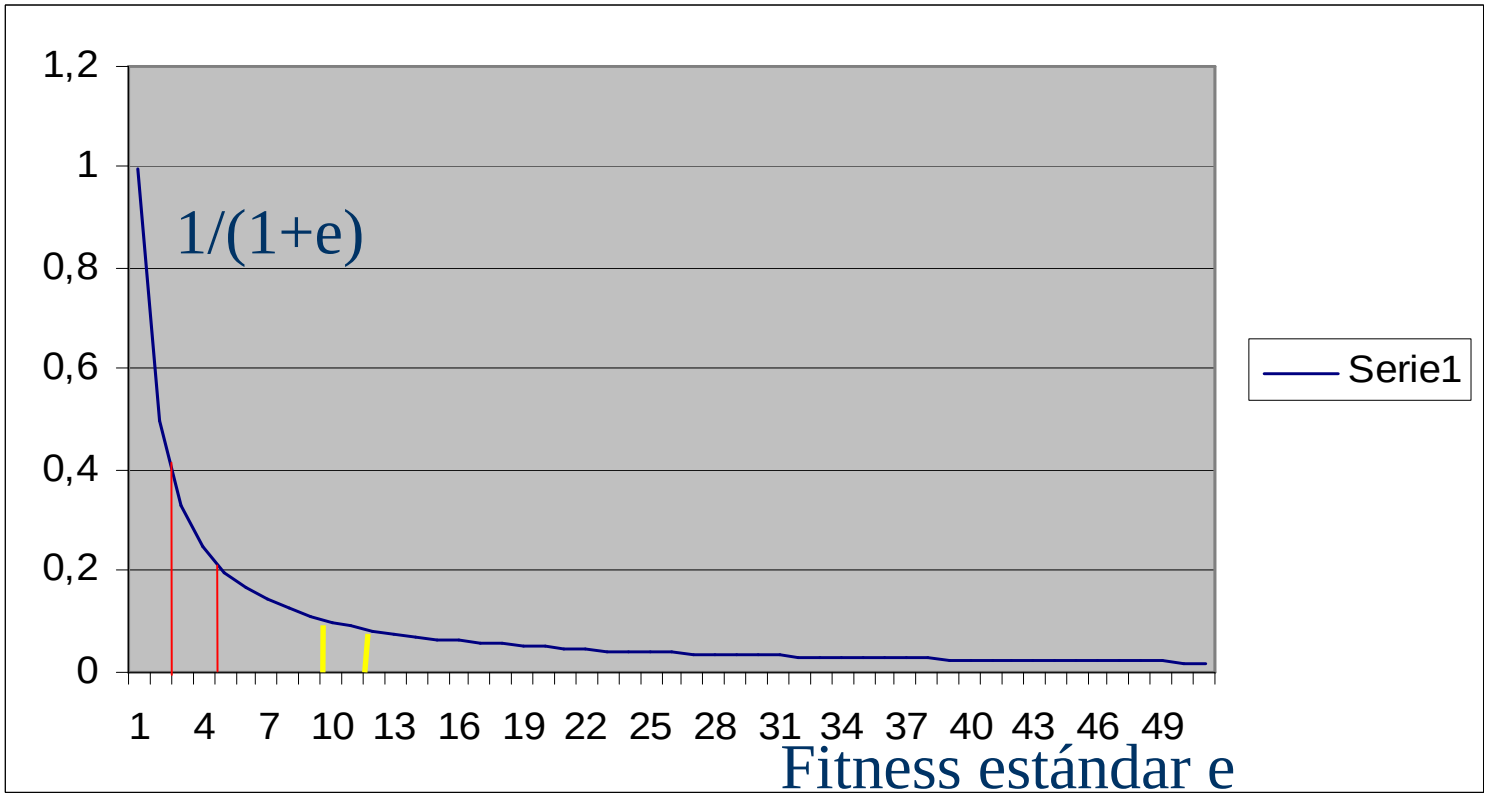
Funciones y Terminales II

- Diferenciar entre **funciones** (evalúan los argumentos. Ej: *suma*) y **macros** (no los evalúan: *if-then-else*).
- Las funciones se deben ejecutar con cualquier argumento y sin producir errores (cierre léxico o **closure**). Ej: división por cero protegida.
- En PG estándar **no** hay tipos de datos. Todas las funciones tienen que estar preparadas para recibir cualquier tipo y valor de argumento

Evaluación de Programas (*fitness*)

- Cruda (*raw*): la más natural para el problema
  - Ej: número de casos acertados o *hits* (paridad-par)
  - Ej: 0.7*puntos+ 0.3*tiempo (Pacman)
- Estándar: por convenio, PG siempre minimiza:
  - *Fitness* estándar = máxima – cruda
- Ajustada: 1/(1+estándar). Exagera *fitness* cercanas a 0. Es bueno al final de la evolución
- Normalizada o relativa: (entre cero y uno)
  - ajustada/(suma *fitnesses* ajustadas en población)

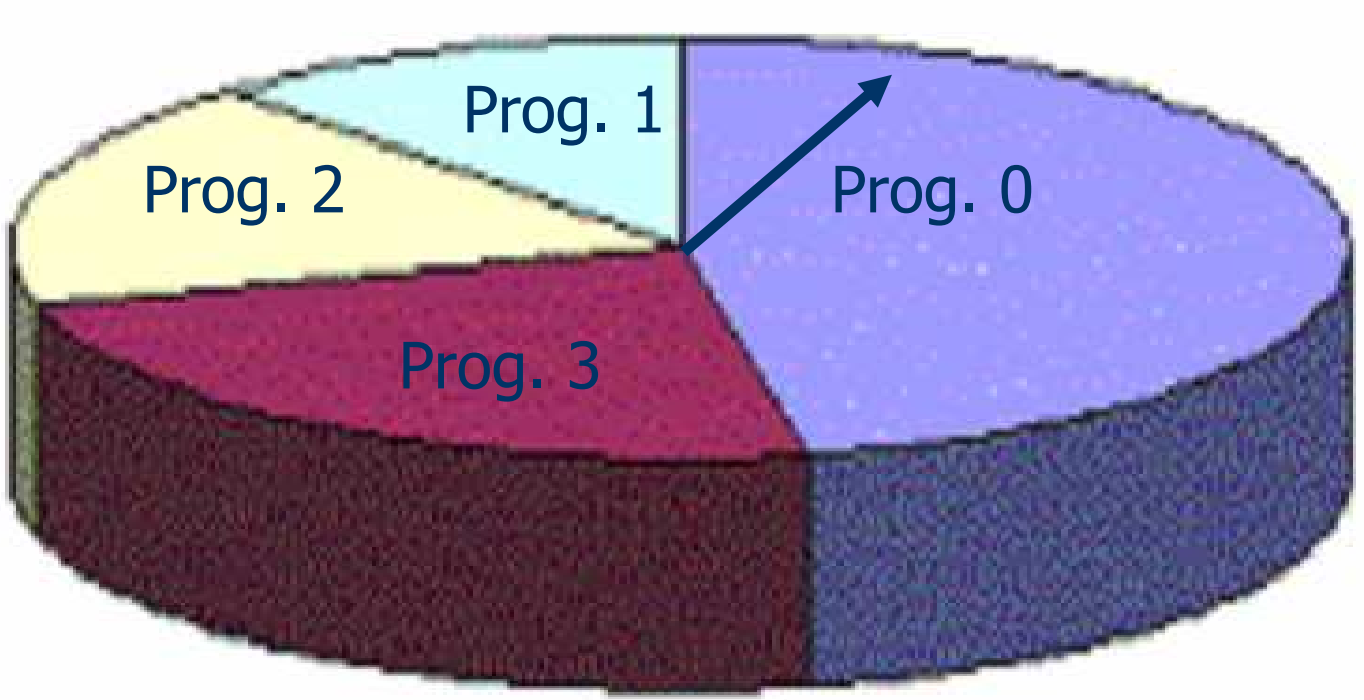
Fitness Ajustada (exagera valores próximos a cero, los mejores)



Selección Darwiniana

- Proporcional a la *fitness* (probabilística): usa *fitness* normalizada
- Torneo
  - Se eligen varios individuos y el mejor se reproduce
- “Greedy Overselection”: 80% selección del grupo I y 20% del grupo 2. Poblaciones grandes (> 1000)

Selección Proporcional a la Fitness

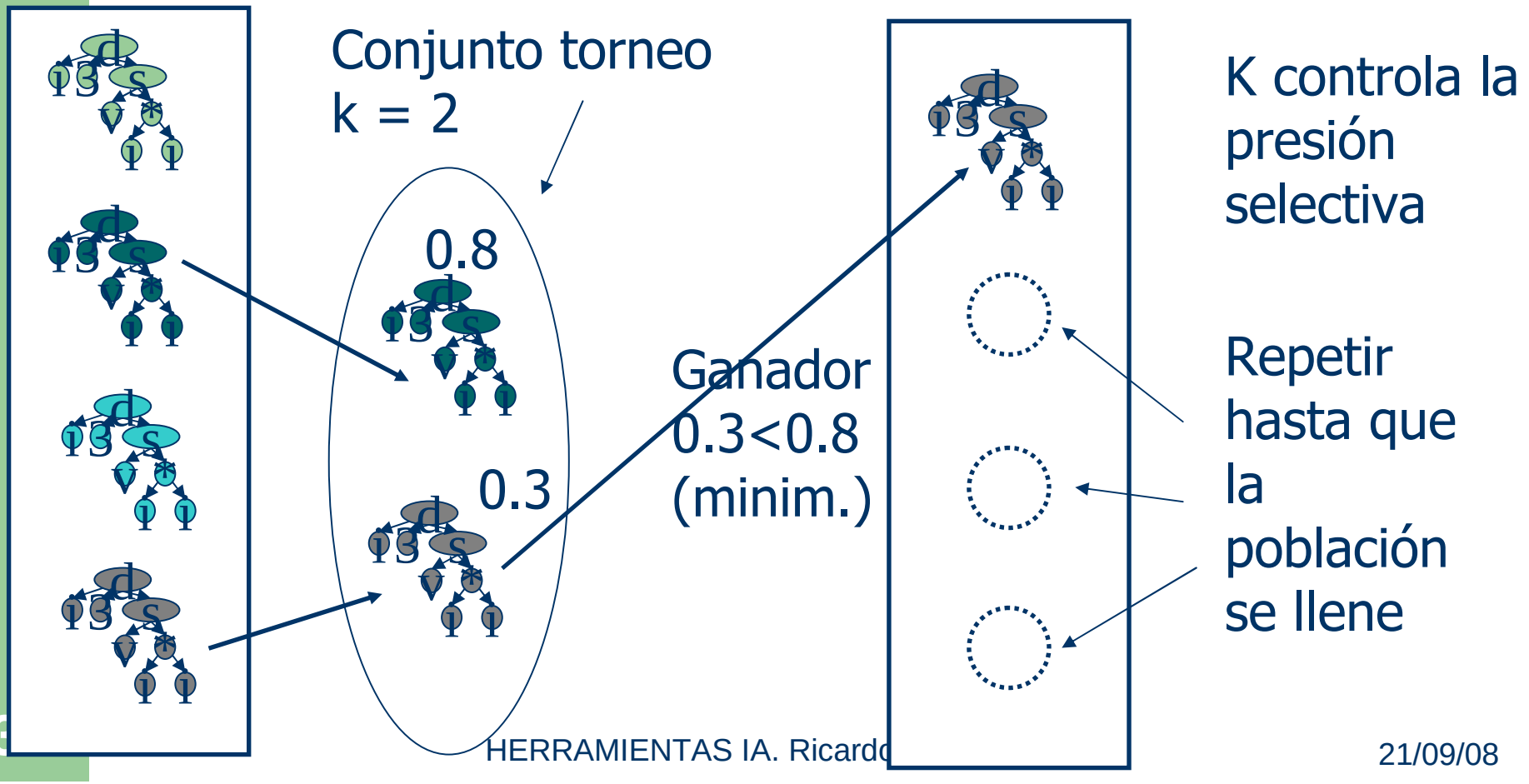


En media, los mejores individuos se seleccionan más frecuentemente (y por tanto, tienen más hijos)
Problema: **superindividuos**, convergencia prematura

Convergencia Prematura

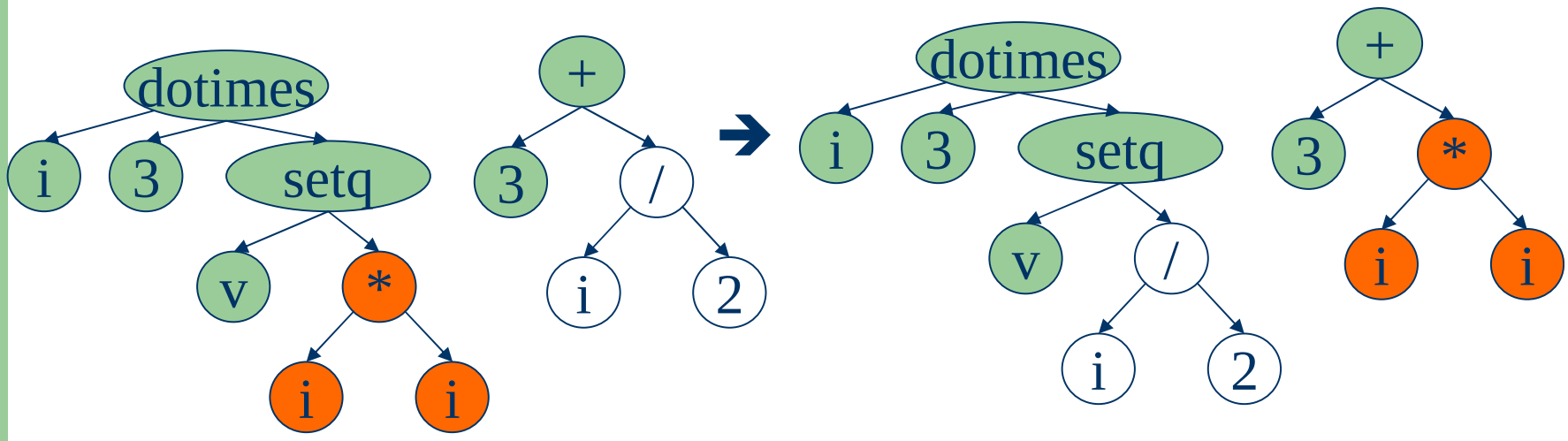
- Algunos métodos de selección convergen prematuramente
- Se llega a una población que deja de generar mejores individuos
- Normalmente esto ocurre porque los individuos son muy parecidos (no hay diversidad) y los operadores genéticos dejan de generar novedad
- La culpa suele ser de los superindividuos

Selección por Torneo



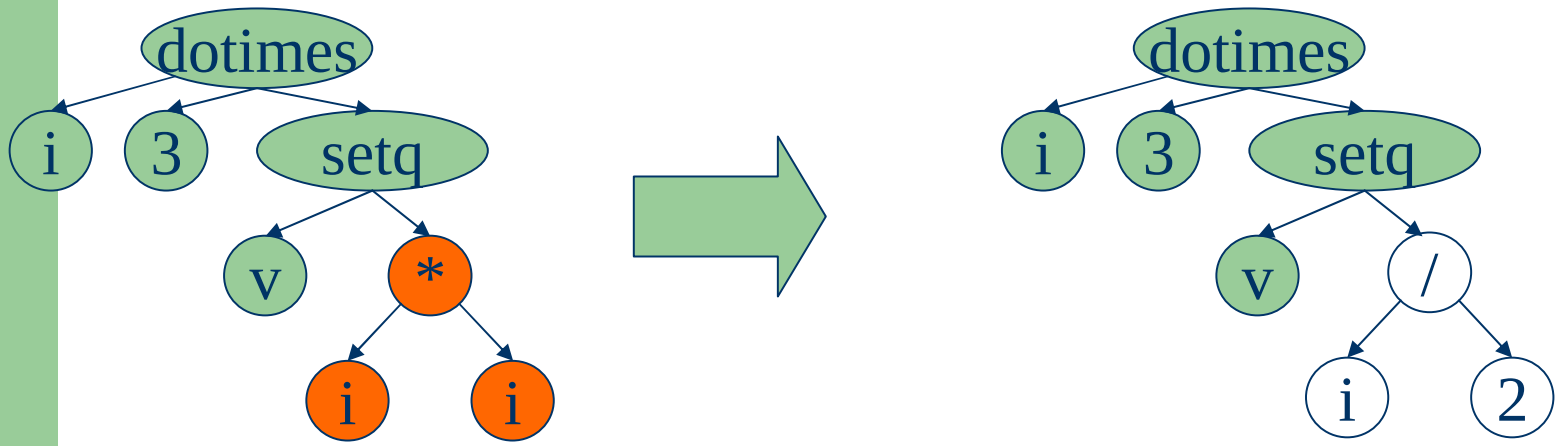
Operadores genéticos

- Reproducción
- Recombinación o cruce (crossover):



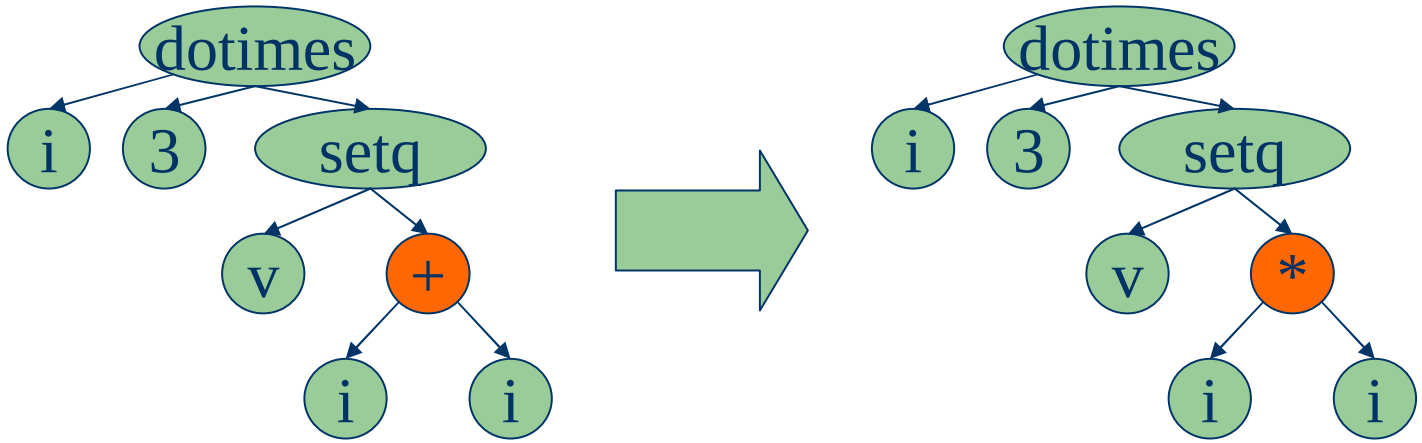
Operadores genéticos

- Mutación de subárbol



Operadores genéticos

- Mutación de punto



Parámetros de control

- Tamaño de la población (M= 500-10000)
- Máximo número de generaciones (G= 50)
- Probabilidades de recombinación, reproducción y mutación (mutación < 5%)
- Método de generación de la población inicial (*grow*, *full*, ***ramped half and half***)
- Profundidad máxima de programas iniciales
- Profundidad máxima para individuos tras recombinación.

La PG es estocástica

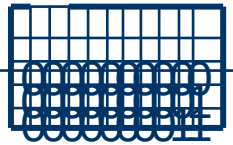
- 1 ejecución = desde la población inicial, varias iteraciones, hasta que se detiene el sistema
- Puntos de aleatoriedad:
  - Creación de la población inicial
  - Selección de los individuos
  - Selección de los operadores genéticos
  - Selección de los puntos de cruce o mutación
- No hay garantía de que una ejecución de PG tenga éxito (Convergencia prematura) → Repetir ejecuciones y quedarse con el mejor

Pasos para utilizar PG

1) Determinar los terminales (valores de entrada y constantes)
2) Determinar las funciones y macros primitivas. Programarlas (asegurar cierre/closure)
3) Determinar los parámetros (M, G, probabilidades)
4) Determinar el método para seleccionar al mejor individuo
5) Ejecutar varias veces, quedarse con el mejor, verificar posteriormente el programa

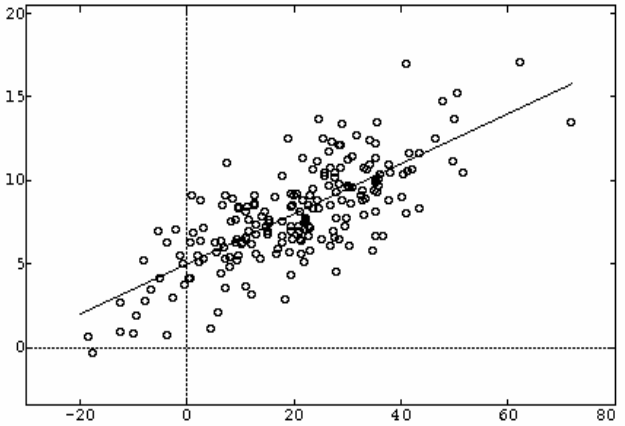
Objetivo	Encontrar un programa que juegue al **Pacman**
Terminales	(avanzar), (girar-izquierda), (girar-derecha)
Funciones	si-random-1(2), si-obstaculo(2), si-punto(2), si-punto-gordo(2), si-fantasma(2), secuencia2(2), secuencia3(3), secuencia4(4)
Casos de *fitness*	Una ejecución del Pacman hasta ser comido o terminar escenario
Raw fitness	Suma de puntos obtenida si vivo.
Standard *fitness*	Máximo puntos – *raw fitness*
Hits	No hay
Wrapper	No hay
Parámetros	M=5000, G=51
Éxito	Obtener máximo de puntos posible (puntos en el escenario)

Objetivo	Encontrar un programa que resuelva **Paridad-10**
Terminales	D0, D1, D2, D3, D4, D5, D6, D7, D8, D9
Funciones	and(2), or(2), not(1), nand(2), nor(2)
Casos de *fitness*	Las 1024 combinaciones de los 10 valores de entrada, junto con la salida apropiada para cada caso
Raw fitness	Número de casos correctos
Standard *fitness*	1024 - *raw fitness*
Hits	Como la *raw fitness*
Wrapper	No hay
Parámetros	M=16000, G=51
Éxito	Obtener máximo número de hits (1024)



Regresión Simbólica

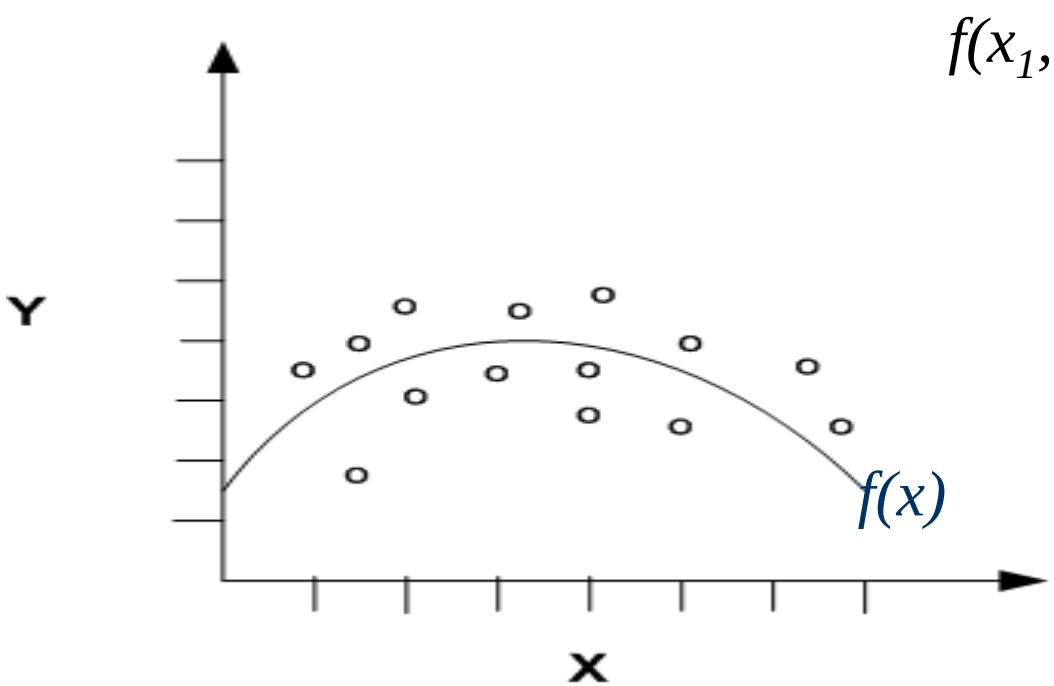
- Regresión: encontrar la dependencia funcional de una variable respecto de otra u otras: *y=f(x)*
- En estadística se hace regresión lineal
- Hay que encontrar los parámetros Beta. Existe una fórmula para ello
- ¿Y la regresión no lineal?



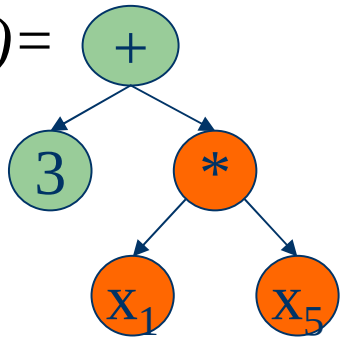
$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_p x_{pi}$$

Regresión no lineal

- A partir de ciertos datos, encontrar la función *f(x)* que “mejor” los describa



$f(x_1, x_2, \ldots, x_5) =$



Regresión Simbólica en PG

- ¿Objetivo?: encontrar una función $f(x_1, x_2, \ldots)$ que describa "bien" los datos
- ¿Funciones?: tenemos libertad, por ejemplo funciones aritméticas: +, -, *, /
- ¿Terminales?: variables de entrada $x_1, \ldots, x_5$ y la Ephemeral random constant *R*
- Recordemos que hay que proteger las funciones para conseguir el **cierre** (o **closure**).

  – Un ejemplo de división protegida es:

  %(a,b) = si (b<> 0) entonces (a/b) sino real-grande

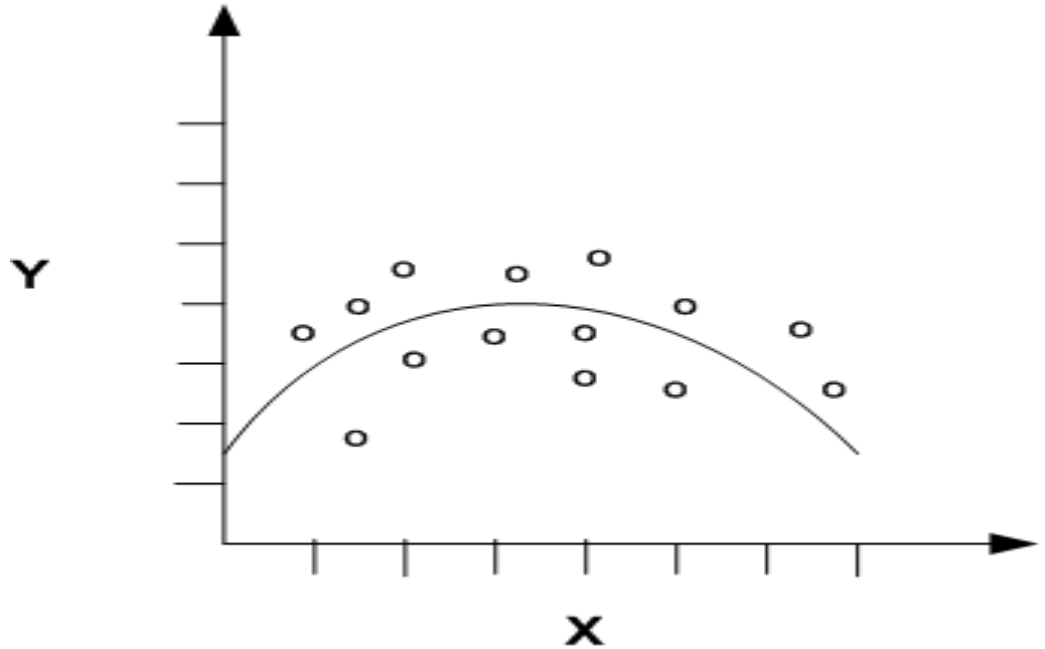
  – Un ejemplo de raíz cuadrada protegida es:

  Sqrt%(x) = si(x>=0) entonces (sqrt(x)) sino 0.0

Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	X_0, X_1, …, X_5, R
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	¿?
Raw fitness	
Standard *fitness*	
Hits	
Wrapper	
Parámetros	
Éxito	

Regresión Simbólica en PG. Casos de fitness

- Los casos de fitness son los datos a aproximar:
  O sea, las tuplas (x_1, …, x_5, valor)
- Seguramente tengan ruido



Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	X_0, X_1, …, X_5, R
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	Datos a aproximar (lista de tuplas (entrada, salida))
Raw fitness	¿?
Standard *fitness*	
Hits	
Wrapper	
Parámetros	
Éxito	

Regresión Simbólica en PG. Raw fitness

- ¿Cómo medir el grado de parecido entre una función evolucionada por PG y los datos?
- $Dato_i = (entrada_i, salida_i) = (x_{i1}, x_{i2}, \ldots, x_{i5}, s_i)$
- $Errores\ e_i = salida(dato_i) - f(entrada(dato_i))$
  - f es un programa evolucionado por PG del cual queremos saber su fitness
- Se suele utilizar el error cuadrático medio: es la media de los errores al cuadrado, entre la función evolucionada y los datos

$$\frac{1}{n}\sum_{i=1}^{n} e_i^2$$

Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	$X_0, X_1, \ldots, X_5, R$
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	Datos a aproximar (lista de tuplas (entrada, salida)) $(x_{i1}, x_{i2}, \ldots, x_{i5}, s_i)$
Raw fitness	$\frac{1}{n}\sum_{i=1}^{n} e_i^2$
Standard *fitness*	*Igual a la Raw Fitness (se trata de minimizar el error)*
Hits	*¿?*
Wrapper	
Parámetros	
Éxito	

Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	$X_0, X_1, \ldots, X_5, R$
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	Datos a aproximar (lista de tuplas (entrada, salida)) $(x_{i1}, x_{i2}, \ldots, x_{i5}, s_i)$
Raw fitness	$\frac{1}{n}\sum_{i=1}^{n} e_i^2$
Standard *fitness*	*Igual a la Raw Fitness (se trata de minimizar el error)*
Hits	*Contar en cuantos casos de fitness* $\lvert s_i - f(e_i)\rvert$ *es pequeño*
Wrapper	NO HAY
Parámetros	
Éxito	

Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	$X_0, X_1, \ldots, X_5, R$
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	Datos a aproximar (lista de tuplas (entrada, salida)) $(x_{i1}, x_{i2}, \ldots, x_{i5}, s_i)$
Raw fitness	$\frac{1}{n}\sum_{i=1}^{n} e_i^2$
Standard *fitness*	*Igual a la Raw Fitness (se trata de minimizar el error)*
Hits	*Contar en cuantos casos de fitness* $\lvert s_i - f(e_i)\rvert$ *es pequeño*
Wrapper	NO HAY
Parámetros	
Éxito	Acabar cuando la raw fitness sea un valor pequeño (ej: <0.05)

Objetivo	Encontrar una composición de funciones que aproxime los datos
Terminales	$X_0, X_1, \ldots, X_5, R$
Funciones	+(2), -(2), %(2), *(2)
Casos de *fitness*	Datos a aproximar (lista de tuplas (entrada, salida)) $(x_{i1}, x_{i2}, \ldots, x_{i5}, s_i)$
Raw fitness	$\frac{1}{n}\sum_{i=1}^{n} e_i^2$
Standard *fitness*	*Igual a la Raw Fitness (se trata de minimizar el error)*
Hits	*Contar en cuantos casos de fitness* $\lvert s_i - f(e_i)\rvert$ *es pequeño*
Wrapper	NO HAY
Parámetros	
Éxito	Acabar cuando el número de hits sea igual al número de casos de fitness

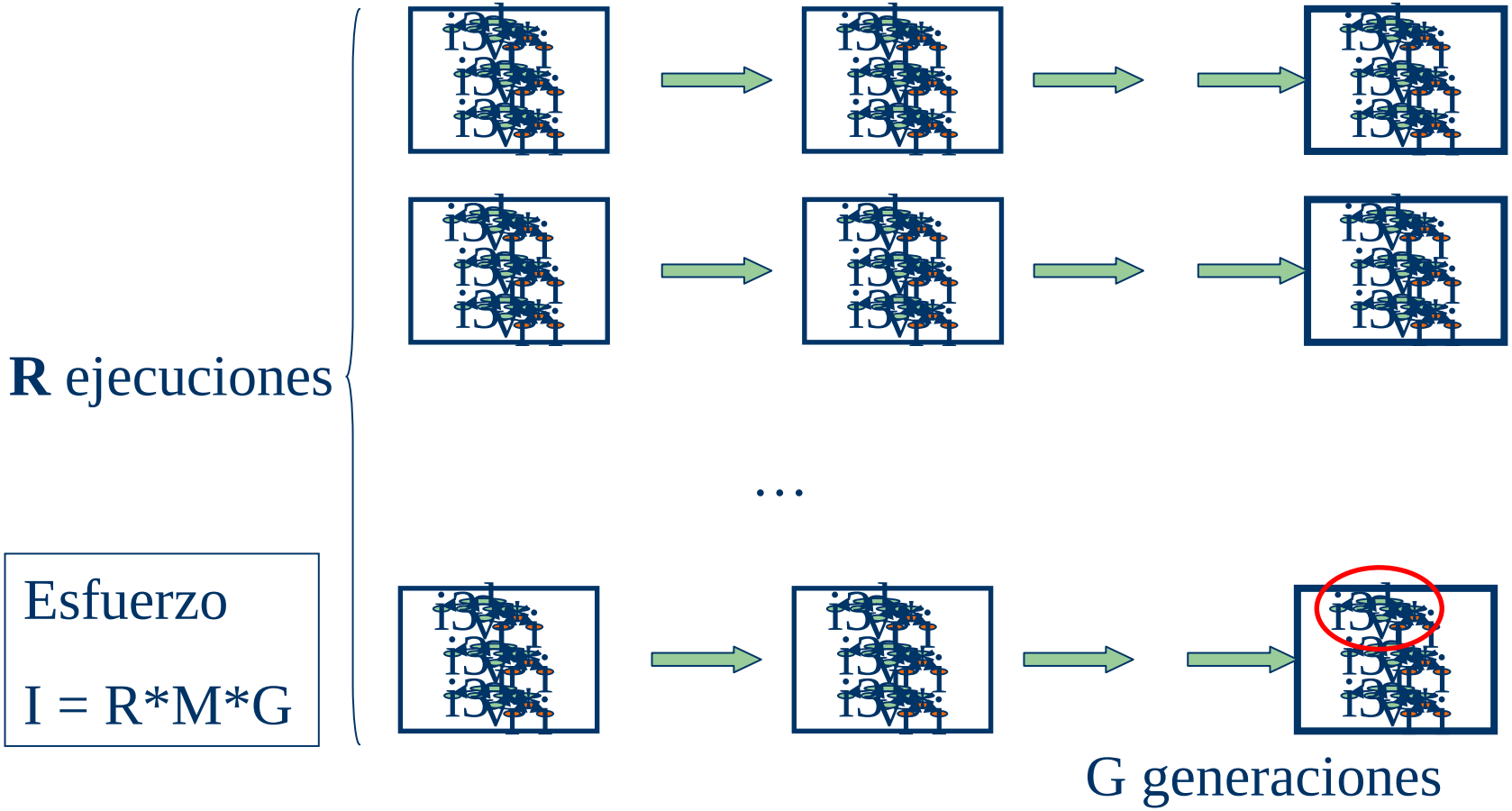
Esfuerzo computacional en PG

- Supongamos que queremos medir cómo de difícil le resulta a la PG resolver un problema (el tiempo que le cuesta)
- Puede servir para comparar un sistema estándar de PG con otro más avanzado, comparándolos sobre el mismo problema
- O para determinar si el Pac-Man es más o menos sencillo de resolver que el problema de la paridad par

Esfuerzo computacional en PG

- Donde más esfuerzo invierte la PG es en el cálculo de la *fitness*, que involucra la ejecución de un programa (individuo)
- Podemos medir la dificultad de resolver un problema, contando el número de individuos que la PG tiene que evaluar para llegar a una solución

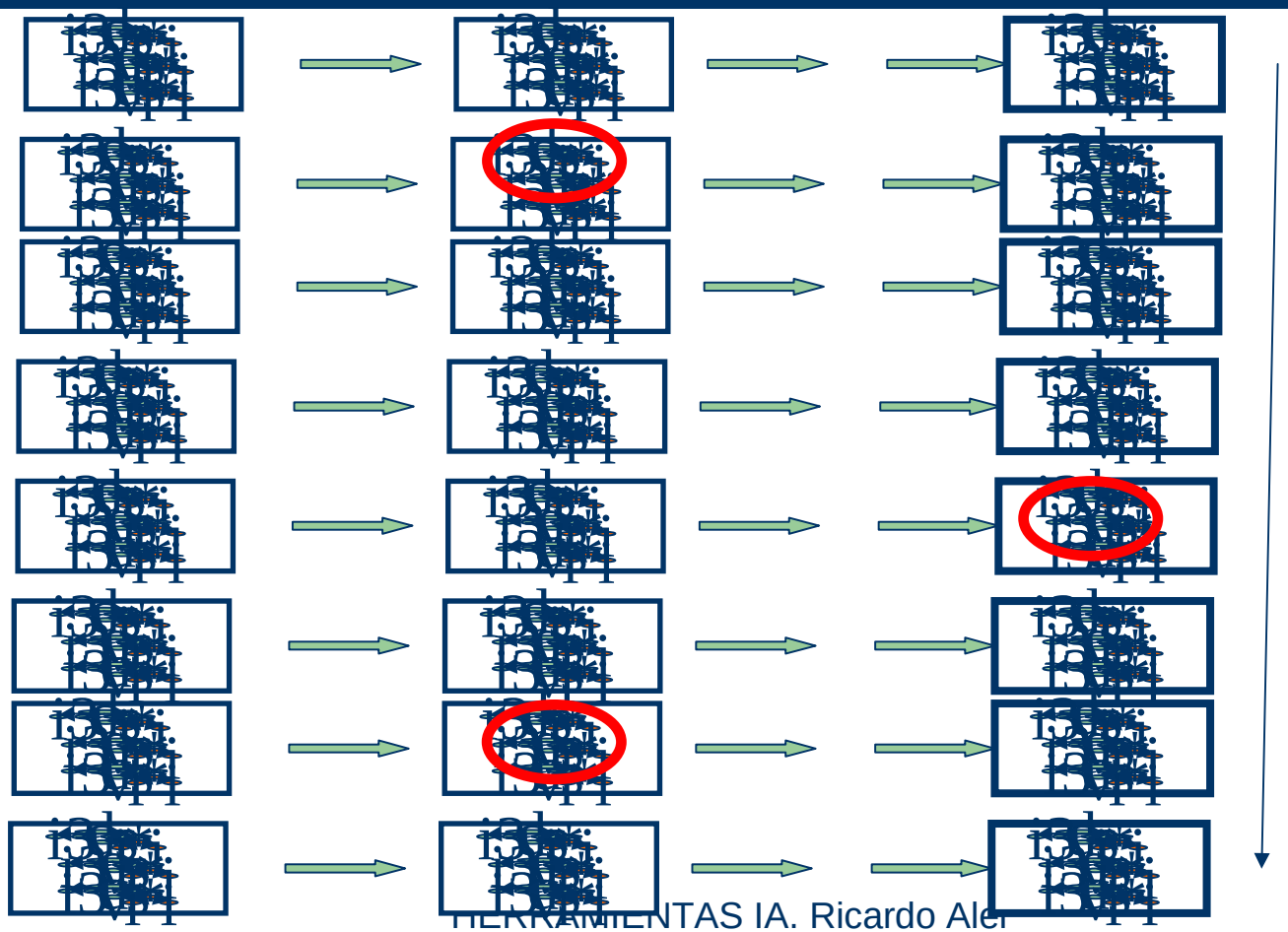
Esfuerzo computacional en PG



Esfuerzo computacional en PG

- Problema: en algunas ocasiones la PG encontrará la solución en 10 ejecuciones, en otras requerirá 15, etc.
- Necesitamos calcular una especie de “esfuerzo computacional medio”
- Solución: hacemos muchas más ejecuciones de PG, en el curso de las cuales, aparecerán varias soluciones

Esfuerzo computacional en PG



Diferentes ejecuciones

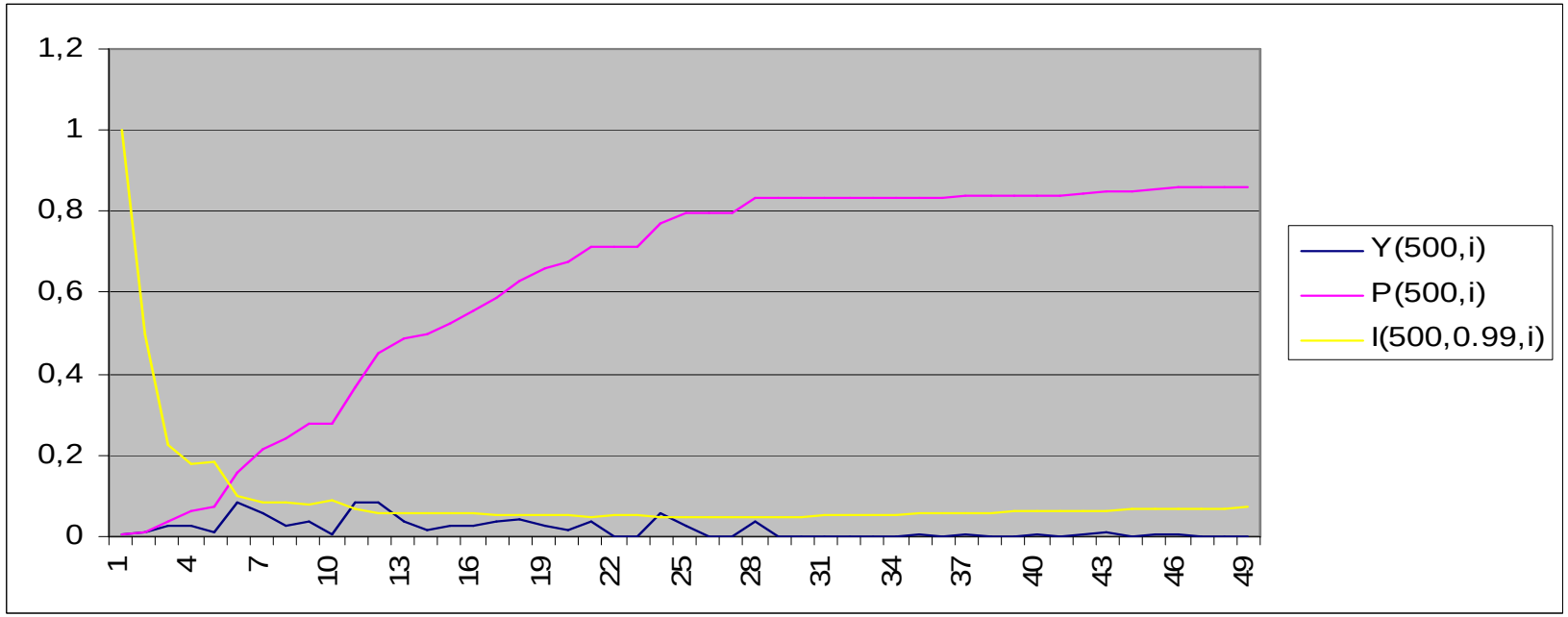
Esfuerzo computacional en PG

- ¿Cuántos programas hay que evaluar para obtener un programa correcto en un problema determinado?
- Estimar *Y(M,i)* = probabilidad programa correcto en generación *i*
- Calcular P(M,i) = prob. acumulada de obtener prog. correcto en generación *i* o antes
- Calcular número de ejecuciones para obtener un prog. correcto con probabilidad *z*:

  $$R(z) = \log(1-Z)/(\log(1-P(M.i))$$

- Esfuerzo total: $I(M,i,z) = R(z) * M * i$

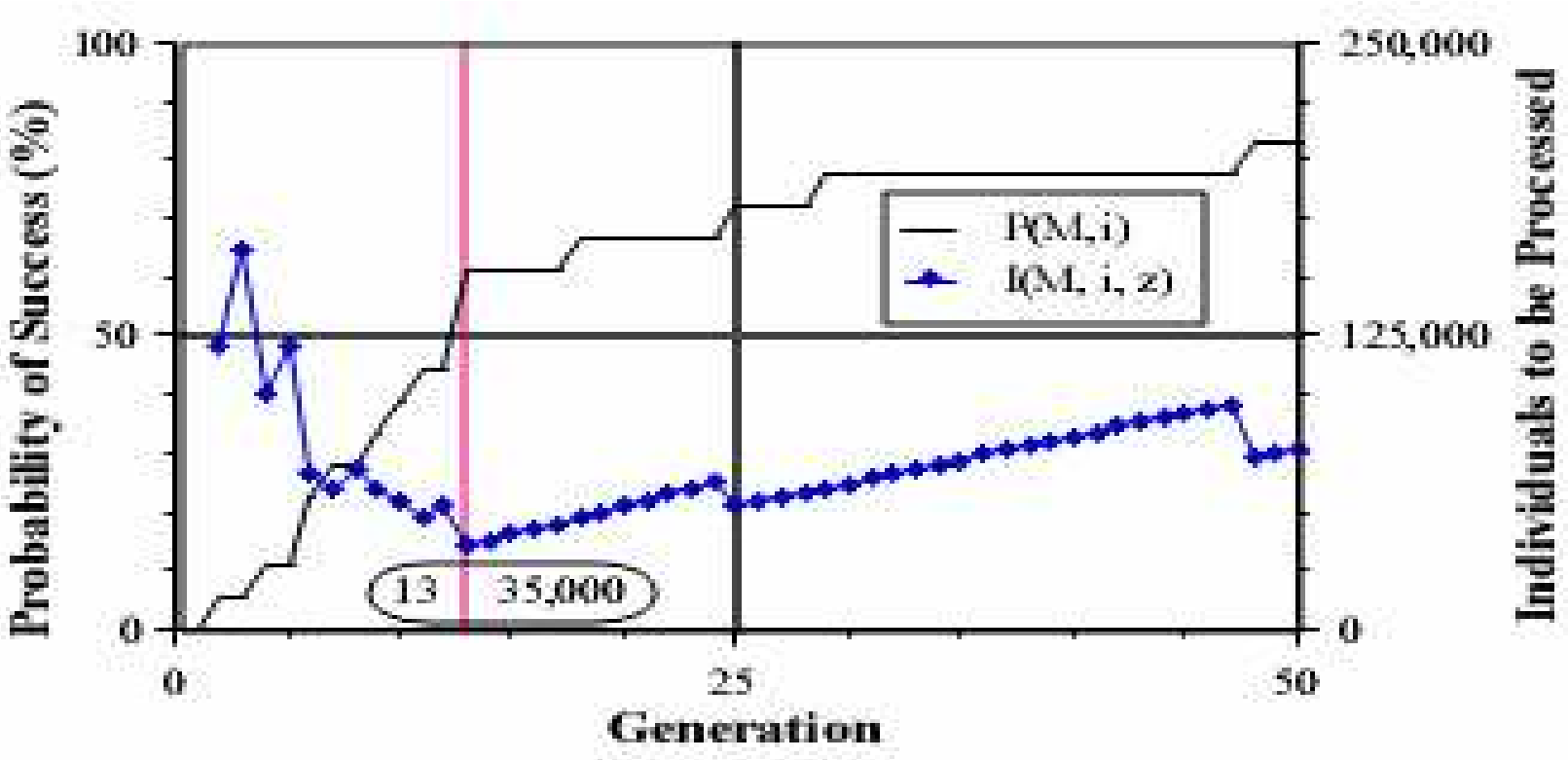
Ejemplo esfuerzo computacional



Esfuerzo computacional = mínimo I(500,0.99,i) = 36084,94534

Generación = i* = 25

Ejemplo esfuerzo computacional



Esfuerzo computacional paridad-par

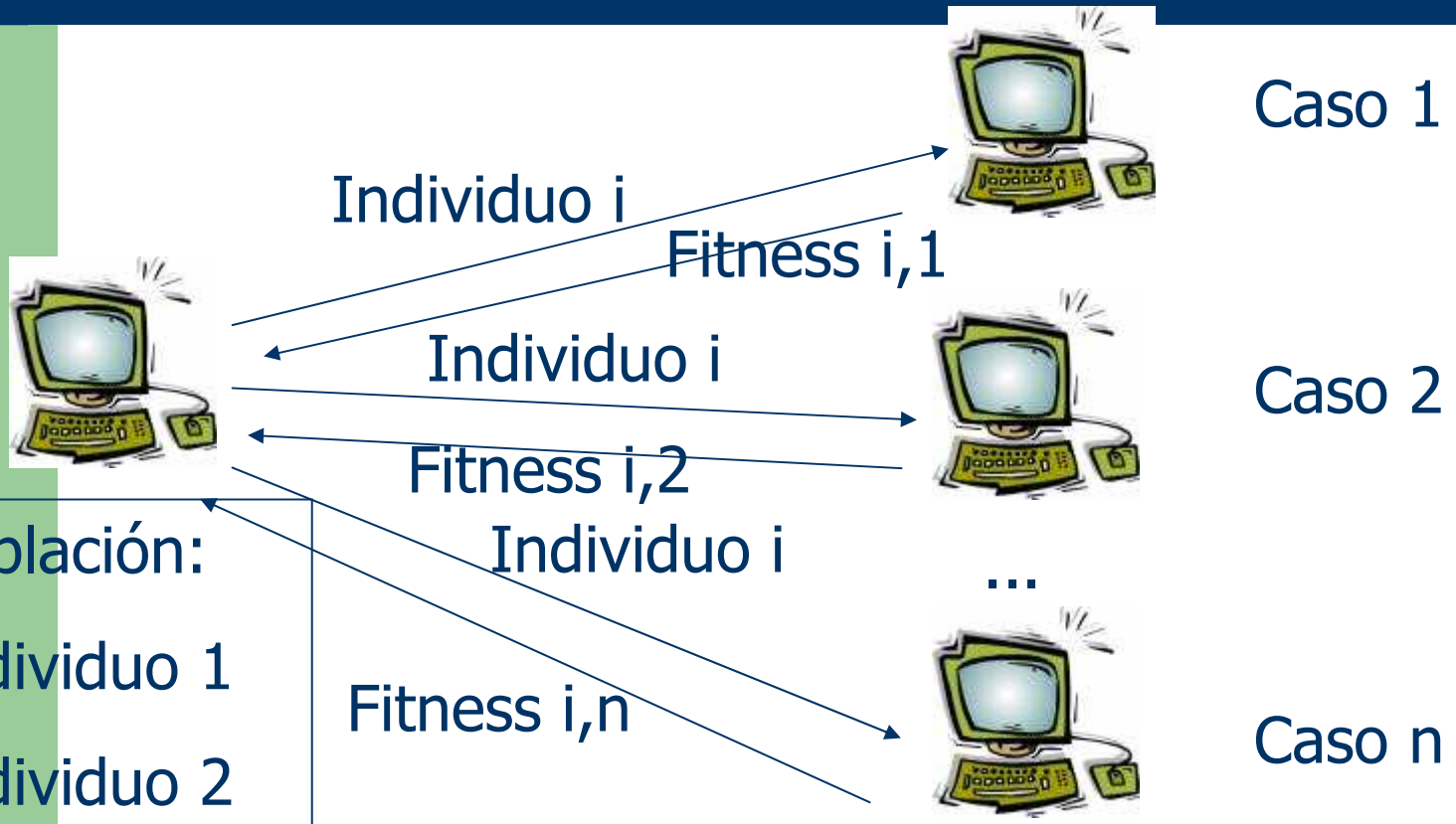
Paridad	Esfuerzo (evaluaciones)	Tiempo (horas)
3	96.000	
4	384.000	
5	6.528.000	
6	70.176.000	5h (P-2GHz)

Notas: No todas las evaluaciones consumen el mismo tiempo
La mayor parte del tiempo se consume en *fitness*

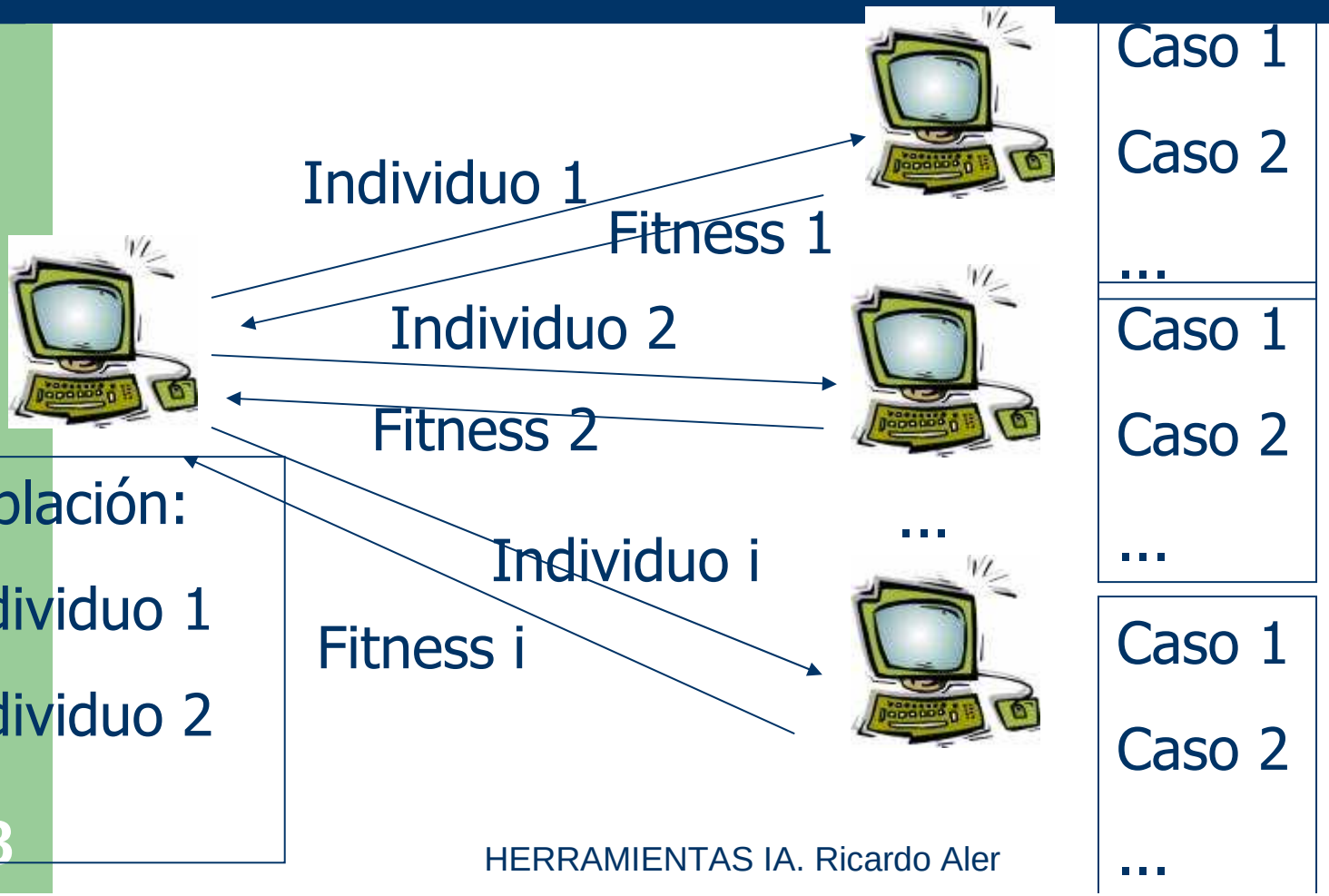
Aceleración de la PG

- La PG requiere **gran cantidad** de esfuerzo computacional
- Pero se pueden conseguir resultados con máquinas actuales
- Ley de Moore: el poder computacional se duplica cada 18 meses
- Evolución de código máquina (x2000 lisp, x100 C)
- Hardware reconfigurable (PGAs)
- Paralelismo:
  - De casos de prueba
  - De evaluación de programas
  - Distintas ejecuciones en distintas máquinas
  - Varias poblaciones + migración

Paralelismo de los Casos de Fitness



Paralelismo de Individuos



Paralelismo de Ejecuciones

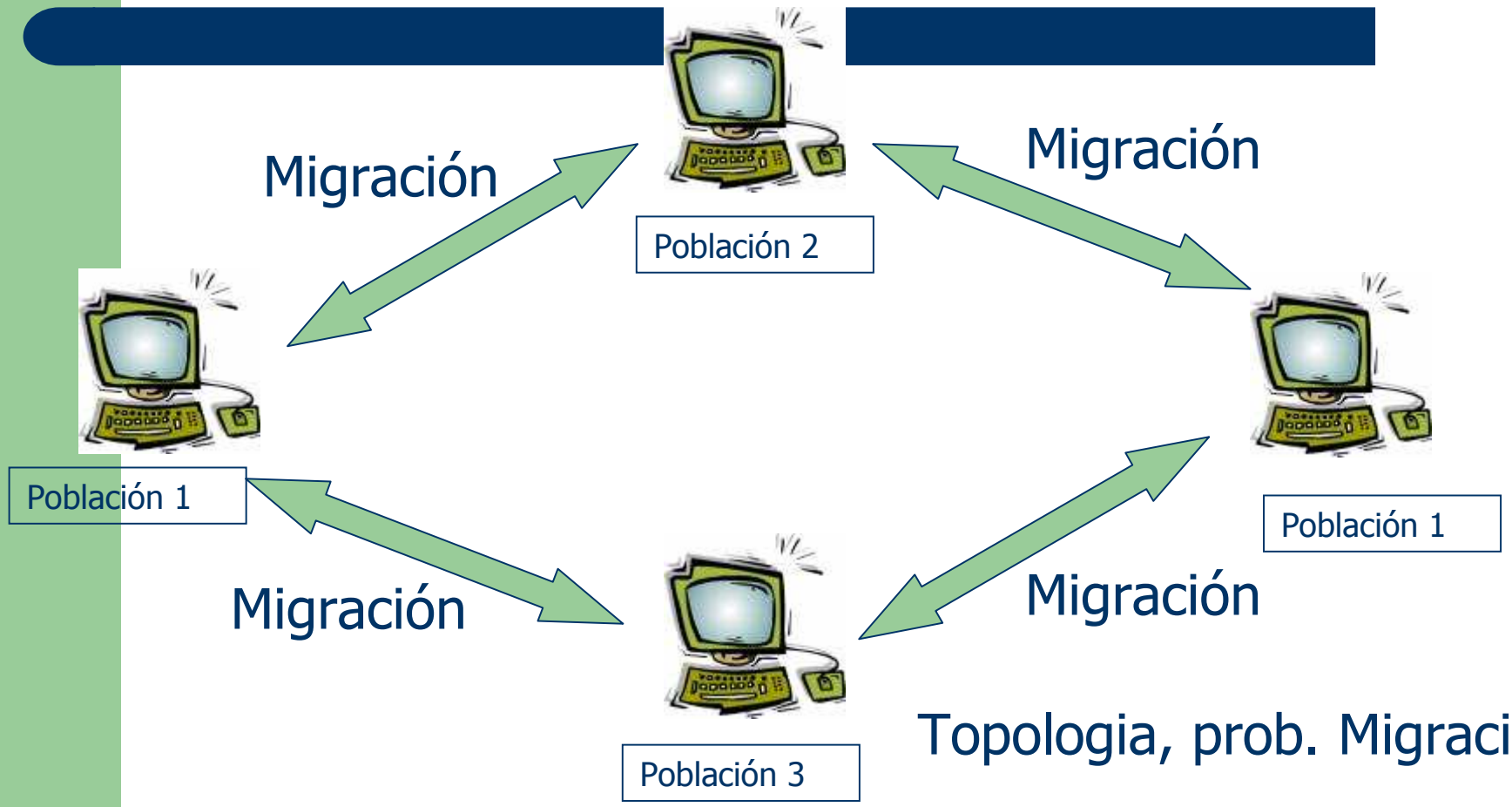
Población 2

Población 1

Población 1

Población 3

Paralelismo con Islas



Topologia, prob. Migración

Mantiene diversidad

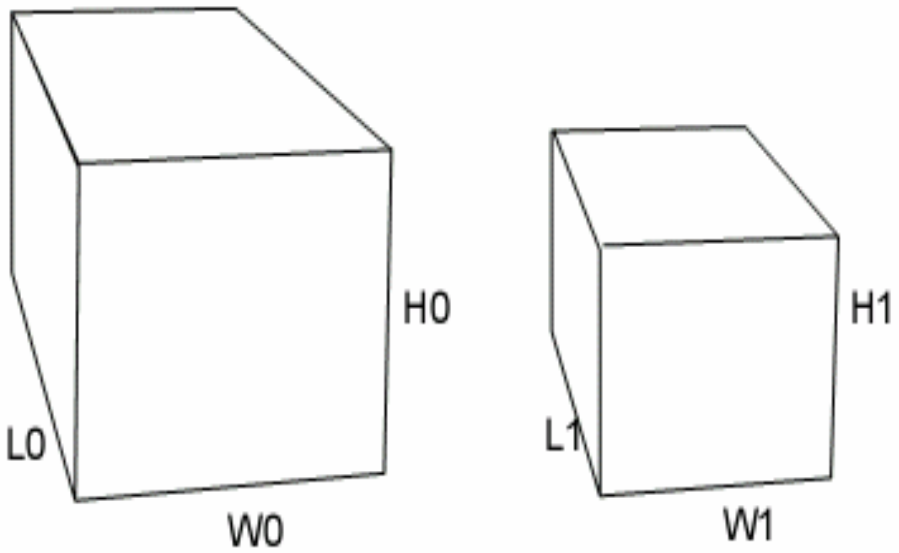
Reutilización

- De cálculos: guardarlos en una **variable** o en una estructura de datos (arrays, pilas, ...)
- De procedimientos: **subrutinas** (ADF: Automatic Defined Functions)
- De repeticiones: iteraciones, **bucles**, recursividad
- Sólo con bucles (o equivalente), la PG es completa en el sentido de Turing

Necesidad de las subrutinas

- Problema: escribir un programa que encuentre la diferencia de volúmenes de dos hexaedros (cubos) C1 y C2 : (ancho x largo x alto)
- Solución: DV = A1*L1*H1- A2*L2*H2
- Un programador humano definiría la subrutina V(A,L,H) = A*L*H
- Y resolvería el problema así: DV=V(A1,L1,H1)-V(A2,L2,H2)

Los Dos Cubos



L = largo, W = ancho, H = alto

Casos de *fitness* (los dos cubos)

Caso	A1	L1	H1	A2	H2	L2	DV
1	3	4	7	2	5	3	54
2	7	10	9	10	3	1	600
3	10	9	4	8	1	6	600
4	3	9	5	1	6	4	111
5	4	3	2	7	6	1	-18
...	...	...	...	...	...	...	...

Nota: este es un problema de regresión simbólica

Automatically Defined Functions (ADF)

- La PG puede evolucionar subrutinas, además del programa principal,
- La PG encuentra subrutinas, aunque no siempre las que una persona hubiera diseñado
- En el caso de los dos cubos, el esfuerzo computacional **decrece** al utilizar subrutinas:
  - 1.176.000 (con)/2.220.000 (sin)
- El tamaño (“complejidad estructural”) del programa solución **decrece** con subrutinas:
  - 17,8 (con) / 33,5 (sin)

Representación de las ADFs

Varios subárboles = 1 cuerpo principal + 1 subrutina

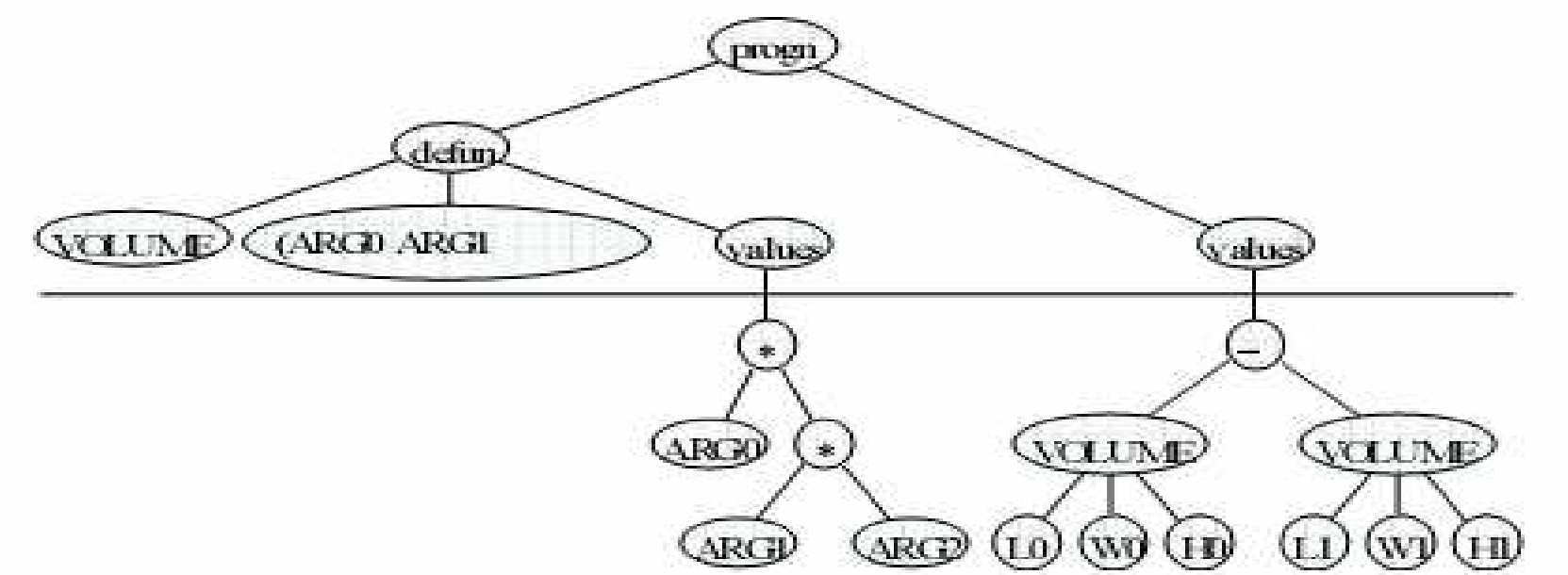


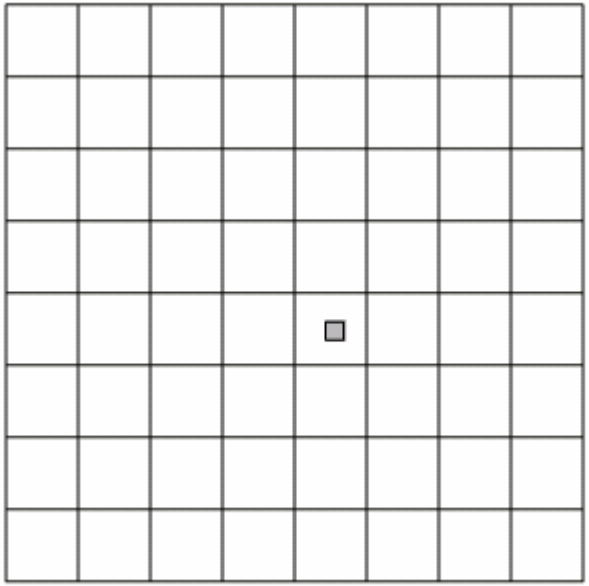
Tabla para problema de los dos cubos con ADFs

Arquitectura	1 programa principal y 1 ADF con 3 argumentos
Terminales prog. Principal	A1, L1, H1, A2, L2, H2
Funciones prog. Principal	+(2), -(2), *(2), %(2), ADF0(3)
Terminales ADF0	ARG0, ARG1, ARG2
Funciones ADF0	+(2), -(2), *(2), %(2)
Fitness	Número de casos acertados

Ejemplo del cortador de cesped (lawnmower)

- Dado un jardín de MxM trozos de cesped (toroidal) y un cortador de césped que puede:
  - Cortar césped
  - Avanzar 1 casilla
  - Girar a la izquierda 90 grados
  - Moverse a cualquier otra casilla
- Encontrar un programa de control que corte todo el cesped

64x64 Cortador de Cesped



Terminales y funciones en el lawnmower

- *Left*: girar a la izquierda (modulo 4)
- *Mow*: corta el cesped (tendremos que guardar una estructura de datos que represente el campo de cesped. Llamemos *g* a la estructura global donde se guardan
- *Vma(a,b):* suma dos vectores
- *Frog(a):* salta a la posición *a*
- *Prog2(a,b)*: secuencia de dos instrucciones. Devuelve lo que devuelva *b*

Peculiaridades lawnmower

- Como algunas funciones utilizan vectores (x,y), este será el tipo de datos para todas
- Cierre/closure: todas las funciones tienen que ser capaces de aceptar vectores sin producir error.

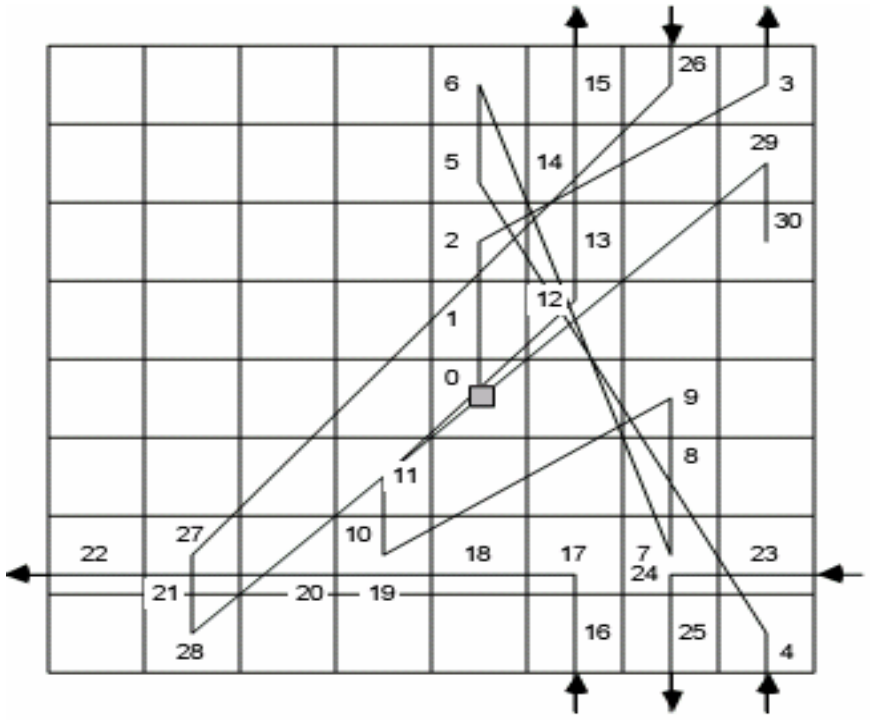
Tabla lawnmower sin ADF

Arquitectura	1 RPB
Terminales	(left), (mow), “vector aleatorio”
Funciones	vma(2), frog(1), prog2(2)
Casos de fitness	1 campo de césped de MxM sin segar
Raw fitness	Cuántos trozos de césped siega
Standard fitness	MxM-raw fitness
Hits	Raw fitness
Éxito	Cortar los MxM trozos de césped

Tabla lawnmower con ADF

Arquitectura	1 RPB, 2 ADFs con 0 y 1 argumentos respectivamente
Terminales	(left), (mow), “vector aleatorio”, **(ADF0)**
Funciones	vma(2), frog(1), prog2(2), ADF1(1)
Terminales ADF1	(left), (mow), “vector aleatorio”, **(ADF0),** , **ARG0**
Funciones ADF1	vma(2), frog(1), prog2(2)
Terminales ADF0	(left), (mow), “vector aleatorio”
Funciones ADF0	vma(2), frog(1), prog2(2)

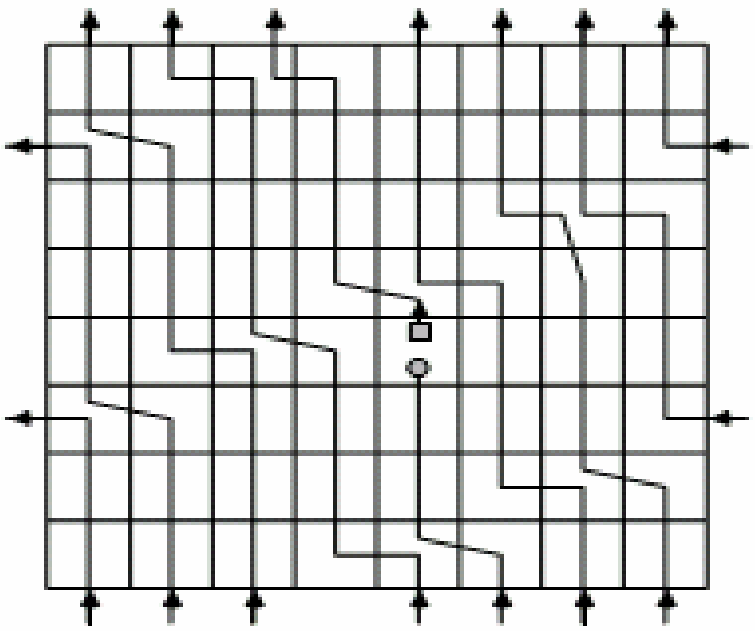
Solución sin ADFs (64x64)



Solución sin ADFs (64x64)

```
(V8A (V8A (V8A (FROG (PROGN (PROGN (V8A (MOW) (MOW)) (FROG #(3
2))) (PROGN (V8A (PROGN (V8A (PROGN (PROGN (MOW) #(2 4)) (FROG
#(5 6))) (PROGN (V8A (MOW) #(6 0)) (FROG #(2 2)))) (V8A (MOW)
(MOW))) (PROGN (V8A (PROGN (PROGN #(0 3) #(7 2)) (FROG #(5
6))) (PROGN (V8A (MOW) #(6 0)) (FROG #(2 2)))) (V8A (MOW)
(MOW)))) (PROGN (FROG (MOW)) (PROGN (PROGN (PROGN (V8A (MOW)
(MOW)) (FROG (LEFT))) (PROGN (MOW) (V8A (MOW) (MOW)))) (PROGN
(V8A (PROGN #(0 3) #(7 2)) (V8A (MOW) (MOW))) (PROGN (V8A
(MOW) (MOW)) (PROGN (LEFT) (MOW)))))))) (V8A (PROGN (V8A
(PROGN (PROGN (MOW) #(2 4)) (FROG #(5 6))) (PROGN (V8A (MOW)
#(6 0)) (FROG #(2 2)))) (V8A (MOW) (MOW))) (V8A (FROG (LEFT))
(FROG (MOW))))) (V8A (FROG (V8A (PROGN (V8A (PROGN (V8A (MOW)
(MOW)) (FROG #(3 7))) (V8A (PROGN (MOW) (LEFT)) (V8A (MOW) #(5
3)))) (PROGN (PROGN (V8A (PROGN (LEFT) (MOW)) (V8A #(1 4)
(LEFT))) (PROGN (FROG (MOW)) (V8A (MOW) #(3 7)))) (V8A (PROGN
(FROG (MOW)) (V8A (LEFT) (MOW))) (V8A (FROG #(1 2)) (V8A (MOW)
(LEFT))))) (PROGN (V8A (FROG #(3 1)) (V8A (FROG (PROGN (PROGN
(V8A (MOW) (MOW)) (FROG #(3 2))) (FROG (FROG #(5 0))))) (V8A
(PROGN (FROG (MOW)) (V8A (MOW) (MOW))) (V8A (FROG (LEFT))
(FROG (MOW)))))) (PROGN (PROGN (PROGN (PROGN (LEFT) (MOW))
(V8A (MOW) #(3 7))) (V8A (V8A (MOW) (MOW)) (PROGN (LEFT)
(LEFT)))) (V8A (FROG (PROGN #(3 0) (LEFT))) (V8A (PROGN (MOW)
(LEFT)) (FROG #(5 4)))))))) (PROGN (FROG (V8A (PROGN (V8A
(PROGN (PROGN (V8A (PROGN (PROGN (MOW) #(2 4)) (FROG #(5 6)))
(PROGN (V8A (MOW) #(1 2)) (FROG #(2 2)))) (V8A (MOW) (MOW)))
(FROG #(3 7))) (V8A (PROGN (PROGN (MOW) #(2 4)) (FROG #(5 6)))
(PROGN (V8A (MOW) #(6 0)) (FROG #(2 2))))) (PROGN (PROGN (V8A
(FROG (MOW)) (V8A #(1 4) (LEFT))) (PROGN (FROG (MOW)) (V8A
(MOW) #(3 7)))) (V8A (PROGN (FROG (MOW)) (V8A (LEFT) (MOW)))
(V8A (FROG #(1 2)) (V8A (MOW) (LEFT)))))) (PROGN (V8A (PROGN
(FROG #(2 4)) (V8A (MOW) (MOW))) (V8A (FROG (MOW)) (LEFT)))
(PROGN #(3 0) (LEFT))))) (FROG (V8A #(7 4) (MOW))))) (V8A
(V8A (PROGN (MOW) #(4 3)) (V8A (LEFT) #(6 1))) (MOW)))
```

Solución con ADFs (64x64)



Solución con ADFs (64x64)

```
(progn (defun ADF0 ()

(values (V8A (PROGN (V8A (V8A (LEFT) #(6 5)) (PROGN
(MOW) (LEFT))) (V8A (PROGN (MOW) (MOW)) (V8A (MOW)
(MOW)))) (V8A (PROGN (V8A #(1 4) (MOW)) (PROGN #(3 1)
(MOW))) (PROGN (PROGN #(3 1) (MOW)) (PROGN (LEFT)
(LEFT))))))))

        (defun ADF1 (ARG0)

(values (V8A (PROGN (FROG (PROGN ARG0 (ADF0))) (V8A
(PROGN (MOW) (ADF0)) (V8A (V8A (ADF0) #(3 4)) (V8A
(ADF0) ARG0)))) (V8A (FROG (FROG (MOW))) (PROGN
(PROGN (MOW) #(3 5)) (PROGN (MOW) (MOW))))))))

        (values (V8A (ADF1 (ADF1 (V8A #(7 1) (LEFT))))
(V8A (V8A (PROGN (LEFT) (LEFT)) (V8A #(7 0) (LEFT)))
(FROG (V8A (ADF0) (MOW)))))))
```

Significado de las Subrutinas

- Las subrutinas pasan a ser las nuevas primitivas
- En términos de las cuales resolver el problema
- Así, las ADFs permiten que la PG evolucione las primitivas más adecuadas para resolver el problema, al mismo tiempo que se resuelve el problema (cuerpo principal del programa)
- Se cambia de manera automática la manera de representar el problema (primitivas)

Jerarquía de ADF en paridad-par

p4(A,B,C,D) = p2(p2(A,B), p2(C,D))
p2(E,F) = nor(f(E,F),f(F,E)) = ADF1(ARG0, ARG1)
f(G,H) = not(G) and H = ADF0(ARG0, ARG1)

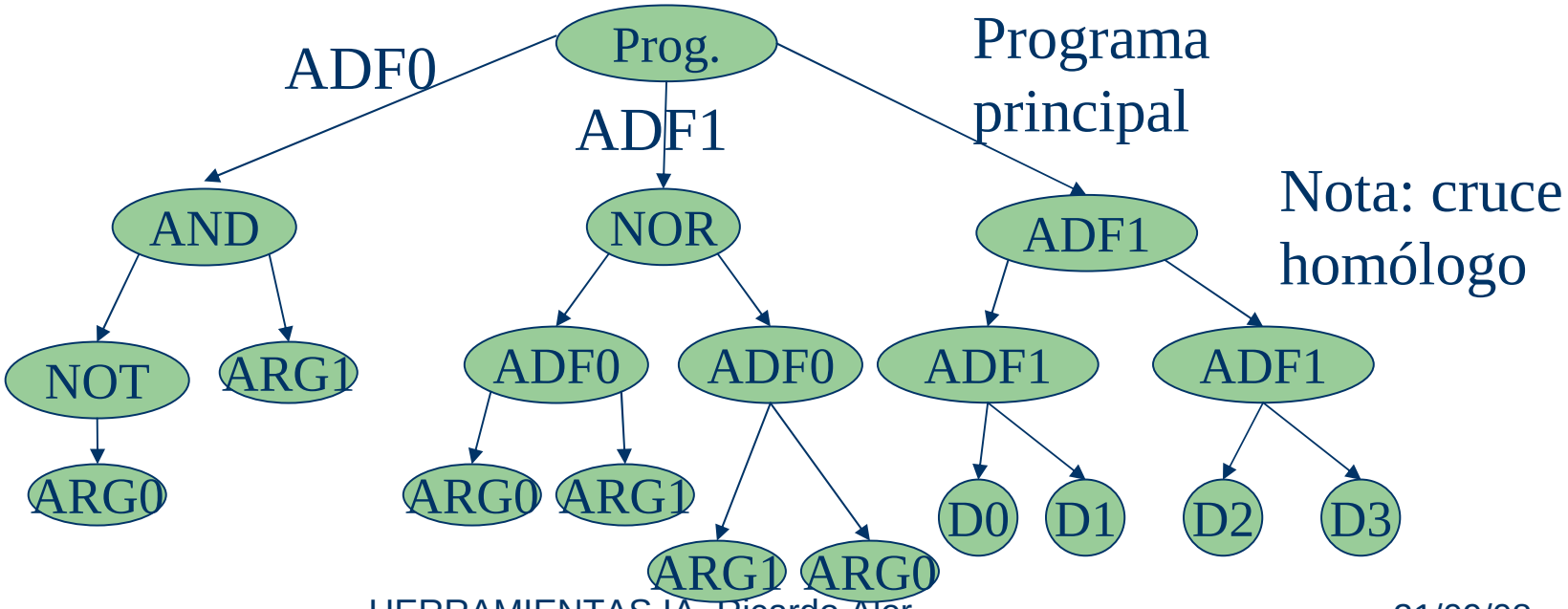


Tabla paridad-par con ADF

Arquitectura	1 programa principal y 2 ADF de dos argumentos cada una
Terminales	D0, D1, D2, D3, D4, D5, D6, D7, D8, D9
Funciones	and(2), or(2), not(1), nand(2), nor(2), **ADF1(2)**
Terminales ADF1	**ARG0, ARG1**
Funciones ADF1	**ADF0(2)**, and(2), or(2), not(1), nand(2), nor(2)
Terminales ADF0	**ARG0, ARG1**
Funciones ADF0	and(2), or(2), not(1), nand(2), nor(2)

Esfuerzo y tamaño en paridad-par

Paridad	Esfuerzo sin ADF	Esfuerzo con ADF	Tamaño sin ADF	Tamaño con ADF
3	96.000	64.000 (x1,5)	44,6	48,2
4	384.000	176.000 (x2,18)	112,6	60,1
5	6.528.000	464.000 (x14,07)	299,9	156,8
6	70.176.000	1.344.000 (x52,2)	900,8	450,3
7 a 11	NO	SI		

¿Cuántas ADFs y cuántos argumentos?

- No hay solución a priori
- Probar con distintas posibilidades, especialmente si tenemos conocimiento sobre el problema
- Utilizar muchas ADFs con muchos argumentos (la PG decidirá qué es lo que no se usa). Problema: se amplia el espacio de búsqueda
- Alteración automática de la arquitectura:
  - Duplicar ADFs/argumentos
  - Quitar ADFs/argumentos

Conclusiones ADFs

- Los programadores humanos simplifican la complejidad del problema escribiendo las subrutinas adecuadas y reutilizándolas en el código
- La PG puede evolucionar individuos compuestos por un programa principal y sus subrutinas asociadas
- Si el problema es lo suficientemente complicado, el esfuerzo computacional y la complejidad estructural (tamaño) del programa final disminuyen

Uso de variables

- Creando funciones que almacenen resultados y los lean
- Ej: Añadir la función (*escribem valor)* y el terminal (*leem)* para escribir y leer de la variable *m*
- Ej: Añadir las funciones *(escribem indice valor)* y *(leem indice)* para escribir y leer de la posición *indice* del array *m*
- Se pueden utilizar primitivas que usen estructuras de datos complejas: pilas, colas, matrices, ...

Variable *m*

- Definir las primitivas de lectura y escritura

```
Funcion escribem (subarbol) {
  m = ejecuta(subarbol);
  return(m);
}

Funcion leem () {
  return(m);
}
```

Uso de iteraciones y bucles

- Creando macros (EXPR) que los incluyan
- Ej: for (i=0; i<veces; i++) {cuerpo;}
- *(bucle veces cuerpo)* sería equivalente a:

```
Funcion bucle (veces, cuerpo) {

  v = ejecuta(veces);
  for (i=0;  i<v; i++) {
    resultado=ejecuta(cuerpo);
  }
  return(resultado);
}
```

Problemas de los bucles

- Se pueden prolongar demasiado tiempo
- Limitar el número de iteraciones o el tiempo máximo que tiene un programa para evaluarse
- Limitar la profundidad de la recursividad o el tiempo máximo del programa
- A diferencia de las ADFs, ni bucles ni recursividad están muy bien investigados

Variable *x* en PROGEN

```
public class DoubleX extends Terminal{

public DoubleX(){

        super(0, "double", "DoubleX");

        setValue((Double)0.0);

        UserProgram.addNewVariable(this); //We make the variable
accesible to the user

    }


public Object execute(PGNode[] children, UserProgram uProgram, PGStack
stack)

{return (Double) getValue();}}
```

Uso de while en PROGEN

```
public class LoopWhileFc extends Function{
      private final int loopLimit = 200;
      public Object execute(PGNode[] children, UserProgram uProgram,
PGStack stack){

    int iterations = 0;
    /* Aquí ejecutamos la condición del bucle (child0) */
    Boolean child_0 = (Boolean) children[0].execute(uProgram, stack);

    while((child_0.booleanValue()) && (iterations < this.loopLimit)) {
        /* Y aquí ejecutamos el cuerpo del bucle (child1) */
        children[1].execute(uProgram, stack);
        /* Aquí volvemos a ejecutar la condición del bucle (child0) */
        child_0 = (Boolean) children[0].execute(uProgram, stack);
    }
return null;}}
```

Variable *m* en LIL-GP

Meter la variable m en la estructura global g

```
DATATYPE f_escribem (int tree; farg *args) {
  g.m = args[0].d;
  return(m);
}

DATATYPE f_leem (int tree; farg *args) {
  return(g.m);
}
```

Uso de iteraciones y bucles en LIL-GP

```
DATATYPE bucle (int tree; farg *args) {
  int i; DATATYPE veces, resultado;
  veces = evaluate_tree (tree, args[0].t);
  for (i=0;  i<veces; i++) {
    resultado=evaluate_tree(args[1].t, tree);
  }
  return(resultado);}
```