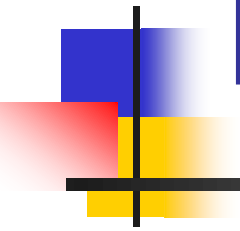
Programación Automática

MÁSTER EN CIENCIA Y TECNOLOGÍA INFORMÁTICA

Ricardo Aler Mur

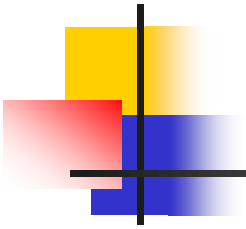


Programación Automática

Ricardo Aler Mur

Universidad Carlos III de Madrid

http://www.uc3m.es/uc3m/dpto/INF/aler



Programación Genética

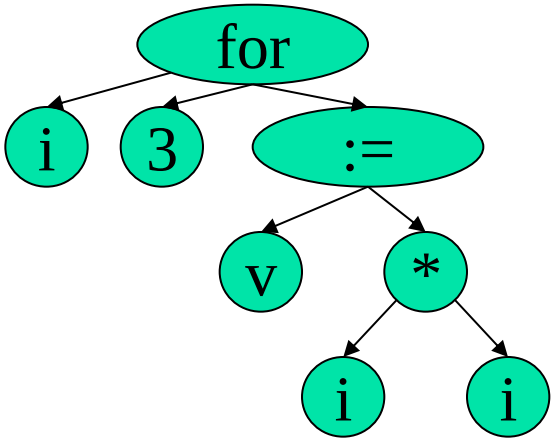
- Algoritmos genéticos para evolucionar programas
- M. Cramer. 1985. A Representation for the Adaptive Generation of Simple Sequential Programs, Proc. of an Intl. Conf. on Genetic Algorithms and their Applications.

Programación Genética

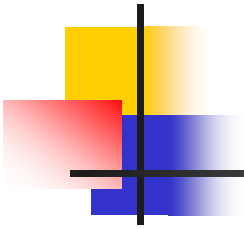
- John R. Koza
- Non-Linear Genetic Algorithms for Solving Problems. United States Patent 4,935,877. Filed May 20, 1988. Issued June 19, 1990.
- 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.
- 1994. Genetic Programming: On the Programming of Computers by Means of Natural Selection MIT Press.
- 1999. Genetic Programming III: Darwinian Invention and Problem *Solving*
- *2003. Genetic Programming IV: Routine Human-Competitive Machine Intelligence*

Representación de programas

- En Programación Genética se suele usar LISP (los programas son listas en notación prefija)
  - (for i 3 (:= v (* i i)))
  - For(i=0;i<3;i++){v=i*i;}
- O equivalentemente, parse trees



- Lenguaje = funciones + terminales



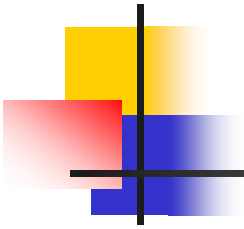
Ejemplo: Paridad par

- 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)

Casos de prueba o casos de fitness

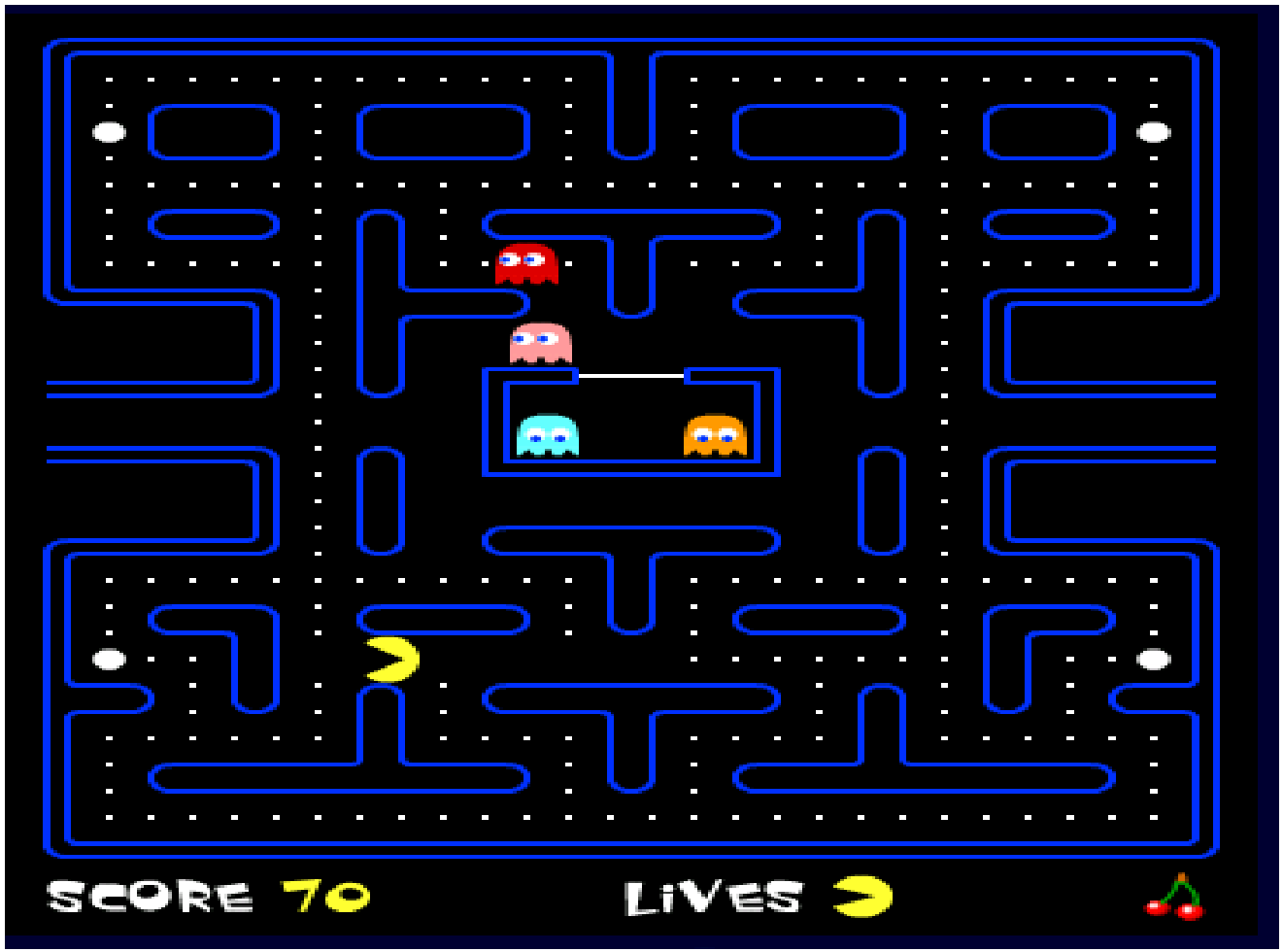
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	SALIDA
0	0	0	0	0	0	0	0	0	0	TRUE
0	0	0	0	0	0	0	0	0	1	FALSE
0	0	0	0	0	0	0	0	1	1	TRUE
…										



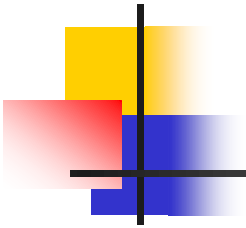
Dificultad de paridad-par

- Un simple cambio de 1 bit de 1 a 0 (o de 0 a 1) en la entrada, cambia la salida (pasa de par a impar). Esto plantea dificultades a muchos otros algoritmos de aprendizaje
- No se dispone de la función XOR o IGUAL que simplificarían el aprendizaje:
  - S = NOT(d0 XOR d1 XOR d2 ... XOR d10)

Ejemplo: Pac-Man



Ejemplo: Estrategia para Pacman



- 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
- **QUÉ**: comer todos los puntos del tablero en un tiempo límite

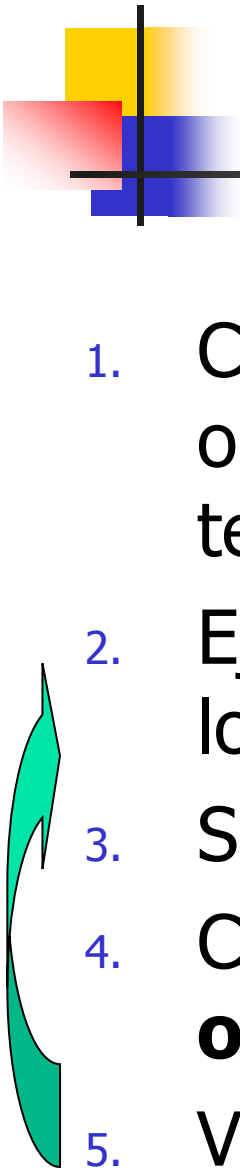
Ejemplo de programa en el Pacman

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

Evolución Darwiniana

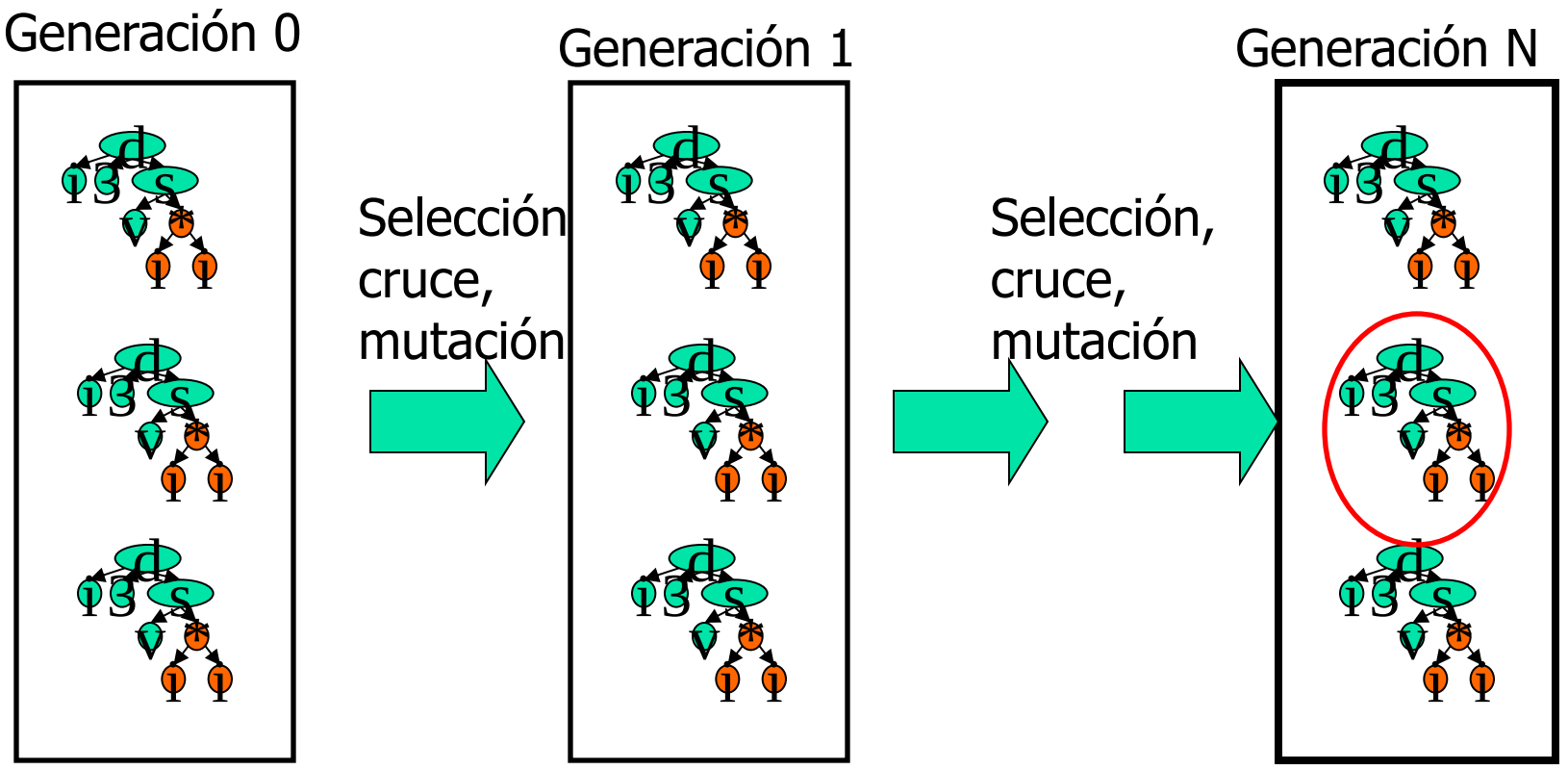
- Se reproducen los individuos más capaces (**selección**)
- Los hijos no son idénticos a los padres. Cambios en el DNA debidos a mezcla, cruce, mutación (**variación**)
- El resultado es que se producen individuos adaptados a su medio
- Evolución = variación + selección

Algoritmo Programación Genética



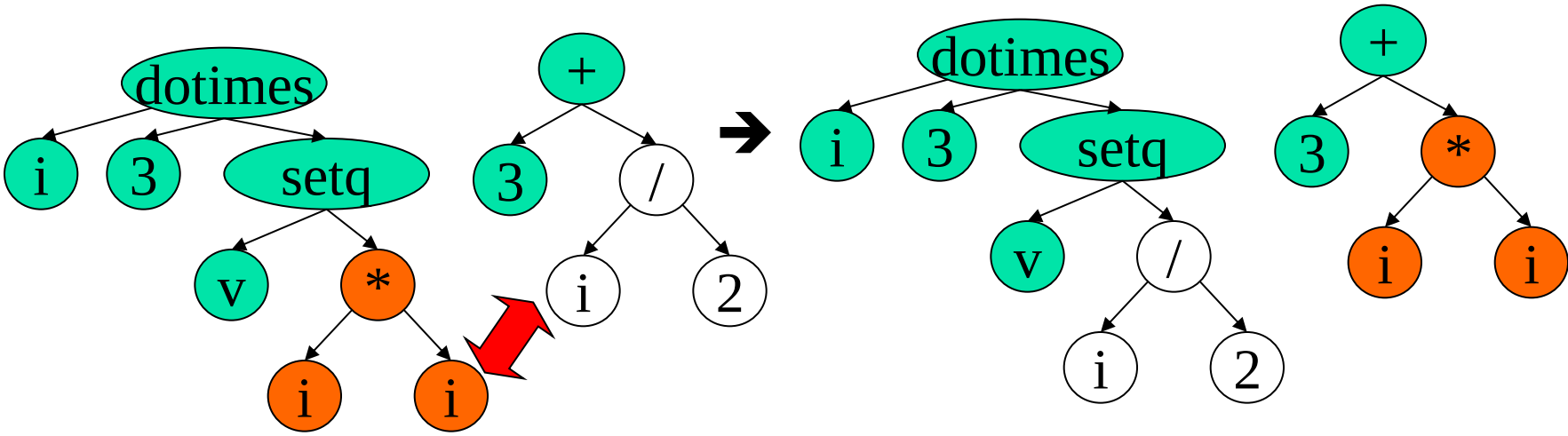
1. Creación de una población de programas de ordenador **aleatoria**, utilizando funciones y terminales
2. Ejecución de todos los programas y evaluación de los mismos (función de ***fitness***)
3. Selección de los **mejores** programas
4. Creación de una nueva población aplicando los **operadores genéticos** a los seleccionados
5. Volver a 2 hasta encontrar un "buen" programa

Evolución modelo generacional

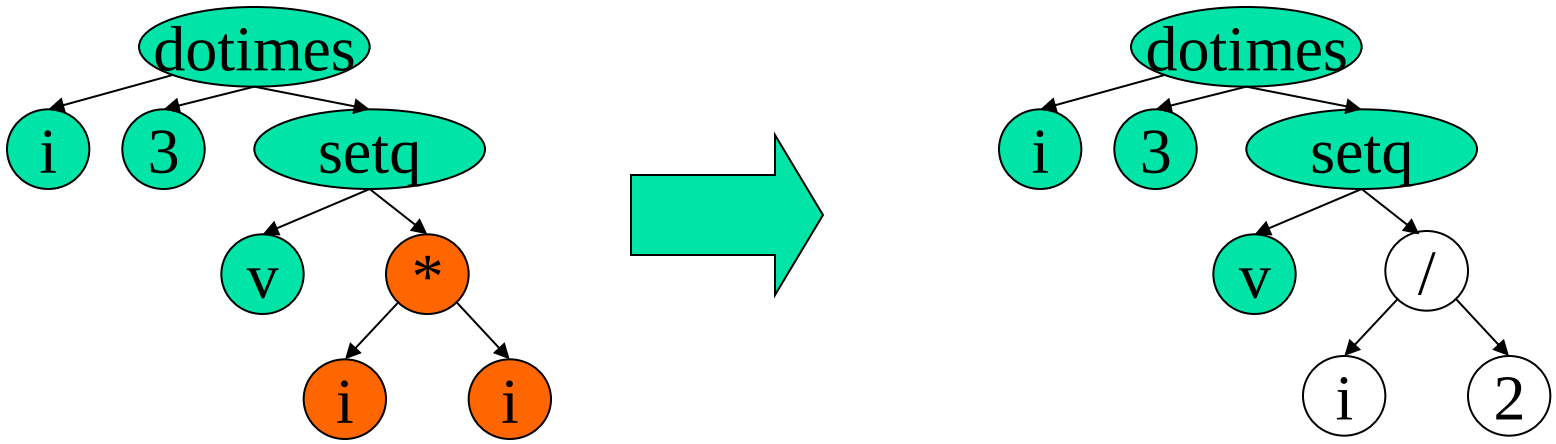


Operadores genéticos. Crossover

- Reproducción (copia sin modificación)
- Recombinación o cruce (crossover):

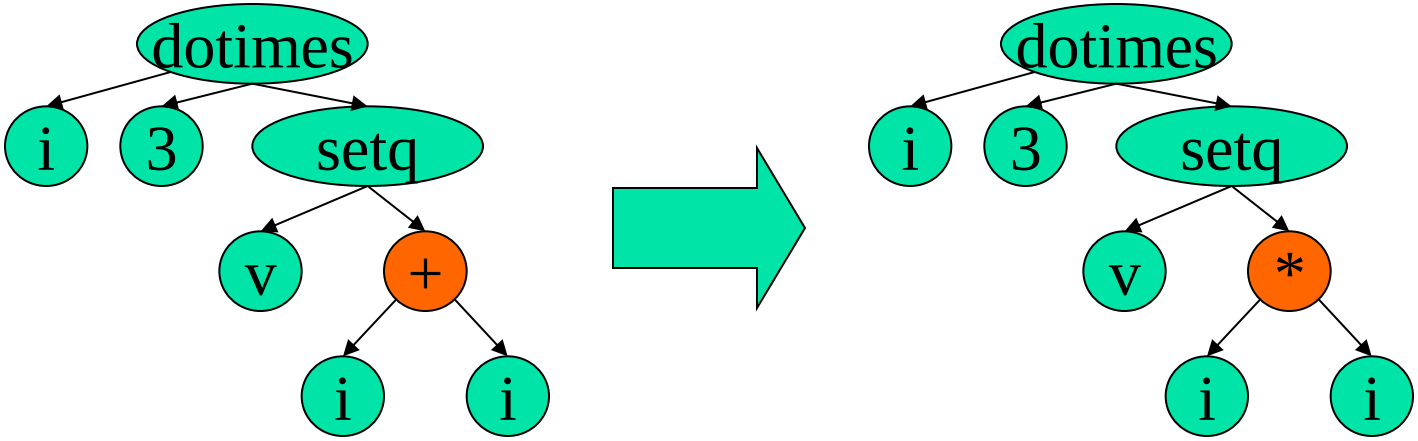


Operadores genéticos. Mutación de subárbol

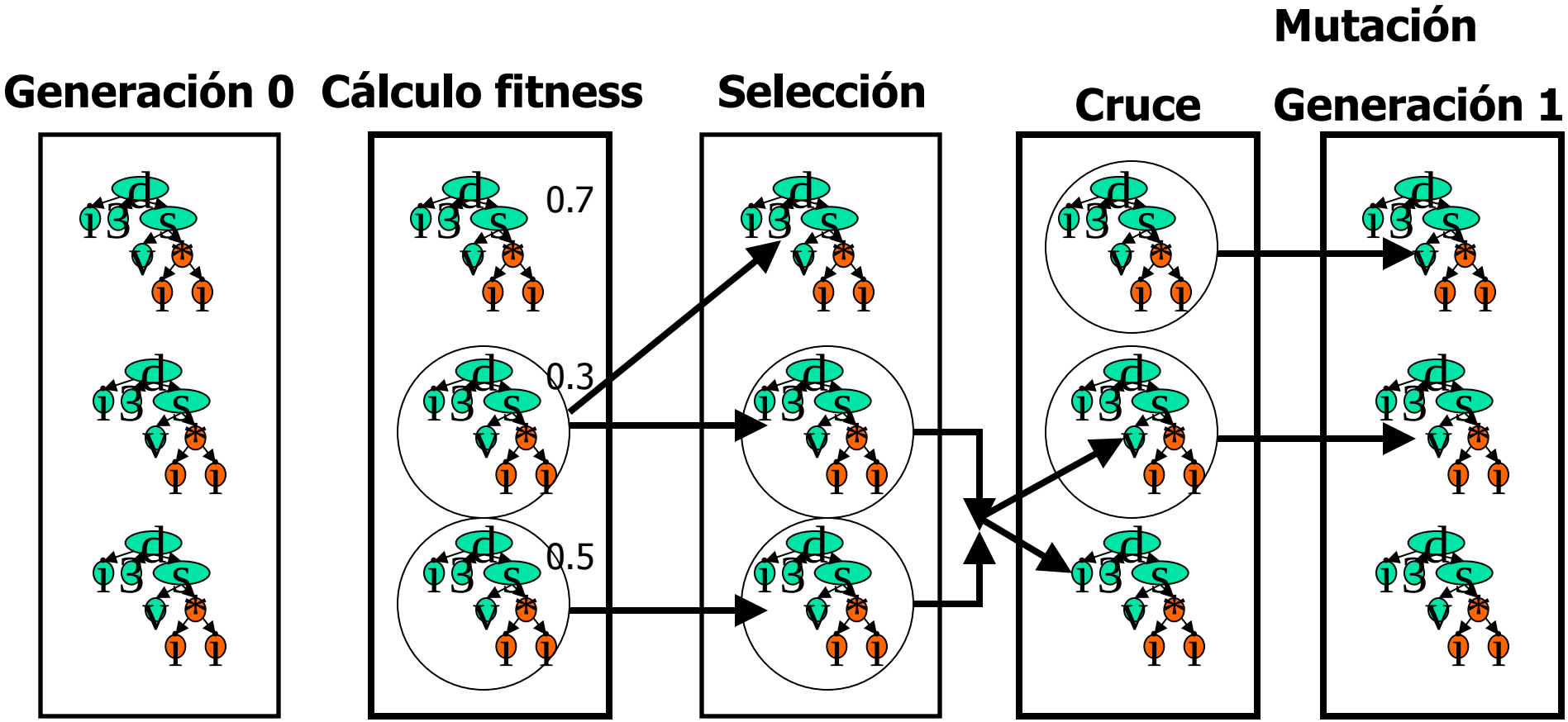


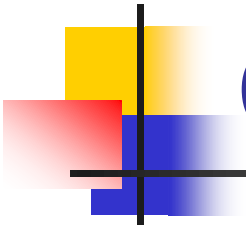
Operadores genéticos. Mutación de punto

Elegir una función con la misma aridad en el punto de mutación



De la generación *i* a la *i+1*

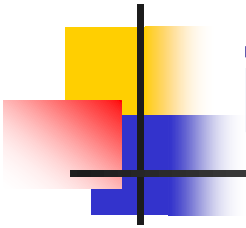




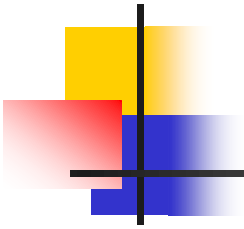
Generación de población inicial

- 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

Métodos de generación de individuos

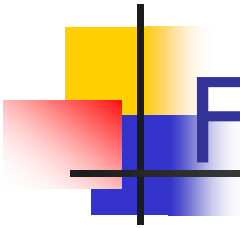


- "Full": misma profundidad para todas las ramas del árbol
- "Grow": profundidad variable
- "Ramped half and half":
  - Se generan árboles para cada posible profundidad
  - El 50% será full y el 50% grow
- Objetivo: maximizar la diversidad



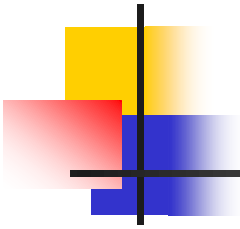
Funciones y Terminales

- Funciones: aquellas funciones o macros que toman argumentos. Ej: *(+ 3 4)*
- Terminales:
  - 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

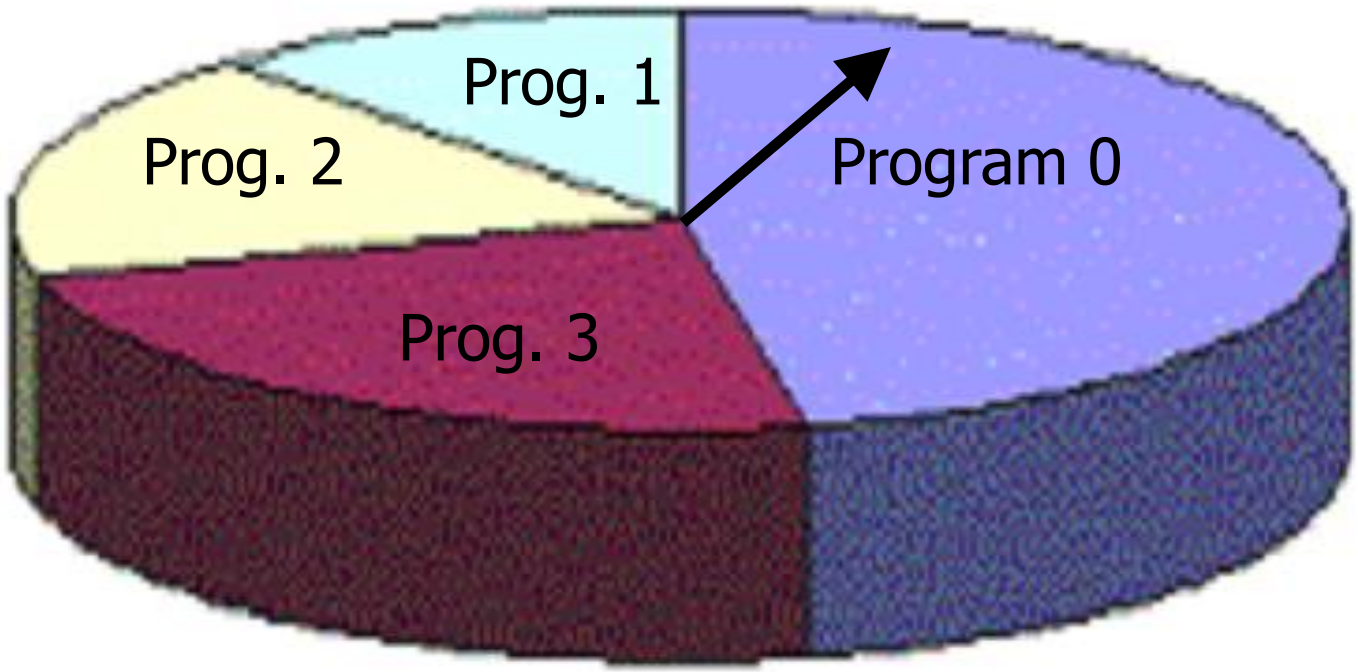
- Las funciones se deben ejecutar con cualquier argumento y sin producir errores (cierre léxico o ***closure***). Ej: división por cero protegida.
- En PG estándar **no** hay tipos de datos. Todas las funciones tienen que estar preparadas para recibir cualquier tipo y valor de argumento



Selección Darwiniana

- Proporcional a la *fitness* (probabilística): usa *fitness* normalizada
  - Problema: superindividuos, convergencia prematura
- Torneo
  - Se eligen varios individuos y el mejor se selecciona
  - Cuanto más grande es el conjunto de torneo, más presión selectiva (Normal: 4)
- Elitismo:
  - Pasar el mejor individuo a la siguiente generación (para evitar perderlo)

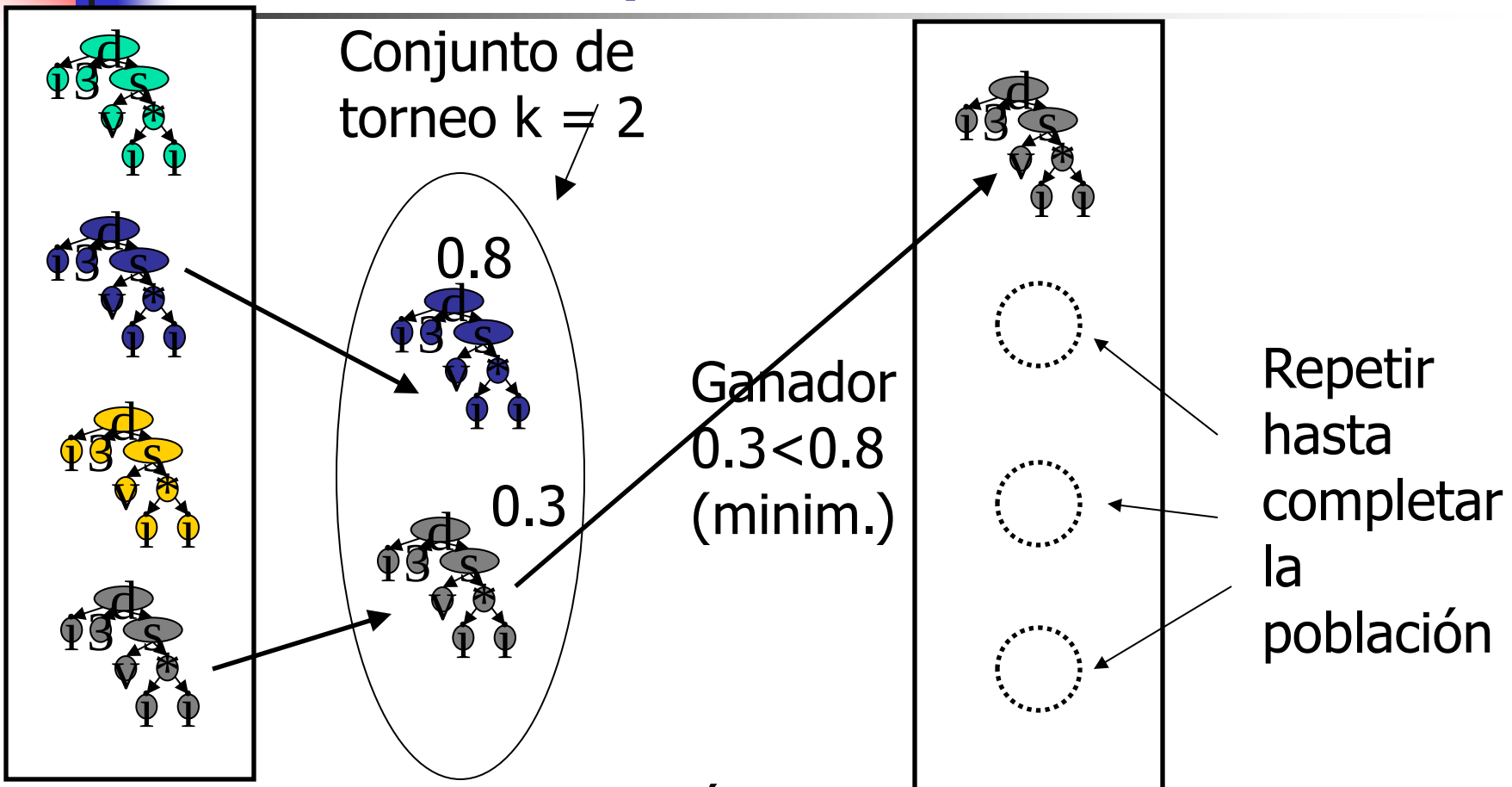
Proporcional a la fitness. Selección por ruleta



En media, los buenos individuos serán seleccionados más a menudo

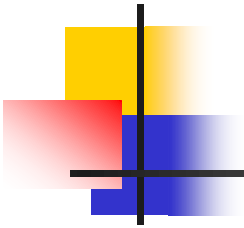
Problema: superindividuos, fin de la variedad, convergencia prematura

Selección por torneo



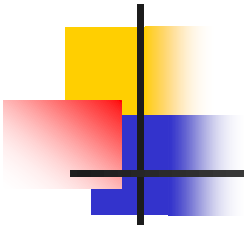
K controla la presión selectiva

A mas grande K, mas presión (4 está bien)



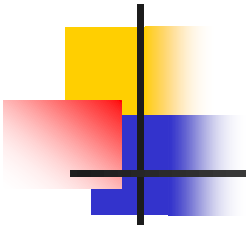
Parámetros de control

- Tamaño de la población (M: 500 a 10000)
- Máximo número de generaciones (G: 50 a 100)
- Probabilidades de recombinación, reproducción y 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.



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 al programa

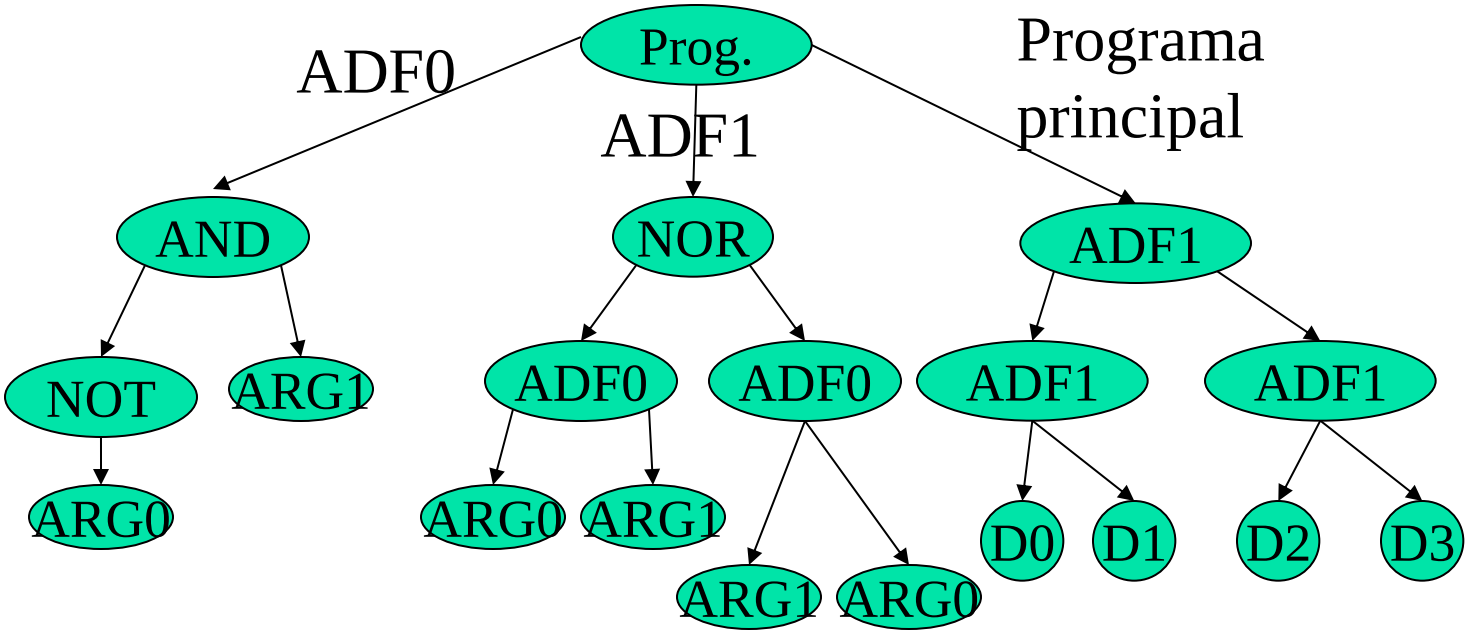


Evolución de Subrutinas

- Los programadores humanos simplifican la complejidad del problema escribiendo las subrutinas adecuadas y reutilizándolas en el código

ADFs (Automatically Defined Functions: subrutinas)

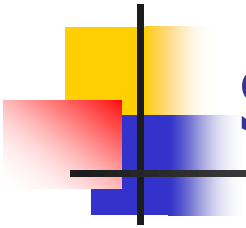
Nota: cruce homólogo



Esfuerzo y tamaño en paridad-par

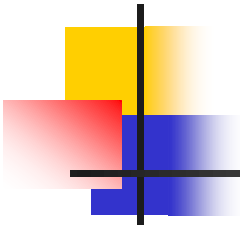
Paridad	Esfuerzo sin ADF	Esfuerzo con ADF	Tamaño sin ADF	Tamaño con ADF
3	96.000	64.000 (x1,5)	44,6	48,2
4	384.000	176.000 (x2,18)	112,6	60,1
5	6.528.000	464.000 (x14,07)	299,9	156,8
6	70.176.000	1.344.000 (x52,2)	900,8	450,3
7 a 11	NO	SI		

Trabajos sobre evolución de subrutinas

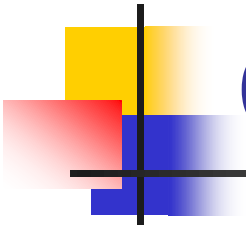


- Angeline PJ and Pollack JB. 1992. The Evolutionary Induction of Subroutines, *The Proceedings of the 14th Annual Conference of the Cognitive Science Society*.
- Rosca & Ballard. 1996. Discovery of Subroutines in Genetic Programming. Advances in Genetic Programming II.
- Ricardo Aler, David Camacho, Alfredo Moscardini. 2004. *"The Effects of Transfer of Global Improvements in Genetic Programming".* Computing and Informatics

¿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

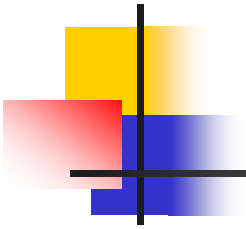


Conclusiones ADFs

- La PG puede evolucionar el programa principal y varias subrutinas
- Si el problema es lo suficientemente complicado, el esfuerzo computacional y la complejidad estructural (tamaño) del programa final disminuyen

Rational Allocation of Trials (RAT)

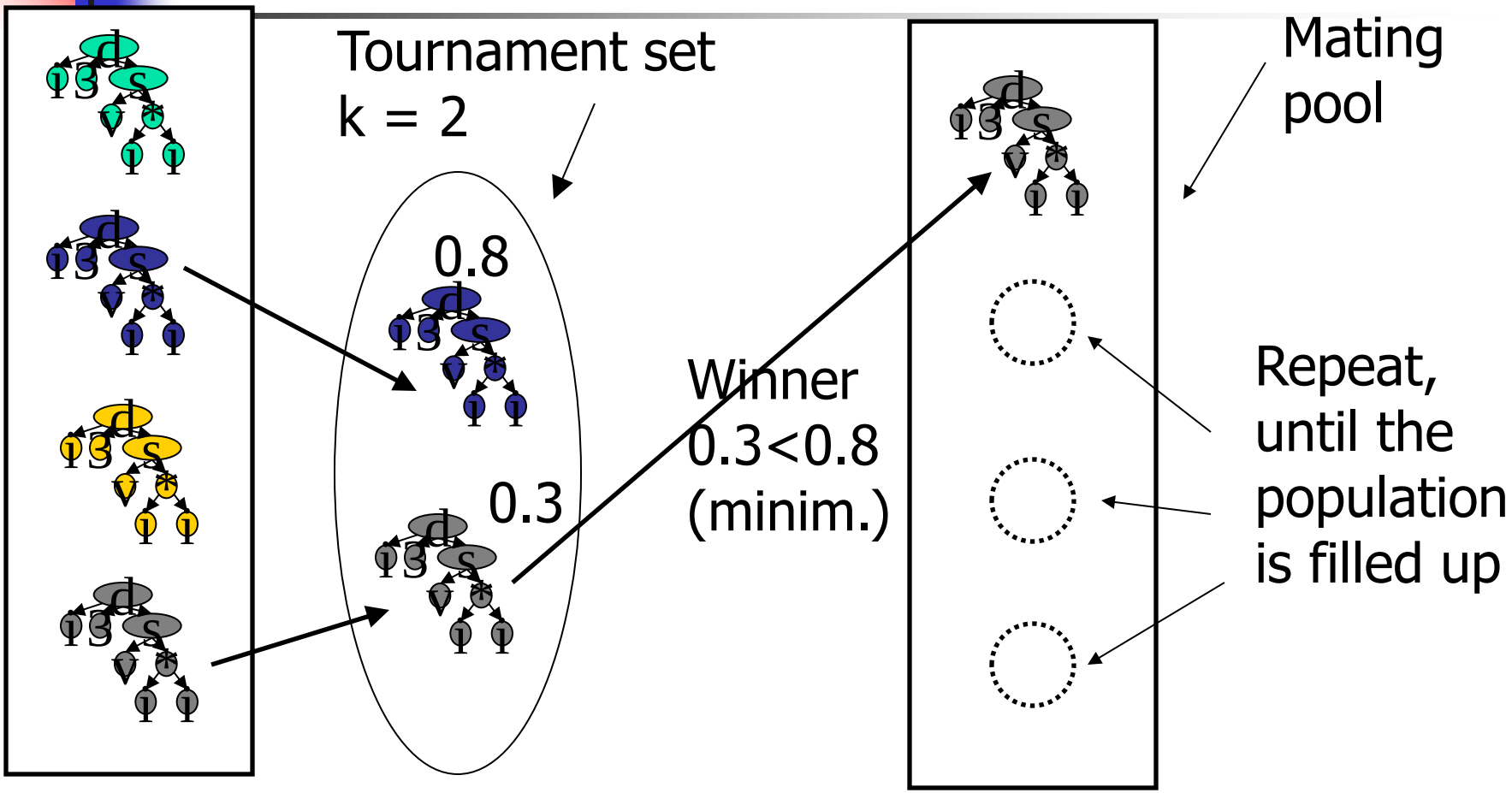
- Every fitness computation requires running many fitness cases. But in order to differentiate between good and bad individuals (as required by tournament selection), maybe not all fitness cases are needed. Using only the right number of fitness cases might improve efficiency.
- Teller, Andre. 1997. **"Automatically Choosing the Number of Fitness Cases: The rational Allocation of Trials".** *GECCO'97*
- Do not use all the fitness cases
- Use only as many as necessary to differentiate between good and bad individuals
- Every individual will evaluate the most appropriate number of fitness cases

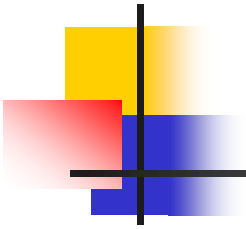


RAT Algorithm

- It uses tournament selection:
  1. Do *M* times (*M* is the population size)
     1. Pick *K* individuals at random from the population P_i
     2. From this set, place a copy of the individual *i* with highest approximated fitness into the mating pool
  2. P_{i+1} is created by the application of genetic operators to the mating pool

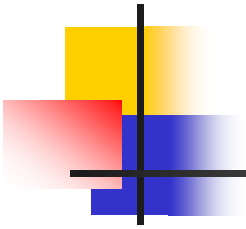
Tournament Selection





RAT Basic Idea

- If an individual *L* in a tournament is unlikely to become the winner, then do not evaluate more fitness cases for *L*
- If no other individual in the tournament set is likely to become the winner *W*, then do not evaluate more fitness cases for *W*



RAT Algorithm. Initialization

- Create M tournaments (all at once)
- Initialize contention list Q with all individuals (it contains individuals for which it is required to evaluate more fitness cases)
- Evaluate all individuals with T_{min} fitness cases (out of a total T fitness cases)

RAT Algorithm

- Do T-T_{min} times:
  - Remove any individual X from Q if for every tournament t that X is in:
    - X is not in the first place
    - AND it is *not likely* to become the winner

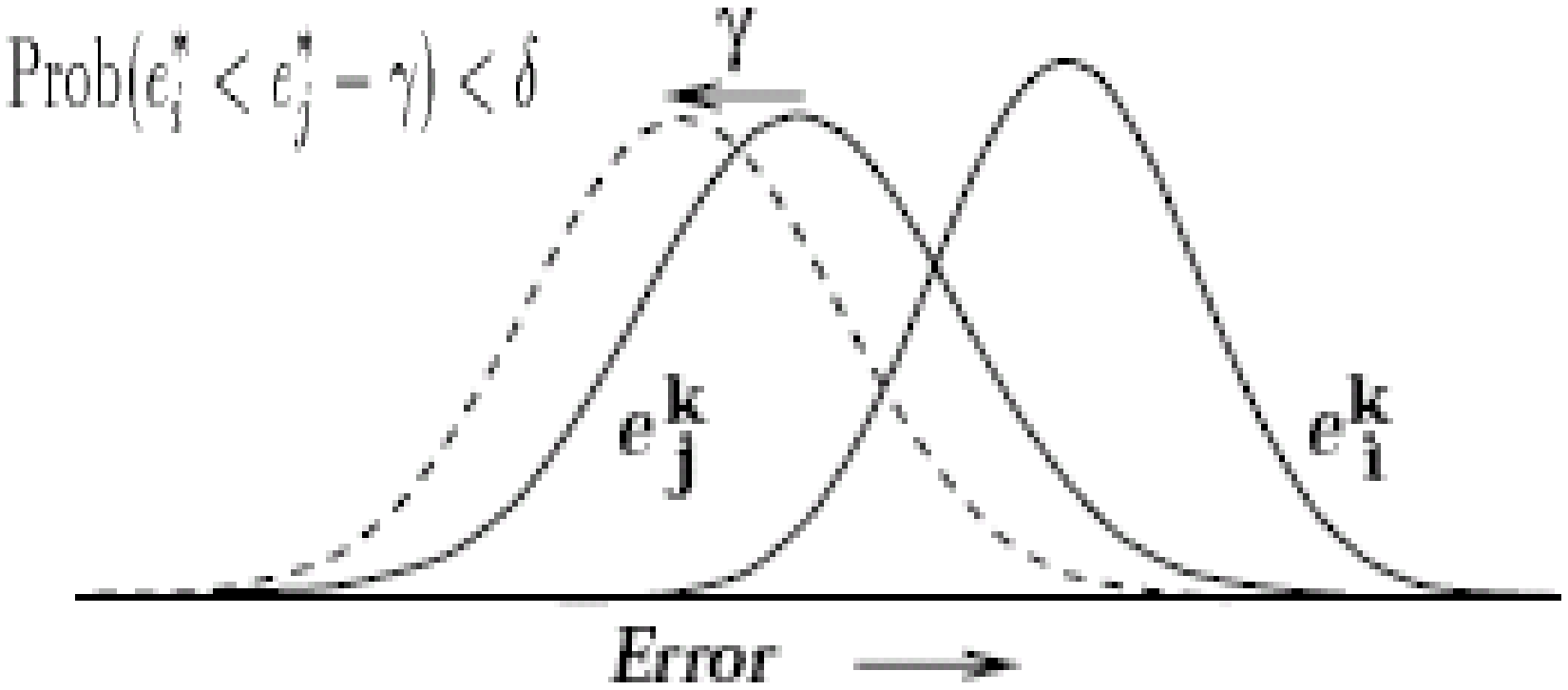
    OR
    - X is in the first place (temporary winner)
    - AND it is not likely that other individuals in tournament t become better than X
  - Evaluate all individuals still in list Q on the current training example

RAT Algorithm

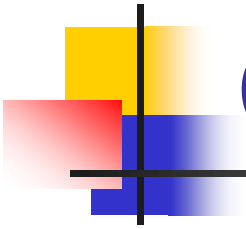
- How to determine if an individual *i* is likely (or not) to be better than another individual *j*?
- If both individuals have been evaluated on a sample with *k* fitness cases
- Then, the average error (e^k_i, e^k_j) and standard deviation can be estimated from the sample
- Assuming normality, we can compute the probability that the true error of *i* (e^*_i) is smaller than the true error of *j* (e^*_j).
- If $Pr(e^*_i < e^*_j - t)$ is small then *i* is not likely to be better than *j*

RAT Algorithm

- The gaussians are the distributions of the true error (assuming normality)

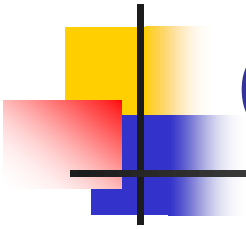


Iteración en Programación Genética



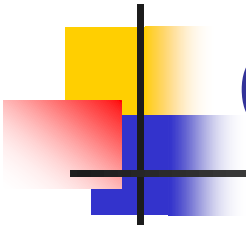
- Resultado teórico:
  - Teller. 1994. “Turing Completeness in the Language of Genetic Programming with Indexed Memory ”. Proceedings of the 1994 IEEE World Congress on Computational Intelligence
- GP+IM (Genetic Programming + Indexed Memory (arrays))

Iteración en Programación Genética [Teller, 1994]



- Supongamos que disponemos de:
- (IF X THEN Y ELSE Z)
- (= X Y)
- (AND X Y)
- (ADD X Y), (SUB X Y)
- Memoria indexada (array):
  - (Read X), (Write Y X)

Iteración en Programación Genética [Teller, 1994]

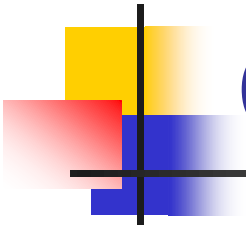


- Entonces, cualquier algoritmo se puede expresar como:

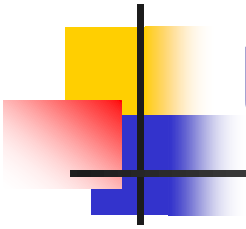
REPEAT <GP+IM function>

UNTIL <ocurre cierto estado en la memoria>

Iteración en Programación Genética [Teller, 1994]



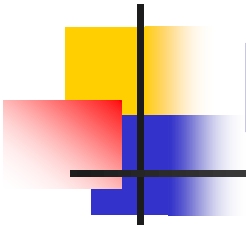
- Es decir, en principio no son necesarios los bucles para expresar algoritmos iterativos
- Sólo hay que evolucionar una función GP+IM que use memoria indexada
- E iterarla hasta que se alcance cierto estado en la memoria
- Pero en la práctica es posible que los programas sean más fácilmente expresables con bucles que con GP+IM



Uso de Bucles

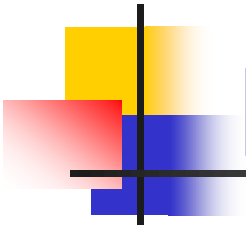
- Uso de bucles: añadir función que implementa el bucle:
  - (bucle veces cuerpo)
  - (bucle 10 (assigna-m (* (lee-m) i))

Limitaciones de los bucles/recursividad



- Con éxito en algunas aplicaciones
- A diferencia de las ADFs, no tan bien estudiados
- Incrementan **enormemente** el tiempo de evaluación de la fitness (incluso no terminación)
- Programas recursivos o iterativos son extremadamente **frágiles**

Exploración del espacio de programas Turing-completos



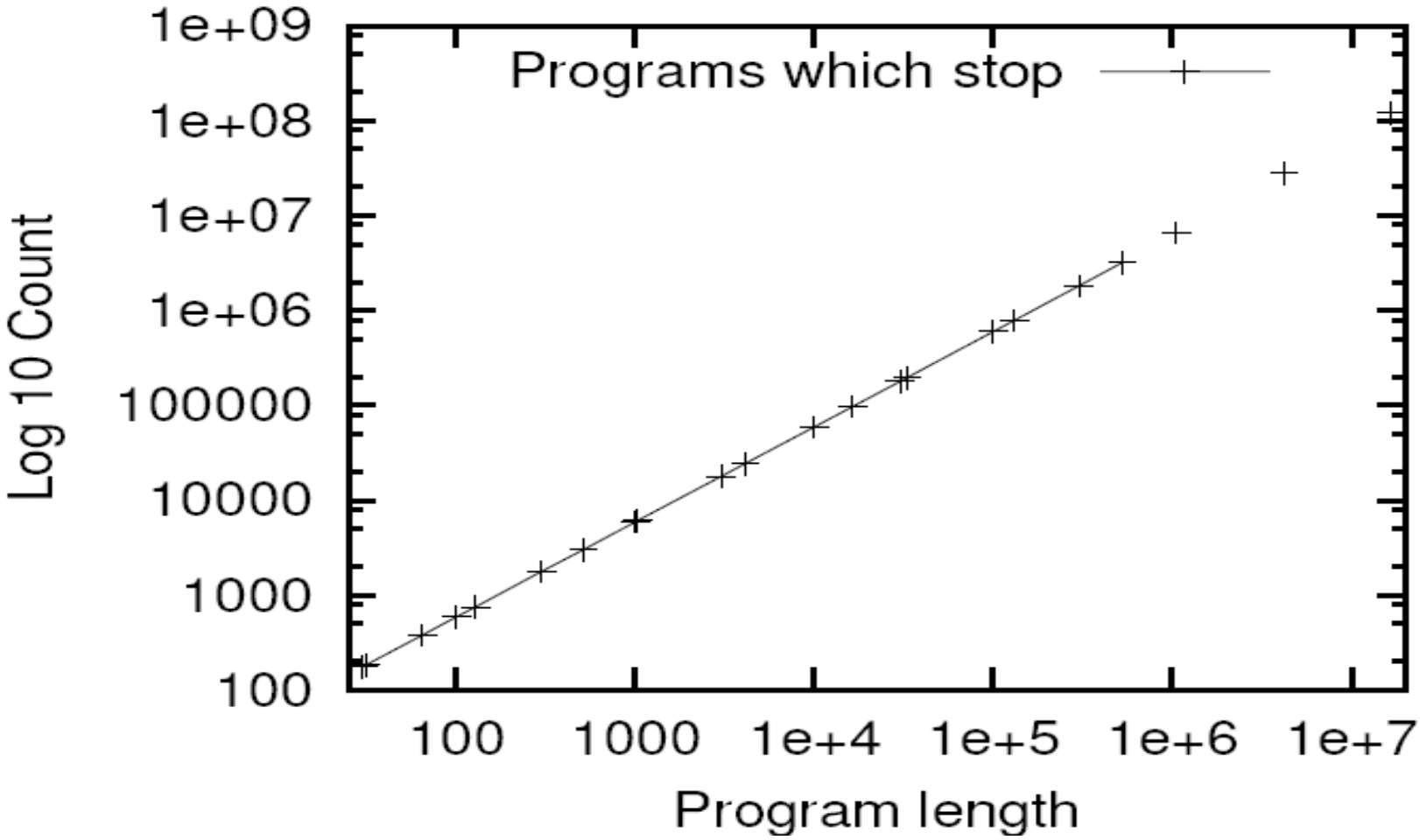
- W. B. Langdon and R. Poli. **The Halting Probability in von-Neumann Architectures**. EuroGP' 06
- Generan aleatoriamente programas en código máquina, de distinta longitud
- Codificación lineal

Instrucciones máquina utilizadas

Table 1. T7 Turing Complete Instruction Set

Instruction	*#operands*	*operation*	*v* set	Every ADD operation either sets or clears the overflow bit *v*.
ADD	3	A + B→C	v	
BVS	1	#addr→pc if *v*=1		LDi and STi, treat one of their arguments as the address of the data. They allow array manipulation without the need for self modifying code. (LDi and STi data addresses are 8 bits.)
COPY	2	A→B		
LDi	2	@A→B		
STi	2	A→@B		
COPY_PC	1	pc→A		
JUMP	1	addr→pc		To ensure JUMP addresses are legal, they are reduced modulo the program length.

Programas que terminan



Número de Programas que Terminan

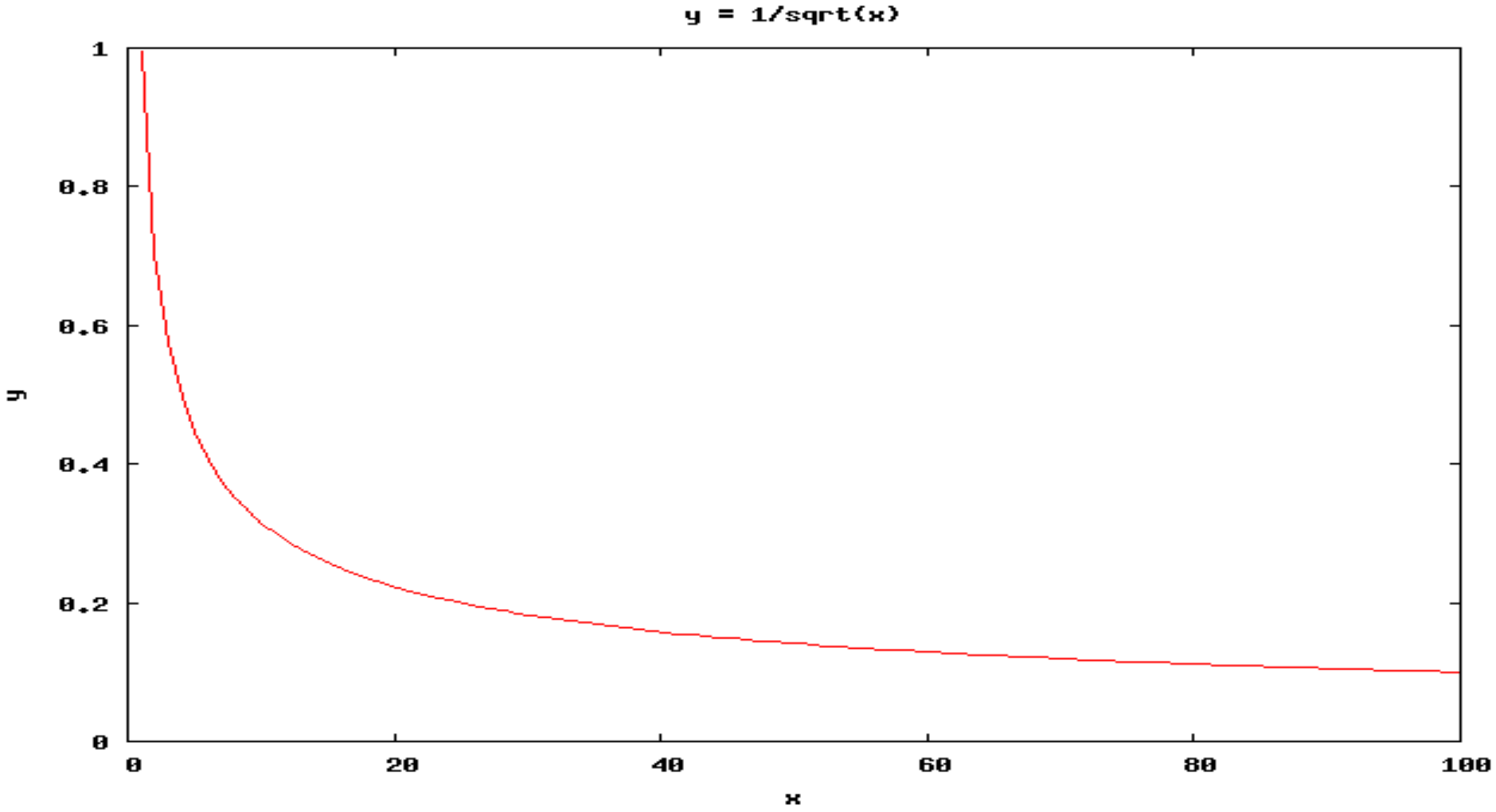
- El número de programas que terminan crece exponencialmente con la longitud
- Pero el número de programas totales crece muchísimo más rápidamente
- La proporción de programas que terminan es:

$$1/\sqrt{\text{length}},$$

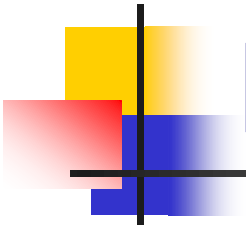
- El tiempo de ejecución de los programas que terminan es proporcional a:

$$\sqrt{\text{length}}.$$

Proporción de Programas que Terminan

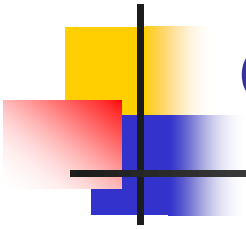


Soluciones a los problemas de bucles y recursividad



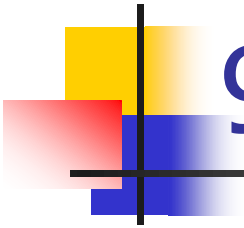
- Limitar:
  - Tiempo de ejecución
  - Número de bucles
  - Número de iteraciones o llamadas recursivas (de manera proporcional al tamaño de la entrada)
  - Anidamiento de bucles
- Modelo de corutinas [Maxwell, 94] Ejecutar individuos en paralelo y cancelar aquellos mucho peores que los ya encontrados

Modelo de corutinas [Maxwell, 94]

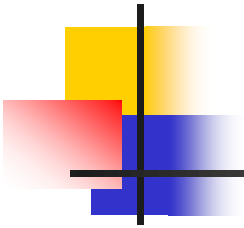


- Maxwell. 1994. “Experiments with a Coroutine Execution Model for Genetic Programming”. IEEE World Congress on Computational Intelligence. 413-417

Modelo de corutinas [Maxwell, 94]

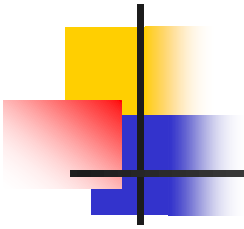


- Problema de limitar tiempo, iteraciones, etc:
  - Puede que no permita encontrar la solución (nos podemos quedar cortos o largos)
- Modelo de corutinas:
  - Permitir la ejecución en paralelo (real o simulado) de los programas
  - No limitar su tiempo de ejecución
  - Modelo steady-state: los individuos buenos / rápidos reemplazan a los malos / lentos



Modelo de corutinas

- Si un individuo ha conseguido en el mismo tiempo una fitness mayor que otro, el primero reemplaza al segundo
- Necesita que los individuos devuelvan la fitness conseguida hasta el momento de manera continua (ej: fitness acumulada en los casos de prueba resueltos hasta el momento)
- Sólo se pueden comparar individuos de la misma edad (tiempo de ejecución)

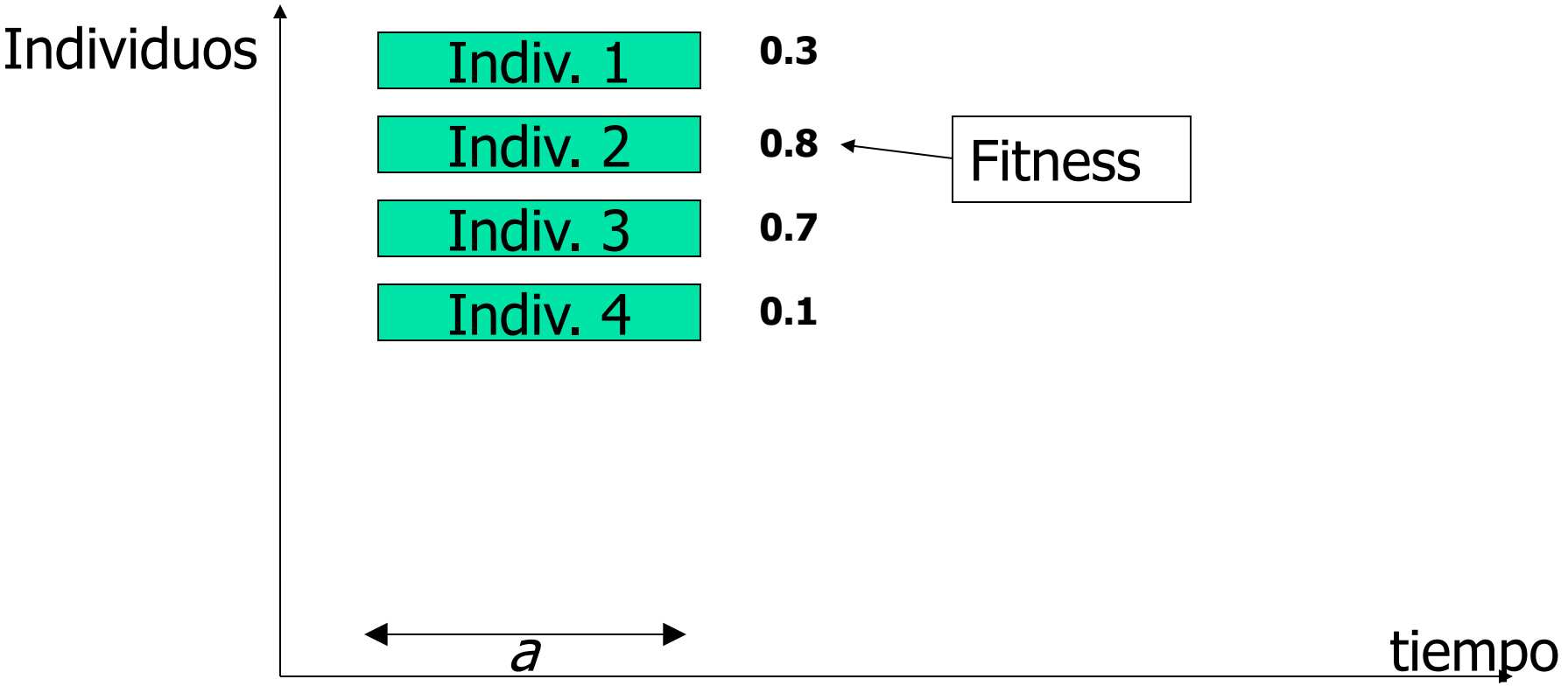


Modelo de corutinas

1. Creación de la población inicial
2. Ejecutar cada individuo un tiempo *a*
3. Si se consigue el éxito, terminar
4. Crear N nuevos individuos (selección-torneo, mutacíón, cruce)
5. Ejecutar los N individuos hasta que tengan la misma edad que los de la población
6. Reemplazar antiguos individuos por los N nuevos mediante torneo
7. Ejecutar individuos otro periodo a
8. Volver a 3

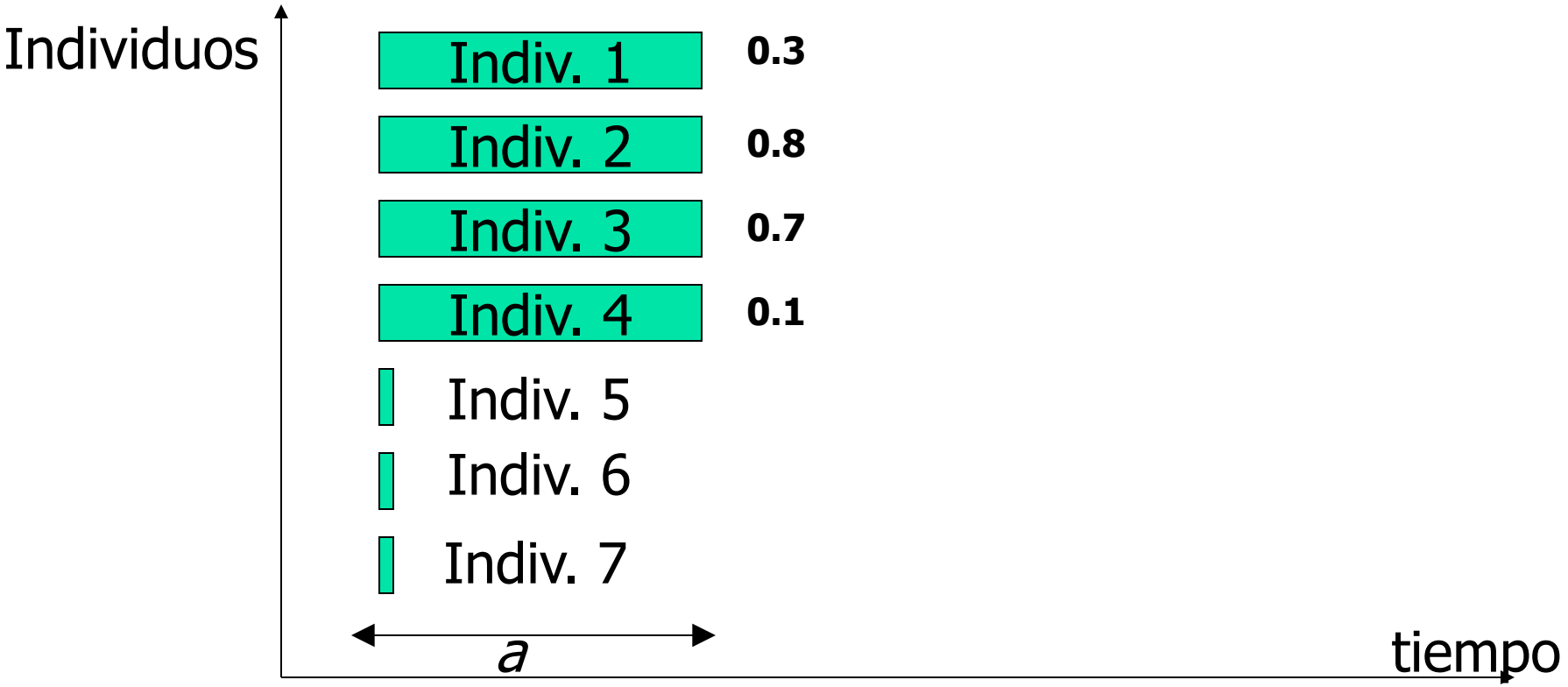
Modelo de Corutinas

- Crea población inicial y ejecútala durante a segundos. Computar fitness parcial.



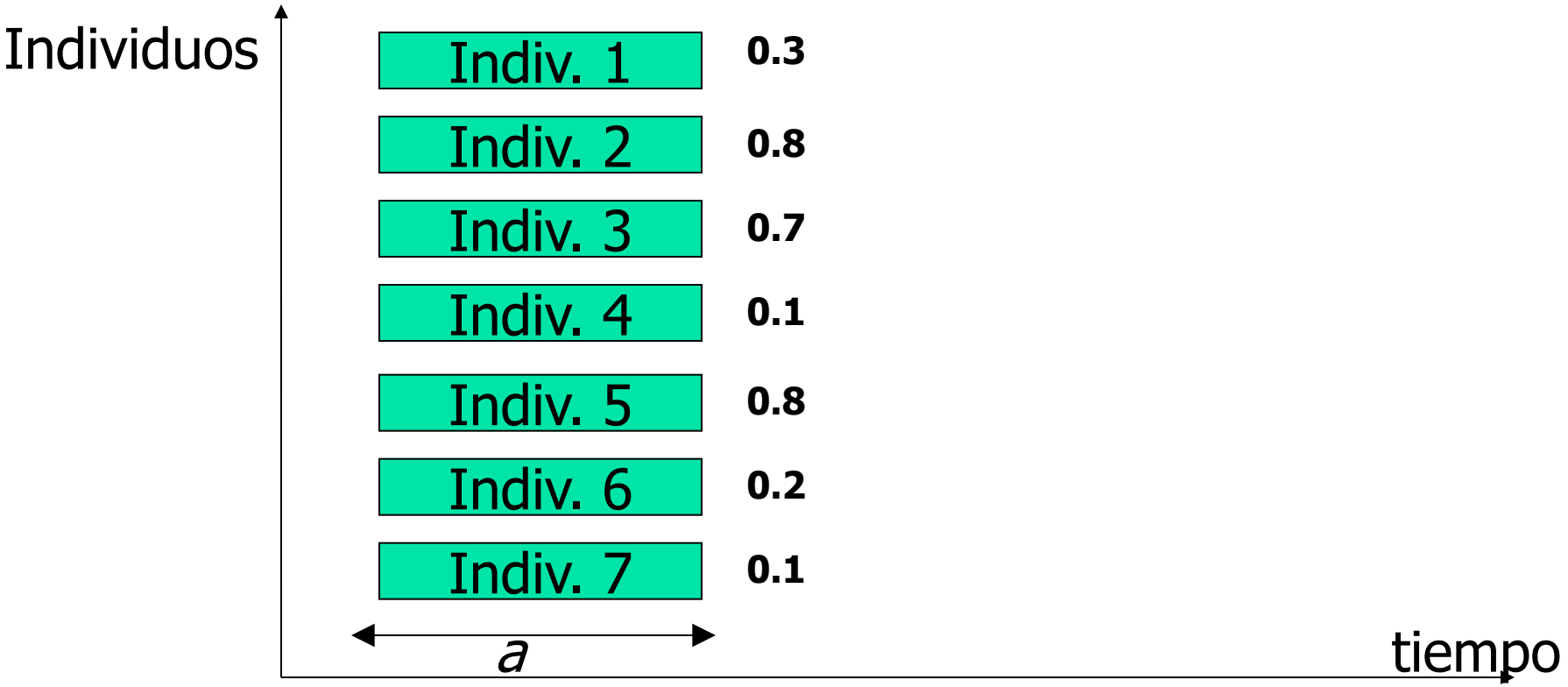
Modelo de Corutinas

- Crea N nuevos individuos por medio de torneo y operadores genéticos.



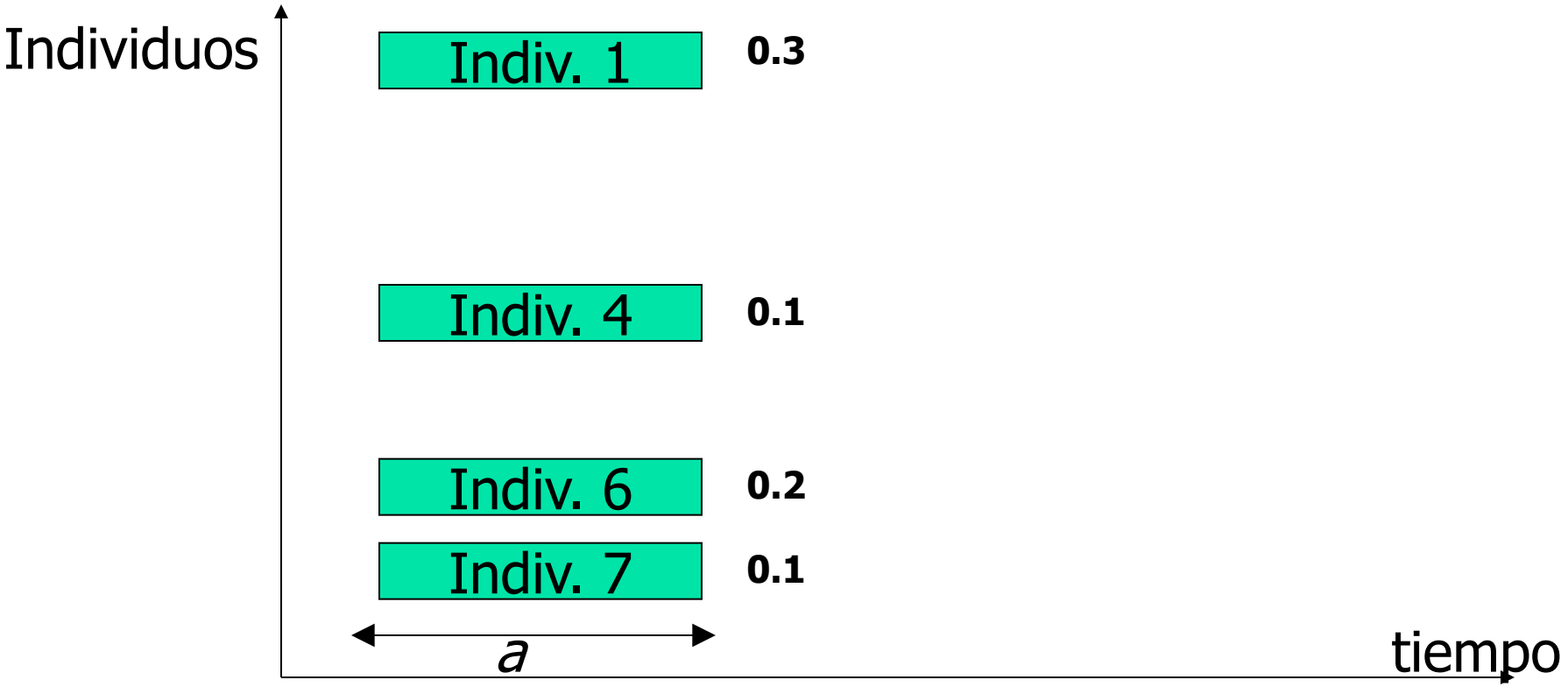
Modelo de Corutinas

- Ejecuta los N individuos el mismo tiempo que el resto



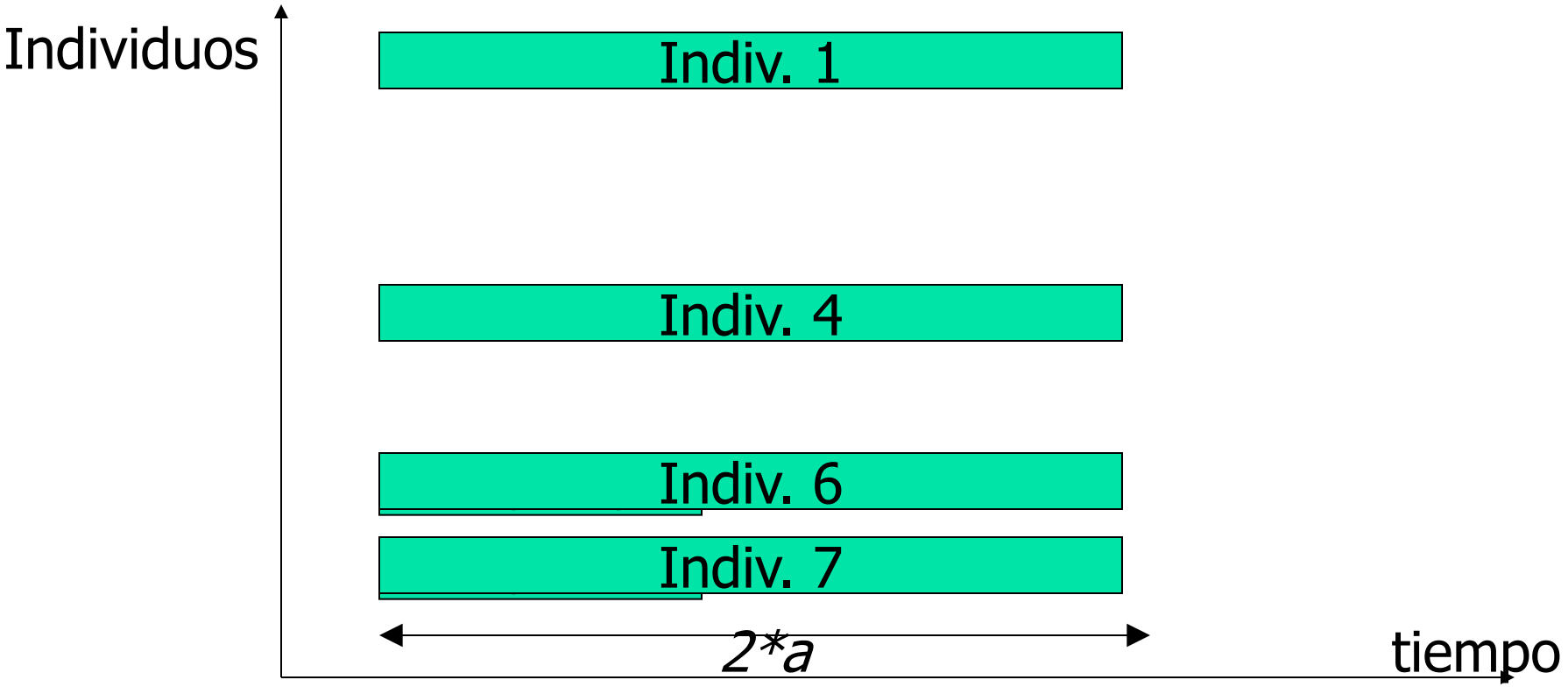
Coroutine Model

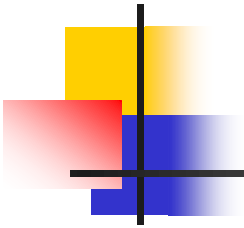
- Borra los peores por torneo



Coroutine Model

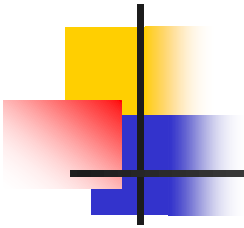
- Ejecútalos otro tiempo *a*. Etcétera ...





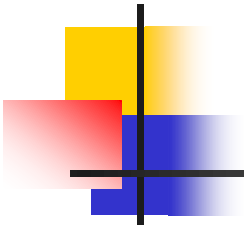
Modelo de corutinas

- Requiere un sistema que permita ejecutar un programa durante un tiempo, quitarlo de la CPU, guardar su estado, y volverlo a ejecutar más adelante
- Comienza a funcionar bien en cuanto aparece al menos un individuo que computa todos los casos de prueba en un tiempo finito
- Parece generar individuos más eficientes
- Pero no hay garantía de que un individuo “rápido” vaya a ser el mejor a la larga



Modelo de corutinas

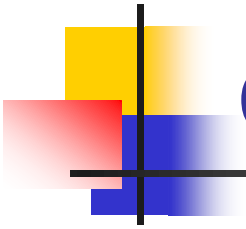
- Sólo funciona si podemos evaluar la fitness conseguida "hasta el momento" de un individuo
- Puede ser interesante intentar construir programas que consiguen un resultado parcial en alguna variable global, en lugar de aquellos que esperan hasta el final para dar una solución (*Anytime* [Teller, 94])



Restricciones: sintáxis y tipos

- La PG estándar requiere “closure”: cualquier función debe aceptar cualquier tipo y valor.
  - Ej: función “/” protegida 3/0 = 1
  - Ej: “perro” + 4 = 4
- El espacio de búsqueda es mayor de lo necesario (esto no siempre es problemático)
- Las soluciones encontradas son poco naturales:
  - Ej: if (3+”cadena”) then {10/0}
- Solución: utilizar gramáticas o tipos

Trabajos principales con gramáticas



- F. Gruau. 1996. “On using syntactic constraints with genetic programming”. Advances in Genetic Programming III.
- P. A. Whigham. 1995. Grammatically-based Genetic Programming. Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications.

Restricciones de sintaxis: gramáticas libres de contexto

```
<axiom> ::= <DNF>

<DNF>[0..6] ::= or (<term>)(<DNF>) | <term>

<term>[0..3] ::= and (<literal>)(<term>) | <literal>

<literal> ::= <letter> | not (<letter>)

<letter> ::= A | B | C | D
```

[Gruau, 96]

Ejemplo de individuo generado:

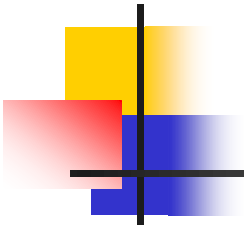
DNF -> (OR (<TERM>) (<DNF>)) ->

(OR (AND (<LETTER> <LETTER> <LETTER>) <DNF>) ->

(OR (AND (A B C) <DNF>) -> (OR (AND (A B C) <TERM>) -> ... ->

(OR (AND (A B C)) D)

(<DNF> (<TERM> (<LETTER> <LETTER> <LETTER>)) <LETTER>)

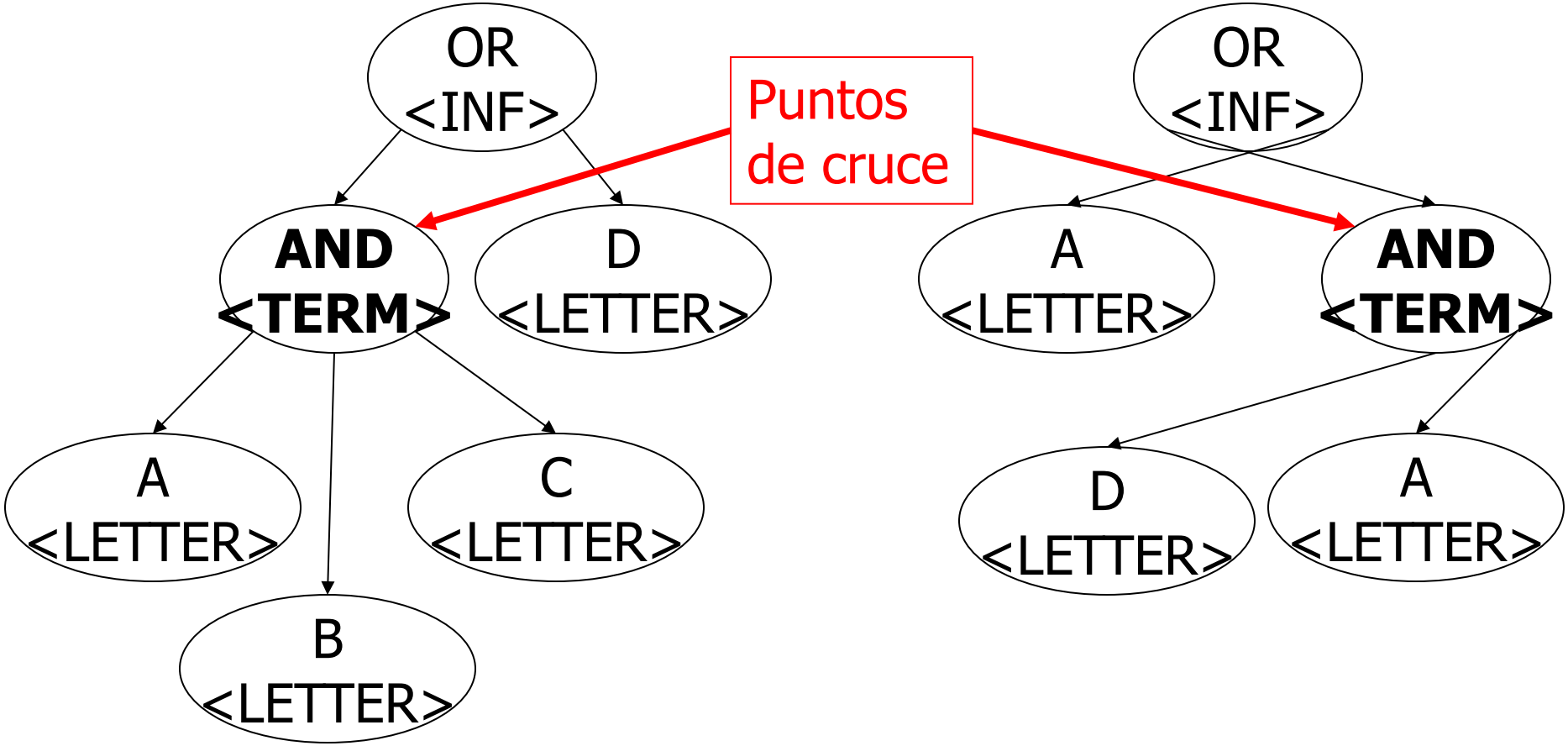


Uso de la gramática

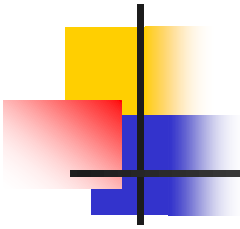
- Para generar los individuos de la población inicial (se utilizan las reglas de manera aleatoria)
- Resulta algo más complicado generar individuos de una profundidad prefijada
- Para generar individuos correctos con el cruce (se eligen dos puntos de cruce que puedan ser generados por la misma regla. ej: A puede ser intercambiado por otro <letter>, como C)
- Es necesario especificar cuántas veces se puede utilizar una regla para limitar el tamaño final del árbol

Ejemplo de cruce con gramática

Seleccionar en el segundo padre un nodo del mismo tipo que el nodo seleccionado en el primer padre

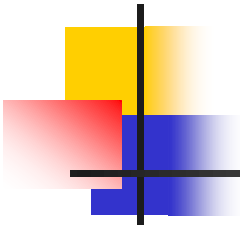


Evolución de algoritmos de ordenación



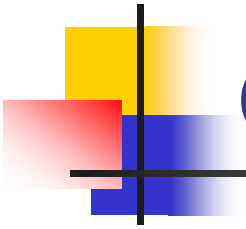
- Kinnear. 1993. **"Generality and Difficulty in Genetic Programming: Evolving a Sort".** *Fifth International Conference on Genetic Algorithms*
- Se evolucionó algoritmos de orden $O(N^2)$
- Fitness function: compleja, pero basicamente cuenta el número de inversiones (swaps): parejas de números desordenadas

Evolución de algoritmos de ordenación [Kinnear, 93]



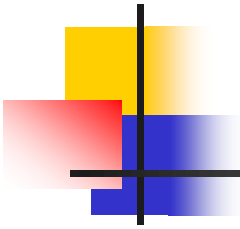
- Population size: 1000
- Generations: 50
- Maximum initial depth: 6
- Fitness cases: 15 (5 fixed and 10 random). Maximum list length: 30
- Probability of success: from 40% (low-level primitives) to 100% (high-level primitives) of runs generated a correct individual

Evolución de algoritmos de ordenación [Kinnear, 93]



- Primitives (de muy bajo nivel):
  - if(test) {body}
  - x<y
  - Swap(x,y)
  - for(start, end, body), index
  - x+1, x-1, x-y
  - *leng*: length of the list of integers

Evolución de algoritmos de ordenación [Ciesielski, Li, 04]



- Ciesielski, Li. 2004. **"Experiments with Explicit For-loops in Genetic Programming".** *CEC'04.*
- No consiguieron evolucionar un algoritmo de ordenación general

Evolución de algoritmos de ordenación [Spector, Klein, Keijzer, 05]

- Spector, Klein, Keijzer. 2005. **"The Push3 Execution Stack and the Evolution of Control"**. GECCO'2005
- Stack-based GP
- Evolucionaron programas generales para invertir una lista, factorial, fibonacci, n-even-parity y ordenación de listas

Evolución de algoritmos de ordenación [Spector, Klein, Keijzer, 05]

- Complejidad: $O(N^2)$: $N*(N-1)/2$
- Fitness cases: listas de 4 to 8 enteros

Evolución de algoritmos de ordenación [Spector, Klein, Keijzer, 05]

- Primitivas (de bajo nivel):
  - List[i] (accesses position i of the list)
  - Length (of the list to be sorted)
  - Swap(i,j) =
    - List[i] = List[j], List[j] = List[i]
  - Max(i,j) = Max(List[i],List[j])

Evolución de algoritmos de ordenación [Agapitos, Lucas, 2006]

- http://ieeexplore.ieee.org/iel5/11108/35623/01688643.pdf?tp=&isnumber=35623&arnumber=1688643
- **Evolving Efficient Recursive Sorting Algorithms,** 2006 IEEE Congress on Evolutionary Computation
- Alexandros Agapitos, Simon M. Lucas

Evolución de algoritmos de ordenación [Agapitos, Lucas, 2006]

Primitivas de mas alto nivel

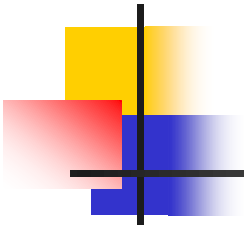
PRIMITIVE ELEMENTS FOR EVOLVING SORTING ALGORITHMS

Method set		
Method	**Argument(s) type**	**Return type**
Head	CList	Comparable
Tail	CList	Comparable
Append	CList, CList	CList
Cons	Object, Object	CList
Compare	Comparable, Comparable	Boolean
EqualTo	Object, Object	Boolean
Filter	CList, MyComp, Comparable	CList

Conditional		
Control flow	**Argument(s) type**	**Return type**
IF-Then-Else	Boolean, CList, CList	CList

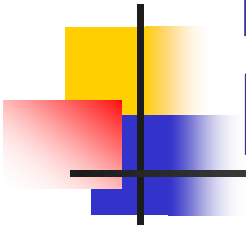
Terminal set		
Terminal	**Value**	**Type**
Parameter[0]	-	CList
Parameter[1]	-	MyComp
Parameter[2]	-	Comparable
Const: GreaterThan	new GreaterThan()	MyComp
Const: NotGreaterThan	new NotGreaterThan()	MyComp
Const: null	null	Object

Evolución de algoritmos de ordenación [Agapitos, Lucas, 2006]

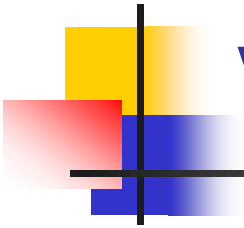


- Casos de fitness: 10 listas aleatorias de 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 elementos únicos, elegidos del rango 0-250
- Población: 12000 individuos
- Cada ejecución: varias horas
- Función de fitness: prueban varias, de mas complejidad que las anteriores (ej: distancia media entre inversiones)

Evolución de algoritmos de ordenación [Agapitos, Lucas, 2006]



- Resultados: consiguen evolucionar programas de orden O(n*log(n)) (como Quicksort, el más eficiente conocido)



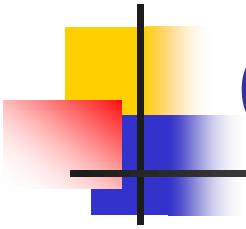
Variantes de PG

- Evolución de estructuras de datos [Langdon, 98]
- **Evolución de código máquina (GP lineal)** [keller, 96], [friedrich, 97]
- Immune Programing [Musilek, 06]: hipermutación adaptativa
- Stack-based GP (lineal) [Perkins, 94] [Spector]
- Cartesian Genetic Programming [Miller et al, 03]
- http://sc.snu.ac.kr/courses/2006/fall/pg/aai/Papers2.html

Antena diseñada por developmental GP (embriones)

- La PG evoluciona programas que construyen el circuito o la antena
- Montados en satélites ST5, lanzados en Marzo del 2006

Conclusiones Programación Genética



- Evolución de expresiones funcionales
- Dificultad con los bucles y la recursividad
- Buena idea: utilidad probada de las ADFs
- Éxito en problemas reales
- Operadores genéticos tal vez poco apropiados