Programación Genética: control del bloat y uso de gramáticas

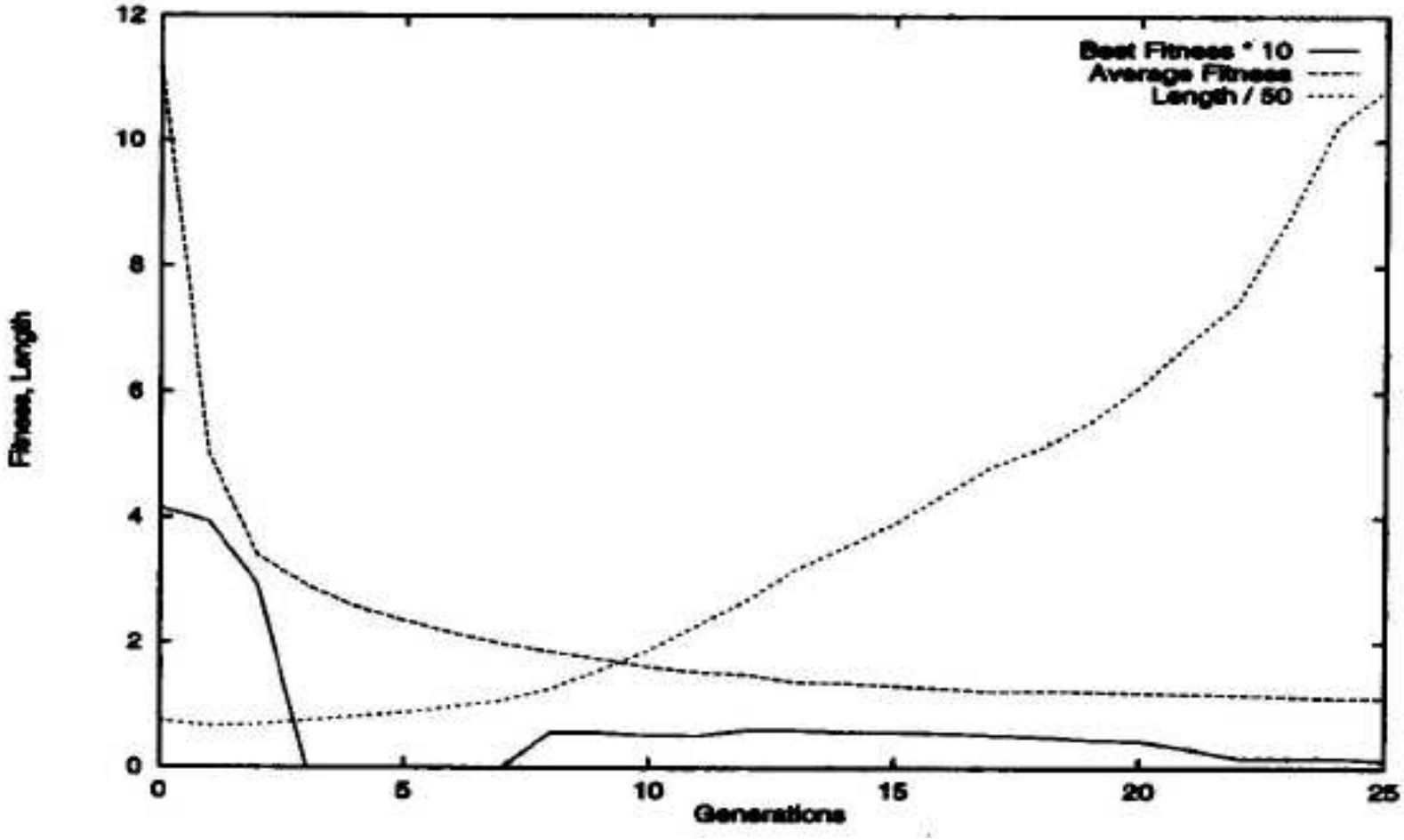
Bloat (“crecimiento”)

- A medida que avanzan las generaciones, el tamaño de los individuos crece (sin necesidad de que también mejore la fitness)
- Crecimiento rápido, prácticamente exponencial [Langdon, 00]
- Está principalmente causado por “intrones” (código que no hace nada).
  - Ej: a+ (a-a+a-a), 0*(a*b+c*d), if(False) then {...}

Bloat (“crecimiento”)

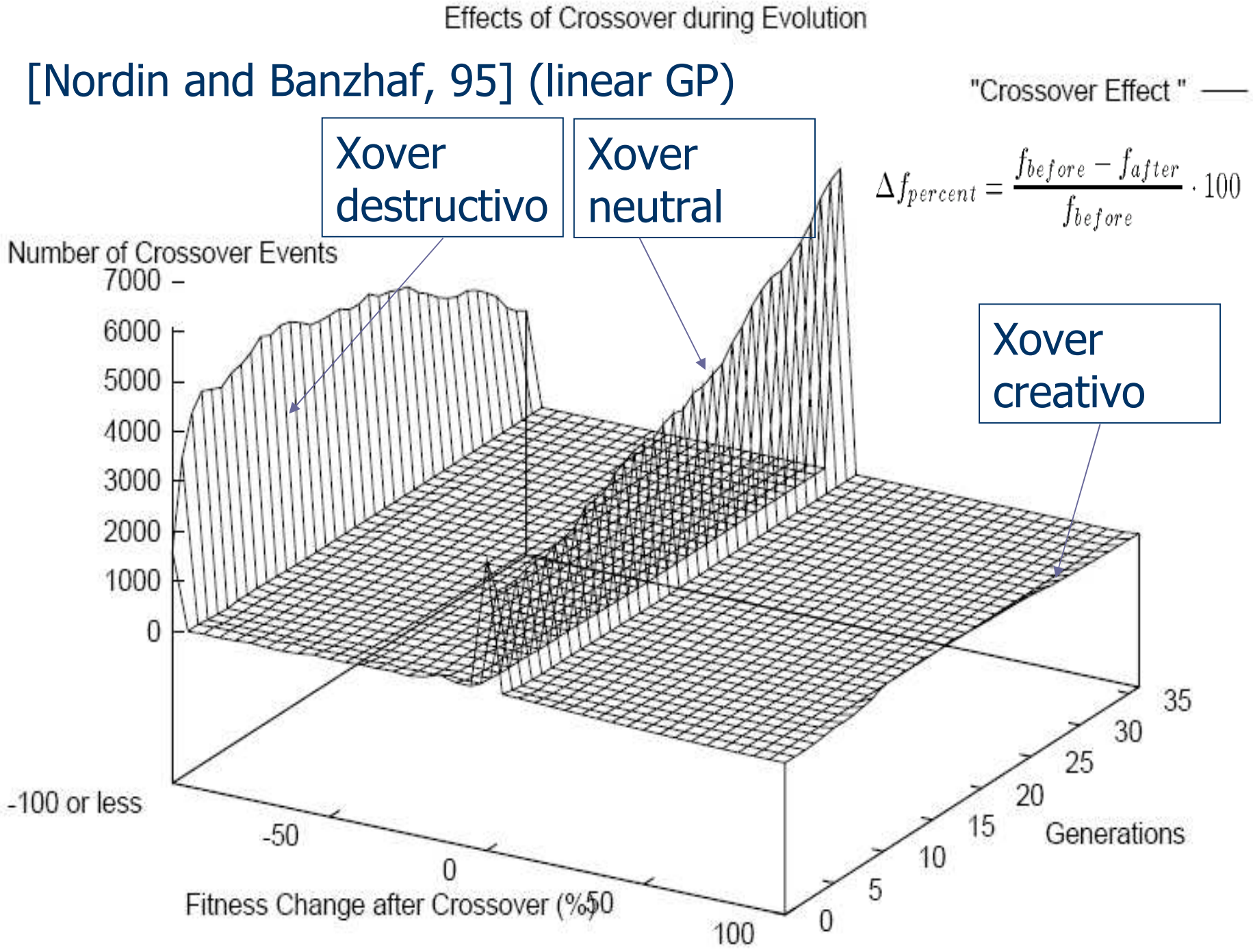
- Esto ocasiona a veces que tarden más en ejecutar
- Y ocupen más memoria
- Y la búsqueda se estanca (GP converge)
- Y sean menos legibles los programas

Bloat. Desarrollo de una ejecución típica



Bloat (“crecimiento”). ¿Porqué?

- El operador de cruce permite que los hijos sean más profundos que los padres
- ***Accuracy theory*: Defensa contra el crossover, especialmente hacia el final de la ejecución, cuando es difícil mejorar la fitness**
- *Removal bias theory*: Es más fácil añadir subárboles grandes en los intrones (que suelen estar cerca de las hojas) que quitar subárboles grandes
- Superado cierto tamaño límite, para la misma fitness hay programas de muy diverso tamaño. Como hay más programas grandes que pequeños, los programas tienden a crecer


Effects of Crossover during Evolution
[Nordin and Banzhaf, 95] (linear GP)
"Crossover Effect "
$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100$
Xover destructivo
Xover neutral
Xover creativo
Number of Crossover Events
7000
6000
5000
4000
3000
2000
1000
0
-100 or less
-50
0
50
100
Fitness Change after Crossover (%)
0
5
10
15
20
25
30
35
Generations

Control del Bloat

- Parsimonia: penalizar a los individuos en la función de fitness (suponiendo minimización):
  - F'(x) = F(x) + k*nodos-en(x)
  - Problema: se supone que los individuos grandes son malos en general, pero lo que queremos es limitar el crecimiento sin sentido (que no lleve aparejado una mejora en fitness)
- Limitar el tamaño máximo (¿pero cuál?):
  - Problema: los árboles van a tender a crecer rápidamente hasta el tamaño máximo.

Control del Bloat

- Hacer torneos con fitness y si hay empate, elegir a los pequeños
  - Esto **no** funciona bien si la fitness es contínua (el individuo puede mejorar muy poco, pero engordar mucho)
- Evitar el cruce destructivo (que se produzcan hijos peores que los padres):
  - Ej: ejecutar la operación de cruce N-veces y quedarse con los mejores hijos (brood recombination)
  - Para hacer el brood eficiente, los hijos se evalúan con menos casos de fitness

Control del Bloat. Método Tarpeian

```
IF size(program) > average_pop_size AND random_int MOD n = 0
THEN
     return( very_low_fitness );
ELSE
     return( fitness(program) );
```

GP "canónico"

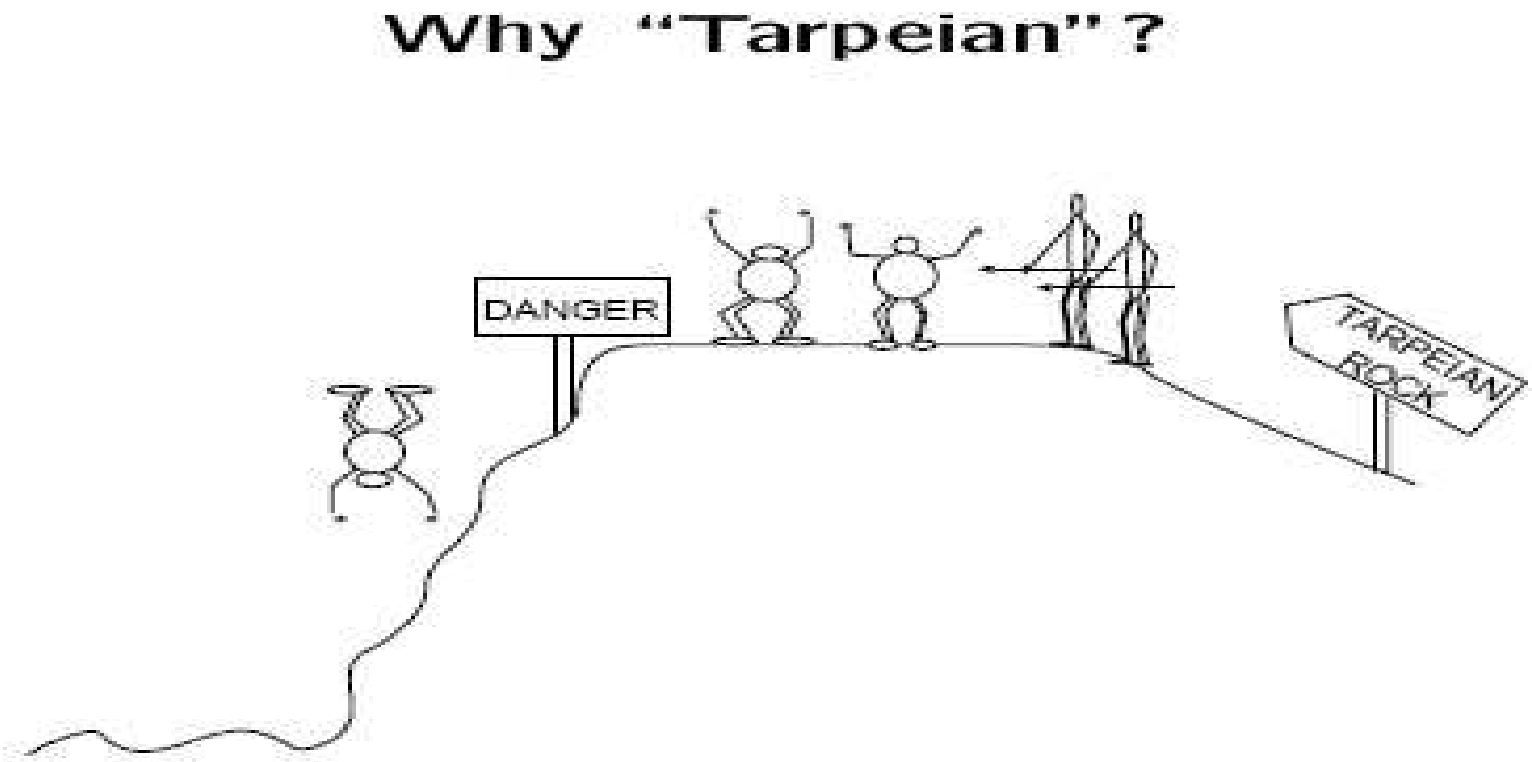
Elimina aquellos programas de longitud mayor que la media, de vez en cuando (con probabilidad 1/N)

n >= 2 (n=2, la mitad de los programas grandes, muere)

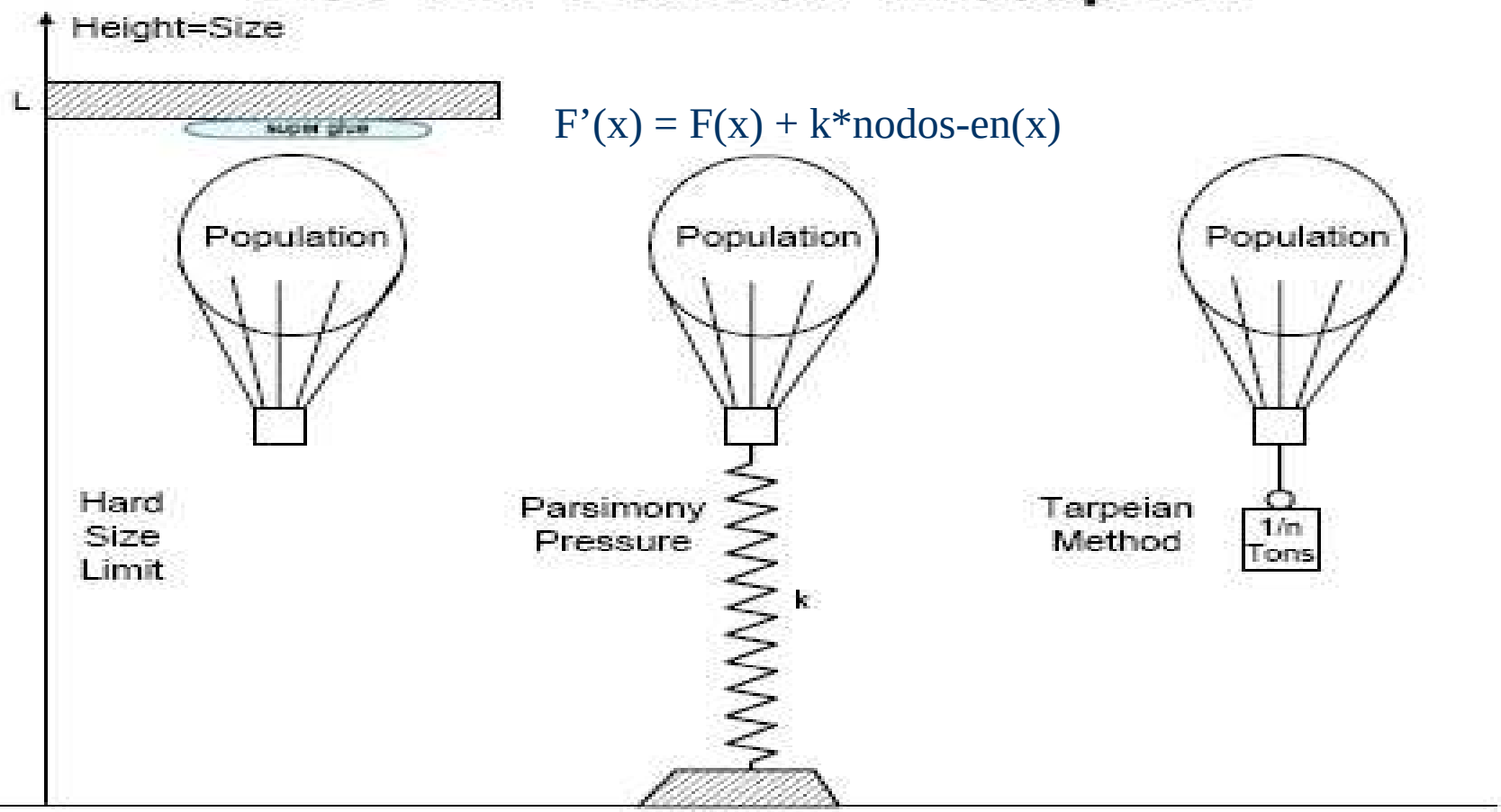
Control del Bloat. Método Tarpeian

- Si se crea un programa más grande que la media, tiene grandes posibilidades de ser “asesinado” (se le asigna una fitness mala), con probabilidad 1/*n* (a *n* menor, más fácil morir)
- Si sobrevive y es mejor (fitness) que la media, se reproducirá, y el tamaño medio de la población se incrementará
- Pero seguirá habiendo una presión contraria a crecer

Tarpeian: se crea un “agujero” para individuos grandes



Tarpeian: metáfora del globo



Características de Tarpeian

- Salva tiempo de cómputo en los individuos grandes (porque algunos no se evalúan)
- Justificación teórica basada en el teorema del esquema
- El método impide que se generen programas más grandes si y sólo si no son mejores que los cortos

Características de Tarpeian

- El método es demostrablemente válido **sin** mutación de subárboles (pero sí con mutación puntual)
- Hay que tener cuidado con la convergencia prematura, que se produce con:
  - Poblaciones pequeñas
  - Presión selectiva grande (ej: conjuntos de torneos grande)
  - Sesgo contra individuos grandes (*n* pequeño)

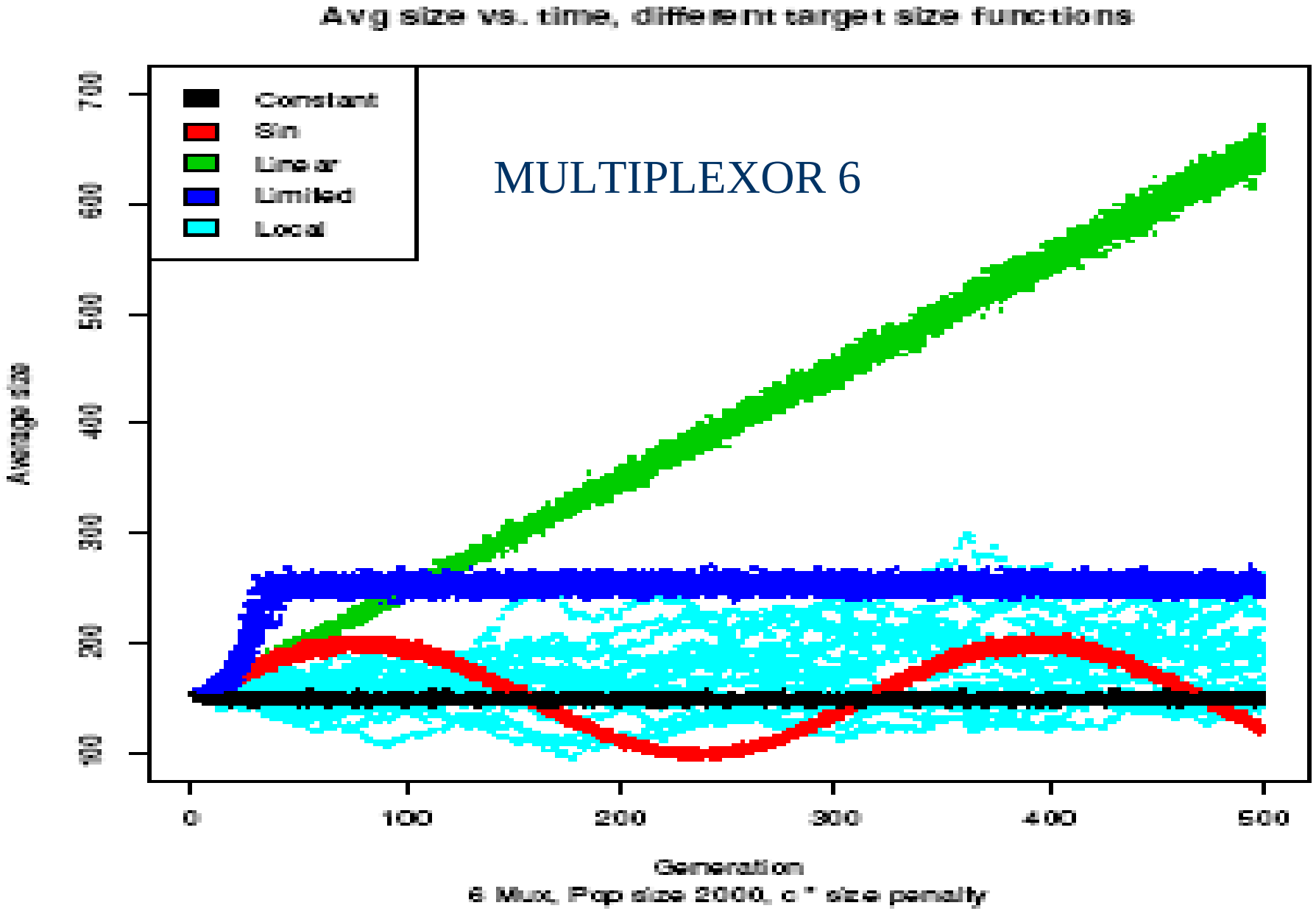
Control de bloat con penalización bien fundamentado (Poli and McPhee, 2008b)

- F'(x) = F(x) + C(t)*nodos-en(x)
- ¿k?
- C(t) = Cov(l,f)/Var(l)
  - l = tamaños de programa
  - f = fitness de individuos
- Hay que recalcular K en cada generación (por eso depende del tiempo)
- Con esto se consigue que el tamaño medio de los individuos en todas las generaciones sea aproximadamente el mismo que el tamaño medio en la generación cero => No hay bloat (aunque hay cierta variabilidad)

Control de bloat con penalización bien fundamentado (Poli and McPhee, 2008b)

- Podemos incluso conseguir que la evolución del tamaño medio de los individuos siga cualquier función predefinida
- $\gamma(t+1)$ es el tamaño medio que deseamos que los individuos tengan en t+1
- $\mu(t)$ es el tamaño medio actual (t)

$$c(t) = \frac{\mathrm{Cov}(\ell, f) - (\gamma(t+1) - \mu(t))\bar{f}}{\mathrm{Var}(\ell) - (\gamma(t+1) - \mu(t))\mu(t)}$$


Avg size vs. time, different target size functions
Constant
Sin
Linear
Limited
Local
MULTIPLEXOR 6
Average size
700
600
500
400
300
200
100
0
100
200
300
400
500
Generation
6 Mux, Pop size 2000, c * size penalty

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

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
```

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>) -> ... ->

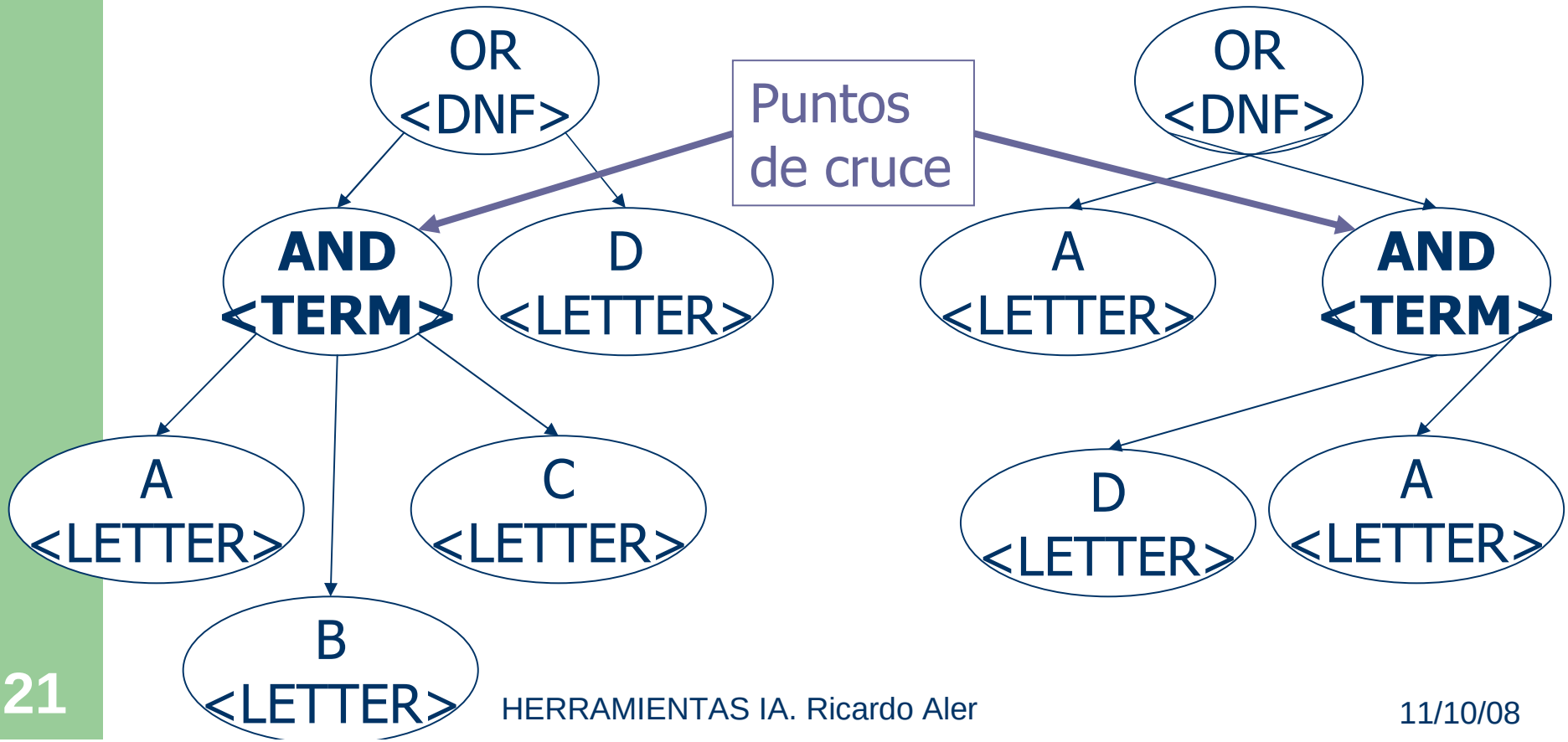
(OR (AND (A B C)) D)

(<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 (es mas complicado hacer *ramped half-and-half*)
- 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



Modelo de corutinas [Maxwell, 94]

- ¿Cómo computar la fitness de individuos que pueden tardar mucho (sobre todo si tienen bucles)?
- Limitar tiempo, iteraciones, etc:
  - Problema: 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

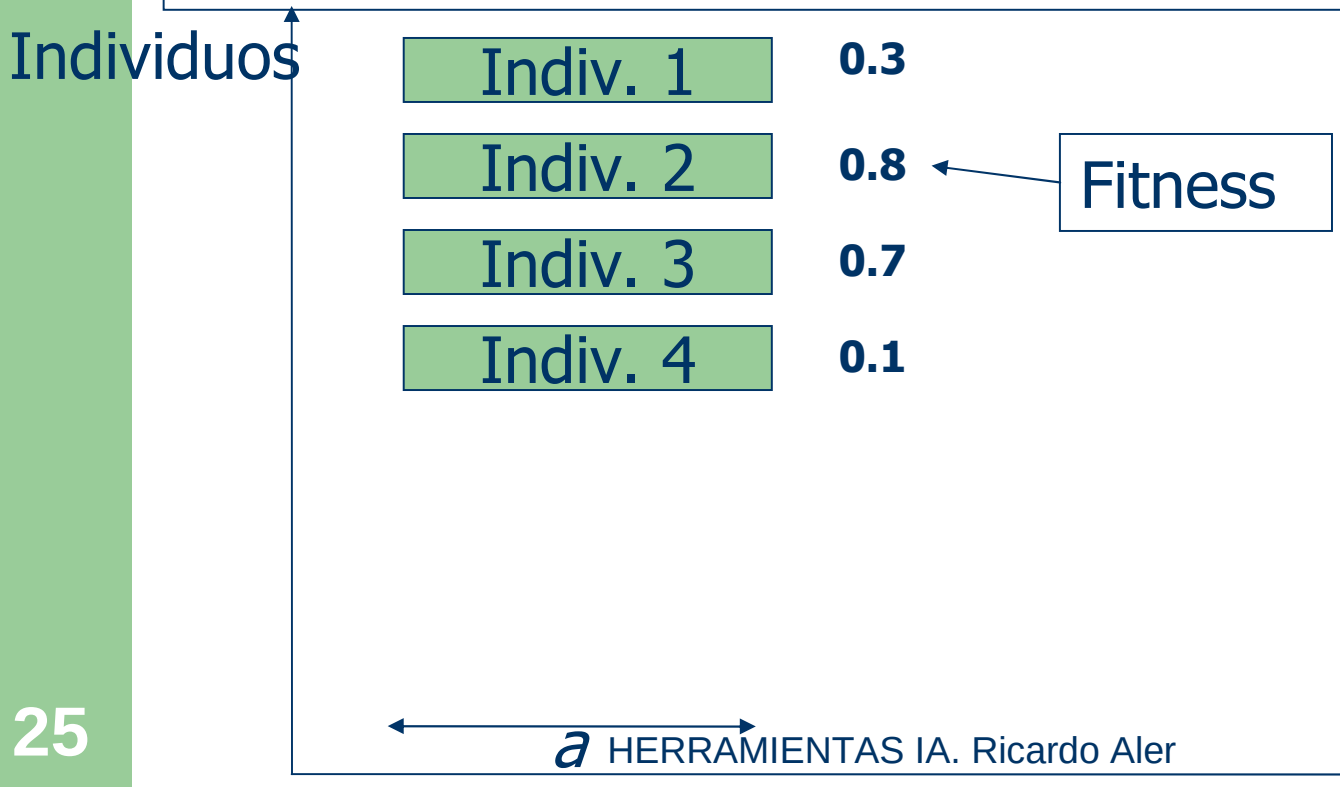
- Si un individuo ha conseguido en el mismo tiempo una fitness mayor que otro, el primero reemplaza al segundo
- Necesita que los individuos devuelvan la fitness conseguida hasta el momento de manera continua (ej: fitness acumulada en los casos de prueba resueltos hasta el momento)
- Sólo se pueden comparar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

Modelo de corutinas

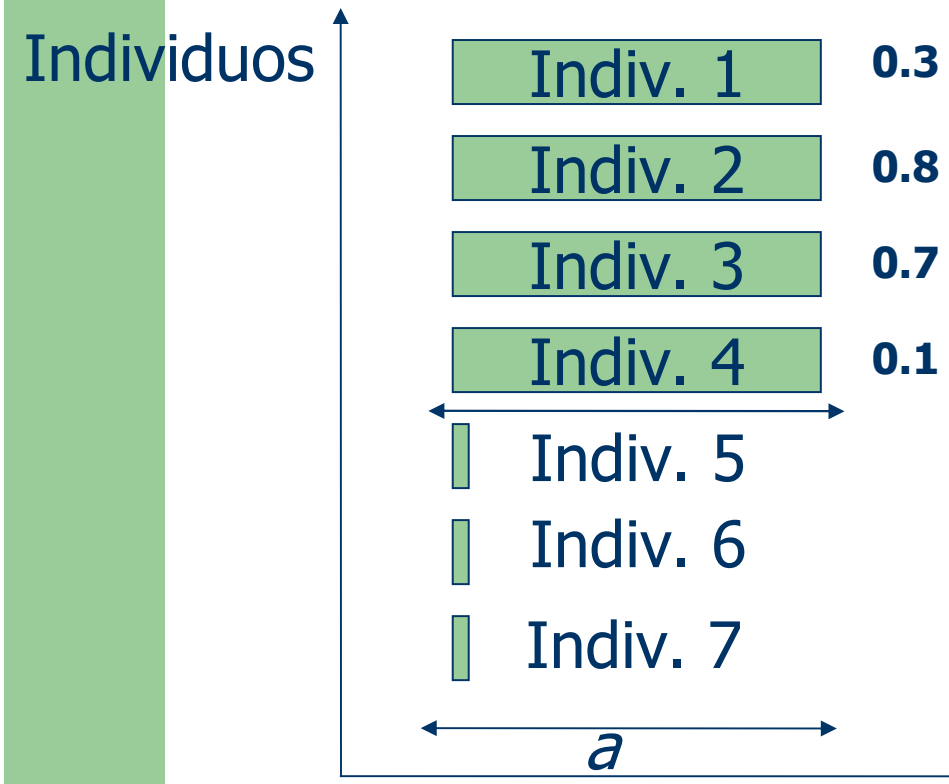
- Creación de la población inicial
- Ejecutar cada individuo un tiempo *a*
- Si se consigue el éxito, terminar
- Crear N nuevos individuos (selección-torneo, mutacíón, cruce)
- Ejecutar los N individuos hasta que tengan la misma edad que los de la población
- Reemplazar antiguos individuos por los N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

Modelo de corutinas

- Crear pob. Inicial y ejecutarla durante *a* segundos. Computar fitness parcial

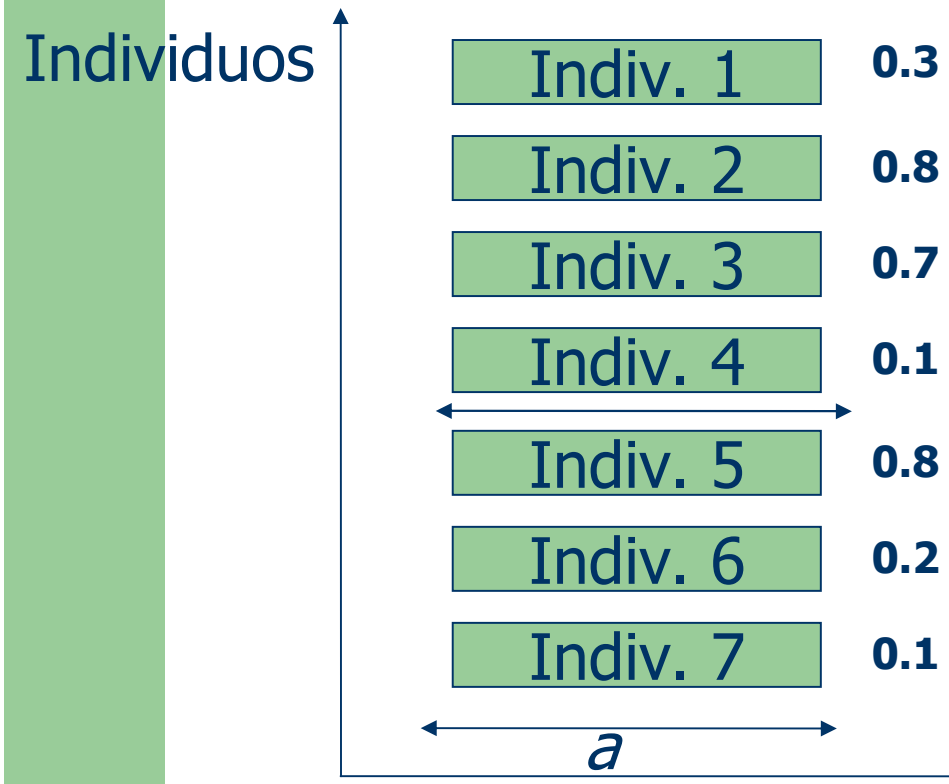


Modelo de corutinas



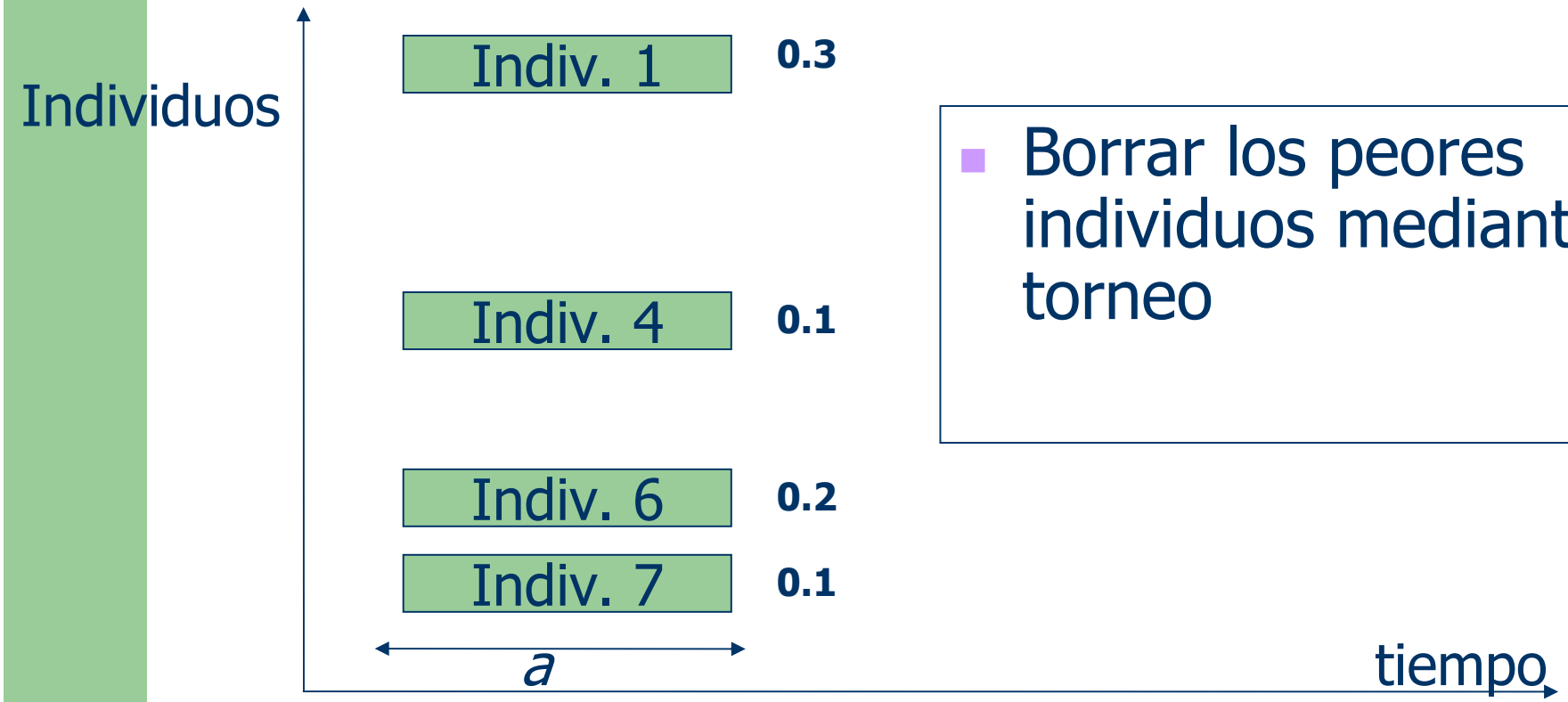
- Crear N nuevos individuos con torneo y operadores genéticos

Modelo de corutinas



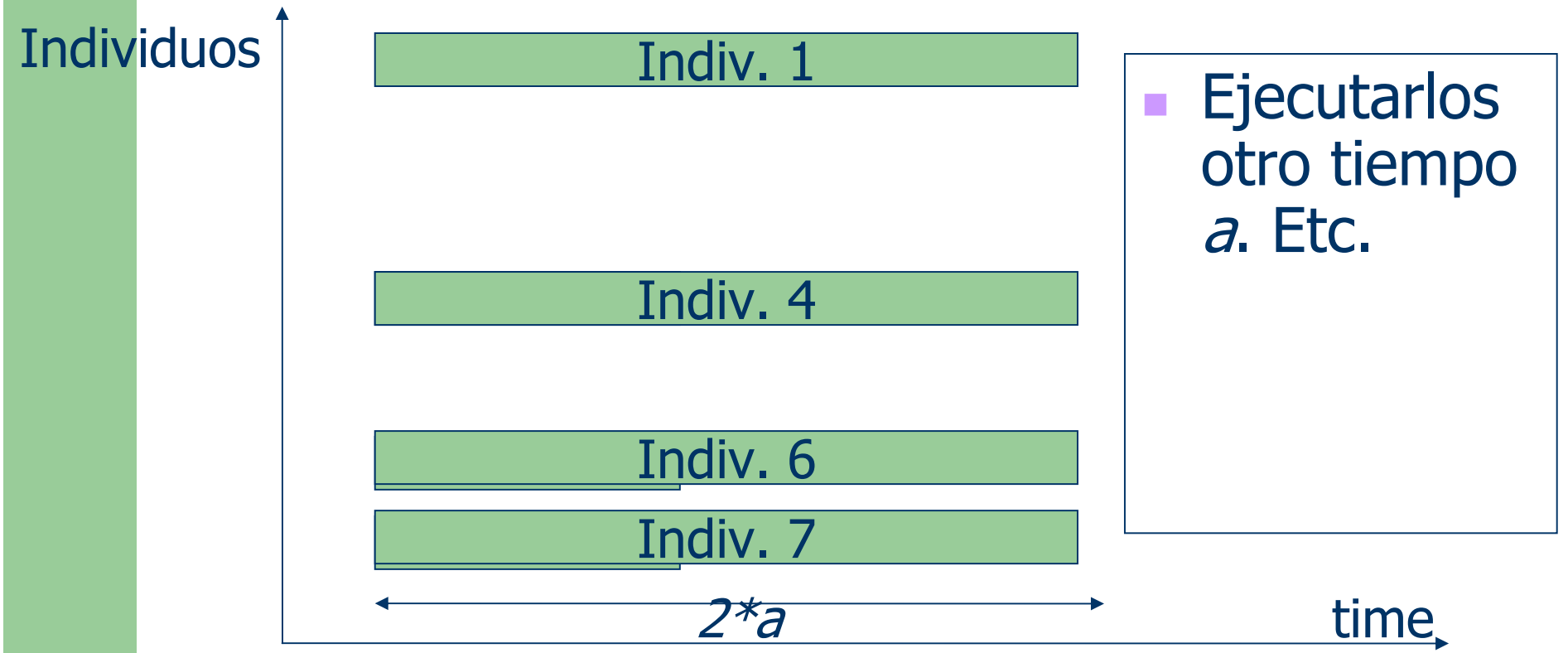
- Ejecutar los N individuos durante el mismo tiempo que el resto de individuos en la población

Modelo de corutinas



- Borrar los peores individuos mediante torneo

Modelo de corutinas



- Ejecutarlos otro tiempo *a*. Etc.

Modelo de corutinas

- El modelo elimina aquellos programas que tardan en conseguir fitness (en comparación con los otros)
- Substituye a los individuos lentos por hijos de individuos rápidos que supuestamente serán mejores
- Comienza a funcionar bien en cuanto aparece al menos un individuo que computa todos los casos de prueba en un tiempo finito. Ese individuo actúa de límite temporal

Modelo de corutinas

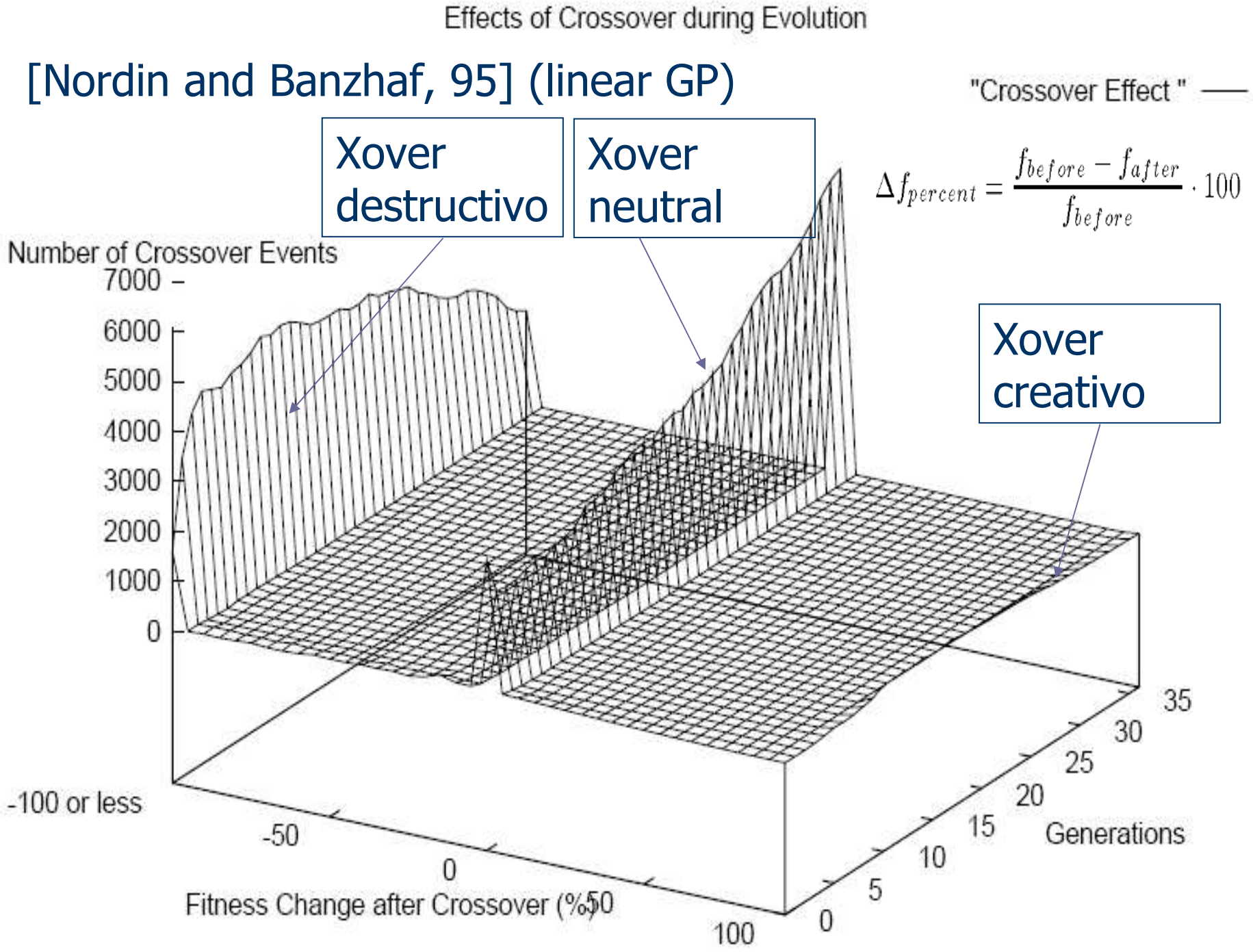
- Requiere un sistema que permita ejecutar un programa durante un tiempo, quitarlo de la CPU, guardar su estado, y volverlo a ejecutar más adelante
- Parece generar individuos más eficientes
- Pero no hay garantía de que un individuo “rápido” vaya a ser el mejor a la larga (el modelo es sólo una heurística)

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])

Modelo de corutinas. Otras posibilidades

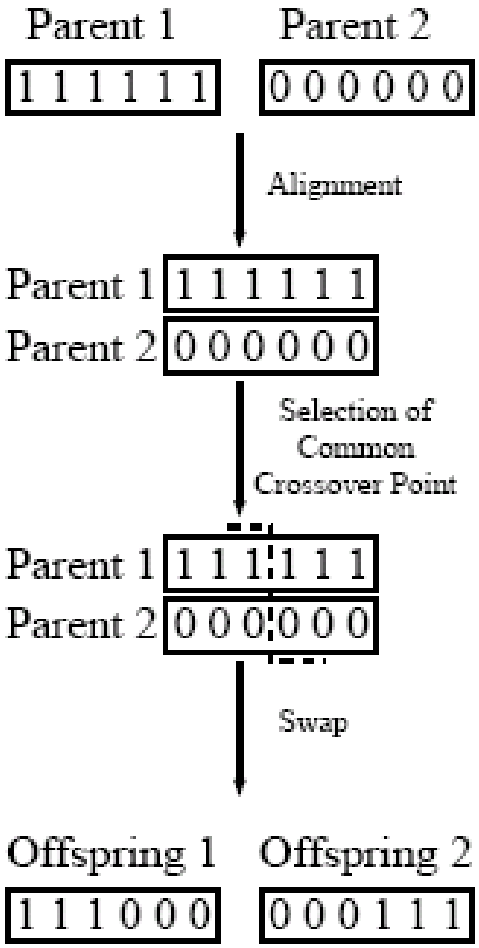
- Las corutinas son difíciles de implementar
- Ejecutar todos los procesos en paralelo y manejar individuos de distintas edades en la misma población
- Guardar la fitness de cada programa en cada generación
- Obtener la mejor fitness de cada tiempo de ejecución
- Cuando un programa alcance ese tiempo de ejecución, cancelarlo si es peor que la mejor fitness obtenida en ese tiempo
- O bien ajustar la prioridad a la baja del hilo


Effects of Crossover during Evolution
[Nordin and Banzhaf, 95] (linear GP)
"Crossover Effect "
$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100$
Xover destructivo
Xover neutral
Xover creativo
Number of Crossover Events
7000
6000
5000
4000
3000
2000
1000
0
-100 or less
-50
0
50
100
Fitness Change after Crossover (%)
0
5
10
15
20
25
30
35
Generations

Headless Chicken Crossover Operator HCCO

- [Luke, Spector, 97,98]: “A [Revised] Comparison of Crossover and Mutation in GP”:
  - El cruce parece funcionar mejor que la mutación, pero la diferencia es muy pequeña
- [Chelapilla, 97]: “Evolving computer programs without subtree crossover”:
  - Se puede evolucionar programas correctos utilizando sólo mutación
- El cruce no recombina subárboles sino que actúa como una macro-mutación. Pero funciona razonablemente bien (**mejor** que una búsqueda aleatoria)

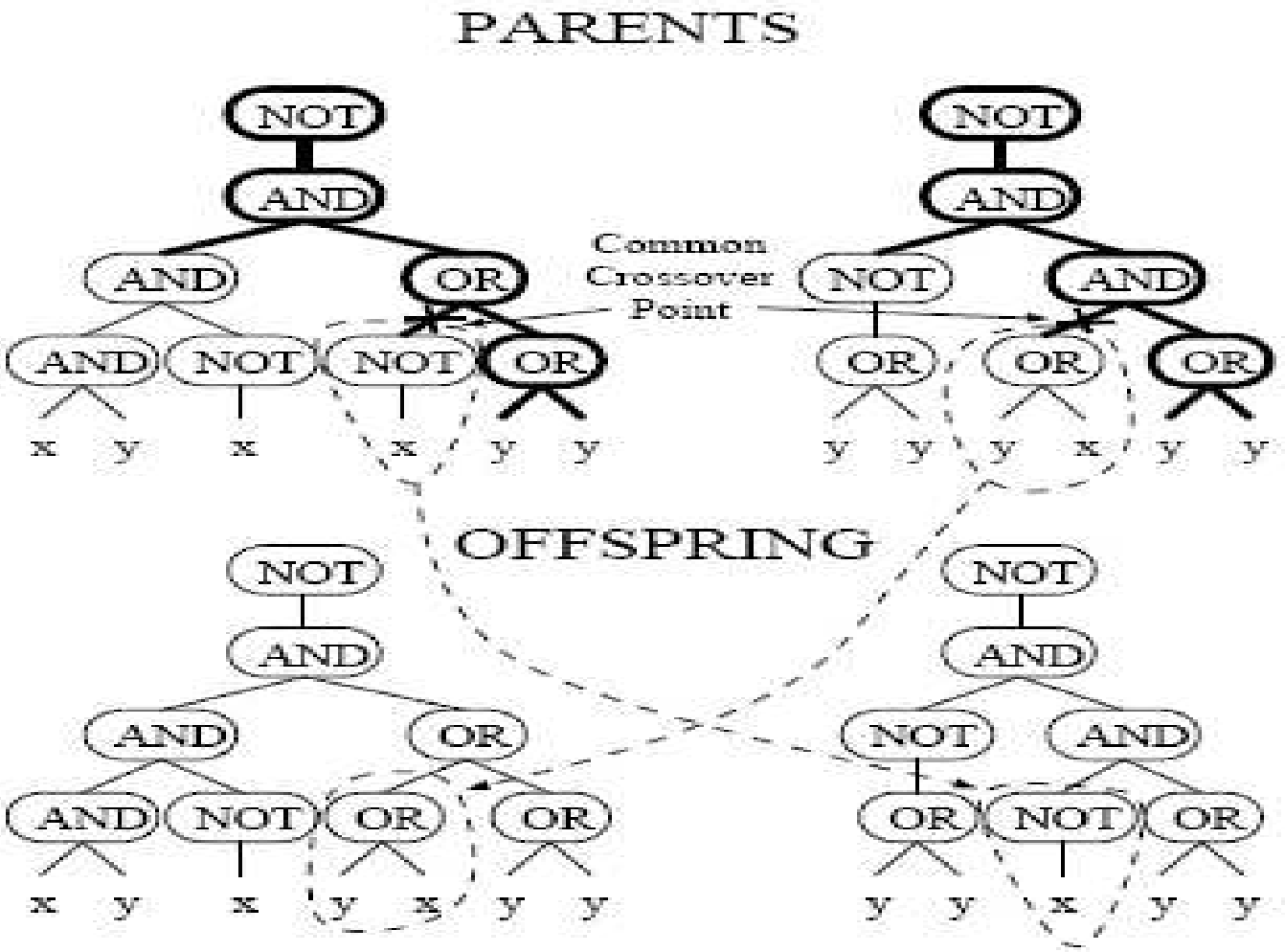
Cruce en algoritmos Genéticos

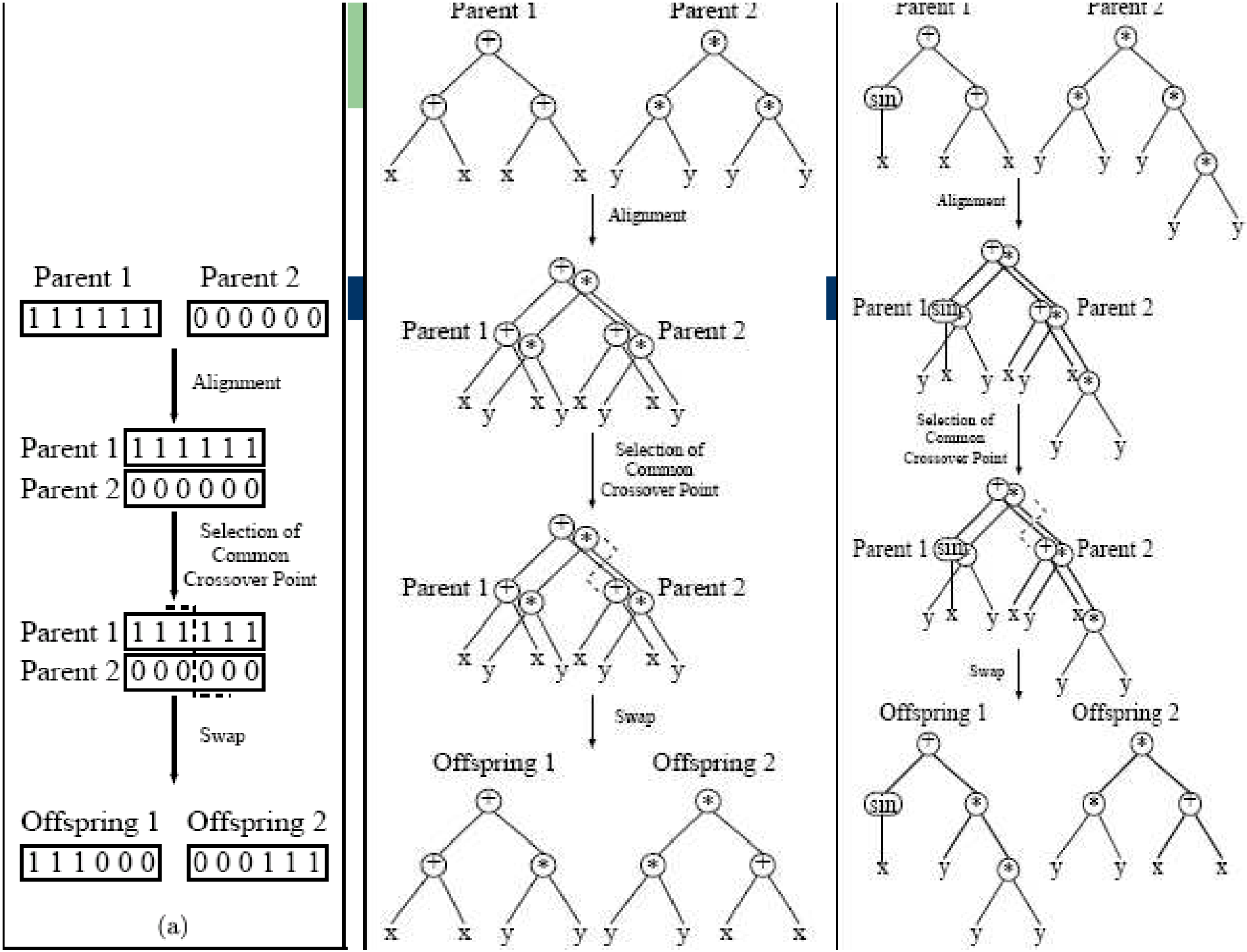


A diferencia de en Programación Genética, el significado de cada bit viene dado por su posición en la cadena de bits (cromosoma)

El cruce siempre mantiene la posición de todos los bits en el cromosoma

One Point Crossover




Parent 1
Parent 2
1 1 1 1 1 1
0 0 0 0 0 0
Alignment
Parent 1 1 1 1 1 1
Parent 2 0 0 0 0 0 0
Selection of Common Crossover Point
Parent 1 1 1 1 1 1
Parent 2 0 0 0 0 0 0
Swap
Offspring 1
Offspring 2
1 1 1 0 0 0
0 0 0 1 1 1
(a)
Parent 1
Parent 2
Alignment
Parent 1
Parent 2
Selection of Common Crossover Point
Parent 1
Parent 2
Swap
Offspring 1
Offspring 2
Parent 1
Parent 2
sin
Alignment
Parent 1
Parent 2
Selection of Common Crossover Point
Parent 1
Parent 2
Swap
Offspring 1
Offspring 2

One-Point Crossover. Funcionamiento

1. Alineamiento: comenzando desde la raíz, busca qué puntos de cruce tienen la misma estructura
2. Elige uno de los puntos de cruce aleatoriamente
3. Intercambia los subárboles

Propiedades del One-point Crossover

- Al comienzo de la evolución, los puntos de cruce pertenecen a la parte superior de los árboles (es imposible que, si los padres son distintos en los nodos superiores, se llegue a elegir un punto de cruce inferior
- En el cruce estándar, los puntos de cruce están cerca de las hojas (porque hay más)

Propiedades del One-point Crossover

- En sucesivas generaciones se fija la estructura superior de los individuos y se comienza a explorar la parte inferior (búsqueda top-down)
- Tiene sentido comenzar por la raíz, porque un cambio ahí, produce cambios más grandes
- Si no se utiliza mutación puntual, la población converge demasiado rápidamente

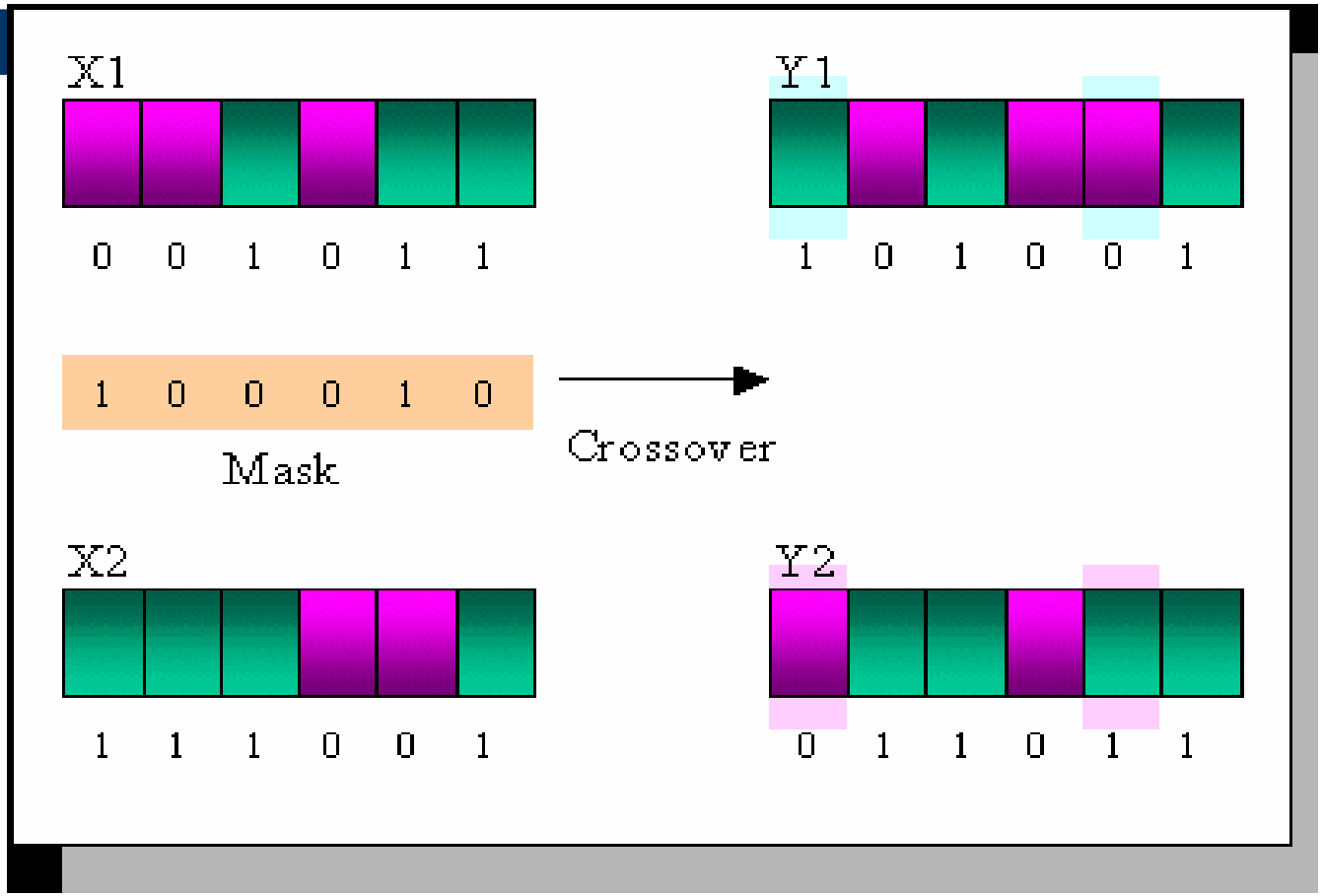
Propiedades del One-point Crossover

- Este cruce no incrementa la profundidad de los hijos con respecto a la de los padres
- Tampoco decrementa su profundidad
- Nunca se crean individuos más profundos o más pequeños que los de la población inicial
- Hay que asegurar que en la población inicial hay una buena mezcla de profundidades (variedad)
- Los hijos heredan la parte superior de los padres

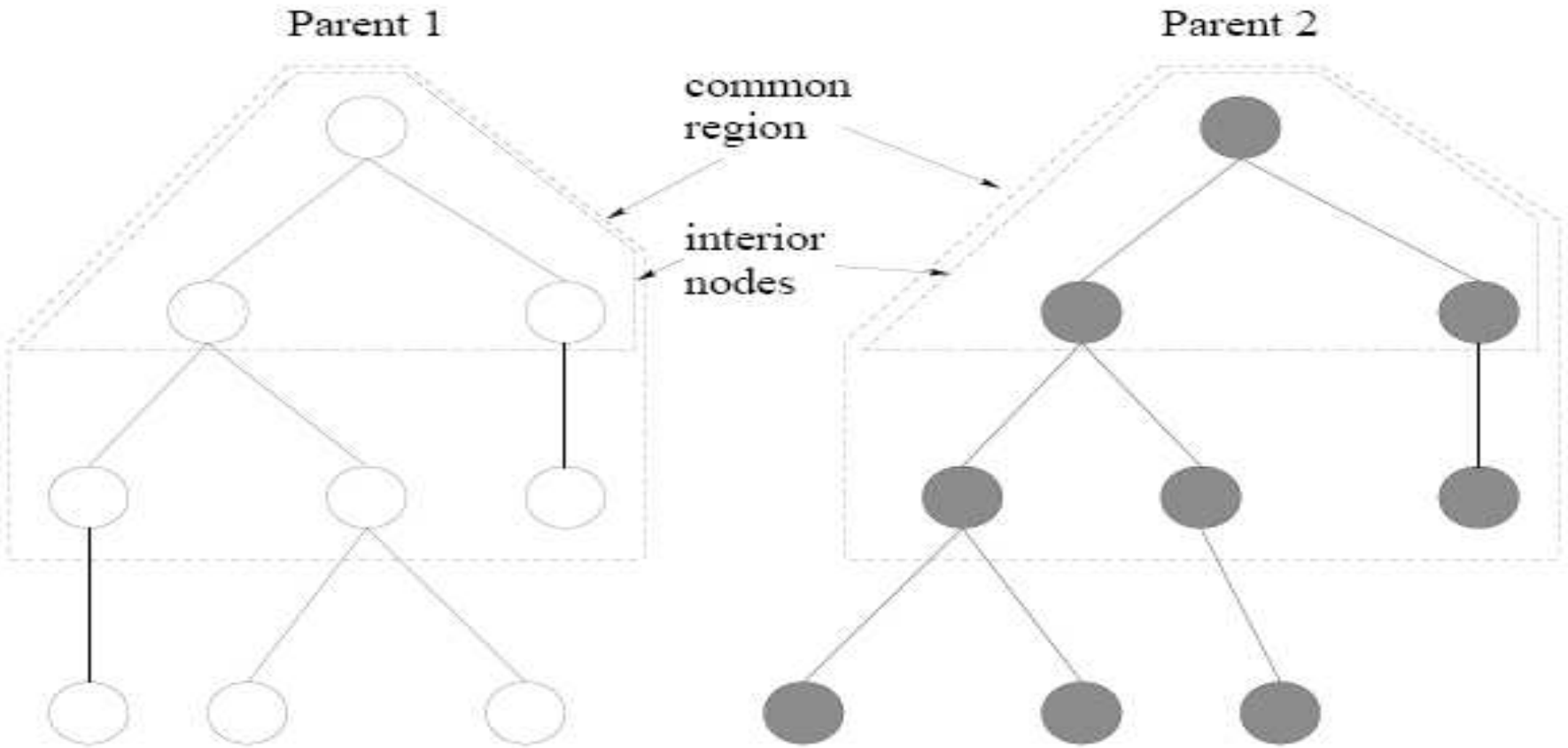
Propiedades del One-point Crossover

- El crossover normal tiende a elegir nodos cerca de las hojas (porque hay más, que cerca de la raíz)
- Es por tanto un operador de búsqueda local, porque los hijos se parecen mucho a los padres
- En contraste, el one-point crossover es global al principio y local hacia el final

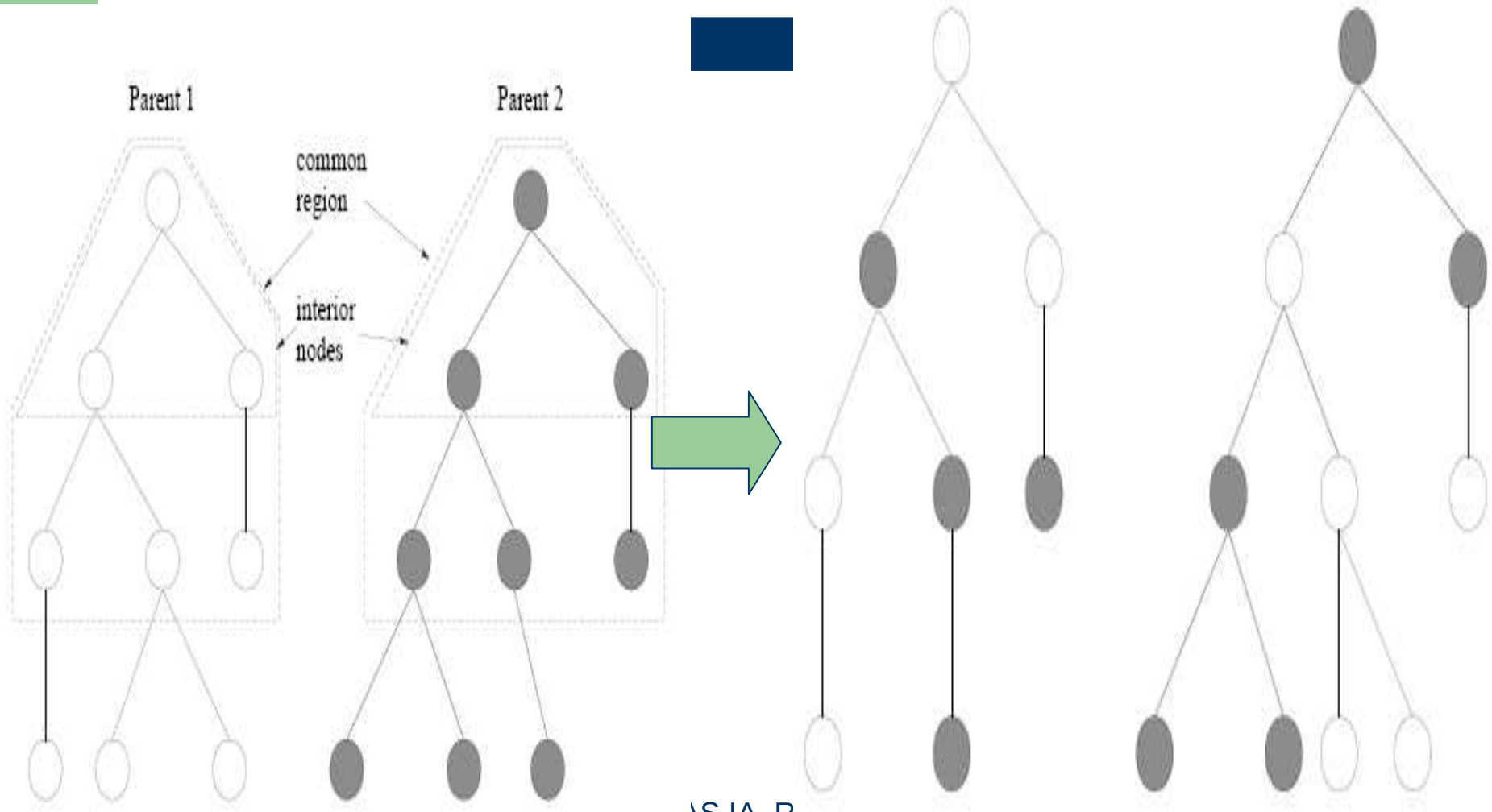
Cruce Uniforme en AG



El cruce uniforme (GPUX)



El cruce uniforme (GPUX)



El cruce uniforme. Funcionamiento

- Determinar la región común y la interior
- Seleccionar qué nodos se van a intercambiar dentro de la zona común
- Si el nodo a intercambiar está dentro del interior, hacerlo
- Si no está en el interior, intercambiar junto con sus nodos descendientes

Propiedades del cruce uniforme

- Hace una búsqueda más global (los hijos se parecen menos a los padres que con el crossover normal o el one-point crossover)