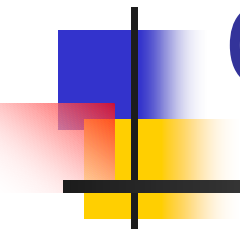
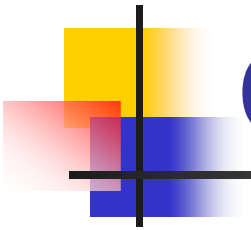


Genetic Programming



Genetic Programming

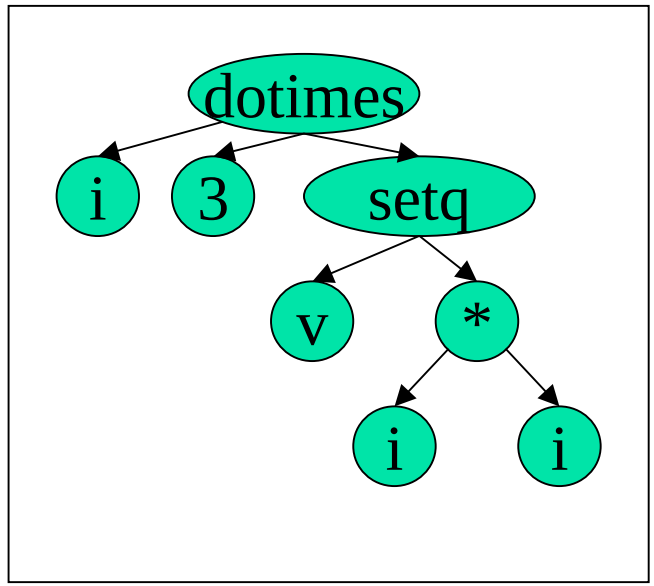
- Genetic algorithms for evolving programs
- Instead of bitstrings, programs are evolved
- M. Cramer. 1985. **A Representation for the Adaptive Generation of Simple Sequential Programs,** *Proc. of an Intl. Conf. on Genetic Algorithms and their Applications.*

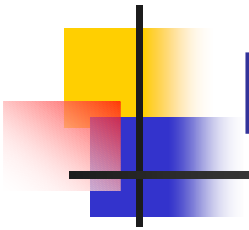
Genetic Programming

- 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.**
- 1999. **Genetic Programming III: Darwinian Invention and Problem *Solving***
- *2003.* ***Genetic Programming IV: Routine Human-Competitive Machine Intelligence***

Program Representation

- LISP / parse trees
- (dotimes i 3 (setq v (* i i)))
- For i := 1 to 3 {v := i*i;}
- Language = functions + terminals



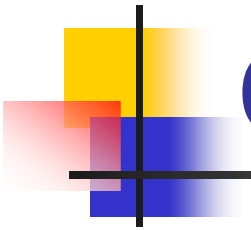


Example: Even-parity

- Even-parity(1,0,0,0,1,0,0,1,1) -> TRUE
- **Language / primitives**:
  - Functions: AND, OR, NAND, NOR, NOT (no XOR available!)
  - Terminals: D0, D1, D2, …, D9
- **Heuristic / fitness**: count number of input/output pairs (fitness cases) solved correctly

Fitness cases 10 bit even-parity

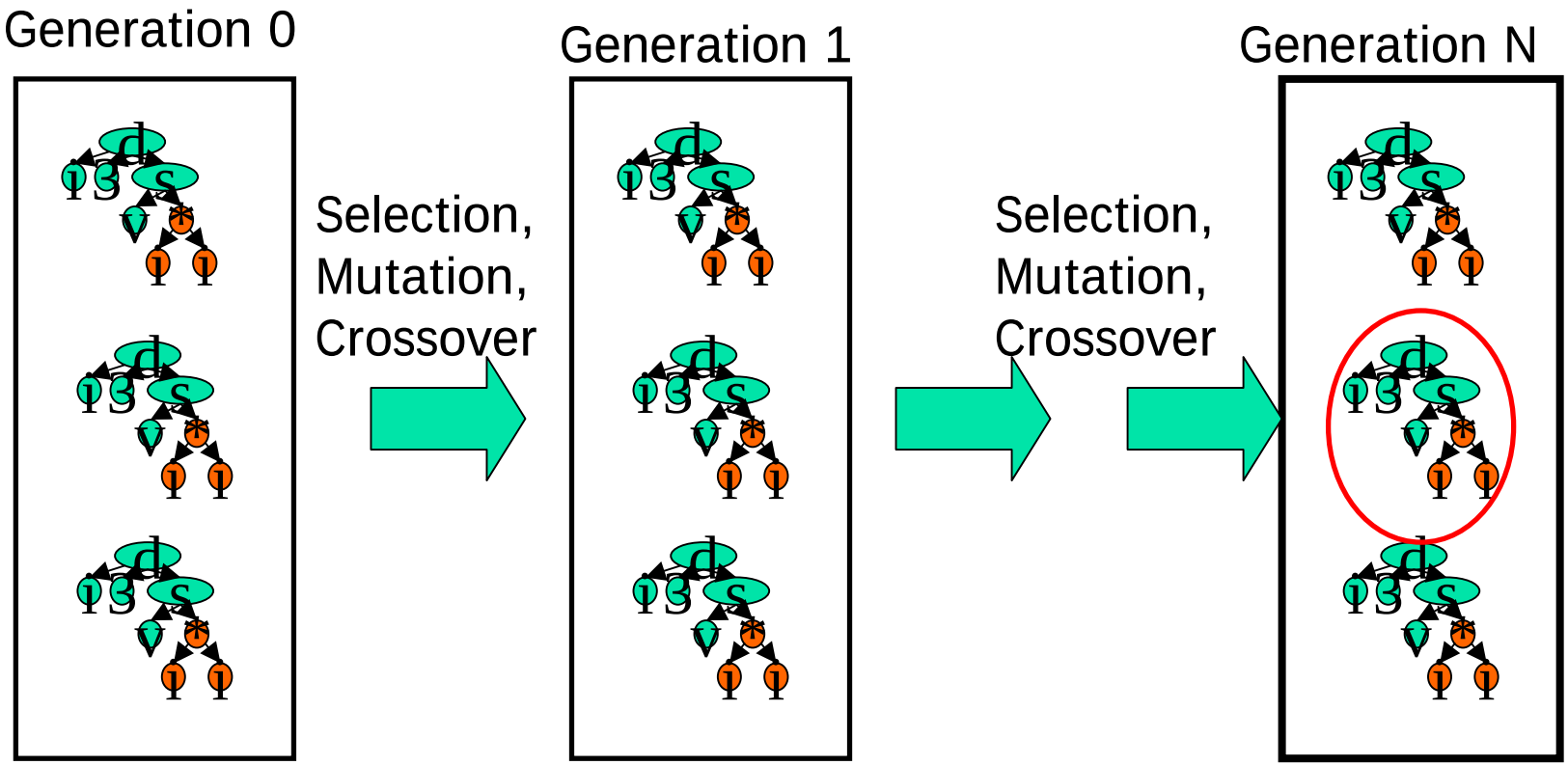
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	OUTPUT
0	0	0	0	0	0	0	0	0	0	TRUE
0	0	0	0	0	0	0	0	0	1	FALSE
0	0	0	0	0	0	0	0	1	1	TRUE
…										



Generational GP Algorithm

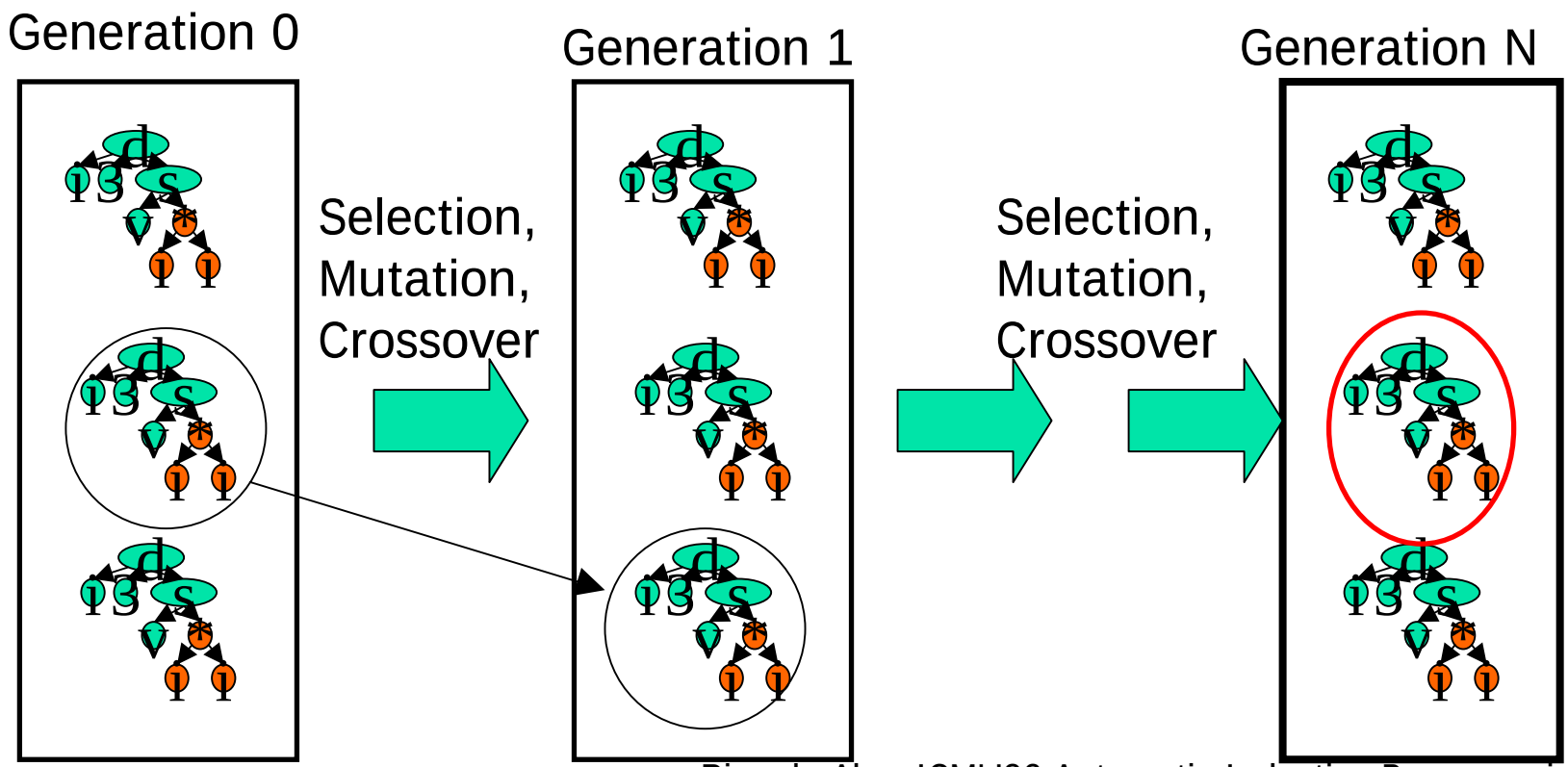
1. Create a **random** population of programs (individuals) using the functions and terminals
2. Run all the programs and compute their fitness
3. Select (stochastically) the **best** ones according to some policy
4. Create a new population by applying the genetic operators to the selected individuals
5. Go to 2, until a “good enough” program is found

Generational GP Algorithm



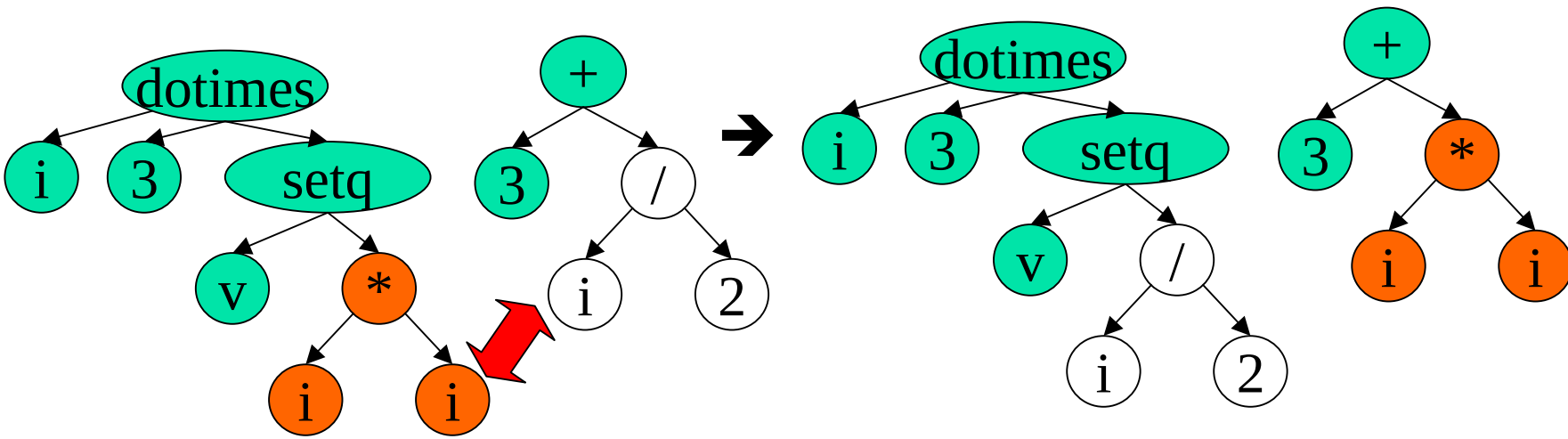
Steady-State GP

Only a few individuals (even 1) change between generations



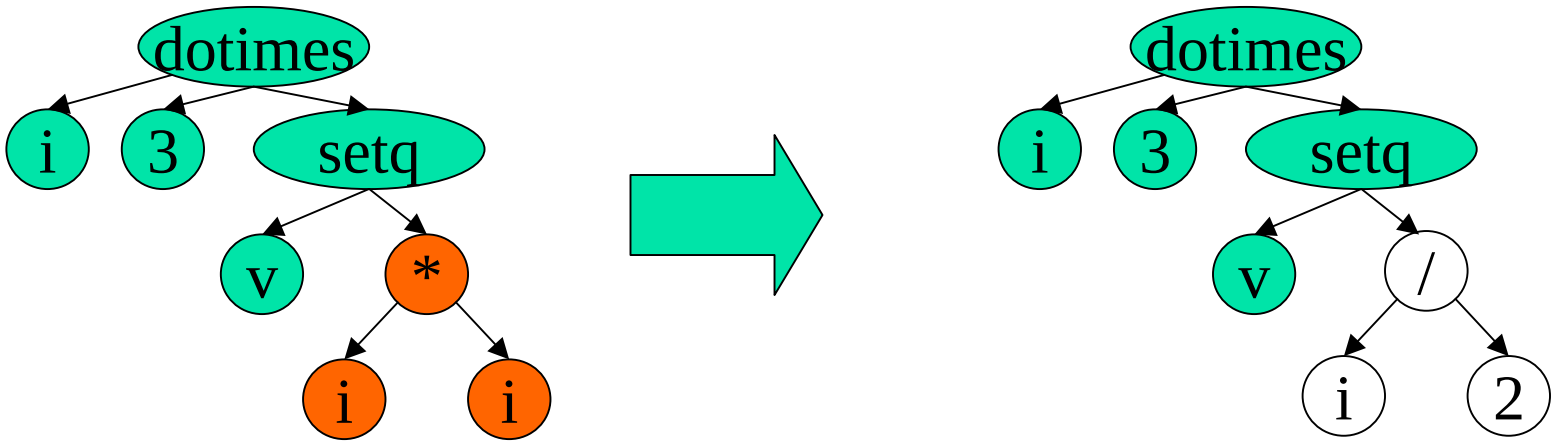
Genetic Operators. Crossover

- Reproduction (just copy the program)
- Crossover (recombination)



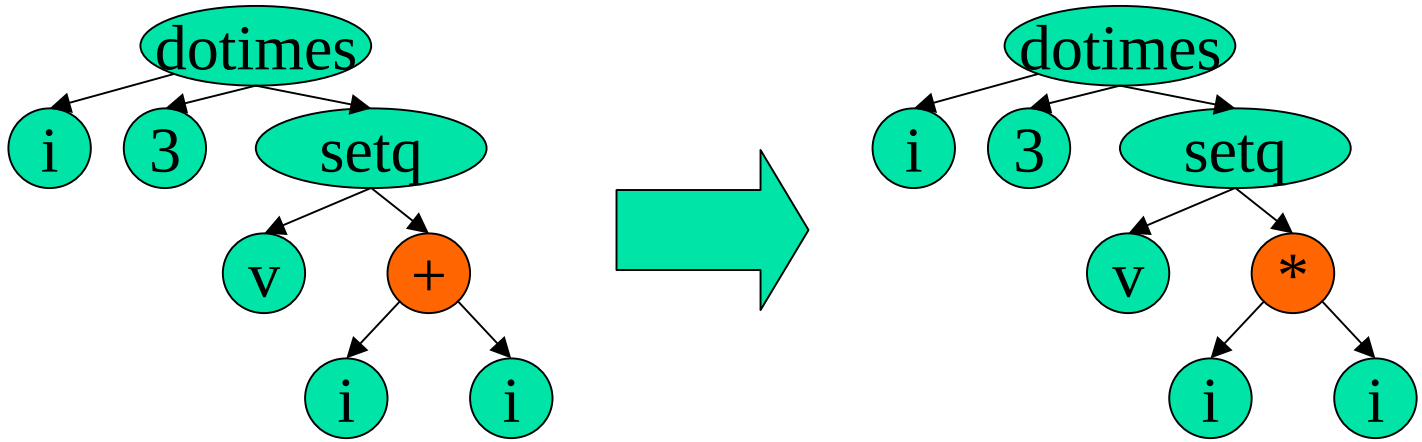
Genetic Operators. Subtree Mutation

Chop off a subtree and grow a random one

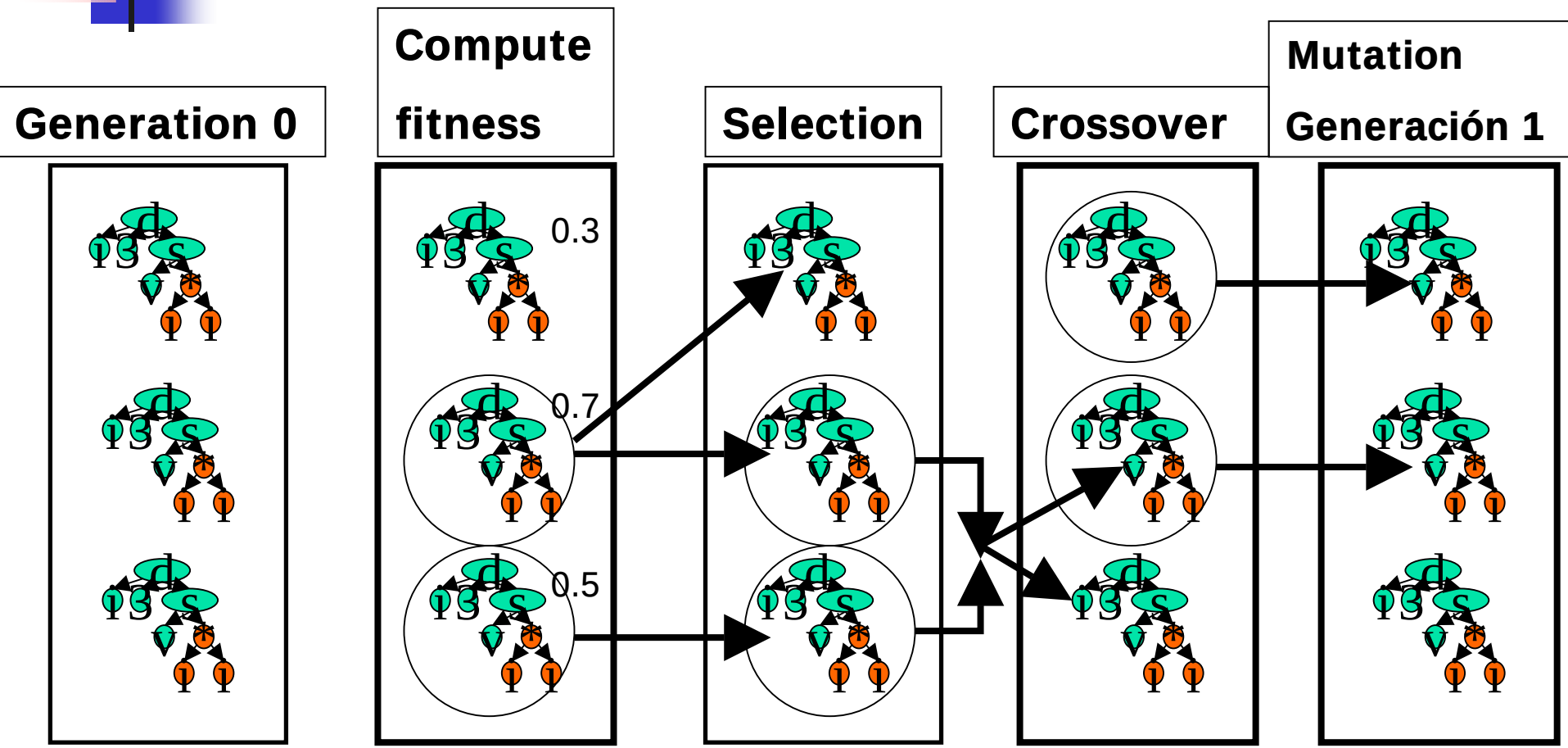


Genetic Operators. Point Mutation

Select a function with the same arity at the mutation point

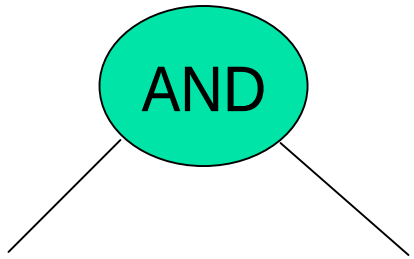


From Generation *i* to *i+1*



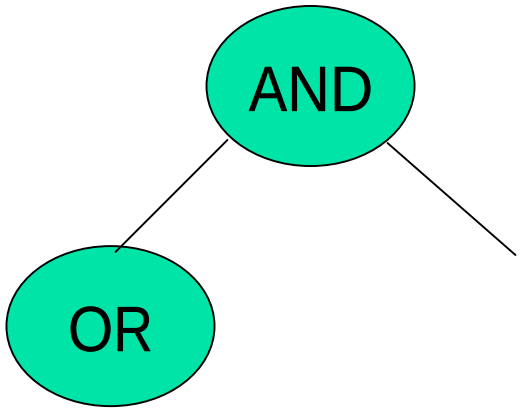
Random Creation of the Initial Population

- Select randomly a function for the tree root from:
  - {AND(1,2), OR(1,2), NAND(1,2), NOR(1,2), NOT(1)}
  - {D0, D1, D2, …, D9}
- Create as many branches as the function's arity



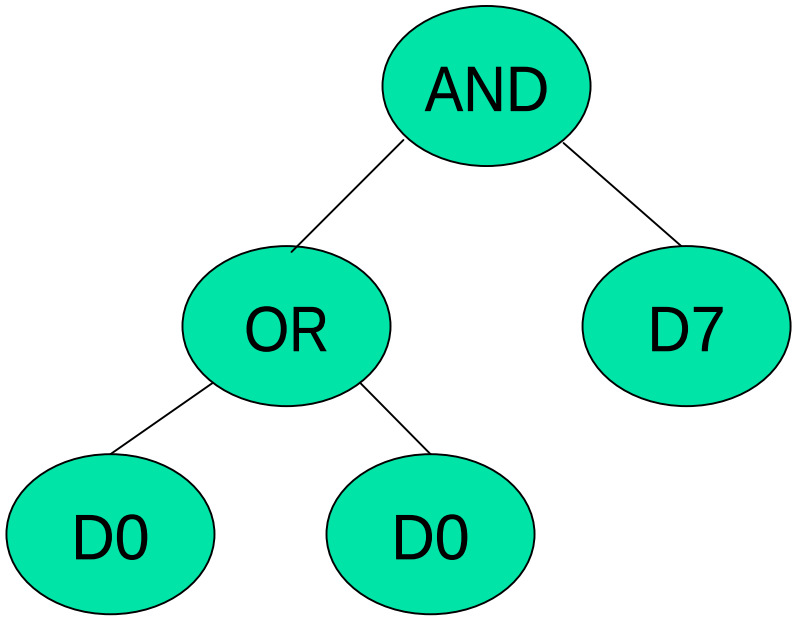
Random Creation of the Initial Population

- Create as many subtrees as branches

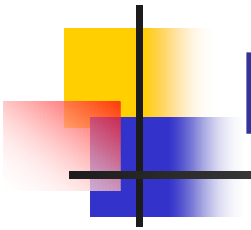


Random Creation of the Initial Population

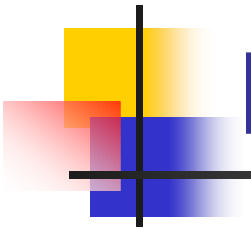
- It is important to create an initial population as diverse as possible:
  - Different tree shapes
  - Different tree depths



Methods for Generating Diverse Individuals

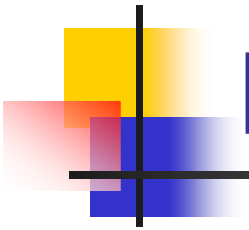


- "Full": all tree paths have the same depth
- "Grow": variable depth
- "Ramped half and half":
  - Individuals are generated for depths 1, 2, 3, …, max-depth
  - 50% full, 50% grow
- Goal: maximize diversity



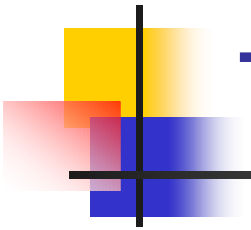
Language

- Language (primitives) = functions + terminals



Functions

- Have 1 or several arguments: +, not, ...
- Kinds:
  - Functions: arguments are evaluated before calling the function:
    - +(3, *(4,5)) = +(3,20) = 23
  - Macros: the macro controls which arguments are evaluated:
    - If(3>5, v:=3, v:=4) -> v:=4
- **Closure**: every function must be able to accept any value (i.e. They must be protected):
  - 3 / 0 = 1
  - 3 + “joe” = 3

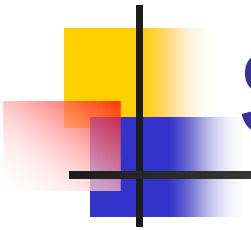


Terminals

- Input variables:
  - D0, D1, ..., D9
- Functions with no arguments:
  - *go-forward*
- Constants:
  - *3*, *a*, ...
- “Ephemeral random constant” *R*:
  - For numerical problems. Everytime R is selected during individual creation, a random real number is created

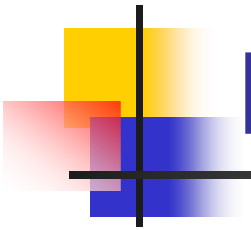
The *fitness* Function

- Raw:
  - Ex: number of fitness cases predicted correctly
- Standard: GP always minimizes:
  - Standard = maximum – raw
- Adjusted: (normalized between 0 and 1)
  - 1/(1+standard).
- Relative:
  - Adjusted/Total fitness in the population



Selection of Best Individuals

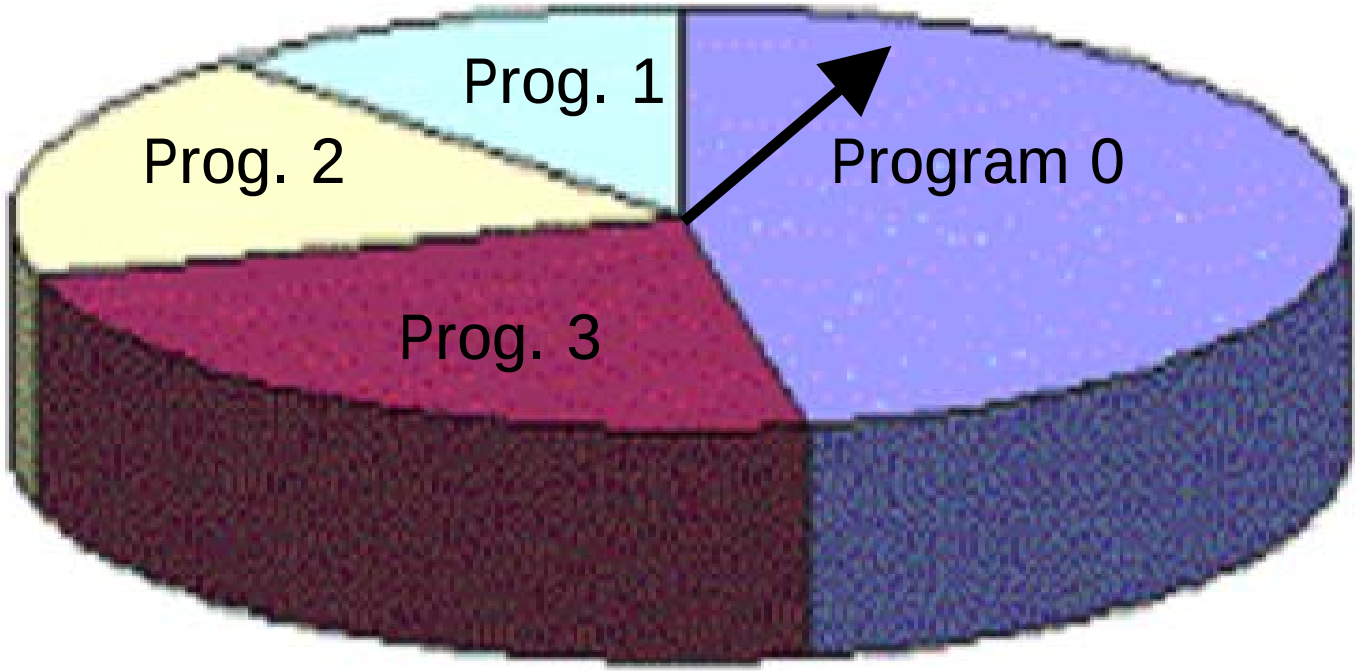
- Fitness proportionated:
  - An individual is selected with a probability proportional to its relative fitness
- Tournament selection
  - K individuals are sampled randomly
  - The best one is selected
- Elitism:
  - The best individual(s) are always selected
  - This is used to make sure that the best individual will not be lost



Premature Convergence

- Some selection methods lead to premature convergence
- That is, we get a population that stops generating better and better individuals

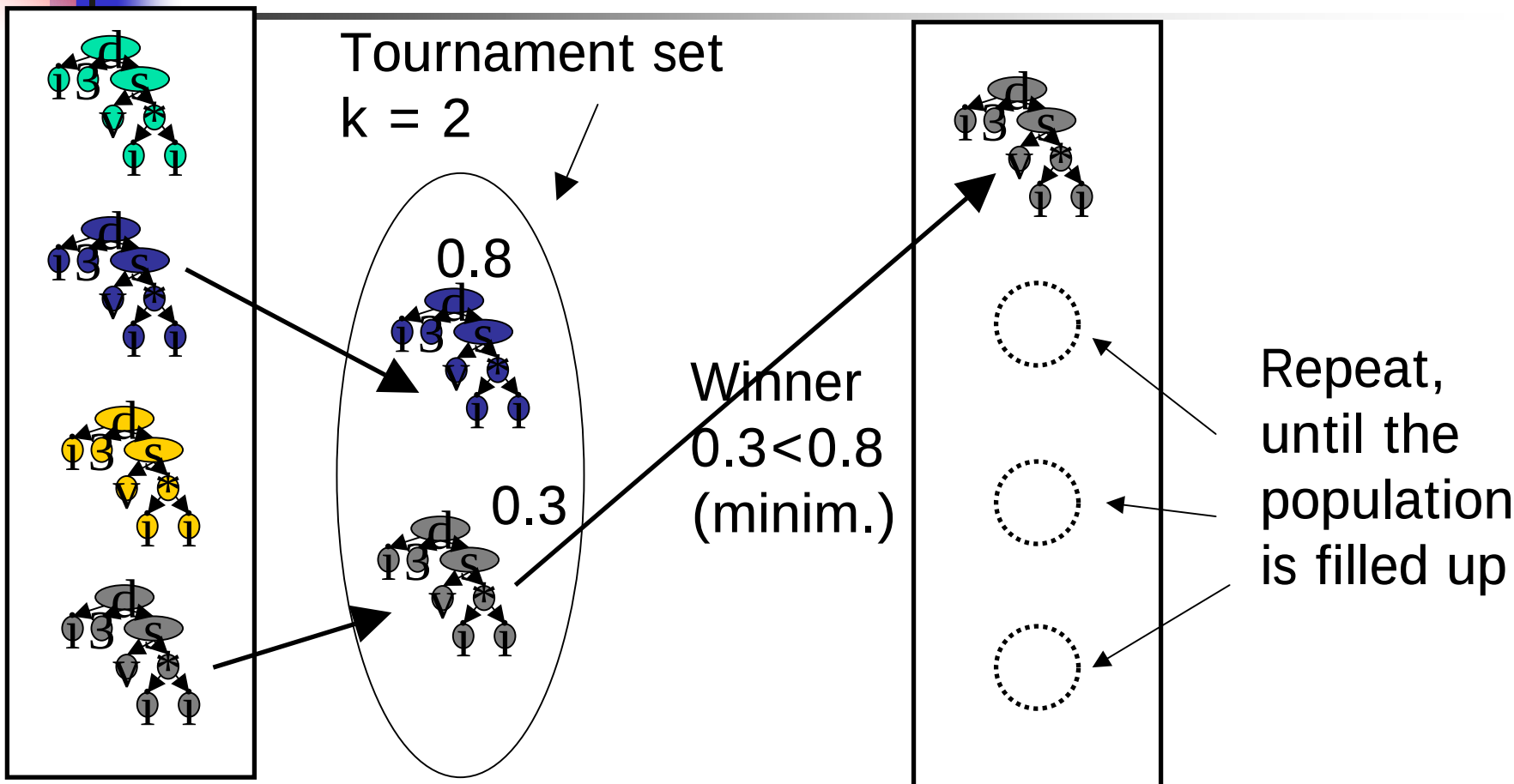
Fitness Proportionated (Roulette Wheel) Selection



On average good individuals will be selected more often

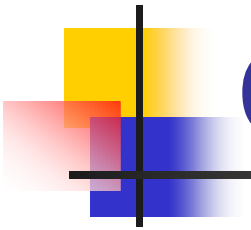
Problem: superindividuals, no variety in the population premature convergence

Tournament Selection



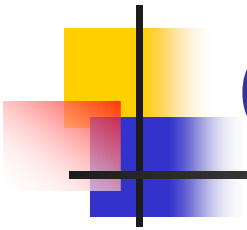
K controls selective pressure

The larger K, the more pressure (4 is ok)



Control Parameters

- Population size (M: 500 to 10000)
- Maximum number of generations (G: 50 to 100)
- Probabilities of crossover, recombination, and mutation (mutation<5%)
- Generation method for the initial population (*grow*, *full*, *ramped half and half*)
- Maximum depth of initial individuals
- Maximum depth of individuals after recombination



GP Tools

- LIL-GP: c
  - http://garage.cse.msu.edu/software/lil-gp/
- ECJ: Java
  - http://cs.gmu.edu/~eclab/projects/ecj/

Example with lil-gp

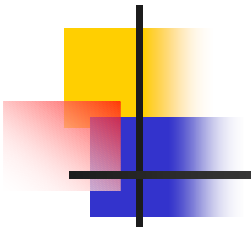
- Individuals in lil-gp are represented as trees. When they are to be executed, lil-gp interprets the tree and calls the primitives as required
- For instance, if the function set includes / and *, the following functions have to be defined:

```
DATATYPE f_protdivide (int tree, farg *args)
    If (args[1].d == 0.0) return 1.0;
    Else return args[0].d / args[1].d

DATATYPE f_multiply (int tree, farg *args)
    Return args[0].d * args[1].d
```

Lil-gp Fitness Function for a Regression Problem dv=f(x)

```
void app_eval_fitness ( individual *ind ){
for ( i = 0; i < fitness_cases; ++i ){
   g.x = app_fitness_cases[0][i];              # x = input
   dv = app_fitness_cases[1][i];               # dv = output
   v = evaluate_tree ( ind->tr[0].data, 0 ); # v=value returned by individual
   disp = fabs ( dv-v ); # difference between correct and predicted value
   ind->r_fitness += disp;                     # Add to the fitness
}
    ind->s_fitness = ind->r_fitness;           # Standard fitness
    ind->a_fitness = 1/(1+ind->s_fitness); # Adjusted fitness
}
```



GP is Stochastic

- GP is stochastic: different runs may provide different results
- No guarantee a GP will end with success (premature convergence)
- Repeat many times, record the best result
- Computational effort (informal definition): minimum number of individuals to be evaluated so that a perfect individual will be obtained with high probability

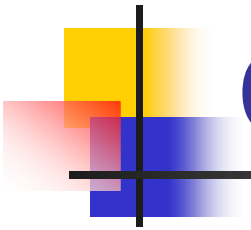
Computational Effort for Even-Parity

Even parity	Computational effort	Time (hours)
3	96.000	
4	384.000	
5	6.528.000	
6	70.176.000	3h (P-1.5GHz)

Note:

Not all fitness evaluation take the same time.

Most of the time is spent on fitness computation



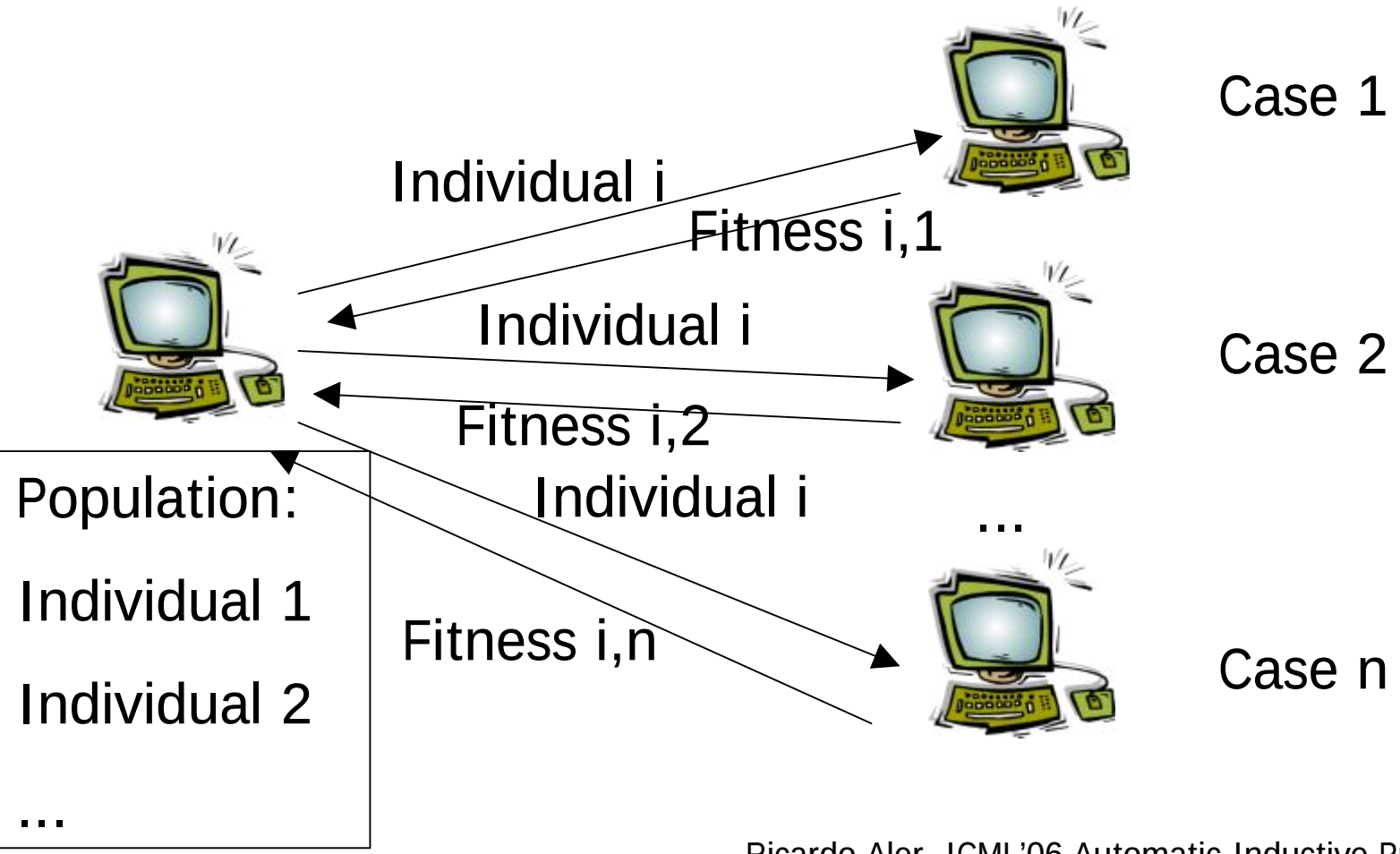
GP Speedup

- Large computational effort required
- Interesting results can be obtained with current machines
- Moore's law: computational power duplicated every 18 months

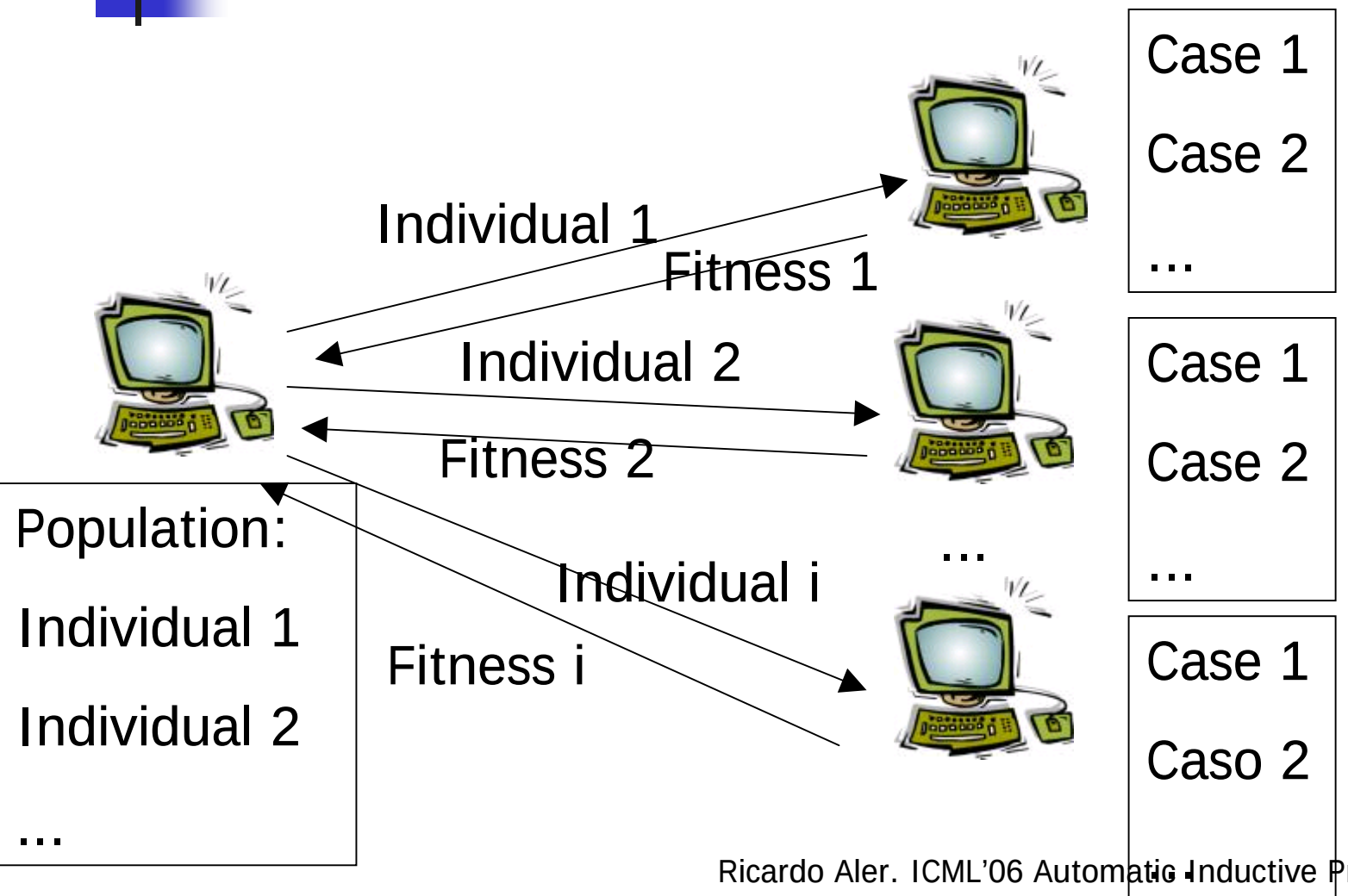
GP Speedup

- Machine Code evolution ([keller, 96], [friedrich, 97], [Nordin, 95] x2000 wrt LISP, x100 wrt C)
- Reconfigurable hardware (PGA)
- Paralelism:
  - 1 run per machine
  - Island model (or demes): 1 population per machine + migration
  - 1 fitness case per machine
  - 1 individual per machine

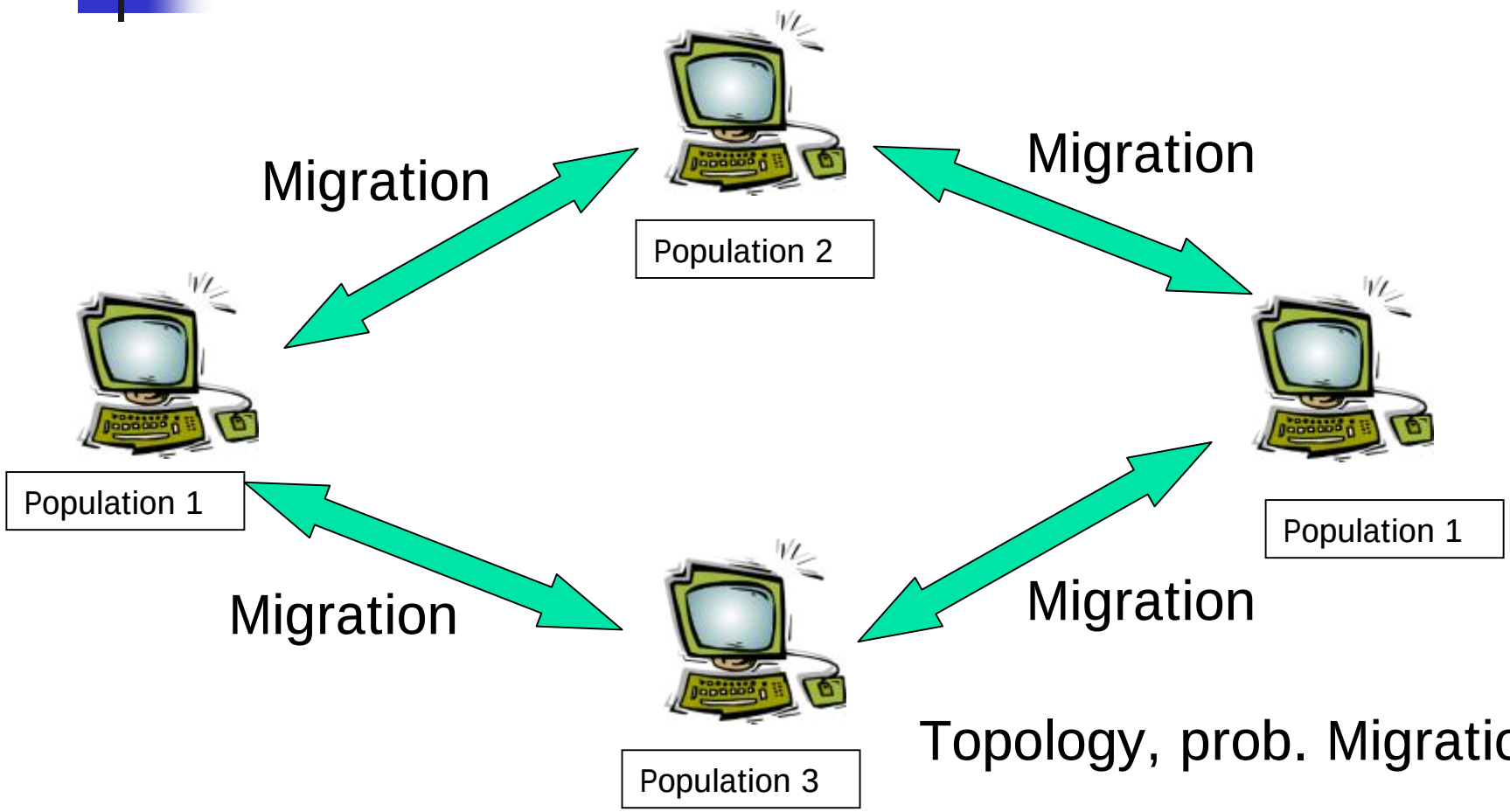
Fitness Case Parallelism



Individual Parallelism



Paralellism with Islands



Topology, prob. Migration

Maintains diversity

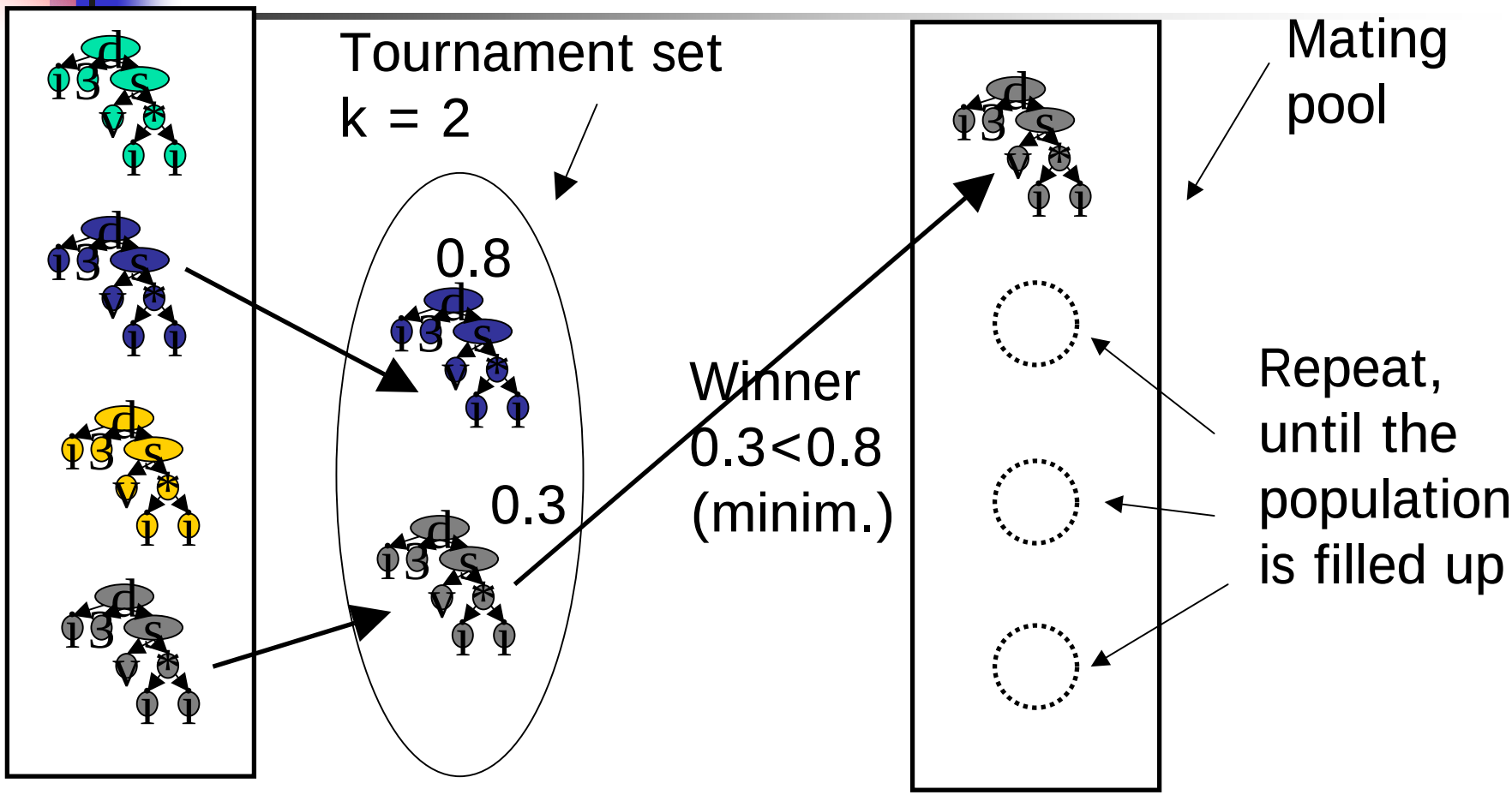
Rational Allocation of Trials (RAT)

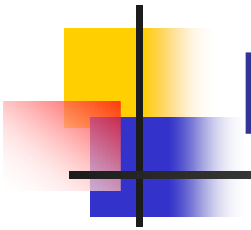
- Teller, Andre. 1997. **"Automatically Choosing the Number of Fitness Cases: The rational Allocation of Trials".** *GECCO'97*
- Do not use all the fitness cases
- Use only as many as necessary to differentiate between good and bad individuals
- Every individual will evaluate the most appropriate number 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

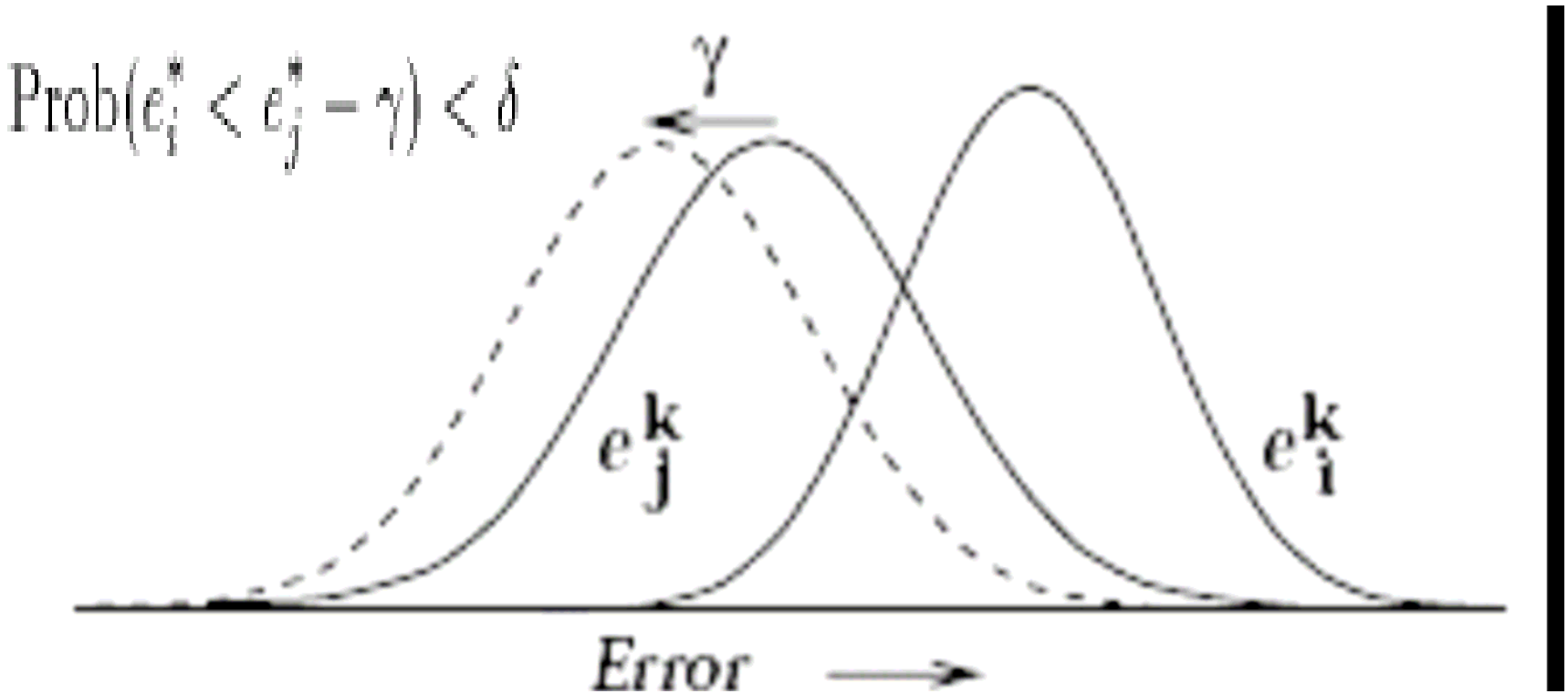
    OR
    - X is in the first place (temporary winner)
    - AND it is not likely that other individuals in tournament t become better than X
  - Evaluate all individuals still in list Q on the current training example

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*

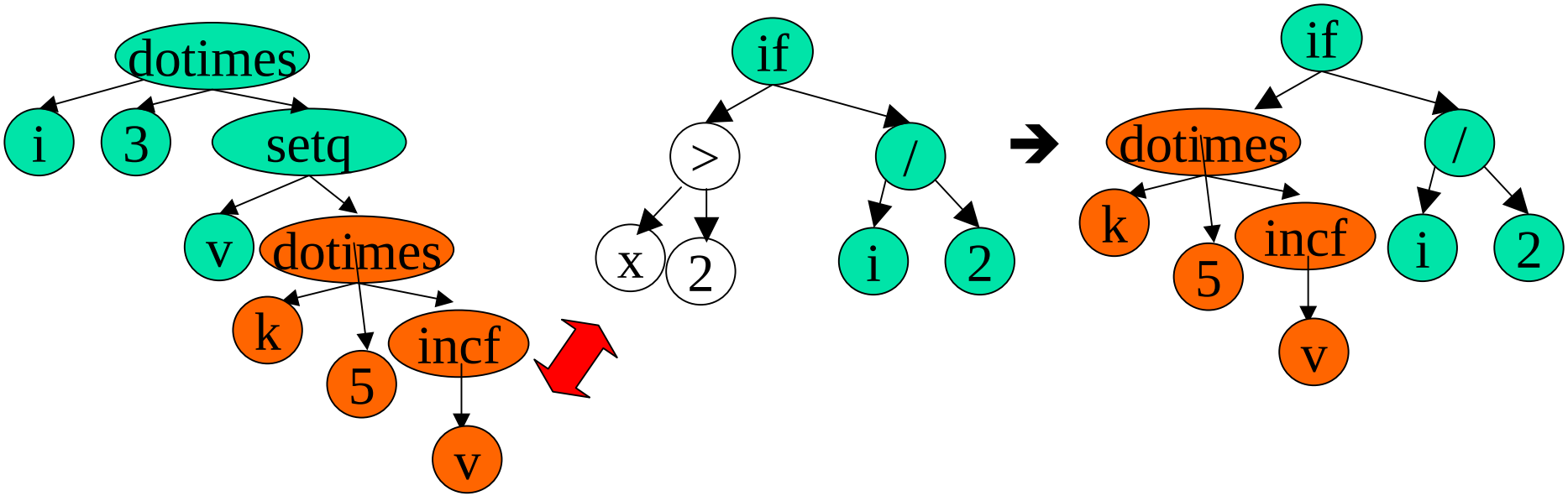
RAT Algorithm

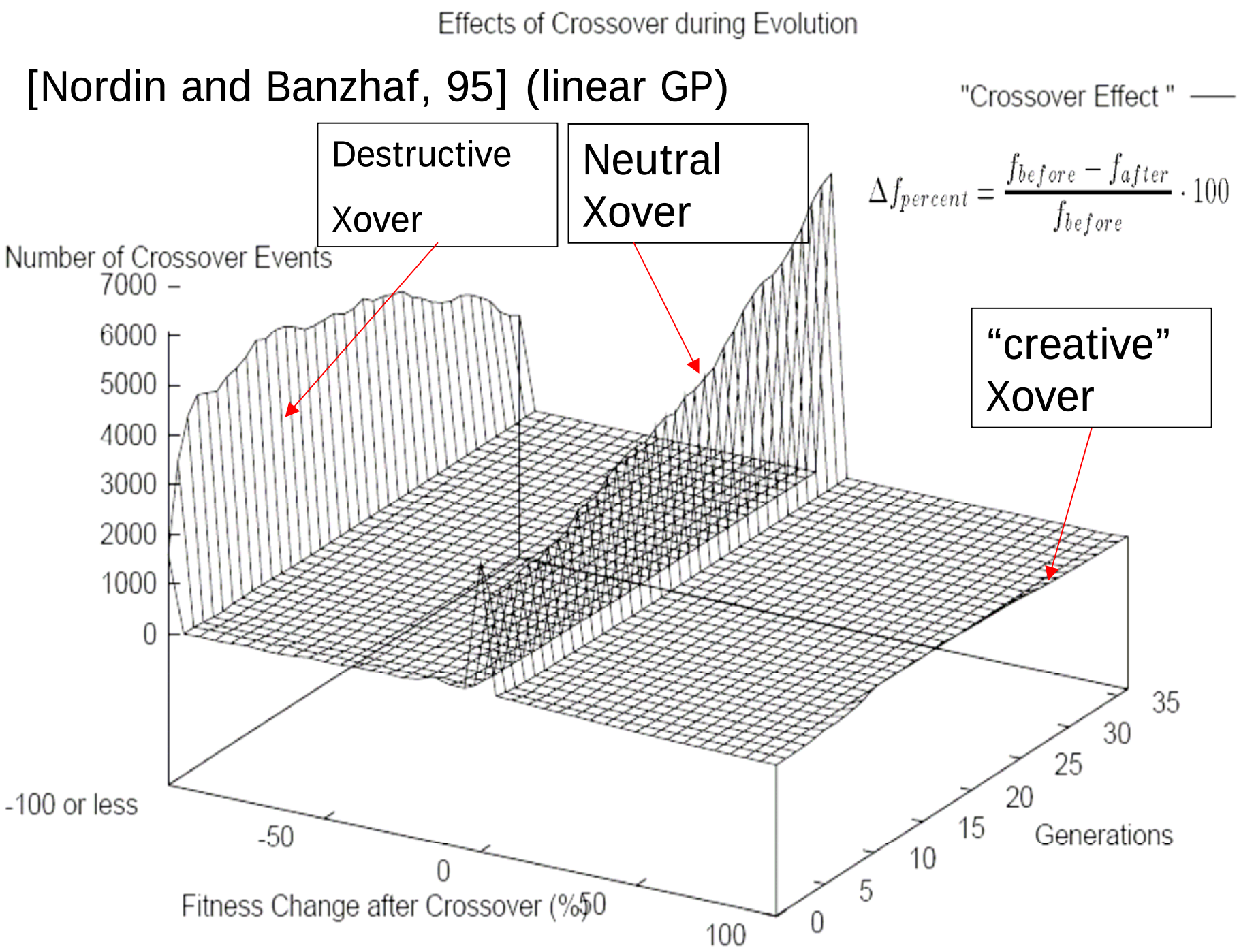
- The gaussians are the distributions of the true error (assuming normality)



Criticisms to Crossover

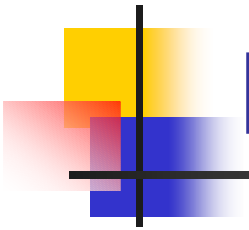
- It is not clear that crossover actually recombines features from both parents (effects of code are very context dependent)




Effects of Crossover during Evolution
[Nordin and Banzhaf, 95] (linear GP)
"Crossover Effect "
$\Delta f_{percent} = \frac{f_{before} - f_{after}}{f_{before}} \cdot 100$
Destructive Xover
Neutral Xover
"creative" Xover
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

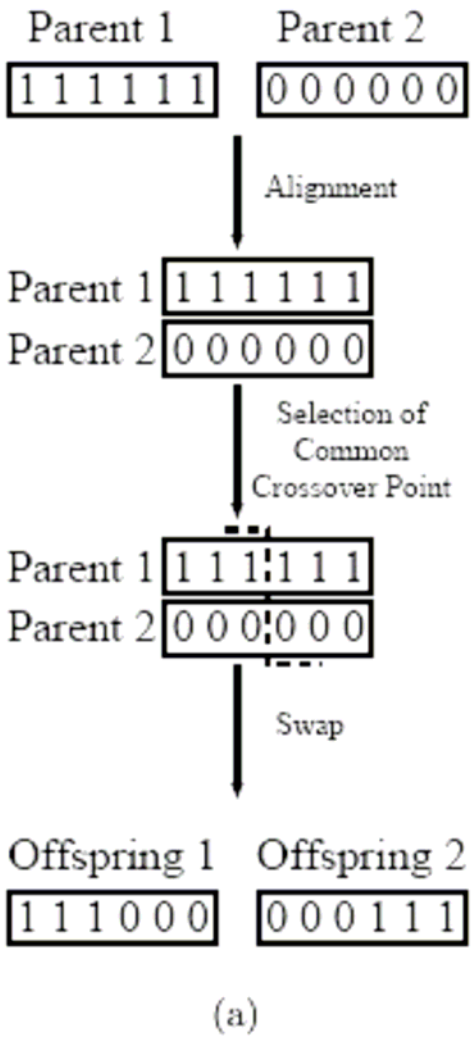
- Some studies show that crossover is basically a macromutation operator, although it works better than random search ([Lang, 95], [O'Reilly & Oppacher, 94], [Angeline, 97])
- [Luke, Spector, 97,98]: "A [Revised] Comparison of Crossover and Mutation in GP":
  - Crossover works slightly better than mutation
- [Chelapilla, 97]: "Evolving computer programs without subtree crossover":
  - Crossover not necessary



New crossover operators

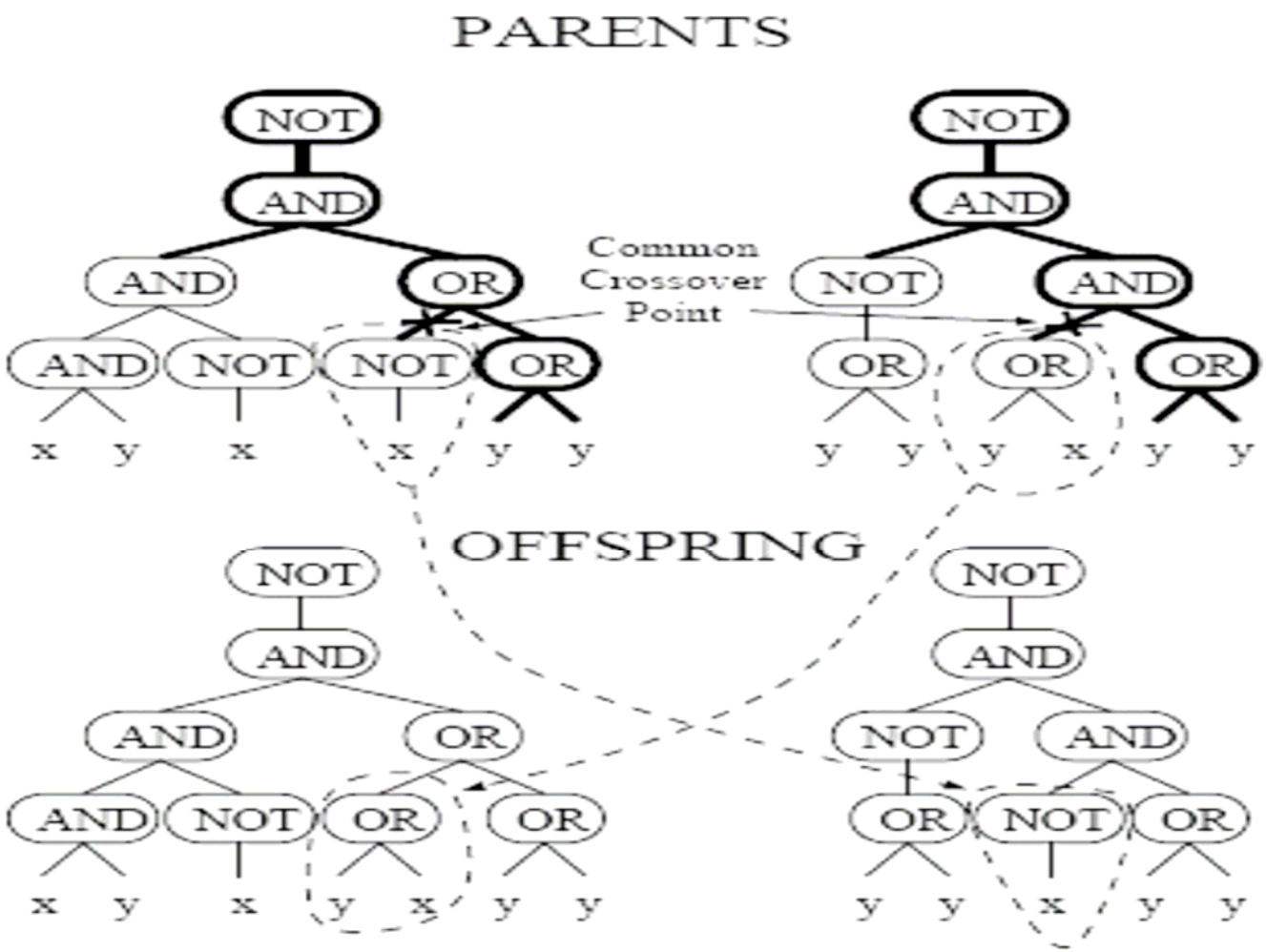
- “Brood recombination”: two parents cross many times, the best offspring is chosen [Tackett, 94]
- “intelligent xover”: choose the crossover point intelligently: PADO [Teller, 95]
- “Homologous xover”: subtrees are exchanged only at the same position [Haeseleer, 94] [Poli & Langdon]

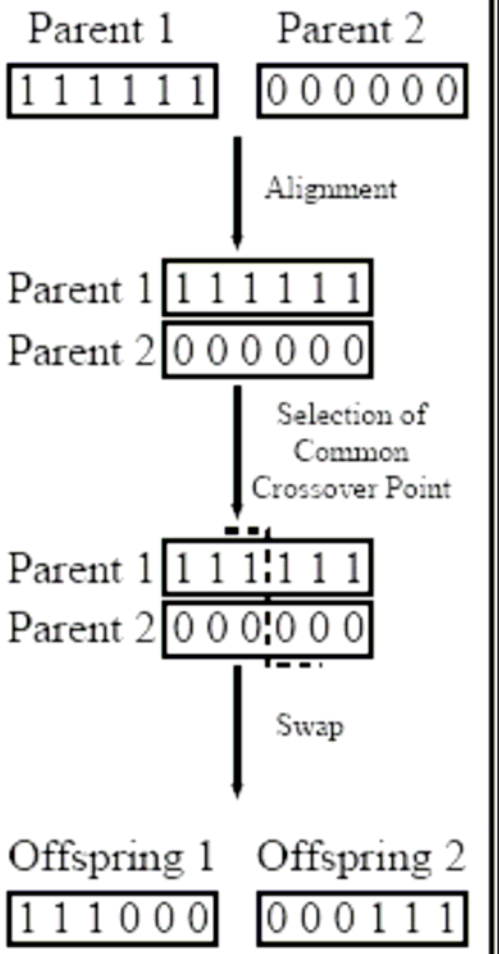
Crossover in Genetic Algorithms



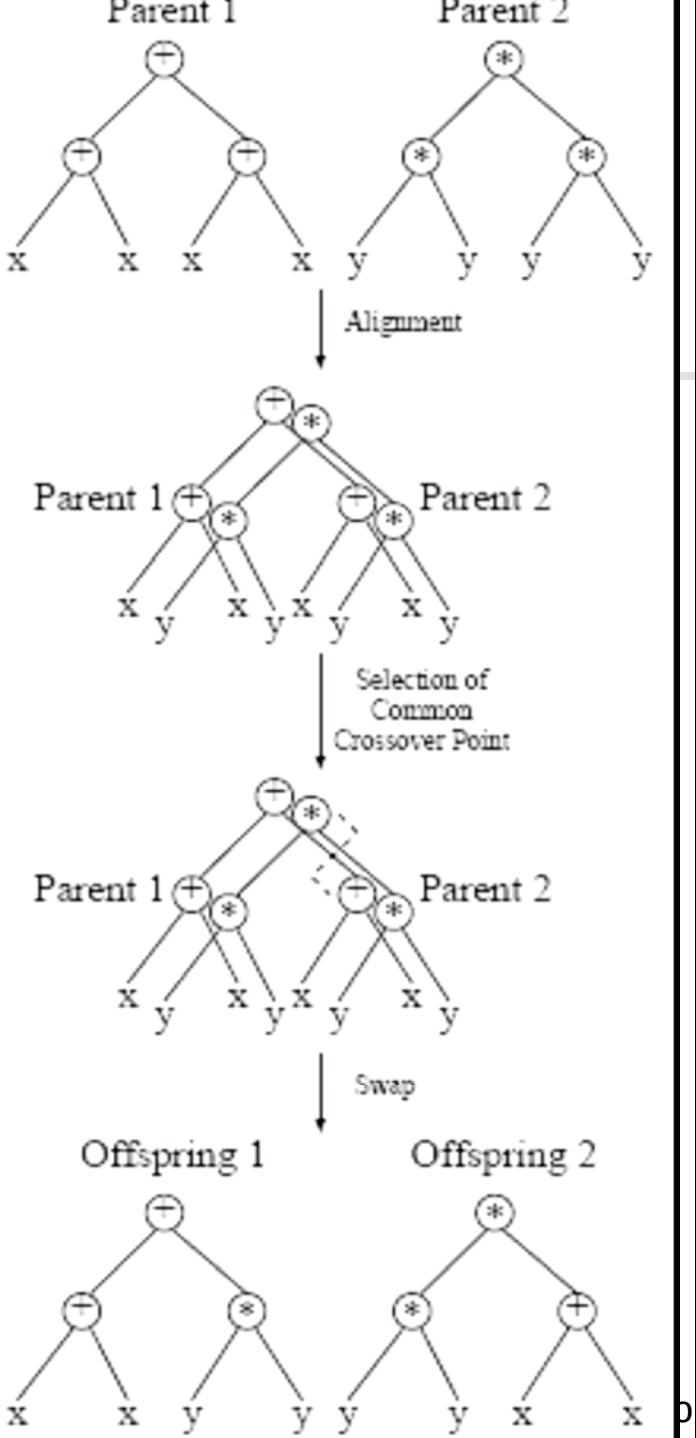
Crossover always maintains the position of bits

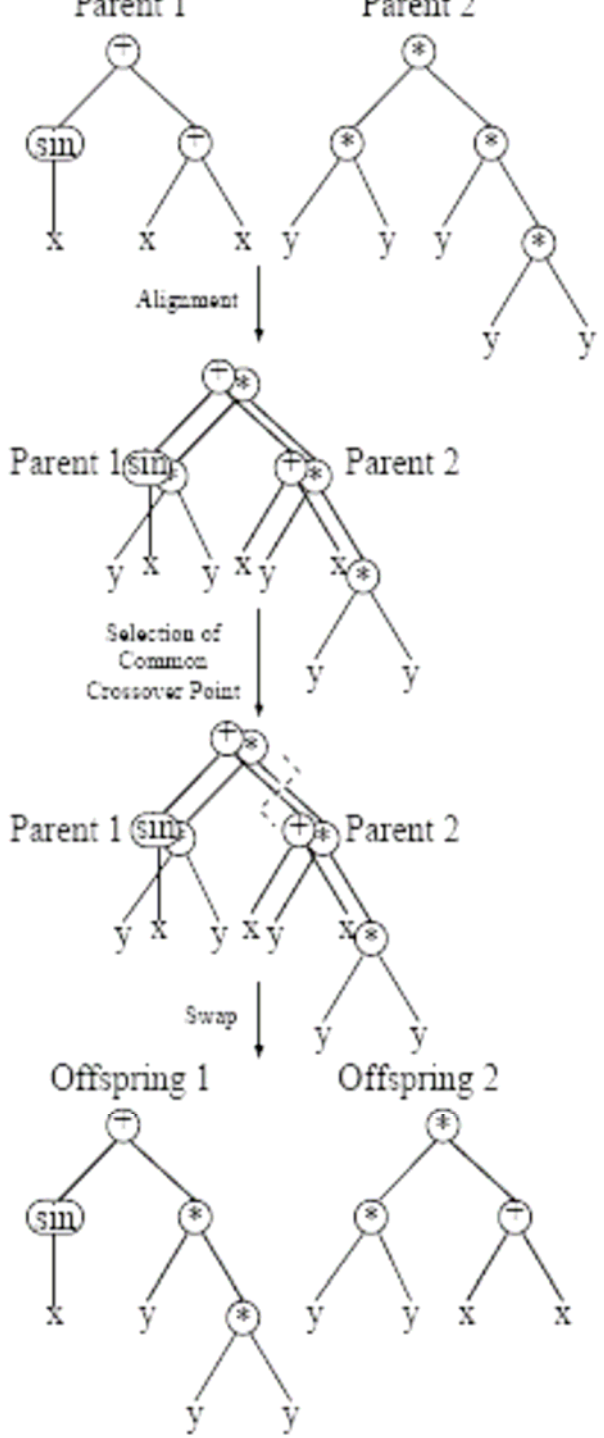
One Point Crossover

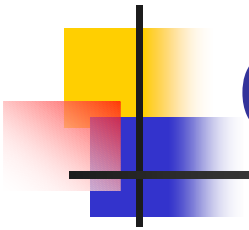



Parent 1
Parent 2
1 1 1 1 1 1
0 0 0 0 0 0
Alignment
Parent 1 1 1 1 1 1
Parent 2 0 0 0 0 0 0
Selection of Common Crossover Point
Parent 1 1 1 1 1 1 1
Parent 2 0 0 0 0 0 0
Swap
Offspring 1
Offspring 2
1 1 1 0 0 0
0 0 0 1 1 1

(a)


Parent 1
Parent 2
Alignment
Parent 1
Parent 2
Selection of Common Crossover Point
Parent 1
Parent 2
Swap
Offspring 1
Offspring 2

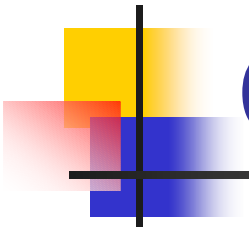

Parent 1
Parent 2
sin
Alignment
Parent 1
Parent 2
Selection of Common Crossover Point
Parent 1
Parent 2
Swap
Offspring 1
Offspring 2



One-Point Crossover.

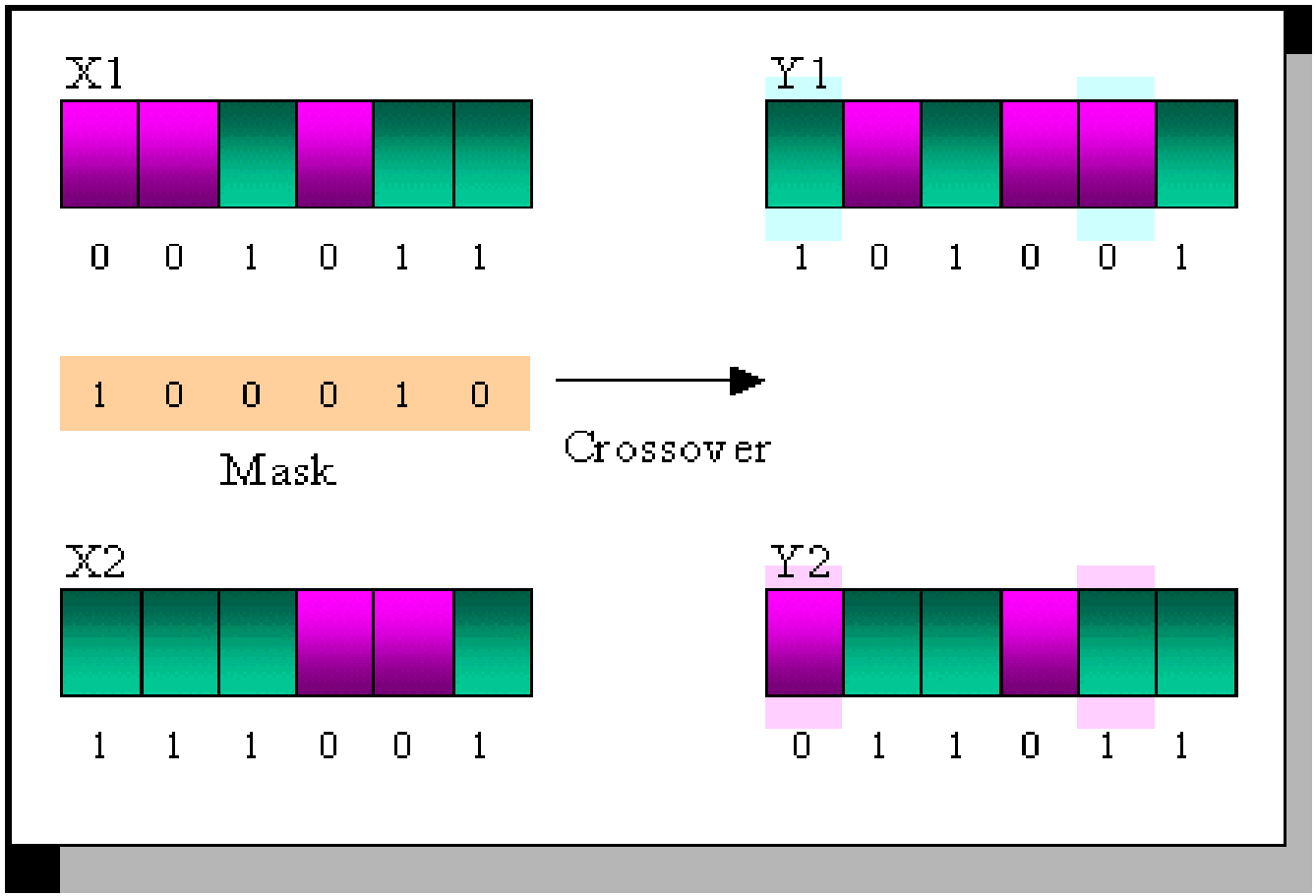
1. Alignment: Look for the common structure in both parents
2. Choose **one** random xover point in the common region
3. Exchange the subtrees

Properties of One-point Crossover

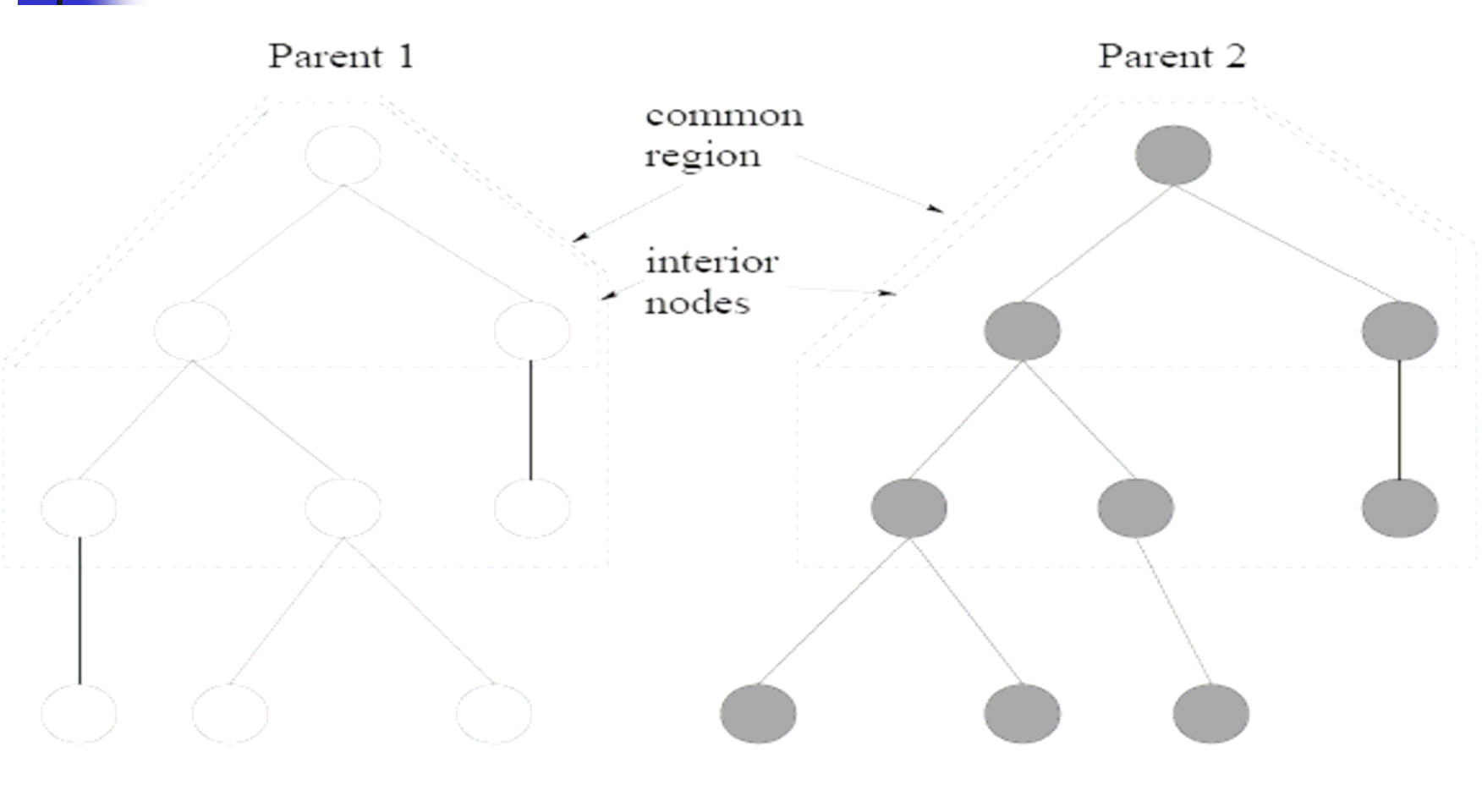


- At the beginning of evolution, xover points belong to the top part of parents (small common structure)
- As evolution progresses, some structures become prevalent and deeper regions are explored
- Top-down exploration, which makes sense for programs
- Point mutation required (like in genetic algorithms)

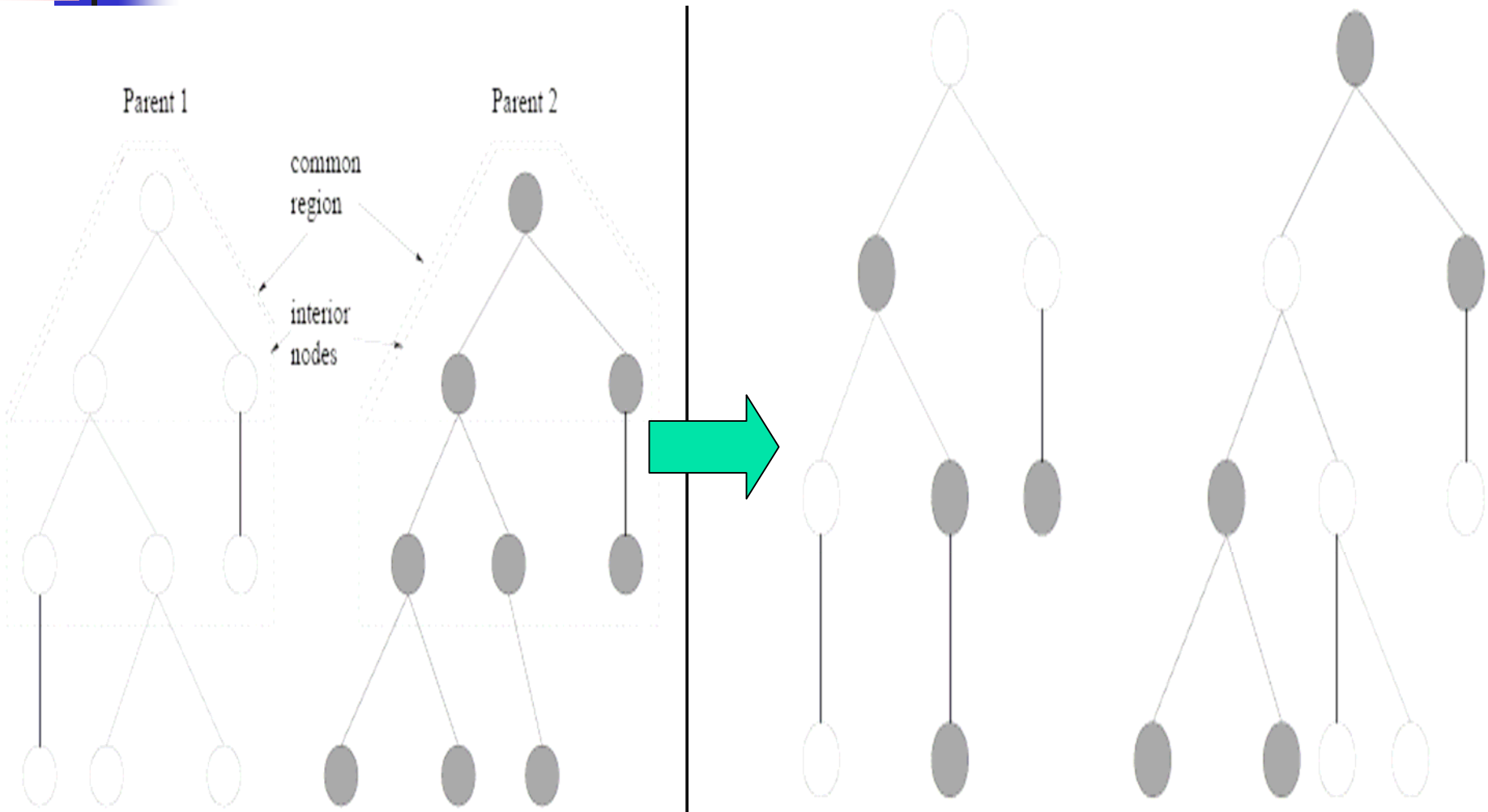
Uniform Crossover in Genetic Algorithms

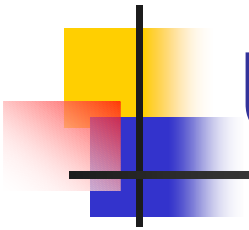


GP Uniform Crossover (GPUX)



GP Uniform Crossover (GPUX)



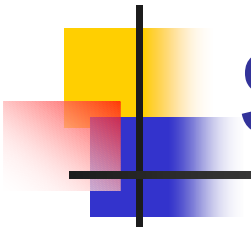


Uniform Crossover

- Determine the common and interior regions
- Exchange nodes:
  - If in the interior region, just exchange the nodes
  - If not, exchange the subtrees as well

Properties of Uniform Crossover

- Search becomes more global (offspring less similar to parents than the 1-point crossover)



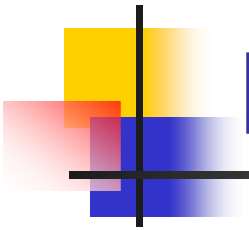
Smooth Operators

- Goal: to make small changes to programs
- Instead of exchanging two functions, they are sort of “averaged”
- GPSUX, GPPM: Smooth crossover and point mutation

Smooth Operators for Boolean Functions

p	q	¬p	p∧q	p∨q	p→q	p↔q
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	V

Smooth Crossover and Mutation



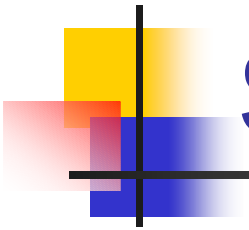
- AND(A,B) = ffft
- OR(A,B) = 0111
- Smooth xover AND/OR = 0f1t
- Smooth mutation AND = f1ft
- Not clear how this could be extended to non-binary problems
- (Note f=0, t=1)

Results on even-parity-5

Operators	Population size	Fitness evaluations	Complexity	Success Rate
Standard GP	50	11,250	428	12%
Standard GP	200	19,000	568	68%
GPUX, GPPM	50	4,200	84	88%
GPUX, GPPM	200	6,000	49	98%
GPPM only	50	4,200	68	80%
GPPM only	200	6,000	49	98%
GPSUX, GPSPM	50	2,250	82	92%
GPSUX, GPSPM	200	2,200	56	100%
GPSPM only	50	2,250	76	98%
GPSPM only	200	8,400	49	98%

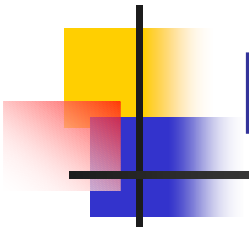
Results on even-parity-6

Operators	Population size	Fitness evaluations	Complexity	Success Rate
Standard GP	50	No solution found	N/A	0%
Standard GP	200	No solution found	N/A	0%
GPUX, GPPM	50	34,850	38	36%
GPUX, GPPM	200	127,600	40	60%
GPPM only	50	35,550	43	44%
GPPM only	200	71,400	51	82%
GPSUX, GPSPM	50	17,000	49	62%
GPSUX, GPSPM	200	19,200	53	80%
GPSPM only	50	16,200	59	67%
GPSPM only	200	18,400	42	82%



Smooth operators

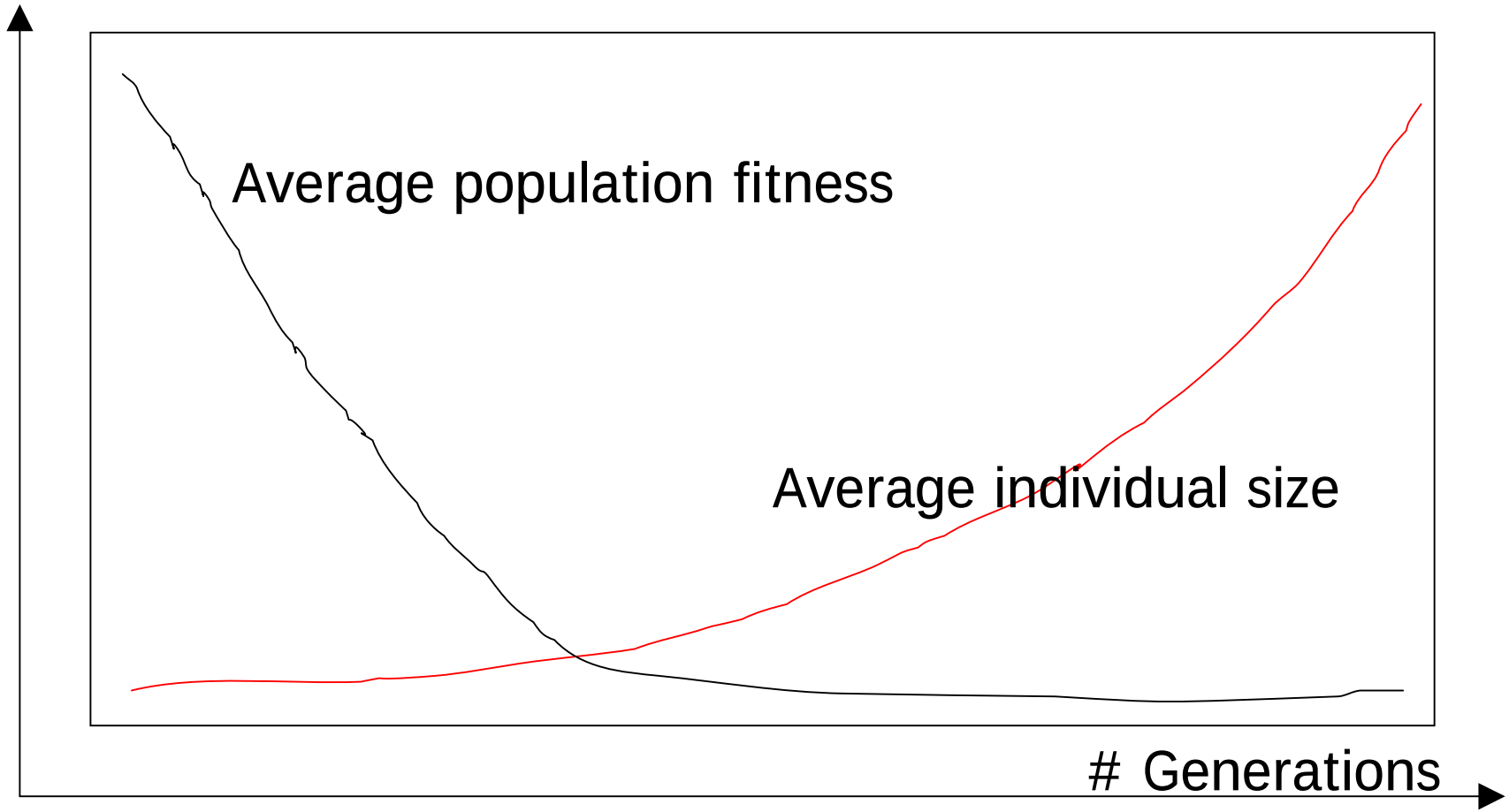
- They seem to work better than non-smooth
- Not clear that xover is required in this problem

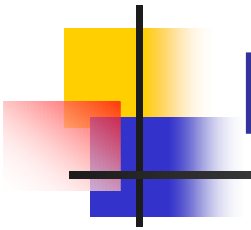


Bloat

- After several generations, individual size tends to grow, with no increase in fitness
- Fast growth, nearly quadratic [Langdon, 00]
- Caused by “introns”: a+ (a-a+a-a), 0*(a*b+c*d), if(F) then {…}

Bloat


Average population fitness
Average individual size
Generations

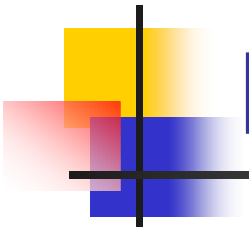


Bloat Problems

- Individuals take longer to run and take more memory
- Search stagnates (genetic operators change unused regions)
- Although it is reported that in some occasions, introns protect sensitive parts of the code

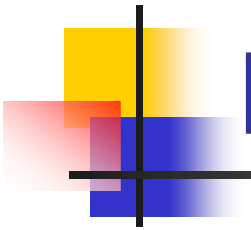
Bloat. Why?

- It is expected it will happen when evolving variable-size structures, with a fixed fitness function
- ***Accuracy theory***: Defence against crossover, specially at the end of the run, when it is difficult to improve fitness
- After a particular size, fitness is independent of fitness. There are more large programs than small ones, so there is a tendency to grow
- ***Removal bias theory***: It is easier to add small subtrees (inside the introns) than to remove large subtrees



Bloat Control

- Parsimony: fitness penalty for large individuals (k?):
  - F'(x) = F(x) + k*size(x)
- Limit maximum size (which one?):
  - Problem: you get bloat!
- Assign a bad fitness to large individuals so that they are not selected in the next generation



Bloat Control

- Tournament selection. If two individual draw in fitness, then select the small ones
- Avoid destructive crossover:
  - brood recombination

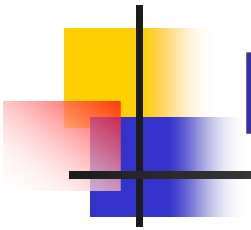
Bloat Control. Tarpeian Method

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

Removes a percentage of larger-than-average programs

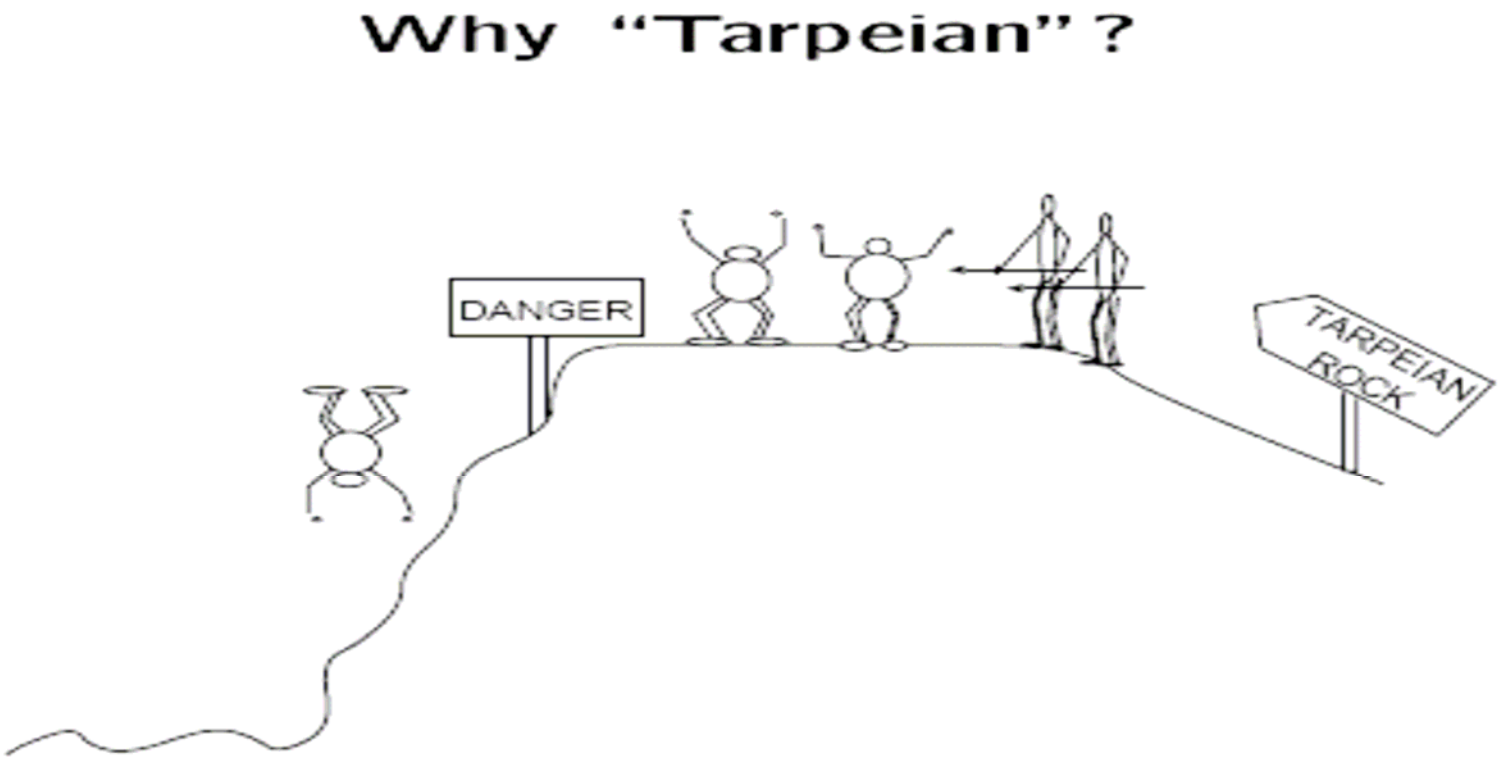
n >= 2 (n=2, half of big programs die)

Bloat Control. Tarpeian Method

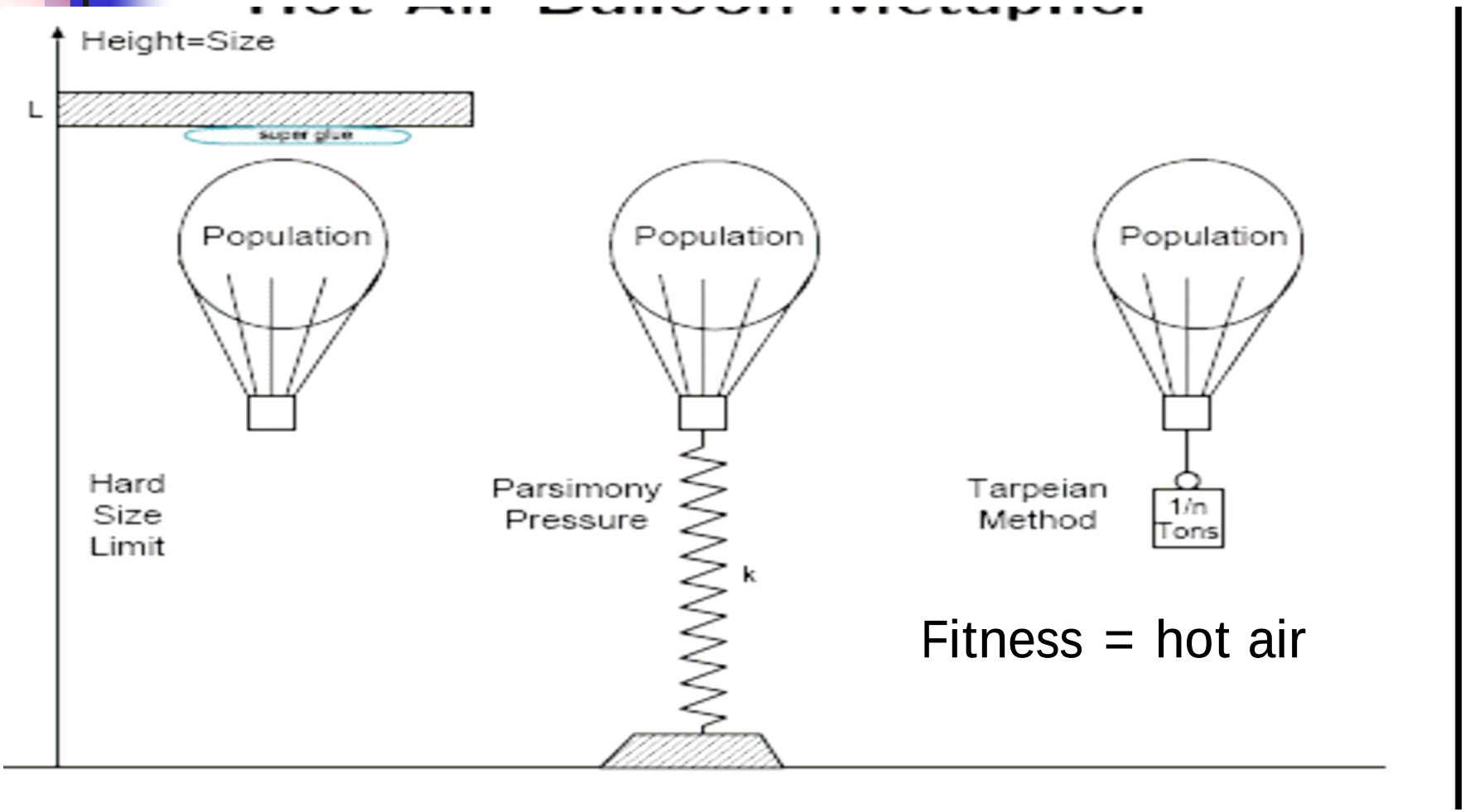


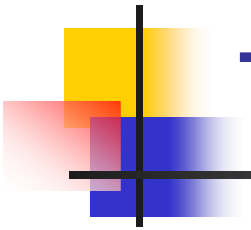
- Larger-than-average individuals are more likely to die
- If it survives and it is a better than average individual, it will reproduce and the average size will increase
- But there will still be opposition to growth
- Saves time on large individuals (not evaluated)
- Justification based on schema theorem

Tarpeian: Fitness Hole for Large Individuals



Tarpeian: Balloon Metaphor



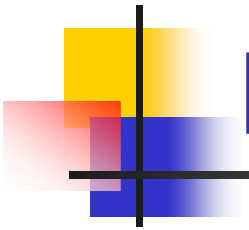


Tarpeian Properties

- Valid with point-mutation (no subtree mutation)
- Beware of premature convergence because of:
  - Small populations
  - Large pressure selection
  - Too large bias against large individuals (with *n* small)

Adding Syntactical Restrictions

- Standard GP requires “closure” (i.e. Protected functions)
  - 3/0 = 1
  - “dog” + 4 = 4
- Search space larger than necessary
- Very unnatural solutions (some believe this is an advantage):
  - if (3+”dog”) then {10/0}
- Solution: Use grammars or types



Main Grammar Works

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

Context Free Grammars

```
<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]

Example of individual:

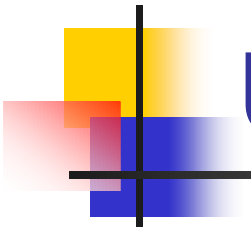
DNF -> (OR (<TERM>) (<DNF>)) ->

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

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

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

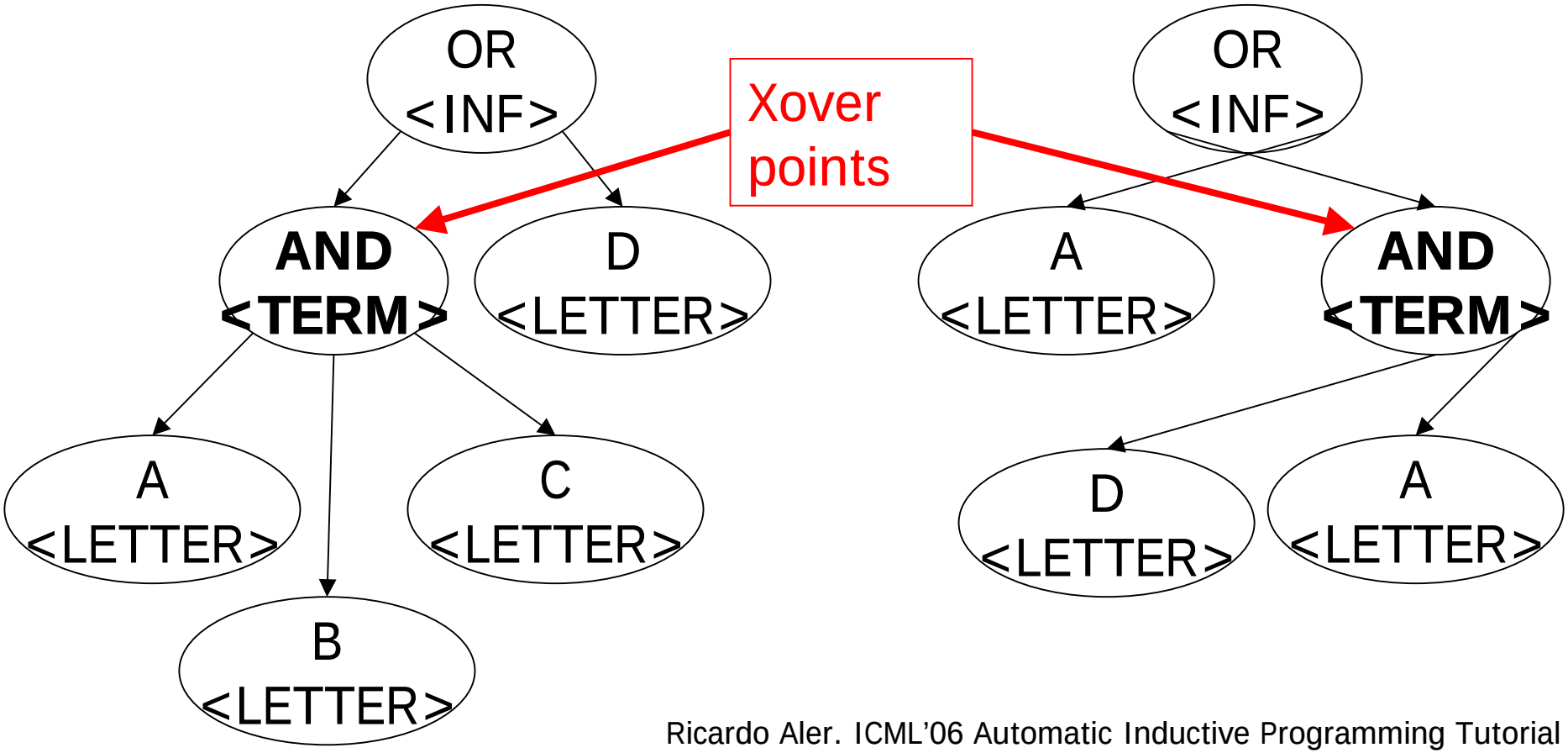
(<DNF> (<TERM> (<LETTER> <LETTER> <LETTER>)) <LETTER>)



Using Grammars

- To generate the initial population (production rules used randomly)
- To generate syntactically correct individuals by crossover: choose two xover points generated by the same production rule (ex: ‘A’ can be exchanged by another <letter>, like C)

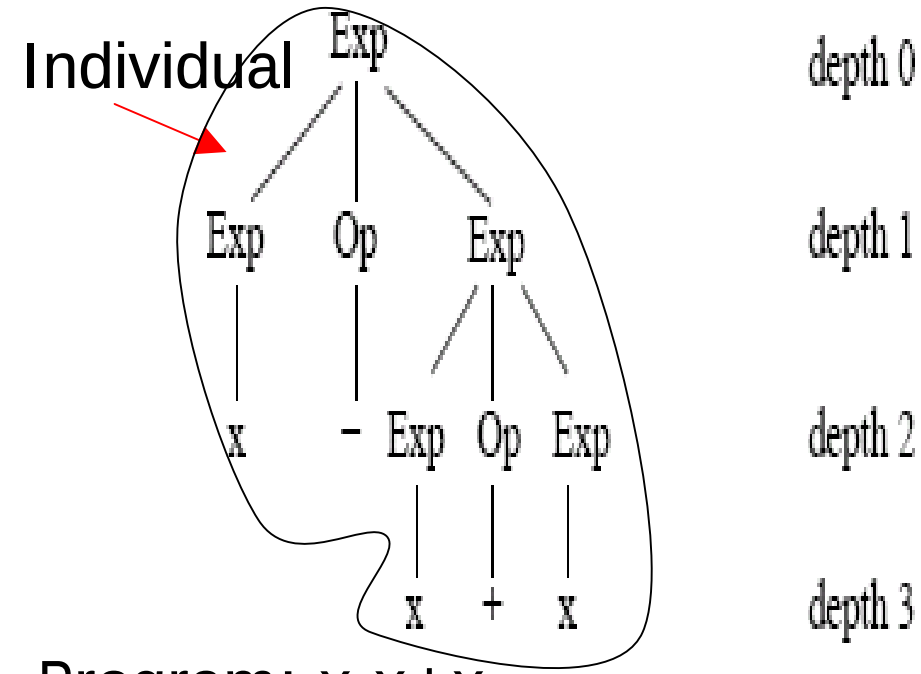
Example of Grammar Guided Crossover



Grammars in [Whigham, 96]

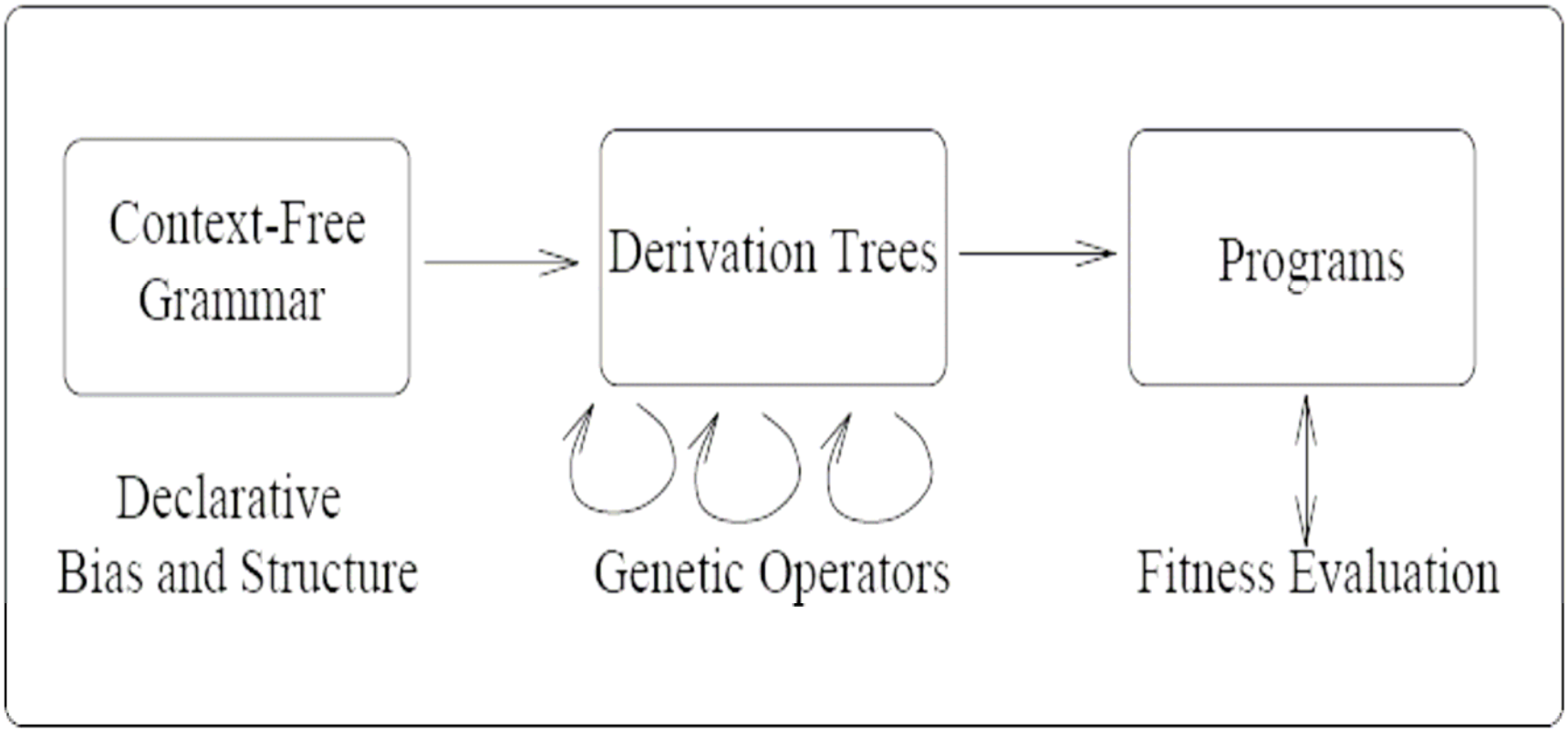
- Individuals are derivation trees
- Makes crossover easier
- It is necessary to build the program for fitness computation

S	→	Exp	(0)
Exp	→	Exp Op Exp |	(1)
	→	Pre Exp |	(2)
	→	x	(3)
Op	→	+ |	(4)
	→	− |	(5)
	→	× |	(6)
	→	/	(7)
Pre	→	sin |	(8)
	→	cos |	(9)
	→	e^ |	(10)
	→	ln	(11)

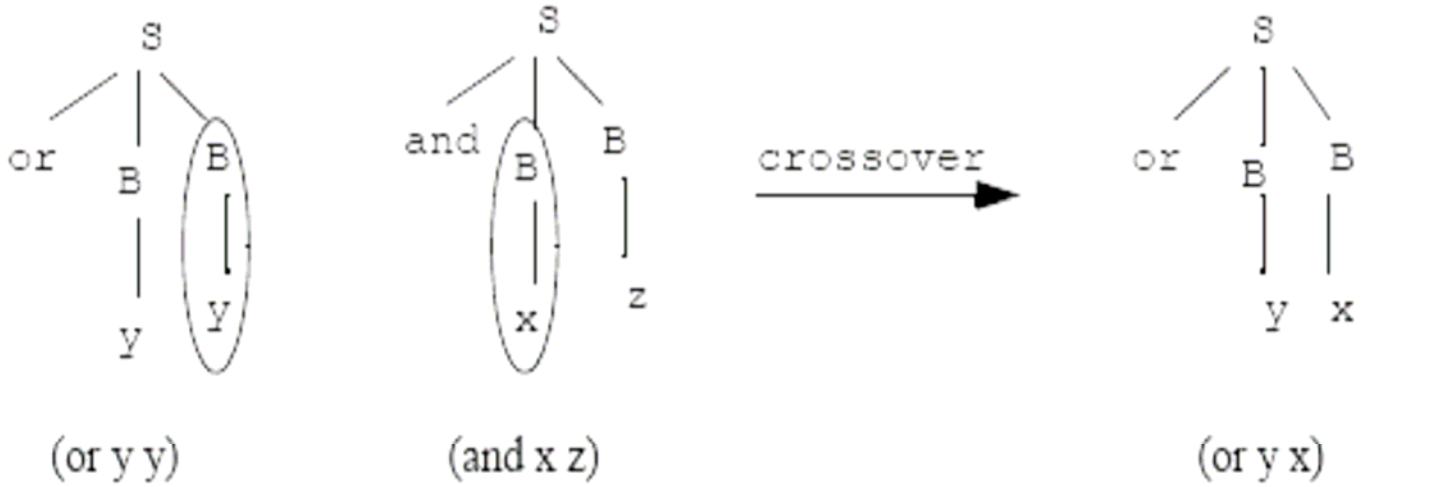


Program: x-x+x

Working with Derivation Trees

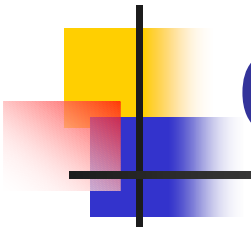


Derivation Trees Crossover




```
S ::= notB | andBB | orBB | x

B ::= x | y | z
```

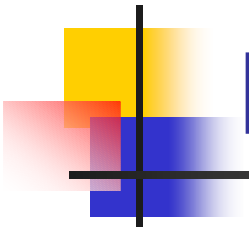


Grammar Adaptation

- P. A. Whigham. 1995. “Inductive Bias and Genetic Programming”.
- Grammars can be changed as evolution progresses, so that they generate good individuals more likely

Other Grammar/Types Work

- M. L. Wong, K. S. Leung. 1995. Genetic Logic Programming and Applications. IEEE Expert, 10(5).
- D. Montana. 1995. Strongly Typed Genetic Programming. Evolutionary Computation



Reuse in GP

- Of computations: store them in a **variable** or data structure (arrays, lifo, fifo, ...)
- Of parameterized code: **subroutines** (ADF: Automatic Defined Functions)
- Of repetitive code: **iterations**, **loops**, **recursivity**
- Only with loops or recursivity, GP is Turing-complete

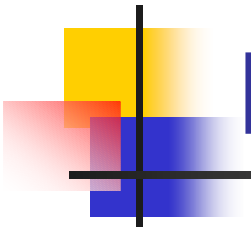
Use of Variables

- Add functions to read and write on variables or arrays:
  - Ex: (write-m 3.0) (read-m)
  - EX: indexed memory (arrays):
    - (write-array-m 5 3.0)
    - (read-array-m 5)
- Other data structures can be used (queues, stacks, …)
- Problems with (global) variables: secondary effects, no functional programming anymore

Example: variable *m*

```
Int write-m (value) {
  m = value;
  return(m);
}

Int read-m () {
  return(m);
}
```

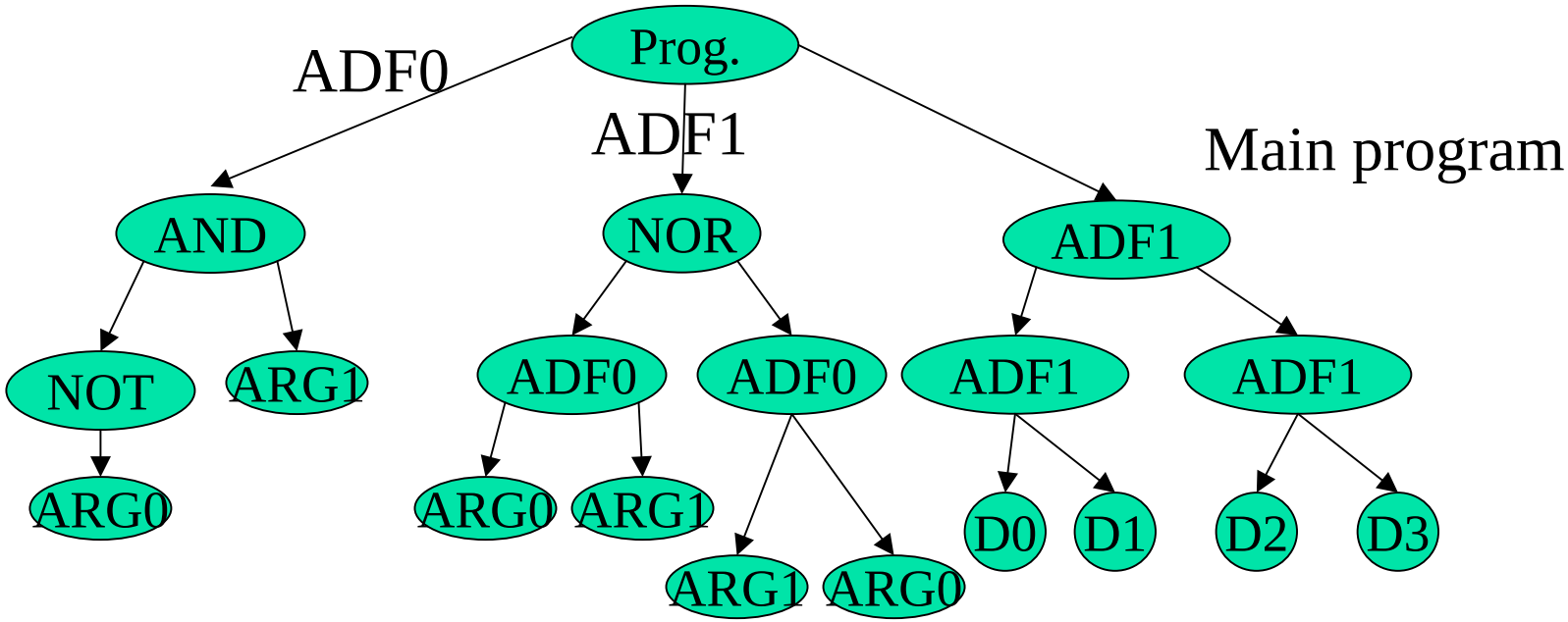


Evolving Subroutines

- Human programmers write subroutines for:
  - Creating new primitives, more amenable to the problem at hand
  - Generalize similar pieces of code found in differents parts of the main program
- By allowing subroutines, it is easier to write code, and the final program is simpler
- Can GP use subroutines?

ADFs (Automatically Defined Functions: subrutinas)

Each individual evolves its own subroutines in different branches of the tree



Homologous crossover

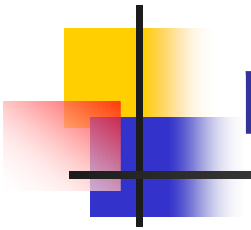
Effort and size for even-parity

Even-parity	Effort w/o ADF	Effort with ADFs	Size w/o ADF	Size with 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	YES		

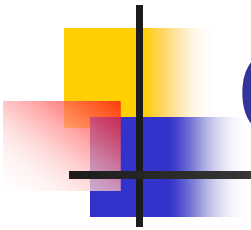
Other Subroutine Works

- 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*

How Many ADFs and how many parameters?



- Try and test, starting with small values
- Use many ADFs and parameters and let GP discover how many are needed
- Add automatic structure alteration operators:
  - Duplicate ADF or arguments
  - Remove 1 ADFs or 1 argument



Conclusions ADFs

- GP can evolve a main program and several subroutines
- If the problem is complex enough, computational effort and final size decrease a lot
- Good idea, use them!

Iteration in GP

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

Iteration in GP [Teller, 1994]

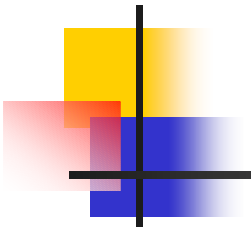
- Available primitives:
- (IF X THEN Y ELSE Z)
- (= X Y)
- (AND X Y)
- (ADD X Y), (SUB X Y)
- Indexed memory (array):
  - (Read X), (Write Y X)

Iteration in GP [Teller, 1994]

- Then, any algorithm can be expressed as:

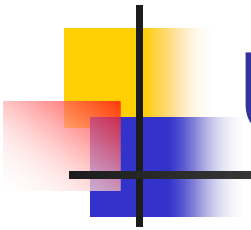
REPEAT <GP+IM function>

UNTIL <some state happens in memory (for instance a flag is raised)>



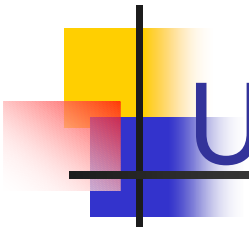
Iteration in GP [Teller, 1994]

- No loops are needed for Turing-completeness! (not completely unexpected)
- Just evolve a GP+IM program
- In practice, it may be easier to evolve programs with explicit loops



Use of Loops

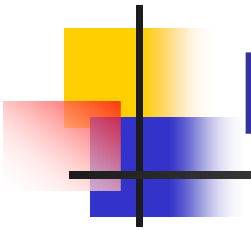
- Add to the function set a function that implements the loop:
  - (loop times loop-body)
  - (loop 10 (write-m (* (read-m) i))



Use of Iterations and Loops

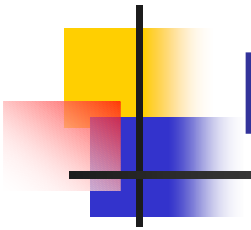
```
int loop (times; body) {
  int i; int times, result;
  for (i=0;  i<times; i++) {
    result=evaluate_tree(body);
  }
  return(result);}
```

Limitations of Loops (and Recursion)



- Not used often
- Not well studied in GP
- Increase a lot fitness computation time
- Iterative programs are fragile

Solutions for Loops and Recursivity



- Limit:
  - Fitness computation time
  - Number of loops
  - Number of iterations or recursive calls
  - Loop nesting
- Coroutine model [Maxwell, 94]: Run programs in parallel and cancel the bad ones
- Implicit recursion by means of high-level funcions: **map**, **foldr**, ... [Yu, 01]

Turing-complete Program Space

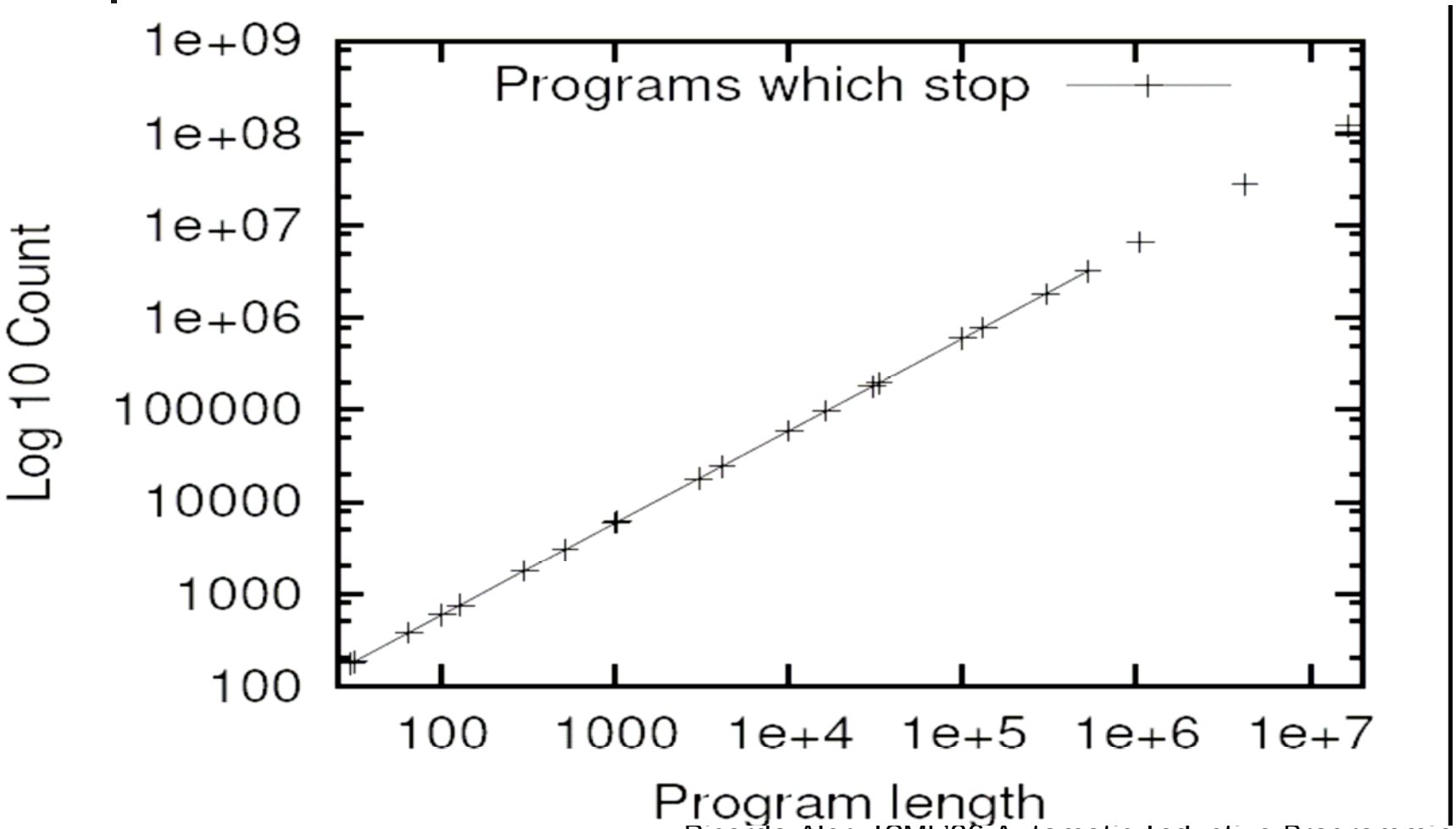
- W. B. Langdon and R. Poli. **"The Halting Probability in von-Neumann Architectures"**. *EuroGP' 06*
- Space made of machine code random programs

Machine Code

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.

Programs that Terminate



Number of Programs that End

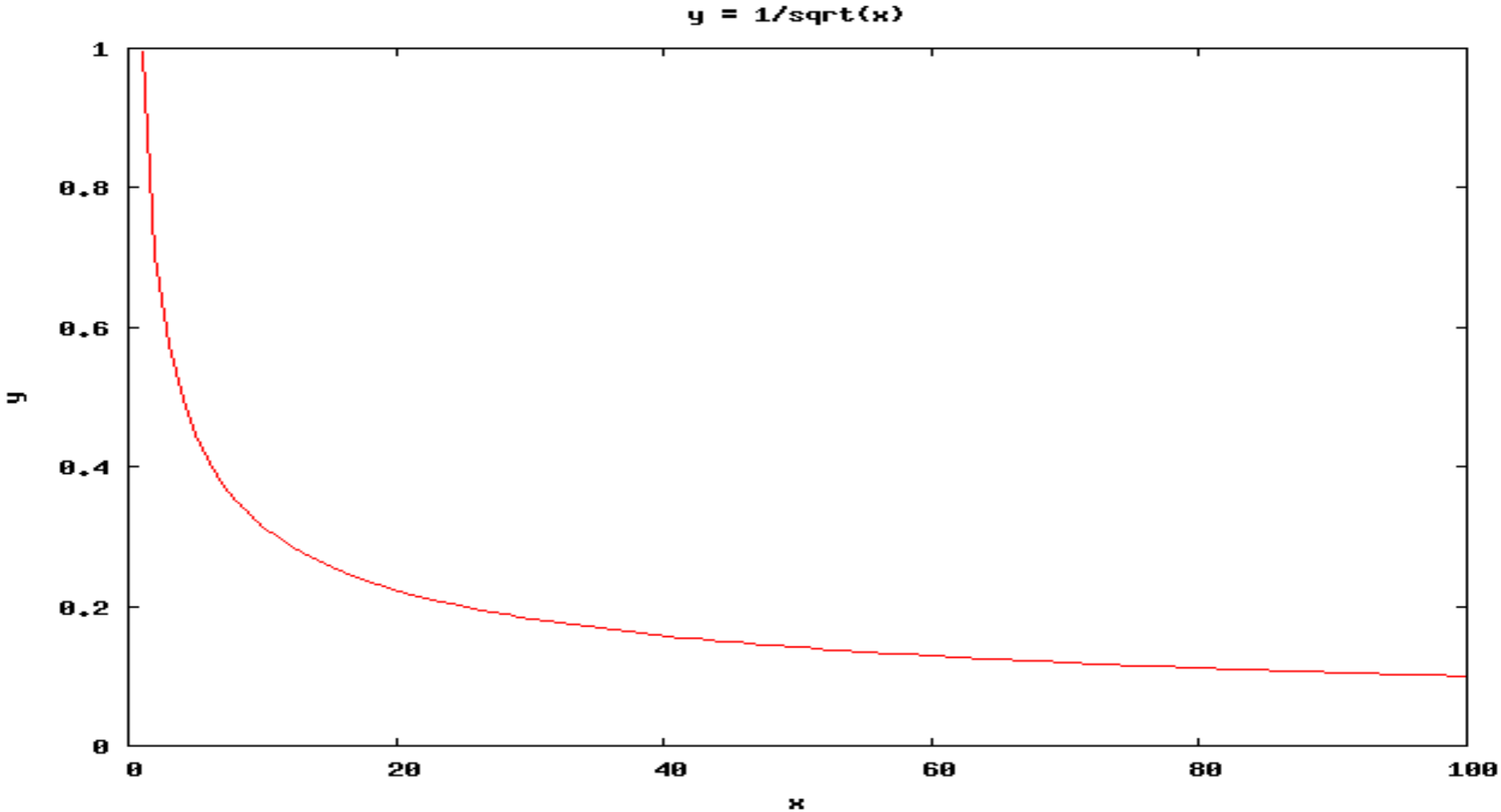
- The number of programs that end grow exponentially with length
- But the total number of programs grows much faster
- The proportion is:

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

- Execution time of programs that end is proportional to:

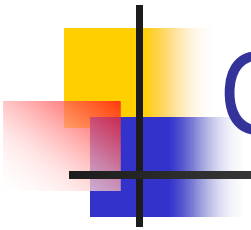
$$\sqrt{\text{length}}.$$

Proportion of programs that end



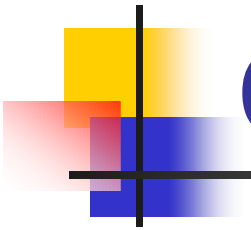
Coroutine model [Maxwell, 94]

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



Coroutine Model [Maxwell, 94]

- Problem of limiting time:
  - Threshold may be too small or too large
- Coroutine model:
  - Allows “parallel” run of programs
  - Do not limit execution time
  - Steady-state model: good/fast individuals replace bad/slow individuals

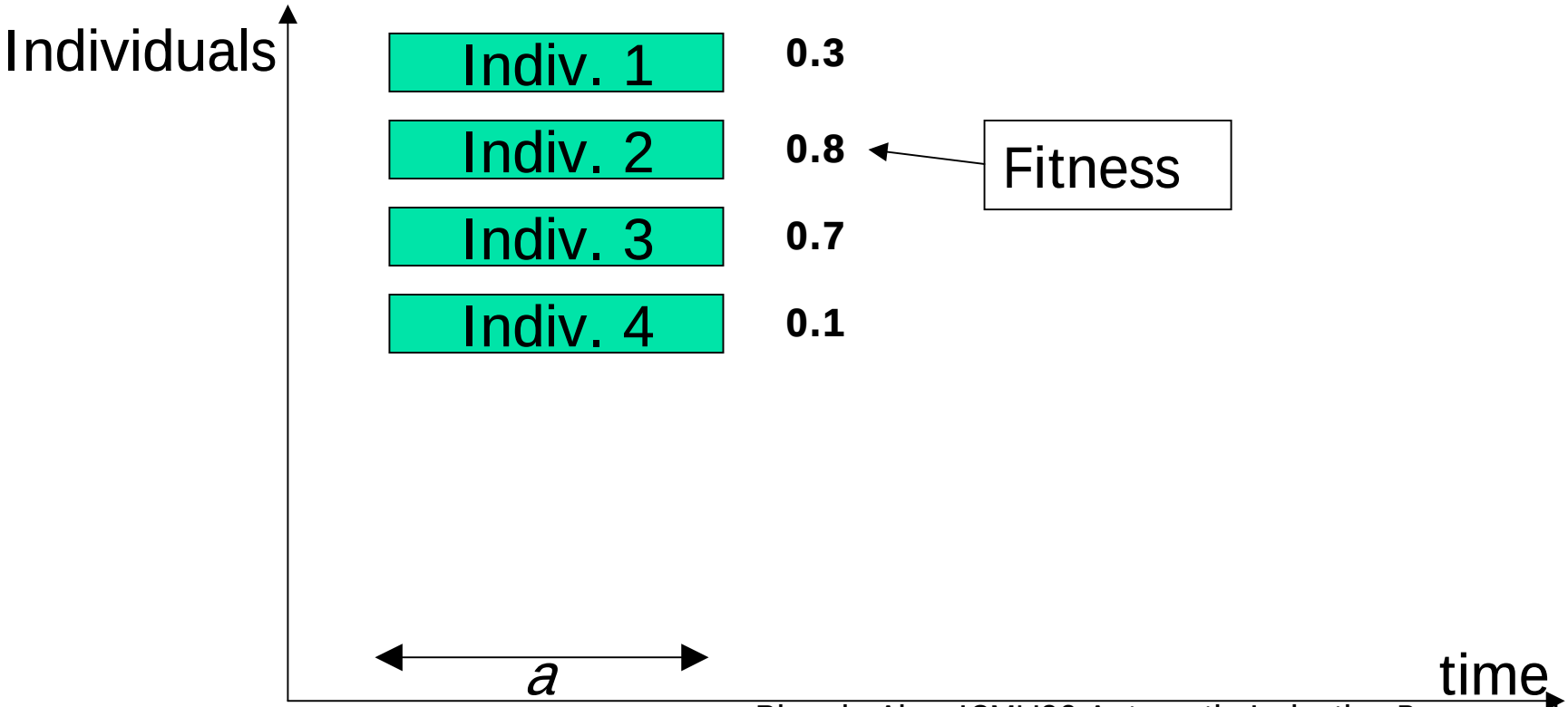


Coroutine Model

- Fair comparison: compare individuals with the same age (running time)
- Requires programs return partial fitness
- Not that difficult if there are many fitness cases (fitness accumulated so far)
- Or in problems similar to the Pacman

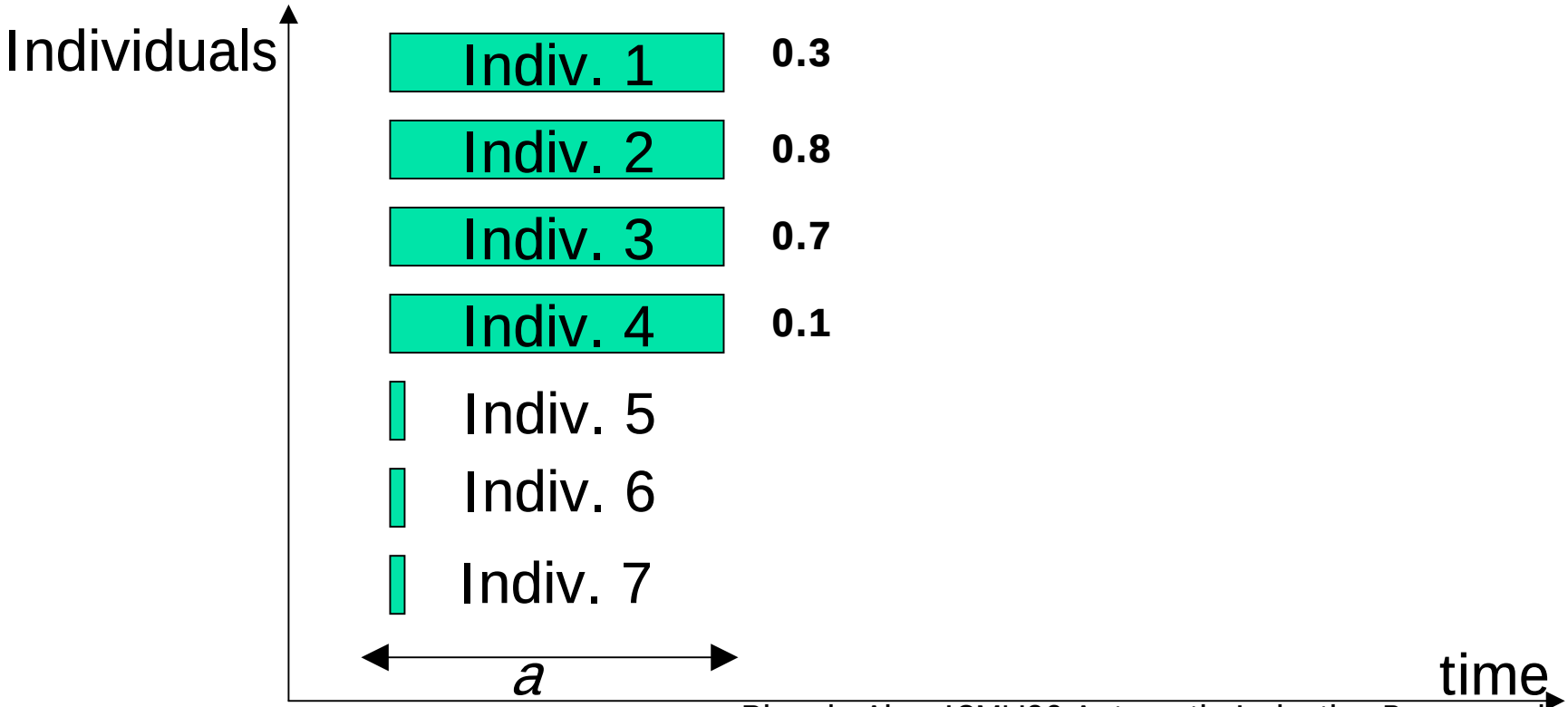
Coroutine Model

- Create initial population and run them for *a* seconds. Compute partial fitness



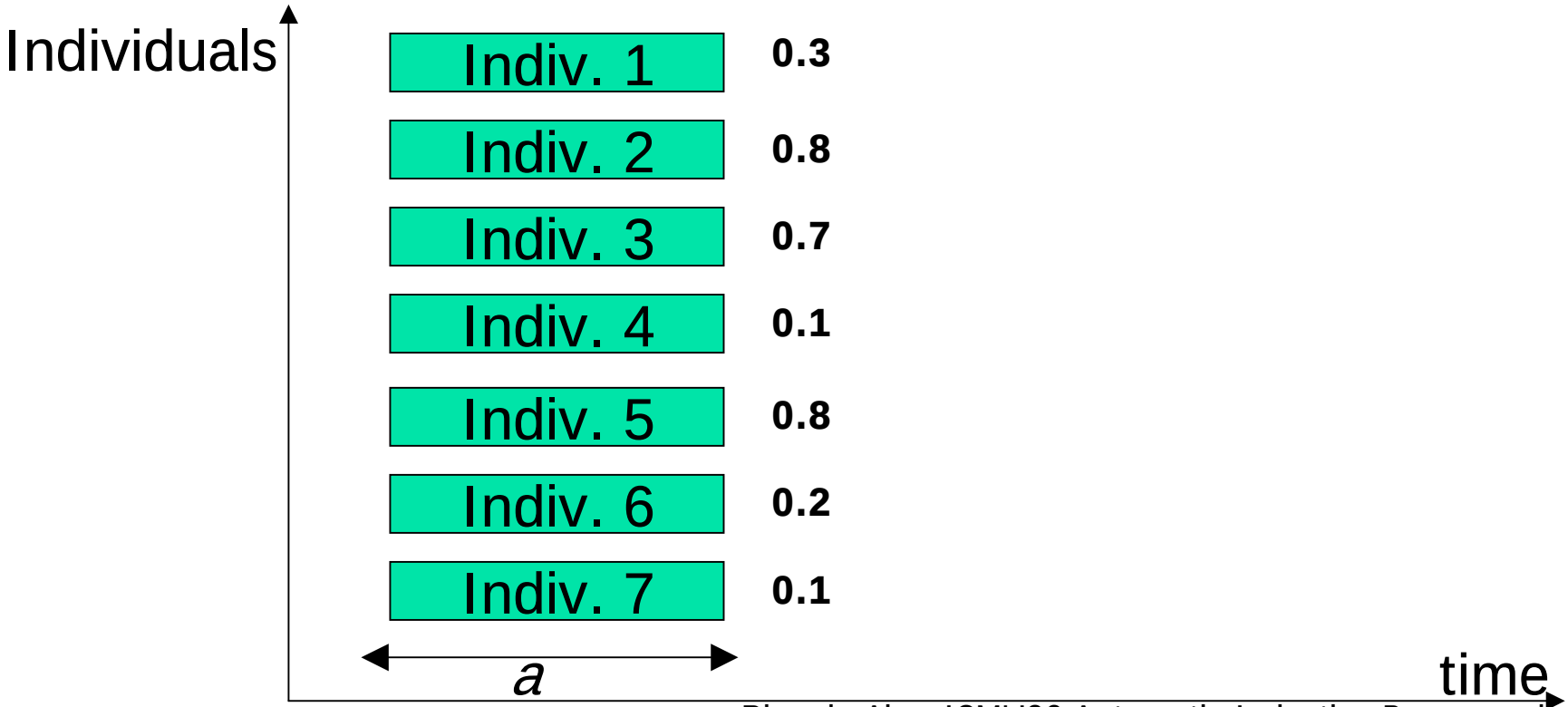
Coroutine Model

- Create N new individuals by means of tournament and genetic operators



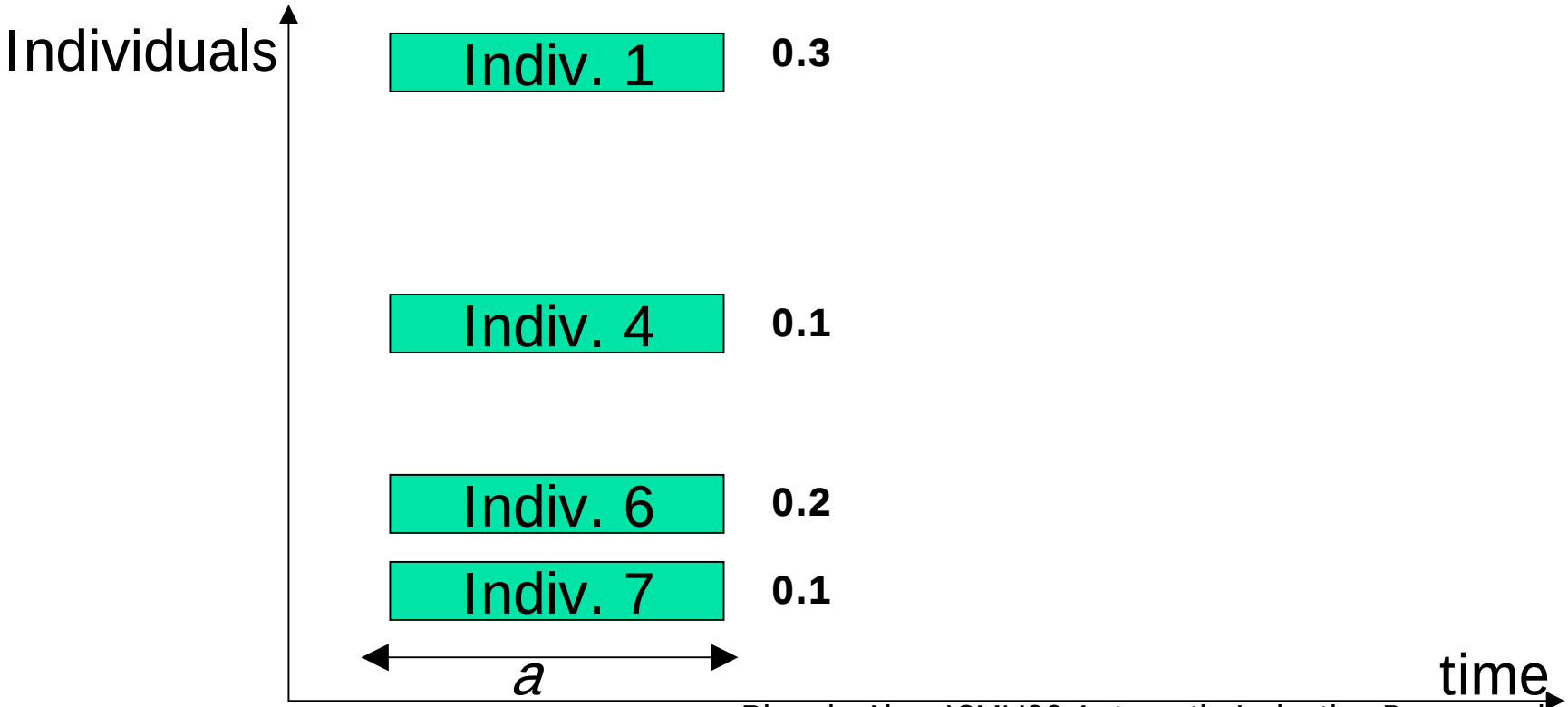
Coroutine Model

- Run the N individuals for the same time than the rest of individuals in the population



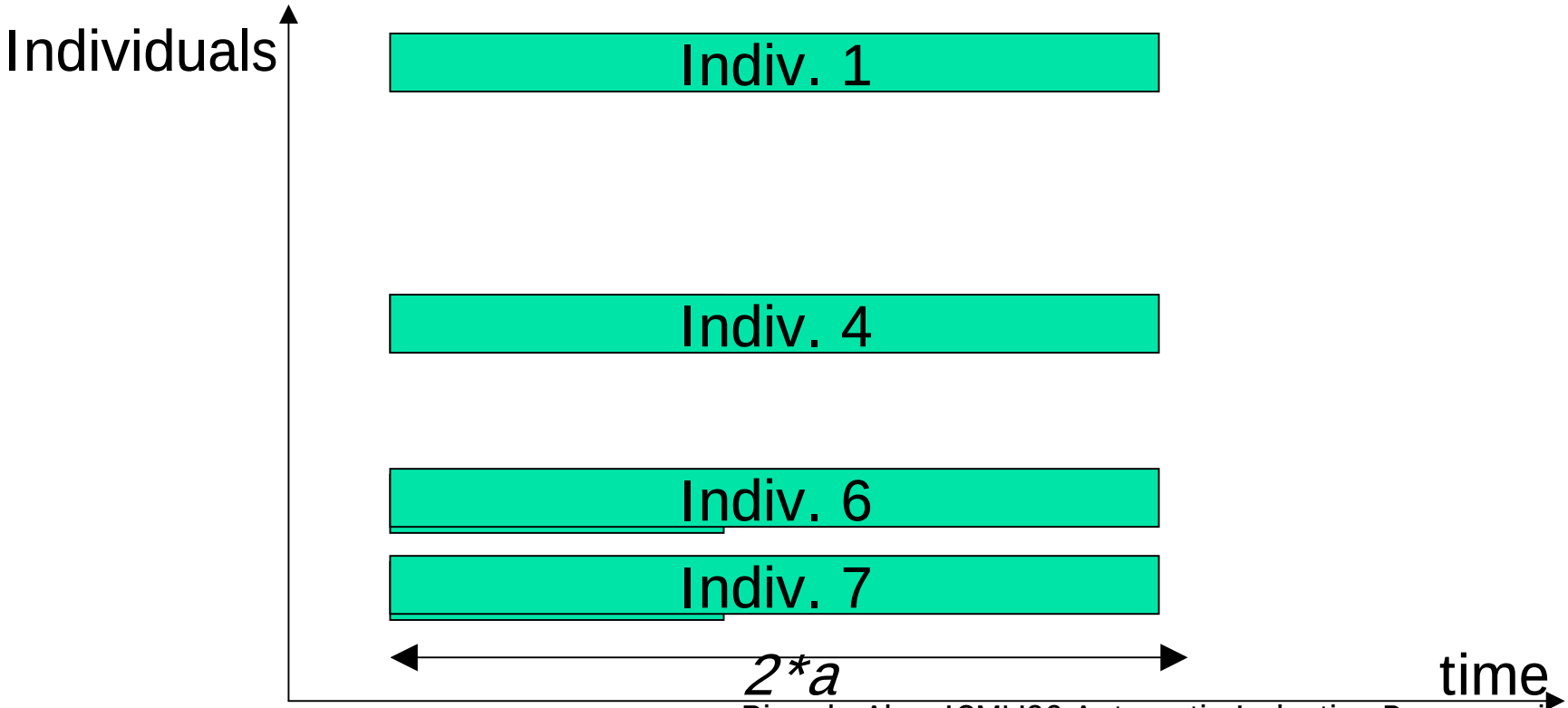
Coroutine Model

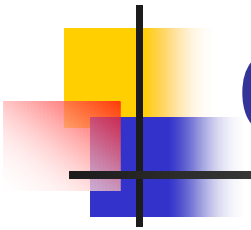
- Remove the worse individuals by tournament



Coroutine Model

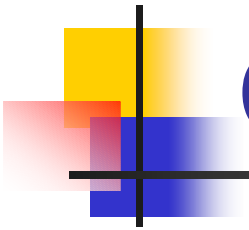
- Run individuals for another time *a*. And so on ...





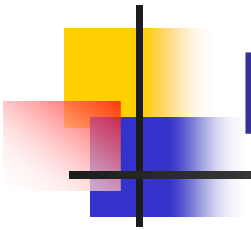
Coroutine Model

- It works well when an individual appears, that solves all fitness cases in a finite time
- Experiments show that it generates more efficient individuals
- But there is no guarantee that a better individual at a particular time *t* will be the best one in the long run. It is just a heuristic



Other Possibilities

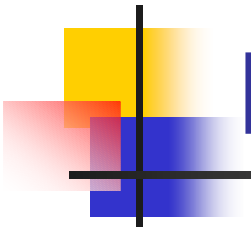
- Coroutines difficult to implement
- Use different threads?
- For every generation, store the best fitness obtained so far
- Cancel all programs that at that time, get a worse fitness than the best obtained so far
- Or downward adjust its priority



Experiments with loops

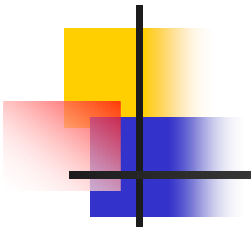
- Not too many! Hard for GP, room for research

Evolving a Sorting Algorithm



- Kinnear. 1993. **"Generality and Difficulty in Genetic Programming: Evolving a Sort".** *Fifth International Conference on Genetic Algorithms*
- Several general sorting algorithms were evolved. $O(N^2)$
- Fitness function: complex, but basically, it counts the number of swaps (how far a number is from its right position)

Evolving a Sorting Algorithm. [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

Evolving a Sorting Algorithm. [Kinnear, 93]

- Primitives (quite low-level):
  - 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

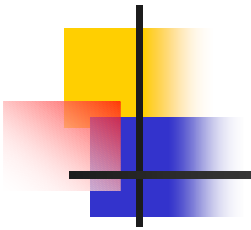
Evolving a Sorting Algorithm. [Ciesielski, Li, 04]

- Ciesielski, Li. 2004. **"Experiments with Explicit For-loops in Genetic Programming".** *CEC'04.*
  - Sorting problem:
    - It is more likely to evolve good sorting programs by using loops. They are also simpler
    - But no general sorting algorithm was evolved!

Evolving a Sorting Algorithm. [Spector, Klein, Keijzer, 05]

- Spector, Klein, Keijzer. 2005. **"The Push3 Execution Stack and the Evolution of Control"**. GECCO'2005
- Stack-based GP
- Evolved general programs for reversing a list, factorial, Fibonacci, N-even-parity, and list sorting

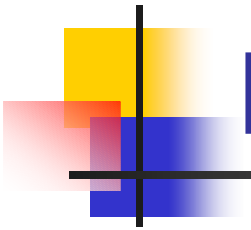
Evolving a Sorting Algorithm. [Spector, Klein, Keijzer, 05]



- Complexity: $O(N^2)$: N*(N-1)/2
- Fitness cases: lists of 4 to 8 integer elements

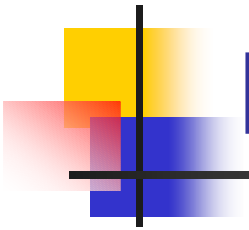
Evolving a Sorting Algorithm. [Spector, Klein, Keijzer, 05]

- Primitives (quite low-level):
  - 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])



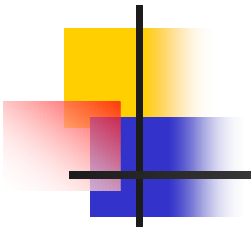
Evolving a Sorting Algorithm

- No O(N*ln(N)) algorithm has been evolved by GP (as far as I know!)



Experiments in Recursivity

- Wong. 2005. **“Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming.”**
- Yu. 2001. **“Hierachical Processing for Evolving Recursive and Modular Programs Using Higher Order Functions and Lambda Abstractions”**
- In: *Genetic Programming and Evolvable Machines*



Implicit Recursion [Yu, 01]

- No explicity recursive calls
- High level functions:
  - Map. Ex: map 3+x (1 2 3) = (4 5 6)
  - Foldr. Ex: foldr + 0 (1 2 3) = 1+2+3+0 = 6
- They are guaranteed to terminate

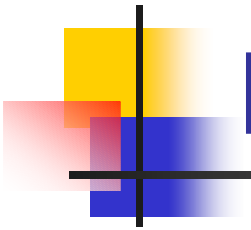
Results Implicit Recursion

Results	Implicit Recursion + λ Abstractions	Generic Genetic Programming	GP with ADFs
Programs	general even-parity	general even-parity	even-7-parity
Runs/Success	60/57	60/17	29/10
Minimum I(M,i,z)	14,000	220,000	1,440,000
Number of Fitness Cases	12	8	128
Fitness Cases Processed	168,000	1,760,000	184,320,00

Solution: $\mathrm{nor}\ (\mathrm{foldr}\ r6\ (\mathrm{head}\ \mathrm{L})(\mathrm{tail}\ \mathrm{L}))\ \mathrm{False}$

Equivalent to: (not (foldr xor (head L) (tail L)))

(xor también ha sido evolucionada)



Limits to Explicit Recursion

- Only for lists?
- It does not work for evolving Fibonacci equation f(n) = f(n-1)+f(n-2)

GP Variants

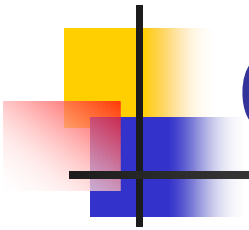
- Evolving Data Structures [Langdon, 98]
- **Machine Code Evolution (linear representation)** [keller, 96], [friedrich, 97]
- Immune Programing [Musilek, 06]:
- Stack-based GP (lineal) [Perkins, 94] [Spector et al. 2005]
- Cartesian Genetic Programming [Miller et al, 03]

Criteria for “human-competitive”

- **(A)** The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
- **(B)** The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
- **(C)** The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
- **(D)** The result is publishable in its own right as a new scientific result — *independent* of the fact that the result was mechanically created.

Criteria for “human-competitive”

- **(E)** The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
- **(F)** The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
- **(G)** The result solves a problem of indisputable difficulty in its field.
- **(H)** The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

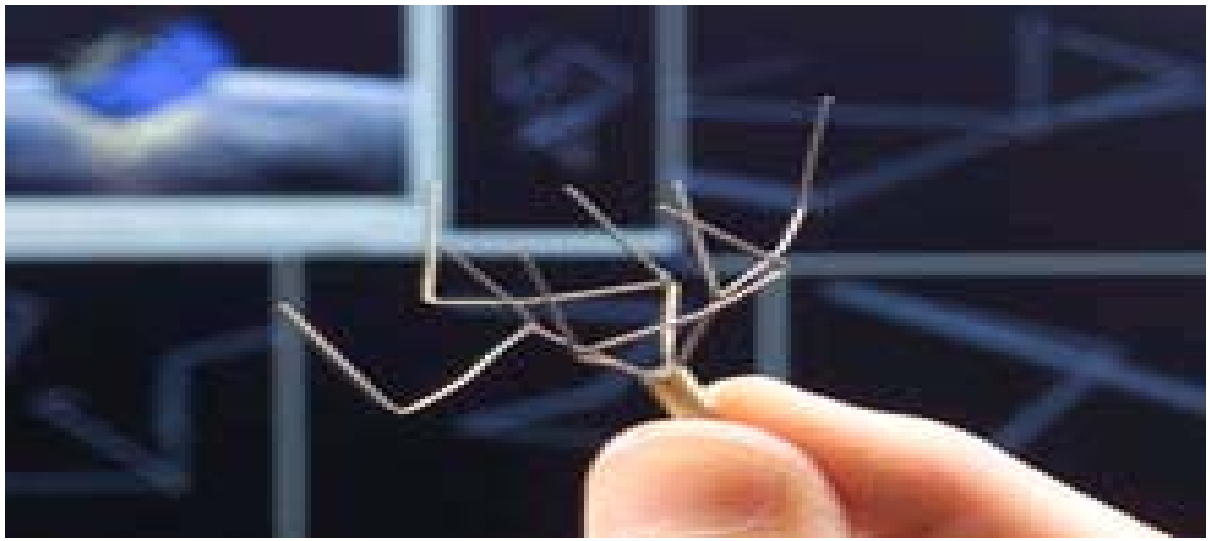


GP successes

- http://www.genetic-programming.com/humancompetitive.html
- Quantum algorithms better than existing ones
- Aplication to Robosoccer
- Aplications to Bioinformatics
- Aplication to analogical circuit design and antennae design (developmental GP)
- Paralelization of computer programs

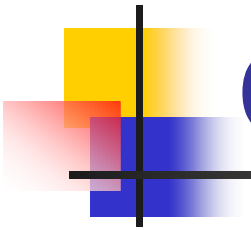
Antenna designed by develomental GP

- GP evolves programs that build the antenna
- Launched in ST5 satellite, launched in March 2006



GP in Quantum Computing

- Creation of a better-than-classical quantum algorithm for the Deutsch-Jozsa "early promise" problem (B, F)
- Creation of a better-than-classical quantum algorithm for Grover's database search problem (B, F)
- Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result (D)
- Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result (D)
- Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication (D)
- Creation of a novel variant of quantum dense coding (D)



GP in Robosoccer

- Creation of a team that won the first two matches at RoboCup 1997, [Luke 1998]
- Creation of a whole team that ranked in the middle of 34 human programmed teams at RoboCup 1998, [Andre and Teller 1999]

Results at Robocup 1998

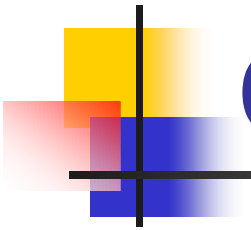
	1	2	3	4
1. PaSo Team (Italy)	-	6:2 [1] [2]	0:1 [1] [2]	5:0 [1] [2]
2. Ulm-Sparrow (Germany)	2:6	-	0:3 [1] [2]	0:1 [1] [2]
3. Miya 2 (Japan)	1:0	3:0	-	0:0 [1] [2]
4. Darwin United (USA)	0:5	1:0	0:0	-

Qualified:

1. Miya2
2. PaSo Team

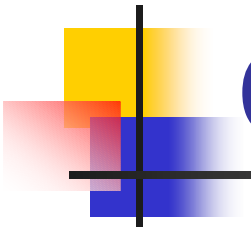
Results at Robocup 1998

- Grupo E:
  - (1) Miya2 (0-0)
  - (2) PasoTeam (0-5)
  - (3) Darwin United
  - (4) Ulm-Sparrow (1-0)
- Darwin United obtained 4 points.



GP in bioinformatics

- Creation of four different algorithms for protein transmembrane identification



Conclusions GP

- Evolution of trees with functions and memory
- Good idea: ADFs
- Not many results with loops and recursivity, so far
- Success in some real problems
- Genetic operators, too low level