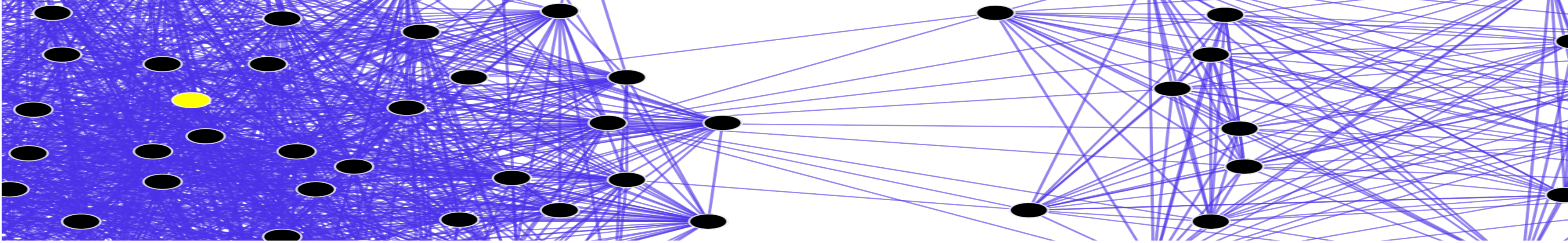


Computational Intelligence

Master in Artificial Intelligence

2021-22

René Alquézar

Introduction to Genetic Programming

Soft Computing Research Group

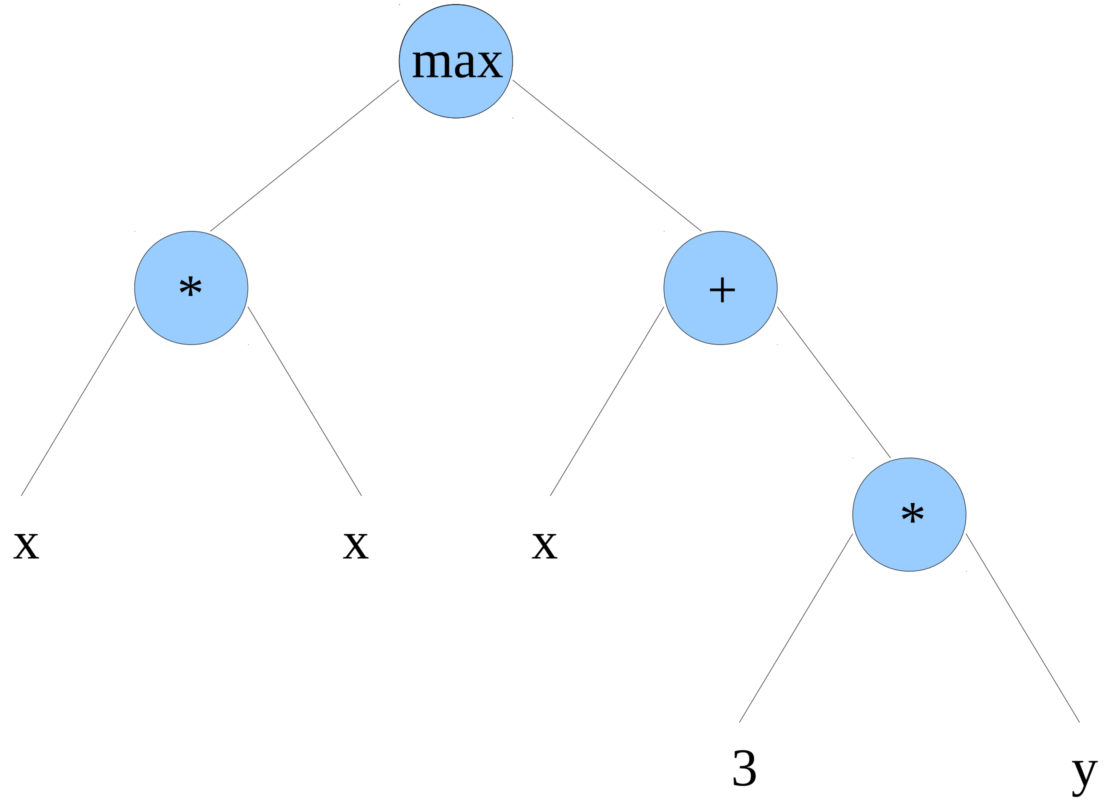
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Genetic Programming

Genetic Programming encompasses a family of techniques within Evolutionary Algorithms for which:

- Individuals are computer programs represented as syntax trees (or as a set of syntax trees)
- Strategy is usually (μ , λ) with $\mu = \lambda$, but some of the offspring may be copies (clones) of the previous individuals
- Selection is usually proportional to fitness
- Crossover, mutation and cloning are alternative genetic operators with specified probabilities to create offspring
- The best-so-far individual is chosen as the result

Syntax tree of a program



(max (* x x) (+ x (* 3 y)))

Genetic Programming

In trying to obtain a program that solves a given problem with the basic version of GP, the user needs to specify five things:

1) The **set of terminals** (independent variables, zero-argument functions and random constants) to be used as tree leaves
2) The **set of primitive functions** to be used as internal nodes (typically including a *conditional operator*, similarly to LISP)
3) The **fitness measure** to evaluate the computer programs
4) Certain **parameters for controlling the run** (population size, operator probabilities, ...)
5) The **termination criterion** (max number of generations, target value of the fitness measure)

Basic GP algorithm

t := 0

Randomly create initial population P(t) of μ programs

Evaluate P(t) using the fitness measure

WHILE NOT (termination condition) DO

t := t+1

Create P'(t) from P(t-1) by applying crossover, mutation and cloning operators with associated probabilitites to individuals selected proportionally to their fitness (up to reach μ offspring individuals in P'(t))

P(t) := P'(t)

Evaluate P(t) using the fitness measure

END

Random generation of programs

```
procedure gen_rnd_expr
  arguments: func_set, term_set, max_d, method /* either Full or Grow */
  results:  expr  /* an expression in prefix notation */
begin
  if max_d=0  or  method=Grow  and rnd_val < term_prb then
    expr := choose_rnd_element (term_set)
  else
    func := choose_rnd_element (func_set);
    for i := 1 to arity (func) do
        arg_i := gen_rnd_expr(func_set, term_set, max_d - 1, method);
    expr := (func, arg_1, arg_2, … )
  endif
end
```

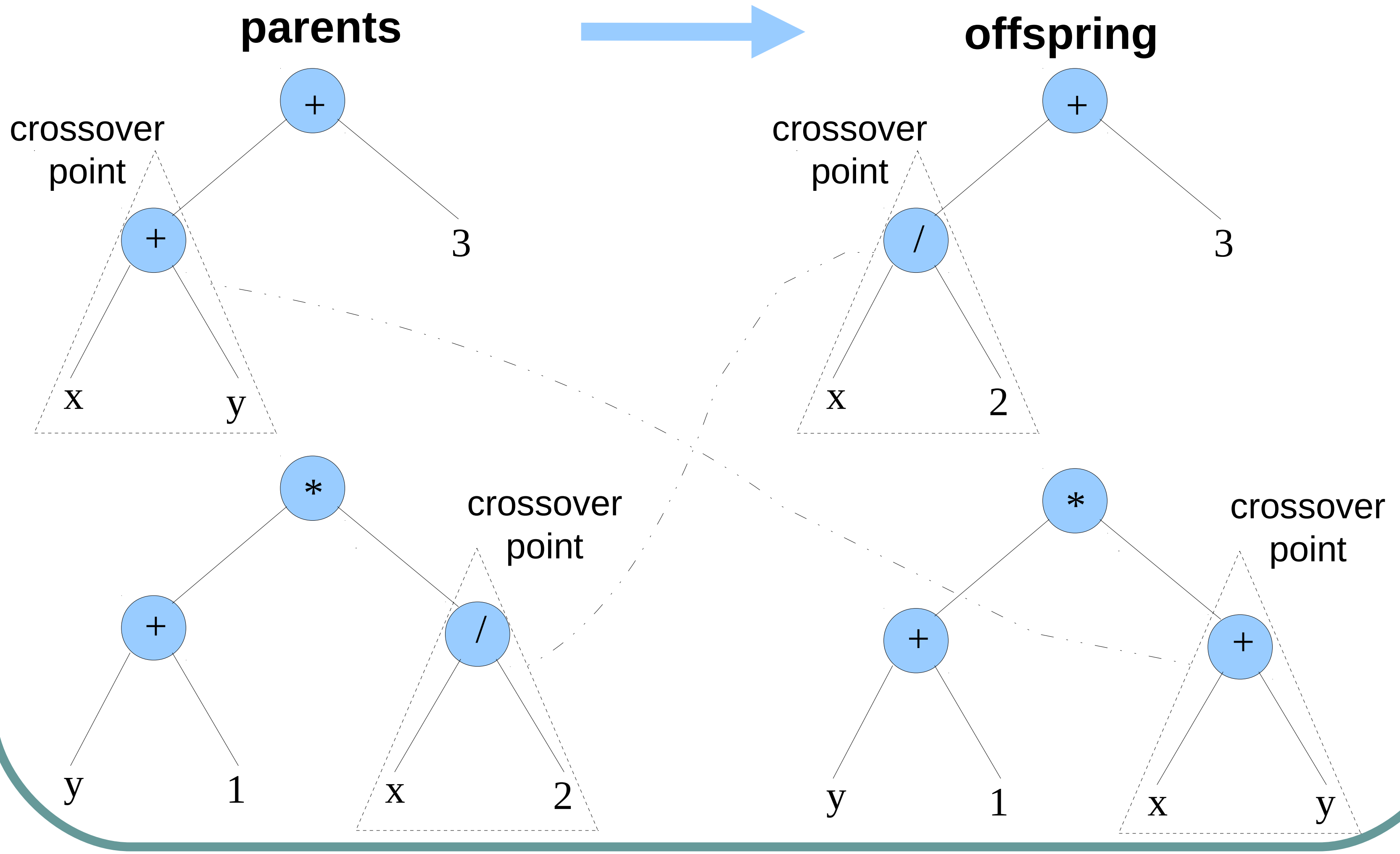
Fitness measure

The ***fitness*** of a program may be measured in many different ways depending on the problem:

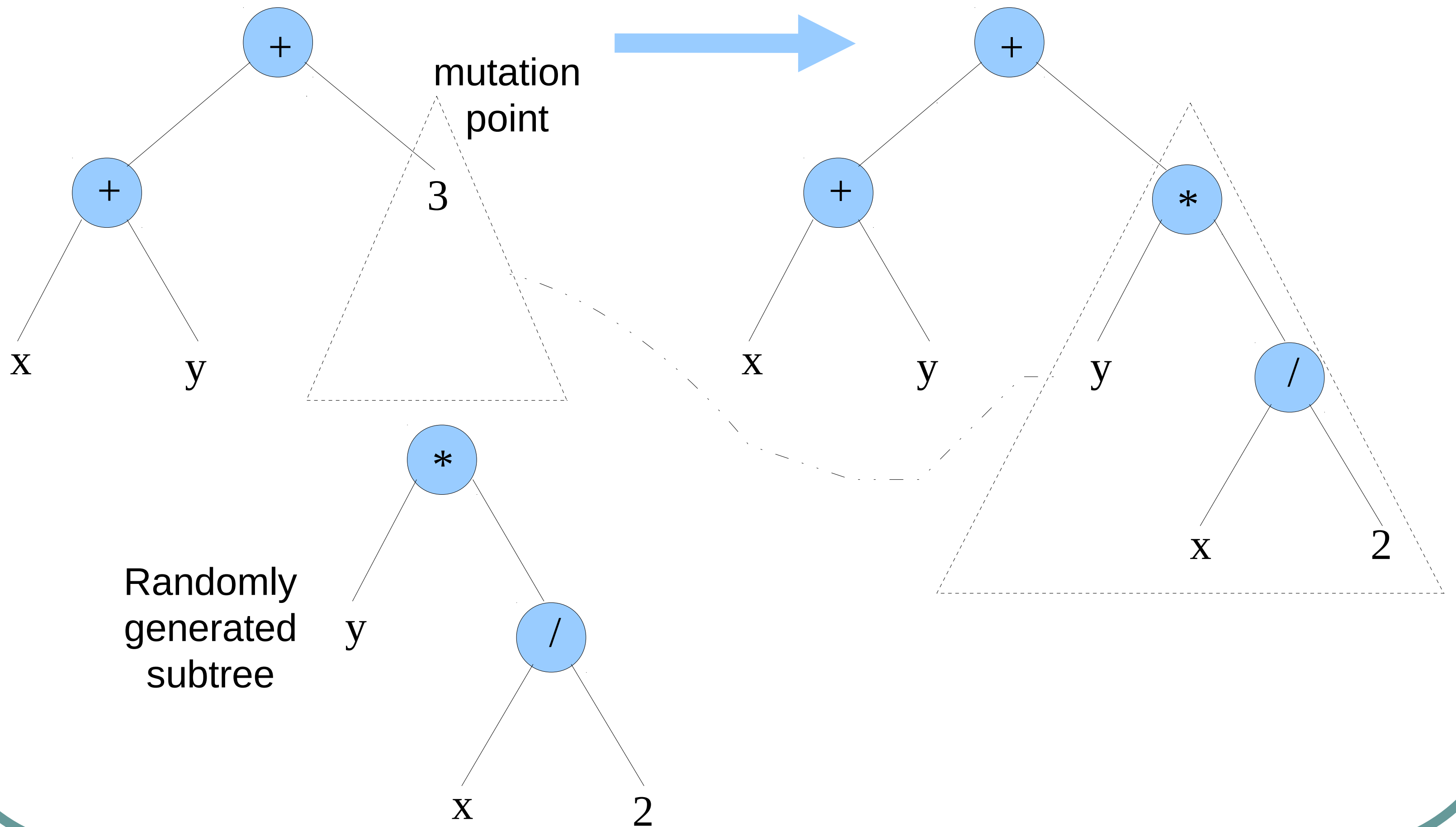
- Amount of **error** between output and desired output when run over a sample of *fitness cases* (test cases)
- **Accuracy** of classifying objects into classes
- **Payoff** that a game playing program produces

The fitness measure **may be multi-objective** in the sense that it combines two or more different elements (e.g. effectiveness and time efficiency)

Crossover


parents
offspring
crossover
point
+
+
3
x
y
*
crossover
point
+
/
y
1
x
2
crossover
point
+
/
3
x
2
*
crossover
point
+
+
y
1
x
y

Mutation



Recommendations for parameters

John Koza recommends the following GP run **parameters**:

- **Population size**: thousands or millions of individuals
- **Operator probabilities**:
  - *Crossover*: 0.9 (90% offspring)
  - *Cloning*: 0.08 (8% offspring)
  - *Mutation*: 0.01 (1% offspring)
  - *Architecture-altering operations*: 0.01 (1% offspring)

Advanced features of GP - I

Constrained syntactic structures (strong typing)

- A grammar specifies the functions or terminals that are permitted to appear as each argument of each function
- There are multiple function sets and multiple terminal sets, which are associated with types
- All the individuals in the initial random population are created so as to comply with the constrained syntactic structure
- All genetic operations are designed to produce offspring that comply with the requirements of the constrained syntactic structure (e.g. crossover points must be of the same type)

Random generation of programs with strong typing

```
procedure gen_rnd_expr_with_type
  arguments: func_set, term_set, max_d, method, res_type
  results:  expr  /* an expression of res_type in prefix notation */
begin
  if max_d=0  or  method=Grow  and rnd_val < term_prb then
    expr := choose_rnd_element (term_set, res_type)
  else
    func := choose_rnd_element (func_set, res_type);
    for i := 1 to arity (func) do
        type_i := arg_i_type (func, i);
        arg_i := gen_rnd_expr(func_set, term_set, max_d - 1, method, type_i);
    expr := (func, arg_1, arg_2, … )
  endif
end
```

Advanced features of GP - II

Automatically Defined Functions (ADFs)

- ADFs allow GP to implement the parameterized reuse and hierarchical invocation of evolved code
- Each ADF resides in a separate function-defining branch within the overall program architecture
- An ADF may possess zero, one, or more dummy variables (formal parameters) and may be called by the program's main result-producing branch, another ADFs or other branches
- A constrained syntactic structure is used to implement ADFs
- ADFs are the focus of *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994, MIT Press)

Advanced features of GP - III

Other types of branches, in addition to ADFs, to reuse code:

- Automatically Defined Iterations (ADIs)
- Automatically Defined Loops (ADLs)
- Automatically Defined Recursions (ADRs)
- Automatically Defined Stores (ADSs) provide means to reuse the result of executing code
- ADIs, ADLs, ADRs and ADSs are described in *Genetic Programming III: Darwinian Invention and Problem Solving* (Koza, Bennett, Andre, and Keane 1999, Morgan Kaufmann)

Advanced features of GP - IV

Program Architecture and Architecture-Altering Operations

- Each program is actually represented as a set of trees called **branches** instead of always being a single tree
- The **architecture** of a program consists of
  - the total number of branches
  - the type of each branch (result-producing branch, ADF, ADI, ADL, ADR or ADS)
  - the number of arguments (if any) possessed by each branch
  - if there is more than one branch, the nature of the hierarchical references (if any) allowed among the branches

Advanced features of GP - V

Program Architecture and Architecture-Altering Operations

- The **architecture** of a program may be
  - prespecified by a human user (this means to perform an additional architecture-defining preparatory step) or
  - automatically and dynamically created during a GP run by means of architecture-altering operations
- The **architecture-altering operations** include the *creation*, *duplication* and *deletion* of branches (Koza, Bennett, Andre, and Keane 1999, Morgan Kaufmann)

Some areas where GP has produced human-competitive results

- Creation of **quantum algorithms** for some problems (e.g. Grover's database search problem)
- Creation of **robot soccer-playing** programs for Robo Cup
- Rediscovery of some DSP (Digital Signal Processing) **filters**
- Synthesis of electric and electronical **circuits**
- Synthesis of PID (proportional, integrative, and derivative) and non-PID **controllers** for industrial control systems