POLITECNICO DI MILANO

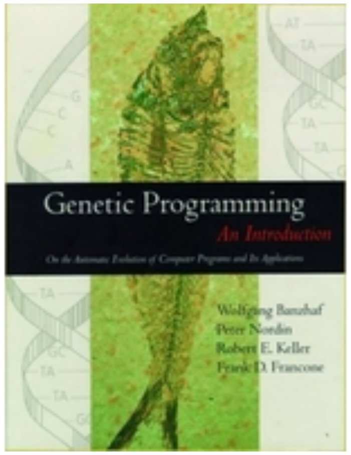
Genetic Programming

Genetic Algorithms and Other Evolutionary Approaches

Prof. Pier Luca Lanzi

References

- Wolfgang Banzhaf et al. Eds. Genetic Programming. An Introduction.: An Introduction (The Morgan Kaufmann Series in Artificial Intelligence)



- CS 5320 “Introduction to Evolutionary Computation” Prof. Martin Pelikan, University of Missouri, St. Louis, MO

Outline

- Introduction
- Representation
- Examples
- Generation
- Variation
- Summary

Motivation of John Koza

Challenge

How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?

Arthur Samuel (1959)

Criterion of success

The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.

Arthur Samuel (1983)

Early intuitions

Alan Mathison Turing, English mathematician, logician, and cryptographer. Often considered to be the father of modern computer science.

"There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value."
Alan M. Turing, Intelligent Machinery, 1948

Early intuitions

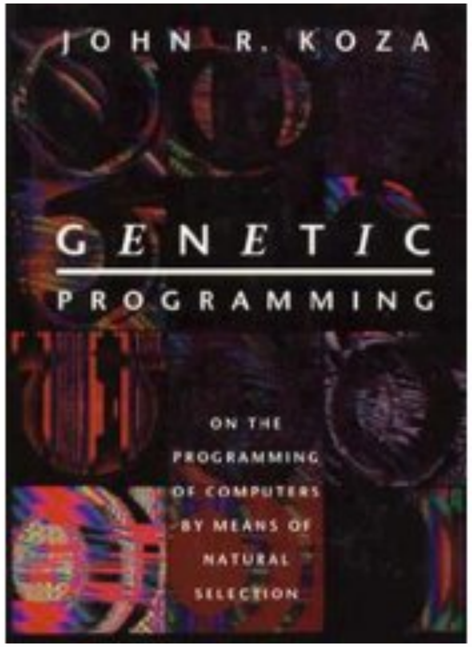
“We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications

“Structure of the child machine = Hereditary material
“Changes of the child machine = Mutations
“Natural selection = Judgment of the experimenter”

Alan M. Turing, “Computing Machinery and Intelligence” 1950.

Genetic Programming

- John R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press


JOHN R. KOZA
GENETIC
PROGRAMMING
ON THE PROGRAMMING OF COMPUTERS BY MEANS OF NATURAL SELECTION

John R. Koza

Genetic Programming

Make the computer program itself

Input

- Syntactic rules for your programs
- Semantics of solutions
- Fitness function

Output

- Best performing program code

Big Questions & Basic Approach

- How to represent syntactic rules?
- How to represent programs?
- How to deal with the semantics?
- How to define the fitness function?
- How to get the best program?

- Basic Approach
  - Modify genetic algorithm to work with program codes.
  - Represent program codes as labeled trees.
  - Implement fitness function to evaluate program codes.
  - Implement crossover + mutation for program trees.

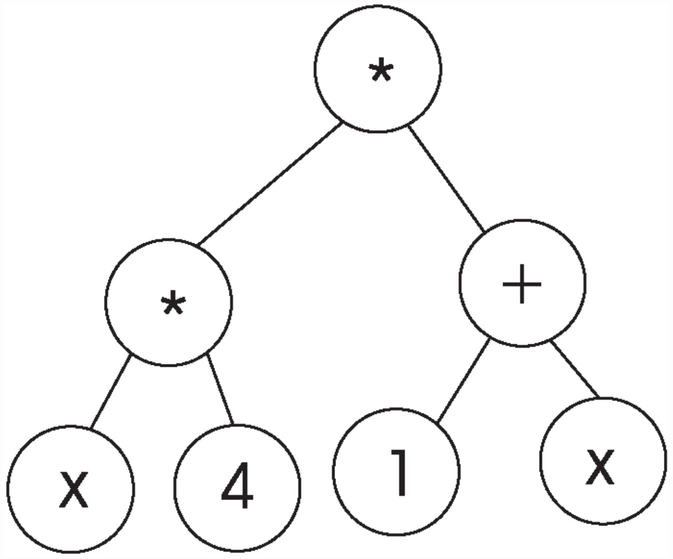
Syntactic Rules

- Two parts
  - Terminals: constants, variables, functions with no arguments
  - Functions: functions with one or more arguments

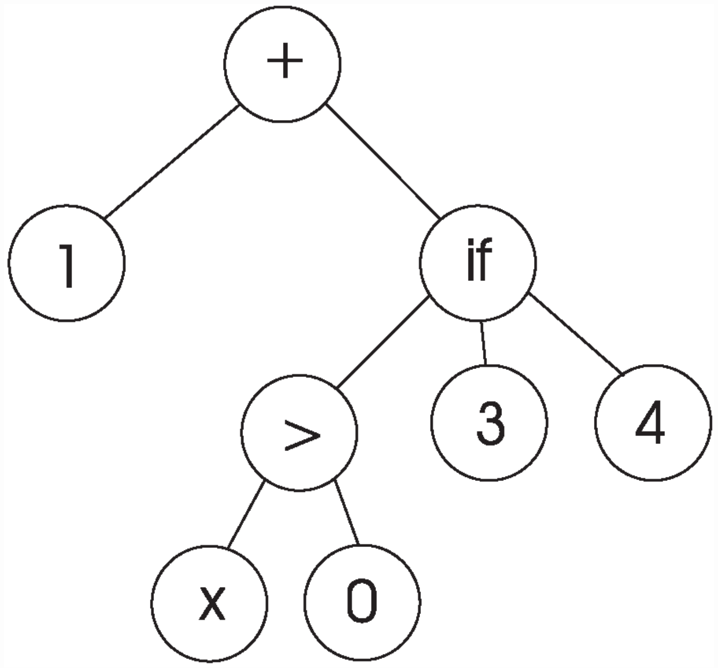
- Valid programs
  - Only the specified terminals and functions are used.
  - All functions have necessary parameters.

Example: Arithmetic Expressions

- Terminals: Real-valued constants, variables.
- Functions with one parameter: sin, cos, tan, log, sqrt.
- Functions with two parameters: +, -, *, /,ˆ .



(x*4)*(1+x)

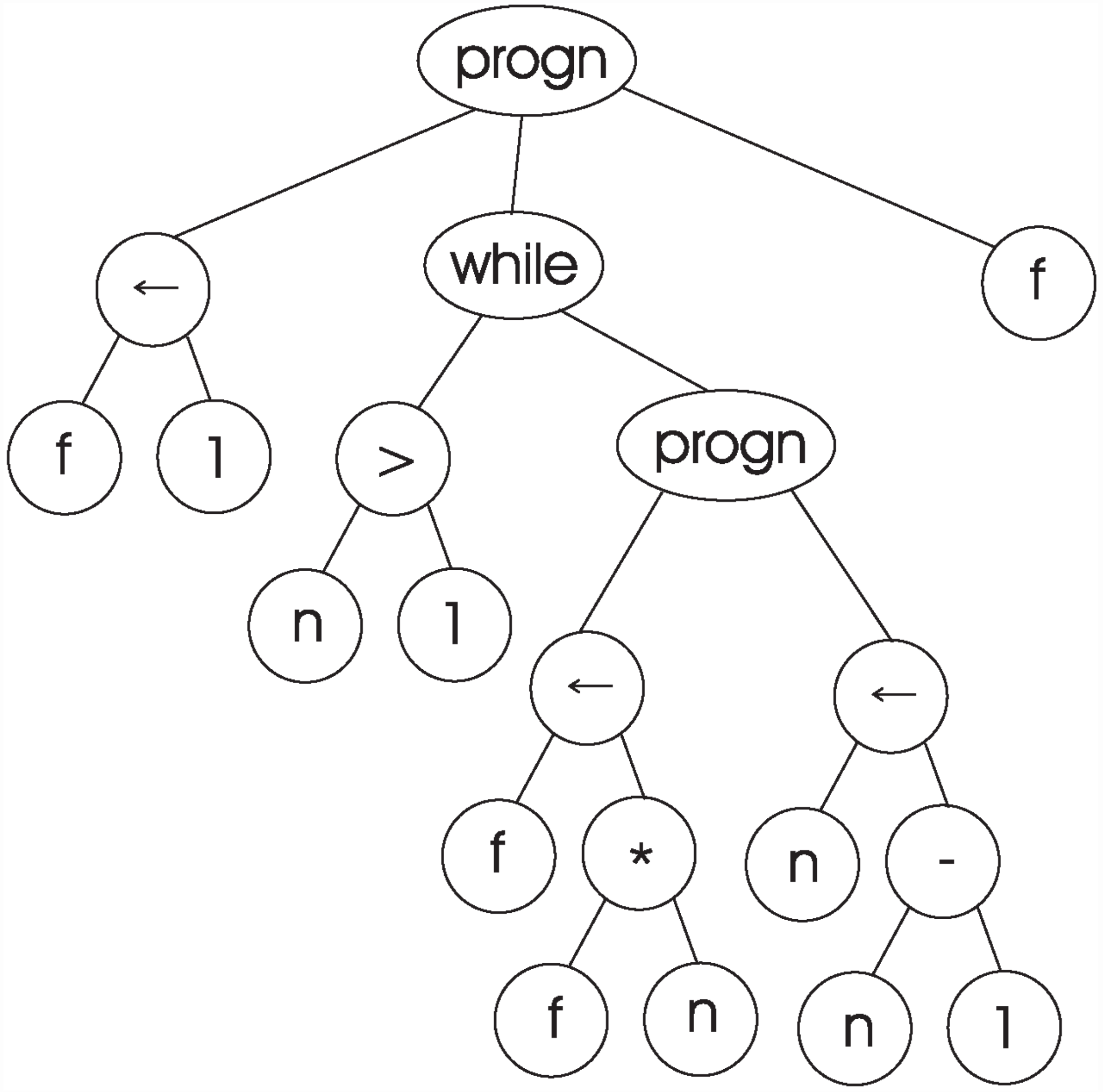


1 + ((x > 0)? 3 : 4)

Example: Programs

- Terminals
  - Same as for arithmetic expressions.
- More Functions
  - $\leftarrow$ assignment of a variable.
  - while(c,b): while loops with condition c and body b
  - if(c,a1,a2): if-then-else statements with condition c, then-statement a1, and else-statement a2
  - $>$, $\leq$, $\geq$, $<$, $=$: compare real numbers
  - progn(a_1, . . . , a_n): execute a sequence of statements from a_1 to a_n,
  - returning the output value of the last statement (a_n).

Example: Factorial n!



Semantics

- Functions
  - Define how each function processes arguments.
  - Each function returns a value.
  - Return value and input parameters should work with all types.
  - Functions should be foolproof (work properly for any parameter values). Example: For division x/y, we would return some value even if y = 0.
- Terminals
  - Define a function that initializes a terminal.
  - Either randomly generate (e.g. numbers).
  - Or set to some fixed value (e.g. π).

Fitness Function

- Fitness function
  - Depends on the problem...
- Examples
  - Symbolic regression.
  - Classification.
  - Game playing.
  - Decision making in agents.

Symbolic Regression

- Input
  - Number of parameters: n.
  - Database of N input-output pairs $(x_{i,1}, \ldots, x_{i,n}) \rightarrow y_i$
  - $i \in \{1, 2, \ldots, N\}$
  - $x_{i,j}$ are real numbers
  - y_i are real numbers
- Task
  - Find the function that represents the input-output pairs most accurately.

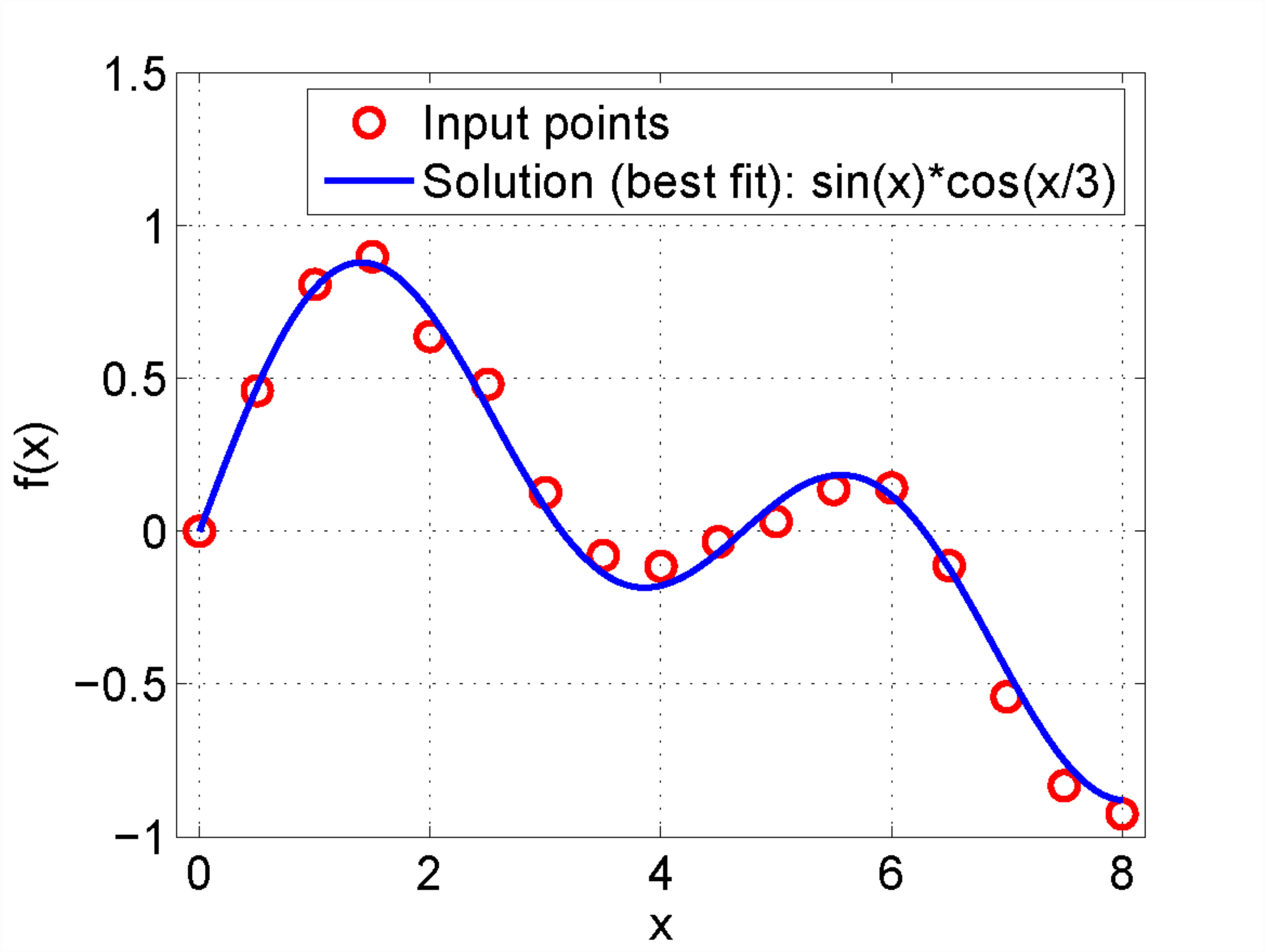
Symbolic Regression

- Representation
  - Use standard functions for arithmetic expressions or a subset of them.
  - Allow terminals for real numbers.
  - Allow terminals for n input parameters: x_1, x_2, ..., x_n.
  - All functions return a real value.

- Fitness function (one possibility)
  - Mi

$$\text{fitness}(f) = \sum_{i=1}^{N} [f(x_{i,1}, x_{i,2}, \ldots, x_{i,n}) - y_i]^2$$

Symbolic Regression



n = 1, N = 17, fitness 0.057 (sum of square differences)

Classification

- Input
  - Similar to symbolic regression
  - Number of parameters: n
  - Number of classes: c
  - Database of N input-output pairs $(x_{i,1}, ..., x_{i,n}) \rightarrow y_i$
  - $i \in \{1, 2, . . . ,N\}$
  - $x_{i,j}$ are parameters of any type
  - y_i are integers from 1 to c
- Task
  - Create a function that correctly classifies as many input-output pairs as possible.

Classification

- Representation
  - Functions allow tests on variables of different types.
  - Arithmetic expressions allowed as well.
  - Terminals contain n input parameters: $x_1, x_2, \ldots, x_n$.
  - Terminals contain c values for different classes: $cl_1, cl_2, \ldots, cl_c$
  - Return value would ideally be a class (if it works).

- Fitness function
  - Maximize the number of correctly classified input pairs (values from 0 to N, N being the best).

Example: Restaurant Selection

Quality (x_1)	Prices (x_2)	Service (x_3)	Classification (y)
High	Average	Good	$+$
Low	High	Average	$-$
Low	Low	Good	$-$
Average	Low	Average	$+$
High	High	Poor	$-$
Average	Average	Good	$+$

- Solution
  - Fitness is 6 (number of correct classifications)

```
if ((x1=average) or (x1=high))
   then if ((x3=average) or (x3=good))
           then +
           else -
   else -
```

Game Playing

- Input
  - Current percepts.
- Example
  - Chess—positions of the pieces of you and your opponent.
  - Poker—cards on your hand, number of cards exchanged, bets.
- Task
  - Decide on the next move (action).

Game Playing

- Representation
  - Depends on the problem, but similar to classification.
- Fitness function
  - Let each candidate solution play against other solution in the populations.
  - Fitness will be the number of games won minus the number of games lost (ties do not count).
  - We want to maximize the fitness.

Coevolution

- No explicit fitness function that gives a solution its fitness.
- Solutions are somehow compared with each other and the results are somehow processed to produce fitness.
- Useful in game playing.
- Useful also in simulations of adaptive and complex systems.

Example: Santa Fe Ant Trial

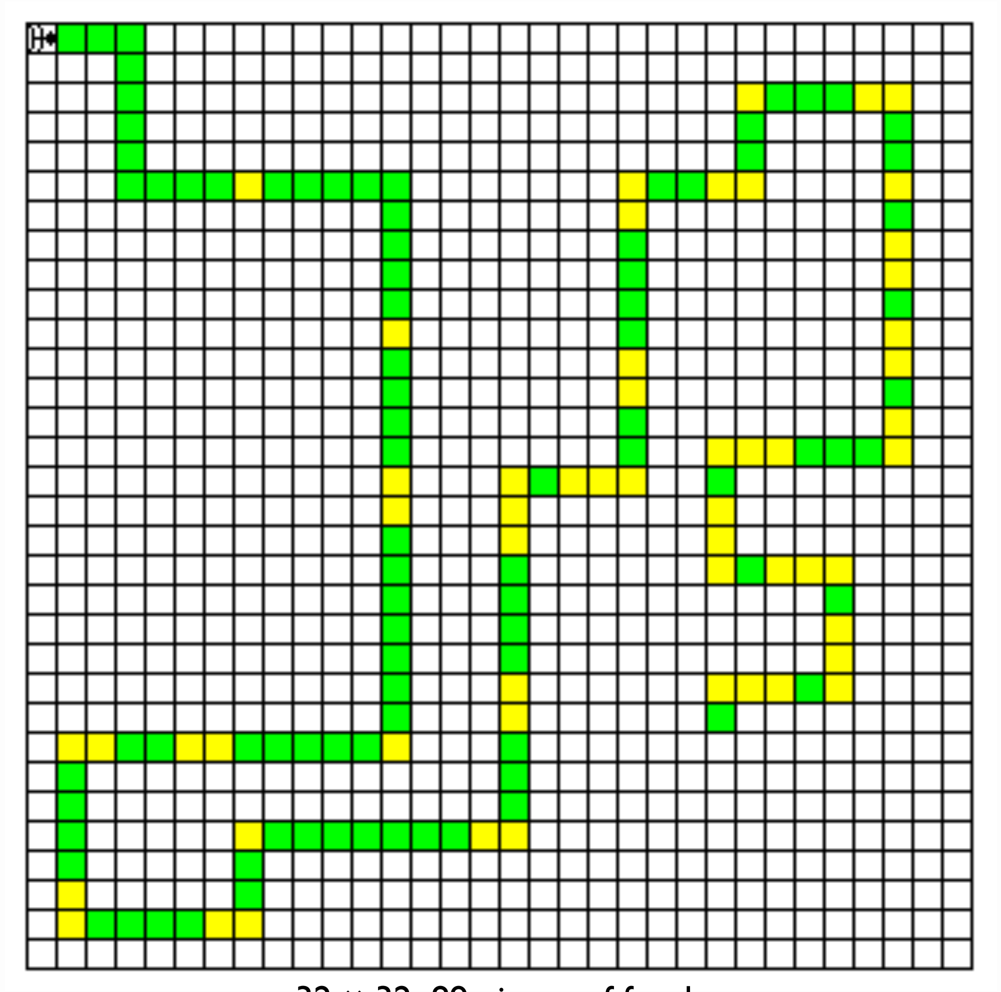
- Goal
  - Create a "brain" of an agent whose task is to collect food distributed on a 2D map.
- Input
  - 2D grid map (can move from square to square).
  - Some squares contain food.
  - Some squares are empty.
- Task
  - Create an agent that can discover the food but must decide only based on the information about the adjacent square.

Example: Santa Fe Ant Trial

- Representation for Santa Fe ant trail
  - move: move ant one step forward
  - left: turn left
  - right: turn right
  - if-food-ahead(a1, a2): if food lies in front of the ant perform a1, otherwise perform a2
  - progn2(a1, a2): first perform a1, then perform a2

- Fitness for Santa Fe ant
  - Let the ant proceed for sufficiently many steps to pick up all the food (+ some extra).
  - Fitness is the number of food pieces collected along the way.
  - Smarter ants find more food and win.

Example: Santa Fe Ant Trial



32 × 32, 89 pieces of food
Green squares denote food, the rest is empty.

```
(Progn2
    (Progn2
        (if-food-ahead
            (Progn2
                (Progn2
                    (if-food-ahead
                        (move)
                        (right))
                    (move))
                (if-food-ahead
                    (if-food-ahead
                        (move)
                        (move))
                    (if-food-ahead
                        (move)
                        (move))))
            (if-food-ahead
                (right)
                (right)))
        ...
```

What Now?

- What we know
  - How to design syntactic rules.
  - How to deal with the semantics.
  - How to design fitness functions (examples).
- Remaining components of GP
  - How to generate random programs trees?
  - How to perform crossover of two programs trees?
  - How to perform mutation of a program tree?

Generation

- Approach 1: Grow a tree
  - Start in the root.
  - Randomly generate a function or terminal.
  - If we generated terminal, terminate generation.
  - If we generated function, recursively generate its arguments.
  - Trees will vary in depth and structure.
- Approach 2: Generate full tree
  - Choose desired depth for terminal nodes.
  - Generate randomly like before.
  - But make sure all terminals are at the same depth.
  - Any traversal from root to leaf has the same length.
  - Structure still not fixed, but much more regular.

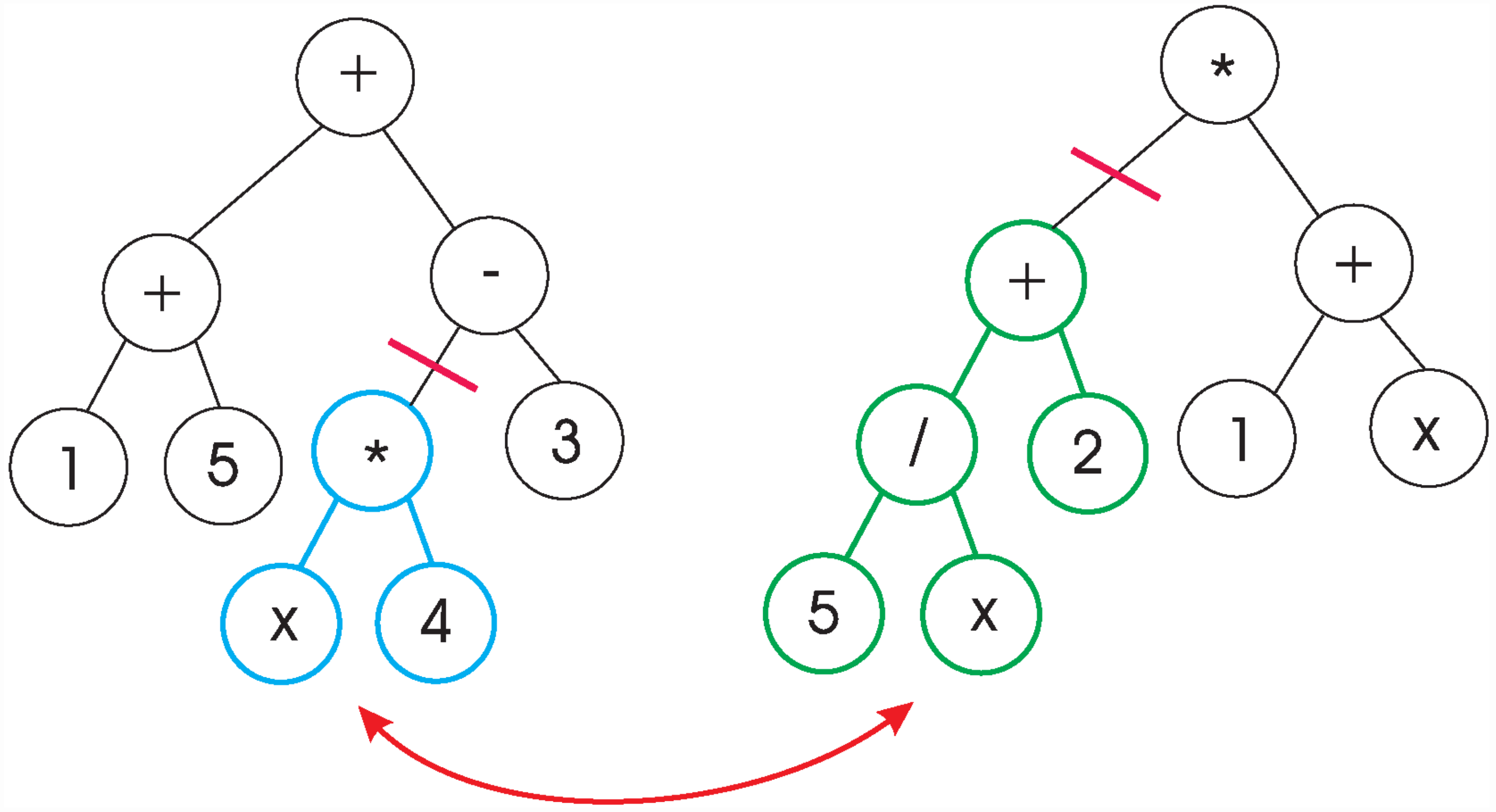
Generation: Ramped Half-And-Half

- Ramped half-and-half
  - Generate half of the population by grow method.
  - Set a range of depths to use: d_{min} to d_{max}
  - Split the other half of the population into equal parts, one for each depth between d_{min} and d_{max}
  - Each of the parts uses full generation with the corresponding depth
  - Creates a range of individuals with fixed and variable structure, and different overall depths
  - Koza and lots of others use this method

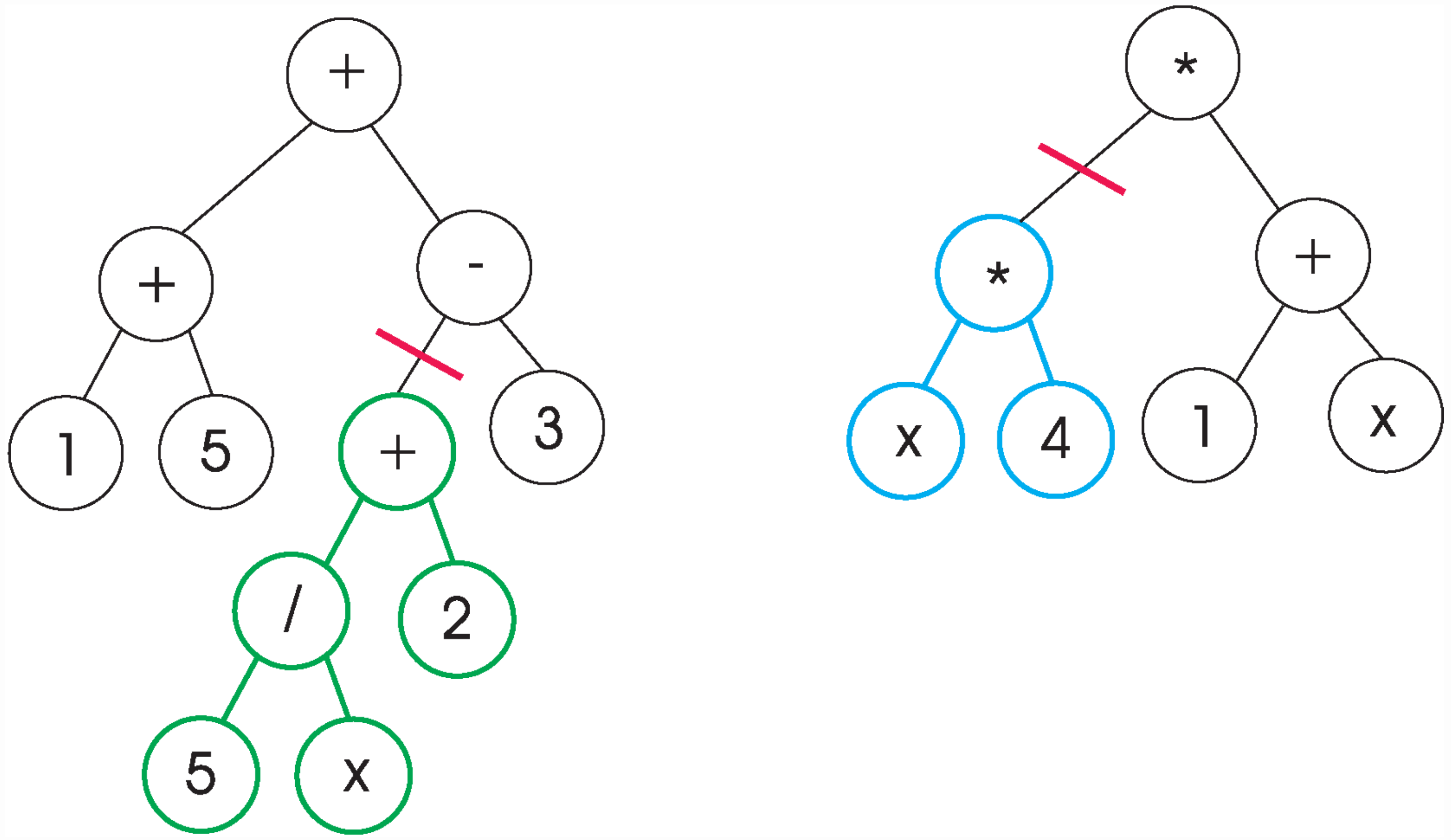
Crossover

- Two steps
  - Randomly select one subtree in each of the two trees.
  - Swap the selected subtrees.

Crossover
Step 1: Select subtrees to swap



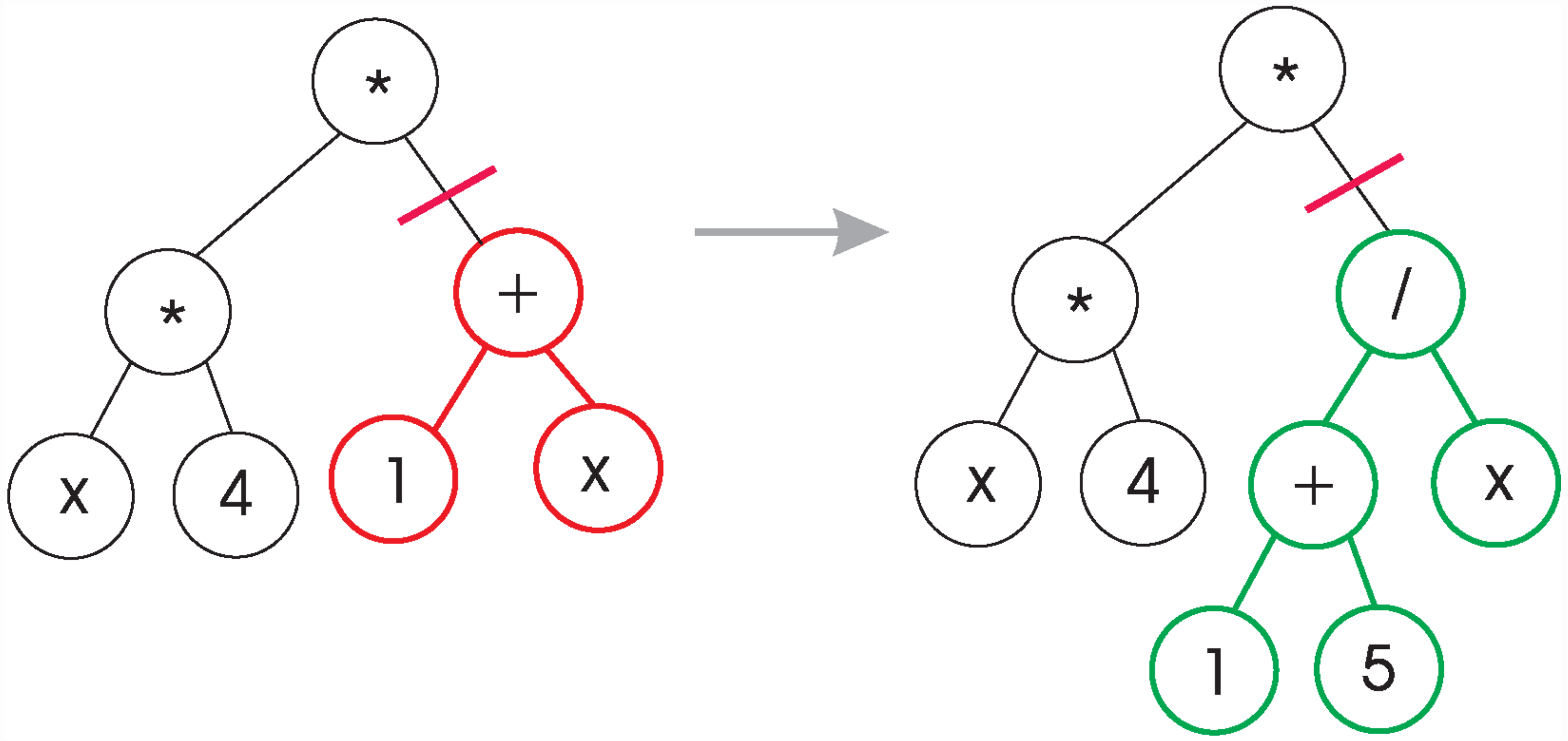
Crossover
Step 2: Swap subtrees



Mutation

- Two steps
  - Randomly select a subtree.
  - Replace the subtree by a randomly generated tree.

Mutation: Example

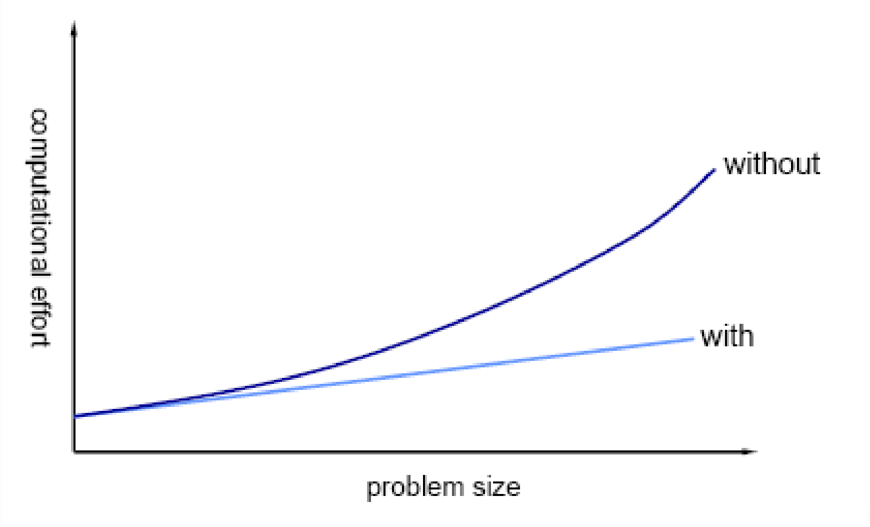


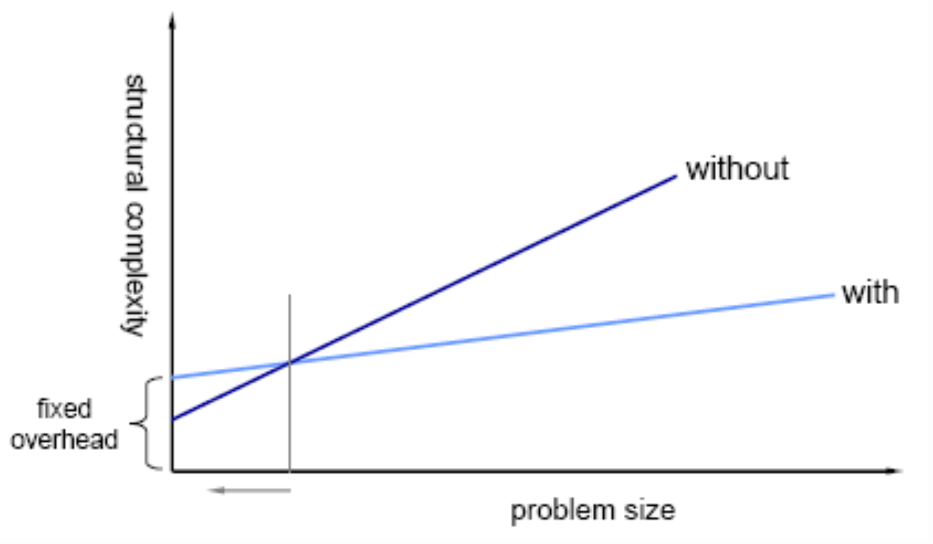
Automatic Function Definition

- Problems have structure.
- Similar computations repeated in a program.
- Discover general functions; use multiple times.
- Efficiency: discover once, use often.
- Depends on problem hierarchy and iterated structure: decomposition (again).

Benefits of Automatic Function Definition

- Lower structural complexity
- Lower computational costs
- Better scalability w.r.t. problem size
- Acceleration of learning





Bloat

- Bloat = “survival of the fattest”
- The tree sizes in the population are increasing over time
- Ongoing research and debate about the reasons
- Countermeasures
  - Prohibiting variation operators
    that would deliver “too big” offspring
  - Parsimony pressure: penalty for being oversized

Questions

- How much sense does it make to...
  - ...take a subtree out of it context?
  - ...exchange subtrees between two trees?
  - ...replace a subtree with a randomly generated one?
- Answer...
  - The above questions are very difficult to answer despite that it seems to be easy at the first sight.
  - Depends on the problem too.
  - GP proved to work very well in a number of applications.
  - GP provided several important results, including patentable inventions!
  - But most of these applications used a LOT of computational power.

Human-Competitive Result

Result of comparable or better quality than was achieved by a human.

Automatic programming procedure beats the human expert.

Human-competitive results make GP an automatic invention machine.

What criteria to use for human competitiveness?

Criteria for Human-Competitive Results

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 Results

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)

Human-Competitive Results

- 36 human-competitive results
- Source: http://www.genetic-programming.com/
- There is an annual competition at Genetic and Evolutionary Computation Conference for human-competitive results in GP.

Human-Competitive Results

- Synthesis of a tunable integrated active filter
  A
  Section 15.4.6 of Genetic Programming IV

- Creation of PID tuning rules that outperform the Ziegler-Nichols and strm-Hgglund tuning rules
  A, B, D, E, F, G
  Chapter 12 of Genetic Programming IV

- Creation of three non-PID controllers that outperform a PID controller using the Ziegler-Nichols or strm-Hgglund tuning rules
  A, B, D, E, F, G
  Chapter 13 of Genetic Programming IV

Human-Competitive Results

- Synthesis of a high-current load circuit
  A
  Section 15.4.3 of Genetic Programming IV
- Synthesis of a voltage-current conversion circuit
  A
  Section 15.4.4 of Genetic Programming IV
- Synthesis of a cubic function generator
  A
  Section 15.4.5 of Genetic Programming IV

Human-Competitive Results

- Rediscovery of negative feedback
  A, E, F, G
  Chapter 14 of Genetic Programming IV
- Synthesis of a low-voltage balun circuit
  A
  Section 15.4.1 of Genetic Programming IV
- Synthesis of a mixed analog-digital variable capacitor circuit
  A
  Section 15.4.2 of Genetic Programming IV

Human-Competitive Results

- Synthesis of a NAND circuit
  A, F
  Section 4.4 of Genetic Programming IV

- Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits
  A. F, G
  Chapter 5 of Genetic Programming IV

- Synthesis of topology for a PID (proportional, integrative, and derivative) controller
  A, F
  Section 9.2 of Genetic Programming IV

Summary

- GP is a GA with labeled trees instead of binary strings.
- Labeled trees can represent programs, arithmetic expressions, classifiers, and all kinds of other things.
- Must define:
  - Syntax.
  - Semantics (relates to syntax).
  - Fitness function (relates to syntax and semantics).
- What differs from GAs?
  - Initialization (random generation).
  - Crossover.
  - Mutation.
- GP enables computers program themselves. But it's not so simple.