Evolutionary Computing

Genetic Programming

Dr. Petr Musilek

Department of Electrical and Computer Engineering
University of Alberta

Fall 2019

Automatic Programming

- Automatic generation of computer programs
- Saying **WHAT** is wanted but not **HOW** to do it
- The programs can be of the most GENERAL form:
  - Subroutines (e.g. sorting, search)
  - Planning (e.g. sequence of moves of robotic arm)
  - Strategy in games (e.g. Pacman, Quake, Civilization ...)
  - Classification (e.g. character recognition)
  - Prediction (e.g. consumption of electricity)
  - Control (e.g. vehicle steering ...)
  - Design of electronic circuits (e.g. chip layout)

Genetic Programming (GP)

- GP is a branch of EC with a goal to generate a computer program (or just function/expression) that best fits user's requirements.
- The computer program acts as a candidate solution (like a binary string in GA).
- GP is computationally expensive and its application area was limited in 1990s.
- Recently, GP has started delivering human-competitive machine intelligence thanks to exponential growth in CPU power.

Representation of programs

- Programming languages: C, Java, Prolog, machine language, LISP
- Special languages: robots, Turing machines
- Typically, data and programs are treated the same way (lists or S-expressions):
  - `(dotimes i 3 (setq v (* i i)))`
  - `(3 4 5 (to b c))`
  - **Language = functions + terminals**

Example: Even parity

- Generate a computer program that takes 10 bits and returns whether the number of 1's is even:
  - `Even-Parity(1,0,0,0,1,0,0,1,1,0)` $\Rightarrow$ `TRUE`
- In terms of:
  - Functions: `AND, OR, NAND, NOR, NOT`
  - Terminals: `B0, B1, B2..., B9`
- Large number of input/output test cases ($2^{10} = 1024$ cases)

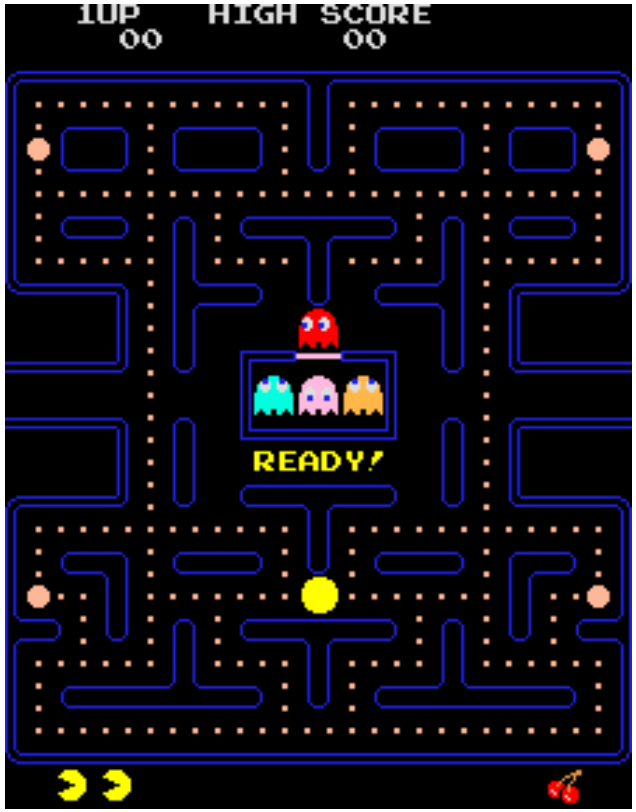
Test cases for 10-bit even parity (1024 cases)

B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	OUT
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
...										
1	1	1	1	1	1	1	1	1	1	TRUE

Example: Strategy for Pacman game

- Functions:
  - `if-obstacle`, `if-pill`, `if-power-pill`, `if-ghost` (they are of the form if-then-else)
  - `sequence2, 3, 4` ...
- Terminals: `advance`, `turn-left, -right`
  - `advance, turn-left, -right`
- **GOAL**: to eat all pills within a time limit

...

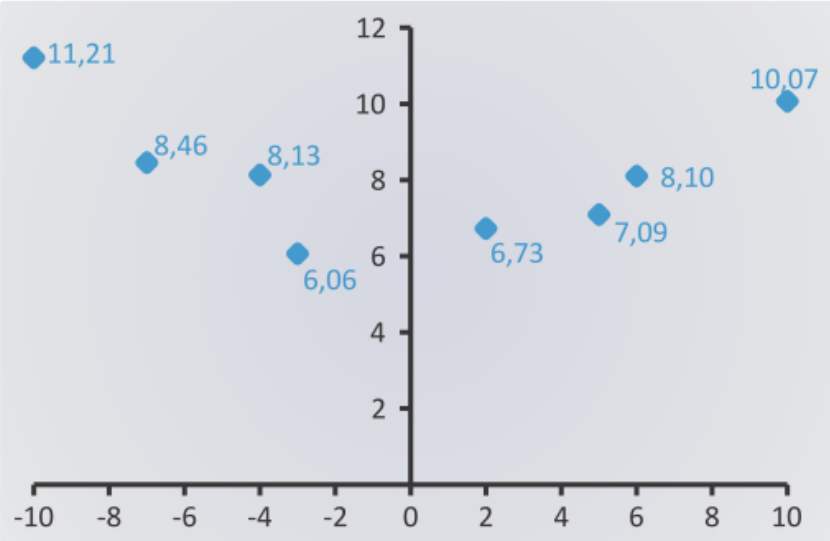


Example of program in Pacman game

```
(if-ghost
    (sequence3 (turn-left)
               (turn-left)
               (advance))
(if-power-pill
    (advance)
    (turn-right)))
```

Symbolic Regression

- Given: Set $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of points
- Wanted: functional relation φ: $\mathbb{R} \mapsto \mathbb{R}$



x	y
−10	11.20975399
−7	8.455665097
−4	8.131449992
−3	6.064537515
2	6.725218415
5	7.091508928
6	8.104694856
10	10.06724264

Symbolic Regression - Alternatives

- Approximations: linear and non-linear Regression
- Parameterization of a given formula blueprint, e.g.:

$$\boldsymbol{y} = \varphi(\boldsymbol{x}) = a_4\boldsymbol{x}^4 + a_3\boldsymbol{x}^3 + a_2\boldsymbol{x}^2 + a_1\boldsymbol{x} + a_0$$

- Method of least squares developed by Gauß in the 1790s at the age of 18 and first published by Legendre:

$$\text{Find values for } a_0 \text{ to } a_4 \text{ that minimize } \sum_{i=1}^{n}\left(y_i - \varphi(x_i)\right)^2$$

- Often, we can do this efficiently with the Levenberg-Marquardt algorithm, the standard for non-linear regression
- If that does not work well, we can combine these methods with DE or CMA-ES ...
- But: Assumption about formula blueprint needed ...

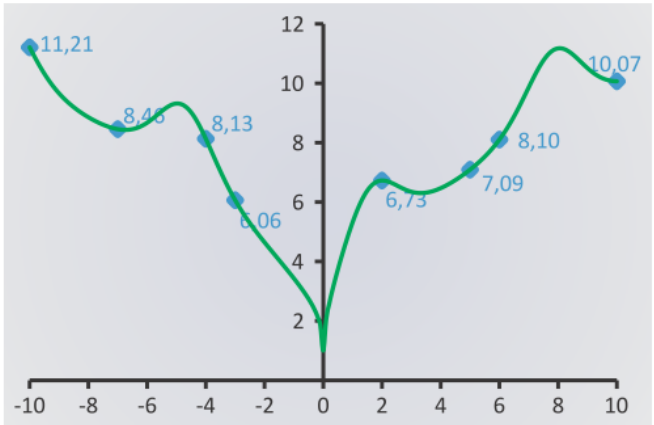
Symbolic Regression - Idea

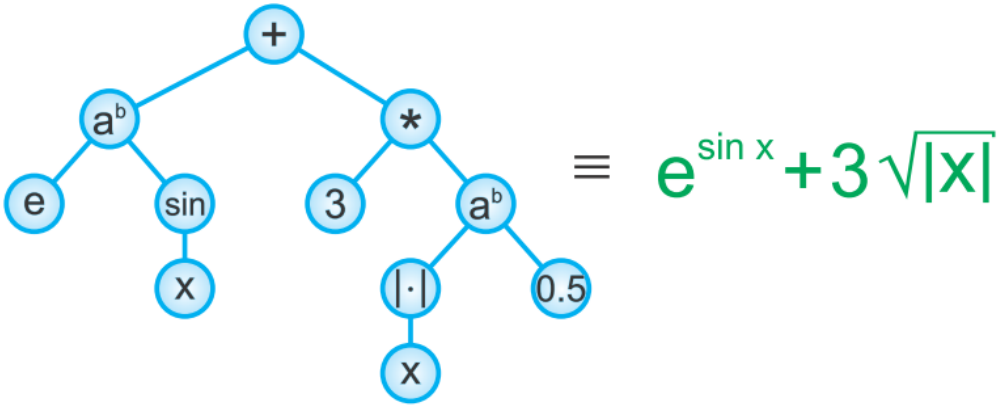
- New kind of optimization problem: Symbolic Regression
- We have:
  - Given set of points $S = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), \ldots, (\boldsymbol{x}_n, \boldsymbol{y}_n)\}$
  - A Function Set with elementary functions which can be combined arbitrarily: $\mathcal{F} = \{+, -, *, \sin, \sqrt{}, \exp, \ldots\}$
  - A Terminal Set, i.e., the available null-ary functions containing things like the input variable $\boldsymbol{x}$ and real constants: $\mathcal{T} = \{\boldsymbol{x}, \Re, ...\}$
- We want: Find a combination of elements from $\mathcal{F}$ and $\mathcal{T}$ which represents a formula that fits to the data.

Symbolic Regression - Representation

Symbolic Regression with Genetic Programming:

- represent formulas as tree data structures
- for example: minimize $f(\varphi) = \sum_{i=1}^{n} \left(y_i - \varphi(x_i)\right)^2$
- construct φ with Genetic Programming!

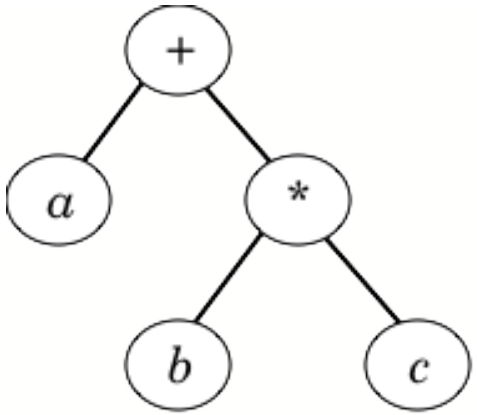




Program Representation

The program is generally represented by a parse tree of the function set and terminal set rather than lines of code.

E.g., a simple expression $a + b \cdot c$ would be represented as:



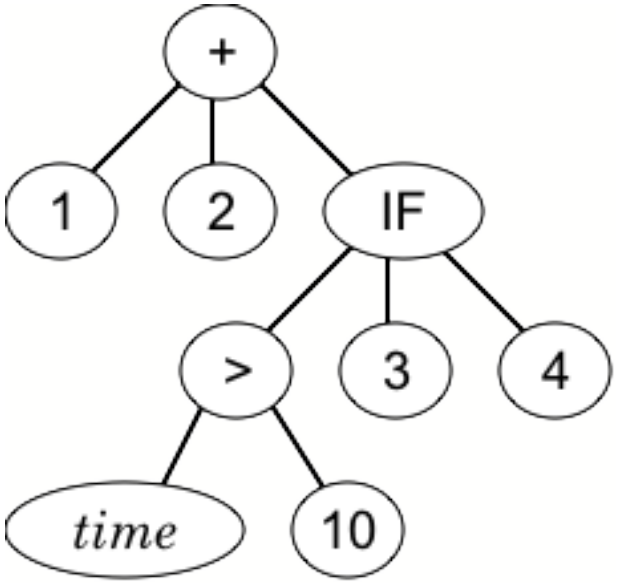
Program Representation

As another example, consider the following simple function in C language:

```
int foo (int time)
{
    int a, b;
    if(time > 10)
        a = 3;
    else
        a = 4;
    b = a + 1 + 2;
    return b;
}
```

Program Representation

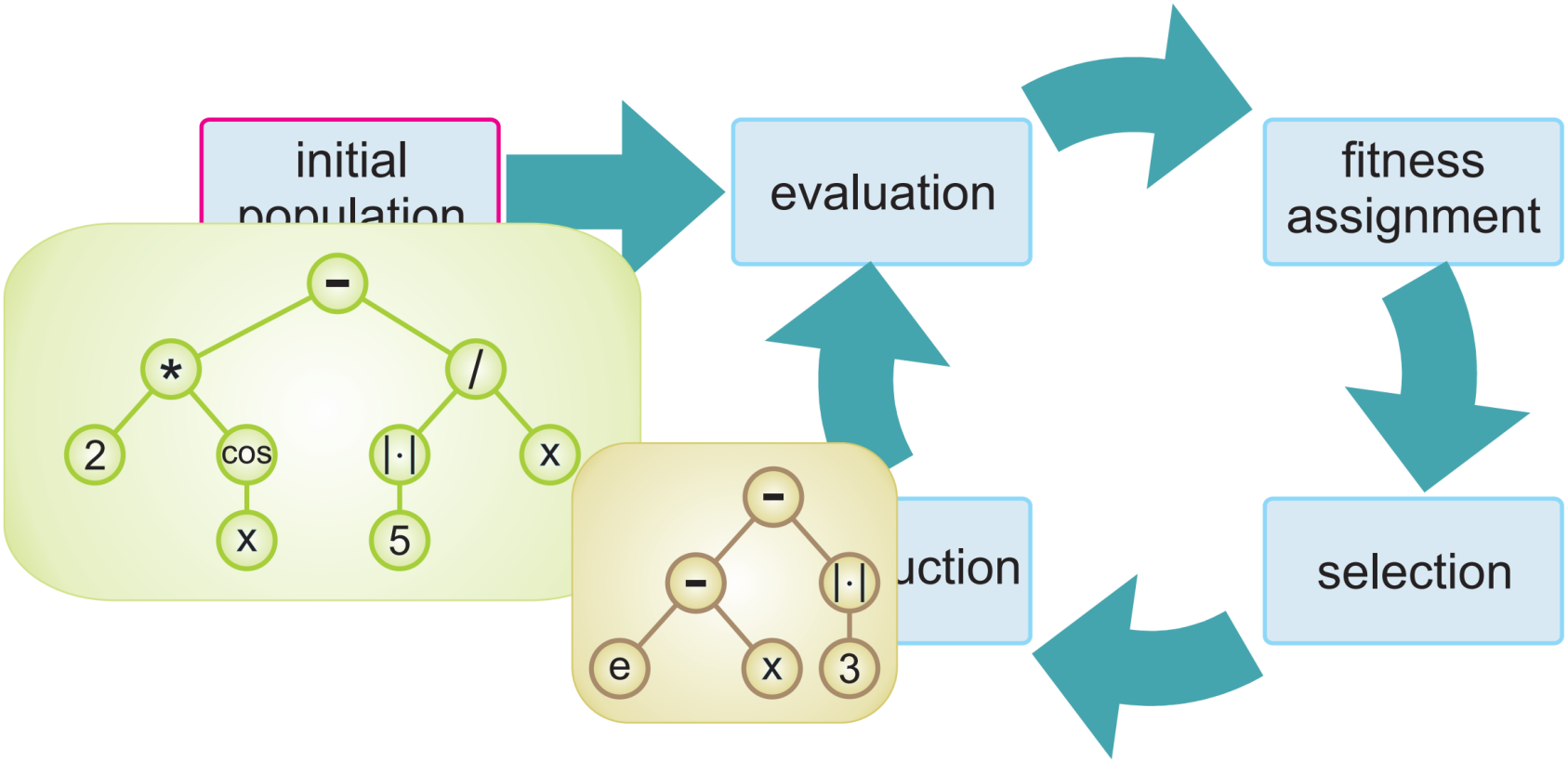
The function can be represented by the following tree:



GP Algorithm

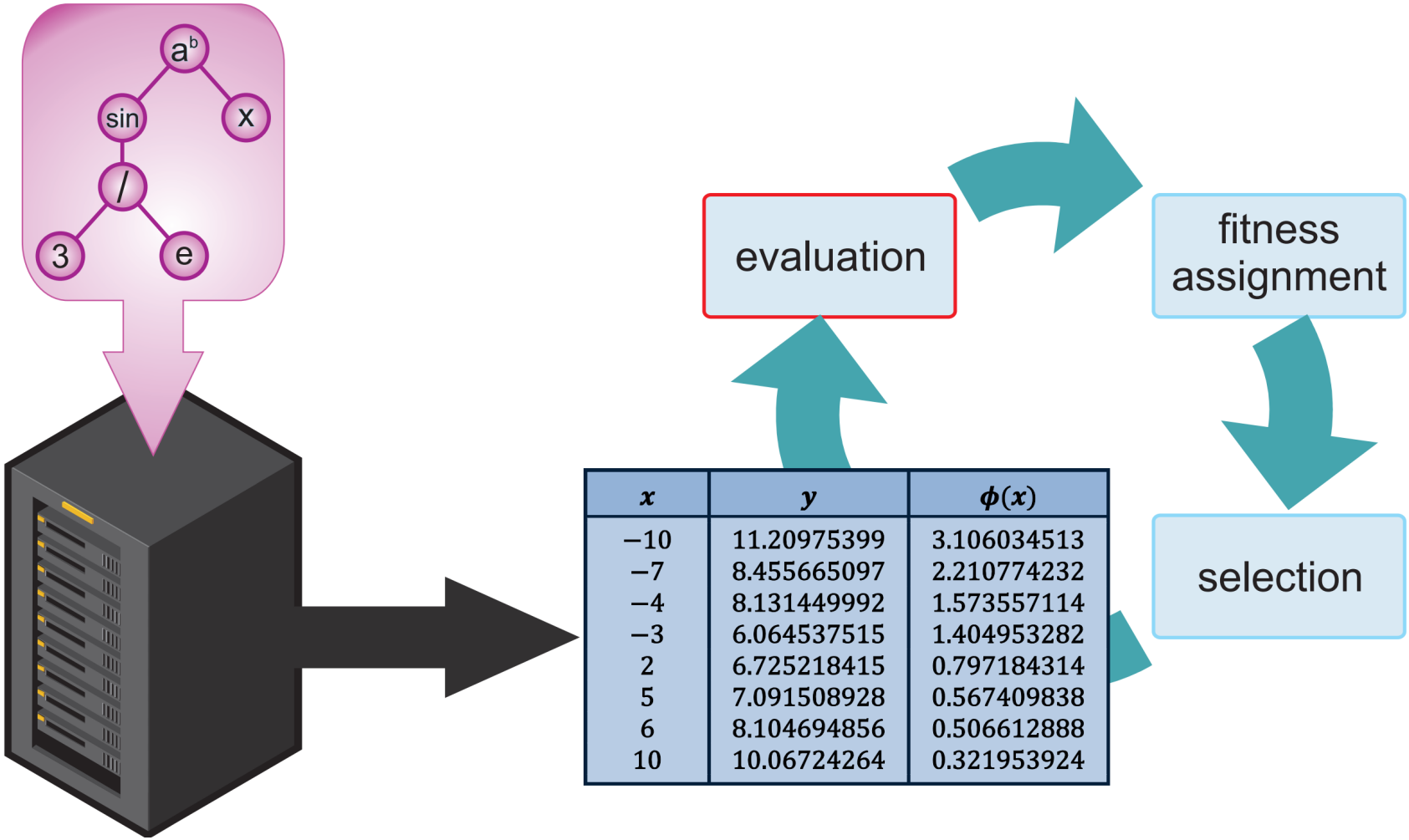
1. Formation of an initial random population computer of programs, using functions and terminals
2. Execution (!) of all programs and their evaluation (fitness function)
3. Selection of well-performing programs
4. Creation of a new population by applying genetic operators to selected programs
5. Return to 2 until finding a “good” program

GP Cycle: Initialization



Start with a random set of programs

GP Cycle: Evaluation



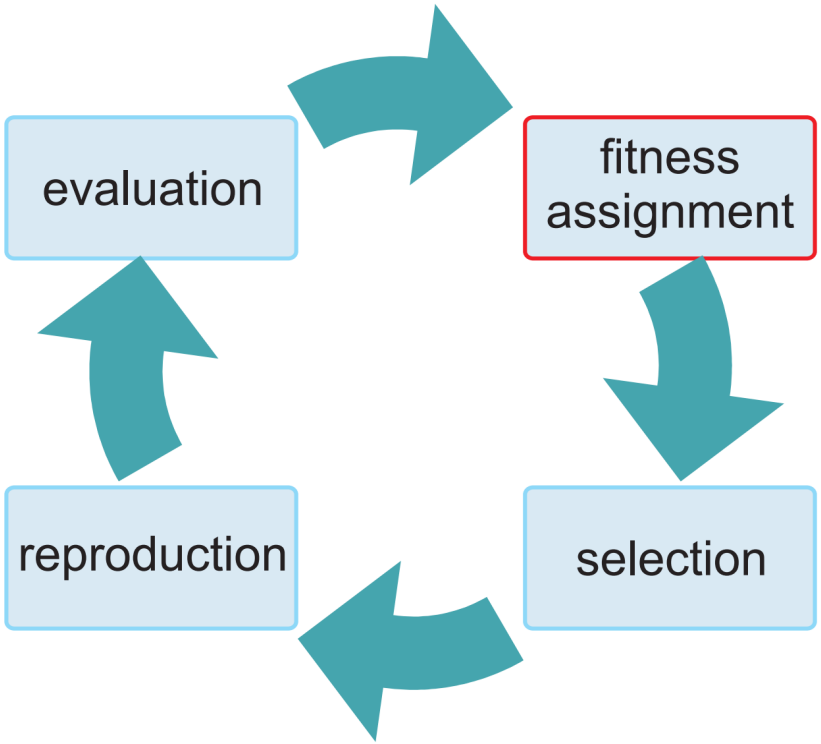
x	y	$\phi(x)$
−10	11.20975399	3.106034513
−7	8.455665097	2.210774232
−4	8.131449992	1.573557114
−3	6.064537515	1.404953282
2	6.725218415	0.797184314
5	7.091508928	0.567409838
6	8.104694856	0.506612888
10	10.06724264	0.321953924

Test the performance of each individual in simulation/evaluation plus criteria like size

GP Cycle: Fitness

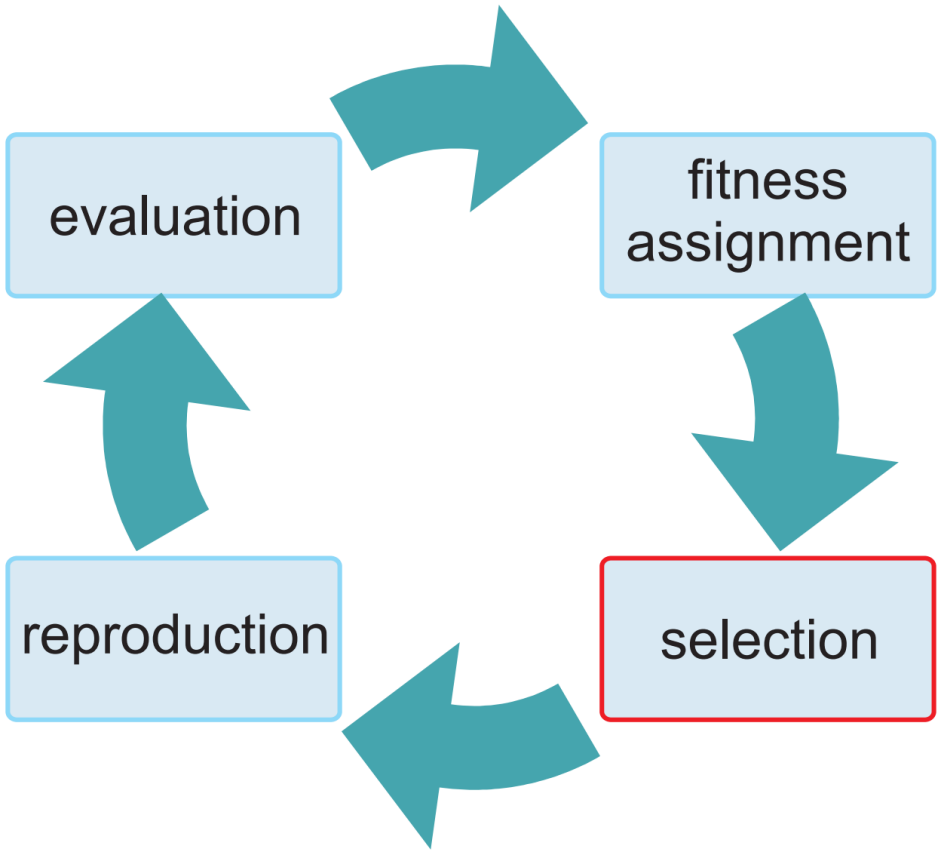
Fitness $\nu(\varphi)$ in Symbolic Regression could be...

- square error $f(\varphi)$
- $\nu(\varphi) = f(\varphi) + \text{penalty} * size(\varphi)$



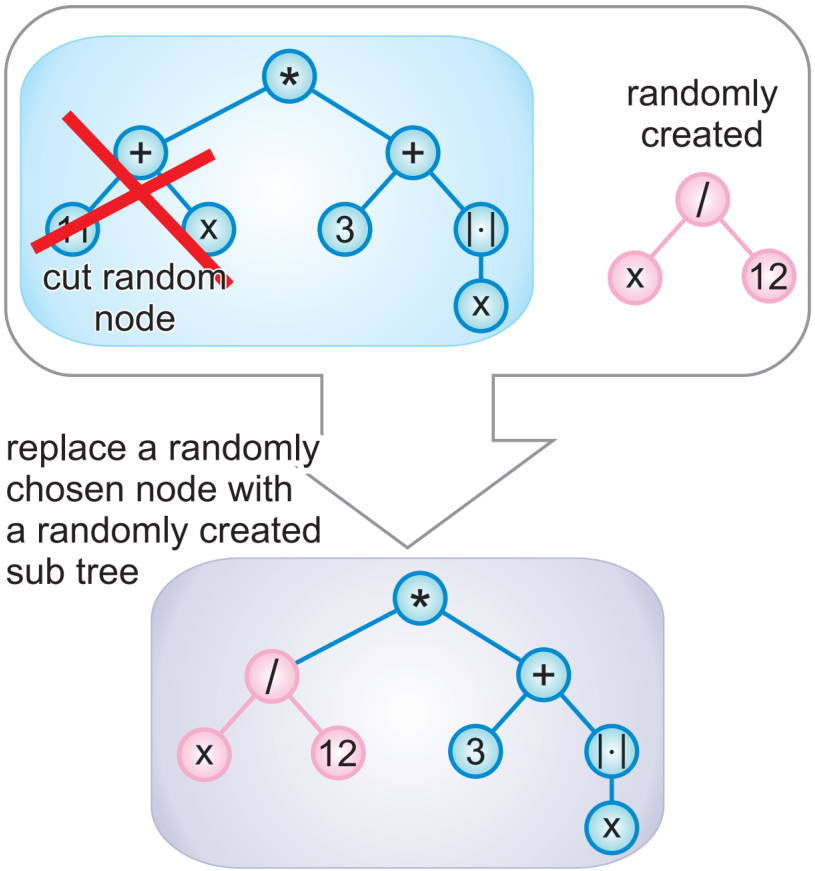
Fitness can be relative, determines expected number of offsprings

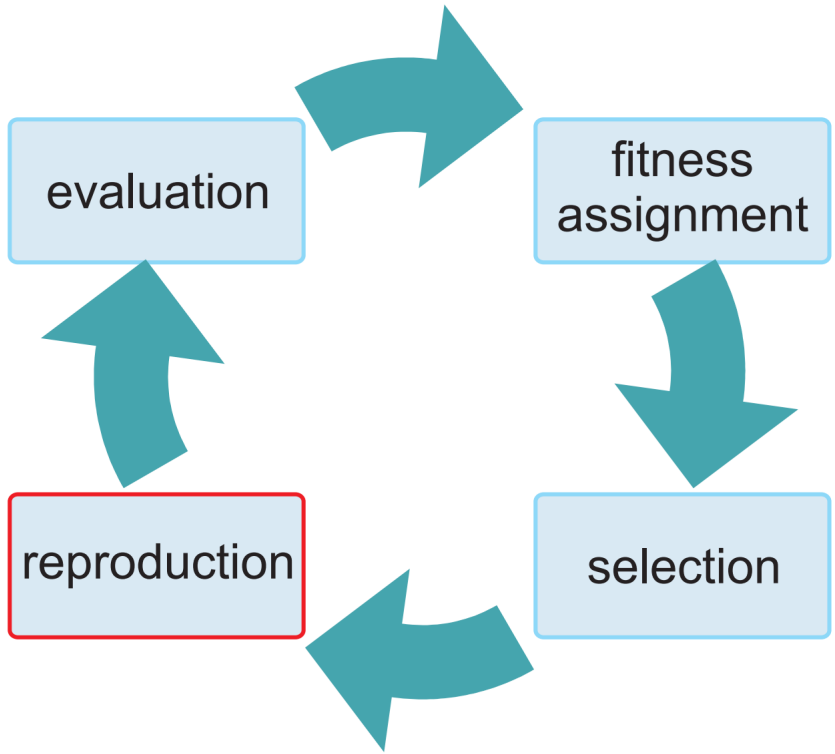
GP Cycle: Selection



Selection determines number of offspring based on fitness
In GP usually either fitness proportionate or tournament selection

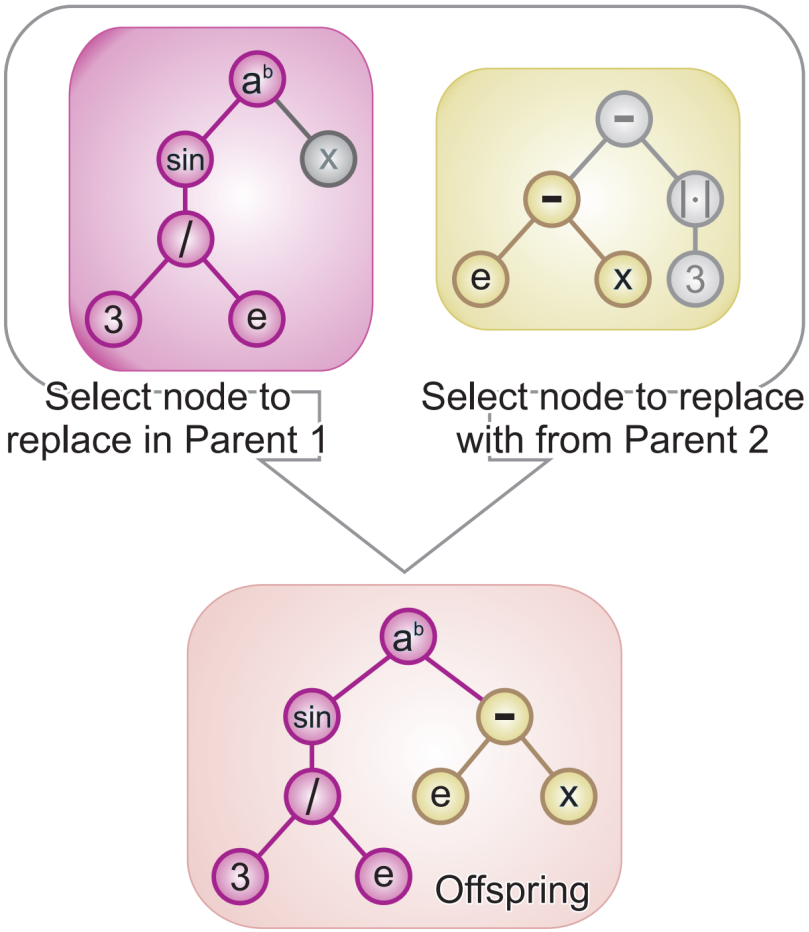
GP Cycle: Asexual Reproduction

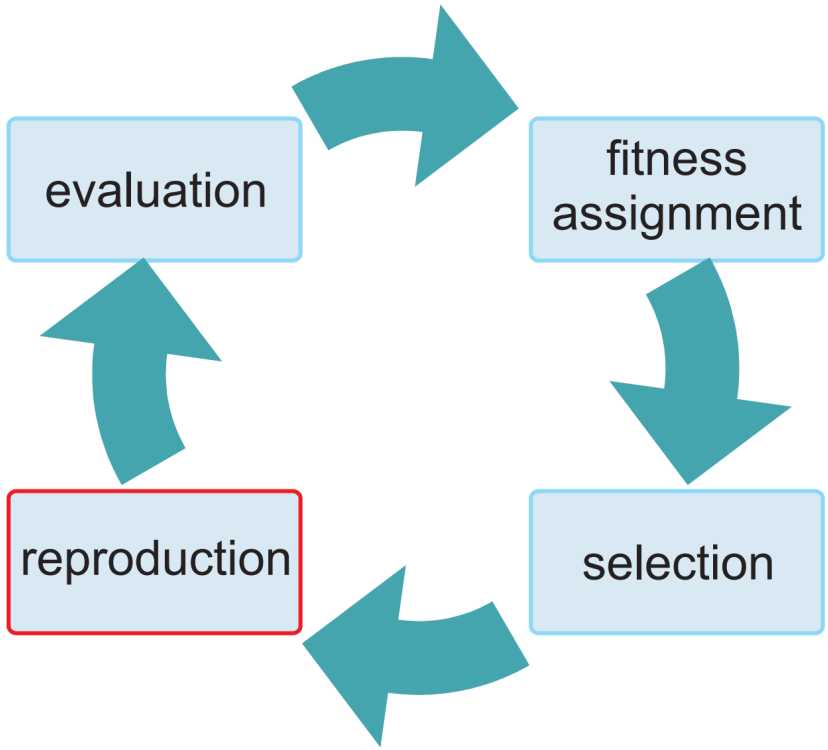




Asexual reproduction: mutation (insert random sub-tree)

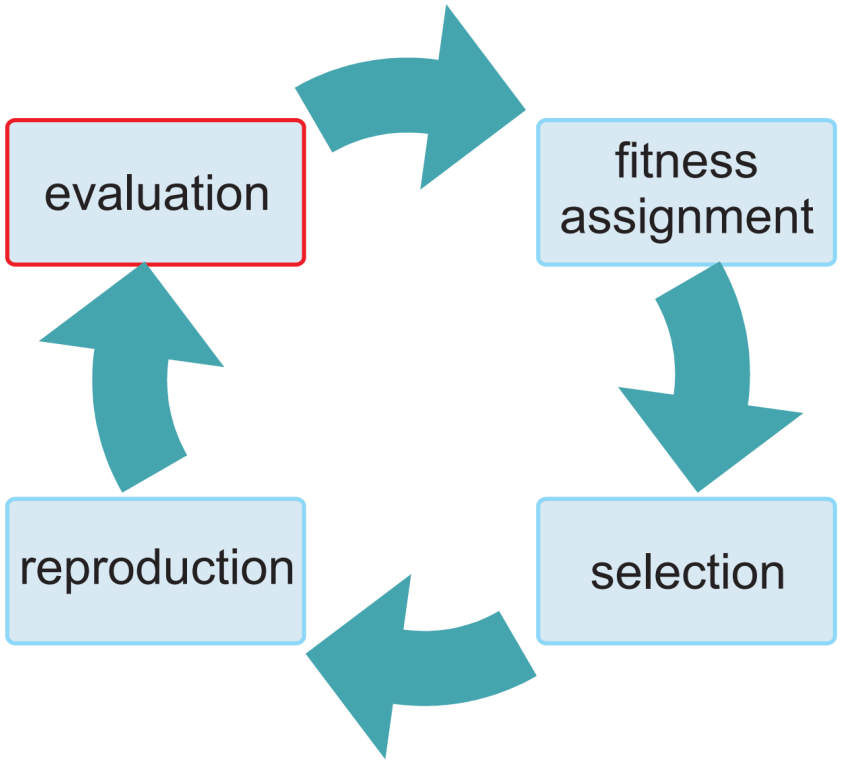
GP Cycle: Sexual Reproduction





Sexual reproduction: recombination (exchange sub trees)

GP Cycle: Itteration



Cycle starts all over until termination criterion is met

GP as search

- Genetic Programming performs a heuristic search in the space of computer programs (a type of “best-first” search with an opened list of limited size – the population)
- The heuristic function is the *fitness function*
- The search operators are the *genetic operators* (mutation and crossover)

Generation of an initial population

- Choose a function for the root of the tree
- Check the arity of the function
- For each argument of the function, generate:
  - a terminal; or
  - a subtree
- Practically, trees deeper that certain constant should not be generated

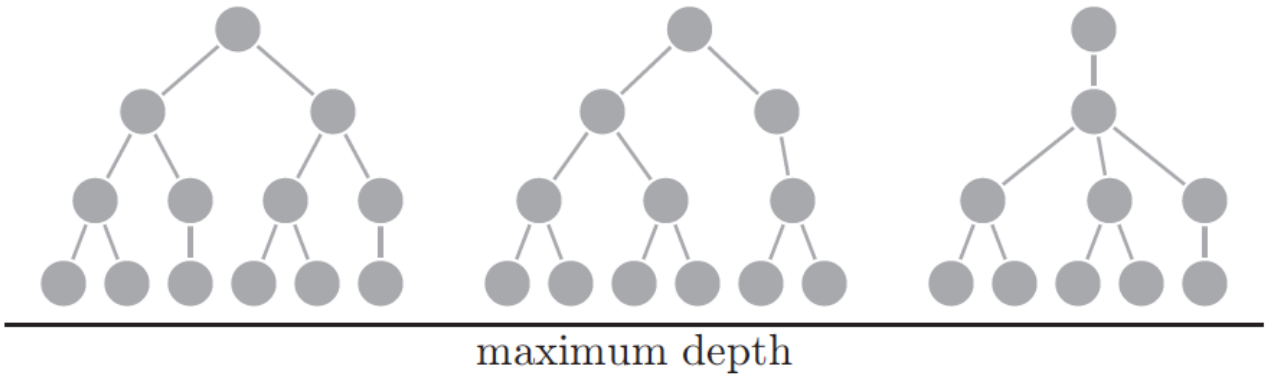
Tree Creation

- Tree Creation is a Nullary Reproduction

- In SGP an initial population composed of randomized individuals is needed (similarly to a set of random bit strings in GA).
- Normally, there is a maximum depth $\hat{d}$ specified that the tree individuals are not allowed to surpass.
- The creation operation will return only trees where the path between the root and the most distant lead node is not longer than $\hat{d}$.
- There are three basic ways for realizing the create() and "createPop" operations for trees - according to the depth of the produced individuals: full, grow, and ramped hal-and-half

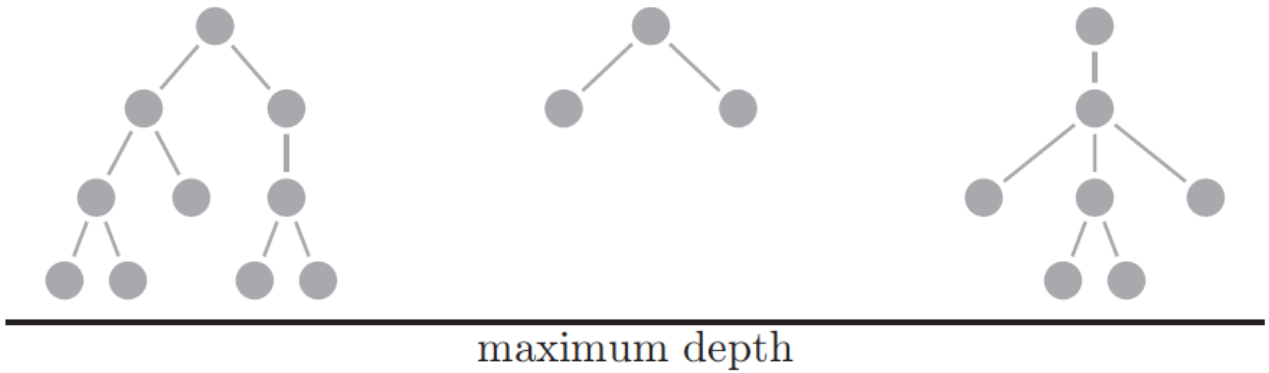
Full Method

- It creates trees where each (non-backtracking) path from the root to the leaf nodes has exactly the lenght $\hat{d}$.
- For each node of a depth below $\hat{d}$, the type is chosen from the non-terminal symbols.
- A chain of function nodes is recursively constructed until the maximum depth minus one.
- If $\hat{d}$ is reached, only leaf nodes (with types from $\sum$) can be attached.



Grow Method

- Creates trees where each (non-backtracking) path from the root to the leaf nodes is not longer than $\hat{d}$ but may be shorter.
- This is achieved by deciding randomly for each node if it should be a leaf or not when it is attached to the tree.
- Of course, to nodes of the depth $\hat{d} - 1$, only leaf nodes can be attached.



Ramped Half-and-Half Method

- Fills population with a good variety of trees of different shapes and sizes
- For each tree to be created, this algorithm draws a number r uniformly distributed between 2 and $\hat{d}$: $(r = \lfloor \text{randomUni}[2, \hat{d} + 1) \rfloor)$.
- Now, either *full* or *grow* is chosen to finally create a tree with the maximum depth r (in place of $\hat{d}$).

Functions and Terminals

- Functions: functions or macros that take arguments
  - e.g. `(+34)`
- Terminals:
  - Functions that do not have arguments; e.g. (advance)
  - Constant: 3, $a, \ldots$
  - (Ephemeral) random constant $\mathcal{R}$: for numerical problems and symbolic regression
  - Input variables: `D0,D1`, ...

Functions

- Funcions/terminals sufficient to express the solution (e.g. for boolean functions, `AND`, `OR` and `NOT` are sufficient)
- It may be useful to include powerful functions (so that the system does not have to rediscover them); e.g. `sin(x)`
- Differentiate between functions (evaluate arguments; e.g. `sum`) and macros (do not evaluate: `if-then-else`);
- Functions have to execute with any argument and without producing errors (closure); e.g. protection against division by zero
- In standard GP there are no data types. All functions must be prepared to receive any type and value of argument

Evaluation of programs (fitness)

- *Raw*:
  - e.g. number of correctly guessed cases or hits (even-parity)
  - e.g. function, such as [0.7 $\times$ points $+$ 0.3 $\times$ time] (Pacman)
- *Standard*: (to be minimized)
  - Standard Fitness = Maximum - raw
- *Adjusted*: $1/(1 +$ standard$)$; it exaggerates fitness near 0
- *Normalized* (or relative):
  - ajusted / (sum of adjusted fitnesses in population)

GP Operations

- Reproduction (selection)
- Mutation
- Crossover
- Architecture-altering operations

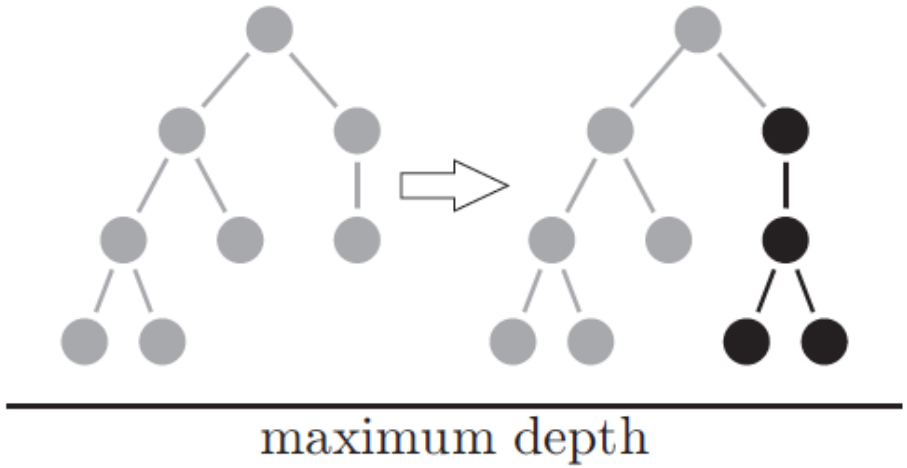
Selection

Selection of capable (well-performing) individuals

- Fitness proportional (probabilistic): uses normalized fitness
  - Problem: super-individuals
- Ranked
- Tournament
  - Match several individuals to compete among themselves: the best one reproduces
- Greedy Overselection (of the fitter individuals): create a high fitness group, H, and a low fitness group, L
  - 80% of the time select the parent from group H
  - 20% of the time select the parent from group L

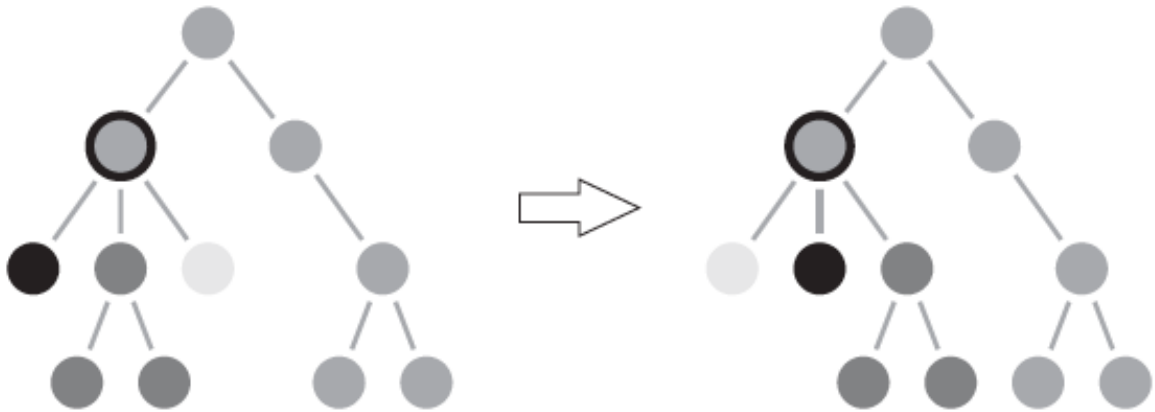
Unary Reproduction: Mutation

- Usually defined as the random selection of a node in the tree, removing this node and all of its children, and finally replacing it with another node.
  - A copy $g \in \mathbb{G}$ from the original genotype $g_p \in \mathbb{G}$ is created.
  - A random parent node t is selected from g.
  - One child of this node is replaced with a newly created subtree.
  - For creating this tree, the ramped half-and-half method is picked with a maximum depth similar to the one of the child which is replaced by it.



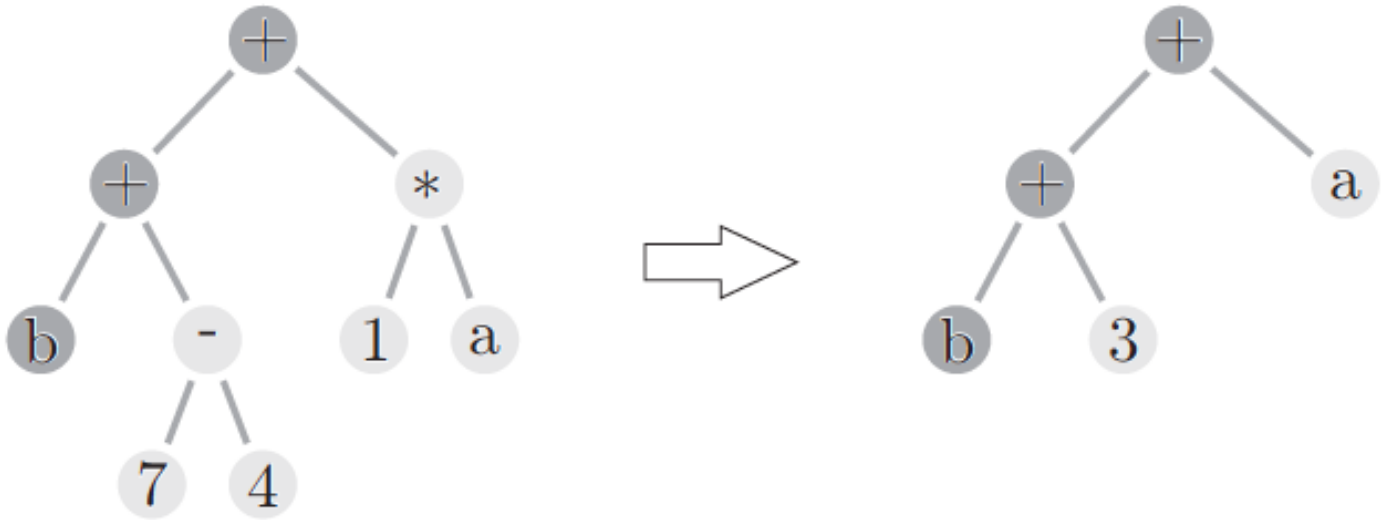
Unary Reproduction: Permutation

- Used to reproduce one single tree.
  - It first selects an internal node of the parental tree.
  - The child nodes attached to that node are then shuffled randomly.
  - The main goal of this operation is to re-arrange the nodes in highly fit subtrees in order to make them less fragile for other operations such as recombination.



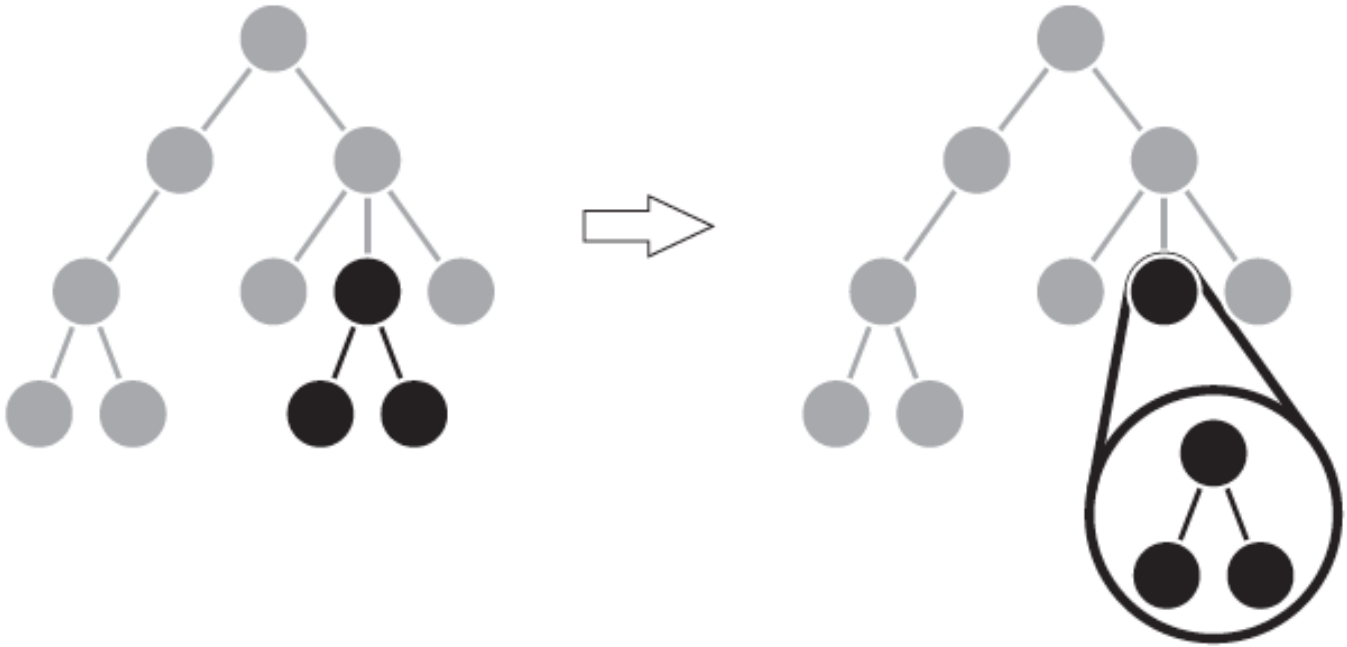
Unary Reproduction: Editing

- Editing means creation of a new offspring tree which is more efficient but equivalent to its parent in terms of functional aspects.
- It also reduces the diversity in the genome which could degrade the performance by decreasing the variety of structures available.



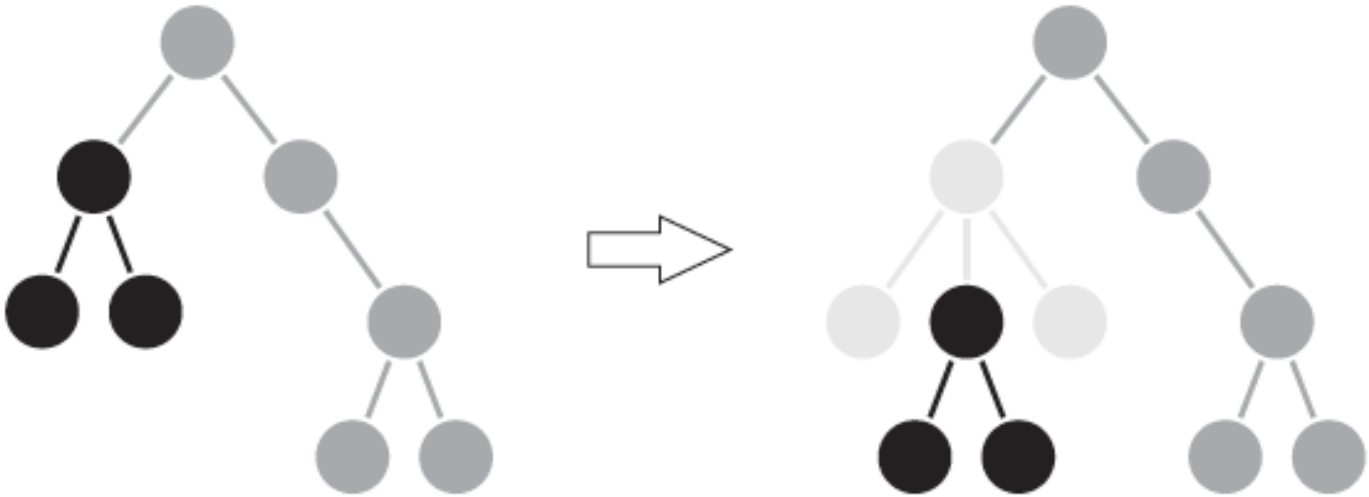
Unary Reproduction: Encapsulation

- The basic idea is to identify useful subtrees and to turn them into atomic building block.
  - New terminal symbols are created (internally hidden), which are trees with multiple nodes.
  - This way, they will no longer be subject to potential damage by other reproduction operations.
  - The new terminal may spread throughout the population in the further course of the evolution.



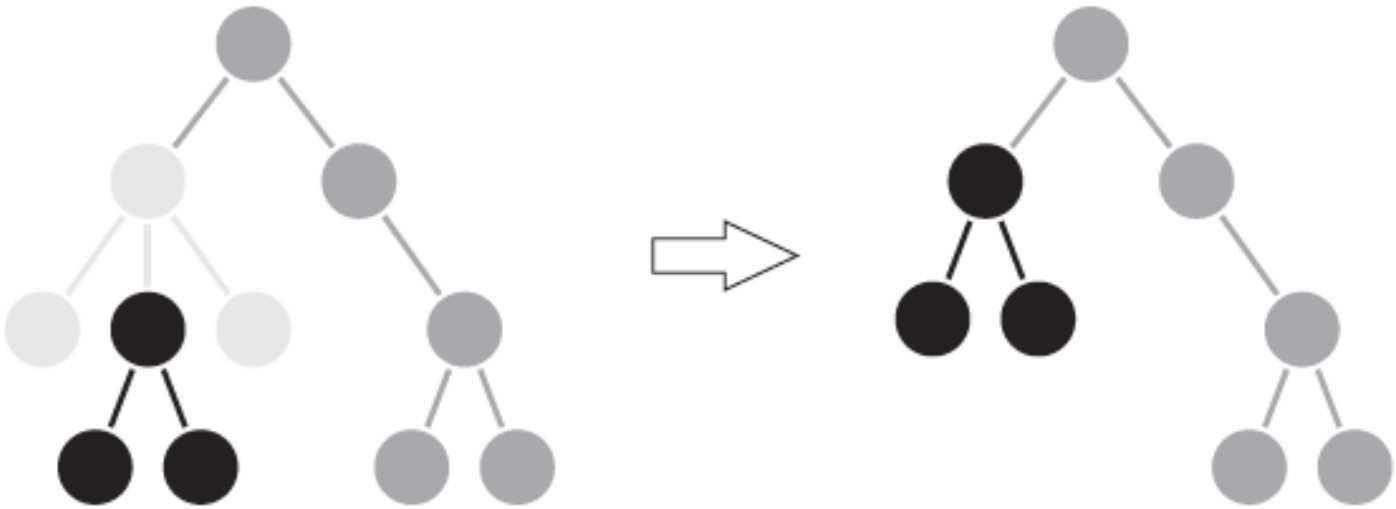
Unary Reproduction: Wrapping

- Its purpose is to allow modifications of non-terminal nodes that have a high probability of being useful.
  - Wrapping first selects an arbitrary node n in the tree.
  - A new non-terminal node m is created outside of the tree.
  - In m, at least one child node position is left unoccupied.
  - Then, n (and all its potential child nodes) is cut from the original tree and append it to m by plugging it into the free spot.
  - Finally, m is hung into the tree position formerly occupied by n.



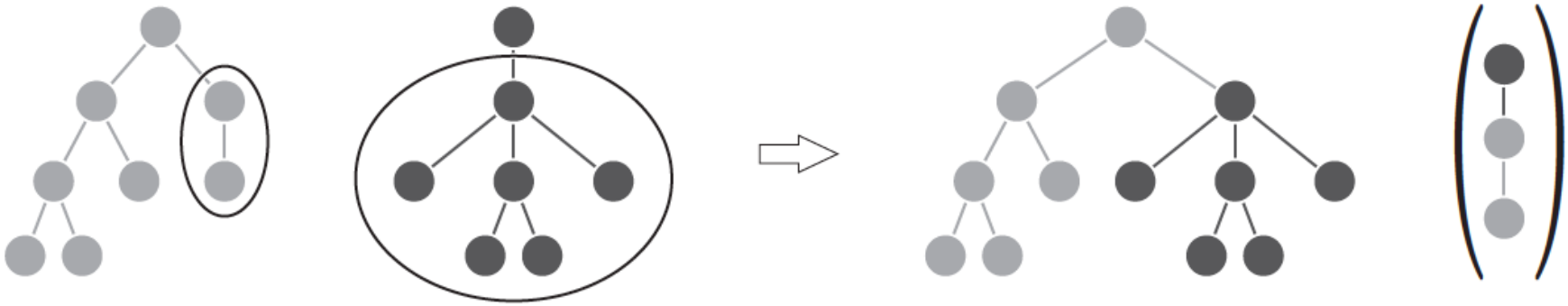
Unary Reproduction: Lifting

- Lifting is the inverse operation to wrapping that can remove nodes in non-terminal positions.
  - It begins with selecting an arbitrary inner node n of the tree.
  - This node then replaces its parent node.
  - The parent node inclusively all of its child nodes (except n) are removed from the tree.
  - With lifting, a tree that represents the mathematical formula $(b + a) * 3$ can be transformed to b*3 in a single step.



Binary Reproduction/Recombination: Subtree crossover

- Applying the default subtree exchange recombination operator to two parental trees means to swap subtrees between them.
- Therefore, single subtrees are selected randomly from each parent and subsequently cut out and reinserted in the partner genotype.
- If a depth restriction is imposed on the genome, both, the mutation and the recombination operation have to respect them.



maximum depth

Binary Reproduction/Recombination: Headless Chicken

Recombination in Standard Genetic Programming can also have a very destructive effect on the individual fitness.

1. *Strong Headless Chicken Crossover (SHCC).* Each parent tree is recombined with a new randomly created tree and the offspring is returned. The new child will normally contain a relatively small amount of random nodes.
2. *Weak Headless Chicken Crossover (WHCC).* The parent tree is recombined with a new randomly created tree and vice versa. One of the offspring randomly selected, and returned. In half of the returned offspring, there will be a relatively small amount of non-random code.

Binary Reproduction/Recombination: Advanced Techniques

Several techniques have been proposedto mitigate the destructive effects of recombination in tree-based representations.

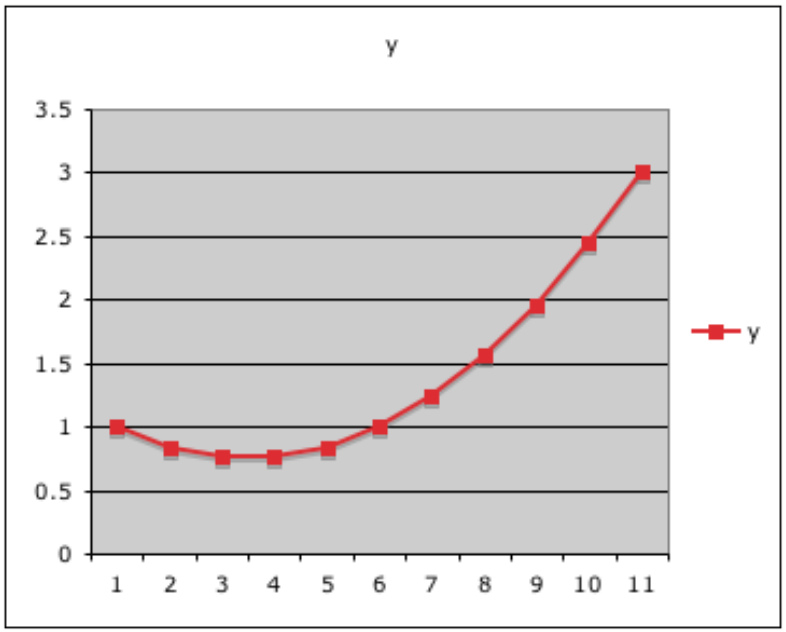
- Strong context preserving crossover - permits only the exchange of subtrees that occupied the same positions in the parents
- Other similar single-point crossovers were proposed for tree genomes with the same purpose: increasing the probability of exchanging genetic material which is structurally and functionally similar $\rightarrow$ decreasing the disruptiveness.
- Related approach was defined for linear Genetic Programming

Modules

- **Automatically Defined Functions(ADFs).** Provide some sort of pre-specified modularity for Genetic Programming. The root of the tree usually loses its functional responsibility and now serves only as glue that holds the individual together and has a fixed number n of children, from which $n-1$ are automatically defined functions and one is the result-generating branch.
- When evaluating the fitness of an individual, often only this first branch is taken into consideration whereas the root and the ADFs are ignored. The result-generating branch, however, may use any of the automatically defined functions to produce its output.
- **Automatically Defined Macros (ADMs)**. Complements ADFs; it only differs from ADFs in the way that their parameters are handled.

Example I: Nonlinear Function Identification

Find a nonlinear function that satisfies the following relationship:



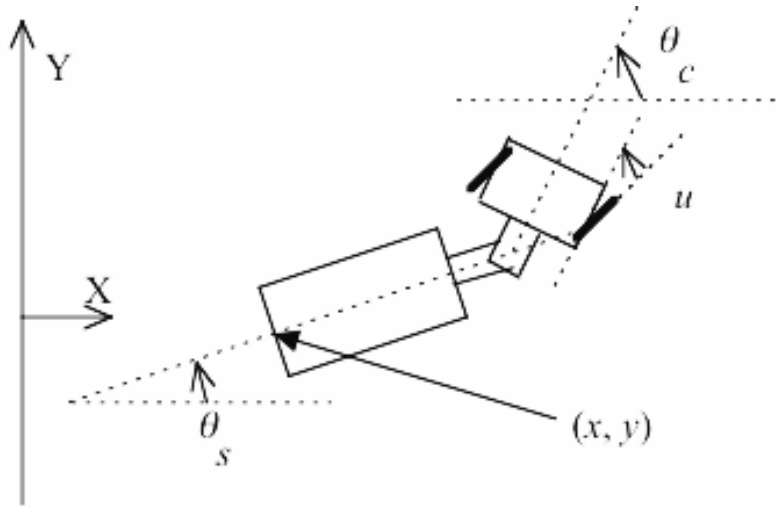
x	y
-1.00	1.00
-0.80	0.84
-0.60	0.76
-0.40	0.76
-0.20	0.84
0.00	1.00
0.20	1.24
0.40	1.56
0.60	1.96
0.80	2.44
1.00	3.00

Example I: Nonlinear Function Identification

Terminal set	$T = \{x, \text{Random-constants}\}$
Function set	$F = \{+, -, *, \%\}$
Fitness	Sum of errors between the candidate program's outputs and y's for all values of x.
Parameter	Population size: 4
Termination	When the sum of errors is less than, e.g. 0.1

Example II: Truck Backer Upper Control Law Design

Objective: Develop a control law $u = (x, y, \theta_S, \theta_C)$ that leads the trailer tail to $(x, y, \theta_S) = (0, 0, 0)$ when the backward linear velocity of the wheels is fixed.



Example II: Truck Backer Upper Control Law Design

There are four input variables of the controller:

- Horizontal position, x
- Vertical position, y
- Trailer angle, θ_S
- Cap angle, θ_C

Thus, *terminals* $T = \{x, y, \theta_S, \theta_C, \text{Random-constants}\}$

Function set: arithmetic and trigonometric functions

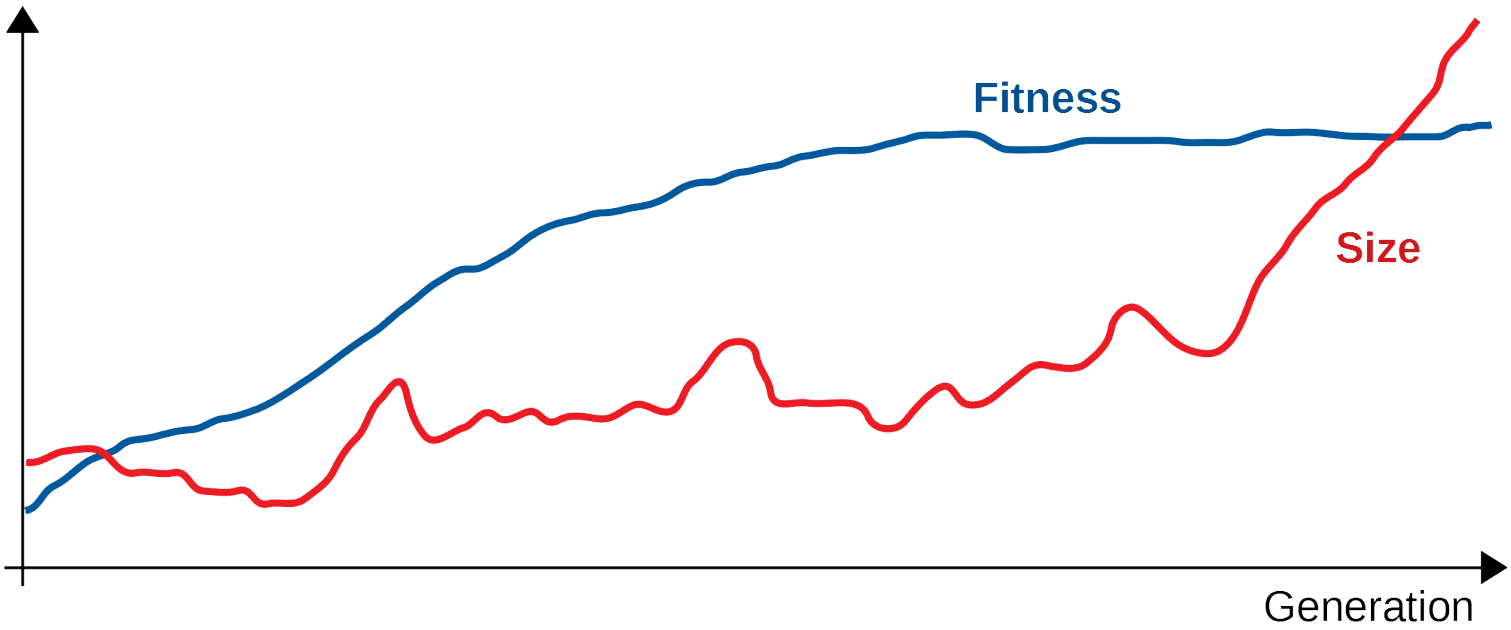
To *evaluate* a solution candidate during evolution process, we need to conduct simulation runs over many initial conditions of the trailer for hundreds of time steps.

Bloat in Genetic Programming

- Tree-representations are of variable size
- There may be big trees, there may be small trees
- What happens with the tree size during the course of the evolution?

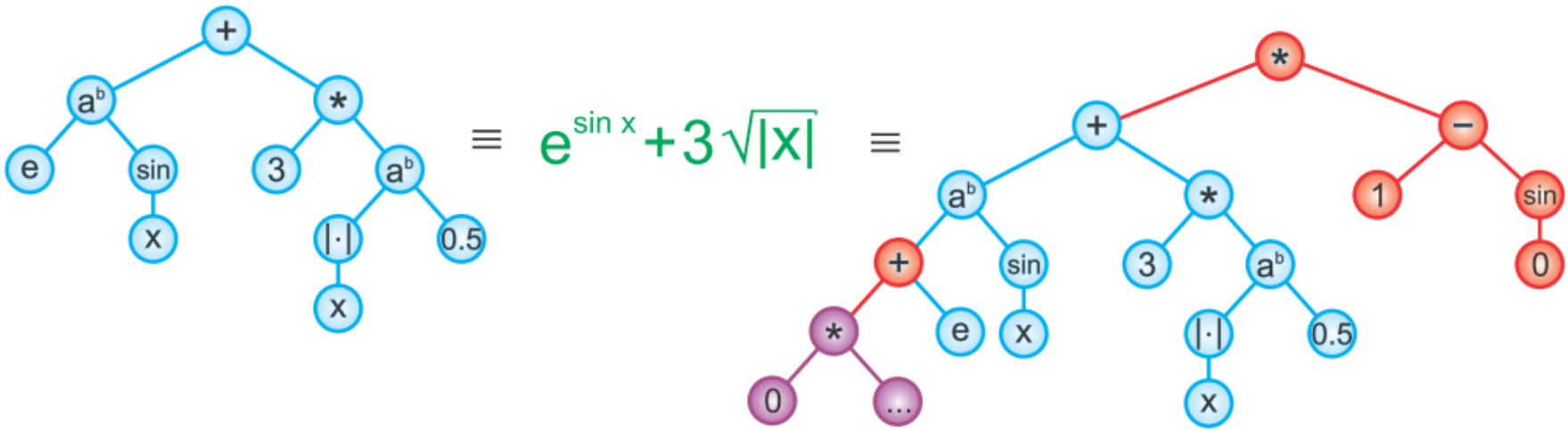
Bloat

Growth of program size without corresponding increase in program fitness (the optimal solution might still be a large program).



Bloat

- **Bloat** is uncontrolled growth of programs
- Example is **intron**, a useless part of program



Bloat

- Bloat is bad
  1. Elegant solutions are always simple and small
  2. Longer processing time for both, reproduction and evaluation
  3. Danger of overfitting
  4. Occupy more memory

Possible reasons	Possible remedies
Useless code hitchhiking and reproducing with good individuals (high pressure $\Rightarrow$ more bloat)	Use MO optimization: minimize also program size
Protection against reproduction ops: mutation in useless code has no impact $\Rightarrow$ program can survive	Use penalties in single-objective optimization
Removal bias: finite portion of removable useless nodes, but no direct limit to amount of nodes which can be inserted	Set a conservative upper bound for program size
Overfitting: code one decision for each training instance - perfect fitness (but no generalization)	Use specialized reproduction operators