Genetic Programming

Jiří Kubalík
Department of Cybernetics, CTU Prague

Intelligent Data Analysis
RESEARCH GROUP

http://cw.felk.cvut.cz/doku.php/courses/a0m33eoa/start

Contents

Genetic Programming (GP)

:: **GP shares with GA** the philosophy of survival and reproduction of the fittest and the analogy of naturally occurring genetic operators.

:: **GP differs from GA** in a representation, genetic operators and a scope of applications.

:: **GP is extension of the conventional GA** in which the structures undergoing adaptation are trees of dynamically varying size and shape representing hierarchical computer programs.

:: **Applications**

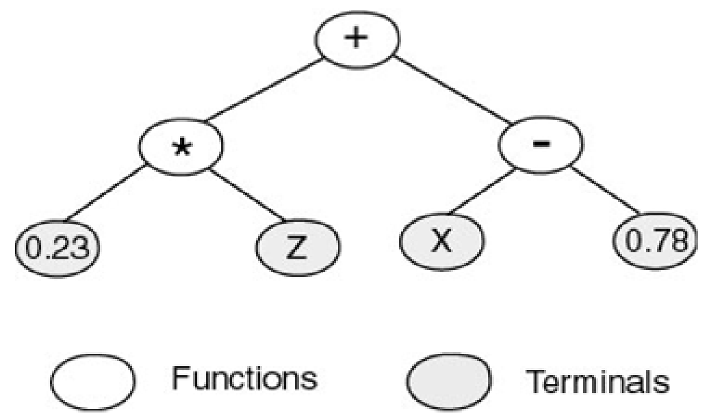
- learning programs,
- learning decision trees,
- learning rules,
- learning strategies,
- ...

GP: Representation

:: All possible trees are composed of **functions** (inner nodes) and **terminals** (leaf nodes) appropriate to the problem domain

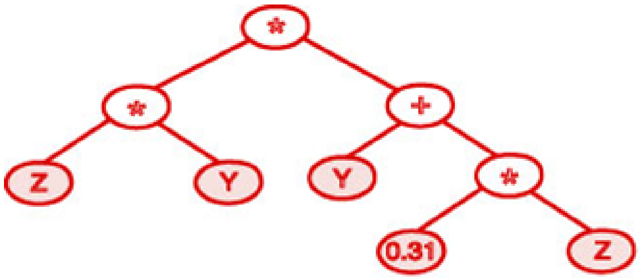
- **Terminals** – inputs to the programs (independent variables), real, integer or logical constants, actions.
- **Functions**
  - arithmetic operators (+, -, *, /),
  - algebraic functions (sin, cos, exp, log),
  - logical functions (AND, OR, NOT),
  - conditional operators (If-Then-Else, cond?true:false),
  - and others.

Example: Tree representation of a LISP S-expression $0.23 * Z + X - 0.78$

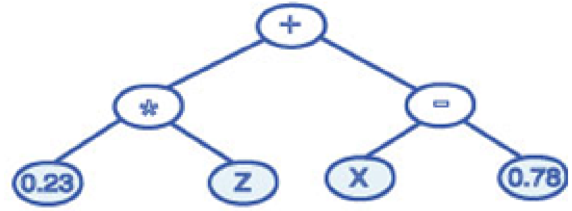


:: **Closure** – each of the functions should be able to accept, as its argument, any value that may be returned by any function and any terminal.

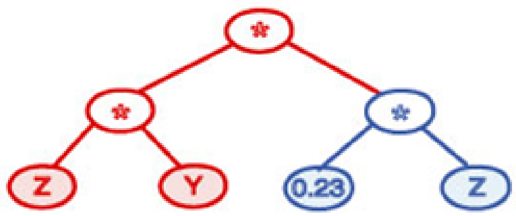
GP: Standard Crossover



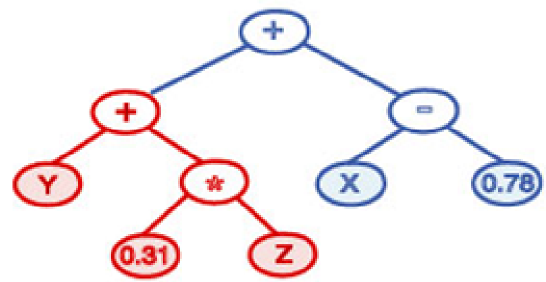
Parent 1: Z * Y * (Y + 0.31 * Z)



Parent 2: 0.23 * Z + X - 0.78



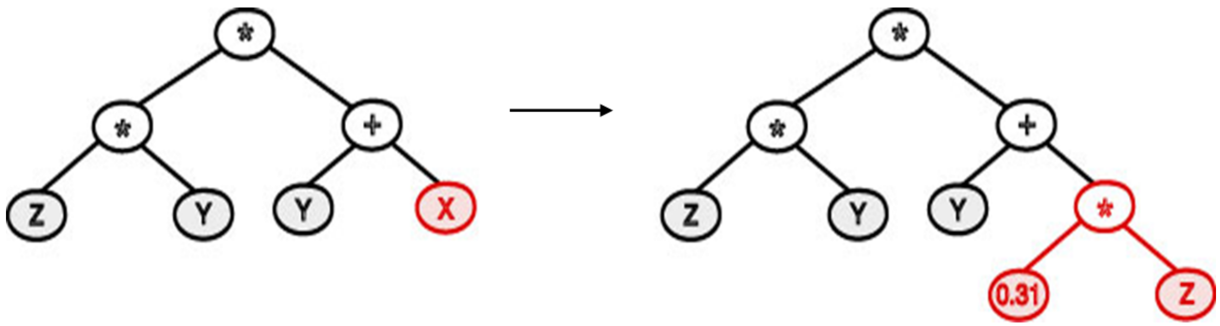
Child 1: 0.23 * Y * Z^2



Child 2: Y + 0.31 * Z + X - 0.78

GP: Subtree-Replacing Mutation

:: **Mutation** replaces selected subtree with a randomly generated new one.



Example of GP in Action: Trigonometric Identity

:: **Task** is to find an equivalent expression to $cos(2x)$.

:: **GP implementation:**

- **Terminal set** $T = \{x, 1.0\}$.
- **Function set** $F = \{+, -, *, \%, sin\}$.
- **Training cases**: 20 pairs (x_i, y_i), where x_i are values evenly distributed in interval $(0, 2\pi)$.
- **Fitness**: Sum of absolute differences between desired y_i and the values returned by generated expressions.
- **Stopping criterion**: A solution found that gives the error less than 0.01.

Example of GP in Action: Trigonometric Identity cont.

1. run, 13^{th} generation

$$(-(-1(*(sinx)(sinx))))(*(sinx)(sinx)))$$

which equals (after editing) to $1 - 2 * sin^2 x$

2. run, 34^{th} generation

$$(-1(*(*(sinx)(sinx))2))$$

which is just another way of writing the same expression.

Example of GP in Action: Trigonometric Identity cont.

1. run, 13^{th} generation

$$(-(-1(*(sin x)(sin x))))(*(sin x)(sin x)))$$

which equals (after editing) to $1 - 2 * sin^2 x$

2. run, 34^{th} generation

$$(-1(*(*(sin x)(sin x))2))$$

which is just another way of writing the same expression.

3. run, 30^{th} generation

$$\begin{array}{lll} (sin & (-(-2(*\ x\ 2)) & \\ & (sin(sin(sin(sin(sin(sin(*(sin & (sin\ 1)) \\ & & (sin\ 1)) \\ ))))))))) & & \end{array}$$

Example of GP in Action: Trigonometric Identity cont.

1. run, 13^{th} generation

$$(-(-1(*(sinx)(sinx))))(*(sinx)(sinx)))$$

which equals (after editing) to $1 - 2 * sin^2 x$

2. run, 34^{th} generation

$$(-1(*(*(sinx)(sinx))2))$$

which is just another way of writing the same expression.

3. run, 30^{th} generation

$$\begin{array}{lll} (sin & (-(-2(*\ x\ 2)) & \\ & (sin(sin(sin(sin(sin(sin(*(sin & (sin\ 1)) \\ & & (sin\ 1)) \\ ))))))))) & & \end{array}$$

The expression on the second and third row returns a value almost equal to $\pi/2$ so the discovered identity is

$$cos(2x) = sin(\pi/2 - 2x).$$

GP: Constant Creation

In many problems exact real-valued constants are required to be present in the correct solution (evolved program tree) $\Longrightarrow$ GP must have the ability to create arbitrary real-valued constant.

Ephemeral random constant $\mathfrak{R}$ – a special terminal.

- Initialization – whenever the ephemeral random constant $\mathfrak{R}$ is chosen for any endpoint of the tree during the creation of the initial population, a random number of a specified data type in a specified range is generated and attached to the tree at that point.

  Each occurrence of this terminal symbol invokes a generation of a unique value.

- Once these values are generated and inserted into initial program trees, these constants remain fixed.

- The numerous different random constants will become embedded in various subtrees in evolving trees.

  Other constants can be further evolved by crossing the existing subtrees, such a process being driven by the goal of achieving the highest fitness.

  The pressure of fitness function determines both the directions and the magnitudes of the adjustments in numerical constants.

GP Initialisation: Common Methods

GP needs a good tree-creation algorithm to create trees for the initial population and subtrees for subtree mutation.

Required characteristics:

- Light computationally complex; optimally linear in tree size.
- User control over expected tree size.
- User control over specific node appearance in trees.

`GROW` method (each branch has $depth \leq D$):

- nodes at depth $d < D_{max}$ randomly chosen from $F \cup T$,
- nodes at depth $d = D_{max}$ randomly chosen from T.

`FULL` method (each branch has $depth = D$):

- nodes at depth $d < D$ randomly chosen from function set F,
- nodes at depth $d = D$ randomly chosen from terminal set T.

`GROW`(depth d, max depth D)
Returns: a tree of depth $\leq D - d$
if $(d = D)$ return a random terminal
else
    choose a random func or term f
    if (f is terminal) return f
    else
        for each argument a of f
            fill a with GROW$(d + 1, D)$
        return f

GP Initialisation: Common Methods

Characteristics of GROW:

- does not have a size parameter – does not allow the user to create a desired size distribution,
- does not allow the user to define the expected probabilities of certain nodes appearing in trees,
- does not give the user much control over the tree structures generated.
- there is no appropriate way to create trees with either a fixed or average tree size or depth.

RAMPED HALF-AND-HALF – GROW & FULL method each deliver half of the initial population.

D is chosen between 2 to 6,

GP Initialization: Probabilistic Tree-Creation Method

Probabilistic tree-creation method:

- An expected desired tree size can be defined.
- Probabilities of occurrence for specific functions and terminals within the generated trees can be defined.
- Fast – running in time near-linear in tree size.

Definitions:

- $\mathbf{T}$ denotes a newly generated tree.
- D is the maximal depth of a tree.
- E_{tree} is the expected tree size of $\mathbf{T}$.
- F is a function set divided into terminals T and nonterminals N.
- p is the probability that an algorithm will pick a nonterminal.
- b is the expected number of children to nonterminal nodes from N.
- g is the expected number of children to a newly generated node in $\mathbf{T}$.

$$g = pb + (1 - p)(0) = pb$$

GP Initialization: Probabilistic Tree-Creation Method 1

PTC1 is a modification of `GROW` that

- allows the user to define probabilities of appearance of functions within the tree,
- gives user a control over expected desired tree size, and guarantees that, on average, trees will be of that size,
- does not give the user any control over the variance in tree sizes.

Given:

- maximum depth bound D
- function set F consisting of N and T
- expected tree size, E_{tree}
- probabilities q_t and q_n for each $t \in T$ and $n \in N$
- arities b_n of all nonterminals $n \in N$

Calculates the probability, p, of choosing a nonterminal over a terminal according to

$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}$$

```
PTC1(depth d)
  Returns: a tree of depth d ≤ D
  if(d = D) return a terminal from T
      (by q_t probabilities)
  else if(rand < p)
      choose a nonterminal n from N
          (by q_n probabilities)
          for each argument a of n
              fill a with PTC1(d + 1)
          return n
  else return a terminal from T
      (by q_t probabilities)
```

Probabilistic Tree-Creation Method PTC1: Proof of p

- The expected number of nodes at depth d is $E_d = g^d$, for $g \geq 0$ (the expected number of children to a newly generated node).
- E_{tree} is the sum of E_d over all levels of the tree, that is

$$E_{tree} = \sum_{d=0}^{\infty} E_d = \sum_{d=0}^{\infty} g^d$$

From the geometric series, for $g \geq 0$

$$E_{tree} = \begin{cases} \frac{1}{1-g}, & \text{if } g < 1 \\ \infty, & \text{if } g \geq 1. \end{cases}$$

Probabilistic Tree-Creation Method PTC1: Proof of p

- The expected number of nodes at depth d is $E_d = g^d$, for $g \geq 0$ (the expected number of children to a newly generated node).
- E_{tree} is the sum of E_d over all levels of the tree, that is

$$E_{tree} = \sum_{d=0}^{\infty} E_d = \sum_{d=0}^{\infty} g^d$$

  From the geometric series, for $g \geq 0$

$$E_{tree} = \begin{cases} \frac{1}{1-g}, & \text{if } g < 1 \\ \infty, & \text{if } g \geq 1. \end{cases}$$

- The expected tree size E_{tree} (we are interested in the case that E_{tree} is finite) is determined solely by g, the expected number of children of a newly generated node.

Probabilistic Tree-Creation Method PTC1: Proof of p

- The expected number of nodes at depth d is $E_d = g^d$, for $g \geq 0$ (the expected number of children to a newly generated node).
- E_{tree} is the sum of E_d over all levels of the tree, that is

$$E_{tree} = \sum_{d=0}^{\infty} E_d = \sum_{d=0}^{\infty} g^d$$

  From the geometric series, for $g \geq 0$

$$E_{tree} = \begin{cases} \frac{1}{1-g}, & \text{if } g < 1 \\ \infty, & \text{if } g \geq 1. \end{cases}$$

- The expected tree size E_{tree} (we are interested in the case that E_{tree} is finite) is determined solely by g, the expected number of children of a newly generated node.

  Since $g = pb$, given a constant, nonzero b (the expected number of children of a nonterminal node from N), a p can be picked to produce any desired g.

  Thus, a g (and hence a p) can be picked to determine any desired E_{tree}.

Probabilistic Tree-Creation Method PTC1: Proof of p

- From

$$E_{tree} = \frac{1}{1 - pb}$$

we get

$$p = \frac{1 - \frac{1}{E_{tree}}}{b}$$

Probabilistic Tree-Creation Method PTC1: Proof of p

- From

$$E_{tree} = \frac{1}{1 - pb}$$

we get

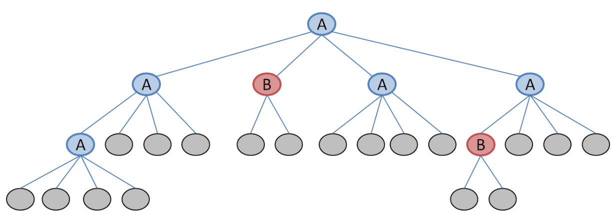
$$p = \frac{1 - \frac{1}{E_{tree}}}{b}$$

After substituting $\sum_{n \in N} q_n b_n$ for b we get

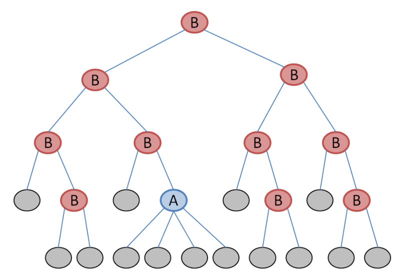
$$p = \frac{1 - \frac{1}{E_{tree}}}{\sum_{n \in N} q_n b_n}.$$

- User can bias typical bushiness of a tree by adjusting the occurrence probabilities of nonterminals with large fan-outs and small fan-outs, respectively.

  Example: Nonterminal A has four children branches, nonterminal B has two children branches.



$p_A > p_B$



$p_A < p_B$

GP: Selection

Commonly used are the fitness proportionate roulette wheel selection or the tournament selection.

Greedy over-selection is recommended for complex problems that require large populations (> 1000) – the motivation is to increase efficiency by increasing the chance of being selected to the fitter individuals in the population

- rank population by fitness and divide it into two groups:
  - group I: the fittest individuals that together account for $x\%$ of the sum of fitness values in the population,
  - group II: remaining less fit individuals.
- 80% of the time an individual is selected from group I in proportion to its fitness; 20% of the time, an individual is selected from group II.
- For population size = 1000, 2000, 4000, 8000, $x = 32\%$, 16%, 8%, 4%.

  %'s come from a rule of thumb.

GP: Selection

Commonly used are the fitness proportionate roulette wheel selection or the tournament selection.

Greedy over-selection is recommended for complex problems that require large populations (> 1000) – the motivation is to increase efficiency by increasing the chance of being selected to the fitter individuals in the population

- rank population by fitness and divide it into two groups:
  - group I: the fittest individuals that together account for $x\%$ of the sum of fitness values in the population,
  - group II: remaining less fit individuals.
- 80% of the time an individual is selected from group I in proportion to its fitness; 20% of the time, an individual is selected from group II.
- For population size = 1000, 2000, 4000, 8000, $x = 32\%, 16\%, 8\%, 4\%$.

  %'s come from a rule of thumb.

Example: Effect of greedy over-selection for the 6-multiplexer problem

Population size	I(M,i,z) without over-selection	I(M,i,z) with over-selection
1,000	343,000	33,000
2,000	294,000	18,000
4,000	160,000	24,000

GP: Crossover Operators

Standard crossover operators used in GP, like standard 1-point crossover, are designed to ensure just the syntactic closure property.

- On the one hand, they produce syntactically valid children from syntactically valid parents.
- On the other hand, the only semantic guidance of the search is from the fitness measured by the difference of behavior of evolving programs and the target programs.

  This is very different from real programmers' practice where any change to a program should pay heavy attention to the change in semantics of the program.

To remedy this deficiency in GP genetic operators making use of the semantic information has been introduced:

- Semantically Driven Crossover (SDC)

  [Beadle08] Beadle, L., Johnson, C.G.: Semantically Driven Crossover in Genetic Programming, 2008.

- Semantic Aware Crossover (SAC)

  [Nguyen09] Nguyen, Q.U. et al.: Semantic Aware Crossover for Genetic Programming: The Case for Real-Valued Function Regression, 2009.

GP: Semantically Driven Crossover

- Applied to **Boolean domains**.
- The semantic equivalence between parents and their children is checked by transforming the trees to *reduced ordered binary decision diagrams* (ROBDDs).

  Eliminates two types of introns (code that does not contribute to the fitness of the program)
  - Unreachable code – `(IF A1 D0 (IF A1 (`*`AND D0 D1`*`) D1))`
  - Redundant code – `AND A1 A1`

  Trees are considered semantically equivalent if and only if they reduce to the same ROBDDs.
- If the children are semantically equivalent to their parents then they are not copied to the next generation – the crossover is repeated until semantically non-equivalent children are produced.
- This way the semantic diversity of the population is increased.

GP: Semantically Driven Crossover

- Applied to **Boolean domains**.
- The semantic equivalence between parents and their children is checked by transforming the trees to *reduced ordered binary decision diagrams* (ROBDDs).

  Eliminates two types of introns (code that does not contribute to the fitness of the program)
  - Unreachable code – `(IF A1 D0 (IF A1 (AND D0 D1) D1))`
  - Redundant code – `AND A1 A1`

  Trees are considered semantically equivalent if and only if they reduce to the same ROBDDs.
- If the children are semantically equivalent to their parents then they are not copied to the next generation – the crossover is repeated until semantically non-equivalent children are produced.
- This way the semantic diversity of the population is increased.

SDC was reported useful in increasing GP performance as well as reducing code bloat (compared to GP with standard Koza's crossover).

- SDC significantly reduces the depth of programs (smaller programs).
- SDC average maximum score is significantly higher than the standard GP.
- SDC significantly increases the standard deviation of score – SDC algorithm is performing wider search, yielding better results.

GP: Semantic Aware Crossover

- Applied to **real-valued domains**.
- Determining semantic equivalence between two real-valued expressions is NP-hard.
- **Approximate semantics are calculated** – the compared expressions are measured against a random set of points sampled from the domain.

  Two trees are considered semantically equivalent if the output of the two trees on the random sample set are close enough – subject to a parameter ε called *semantic sensitivity*.

  ```
  if (Abs(output(P1) - output(P2)) < ε)
      then return "P1 is semantically equivalent to P2"
  ```

  Equivalence checking is used both for individual trees and subtrees.
- Scenario for using the semantic information – motivation is *to encourage individual trees to exchange subtrees that have different semantics*.

  Constraint crossover:
  - Crossover points at subtrees to be swapped are found at random in the two parental trees.
  - If the two subtrees are semantically equivalent, the operator is forced to be executed on two new crossover points.

GP: Semantic Aware Crossover

Constraint crossover:

choose at random crossover points at $Subtree_1$ in P1, and at $Subtree_2$ in P2
if ($Subtree_1$ is not equivalent with $Subtree_2$) execute crossover
else
    choose at random crossover points at $Subtree_1$ in P1, and $Subtree_2$ in P2
    execute crossover

Semantic guidance on the crossover (SAC) was reported

- to improve GP in terms of the number of successful runs in solving a class of *real-valued symbolic regression problem*,
- to increase the semantic diversity of population,
- to carry out much fewer semantically equivalent crossover events than standard GP,

  Given that in almost all such cases the children have identical fitness with their parents, SAC is more semantic exploratory than standard GP.
- to help reduce the crossover destructive effect.

  SAC is more fitness constructive than standard GP – the percentage of crossover events generating a better child from its parents is significantly higher in SAC.

Automatically Defined Functions: Motivation

Hierarchical problem-solving ("divide and conquer") may be advantageous in solving large and complex problems because the solution to an overall problem may be found by decomposing it into smaller and more tractable subproblems in such a way that the solutions to the subproblems are reused many times in assembling the solution to the overall problem.

Automatically Defined Functions: Motivation

Hierarchical problem-solving ("divide and conquer") may be advantageous in solving large and complex problems because the solution to an overall problem may be found by decomposing it into smaller and more tractable subproblems in such a way that the solutions to the subproblems are reused many times in assembling the solution to the overall problem.

Automatically Defined Functions (Koza94) – idea similar to reusable code represented by subroutines in programming languages.

- Reuse eliminates the need to "reinvent the wheel" on each occasion when a particular sequence of steps may be useful.
- Subroutines are reused with different instantiation of dummy variables.
- Reuse makes it possible to exploit a problem's modularities, symmetries and regularities.
- Code encapsulation – protection from crossover and mutation.
- Simplification – less complex code, easier to evolve.
- Efficiency – acceleration of the problem-solving process (i.e. the evolution).

[Koza94] Genetic Programming II: Automatic Discovery of Reusable Programs, 1994

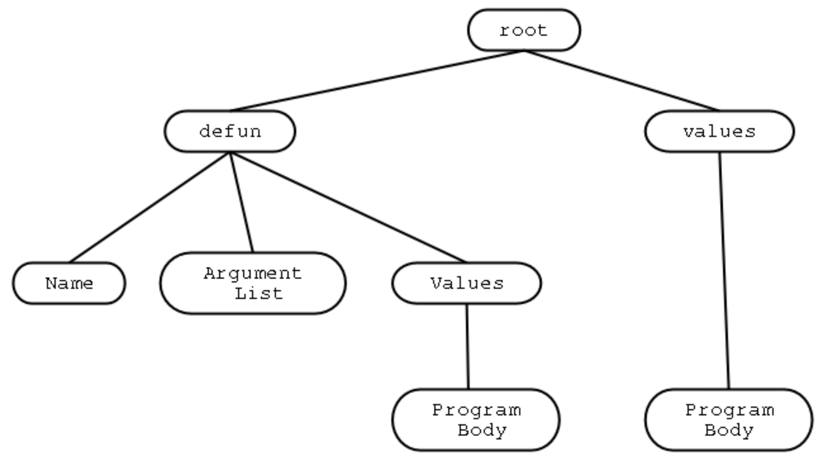
Automatically Defined Functions: Structure of Programs with ADFs

Function defining branches (ADFs) – each ADF resides in a separate function-defining branch.

Each ADF

- can posses zero, one or more formal parameters (dummy variables),
- belongs to a particular individual (program) in the population,
- may be called by the program's result-producing branch(es) or other ADFs.

Typically, the ADFs are invoked with different instantiations of their dummy variables.

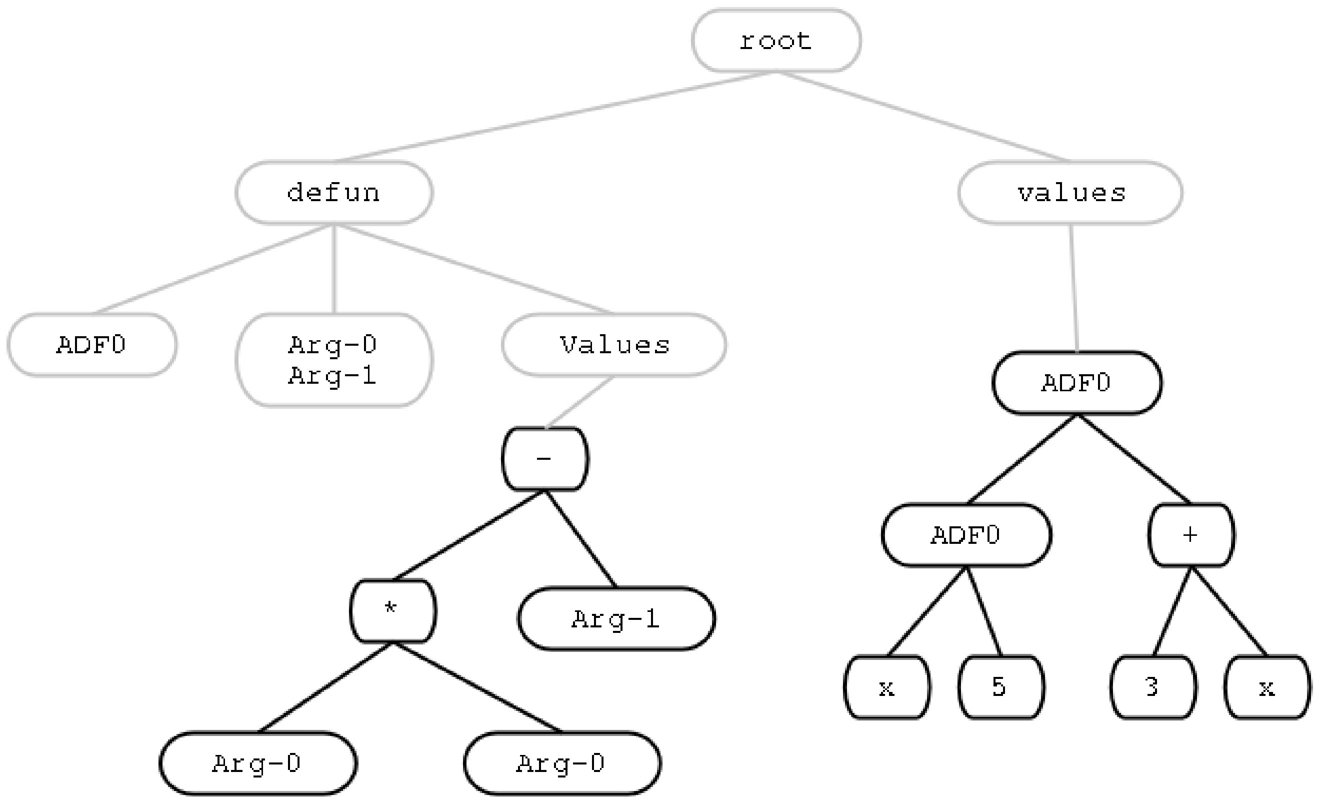


Result-producing branch (RPB) – "main" program (Can be one or more).

The RPBs and ADFs can have different function and terminal sets.

ADFs as well as RPBs undergo the evolution through the crossover and mutation operations.

ADF: Tree Example for Symbolic Regression of Real-valued Functions



Terminal set: Arg-0 Arg-1
Function set: task-dependent
(e.g. +,-,*)

Terminal set: task-dependent
(e.g. X, 1, 2, 3, 4, 5)
Function set: ADF0 + task-dependent
(e.g. ADF0, +, -, *)

ADF: Symbolic Regression of Even-Parity Functions

Even-n-parity function of n Boolean arguments returns `true` if the number of `false` arguments is even and returns `false` otherwise.

- function is uniquely specified by the value of the function for each of the 2^n possible combinations of its n arguments.

Even-3-parity – the truth table has $2^3 = 8$ rows.

	D2	D1	D0	Output
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Even-3-Parity Function: Blind Search vs. Simple GP

Experimental setup:

- Function set: F $=$ {AND, OR, NAND, NOR}
- The number of internal nodes fixed to 20.
- Blind search – randomly samples 10,000,000 trees
- GP without ADFs
  - Population size $M = 50$.
  - Number of generations $G = 25$.
  - A run is terminated as soon as it produces a correct solution.
  - Total number of trees generated 10,000,000.

Even-3-Parity Function: Blind Search vs. Simple GP

Experimental setup:

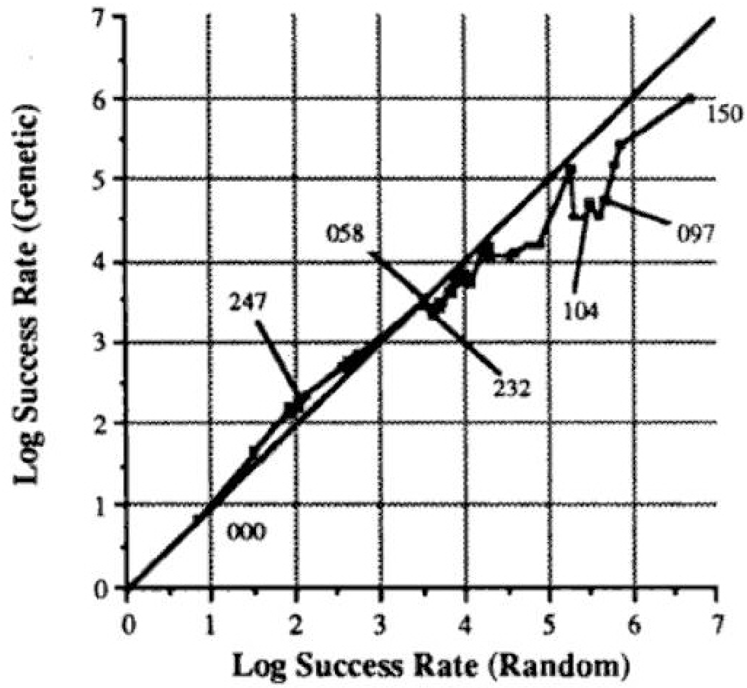
- Function set: F = {AND, OR, NAND, NOR}
- The number of internal nodes fixed to 20.
- Blind search – randomly samples 10,000,000 trees
- GP without ADFs
  - Population size $M = 50$.
  - Number of generations $G = 25$.
  - A run is terminated as soon as it produces a correct solution.
  - Total number of trees generated 10,000,000.

Results – number of times the correct function appeared in 10,000,000 generated trees:

Blind search	0
GP without ADFs	2

Even-3-Parity Function: Blind Search vs. Simple GP

Log-log scale graph of the number of trees that must be processed per correct solution for blind search vs. GP for 80 Boolean functions with three arguments.



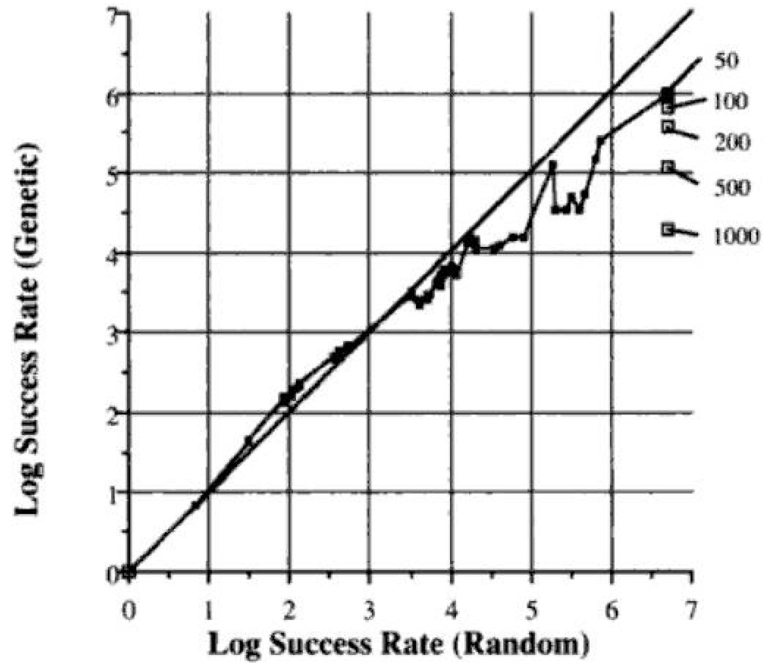
Points below the 45° line represent functions solved easily by GP than by blind search.

Even-3-Parity Function: Blind Search vs. Simple GP

Log-log scale graph of the number of trees that must be processed per correct solution for blind search vs. GP for 80 Boolean functions with three arguments.

Effect of using larger populations in GP.

- $M = 50$: 999,750 processed ind. per solution
- $M = 100$: 665,923
- $M = 200$: 379,876
- $M = 500$: 122,754
- $M = 1000$: 20,285

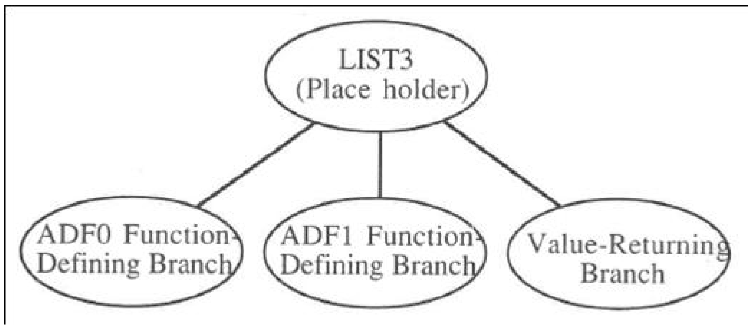


Conclusion: The performance advantage of GP over blind search increases for larger population sizes – again, it demonstrates the importance of a proper choice of the population size.

Comparison of GP without and with ADFs: Even-4-Parity Function

Common experimental setup:

- Function set: F = {AND, OR, NAND, NOR}
- Population size: 4000
- Number of generations: 51
- Number of independent runs: $60 \sim 80$



Setup of the GP with ADFs:

- ADF0 branch
  - Function set: F={AND, OR, NAND, NOR}
  - Terminal set: A2 = {ARG0, ARG1}
- ADF1 branch
  - Function set: F = {AND, OR, NAND, NOR}
  - Terminal set: A3 = {ARG0, ARG1, ARG2}
- Value-producing branch
  - Function set: F={AND, OR, NAND, NOR, ADF0, ADF1}
  - Terminal set: T4 = {D0, D1, D2, D3}

Observed GP Performance Parameters

Performance parameters:

- $P(M, i)$ – cumulative probability of success for all the generations between generation 0 and i, where M is the population size.
- $I(M, i, z)$ – number of individuals that need to be processed in order to yield a solution with probability z (here $z = 99\%$).

  For the desired probability z of finding a solution by generation i at least once in R runs the following holds

  $$z = 1 - [1 - P(M, i)]^R.$$

  Thus, the number $R(z)$ of independent runs required to satisfy the success predicate by generation i with probability $z = 1 - \varepsilon$ is

  $$R(z) = [\frac{\log \varepsilon}{\log(1 - P(M, i))}].$$

GP without ADFs: Even-4-Parity Function

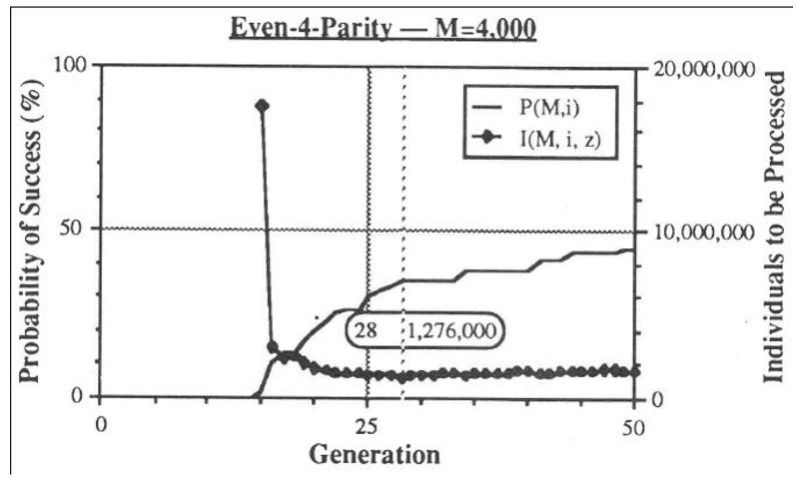
GP without ADFs is able to solve the even-3-parity function by generation 21 in all of the 66 independent runs.

GP without ADFs on even-4-parity problem (based on 60 independent runs)

- Cumulative probability of success, $P(M, i)$, is 35% and 45% by generation 28 and 50, respectively.
- The most efficient is to run GP up to the generation 28 – if the problem is run through to generation 28, processing a total of

  $4{,}000 \times 29 \text{ gener} \times 11 \text{ runs} = 1{,}276{,}000$

  individuals is sufficient to yield a solution with 99% probability.



GP without ADFs: Even-4-Parity Function

An example of solution with 149 nodes.

```
(AND (OR (OR (OR (NOR D0 (NOR D2 D1)) (NAND (OR (NOR (AND
D3 D0) D2) (NAND D0 (NOR D2 (AND D1 (OR D3 D2))))) D3))
(AND (AND D1 D2) D0)) (NAND (NAND (NAND D3 (OR (NOR D0
(NOR (OR D3 D2) D2)) (NAND (AND (AND (AND D3 D2) D3) D2)
D3))) (NAND (OR (NAND (OR D0 (OR D0 D1)) (NAND D0 D1))
D3) (NAND D1 D3))) D3)) (OR (OR (NOR (NOR (AND (OR (NOR
D3 D0) (NOR (NOR D3 (NAND (OR (NAND D2 D2) D2) D2)) (AND
D3 D2))) D1) (AND D3 D0)) (NOR D3 (OR D0 D2))) (NOR D1
(AND (OR (NOR (AND D3 D3) D2) (NAND D0 (NOR D2 (AND D1
D0)))) (OR (OR D0 D3) (NOR D0 (NAND (OR (NAND D2 D2) D2)
D2)))))) (AND (AND D2 (NAND D1 (NAND (AND D3 (NAND D1
D3)) (AND D1 D1)))) (OR D3 (OR D0 (OR D0 D1)))))).
```

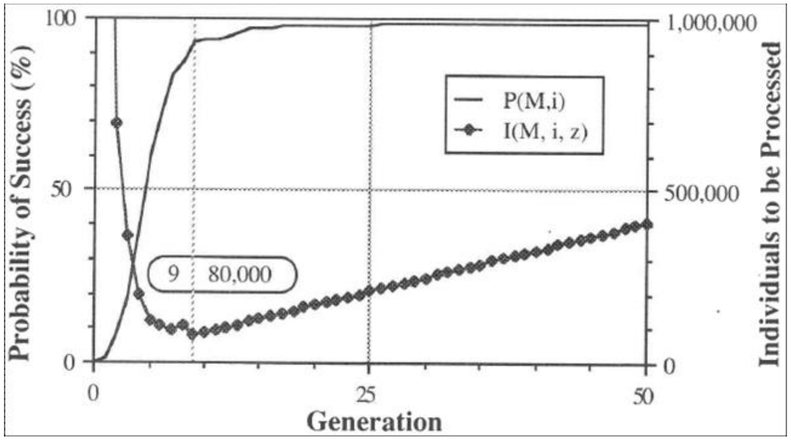
GP with ADFs: Even-4-Parity Function

GP with ADFs on even-4-parity problem (based on 168 independent runs)

- Cumulative probability of success, $P(M, i)$, is 93% and 99% by generation 9 and 50, respectively.
- If the problem is run through to generation 9, processing a total of

  $4{,}000 \times 10 \text{ gener} \times 2 \text{ runs} = 80{,}000$

  individuals is sufficient to yield a solution with 99% probability.



This is a considerable improvement in performance compared to the performance of GP without ADFs.

GP with ADFs: Even-4-Parity Function

An example of solution with 74 nodes.

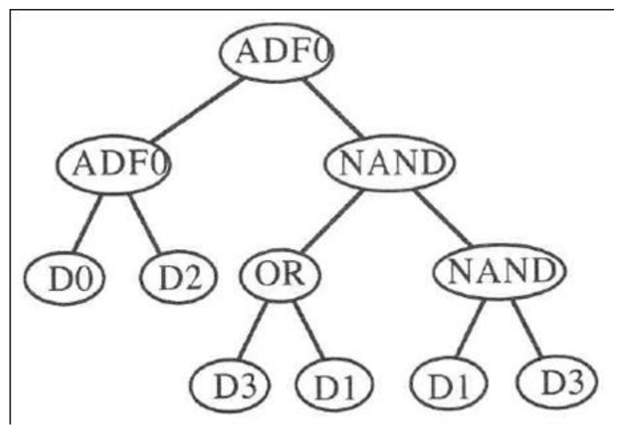
```
(LIST3 (NAND (OR (AND (NOR ARG0 ARG1) (NOR (AND ARG1
          ARG1) ARG1)) (NOR (NAND ARG0 ARG0) (NAND ARG1
          ARG1))) (NAND (NOR (NOR ARG1 ARG1) (AND (OR
          (NAND ARG0 ARG0) (NOR ARG1 ARG0)) ARG0)) (AND
          (OR ARG0 ARG0) (NOR (OR (AND (NOR ARG0 ARG1)
          (NAND ARG1 ARG1)) (NOR (NAND ARG0 ARG0) (NAND
          ARG1 ARG1))) ARG1))))
       (OR (AND ARG2 (NAND ARG0 ARG2)) (NOR ARG1 ARG1))
       (ADF0 (ADF0 D0 D2) (NAND (OR D3 D1) (NAND D1
          D3)))).
```

GP with ADFs: Even-4-Parity Function

An example of solution with 74 nodes.

```
(LIST3 (NAND (OR (AND (NOR ARG0 ARG1) (NOR (AND ARG1
          ARG1) ARG1)) (NOR (NAND ARG0 ARG0) (NAND ARG1
          ARG1))) (NAND (NOR (NOR ARG1 ARG1) (AND (OR
          (NAND ARG0 ARG0) (NOR ARG1 ARG0)) ARG0)) (AND
          (OR ARG0 ARG0) (NOR (OR (AND (NOR ARG0 ARG1)
          (NAND ARG1 ARG1)) (NOR (NAND ARG0 ARG0) (NAND
          ARG1 ARG1))) ARG1))))
       (OR (AND ARG2 (NAND ARG0 ARG2)) (NOR ARG1 ARG1))
       (ADF0 (ADF0 D0 D2) (NAND (OR D3 D1) (NAND D1
          D3)))).
```

- ADF0 defined in the first branch implements two-argument XOR function (odd-2-parity function).
- Second branch defines three-argument ADF1. It has no effect on the performance of the program since it is not called by the value-producing branch.
- VPB implements a function equivalent to
  ADF0 (ADF0 D0 D2) (EQV D3 D1)



GP with Hierarchical ADFs

Hierarchical form of automatic function definition – any function can call upon any other already-defined function.

- Hierarchy of function definitions where any function can be defined in terms of any combination of already-defined functions.
- All ADFs have the same number of dummy arguments. Not all of them have to be used in a particular function definition.
- VPB has access to all of the already defined functions.

Setup of the GP with hierarchical ADFs:

- ADF0 branch

  Functions: F={AND, OR, NAND, NOR}, Terminals: A2 = {ARG0, ARG1, ARG2}
- ADF1 branch

  Functions: F = {AND, OR, NAND, NOR, ADF0}, Terminals: A3 = {ARG0, ARG1, ARG2}
- Value-producing branch

  Functions: F={AND, OR, NAND, NOR, ADF0, ADF1}, Terminals: T4 = {D0, D1, D2, D3}

GP with Hierarchical ADFs: Even-4-Parity Function

An example of solution with 45 nodes.

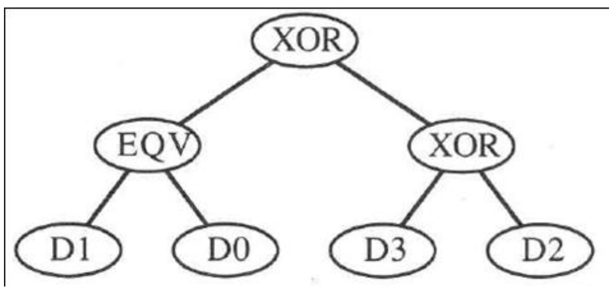
```
(LIST3 (NOR (NOR ARG2 ARG0) (AND ARG0 ARG2))
       (NAND (ADF0 ARG2 ARG2 ARG0)
             (NAND (ADF0 ARG2 ARG1 ARG2)
                   (ADF0 (OR ARG2 ARG1)
                         (NOR ARG0 ARG1)
                         (ADF0 ARG1 ARG0 ARG2))))
       (ADF0 (ADF1 D1 D3 D0)
             (NOR (OR D2 D3) (AND D3 D3))
             (ADF0 D3 D3 D2))).
```

GP with Hierarchical ADFs: Even-4-Parity Function

An example of solution with 45 nodes.

```
(LIST3 (NOR (NOR ARG2 ARG0) (AND ARG0 ARG2))
       (NAND (ADF0 ARG2 ARG2 ARG0)
             (NAND (ADF0 ARG2 ARG1 ARG2)
                   (ADF0 (OR ARG2 ARG1)
                         (NOR ARG0 ARG1)
                         (ADF0 ARG1 ARG0 ARG2))))
       (ADF0 (ADF1 D1 D3 D0)
             (NOR (OR D2 D3) (AND D3 D3))
             (ADF0 D3 D3 D2))).
```

- ADF0 defines a two-argument XOR function of variables ARG0 and ARG2 (it ignores ARG1).
- ADF1 defines a three-argument function that reduces to the two-argument equivalence function of the form (NOT (ADF0 ARG2 ARG0))
- VPB reduces to (ADF0 (ADF1 D1 D0) (ADF0 D3 D2))



Value-producing branch

Reading

- Poli, R., Langdon, W., McPhee, N.F.: *A Field Guide to Genetic Programming*, 2008, http://www.gp-field-guide.org.uk/
- Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.