Grammatical Evolution

Jiří Kubalík
Department of Cybernetics, CTU Prague

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

Contents

- GP and "closure" problem
  - Strongly-typed GP
- Grammatical Evolution
  - Representation and genotype-phenotype mapping
  - Crossover operators
  - Automatically defined functions
  - Examples: Symbolic regression, Artificial ant problem

GP and Closure Problem: Motivation Example

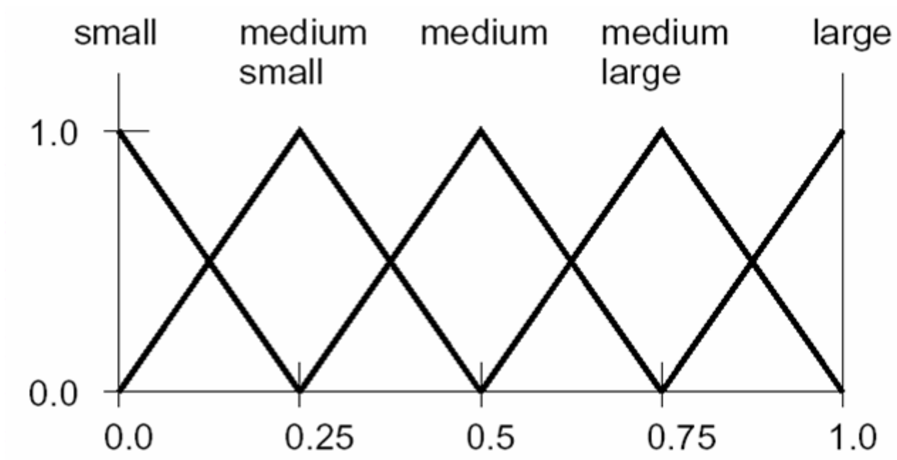
Closure - *any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal*.

Fuzzy-rule based classifier consists of fuzzy if-then rules of type

$$\text{IF}(x1 \text{ is } medium) \text{ and } (x3 \text{ is } large) \text{ THEN } class = 1 \text{ with } cf = 0.73$$

where

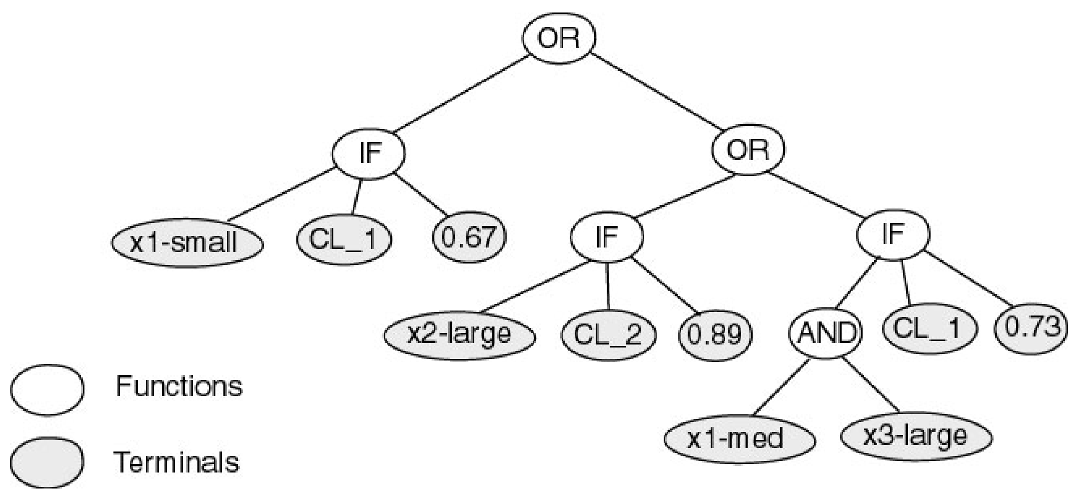
- Linguistic terms – small, medium small, medium, medium large, large,
- Fuzzy membership functions – approximate the confidence in that the crisp value is represented by the linguistic term.



GP and Closure Problem: Motivation Example

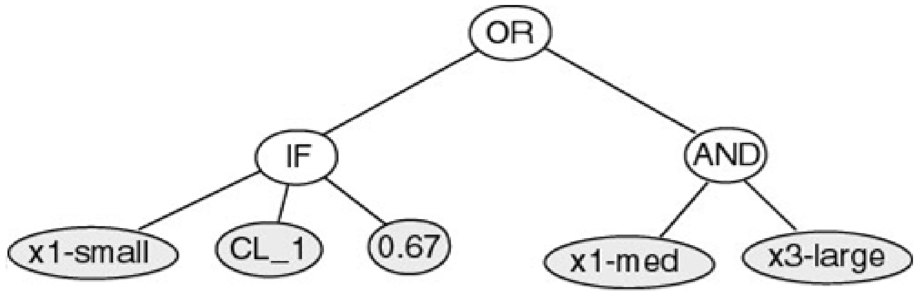
A syntactically correct tree representing a classifier as a disjunction of the three rules:

- IF$(x1$ is $small)$ THEN $class = 1$ with $cf = 0.67$,
- IF$(x2$ is $large)$ THEN $class = 2$ with $cf = 0.89$,
- IF$(x1$ is $medium)$ and $(x3$ is $large)$ THEN $class = 1$ with $cf = 0.73$.



GP and Closure Problem: Motivation Example

Using simple subtree-swapping crossover or subtree-regeneration mutation an invalid tree that does not represent a syntactically correct rule base can be generated.

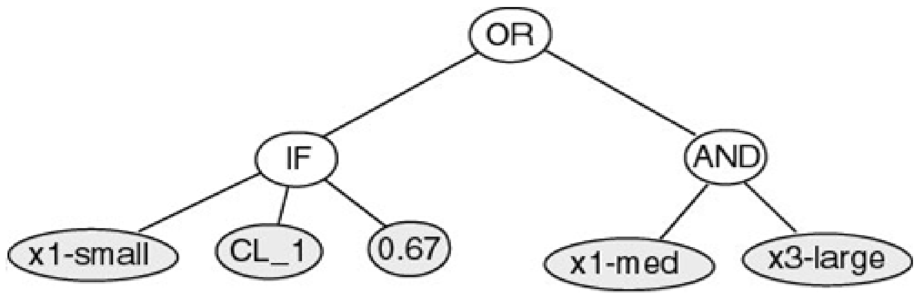


Obviously, this happens due to the fact that the **closure constraint does not hold here**.

Standard GP is not designed to handle a mixture of data types.

GP and Closure Problem: Motivation Example

Using simple subtree-swapping crossover or subtree-regeneration mutation an invalid tree that does not represent a syntactically correct rule base can be generated.



Obviously, this happens due to the fact that the **closure constraint does not hold here**.

Standard GP is not designed to handle a mixture of data types.

What can we do to get around the problem with the closure constraint?

Closure with Avoiding Multiple Data Types

Example: **Boolean** data type

- Avoid using predicates such as FOOD-AHEAD, LESS-THAN, which return Boolean values.
- Instead, use branching constructs such as IF-FOOD-AHEAD or IF-LESS-THAN, which return real values.

  Real valued data are treated as Boolean considering a subset of the reals to be "*true*" and its complement to be "*false*".

  For example, an IF-LESS-THAN construct evaluates and returns its second argument if the first argument is less than zeroand evaluates and returns its third argument otherwise.

However, sometimes there is no way to avoid introducing multiple data types, because it might not be possible to transform one data type to another.

Closure with Multiple Data Types

Dynamic typing - uses data types at program execution time to allow programs to handle data differently based on their types.

1. Functions that perform different actions with different argument types.

   Applicable when there are natural ways to cast any of the data types to any other.

   Example: data types - *real* and *complex*

   Again, often there are not natural ways to cast from one data type to another (for ex. 3D vector and 4x2 matrix).

2. Functions that signal an error when the arguments are of inconsistent type and assign an infinitely bad evaluation to this parse tree.

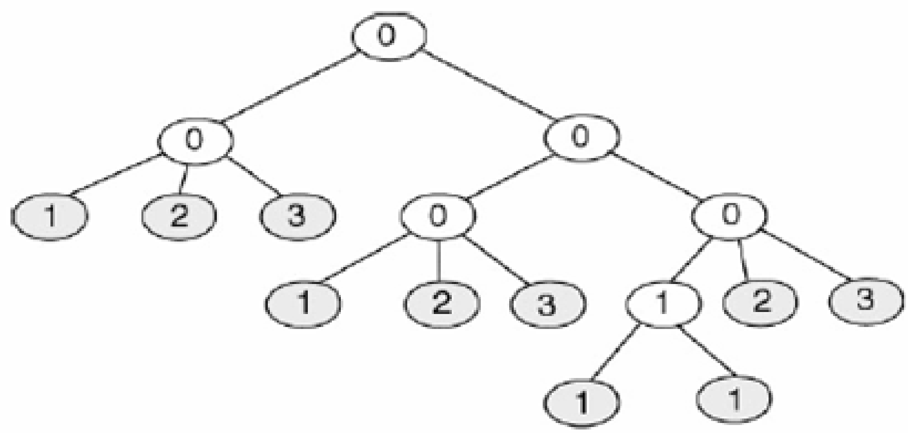
   This can be terribly inefficient, spending most of the time evaluating illegal trees.

Strongly-Typed Genetic Programming

STGP - defines syntax of evolved tree structures by specifying the data types of each argument of each non-terminal and the data types returned by each terminal and non-terminal.

- prevents generating illegal individuals,
- quite a big overhead $\Longrightarrow$ inefficient for manipulating large trees.

F / T	Output	Input
OR	0	0, 0
IF	0	1, 2, 3
AND	1	1, 1
IS	1	None
CLASS	2	None
CF	3	None



Any other elegant way to get around the closure constraint?

Grammatical Evolution

Grammatical Evolution (GE) – a grammatical-based GP system that can **evolve complete programs in an arbitrary language** using a variable-length binary/integer strings.

- The evolutionary process is performed on **variable-length binary strings**.
- A **genotype-phenotype mapping** process is used to generate programs (expression trees) in any language by using the binary strings to select production rules in a Backus-Naur form (BNF) grammar definition.
- By the use of the mapping between linear chromosomes and programs the "closure" problem is overcome, **only valid programs are generated**.

  The syntactically correct programs are evaluated by a fitness function.

Grammatical Evolution

Grammatical Evolution (GE) – a grammatical-based GP system that can **evolve complete programs in an arbitrary language** using a variable-length binary/integer strings.

- The evolutionary process is performed on **variable-length binary strings**.
- A **genotype-phenotype mapping** process is used to generate programs (expression trees) in any language by using the binary strings to select production rules in a Backus-Naur form (BNF) grammar definition.
- By the use of the mapping between linear chromosomes and programs the "closure" problem is overcome, **only valid programs are generated**.

  The syntactically correct programs are evaluated by a fitness function.

A user can specify and modify the grammar, whilst ignoring the task of designing specific genetic search operators.

Backus-Naur Form

Backus-Naur form (BNF) is a notation for expressing the grammar of a language in the form of production rules.

BNF is represented by the tuple $\{N, T, P, S\}$, where

- T is a set of terminals – items that can appear in the language ($+$, -, X, ...),
- N is a set of nonterminals – items that can be further expanded into one or more terminals or nonterminals,
- P is a set of production rules that map the elements of N to T,
- S is a start symbol that is a member of N.

Remark: Do not mistake terminals/nonterminals used in GP for terminals/nonterminals used in GP!

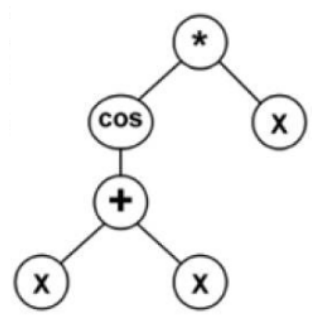
BNF: Arithmetical expressions

$N = \{expr, op, pre_{op}\}$
$T = \{sin, +, -, /, *, X, 1.0, (,)\}$
$S = \langle expr \rangle$

```
P: <expr>    ::= <expr><op><expr>     (0)
              |  (<expr><op><expr>)   (1)
              |  <pre-op>(<expr>)     (2)
              |  <var>                (3)
   <op>      ::= +                    (0)
              |  -                    (1)
              |  /                    (2)
              |  *                    (3)
   <pre-op>  ::= cos                  (0)
   <var>     ::= X                    (0)
              |  1.0                  (1)
```

Each nonterminal has one or more possible ways of expansion.

Example of a tree compliant with the BNF.



Genotype-Phenotype Mapping Process: Modulo Rule

Variable-length binary chromosomes

11011100 | 11110000 | 11011100 | . . . | 11100110

are transcribed to **codons** (a consecutive group of 8 bits encoding an integer number)

220 | 240 | 220 | . . . | 102.

The **codons are used** in a mapping function **to select an appropriate production rule** from the BNF definition to expand a given nonterminal using the following mapping function

rule=(codon integer value)
`modulo`
(number of rules for the current nonterminal)

This implies that **only syntactically correct programs can be generated!!!**

Genotype-Phenotype Mapping Process: Modulo Rule

Example: Assume that a nonterminal <op> is to be expanded, and the codon being read produces the integer 6.

There are four production rules to choose from

$$
\begin{array}{lccc}
\texttt{<op>} & ::= & + & (0) \\
 & | & \texttt{-} & (1) \\
 & | & / & (2) \\
 & | & * & (3)
\end{array}
$$

Then

$$6 \ \texttt{modulo} \ 4 = 2$$

would select rule (2).

Genotype-Phenotype Mapping Process: Wrapping

Mapping finishes when all of the nonterminals have been expanded.

During the genotype-phenotype mapping process it is possible for individuals to run out of codons and in this case so called **wrapping** is used – the chromosome is being traverse multiple times, so the codons are reused several times.

Each time the same codon is expressed

- it will always generate the same integer value,
- but depending on the current nonterminal to which it is being applied, it may result in the selection of a different production rule.

  The codon 240 can be read one time to expand the nonterminal `<expr>`, another time to expand the nonterminal `<pre-op>`, etc.

chromosome: 220 | **240** | 220 | . . . | 102.

A maximum number of wrapping events is specified – if an incomplete mapping occurs after the specified number of wrapping events, the individual is assigned the lowest possible fitness value.

- To reduce the number of invalid individuals in the population, a steady-state replacement is used.

Genotype-Phenotype Mapping Process: Example

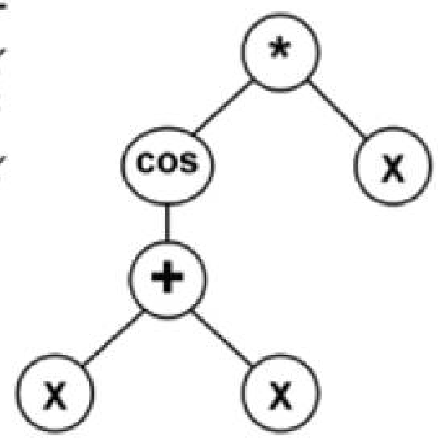
:: Grammar in Backus-Naur Form

N = {expr, op, pre-op}
T = {+, −, ∗, /, sin, cos, exp, log, X}
expr = starting symbol
P:

<expr>	::=	<expr> <op> <expr>	[0]
	|	<pre-op> <expr>	[1]
	|	X	[2]
<op>	::=	+	[0]
	|	−	[1]
	|	*	[2]
	|	/	[3]
<pre-op>	::=	sin	[0]
	|	cos	[1]
	|	exp	[2]
	|	log	[3]

:: Chromosome: 6 4 9 3 5 8 8 6 2

6 → <expr> <op> <expr>
4 → <pre-op> <expr>
9 → cos
3 → <expr> <op> <expr>
5 → x
8 → +
8 → x
6 → *
2 → x



Grammatical Evolution: Genetic Code Degeneracy

Neutral mutations – mutations that have no effect on the phenotypic fitness of an individual.

- cause diversity within equally fit individuals in the **search space**.

Adaptive mutations – mutations that change the phenotype.

- explore the **solution space**.

Degenerate genetic code – a genetic code with redundancy.

- Each codon can represent a number of distinct integer values – many of these values can represent the same production rule.

  Various genotypes can represent the same phenotype, thus facilitating the maintenance of genetic diversity within a population.

Example: Subtle changes in the genotype may have no effect on the phenotype.

A rule for operator selection:

```
<op> ::= +  (0)
       | -  (1)
       | /  (2)
       | *  (3)
```

If the current codon value were 8, then

$$8 \ \texttt{modulo} \ 4 = 2$$

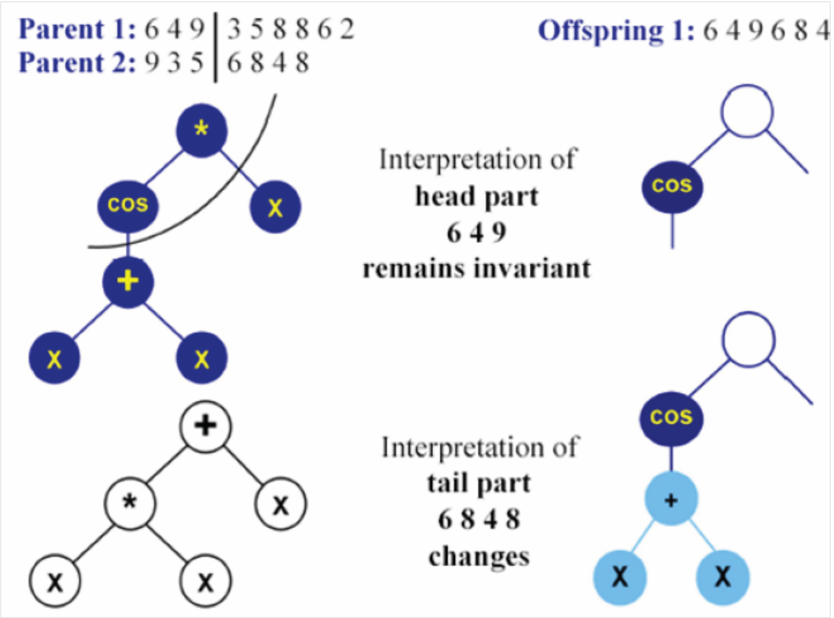
would select rule (0); the same rule that would be selected by codon value 4, 12, 16, etc.

Grammatical Evolution: 1-point Crossover

Ripple effect – a single crossover event can remove any number of subtrees to the right of the crossover point.

- It is more exploratory than the subtree crossover used in GP.
- It transmits on average half of the genetic material for each parent,
- It is equally recombinative regardless of the size of the individuals involved,

  The subtree crossover exchanges less and less genetic material as the trees are growing.
- It is less likely to get trapped in a local optimum than the subtree crossover.



The head sequence of codons does not change its meaning, while the tale sequence may or may not change its interpretation (the function of a gene depends on the genes that precede it) implying a limited exploitation capability of the recombination operation.

Grammatical Evolution: Evolutionary Algorithm

Typically, the search is carried out by an EA. However, any search method with the ability to operate over variable-length binary strings could be employed.

- Grammatical Differential Evolution,
- Grammatical Swarm.

Grammatical Evolution for Symbolic Regression

The grammar used

$$N = \{expr, op, pre_{op}\}$$
$$T = \{sin, cos, exp, log, +, -, /, *, X, 1.0, (,)\}$$
$$S = \langle expr \rangle$$

P:

`<expr>`	::=	`<expr><op><expr>`	(0)
	|	`(<expr><op><expr>)`	(1)
	|	`<pre-op>(<expr>)`	(2)
	|	`<var>`	(3)
`<op>`	::=	+	(0)
	|	-	(1)
	|	/	(2)
	|	*	(3)
`<pre-op>`	::=	sin	(0)
	|	cos	(1)
	|	exp	(2)
	|	log	(3)
`<var>`	::=	X	(0)
	|	1.0	(1)

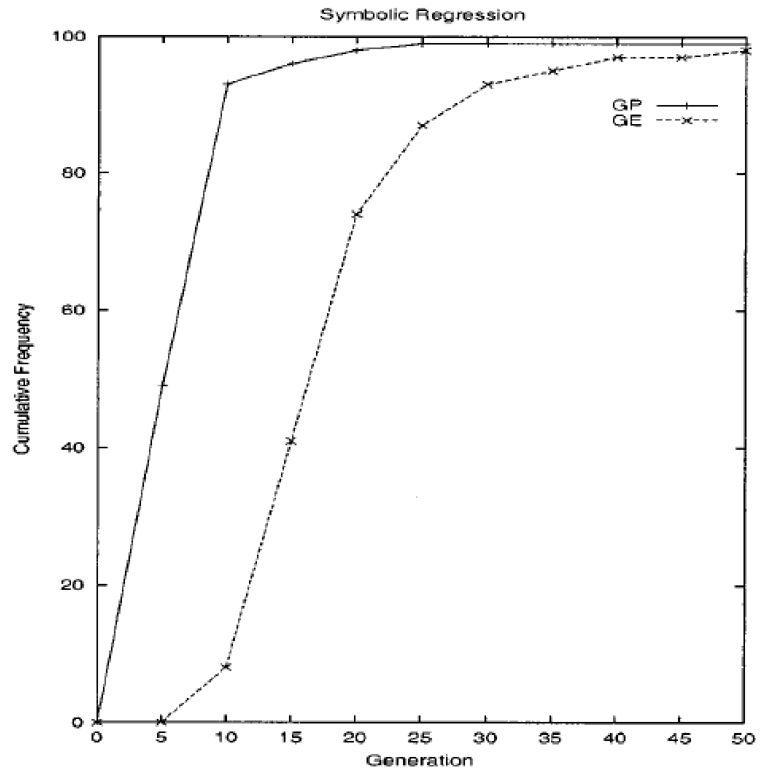
Grammatical Evolution for Symbolic Regression

Experimental setup:

Objective :	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands:	X (the independent variable), 1.0
Terminal Operators	The binary operators $+, *, /$, and $-$ The unary operators Sin, Cos, Exp and Log
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$ i.e. { -1, -.9, -.8, -.76, -.72, -.68, -.64, -.4, -.2, 0, .2, .4, .63, .72, .81, .90, .93, .96, .99, 1 }
Raw Fitness	The sum, taken over the 20 fitness cases, of the error
Standardised Fitness	Same as raw fitness
Wrapper	Standard productions to generate C functions
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01

Grammatical Evolution for Symbolic Regression

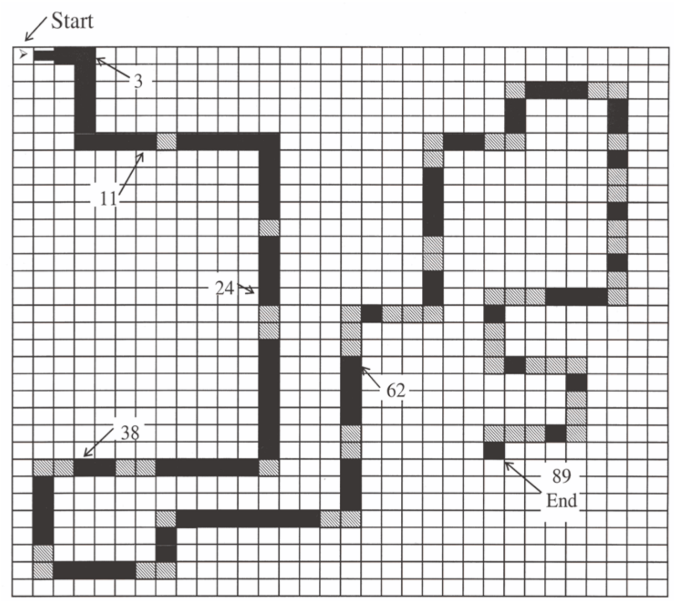
Results: GE compared to standard GP.



- GE successfully found the target function.
- GP slightly outperforms GE – this might be attributed to more "careful" initialization of the initial population in the GP.

Ant capabilities

- detection of the food right in front of him in direction he faces.
- actions observable from outside
  - `MOVE` – makes a step and eats a food piece if there is some,
  - `LEFT` – turns left,
  - `RIGHT` – turns right,
  - `NOP` – no operation.



Goal is to find a strategy that would navigate an ant through the grid so that it finds all the food pieces in the given time (600 time steps).

Santa Fe trail

- 32×32 toroidal grid with 89 food pieces.
- Obstacles: $1\times$, $2\times$ strait; $1\times$, $2\times$, $3\times$ right/left.

Grammatical Evolution for Artificial Ant Problem

The grammar used

$$N = \{code, line, if\text{-}statement, op\}$$

$$T = \{left(), right(), move(), food\_ahead, else, if, \{, \}, (,), ;\}$$

$$S = \langle code \rangle$$

P:

<code>	::=	<line>	(0)
	|	<code><line>	(1)
<line>	|	<if-statement>	(0)
	|	<op>	(1)
<if-statement>	::=	if (food_ahead()) {<line>} else {<line>}	(0)
<op>	::=	left()	(0)
	|	right()	(1)
	|	move()	(2).

Grammatical Evolution for Artificial Ant Problem

Experimental setup:

Objective :	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators:	left(), right(), move(), food_ahead()
Terminal Operands:	None
Fitness cases	One fitness case
Raw Fitness	Number of pieces of food before the ant times out with 600 operations.
Standardised Fitness	Total number of pieces of food less the raw fitness.
Wrapper	None
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01

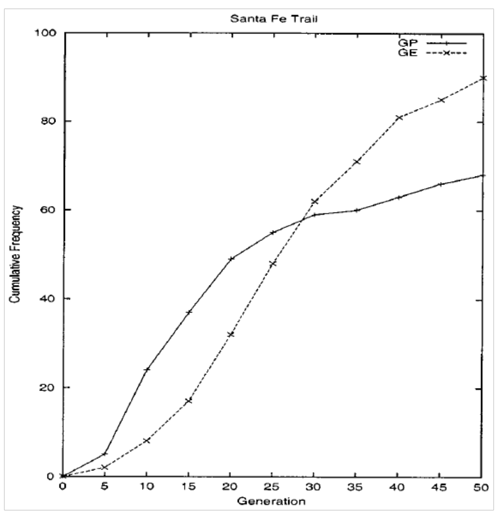
Grammatical Evolution for Artificial Ant Problem

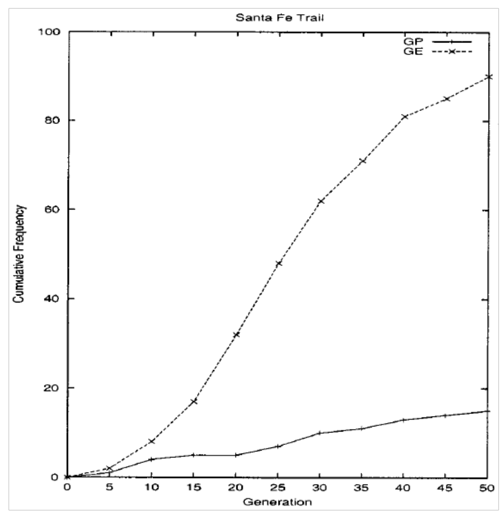
GE was successful at finding a solution to the Santa Fe trail.

- Solutions have a form of a multiline code.

GE outperforms GP:

- The left side figure shows the performance of the GP using solution length and the number of food pieces eaten in the fitness.
- The right side figure shows the performance of the GP using just the number of food pieces eaten in the fitness measure.






```
move();
left();
if(food_ahead())
  left();
else
  right();
right();
if(food_ahead())
  move();
else
  left();
```

Each solution is executed in a loop until the number of time steps allowed is reached.

Grammatical Evolution and Automatically Defined Functions

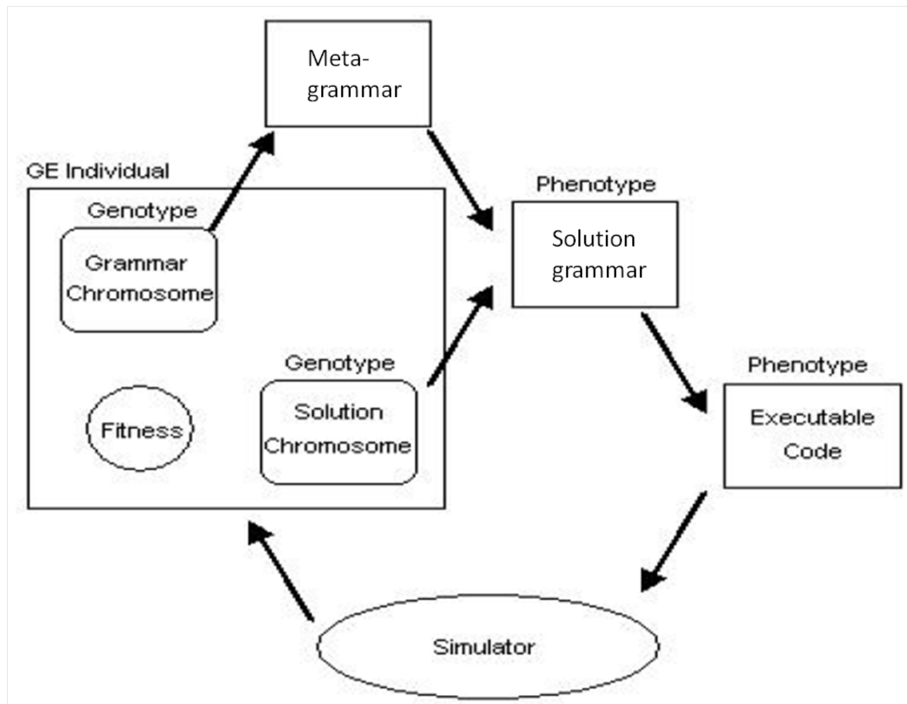
Three approaches illustrating the possibilities to adopt the principles of automatically capturing modularity from GP

1. Grammatical Evolution by Grammatical Evolution or meta-Grammar GE $(GE)^2$
   - the input grammar is used to specify the construction of another syntactically correct grammar, which is then used in a mapping process to construct a solution.
2. GE grammar with the ability to define one ADF.
3. GE grammar with the ability to define any number of ADFs.

Grammatical Evolution by Grammatical Evolution

$(GE)^2$ is a variant of GE which evolves the input grammar itself, thus it has an ability to automatically define and evolve a number of ADFs.

- The **meta grammar** dictates the construction of the **solution grammar**.
- Two separate, variable-length binary chromosomes are used
  - Solution grammar chromos. – to generate the solution grammar from the meta grammar.
  - Solution structure chromosome – to generate the solution itself.
- Crossover operates between homologous chromosomes.
- The solution grammar and the structure of the solution are evolved simultaneously.



Grammatical Evolution by Grammatical Evolution

Example: Meta grammar for the artificial ant problem.

Main program

ADFs' definitions

```
<g> ::=
       <def_fun_u>
       "<prog>      ::= public Test() { while(get_Energy_Left()) { <code>} } "
       "<code>      ::= <line> | <code> <line>"
       "<line>      ::= <condition> | <op>"
       "<condition> ::= if (food_ahead()==1) { <line> } else { <line>}"
       "<op>        ::=  left(); | right(); | move(); | adf*();"
<def_fun_u>    ::= <def_fun_s> | <def_fun_u> <def_fun_s>
<def_fun_s>    ::= "public void adf*() {" <adfcode> "}"
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition> | <adfop>
<adfcondition> ::= if (food_ahead()==1) { <adfline> } else { <adfline> }
<adfop>        ::= left(); | right(); | move();
```

- $adf \star ()$ is a function call to a defined function.

  A codon from the solution chromosome is used to select which function is called.

In a solution grammar the multiple defined ADFs are post-proceed to make each function signature unique.

GE Grammar with one ADF

Main program definition

```
<prog>         ::= "public Ant() { while(get_Energy() > 0) {"<code>"}} "
                   "public void adf0() {"<adfcode>"}"
<code>         ::= <line> | <code> <line>
<line>         ::= <condition>  | <op>
<condition>    ::= if (food_ahead()==1) {<line>} else {<line>}
<op>           ::=  left(); | right(); | move(); | adf0();
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition>  | <adfop>
<adfcondition> ::= if (food_ahead()==1) {<adfline>} else {<adfline>}
<adfop>        ::= left(); | right(); | move();
```

ADF’s definition

A header of a single ADF, `adf0()`, is defined so only a body of this particular ADF can be generated using the chromosome.

GE Grammar with Multiple ADFs

Main program definition

```
<prog>          ::= "public Ant() { while(get_Energy() > 0) {"<code>"} }"<adfs>
<adfs>          ::= <adf_def>  | <adf_def> <adfs>
<adf_def>       ::= " public void adf*() {"<adfcode>"}"
<code>          ::= <line> | <code> <line>
<line>          ::=  <condition>  | <op>
<condition>     ::= "if(food_ahead()==1) {"<line>"} else {"<line>"}"
<op>            ::=  adf*();
<adfcode>       ::= <adfline> | <adfcode> <adfline>
<adfline>       ::= <adfcondition>  | <adfop>
<adfcondition> ::= "if (food_ahead()==1) {"<adfline>"} else {"<adfline>"}"
<adfop>         ::= left(); | right(); | move();
```

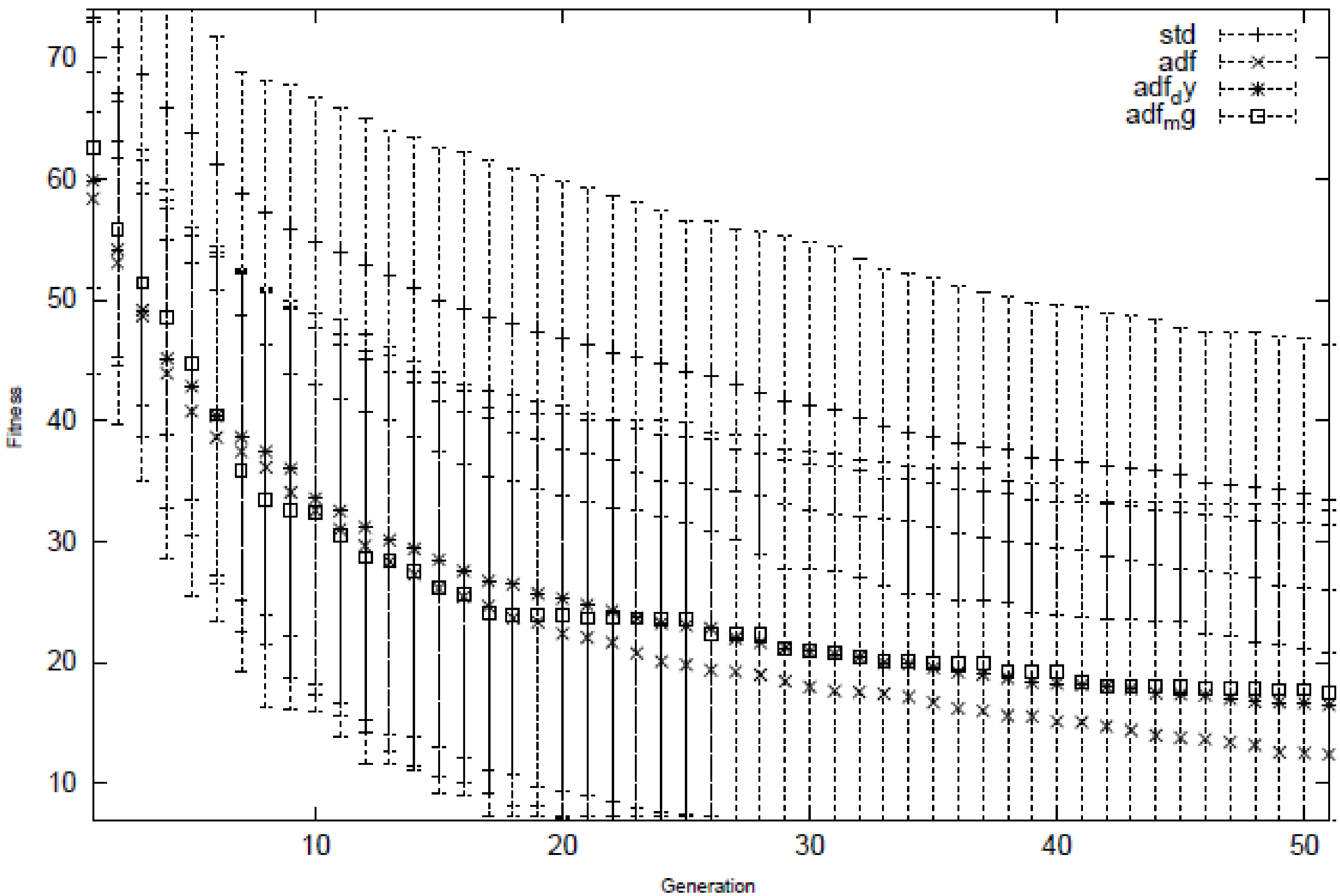
ADFs' definitions

A number of ADFs can be generated via the non-terminal `<adfs>` using the chromosome.

`adf()` is expanded to enumerate all the allowed ADFs.

GE with ADFs: Results on Santa Fe Ant Trail

Irrespective of the ADF representation, the presence of ADFs alone is sufficient to significantly improve performance of the GE.



Reading

- Poli, R., Langdon, W., McPhee, N.F.: *A Field Guide to Genetic Programming*, 2008
  http://www.gp-field-guide.org.uk/
- O'Neill, M., Ryan, C.: Grammatical Evolution. IEEE Transactions on Evolutionary Computation, VOL. 5, NO. 4, AUGUST 2001
  http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=942529
- Grammatical-Evolution.org
  http://www.grammatical-evolution.org/pubs.html