CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering
Department of Cybernetics

A0M33EOA
Grammatical Evolution. Cartesian GP.

Petr Pošík

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Heavilly using slides from **Jiří Kubalík**, CIIRC CTU, with permission.

Introduction

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
- Cartesian Genetic Programming
  - Representation
  - Genotype-phenotype mapping
  - Examples: Design of boolean Circuits

GP and Closure Problem: Motivation Example

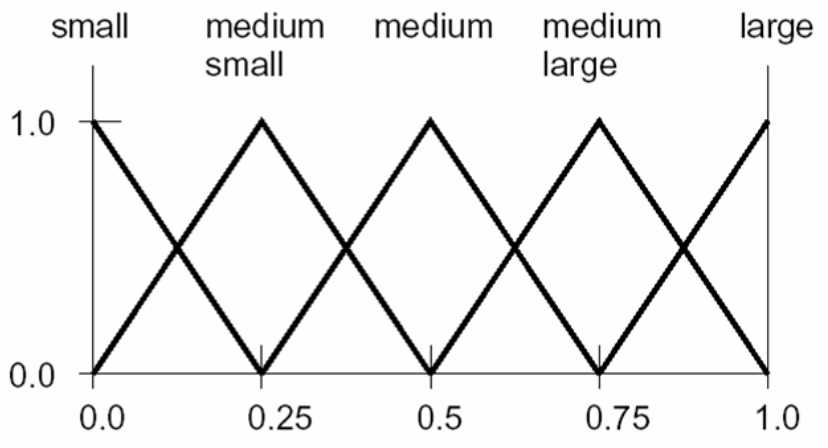
Closure property: any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal.

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

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

Fuzzy rules use

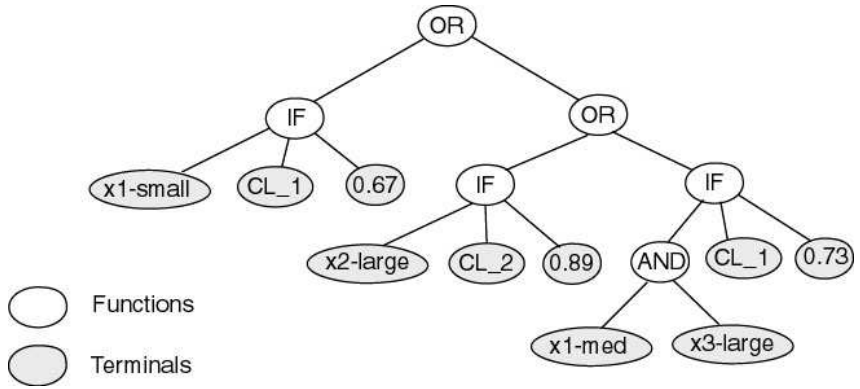
- linguistic terms: small, medium small, medium, medium large, large,
- fuzzy membership functions: approximate the confidence in that the crisp value is represented by the linguistic term.



GP and Closure Problem: Motivation Example (cont.)

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

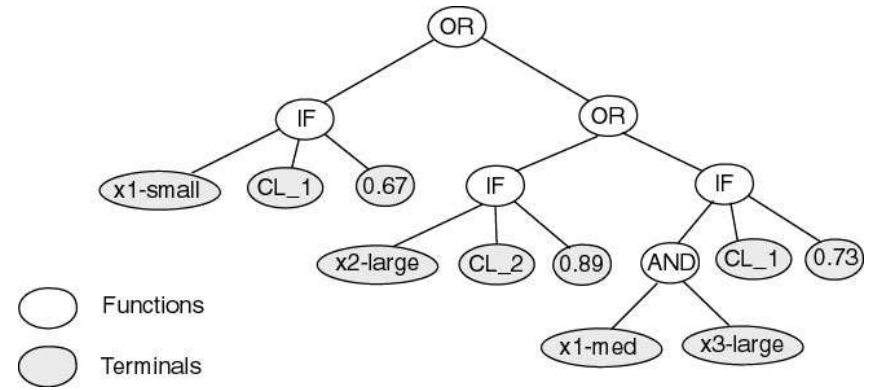
- IF$(x1$ is $small)$ THEN $class = 1$ with $cf = 0.67$,
- IF$(x2$ is $large)$ THEN $class = 2$ with $cf = 0.89$,
- IF$(x1$ is $medium)$ and $(x3$ is $large)$ THEN $class = 1$ with $cf = 0.73$.



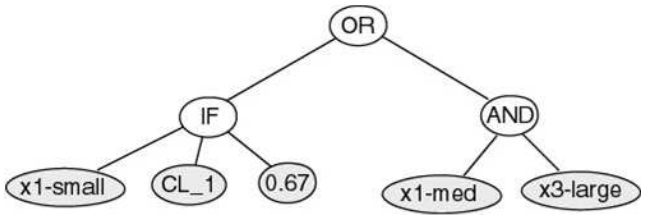
GP and Closure Problem: Motivation Example (cont.)

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 $cf = 0.73$.



Subtree crossover or subtree mutation can produce an invalid tree (syntactically incorrect rule base).



Closure property does not hold here and standard GP is not designed to handle a mixture of data types.

How to get around the closure constraint?

Several options:

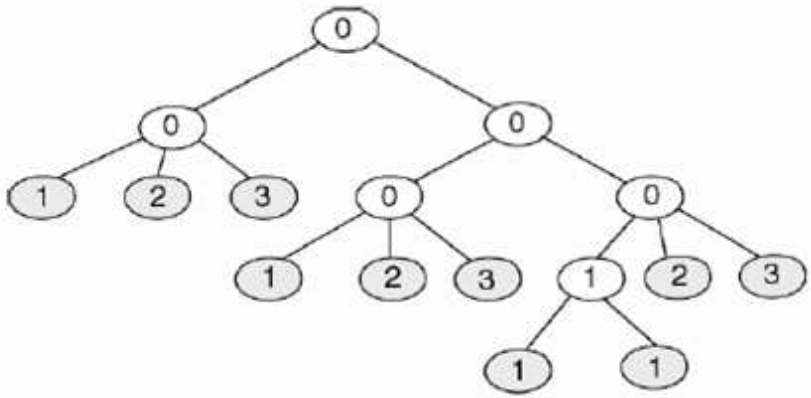
- Embed one data type into another, i.e., actually use only a single data type. This is not always possible.
- Use a different kind of GP system that allows multiple data types:
  - Strongly-typed GP
  - Grammatical Evolution
  - ...

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.

- It 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

Grammatical Evolution (GE) [RCO98, OR01]:

- A grammar-based GP system that can *evolve complete programs in an arbitrary language*.
- The evolutionary process is performed on *variable-length binary/integer strings*.
- A *genotype-phenotype mapping*
  - uses the numbers from chromosome
  - to select production rules from a grammar in **Backus-Naur form (BNF)**, and
  - generates a program (expression tree) in a language described by the grammar.
- The "closure problem" is solved by **generating only valid programs**.
- The user specifies the grammar; no need to design any specific genetic search operators.

[OR01] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[RCO98] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 1998. Springer-Verlag.

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 a tuple $\{T, N, P, S\}$, where

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

Remark: Do not confuse (non)terminals used in GE and (non)terminals used in GP!

BNF: Arithmetical expressions

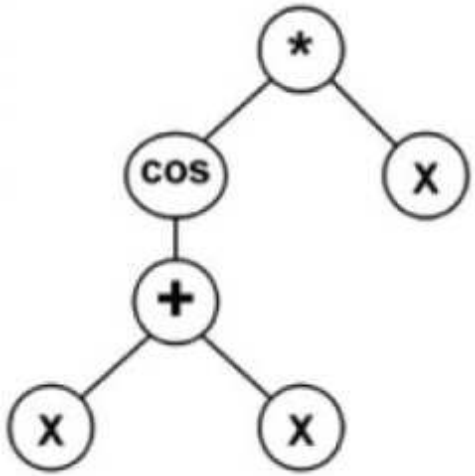
$N = \{$<expr>, <op>, <pre-op>, <var>$\}$
$T = \{$cos, +, -, /, *, X, 1.0$\}$
$S =$ <expr>
$P = \{$

```
    <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

Mapping variable-length binary chromosomes into programs using a grammar:

1. Transform the binary chromosome into a list of integers (codons).
2. Set *program* to *start symbol S* of the grammar.
3. While *program* contains any nonterminal:
4. Let *s* be the first nonterminal in *program*.
5. Read the next *codon c*.
6. Use *c* to choose a particular rule from the available production rules for symbol *s*.
7. Replace symbol *s* in *program* with the symbol expansion (the RHS of the rule).
8. Return *program*.

Genotype-Phenotype Mapping Process: Modulo Rule

- Variable-length binary chromosomes

  `11011100|11110000|11011100|...|11100110`

  are transcribed into **codons** (each group of 8 bits encodes an integer number)

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

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

  chosen rule = (codon 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

- Variable-length binary chromosomes

  `11011100|11110000|11011100|...|11100110`

  are transcribed into **codons** (each group of 8 bits encodes an integer number)

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

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

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

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

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

There are 4 production rules for `<op>`:

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

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.
- If the mapping process *runs out of codons*, **wrapping** is used: the chromosome is traversed from the beginning again. The codons may be reused several times.
- Each time the same codon is expressed
  - it represents the same integer value, but
  - it may select a different production rule depending on the nonterminal being expanded.
  - E.g., codon 240 can be read one time to expand nonterminal `<expr>`, another time to expand 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.

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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

S: `<expr>`

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$<expr>, <op>, <pre-op>$\}$
$T = \{$+, -, *, /, sin, cos, exp, log, X, (,)$\}$
$S =$ <expr>

$P = \{$

<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

<expr> $\longrightarrow$ S: 6 mod 3 = 0 $\longrightarrow$ <expr><op><expr>

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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

	S:	
		`<expr>`
`<expr>`	$\longrightarrow$ 6 mod 3 = 0 $\longrightarrow$	`<expr><op><expr>`
`<expr><op><expr>`	$\longrightarrow$ 4 mod 3 = 1 $\longrightarrow$	`<pre-op>(<expr>)<op><expr>`

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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

	S:		
			`<expr>`
`<expr>`	$\longrightarrow$ 6 mod 3 = 0	$\longrightarrow$	`<expr><op><expr>`
`<expr><op><expr>`	$\longrightarrow$ 4 mod 3 = 1	$\longrightarrow$	`<pre-op>(<expr>)<op><expr>`
`<pre-op>(<expr>)<op><expr>`	$\longrightarrow$ 9 mod 4 = 1	$\longrightarrow$	`cos(<expr>)<op><expr>`

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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

	S:	`<expr>`
`<expr>`	$\longrightarrow$ 6 mod 3 = 0 $\longrightarrow$	`<expr><op><expr>`
`<expr><op><expr>`	$\longrightarrow$ 4 mod 3 = 1 $\longrightarrow$	`<pre-op>(<expr>)<op><expr>`
`<pre-op>(<expr>)<op><expr>`	$\longrightarrow$ 9 mod 4 = 1 $\longrightarrow$	`cos(<expr>)<op><expr>`
`cos(<expr>)<op><expr>`	$\longrightarrow$ 3 mod 3 = 0 $\longrightarrow$	`cos(<expr><op><expr>)<op><expr>`

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$<expr>, <op>, <pre-op>$\}$
$T = \{$+, -, *, /, sin, cos, exp, log, X, (,)$\}$
$S =$ <expr>

$P = \{$

```
<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

	S:	<expr>
<expr>	$\longrightarrow$ 6 mod 3 = 0 $\longrightarrow$	<expr><op><expr>
<expr><op><expr>	$\longrightarrow$ 4 mod 3 = 1 $\longrightarrow$	<pre-op>(<expr>)<op><expr>
<pre-op>(<expr>)<op><expr>	$\longrightarrow$ 9 mod 4 = 1 $\longrightarrow$	cos(<expr>)<op><expr>
cos(<expr>)<op><expr>	$\longrightarrow$ 3 mod 3 = 0 $\longrightarrow$	cos(<expr><op><expr>)<op><expr>
cos(<expr><op><expr>)<op><expr>	$\longrightarrow$ 5 mod 3 = 2 $\longrightarrow$	cos(X<op><expr>)<op><expr>
cos(X<op><expr>)<op><expr>	$\longrightarrow$ 8 mod 4 = 0 $\longrightarrow$	cos(X+<expr>)<op><expr>
cos(X+<expr>)<op><expr>	$\longrightarrow$ 8 mod 3 = 2 $\longrightarrow$	cos(X+X)<op><expr>
cos(X+X)<op><expr>	$\longrightarrow$ 6 mod 4 = 2 $\longrightarrow$	cos(X+X)*<expr>
cos(X+X)*<expr>	$\longrightarrow$ 2 mod 3 = 2 $\longrightarrow$	cos(X+X)*X

Genotype-Phenotype Mapping Process: Example

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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

		S:		`<expr>`
`<expr>`	$\longrightarrow$	$6 \bmod 3 = 0$	$\longrightarrow$	`<expr><op><expr>`
`<expr><op><expr>`	$\longrightarrow$	$4 \bmod 3 = 1$	$\longrightarrow$	`<pre-op>(<expr>)<op><expr>`
`<pre-op>(<expr>)<op><expr>`	$\longrightarrow$	$9 \bmod 4 = 1$	$\longrightarrow$	`cos(<expr>)<op><expr>`
`cos(<expr>)<op><expr>`	$\longrightarrow$	$3 \bmod 3 = 0$	$\longrightarrow$	`cos(<expr><op><expr>)<op><expr>`
`cos(<expr><op><expr>)<op><expr>`	$\longrightarrow$	$5 \bmod 3 = 2$	$\longrightarrow$	`cos(X<op><expr>)<op><expr>`
`cos(X<op><expr>)<op><expr>`	$\longrightarrow$	$8 \bmod 4 = 0$	$\longrightarrow$	`cos(X+<expr>)<op><expr>`
`cos(X+<expr>)<op><expr>`	$\longrightarrow$	$8 \bmod 3 = 2$	$\longrightarrow$	`cos(X+X)<op><expr>`
`cos(X+X)<op><expr>`	$\longrightarrow$	$6 \bmod 4 = 2$	$\longrightarrow$	`cos(X+X)*<expr>`
`cos(X+X)*<expr>`	$\longrightarrow$	$2 \bmod 3 = 2$	$\longrightarrow$	`cos(X+X)*X`

The resulting "program": $x \cos(2x)$

Quiz: GE Mapping

Grammar in BNF:

$N = \{$`<expr>`, `<op>`, `<pre-op>`$\}$
$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `(`, `)`$\}$
$S =$ `<expr>`

$P = \{$

`<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)

$\}$

Given the above grammar, which of the following codon sequences represents an incomplete mapping?

A 2 3

B 4 5 5

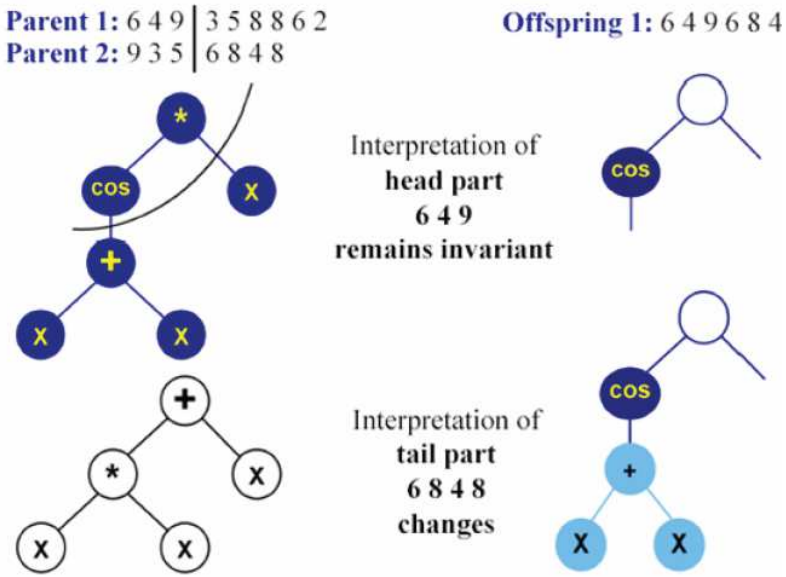
C 0 2

D 1

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

Grammar used by GE for SR:

$N = \{$`<expr>`, `<op>`, `<pre-op>`, `<var>`$\}$

$T = \{$`+`, `-`, `*`, `/`, `sin`, `cos`, `exp`, `log`, `X`, `1.0`, `(`, `)`$\}$

$S =$ `<expr>`

$P = \{$

`<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)

$\}$

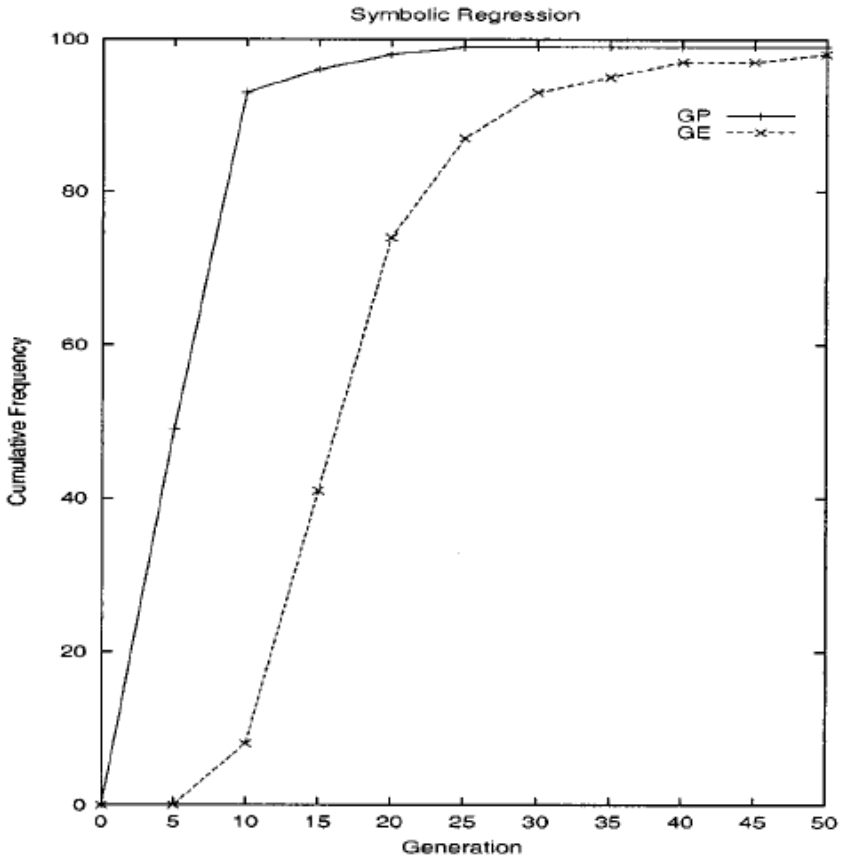
Grammatical Evolution for Symbolic Regression

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.

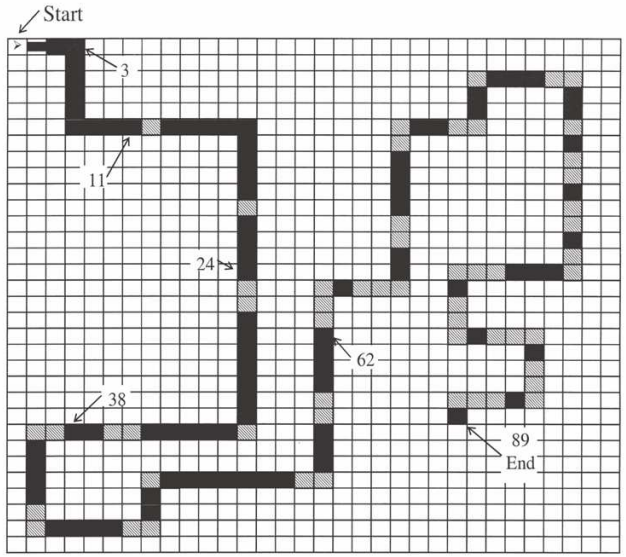


- GE successfully found the target function.
- GP outperforms GE: this might be attributed to more ”careful” initialization of the initial population in the GP.

Grammatical Evolution for Artificial Ant Problem

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: find a strategy that navigates 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 by GE for Artificial Ant:

$N = \{$`<code>`, `<line>`, `<if-statement>`, `<action>`$\}$

$T = \{$`left()`, `right()`, `move()`, `food_ahead()`, `if`, `else`, `{`, `}`, `(`, `)`, `;`$\}$

$S =$ `<code>`

$P = \{$

```
          <code> ::= <line>                    (0)
                   | <code><line>              (1)
          <line> ::= <if-statement>            (0)
                   | <action>                  (1)
 <if-statement> ::= if (food_ahead()) {        (0)
                      <line>
                    } else {
                      <line>
                    }
        action ::= 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.

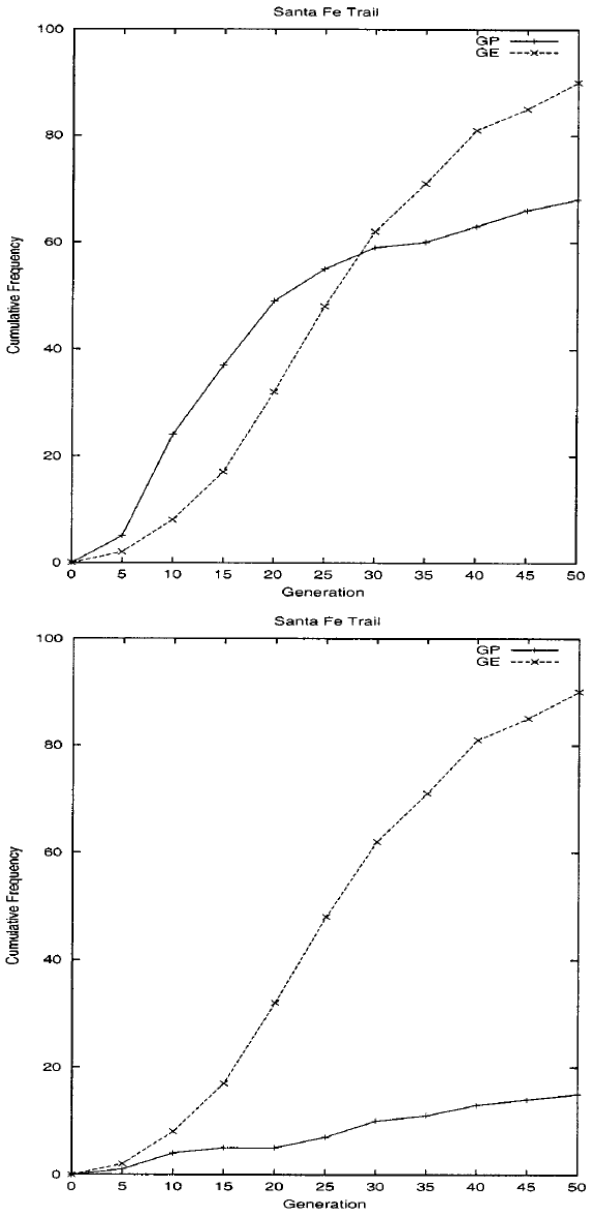
- Solutions have a form of a multiline code:

```
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.

GE outperforms GP:

- The top figure shows the performance of GP using a solution length constraint component in the fitness measure.
- The bottom figure shows the performance of GP without the constraint on the solution length.



Grammatical Evolution and Automatically Defined Functions

Many options to employ ADFs in GE, e.g.:

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.

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();
```

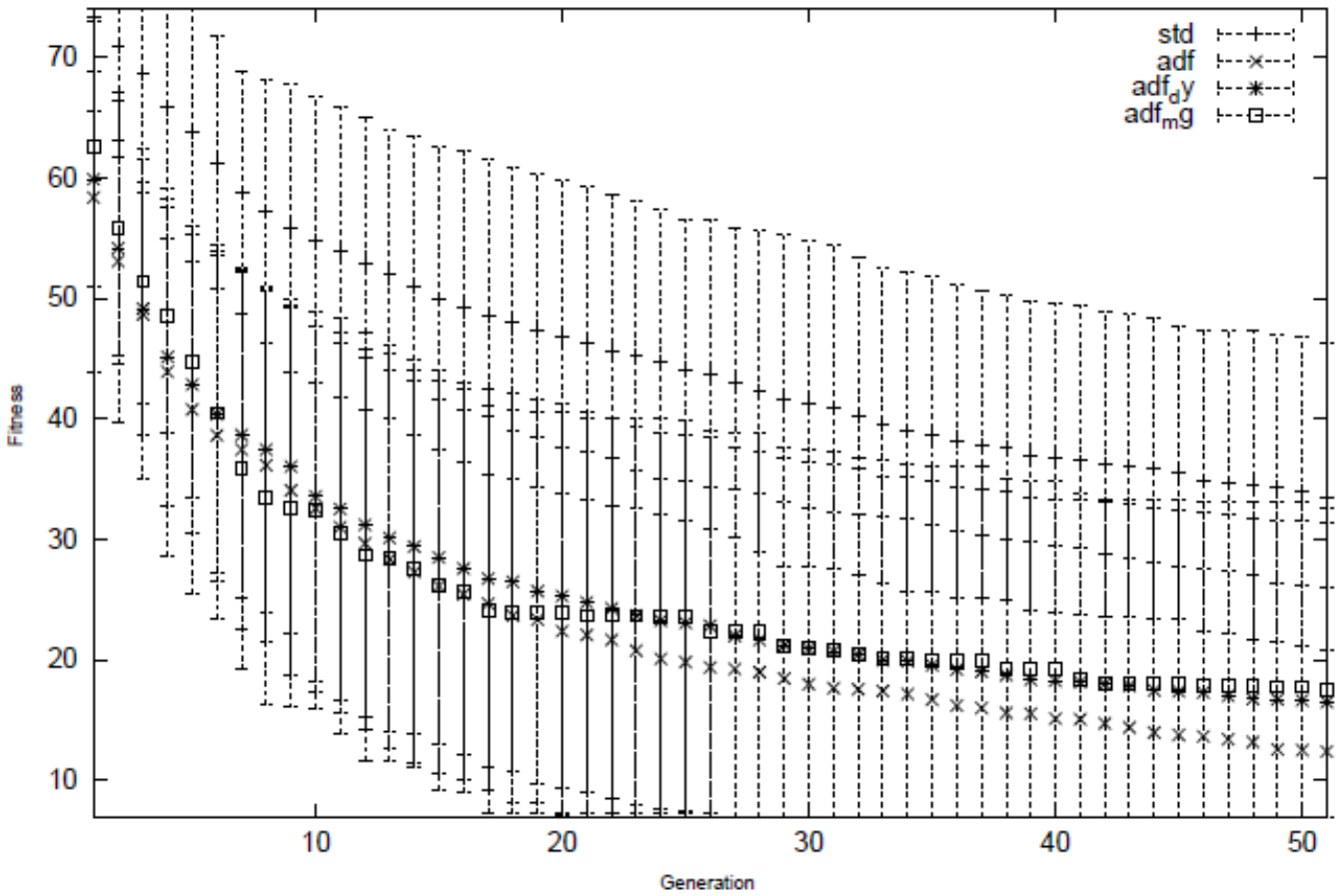
ADFs' definitions

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.



Cartesian Genetic Programming

Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [Mil11]

- A GP technique evolving *programs in the form of directed graphs*.
- The genotype is a list of integers that represent the program primitives and their connections.
- The genotype usually contains **many non-coding genes**.
- The genes are
  - addresses in data (connection genes),
  - addresses in a look up table of functions.
- The representation is very simple, flexible and convenient for many problems.

[Mil11] Julian Francis Miller, editor. *Cartesian Genetic Programming*. Springer, 2011.

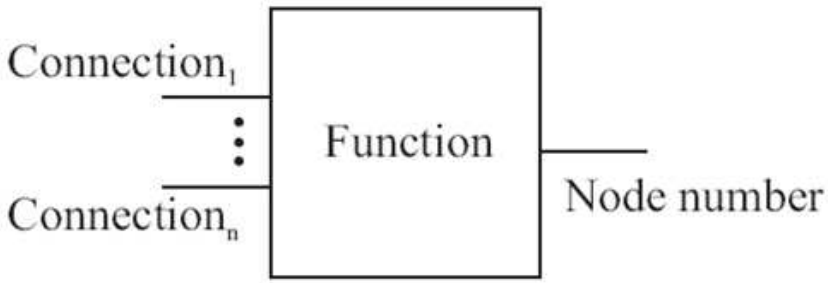
CGP Node

CGP program is a set of interconnected nodes.

A CGP node contains

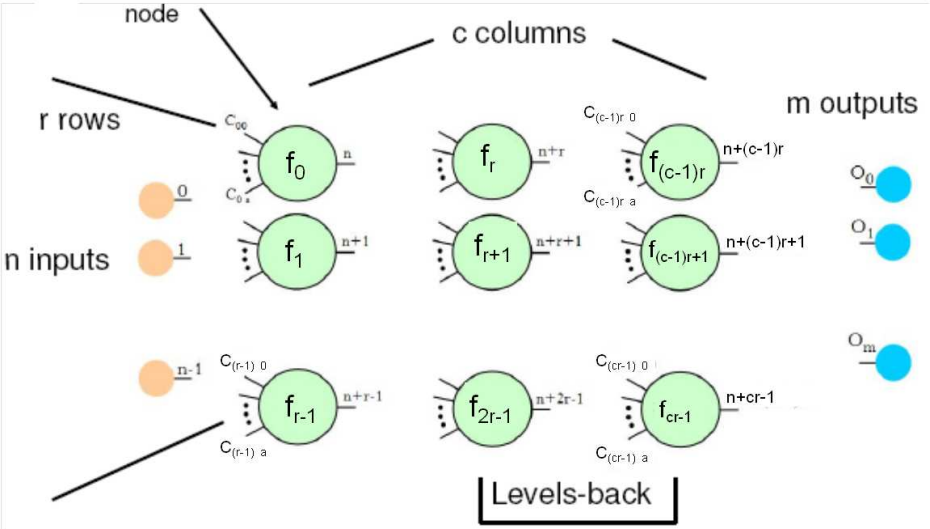
- **function symbol** (specifies the operation performed by the node), and
- **connections** (pointers toward nodes providing input for the function of the node).

Each CGP node has an output with its unique number assigned that may be used as an input for another node.



CGP General Form

CGP is Cartesian in the sense that the graph nodes are placed in **Cartesian coord. system**



Each CGP program is defined by

- number of rows r,
- number of columns c,
- number of inputs n,
- number of outputs m,
- number of functions f,
- maximum arity of function a,
- nodes interconnectivity l.

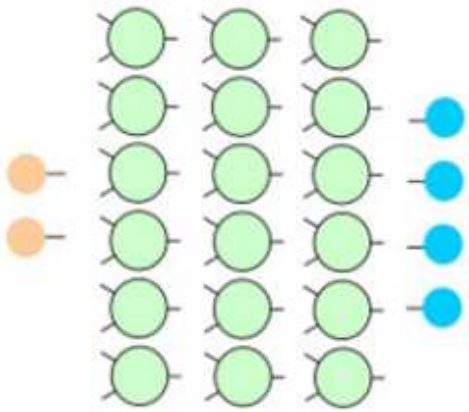
Nodes in the same column are not allowed to be connected to each other.

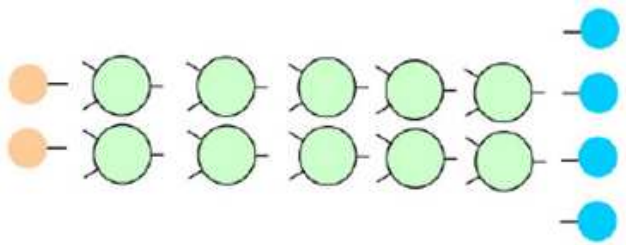
The **nodes interconnectivity** defines the maximum distance (in terms of the number of columns) between two connected nodes.

- If equal to 1, each node can be connected only with nodes in the previous column.
- If equal to c, each node can be connected to any other node in the previous columns.

CGP: Variety of Graphs

Depending on r, c and l a wide range of graphs can be generated.

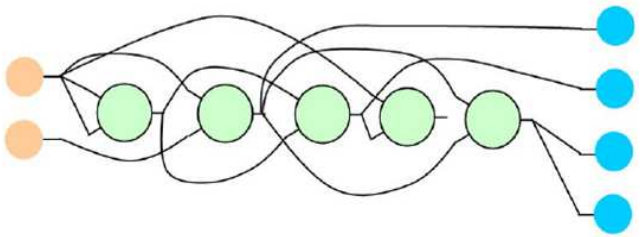




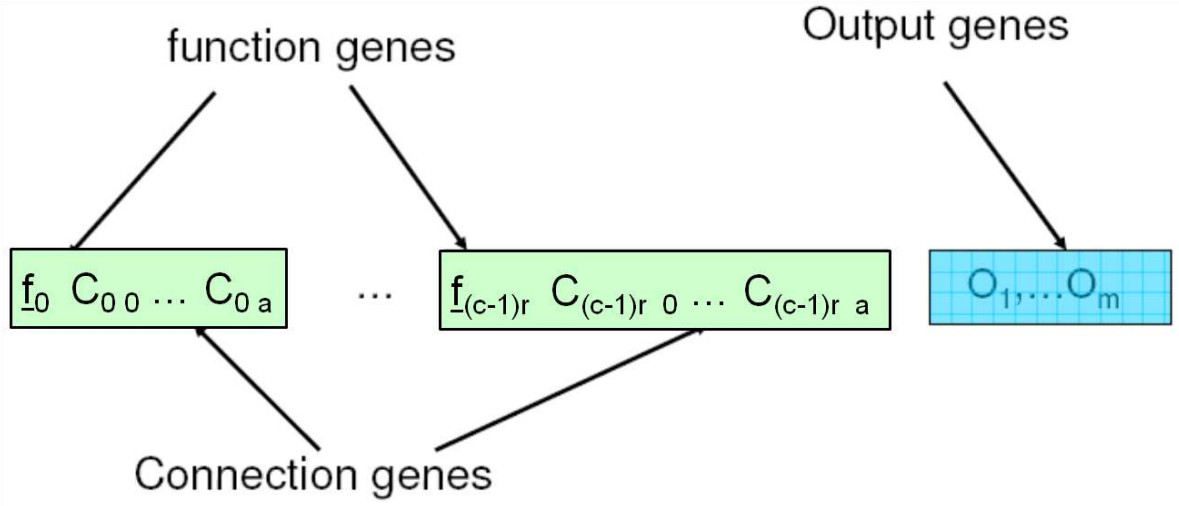
The length of the genotype (i.e. the maximum size of the CGP program) is fixed, however the **actual size and structure of the program can vary**.

The most general choice is $r = 1$ and $l = c$:

- Arbitrary directed graphs can be created with a maximum depth.
- Suitable when no prior knowledge about the solution is available.



CGP Genotype



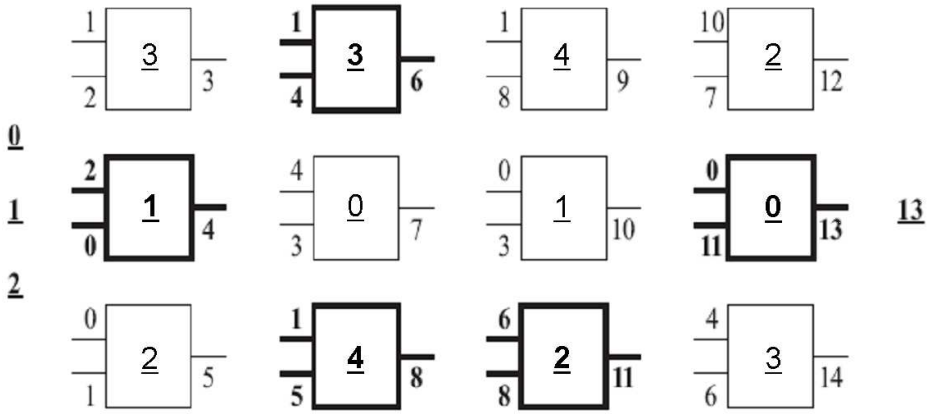
Usually, all functions have as many inputs as the maximum function arity.

Unused connections are ignored.

CGP Program Example

CGP program with 3×4 architecture, 3 inputs and 1 output.

- number of rows $r = 3$
- number of columns $c = 4$
- number of inputs $n = 3$
- number of outputs $m = 1$
- number of functions $f = 5$
- maximum arity of function $a = 2$



Look up table of 5 functions:

0	+	Add the arg1 to arg2
1	-	Subtract arg2 from arg1
2	*	Multiply arg1 to arg2
3	/	Divide arg1 by arg2
4	`sin`	Calculate sin of arg1

CGP chromosome:

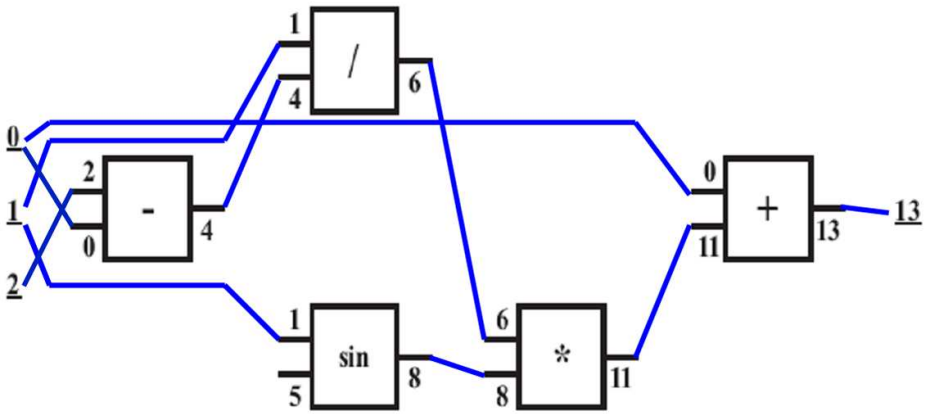
$$C = (\underbrace{\underline{3},1,2}_{3},\underbrace{\underline{\mathbf{1}},\mathbf{2},\mathbf{0}}_{4},\underbrace{\underline{2},0,1}_{5},\underbrace{\underline{\mathbf{3}},\mathbf{1},\mathbf{4}}_{6},\underbrace{\underline{0},4,3}_{7},\underbrace{\underline{\mathbf{4}},\mathbf{1},\mathbf{5}}_{8},\underbrace{\underline{4},1,8}_{9},\underbrace{\underline{1},0,3}_{10},\underbrace{\underline{\mathbf{2}},\mathbf{6},\mathbf{8}}_{11},\underbrace{\underline{2},10,7}_{12},\underbrace{\underline{\mathbf{0}},\mathbf{0},\mathbf{11}}_{13},\underbrace{\underline{3},4,6}_{14},13)$$

Its length: $rc(1+2)+m = 3 \cdot 4 \cdot (1+2)+1 = 37$

CGP Program Example

CGP program with 3×4 architecture, 3 inputs and 1 output.

- number of rows $r = 3$
- number of columns $c = 4$
- number of inputs $n = 3$
- number of outputs $m = 1$
- number of functions $f = 5$
- maximum arity of function $a = 2$



Look up table of 5 functions:

0	+	Add the arg1 to arg2
1	-	Subtract arg2 from arg1
2	*	Multiply arg1 to arg2
3	/	Divide arg1 by arg2
4	`sin`	Calculate sin of arg1

CGP chromosome:

$$C = (\underbrace{\underline{3},1,2}_{3},\underbrace{\underline{\mathbf{1}},\mathbf{2},\mathbf{0}}_{4},\underbrace{\underline{2},0,1}_{5},\underbrace{\underline{\mathbf{3}},\mathbf{1},\mathbf{4}}_{6},\underbrace{\underline{0},4,3}_{7},\underbrace{\underline{\mathbf{4}},\mathbf{1},\mathbf{5}}_{8},\underbrace{\underline{4},1,8}_{9},\underbrace{\underline{1},0,3}_{10},\underbrace{\underline{\mathbf{2}},\mathbf{6},\mathbf{8}}_{11},\underbrace{\underline{2},10,7}_{12},\underbrace{\underline{\mathbf{0}},\mathbf{0},\mathbf{11}}_{13},\underbrace{\underline{3},4,6}_{14},13)$$

Its length: $rc(1+2)+m = 3 \cdot 4 \cdot (1+2)+1 = 37$

The chromosome represents function $y = x_0 + \dfrac{x_1}{x_2 - x_0} \cdot \sin x_1$

CGP: Algorithm

Classic form of CGP uses a variant of $(1 + \lambda)$-EA

- with a point mutation variation operator;
- usually $\lambda = 4$.

$(1 + \lambda)$-EA:

1. Generate a random solution S
2. While not stopping criterion do
3. Generate λ mutated versions of S
4. Replace S by the best individual individual out of the λ new solutions iff it is **not worse** than S.
5. Return S as the best solution found

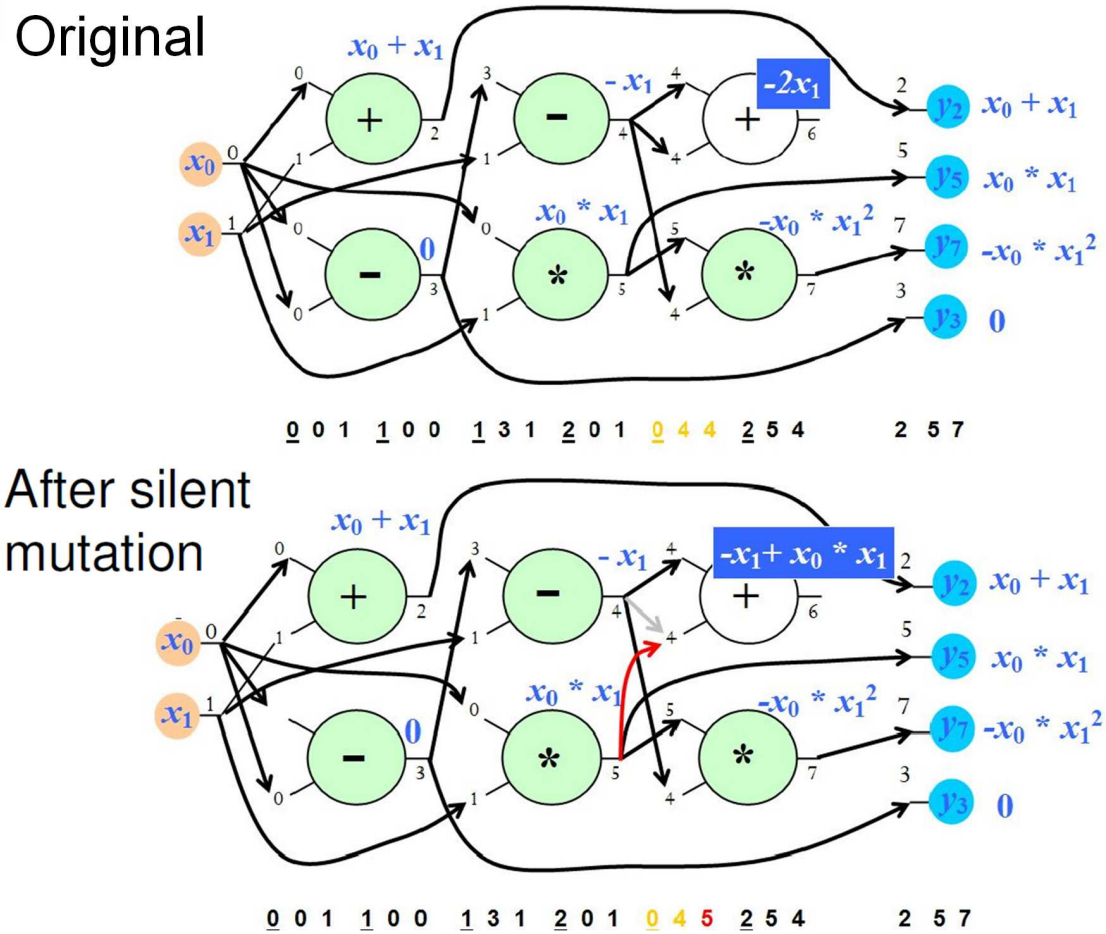
Neutral search:

- The algorithm accepts moves to new states of the solution space (step 4) that do not necessarily improve the quality of the current solution.
- This allows an introduction of new pieces of genetic code that can be plugged into the functional code later on.

If only improving steps are allowed then the search would not be neutral.

CGP: Point Mutation

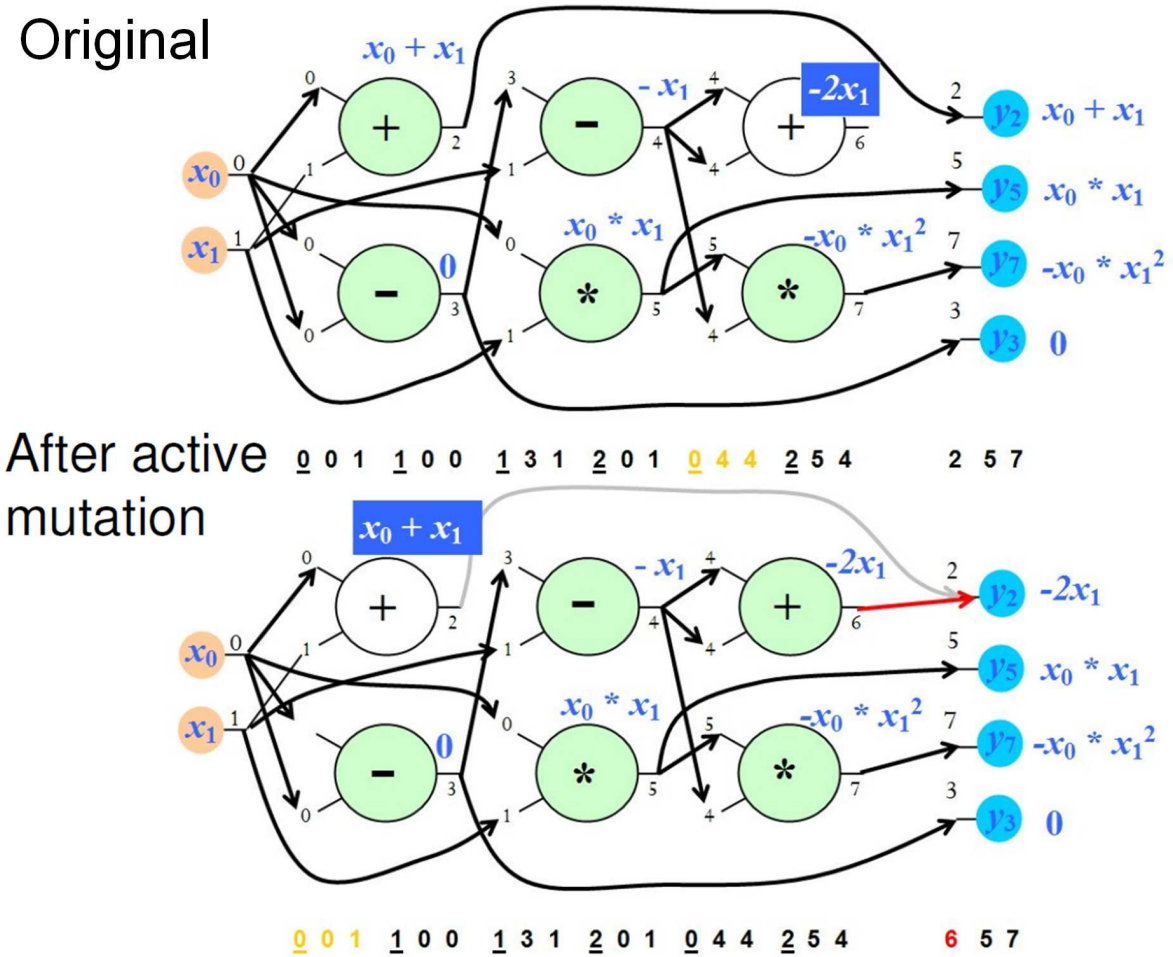
Silent mutations and their effects:



No change in phenotype but it changes the programs *accessible* through *subsequent* mutational change

CGP: Point Mutation

Non-silent mutations and their effects:

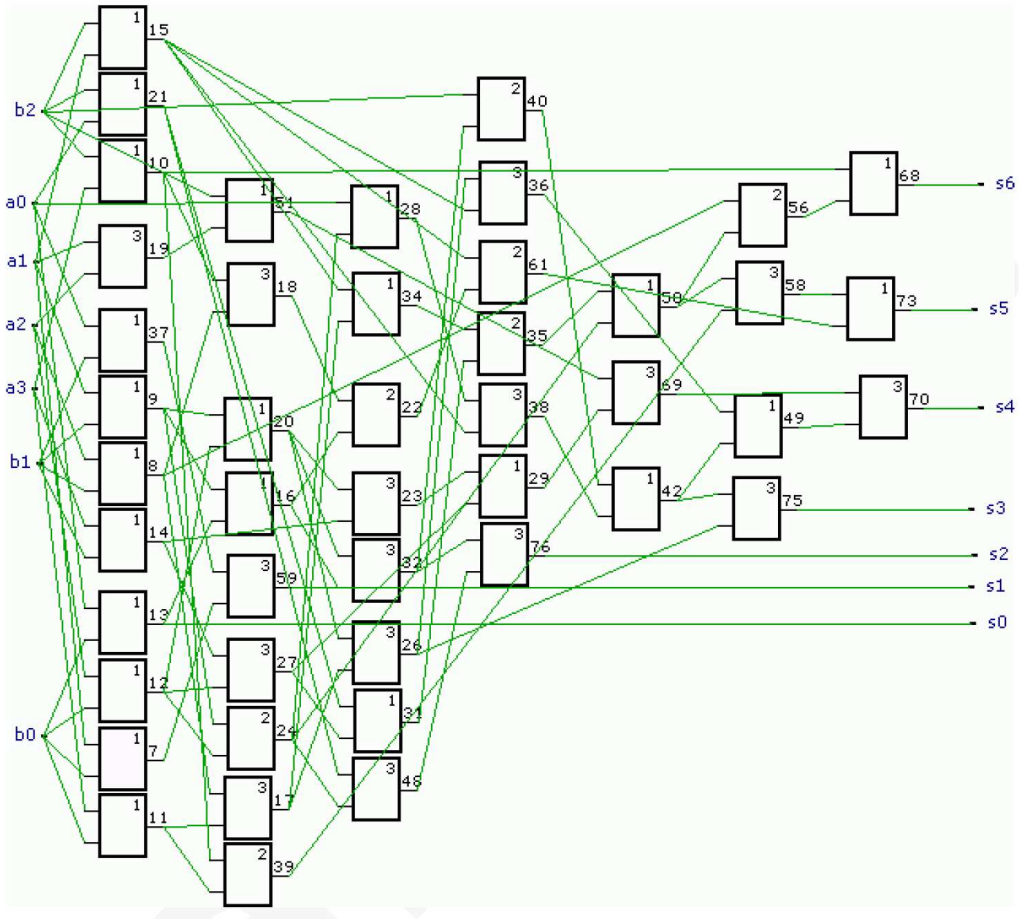


Massive change in phenotype is possible through simple mutation

CGP: Evolutionary Design of Boolean Circuits

CGP for evolution of **3x4-bit multiplier**

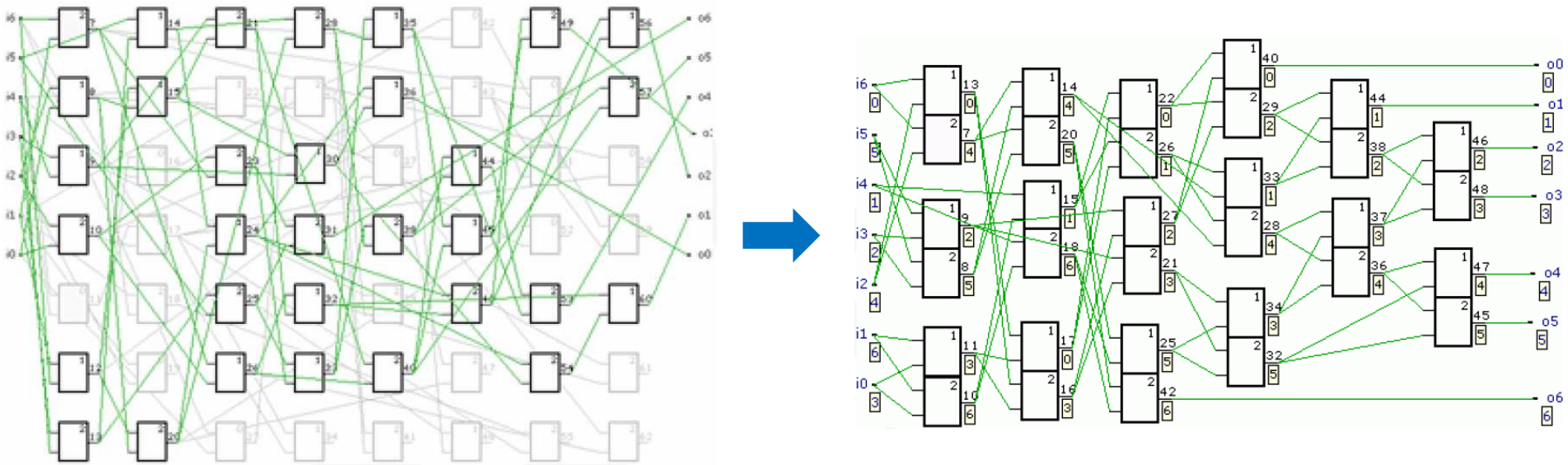
- $F = \{\texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{Wire-Jumper}\}$
- $T = \{a_0, \ldots, a_3, b_0, \ldots, b_2\}$
- (1+4)-EA
- $r = 10, c = 7, l = 7$



CGP: Evolutionary Design of Boolean Circuits

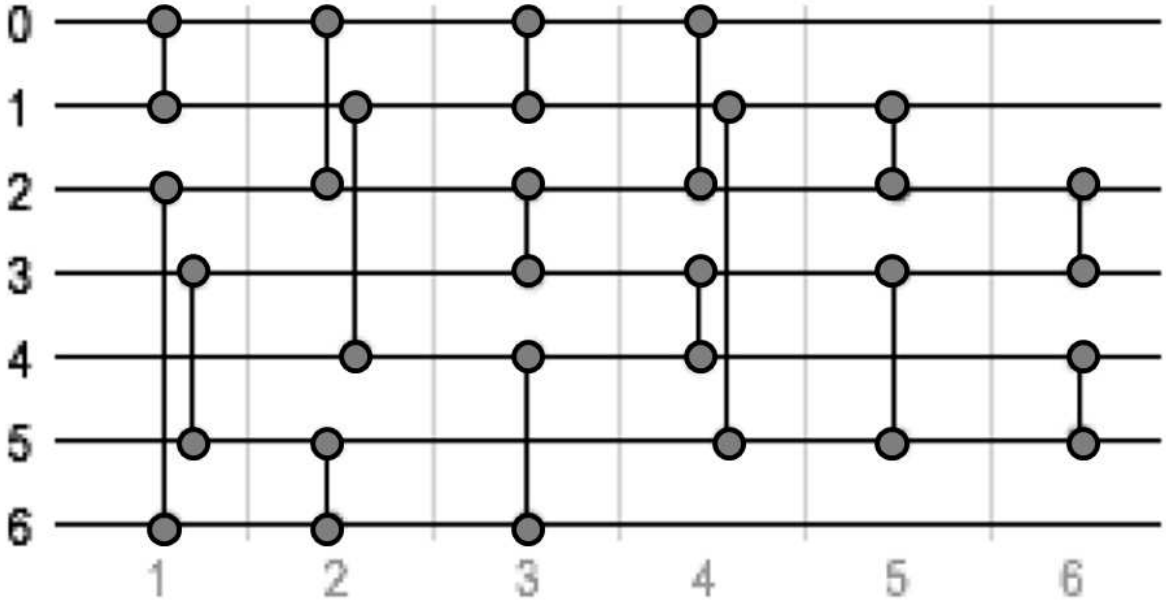
CGP for evolution of **7-bit sorting network**

- $F = \{\texttt{Compare\&Swap}, \texttt{Wire-Jumper}\}$ realized by `AND-OR` units
- $T = \{a_0, \ldots, a_6\}$
- (1+4)-EA
- $r = 7, c = 8, l = 8$



CGP: Evolutionary Design of Boolean Circuits

7-bit sorting network found by the CGP from previous slide realized by 16 `C&S` operations



CGP: Summary

Application areas

- Digital Circuit Design – parallel multipliers, digital filters, analogue circuits
- Mathematical functions
- Control systems – Maintaining control with faulty sensors, helicopter control, simulated robot controller
- Artificial Neural Networks – Developmental Neural Architectures
- Image processing – Image filters

Pros/cons:

- (+) Flexible program representation – genotype-phenotype mapping allows for a neutral evolution
- (+) Fixed genotype size but variable size and structure of the programs
- (+) Explicit automatic code reuse
- (+) Allows for an evolution of modules
- (–) Does not allow for multi level hierarchy in the ADFs

Summary

Learning outcomes

After this lecture, a student shall be able to

- implement a variable-length linear representation and a genotype-phenotype mapping used in GE;
- describe a representation of a program in CGP in the form of a directed graph;
- explain the neutral mutations in GE and CGP and their effect on the search process;
- describe the ripple crossover in GE;
- write a high-level pseudocode of GE and CGP;
- implement a concept of automatically defined functions into GE (Grammatical Evolution or meta-Grammar GE, GE grammar with the ability to define one more ADFs);
- explain the explicit automatic code reuse in CGP;

References

Grammatical Evolution

[OR01] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas F. Mcphee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, March 2008.

[RCO98] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 1998. Springer-Verlag.

Cartesian Genetic Programming

[Mil11] Julian Francis Miller, editor. *Cartesian Genetic Programming*. Springer, 2011.

[MT15] Julian Miller and Andrew Turner. Cartesian genetic programming. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, page 179–198, New York, NY, USA, 2015. Association for Computing Machinery.