Cartesian Genetic Programming

Jiří Kubalík
Department of Cybernetics, CTU Prague

Substantial part of this material is based on slides for tutorial 'Cartesian Genetic Programming'
presented at GECCO 2013 by J.F. Miller,
see http://dl.acm.org/citation.cfm?id=2464578
and the paper J. A. Walker and J. F. Miller: The Automatic Acquisition, Evolution and Reuse of Modules in CGP

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

Contents

Cartesian Genetic Programming

- Representation
- Genotype-phenotype mapping
- Examples: Design of boolean Circuits
- Embedded CGP

Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a GP technique that, in its classic form, uses a very simple integer based genetic representation of a **program in the form of a directed graph**.

- The genotype is a list of integers that represent the program primitives and how they are connected together.

  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.

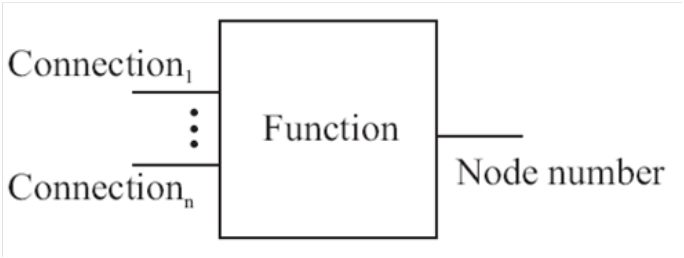
CGP Node

CGP program is a set of interconnected nodes.

A CGP node contains

- a `function symbol` – specifies the operation performed by the node,
- **connections** – pointers toward nodes providing input for the function of the node.

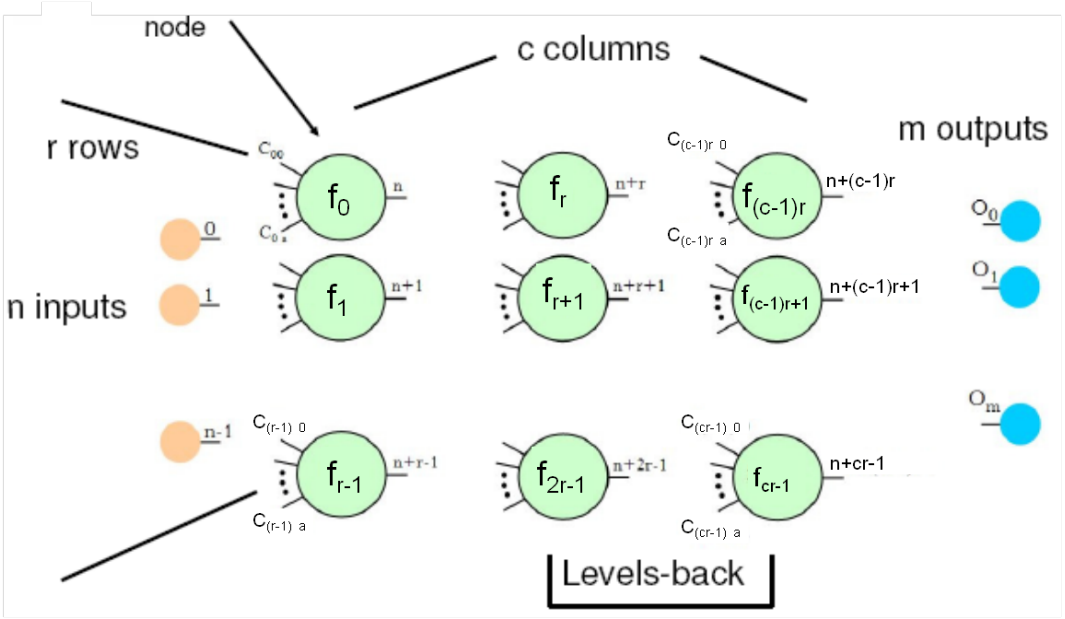
Each CGP node has an output that may be used as an input for another node.



CGP node

CGP General Form

CGP is Cartesian in the sense that the graph nodes are represented 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,
- nodes interconnectivity l

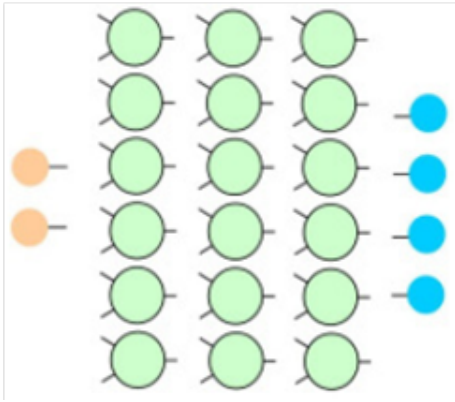
Nodes in the same column are not allowed to be connected to each other.

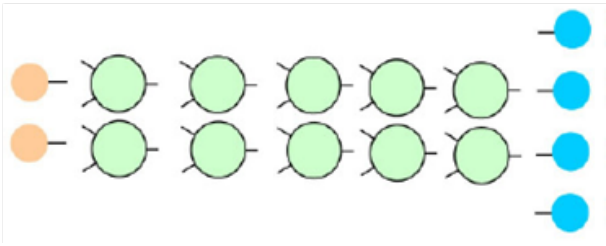
The **nodes interconnectivity** defines the maximum distance (in terms of the number of columns) between two connected nodes.

- 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

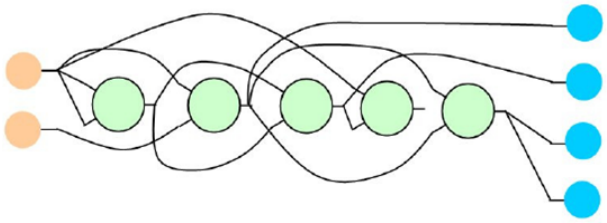




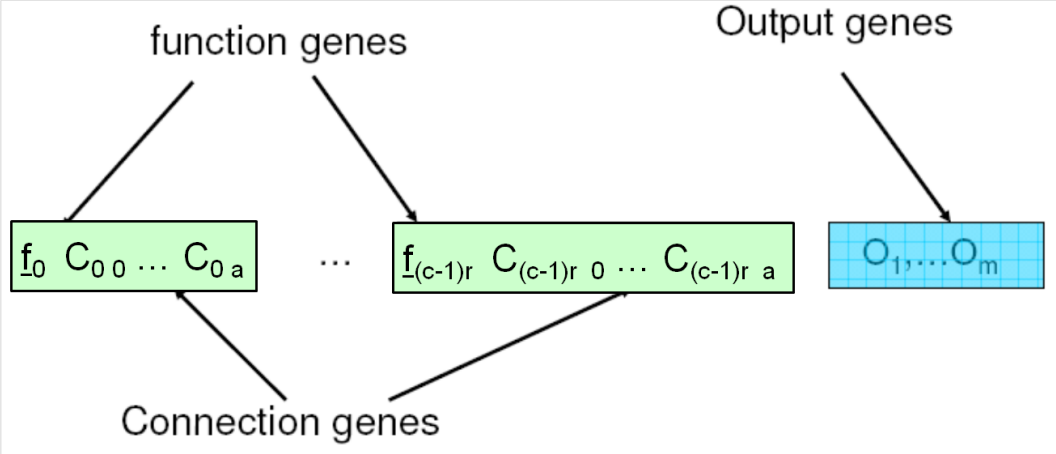
The length of the genotype (i.e. the maximum size of the CGP program) is fixed, however the **actual size and structure of the program can vary**.

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

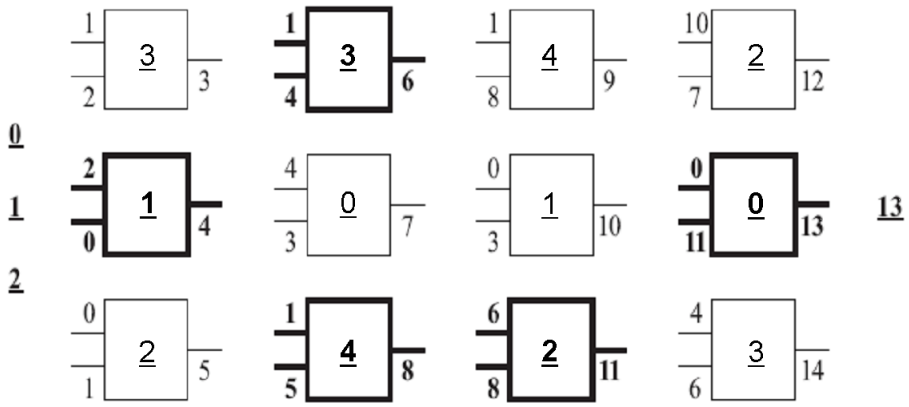


Usually, all functions have as many inputs as the maximum function arity.

Unused connections are ignored.

CGP Program Example

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



Look up table of 5 functions

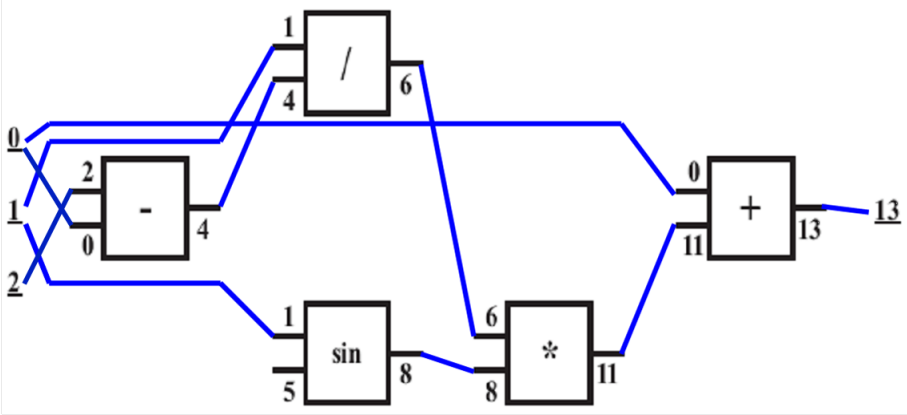
0	+	Add the arg1 to arg2
1	-	Subtract arg2 from arg1
2	*	Multiply arg1 to arg2
3	/	Divide arg1 by arg2
4	`sin`	Calculate sin of arg1

CGP chromosome

C=(3,1,2, **1,2,0**, 2,0,1, **3,1,4**, 0,4,3, **4,1,5**, 4,1,8, 1,0,3, **2,6,8**, 2,10,7, **0,0,11**, 3,4,6, 13)

CGP Program Example

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



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=(3,1,2, **1,2,0**, 2,0,1, **3,1,4**, 0,4,3, **4,1,5**, 4,1,8, 1,0,3, **2,6,8**, 2,10,7, **0,0,11**, 3,4,6, 13)

Graph function: $y = x_0 + (x_1/(x_2 - x_0)) * \sin x_1$

CGP: Algorithm

In its classic form, CGP uses a variant of a simple algorithm called $(1 + \lambda)$-Evolutionary Strategy with a point mutation variation operator, where λ is usually 4.

$(1 + \lambda)$-ES:

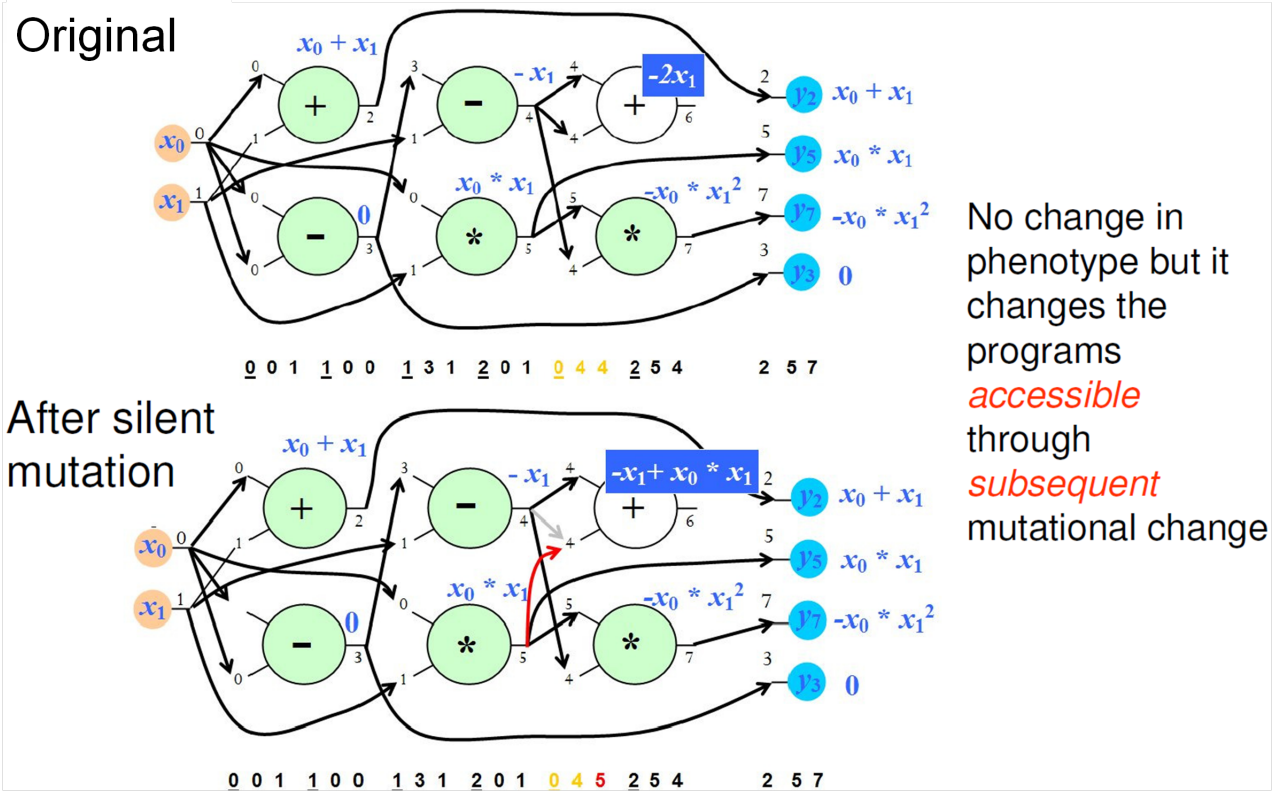
1. Generate a random solution S
2. while not stopping criterion do
3. Generate λ mutated versions of S
4. Replace S by the best individual individual out of the λ new solutions iff it is not worse than S.
5. Return S as the best solution found

Neutral search – in step 4 we accept move to new states of the solution space that do not necessarily improve the quality of the current solution.

If only improving steps are allowed then the search would be non-neutral.

CGP: Point Mutation

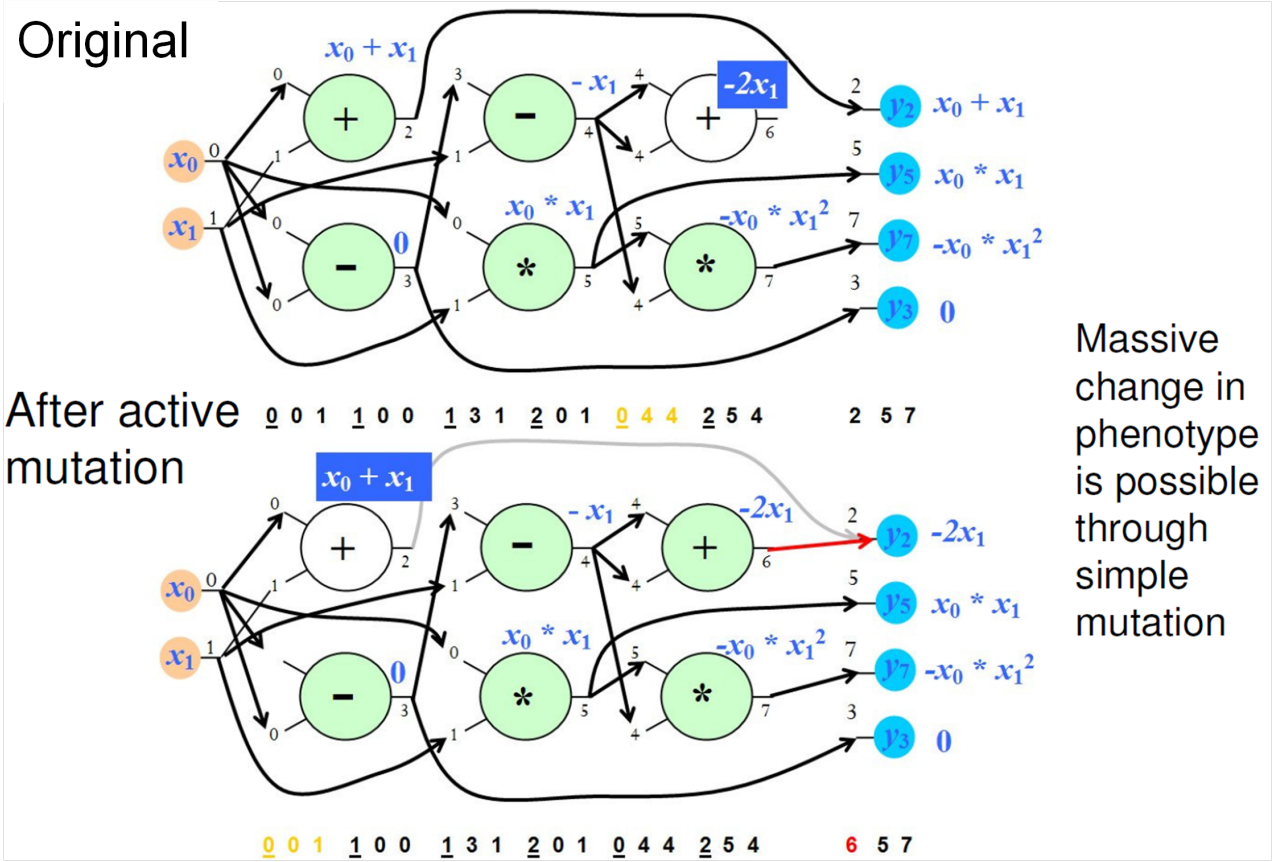
Silent mutations and their effects



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

CGP: Point Mutation

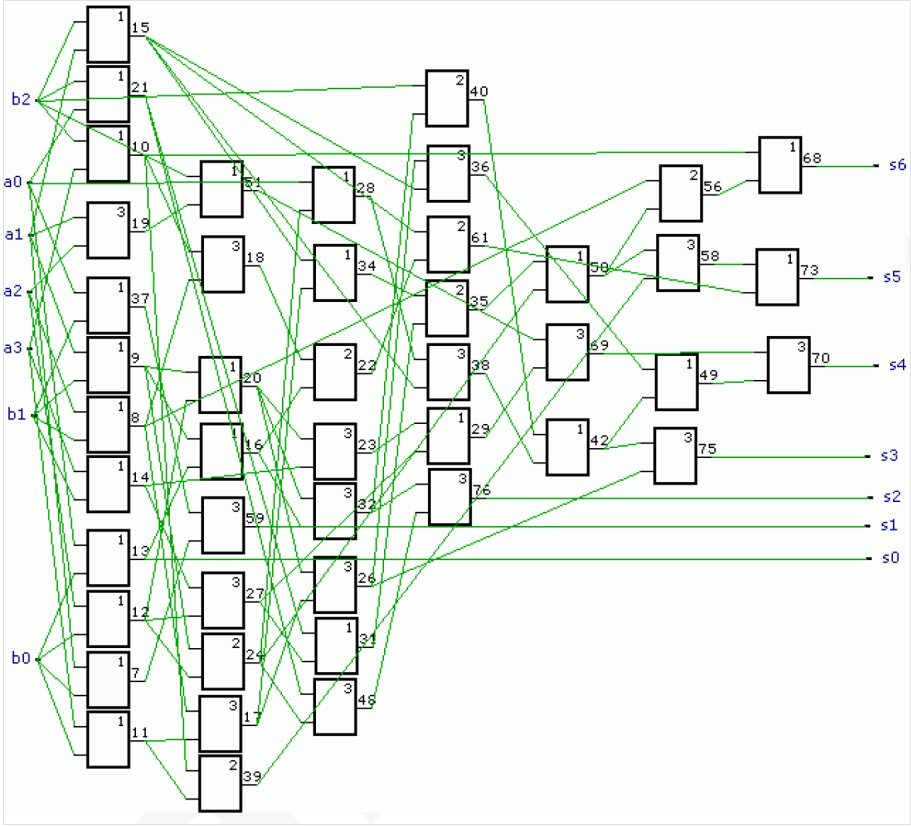
Non-silent mutations and their effects



CGP: Evolutionary Design of Boolean Circuits

CGP for evolution of 3×4-bit multiplier

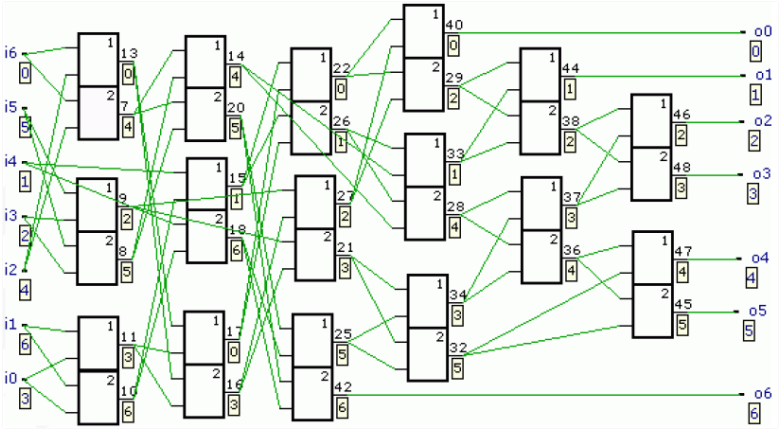
- $F = \{\texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{Wire-Jumper}\}$
- $T = \{a_0, \ldots, a_3, b_0, \ldots, b_2\}$
- (1+4)-ES
- $r = 10$, $c = 7$, $l = 7$



CGP: Evolutionary Design of Boolean Circuits

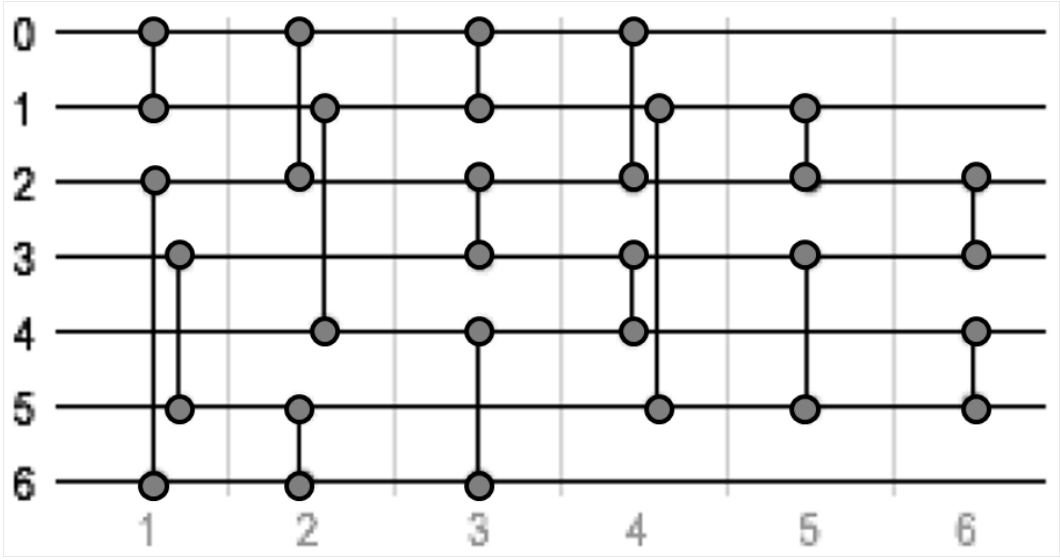
CGP for evolution of 7-bit sorting network

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



CGP: Evolutionary Design of Boolean Circuits

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



Embedded Cartesian Genetic Programming (ECGP)

ECGP incorporates a concept of ADFs by automatic acquisition, evolution and reuse of partial solutions (referred to as modules) implemented through:

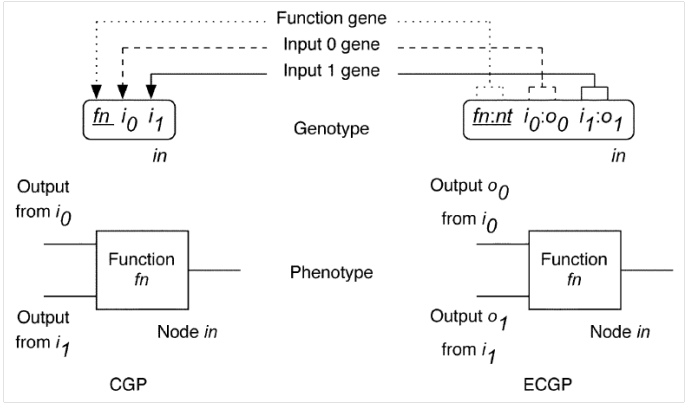
- extended representation,
- compress and expand operators, genotype point mutation,
- module operators – module point mutation, add-input, add-output, remove-input, remove-output,
- evolution strategy model that introduces an implicit pressure for good modules.

ECGP: Representation

A genotype is a bounded variable length representation.

Node representation:

- function $fn : nt$, where
  - fn is either a primitive function or a module,
  - nt is a node type.
- Each gene is encoded using a pair of integers $[o_j : i_j]$, interpreted as o_j-th output of node i_j (for example $[o_0 : i_0]$ in the figure).



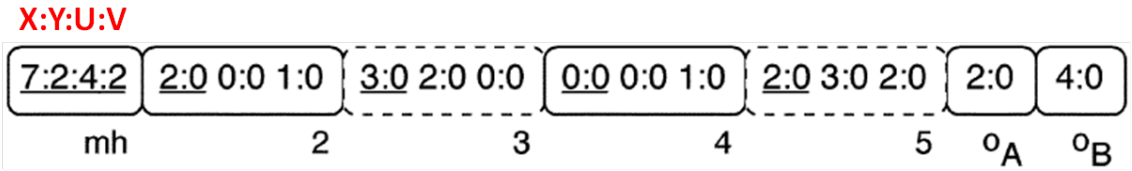
Node types:

- Type 0 – primitive functions,
- Type 1 – modules created by a compress operator; modules that contain an original section of the genotype,
- Type 2 – modules that have been reused.

ECGP: Module Representation

Module is represented as a bounded length genotype; the number of nodes in the module genotype remains fixed.

- **Module header A:B:C:D**
  - A – the module identifier,
  - B – the number of module inputs,
  - C – the number of nodes contained in the module,
  - D – the number of module outputs.
- **Module body** – encodes the connections and functions of the nodes contained in the module, and the module outputs. **Module nodes can represent only primitive functions.**



Module dimensions:

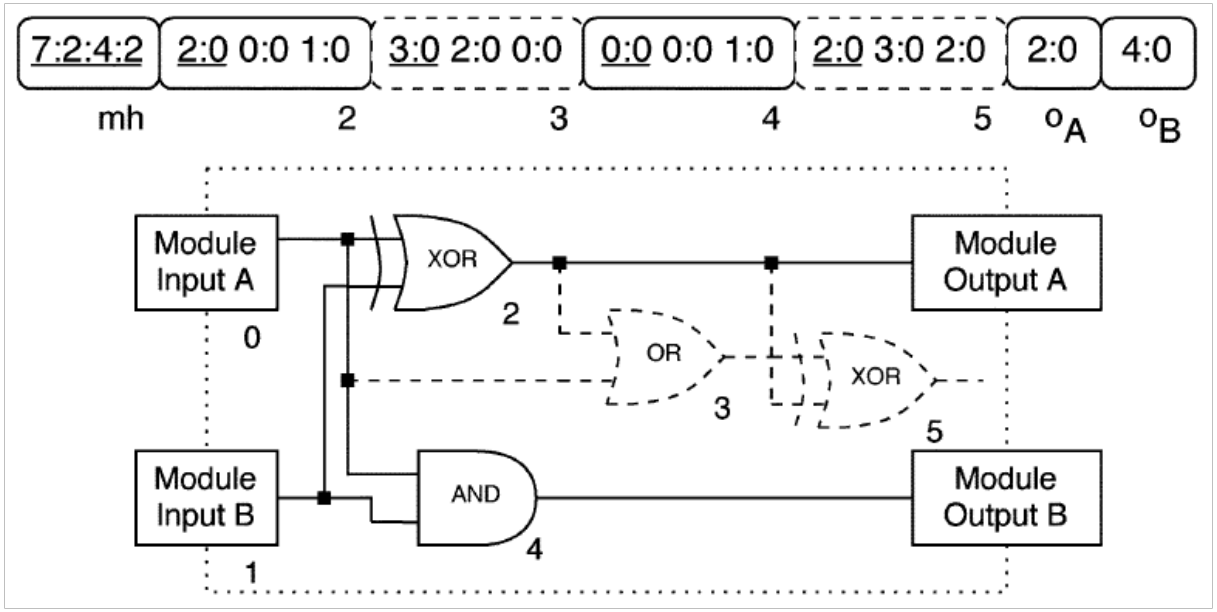
- Number of nodes – between 2 and MAX_NODES
- Number of outputs – between 1 and the number of nodes in the module, n,
- Number of inputs – between 2 and $2n$.

ECGP: Module Representation

Ex.: The genotype and corresponding phenotype of a module with

- 2 inputs,
- 2 outputs, and
- 4 nodes

constructing a half adder circuit.



ECGP: Module List

All active modules are stored in a **module list**.

The module list

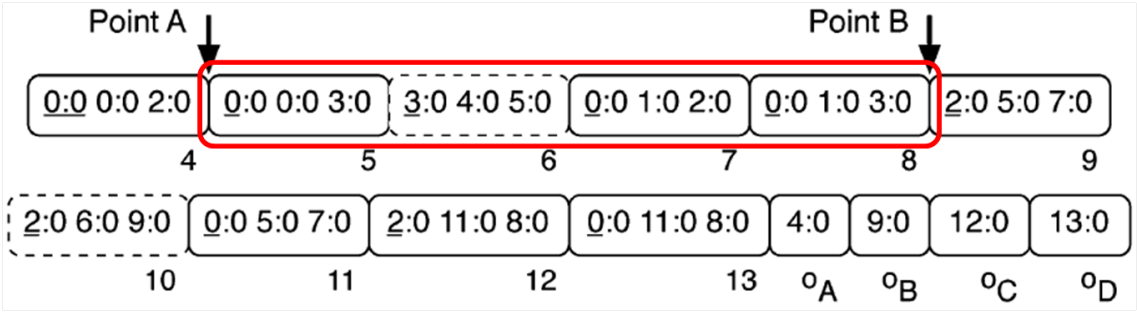
- is shared by all individuals in the population,
- is dynamic and has no restrictions on its maximum size,
- is updated in each generation to include only those modules present in the fittest individual.

Any node in a genotype can represent any primitive function or module.

ECGP: Compress Operator

Compress operator constructs modules through the following steps:

1. A section between two randomly chosen points A and B is removed from the genotype.



   The section can contain **only nodes representing primitive functions**.

2. The extracted section is converted into a module – header and body.

   All nodes and their inputs are relabeled.

3. A node of type 1 representing the new module is added to the genotype in place of the removed section.

   All nodes, node inputs, and outputs occurring after the inserted node are relabeled.

A module created by the compress operator is represented in the genotype as a **type 1 node**.

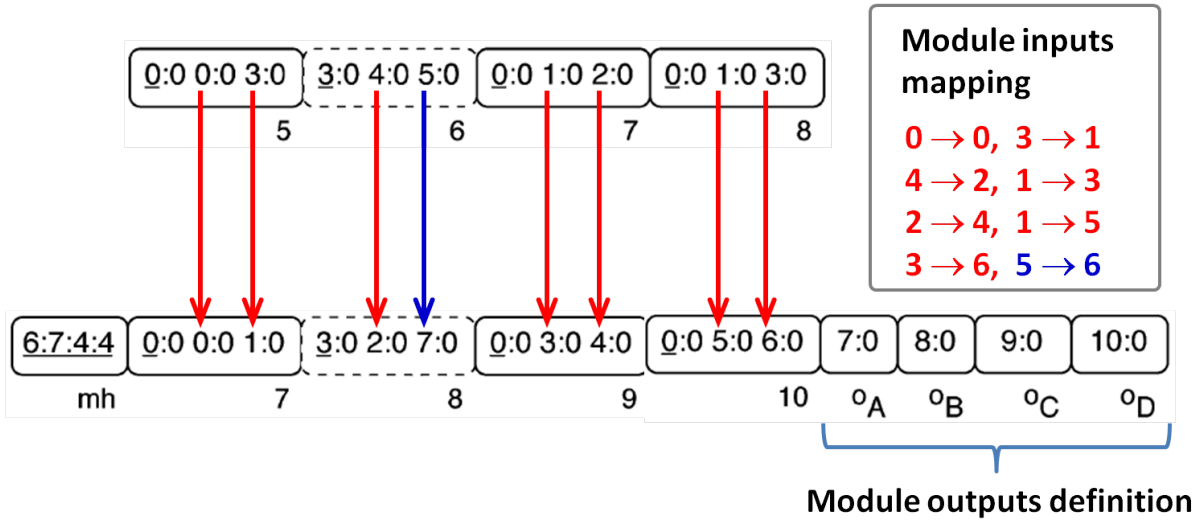
ECGP: Compress Operator - Step 2

The number of module inputs that a module is initialized with is determined by the number of connections leading to the inputs of the nodes being encapsulated into a module.

If there are repeated connections to the output of a previous node, each connection is assigned it's own module input.

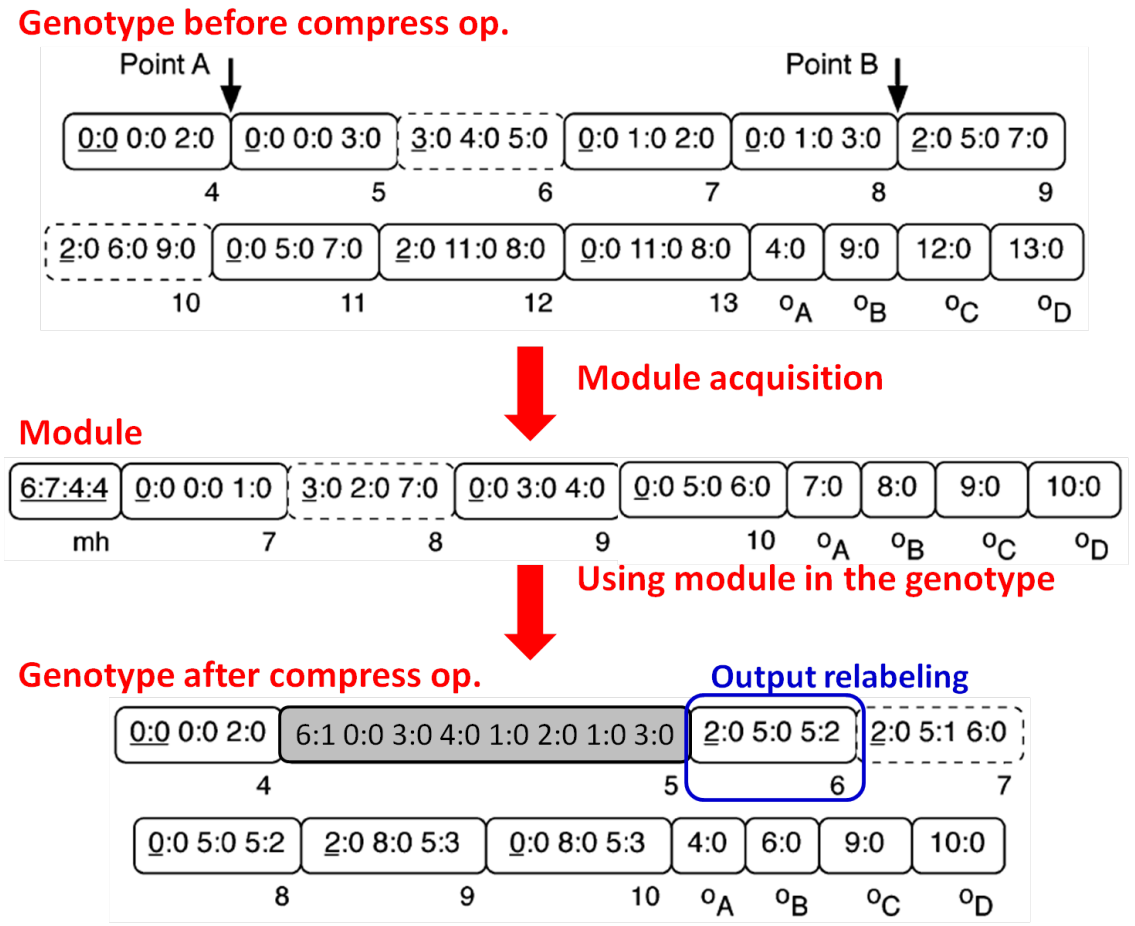
There are in total 7 inputs and 4 outputs of the module in the example below.

The number of module outputs is determined by the number of connections between the inputs of the later nodes in the genotype (nodes behind the section of the module) and the outputs of the nodes that are going to be encapsulated in the module.

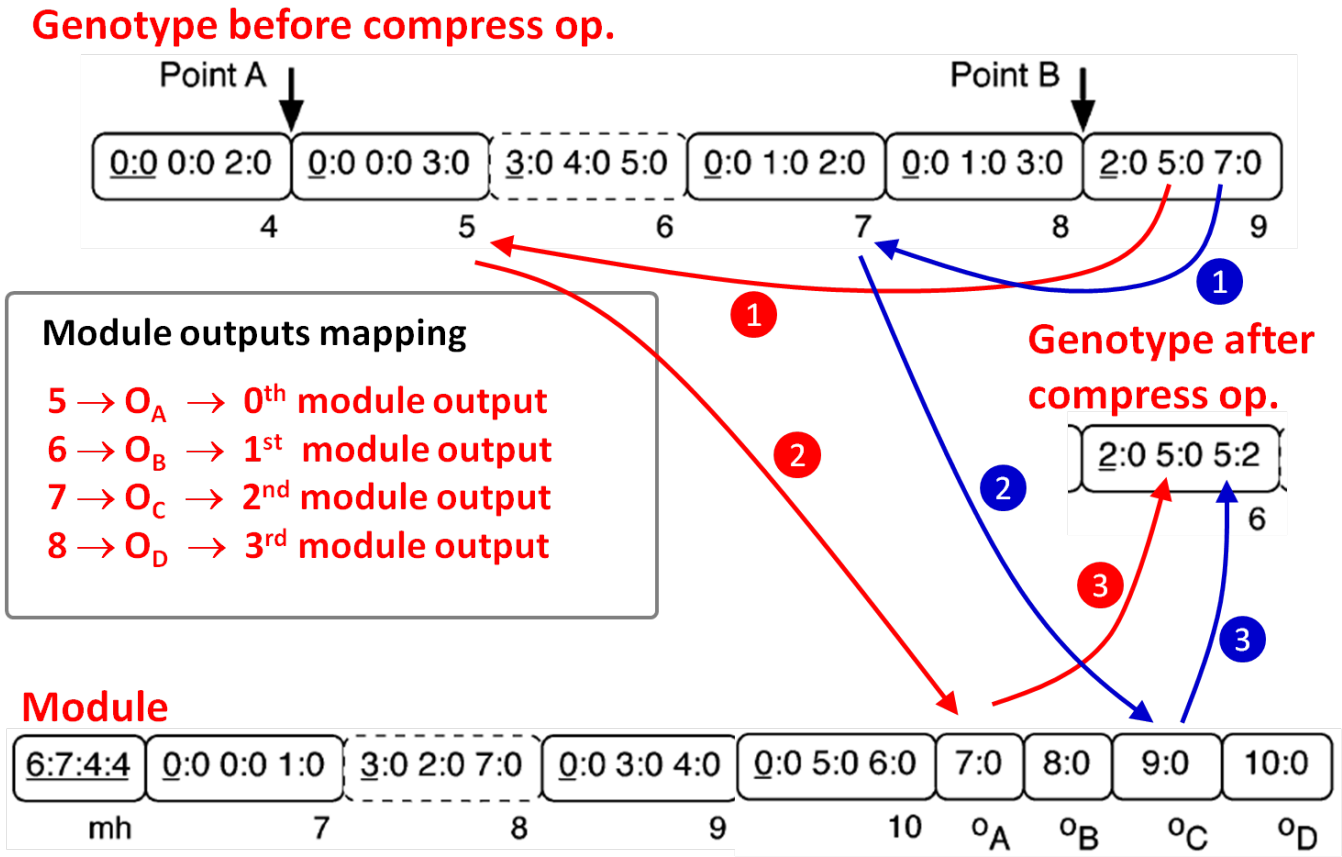


ECGP: Compress Operator - Step 3

A node of type 1 representing the new module is added to the genotype in place of the removed section of nodes.



ECGP: Compress Operator - Module Outputs Mapping



ECGP: Expand Operator

An **expand operator** is the reverse of the compress operator, it **destructs modules**. It can be **applied only to type 1 nodes**.

The expand operator randomly selects a type 1 node in the genotype and replaces it with the nodes contained in the module represented by the type 1 node.

If this was the only use of the module, the module is removed from the module list.

ECGP: Remarks on Compress and Expand Operators

The inputs of all of the later nodes in the genotype are updated in the final stage of the compress and expand operators, so all of the connections remain intact.

The operators **only make a structural changes to the genotype**, they have no affect on genotype fitness.

ECGP: Genotype Point Mutation

Genotype Point Mutation provides the only way the modules can replicate within the genotype.

- It can mutate the function of a node to any primitive function or module.

  If the function node is mutated to represent a module, it becomes a **type 2 node**.

  Type 2 nodes are immune from the expand operator. This is to avoid an excessive growth of the genotype.

  Once the module is represented by a type 2 node, it is harder to be removed from the module list since **it cannot be expanded**.

  Whenever a node changes its type ($0 \longleftrightarrow 2$), extra inputs are randomly generated if necessary.
- Mutates the inputs of any type 0, type 1, or type 2 node.

ECGP: Module Operators

Modules can be evolved through the following operators:

- **Module Point Mutation** – mutates the input, output and function genes of any node encoded in the module.

  It is **not allowed to introduce any type 2 node into the module**.

- **Add-Input**, **Add-Output** – changes connectivity to and from the module by increasing the number of module inputs or outputs by 1 (making sure that the constraints for the maximum number of inputs/outputs are obeyed).

- **Remove-Input**, **Remove-Output** – decreases the number of inputs or outputs by 1 (making sure to obey the constraints for the minimum number of inputs/outputs).

ECGP: Evolutionary Strategy

(1+4)Evolutionary strategy

1. Randomly generate an initial population of five genotypes and select the fittest.
2. Carry out mutation on the winning parent to generate four offspring.
3. Select a winner from the current individual and its four offspring using the following rule
   (a) If any offspring has a better fitness than the parent; the best becomes the winner.
   (b) Otherwise, an offspring with the same fitness as the parent is randomly selected.
   (c) Otherwise, the parent remains as the winner.
4. Go to Step 2 unless the maximum number of generations is reached or a solution is found.

ECGP: Even Parity Problem

ECGP compared to CGP, GP and GP with ADFs using the computational effort statistic ($I(M, i, z)$).

Parity	CGP	ECGP	GP	GP with ADFs
3	33,282	37,446	96,000	64,000
4	151,683	201,602	384,000	176,000
5	776,002	512,002	6,528,000	464,000
6	3,044,162	978,882	70,176,000	1,344,000
7	11,451,202	1,923,842	-	-
8	31,187,842	4,032,002	-	-

CGP as well as ECGP always produces 100% successful solution.

As the complexity of the problem increases, **ECGP performs significantly better than CGP**.

GP with ADFs allows a two hierarchy in the ADFs, while the **ECGP uses just single-level hierarchy**.

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

CGP: Sources

- Miller, J.F.: GECCO 2013 Tutorial: Cartesian Genetic Programming
  `http://portal.acm.org/citation.cfm?id=1389075`
- Home site: `http://www.cartesiangp.co.uk`
- Julian Miller: `http://www.elec.york.ac.uk/staff/jfm7.html`
- Simon Harding: `http://www.cs.mun.ca/~simonh/`
- Lukas Sekanina: `http://www.fit.vutbr.cz/~sekanina/`

Sekanina L., Vašíček Z., Růžička R., Bidlo M., Jaroš J., Švenda P.: Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům. Academia Praha 2009