GECCO 2010 Tutorial: Cartesian Genetic Programming

Julian F. Miller
Dept of Electronics
University of York, UK
jfm7@ohm.york.ac.uk

Simon L. Harding
Dept of Computer Science
Memorial University of Canada
slh@evolutioninmaterio.com

Copyright is held by the author/owner(s).
GECCO ’10, July 7–11, 2010, Portland, Oregon, USA.
ACM 978-1-4503-0073-5/10/07.

Abstract
Cartesian Genetic Programming is a form of genetic programming. It is increasing in popularity. It was developed by Julian Miller with Peter Thomson in 1997. In its classic form it uses a very simple integer based genetic representation of a program in the form of a directed graph. In a number of studies, it has been shown to be efficient in comparison with other GP techniques.

Since then, the classical form of CGP has been enhanced in various ways by including automatically defined functions.

Most recently, it has been developed by Julian Miller, Wolfgang Banzhaf and Simon Harding to include self-modification operators. This again has increased its efficiency.

The tutorial will cover the basic technique, advanced developments and applications to a variety of problem domains.

Contents
- Classic CGP
- Modular CGP
- Self-modifying CGP
- Developmental CGP
- Cyclic CGP
- Applications
- Resources
- Bibliography

Genetic Programming
- The automatic evolution of computer programs
  - Tree-based, Koza 1992
  - Stack-based, Perkis 1994, Spector 1996 onwards (push-pop GP)
  - Linear GP, Nordin and Banzhaf 1996
  - Cartesian GP, Miller 1997
  - Parallel Distributed GP, Poli 1996
  - Grammatical Evolution, Ryan 1998
  - Lots of others…
Origins of Cartesian Genetic Programming (CGP)

- Originally, represents programs or circuits as a two dimensional grid of program primitives.
- This is loosely inspired by the architecture of digital circuits called FPGAs (field programmable gate arrays).

What defines CGP?

- The genotype is a list of integers (and possibly parameters) that represent the program primitives and how they are connected together.
  - CGP represents programs as graphs in which there are non-coding genes.
- The genes are:
  - Addresses in data (connection genes)
  - Addresses in a look up table of functions
  - Additional parameters
- This representation is very simple, flexible and convenient for many problems.

CGP General form

- The genotype is a list of integers (and possibly parameters) that represent the program primitives and how they are connected together.
- CGP represents programs as graphs in which there are non-coding genes.
- The genes are:
  - Addresses in data (connection genes)
  - Addresses in a look up table of functions
  - Additional parameters
- This representation is very simple, flexible and convenient for many problems.

Allelic constraints for directed acyclic graphs

All function genes $f_i$ must take allowed function alleles:

$$0 \leq f_i \leq n_f$$

Nodes connections $c_{ij}$ of a node in column $j$, and levels-back $l$, must obey (to retain directed acyclicity):

- $j \geq 1$:
  $$n + (j-1)r \leq c_{ij} \leq n +jr$$
- $j < 1$:
  $$0 \leq c_{ij} \leq n + jr$$

Output genes (can connect to any previous node or input):

$$0 \leq o_i \leq n + cr -1$$
**Types of graphs easily controlled**

- Depending on rows, columns and levels-back a wide range of graphs can be generated.

- When rows = 1 and levels-back = columns arbitrary directed graphs can be created with a maximum depth.
  - In general choosing these parameters imposes the least constraints. So without specialist knowledge this is the best and most general choice.

**CGP genotype**

- Function genes
- Output genes
- Connection genes

Usually, all functions have as many inputs as the maximum function arity.
Unused connections are ignored.

**Example**

- Encoding of graph as a list of integers (i.e. the genotype)

0 0 1 1 0 0 1 3 1 2 0 1 0 4 4 2 5 4 2 5 7 3

**Example: Function look up table**

The function genes are the addresses in a user-defined lookup table of functions:

0 + Add the data presented to inputs
1 - Subtract the data presented to inputs
2 * Multiply data presented to inputs
3 / Divide data presented to inputs (protected)
Obtaining the graph

Encoding of graph as a list of integers (i.e. the genotype)

So what does the graph represent?

What happened to the node whose output label is 6?

The node was not used so the genes are silent or non-coding

The CGP genotype-phenotype map

- When you decode a CGP genotype many nodes and their genes can be ignored because they are not referenced in the path from inputs to outputs
- These genes can be altered and make no difference to the phenotype, they are non-coding
- Clearly there is a many-to-one genotype to phenotype map
- How redundant is the mapping?
A mathematical aside: CGP and Stirling numbers

- Assume that a CGP graph has the following parameters
  - Number of rows = 1
  - Levels-back = num_cols = n
  - There is one input
  - Assume that the output is taken from the last node

The number of genotypes, \( G \), that have a phenotype of size \( k \) (nodes) can be shown to obey a recurrence relation obeyed by unsigned Stirling numbers of the first kind.

\[
G(n+1, k) = nG(n, k) + G(n, k-1)
\]

How many genotypes of length \( n \) map to a phenotypes of length \( k \)?

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>11</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>50</td>
<td>35</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>274</td>
<td>225</td>
<td>85</td>
<td>15</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>720</td>
<td>1764</td>
<td>1624</td>
<td>735</td>
<td>175</td>
<td>21</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5040</td>
<td>13068</td>
<td>13132</td>
<td>6759</td>
<td>1960</td>
<td>322</td>
<td>28</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>40320</td>
<td>109584</td>
<td>118124</td>
<td>67284</td>
<td>22449</td>
<td>4536</td>
<td>546</td>
<td>36</td>
<td>1</td>
</tr>
</tbody>
</table>

Average number of active nodes in a genotype of length 9 is 2.83

Clearly, with say a genotype of 100 nodes, the number of genotypes that map to a phenotype with say about 10 nodes is an astronomical number

Decoding CGP chromosomes is easy

```plaintext
// L = MaxGraph.Length
// I = Number of program inputs
// N = Number of program outputs
bool ToEvaluate[L]
double NodeOutput[L+I]

// identify initial nodes that need to be evaluated
p = 0
do
    ToEvaluate[OutputGene[p]] = true
    p = p + 1
while (p < N)

// work out which nodes are used
p = L-1
do
    if (ToEvaluate[p])
        x = Node[p].Connection1
        y = Node[p].Connection2
        ToEvaluate[x] = true
        ToEvaluate[y] = true
    endif
    p = p - 1
while (p >= 0)

// load input data values
p = 0
do
    NodeOutput[p] = InputData[p]
    p = p + 1
while (p < I)

// Execute graph
p = 0
do
    if (ToEvaluate[p])
        x = Node[p].Connection1
        y = Node[p].Connection2
        z = NodeFunction[p].Function
        NodeOutput[p+I] = ComputeNode(NodeOutput[x], NodeOutput[y], z)
    endif
    p = p + 1
while (p < L)
```

Point mutation

- Most CGP implementations only use mutation.
- Carrying out mutation is very simple. It consists of the following steps. The genes must be chosen to be valid alleles

```plaintext
// Decide how many genes to change:
num_mutations
while (mutation_counter < num_mutations)
{
    get gene to change
    if (gene is a function gene)
        change gene to randomly chosen new valid function
    else if (gene is a connection gene)
        change gene to a randomly chosen new valid connection
    else
        change gene to a new valid output connection
}
```
Evolutionary Strategy

- CGP often uses a variant of a simple algorithm called (1 + 4) Evolutionary Strategy
  - However, an offspring is always chosen if it is equally as fit or has better fitness than the parent

Crossover or not?

- Recombination doesn’t seem to add anything (Miller 1999, “An empirical study…”)
- However if there are multiple chromosomes with independent fitness assessment then it helps a LOT (Walker, Miller Cavill 2006)
- Some work using a floating point representation of CGP has suggested that crossover might be useful (Clegg, Walker, Miller 2007)

Silent mutations and their effects

Original

After silent mutation

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

Non-silent mutations and their effects

Original

After active mutation

Massive change in phenotype is possible through simple mutation
Neutral search is fundamental to success of CGP

- A number of studies have been carried out to indicate the importance of neutral search

Neutral search and the three bit multiplier problem (Vassilev and Miller 2000)

Importance of neutral search can be demonstrated by looking at the success rate in evolving a correct three-bit digital parallel multiplier circuit.

Graph shows final fitness obtained in each of 100 runs of 10 million generations with neutral mutations enabled compared with disabling neutral mutations.

Effectiveness of Neutral Search as a function of mutation rate and Hamming bound (Yu and Miller 2001)

- Hamming Distance $H(g,h)$
  - $g_1 = 213 \ 012 \ 130 \ 432 \ 159$
  - $g_2 = 202 \ 033 \ 132 \ 502 \ 652$
  - Hamming distance $H(g_1, g_2) = 9$.
- If genotypes are selected so that $H(g_{new}, g_{old}) = 0$. No neutral drift is permitted.
- If genotypes are selected so that $H(g_{new}, g_{old}) = \text{length}(g)$. Any amount of neutral drift is permitted.

In CGP, large genotypes and small mutation evolve solutions to problems more quickly [Miller and Smith 2006]

Even 3 parity with gate set (AND, OR, NAND, NOR).

Two-bit multiplier with gate set (AND, OR, NAND, NOR).

- However big genotypes does NOT mean big phenotypes (programs)....
Phenotype length versus genotype length (two-bit multiplier)

Average proportion of active nodes in genotype at the conclusion of evolutionary run for all mutation rates versus genotype length

SEARCH MOST EFFECTIVE WHEN 95% OF ALL GENES ARE INACTIVE!!

Average phenotype length for the initial population contrasted with the average phenotype length at conclusion of evolutionary run versus genotype length with 1% mutation

NO BLOAT

Modular/Embedded CGP (Walker, Miller 2004)

- So far have described a form of CGP (classic) that does not have an equivalent of Automatically Defined Functions (ADFs)
- Modular CGP allows the use of modules (ADFs)
  - Modules are dynamically created and destroyed
  - Modules can be evolved
  - Modules can be re-used

MCGP Example

Genotype

Module List

Module Creation

Representation Modification 1

- Each gene encoded by two integers in M-CGP
  - Function/module number and node type
  - Node index and node output
    - nodes can have multiple outputs
**Representation Modification 2**

M-CGP has a bounded variable length genotype
- Compression and expansion of modules
  - Increases/decreases the number of nodes
- Varying number of module inputs
  - Increases/decreases the number of genes in a node

**Modules**

- Same characteristics as M-CGP
  - Bounded variable length genotype
  - Bounded variable length phenotype

- Modules also contain inactive genes as in CGP
- Modules can not contain other modules!

**Node Types**

- Three node types:
  - Type 0
    - Primitive function
  - Type I
    - Module created by compress operator
  - Type II
    - Module replicated by genotype point-mutation

**Creating and Destroying a Module**

- Created by the *compress* operator
  - Randomly acquires sections of the genotype into a module
    - Sections must ONLY contain type 0 nodes
- Destroyed by the *expand* operator
  - Converts a random type I module back into a section of the genotype
Module Survival

- Twice the probability of a module being destroyed than created
- Modules have to replicate to improve their chance of survival
  - Lower probability of being removed
- Modules must also be associated with a high fitness genotype in order to survive
  - Offspring inherit the modules of the fittest parent

Evolving a Module I

- Structural mutation
  - Add input
  - Remove input
  - Add output
  - Remove output

Evolving a Module II

- Module point-mutation operator
  - Restricted version of genotype point-mutation operator
  - Only uses primitive functions

Re-using a Module

- Genotype point-mutation operator
  - Modified CGP point-mutation operator
- Allows modules to replicate in the genotype
  - Primitive (type 0) → module (type II)
  - Module (type II) → module (type II)
  - Module (type II) → primitive (type 0)
- Does **NOT** allow type I modules to be mutated into primitives (type 0) or other modules (type II)
  - Type I modules can only be destroyed by Expand
Experimental parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>5</td>
</tr>
<tr>
<td>Initial genotype size</td>
<td>100 nodes (300 genes)</td>
</tr>
<tr>
<td>Genotype point mutation rate</td>
<td>3% (9 genes)</td>
</tr>
<tr>
<td>Genotype point mutation probability</td>
<td>1</td>
</tr>
<tr>
<td>Compress/Expand probability</td>
<td>0.1/0.2</td>
</tr>
<tr>
<td>Module point mutation probability</td>
<td>0.04</td>
</tr>
<tr>
<td>Add/Remove input probability</td>
<td>0.01/0.02</td>
</tr>
<tr>
<td>Add/Remove output probability</td>
<td>0.01/0.02</td>
</tr>
<tr>
<td>Module list initial contents</td>
<td>Empty</td>
</tr>
<tr>
<td>Number of independent runs</td>
<td>50</td>
</tr>
</tbody>
</table>

NOTES: ◊ these parameters only apply to Modular (Embedded) CGP

The results are heavily dependent on the maximum number of nodes allowed. Much better results are obtained when larger genotype lengths are used.

---

**Digital Multiplier**

- Two digital multiplier problems:
  - 2-bit and 3-bit
- Function set:
  - AND, AND (one input inverted), XOR, OR
- Fitness Function:
  - Number of phenotype output bits that differ from the perfect \( n \)-bit digital multiplier solution
  - Perfect solution has a fitness of zero
- Results are averaged over fifty independent runs

---

**Even Parity Results**

![Graph showing Even Parity Results](image)

---

**Multiplier Results**

![Graph showing Multiplier Results](image)
Symbolic Regression

- Two problems:
  - $x^6 - 2x^4 + x^2$
  - $x^6 - 2x^3 + x$

- Function set:
  - +, -, *, / (protected)

- Fitness Function:
  - Absolute error over all fifty points in the input set
  - Solution found when absolute error is within 0.01 of each point

- Results averaged over fifty independent runs

---

Self-modifying Cartesian Genetic programming

- A developmental form of Cartesian Genetic Programming (CGP)
  - Includes self modification functions.
  - ‘General purpose’ GP system
  - Phenotype can vary over time (with iteration)
  - Can switch off its own self-modification

---

Changes to CGP: relative addressing

- Replaced direct node addressing with relative addressing
  - Always use 1 row (not rectangular)
  - Connection genes say how many nodes back

---
Changes to CGP: Inputs

- Replace input calls with a function.
  - We call these functions INP, INPP, SKIPINP
- Pointer keeps track of ‘current input’.
  - Call to INP returns the current input, and moves the pointer to the next input.
- Connections beyond graph are assigned value 0.

Changes to CGP: Outputs

- Removed output nodes.
- Genotype specifies which nodes are outputs.
  - If no OUTPUT function then last active node is used
    - Other defaults are used in situations where the number of outputs does not match the number required

Changes to CGP: Arguments

- Nodes also contain a number of ‘arguments’.
  - 3 floating point numbers
  - Used in various self-modification instructions
  - Cast to integers when required

SMCGP Nodes

- In summary, each node contains:
  - Function type
  - Connections as relative addresses
  - 3 floating point numbers
Functions

- Two types of functions:
  - Computational
    - Usual GP computational functions
  - Self-modifying
    - Passive computational role (see later)

Some Self-Modification Functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Parameters: use node address and the three node arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>Start, End, Insert</td>
<td>Moves each of the nodes between Start and End into the position specified by Insert</td>
</tr>
<tr>
<td>DUP</td>
<td>Start, End, Insert</td>
<td>Inserts copies of the nodes between Start and End into the position specified by Insert</td>
</tr>
<tr>
<td>DELETE</td>
<td>Start, End</td>
<td>Deletes the nodes between Start and End indexes</td>
</tr>
<tr>
<td>CHF</td>
<td>Node, New Function</td>
<td>Changes the function of a specified node to the specified function</td>
</tr>
<tr>
<td>CHC</td>
<td>Node, Connection1, Connection2</td>
<td>Changes the connections in the specified node</td>
</tr>
</tbody>
</table>

SMCGP Execution

- Important first step:
  - Genotype is duplicated to phenotype.
  - Phenotypes are executed:
    - Self modifications are only made to the phenotype.

Self Modification Process: The To Do list

- Programs are iterated.
- If triggered, self modification instruction is added to a To Do list.
- At the end of each iteration, the instructions on this list are processed.
- The maximum size of the To Do list can be predetermined
Computation of a SM node

- Functions are appended to the To Do list if:
  - The first input > the second input.
- And:
  - The To Do list isn’t too big.

Publications using SMCGP

- General Parity Problem (CEC 2009)
- Mathematical Problems (EuroGP 2009, GECCO 2007)
- Learning to Learn (GECCO 2009)
- Generating Arbitrary Sequences (GECCO 2007)
- Computing the mathematical constants pi and e (GECCO 2010 in GDS track)
- General adder and many other problems (GPEM Tenth Anniversary Special Issue, 2010)

Evolving Parity

- Each iteration of program should produce the next parity circuit.
  - On the first iteration the program has to solve 2 bit parity. On the next iteration, 3 bit ... up to 22 parity
  - Fitness is the cumulative sum of incorrect bits
- Aim to find general solution
- CGP or GP cannot solve this problem as they have a finite set of inputs (terminals)

Parity results: SMCGP versus CGP and ECGP
Scaling behaviour of SMCGP

A evolved general solution to parity

Evolving pi

- Iterate a maximum of 10 times
- If program output does not get closer to pi at the next iteration, the program is stopped and large fitness penalty applied
- Fitness at iteration $i$, is absolute difference of output at iteration $i$ and pi
- One input: the numeric constant 1.

Evolving pi: an evolved solution

- Evolved solution
  
  $$f(i) = \begin{cases} 
  \cos(\sin(\cos(0))) & i = 0 \\
  f(i-1) + \sin(f(i-1)) & i > 0 
  \end{cases}$$

- $f(10)$ is correct to the first 2048 digits of pi
- It can be proved that $f(i)$ rapidly converges to pi in the limit as $i$ tends to infinity
Further results

- Mathematically provable results found so far:
  - Evolved a program that can carry out the bitwise addition of an arbitrary number of inputs
  - Evolved a sequence that converges to $e$

- Other results
  - Evolved a sequence function that generates the first 10 Fibonacci numbers
  - Evolved a power function $x^n$
  - Bioinformatics classification problem (finite inputs)
    - SMCGP performed no worse than CGP

Other Developmental CGP

- Various types of CGP inspired by biological development, graph re-writing and neuro-development have been devised
  - Cellular developmental (Miller 2003, 2004)
  - Graph re-writing (Miller 2003)

Bio-inspired developmental CGP

- Various types of CGP inspired by biological development, graph re-writing and neuro-development have been devised
  - Cellular developmental (Miller 2003, 2004)
  - Graph re-writing (Miller 2003)

Graph-rewriting CGP
Neuro-inspired developmental CGP

- Real neurons are immensely complex
- Like other cells, they replicate and change
- They also have complex electrical behaviour and communicate with each other
- Most learning occurs through morphological changes (new neurons, dendritic, axonal growth and new synapses)
- In neuro-inspired developmental CGP a neuron has been represented by seven CGP chromosomes
  - Applied to wumpus world, checkers, maze solving

Three ‘electrical’ CGP chromosomes

- Soma potential
- Updated soma potential
- Axonovasal electrical CGP program
- Potential of connected dendrite branches
- Synapse potential
- Potential of neighbouring dendrite branches
- Average potential of connected dendritic branches

Three developmental CGP chromosomes

- Branch Weight
- Soma Weight
- Updated Branch Weight
- Updated Soma Weight
- Branch Health
- Soma Health
- Updated Branch Health
- Updated Soma Health
- Resistance: determines whether branches grow or shrink
- Health: decides whether component will replicate, stay the same or die
- Weights: used in electrical programs
One weight adjustment chromosome

Cyclic CGP

- When outputs are allowed to connect to inputs through a clocked delay (flip-flop) it is possible to allow CGP to include feedback.
- By feeding back outputs generated by CGP to an input, it is possible to get CGP to generate sequences
  - In this way iteration is possible
- This has not, so far, been studied in any detail

Applications of CGP

- Digital Circuit Design
  - ALU, parallel multipliers, digital filters, analogue circuits
- Mathematical functions
  - Prime generating polynomials
- Control systems
  - Maintaining control with faulty sensors, helicopter control, simulated robot controller
- Image processing
  - Image filters
  - Mammary Tumour classification
- Bio-informatics
  - Molecular Post-docking filters
- Artificial Neural Networks
- Developmental Neural Architectures
  - Wumpus world, checkers, maze solving
- Evolutionary Art
- Artificial Life
  - Regenerating ‘organisms’
- Optimization problems
  - Applying CGP to solve GA problems

CGP Resources

- 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/
- CGP Book is in preparation, published by Springer
Conclusions

- Cartesian Genetic Programming is a graph based GP method.
- Genetic encoding is compact, simple and easy to implement and can handle multiple outputs easily.
- The unique form of genetic redundancy in CGP makes mutational search highly effective.
- The effectiveness of CGP has been compared with many other GP methods and it is very competitive.

CGP Bibliography