PPSN 2014 Tutorial: Cartesian Genetic Programming

Julian F. Miller
Dept of Electronics
University of York, UK
julian.miller@york.ac.uk
Abstract

Cartesian Genetic Programming (CGP) is an increasingly popular and efficient form of Genetic Programming. Cartesian Genetic Programming is a highly cited technique that was developed by Julian Miller in 1999 and 2000 from some earlier joint work of 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. Graphs are very useful program representations and can be applied to many domains (e.g. electronic circuits, neural networks). In a number of studies, CGP has been shown to be comparatively efficient to other GP techniques. It is also very simple to program.

Since then, the classical form of CGP has been developed made more efficient in various ways. Notably by including automatically defined functions (modular CGP) and self-modification operators (self-modifying CGP). SMCGP was developed by Julian Miller, Simon Harding and Wolfgang Banzhaf. It uses functions that cause the evolved programs to change themselves as a function of time. Using this technique it is possible to find general solutions to classes of problems and mathematical algorithms (e.g. arbitrary parity, n-bit binary addition, sequences that provably compute pi and e to arbitrary precision, and so on).

This tutorial is will cover the basic technique, advanced developments and applications to a variety of problem domains. The first edited book on CGP was published by Springer in September 2011. CGP has its own dedicated website http://www.cartesiangp.co.uk
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 (closely related to CGP)
  - Grammatical Evolution, Ryan 1998
  - Lots of others…
Origins of Cartesian Genetic Programming (CGP)

  - First actual mention of the term *Cartesian Genetic Programming* appeared at GECCO in 1999.
- 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

- **r** rows
- **c** columns
- **m** outputs
- **n** inputs

**Note:** Nodes in the same column are not allowed to be connected to each other
Allelic constraints for directed acyclic graphs

All function genes $f_i$ must takes 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)

$$j \geq l \quad n + (j-l)r \leq C_{ij} \leq n + jr$$

$$j < l \quad 0 \leq C_{ij} \leq n + jr$$

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

$$0 \leq 0_i \leq n + cr -1$$
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

Connection genes

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

Output genes
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)

0 0 1 1 0 0 1 3 1 2 0 1 0 4 4 2 5 4 2 5 7 3
So what does the graph represent?

\[ y_2 = x_0 + x_1 \]
\[ y_5 = x_0 \times x_1 \]
\[ y_7 = -x_0 \times x_1^2 \]
\[ y_3 = 0 \]
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
- Arity of functions = 1
- 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>Genotype length $n$</th>
<th>phenotype length $k$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
</tr>
<tr>
<td>7</td>
<td>720</td>
</tr>
<tr>
<td>8</td>
<td>5040</td>
</tr>
<tr>
<td>9</td>
<td>40320</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

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

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

// determine nodes needed
p = L - 1
dq = 0
do
    if (ToEvaluate[p])
        x = Node[p].Connection1
        y = Node[p].Connection2
        ToEvaluate[x] = true
        ToEvaluate[y] = true
        q = q + 1
        NodesUsed[q] = p
    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
for p = I to p < q + I
    x = Node[NodesUsed[p]].Connection1
    y = Node[NodesUsed[p]].Connection2
    z = Node[NodesUsed[p]].Function
    NodeOutput[p] = ComputeNode(NodeOutput[x], NodeOutput[y], z)
endfor

Decoding CGP chromosomes is easy
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.
  - Of course, it can also be done probabilistically.

// Decide how many genes to change: num_mutations
while (mutation_counter < num_mutations)
{
    get random 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
}
A new parameter less mutation procedure

- Goldman and Punch 2013, Eurogp2013 (see refs)
- Exactly one active gene is mutated for all offspring.
- Active genes will be mutated more frequently than inactive
  - Zero or more inactive genes can be mutated
- No mutation rate is required!

```java
//mutate randomly until active gene changed: single active strategy
gene_is_active = false
do {
    get random 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
    if (gene is active) gene_is_active = true
} while (gene_is_active = false)
```
Single active gene mutation strategy: results (Goldman and Punch 2013)

- **3bit parallel multiplier**
  - Multiples two three-bit numbers in parallel
  - Hard problem

- **Normal** = standard CGP
- **Skip**: set offspring’s fitness to parent if identical
- **Accumulate**: apply mutation operator until an offspring is generated with some active gene changed.
- **Single** 29% less real-computation than Normal!

![Graph showing comparison between Normal, Skip, Accumulate, and Single strategies across mutation rates.](image)
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, Walker, Völk, Smith, Miller, 2009*)
- 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

\[ \begin{align*}
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
\text{Original} & : \quad x_0 + x_1 \\
\text{Silent mutations} & : \\
Silent mutations and their effects

After silent mutation
Non-silent mutations and their effects

Original
Non-silent mutations and their effects

After active mutation
Neutral search is fundamental to success of CGP

- A number of studies have been carried out to indicate the importance to neutral search
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 disabled neutral mutations.
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 phenotype length for the initial population contrasted with the average phenotype length at conclusion of evolutionary run versus genotype length with 1% mutation

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!!
How big should the genotype be?

- Even parity with gate set \{AND, OR, NAND, NOR\}.
- Mutation type: probabilistic
- Mutation probability: 0.03
Modular/Embedded CGP (Walker, Miller 2004, 2008)

- 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
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
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

❖ Control excessive code growth
  • Genotype can return to original length at any time
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
    - Uses only 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
- Results heavily dependent on the maximum number of nodes allowed. Much better results are obtained when larger genotype lengths are used.
CGP versus Modular CGP?

- In work published by Walker and Miller (IEEE Trans. 2008) it was shown that Modular CGP appeared to outperform standard CGP for harder problems. Here are some results.
Even Parity Results

The graph shows the results for even parity with various bit lengths ranging from 3-bit to 8-bit. Different methods are compared, including CGP, M-CGP(5), GP, ADF, EP, and EP ADF. The x-axis represents the parity bit length, and the y-axis shows the count of occurrences. Each method is represented by a different color, allowing for easy comparison across bit lengths.
CGP versus Modular CGP? An experimental flaw

- When you compare MCGP with CGP one must ensure that the *maximum* number of primitive function nodes available to both approaches is the same
  - Because maximum genotype length is a highly important factor in the effectiveness of the evolutionary search
  - This was not done. Here are some indicative results comparing CGP with modular CGP when they do have the same maximum number of primitive function nodes

100 nodes (max)  500 nodes (max)
Self-modifying Cartesian Genetic programming

- A developmental form of CGP
  - Includes self modification functions in addition to computational functions
  - ‘General purpose’ GP system
  - Phenotype can vary over time (with iteration)
  - Can switch off its own self-modification

- Some representational changes from classic CGP…
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: summary

- Each node contains:
  - Function type
  - Connections as relative addresses
  - 3 floating point numbers
SMCGP: 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 can be appended to the To Do list under a variety of conditions
  - If active
  - If $\text{value(first input)} > \text{value(second input)}$.
- 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)
- General adder and many other problems (GPEM Tenth Anniversary Special Issue, 2010)

Authors: Harding, Miller, Banzhaf
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
  - Solutions can be proved to general
    - See GPEM 2010 paper

- 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
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.
An evolved solution

\[ f(i) = \begin{cases} 
\cos(\sin(\cos(\sin(0)))) & i = 0 \\
 f(i - 1) + \sin(f(i - 1)) & i > 0 
\end{cases} \]

- 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

❖ Other 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 (probably general)
  • Evolved a power function $x^n$
  • Bioinformatics classification problem (finite inputs)
    – SMCGP performed no worse than CGP
Two dimensional SMCGP (SMCGP2)

- Harding, Miller and Banzhaf 2011
- SMCGP2: genes
  - Function
  - Connections
  - Numeric Constant
- Arguments are now 2 D vectors
  - SM size (SMS)
  - SM location (SML)
SMCGP2: Vector relative addressing and Empty nodes

- There are **empty nodes** are represented by X.
- The relative address from C to B is (2, 1)
  - meaning 2 nodes to the left, and one node up.
- The relative address of C to A is (4,1).
- Note how the empty nodes are not counted when computing how many nodes back to connect.
SMCGP2: Self Modifying Functions

- Simplified SM function set
  - Duplicate section, insert elsewhere.
  - Duplicate section, overwrite elsewhere.
  - Crop to a section.
  - Delete a section.
  - Add a row or column.
  - Delete a row or column.
  - NULL
SMCGP2: Solving even-n parity

Time

n = 2  n = 3  n = 4  n = 5  n = 12
SMCGP2 versus SMCGP: Results

❖ Parity
  • Two functions sets used:
    – FULL: All 2-input Boolean functions used
    – REDUCED: only AND, OR, NAND, NOR used
  • SMCGP2 solves general parity 6.3 times faster than SMCGP using the FULL functions set but is slower for the REDUCED function set

❖ N bit binary adder
  • SMCGP2 solves it approximately 6 times faster than SMCGP
Multi-type CGP (MT-CGP)

Harding, Graziano, Leitner, Schmidhuber 2011

- Genotype pretty much classic CGP
  - Genotype is a (partly connected, feed-forward) graph
  - Graph is a list of nodes
    - Each node contains:
      - Function (from a function set)
      - Two connections (to other nodes)
      - real number (to use for parameters)

- Handles multiple data types
  - So far: reals and vectors

- Adds lots of functionality
  - List processing, statistical, specialist domain specific
MT-CGP: Example

Evolved program:

- **Inputs:**
  - (1,2,3,4)
  - 11
  - (5,6,7,8)

- **Evolved program nodes:**
  - **INPUT**
  - **ADD**
  - **SUM**
  - **HEAD**
  - **ADD**
  - **OUTPUT**

- **Output value of each node:**
  - (1,2,3,4) 11 (12,13,14,15) 54 1 55 55
MT-CGP

- Has a *big* function set
- Trying to incorporate *domain knowledge*
  - Easy to add new functions to help with a particular problem
- Functions deal with multiple data types
  - Functions are *overloaded*
  - Attempts are made at *human readable consistency*
- Evaluated on a suite of classification problems and is competitive with other methods
  - Can produce simple human readable classifiers
Application 1: Digital circuit synthesis with CGP

- Digital Circuits with hundreds of variables can be optimized using CGP (Vassicek and Sekanina 2011)
  - Won the $3000 silver award in human competitive workshop at GECCO 2011
- The method employs a SAT solver to identify whether two circuits are logically equivalent
  - In many cases this can be done in polynomial time
If C1 and C2 are not functionally equivalent then there is at least one assignment of the inputs for which the output of G is 1.
CGP for optimizing conventionally synthesized circuits

The seed for CGP is provided by using the logic synthesis package, ABC ([http://www.eecs.berkeley.edu/~alanmi/abc/](http://www.eecs.berkeley.edu/~alanmi/abc/)).

The fitness function is as follows:

- Use a **SAT solver** to decide whether candidate circuit $C_i$ and reference circuit $C_1$ are functionally equivalent.
  - If so, then $\text{fitness}(C_i) = \text{number of nodes} - \text{number of gates in } C_i$;
  - Otherwise: $\text{fitness}(C_i) = 0$. 
Application 2: Evolving Image Filters with CGP

- Detecting/locating objects with the iCub cameras
- Done by evolving image filters that take a camera image, and return only the objects of interest

Input → Evolved filter → Target
Input data

Image from camera

Grey

Red

Green

Blue

Hue

Saturation

Luminosity

Evolved filter

Split colour image is used as inputs
Genotype representation (like SMCGP but no SM functions)
## Large Function Set

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>LOG</td>
<td>TRIANGLES</td>
</tr>
<tr>
<td>INP</td>
<td>MAX</td>
<td>LINES</td>
</tr>
<tr>
<td>INPP</td>
<td>MIN</td>
<td>SHIFTDOWN</td>
</tr>
<tr>
<td>SKIP</td>
<td>EQ</td>
<td>SHIFTUP</td>
</tr>
<tr>
<td>ADD</td>
<td>GAMMA</td>
<td>SHIFLEFT</td>
</tr>
<tr>
<td>SUB</td>
<td>GAUSS</td>
<td>SHIFTRIGHT</td>
</tr>
<tr>
<td>CONST</td>
<td>SOBELX</td>
<td>SIFTα</td>
</tr>
<tr>
<td>MUL</td>
<td>SOBELY</td>
<td>GABOR</td>
</tr>
<tr>
<td>ADDC</td>
<td>AVG</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>SUBC</td>
<td>UNSHARPEN</td>
<td>RESCALE</td>
</tr>
<tr>
<td>MULC</td>
<td>THRESHOLD</td>
<td>GRABCut</td>
</tr>
<tr>
<td>ABSDIFF</td>
<td>THRESHOLDBW</td>
<td>MINVALUE</td>
</tr>
<tr>
<td>CANNY</td>
<td>SMOOTHMEDIAN</td>
<td>MAXVALUE</td>
</tr>
<tr>
<td>DILATE</td>
<td>GOODFEATURESTOTRACK</td>
<td>AVGVALUE</td>
</tr>
<tr>
<td>ERODE</td>
<td>SQUARES</td>
<td>RESCALE</td>
</tr>
<tr>
<td>LAPLACE</td>
<td>CIRCLES</td>
<td>RESIZETHENGABOR</td>
</tr>
</tbody>
</table>
Fitness

- Fitness = sum of mean square error of pixel values between each input/target
public class MyEvolvedFilter : GpImageFilterRunner
{
    public override GpImage RunFilter() {
        GpImage node0 = InputImages[0];
        GpImage node1 = InputImages[1];
        GpImage node2 = node0.erode(1);
        GpImage node3 = node2.ShiftDown();
        GpImage node4 = node0.absdiff(node2);
        GpImage node5 = node0.avg(node2);
        GpImage node7 = node5; // NOP
        GpImage node8 = node1.erode(3);
        GpImage node9 = node3.sub(node8);
        GpImage node12 = node4.add(node4);
        GpImage node13 = node7.min(node12);
        GpImage node16 = node13.absdiff(node9);
        GpImage node24 = node16.sub(node9);
        GpImage node50 = node24.gauss(15);
        GpImage node78 = node50.gauss(15);
        GpImage node89 = node78.threshold(64);
        GpImage node99 = node89.gauss(15);
        return node99;
    }

    public override void SetUsedInputs() {
        this.UsedInputs.Add(1);
        this.UsedInputs.Add(0);
    }
}
Evolved Filter Dataflow
Things we can do already:

- Generate different filters for other objects.
  - Recently, allowing icub to detect its fingers (Leitner et al 2013)
- Find fast running filters.
- Find them quickly.
- Show that filters are robust.
- Transfer code from offline learning to yarp module.
  - Software emits C# and C++ code
  - Running on Windows/Linux/Mac.
Tea-box filter: demonstration
Application 3: CGP encoded Artificial Neural Networks (CGPANN)

- CGP has been used to encode both feed-forward ANNs and recursive ANNs. The nodes genes consist of:
  - Connection genes (as usual)
  - Function genes
    - Sigmoid, hyperbolic tangent. Gaussian
  - Weights
    - Each connection gene carries a real-numbered weight

- Pole balancing, Arm Throwing
  - Very competitive results with other TWEANN methods (*Khan, Khan and Miller 2010, Turner and Miller 2013*)

- Breast cancer detection (*Ahmad et al 2012, Turner and Miller 2013*)
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
- There are a couple of publications using iteration in CGP (Khan, Khan and Miller 2010, Walker, Liu, Tempesti, Tyrrell 2010, Minarik, Sekanina 2011)
Recurrent CGP

- By allowing nodes to receive inputs from the right, CGP can be easily extended to encode recursive computational structures.
- Recurrent CGP Artificial Neural Networks can be explored in this framework.
- Only just begun to be explored (in 2014).
Recurrent CGP: Details

- Probability of recursive links controlled by a user-defined parameter recurrent connection probability (rcp)

- Decoding
  1. set all active nodes to output zero
  2. apply the next set of program inputs
  3. update all active nodes once from program inputs to program outputs
  4. read the program outputs
  5. repeat from 2 until all program input sets have been applied
Recurrent CGP: publications

- There are a three publications using recursion in CGP

- *Turner, Miller  EuroGP2014*
  - Looked at why CGP does not bloat: disproves Neutral genetic drift or length bias as the reasons

- *Turner, Miller  PPSN2014 – here*
  - Introduces recurrent CGP applies it to partially observable task: artificial ant and sunspot prediction

- *Turner, Miller YDS2014*
  - Applies classic and recurrent and to create equations that predict famous mathematical sequences
A general method for applying CGP to GA problems

- Choose fixed constants in interval [-1, 1]
- Choose CGP function nodes to be mathematical functions operating on numbers in interval [-1, 1]
- Choose as many outputs as you need to define solution vector
- Linearly map outputs to problem domain
- This generic approach that can solve many problems
  - Two papers here at PPSN 2014 which use this technique (TSP: Klegg et al, Classification: Mohid et al)
CGP acceleration (Vassicek and Slany 2012)

- CGP decoding step is replaced with native machine code that directly calculates response for a single training vector.
- Requires little knowledge of assembly language or target machine code.
- Integration of the machine code compiler requires modifying only a few lines of code.
- Achieves 5 times speedup over standard implementation.
Applications of CGP

- Circuit Design
  - ALU, parallel multipliers, digital filters, analogue circuits, circuit synthesis and optimization
- Machine Learning
  - classification
- Mathematical functions
  - Prime generating polynomials
- Control systems
  - Maintaining control with faulty sensors, helicopter control, general control, simulated robot controller
- Image processing
  - Image filters
  - Mammary Tumour classification
- Robotics
  - gait
- 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 I:
http://www.cartesiangp.co.uk

- **Julian Miller**: C implementations of CGP and SMCGP available at http://www.cartesiangp.co.uk
- **Andrew Turner**: Easy to use, highly extendable, C implementation that includes CGPANNs http://www.cgplibrary.co.uk/
- **Eduardo Pedroni**: Java implementation with GUI https://bitbucket.org/epedroni/jcgp/downloads
- **Zdenek Vassicek**: Highly optimised C/Machine Code implementation http://www.fit.vutbr.cz/~vasicek/cgp/
- Cartesian Genetic Programming book
  - Published in 2011 by Springer
CGP Resources II:

- **David Oranchak** has implemented CGP in Java. Documentation is available at http://oranchak.com/cgp/doc/

- **Brian Goldman** has implemented CGP in Python https://github.com/brianwgoldman/ReducingWastedEvaluationsCGP

- **Jordan Pollack** has implemented symbolic regression in CGP with Matlab
  - See CGP web site

- **Lawrence Ashmore** has implemented a Java evolutionary art package using CGP
  - See CGP web site
Conclusions

- Cartesian Genetic Programming is a graph based GP method capable of representing many computational structures
  - programs, circuits, neural networks, systems of equations…

- 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.

Ashmore L. An investigation into cartesian genetic programming within the field of evolutionary art. http://www.emoware.org/evolutionary_art.asp, Department of Computer Science, University of Birmingham (2000)


DiPaola S., Gabora L. Incorporating characteristics of human creativity into an evolutionary art algorithm, Genetic Programming and Evolvable Machines (2009) Vol. 10. For further info see: http://dipaola.org/evolve/


