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...

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)
- The genotype is a list of integers that represent the program primitives and how they are connected together
  - CGP represents programs as **graphs** in which there are **non-coding genes**

Types of CGP

- Classic
- Modular
- Self-modifying
- Developmental
- Cyclic
CGP General form

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

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

---

CGP genotype

- **function genes**
- **Output genes**
- **Connection genes**

Usually, all functions have as many inputs as maximum function arity.
Unused connections are ignored (see later).

---

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
```

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
```
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)

So what does the graph represent?

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

The role of the geometric parameters: rows, columns and level-back

- Tall and thin graphs
- Short and wide graphs
- Layered graphs (levels-back =1)
- Less layered graphs (levels-back =3)
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

Allelic constraints

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)

For $j \geq l$

\[ n + (j-1)r \leq C_{ij} \leq n + jr \]

For $j < l$

\[ 0 \leq C_{ij} \leq n + jr \]

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

\[ 0 \leq o_i \leq n + cr -1 \]

Non-coding genes in CGP

- Contains active and inactive regions (rather than coding or non-coding)
- Mutations can make active genes become inactive and inactive genes become active
- A single gene change can thus cause large phenotypic changes
- When a gene is changed by mutation several things can happen
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 (as in slide 14).

```
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
}
```

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 – see later (Walker, Miller Cavill 2006)
- Recent work using a floating point representation of CGP has suggested that crossover might be useful (Clegg, Walker, Miller 2007)

Program changes caused by mutations

<table>
<thead>
<tr>
<th>Gene was</th>
<th>Gene is</th>
<th>Genotypic change</th>
<th>Phenotypic change</th>
<th>Fitness change</th>
</tr>
</thead>
<tbody>
<tr>
<td>silent</td>
<td>silent</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>active</td>
<td>silent</td>
<td>Yes</td>
<td>Yes</td>
<td>Likely</td>
</tr>
<tr>
<td>silent</td>
<td>active</td>
<td>Yes</td>
<td>Yes</td>
<td>Likely</td>
</tr>
<tr>
<td>active</td>
<td>active</td>
<td>Yes</td>
<td>Yes</td>
<td>Likely</td>
</tr>
</tbody>
</table>

When genetic changes occur without any fitness change it is often referred to a neutral change.

The very interesting aspect is that in CGP most neutral change occurs externally to the phenotype, so it does not have to be processed in any fitness calculation (unlike many other forms of GP)

Silent mutations and their effects

Original

No change in phenotype but it changes the programs accessible through subsequent mutational change
Non-silent mutations and their effects

Massive change in phenotype is possible through simple mutation

Evolutionary Strategy

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

Neutral search is fundamental to success of CGP

- A number of studies have been carried out to indicate the importance to neutral search (Miller and Thomson 2000, Vassilev and Miller 2000, Yu and Miller 2001, Miller and Smith 2006)

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.

---

Computational effort versus Genotype length and mutation rate

Evolutionary search is most effective at low mutation rates and large genotype lengths. The larger the genotype length, the lower should be the value chosen for mutation rate.

---

Minimum Computational Effort (over all mutation rates) versus genotype length (in nodes)

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

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

So provided you choose the ‘best’ mutation rate, problems are more easily solved with large genotypes. 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

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
- 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
  - 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) \(\rightarrow\) module (type II)
  - Module (type II) \(\rightarrow\) module (type II)
  - Module (type II) \(\rightarrow\) 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 (200 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.
Even Parity Results

Digital Adder

- Three digital adder problems:
  - 1-bit, 2-bit, and 3-bit

- Function set:
  - AND, NAND, OR, NOR

- Fitness Function:
  - Number of phenotype output bits that differ from the perfect n-bit digital adder solution
  - Perfect solution has a fitness of zero

Adder Results

Digital Multiplier

- Two digital multiplier problems:
  - 2-bit and 3-bit

- Function set:
  - AND, AND (on 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
Multiplier Results

Symbolic Regression

- Two problems:
  - $x^6 - 2x^4 + x^2$
  - $x^5 - 2x^3 + x$

- 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

Symbolic Regression Results

Lawnmower Problem

- Guide a lawnmower around a lawn cutting the grass
  - Lawn divided into $n \times m$ squares
  - Cuts all the grass in a square, when the square is visited
  - Starts in the centre square
  - If the lawnmower leaves one side, it reappears on the opposite side

- Problem solved when all squares have been visited
Lawnmower Problem Results

Parameter Sweeps

Multi-chromosome Approach

- A multi-chromosome genotype is divided up into \( n \) equal length sections called "chromosomes"
  - Each chromosome contains an equal number of nodes
- The no. of chromosomes \( (n) \) is dictated by the no. of outputs of the given problem
  - Each chromosome has a single output
- The entire problem is still represented in a single genotype

Multi-chromosome CGP Example

- A node in a chromosome can connect to:
  - A program input
  - The output of a previous node in the SAME chromosome
- Creates a form of compartmentalisation in the genotype
  - Removes any connections between the smaller problems in each chromosome
Multi-chromosome (1 + 4) ES

• Calculate fitness for each chromosome
• Select best chromosome from each position
  – If tied, choose offspring over parent
• Promoted individual consists of the best chromosomes
  – May not have been in the original population
  – May have a higher overall fitness than parents

Multi-chromosome experiments and Parameters

• Adder †
  – 2-bit (3 chromosomes)
  – 3-bit (5 chromosomes)
• Multiplier *
  – 2-bit (4 chromosomes)
  – 3-bit (6 chromosomes)
• De-multiplexer †
  – 3:8-bit (8 chromosomes)
• Comparator †
  – 4 x 1-bit (18 chromosomes)
• Arithmetic Logic Unit *
  – 3-bit (17 chromosomes)

• Each chromosome contained 100 nodes
  (300 genes)
• Function set 1 (‘)
  – AND, AND (one input inverted), XOR, OR
• Function set 2 (†)
  – AND, NAND, OR, NOR

Results

Modules within Modules?

• Currently only allow primitive functions in modules
  – Single level hierarchy
• Allow modules within modules
  – Multi-level hierarchy
  – Produce larger building blocks
  – Improve performance
  – Evolve solutions to larger, more complex problems
• ADFs occur inside ADFs in GP, why not have modules inside modules?
Multi-level Hierarchy (Walker 2008)

- Introduce level types into modules
  - L1 → primitive functions
  - L2 → primitive functions, L1 modules
  - L3 → primitive functions, L1 and L2 modules
  - etc...
- Each level is created by the compress operator
  - User sets the number of levels in the hierarchy
- Nodes in a module can only be mutated to functions with a lower level type
- Nodes in a genotype can be mutated to a function of any level

Multi-level modular CGP Even Parity

Self-modifying CGP
(Harding, Miller and Banzhaf 2007)

- SMCGP is a form of developmental CGP
- To be more specific: a form of genetic programming where an individual’s phenotype can vary over time
  - It is iterated
Representational differences

- In CGP nodes connect explicitly
  - i.e. This node connects to node 12.
- In SMCGP nodes have a relative address
  - i.e. This node connects to one 4 nodes back.
  - Useful for moving pieces of cgp code around
- CGP node:
  - function & connections
- SMCGP node:
  - function, connections & 3 parameters.

Other representational differences

- Input/Outputs handled differently.
  - In SMCGP typically the last N-nodes in the graph are used as output nodes
- If a node addresses a node of a negative index, then this is mapped to an input (using modulo arithmetic)

Visualization

Self-modification

- In addition to functional nodes, SMCGP contains nodes that modify its own graph
  - For example, a function may add, delete or move a section of the program
- Self modification nodes pass the larger numerical input unchanged
- Phenotype is initially the same as the genotype graph, however
  - It is iterated, which with modification, causes it to diverge from the original graph
Self-modification process

1. Evaluate CGP graph
   • Get computational output
2. If a node is a modification node and it is activated, add to ‘ToDo’ list
   • Activated means: If the first input is greater or equal to value to the second input
3. When finished evaluating entire graph, parse ‘ToDo’ list.
4. Perform each operation to build modified graph for next iteration

Some SMCGP operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Parameters</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>DUPE</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>ADD</td>
<td>Insert, Count</td>
<td>Adds Count number of NOP nodes at position Insert</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>
<tr>
<td>CHP</td>
<td>Node, Parameter, New Value</td>
<td>Changes the specified parameter and a given node</td>
</tr>
<tr>
<td>FLR</td>
<td></td>
<td>Clears any entries in the pending modifications list</td>
</tr>
<tr>
<td>OVR</td>
<td>Start, End, Insert</td>
<td>Moves each of the nodes between Start and End into the position specified by Insert, overwriting existing nodes</td>
</tr>
<tr>
<td>DU2</td>
<td>Start, End, Insert</td>
<td>Similar to DUPE, but connections are considered to absolute, rather than relative</td>
</tr>
</tbody>
</table>

Example: Duplication

Example: Deletion and changing function
A special function can call another part of the graph as a procedure.
This section of graph could be made up of active nodes, or nodes neutral to the main graph.
Procedures can call other procedures.
Procedures can self modify.
Inputs to this procedure are the inputs to the calling node.

We limit the functional nodes to + and –.
The task is on each iteration (0,1,2,…) to produce the next number in a sequence.
– Here, we ask for the squares: 1,4,9,16,25 etc.
The only input was the iteration, i.
Fitness calculated by iterating from 0 to 9 and counting the longest sequence from zero that were correct.
Without self-modification this task is impossible.

Example: Generating Sequences

<table>
<thead>
<tr>
<th>Input, i</th>
<th>Evolved program</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$0 + i$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$0 + i$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$0 + i + i$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>$0 + i + i + i$</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>$0 + i + i + i + i$</td>
<td>16</td>
</tr>
</tbody>
</table>

Squares program

Evolving Digital Circuits

In this experiment we tackled the well-known problem of evolving circuits for solving even-parity.
We used a restricted function set, that is well studied in the literature:
AND OR NAND NOR
This set of functions make the problem very hard to solve.
Evolving Parity Circuits

<table>
<thead>
<tr>
<th>Number of inputs</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMCGP</td>
<td>28811</td>
<td>58194</td>
<td>191493</td>
<td>352901</td>
<td>583712</td>
</tr>
<tr>
<td>Speedup compared with MCGP</td>
<td>2.27</td>
<td>3.13</td>
<td>1.44</td>
<td>0.76</td>
<td>0.5</td>
</tr>
<tr>
<td>Speedup over CGP</td>
<td>2.84</td>
<td>5.04</td>
<td>4.88</td>
<td>8.53</td>
<td>10.13</td>
</tr>
</tbody>
</table>

Evolving big parity functions

- The largest parity problem solved to date with a direct GP approach appears to be 22 inputs
  - Although general solutions have been found.
  - The function set used includes many more bitwise operations – including XOR
- We attempted to produce a solution that could be iterated to find any size circuit

Evolving big parity functions

- The challenge:
  - Evolve an SMCGP program that solves 2-input even parity. After iterating the growth algorithm once, it should solve 3-input. After a second time, 4-input and so on
- We were able to evolve (and test) to 24 bit input. We think it is a general solution but haven’t verified this yet

SMCGP Conclusions

- Discussed a promising new variant of CGP that bridges the divide between artificial developmental systems and genetic programming
- This is done by directly producing a phenotype capable of performing a computation
- We have shown we can solve problems that cannot be solved by a conventional GP system.
- In other experiments we have shown that performance appears to be similar on problems where there is no inherent advantage for the self-modification
Developmental CGP

• Various types of CGP inspired by biological development, graph re-writing and neuro-develop have been devised
  – Biological developmental (Miller 2003, 2004)
  – Graph re-writing (Miller 2003)
  – Neuro-developmental (Khan, Miller and Halliday 2007, 2008)

Bio-inspired developmental CGP

Graph-rewriting CGP

Neuro-inspired developmental CGP
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.

GPU Implementation (Harding and Banzhaf 2007)

- A guaranteed maximum program length makes it easy to use CGP on more limited platforms.
- We have developed a version of CGP that runs on Graphics Processing Units
  - Limited program length
  - Memory constraints
  - But fast, parallel architecture

GPU Speed up on a regression problem
Image Processing

Image processing

22.4
Fitness

Applications of CGP

- Digital Circuit Design
  - ALU, parallel multipliers, digital filters
- Mathematical functions
  - Prime generating polynomials
- Control systems
  - Maintaining control with faulty sensors, helicopter control, simulated robot controller
- Image processing
  - Image filters
- Bio-informatics
  - Molecular Post-docking filters
- Developmental Neural Architectures
  - Wumpus world, checkers
- Evolutionary Art
- Artificial Life
  - Regenerating ‘organisms’
- Optimization problems
  - Applying CGP to solve GA problems
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
• The CGP method is still being developed (i.e. modular CGP, self-modifying CGP, neuro-developmental CGP)
• A method has been developed for CGP to output lists of numbers so that it can be applied to any problem that genetic algorithms can be applied to (see Walker and Miller 2007)

References

References


References

52. 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)

References