Genetic Programming: Language and Representation

Dr. Michael Lones
Room EM.G31
M.Lones@hw.ac.uk
How to make tree GP more expressive
- Memory, iteration and modularity

Approaches to modularity in GP
- ADFs
- Refactoring mutations
- Analytical approaches

Cartesian GP
- Graph-based GP
- Role of redundancy
Other ways of representing programs
▶ Linear GP: lists of instructions
▶ Grammatical evolution: lists of grammar transitions

Developmental genetic programming
▶ Growing programs and other structures
▶ Transitioning between representations
▶ Addressing scalability
▶ Computing more like biology
Tree GP doesn’t use standard languages

- Requires special interpreters
- Challenging to integrate with existing code
- Typically not Turing-complete
- Language features appear a little *ad hoc*

Other approaches do use conventional languages

- Machine languages (e.g. x86 assembler): **Linear GP**
- Imperative languages (e.g. C): **Grammatical evolution**
- Functional languages (e.g. Haskell): **PolyGP**
- Logic languages (e.g. Prolog): **DCTG-GP**
- Evolves lists of machine language instructions
  - So, works a lot like a genetic algorithm
  - Variation operators less likely to break syntax
  - Often no need for an interpreter or a compiler
  - However, semantic (e.g. runtime) errors are possible

```
push r1
mov r1, r2
mov 2, r1
mov r1, r2
pop r2
...
```

```
push r1
mov r1, r2
mov 2, r2
mov r1, r2
pop r2
...
```

*mutate*
Languages that have been targeted include:

- Sun SPARC assembler (RISC)
- Intel i386 assembler (CISC)
- Java bytecode
- CUDA instructions
- Novel hardware languages
- GP-specific languages, e.g. Push

FINCH [Orlov & Sipper 2010]

- Special crossover/mutation operators preserve type-compatibility and stack depth-compatibility:
Evolved bytecode programs can be converted to Java

- e.g. Koza’s intertwined spirals problem, which involves finding a classifier that can distinguish the points from two spirals:

```java
boolean isFirst(double x, double y) {
    double a, b, c, e;
    a = Math.hypot(x, y);  e = y;
    c = Math.atan2(y, b = x) +
        -(b = Math.atan2(a, -a))
        * (c = a + a) * (b + (c = b));
    e = -b * Math.sin(c);
    if (e < -0.0056126487018762772) {
        b = Math.atan2(a, -a);
        b = Math.atan2(a * c + b, x);  b = x;
        return false;
    }
    else
        return true;
}
```
Protein localisation [Heddad et al. 2004]

Predicting which part of a cell a protein will be located in based upon patterns in its amino acid sequence

Amino acid sequence: GIVEQCCTSICSLYQLENYN...

```c
void gp(double *r, char *seq)
{
    r[3] = match("SA", seq);
    r[4] = match("[RK]R", seq);
    r[5] = match("[P]GP", seq);
    r[6] = match("[FWV]", seq);
    r[10] = match("[QS]", seq);
    r[0] = r[4] - r[8];
    r[0] = r[10] + r[0];
    r[0] = r[4] * r[0];
}
```
◫ A language designed for GP  [Spector 2001]
  ▶ http://faculty.hampshire.edu/lspector/push.html
  ▶ A stack-based language, with one stack per type
  ▶ Handles syntax and semantics in a flexible manner
  ▶ Code stack can be used for modularity
  ▶ Turing complete, relatively good evolvability

int.+  # 52 + 10, 62 goes on int stack
code.do  # calls float.dup (duplicate)
float.*  # 3.0 * 3.0, 9.0 on float stack
...

Typed stacks
Grammatical Evolution (GE)

- Evolves lists of grammar transitions [Ryan 1998]
  - Programming language is expressed by a grammar
  - Described in Backus Naur form
  - GE can then evolve any program in that grammar

Problem we want to solve

Grammar to express solution in

List of grammar transitions that construct a program
<exp> ::= 0<exp><op><exp> | 1(<exp><op><exp>) | 2<pre-op>(<exp>) | 3<var>
<op> ::= 0AND | 1OR | 2XOR
<pre-op> ::= 0NOT
<var> ::= 0IN1 | 1IN2 | 2IN3

Evolved chromosome: 10 18 41 27 6 39 21 47 13 31 19 8
Evolved chromosome:

For the first step, start with the most general term

Apply transition 2 to the current term

Number of transitions in current grammar rule

10 % 4 = 2

10 | 18 | 41 | 27 | 6 | 39 | 21 | 47 | 13 | 31 | 19 | 8
Evolved chromosome:

Number of transitions in current grammar rule

Apply transition 2 to the current term

For the first step, start with the most general term

Grammar:

\[
\begin{align*}
<\text{exp}> & ::= \ 0<\text{exp}><\text{op}><\text{exp}> \mid 1(\text{<exp><op><exp>}) \mid 2<\text{pre-op}>(\text{exp}) \mid 3\text{<var>}
\\
<\text{op}> & ::= \ 0\text{AND} \mid 1\text{OR} \mid 2\text{XOR}
\\
<\text{pre-op}> & ::= \ 0\text{NOT}
\\
<\text{var}> & ::= \ 0\text{IN1} \mid 1\text{IN2} \mid 2\text{IN3}
\end{align*}
\]
Evolved chromosome:

At each step, match the left-most unmatched term

Number of transitions in current grammar rule

Apply transition 0 to the current term

18 \mod 1 = 0

10 18 41 27 6 39 21 47 13 31 19 8
Grammar for expressions:

\[
\text{<exp>} ::= \begin{cases} 
0\text{<exp>\text{<op>\text{<exp>}}}, & 1(\text{<exp>\text{<op>\text{<exp>}}}), \\
2\text{pre-op}(\text{<exp>}), & 3\text{<var>}
\end{cases}
\]

\[
\text{<op>} ::= \begin{cases} 
0\text{AND}, & 1\text{OR}, \\
2\text{XOR}
\end{cases}
\]

\[
\text{<pre-op>} ::= \begin{cases} 
0\text{NOT}
\end{cases}
\]

\[
\text{<var>} ::= \begin{cases} 
0\text{IN1}, & 1\text{IN2}, & 2\text{IN3}
\end{cases}
\]

Evolved chromosome:

At each step, match the left-most unmatched term

18 mod 1 = 0

Number of transitions in current grammar rule

Apply transition 0 to the current term
<exp> ::= 0<exp><op><exp> | 1(<exp><op><exp>) | 2<pre-op><exp>) | 3<var>

<op> ::= 0AND | 1OR | 2XOR

<pre-op> ::= 0NOT

<var> ::= 0IN1 | 1IN2 | 2IN3

Evolved chromosome:

10 18 41 27 6 39 21 47 13 31 19 8

41 mod 4 = 1

NOT(IN1 AND (IN2 OR IN3)
Evolving Super Mario levels  [Shaker et al. 2012]

\[
\begin{align*}
\langle \text{level} \rangle & ::= \langle \text{chunks} \rangle \quad \langle \text{enemy} \rangle \\
\langle \text{chunks} \rangle & ::= \langle \text{chunk} \rangle \mid \langle \text{chunk} \rangle \langle \text{chunks} \rangle \\
\langle \text{chunk} \rangle & ::= \text{gap}(<x>,<y>,<w_g>,<w_{\text{before}}>,<w_{\text{after}}>) \\
& \mid \text{platform}(<x>,<y>,<w>) \\
& \mid \text{hill}(<x>,<y>,<w>) \\
& \mid \text{cannon}_{\text{hill}}(<x>,<y>,<h>,<w_{\text{before}}>,<w_{\text{after}}>) \\
& \mid \text{tube}_{\text{hill}}(<x>,<y>,<h>,<w_{\text{before}}>,<w_{\text{after}}>) \\
& \mid \text{coin}(<x>,<y>,<w_c>) \\
& \mid \text{cannon}(<x>,<y>,<h>,<w_{\text{before}}>,<w_{\text{after}}>) \\
& \mid \text{tube}(<x>,<y>,<h>,<w_{\text{before}}>,<w_{\text{after}}>) \\
& \mid \langle \text{boxes} \rangle \\
\langle \text{boxes} \rangle & ::= \langle \text{box\_type} \rangle (\langle x \rangle,\langle y \rangle) \mid \ldots \\
& \mid \langle \text{box\_type} \rangle (\langle x \rangle,\langle y \rangle)^6 \\
\langle \text{box\_type} \rangle & ::= \text{block}_{\text{coin}} \mid \text{block}_{\text{powerup}} \\
& \mid \text{rock}_{\text{coin}} \mid \text{rock}_{\text{empty}} \\
\langle \text{enemy} \rangle & ::= (\text{koopa} \mid \text{goompa})(\langle x \rangle) \mid \ldots \\
& \mid (\text{koopa} \mid \text{goompa})(\langle x \rangle)^{10} \\
\langle x \rangle & ::= [5..95] \quad \langle y \rangle ::= [3..5]
\end{align*}
\]
A flexible and expressive approach
- Since grammars can be defined for all languages
- Programs have been evolved in C, for example

Though modern languages are problematic
- Complicated syntax and large APIs
- Typically only a subset of a language is used

Some concerns about evolvability
- Sensitive to mutations at the left of a chromosome
- These have a large effect upon the final expression
Evolve something that constructs something else

- Grammatical evolution does this, in a limited sense
  i.e. evolve grammar transitions that construct a program
- You also saw it with genetic algorithms
  e.g. rule sets that construct bin packing solutions
Developmental GPs also use indirect encodings

- Evolve programs that construct other entities, e.g. structures, circuits, or other programs
- Often inspired by biological models of development
- Often more scalable, able to solve bigger problems

This is often referred to as a 'developmental process' or a 'genotype-phenotype mapping'
Using tree-based GP to design circuits  [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping
Using tree-based GP to design circuits  [Koza 2003]
- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping

Create three nodes in series
Using tree-based GP to design circuits  [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping
Using tree-based GP to design circuits  [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping

Add three-way link to ground

\[ V_{IN} \]
Using tree-based GP to design circuits  [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping

Diagram showing a GP tree with nodes and connections, including a node with two leads.
Using tree-based GP to design circuits [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping

Instantiate

\( V_{\text{IN}} \)  
105500\( \mu \)H
Using tree-based GP to design circuits [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping
Using tree-based GP to design circuits [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping
Using tree-based GP to design circuits [Koza 2003]

- This approach uses a GP tree to construct a graph
- An early example of a developmental mapping
Scalability is a problem for genetic programming

- Search space grows exponentially with program size
- Development allows programs to be compressed
- Particularly when solutions have repetitive structure:

Even-4-parity circuit:

![Even-4-parity circuit](http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html)

Even-8-parity circuit:
CGP programs that modify themselves  [Harding’11]
  ▶ Contains normal functions and self-modifying functions
    • E.g. delete nodes, duplicate nodes, change connection …
  ▶ A new program is created at each program iteration
    • Can generate general solutions, e.g. even-n-parity circuit:

Colours represent different functions in the CGP program

Step 1 solves for 2 inputs
Step 2 solves for 3 inputs
This program solves 12-input parity
Programs build their own offspring [Spector 2010]

- i.e. they include code to copy and modify themselves
- Based on the idea that programs can learn good patterns or regions of variation, for instance mutation hotspots
- e.g. Autopush: autoconstructive evolution using Push
- Modifies the program’s “code” stack to construct a child

```
(((integer.stackdepth (boolean.and code.map)) (integer.sub
  (integer.stackdepth (integer.sub (in (code.wrap (code.if (code.noop)
  Boolean.fromfloat (2) integer.fromfloat) (code.rand integer.rot)
  exec.swap code.append integer.mult)))))))
```
A way of transitioning between representations
  ▶ e.g. using a tree to represent a graph
  ▶ Choose an evolvable representation, rather than one that is required by the problem domain

Also a way of achieving scalability
  ▶ Genotype space can be smaller than phenotype space
  ▶ Human analogy: 30,000 genes encode 100 trillion cells

And potentially for achieving complexity
  ▶ Not limited to repetitive modular structures
Things you should know

- Linear GP and grammatical evolution
  - How programs are represented
  - When they should be used, strengths and weaknesses

- Developmental GP
  - The meaning of indirect/developmental representation
  - Advantages over non-developmental GPs

- I don’t expect you to know
  - Low-level details, e.g. translating a GE expression
  - Push, the different kinds of developmental GP
Biological representations of computation

- Most of the approaches discussed so far use conventional representations of computation
- Conventional representations are fragile
- Developmental GP hints at the potential of representing computation in unconventional ways

- *This is the topic of the next few lectures…*

- Next week: Cellular Automata
  - Complex behaviour from simple systems
Things to try out

◊ Have a look at Grammatical Evolution
  ▶ `ec.gp.ge` package in ECJ
    • See section 5.3

◊ Have a look at Push
  ▶ `ec.gp.push` package in ECJ
    • See section 5.4
S. Harding et al, SMCGP2: self modifying Cartesian genetic programming in two dimensions, GECCO 2011
http://www.cartesiangp.co.uk/papers/gecco2011a-harding.pdf

A. Heddad et al, Evolving Regular Expression-based Sequence Classifiers for Protein Nuclear Localisation, EvoBIO 2004

J. Koza et al., The Importance of Reuse and Development in Evolvable Hardware, 2003 NASA/DoD Conference on Evolvable Hardware, 2003
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5365


