

Lecture 5: Overview of Genetic Programming

Genetic Programming

- ▶ Genetic Programming (GP) is the field of EC pioneered by John Koza
 - ▶ Koza, J. Genetic Programming: On the Programming of Computers by Means of Natural Selection. 1992.
- ▶ GP is based on the following proposals
 - ▶ Many problems from various fields can be interpreted as the problem of discovering an appropriate computer program that maps some input to some output
 - ▶ I.e. program induction
 - ▶ GP is a general way to do program induction

The Program Induction Paradigm

- ▶ Many diverse problems can be viewed as program induction
 - ▶ Optimal control
 - ▶ Planning
 - ▶ Sequence induction
 - ▶ Symbolic regression
 - ▶ Empirical discovery
 - ▶ Decision tree induction
 - ▶ Evolution of emergent behaviour

Trees as Programs

- ▶ Programs can be represented with a commonly used data structure - Trees
 - ▶ E.g. Reverse Polish Notation expressions
 - ▶ $5\ 3 + 7 * 4 /$
 - ▶ E.g. LISP S-expressions
 - ▶ $(+ 1\ 2\ (IF\ (>\ TIME\ 10)\ 3\ 4))$
- ▶ Genetic Programming is GA with trees
 - ▶ The use of trees as chromosomes leads to some interesting differences with GAs
 - ▶ In particular, chromosomes are of variable length

Characteristics of Program Trees

- ▶ Program trees are built from a set of functions F and a set of terminals T
 - ▶ Functions in F have specific *arities*
 - ▶ E.g. binary functions such as + have arity 2
 - ▶ Ternary functions such as IF condition THEN action1 ELSE action2 have arity 3
 - ▶ Terminals in T are functions having arity 0
- ▶ There is a great variety of possible program trees, in fact there is an infinite variety
 - ▶ The number of possible recursive compositions of functions and terminals is infinite if we do not limit the tree's depth
 - ▶ In contrast the standard fixed chromosome-length encoding of most GAs give a finite (but possibly very large) number of possible chromosomes

Closure of the Function and Terminal Set

- ▶ For GP it is important that the function and terminal sets have the *closure* property
- ▶ Closure is the property that the functions in F accept any possible value that may be output from any composition of functions in F and terminals in T
 - ▶ This is easy to achieve for some function and terminal sets, such as boolean functions
 - ▶ More care is needed in most domains
 - ▶ E.g. with arithmetic functions care is needed to prevent division by zero
- ▶ If the closure property does not hold over $F \cup T$ then we are faced with a *constrained optimisation problem*
 - ▶ We can apply various techniques such as those we have seen in the GA lectures previously

Closure of the Function and Terminal Set

- ▶ Various strategies for ensuring closure are possible...
 - ▶ Implement protected versions of vulnerable operators
 - ▶ E.g. protected version of division that returns 1 or 'undefined' for division by zero
 - ▶ E.g. protected versions of log and $\sqrt{\quad}$ that work on the absolute value of their arguments
 - ▶ Combine different function types
 - ▶ E.g. combine numerical and boolean values in numerical logic (FALSE=0, TRUE \neq 0)
 - ▶ Implement conditional comparative operators
 - ▶ E.g. ILTZ (If Less Than Zero)
 - ▶ Implement conditional branching operators
 - ▶ E.g. evaluate one of several functions based on some state and return its value

Sufficiency of the Function and Terminal Set

- ▶ If GP is to make any progress in tackling a problem, the function and terminal set must be *sufficient* to find an appropriate program
- ▶ E.g. in boolean logic the sets $F = \{\wedge, \vee, \neg\}$ and $F = \{\wedge, \neg\}$ are sufficient to represent any boolean function, but the set $F = \{\wedge\}$ is not
- ▶ This is a universal problem in machine learning, sometimes called attribute selection
- ▶ Typically it is hard to select the minimal sufficient set in advance and it is necessary to include more functions than are strictly necessary

Population Initialisation

- ▶ As in a GA, in GP we need to initialise a population of individuals
- ▶ There are three basic ways of doing this
 - ▶ 'Grow' method
 - ▶ 'Full' method
 - ▶ 'Ramped half-and-half' method
- ▶ For the 'grow' and 'full' methods an individual is initialised recursively

```
Initialise(root, 1);
```

- ▶ As in GAs a 'no duplicates' policy may be implemented to avoid wasted computational effort

'Grow' Method

```
InitialiseG(node,depth)
```

```
switch depth do
```

```
  case 1
```

```
  | Uniformly randomly select node type from  $F$ ;
```

```
  case maxdepth
```

```
  | Uniformly randomly select node type from  $T$ ;
```

```
  case otherwise
```

```
  | Uniformly randomly select node type from  $F \cup T$ ;
```

```
end
```

```
if node type  $\in F$  then
```

```
  for  $n=1$  to arity of node type do
```

```
  | InitialiseG(child  $n$ ,depth+1);
```

```
  end
```

```
else
```

```
  return;
```

```
end
```

'Full' Method

```
InitialiseF(node,depth)  
if depth<maxdepth then  
  | Uniformly randomly select node type from F;  
  | for n=1 to arity of node type do  
  | | InitialiseF(child n,depth+1);  
  | end  
else  
  | Uniformly randomly select node type from T;  
  | return;  
end
```

'Ramped Half-and-Half' Method

- ▶ 'Ramped half-and-half' combines the 'full' and 'grow' methods
 - ▶ Generates equal numbers of full and grown trees with all possible depths between 2 and *maxdepth*
 - ▶ Hence a great variation of tree sizes and shapes are constructed
 - ▶ Evolution needs variability to work with

Data: *maxdepth* — maximum depth of any individual in population, ≥ 2

Data: *N* — population size, an even multiple of *maxdepth*

for *i=2 to maxdepth do*

```
|   for 1 to  $N/(2*(maxdepth-1))$  do  
|   |   Add InitialiseF(root,i) to population;  
|   |   Add InitialiseG(root,i) to population;  
|   end  
end
```

Fitness

- ▶ Depending on the problem we are applying GP to, there are two main ways of calculating raw fitness
 - ▶ Absolute performance (for control, optimisation problems, etc.)
 - ▶ Error (for regression-type problems)
- ▶ Absolute performance is familiar from GAs
- ▶ Error is interesting
 - ▶ For problems such as symbolic regression we can compare the output from an individual program against the correct answer (the *ground truth*)

$$r_i = \sum_{j=1}^{j=n} |S(i, j) - c_j|,$$

where n is the number of *fitness cases*, r_i is the fitness of individual i , $S(i, j)$ is the output of the individual i on input j , and c_j is the correct output for input j

Fitness

- ▶ N.B. The error-based definition of fitness

$$r_i = \sum_{j=1}^{j=n} |S(i, j) - c_j|,$$

- ▶ is similar but slightly different to the traditional *sum of squares* that we usually minimise in regression

$$\sum_{j=1}^{j=n} (Y_j - \hat{Y}_j)^2,$$

- ▶ where Y_j is the value of sample point j , and \hat{Y}_j is the value predicted for sample j by our regression estimator

Fitness

- ▶ The symbolic regression (i.e. *function approximation*) problem for GP is equivalent to statistical regression
- ▶ We can see this be relabelling the regression line as the prediction from an individual GP-tree on a given input, and the sample point as the ground-truth for that input
 - ▶ E.g. if we were trying to use GP to approximate a set of points on a quadratic line, only allowing linear functions, the compromise function achieving the *best fit* would be the one that minimises the sum of squares
- ▶ So perhaps sum of squares is a better fitness measure for GP
 - ▶ One nice feature is that it penalises large errors more than small errors

GP Operators

- ▶ The main GP operators are
 - ▶ Clonal reproduction
 - ▶ Crossover
- ▶ Why no mutation?
 - ▶ Loss of functions and terminals from the population is rare
 - ▶ Functions and terminals are not limited to particular loci
 - ▶ There are typically far fewer functions and terminals than there are combined loci in the population
 - ▶ Also, as we shall see, convergence is not so accute in GP
- ▶ Clonal Reproduction
 - ▶ As its name suggests, this results in the insertion of a cloned offspring into the population
 - ▶ Why? We wanted to avoid duplicated individuals in the initial population.
 - ▶ But this is more useful than it sounds, as selection for reproduction is fitness-proportional...

Crossover

- ▶ Crossover in GP is performed by exchange of subtrees between two parents
 - ▶ A node (crosspoint) in each parent is randomly selected, often with a 9:1 bias in favour of choosing a function node over a terminal node
 - ▶ The entire subtree that has its root as the selected node in one of the parents is swapped with the corresponding subtree in the other parent
 - ▶ This always produces legal offspring as long as the closure property holds over $F \cup T$
 - ▶ Selection of root crosspoints in both parents simply clones both parents...
 - ▶ but incest produces non-clonal offspring, as long as the crosspoints are not the same
 - ▶ This counteracts the convergence pressure exerted by the clonal reproduction operator
 - ▶ It also tends to render mutation unnecessary for the same reason
 - ▶ Typically a maximum depth for offspring is enforced, to prevent unmanageable individuals being created
 - ▶ A clone of one of the parents is inserted into the population in the place of the over-sized offspring