

Lecture 6: Examples of Genetic Programming

GP Example - Learning the 11-Multiplexer

- ▶ Let us look at an example application of GP, learning a multiplexer
- ▶ Multiplexers are a class of addressing problems
 - ▶ Instances have k 'address' bits, followed by 2^k 'data' bits
 - ▶ The address bits index a bit in the data bits, and the class of the instance is the value of that bit
 - ▶ The 11-multiplexer has 11-bit instances, of 3 address bits plus 2^3 data bits, e.g.
 - ▶ 01001011011 \Rightarrow Class 0
 - ▶ 10101011011 \Rightarrow Class 1
 - ▶ 00001010011 \Rightarrow Class 0
 - ▶ 01101010011 \Rightarrow Class ?
 - ▶ 10001010011 \Rightarrow Class ?

GP Example - Learning the 11-Multiplexer

- ▶ The first step is to choose the function and terminal sets
 - ▶ Terminal set
 - ▶ Choosing the terminal set is straightforward, it is simply each of the data bits in the problem
 - ▶ $T = \{A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7\}$
 - ▶ Function set
 - ▶ Choosing the function set is slightly more interesting
 - ▶ As the multiplexer is a boolean function we choose some boolean operators
 - ▶ $F = \{\wedge, \vee, \neg, IF\}$
 - ▶ Note that $F \cup T$ satisfies the closure property; only 0s or 1s will appear as input to any function, and all functions are defined for both of these values

GP Example - Learning the 11-Multiplexer

- ▶ We also need to select a fitness function
 - ▶ We shall define raw fitness to be the number of correct predictions an individual makes, summed over all 2^{11} fitness cases
 - ▶ Raw fitness therefore varies between 0 and 2048
- ▶ Hence we are using GP to perform function approximation, or *concept learning* on the 11-multiplexer
 - ▶ This contrasts with the traditional Machine Learning problem of predicting the classes of new unseen instances (*generalisation*)

GP Example - Learning the 11-Multiplexer

- ▶ After 9 generations the best individual so far correctly classifies all fitness cases
 - ▶ (IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0)) (IF A0 (IF A1 (IF A2 D7 D3) D1) D0)) (IF A2 (IF A1 D6 D4) (IF A2 D4 (IF A1 D2 (IF A2 D7 D0))))))
- ▶ This is rather hard for a human to parse in their head...
- ▶ We can simplify it by applying an *editing* operation, to give the logically equivalent
 - ▶ (IF A0 (IF A2 (IF A1 D7 D5) (IF A1 D3 D1)) (IF A2 (IF A1 D6 D4) (IF A1 D2 D0)))

Editing

- ▶ Editing is an asexual operation that simplifies a single individual
- ▶ Simplification is done using a combination of a domain-independent editing rule...
 - ▶ *Universal editing rule*: if a function without side-effects occurs in a tree with constant arguments, evaluate the function and replace with the result
- ▶ ...and domain-dependent editing rules
 - ▶ E.g. for boolean logic
 - ▶ $\neg(\neg A) \rightarrow A$
 - ▶ $A \wedge A \rightarrow A$
 - ▶ $A \vee A \rightarrow A$
 - ▶ De Morgan's Laws
 - ▶ etc.
- ▶ Editing can be used as an operator during the GP search
 - ▶ Reduces *bloat* of individuals...
 - ▶ at the expense of diversity
- ▶ Typically editing is only applied when interpreting the results of a GP run, as here

GP Example - Learning the 11-Multiplexer

- ▶ It is informative to notice that the best solution found by GP is hierarchical
- ▶ Two multiplexer problems exist that are smaller than 11-multiplexer
 - ▶ 6 multiplexer - 2 address bits + 2^2 data bits
 - ▶ 3 multiplexer - 1 address bit + 2^1 data bits
- ▶ Looking again at our simplified solution, we see that it is a composition of two 6-multiplexers
 - ▶ A0 is used to decide whether the A1 and A2 address bits should index into D7,D5,D3,D1, or into D6,D4,D2,D0
- ▶ Furthermore, at the lowest level these 6-multiplexers are each a composition of two 3-multiplexers
 - ▶ The 3-multiplexer can be solved by a single IF-ELSE function

GP Example - Learning the 11-Multiplexer

- ▶ The best individual in generation 9 is a boolean function that perfectly matches all our fitness cases
- ▶ How likely is it that random search would have found this particular boolean function so quickly?
- ▶ We can estimate this by calculating the number of distinct boolean functions
 - ▶ *Exercise:* Derive an expression for the number of possible boolean functions operating on n variables and returning one boolean value as the result. Evaluate this for $n = 11$ to calculate the probability of finding the 11-multiplexer function by random search in that function space
- ▶ N.B. actually the number of GP trees implementing the same boolean function is high (infinite if we do not limit the tree depth), so the probability of finding a GP tree to solve the 11-multiplexer by random search will be even smaller

GP Example - Learning the 11-Multiplexer

- ▶ We have already established that $F \cup T$ satisfies the closure property
- ▶ We have also demonstrated its sufficiency, by using it to find an individual that completely solves the problem
- ▶ Would a subset of F have the sufficiency property? What is the smallest subset that is sufficient to solve the problem?
 - ▶ *Exercise:* Determine analytically the minimum subset(s) of F that are sufficient to solve the multiplexer problem. Run a GP system on the 11-multiplexer, experimenting with the various sufficient subsets of the function set F and examining the best-of-run individuals produced

Hierarchy in GP

- ▶ The multiplexer example illustrates a general principle of GP
 - ▶ GP works with hierarchies of building blocks
- ▶ In the multiplexer the building blocks are repeatedly and independently discovered by the GP algorithm
- ▶ It would seem useful to be able to re-use building blocks once discovered
- ▶ Two approaches to this exist
 - ▶ Encapsulation
 - ▶ Automatically Defined Functions

Encapsulation

- ▶ The *encapsulation* operator works quite simply on a single parent, to produce a single offspring
- ▶ As its name suggests, it encapsulates some part of the parent, making it available for use in other individuals
 - ▶ Clone the parent
 - ▶ In the cloned offspring select one internal node at random and remove that subtree
 - ▶ Define a new member of the function set F that evaluates that subtree
 - ▶ Put the new function in the offspring at the point at which the subtree was removed

Automatically Defined Functions

- ▶ Encapsulation allows useful functions to be created, which *do not* take any parameters
- ▶ For the multiplexer, the building blocks *do* take parameters
 - ▶ The address and data bits used in a smaller multiplexer
- ▶ To create building blocks with parameters we can use *Automatically Defined Functions (ADFs)*
 - ▶ Like encapsulation, we define a new function based on a subtree, but allow that function arguments which will be mapped onto *dummy variables*
 - ▶ This works in a slightly more complicated way than encapsulation

ADF - Initialisation

- ▶ We first decide on how many ADFs the GP algorithm will be allowed for the problem
 - ▶ We also specify how many arguments each ADF should be allowed to use
- ▶ We then put a placeholder 'LIST' node of appropriate arity at the root of all trees in the population
 - ▶ One branch is required for each ADF, and one for the *value-returning subtree*
- ▶ Finally, we randomly initialise the population with the following function and terminal sets
 - ▶ ADF subtrees
 - ▶ Function set - standard function set F selected for problem
 - ▶ Terminal set - dummy variables $\{ARG0, ARG1, \dots, ARGn\}$
 - ▶ Value-returning subtree
 - ▶ Function set - standard function set plus ADF set, $F \cup \{ADF0, ADF1, \dots, ADFn\}$
 - ▶ Terminal set - standard terminal set T selected for problem

ADF - Crossover

- ▶ We must be careful in applying crossover to trees containing ADFs
- ▶ In particular, what happens if we select a crosspoint in an ADF subtree in one parent, and in the value-returning subtree in the other parent?
- ▶ To avoid this, we must implement a *structure preserving crossover* operator
 - ▶ A crosspoint is randomly selected in one parent
 - ▶ A crosspoint of the same *type* is then randomly selected in the other parent
- ▶ The types for GP trees with ADFs are
 - ▶ Root 'LIST' node
 - ▶ Nodes inside a particular ADF subtree
 - ▶ Nodes inside the value-returning subtree

Symbolic Regression with Constants

- ▶ So far we have been looking at symbolic regression on boolean functions
- ▶ These are the simplest functions
 - ▶ In particular, there are only two values members of the terminal set T can take, 0 and 1
- ▶ We might also want to do symbolic regression on more complicated functions, such as those whose domain and range are real numbers
- ▶ E.g. imagine we want to learn the function for the area of a circle
 - ▶ $a = \pi r^2$
- ▶ Solution of this problem requires us to put the constant π in the terminal set
 - ▶ π is a pretty useful constant, so if we know we're doing geometry it's not unreasonable to put it in
 - ▶ Not all constants are so obvious... what should we do?

Symbolic Regression with Constants

- ▶ Clearly it's not practical to include all the real numbers we might need in the terminal set
- ▶ Even for integers we face the same problem
- ▶ The solution is to use *random ephemeral constants*
 - ▶ The ephemeral random constant is an additional terminal in the set T
 - ▶ It is labelled \mathfrak{R}
 - ▶ At population initialisation, whenever a \mathfrak{R} is chosen as a terminal it is assigned a random value in some appropriate range
 - ▶ These random constants can then be moved between trees by crossover and combined together with arithmetic and mathematical operators to give new values
 - ▶ In doing so new constants can be created
 - ▶ We can optionally use the edit or encapsulation operations to protect them from destruction by crossover

Advanced Genetic Programming

- ▶ We have now covered the basics of Genetic Programming
- ▶ Various advanced techniques exist for you to investigate, if you're interested, e.g.
 - ▶ Iteration
 - ▶ Recursion
 - ▶ Cascading variables
 - ▶ Hierarchical ADFs
 - ▶ Strongly-typed GP
 - ▶ ...