Modularity in Genetic Programming

John R. Woodward

School of Computer Science, University of Birmingham, B15 2TT, UK Email: J.R.Woodward@cs.bham.ac.uk

Abstract. Genetic Programming uses a tree based representation to express solutions to problems. Trees are constructed from a primitive set which consists of a function set and a terminal set. An extension to GP is the ability to define modules, which are in turn tree based representations defined in terms of the primitives. The most well known of these methods is Koza's Automatically Defined Functions. In this paper it is proved that for a given problem, the minimum number of nodes in the main tree plus the nodes in any modules is independent of the primitive set (up to an additive constant) and depends only on the function being expressed. This reduces the number of user defined parameters in the run and makes the inclusion of a hypothesis in the search space independent of the primitive set.

1 Introduction

Genetic programming (GP) [1, 2] is inspired by natural evolution. Candidate solutions are generated and their performance is measured against a specific target task. The better solutions are then taken and altered typically by mutation and crossover operators and tested again. This process is repeated until terminating conditions are met. All evolutionary algorithms follow this test-and-generate approach to produce new solutions in the hope that eventually ever improving solutions will give a satisfactory solution.

GP is plagued by the number choices a user has to make at the start of a run and it is often difficult to know in advance what effect these choices may have on the success of a run. For example, trees can potentially grow without limit and this would cause the program to crash so typically the depth of tree is limited. Operators are designed that produce new trees from old ones. A suitable representation is required in which potential solutions are expressed. This is usually tree based and is expressed in terms of a set of primitives. All of these choices affect the dynamics of a genetic run. This paper is concerned with representation.

In GP, the typical representation is tree based (see Figure 1). This is for largely historical reasons due to the many of the first GP systems being coded in LISP which uses tree based representations. Other data structures have been used, namely linear and graph based structures [3, 4]. Tree structures are used for illustration, as they are the most common, but the result (Theorem 2) holds for any data structure. In this paper, non-modular representations will be

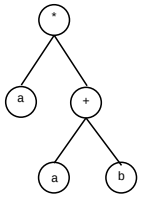

Fig. 1. A tree representation of the function a*(a+b). The function set is $\{+, *\}$ and the terminal set is $\{a, b\}$. There is no modularity within a tree, repeated subtrees have to be expressed explicitly.

referred to as trees (see fig 1) and modular representations will be referred to as forests (see fig 2), in order to make the distinction explicit.

Tree structures are constructed from a set of primitives. This set consists of two sets, a function set and a terminal set. The function set consists of a number of functions which can be linked together to form larger more complicated functions. Each function has an arity, which is the number of inputs, or arguments, it takes. For example each of the following arithmetic operators take two { +, -, *, / }. A typical function set for a maths problem may include { +, -, *, / } and a typical function set for a logic problem may include { and, or, xor , not } (i.e. a logically complete set). The terminal set corresponds to the input variables of the problem. The function set and terminal set together should be powerful enough to express the solution to the problem, if not then the solution lies outside the search space and GP will not be able to find the solution. Typically it is not difficult to know if the primitive set is powerful enough and is fairly obvious from the nature of the problem.

In the GP literature, the function set and terminal set are referred to as two separate sets. One reason is possibly the semantic and syntactic difference between the two sets. The function set represents what processing can be done and the terminal set represents the input data of the problem. Also, in the tree representation, terminals are always found at the leaf nodes and functions are found at the non-terminal nodes. Terminals could be considered as functions with arity zero. The function set F and terminal set T could be described as a single set called the primitive set P i.e. if $F = \{f1, f2\}$ and $T = \{t1, t2\}$ then $P = \{F, T\} = \{f1, f2, t1, t2\}$. The term primitive set will be used unless the distinction is necessary.

The outline of the paper is as follows. In section 2 modularity is defined and commented on. Section 3 looks at the current modular techniques available in GP. Definitions and proofs are laid down in section 4, followed by the implications for learning in section 5 and a discussion in section 6.

2 Modularity

Definition 1. *A module is a function that is defined in terms of a primitive set or previously defined modules.*

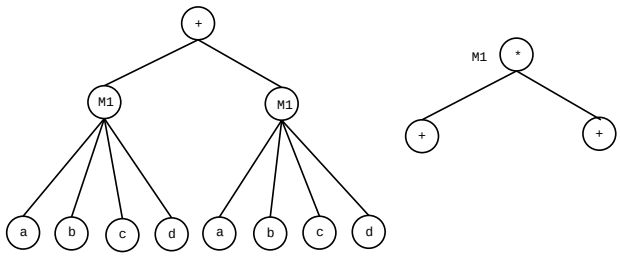

Fig. 2. A function is represented in modular form as a forest (i.e. a collection of trees). The tree on the left is the main tree and the tree on the right is a module. The function set is $\{+, *\}$ and the terminal set is $\{a, b, c, d\}$. The function represented is ((a+b)*(c+d))+((a+b)*(c+d)). This is arrived at by substituting in the module M1, defined by the right tree, at the positions of the two nodes in the left tree. There is modularity within a forest, modules are represented explicitly by the addition of trees to the forest. In this example the terminals could also be included in the module M1, obtaining further compression.

The ability of a representation to include modularity does not add expressiveness, it simply makes the expression more efficient in terms of its size. The following observations about modules are made. Firstly, they can be considered as primitives. Secondly, they interface like primitives. Finally, they can be treated as a black boxs. The above definition does not permit recursion, which may be considered slightly restrictive, but this restriction does not affect the main result of the paper.

In the context of GP with an addressable memory [3] it may be more natural to talk about instructions rather than function and terminal sets. Technically functions and terminals all return a value whereas instructions do not. Also, instructions can alter the memory, whereas functions do not.

A module could be defined in terms of itself or in terms of modules that 'chain back' to the original module. While this can be allowed in some systems it is generally prohibited in GP to avoid problems of infinite recursion (see section 3).

There are a number of reasons why it is desirable to represent a solution in a compact modular form rather than in a cumbersome repetitious way. Firstly, it is well accepted the shorter solutions generalize better than longer solutions. If two solutions account for the given data, nothing can be gained by using the larger solution rather than the smaller solution. Secondly, modularity itself is a simple process to comprehend and implement. Surely, if a solution can be expressed in a neat, more elegant form this appeals to our notion of a conciseness rather than a longer solution with unnecessary repetition. Thirdly, if there is structure in the problem, then this should be exploited in the solution. Shortest solutions have no structure, if they did any repeated structure could be removed by replacing the repeated structure by appropriate modules. Later, it is argued that if there is no structure in a problem then it is effectively unlearnable in the sense that

no predictions can be made about unseen inputs (see section 6). Finally, the author can see no reason for not expressing something in a modular form!

The representation defines the set of all possible solutions that a GP may explore. If, however, the GP system is allowed to modularize, it can effectively define its own primitives during evolution. In other words it can alter it representation to suit itself.

In artificial intelligence one often talks about dividing a problem into its subproblems, solving these smaller (and hopefully easier!) problems in isolation, and then combining these solutions into a solution to the original problem. Modularity is one way to attempt this, and each module could be considered as a solution to each of the subproblems. If one subproblem has been solved and occurs again later, it does not have to be solved again; the solution can simply be referred to again. Surely, the reuse of solutions to previously solved problems must appeal to our intuition of what intelligence is.

Solutions are often represented as trees and modules correspond to branches off the root node of the tree. For example, Automatically Defined Functions (see section 3) have a main result-producing branch along with one or more function-defining branches. In this paper, a non-modular representation will be represented as and referred to as a tree (see fig. 1). A modular representation will be represented as and referred to as a forest (see fig. 2). Thus a module will appear explicitly as a separate tree in the forest.

3 Related Work

Here a number of modularization methods are discussed. For each of these methods a representation is needed along with an operation which produces new candidate solutions from old ones and dynamically produces new modules. This paper is only concerned with the representation and not with the process of moving from one point to another in the search space.

Automatically Defined Functions (ADFs) [5, 2] are probably the most popular of the modularization methods used in GP. Along with the main tree, additional branches are allowed to hang down from the root of the tree which define ADFs. ADFs are called just as if they belonged to the primitive set. Before starting the run, the user has to decide on the number of ADFs, the number of parameters each ADF takes and which ADFs can call which other ADFs. This final point is important to avoid infinite recursion and circular definitions. These choices all affect what structures are included in the search space.

Architecture altering operations [6] are a natural extension to ADFs which allow the structure of a solution to alter thus freeing the user from having to make these choices. ADFs can be created or deleted and new parameters added or removed from the argument lists of ADFs. Thus the number of ADFs and the parameters each one takes can change during the run. As pointed out in [1] (page 288), this has the benefit that no extra parameters are required in addition to the parameters required for standard GP run with no ADFs.

Adaptive Representation [7] creates modules on-the-fly during evolution according to population statistics. In a sense this is very similar to architecture altering operations in that the form of the representation is free to alter during the run.

Encapsulation [5] and module acquisition [8] are two similar modularization methods. With encapsulation, a point in a tree is chosen and the subtree from that point down to all the terminals in that subtree is 'encapsulated'. This encapsulated subtree is then treated as a new terminal (because it takes no arguments). With module acquisition, a point in a tree is chosen and the subtree (down to a given depth) is promoted as a module. Due to the depth restriction a module may or may not take arguments.

These two methods only allow identification of a subset of possible modules. Encapsulation forces all modules to behave like terminals as the encapsulation process captures the whole of a subtree from some point down to all the terminals in that subtree. Module acquisition is similarly limited in that a predefined depth determines what can and cannot be promoted as a module. Thus the space of potential modules is restricted.

The dynamics of how each of these methods select modules are different. The representation used with both of these methods is essentially the same; a main tree along with any modules which are also represented as trees.

If an instruction set is Turing Complete then it is implicitly capable of expressing modular solutions. There is no need to supply and additional vehicle for the purpose of modularity. Work on GP with Turing Complete instruction sets has been done, however we are not aware of any results that could be identified as having modules. Only programs with single loops have been evolved [3].

4 Definitions and Proofs

We need to be able to convert an expression in terms of one primitive set into an expression in terms of a different primitive set. This is done using a dictionary.

Definition 2. *A dictionary $D_{P_1P_2}$ is a set of trees or forests which express each member of primitive set P_2 in terms of primitive set P_1.*

The existance of $D_{P_1P_2}$ does not imply the existance of $D_{P_2P_1}$. The size of $D_{P_1P_2}$ is $|D_{P_1P_2}|$, and in general, $|D_{P_1P_2}| \neq |D_{P_1P_2}|$. The dictionary must be of finite size.

Definition 3. *Two primitive sets P_1 and P_2 are equally expressive if and only if a pair of dictionaries $D_{P_1P_2}$ and $D_{P_2P_1}$ exist.*

In terms of what can be expressed with the two sets, there is no difference, only the number of primitives in the expression will differ. There are many examples of equally expressive primitive sets. For example any pair of logically complete sets are equally expressive. Similarly, most programming languages are equally expressive as they are Turing Complete.

Definition 4. *A primitive set is a minimal primitive set if no member of the set can be expressed in terms of the other members of the set.*

Within a primitive set, one or more of the members may be expressed in terms of other members of the set. This implies that there is some redundancy in the set and these members could be removed without affecting what could be expressed with that set. An example of redundancy would be the `for` loop construct present in most programming languages. It could be removed from the language without affecting what could be expressed in the language as a `for` loop could be implemented in terms of a `do while` loop. A logically complete primitive set which is minimal is referred to as minimally logically complete. A set of instructions which are minimal and Turing Complete is referred to as minimally Turing Complete. If a primitive set is minimal it implies there is no redundancy in the set.

Definition 5. *The size $S_P(f)$ of a tree or forest, which expresses a function f, in terms of a primitive set P, is the number of nodes it contains.*

Definition 6. *The complexity $C_P(f)$ of a function f is the size of the smallest tree or forest expressing f, in terms of a primitive set P.*

A function can be expressed as different trees or forests in terms of a primitive set and each of of which may have a different size. For a given function and a given primitive set there will exist a minimum size which can express the function. This is the complexity of the function for the given primitive set.

Theorem 1. *Given two equally expressive primitive sets P_1 and P_2, the complexity $C_{P_1}(f)$ has an upper bound which is within a constant factor, $K_{P_2P_1}$, of the complexity $C_{P_2}(f)$, provided modularity is not permitted (i.e. only trees are permitted).*

$$C_{P_1}(f) \leq C_{P_2}(f) * K_{P_2P_1}$$

Proof. Given a function expressed in terms of a primitive set P_2, it can be reexpressed in terms of the primitive set P_1 by directly substituting in the definitions of each of the primitives from the dictionary. In the worst case, each node can be rewritten in terms of a tree of size $K_{P_2P_1}$. $K_{P_2P_1}$ is the complexity of the dictionary $D_{P_2P_1}$.

Theorem 2. *Given two equally expressive primitive sets P_1 and P_2, the complexity $C_{P_1}(f)$ is equal to the complexity $C_{P_2}(f)$ within an additive constant provided modularity is permitted (i.e. forests are permitted).*

$$C_{P_1}(f) \leq C_{P_2}(f) + K_{P_2P_1}$$

Proof. Given a function expressed as a forest in a primitive set P_1, it can be expressed as a forest in the primitive set P_2 by adding on the definitions of each of the primitives. Thus in the worst case all of the primitives need to be rewritten in terms of a tree of size $K_{P_2P_1}$, $K_{P_2P_1}$ is the complexity of the dictionary $D_{P_2P_1}$.

The dictionary $D_{P_1P_2}$ has complexity $K_{P_1P_2}$ and is defined to be the number of nodes in the smallest dictionary. In general $K_{P_1P_2} \neq K_{P_2P_1}$

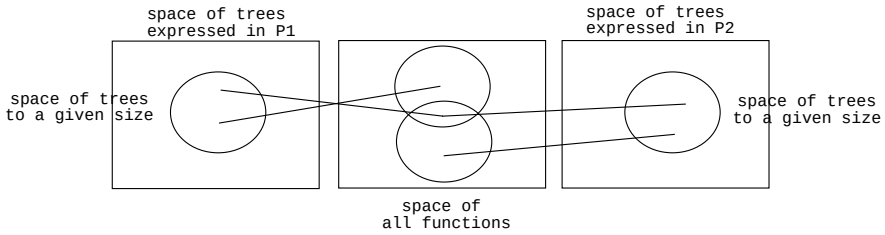

Fig. 3. In the tree representation, modularity is not permitted and the two spaces of trees of limited size map to two different spaces of functions (which overlap slightly).

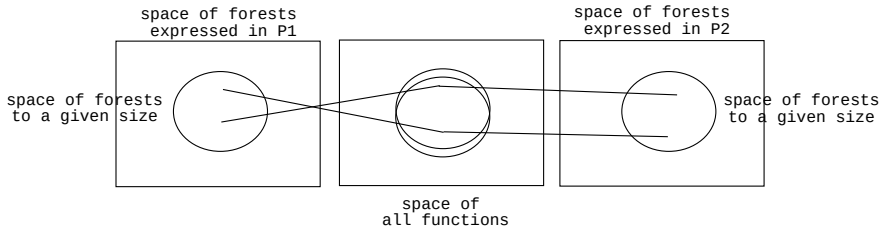

Fig. 4. In the forest representation, modularity is permitted and the two spaces of forests of a limited size map to almost the same space of functions. The smaller the complexity of the dictionaries involved the more the two spaces of functions overlap.

The definitions of size and complexity are the same as those given in Wegener (page 9 [9]), except they apply to Boolean Circuts. He also states that the complexity and depth of a Boolean Function can only increase by a constant factor if we switch from one primitive set to another.

Corollary 1. *Given two equally expressive primitive sets, the set of functions expressed by forests using either primitive set is approximately the same if the size limit on the forests is large compared to the complexity of the dictionaries.*

This follows as a consequence of Theorem 2 as the size of the dictionary can be ingored if the size limit on a forest is much bigger than the complexity of the dictionaries. This additive constant will be ingored from now on and so the set of functions defined by forests (of fixed upper size) is the same. This is realistic for most primitive sets used in GP and target functions we expect GP to find.

5 Implications for Learning

Evolution in GP can be thought of as the movement of a population of points through a search space. Each point is represented by a tree or forest and corresponds to a function. The aim is to find a point that corresponds to the target function. The terms hypothesis space and and search space are used interchangeable and a point in a search space represents a particular hypothesis. A point in

the search space is the genotype and the function it represents is the phenotype. The mapping between these two spaces is many to one as many trees represent the same function. It is interesting to note that it has been suggested that the No Free Lunch Theorem does not hold in this space [10]. The structure of the search space is defined by the representation, the operators used to move around the space and the fitness function and is often considered using the landscape metaphor [11].

Two different sets of primitives define two different search spaces. These two spaces will map to the same set of functions if the two primitive sets are equally expressive (and the size is unlimited). However, if a maximum depth or size is imposed on a tree, this will effectively cut off part of the space, and therefore limit which functions can be represented. In general, if modularity is not permitted, the set of functions defined by trees in the two spaces will be *different*. If however the representation does allow modularity, and the size of forests in the search space is limited, the set of functions will *the same*. In other words, if we do not have modularity, the functions expressed in two search spaces defined by two different primitive sets will diverge, whereas if the ability to modularize is included the sets of functions will converge as the maximum size of a tree or a forest increases.

Figure 3 shows how trees expressed in terms of two different primitive sets map to different parts of the function space. Figure 4 shows how forests (i.e. trees with the ability to modularize) expressed in terms of two different primitive sets map to the same part of the space of functions. The left and right rectangles represent the space of all trees (in fig 3) and forests (in fig 4) without a size limit, expressed in terms of P_1 and P_2 respectively. The central rectangle represents the space of all functions that can be represented. Ellipses in figure 3 on the left and right represent trees up to a fixed size. These map to two *different* subsets of functions. Note that the small overlap signifies that some functions can be represented by trees of a fixed size in either primitive set. Ellipses in figure 4 on the left and right represent forests up to a fixed size. These map to *the same* subsets of functions which is illustrated as two ellipses which largely overlap.

In summary, given two equally expressive primitive sets with a representation which permits modularity, if the size of the forests is limited then the two search spaces map to the same space of functions. If modularity is not permitted, the two search spaces will map to different spaces of functions. In general the size of the two search spaces will be different. No claims are made at this stage about the relationship between the structure of the two search spaces or appropriate operators to move around these spaces.

6 Discussion

One of the aims of GP is to synthesize a target function from a basic primitive set to express the relationship between given input and output data. Perhaps deeper aim is generalization, which is the ability to predict what should happen on unseen data (i.e. data that was not included in the training set). To express

a given set of data a lookup table could be constructed recording all the inputs against all the outputs, but this says nothing about what rule, if any, there is connecting the input and output data. If we are to make predictions we need access to this rule or an approximation to it. If there is a pattern in the training data, then this pattern can be exploited and used to predict the output data, and if there is no pattern then it is impossible to make any predictions. Modularity is one vehicle which can help express regularity and patterns in data.

The Kolmogorov complexity of a bit string is the length of the shortest program, in bits, that prints out the bit string [12]. Kolmogorov complexity is thus associated with Turing Complete instruction sets, and is therefore undecidable. In a sense, Kolmogorov complexity is a special case of Theorem 2 for the case when the primitive sets are Turing Complete.

Theorem 2 is in terms of numbers of nodes in forests. Each node corresponds to either a function or a terminal in the primitive set. It may be conceptually easier to think of the complexity of a function in terms of the number of primitives required to express it rather than the number of bits.

The complexity of functions expressed in Turing Complete systems is undecidable. The complexity of, for example, a logical function expressed in terms of a logically complete primitive set and a modular representation is decidable (the space of all forests could be enumerated and searched, and each forest terminates as there is no self-referencing in the system). This may therefore be a suitable arena in which to study some aspects of complexity and modularity.

Further work includes examing learning using two equally expressive primitve sets. Comparing a minimal and non-minimal primitive set to investigate the effect of redundancy in the primitive set. Also being considered are suitable operators to move around a space of modular representations.

7 Conclusion

The main result of this paper is that the size of a solution is independent of the primitive set used provided modularity is permitted. In practice, the size of a search space is limited and the implication for the GP practioner is that the inclusion or exclusion of a target function from a search space depends only on a size parameter and not on the primitve set and hence can potentially remove any bias due to the choice of primitive set.

Modularization is important in the expression of a solution. It not only appeals to our intuition of conciseness but is also vital in expressing regularities in data and therefore the ability of solutions to make accurate predictions.

A number of modularization techniques in GP are discussed. Most of them require the user to supply parameters concerning the sizes and numbers of of modules. We prove that the minimum number of primitives needed to express a target function is independent of the primitive set if a modular representation is permitted. It should be noted that no claim is being made at this stage about the dynamics of GP, or that modularity should be enforced during a GP run.

8 Acknowledgements

Mark Ryan, Achim Jung, Xin Yao, Stefano Cattani.

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag (1998)
2. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts (1994)
3. Huelsbergen, L.: Toward simulated evolution of machine language iteration. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 315–320
4. Miller, J.F., Thomson, P.: Cartesian genetic programming. In Poli, R., Banzhaf, W., Langdon, W.B., Miller, J.F., Nordin, P., Fogarty, T.C., eds.: Genetic Programming, Proceedings of EuroGP 2000. Volume 1802 of LNCS., Edinburgh, Springer-Verlag (2000) 121–132
5. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
6. Koza, J.R.: Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, J.R., Reynolds, R.G., Fogel, D.B., eds.: Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming, San Diego, CA, USA, MIT Press (1995) 695–717
7. Rosca, J.P., Ballard, D.H.: Learning by adapting representations in genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, Orlando, Florida, USA, IEEE Press (1994)
8. Angeline, P.J., Pollack, J.B.: The evolutionary induction of subroutines. In: Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Bloomington, Indiana, USA, Lawrence Erlbaum (1992)
9. Wegener, I.: The Complexity of Boolean Functions. Wiley Teubner (1987)
10. Woodward, J.R., Neil, J.R.: No free lunch, program induction and combinatorial problems. In: Genetic Programming, Proceedings of EuroGP 2003, Essex, UK, Springer-Verlag (2003)
11. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002)
12. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA (1991)