Complexity and Cartesian Genetic Programming

John R. Woodward

The University of Birmingham, Birmingham B15 2TT, UK.
http://www.cs.bham.ac.uk/ jrw/

Abstract. Genetic Programming (GP) [1] often uses a tree form of a graph to represent solutions. An extension to this representation, Automatically Defined Functions (ADFs) [1] is to allow the ability to express modules. In [2] we proved that the complexity of a function is independent of the primitive set (function set and terminal set) if the representation has the ability to express modules. This is essentially due to the fact that if a representation can express modules, then it can effectively define its own primitives at a constant cost.
Cartesian Genetic Programming (CGP) [3] is a relative new type of representation used in Evolutionary Computation (EC), and differs from the tree based representation in that outputs from previous computations can be reused. This is achieved by representing programs as directed acyclic graphs (DAGs), rather than as trees. Thus computations from subtrees can be reused to reduce the complexity of a function. We prove an analogous result to that in [2]; the complexity of a function using a (Cartesian Program) CP representation is independent of the terminal set (up to an additive constant), provided the different terminal sets can both be simulated. This is essentially due to the fact that if a representation can express Automatic Reused Outputs [3], then it can effectively define its own terminals at a constant cost.

1 Introduction

GP is a test-and-generate paradigm where a representation is chosen to express programs, and programs are generated and tested against a set of test cases. A broad variety of representations have been used in GP, e.g. lists, trees and graphs. Research in EC involves choices about the fitness function, representation and genetic operators, however in this paper we are only concerned with representation. Typically, GP is supplied with a function set and a terminal set and new functions are created by stringing together these primitives so the output of one is fed into another. The choice of representation determines how new functions can be created from the primitive set. In standard tree style GP, each time a new 'sub-function' is needed it must be represented again as a new subtree. ADFs, on the other hand, can arbitrarily define new functions which can be reused as and when needed. CGP, which we now introduce, lies somewhere between these two types of representation in terms of the type of reuse allowed.

In CGP, a Cartesian Program (CP) representation is evolved. In a CP a set of function nodes are placed on a 2D grid arranged in rows and columns (see

figure 1). The inputs to the CP are fed in from the left and the outputs are taken from the right. There can be one output or many. Information flows from left to right through the representation. Each function node can take its input from the output of any of the function nodes to the left of it. The functions at each location in the grid and the connectivity of the function nodes (i.e. which function node is connected to which other function node) are evolved. It maybe that some function nodes are not connected at some point during the evolution. For example, the function nodes in the final column could be directly connected to the inputs and so no intermediate computation takes place. While the overall structure has fixed size (effectively the number of rows times the number of columns, as there is a function node at each point in the grid), the 'actual' size may be much smaller as some function nodes are not connected. For example, in figure 1, the middle node in the first column is not connected (i.e. its output, labelled 5, is not used).

There are a number of important points which can be made about this representation. Firstly, large parts of the representation may not be connected to the rest of the program, and so a single mutation may connect in large 'subprograms' allowing 'large jumps' in the search space. Secondly, as information flows from left to right through this representation, the output of a computation (i.e the output value at some point in the 2D grid) can be used multiple times by any function nodes appearing to the right of it. This is in contrast to tree based GP where no reuse occurs. It is this second point which concerns us in this paper.

CGP has been applied to a variety of problems including function regression and boolean problems. A CP typically has more than one output. Mathematically a function has only a single output (i.e. an element in one set maps to an element in a second set). However, we can think of a CP representing a number of functions (which may share parts of the same representation). For example, the CP may be used to control a robot with two wheels, the two outputs controlling each of the wheels. This is similar to artificial neural networks, where there are typically multiple outputs.

In the remainder of this paper, we look at related work using DAGs as a representation in section 2. In section 3 we introduce CP, and in section 4 we examine reuse in different types of representation. Before proving the main results of this paper in section 6 we provide some preliminary definitions in section 5. We end with a section summarising and concluding.

2 Related work

Handley [4] uses a DAG to store a population of trees, rather than a forest of trees (i.e. a set of trees). This has a two benefits. Firstly, identical subtrees (down to the leaves) only need to be represented once. Miller [3] does this within an individual, but Handley [4] does this across the whole population. While there may not be identical subtrees within an individual program, there may be identical subtrees in individuals in the population. Secondly, time is saved by caching the value for each subtree for each fitness case. This information is copied to the next

generation so values only need to be computed once. Handley's method achieves good improvements in the amount of space needed and the run time of a GP algorithm.

Keijzer [5] continues Handley's [4] work using minimal DAGs to store a population. Roberts [6] also applies this method of representing a population as a DAG to a medical imaging problem. He identifies a trade off between the time to find a subtree in the cache and the time to evaluate the subtree. The method is developed by introducing ways to add and remove subtrees from the cache. One interesting point common to both of these works is that as time increases (i.e. the number of evaluations), the accumulated number of evaluations increases *linearly*, whereas uncached versions climb much more steeply.

CGP represents programs as DAGs, rather than as trees. As a CP has a number of outputs, a CP could be considered as representing a population of programs. The same idea of representing duplicate subtrees once by referring to an index of a subtree is the same in CP as it is with storing a population as a DAG.

3 Cartesian genetic programming

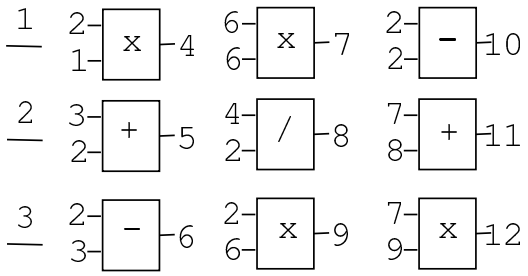

Fig. 1. Visual representation of a CP.

In CGP the genotype is a string. User defined parameters set the number of rows and columns. In this example (see figure 1), we have 3 rows and 3 columns. Each node is described by 3 integers, the first two are the inputs and the third is which function from the function set is to be placed at the location. The first 3x3 integers thus describe the function nodes found in the first column of the CP. For example, the function set $\{+, -, *, /\}$ is represented by the integers $\{1, 2, 3, 4\}$ respectively. Each of these functions has arity 2, i.e. takes two inputs. The inputs to the whole program are labelled 1, 2 and 3 which are listed on the left. An example of a genotype is the following integer string; (We have put commas in to make the list more readable, but do not appear in the genotype). The interpretation of this string is shown in figure 1.

2 1 3, 3 2 1, 2 3 2, 6 6 3, 4 2 4, 2 6 3, 2 2 2, 7 8 1, 7 9 3

The outputs of each function node are labelled incrementally, starting with the top right node with output labelled 4 (as we have program inputs 1, 2, 3) and the final node, bottom right has output labelled 12. The integer string is interpreted in groups of 3 integers. Consider the first set of 3 integers from the genotype, namely 2 1 3. The first two numbers describe the input to the function, in this case inputs 2 and 1. The next number, 3, is the function this node performs, in this case multiplication (∗). There are 3 outputs from this program. Some nodes are not connected e.g. the middle node in the first column (function node with output 5). There is reuse, the output from the bottom node in the first column (labelled output 6) is used 3 times by function nodes in the middle column. Note that in the proof and in figure 3, we represent a CP as a tree with overlapping subtrees (i.e. we are only concerned with the connected nodes). The number of leaves in a CP never needs to exceed the number of terminals as function nodes in the program can refer directly to the terminals.

4 Reuse in different types of representations

Some representations allow reuse of component parts, other representations do not. In this section we contrast three different types of representation used in the evolution of programs. Each have increasing flexibility in the type of reuse which is allowed. In the first type of representation, tree based, no reuse is supported at all. In the second type, CP, reuse of previously expressed subtrees (down to the leaves) is permitted. In the third type of representation, ADFs, arbitrary subtrees may be reused. The ability of reuse has effects on the complexity of a function expressed with each representation.

To describe these representations the following terminology is used. The node at the top is called the root node. A tree has a single root node. A CP may have more than one root node as it may have more than one output. The nodes at the bottom are called leaf nodes. The remaining nodes in the 'body' of the tree are called non-terminal nodes. The root node is a special case of a non-terminal node. The leaf nodes correspond to terminals from the terminal set and the non-terminals correspond to functions from the function set.

In a tree there is no reuse (see figure 2). Each node in a tree is executed once or not at all. There are two ways a tree representation may be executed, top down or bottom up. In the bottom up case, each of the leaf nodes would be assigned a value and these are passed up the tree to the nodes above which makes their calculation and pass the output value up to non-terminal nodes above. Eventually a value emerges from the root node. Each node is executed exactly once. In the top down case, we start at the top and evaluate downwards. The important difference between top down and bottom up evaluation can be illustrated with the following example. If a node represents the logical function AND, and the left hand subtree evaluates to $false$, time can be saved by not evaluating the right hand branch as we can already determine the value of this

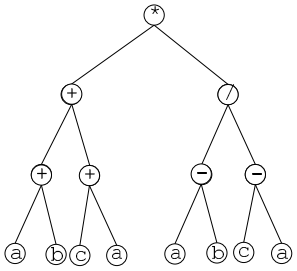

Fig. 2. A function represented as a tree. Each node may be executed once during a single computation and no reuse is permitted.

computation ($false$). Similarly if a node represents an IF-THEN-ELSE function, once the condition part of the subtree is evaluated, we do not need to evaluate both the THEN and ELSE subtrees as only one of the computations is required depending on the condition of the IF subtree. In the top down case each node may be evaluated once or not at all, whereas in the bottom up case each node is evaluated exactly once. If we were to trace out the path from a leaf node to the root node, there would only be one path.

Given a function expressed as a tree using a given primitive set, what happens to the complexity of the function when we express it as a tree in terms of a different primitive set. The depth is bounded by a multiplicative constant and the number of nodes may potentially increase exponentially.

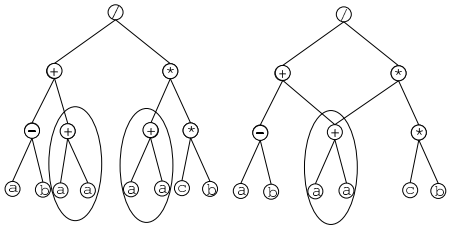

Fig. 3. A tree based representation on the left and a CP of the same function on the right. In tree based representation on the left, two subtrees down to the roots are identical (both are circled by ellipses). On the right, in a CP representation identical subtrees can be represented once (shown by a single subtree circled by an ellipse)

In order to illustrate the similarities and distinctions of CP with tree based data structures and ADFs, we represent them in a similar fashion. Instead of information entering from left and leaving on the right (as in figure 1) we rotate the diagram so information flows from bottom to top. We also remove the grid and only show the connected nodes. Thus we drop the left-to-right grid based

representation of figure 1 and represent CPs as trees with overlapping subtrees (where the overlap is all the way down to the terminals) (as in figure 3).

As data enters at the leaf nodes, it is processed and passed further up the tree. Any node further up the tree may make use of any output further down the tree. In [3] this is referred to as 'Automatic Re-used Outputs' (AROs), where the point is made, "A potential disadvantage is that AROs are not as general as ADFs as they can only re-use an output *with the same inputs*." Indeed, it is at the heart of the proof that ADFs *with the same inputs* (i.e. terminals) is the reason why complexity is invariant with respect to the terminal set.

If we were to trace out the path between a leaf node and an output node, we would find that there could be multiple paths between an input and an output node (see figure 3). If two paths overlap, they must overlap all the way down to the leaf node. CPs can be evaluated from the bottom up or top down.

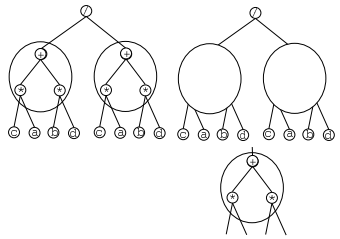

Fig. 4. A tree based representation on the left and a program with an ADF on the right. Two identical subtrees on the right, are represented once on the left but referred to twice. The module contains 3 nodes and takes 4 arguments.

With ADFs, arbitrary subtrees can be defined as modules and an ADF can take different arguments as inputs. This is in contrast to CPs, where subtrees can be reused, but must be subtrees down to the leaf nodes. On the left of figure 4 is a tree, with two subtrees which are identical, indicated by the ellipses. In a representation capable of expressing modules, arbitrary repeated structures can be defined once, and called when needed with the appropriate arguments. On the right, the same function is represented, but the identical structures in the previous tree based representation are represented once (in the ellipse), and is referred to twice in the main tree.

The main difference between CP and ADFs is that 'modules' defined in CP do not take any arguments (and are therefore modules with zero arity i.e. terminals). ADFs are modules which may have non-zero arity, and are therefore modules that can be called with *different* arguments each time.

If we trace a path down from the root node in an ADF representation, two paths may overlap. If the do overlap, there is no guarantee that they will terminate at the same leaf node (essentially as ADFs can be called with different

arguments). This is in contrast to CP, where if paths do overlap, they are guaranteed to end at the same leaf node.

In summary, ADFs are the vehicle for reuse. Arbitrary functions can be defined (using the operation of composition and a predefined set of primitives), and can be used as if they are new primitives, i.e. called when needed with different arguments each time. With CGP, a new set of terminals can effectively be defined and this set can be called when needed. A terminal is effectively a function with no input (i.e. arity zero). Trees, on the other hand, have no mechanism for reused.

5 Preliminary definitions

In this section we give a number of definitions which are similar in nature to those in [2]. These definitions are needed for the proof in the following section.

Definition 1 (terminal set, function set, primitive set). *The terminal set t is the set of inputs to the program. These are typically problem variables and/or constants. The function set f is the basic functions GP uses to construct more complex functions. The primitive set p is the union of the function set f and the terminal set f, i.e $p = t \cup f$. These definitions are taken from [1] (section 5.1).*

Definition 2 (size). *The size of the instance of a CP representation is the number of nodes it contains.*

Note that under this definition, if nodes are not connected in to the overall representation then they still contribute to the size. Alternatively we could have defined size to be the number of connected nodes, but this makes no difference when we consider complexity which is defined in terms of the minimum size.

Definition 3 (equally expressive). *Two primitive sets are equally expressive if they can express the same set of functions in finite size.*

For example, the sets $\{NAND\}$ and $\{NOT, AND, OR\}$ are logically complete (i.e. can express all logical functions) and are therefore equally expressive. The programming languages Java and C are both Turing Equivalent and are therefore equally expressive. While this definition tells us if two primitive sets are equally expressive or not, we need to know how to construct a function in a new primitive set given the function expressed in an old primitive set. This is done using a special CP we call a dictionary.

Definition 4 (dictionary). *A dictionary, $D_{t1,t2}$, is the CP which takes the inputs $t1$ and has outputs $t2$, where $t1$ and $t2$ are terminal sets. The function nodes are from the primitive set$f$. The size of the dictionary must be finite.*

Each member of the set $t1$ can be expressed in terms of $p2 = t2 \cup f$. In this paper we are not concerned with switching function sets, only terminal sets. The existence of the pair of dictionaries $D_{t1,t2}$ and $D_{t2,t1}$ is a necessary and sufficient condition to imply that $p1$ $(= t1 \cup f)$ and $p2$ $(= t2 \cup f)$ are equally expressive.

Definition 5 (complexity). *The complexity, C, of a function (or set of functions) f under the CP representation, with respect to a fixed primitive set p, is the size of the smallest CP which can represent the function (or set of functions), denoted by* $C_p(f)$.

Definition 6 (complexity of a dictionary). *The complexity of a dictionary* $D_{t1,t2}$ *is the size of the smallest dictionary which expresses the set of terminals* t_2 *in terms of a CP using primitive set* $p1$ $(= t1 \cup f)$. *We write* $K_{t1,t2}$ *for the complexity of dictionary* $D_{t1,t2}$.

Strictly, this definition is not necessary as it is implied by the previous two definitions of dictionary and complexity. However, we make this definition explicit as it is made use of in the proofs.

Definition 7 (translate terminal set). *Given a function expressed in terms of one primitive set,* $p1$ $(= t1 \cup f)$, *we can express the same function in terms of a second primitive set,* $p2$ $(= t2 \cup f)$. *As the function sets are the same we need only consider the terminal sets. The process of re-expressing the function in terms of* $p2$ *is called translation of terminal set from* $p1$ *to* $p2$.

6 Complexity

We present 3 theorems regarding the complexity of a function when expressed using a CP representation. We prove that the complexity of a function is invariant under translation of terminal set, if the two primitive sets are equally expressive. We then go onto prove the tightest upper and lower bounds on the complexity of an arbitrary function when expressed using primitive sets with a different terminals sets but the same function set.

Theorem 1 (complexity). *The complexity of a function under the CP representation is invariant under translation of terminal set, within a constant* $K_{t1,t2}$ *(the complexity of the dictionary* $D_{t1,t2}$*) provided the primitive sets are equally expressive.*

$$C_{t2}(f) \leq C_{t1}(f) + K_{t1,t2} \text{ (eq. 1)}$$

Note, strictly we should talk about complexity with respect to a primitive set, but as the function set remains fixed here we drop reference to it.

Proof. Given that the two primitive sets are equally expressive (i.e. $p1 = t1 \cup f$ and $p2 = t2 \cup f$), we can simulate the set of terminals $t2$ using the dictionary $D_{t1,t2}$. As $p1$ and $p2$ are equally expressive, the dictionaries $D_{t1,t2}$ and $D_{t2,t1}$ both exist. These are represented as subtrees of the new CP. As we are using a CP, we only need to represent the new set of terminals once. We illustrate this graphically (see figure 5).

Theorem 2 (smallest bound). $K_{t_2 t_1}$ *is the smallest bound.*

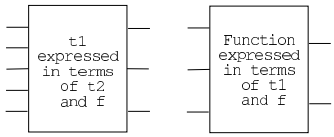

Fig. 5. A visualization of the proof. The box on the right is a CP expressed in terms of a primitive set $f \cup t_1$. The program has 2 outputs and 3 inputs. The box on the left is the CP expressing the set of functions t_1 in terms of $f \cup t_2$. Hence, the combination of CPs expresses the function in terms of $f \cup t_2$ without any reference to t_1.

Proof. Some functions will not depend on all of the terminals in a given terminal set, therefore these terminals do not need to be translated and do not need to be expressed by the dictionary. However in the worst cases, all of the terminals are required to be translated and the complete dictionary is needed. Therefore the smallest size of the bound is the complexity of the dictionary (i.e. the size of the smallest dictionary which translates all of the terminals).

Theorem 3 (lower bound). $C_{t2}(f) \geq C_{t1}(f) - K_{t2,t1}$

Proof. Consider the above equation (eq. 1). Consider the translation from terminal set $t2$ to terminal set $t1$, i.e. $C_{t1}(f) \leq C_{t2}(f) + K_{t2,t1}$ then rearrange the equation $C_{t1}(f) - K_{t2,t1} \leq C_{t2}(f)$ We can also say this is the tightest lower bound by an identical argument to that in the previous proof.

We can combine the above results into a single expression

$$C_{t1}(f) - K_{t2,t1} \leq C_{t2}(f) \leq C_{t1}(f) + K_{t1,t2}$$

and say that these bounds are the tightest obtainable.

7 Summary and conclusions

CP uses a DAG to represent programs, rather than using trees which are typically used in GP. DAGs allow some reuse of subtrees, and this reuse has an affect on the complexity of a function when expressed using a DAG. In contrast, a tree representation offers no reuse as each branch of the computation must be represented explicitly, even if subtrees are identical.

We proved that the complexity of a function, when expressed by a CP, is independent of the terminal set (within an additive constant), provided the primitive sets are equally expressive. This was essentially done with a simulation argument. We also proved that the tightest upper bound on the complexity is the complexity of the set of functions (i.e. the complexity of the dictionary) corresponding to the new terminal set. We also proved the tightest lower bound on complexity is symmetric to this. Thus the complexity of a function is sandwiched symmetrically between the complexity of the function with reference to a different terminal set.

In [2] we proved that the complexity of a function is independent (within an additive constant) of the function set and terminal set provided the sets are equally expressive and the representation is capable of expressing modules (i.e. ADFs). The results in this paper are completely analogous to those in [2], and can be thought of as a special case. As CPs are not as flexible as ADFs in the amount of reuse allowed, one would not expect the same degree of robustness regarding the translation of function set and/or terminal set.

What we have not addressed in this paper is how to use these results to design more efficient search algorithms for CGP. There are two benefits from representing a program as a DAG; the amount of space needed to express a given function is reduced and also the amount of time needed to compute a function for a given input is reduced. We also believe that similar results regarding complexity exist for Binary Decision Diagrams and Artificial Neural Networks as there is the possibility of reuse in these representations.

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. Woodward, J.R.: Modularity in genetic programming. In: Genetic Programming, Proceedings of EuroGP 2003, Essex, UK, Springer-Verlag (2003)
3. 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
4. Handley, S.: On the use of a directed acyclic graph to represent a population of computer programs. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, IEEE Press (1994) 154–159
5. Keijzer, M.: Efficiently representing populations in genetic programming. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 259–278
6. Roberts, M.E.: The effectiveness of cost based subtree caching mechanisms in typed genetic programming for image segmentation. In Raidl, G.R., Cagnoni, S., Cardalda, J.J.R., Corne, D.W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E., Meyer, J.A., Middendorf, M., eds.: Applications of Evolutionary Computing, EvoWorkshops2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM. Volume 2611 of LNCS., University of Essex, UK, Springer-Verlag (2003) 444–454