# Function Set Independent Genetic Programming

John R. Woodward

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

**Abstract.** The choice of function set can significantly affect the performance of a GP run, possibly due to the difference in complexity of a function when expressed in different function sets. If modules are used in the representation, the complexity of a solution is independent of the function set (up to a fixed constant). This result motivates a representation which can be searched independently of the function set.
We represent modules as look up tables, providing the definition of the module. This avoids having to invent heuristics to identify modules. It also delays the implementation details of a module until a decomposition of the problem has been obtained. We present a hierarchical modular search algorithm which decomposes a problem and reduces the impact of the choice of function set.

## 1 Introduction

Genetic Programming (GP) is a potential method of avoiding the task of explicitly programming a computer, by specifying *what* we want but not *how* to do it. A set of training examples are supplied, which define *what* we want the program to do, and it is up to GP to search the space of computer programs in order to find a program which meets the requirements of the training examples.

In GP the choice of function set is critical. (We will use the term primitive set to mean the union of the function and terminal set). For a given problem it is often easy to define a primitive set expressive enough to solve the problem, but this choice can greatly affect the performance of a run. The authors are only aware of one paper which addresses this issue [Wan04].

**The Approach** The originality of this algorithm comes from its representation, not the way it searches the space. A module representation is used, in which the main tree is represented in terms of the primitive set, and the modules are represented as look up tables (LUTs). An outline of the algorithm follows:

1. Firstly, evolve a solution with a module represented as LUT. The module is then passed to the second stage.
2. Secondly, given the module represented as a LUT, evolve a solution (in the specified primitive set) that meets this specification (i.e. the LUT is treated as a set of test cases).
3. Finally, construct the entire solution by replacing the module in stage 1 (represented as a LUT) with the module in stage 2 (represented in the specified primitive set).

**Outline of Paper** Modularity and modular techniques are reviewed. We contrast two types of representation; look up tables and modules. The algorithm is described. Two experiments are conducted, comparing tree based GP with modular GP. The results are analyzed and finally discussed.

## 2 Modularity and Module Identification

An extension to standard GP is the use of modules. In these approaches, each module is represented as a tree and this type of representation can therefore be called a forest. The complexity of a function, which can be defined as the minimum number of nodes in a tree or forest which can represent that function, may be some indication of the difficulty of that problem (i.e. more complex problems are harder to solve). If a modular representation is used, the complexity of a function is independent of the primitive set used (up to a constant) [Woo03]. Thus, if a tree based representation is used, the choice of primitive set would affect the difficulty. Whereas, if a forest representation is used, this dependence would largely be removed. This argument it is used as an inspiration to find a representation which can be searched effectively irrespective of the choice primitive set.

One of the central difficulties with modular methods is how to identify modules. In GP, a score is assigned to the individual as a whole, without considering its component parts. (There is nothing stopping us examining components in isolation, but we will in general not know what components we are looking for in advance). This is general problem is referred to as the Credit Assignment problem. How can credit be given to individual components when it is unclear how much each component contributes to the overall performance of the individual? Heuristics are discussed in sect 3. It is also worth noting that this is also the problem faced by standard GP; how can useful sub trees be identified? (the only difference being that subtrees are called once where modules can be called multiple times).

## 3 Related Work

All modular representations in GP use essentially the same forest representation. A main tree is represented as a tree along with any modules which are also represented as trees (fig 1). The modularization methods differ in the heuristics they use to form modules and move through the search space.

Koza's Automatically Defined Functions (ADFs) [Koz92] have a main result producing branch, and a function defining branch which defines the ADFs. Crossover is done by selecting the corresponding ADFs in two individuals and exchanging subtrees. The number of ADFs and their arguments are decided at the start of the run. Architecture Altering Operations (AAO) [Koz95] remove these extra parameters as they are allowed to evolve, rather than being predefined by the user. Adaptive Representation [RB94] creates modules on-the-fly

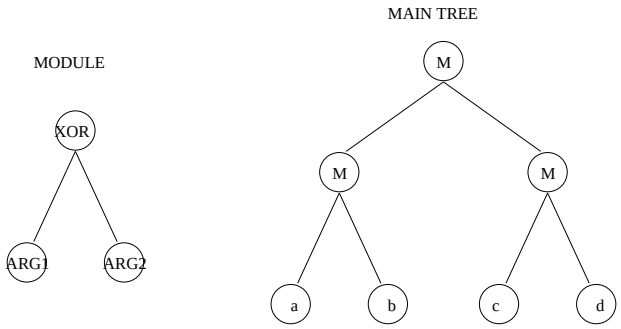


**Fig. 1.** A typical modular representation. A module (left) performs the XOR operation on ARG1 and ARG2. The main tree (right) calls the module 3 times. The overall function performed is even parity with 4 inputs.

during evolution according to population statistics and is similar to AAO in that the form of the representation is free to alter during the run.

In Module Acquisition (MA) [AP92], a subtree is selected from an individual (down to a predefined depth) and is selected as a new *function*, which is made available to other individuals in the population and does not undergo further evolution. With encapsulation[Koz92], however an entire sub tree is chosen and is made it into a new globally defined *terminal*. Howard [How03] maintains a database of subtrees during an initial set of runs and the most commonly observed subtrees are then used as terminals in later runs. However, if subtrees are randomly selected from the database, there is an improvement in performance.

Let us consider what these methods have in common. These methods do not attempt to solve the problem in stages by problem decomposition (as described in sec. 5). They attempt to solve the problem in one go, presenting the solution as a whole. Also, they attempt to discover useful modules by only observing the behavior of the whole individual.

## 4 Look Up Tables and Modules

**Look Up Tables** A look up table (LUT) is an ordered list of input values and corresponding output values (see left hand side of fig 2). If the input values to the LUT are 0 and 1 (2nd row), 1 is returned. A set of test cases is effectively a LUT; it is a list of inputs with corresponding outputs. A LUT and a set of test cases are equivalent and a LUT can be considered as the specification of a module.

A LUT has two properties which make it useful. Firstly, as the inputs are ordered, there is only one way to represent a given function. This is unlike other representations (e.g. tree based representations) where there are many ways of representing the same function. The genotype-phenotype mapping between a LUT and the function it represents is one to one.

Secondly, one can directly manipulate the functionality of a LUT. If we simply want to change the output for a given input we can directly alter the row in the LUT. There is a smooth mapping between phenotype and genotype of a LUT. With a tree based representation, slightly altering the genotype will in general cause a large change in the phenotype. Unfortunately this property is smeared out in this paper as the LUT is embedded in a function which manipulates the output of the LUT.

The downside of LUTs is that we are not able to make predictions about the output of the function on inputs not listed. A LUT may not have entries for some inputs and is therefore undefined for certain values. A LUT which does not list all possible inputs is incomplete and therefore represents a partial function. For example, in the LUT in fig 2 , the input 1 and 0 (3rd row) does not have a output value listed (indicated by'?'). In contrast, modules represent total functions (assuming the primitives they are expressed in are total ).

**Modules** We can consider modules from two different perspectives, building them or using them. If we are concerned with construction, then modules can be constructed directly from these primitives or in terms of other previously defined modules, to build more complex modules. This is how all previous modularization methods view modules.

In contrast, if we are using a module we do not care about its internal workings, we treat it as a black box. In fact we could consider a module, in this respect, as a LUT as we are not concerned with its implementation. This is how our algorithm will view modules.

## 5 The Algorithm

The representation used in this algorithm is motivated by the fact that it is easier to specify *what* we want a program to do (i.e. simply list the inputs and outputs) rather than *how* to do it (i.e. produce the program - which is the aim of GP). Therefore it is proposed that we represent a module as a LUT, rather than implementing it immediately in terms of the primitive set, and when a satisfactory LUT is found we can then begin to search for an implementation of the LUT in the primitive set.

**Problem Decomposition**

Problem decomposition is a 3 stage process,

- Firstly, the problem is decomposed into smaller problems.
- Secondly, these simpler isolated sub problems are solved independently.
- Finally, a complete solution to the original problem is constructed by recomposing the sub solutions, mirroring the first stage.

This process can be done recursively, i.e. sub problems may still be too large to solve and need to be decomposed further before becoming solvable ([Koz94] chapter 1). Koza illustrates this idea with differentiation, where a large expression (which *cannot* be differentiated in one go) is broken down into smaller expressions (which *can* be differentiated in one go) ([Koz94] chapter 3).

Module as look up table

| arg1 | arg2 | output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | ? |
| 1 | 1 | 0 |

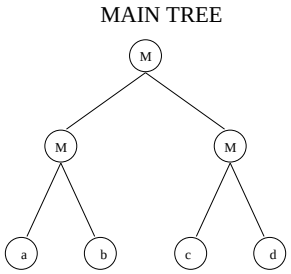


**Fig. 2.** The main tree (right), makes 3 calls to the module (left) represented as a look up table. Given two inputs (labeled arg1 and arg2), the corresponding entry is found in the table and relevant output value is returned. The overall function computed is even parity with 4 inputs, but is undefined for some inputs as the LUT has a missing output value, indicated by a '?'.

**Description of the Algorithm**

A complete individual is represented as a main tree and a LUT (fig 2). (in this paper *one* LUT is used). The search process is a three part process, mirroring the 3 stages of problem decomposition described above.

The first stage consists of two parts, concerned with generating the tree part then the LUT part. First, a tree is generated (by a mutation operator described below), which is composed of primitives and calls to the LUT. Second, a LUT is generated. The inputs for the LUT are generated by presenting the test cases to the whole individual and recording the input values the LUT is exposed to. Then outputs for the LUT are generated at random, a certain number of times. This process of generating a tree, then generating LUTs is repeated until either an individual is found which satisfies the test cases, in which case the second stage can begin, or a termination condition is met. This completes the first stage, which corresponds to the first stage of problem decomposition. If the LUT is complete, we can stop here (as we have a total function), if the LUT is incomplete, we must continue to state two, to find an implementation (which will be a total function).

Secondly, the LUT representation of the module is used as a set of test cases in order to find an implementation of the module using standard tree based GP. The LUT may not be defined for all inputs and only the inputs with defined outputs are used as test cases. In fig 2 only 3 of the rows in the LUT could be used as test cases.

Finally, the LUT in the initial stage is replaced by the implementation of the module found in the second stage. As the LUT (which may not be defined for all values) is replaced by an implementation of the LUT (which is defined for all input values), the overall individual is defined for all values.

The above description applies to the generation of a single individual. We evolve a population of such individuals using replacement selection and a mutation operator. The replacement operator selects a pair of individuals at random, and a mutated copy of the better one replaces the worse of the pair. The muta-

tion operator is used which selects a node in the tree with decreasing probability with depth (i.e. there is a 1/2 chance of selecting a node at depth one, and a 1/4 chance of selecting a node at depth two, and so on). The subtree below the selected node is then replaced with a randomly generated subtree.

To speed up the runs, an incremental approach is used for testing. When a solution is found to the current set of test cases, a new test case is added to the testing set. Initially only one test case is used and this is increased until a predefined upper limit is reached.

## 6 Experiments

**Table 1.** Parameter settings for experiment 1

| | |
|---|---|
| Population size | 10 |
| Number of generations | 1000 |
| Number of runs | 1000 |
| Initial number of test cases | 1 |
| Final number of test cases | 20 |
| Problem | even parity 6 |
| P0 | XOR |
| P1 | OR AND NAND NOR |
| Number of times random output generated for LUT | 100 |
| Number of arguments for LUT | 3 |

The hypothesis that drives this work is that, if a modular representation is used, the choice of primitive set will make less of a difference to the performance of a GP run, whereas if a tree based representation is used there will be a greater difference in performance. Hence 4 types of run were conducted; the two types of representation (tree and modular) with two different primitive sets (P0 and P1) which we will refer to as tree P0, tree P1, modular P0 and modular P1. The same mutation operator is used to manipulate the tree structure of both tree and modular representations. The primitives and parameter settings are listed in table 1.

With this set up, the modular search is potentially 100 times more expensive, because each time a tree is generated, a maximum of 100 randomly generated LUTs are tested with that tree. This is taken into account in the analysis by dividing the number of evaluations by the number of successes to get an estimate of the number of evaluations required to obtain a success. These ratios are presented in table 3, and the lower the number, the more efficient it is. This is the set up in experiment 1 ( table 1).

An alternative would be to put a maximum limit on the number of evaluations per run. This experiment was done with the parameter setting in table 2 and

is called experiment 2. Thus we list the maximum number of evaluations rather than the number of generations.

**Table 2.** Parameter settings for experiment 2

| | |
|---|---|
| Population size | 100 |
| Maximum number of evaluations | 1000000 |
| Number of runs | 100 |
| Initial number of test cases | 1 |
| Final number of test cases | 16 |
| Problem | even parity 4 |
| P0 | XOR NAND |
| P1 | OR AND NAND NOR |
| Number of times random output generated for LUT | 4 |
| Number of arguments for LUT | 2 |

Note that the parameters were not optimized in either experiment. In the first experiment P0 is not logically complete but is expressive enough to solve the problem. In the second experiment P0 is XOR and NAND. Each of these set ups was repeated 5 times to produce a set of results for statistical analysis.

## 7 Results

**Table 3.** Number of evaluations per success for expt 1 for 4 different types of run.

| | | | | | |
|---|---|---|---|---|---|
| Tree P0 | 183418 | 188737 | 190370 | 184033 | 180592 |
| Tree P1 | NA | NA | NA | NA | NA |
| Modular P0 | 315212 | 323165 | 306667 | 288055 | 296379 |
| Modular P1 | 6651595 | 8242125 | 6432609 | 6692874 | 6408986 |

Each row in tables 3 and 4 show the the number of evaluations required to reach a solution (averaged over the respective number of runs) for tree GP with primitive sets P0 and P1 and modular GP with primitive sets P0 and P1 respectively. Figures are rounded down to the nearest integer. NA implies that no solutions were obtained on any of the runs (this is commented on in sec. 8).

**Statistical Analysis** We use the Mann Whitney test to determine if the results were drawn from distributions with the same central tendency.

**Table 4.** Number of evaluations per success for expt 2 for 4 different types of run.

| | | | | | |
|---|---|---|---|---|---|
| Tree P0 | 31054 | 17919 | 21048 | 32811 | 26114 |
| Tree P1 | NA | NA | NA | NA | NA |
| Modular P0 | 31037 | 37799 | 43168 | 35026 | 41381 |
| Modular P1 | 482593 | 437163 | 434827 | 430037 | 553851 |

For experiment 1, the results from the 4 types of run show clear separation with no overlap. We can be confident at the 0.005 level (rank sum of 15) that each of the 4 types of run have a different central tendency.

For experiment 2, the results from the 4 types of run also show clear separation. Only one of the modular P0 set of runs beats tree P0. We can be confident at the 0.010 level (rank sum of 16) that modular P0 runs and tree P0 runs and have different central tendency. We can be confident at the 0.005 level (rank sum of 15) that the other pairs of set ups have different central tendency.

We have shown that different representations, tree P0, modular P0, modular P1, tree P1, perform in this order, to an overall confidence level of 0.010. As the results from the two modular representations are sandwiched between the results from the two tree representations, we can say that this set of experiments supports our hypothesis; modularity reduces the inpact of function set.

## 8 Discussion

It is perhaps not surprising that the 4 different representations perform in the order they do, as this is the ordering of complexity of the solutions. The tree representation with P0 outperforms the forest representation with P0. We conjecture that this is because the module with P0, not only has to find a decomposition but then find a suitable module. With tree with P0, we already have an ideal representation, and we just need GP to string together a solution with all of the variables.

The aim of this paper is to show that if this modular method is used, there is less dependence on the primitive set. We use a basic method to search the space (mutation for the trees and random search for the LUT). This algorithm could be improved by perhaps using a more sophisticated method of determining the output of a LUT. For example, using exhaustive search rather than random search to avoid potentially generating the same LUT. However with random search we still get results which support out hypothesis.

In all of the solutions obtained the LUT was complete (i.e. defined for all values) so we did not continue into stages 2 and 3 of the algorithm. As the LUT was completely defined, no additional predictions could be made even if the LUT was implemented in the primitive set.

It is worthwhile examining the LUTs produced in the evolution. When P1 is used, all of the LUTs (sizes 3 and 2 in experiments 1 and 2 respectively) consist

of even parity or odd parity solutions for the respective sizes. GP has discovered that parity functions make a good modules; *no other modules were observed.* If a human was faced with the task of solving the even parity problem with P1, one would probably construct a module which performs a parity function and then use this to construct a solution.

*Every* modular solution using P1 contained calls to the LUT. However, some modular solutions using P0 did not contain any calls to the LUT (and so no output values were generated as the LUT was not exposed to any input values). This reflects the fact that it is easier for P0 to simply use the primitive XOR directly than it is to reinvent the wheel by evolving a LUT to do the same job. It also indicates that the problem does not need to be further subdivided as it is using primitives rather than LUTs.

There are extra parameters associated with this search algorithm, e.g.the number of LUTs (set to one in this paper). These are the same parameters associated with ADFs. The main difference between this algorithm and ADFs is the representation. To avoid setting these parameters, we could adopt the approach like AAO [Koz95].

The motivation for this work is to find a primitive independent representation and search mechanism. One may have been tempted to propose using different function sets, and then deciding with the benefit of hindsight which function sets are the best. This approach was proposed recently [Wan04], and also demonstrates the interesting phenomena of function groups. Their work differs in that it does not use a modular approach and the are looking for the best function set. Their work could be extended, to implement the best function set in terms of the primitive set of your choice, and this would also achieve our aim (i.e. we can directly translate between primitive sets). However this proposal would miss out on the use of LUTs which have a unique representation for a function and allow us to breakdown a problem.

The frequency distribution of subtrees in Howard's [How03] database is exponential in nature. He conjectures that runs containing subtrees randomly selected from the database (rather than selecting the most frequently observed) perform better due to higher diversity. Wang [Wan04] states that the most appropriate function set is optimally diverse. If we represent modules as LUTs, this naturally maintains diversity as each LUT represents a unique function. The frequency distribution of functionality of LUTs generated at random is uniform due to the one to one mapping, indicating a more effective sampling of functions.

No solutions were found with the tree P1 representation. This maybe because the mutation operator is more likely to mutate a subtree at the top of the tree, preventing trees of sufficient size to evolve. However, using the same mutation operator we can produce solution using the modular P1 representation.

## 9 Further Work

The representation proposed here achieves our aim on this problem. However, we have not investigated the size of the final solutions produced in the different

primitive sets. Given a solution in one primitive set, a solution exists in another primitive set, where the size of the solutions differ by less than a known fixed constant [Woo03]. It would be interesting to see if these solutions could be found.

In this search algorithm, the 'top' level of the program is represented as a tree, and the module is represented as a LUT. Perhaps a better approach would be to also represent the top level as a LUT too, and when a suitable solution is found using this representation, the next stage if implementing *both* the top level LUT and module could begin. Thus the first stage of the algorithm would not involve any primitives at all. We suggest this would further reduces the dependence on the primitive set.

Boolean functions can be represented by finite sized LUTs. Extending the LUT idea to integer and continuous valued functions could prove problematic as they may require infinite sized LUTs. We intend to investigate this by incrementally increasing the number of test cases until we can be confident we have a LUT from which we can induce a module which will capture the underlying function. This was partially why the incremental approach to testing was incorporated.

**Acknowledgements** X. Yao, S. Cattani, D. Gurnell, G. Brown, M. Roberts.

## References

[AP92] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[How03] Daniel Howard. Modularization by multi-run frequency driven subtree encapsulation. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 10, pages 155–172. Kluwer, 2003.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[Koz95] John R. Koza. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In John Robert McDonnell, Robert G. Reynolds, and David B. Fogel, editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, San Diego, CA, USA, 1-3 1995. MIT Press.

[RB94] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Wan04] Soule T. Wang, G. How to choose appropriate function sets for genetic programming. In *Genetic Programming, Proceedings of EuroGP 2004*, Coimbra, Portugal, 2004. Springer-Verlag.

[Woo03] J. R. Woodward. Modularity in genetic programming. In *Genetic Programming, Proceedings of EuroGP 2003*, Essex, UK, 14-16 2003. Springer-Verlag.