Deriving Genetic Programming Fitness Properties by Static Analysis

Colin G. Johnson

Computing Laboratory.
University of Kent at Canterbury.
Canterbury, Kent, CT2 7NF, England.
Email: C.G.Johnson@ukc.ac.uk

Abstract. The aim of this paper is to introduce the idea of using *static analysis* of computer programs as a way of measuring fitness in genetic programming. Such techniques extract information about the programs without explicitly running them, and in particular they infer properties which hold across the whole of the input space of a program. This can be applied to measure fitness, and has a number of advantages over measuring fitness by running members of the population on test cases. The most important advantage is that if a solution is found then it is possible to formally trust that solution to be correct across all inputs. This paper introduces these ideas, discusses various ways in which they could be applied, discusses the type of problems for which they are appropriate, and ends by giving a simple test example and some questions for future research.

1 Introduction

Genetic programming (GP) [2, 11] is concerned with the application of evolutionary algorithms to the evolution of program code. In order to apply evolutionary methods a notion of solution quality (fitness) is needed. Traditionally GP and related techniques have used performance measures on sets of test data as a way of measuring the quality of solutions. However this leads to a number of problems, in particular programs can become overspecialized to their training set and they cannot be (formally) trusted to perform correctly beyond that training set.

This paper discusses the prospects for applying various kinds of static program analysis techniques to this problem. A background section describes these ideas and gives some motivating examples, then two further sections outline reasons why these techniques are potentially useful in genetic programming, and how they can be so applied. A simple example is implemented, and the paper concludes by summarizing these ideas and suggesting a number of questions for future research.

2 Background

Static analysis [16] is a set of techniques which provide information about a program and how it will behave once run, without actually running the program.

In particular static analysis techniques are able to provide information about program behaviour across the whole of the input space of the program, rather than providing information for a particular test case. A number of different kinds of information can be gained by carrying out different kinds of static analysis, e.g.:

- constraint information about relationships between variables [6]
- information about the extreme values which a variable could possibly take at each point in the execution of a program [4]
- usage information about whether facts vital to the solution of a problem have been used
- complexity information, e.g. the number of potential paths through a piece of code [15]
- performance information [17, 21]

As a toy example consider the following fragment of pseudocode (all variables are integers):

```
x = 0;
read y;
z = y+20;
if (y<10)
  for (i=0;i<y;i++)
    x += 2;
```

At the end of the first line it can be seen that `x` must equal 0. By the end of the fragment, it must be at least 0 and at most 20. The variable `z` must be greater than `y` by the end of line 3. All this information can be inferred by systematically tracking the extreme values that variables can possibly take at each line in the program. This process can be automated.

3 Why should we apply static analysis to genetic programming?

GP has proven to be a powerful technique for the solution of problems from a large number of domains. However there are a number of difficulties with GP, in particular relating to the measurement of fitness. The power of GP comes from being able to produce programs which are able to solve a parameterized space of problems. In order to assess the fitness of a member of the population, it is traditional to run the program on a number of test cases, and use the accuracy of the output as the fitness measure.

The first problem with this method of measuring fitness is that it gives no formal assurance that the program will operate on data outside of the test set. Moreover the programs generated by GP are typically incomprehensible to human programmers, so informal *post hoc* analysis of the programs is impossible. Measures of fitness derived from static analysis could be used to ensure that the results will be applicable across the whole of the input space.

Also this may lead to new kinds of fitness measures which make specific use of static features of the program which cannot be discovered simply by running the program. An example of this would be the use of cyclomatic complexity [15] as a component of a fitness measure, which measures the complexity of programs by calculating the number of routes through the code.

Another important problem with GP is that the programs generated can be overfitted to the training data used. Experimental evidence for this is given e.g. by Paterson and Livesey [20]. If the fitness of members of the population is created using measures which apply to the whole of the input space, this problem is removed.

There are also ways in which these techniques could improve the performance of GP itself as well as eliminating problems with existing GP techniques, for example it may be the case that a static analysis may be quicker to perform than running a complex program on many test cases.

Also these concepts have the potential to expand the range of problems which can be tackled using GP. Many problems solved by GP are of the form in which a small sample of inputs gives results which give us confidence about the remainder of the input space. Consider the evolution of high-pass and low-pass filter circuits [3]. In such a problem is it informally reasonable to say if a random sample of inputs gives a reasonable response, then the response will be reasonable across the frequency range. Related arguments can be made for problems like evolving a range of controllers parameterized by a free variable [12]. However in many applications there are too many variables to do this, or else the relationship between input and output is too complex to generalize in a simple way, and these statements will never be rigourously correct.

The kinds of information that can be generated by the static analysis of programs can be divided into two kinds. The first type is results which say something about the *performance* of the program, e.g. its use of cache memory. Secondly the second type is results which measure the *functioning* of the program itself, e.g. results about bounds on output values.

There are a number of potential difficulties with the application of these techniques. The most significant is that the programmer might not be able to get the information which is required in order to assess fitness by using a static measure. This might be for pragmatic reasons (e.g. not being able to write the fitness function in terms of data which can be obtained from the static analysis tools available), or it may be because the problem is *defined* implicitly in terms of a particular data set. Another problem could be that the time taken to carry out the analysis may make it impractical; however for most simple analysis techniques this does not seem to be a problem.

4 Ways in which static analysis can be applied

There are a number of ways in which these ideas might be applied. The applicability of these different techniques is likely to be problem dependent. Three ideas are detailed in this paper.

4.1 Using multicriterion optimization to combine test-driven and static fitness aspects

For some problems some characteristics of a good program may be measurable using static techniques, whilst other characteristics might only be measurable using testing. This might be because the programmer cannot find an appropriate static measure. Alternatively, it may be because the problem is defined implicitly by data, e.g. certain kinds of function regression problems or the "non-programmed computation" problems discussed by Partridge [19].

In particular static analysis lends itself to measuring *performance* aspects of a solution, e.g. measuring the efficiency of memory usage. Such a static measure of a performance aspect can be factored into the fitness measure for any program.

Therefore the first way in which the two ideas might be combined would be to measure some aspects of the program's fitness statically, and some aspects using a traditional test-data approach. These results would then be combined using one of the many techniques which have been developed for multi-criterion optimization [9]. The simplest such technique is to measure the various fitness characteristics of the programs separately, and then to create a fitness from a weighted sum of the measures, where the weights are given by inverse population averages for that factor, so that the total average contribution for each factor is the same.

Other, more sophisticated, techniques could be used to carry out this multi-criterion optimization. An example is Ryan's [23] "Pygmies and Civil Servants" algorithms, which takes two parent populations, one of which has been selected for each of two criteria, and creates children by recombining pairs from each of the two populations. This could be applied to the situation described above by creating two rankings on the population, one a statically-derived measure and one a measure derived from testing.

An example of the kind of performance characteristics which could be measured statically would be the behaviour of the cache memory (e.g. as described in [1, 7, 17, 21]), the efficient distribution of tasks between a number of parallel processors, or the efficiency of garbage collection in a program.

Another example of this would be in evolving solutions to problems where there is an important safety constraint. Two examples will illustrate this. The first of these is evolving some behaviours for a mobile robot, where it is desirable for safety reasons that the robot be not allowed to leave a particular physical area. A second example is in evolving some control mechanism, where some critical value (like the temperature of a machine) is not allowed to go outside a critical range. In both of these examples there are two components to fitness which need to be combined into a single fitness measure. The first of these is a measure of the success of the solution on the task at hand, which could be measured by running the program on test cases. The second is a measure of the safety of the system, which could be derived from a static analysis of intervals and inequalities [4, 5] which the critical value takes whilst the program is running. The inclusion of this second component in the fitness would bias the population towards those solutions which lie within the safe region, and would provide a way of knowing when solutions are within the safe region.

4.2 Improving existing programs

A second way in which these ideas could be used is in improving the performance of existing code whilst maintaining functionality. An example of this would be improving memory performance of an existing program by applying static measures of cache performance [1, 7, 17, 21] to the members of the population. The maintenance of functionality would be by ensuring that the only operations performed on the program are the interchange of *basic blocks* [13] within a program (as a kind of mutation operator), functionality preserving exchanges within those blocks (such as the unrolling of loops) [10] and the interchange of functionally-identical basic blocks between programs (as a kind of crossover operator).

4.3 Using only static measures

The ideal application of these ideas would be to problems where the entirety of the fitness can be derived from static measures. In such a case when solutions are found they can be trusted to be correct across the whole of the input space.

Clearly the types of problem for which this could be applied depend on the sort of information that can be obtained via various forms of static analysis. For certain problems fitness can be defined in terms of variables satisfying certain inequalities or falling within certain ranges, and it would be possible to extend this to optimization problems by creating a hierarchy of inequalities. An example of a problem whose fitness can be defined in this way is a "placement problem" like those outlined in section 5 below.

One perspective on this is that this process is producing programs by specifying *guidelines*, and evolving programs which produce outputs which satisfy those guidelines. So for example the user creates a list of inequalities which the various variables in their program must satisfy at the end of the program. The fitness measure counts the number of these inequalities which are satisfied by the end of the program, *regardless* of input values, by techniques such as tracking the extreme values of intervals and tracking whether changes to variables change a list of inequalities associated with that value [5].

5 A simple example

The main point of this paper has been to discuss the type of problems which can be tackled using this approach rather than to give specific examples. However to finish we shall give a report on some preliminary experiments in which we have implemented a simple example which illustrates these ideas.

The example is a 2-dimensional "placement problem". Given a number of shapes (rectangles for the purposes of this example) and a number of desired relations between those shapes (e.g. "rectangle A must be completely to the right of rectangle B"), the aim of the algorithm is to find a placement of those shapes on a given background region so that the relations are satisfied. The way in which GP is applied to this problem is to derive a program which will solve that problem, parameterised by the lengths of the various rectangles, but with the relationships being fixed. Note that the aim of this is not to apply evolutionary algorithms to the problem directly (as in e.g. [14]), but to find algorithms which can solve a large space of such problems. These types of problems occur in a number of applications: VLSI layout [14], packing problems, and automated layout of windows on a computer screen [8, 22] or widgets within a window. A sample problem is illustrated in figure 1. For small numbers of constraints the problem is trivial, but as more constraints and shapes are added it becomes more difficult.

This problem has been tackled using the GP-like technique of O'Neill and Ryan known as *grammatical evolution* [18]. This uses a BNF grammar to transform a bitstring into a valid program in an arbitrary language, which provides a powerful extension to standard GP. A tableau describing how the grammatical evolution algorithm is applied to this problem is described in table 1.

It is possible to solve this problem using traditional fitness measures, by taking a list of sample cases which satisfy the conditions. However there are a number of problems with this. Firstly there are the problems outlined above; there can be no certainty that the program will work outside our test set, *et cetera*. This is illustrated by attempts to change the size of the space into which the shapes are placed (this could represent placing shapes on a different sized screen, but with the same desired spatial relationships). Also it is difficult to create these test data, so being able to generate the program directly from the constraints is valuable.

For this problem fitness is derived statically in the following way. For each line of the function, a set of data is updated containing information about the relationships between the variables. In this example two pieces of data are tracked: one is the extreme values which the variable values can take, and the other is a boolean variable which tracks whether a variable must be greater than or equal to zero at that point, regardless of which route through the program was used to access that point. The following are examples of the kind of update rules which are used to update the latter variable (space precludes the inclusion of a full list):

- If a positive constant is assigned to the variable, then the "known to be positive or zero" flag for that variable is set to true.
- If the flag is currently true and the variable is incremented by a value (variable or constant) which is itself known to be positive, then the flag remains true.
- If the variable is decremented by a variable about which nothing is known, then the flag is set to false. This illustrates the "conservative" nature of the flag—it is not measuring for certain whether the value is positive/zero at a particular program point, it is measuring whether, regardless of which route

the program had taken to reach the current point, it is possible to make the statement at that point.
- If the variable is assigned to a sum of values (constant or variable) which are known to be positive and values to which the absolute value function has been applied, then the flag is true.

It would be possible to add more transformations to the list; however the list can always be finished by saying "in all other cases the value becomes false", i.e. it is always possible to say that no statement is being asserted with confidence at that point. In this way an analyser can be gradually built up by replacing conservative approximations with more concrete statements as the analysis program is improved.

The complete analyser contains a large number of such update rules. At the end of a run of the analyser on a program, a list of conditions is produced which the output from the program must satisfy, regardless of input. These can then be checked for compatibility against the desired conditions. For each condition which is met, the fitness is incremented. In the implementation described here, different kinds of conditions are weighted, so that the fitness function is not swamped by lots of easy-to-satisfy conditions early on.

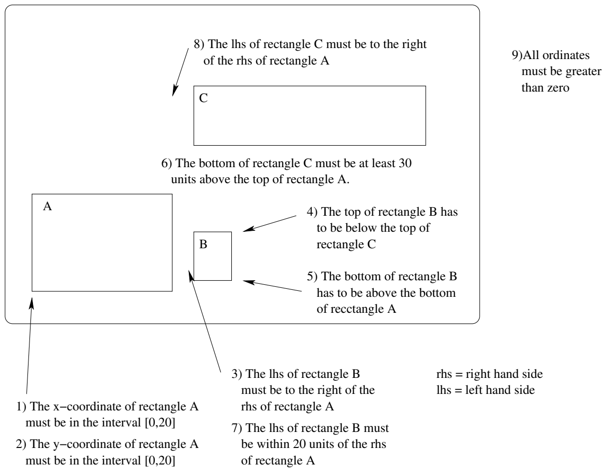

Fig. 1. A sample placement problem.

Results from two sample runs of a test example (showing best fitness and average fitness for a weighted sum of constraints) are presented in figure 2. The two runs have different mutation probabilities. Clearly evolution is working to ensure that more conditions are being rigourously satisfied with time. However three conditions (out of 16) are not satisfied, and these are some of the more complex conditions. What appears to be happening is that evolution is converging on a number of rather short programs which satisfy a reasonable number of

Objective.	To find a placement for a number of rectangles, given the length and width of each rectangle, so that they satisfy a set of conditions stated as intervals and inequalities.
Terminal operators	The binary operators $+, -, \times$, and the unary operators increment, decrement, absolute value.
Terminal operands.	LValues: Position of each rectangle and a number of scratch variables. RValues: Length and width of each rectangle, position of each rectangle, the scratch variables and a number of fixed constant values.
Fitness cases.	The fitness is not measured by running the program on test cases. Instead it is measured by keeping track of the extreme values which variables can take, and keeping track of whether it is possible to say rigourously that variables are positive or negative at each program point, and using these intervals and inequalities to compare with the list of required constraint values.
Raw fitness.	A weighted sum of the number of conditions satisfied by the rectangles in their final position at the end of program execution. Interval conditions are weighted 1, simple inequalities between two variables are weighted 2, and more complicated inequalities are weighted 5.
Wrapper.	C code to transform the list of arithmetic statements into a C function.
Parameters.	Population size = 500, termination after 50 generations, probability of mutation = 0.001 or 0.01 per bit, probability of crossover = 1.0, elitist selection, one point crossover.
Success predicate.	The program can be terminated if all the conditions become satisfied.

Table 1. Grammatical evolution tableau for a sample placement problem.

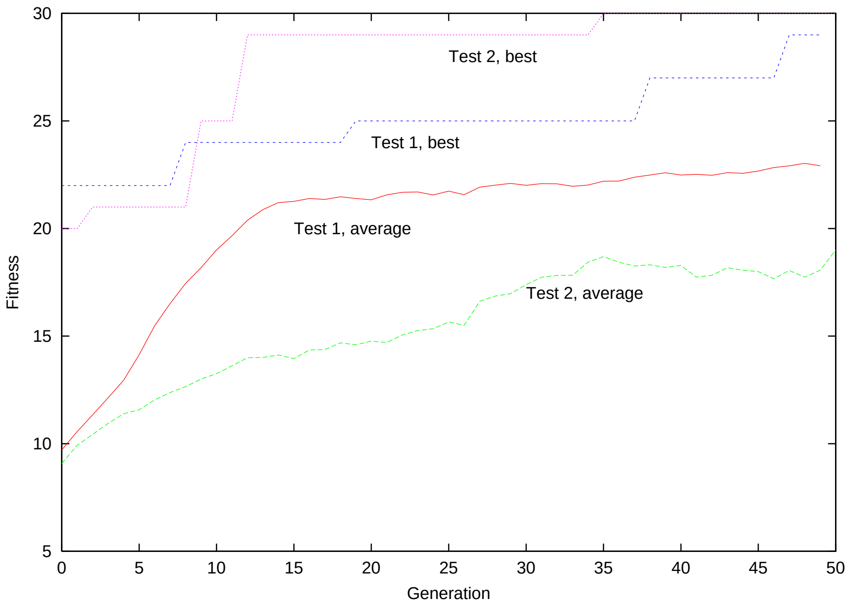

Fig. 2. Two fitness results for a placement problem (test 1: mutation probability 0.001 per bit, test two mutation probability 0.01 per bit).

conditions, and then crossover cannot work to swap information into these short programs.

Two strategies are being employed to attempt to alleviate this. Firstly explicit length information is being factored into the fitness function. Secondly a fitness function is being developed where the weightings given to each constraint depend on the comparative rarity of the constraint in the population, and this is being further emphasized by the use of fitness scaling. Preliminary results from the latter suggest that this is effective in maintaining diversity of constraints satisfied in the population.

6 Conclusions and research questions

In this paper we have discussed various ways in which static analysis techniques can be used in genetic programming and related areas. A simple example has been given, however the main piece of future work is to apply these ideas to a broad range of problems and gather detailed information about performance. Another piece of work which would help in applying these methods would be to gather together details about the different types of information which can be gained from different types of static analysis, and bring these together into some common format so that they can be combined into composite fitness measures. Once such a structure is in place it will be easier to get a feel for the range of problems to which these ideas can be applied.

References

1. Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium 1996*, pages 52–66. Springer, 1996.
2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, 1998.
3. Forrest H Bennett III, Martin A. Keane, David Andre, and John R. Koza. Automatic synthesis of the topology and sizing for analog electrical circuits using genetic programming. In Kaisa Miettinen, Marko M. Mäkelä, Pekka Neittaanmäki, and Jacques Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 199–229, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons.
4. P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 224 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.
5. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

6. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.
7. Saumya Debray. Resource-bounded partial evaluation. In *Proceedings of the 1997 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 179–192, 1997.
8. Bjørn Freeman-Benson. Converting an existing user interface to use constraints. In *ACM Symposium on User Interface Software and Technology*, pages 207–215, 1993.
9. Jeffery Horn. Multicriterion decision making. In Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors, *Handbook of Evolutionary Computation*, pages F1.9.1–F1.9.15. Oxford University Press / Institute of Physics, 1997.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
11. John R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection.* Series in Complex Adaptive Systems. MIT Press, 1992.
12. John R. Koza, Jessen Yu, Martin A. Keane, and William Mydlowec. Evolution of a controller with a free variable using genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty, editors, *Proceedings of the 2000 European Conference on Genetic Programming*, pages 91–105. Springer, 2000. LNCS 1802.
13. James Larus. Whole program paths. In *Programming Language Design and Implementation*, 1999.
14. Pinaki Mazumder and Elizabeth M. Rudnick. *Genetic Algorithms for VLSI Design, Layout and Test Automation.* Prentice-Hall, 1998.
15. Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.
16. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.
17. K. D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. In *ACM PLDI Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.
18. Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.
19. Derek Partridge. Non-programmed computation. *Communications of the ACM*, 43(11):293–302, 2000.
20. Norman Paterson and Mike Livesey. Evolving caching algorithms in C by genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference.* Morgan Kaufman, 1997.
21. P. Puchner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1:159–176, 1989.
22. Michael J. Rees. Comparison of user interface design constraints for CGI and java applet web applications. In *Australian World Wide Web Technical Conference*, pages 1–14, 1997.
23. Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 243–263. MIT Press, 1994.