Hierarchical Genetic Programming Based on Test Input Subsets

David Jackson
Dept. of Computer Science
University of Liverpool
Liverpool L69 3BX, United Kingdom
Tel. +44 151 795 4297
d.jackson@csc.liv.ac.uk

ABSTRACT
Crucial to the more widespread use of evolutionary computation techniques is the ability to scale up to handle complex problems. In the field of genetic programming, a number of decomposition and reuse techniques have been devised to address this. As an alternative to the more commonly employed encapsulation methods, we propose an approach based on the division of test input cases into subsets, each dealt with by an independently evolved code segment. Two program architectures are suggested for this hierarchical approach, and experimentation demonstrates that they offer substantial performance improvements over more established methods. Difficult problems such as even-10 parity are readily solved with small population sizes.

Categories and Subject Descriptors
D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence]** Automatic Programming – *program synthesis*; I.2.6 [**Artificial Intelligence**] Learning – *induction*.

General Terms
Algorithms, Design, Performance, Experimentation.

Keywords
Genetic programming, hierarchical GP, decomposition, program architecture.

1. INTRODUCTION
A key area of research in genetic programming (GP) and in other areas of evolutionary computation is that of discovering how to scale up such techniques to deal with complex, real-world problems. The usual approach of the human programmer is to decompose the original problem into sub-tasks that are simpler to solve, and then combine the resulting lower-level modules into a solution to the full problem. It seems natural, then, that similar divide-and-conquer approaches should form the basis for solving difficult problems by evolutionary means.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'07, July 7-11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007…$5.00.

In GP, decomposition and reuse techniques can take various forms, the best-known paradigm being that of Koza's Automatically Defined Functions (ADFs) [10-12]. In this, the architecture of programs is defined in such a way that a number of function-defining branches are evolved in tandem with a main value branch that may make calls to the functions. For many problem domains, it has been found that ADF-based systems can provide much better performance than conventional GP systems [15].

An alternative to the use of ADFs is the technique of Module Acquisition introduced by Angeline and Pollack [1, 2]. In this, portions of individuals are randomly encapsulated as modules that are protected from the effects of the usual evolutionary operators. Modules are stored in a library, their value being determined by how often they are used by evolving individuals.

Other approaches are more systematic (i.e. less random) in the modularisation decisions that are made. The Adaptive Representation through Learning (ARL) algorithm [16], for example, identifies and extracts subroutines from offspring which exhibit the best improvements on the fitness of their parents. Similarly, Roberts et al [14] describe a two-stage scheme in which module selection is based on sub-tree survival and frequency.

The importance of modularisation has also been recognised for genetic programming systems that use representations which differ from the tree structure usually employed. In his work on Cartesian Genetic Programming (CGP), Miller has described bottom-up techniques for building hardware circuits from simpler 'cells' [13], and for encapsulation of modules in evolving CGP programs [18, 19].

In much of the existing research, the emphasis is on attempts to encapsulate useful code segments as they evolve, and it is usually not known in advance what functionality or structure these fragments will possess. A very different approach is one in which modules are associated with predetermined sub-tasks, so that the evolution of each module is guided towards a specific goal. However, if we are to avoid having to make use of intelligent intervention to define these sub-problems, then other, more mechanistic criteria are needed. In the next section, we explore these alternatives more fully, and in particular we describe and assess some previous work on a problem decomposition technique based on the input test cases used to evaluate the fitness of individuals in a GP population. In subsequent sections we go on to propose two novel program architectures aimed at addressing the limitations of the earlier work, and we assess the approaches

experimentally before finishing with some concluding remarks and suggestions for future research.

2. PREVIOUS WORK

In a previous paper [9] we examined the use of *layered learning* [3, 4, 17] as a means for solving GP problems in a hierarchical fashion. The essence of the layered learning approach is to decompose a problem into subtasks, each of which is then associated with a layer in the problem-solving process. The idea is that the learning achieved at lower layers when solving the simpler tasks directly facilitates the learning required in higher subtask layers.

In Layered Learning Genetic Programming (LLGP) [5-9], evolution begins in an initial layer that proceeds towards the solution of a sub-task of the overall problem. The stopping criterion for this may be, say, that a solution to the sub-task is found, or that a pre-defined limit on the number of generations has been reached. The genetic material produced at the end of this layer then forms the basis for the initial population of the next layer, which uses what has been 'learnt' in layer 1 to help it evolve towards a solution of the original problem. In practice, two layers may be sufficient, although the approach could in principle be extended to multiple layers. There are also various ways in which genetic material can be propagated from one layer to the next: the population as it stands at the end of one layer may be simply re-cast as generation zero of the subsequent layer, or the best individuals from the lower layer may be used exclusively to seed the higher layer.

A key issue in layered learning is deciding how to decompose a problem into tasks to be associated with the lower layers. There are at least three ways in which this can be done:

1. Based on the original problem specification, identify a component sub-task.

2. Attempt to solve a lower-order form of the same problem.

3. Make use of a subset of the test cases normally used to evaluate the fitness of individuals.

Most of the published work on layered learning adopts the first of these approaches. For example, in research done on the application of layered learning to the game of *keep-away soccer* [5-8], it was decided that in attempting to solve the problem of preventing the opposing team from gaining possession of the ball, it would be useful firstly to evolve accurate passing between team-mates without an opposing player present. The drawback of such prior decomposition is that it requires an intelligent understanding of the problem at hand, plus at least some insight into the components that would be useful in forming a solution.

The other two strategies are far more mechanistic, and were the subject of our investigations in the earlier paper [9]. As described there, approach number two involves the solution of a problem (e.g. the even-5 parity problem) by reusing genetic material obtained from solving lower order versions of that problem (e.g. even-2 or even-3 parity). Experimentation showed that the technique significantly out-performs not only conventional GP systems, but also those that make use of Automatically Defined Functions (ADFs).

The way that approach number three works is that only a subset of the test cases normally used to evaluate the fitness of an individual is considered in the initial layer. Once a solution to this sub-problem has been found, or another stopping criterion reached, the process is ramped up to a second layer in which all test cases are used to drive evolution. For example, in the even-4 parity problem, exhaustive testing involves 16 combinations of values on the binary inputs {D0, D1, D2, D3}. In the layered learning approach as we tried it, a lower layer attempts to solve for, say, 4 or 8 of these input combinations. The whole of the population at the end of layer 1 then becomes the initial population of the upper layer, which searches for programs that work for all 16 test cases.

In a range of experiments, layer 1 was allowed to proceed until given saturation levels of solutions to the sub-task were reached. However, the performance of this strategy was disappointing: although modest improvements were seen in some cases, in general the approach fared little better than a standard GP system.

In the following sections we describe how, rather than abandoning the test subset approach entirely, we have investigated alternative ways of employing it.

3. A FUNCTION-BASED ARCHITECTURE

A possible reason for the failure of the test-subset approach to improve on performance is that although potentially useful pieces of code are evolved in the first layer, there is then no mechanism for subsequently protecting those fragments from the deleterious effects of crossover and other evolutionary operators. Perhaps what is needed, then, is some form of encapsulation to turn the useful code sections into indivisible modules. This suggests a hierarchical approach to the form of genetic programming that is used.

As mentioned earlier, the best-known of the hierarchical approaches to GP is that of Koza's Automatically Defined Functions (ADFs) [10-12]. In implementing this, the structure of population members is more tightly specified than is usual in GP. Figure 1 shows a typical architecture for programs in an ADF-based system.

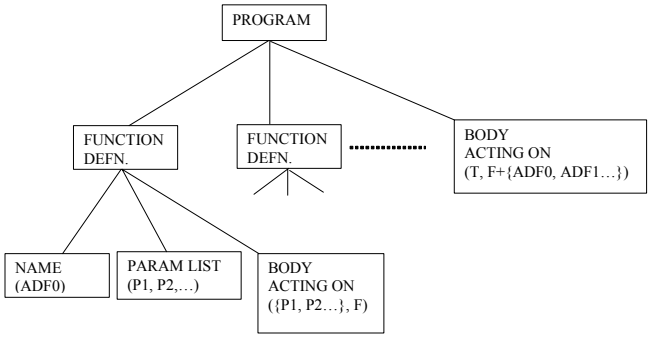

Figure 1. Typical architecture of an ADF-based GP system

In this architecture, a program tree comprises one or more function-defining branches and a main branch that may invoke those functions. Each function definition has a formal parameter list, and only members of this list can appear as terminals in the body of the function; the terminal set of the problem itself is not accessible within an ADF. The body of the program as a whole – its main branch – is built from the usual terminal set, members of the problem function set, and members of the set of ADFs. During evolution, operators are usually subject to some context restrictions; in crossover, for example, if the first parent's crossover point is within, say, ADF0, then the second parent's crossover point is also confined to ADF0. In this way, all of the function branches and the main branch evolve simultaneously towards a solution to the problem.

For our own purpose, a slightly different hierarchical architecture is proposed; this is shown in Figure 2.

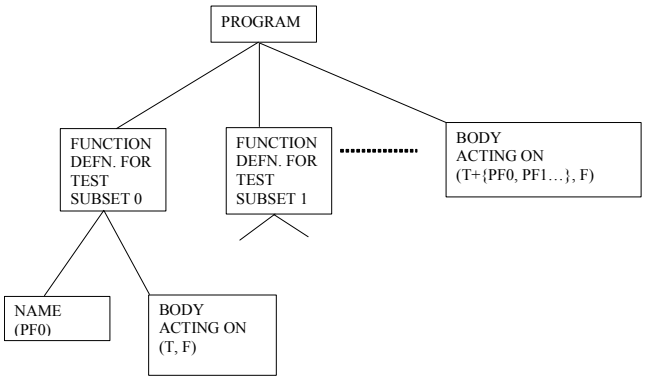

Figure 2. Function-based architecture for test-subset approach

As before, each program consists of several function defining branches plus a main branch. However, there are several key differences between this architecture and that of Figure 1. Firstly, in the ADF approach the number of function branches is arbitrary, whilst in our approach it is determined by the number of subsets of test input cases. Secondly, each ADF does not evolve towards a particular fixed and independent goal; rather, its evolutionary worth is judged according to its contribution to the fitness of the individual as a whole. In our new architecture, on the other hand, each function is assessed separately, its fitness being determined by the number of test cases it fulfils within its assigned subset. Thirdly, as is made clear in the diagrams, ADFs are defined in terms of formal parameter lists, whereas the functions in the new architecture are parameterless functions (PFs). This in turn leads to a fourth difference, which is in the composition of the function and terminal sets. In the ADF approach, the terminals appearing in the body of a function are taken solely from the formal parameter list, and the function set in the main branch is augmented with the set of ADFs. In the new architecture, the normal terminal set for the problem is used to build function bodies. Moreover, since these functions are parameterless, they act as terminal nodes of the main branch, and so it is the terminal set rather than the function set which is expanded to include them. (Indeed, the phrase Automatically Defined Terminals (ADTs) has been considered for these new entities).

Other differences lie in the way the evolutionary process is conducted. Rather than evolving all branches in concert as in an ADF-based system, the proposed approach focuses all of its evolutionary effort on one branch at a time (at least, for a uniprocessor machine). As soon as a PF has evolved, it is propagated to all members of the population, since there is little point in evolving multiple versions of functionally identical code. Then, when all PFs have been produced, evolutionary effort is shifted onto the main program branch.

In evaluating this architecture, we begin with the even-parity problem, one of a class of Boolean problems that is known to be difficult for GP to solve. In the even-4 version, the aim is to evolve a Boolean design that returns a TRUE output if the number of logic one values on its 4 inputs D0-D3 is even, FALSE otherwise. The parameters for the problem as we have implemented it in our GP systems (before any expansion of the function and terminal sets) are given in Table 1.

Table 1. GP parameters for the even-4 parity problem

Objective	To evolve a program capable of determining if the number of logic 1s on the 4 inputs is even
Terminal set	D0, D1, D2, D3
Function set	AND, OR, NAND, NOR
Initial population	Ramped half-and-half
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	16, representing all combinations of inputs
Fitness	Number of mismatches with expected outputs (0-16)
Success predicate	Zero fitness (solution found)
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

The performance of our subset-based system can be compared against a conventional GP system, and also against one which makes use of ADFs. For the latter, we have followed Koza's precept [10] of enabling the evolution of one function for each of the arities from 2 up to n-1, where n is the size of the terminal set for the problem. Hence, for the even-4 parity problem, we allow for one ADF with 2 parameters, and a second with 3 parameters (although not all formal parameters need be accessed within the body of a function).

For our parameterless function approach, we also need to make a decision as to how many functions are required, and this in turn is governed by how we choose to partition the test cases. For even-4

parity, exhaustive testing using all combinations of the four inputs {D0, D1, D2, D3} requires 16 test cases. In this experiment, we will assess the effectiveness of declaring 2 functions, the first dealing with the integer values 0-7 on the four binary inputs, the other dealing with the values 8-15. We will also evaluate the effect of using four PFs, dealing with values 0-3, 4-7, 8-11 and 12-15, respectively.

In comparing approaches, we make use of the success rate at finding solutions over 100 runs, each of 50 generations. We also make use of Koza's metric of computational effort [10], defined as the minimum number of individuals that must be processed to achieve a 0.99 probability that a solution will be found. Table 2 presents these figures for each of the systems we have described.

Table 2. Comparative performance of the PF approach on the even-4 parity problem

Approach	Success rate (%)	Comp. Effort
Standard GP	14	700,000
ADF GP	43	97,500
2 PFs, 8 cases each	55	116,000
4 PFs, 4 cases each	59	72,000

When only 2 PFs are used, the solution-finding success rate of the subset-based approach is substantially better than standard GP; it is also significantly better than ADF-based GP, although more computational effort is required. When 4 PFs are used, both the success rate and the computational effort are better than either of the more established approaches.

Figure 3 shows the graph of best fitness for one successful run of the subset-based approach, using 2 PFs dealing with 8 cases each. From generations zero to 7, the system is evolving the code for PF0, which handles test cases 0-7. In the initial population, the best individual has a fitness value of 3, i.e. it cannot solve for 3 of the 8 test cases. The code for PF0 is evolved at generation 7, and in generation 8 a new population is created to evolve PF1, which handles test cases 8-15. This is completed by generation 12, and in generations 13-17 the main branch of the program is evolved, the code for which must pass all 16 test cases.

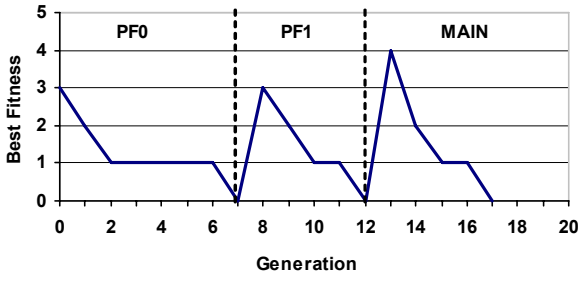

Figure 3. Graph of best fitness for one run of the PF approach on even-4 parity

It is important to bear in mind that the code evolved for a PF is specific to the test cases for which it is responsible, and is in no sense a generalised solution. For example, in a two-PF program for even-4 parity, PF1 will give correct answers for all of the integer values 8-15 encoded on the inputs {D0, D1, D2, D3}; however it may very well give incorrect results for other integer encodings. In a sense, these other results are 'don't care' values, and it is the job of the main program branch to screen these out and ensure that only valid outputs are reflected in the program as a whole.

The success of the approach for even-4 parity encouraged us to try it for the more difficult even-5 parity problem. The only changes to the problem parameters given in Table 1 are an additional input D4, a corresponding increase in the fitness cases to 32, and an increase in population size from 500 to 2000. As before, we experimented with two versions of the subset-based system: one with 4 PFs dealing with 8 of the 32 test cases each, and one with 8 PFs handling 4 cases each. The results are compared in Table 3.

Table 3. Comparative performance of the PF approach on the even-5 parity problem

Approach	Success rate (%)	Comp. Effort
Standard GP	0	-
ADF GP	32	864,000
4 PFs, 8 cases each	58	564,000
8 PFs, 4 cases each	10	3,584,000

Like Koza [10], we found that discovering a solution to the even-5 parity problem using standard GP is extremely difficult. By incorporating an ADF mechanism we were able to get much better results, with a success rate of 32%. When we try the test subset approach using 4 PFs, the success rate is almost double that achieved in the ADF system, leading to a large decrease in the computational effort. However, when we attempt to evolve solutions with 8 PFs, the performance drops markedly. We shall return to the reasons for this in the next section.

To evaluate the approach further, we applied it to the majority-on problem. In this, the aim is to evolve a program that is capable of determining whether the majority of its Boolean inputs are set to logic-one. Thus, in the 5-input version, a solution will deliver TRUE if three or more inputs are logic-one, and FALSE otherwise. The function set for the problem is F={AND, OR, NOT}, but other parameters for the problem as we have implemented it are the same as given for the even-parity problem. In creating the ADF version of the GP code, we have again used Koza's rule of thumb, so that for the majority-5-on problem there are three ADFs, with arities 2, 3 and 4. For the subset-based approach, we have experimented with 2 PFs of 16 cases each, and 4 PFs of 8 cases each. The results are given in Table 4.

Table 4. Comparative performance of the PF approach on the majority-5-on problem

Approach	Success rate (%)	Comp. Effort
Standard GP	62	49,000
ADF GP	7	945,000
2 PFs, 16 cases each	86	24,000
4 PFs, 8 cases each	41	91,000

An unusual characteristic of this problem is that the performance of the ADF version is substantially worse than that of the conventional GP approach. Much better than either of these is the subset-based approach with 2 PDFs, with a computational effort only half that of standard GP. Again, however, when the number of PFs is increased to 4, the performance drops: although still overwhelmingly better than the ADF system, it is somewhat worse than standard GP.

4. A SELECTION ARCHITECTURE

It is clear from the experimentation of the previous section that the test subset approach can lead to substantially better performance than more conventional GP methods. However, the number of parameterless functions employed is critical to optimum performance, and this phenomenon merits further attention.

When the number of PFs is low, the size of the test subset assigned to each PF must be quite high. This means that the PF itself must be reasonably complex to deal with the relatively large number of cases, and so it may take some time to evolve. On the plus side, the main branch has to accommodate only a small number of additional functions, and so may evolve quite quickly.

Conversely, a large number of PFs means that each function is comparatively easy to evolve, because it has to deal with only a small number of test cases. However, it becomes correspondingly more difficult to evolve the main branch. Not only does it have to select terminals from a greatly expanded set, it also has to combine these in such a way that the useful outputs from the PFs are given prominence, while the 'don't care' outputs described earlier are masked out. The difficulty of this is seen time and time again in the experiments: the PFs evolving very quickly, often within a generation, with the necessarily more complex main branch subsequently failing to evolve at all.

But what if there were ways to simplify the main branch? In fact, what if it were possible to do away with the main branch completely?

Once a PF has evolved successfully, what is generated is a self-contained program which, on application to a pre-determined set of test inputs, always produces the correct results, whilst for other test inputs it produces garbage. In other words, function X may work for test case A, but not B. However, the complete partitioning of test cases means that there will exist another function Y that will work for test case B. Suppose, then, that we define our program architecture in such a way that test case A is always referred to function X for solving, while test case B is always passed to function Y. If this were in place, each PF would receive only the input values it had evolved to handle, and so would never generate any garbage outputs.

The architecture we propose is shown in Figure 4. Since there is no longer a main branch, there is no function calling, and code fragments that were previously encapsulated as parameterless functions now simply become branches of the main program. The root node of this program takes the form of a 'select' function. Its job is essentially to route control flow to the appropriate branch, according to the current program input values. Its precise form is left deliberately vague; its exact nature will depend on the problem being solved and the language being used to encode evolved programs. It will usually take the form of a 'switch' or 'case' statement, or perhaps even a nested 'if-then-else' construct.

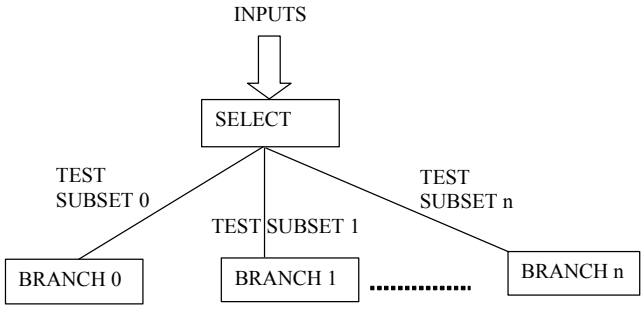

Figure 4. Selection architecture for test-subset approach

The primary advantage of using this architecture is that we need evolve only the branches responsible for the test case subsets, exactly as we did earlier for the parameterless functions. The select node at the root of the program tree takes care of everything else. The total amount of evolutionary effort required to discover a solution may therefore be greatly reduced.

Table 5 shows the performance results obtained for the even-4 parity problem when using this selection architecture. The experiment was performed with 2 branches dealing with 8 test cases each, and 4 branches dealing with 4 cases each.

Table 5. Performance of the selection approach on the even-4 parity problem

Approach	Success rate (%)	Comp. Effort
2 branches, 8 cases each	71	59,500
4 branches, 4 cases each	100	3,500

These figures should be compared with those given for conventional, ADF-based and PF-based GP in Table 2. It is patent that the selection architecture leads to substantially improved performance. When four branches are used, the branches are trivially easy to evolve, several often appearing together in the initial population. Since we have eliminated the need to evolve

additional code to combine these four fragments, the overall computational effort is tiny.

The following shows the form of code that is generated as a solution to the even-4 parity problem when using four branches:

```
SWITCH (INT(D3..D0))
CASE 0..3:
NAND(OR(D0 D1) NAND(D0 D1))
CASE 4..7:
NOR(NOR(OR(D1 D2) OR(D1 D1)) NAND(OR(D0 D1)
NAND(D0 D1)))
CASE 8..11:
AND(AND(OR(NAND(OR(D1 D0) AND(D1 D1))
OR(NAND(D1 D0) NAND(D3 D1))) OR(NOR(NOR(D2
D0) NAND(D2 D2)) OR(D1 D0))) OR(NAND(NOR(D2
NOR(D0 D0)) NOR(NAND(D0 D0) NOR(D0 D2)))
NAND(NAND(AND(D1 D2) NOR(D0 D3)) AND(AND(D2
D3) NAND(D3 D1)))))
CASE 12..15:
OR(AND(AND(NAND(AND(D1 D0) OR(D2 D1))
AND(NAND(D0 D3) NAND(D1 D2))) AND(OR(AND(D2
D1) NAND(D3 D0)) OR(NAND(D1 D3) OR(D2 D3))))
AND(D1 D0))
```

and here is a solution when 2 branches are used:

```
SWITCH (INT(D3..D0))
CASE 0..7:
NAND(NAND(NOR(D3 D1) AND(AND(NAND(D2 D2)
NOR(D2 D0)) OR(NOR(D3 D0) NAND(D0 D2))))
NAND(NOR(NOR(NOR(AND(D1 D2) AND(D1 D3))
NOR(AND(D0 D1) AND(D2 D3))) AND(NOR(D1 D2)
NAND(D0 D1))) OR(AND(AND(D2 D1) OR(D1 D1))
OR(NOR(D1 D2) D0))))
CASE 8..15:
NOR(NOR(NOR(AND(AND(D2 D1) NAND(OR(D1 D3)
AND(D1 D0))) OR(NAND(D2 D2) OR(D3 D1)))
NAND(AND(NAND(D0 D0) AND(D2 D1)) D2))
NOR(AND(NOR(NAND(D1 D1) AND(D0 D1)) OR(D1
NAND(D0 D1))) NAND(OR(AND(D0 D2) NAND(D2
D2)) NAND(NAND(NAND(NOR(AND(D2 D2)AND(D0
D0)) OR(NAND(D0 D0) D2)) OR(AND(AND(D0 D2)
NAND(D2 D1)) AND(AND(D1 OR(D1 D0)) NAND(D2
D1)))) OR(D1 D0)))))
```

In each case, the integer value on the four binary inputs {D0, D1, D2, D3} is used to select the appropriate branch to execute. An important point to note here is that each branch is a stand-alone piece of code that deals with a particular test case subset, and that the garbage values it produces for unrecognised inputs are no longer an issue. Because of this, it becomes possible to combine branches from different programs in a 'mix and match' approach to solution generation. For example, if we are looking for the shortest solution, we can bring together the shortest branches obtained over a sequence of runs. If we do this for the 4-branch version of our even-4 parity problem, we can build the following program from the various solutions:

```
SWITCH (INT(D3..D0))
CASE 0..3:
NAND(OR(D0 D1) NAND(D0 D1))
CASE 4..7:
NOR(NOR(D1 D0) AND(D1 D0))
CASE 8..11:
AND(OR(D0 D1) NAND(D0 D1))
CASE 12..15:
NAND(NAND(D0 D1) OR(D1 D0))
```

An interesting feature of this concocted program is that although it is a solution to a four-input problem on D0-D3, the branches are expressed in terms of the two inputs D0 and D1 only. The size of the solution is 29 nodes (including the root node); this compares with the 59-node smallest program found in the standard GP runs, and the 33-node shortest solution in the ADF-based GP runs.

Table 6 shows the performance of the selection architecture when applied to the even-5 parity problem; this can be compared with the results given earlier in Table 3. Similarly, Table 7 shows how the selection architecture fares on the majority-5 problem; this can be contrasted with the results of Table 4. In both cases, the superiority of the new approach is evident.

Table 6. Performance of the selection approach on the even-5 parity problem

Approach	Success rate (%)	Comp. Effort
4 branches, 8 cases each	91	192,000
8 branches, 4 cases each	100	16,000

Table 7. Performance of the selection approach on the majority-5-on problem

Approach	Success rate (%)	Comp. Effort
2 branches, 16 cases each	90	19,500
4 branches, 8 cases each	100	6,500

In general, as the number of test cases per branch is lowered, each branch becomes correspondingly easier to evolve, and solutions can be found with a comparatively small population size. At the same time, small test subsets imply a large branch count, so that even though only a small number of generations may be needed to evolve each branch, the maximum number of generations per run may have to be greatly increased in order to allow enough evolutionary time to generate all branches. Utilising this knowledge allows us to solve comparatively difficult problems by using only small GP populations, simply by extending run lengths to include a sufficient number of generations. A final demonstration of the efficacy of the selection architecture approach is presented in Table 8, which gives the performance figures obtained from solving the even-10 parity problem using a population size of only 2000. To achieve this, the program architecture has been given 256 branches, each handling just 4 test cases, and the maximum run length is set at 500 generations. It is perhaps worth remarking that the figure of computational effort for this approach to the even-10 parity problem is lower than that required for standard GP to solve the even-4 parity problem.

Table 8. Performance of the selection approach on the even-10 parity problem

Approach	Success rate (%)	Comp. Effort
256 branches, 4 cases each	100	628,000

5. CONCLUSIONS

In this paper we have proposed two novel program architectures for enabling the hierarchical decomposition and evolutionary solution of problems, via consideration of subsets of the test input cases normally used to evaluate the fitness of individuals.

In the first of these approaches, parameterless functions (PFs) are evolved to handle each of the test case subsets. The main program branch can include calls to these new entities via an expanded terminal set. Our experiments have shown that it is possible to achieve performance levels that are much improved over standard and ADF-based GP, but that crucial to this is judicious selection of the number of PFs, which may be difficult to determine in advance.

In the second approach, the complexity and associated problems of the main program branch are removed by eliminating that branch entirely. This is achieved via an architecture which contains a selection primitive at its root node. Experimental results for this approach are impressive, and offer the ability to solve complex problems such as even-10 parity with relative ease.

The proposed architectures also have additional advantages over more conventional systems. Unlike ADFs, for example, each PF and selection branch is evolved according to its own pre-defined goal and fitness criterion. As such, these code fragments are evolved independently of each other and of any main branch. This provides several benefits. Firstly, as we saw earlier, it becomes possible to mix-and-match fragments from different solutions. Secondly, it becomes possible to apply different evolutionary techniques to different fragments. In each of our experiments, all test subsets were of the same size, but in principle it would be possible to define test subsets of different sizes and other characteristics, and to employ a variety of techniques in solving them. Thirdly, the opportunities for parallel execution are obvious. In the PF-based architecture, it would be a simple matter to evolve all PFs in parallel, and then evolve the main branch. In the selection architecture, all branches can evolve in parallel.

Another advantage not brought out in previous discussion is that of storage requirements. The abstract views of the proposed architectures as we have presented them imply that all branches are contained within all individuals. In fact, this is not necessary in an implemented system. Once a PF or selection branch has evolved, it is propagated to all individuals, but this does not require a physical copying of code to every member of the population. Instead, the evolved code fragment is moved to a global data structure, accessible by all individuals. In this way, only code for the currently evolving PF or selection branch is ever contained within individuals. This is in contrast to, say, an ADF system, in which each individual must have physical sub-trees for each function and the main value branch.

An argument that could be levelled against the selection architecture is that it is an artificial device that absorbs some of the complexity that would otherwise have to be dealt with by the evolutionary process. Our counter-argument to this is that, far from being an exotic and problem-specific device, the selection construct is something that is found in all programming languages. In its implementation it is nothing more than a standard case statement or nested if-then-else clause. As such, its form remains constant across all applicable problems, and does not have to be tailored to a specific domain. Finally, a selection function could, if one wished to be so purist, be evolved as a separate exercise.

That said, it is clear that any solution generated using our architectures is a very different animal from that which would arise in a more conventional GP system. A program takes the form of an interconnected set of relationships, rather than a single relationship over all inputs, and this gives rise to a number of questions about its properties. For example, how generalisable is the approach? How effective is it when the test input set is not exhaustive, but contains ‘gaps’ ? How well, if at all, does the approach extend to problems in which fitness is assessed not by input-output relationships but by dynamic behaviour over time? These questions and many others we hope to address in on-going work.

6. REFERENCES

[1] Angeline, P.J. and Pollack, J. Evolutionary Module Acquisition. In *Proc. 2nd Annual Conf. on Evolutionary Programming*, La Jolla, CA, 1993, 154-163.

[2] Angeline, P.J. and Pollack, J. Coevolving High-Level Representations. In *Artificial Life III,* Langton, C.G. (ed.), Addison-Wesley, 1994, 55-71.

[3] Asada, M., Noda, S., Tawaratsumida, S. and Hosoda, K. Purposive Behaviour Acquisition for a Real Robot by Vision-Based Reinforcement Learning. In *Machine Learning*, vol. 23, 1998, 279-303.

[4] deGaris, H.: Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules. In *Proc. Seventh International Conf. on Machine Learning (ICML-90),* Porter, B.W. et al (eds), 1990, 132-139.

[5] Gustafon, S.M. Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem. M.S. Thesis, Dept. of Computing and Information Sciences, Kansas State University, USA, 2000.

[6] Gustafon, S.M. and Hsu, W.H. Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem. In *Proc. EuroGP 2001, Lecture Notes in Computer Science vol. 2038*, Miller, J.F. et al (eds), Springer-Verlag, Berlin Heidelberg, 2001, 291-301.

[7] Hsu, W.H. and Gustafon, S.M. Genetic Programming and Multi-Agent Layered Learning by Reinforcements. In *Proc. GECCO 2002*, New York, NY, USA, 2002, 764-771.

[8] Hsu, W.H., Harmon, S.J., Rodriguez, E. and Zhong, C. Empirical Comparison of Incremental Reuse Strategies in Genetic Programming for Keep-Away Soccer. In *GECCO 2004 late-breaking papers*, 2004.

[9] Jackson, D. and Gibbons, A.P. Layered Learning in Boolean GP Problems. In *Proc. EuroGP 2007, Lecture Notes in Computer Science, vol. 4445,* Springer-Verlag, Berlin Heidelberg, 2007, 148-159.

[10] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[11] Koza, J.R. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[12] Koza, J.R. Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming. In *Proc. 5th International Conf. Genetic Algorithms (ICGA-93)*, 1993, 295-302.

[13] Miller, J.F. and Thomson, P. A Developmental Method for Growing Graphs and Circuits. In *Proc. 5th International Conf. on Evolvable Systems*, Trondheim, Norway, 2003, 93-104.

[14] Roberts, S.C., Howard, D. and Koza, J.R. Evolving Modules in Genetic Programming by Subtree Encapsulation. In *Proc. EuroGP 2001, Lecture Notes in Computer Science, vol. 2038*, Miller, J. et al (eds), Springer-Verlag, Berlin Heidelberg, 2001, 160-175.

[15] Rosca, J.P. and Ballard, D.H. Hierarchical Self-Organization in Genetic Programming. In *Proc 11th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 1994, 251-258.

[16] Rosca, J.P. and Ballard, D.H. Discovery of Subroutines in Genetic Programming. In *Advances in Genetic Programming 2*, Angeline, P. and Kinnear, K.E. Jr. (eds), ch. 9, MIT Press, Cambridge, MA, 1996, 177-202.

[17] Stone, P. and Veloso, M. Layered Learning. In *Proc. 17th International Conf. on Machine Learning*, Springer-Verlag, Berlin Heidelberg, 2000, 369-381.

[18] Walker, J.A. and Miller, J.F.: Evolution and Acquisition of Modules in Cartesian Genetic Programming. In *Proc. EuroGP 2004, Lecture Notes in Computer Science vol. 3003*, Keijzer, M. et al (eds), Springer-Verlag, Berlin Heidelberg, 2004, 187-197.

[19] Walker, J.A. and Miller, J.F. Improving the Performance of Module Acquisition in Cartesian Genetic Programming. In *Proc. GECCO 2005*, Beyer, H-G. and O'Reilly, U-M. (eds), ACM Press, New York, 2005, 1649-1656.