

An Adaptive Mapping for Developmental Genetic Programming

Steve Margetts and Antonia J. Jones

Department of Computer Science, Cardiff University,
Queen's Buildings, Newport Road, PO Box 916,
Cardiff CF24 3XF, U.K.

Email: S.Margetts@cs.cf.ac.uk

Phone: +44 (0)29 2087 4812

Abstract. In this article we introduce a general framework for constructing an adaptive genotype-to-phenotype mapping, and apply it to developmental genetic programming. In this preliminary investigation, we run a series of comparative experiments on a simple test problem. Our results show that the adaptive algorithm is able to outperform its non-adaptive counterpart.

1 Introduction

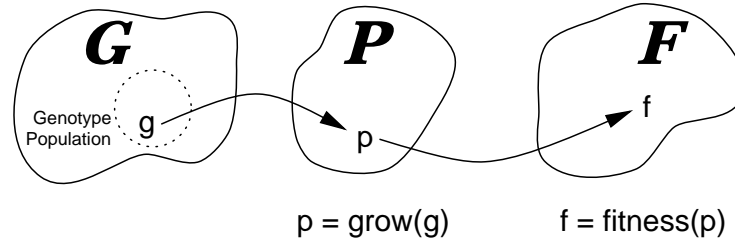
Genetic programming can be thought of as the construction of programs guided by evolutionary search. The fitness of a program is measured by executing it, and the aim of the system is to find a program with some desired functionality. In the case of traditional genetic programming, programs are represented by LISP S-expressions [5]. New programs are created from old by the application of various genetic operators acting directly on the LISP expressions themselves. There is therefore no distinction between the structures that undergo evolutionary modification (the genotypes) and the structures that are used to calculate fitness (the phenotypes).

This does not have to be the case: we could use binary strings [1], or even executable graphs [13] as our genotypes. The study of genetic programming with alternative genotype structures is known as *developmental genetic programming* [3] [11].

One of the advantages of separating the genotype \mathbb{G} from the phenotype \mathbb{P} is that the genetic operators used by the system can be simpler, and better suited to the structures on which they act. The general scheme is given in figure 1, where the mapping from genotype to phenotype is represented by the function **grow**. The dotted circle represents the current population.

One way to do this is to specify a *fixed* mapping from the genotype to the phenotype. But as each problem is different and requires a different problem-specific function set, we must find a new mapping for each. Unfortunately, there are often many different ways to convert a particular type of genotype into a program, and not all of these are equal [4]. Given that we want to obtain the

Fig. 1. An illustration of a genotype to phenotype mapping. The symbols \mathbb{G} , \mathbb{P} and \mathbb{F} denote the set of genotypes, phenotypes and fitness values respectively (we usually take \mathbb{F} to be the positive real numbers). The function `grow` converts a given genotype into its corresponding phenotype, which can then be assessed using the fitness function `fitness`.



best possible performance from our genetic programming systems, selecting the best genotype-to-phenotype mapping is a critical but difficult task.

In this article we present an *adaptive* genotype-to-phenotype mapping for genetic programming. This mapping is able to adjust itself in response to feedback on its progress, and thus side-steps many of the problems of a fixed mapping. We will start by outlining the nature of the genetic programming system we are interested in. After describing our adaptive mapping, we will then compare it with a fixed mapping on a simple test problem.

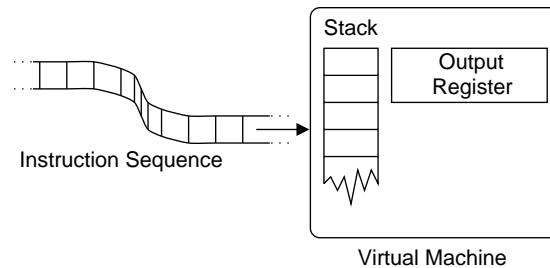
2 Stack-Based Genetic Programming

One of the simplest abstract computing devices is a stack machine. The field of genetic programming with such machines is termed *stack-based genetic programming*, and has been shown to be at least as effective as traditional genetic programming [8] [12]. A program for a stack-machine is simply a list of instructions, each of which alters the state of the machine. In this respect, a program can be thought of as a type of “machine-code”. If we regard each machine instruction as an element of a function set \mathbb{F} , then the phenotype for our system is \mathbb{F}^* , the set of all possible strings drawn from \mathbb{F} .

The machine we will use here consists of a general-purpose stack combined with an output register, and a schematic is shown in figure 2. We can broadly classify the instruction set for this machine into the following: constants, variables, problem-specific functions, and general-purpose functions that affect the stack directly.

Given a program in the form of a list of instructions, we process each one in turn. If we encounter a constant or a variable we push its associated value onto the stack. If we encounter a function, we take the required number of arguments from the stack, present them to the function, and push any return value back onto the stack. (If there are insufficient arguments on the stack, then the function simply does nothing.)

Fig. 2. A schematic of the virtual stack machine used in our developmental genetic programming system. The value in the register at the end of the run is used as the program’s output.



To evaluate the fitness of a program, the genetic programming system sets the values of any variables, sets the output register to zero, and submits the program to the stack-machine. The output of the program is taken to be the value in the register at the end of the program. Coupled with the fact that functions with insufficient arguments are ignored, this means that *any* sequence of instructions forms a valid program.

Stack-based genetic programming provides a natural distinction between genotype and phenotype. Having specified the computational engine and the phenotype we hope to use, we must now turn our attention to the genotype. Following the work of [1], we will choose fixed-length binary strings. This means that we can use a standard genetic algorithm [2] as our genetic programming system.

3 An Adaptive Genotype to Phenotype Mapping

We now need some way to generate an instruction sequence (the phenotype) from a fixed-length binary string (the genotype). One possible way to do this is to associate each instruction in the function set \mathbb{F} with a unique binary sequence. We would like the number of binary digits representing each instruction to be as small as possible, as this will ensure that the system can make the best use it can from the fixed-length binary string. However, if \mathbb{F} has n elements, we will need at least $\lceil \log_2 n \rceil$ digits to represent each instruction.

If the number of instructions is an exact power of two, then there is no problem: we can easily assign one sequence to each symbol. If not, then we will have some “spare” sequences, and we must decide how to deal with these. We believe that what we do with these spare sequences in the decoding process is critical to the performance of the algorithm as a whole. If we encounter an unrecognised binary sequence when decoding a particular binary string, our options are:

- Ignore it, i.e. skip over a entire block of bits
- Skip single bits until we come to a sequence we recognise
- Encode one or more functions multiple times

All of these options have disadvantages. Skipping over unassigned blocks or enough bits until we reach a valid code is wasteful in that we are not using the binary string to its fullest. And encoding one or more functions multiple times necessarily biases the system towards these functions as multiply-encoded symbols are more likely to appear in a binary string drawn at random.

In effect, encoding a symbol multiple times increases its relative importance, and imposes a ranking upon the function set. This is not necessarily a bad thing: if we could arrange the ranking in such a way as to emphasize those functions that are “important” for a given problem, we may be able to improve the performance of our algorithm. Of course, the difficulty here is in choosing an appropriate ranking. The “best” ranking of the functions in the function set is likely to be problem dependent, and may even change during the execution of the algorithm.

The function set could also contain elements that are of little or no use for the problem being tackled. This situation is particularly relevant to symbolic regression problems, as we often do not know which inputs are important in terms of accurately modelling the training data. Each “useless” function makes the problem harder, as not only must the algorithm learn to solve the problem, it must also learn to avoid using these functions.

3.1 Using a Huffman-Decoding Mapping

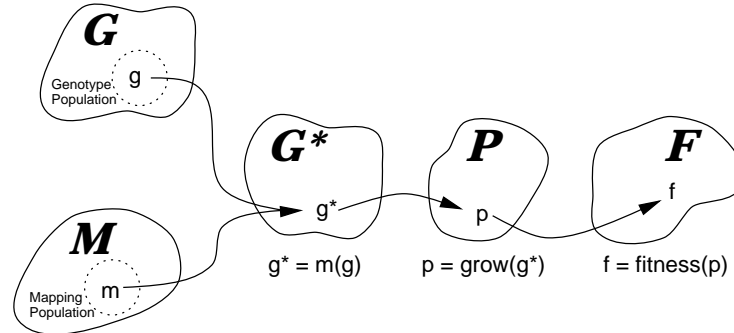
The solution is to allow the algorithm to choose its own assignment of binary sequences to function symbols. One way we can do this is by *Huffman encoding* (e.g. [10]), which works in the following manner. Imagine for a moment that we have a message consisting of a sequence of symbols that we wish to send over an error-free communication channel. Suppose also that the channel can only send binary signals, and that we require the transmission time to be minimised. The problem is then to represent the message efficiently as a binary string.

The simplest solution is to assign each symbol in the message a unique binary sequence. To create the message we simply convert each symbol into its binary sequence and transmit it. Provided the receiver has a copy of the encoding table used, it is easy to decode the message. However, this message is unlikely to be the shortest possible.

A better solution is to take account of the composition of the message. One approach is to note that the frequencies of symbols in any “typical” message are likely to be non-uniform. That is, we expect certain symbols to occur more often than others (for example, the most common letter in a typical piece of English prose is ‘e’). If we can arrange for these frequently-used symbols to be transmitted using a small number of bits, we will reduce the overall message length. The idea of Huffman encoding is a simple extension of this: we aim to transmit each symbol with a number of bits that is proportional to its frequency in the message. Again, once we have decided on an encoding, the message can easily be converted into a binary string and transmitted.

Using this scheme, each symbol may now be represented using a different number of bits. To decode a symbol we can use a *decoding tree*, which has as its branches the values ‘0’ and ‘1’, and symbols at its leaves. To decode a binary

Fig. 3. An illustration of an adaptive genotype to phenotype mapping. The symbols \mathbb{G} , \mathbb{M} , \mathbb{G}^* , \mathbb{P} and \mathbb{F} denote the set of genotypes, mappings, intermediate genotypes, phenotypes and fitness values respectively. An intermediate genotype is constructed by combining an element of \mathbb{G} (a “solution”) with an element of \mathbb{M} (a mapping). This is then converted into a phenotype by the function `grow`, and can then be assessed by the fitness function `fitness`.



sequence we follow the path through the tree until we reach a leaf node, where we return the associated symbol. Provided the receiver has the encoding we used to create the transmission, it is a simple matter to recover the original message.

The shape of the decoding tree, and thus the binary sequence for each symbol, is controlled entirely by the frequency information used to construct it. Hence we can adjust the encoding simply by adjusting these frequencies. However, we are not actually interested in *encoding* a sequence of symbols, but in *decoding* the binary string, so the frequencies here refer to the average number of occurrences of a symbol in a program generated from a random binary string, rather than the frequency of a symbol in a message.

To recap, what we have here is a way of representing the encoding of a set of function symbols into binary strings as a real-valued vector. Our overall aim is to let the algorithm control this vector, so that it can choose which mapping it should use on a per-problem basis. To do this, we turn to the idea of an *adaptive mapping*.

3.2 A Coevolutionary Model for Adaptive Mappings

Our basic idea is very simple: we use coevolution (in the sense of [9]) to search for a useful mapping at the same time as searching for a solution to the problem. To do this, we therefore establish *two* populations: one to represent the possible solutions (i.e. what would be the set of genotypes in a standard genetic programming system), and the other to represent the possible mappings.

A general framework for this scheme is given in figure 3. Here, the sets of possible solutions and mappings are denoted by \mathbb{G} and \mathbb{M} respectively, and the dotted circles within each set represent the individuals of the current population.

To evaluate the fitness of a candidate genotype g , we must first select a member of the mapping population m . In our implementation, we have chosen to select the m with the current highest fitness, although other schemes are possible. However we choose m , the result is g^* , a member of the “intermediate” genotype space, to which we can apply the (fixed) mapping function `grow`. This yields the phenotype p , to which the fitness measure is applied.

The intermediate genotype space \mathbb{G}^* is intended to separate the actual genotype from the phenotype. The reason for this is that in many cases, the set of valid phenotypes is a subset of all possible phenotypes. In such cases it would be rather difficult to constrain each possible mapping the algorithm might generate to produce only valid phenotypes. We can then implement any restrictions on the allowable phenotypes in `grow`, the mapping between \mathbb{G}^* and \mathbb{P} . Note that in most cases, we can take $\mathbb{G} = \mathbb{G}^*$.

While we can see how to find the fitness of a member of \mathbb{G} , it is less clear how we can evaluate the fitness of a candidate mapping m . Perhaps the best method is to evaluate the fitnesses of the entire population of solutions in the context of m , and take the average fitness of these as the fitness of m . Unfortunately, this is too computationally expensive to be practical.

Instead, we will evaluate the fitness of the member of \mathbb{G}^* created by applying the current best member of the solution population to m . Although this does not return as much information about the mapping as an exhaustive evaluation of the entire population of solutions, it does mean that the best mapping will match itself to the best solution and vice-versa. The resulting algorithm is also pleasantly symmetric.

In summary, the use of coevolution in this scheme naturally breaks the problem into two symbiotic parts: one being the search for a mapping and the other being the search for the solution.

4 Experimental Setup

To evaluate this idea, we will use the *maximum output* problem (e.g. [6]). The task here is to produce a program that takes no arguments but which outputs a numeric value that is as large as possible, using just simple arithmetic operators and constants. The function set for our version of this problem is listed in table 1.

We ran a set of comparative experiments between a non-adaptive algorithm and our adaptive algorithm on this domain. For the non-adaptive algorithm, we generated a random encoding of binary strings to the function set, as given in table 1. The adaptive algorithm was free to evolve its own encoding in the manner outlined above.

In using *fixed-length* binary strings as genotypes we are effectively constraining the maximum size of the programs the system can produce. To investigate the effects of this we ran experiments using genotypes with 50, 100, 150, 200 and 250 bits. The binary strings used for the population of adaptive mappings were fixed at 70 bits, using 10 bits for each of the 7 frequencies in the mapping. All

Table 1. The function set and default encoding used by the non-adaptive algorithm on the maximum output problem.

FUNCTION SYMBOL	ENCODING	DESCRIPTION
plus	011	Addition: pop two items from stack, add and push result onto stack
times	00	Multiplication: pop two items from stack, multiply and push result onto stack
const	110	Constant: interpret next N bits as number and push value onto stack
dup	100	Duplicate item at top of stack
pop	111	Remove item at top of stack
s2r	101	Copy item at top of stack into output register (does not affect stack)
r2s	010	Push output register onto stack

Table 2. The results of the non-adaptive algorithm on the maximum output problem. The mean and best outputs were calculated over the 10 independent runs for each genotype size, each run being 5000 function evaluations. The mean number of instructions used were calculated from best programs produced at the end of each run.

No. BITS	MEAN VALUE	BEST VALUE	MEAN NUMBER OF INSTRUCTIONS
50	19797	65536	13.4
100	515367	2.0×10^6	25.7
150	2.0×10^7	8.4×10^7	40.9
200	3.3×10^7	1.9×10^8	56.3
250	6.5×10^7	3.4×10^8	67.9

real values were generated in the interval $[0, 1]$, using the counting-ones mapping described in [7].

To ensure a fair comparison all experiments used a population of 200 individuals in total (i.e. the adaptive algorithm used two populations of 100 individuals). We ran each experiment for a total of 5000 function evaluations, repeating each one 10 times. The best and average value of the output at the end of each experiment was recorded.

5 Results

The results for the non-adaptive algorithm are shown in table 2. This table gives the mean and best output values achieved, along with the mean number of instructions used by the best programs in each case.

Table 3. The results of the adaptive algorithm on the maximum output problem. The mean and best outputs were calculated over the 10 independent runs for each genotype size, each run being 5000 function evaluations. The mean number of instructions and the mean function frequencies were both calculated from the best programs produced at the end of each run.

No. BITS	MEAN VALUE	BEST VALUE	MEAN INST.	MEAN FREQUENCIES						
				const	pop	dup	s2r	r2s	plus	times
50	5953	59049	12.9	0.48	0.37	0.50	0.50	0.40	0.44	0.35
100	2.5×10^6	8.0×10^6	29.5	0.39	0.29	0.51	0.54	0.30	0.43	0.34
150	1.7×10^7	7.2×10^7	43.2	0.30	0.23	0.52	0.46	0.38	0.42	0.26
200	2.2×10^8	1.1×10^9	59.3	0.39	0.42	0.56	0.66	0.54	0.68	0.45
250	6.8×10^9	4.3×10^{10}	85.1	0.34	0.28	0.39	0.51	0.38	0.56	0.38

The results for the adaptive algorithm are shown in table 3. This table gives the mean and best output values achieved, and the mean number of instructions used by the best programs as before. It also gives the mean frequency of each symbol in the mappings used by the best programs at the end of the run.

It is clear from these tables that increasing the string length allows the programs to output larger values. We can also see that in all but the first case, the adaptive algorithm tends to produce longer programs than its non-adaptive counterpart, particularly when using longer genotypes. Perhaps because of this, the adaptive algorithm tends to produce greater output values for most of these cases.

It is worth taking a quick look at the case where the adaptive algorithm was outperformed by the standard one in more detail. The best result for the 50-bit string is shown below. (Instructions in italic type do not contribute to the final value returned by the program.)

```
const(1) dup dup plus dup plus s2r dup times dup times dup
times s2r
```

Note particularly the repeated “dup times” sequence, which squares the value at the top of the stack. This short sequence is a good way to produce a large number, and similar expressions were found in all the best programs. The number of wasted instructions here is small: only the central *s2r* function does not contribute to the overall value. By using only 50 bits to represent a program, we appear to be forcing the system to be economical in its usage of functions. With such short bit-strings, it is likely that any individuals which do not maximise their use of their genotype will be out-evolved.

The corresponding program for the adaptive algorithm is given next, along with the encoding used to construct this solution. It is clear that this program is longer than the one above, even though it produces a smaller output.

```
dup const(1) dup dup dup plus plus dup times dup dup times dup
times times s2r
```

Encoding						
const	pop	dup	s2r	r2s	plus	times
111	011	00	10	-	010	110

This program is longer because the mapping ignores the `r2s` function. As this instruction pushes the contents of the output register onto the stack, it is of little direct use in this problem. By ignoring this function, two of the remaining symbols can be represented using only two bits. As one of these is the useful `dup` function, this represents a substantial saving.

As a comparison, we can look at the best program generated by the non-adaptive algorithm on a 250-bit genotype (below). We can see that this program has a higher proportion of “useless code”; in one place it even calls the `pop` function, which removes the item at the top of the stack.

```
times plus const(0.6) times plus const(1) times const(0.9)
plus times plus s2r r2s times r2s plus s2r dup r2s times dup
dup pop s2r dup times plus r2s times dup s2r times dup dup
s2r r2s times times dup plus r2s plus plus const(0.9) s2r s2r
plus dup s2r plus plus times r2s plus dup plus r2s plus times
times times s2r r2s plus dup plus s2r times times dup plus
r2s plus s2r r2s
```

To complete the set we have the best program and mapping found by the adaptive algorithm when using 250 bits.

```
dup dup dup plus pop const(0.8) s2r plus dup plus dup plus
r2s plus s2r r2s times dup times dup times s2r s2r r2s times
dup times plus dup r2s plus plus dup s2r plus plus r2s r2s
plus s2r r2s dup plus plus dup dup plus plus dup plus s2r dup
plus r2s plus dup dup plus s2r s2r r2s dup plus plus dup plus
dup plus plus dup plus dup plus dup plus s2r r2s s2r r2s
```

Encoding						
const	pop	dup	s2r	r2s	plus	times
0000	0001	111	01	001	10	110

The mapping found here is slightly different to that found for the 50-bit program above: instead of removing a symbol entirely, the mapping represents both the `const` and the `pop` instructions using four bits. As the `const` symbol is used only once in the program, and the `pop` symbol is not used at all, the program does not need to worry too much about using four bits to represent this symbol. However, doing so allows the remaining symbols to be represented with only two or three bits. The resulting program is longer, and so returns a larger value than its fixed-mapping counterpart.

6 Conclusions

The adaptive algorithm is able to outperform the non-adaptive algorithm as it is able to maximise the length of the program that it can extract from a fixed-length binary string. In addition, it is able to bias the function set to those instructions that are best suited to the problem at hand.

7 Future Work

This work is perhaps best regarded as a preliminary study – it has always been our intention to apply this idea to other genetic programming problems, for example that of symbolic regression. The ability to remove functions from the function set in this domain provides a form of feature-selection. This is because we have instructions that represent the inputs from the dataset. It may be that inputs that are not “important” in predicting the output will be dropped from the function set.

In investigating this simple problem, we have noticed that the simple iterative mapping from binary strings to instruction sequences is inherently brittle. By this, we mean that making a small change near the beginning of the string can cause a huge change in the functionality of the program, simply because each instruction can affect all those that follow it. One idea that we are currently considering is a way to use the idea of an adaptive mapping to evolve a more robust encoding.

References

1. Wolfgang Banzhaf. Genotype-phenotype-mapping and neutral variation – a case study in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of Parallel Problem Solving from Nature III*, pages 322–332, Berlin, 1994. Springer.
2. David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley Publishing Company Inc., 1989.
3. Robert E Keller and Wolfgang Banzhaf. Genetic programming using genotype-phenotype mappings from linear genomes to linear phenotypes. In John R Koza, David E Goldberg, David B Fogel, and Rick L Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, Cambridge, USA, 1996. MIT Press.
4. Robert E Keller and Wolfgang Banzhaf. The evolution of genetic code in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E Eiben, Max H Garzon, Mark Jakiela, and Robert E Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1077–1082, San Francisco, California, 1999. Morgan Kaufmann. ISBN 1-55860-611-4.
5. John R. Koza. *Genetic Programming - On the Programming of Computers by Means of Natural Selection*. The MIT Press, Massachusetts, Cambridge, 1992. ISBN 0-262-11170-5.

6. William B. Langdon and Ricardo Poli. An analysis of the MAX problem in genetic programming. In John R. Koza, K. Deb, Marco Darigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *GP97: Proceedings of the Second Annual Conference on Genetic Programming*, pages 222–230, Stanford University, USA, July 1997. Morgan-Kaufmann.
7. Steve Margetts and Antonia J. Jones. Phlegmatic mappings for function optimisation with genetic algorithms. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 82–89, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.
8. Timothy Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 148–153. IEEE Press, 1994.
9. Mitchell A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution*. Phd thesis, George Mason University, Fairfax, Virginia, Spring 1997. Supervised by Kenneth A. De Jong.
10. Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company Inc., 1992. ISBN 0-201-51059-6.
11. Lee Spector and Kilian Stoffel. Ontogenetic programming. In John R Koza, David E Goldberg, David B Fogel, and Rick L Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, Cambridge MA, 1996. MIT Press.
12. Kilian Stoffel and Lee Spector. High-performance, parallel, stack-based genetic programming. In John R Koza, David E Goldberg, David B Fogel, and Rick L Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Cambridge MA, 1996. The MIT Press.
13. Astro Teller. *Advances in Genetic Programming II*, chapter 3: Evolving Programmers: The Co-evolution of Intelligent Recombination Operators. MIT Press, 1996.