

Memory with Memory in Tree-Based Genetic Programming

Riccardo Poli¹, Nicholas F. McPhee², Luca Citi¹, and Ellery Crane²

¹ School of Computer Science and Electronic Engineering, University of Essex, UK
{rpoli, lciti}@essex.ac.uk

² Division of Science and Mathematics, University of Minnesota, Morris, USA
mcphee@morris.umn.edu, seidaku@gmail.com

Abstract. In recent work on linear register-based genetic programming (GP) we introduced the notion of Memory-with-Memory (MwM), where the results of operations are stored in registers using a form of soft assignment which blends a result into the current content of a register rather than entirely replace it. The MwM system yielded very promising results on a set of symbolic regression problems. In this paper, we propose a way of introducing MwM style behaviour in tree-based GP systems. The technique requires only very minor modifications to existing code, and, therefore, is easy to apply. Experiments on a variety of synthetic and real-world problems show that MwM is very beneficial in tree-based GP, too.

1 Introduction

In the vast majority of programming models, assignments are entirely destructive in the sense that an instruction that stores a result in a register or in a memory location completely overwrites their previous value. This overwriting model of assignment was carried over to most versions of genetic programming (GP) that had state and assignments (see [8] for a review). This includes linear GP [2], which evolves sequences of (virtual or real) machine code instructions that usually act by destructively writing to registers or memory locations. Other examples include indexed memory [1,3,11], and work on evolving data structures (such as stacks) that have internal state [4,5]. Similarly, systems that use data structures such as stacks in their computational model, such as PushGP [9,10], manipulate internal states with destructive assignments.

This is in contrast to most biological systems, where the state of such a system is rarely if ever completely replaced with a new state with no regard for or “memory” of the previous state. Changes in protein concentrations within a cell, for example, can happen quickly but are typically incremental in nature, each new state being constructed via modification of the previous state rather than the complete replacement of it.

Noting this dichotomy between nature and GP, in previous work [7] we introduced the idea that the results of operations in a linear register-based GP system could be stored in registers using a form of *soft assignment*. Instead of having the new value completely overwrite the old value of the register, these soft assignments combine the old and new values. We called this technique *Memory-with-Memory (MwM)*.

Although there are many approaches that could be taken to combining the old values with the new when performing assignments, in [7] we found that using a simple weighted average of the old value of a register with the new value being assigned

worked well. In particular, if v_{old} is the original value of the register, and v_{new} is the new value being assigned to the register, the resulting value v_{result} is given by

$$v_{\text{result}} = \gamma v_{\text{new}} + (1 - \gamma)v_{\text{old}} \quad (1)$$

where γ is a constant that indicates the *assignment hardness*, allowing users to determine how “soft” the assignment operator is.¹ In [7] we showed that including this new type of assignment in a linear GP system can significantly improve performance on a variety of symbolic regression problems. Also, MwM was found to change the nature of bloat, reducing the amount of ineffective code.

A question that naturally comes to mind is whether an extension of the MwM idea to mainstream tree-based GP is possible. Obviously, MwM could be used in any GP system which includes primitives that read and write into some form of memory: all one needs to do is to make assignments to memory soft. However, memory is rarely used in tree-based GP, where most applications evolve functions with no side effects rather than programs. For these, the standard form of MwM cannot be used.

In this paper we propose an alternative way of introducing MwM type of behaviour in mainstream, tree-based GP. This requires only very minor modifications to existing systems, making it easy to add. We describe the technique and our GP system in Secs. 2 and 3. Sec. 4 shows that MwM is beneficial in a variety of settings, including a real EEG reconstruction problem. We conclude in Sec. 5.

2 Memory with Memory in Tree-Based GP

The MwM technique for register-based GP is essentially a way of ensuring that previously computed results cannot entirely be wiped by a single instruction. In tree-based GP, instructions do not write their results into registers. Instead, they pass them as a return value to other instructions higher up in the tree. Those instructions, in turn, use the results to compute a new return value, and so on. To see how one could apply the MwM idea to this style of GP, we need to dissect MwM into its most elementary components.

With MwM, executing instructions effectively involves two steps: a) a calculation and b) a soft assignment of the result of the calculation into a register or memory location. In tree-based GP, executing instructions also consists of two operations. One is, again, a calculation, while the second consists of passing of the result of the calculation to a parent instruction higher up in the tree. Hence, in a tree-based GP system, because calculation results are not stored in memory, what matters is how the results of calculations are turned into return values. If we allow the result of a calculation to entirely determine the value returned by a node in the tree, we essentially have a “*hard*” *return operation*. If, instead, we allow the return value of a node to be a blend between the result of a calculation and previously returned results (we will explain what this means in a moment), then we create a “*soft*” *return operation*. This allows us to achieve the objectives of MwM for tree-based GP systems where instructions have no memory or side effects. For this reason, we will call this technique *MwM for tree-based GP*.²

¹ If $\gamma = 1$ the assignment is completely “hard” (as in traditional GP).

² Although “memory” is not strictly accurate when dealing with return values instead of memory cells, we choose to use this term to preserve the connection to our earlier work.

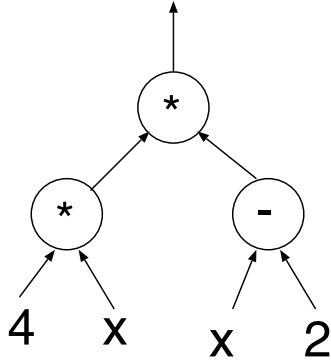


Fig. 1. Example program

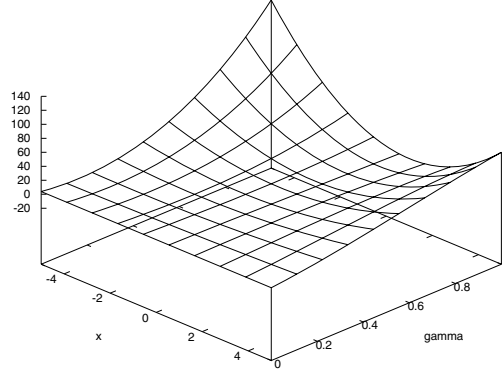


Fig. 2. Illustration of how the behaviour of the program in Fig. 1 varies from the constant function 4 to the parabola $4x^2 - 8x$ as γ varies from 0 to 1

An important question that needs answering to turn this idea into practice is: what do we mean by “blending return values with previously returned results”? Many approaches are possible. However, in the light of the positive experience of [7], before trying something sophisticated we thought we should try a simple approach: function nodes in a tree return a weighted average of their first argument with the result of the calculation corresponding to an instruction. In particular, if IN_1, IN_2, \dots represent the inputs to a particular instruction, F , its output, OUT , is given by

$$OUT = \gamma F(IN_1, IN_2, \dots) + (1 - \gamma)IN_1 \tag{2}$$

where γ is a constant that indicates the *hardness of the return operations*. In the extreme case where $\gamma = 1$ the return is completely “hard” and so $OUT = F(IN_1, IN_2, \dots)$ as in traditional GP. Values of $\gamma < 1$ provide a smoother behaviour. Note that there isn’t any specific reason for always choosing a weighted sum with the first argument of a function. The choice of argument could be randomised or there could be versions of these softer instructions for every possible argument. In the future we will investigate whether this has an effect on performance. However, in this first exploration we decided to go for the simplest possible strategy.

Fig. 1 shows an example program. We would normally interpret it as a representation of the expression $(4 \times x) \times (x - 2) = 4x^2 - 8x$. However, when MwM is used, the tree in Fig. 1 represents the function $\gamma(a \times b) + (1 - \gamma)a$ where $a = \gamma(4 \times x) + (1 - \gamma)4$ and $b = \gamma(x - 2) + (1 - \gamma)x$. Substitution of a and b into that expression produces $\gamma((\gamma(4 \times x) + (1 - \gamma)4) \times (\gamma(x - 2) + (1 - \gamma)x)) + (1 - \gamma)(\gamma(4 \times x) + (1 - \gamma)4)$ which simplifies to $8\gamma^3 + 4\gamma^2x^2 - 8x\gamma^3 + 8\gamma x - 8x\gamma^2 - 4\gamma^2 - 8\gamma + 4$. As γ varies from 1 to 0 the expression gradually morphs from the original $4x^2 - 8x$ to the value of the leftmost leaf of the tree, 4.³ If, for example, we compute this expression for $\gamma = 0.5$, we obtain $x^2 + x$, while

³ This behaviour is consistent with what happens in the original MwM system for linear GP, where for $\gamma = 0$ all programs become identity functions (the input register).

for $\gamma = 0.1$ we obtain $0.04x^2 + 0.712x + 3.168$. This morphing process is illustrated in Fig. 2.

3 GP System and Parameters

For our experiments we used a version of the TinyGP system in Java [8] which we modified so that it can handle MwM instructions and use validation sets.⁴ This is a tree-based system with a steady state control strategy, tournament selection and negative tournaments as a replacement strategy. More details and source code can be found in [8]. All that was required to adapt TinyGP to MwM was to change four lines of code of the interpreter (lines 9, 11, 13 and 19 below), which becomes:

```

1  double run() { /* Interpreter */
2      char primitive = program[PC++];
3      double input;
4      if ( primitive < FSET_START )
5          return(x[primitive]);
6      input = run();
7      switch ( primitive ) {
8          case ADD :
9              return( input * ( 1.0 - GAMMA ) + (input + run()) *
10                 GAMMA );
11             case SUB :
12                 return( input * ( 1.0 - GAMMA ) + (input - run()) *
13                    GAMMA );
14             case MUL :
15                 return( input * ( 1.0 - GAMMA ) + (input * run()) *
16                    GAMMA );
17             case DIV : {
18                 double den = run();
19                 if ( Math.abs( den ) <= 0.001 )
20                     return( input );
21                 else
22                     return( input * ( 1.0 - GAMMA ) + (input / den)
23                        * GAMMA );
24             }
25         }
26     }
27     return( 0.0 );
28 }

```

Tab. 1 shows the parameters and primitive sets we used for the experiments described in the following section.

⁴ The system can also work in a multi-deme configuration where runs execute on different machines in a cluster and pass their best individuals to a central store, which in return passes individuals back to the demes. However, we did not use this feature in the work reported here.

Table 1. Parameter settings used in our experiments

<i>Parameter</i>	<i>Value</i>
Function set	ADD, SUB, MUL, DIV (protected)
Terminal set	100 random constants uniformly distributed in the range $[-5, 5]$ plus 1 to 64 variables (depending on problem), except for the EEG prediction problem where 40 constants in the range $[-1, 1]$ were used
Independent runs	100, 500 or 2000 (depending on problem)
Max initial depth	5
Max size after crossover	10,000
Crossover point selection	uniform
Point mutation rate (per primitive)	0.05
Population size	10,000 or 100,000
Generations	100
Fitness evaluations per run	population size \times generations
Crossover rate (per individual)	0.1
Mutation rate (per individual)	0.9
Tournament size	2
Hardness of return operation (γ)	1.0, 0.7, 0.5, 0.3, 0.1
Fitness cases	11 to 5000 (depending on problem)
Fitness	sum of absolute errors
Stopping criterion	<code>fitness < 0.05 \times # fitness cases</code>

4 Problems and Results

To test the behaviour of our tree-based GP version of MwM, we used five classes of test problems: 1) symbolic regression with a sine target function, 2) one symbolic regression and two prediction problems involving the Mackey-Glass time series, 3) a prime prediction problem, 4) symbolic regression with two different polynomial targets, and 5) a real-world problem involving the reconstruction of EEG ear-lobe electrodes from other electrodes. The problems and the results we obtained on them using different values of the parameter γ are described in the following sections. As we will see, except in one case, the use of MwM helps GP to either improve its success rate or the accuracy of its solutions (or both).

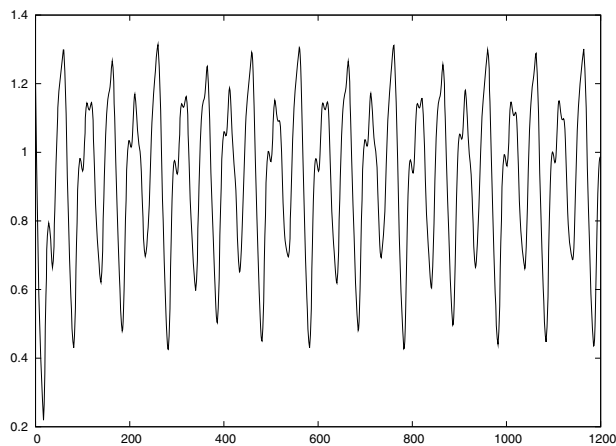
4.1 Sine Problem

The sine symbolic regression problem was used for illustration of the TinyGP system in [8]. The problem requires programs to fit the sine function over a full period of oscillation. We used 63 fitness cases obtained by sampling $\sin(x)$ for $x \in \{0.0, 0.1, 0.2, \dots, 6.2\}$. Based on the stopping criterion indicated in Tab. 1, we define a *success* to be a run where the best fitness was less than 3.15, or an average error of less than 0.05 over the 63 test cases.

For each configuration of γ we performed 500 independent runs with populations of size 10,000. Our results are reported in Tab. 2.

Table 2. Success rates and average end-of-run program size vs. hardness of return operation (γ) in the Sine symbolic regression problem

γ	Success rate	Average program size
1.0	0.296	54.88
0.7	0.442	56.24
0.5	0.618	67.59
0.3	0.540	88.16
0.1	0.066	140.54

**Fig. 3.** Mackey-Glass chaotic time series

The results show that most GP configurations that use MwM performed better than the standard GP case ($\gamma = 1$), with the best MwM configuration ($\gamma = 0.5$) effectively doubling the success rate of the system. It is also apparent from the results that as the value of γ decreases, the average size of the programs at the end of the run increases. As we will see, this is a common feature in our experiments with MwM (which we also observed in [7]). This behavior occurs, we theorize, because MwM causes every instruction to contribute to the output of a program, making it possible to continue to incrementally improve a program by appropriately extending it.

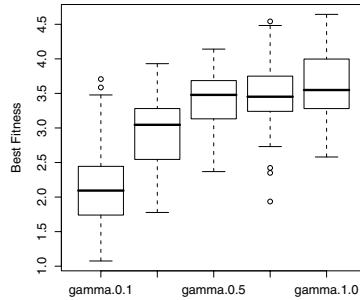
4.2 Mackey-Glass Problems

The Mackey-Glass chaotic time series is often used for testing prediction algorithms. We show the first 1,200 samples of this time series in Fig. 3.

In a prediction setting, algorithms are typically required to predict the next sample in the series given any number of previous samples. In principle, then, the estimated sample can be fed back into the algorithm to produce a further sample into the future and so on. However, the time series can also be used as a target for symbolic regression where the independent variable is time, and the dependent variable is the value taken by

Table 3. Success rates and average end-of-run program size vs. hardness of return operation (γ) (left) and box plot of best of run fitnesses (right) in the Mackey-Glass regression problem

γ	Success rate	Average program size
1.0	0.00	62.74
0.7	0.03	73.90
0.5	0.04	75.31
0.3	0.26	87.46
0.1	0.78	104.87



the series at that particular time. In this work we have used the Mackey-Glass chaotic time series for both types of applications.

We tested three configurations: two for prediction and one for regression. For each configuration and value of γ we performed 100 independent runs with populations of 100,000 individuals.

In the symbolic regression problem we gave GP the first 51 samples from the series and asked it to find a function which transformed the sample number (i.e., the numbers 0, 1, etc. up to 50) into the corresponding value in the Mackey-Glass chaotic time series. Tab. 3 shows the results. Again, the addition of MwM helps evolution considerably. In fact, $\gamma = 0.1$ makes the problem problem easy, while the results from $\gamma = 1.0$ might lead one to deem the problem impossible to solve. As before, we see the relationship between γ and the average end-of-run program size, with smaller γ 's leading to bigger programs.

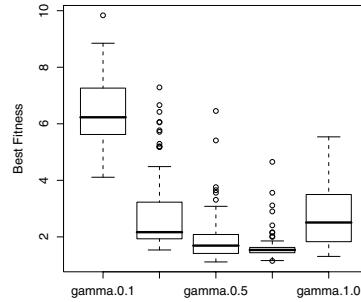
In the first prediction problem we constructed a training set including 1,192 fitness cases. Each fitness case had 8 independent variables representing the values taken by the time series in 8 consecutive samples. The corresponding target was simply the value of the following sample (which we wish to predict). For example, the first fitness case provides the values of the series at times 0 through to 7 to GP and asks GP to find a function that produces as output the value of sample 8. By sliding this 9-sample window over the time series, we obtained the rest of the training set. The results of our experiments are shown in Tab. 4.

All GP configurations were very successful at solving the problem. This is not surprising given the high correlation between samples present in the Mackey-Glass series and our stopping criterion which required absolute average errors per sample of less than 0.05. Looking at the mean best fitnesses, however, we find that, again, MwM helps evolution, as the settings $\gamma = 0.7$ and $\gamma = 0.5$ provide significant reductions in prediction error over the standard GP case. We also find that excessively soft return operations hinder performance. Size-wise, very small values of γ lead to bigger programs, but size seems to depend less on γ than for other problems (probably because runs were stopped early as a result of being successful).

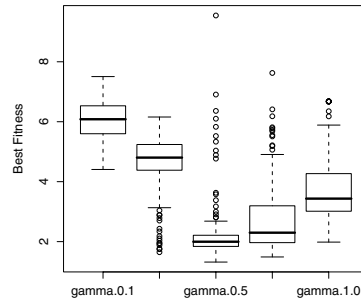
The second prediction problem was similarly constructed. The difference here was that we used 7 input taps (samples) and taps were not consecutive, but at distances of 1,

Table 4. GP performance for different values of hardness of return operation (γ) in Mackey-Glass prediction problem with consecutive taps

γ	Success rate	Mean best-end-of-run fitness	Average program size
1.0	1.0	2.69012	42.48
0.7	1.0	1.61918	35.89
0.5	1.0	1.89431	40.79
0.3	1.0	2.79318	42.26
0.1	0.96	6.44385	98.10

**Table 5.** GP performance for different values of hardness of return operation (γ) in Mackey-Glass prediction problem with non-consecutive taps

γ	Success rate	Mean best-end-of-run fitness	Average program size
1.0	1.0	3.79617	45.86
0.7	1.0	2.83241	42.25
0.5	1.0	2.45872	54.50
0.3	1.0	4.51185	75.97
0.1	1.0	6.0517	105.00



2, 4, 8, 16, 32 and 64 from the sample we wanted to predict. This produced a training set of 1,137 fitness cases. The results of our experiments are shown in Tab. 5.

Again we find that all systems are successful at finding solutions, but that $\gamma = 0.7$ and $\gamma = 0.5$ help produce better solutions, while excessively small γ 's hinder performance. Size-wise, very small values of γ lead to bigger programs, but other values affect size less markedly (again likely because all runs were successful and were stopped early).

4.3 Prime Prediction Problem

The prime prediction problem was originally suggested as a competition for the GECCO 2006 conference. The original version of the problem consisted in evolving a polynomial with integer coefficients such that given an integer value i as input produced the i -th prime number, $p(i)$, for the largest possible value of i . The evolved functions are therefore required to produce consecutive primes for consecutive values of the input i .

Despite its simple statement and the monotonicity of the function to be evolved, it turned out that the problem is extremely difficult and so solutions deviated significantly from the ideal (polynomial with integer coefficients). For example, the winner of the GECCO 2006 competition, David Joslin, proposed the polynomial $1.6272 + 0.7747 \times$

Table 6. Success rates and average end-of-run program size vs. hardness of return operation (γ) in prime prediction problem

γ	Success rate	Average program size
1.0	0.00	62.64
0.7	0.04	86.74
0.5	0.08	87.60
0.3	0.12	94.30
0.1	0.02	114.76

$x + 0.05215 \times x^3 - 2.7092 \times 10^{-6} \times x^8 + 1.5748 \times 10^{-14} \times x^{18} - 2.1966 \times 10^{-16} \times x^{20}$ that correctly predicts only the first 8 primes. Walker and Miller, the runner-ups, did much better but with a solution that wasn't even a polynomial.

In our version of the problem we treated the problem as a symbolic regression problem. We provided the first 11 primes as fitness cases. So, the problem requires GP to evolve a program that maps 1 into 2, 2 into 3, 3 into 5, 4 into 7, 5 into 11, and so on. To make the problem easier we did not require programs to exactly match the target output, but to exhibit a total sum of absolute errors of less than $0.05 \times 11 = 0.55$.

We performed 100 runs with populations of size 100,000 for each assignment of γ . Results are shown in Tab. 6. As was the case in the Mackey-Glass problem, MwM increases the success rate for the problem significantly turning a problem which we would have deemed impossible to solve for standard GP into a problem of moderate difficulty. We again see that solutions are bigger for smaller γ 's.

4.4 Polynomial Symbolic Regression

We applied the MwM GP system to polynomial symbolic regression problems using two target polynomials: $x^2 + 1.419x + 1.009$ which is of medium difficulty and $8x^5 + 3x^3 + x^2 + 6$ which is much harder.

For each target polynomial the fitness is the sum of the absolute error of the evolved function on 21 evenly spaced test points in the range $[-1, 1]$: $\{-1.0, -0.9, -0.8, \dots, 0.8, 0.9, 1.0\}$. The target then is to minimise this error.

We used population of 10,000 individuals and we performed 500 independent runs for each value of γ . We also did 100 runs (for each γ) with populations of 100,000 for the harder polynomial. The results are shown in Tab. 7.

In this instance, MwM results were mixed. While for the easier polynomial, all GP configurations which used MwM performed better than the standard GP case ($\gamma = 1$), in the harder polynomial MwM did not help.⁵

4.5 EEG Reconstruction Problem

Brain electrical activity is typically recorded from the scalp using Electroencephalography (EEG). This is used in electro-physiology, in psychology, as well as in

⁵ While it is somehow disappointing to find a problem where MwM does not help given the positive results obtained with it in all other problems, we should not be surprised to find such problems in the light of the no-free lunch theory.

Table 7. Success rates and average end-of-run program size vs. hardness of return operation (γ) in polynomial regression problems

Target $x^2 + 1.419x + 1.009$ (popsize=10,000)		Target $8x^5 + 3x^3 + x^2 + 6$ (popsize=10,000)		Target $8x^5 + 3x^3 + x^2 + 6$ (popsize=100,000)	
γ	Success rate	γ	Success rate	γ	Success rate
1.0	0.888	1.0	0.116	1.0	0.63
0.7	0.976	0.7	0.030	0.7	0.19
0.5	0.996	0.5	0.010	0.5	0.00
0.3	1.000	0.3	0.002	0.3	0.00
0.1	0.982	0.1	0.000	0.1	0.01
γ	Avg Prog Size	γ	Avg Prog Size	γ	Avg Prog Size
1.0	39.81	1.0	47.88	1.0	50.90
0.7	53.70	0.7	54.78	0.7	55.56
0.5	73.90	0.5	63.86	0.5	66.17
0.3	96.16	0.3	74.41	0.3	72.47
0.1	153.62	0.1	110.37	0.1	100.57

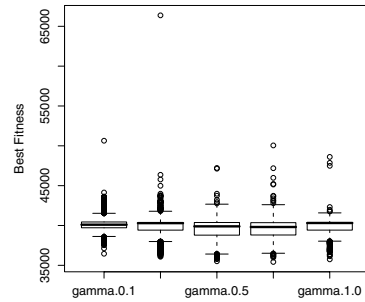
brain-computer interfaces research. Voltages at the EEG electrodes are always recorded relative to some reference. Traditionally the reference has been either one ear electrode or the average of the two ears [6]. This is because there is neither muscular activity (which produces very large potentials) nor, obviously, neural activity in the ears. Note that in some EEG equipment, the ear electrodes are actually the reference voltage against which the voltage of other electrodes is measured. In others they are not, but it is still common to refer signals back to the ears to be consistent with the large body of literature on EEG analysis.

As a result of using the ears as references, any changes occurring in the impedance of the ear electrodes can produce huge artifacts in every signal recorded. If, for example, a single electrode detaches slightly, then every other channel can be flooded with noise, leading to huge baseline shifts. It is, therefore, very important to come up with ways of verifying if the ear electrodes are working properly, for example by comparing them against a prediction of what their voltage should be. There are also systems which do not require the use of ear electrodes. In these systems it is then impossible to refer the recorded signals (for example for comparison with other research) back to the ears. In such systems, if one could predict what the ear electrodes would measure, based on the signals recorded from other electrodes, then one could refer signals back to the reconstructed ear electrodes.

In this last set of experiments we compared GP systems with different degrees of MwM to see how well they can construct a soft-sensor for the left ear from real EEG recordings. We constructed a dataset using 64-channel EEG recordings from 5 different subjects acquired over a period of around a month. From each subject's recording we extracted two fragments of approximately 20 seconds each. The fragments were approximately half an hour apart. The original data was sampled at 2KHz. After band-pass filtering we subsampled it at 128 samples per second. We then chose 250 random time steps within each fragment. At each time step we saved the voltages recorded at the 64 channels plus the left ear reference voltage as a fitness case. This gave us a training

Table 8. GP performance for different values of hardness of return operation (γ) in EEG ear-electrode reconstruction problem

γ	Mean Generalisation Fitness	Fitness Std Dev	Std Error of the Mean	Avg Prog Size
1.0	39886.0	885.42	19.80	5.16
0.7	39637.5	1039.81	23.25	5.74
0.5	39485.6	1252.81	28.01	6.41
0.3	39831.0	1275.56	28.52	7.47
0.1	40342.9	1101.98	24.64	9.25



set of 5,000 fitness cases. Using different fragments from the same 5 subjects we also constructed a validation set of 5,000 fitness cases. This set was not used to compute fitness but only to decide when runs should be stopped.

We performed 2,000 runs for each configuration of γ with a population of size 10,000. GP had 64 input variables (the voltage values of the 64 channels) and one output (the left ear voltage). Tab. 8 reports the generalisation results obtained in different configurations of MwM. Again, MwM significantly improves performance (analysis of variance shows that performance differences are statistically highly significant), slightly affecting program size.

5 Conclusions

In this paper we have extended the idea of memory-with-memory, originally applied to linear, register-based GP, to the domain of tree-based GP with instructions without side effects and memory. This setup is very common, and is used, for example, in virtually all symbolic regression applications of tree-based GP. We achieve this by using a “soft” return operation to pass the values computed by instructions up the tree instead of the standard, “hard”, return operation which is used in most computer science (including traditional GP). Instead of having the new value completely determine the output of a node, the computed value is first combined (using a weighted average) with the value of the first argument to a function and then returned.

In tests with a variety of symbolic regression and prediction problems, tree-based GP with MwM almost always does as well as traditional GP, while significantly outperforming it in several cases. Particularly striking are the very small values of γ (corresponding to a *very* soft form of value return). In several of the cases the greatest success was obtained with γ values of 0.3, and in one case (the Mackey-Glass regression) $\gamma = 0.1$ provided the best performance. These are extremely low values and represent a radically different notion of value return than standard GP. This suggests that these kinds of semantics play a crucial role, and that modifying them can have a powerful impact on system performance.

The data suggest that MwM GP can continue to improve (if slowly) solutions over time where traditional GP may in fact get stuck in local optima. As a result, solutions

evolved using MwM tend to be slightly bigger than without it. Unlike with other systems, this cannot be attributed to introns, since with MwM every instruction contributes to a program's output. In future research we will try to understand what makes a problem hard for systems with MwM.

Acknowledgments

We would like to thank EPSRC (grant EP/G000484/1) for financial support.

References

1. Angeline, P.J.: An alternative to indexed memory for evolving programs with explicit state representations. In: Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L. (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA, July 13-16, pp. 423–430. Morgan Kaufmann, San Francisco (1997)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco (1998)
3. Brave, S.: Evolving recursive programs for tree search. In: Angeline, P.J., Kinnear Jr., K.E. (eds.) *Advances in Genetic Programming 2*, ch. 10, pp. 203–220. MIT Press, Cambridge (1996)
4. Bruce, W.S.: Automatic generation of object-oriented programs using genetic programming. In: Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, July 28–31, pp. 267–272. MIT Press, Cambridge (1996)
5. Langdon, W.B.: *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Genetic Programming, vol. 1. Kluwer, Boston (1998)
6. Luck, S.J.: *An Introduction to the Event-Related Potential Technique*. MIT Press, Cambridge (2005)
7. McPhee, N.F., Poli, R.: Memory with memory: Soft assignment in genetic programming. In: Keijzer, M., Antoniol, G., Congdon, C.B., Deb, K., Doerr, B., Hansen, N., Holmes, J.H., Hornby, G.S., Howard, D., Kennedy, J., Kumar, S., Lobo, F.G., Miller, J.F., Moore, J., Neumann, F., Pelikan, M., Pollack, J., Sastry, K., Stanley, K., Stoica, A., Talbi, E.-G., Wegener, I. (eds.) *GECCO 2008: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, GA, USA, pp. 1235–1242. ACM, New York (2008)
8. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via (2008) (With contributions by Koza, J.R.), <http://lulu.com>, <http://www.gp-field-guide.org.uk>
9. Spector, L., Klein, J., Keijzer, M.: The push3 execution stack and the evolution of control. In: Beyer, H.-G., O'Reilly, U.-M., Arnold, D.V., Banzhaf, W., Blum, C., Bonabeau, E.W., Cantu-Paz, E., Dasgupta, D., Deb, K., Foster, J.A., de Jong, E.D., Lipson, H., Llorca, X., Mancoridis, S., Pelikan, M., Raidl, G.R., Soule, T., Tyrrell, A.M., Watson, J.-P., Zitzler, E. (eds.) *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Washington DC, USA, vol. 2, pp. 1689–1696. ACM Press, New York (2005)
10. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines* 3(1), 7–40 (2002)
11. Teller, A.: The evolution of mental models. In: Kinnear Jr., K.E. (ed.) *Advances in Genetic Programming*, ch. 9, pp. 199–219. MIT Press, Cambridge (1994)