Hindawi Publishing Corporation
Journal of Artificial Evolution and Applications
Volume 2009, Article ID 570606, 16 pages
doi:10.1155/2009/570606

Research Article

Memory with Memory in Genetic Programming

Riccardo Poli,[1] Nicholas Freitag McPhee,[2] Luca Citi,[1] and Ellery Crane[2]

[1] *School of Computer Science and Electronic Engineering, University of Essex, Colchester, CO4 3SQ, UK*
[2] *Division of Science and Mathematics, University of Minnesota, Morris, MN 56267, USA*

Correspondence should be addressed to Riccardo Poli, rpoli@essex.ac.uk

Received 24 April 2009; Accepted 20 May 2009

Recommended by Leonardo Vanneschi

We introduce Memory with Memory Genetic Programming (MwM-GP), where we use soft assignments and soft return operations. Instead of having the new value completely overwrite the old value of registers or memory, soft assignments combine such values. Similarly, in soft return operations the value of a function node is a blend between the result of a calculation and previously returned results. In extensive empirical tests, MwM-GP almost always does as well as traditional GP, while significantly outperforming it in several cases. MwM-GP also tends to be far more consistent than traditional GP. The data suggest that MwM-GP works by successively refining an approximate solution to the target problem and that it is much less likely to have truly ineffective code. MwM-GP can continue to improve over time, but it is less likely to get the sort of exact solution that one might find with traditional GP.

1. Introduction

In the vast majority of programming models, dating back to the Turing machine [1] and the earliest electronic computer architectures (e.g., [2]), assignments are entirely destructive in the sense that an instruction of the form `x:=y` or `LOAD R2 R1` completely overwrites the previous value of a memory location or register. That earlier value is lost forever, and has no impact on the future behavior of the system (unless it was copied elsewhere before it was overwritten). This overwriting model of assignment was carried over to most versions of genetic programming (GP) that had state and assignments (see [3] for a review). This includes linear GP [4], which evolves sequences of (virtual or real) machine code instructions that usually act by destructively writing to registers or memory locations.

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 in the cell, for example, can happen quickly, but are still typically incremental in nature, with each new state being constructed via modification of the previous state rather than a complete replacement of it. Even dramatic state changes, such as the transformation of a caterpillar into a butterfly, take time and involve large numbers of small, local changes.

This difference with biology might be sufficient reason on its own to explore other models of assignment in GP. There are, however, practical concerns that also suggest that there might be value in alternative approaches. Linear register-based GP systems with hard assignments, for example, can be quite fragile with respect to certain changes. A program that works by incrementally building up a result in a register can have its behavior radically altered by something like a point mutation that writes a 0 to the accumulating register late in the process. These hard assignments can also act as powerful intron creation mechanisms, as overwriting registers can render whole sequences of preceding instructions irrelevant to the final output of the evolved program.

Here we propose an alternative *Memory with Memory* (MwM) model, with "soft" assignments that merge new values with previous values instead of overwriting them. We show that including this new type of assignment in a register-based GP system can significantly improve performance on a variety of symbolic regression problems. We also find that Memory with Memory changes the nature of bloat, reducing the amount of ineffective code while at the same time tending to increase the mean program sizes as GP continues to refine the evolved approximations.

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 not used frequently in tree-based GP, where most applications evolve functions with no side effects rather than assignment-based programs. For these, the soft-assignment-based form of MwM cannot be used. There is, however, a hard operation in tree-based GP which is ubiquitously used: the return operation which is used by the code implementing function nodes to communicate results back to their caller. This operation is hard in the sense that it assigns the value computed by a function node to one of the arguments of another function node (or to the fitness evaluator) completely disregarding the value that would have been passed to the caller had the instruction not been executed. As we will illustrate, softening return operations, by making sure the value of a function node is a blend of the result of a calculation and previously returned results effectively achieves in tree-based GP what soft assignments achieve in linear register-based GP: in both cases, MwM provides significant performance improvements. Additionally, both forms of Memory with Memory require only very minor modifications to existing systems, making them easy to add.

The paper is organised as follows. We begin by reviewing related work in the next section. We then define our approach to Memory with Memory both for linear register-based GP systems and for tree-based representations in Section 3. We describe the GP systems we used to test the approach in Section 4. Our empirical results are presented in Section 5. We summarise our findings in Section 6 and we provide some indications for interesting future work in Section 7.

2. Related Ideas

While many GP systems are expression based, numerous GP systems have used some kind of memory or state. An obvious instance is linear GP [4], but other examples include indexed memory [5–7], and work on evolving data structures (such as stacks) that have internal state [8, 9]. Similarly, systems such as PushGP [10, 11] that use (rather than evolve) data structures such as stacks in their computational model are manipulating internal state in an important way. We are unaware, however, of any memory or (internal) state-based GP system that uses a soft assignment or a soft return operation of the type proposed here.

Probably more similar to this work are systems where evolved agents rely primarily on external state (known as stigmergy). Here the state of an individual is rarely if ever completely replaced with a new state with no regard to the previous one. In swarm intelligence [12] and ant colony optimization [13], for example, changes in state such as pheromone levels are typically incremental, basing the new levels on the old. Similarly, the position of agents in particle swarm optimization systems [14] is almost always adjustments of the previous position rather than arbitrary jumps. This connection would also hold for many traditional GP systems used to evolve agents whose behavior is driven primarily by external state. In the classic artificial ant problem using the function set given in [15], for example, the new state of the ant following the trail is always a minor alteration of the previous state as the ant either turns 90° or moves forward one square. Similarly, many systems that evolve strategies for games like RoboCup Soccer [16] rely largely or entirely on external state which updates in an incremental fashion. Reference [17] extends the idea of indexed memory by allowing multiple individuals in the population to read from and write to a shared memory space, which allows them to have both internal and external (shared) states; the write instructions in this system were traditional hard assignments, however, so the state was not constrained to incremental changes.

3. Memory with Memory

3.1. Memory with Memory via Soft Assignment. There are numerous approaches that could be taken to combining the old values with the new when performing assignments. In this research we take a simple but flexible approach: weighted averaging of the old value of a register with the new value being assigned. 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

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

where γ is a constant that indicates the *assignment hardness*, allowing us to determine how "hard" or "soft" the assignment operator is. If $\gamma = 1$, for example, we get a completely "hard" assignment, and have traditional register-based GP. For $\gamma = 0.5$, on the other hand, we have a simple average of the new and old values. For $\gamma = 0$ all registers are effectively write protected, making all instructions behave like no-ops; in this situation all programs compute the identity function.

While having the advantages of simplicity and flexibility, this does introduce yet another parameter into the GP system. We make no attempt at comprehensively optimizing that new parameter here, but do perform extensive tests using a set of four values of γ: 1.0, 0.7, 0.5, and 0.3.

3.2. Memory with Memory via Soft Return Operations. In register-based GP systems executing instructions effectively involves two steps: (a) a calculation and (b) an assignment of the result of the calculation into a register or memory location. The MwM technique for register-based GP acts on this second component, ensuring that previously computed results cannot entirely be wiped out 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. This is very different from what happens in a register-based system. Executing instructions in tree-based GP, however, 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. To introduce a form of MwM we again act on the second stage of execution, that is, on the process which determines how 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 also call the use of soft return operations *MwM*.

To make this practical, an important question is what do we mean by "blending return values with previously returned results"? As was the case for soft assignment, many approaches are possible. We again chose 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 $\texttt{IN}_1$, $\texttt{IN}_2$, and so forth represent the inputs to a particular instruction, F, its output, OUT, is given by

$$\text{OUT} = \gamma \text{F}(\text{IN}_1, \text{IN}_2, \ldots) + (1-\gamma)\text{IN}_1 \tag{2}$$

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 $\texttt{OUT} = F(\texttt{IN}_1, \texttt{IN}_2, \ldots)$ as in traditional GP. Values of $\gamma < 1$ provide a smoother behaviour. Note that there is not 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 study we chose the simplest possible strategy.

Figure 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 Figure 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 . (This behaviour is consistent with what happens in the original MwM system for linear GP, where for $\gamma = 0$ all programs become identity functions.) If, for example, we compute this expression for $\gamma = 0.5$, we obtain $x^2 + x$, while for $\gamma = 0.1$ we obtain $0.04x^2 + 0.712x + 3.168$. This morphing process is illustrated in Figure 2.

4. GP Systems and Parameters

To test our ideas we used a linear register-based GP system and a tree-based GP system. These are described in Sections 4.1 and 4.2, respectively.

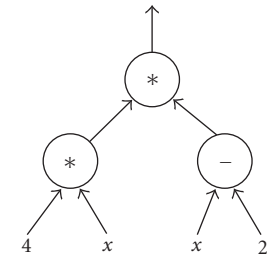


Figure 1: Example program.

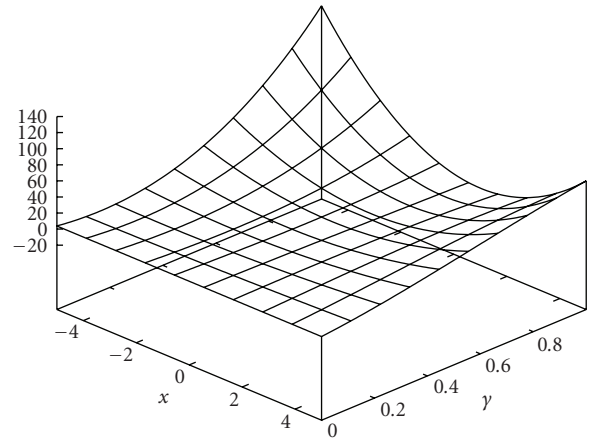


Figure 2: Illustration of how the behaviour of the program in Figure 1 varies from the constant function 4 to the parabola $4x^2 - 8x$ as γ varies from 0 to 1.

4.1. Linear Register-Based GP. We used a simple, register based linear GP system similar to that described in [4]. Table 1 lists the parameter settings used for our experiments; these values have proven useful in prior work, and no effort was made to optimize them for any of the systems used here.

The evolved programs are variable length linear sequences of "machine instructions" acting on a set of up to six registers (R1-R6); the full instruction set is given in Table 3. Note that when using traditional, hard assignment, R3-R6 can only be used to store temporary results; all arithmetic is performed using just R1 and R2. When using soft assignment, however, instructions like R4:=R1 implicitly perform whatever calculations are used to implement soft assignments.

While there has been some study of function selection in GP (e.g., [18, 19]), this remains more art than science. No particular attempt was made to fine tune our instruction set for the target functions used in this study. To see how soft assignment would perform with different instructions, however, we did use four progressively larger instruction sets F1 (group A from Table 3), F2 (groups A and B), F3 (groups A, B, C), and F4 (groups A, B, C, D). F1 is a basic two register system with just addition, multiplication, and swap. F2 adds the constants 0 and 1, subtraction and division, instructions to copy data to/from R1 and R2, and swaps with R3. F3 adds hard assignment from the input variable x and the constants 0 and 1. F4 adds hard and soft assignment of $-x$ and -1 to R1 and R2.

Table 1: Parameter settings used in the experiments with soft assignment in register-based GP. These are fairly generic parameter choices and were not optimized for any of the systems used in these experiments.

Parameter	Value
Primitive set	see Table 3
Independent runs	50
Max initial length	50
Max length after XO	500
Point mutation rate (per primitive)	$1/\ell$
Population size	1000
Generations	40
Fitness evaluations per run (pop. size $\times$ gens)	40,000
Crossover rate (per individual)	0.9
Mutation rate (per individual)	0.1
Tournament size	2
Assignment hardness (γ)	1.0, 0.7, 0.5, 0.3
Fitness cases	21
Fitness	sum of absolute errors
Stopping criterion	fitness $< 0.05 \times$ # fitness cases

Our linear GP system uses a steady-state control strategy with binary tournament selection for choosing parents and negative binary tournament selection for replacement. The initial random individuals had lengths chosen uniformly from the range $[1, 50]$.

New individuals are constructed using mutation 10% of the time, and subtree crossover 90% of the time; we use no reproduction. If mutation is chosen, a parent individual is selected via tournament selection, and an offspring is generated using either point mutation or subtree mutation with equal probability. With point mutation each instruction has a $1/\ell$ chance (where ℓ is the length of the program) of being replaced by a randomly selected instruction. With subtree mutation, a random point is chosen and all the instructions after that point are replaced by a new randomly generated sequence of instructions of length between 1 and 500 (the maximum allowed length after crossover).

If crossover is chosen, we select two parents (again, via tournament selection) and apply homologous two-point crossover with 50% probability, and subtree crossover with 50% probability. Subtree (or variable length) crossover involves the selection of one crossover point in each parent, and swapping the instructions following the crossover points. Since the crossover points are chosen independently, the length of the swapped suffixes can be different, leading to offspring of varying length. Homologous crossover [3, Section 7.1.3], requires choosing two crossover points which are used to divide both parents into three sections. The offspring is then formed by swapping the "middle" sections of the two parents, which means that the offspring is guaranteed to have the same length as the parent donating the prefix and suffix portions.

4.2. Tree-Based GP. We used a version of the TinyGP system in Java [3] which we modified so that it can handle MwM instructions and use validation sets. (The system can also work in a multideme 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.) 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 [3]. Algorithm 1 shows the minimal changes necessary to adapt TinyGP to MwM.

Table 2 shows the parameters and primitive sets we used for the experiments described in the following section.

5. Test Problems and Results

We performed two large sets of experiments: one using soft assignment within the register-based GP system described above and one using soft return operations in TinyGP. The problems we used for each system and the results we obtained are described Sections 5.1 and 5.2.

5.1. Soft Assignment in Register-Based GP

5.1.1. Problems. We applied the register-based GP system with and without MwM to several classes of polynomial symbolic regression problems, using target polynomials of several different degrees for each class. In total we tested each value of γ and each instruction set on a total of 16 different symbolic regression problems of three different types, with degrees ranging from 4 to 15 for each of these three classes.

The simplest class of problems we explored was "regular" polynomials of the form

$$\sum_{1 \le n \le d} x^n = x + x^2 + x^3 + \cdots + x^d. \tag{3}$$

We used six target polynomials of this form, having degrees $d = 5, 7, 9, 11, 13$, and 15; two examples of regular polynomials having degrees 7 and 13, are listed below:

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 \tag{4}$$

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{12} + x^{13} \tag{5}$$

Low-degree polynomials of this type have often been used in TP studies [3, 15].

A slightly more complex class of problems had coefficients that were randomly generated from the set $\{0, 1\}$. (All random coefficients used in our test polynomials were generated via http://www.random.org/.) Thus they are essentially the same as the "regular" polynomials, but with random terms removed. We used five such polynomials, with degrees 4, 7, 9, 12, and 15; the specific polynomials we used

Table 2: Parameter settings used in our experiments with soft return operations in tree-based GP.

Parameter	Value
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 × 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	fitness < 0.05 × # fitness cases

are as follows:

$$x + x^2 + x^3 + x^4 \tag{6}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 \tag{7}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 \tag{8}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 + x^{12} \tag{9}$$

$$x + x^2 + x^3 + x^4 + x^6 + x^7 + x^8 + x^9 + x^{12} + x^{15} \tag{10}$$

Finally, we created a third class of polynomials by adding -1 in the set of possible coefficients; then by randomly choosing coefficients from the set $\{-1, 0, 1\}$, we obtained the following polynomials:

$$1 - x + x^2 + x^3 - x^5 \tag{11}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 \tag{12}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} \tag{13}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} - x^{12} \tag{14}$$

$$1 - x + x^2 + x^3 - x^5 + x^8 - x^9 + x^{10} + x^{11} - x^{12} + x^{15} \tag{15}$$

Note that in each class the polynomials of higher degrees are "extensions" of the lower degree polynomials in the sense that the higher degree polynomials are equal to the lower degree polynomials plus some new higher order terms. For example, the degree 7 polynomial (7) from (6)–(10). is the degree 4 polynomial (6) with two additional terms ($x^6 + x^7$).

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 $[-1, 1]$: $\{-1.0, -0.9, -0.8, \ldots, 0.8, 0.9, 1.0\}$. The target then is to minimize this error. We define a *success* to be a run where the best fitness was less than 1.05, or an average error of less than 0.05 over the 21 test cases.

5.1.2. Results. To better understand what impact Memory with Memory and the particular value of γ has on symbolic regression problems, we performed 50 independent runs for each combination of the following parameters:

(i) 4 values of γ (1.0, 0.7, 0.5, 0.3)

(ii) 4 instruction sets (F1-F4 from Table 3)

(iii) 16 different polynomials (Section 5.1.1)

leading to a total of 12,800 runs and 512,000,000 fitness evaluations.

Impact of Soft Assignment on Fitness. Figure 3 shows the distribution of the best fitness values from each of those 12,800 runs for each of the four values of the assignment hardness γ. $\gamma = 0.7$ has the best results of the four, with $\gamma = 0.5$ second, and $\gamma = 1$ (traditional GP) and $\gamma = 0.3$ being statistically indistinguishable (Throughout this paper all tests for statistical significance are at 95% confidence levels.). The plot also indicates that the variance for the assignment hardnesses less than 1 are significantly smaller than for traditional GP, meaning that using soft assignment gives us more consistent results.

Table 3: Linear GP instructions used in these experiments. All assignments are soft except those suffixed with "(H)". % is protected division, which returns its first argument if the second is less than 10^{-6}. The instructions are divided into four groups of related instructions (*A*, *B*, *C*, and *D*, indicated by horizontal lines). We used four progressively larger sets of instructions: F1 (group *A*), F2 (groups *A* and *B*), F3 (groups *A*, *B*, *C*), and F4 (groups *A*, *B*, *C*, *D*).

Group	Description	Instructions
A	Read input	`R1:=X, R2:=X`
	Plus, times	`R1:=R1+R2, R2:=R1+R2, R1:=R1*R2, R2R1*R2`
	Accum. swap	`Swap R1 R2`
B	Constants	`R1:=0, R2:=0, R1:=1, R2:=1`
	Minus, divides	`R1:=R1-R2, R2:=R1-R2, R1:=R1%R2, R2:=R1%R2`
	Copy to R1	`R1:=R2, R1:=R3, R1:=R4, R1:=R5, R1:=R6`
	Copy to R2	`R2:=R1, R2:=R3, R2:=R4, R2:=R5, R2:=R6`
	Copy from R1	`R2:=R1, R3:=R1, R4:=R1, R5:=R1, R6:=R1`
	Copy from R2	`R1:=R2, R3:=R2, R4:=R2, R5:=R2, R6:=R2`
	Swap R3	`Swap R1 R3, Swap R2 R3`
C	Hard input read	`R1:=X(H), R2:=X(H)`
	Hard constants	`R1:=0(H), R2:=0(H), R1:=1(H), R2:=1(H)`
D	Negative input read	`R1:=-X, R2:=-X, R1:=-X(H), R2:=-X(H)`
	Negative one	`R1:=-1, R2:=-1, R1:=-1(H), R2:=-1(H)`

```
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( ) * GAMMA);
10   case SUB:
11     return (input * ( 1.0 - GAMMA ) + (input - run( ) * GAMMA);
12   case MUL:
13     return (input * (1.0 - GAMMA) + (input * run( ) * GAMMA);
14   case DIV:
15     double den = run ( );
16     if   (Math.abs(den) <= 0.001)
17       return (input);
18     else
19       return(input * (1.0 - GAMMA) + (input / den) * GAMMA);
20     }
21    }
22   return (0.0);
23 }
```

Algorithm 1: The minimal changes to the TinyGp system needed to implement MwM. Only four lines of code in the interpreter needed to be altered (lines 9, 11, 13 and 19 below).

This is supported by the estimated success rates listed in Table 4. Both $\gamma = 0.7$ and $\gamma = 0.5$ have significantly better success rates than traditional GP ($\gamma = 1.0$). $\gamma = 0.7$, for example, has an estimated success rate of over 10% across all the combinations we ran, while the estimated success rate of traditional GP was just over 8%.

Figure 4 shows that the advantage of soft assignment is quite consistent across a wide range of degrees. The median for $\gamma = 0.7$ and $\gamma = 0.5$ are better than for $\gamma = 1$ in all cases except the degree 4 polynomial ((6) and the combination of the two degree 7 polynomials (4) and (7) where they effectively tie. $\gamma = 0.7$ also does better than $\gamma = 0.5$ for several degrees, and never does worse. Note that the best (minimum) for traditional GP is consistently nearly perfect (very nearly 0), while the soft assignment runs were generally unable to reach a perfect solutions. This is not surprising given that soft assignment tends to encourage refinement over time, which is likely to lead to

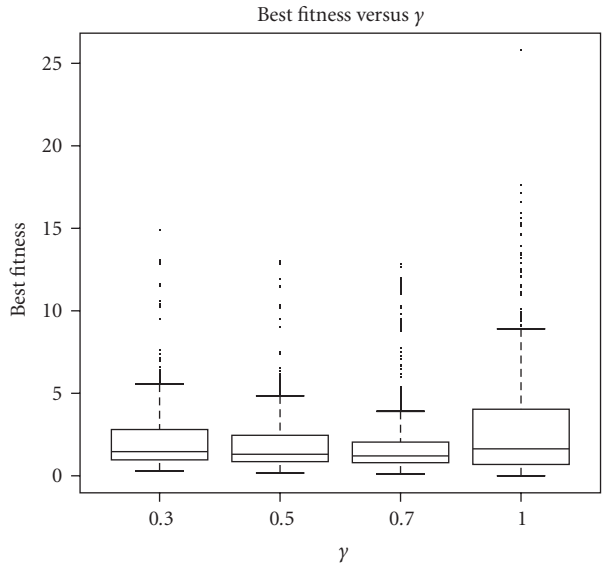


Figure 3: Boxplot showing the distribution of best fitnesses across all runs (all polynomial classes and degrees, all instruction sets). The differences in best fitnesses are statistically significant (using a pairwise Wilcoxon test) for all pairs except $\gamma = 0.3$ versus $\gamma = 1$ (traditional GP). The differences in variances are also statistically significant (using the Fligner-Killeen test of homogeneity of variances).

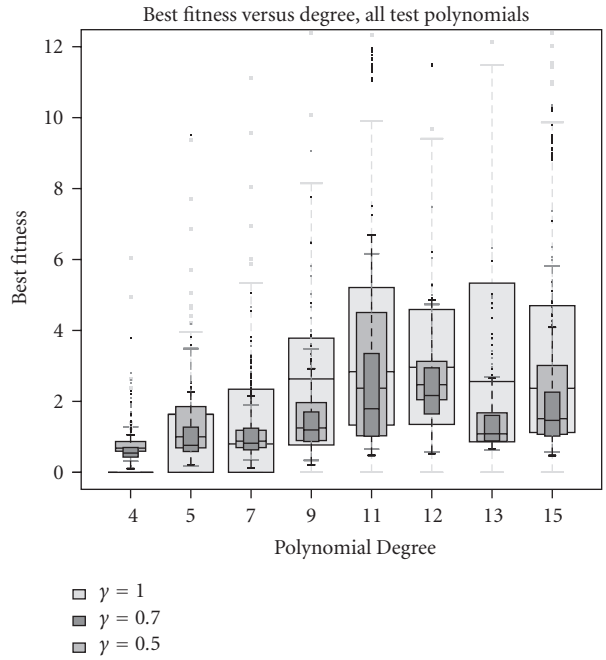


Figure 4: Boxplots of the best end-of-run fitness by degree for $\gamma \in \{1.0, 0.7, 0.5\}$. There are several outlier fitness values between 12 and 26 that are not visible in this plot.

Table 4: Estimates of the success rates for each of the four assignment hardnesses using Wilson's method for computing (95%) confidence intervals for binomial probabilities. We define a success to be a run where the best fitness was less than 1.05, or an average error of less than 0.05 for each of the 21 test cases.

γ	Estimated success rate	Lower	Upper
1.0	8.148%	7.686%	8.635%
0.7	10.578%	10.057%	11.122%
0.5	9.429%	8.935%	9.948%
0.3	7.679%	7.231%	8.153%

approximate solutions. Those approximations are generally very close in our runs, but linear GP with soft assignment does not seem to have the "killer instinct" needed to finally reach the target, at least with the parameters used here. (Given that we only used 40,000 fitness evaluations per run, it is entirely possible that more generations would allow our system to further refine the solutions, but we have not explored the impact of either increasing the population size or the number of generations.) This tendency to approximate may account for the advantage of hard assignment on the degree 4 polynomial; that polynomial is sufficiently easy that traditional GP solves it exactly with a high probability, while the Memory with Memory approximations are slightly worse. The variance trend noted earlier continues here, with the variance for traditional GP being similar in a few cases, but dramatically greater in others (e.g., degree 13 cases).

Figure 5 shows the proportion of successful runs by degree. Here again $\gamma = 0.7$ consistently does as well or better than all the other settings, and $\gamma = 0.3$ and traditional GP typically do the worst.

With one exception, all the instruction sets F1–F4 had at least moderate levels of success on the different polynomial classes. As can be seen in Figure 6, the instruction set F1 was generally incapable of solving any of the polynomials in the -1, 0, 1-polynomial class.

Figure 7 plots the distribution of best fitnesses across the 40 generations for $\gamma = 0.7$ and traditional GP ($\gamma = 1$). To help clarify the pattern, we plotted the application of a single instruction set (F2) to a single polynomial ((5), the regular polynomial of degree 13). We again see that the variance is much smaller with soft assignment. We also see that the fitness improves much more quickly in the early generations than traditional GP.

In Figure 8 we have the proportion of these F2-Degree 13 runs that showed any improvement from one generation to the next. $\gamma = 0.7$ shows a consistently higher proportion of improvements across all the generations than traditional GP ($\gamma = 1$). Both the hard and soft assignments showed a steady drop in the proportion of runs showing improvement up to around generation 20, where in both cases the slope flattens off somewhat. For $\gamma = 0.7$, however, the proportion of runs stays almost constant in the later generations, while it continues to drop noticeably for traditional GP. At generation 40 the proportion of runs still showing improvement in any given generation is over 30% for the soft assignment runs, while itis only around 10% for the traditional GP runs. It is important to realize, however, that

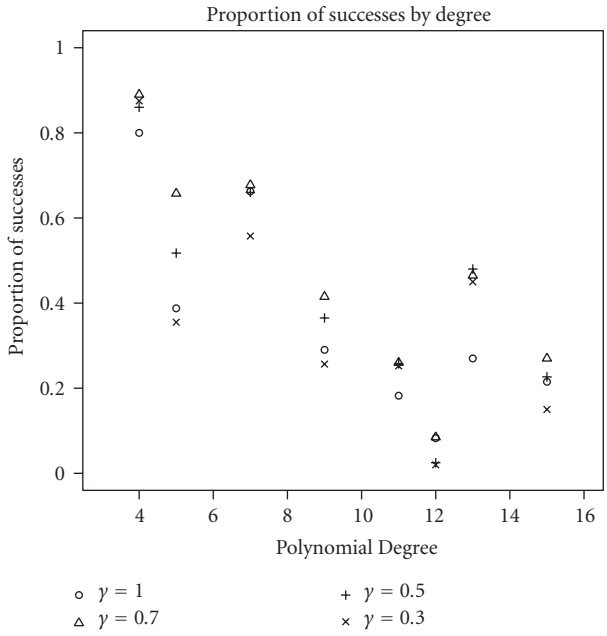


Figure 5: Proportion of successes by degree for each of the four values of γ. The proportion of successes for a configuration is the proportion of runs with that configuration that had a total error of less than 1.05.

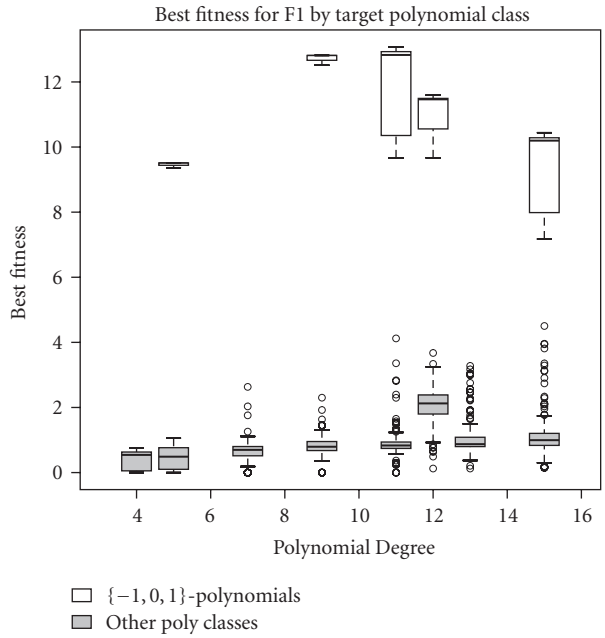


Figure 6: Distribution of end-of-run best fitnesses by class of target polynomial, restricted to runs using the instruction set F1. (All function sets had reasonable levels of success on all test classes except F1 on the $\{-1, 0, 1\}$-polynomials.) The white boxplots show the distribution for the $\{-1, 0, 1\}$-polynomials, while the gray boxplots show the aggregate results for all the polynomials in the other two classes.

most of these improvements in the later generations are quite small, as is suggested by Figure 7. Thus while we would expect continued improvements in best fitness if we let the soft assignment runs continue for additional generations, many of those improvements would be extremely small improvements in the evolved approximations.

Impact of Soft Assignment on Length. Hard assignment is one potential source of ineffective or "intron" code in GP; that is, sequences of instructions which do work, but ultimately have no impact on the program's final output. An instruction like R10, for example, effectively undoes any preceding operations that collected results in R1, potentially making long sequences of previous instructions ineffective. This suggests that hard assignment could play a role in bloat [3, Section 11.3], by providing a mechanism for program lengths to increase without improving (or even changing) their functionality. With soft assignment, on the other hand, every instruction has some impact on the future state of the system. While an early instruction might have a very limited impact in a long program, it will have some impact, implying that there is ultimately no truly ineffective code when using soft assignment. The ability for GP to use soft assignment to incrementally approximate solutions, however, suggests that programs could grow longer and longer over time as GP works to improve its approximations.

Figure 9 shows the distribution of both mean and best-of-run program lengths for all four values of γ across all the runs. Both the mean and best-of-run program lengths were substantially larger for $\gamma = 0.7$ and $\gamma = 0.5$ than for traditional GP. The maximum size allowed after crossover (500) would push down the the amount the ineffective code one would expect in traditional GP. In the case of soft assignment, however, where all instructions have an impact and where there are steady, if small, improvements in fitness over time (as seen in Figures 7 and 8), there was presumably more fitness correlated pressure to grow.

Figure 10 shows that the tendency for soft assignment to lead to larger programs holds for all four function sets, and is much more pronounced for F3 and F4 than F1 and F2. The best-of-run sizes with F1 were substantially smaller than with the other three instruction sets for both traditional GP and $\gamma = 0.7$. The best-of-run sizes in fact showed statistically significant differences across all four function sets when using traditional GP. The best-of-run sizes were much more similar, however, when using $\gamma = 0.7$, especially for F2 and F3; in fact, the differences were not statistically significant for the pairs (F2, F3) and (F2, F4).

Table 5 shows the differences in the proportions of the different instructions (from set F2) as they appeared in the best-of-run individuals for the 50 runs with F2 on the regular degree 13 polynomial (5) between $\gamma = 0.7$ and traditional GP ($\gamma = 1$). Negative values at the top of the table (e.g., `R1:=X`, `R2:=X`, and `R1:=R1+R2`) all appear more frequently in the best-of-run individuals in the soft assignment runs. This indicates that runs with soft assignment used considerably higher proportions of reads from the input variable. In traditional GP, such a read would completely overwrite the contents of the register being read

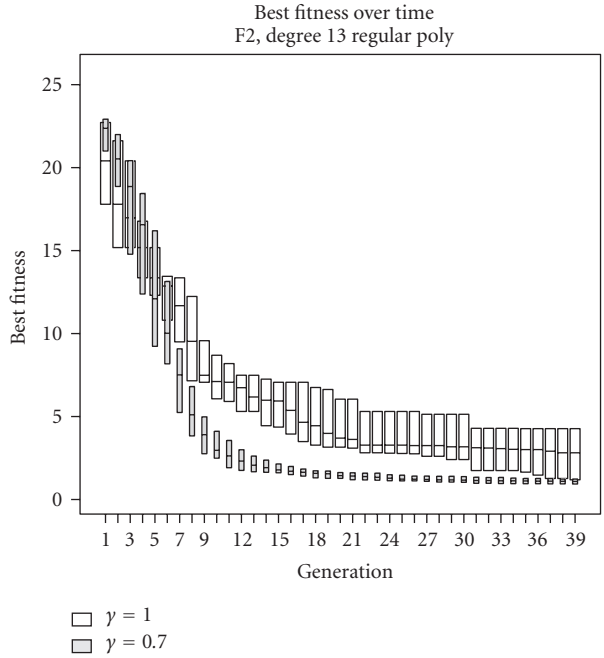


Figure 7: Boxplots showing the middle 50% of the data for the best fitness over time for $\gamma = 0.7$ and for traditional GP ($\gamma = 1$). These results are for a single instruction set (F2) and single polynomial ((5), the regular polynomial of degree 13).

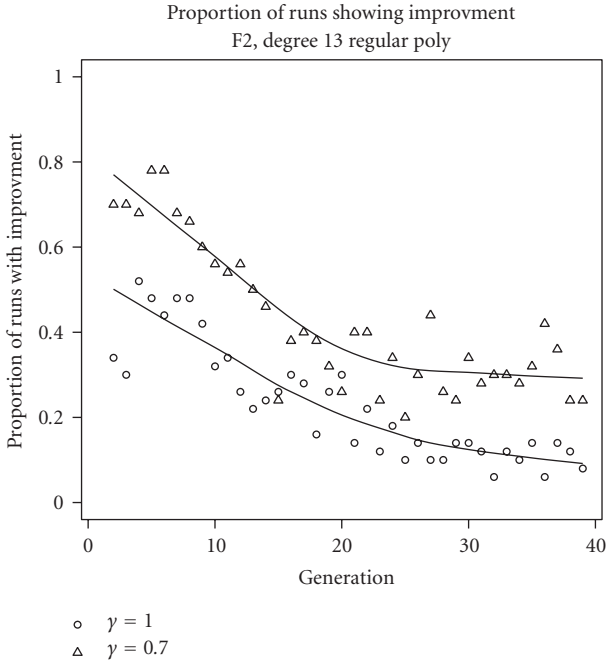


Figure 8: Proportion of runs showing improvement in best fitness over time for $\gamma = 0.7$ and for traditional GP ($\gamma = 1$). Each set of points is overlaid with a curve generated using local polynomial regression fitting (a lowess curve). These results are for a single instruction set (F2) and a single polynomial ((5), the regular polynomial of degree 13).

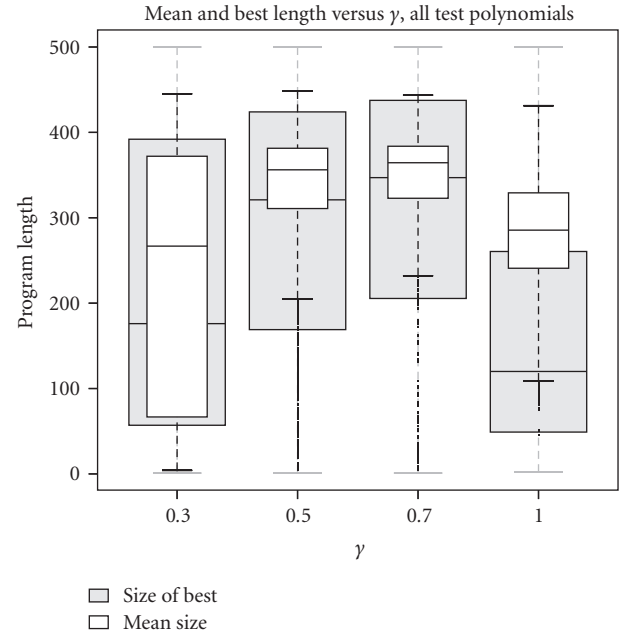


Figure 9: Boxplots showing the distribution of both mean and best-of-run program lengths for the four different values of gamma used in this study, across all runs (all polynomial classes and degrees, all instruction sets). The differences across the γ for both the mean and best-of-run lengths are statistically significant using pairwise Wilcoxon tests. The differences in variances across both the mean and best-of-run lengths are also statistically significant (using the Fligner-Killeen test of homogeneity of variances).

into, and would consequently need to be used with some caution. In our Memory with Memory system, however, such an instruction blends the contents of a register with the input variable x (which is not in fact a terrible approximation of the target function). Conversely, traditional GP was more likely to include the division instructions (near the bottom of the table) as well as, to a lesser degree, subtraction and multiplication. Given that neither division or subtraction are needed to solve the problem, their increased presence in the traditional GP runs could help explain traditional GP's generally poorer performance on this problem.

5.2. Soft Return Operations in Tree-based GP. 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 earlobe 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).

Table 5: The difference in proportions of the different instructions (from F2) as they appeared in the best-of-run individuals in the runs on the degree 13 regular polynomial (5) using $\gamma = 0.7$ and traditional GP ($\gamma = 1.0$). Negative values mean that those instructions appeared with higher proportion in the soft assignment ($\gamma = 0.7$) runs, while positive values mean that those instructions appeared with a higher proportion in the traditional GP runs.

Used more in soft assignment		Neutral		Used more in hard assignment	
Instruction	Difference in proportions	Instruction	Difference in proportions	Instruction	Difference in proportions
`R1 := X`	−0.025	`R1 := 1`	0.000	`R4 := R1`	0.001
`R2 := X`	−0.022	`R6 := R1`	0.000	`R1 := R1 * R2`	0.002
`R1 := R1 + R2`	−0.012	`Swap R1 R3`	0.000	`R6 := R2`	0.002
`R5 := R1`	−0.009	`Swap R2 R3`	0.000	`R1 := R3`	0.003
`R4 := R2`	−0.009	`R2 := R5`	0.000	`R5 := R2`	0.004
`R2 := 1`	−0.009	`R2 := R6`	0.000	`R1 := 0`	0.004
`R2 := R1 + R2`	−0.007			`Swap R1 R2`	0.004
`R3 := R2`	−0.006			`R1 := R4`	0.004
`R2 := R1`	−0.005			`R2 := R1 - R2`	0.005
`R3 := R1`	−0.005			`R2 := R3`	0.006
`R1 := R2`	−0.001			`R1 := R1 - R2`	0.006
				`R2 := R4`	0.006
				`R1 := R6`	0.007
				`R2 := R1 * R2`	0.008
				`R1 := R5`	0.009
				`R2 := R1 % R2`	0.011
				`R2 := 0`	0.012
				`R1 := R1 % R2`	0.013

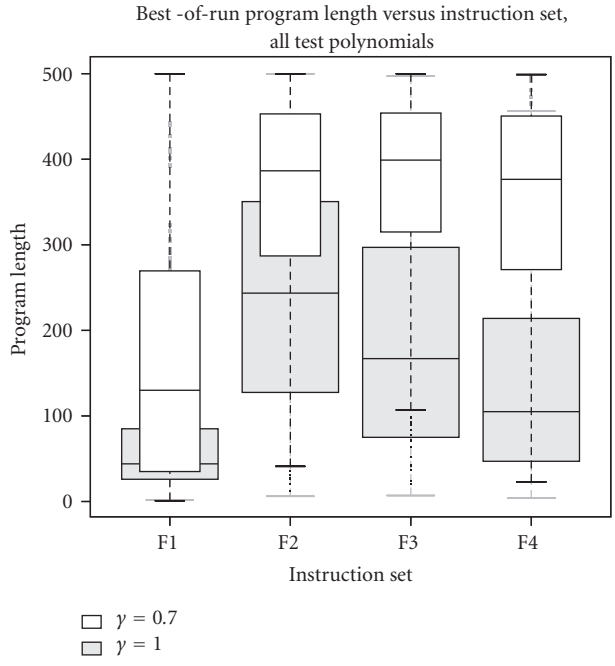


Figure 10: Boxplots showing the distribution of the best-of-run program lengths for the four different instruction sets F1-F4, for $\gamma = 1.0$ and $\gamma = 0.7$, across all polynomial classes and degrees. All differences in the best-of-run sizes for $\gamma = 1.0$ are statistically significant using a pairwise Wilcoxon test. For $\gamma = 0.7$, however, all pairwise differences were statistically significant (again using a pairwise Wilcoxon test) *except* the pairs (F2, F3) and (F2, F4).

5.2.1. Problems

Sine Problem. The sine symbolic regression problem was used for illustration of the TinyGP system in [3]. 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 $x \in \{0.0, 0.1, 0.2, \ldots 6.2\}$. Based on the stopping criterion indicated in Table 2, 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.

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 Figure 11.

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 the series at that particular time. In this work we have used the Mackey-Glass chaotic time series for both types of applications. We ran three types of tests: one for regression and two for prediction.

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,

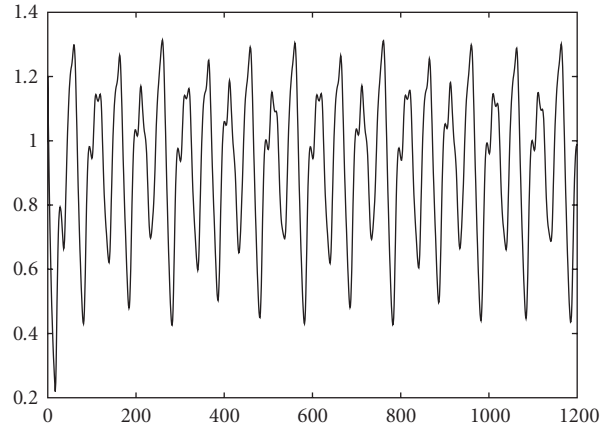


Figure 11: Mackey-Glass Chaotic Time Series.

1, etc. up to 50) into the corresponding value in the Mackey-Glass chaotic time series.

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 second prediction problem was similarly constructed. The difference here was that we used 7 input samples that were not consecutive, but at distances of 1, 2, 4, 8, 16, 32 and 64 from the sample we wanted to predict. This produced a training set of 1,137 fitness cases.

Prime Prediction Problem. The prime prediction problem was originally suggested as a competition for the GECCO 2006 conference. That version of the problem asked participants to evolve a polynomial with integer coefficients such that given an integer value i as input produced the ith prime number, $p(i)$, for the largest consecutive range of values $1 \ldots k$. 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 $1.6272 + 0.7747 \times x + 0.05215 \times x^3 - 2.7092 \times 10^{-6} \times x^8 + 1.5748 \times 10^{-14} \times x^{18} - 2.1966 \times 10^{-16} \times x^{20}$ that correctly predicts only the first 8 primes. Walker and Miller, the runner-ups, did much better but with a solution that wasnot 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 $0.05 \times 11 = 0.55$.

Polynomial Symbolic Regression. We applied the MwM tree-based 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 $[-1, 1]$: $\{-1.0, -0.9, -0.8, \ldots, 0.8, 0.9, 1.0\}$. The target then is to minimise this error.

EEG Reconstruction Problem. Brain electrical activity is typically recorded from the scalp using Electroencephalography (EEG). This is used in electrophysiology, in psychology, as well as in 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 [20]. This is because there is neither muscular activity (which produces very large potentials) nor, obviously, neural activity in the external ear. 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 back to the ears (for comparison with other research, for example). 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 2 KHz. 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 set of 5,000 fitness cases. Using different fragments

Table 6: Success rates and average end-of-run program size versus hardness of return operation (γ) in the Sine symbolic regression problem. All pairwise differences in success rate are statistically significant.

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

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.

The tree-based GP system had 64 input variables (the voltage values of the 64 channels) and one output (the left ear voltage).

5.2.2. Results

Sine problem. In the sine symbolic regression problem, for each configuration of γ we performed 500 independent runs with populations of size 10,000. Our results are reported in Table 6.

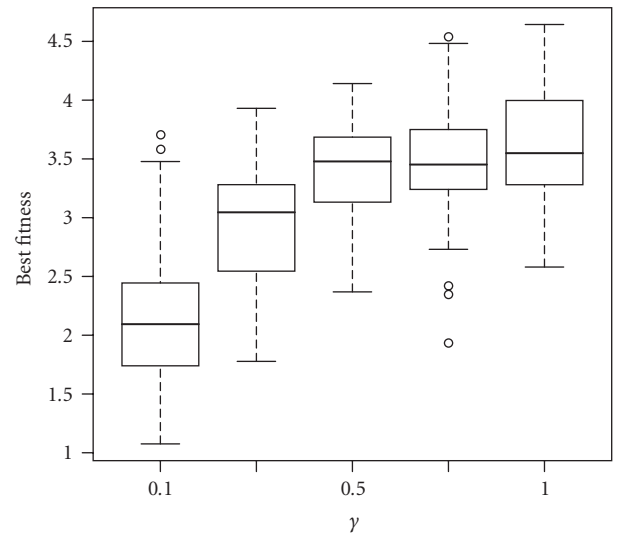
The results show that most GP configurations that use MwM performed better than the standard GP case ($\gamma = 1$), with the best MwM configuration ($\gamma = 0.5$) effectively doubling the success rate of the system. It is also apparent from the results that as the value of γ decreases, the average size of the programs at the end of the run increases. As in the case of our register-based GP system with soft assignment, this is a common feature in our experiments with tree-based GP with MwM. Also in the tree-based case, 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.

Mackey-Glass Problems. With the Mackey-Glass chaotic time series, as we mentioned above, 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. Table 7 shows the results with the symbolic regression version of the problem. 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 with the Mackey-Glass time series we had 8 independent variables representing the values taken by the time series in 8 consecutive samples. The results of our experiments are shown in Table 8. All GP configurations were very successful at solving the problem. This is not surprising given the high correlation between

Table 7: Success rates and average end-of-run program size versus hardness of return operation (γ) (left) and box plot of best of run fitnesses (right) in the Mackey-Glass regression problem. The success rates for $\gamma = 0.3$ and $\gamma = 0.1$ are statistically significantly different from all others and each other.

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



samples present in the Mackey-Glass series combined with the fact that our stopping criterion 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. 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).

In the second prediction problem, where we used 7 nonconsecutive samples at distances of $1, 2, \ldots, 64$ from the target sample we obtained the results shown in Table 9. 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. 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).

Prime Prediction Problem. In the prime prediction problem we performed 100 runs with populations of size 100,000 for each assignment of γ, with the results summarised in Table 10. 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

Table 8: GP performance for different values of hardness of return operation (γ) in Mackey-Glass prediction problem with consecutive samples.

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

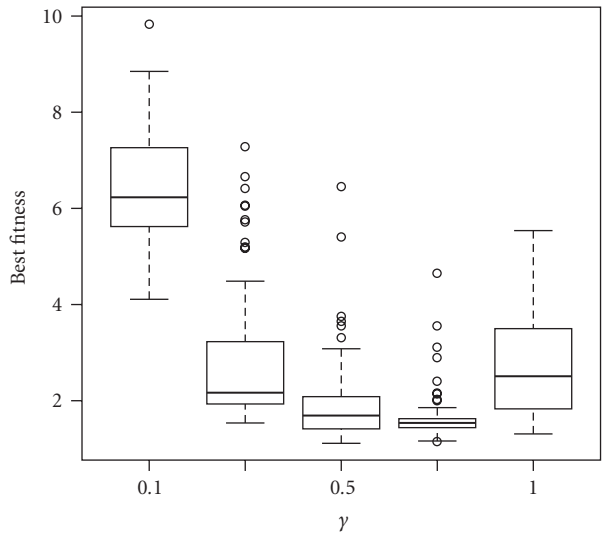
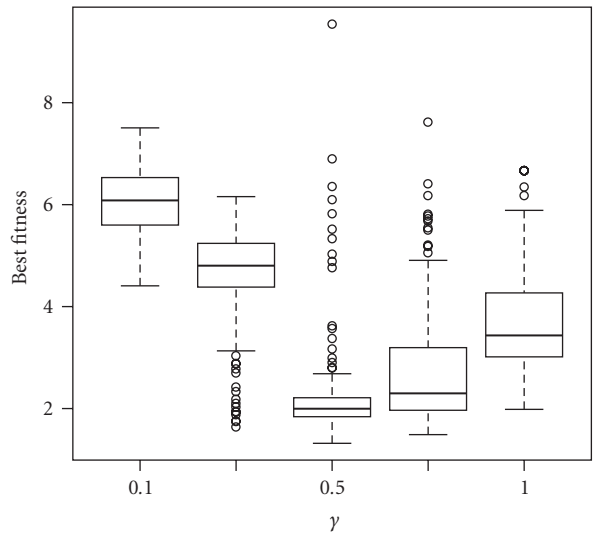


Table 9: GP performance for different values of hardness of return operation (γ) in Mackey-Glass prediction problem with nonconsecutive samples.

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



to solve for standard GP into a problem of moderate difficulty. We again see that solutions are bigger for smaller γ's.

Polynomial Symbolic Regression. With the two polynomial symbolic regression problems using two target polynomials $x^2+1.419x+1.009$ and $8x^5+3x^3+x^2+6$ 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 second polynomial which is much harder. The results are shown in Table 11.

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

EEG Reconstruction Problem. In the EEG reconstruction problem we performed 2,000 runs for each configuration of γ with a population of size 10,000. Table 12 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), with smaller values of γ again leading to slightly larger program sizes.

Table 10: Success rates and average end-of-run program size versus hardness of return operation (γ) in prime prediction problem. The success rates for $\gamma = 0.3$ and $\gamma = 1.0$ are statistically significantly different.

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

6. Conclusions

In this paper we have introduced the idea of Memory with Memory GP, where we use "soft" assignments to registers instead of the "hard" assignments used in most computer science (including traditional GP). Instead of having the new value completely overwrite the old value of the register, these soft assignments combine the two values, using a weighted average in the work reported here.

In addition, we have extended the idea of memory-with-memory 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",

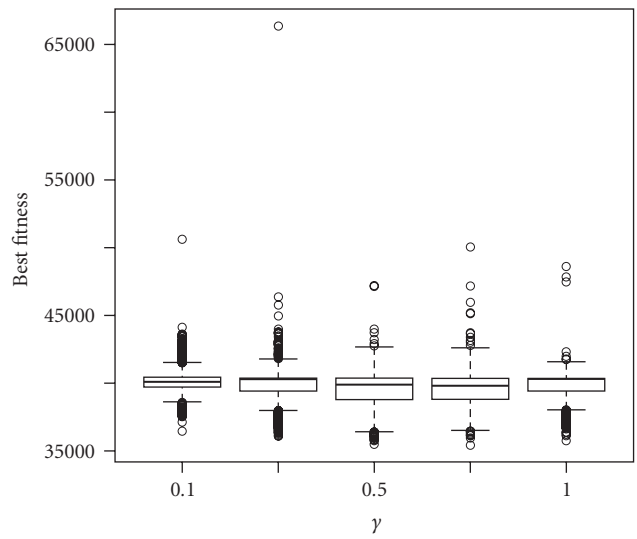
Table 11: Success rates and average end-of-run program size versus hardness of return operation (γ) in polynomial regression problems.

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[(a)]	γ	Success rate[(b)]	γ	Success rate[(c)]
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

[(a)]Differences are significant except for $\gamma = 0.7$ versus $\gamma = 0.5$, $\gamma = 0.7$ versus $\gamma = 0.1$, $\gamma = 0.5$ versus $\gamma = 0.3$, and $\gamma = 0.5$ versus $\gamma = 0.1$. [(b)]Differences are significant except for $\gamma = 0.7$ versus $\gamma = 0.5$, $\gamma = 0.5$ versus $\gamma = 0.3$, $\gamma = 0.5$ versus $\gamma = 0.1$, and $\gamma = 0.3$ versus $\gamma = 0.1$. [(c)]Differences are significant except for $\gamma = 0.5$ versus $\gamma = 0.1$, $\gamma = 0.5$ versus $\gamma = 0.3$, and $\gamma = 0.3$ versus $\gamma = 0.1$.

Table 12: 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



return operation. 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.

Our extensive empirical tests with symbolic regression problems show that in a linear register-based GP system Memory with Memory GP almost always does as well as GP with hard assignments, while significantly outperforming it in several cases. Memory with Memory GP also tends to be far more consistent, having much less variation in its best-of-run fitnesses than traditional GP.

In tests with a variety of symbolic regression and prediction problems using tree-based GP, again we found that MwM GP almost always does very well compared to traditional GP. 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.

Overall the data suggest that Memory with Memory GP works by successively refining an approximate solution to the target problem. Where traditional GP may get stuck in local optima, MwM GP can continue to improve solutions over time even if slowly and in small increments. This means that it is less likely to get the sort of exact solution that one might find with traditional GP. Also, solutions evolved using MwM tend to be bigger than without it. Unlike with other systems, however, this cannot be attributed to introns since with MwM every instruction contributes to a program's output.

7. Future Work

This is a first exploration of a new approach to state updates in GP, and we could only examine a handful of the many alternatives.

An obvious area for future exploration would be the specific implementation of soft assignment and soft return operations. We used a weighted average between the previous and new value (Section 3), but one could, for example, use moving averages, where only the last k values impact the new value, allowing old values to be completely "forgotten" over time. Other approaches such as nonlinear combinations could certainly be explored. We also didn't perform a detailed exploration of different values of γ, or its interaction with other parameters of our evolutionary system.

Another issue not addressed here is whether it would be beneficial to distinguish between different kinds of assignments and return operations, making some soft and some hard. An instruction like `R1:=R1+R2`, for example, already incorporates the old value of R1, and it could be argued that soft assignment is unnecessary there. The instruction `R1:=R1*R2` also typically includes the old value of `R1`, but if `R2:=0` then the old value of `R1` is completely overwritten. Function sets F3 and F4 both contain hard and soft versions of some assignments, so we can get a sense of how evolution combines soft and hard assignment operators. A more sophisticated option would be to allow each instruction to have its own value of γ which could be adjusted over time via some process (e.g., evolution or backpropagation).

One of the challenges with MwM is that it makes the evolved solutions harder to represent and analyze since every assignment and every return operation is in fact a linear combination of two values. An interesting possibility would be to start with $\gamma < 1$, but progressively move it to 1 over the course of a run. This might have the effect of smoothing the fitness landscape, but itis not clear how easily the system would transition from the approximations generated by soft assignment/return to a successful (exact) solution using hard assignment/return.

Similarly, one could start with hard assignments/returns, but when a run appears stuck, decrease γ (making assignments/returns softer) in the hope of introducing a gradient or at least a neutral network that would allow for additional progress, possibly increasing γ again when progress has resumed.

Another interesting direction to take this work is to look at how it performs on noisy and dynamic problems, e.g., problems where the target function changes over time.

In future research we would like also to try to understand what makes a problem hard or easy for systems with different forms of MwM.

Acknowledgments

The authors would like to thank EPSRC (Grant EP/G000484/1) for financial support, and the Dagstuhl seminar 08051 on the Theory of Evolutionary Algorithms, where the initial study of these ideas was finalized. Nic would also like to thank Riccardo and the University of Essex for being such gracious hosts during his research sabbatical.

References

[1] A. M. Turing, "The essential turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of enigma," in *On Computable Numbers, with an Application to the Entscheidungsproblem*, pp. 58–87, Oxford University Press, Oxford, UK, 2004.

[2] J. von Neumann, "First draft of a report on the EDVAC," Tech. Rep., United States Army Ordnance Department and the University of Pennsylvania, 2008, http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf.

[3] R. Poli, W. B. Langdon, and N. F. McPhee, "A field guide to genetic programming," (With contributions by J. R. Koza), 2008, http://www.gp-field-guide.org.uk/.

[4] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction; On the Automatic Evolution of Computer Programs and Its Applications*, Morgan Kaufmann, San Francisco, Calif, USA, 1998.

[5] A. Teller, "The evolution of mental models," in *Advances in Genetic Programming*, K. E. Kinnear Jr., Ed., chapter 9, pp. 199–219, MIT Press, Cambridge, Mass, USA, 1994.

[6] S. Brave, "Evolving recursive programs for tree search," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear Jr., Eds., chapter 10, pp. 203–220, MIT Press, Cambridge, Mass, USA, 1996.

[7] P. J. Angeline, "An alternative to indexed memory for evolving programs with explicit state representations," in *Proceedings of the 2nd Annual Conference on Genetic Programming*, J. R. Koza, K. Deb, M. Dorigo, et al., Eds., pp. 423–430, Morgan Kaufmann, Stanford University, CA, USA, July 1997.

[8] W. B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1 of *Genetic Programming*, Kluwer Academic Publishers, Boston, Mass, USA, 1998.

[9] W. S. Bruce, "Automatic generation of object-oriented programs using genetic programming," in *Proceedings of the 1st Annual Conference on Genetic Programming*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., pp. 267–272, MIT Press, Stanford University, Calif, USA, July 1996.

[10] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.

[11] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO '05)*, H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, et al., Eds., vol. 2, pp. 1689–1696, ACM Press, Washington, DC, USA, June 2005.

[12] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence : From Natural to Artificial Systems (Santa Fe Institute Studies on the Sciences of Complexity)*, Oxford University Press, San Diego, Calif, USA, 1999.

[13] M. Dorigo and T. Stützle, *Ant Colony Optimization (Bradford Books)*, The MIT Press, Cambridge, Mass, USA, 2004.

[14] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

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

[16] S. Luke, "Genetic programming produced competitive soccer softbot teams for robocup97," in *Proceedings of the 3rd Annual Conference on Genetic Programming*, J. R. Koza, W. Banzhaf, K. Chellapilla, et al., Eds., pp. 214–222, University of Wisconsin, Madison, Wis, USA, July 1998.

[17] L. Spector and S. Luke, "Cultural transmission of information in genetic programming," in *Proceedings of the 1st Annual Conference on Genetic Programming*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., pp. 209–214, MIT Press, Stanford University, Calif, USA, July 1996.

[18] W. Gang and T. Soule, "How to choose appropriate function sets for GP," in *Proceedings of 7th European Conference on Genetic Programming (EuroGP '04)*, M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, Eds., vol. 3003 of *Lecture Notes in Computer Science*, pp. 198–207, Springer, Coimbra, Portugal, April 2004.

[19] S. Besetti and T. Soule, "Function choice, resiliency and growth in genetic programming," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO '05)*, H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, et al., Eds., vol. 2, pp. 1771–1772, ACM Press, Washington, DC, USA, June 2005.

[20] S. J. Luck, *An Introduction to the Event-Related Potential Technique*, MIT Press, Cambridge, Mass, USA, 2005.