A Study in Program Response and the Negative Effects of Introns in Genetic Programming

David Andre
andre@flamingo.stanford.edu
(415) 941-9137
Visiting Scholar - Dept. of CS
Stanford University
860 Live Oak, #4 Menlo Park, CA 94025

Astro Teller
astro@cs.cmu.edu
(412) 268-7123
Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

ABSTRACT

The standard method of obtaining a response in tree-based genetic programming is to take the value returned by the root node. In non-tree representations, alternate methods have been explored. One alternative is to treat a specific location in indexed memory as the response value when the program terminates. The purpose of this paper is to explore the applicability of this technique to tree-structured programs and to explore the intron effects that these studies bring to light. This paper's experimental results support the finding that this memory-based program response technique is an improvement for some, but not all, problems. In addition, this paper's experimental results support the finding that, contrary to past research and speculation, the addition or even facilitation of introns can seriously degrade the search performance of genetic programming.

1 Introduction

The traditional method for extracting the response from each individual in GP was inspired by the functional programming of LISP. The most common way to get a program's response is to take the value returned by the root node. The tree whose value is taken to be the answer, especially when an individual consists of multiple trees (such as automatically defined functions (ADFs)), is usually called the Result Producing Branch (RPB). We label this traditional method Result Producing Branch Response, or RPBR. While using the return value of the RPB is the simplest thing to do, it gives exponentially decreasing importance to nodes further down the tree. In other words, nodes lower in the tree have exponentially less of an effect on the outcome in RPBR. We conjecture that this situation can increase the complexity of the search.

The genesis of this paper was the idea of a Memory-Based Program Response (MBPR) paradigm. This idea has already been suggested and found to be successful in non-tree representations (e.g., [Teller and Veloso, 1995a, Teller and Veloso, 1995b]). The hypothesis that this paper addresses is that the advantages of MBPR outweigh its disadvantages for many problems. This paper includes a description of the MBPR mechanism, presentation of experiments designed to test the hypothesis, and analysis of those experimental results.

Section 2 describes an alternative to RPBR. Section 3 presents a set of runs on a classic GP benchmark on which MBPR does well. Then sections 4 and 5 present two other benchmark domains for which the MBPR costs are negligible and high, respectively. The following two sections (6 and 7) describe the important effects of introns in these reported results and comment on the implications. Finally, section 8 concludes with a summary and future research directions.

2 Introduction to the Memory Based Programming Response Method

The idea of MBPR is that one or more memory elements be considered to be the "Answer Location" in memory. After the program tree has been evaluated the value of this memory location or locations is taken to be the program's response. For this paper we will take indexed memory [Teller, 1994] as our memory model and use cell 0 of memory to be the answer location. Thus, in this version of MBPR, the tree is executed and then `Memory[0]` is taken to be the program's response.

2.1 MBPR: Non Tree Representations

Teller and Veloso have previously reported success using the MBPR paradigm in GP with representations that are not tree-structured [Teller and Veloso, 1995a, Teller and Veloso, 1995b]. While this past research is a good justification for careful investigations into the actual effectiveness of MBPR in alternate program representations, we will mention here a few persuasive reasons for the use of MBPR in non-standard program response situations.

The first non-standard response situation that is becoming increasingly common in the GP field is to have a single program return more than one value. Having separate trees that evolve together is one solution to this problem, but this is really a group of programs that share a fitness measure, not a single program with multiple outputs (e.g., [Langdon, 1995]). A simple way to achieve this effect is to use MBPR and use multiple memory locations for the multiple return values. For example, if we want our programs to return three values, we can treat `Memory[0]`, `Memory[1]`, and `Memory[2]` as these three values upon program completion.

A second non-standard response situation is one in which the programs must report a value periodically while running continuously or in which the programs may run for a long or infinite amount of time while the answer is required after a fixed or at least limited amount of time. In such cases, there is no way to force the programs to output a value from some special function node (e.g., the root node of a tree) when desired. If the MBPR paradigm is used, the value of `Memory[0]`, for example, can be polled as needed without interruption of the program or concern for its state.

A third circumstance is one in which the representation itself is not isomorphic to a tree-structured representation. In such cases, as long as memory is being used, the MBPR paradigm is trivial to implement. Conversely, if an RPBR strategy is to be used, then you have to pick a special unit of the program to treat as the answer and have some reassurance that that unit will be activated within the time constraints of the problem. In the MBPR method, the program becomes responsible, not for returning a best value, but for updating its memory with its best guess.

The three reasons just outlined, and others, clearly show MBPR to be more flexible than RPBR. Thus, only a high computational overhead should prevent the use of the MBPR paradigm.

2.2 MBPR: Tree Representations

The following two definitions will be useful in discussing the relative merits of the RPBR and MBPR paradigms.

Vertical Node Interference is the blocking of a subtree result by one of its ancestors. This phenomena is aggravated by representations in which there is an non-uniform distribution of control among the nodes. The tree representation is such a representation. The tree representation provides an inverse exponential distribution of control among the nodes as a function of depth. The root controls the return value of all N nodes. If the root has arity A then each of its children has, on average, control over N/A nodes.

Sibling Coordination Interference is the blocking of a subtree's result by a node that is *not* an ancestor. This is the search difficulty brought about by representations in which nodes that are not hierarchically related (e.g., ancestor-descendant relation) to each other can still affect each other's results through side-effects. The ant problem representation in [Koza, 1992] suffers from this. In fact, any representation that includes memory usage (e.g., indexed memory) potentially suffers from this effect.

The argument for MBPR *in the tree-structured GP representation* follows. If a program has some portion of it that computes an accurate or perfect answer, it must (independent of the representation and response collection method) rely on the rest of the program not to interfere with that answer until the environment collects it. For example, in tree-GP using RPBR, if a subtree computes a good answer, it must depend on the path from the root of that subtree to the root of the tree to be non-destructive. This is the vertical node interference problem mentioned in the previous section.

Clearly, MBPR suffers from an analogous problem. If a program is allowed to update its "answer" memory cell in any part of its program, then clearly a "destructive" program fragment can overwrite a correct answer that the "good" part of the program had already written to the "answer" location in memory. This is the sibling coordination interference problem.

Because the MBPR approach is largely (although not entirely) position independent, it seems to have a natural advantage over RPBR. A response collection policy is position independent to the extent that a subtree that computes "the answer" for a given individual can be moved without affecting the subtree's ability to produce "the answer" in the new individual. In the MBPR paradigm, a subtree that includes a WRITE to the answer cell in memory can be moved to a new individual program and that subtree will still be able to store its value into the answer cell in memory. Obviously, the high degree of vertical node interference means no such guarantee can be made when RPBR is used. MBPR is not entirely position independent in the sense that while a transplanted subtree can compute an answer and store it in `Memory[0]`, that value could later be overwritten by another subtree.

To believe that the MBPR paradigm will be better than RPBR paradigm for a particular problem, we must believe that searching with high sibling coordination interference will be easier than searching with high vertical node interference. It would seem that the a priori argument for MBPR is that, in the worst case, any tree that is a solution in RPBR space, by adding a WRITE to the top, becomes a solution in the MBPR space. Even if there are other WRITEs to the answer location in the tree, this root node, since it is last, cannot be overwritten. It seems that this "capping" procedure would be simple to evolve if it were necessary. However, the side-effects of introns introduced by memory (sibling coordination interference) complicates this analysis.

All these reasons contribute to our belief that MBPR is a more flexible strategy for program response collection. Now, experiments will help us to understand when, if ever, the costs associated with MBPR outweigh the arguments for using it.

3 Positive Results with MBPR

All of the experiments in this paper refer to four different experimental conditions that were run together for comparisons. Below is a list of the four types of experiments as they are annotated in the tables and an explanation of what they mean.

- *ProblemType*-**RPBR** indicates that the problem was run with the usual function set and the result was taken from the return value of the root node of the tree.
- *ProblemType*-**RPBR-X** (N) indicates that the problem was run with the usual function set plus a perversion of the indexed memory functions in which (READ V) and (WRITE U V) both return their last argument (V). N indicates the number of indexed memory cells (see below for the effect $N < 2$ on these run types). In this case the result is still taken from the return value of the root node of the tree.
- *ProblemType*-**RPBR-M** (N) indicates that the problem was run with the usual function set plus the standard indexed memory functions (READ *index*) and (WRITE *index new-value*) with N possible indexed memory cells. In this case the result is still taken from the return value of the root node of the tree.
- *ProblemType*-**MBPR** (N) indicates that the problem was run with the usual function set plus the indexed memory functions (READ *index*) and (WRITE *index new-value*) and N cells. In this case the result is *not* taken from the return value of the root node of the tree. Instead, after the tree is evaluated, the current value of `Memory[0]` is taken to be the program's result.

When $N = 1$, no indexing is necessary, and READ becomes a terminal and WRITE becomes a single arity function. When $N = 0$, READ is no longer included in the function set. Single arity WRITE is left in the function set to save values, but they can not be referenced later by the program. The *ProblemType*-**RPBR-M** runs were only performed when $N > 0$, because when $N = 0$, the memory functions have no effect on the output in RPBR case, and are therefore equivalent to their inert counterparts used in the *ProblemType*-**RPBR-X** runs.

Table 1: **The 5-Parity problem**

Basic 5-Parity Function Set

functions	and	or	nand	nor	
Terminals	d0	d1	d2	d3	d4

5-Parity Problem Parameters
Max Generations = 75
Max Tree Size = 500
Pop Size = 32,000

5-Parity Problem Results

Problem Type	Comp. Effort	#Solved	#Run
5Parity-RPBR	3,435,185	26	27
5Parity-RPBR-X (2)	17,820,172	12	27
5Parity-RPBR-M (2)	1,640,000	26	26
5Parity-MBPR (2)	1,189,000	26	26

All the results presented in this paper had several methodological points in common. Where the methodologies differed, the differences will be mentioned when the data is presented. For all of the experiments, we used tournament selection with sample size of 8. Unless otherwise noted, the maximum number of generations was 101. All the runs discussed in this paper with a maximum node limit of 2000 or above had an initial maximum depth of 10. In addition, the runs with the Gudermannian[1] function on the $N = 0$ case with a maximum node limit of 1000 also used an initial maximum depth of 10. All other runs used had an initial maximum depth of 8. The recombination percentages for all experiments were: 10% copy, 1% mutation, 89% crossover (10% leaves, 79% internal nodes).

The first problem on which we tested the MBPR paradigm was the 5-Parity problem. There were two motivations for choosing this problem. The first was that, as the most classic GP benchmark, it is one that most readers are familiar with. The second reason for choosing a parity problem was the desire for a simple, but not trivial, problem where a subtree has often been observed to express a correct calculation.

This problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80 MHz Power

[1]This paper's regression problem is called the "inverse Gudermannian", referred to here simply as the "Gudermannian". The function is $(log((1/cos(X)) + tan(X)))$.

PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer. The so-called distributed genetic algorithm for parallelization was used with a population size of $Q = 500$ at each of the $D = 64$ demes. On each generation, four boatloads of emigrants, each consisting of $B = 5\%$ (the migration rate) of the node's subpopulation (fitness-based selection) were dispatched to the four toroidally adjacent processing nodes [Andre and Koza, 1996]. This hardware was only used on the Boolean runs. All other runs were performed on a Pentium.

The results on the 5-parity problem are presented in table 1. Not only does the MBPR take 33% of the computational effort of RPBR on the 5-parity problem, but the MBPR style of response actually solves 100% of the time instead of the 96% shown here for RPBR. This large improvement for MBPR may be due to either one or both of the differences between MBPR and RPBR: the addition of memory or the memory based response. In the case where memory is added (5-parity-RPBR-M), 100% of the runs found a perfect solution, and it requires only 48% of the computational effort of the 5-parity RPBR. Taking the answer from memory helps, but just adding memory accounts for more than half of the difference in computational expense.

Clearly, the most dramatic aspect of the data in table 1 is the large amount of computational effort and the low number of solved runs for the 5-Parity-RPBR-X problem. When the inert READ and WRITE operators are added, computational effort increases by a factor of five and solutions occur less than half the time. This is surprising. Why would adding READ and WRITE functions that just passed back their last argument cause such a problem? This question will be addressed in sections 6 and 7. The results from the parity problem indicate that the MBPR approach works well on the most classic GP benchmark.

4 Minimal Cost Results with MBPR

The next benchmark problem we chose for further validation of the MBPR results shown in the previous section was the lawn-mower problem. The lawn-mower problem was chosen for two important reasons. The first is that, as will be described, this version of the lawn-mower problem requires the use of memory. This required use of memory makes it possible to at least partly separate the MBPR paradigm from the general helpfulness of memory (table 1). The second reason was to pick a problem with more complexity than the 5-Parity problem and with a much richer function set.

A version of the lawn-mower problem is described in [Koza, 1992]. The basic concept is that there is a lawn-mower in an 8x8 toroidal grid world. In our version of the problem, the mower only moves once per execution of the tree, and is allowed 200 time steps. If the value returned is less than zero, then the mower turns left. If the value is greater than zero, then it moves forward and mows. If the return value is zero, it does nothing. The lawn-mower's goal is to mow as much of the world as possible during those 200 time steps. Since it receives no sensory input, it must rely on its memory to help it create a path that hits most or all of the grid positions. This description (and the function set shown in table 2) should make it clear that this problem only shares surface level details with the lawn-mower problem in [Koza, 1992]. The main differences are: no state except indexed memory (i.e., no progn), one move at a time (i.e. one move per tree evaluation), and no jumping allowed.

The lawn-mower problem (results for which are shown in table 2:1) was set up so that memory was required[3]. We see that the MBPR takes 1.62 times as much computational effort as RPBR. This is surprising given that the programs are using their memory anyway. One potential explanation for this is that because this is actually a memory critical problem, and zero is a very likely value to occur[4], different program pieces may initially "fight" for control of memory cell 0: some to store the answer, and others to keep track of the mowing movements.

Table 2: **The Lawn-mower Problem**

Basic Lawn-mower Function Set

functions	or	and	if > 0	not	if $= 0$
	pdiv	add	sub	mult	write
	read	incm[2]	dprog2		
Terminals	move	left	-10...10		

Lawn-mower Problem Parameters
Max Generations = 101
Max Tree Size = 500
Pop Size = 2,000

Lawn-mower Problem Results

Section 1: 20 Indexed Memory Cells

Problem Type	Comp. Effort	#Solved	#Run
Lawn-RPBR-M (20)	15,099	143	143
Lawn-MBPR (20)	24,372	136	136

Section 2: 20 IM Cells and 1 dedicated *answer* cell

Problem Type	Comp. Effort	#Solved	#Run
Lawn-MBPR (20/1)	15,516	209	209

[2](INCM(X) adds 1 to `Memory[X]` and then returns new value of `Memory[X]`)

[3]Lawn-RPBR (no memory) and Lawn-RPBR-X (useless memory functions) are not shown because memory is required for the problem.

[4]Zero is especially likely to occur with a function set such as this where aspects of it, like boolean functions and the initally all zeroed valued memory, contribute to the likliness of zero as a value

The above hypothesis suggests the following experiment: continue to use 20 indexed memory cells, but introduce one new dedicated memory cell for the MBPR "answer." To dedicate this memory cell, a new function was introduced into the function set: WRITE-ANSWER. WRITE-ANSWER takes one parameter, the value, and stores that value in the dedicated memory cell. The results of this experiment are shown in table 2:2. The addition of this dedicated memory cell has successfully reduced the computational cost of the MBPR runs to almost exactly that of the RPBR runs. This change in computational cost demonstrates that contention over the "answer" position in indexed memory (memory cell 0) was indeed the expensive aspect of the MBPR runs shown in table 2:1. In short, MBPR does not reduce computational effort in this domain, but does not increase it either.

5 Negative Results with MBPR

Symbolic Regression was chosen for a third benchmark on which to test the MBPR paradigm both because it is a traditional GP benchmark problem, and more importantly, because it has characteristics that suggest that MBPR will not help. Table 3 shows the basic function set that all the regression problems discussed in this paper used (except where explicitly noted).

The relative simplicity of the function set was an important factor in the decision to choose regression as our third problem domain. The algebraic primitives do not lead directly to intron activity (this is discussed more fully in the next section) because 0 and 1 are not provided as terminals. In addition, the algebraic primitives for regression tend to lead to most or all of the tree being used. This set of functions is relatively homogeneous. This homogeneity leads us to suspect that MBPR will hamper rather than help the search process.

Table 3:1 presents the results of the first set of regression runs with the Gudermannian function and 20 indexed memory units. Not only does the MBPR paradigm not help, it is extremely detrimental to performance. In fourteen runs, it never finds a solution. While this is in the direction that was predicted, it is so negative that it is puzzling. Again, we see that adding useless READ and WRITE functions (the Reg-RPBR-X runs) causes a factor of five rise in computational cost and a 28% reduction in number of solutions. It is at this point that we may begin to suspect that the MBPR effects are being swamped by some other effect.

In an effort to find the cause of the performance drop, several experiments were run in which the memory size was reduced. The rationale is that, since the the MBPR paradigm did well on the 5-Parity and had only two memory cells whereas the MBPR method did less well on the lawn-mower problem where there were 20 memory cells, perhaps reducing the indexed memory size will, for this problem, lower the cost of using MBPR. Table 3:2-4 shows the results.

Table 3: **Regression**

Basic Regression Function Set

functions	Add Sub Mult Div	
Terminals	X	Random Ephemeral Constants

Regression Problem Parameters

Function = $(log((1/cos(X)) + tan(X)))$.
Max Tree Size = 1000

Regression Problem Results

Section 1: 20 Indexed Memory Cells

Problem Type	Comp. Effort	# Solved	# Run
Reg-RPBR	35558	28	29
Reg-RPBR-X (20)	183380	19	28
Reg-RPBR-M (20)	129946	10	15
Reg-MBPR (20)	NA	0	14

Section 2: 2 Indexed Memory Cells

Problem Type	Comp. Effort	# Solved	# Run
Reg-RPBR	35558	28	29
Reg-RPBR-X (2)	183380	19	28
Reg-RPBR-M (2)	142710	6	16
Reg-MBPR (2)	1975913	1	17

Section 3: 1 Indexed Memory Cells

Problem Type	Comp. Effort	# Solved	# Run
Reg-RPBR	22589	32	32
Reg-RPBR-X (1)	178732	22	31
Reg-RPBR-M (1)	114506	25	33
Reg-MBPR (1)	540994	12	32

Section 4: 0 Indexed Memory Cells

Problem Type	Comp. Effort	# Solved	# Run
Reg-RPBR	34408	57	57
Reg-RPBR-X (0)	57550	112	114
Reg-MBPR (0)	212157	43	57

It has been repeatedly suggested, though never demonstrated, that unnecessarily large indexed memory sizes can degrade performance. Table 3:2-4 indicates that reducing memory size does not seem to explain the poor performance of the RPBR-X, RPBR-M, and MBPR runs.

The last set of runs in table 3:4 (0 cells for indexed memory) shows runs in which there is no indexed memory to confuse the program while it is computing an answer. Since (Write-Answer x) returns X, it basically has no effect except to save off the answer. In this situation, any tree that has WRITE-ANSWER as its root is *exactly* the same as a tree using RPBR from the point of view of the return value. This is true even if there are other WRITE-ANSWERs in the tree, since the root is evaluated last. It is hard to believe that the problem is eight times easier than "Solve the Gudermannian with Add, Subtract, etc and put a WRITE-ANSWER at the root node."

The runs in table 3:4 have two main points of interest. Notice that while the computational effort of the RPBR-X runs is worse than that of RPBR runs, it is much closer to that of the RPBR runs that to that of the MBPR runs. The former ratio is 1.67; the later is 3.68. This means two things. First, even inert WRITE-ANSWER alone (RPBR-X) is enough to cause a performance degradation. Second, and at least as importantly, MBPR is causing more of a problem than the the intron effect caused by inert WRITE-ANSWER in this case.

6 The Mystery Cause Unmasked

Introns are almost always possible. Whenever a subtree's return value is ignored, that subtree is an intron. There are, however, ways of making introns easier to generate, and therefore more prevalent in programs. For example, in Regression-RPBR-X (20), (WRITE X Y) not only has no effect on memory, but in simply returning Y, it makes the X subtree an intron. The (READ X) function in the RPBR-X (20) runs does not create an entire intron subtree, but merely is an intron itself.

We see the following two ways in which introns can be detrimental to the search process:

1. They can hamper search by providing locations where crossover can happen and have no effect on behavior.

2. They can take up space in the tree and, if tree space is limited, may cramp the actual space for solutions.

In the experiments reported in this section, we attempt to differentiate between these two types of intron effects. Allowing larger maximum tree sizes should only alleviate the cramping effect of introns and not the search degradation effect. These experiments directly compare the effects of increasing maximum tree size on two different types of introns-those created by WRITE-ANSWER in the RPBR-X (0) case, and those created by sibling coordination interference in the MBPR (0) case. We assume (though there is disagreement about this) that the first of these two problems does occur; in our opinion, a wasted search step can do nothing but slow the search process.

The runs in table 4:1 show the results of the Gudermannian problem runs, where the tree sizes are limited to 2000 nodes per tree (as opposed to 1000 nodes per tree in table 3). Notice that the computational effort of the RPBR and MBPR runs in table 3:4 and table 4:1 increases, but their ratio does not change significantly.

The ratio of the computational effort of the RPBR-X runs to the computational effort of the RPBR runs drops from 1.67 to 1.46 when the maximum tree size is increased from 1000 to 2000. Since nothing but the maximum tree-size has changed, and the control for the experiment (RPBR) takes approximately the same amount of time, this means that larger maximum tree sizes made it easier to solve the problem. The RPBR-X runs shown in table 4 differ from the RPBR runs only in that there is a single, useless function. The inert WRITE-ANSWER, because it passes its argument back up, does not even cause introns below it. This is very significant. Adding a single function can cause a run to take 1.67 times as long to complete (table 4:4) and increasing the maximum tree-size can reduce the penalty to 1.46 (2000 node limit). The question arises, "What is the floor for this ratio?"

We see in table 4:2 that moving the maximum tree size to 5000 again has little effect on the RPBR runs or on the MBPR runs. The computational effort of the RPBR-X runs drops to 1.41 times that of the RPBR runs. This confirms our conclusion that the useless functions take up space and force the useful program fragments into a smaller space. By increasing the maximum tree size, this problem is greatly reduced. In the runs with 10000 node maximum tree size, the computational effort of the RPBR-X runs is 1.36 times the computational effort of the RPBR runs. While this is only slightly lower than the 1.41 just mentioned, it is close to the theoretical penalty for adding one useless function (X+1/X

Table 4: **Regression with increased tree size limits**

Regression. $(log((1/cos(X)) + tan(X)))$.

Section 1: Max Tree Size (MTS): 2000

Problem	Comp. Effort	# Solved	# Run
Reg-RPBR	32564	47	51
Reg-RPBR-X (0)	47520	101	103
Reg-MBPR (0)	230615	37	52

Section 2: Max Tree Size (MTS): 5000

Problem	Comp. Effort	# Solved	# Run
Reg-RPBR	37525	43	43
Reg-RPBR-X (0)	53001	85	86
Reg-MBPR (0)	208426	30	43

Section 3: Max Tree Size (MTS): 10000

Problem	Comp. Effort	# Solved	# Run
Reg-RPBR	44116	121	121
Reg-RPBR-X (0)	60064	239	243
Reg-MBPR (0)	304157	67	121

Section 4: Increasing MTS for Reg-RPBR-X (0)

MTS	Comp. Effort	solved/runs	Ratio to RPBR
1000	57550	112/114	1.67
2000	47520	101/103	1.46
5000	53001	085/086	1.41
10000	60064	239/243	1.36

where X is the number of useful functions) which in this case is 5/4 or 1.25. Table 4:4 summarizes this trend.

Comparing the computational effort of the RPBR and the RPBR-X runs for the experiments shown in this section, we see a gradual reduction of the difference between the two as the maximum allowed tree sizes increases. In fact, it seems asymptotic to the theoretical penalty for adding a useless function to the function set.

The computational effort using the MBPR paradigm does not improve noticeably as the maximum tree size increases (table 3:4 and table 4). Thus, the negative effects caused by using the MBPR paradigm are not the cramping effects. This leaves only the "search degradation" intron effect as the explanation for MBPR's higher computational expense. This corresponds with the picture we have painted of the trade-offs between RPBR and MBPR. As the tree grows, the relative importance of vertical node interference and sibling coordination interference does not change.

7 Discussion

7.1 MBPR: Tree Representations

In the previous two sections, we showed three things about the MBPR paradigm:

- For the boolean 5 parity problem, MBPR works well, but much of its advantage was because of the introduction of memory.
- For the iterative lawn-mower problem, a problem that already required memory, MBPR was neutral with respect to computational effort.
- For the regression problems, problems that by their nature (homogeneity of function set) are unlikely to benefit from memory, MBPR was very expensive. Even when WRITE-ANSWER was used instead of indexed memory, the computational effort increase was always at least a factor of 2.

The explanation for MBPR's performance on the 5-parity problem is relatively simple. There is a large amount of redundancy in the parity problems that can be exploited through a technique like ADFs [Koza, 1992]. Since ADFs were not used, the introduction of memory served the same purpose. Values can be stored once in memory and then retrieved in several different places.

The performance improvement observed when using MBPR on the 5-parity problem provides evidence that the vertical node interference is more harmful for the 5-parity problem than is sibling coordination interference. Given that the runs with indexed memory (RPBR-M) differ only from the MBPR runs only in that the representation in the RPBR-M runs allows more vertical node interference, the improvement when using MBPR must be due to that difference.

Another way of understanding the success of MBPR on the 5 parity problem is by investigating its function set. In the 5-Parity problem, all four of the functions allow intron subtrees to form if the other subtree always returns a one or a zero. For example, if a subtree to an AND always returns a 0, the result of the other subtree is ignored. Since the 5-Parity is a boolean problem, constant values of 0 or 1 occur quite often. In contrast, in the regression problem, only a return value of 0 (out of infinitely many possible values) can cause a TIMES to have an intron subtree. Because it is relatively easy to make intron subtrees in the Boolean problem, vertical node interference is much more of a problem than is sibling coordination interference. This explains the success of MBPR on this problem

In the lawn-mower problem, it seems to be that the vertical node interference and the sibling coordination interference are approximately equal; moving from RPBR to MBPR does not make a significant difference. We suggest that this approximate equality of the two interference measures exists because memory is already being used in the RPBR lawn-mower problem. As shown in section 4, the moderate increase in computational effort was caused by contention over use of the `Memory[0]` when a dedicated memory cell is not provided. The lawn-mower problem uses memory in a very different way than does the 5-Parity boolean problem. The lawn-mower must use memory to communicate with itself on a later time step, whereas the 5-Parity problem only uses memory to pass values around. Using memory simulates ADFs for the 5-Parity problem, but not for the lawn-mower problem. This point may help to explain some of the difference between MBPR's effect on the 5-Parity and lawn-mower problems. `Memory[0]` could be a point of contention in either case (possibly more so in the 5-Parity problem when there are only 2 cells). However, these different memory requirements may change the dynamics of memory use in ways that we do not yet fully understand.

Most of the results shown in section 5 used WRITE-ANSWER instead of full indexed memory in a effort to disentangle the issues. One of the aspects of this data that we can not yet fully explain is why WRITE-ANSWER, even when it is not used for MBPR, causes such a noticeable jump in computational effort. Section 7.2 gives a plausible explanation for the underlying cause.

7.2 Introns

It should be clear from the previous section that the MBPR results were being confused by (arguably even obscured by) another effect: introns. This paper has suggested that introns can affect performance in a very negative way. The purpose of this section is to summarize the experimental evidence for introns and to present a partial explanation for their effects. We should mention here that our definition of intron is a behavioral definition. That is, we do not care where the genotype/phenotype line is drawn; an uncalled subtree is an intron, as is a subtree that is called and whose result is ignored.

Before we explain our findings, however, we will summarize what we believe to be the contemporary view of introns in the GP field. Introns have recently been often described as a beneficial aspect of the genetic search that serves at least two important functions. The first "benefit" of introns is often stated to be their ability to protect the program of which they are a part from "destructive" crossover. The second "benefit" of introns is to allow the population as a whole to protect and thereby preserve the best "building blocks" in the population. Efforts to insert extra introns into the population as a search aid [Nordin and Banzhaf, 1995] indicate the current favorable standing of introns in GP.

Many researchers (e.g., [Nordin and Banzhaf, 1995, McPhee and Miller, 1995, Rosca and Ballard, 1995]) have argued that introns are helpful, in that they prevent destructive crossover. However, their argument assumes that the best of the current generation are in the right part of the search space (i.e, that destructive crossover is necessarily bad). While destructive crossover may be bad for natural evolution, the goal of GP is often optimization of a single program, not the survival of any particular individual of individuals.

There are at least three types of introns associated with functions. There are also introns associated with useless terminals, but since this paper had only one experiment with a change in the terminal set ((READ) returning the value of memory for $N = 1$), we will concentrate on the following three function intron types:

- **Local:** Useless functions that simply pass on the values they are passed.
- **Hierarchical:** functions that force one or more of their subtree arguments to be ignored. WRITE in the RPBR-X ($N > 0$) runs done in this paper is an example of this intron type.
- **Sibling (horizontal):** functions that may cause the effects of a sibling subtree to become moot at a later point in the program. An example of this intron type is when a subtree has its contributions to memory overwritten by a subtree that is executed later (as can happen in the RPBR-M and MBPR runs in this paper).

Table 5 summarizes the effect that adding inert versions of READ and WRITE has on the computational effort required to solve various problems. In all of the runs in table 5, READ is useless (because it simply passes its one argument up the tree) (causing Local introns). WRITE actually creates an intron by ignoring its first argument and passing its second argument along (causing Hierarchical introns). Measured by either computational effort or by the likelihood that a perfect solution will be found in the required number of generations, the addition of this useless function and this intron causing function have a very strong negative impact on the search process. Since these two functions are not actually acting as memory conduits, the extra computational expense must be caused by the intron effects.

Table 5: **Result Comparisons**

Run results: Inert Memory (RPBR-X) relative to RPBR MTS is Max Tree Size.

Section 1: using READ and WRITE

MTS	Problem	Comp. Effort Δ	Solution Rate Δ
1000	5Parity (2)	5.18	2.27
1000	Reg (20)	5.23	1.43
1000	Reg (2)	5.23	1.43

Section 2: using READ and WRITE

MTS	Problem	Comp. Effort Δ	Solution Rate Δ
1000	Reg (1)	7.91	0.70

Section 3: using WRITE-ANSWER

MTS	Problem	Comp. Effort Δ	Solution Rate Δ
1000	Reg (0)	1.67	1.02
2000	Reg (0)	1.46	0.94
5000	Reg (0)	1.41	1.01
10000	Reg (0)	1.36	1.02

In table 5:2 we see that when READ becomes a 0 argument function (i.e. a terminal) and WRITE becomes a single argument function (in the $N = 1$ case), and so both cause only local introns, the performance degradation remains. However, when we remove READ entirely and are left with only WRITE-ANSWER (causing only local introns) ($N = 0$) the performance degradation is lessened, although still negative. Further, with only WRITE-ANSWER we no longer have a function that explicitly causes whole subtrees to become introns; WRITE-ANSWER is itself simply an intron since it does nothing useful. This suggests that the addition of one useless function can have a detrimental effect on the GP search process.

The zero arity READ (table 5:2) returns a zero since it has no value to pass along. Since the presence of (READ) in Reg (1) is the only thing that distinguished it from the Reg (0) runs, the presence of zero as a terminal obviously has a large negative effect. The reason for this can again be explained by intron effects. With random ephemeral constants and X, making an intron is possible using the algebraic primitives, but not as easy as when 0 is also readily available. It is the difference between the likelihood of (* Y 0) and (* Y (- Z Z)). This may not seem like a large difference, but it has a large effect in this case, as shown in the difference between table 5:2 and the first line of table 5:3.

Table 5:3 shows the effect of the useless function (WRITE-ANSWER) as the maximum size allowed for a tree is raised from 1000 to 2000, 5000, and then 10000. In all three cases, the expense of local introns has been reduced to a modest cost. The conclusion is that for populations with sufficiently large members, this particular intron effect (addition of inert functions) ceases to be expensive. This says something important about tree size. The prevailing wisdom seems to be that as long as the tree size maximum is large enough for a non-compacted solution to fit, the search will go fine. Table 5:3 implies that the necessary tree size ceiling is also a function of the ease with which the functions can form introns.

It is important to notice that the conclusion of the previous paragraph is made in the context of $N = 0$. By increasing the maximum tree size, we alleviate this intron "cramping effect". In the case where $N > 0$ and/or MBPR is used, the intron expense incurred from the search degradation effect dominates (as entire subtrees can be intron-ized). In this, the $N > 0$, case, increasing the maximum tree size limit has little effect, as is shown in table 4. Now we have an answer to the question "Why won't the 'capping procedure' (proposed in section 3) guarantee a minimal cost increase for MBPR?" The answer is, the 'capping procedure' presupposes a correct solution to cap. The search degradation caused by introns appears to seriously impede the evolution of such solutions.

8 Conclusions

The hypothesis of this paper is that the greater flexibility and reduction of vertical node interference in MBPR outweighs the possible negative effects of increased sibling coordination interference and the possible cost of introducing memory.

The experimental results presented show that the intron effects from MBPR are too damaging for those problems (e.g., regression) that are very homogeneous. This correlates well with whether or not those problems benefit from the introduction of memory; When introducing memory to a problem does not help or even hurts, MBPR is unlikely to provide a benefit since memory must be added for MBPR to be accomplished.

When memory is necessary, the experimental results indicated that, for the test problem, MBPR did not have a dramatic performance impact. Given the negligible cost, the flexibility and generality of MBPR certainly makes it worth consideration for these problem types. However, if the problem is more "horizontal", the likelihood of introns is already present, and vertical node interference is a frequent occurrence then (e.g., the Parity problem) MBPR is likely to give a boost to the search process.

The take home messages are two fold. First, MBPR does not seem to be a problem independent improvement for tree representations, although it seems to be an advantage when memory use is simple, helpful, or required. In addition, the MBPR paradigm, as described in section 2.1, has several other points in its favor. To verify these conclusions, for both tree-GP and non-tree-GP representations, further experiments should be done with more complex problems.

The second conclusion of this paper is that introns are probably damaging. This goes against conventional wisdom, but in the data presented in this paper, the control experiments identify intron effects as the only possible cause of significant performance degradation under a variety of conditions and on a variety of problems. Results were presented that showed this change, to varying degrees, with local, hierarchical, and sibling introns.

The corollary to this conclusion is that simple changes in the function set can have large effects on computational effort. In particular, the addition of one intron causing function can make it difficult or impossible to solve what otherwise would have been an easy problem in genetic programming.

Acknowledgements

This work is supported in part by the Hertz Foundation whom Astro thanks for their kind support. The support and comments of John Koza and Forrest Bennett are greatly appreciated by both David and Astro.

References

[Andre and Koza, 1996] D. Andre and J. Koza. Parallel genetic programming: A scalable implementation using the transputer architecture. In P. Angeline and K. Kinnear, editors, *Advances In Genetic Programming II*. MIT Press, 1996.

[Koza, 1992] John Koza. *Genetic Programming*. MIT Press, 1992.

[Langdon, 1995] William Langdon. Evolving data structures with genetic programming. In Stephanie Forrest, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kauffman, 1995.

[McPhee and Miller, 1995] N. McPhee and J. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 303–309. Morgan Kauffman, 1995.

[Nordin and Banzhaf, 1995] P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 310–317. Morgan Kauffman, 1995.

[Rosca and Ballard, 1995] J. Rosca and D. Ballard. Causality in genetic programming. In L. Eshelman, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 256–263. Morgan Kauffman, 1995.

[Teller and Veloso, 1995a] Astro Teller and Manuela Veloso. PADO: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1995.

[Teller and Veloso, 1995b] Astro Teller and Manuela Veloso. Program evolution for data mining. In Sushil Louis, editor, *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases*. JAI Press, 1995.

[Teller, 1994] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 199–220. MIT Press, 1994.