
Iteration Over Vectors in Genetic Programming

Evan Kirshenbaum

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
kirshenbaum@hpl.hp.com

Abstract

This paper describes the results of using genetic programming with bounded iteration constructs, which allow the computational complexity of the solution to be an emergent property. It is shown that such operators render the even-6-parity problem trivial, and the results of experiments with other, harder, problems that require $O(n)$ complexity are shown. This method is contrasted with Automatically Defined Iterators.

1 BACKGROUND

This paper builds on the work presented in (Kirshenbaum 2000), which demonstrated that genetic programming could be used to evolve programs that contained statically scoped local variables. In this paper, we describe how the presence of such variables makes it straightforward to add new classes of operators which make it possible to easily (and in some cases trivially) solve problems previously recognized as difficult as well as to attack whole new classes of problem.

One of the main limitations of traditional genetic programming (Koza 1990, Banzhaf, et al. 1998) is that the solutions that are discoverable are limited to the class of algorithms known as *constant time* or $O(1)$ algorithms, those that make a single pass through the operators, with no loops or recursion. This is an especially severe limitation when the problem naturally presents its inputs in terms of complex data structures such as lists, sets, vectors, or arrays.

When attacking a problem known to require more computational complexity, the typical approach is for the experimenter to hand-craft a “harness” which, for example, evaluates the candidate programs once for each element of an input data sequence and averages the result. The main problem with this is that it requires the experimenter to assert the basic control flow beforehand rather than allowing this to be discovered as an emergent property of the run.

In other work, the higher complexity is encapsulated in special purpose operators which perform specific operations such as summing or finding the mean of a

vector of numbers. While adding such operators expands the class of problem that can be addressed, the class is still bounded by the particular operators chosen.

Another, more significant, move in the direction of emergent computational complexity is demonstrated by the automatically defined iterations, loops, and recursions of (Koza, et al. 1999). We will have more to say about the comparison of their mechanisms and ours in section 12.

In this paper, we present *bounded iteration* operators that allow the discovery of programs with arbitrary polynomial-time complexity. The fact that the iteration is bounded means that we do not have to be concerned that the presence of iteration will result in candidates containing infinite loops, although the fact that the programs can be arbitrarily (and therefore atrociously) complex does have some performance implications, which will be addressed in section 11.

The paper is organized as follows. In section 2 we detail the default parameters used in the experiments reported in the paper. In section 3 we present the first and more general of the iteration operator schemata. In sections 4 and 5 we demonstrate the utility of this operator by showing that it leads to trivial solutions of the even-6-parity problem as well as higher-order and mixed-order parity problems. In sections 6 and 8 we show experimental results for a more difficult Boolean problem, the mixed-order majority problem, solvable with the more general operator, but trivial with a somewhat more specific operator described in section 7. In section 9 we investigate the general applicability of these operators to Boolean function discovery as well as the implications of certain experiment parameters. In section 10 we present results for a significantly more difficult problem: the regression from a sequence of real numbers to the statistical variance of the sequence.

In section 11 we discuss the performance considerations as well as the actual setup used in the experiments described. In section 12 we compare this mechanism with a similar mechanism described in (Koza, et al. 1999). In section 13 we describe other bounded iteration operators that can also be easily added using static local variables, and finally, in section 14 we present our conclusions.

2 EXPERIMENT PARAMETERS

Unless stated otherwise, all experiments described in this paper involved a single population of 5,000 candidate programs evolving in lock-step generations for a maximum of 25 reproduced generations. Experiments consisted of 25 runs. Effort numbers given are those required to achieve 99% confidence of finding a solution.

Candidates were chosen to reproduce using tournament selection with a tournament size of five. For Boolean problems, the fitness was taken to be the number of mismatches. For arithmetic problems, the fitness was the mean of the relative error. 80% of the children were produced by uniform single-point crossover, 8% by straight reproduction, 8% by point mutation (the replacement of the operator of a randomly selected node by another operator type-compatible with the actual arguments and required result type), and 4% by constant drift (Boolean flipping or integer nudging ± 1). Traditional mutation was not used.

The system used in the experiments implements strongly-typed GP (Montana 1995) with hierarchical types. For problems involving Boolean operators, `and`, `or`, `nand`, `nor`, and `not` were used, as well as the Boolean constants `true` and `false`. For problems involving arithmetic operators, addition, subtraction, multiplication, and division (with $x/0=1$) over the real numbers, as well as the integer constants from -10 to 10 . When both sets are used, greater-than and less-than over reals are added as well as a real-valued `if`.

For the initial generation, the trees in the population are generated with a ramped target depth from 4 to 7, with two thirds of the trees containing at least one longest path that extends exactly to the target level and the remaining third bounded by the target level.

For each problem (except even-6-parity) 1000 random fitness cases were generated with no noise. For each run, 100 cases were randomly selected as training cases and (a non-overlapping) 100 were randomly selected as validation cases. Each candidate was presented with the first twenty training cases, and as long as a candidate scored at least 90% hits on presented cases, new cases were presented in batches of 10 until the full complement of 100 training cases was seen. For real-valued problems, a hit is defined as a value within 10^{-6} of the target.

Finally, the tree generation algorithm was biased 4:1 toward selecting non-leaf operators and 5:1 toward selecting variables, when possible. The importance of these parameters will be discussed in section 9.

3 THE ACCUM OPERATOR SCHEMA

The first iteration operator schema¹ we describe is a generic “accumulate”:

```
(accum (elt-var over vector)
      (result-var from initial)
      body)
```

Such a form takes three subtrees. The first is evaluated and returns a vector. The second is evaluated and the value it returns is bound to the result variable as its initial value. Then the third is repeatedly evaluated, once per element of the vector argument. During each evaluation, the element variable is bound to a specific element of the vector, and both variables are available for use within the body. After each evaluation, the result returned by the body is stored as the new value of the result variable, and when the vector is exhausted, the value of the result variable is returned as the value of the form as a whole.

As an example

```
(accum (R1 over V2)
      (R2 from -infinity)
      (if (< R1 R2) R2 R1))
```

selects the maximum value of the vector V2 while

```
(accum (R15 over V1) (R42 from 0)
      (+ R42 R15))
```

computes the sum of V1. Note that these forms can nest, so, for example

```
(/ (accum (R1 over V1) (R2 from 0)
      (+ R2
        (accum (R3 over V2)
              (R4 from 0)
              (if (< R1 R3)
                  (+ R4 1)
                  R4))))
   (accum (R5 over V1) (R6 from 0)
      (+ R6 1)))
```

computes the average number of elements of V2 that are greater than an element of V1. The inner loop counts up the number of elements greater than the current element of the outer loop, the outer loop sums these numbers, and the result is divided by the value of the third loop, which simply computes the length of the V1.

4 PROBLEM 1A: EVEN-6-PARITY

One of the classic problems in machine learning is to discover a model which detects, given a sequence of Boolean arguments whether the sequence contains an even number of true values. In the context of genetic programming, it was first explored in (Koza 1992), which demonstrated that the GP approach is capable of solving the problem, at least for small numbers of arguments. Later works (Koza 1994a, Koza 1994b, Koza et al. 1999) demonstrated that with extensions to the basic GP paradigm, the problem could be solved more efficiently.

Using only the canonical GP paradigm, Koza (1994a) was able to solve the parity problem with up to five arguments. The even-6-parity problem, however, remained out of reach, and an optimistic minimum estimate

¹ Following (Kirshenbaum 2000), `accum` is an operator schema while “the `accum` binding B17 and R12” is an actual operator.

of the effort that would be needed to find such a solution was given as 70,176,000 candidate evaluations.

In the same work, Koza showed that with the addition of automatically-defined functions, and in particular, with a specific architecture of two ADFs each taking five arguments, the problem could be solved with an expected effort of 1,344,000 candidate evaluations.

In (Koza et al. 1999), the authors recast the problem from one involving six input parameters to one taking a single vector of six Boolean values and applied their GPPS 1.0 to it. They showed that with the availability of automatically-defined functions, recursions, and loops, the problem could be solved with an expected effort of 31,250,000 candidate evaluations.²

When we added an operator schema to walk a Boolean vector and return a Boolean value to the operator set, the results were astounding. In 25 runs,³ every single run found a solution by generation 3, and 72% of the runs (18) found a solution *in the initial random generation*. The problem is therefore guessable with 99% confidence in a population of 20,000, and doing so is, in fact, an optimal strategy (equivalent in cost to a single run to generation 3).

Clearly, something interesting is going on here. To see what it is, consider an unsimplified, randomly-generated correct solution:

```
(nor
  (and (accum (B3 over Data)
          (B4 from FALSE)
        (and (or B3 B4)
              (nand B3 B4)))
    (and (and (accum (B5 over Data)
                    (B1 from TRUE)
                  TRUE)
          (nand FALSE TRUE))
        TRUE))
  FALSE)
```

The second loop simply computes the constant value TRUE, and the logic of the rest of the constant operations render the entire form equal to the complement of the first loop, which must, therefore, implement odd-6-parity. If we look at that loop we see that its body computes the odd-2-parity (XOR) function and that by starting with a value of false, the loop as a whole does, indeed, compute odd-6-parity when Data is a six-element vector.

What has happened is that a very difficult search has been decomposed into two extremely simple searches. On the one hand, the basic behavioral structure is a single pass over the data, while on the other hand, the function to be computed at each iteration is just even-2-parity. Genetic programming can discover solutions to these two prob-

lems in parallel and combine them into a complete solution to the more difficult problem.

5 PROBLEM 1B: GENERALIZATIONS

One of the most intriguing things to note about this result is that nowhere does it depend on the fact that the input is a vector of precisely six elements. In fact, the same method will serve to solve even-parity problems of any size. In 25 runs of the even-15-parity problem, results were found in 23 of the runs by generation 2, with 80% (20) having solutions in the initial random generation, rendering the problem guessable in a population of 15,000.⁴ Note that even though there are 32,000 possible cases for this problem, every asserted solution to the 100 training cases was a perfect solution to the 100 validation cases. In fact, every solution that correctly solved the initially-proffered 20 training cases also correctly solved the remainder of the 100 training cases and the 100 validation cases.

That the approach scales should not be a surprise since as far as the candidate solutions are concerned, *it is the same problem*. The only thing that will change is the absolute amount of time it takes to walk the different sized vectors. Indeed, with the input given as a vector, the problem can be cast as a general “even-parity” problem, with fitness cases containing vectors of varying length.

In 25 runs in which the input vectors varied from length seven to length 20, solutions were found in 22 runs with 32% (8) solutions in the initial random generation and the latest found in generation 12. The optimal strategy is to guess in a population of 60,000.

This difference in performance is somewhat perplexing. On the one hand, the increase in complexity is understandable, as there is now more information contained in each fitness case: the length of the vector. Indeed, some of the candidate solutions appear to be considering whether the vector has an even or odd number of elements by including code similar to

```
(accum (B1 over Data)
      (B2 from TRUE)
      (not B2))
```

and some candidates appear to be attempting to solve the problem separately for even- and odd-length fitness cases.

On the other hand, though, the drop in performance is surprising because, since a solution to the earlier problems will be a solution to this one, it should be just as easy to guess an answer. It is therefore surprising that the number of correctly guessed solutions has dropped from 72–80% down to 32%. This is likely simply a statistical anomaly, but more work needs to be done to confirm this.⁵

²They also show the application of GP using architecture-altering operators to parity problems, but we were unable to find a number for the even-6-parity problem. Presumably it would be higher than the figure of 1,789,500 given for the even-5-parity problem.

³Using all 64 cases as training cases.

⁴ The two runs which did not yield solutions will be discussed in section 9.

⁵ In other experiments on the same data (with different parameters), solutions were guessed in 44% (11) and 68% (17) of the 25 runs.

When cast as vector problems, parity problems can also be solved using arithmetic, rather than Boolean, operators. Using the arithmetic operators of addition, subtraction, multiplication, and division, integer constants ranging from -10 to 10 , a real-valued `if`, and an `accum` that ranges over Boolean vectors and returns reals, solutions were found in all 25 runs, the last in generation 10, with 64% (16) of the runs guessing the solution in the initial random generation. The optimal strategy is to guess in a population of size 25,000.

An example of such an arithmetic solution is

```
(< (- (/ 5 9) -4)
  (accum (B7 over Data)
    (R4 from 4)
    (* (- (if B7 -9 1) (/ 6 10))
      (- (accum (B10 over Data)
        (R9 from -7)
        R4)
        (/ R4 R4))))))
```

The loop here starts with a value of four and multiplies by a negative number every time it sees a true value. At the end it checks the sign. While this is not a method for computing parity that would normally occur to a programmer, it is completely correct and one of the most frequently discovered in our runs.

With both Boolean and arithmetic operators added to the function set, solutions were found in 24 of 25 runs, the latest in generation 17 and 36% (9) in the initial random population. The optimal strategy is to guess in a population of size 55,000. Of the 24 successful runs, only one solved the problem primarily as a Boolean function.

6 PROBLEM 2A: MAJORITY

A somewhat more difficult problem is to compute, given a collection of Boolean values, whether or not a majority of the values are true.

In the first experiment using this problem, we ran 25 runs with fitness cases ranging from eight to 20 elements and with arithmetic operators and `accum`. Solutions (all validated) were found in 14 runs, the earliest in generation 5, and the latest in generation 25. The optimal strategy is to run eight runs to generation 16, with a 40% probability of success in each run, for a total of 680,000 evaluations.

7 THE REDUCE OPERATOR SCHEMA TEMPLATE

While `accum` is a very general operator schema and avoids the introduction of experimenter bias, we found that it was often worthwhile to help the system out by adding in iteration schemata that encapsulated behavior that is common in human-written programs. In particular, the higher-order operator schema

```
(reduce<op,init> (var over vec)
  body)
```

is a big help. This form is roughly equivalent to

```
(accum (var over vec)
  (result from init)
  (op result body))
```

except that the result variable is not visible in the body. The actual schemata introduced into an experimental run will instantiate the two parameters. Using this templated schema, one can build many useful control operators. For example, `reduce<+,0>` can be thought of as `sum`, `reduce<*,1>` is `product`, `reduce<min,∞>` finds the minimum term, and `reduce<and,true>` returns true if all terms are true.

Note that unlike the superficially-similar special-purpose operators discussed in section 1, these operators still contain bodies which can be arbitrarily complex functions of their control variables. So, for instance

```
(sum (R1 over V1)
  (* R1 R1))
```

computes the sum of the squares of the elements of a vector. The combining operator is folded into the iteration, but the terms being combined are still subject to the evolutionary process. Note also that, like `accum`, these forms can nest, with inner loops having access to the control variables from enclosing loops. The example from section 3 could be written

```
(/ (sum (R1 over V1)
  (sum (R3 over V2)
    (if (< R1 R3) 1 0)))
  (sum (R5 over V1) 1))
```

Finally, since the term is an arbitrary expression, the values in the vector need not be of the type required by the combining operator, so forms like

```
(all (R1 over V1)
  (> R1 0))
```

can evolve.

8 PROBLEM 2B: MAJORITY WITH SUM

With `accum` replaced by `sum`, the majority problem becomes trivial, with solutions found in the initial population of 23 of 25 runs and the remaining two runs finding the solution in the next generation. The canonical solution is for the term to be n when the element of the vector is true and $-n$ when it is false, for some number n . This will result in a sum with the same sign as n just in case the number of true values exceeds the number of false values.

9 DISCUSSION

9.1 POWER OF THE APPROACH

Before continuing, it is worthwhile to note that, while the addition of iteration operators appears to make the discovery of some otherwise difficult-to-find Boolean

functions trivial, it is not a panacea. Clearly, one cannot hope to discover all $2^{(2^6)} \approx 1.9 \times 10^{19}$ six-argument Boolean functions with an expected effort of a few tens of thousands of evaluations. So there must be something about the particular functions investigated that makes them particularly amenable to this approach.

The most likely answer is that these functions are *totalistic*—the value of the function only depends on the *number* of values that are true and the size of the input, rather than the specific *pattern* of values. For an input vector of length n , there are only 2^{n+1} possible totalistic functions, which is a substantially smaller number. For $n=6$, this is 128, so under this analysis it becomes unsurprising that all (or substantially all) of the possible totalistic functions will be represented in a population of 5,000 functions, most of which will, of course, not be totalistic.

There are two potential problems with this line of reasoning. First, even the number of totalistic functions grows quickly. For $n=20$, there are 2,097,152 of them. Second, one would expect that one of the most common ways of solving a problem would involve counting the number of true values in the input, especially as this can be trivially done. Indeed, the most likely way that a human programmer would solve the majority problem would be something like

```
(> (* 2 (sum (B1 over Data)
             (if B1 1 0)))
     (sum (B2 over Data) 1))
```

but no solution even remotely taking this form was discovered in any run.

It is probably also not wise to jump to the conclusion that *only* totalistic functions are easily solvable. The fact that, for example, the solution to the n -multiplexer problem can be stated simply as

```
(element Data
 (accum (B1 over Addr) (R1 from 0)
 (+ (* 2 R1) (if B1 1 0))))
```

gives us hope that other useful non-totalistic functions can also be easily discovered. Indeed, it may well turn out that the set of “useful” Boolean functions is largely contained within the set of easily discoverable functions, if only because the useful functions are typically those we can simply characterize.

This brings up what can be seen as another advantage of this approach. By altering the search landscape so that useful or interesting functions are simpler to represent and discover than idiosyncratic ones, the useful functions are discovered more quickly, and the tendency to overfit the particular fitness cases is reduced considerably. One of the biggest objections to early work on evolving Boolean functions is that they can be characterized as little more than “exercises in overfitting the data,” as progress is made by repeatedly solving “one more case”. By making the generalized cases so much easier to find, there isn’t time to get stuck in such ruts before either a solution is

found or you have passed the point at which the optimal strategy is to restart the experiment.

9.2 UNSUCCESSFUL RUNS

While almost all of the “trivial” runs above produced solutions, not all of them did, and it is instructive to look at some of those that did not. In the 15-parity experiment, there were two unsuccessful runs. In both of them the best-of-generation individual from the initial random population was a one-point solution. In run 10, `true` solved 13 of its initial 20 fitness cases, while in run 24, `false` solved 12 of its.

In these experiments structural complexity was only taken into account as a tie-breaker, but that appeared to be enough in these cases. This effect can probably be mitigated by either providing a larger number of training cases (thereby making it less likely that they will be biased one way or the other), ensuring that the training cases are drawn in a balanced fashion, or shuffling the training cases each generation or before each candidate evaluation.

9.3 BIAS PARAMETERS

As mentioned in section 2, two parameters are used to bias the initial tree generation toward choosing non-terminals over terminals and choosing variables over other terminals. For all the runs described above, these parameters were left at their canonical values of 4:1 and 5:1 respectively. While we do not present a full sensitivity analysis for these parameters, we did do one even-6-parity experiment with the biases turned off (set to 1:1).

The results are interesting. Only 24 of 25 runs get solutions by generation 25, with the last occurring in generation 20. More importantly, *not a single solution was found in an initial population*. The optimal solution, three runs to generation nine, requires an estimated effort of 150,000 evaluations. Recall that in section 4 we showed that with the bias, all runs were successful by generation three, and 72% found solutions in the initial population, with an estimated effort of 20,000 evaluations. These parameters are clearly important, and more work needs to be done to determine their optimal values.

10 PROBLEM 3: VARIANCE

It could easily be argued based on the above presentation that, while the addition of iteration operators may indeed make some problems trivial to solve, it does not actually contribute much to the paradigm as a whole, as little actual evolution is required, and we may simply be “getting lucky” by having it be so likely that there is either a solution or a candidate near to the solution in the initial population.

To address these concerns, we ran an experiment to discover a solution to a significantly harder problem: regressing from a vector of real numbers to the statistical variance of the values. The training cases ranged in size from five to ten elements and each element was a real number (to four decimal places) between -10 and 10 . The target was computed as⁶

$$\frac{n \sum_i x_i^2 + (\sum_i x_i)^2}{n(n-1)}$$

where n is the number of elements. For this experiment, the arithmetic operators and sum were available as a size operator that takes a vector and returns the length of the vector as an integer. As this is a more complex problem, the maximum number of generations was raised to fifty and each candidate was initially presented with fifty training cases.

We ran 20 runs of this experiment, and solutions were found in half (10) of them⁷. That the problem is difficult is attested by the fact that the earliest solution found was not until generation 16 and the latest was in generation 46. The optimal strategy is to run 11 runs to generation 28, by which point 35% of the runs are successful, for an expected effort of 1,595,000 evaluations. Things look even better when the unsuccessful runs are examined in detail:

Run	Training Fitness	Validation Fitness	Best-of-Run Generation	Initial Validation Fitness ⁸
2	0.036%	0.036%	49	16.7%
4	1.8%	3.0%	50	45.7%
6	5.5%	10.5%	39	24.1%
8	0.60%	2.5%	49	11.2%
9	6.1%	9.7%	50	37.1%
11	1.9%	2.3%	47	16.4%
13	2.2%	3.5%	50	24.5%
15	0.41%	0.48%	49	31.5%
18	0.042%	0.044%	50	12.2%
20	0.16%	0.21%	49	27.4%

⁶ Yes, this is incorrect. The second term should be subtracted rather than added. We discovered while simplifying the solutions that there had been a bug in the fitness case generator. This should not invalidate the results, and, indeed, it is heartening to know that the process found what we actually asked for rather than what we wanted.

⁷ One of the solutions was not perfect, resulting in hits on only 99 of the 100 validation cases. The mean relative error on the validation cases was 0.00036%.

⁸ The fitness of the best candidate of the random generation on the validation cases.

In all of these runs, both the training fitness and the validation fitness improved substantially, and there is no indication that the problem was overfitting the particular training cases. Four of the unsuccessful runs had the remaining error on the validation cases less than 1%, and in two cases it was less than 0.1%, and in nine of the ten runs the best candidate seen was found in the last four generations, implying that the runs were still making progress and that it may simply be that fifty generations is too small of a cut-off for this problem. Only one of the twenty runs, number 6, appeared to have gotten stuck, having gone twelve generations without improvement and having the highest remaining error on both training and validation sets.

One particularly nice solution was found in generation 28 of run 3:

```
(let ((R2 (size Data)))
  (let
    ((R2 (- (let ((R5 (sum (R2 over Data)
                        R2)))
              (+ 0 (/ (* R5 R5) R2)))
           2)))
    (/ (+ (+ 4 0)
          (- (+ R2 (sum (R1 over Data)
                        (* R1 R1)))
              (- (+ 1 2) 1)))
       (- (size Data) 1))))
```

The elements of the vector are summed in the first loop and the let-binding, introduced by the crossover described in (Kirshenbaum 2000), binds this value to R5. The underlined portion of the code uses this variable twice, as well as R2, which is bound to the length of the vector, to compute the square of the sum divided by the length. The second loop computes the sum of the squares of the elements, and the resulting logic adds the two (along with constants that cancel out) and divides the result by $n-1$ to obtain a perfect solution.

11 PERFORMANCE CONSIDERATIONS

The experiments in this paper were run on a 700MHz Hewlett-Packard Omnibook 6000 running Windows 2000 and using GPLab, a flexible framework for genetic programming developed and used for data mining research at Hewlett-Packard Laboratories. The time taken to evaluate the population ranged from approximately 3 seconds/generation to about 55 seconds/generation. Variance runs tended to start at about 15 seconds/generation and grow to about 40 seconds/generation.

The flip side of allowing the computational complexity to become an emergent property is that different candidate programs will try out different strategies, and some of the attempts will almost certainly be horrendously complex.

Since the iteration operators we introduce are bounded in the number of iterations by the length of the vectors they iterate over, we do not have to worry about a candidate going into an infinite loop, but it is quite reasonable to

expect that we will encounter programs that will contain, say, five sequentially nested loops, for a computational complexity of $O(n^5)$. If left to themselves, such candidates would make runs take an unacceptably long time, so we do two things to reduce their impact.

The first thing we do is to impose a *computation budget* on each fitness case evaluation. This is simply a counter that counts down with every operator invocation. If the counter hits zero, the candidate is considered to be infinitely bad. For the experiments described in this paper, the computation budget was set to 500 operations. This number is somewhat arbitrary, and further work needs to be done to learn how to tell if the budget is set correctly. Note that the budget is reset for each fitness case, although it may be worthwhile to experiment with allowing some or all of the unused budget from one fitness case to carry over to the next to allow a candidate to be occasionally computationally complex.

In addition to limiting the complexity, we also reduce it by noting that certain computation is unnecessary. For the `accum` operator schema, we note that if the result variable is not needed in the computation of the value of the body, then the results of all iterations before the last are discarded and the initial value of the result variable also need not be computed. In the absence of side-effects in the body, this means that the loop can be reduced to a single execution of the body, with the element variable bound to the last element of the vector.⁹ If the element variable is also unneeded, then the vector argument also need not be evaluated.

For the `sum` operator schema, a similar optimization notes that if the element variable is not required to compute the value of the term, then the term need only be computed once and the value of the `sum` is simply this value multiplied by the length of the vector.

The result of these optimizations is that many syntactically-present loops are skipped entirely. For instance, in

```
(accum (B1 over V2)
      (R1 from (sum (R2 over V2)
                  (sum (R1 over V3)
                      (* R1 R2))))
      17)
```

`R1` is not needed to determine the value of the outer loop, and so its complex initialization is not performed. Other loops have their complexity simplified. In

```
(sum (R1 over V)
     (sum (R2 over V) R1))
```

the apparent complexity is $O(n^2)$, but the actual complexity is only $O(n)$ since the body of the inner loop does not refer to `R2`.

⁹ If there are side-effects in the body, it must be executed for side-effects for each of the iterations before the last. Since we keep separate track of variables needed for value and side-effect, this can often be considerably cheaper than fully evaluating the body.

The advantage of such optimizations for evolutionary methods is that it means that an otherwise-good program is not penalized because it contains within it a complex calculation whose value would not have contributed to the final value.

In the current implementation, the optimizations are detected at the time the tree is created and stored as annotations on the nodes.

With these optimizations in place, in the variance experiment we found that between 8% and 12% of the candidates in the random generation tended to exceed their budget in the initial random population, essentially removing them from the breeding pool. This number invariably dropped to approximately 1% for the next four generations and then began to rise again, typically remaining between 2% and 15% for the remainder of the run. Interestingly, the one “bad” run was the only one in which the number exhausting their budget ever exceeded 20%, and the number was in fact growing to the end, with 1,102 of the 5,000 candidates doing so. On the other hand, a high number does not apparently necessarily imply problems, as seen by the fact that in the generation before finding a solution, the number exceeding budget was as high as 16.5% and (in that run) had been growing steadily from a low of 1% for 28 generations.

12 COMPARISON WITH AUTOMATICALLY DEFINED ITERATIONS

The closest mechanism in the genetic programming literature to the iteration schemata described in this paper are the *automatically defined iterators* (ADIs) of (Koza et al. 1999). Like iteration schemata, ADIs perform single-pass iteration over data structures executing terms that are arbitrary expressions built up under evolutionary pressure. Also like iteration schemata, any candidate program may contain an arbitrary number of ADIs.

The principal advantages of iteration schemata over ADIs¹⁰ are

1. All ADIs iterate over a single data structure fixed ahead of time by the experimenter. With iteration schemata the data structure¹¹ is an argument to the iterator, computable by an arbitrary expression. This allows the program to evolve to select or even create the structure to use for each loop, with nested loops iterating over different structures.
2. With iteration schemata, loops may syntactically nest, facilitating their movement as a unit during crossover. While one ADI may refer to another,

¹⁰ As defined. The actual implementation used in (Koza et al. 1999) was a simplified version which did not allow parameters or nesting, limiting the complexity of evolvable programs to $O(n)$.

¹¹ Or, potentially, structures, given schemata that walk in lockstep down multiple sequences.

achieving the behavior of nested loops, they are in different parts of the trees and move independently, so it is likely difficult for the importance of the coupling to be preserved and communicated by means of crossover once it is discovered.

3. With iteration schemata, inner loops may see and even modify the control variables from enclosing loops. ADIs have arguments, which allow information to be passed from outer loops to inner loops, but (1) the number of such arguments is fixed at ADI creation time, and (2) the arguments are passed by value, making it difficult for the inner loop to modify the outer.
4. With iteration schemata, the iterators are part of the initial random population, and so there is a range of computational complexity from the very beginning. ADIs are only created by abstraction from trees already present.

The principal advantage of ADIs would appear to be that they are named constructs. This means that an ADI that would be useful more than once will be easier to reuse once discovered. It also means that having named a loop, different candidates can compete to decide what its implementation should be.

13 FUTURE WORK

13.1 OTHER ITERATION OPERATORS

With support for local variables and data structures in GPLab, the addition of new iteration schemata is reasonably trivial. Functional languages such as Scheme and APL provide a rich source of iterators that human programmers have found useful, and it will be interesting to see how these fare in the genetic programming environment. Besides specific operators, we intend to try out (1) operators that only iterate over *part* of a sequence, (2) operators that create new sequences by mapping or selecting values from a sequence, (3) operators that iterate over multiple structures simultaneously, and (4) generalized `reduce` taking its combinator as a (function-valued) argument,

13.2 OTHER PROBLEMS

Genetic programming with iteration schemata appears to work well for some types of problem, but we have not yet characterized what this class of problem is. It will be instructive to try it out on other problems, such as non-totalistic Boolean functions or problems known to require superlinear complexity.

13.3 OPTIMIZATION

The ad hoc optimizations currently implemented are definitely helpful, but a more principled approach could

almost certainly discover other computation that can be avoided.

13.4 UNDERSTANDING PARAMETERS

As pointed out in sections 9.3 and 11, there are several parameters whose values appear to influence the performance of the system, but this influence has not yet been adequately characterized.

14 CONCLUSIONS

The addition of iteration schemata to genetic programming makes it possible for the computational complexity of the solution to be an emergent property just as are the size and particular configuration of operators. Such operators can render otherwise difficult operators trivial and make new classes of problem solvable.

References

- Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert E.; and Francone, Frank D. 1998. *Genetic Programming: An Introduction*. San Francisco, CA: Morgan Kaufmann.
- Kirshenbaum, Evan. 2000. *Genetic Programming with Statically Scoped Local Variables*. In David Whitley, et al., (eds.) *GECCO 2000: Proceedings of the Genetic and Evolutionary Computation Conference, July 10–12, 2000, Las Vegas, Nevada*. San Francisco, CA: Morgan Kaufman Publishers, pp. 459–468.
- Koza, John R. 1990. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University Computer Science Department Technical Report STAN-CS-90-1314. June, 1990.
- Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, MIT Press.
- Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.
- Koza, John R. 1994b. *Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming*. Stanford University Computer Science Department technical report STAN-TR-CS-94-1528. October 21, 1994
- Koza, John R. ; Bennett, Forrest H, III; Andre, David; and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.
- Langdon, W. B. 1996, Using data structures within genetic programming. In John R. Koza, et al., (eds.) *Genetic Programming 1996: Proc. of the First Annual Conference, July 28–31, 1996, Stanford University*. Cambridge, MA: MIT Press, pp. 141–149.
- Montana, David J. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3(2):199–230.