Learning Recursive Programs with Cooperative Coevolution of Genetic Code Mapping and Genotype

Garnett Wilson and Malcolm Heywood
Faculty of Computer Science
Dalhousie University, Halifax
NS, Canada B3H 1W5
gwilson@cs.dal.ca, mheywood@cs.dal.ca

ABSTRACT
The Probabilistic Adaptive Mapping Developmental Genetic Programming (PAM DGP) algorithm that cooperatively coevolves a population of adaptive mappings and associated genotypes is used to learn recursive solutions given a function set consisting of *general* (not implicitly recursive) machine-language instructions. PAM DGP using redundant encodings to model the evolution of the biological genetic code is found to more efficiently learn 2nd and 3rd order recursive Fibonacci functions than related developmental systems and traditional linear GP. PAM DGP using redundant encoding is also demonstrated to produce the semantically highest quality solutions for all three recursive functions considered (Factorial, 2nd and 3rd order Fibonacci). PAM DGP is then shown to have produced such solutions by evolving redundant mappings to select and emphasize appropriate subsets of the function set useful for producing the naturally recursive solutions.

Categories and Subject Descriptors
I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *Heuristic methods.*

General Terms
Algorithms, Performance, Experimentation.

Keywords
Recursion, developmental genetic programming, genetic code, genotype-phenotype mapping, redundant representation, cooperative coevolution.

1. INTRODUCTION
In this work, we address the problem of automatically evolving recursive solutions given a generic (machine language) function set that contains no operators designed to enable recursion or functions that explicitly accept other functions as arguments so as to enable recursion. The work of Huelsbergen appears to be the first to have evolved machine language-based recursion [4] using such a function set. No evolutionary algorithm approaches known to Huelsbergen [4] in 1997 involved the generation of recursive solutions without recursion-enabling operators in the function set. The early works of Koza [8] and Handley [3] relied on introducing specialized recursive operators into their function sets and thus avoided automatically synthesizing recursion. Since that time, researchers have continued to work on evolving recursive solutions. Koza has recently implemented more specialized functions (automatically defined functions, or ADFs) to perform recursion in [9]. Other authors such as Brave [2] and Yu [18] have opted to evolve recursive programs by including the name of the function on which they want to perform recursion in the function set. Similarly, Wong and colleagues [15, 16] have implemented GP systems using logic grammars that include a grammar rule capable of recursion. Whigham [12] has used directed mutation operators to evolve a recursive function, but operators are both problem specific and incorporate knowledge of the solution. Yu and Clack have also presented an interesting technique that uses implicit recursion via higher order functions to avoid explicit recursive calls [17]. In their work, the code content of a recursive loop is passed as an argument to the higher-order function that iteratively applies the code. While avoiding explicit recursion calls, the recursive mechanism is built into the higher order function and is thus not automatically generated. (The use of higher order functions does have the benefit that it implicitly provides a termination mechanism.)

It thus seems that Huelsbergen has been the only researcher focusing on automatic generation of recursion using a generic function set. In contrast, the focus of other researchers has been the issues of measuring good "semantics" in recursive solution program structures and handling non-terminating recursive cases. Huelsbergen's concern (and that of this paper) is to actually discover recursive solutions using a function set that does not directly imply recursion in any way. This paper does address the issue of semantics through a simple metric (correct sequence output length prior to program termination) to indicate semantic "goodness" of solutions. The termination issue of recursive solutions is handled in the usual way—by reaching a maximum number of program steps executed (this method is used in most of the literature on recursion, with the notable exception of [17]).

Central to the approach adopted in this work is the utility of a developmental model of evolution in which function set and genotype are cooperatively evolved under a symbiotic model, Section 2. Section 3 defines the Factorial and Fibonacci (2^{nd} and 3^{rd} order) problems used to benchmark the paradigm. Section 4 provides results, with Section 5 demonstrating explicit contributions made by the coevolutionary model of development. Conclusions and Future Work follow in Section 6.

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

2. EVOLVING GENETIC CODE MAPPINGS WITH GENOTYPES

A number of researchers have advocated the benefits of developmental systems that evolve both a mapping that models the biological genetic code and an associated genotype [1, 5-7, 10, 11, 13, 14]. In particular, evolution of a genetic code can adaptively bias search toward function set symbols useful for the solution, reducing search space and biasing search toward appropriate regions of the space. Banzhaf and Keller initially demonstrated the benefits of separating the genotype and phenotype spaces in an implementation where genotypes were mapped onto phenotypes using redundant encodings for emphasizing certain members of the function and terminal sets over others. The mapping was coupled with a single genotype individual during the entire tournament, and it was mutated, reproduced, or selected along with the genotype that carried it [6, 7]. Their work was followed by an implementation (here called the Standard Adaptive Mapping GP) of Margetts and Jones [10], where separate populations of mappings and genotypes co-evolved in a search for a genotype-mapping pairing that produced an appropriate phenotype. The individuals in the mapping population corresponded to one-to-one (non-redundant) mappings that used the Huffman compression algorithm. O'Neill and Ryan [11] have also introduced an interesting new developmental system that evolves a genetic code along with genotypes, although their mapping models the genetic code using a grammar rather than a codon (genotype subsequence) to symbol mapping. Wilson and Heywood recently introduced a new developmental system – Probabilistic Adaptive Mapping Developmental Genetic Programming (PAM DGP) – that corrected coevolutionary pathologies and search issues of the Standard Adaptive Mapping and demonstrated efficient search of the mapping and genotype spaces using separate populations in a coevolutionary framework [13, 14]. A more developmental redundant (genetic code-based) adaptive mapping scheme was introduced in [14]. We review the PAM DGP algorithm in this section, and describe its application to learning recursive sequences in the remainder of the work.

In the PAM DGP algorithm, there are two separate populations of genotypes and mappings that symbiotically cooperatively coevolve. A probability table is maintained with entries for each combination of genotype and mapping. Entries represent frequencies corresponding to the probability that roulette selection in a steady state tournament will select the genotype-phenotype pairing of individuals dictated by the indices of the table. A small degree of elitism is used in that genotype and mapping individual that are members of the current best genotype-mapping pairing are protected from mutation and crossover. Each tournament round involves the selection of four unique genotype-mapping pairings. Table columns associated with the winning combinations have the winning combination in that column updated using Equation 1 and the remaining combinations in that column updated using Equation 2

$$P(g,m)_{new} = P(g,m)_{old} + \alpha(1 - P(g,m)_{old}) \quad (1)$$

$$P(g,m)_{new} = P(g,m)_{old} - \alpha(P(g,m)_{old}) \quad (2)$$

where g is the genotype index, m is the mapping index, α is the learning rate (or how much emphasis is placed on current values as opposed to previous search), and $P(g,m)$ is the probability in location [g, m] of the table. To prevent premature convergence, the algorithm also features a noise threshold. If the threshold is exceeded by an element in the table following a tournament round, a standard Gaussian probability adjustment in the interval [0, 1] is added to that element and all values in its column are re-normalized so that the column elements sum to unity. An overview of the PAM DGP algorithm is depicted in Figure 1.

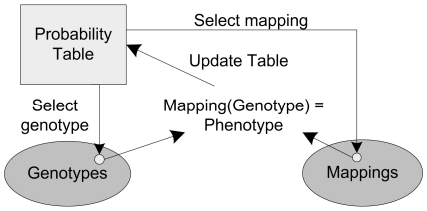

Figure 1. Overview of the PAM DGP algorithm.

Genotypes in PAM DGP are binary strings, with interpretation being instruction-dependant (see next Section). Two types of mappings are benchmarked: Huffman and Redundant. In the Huffman mapping, as advocated by Margetts and Jones [10], mapping individuals consist of s binary sections of 10 bits for each of s function set symbols. All the ones in each 10 bit section are summed and normalized to provide a frequency for each symbol. The function set, associated frequencies, and genotype are provided as arguments to the standard Huffman compression algorithm which returns the symbol-encoding mapping. Given the Redundant mapping, individuals are composed of $b \geq s$ 10-bit binary strings, where b is the minimum number of binary sequences required to represent a function set of s symbols. Each 10 bit mapping section is interpreted as its decimal equivalent, normalized to the range [0…1], and mapped to an ordered function set index by multiplying by s and truncating to an integer value (allowing redundant encoding of symbols). The Huffman and Redundant mappings schemes are shown in Figure 2.

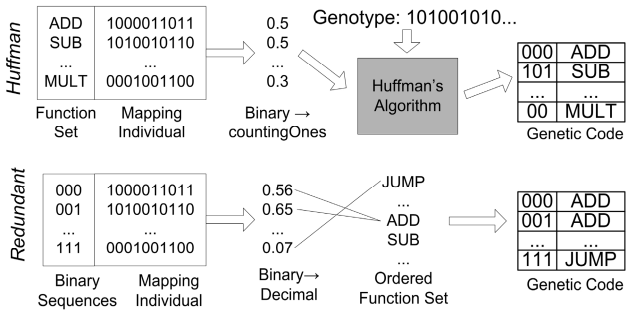

Figure 2. Huffman and Redundant mapping schemes.

3. RECURSIVE PROBLEM DEFINITIONS

In [4], Huelsbergen compares the abilities of random search (Random), genetic programming using solely the crossover operator (XO), exhaustive iterative hill climbing (EIHC), and a hybrid system of his own design that uses the two latter techniques (XO-EIHC) to learn recursive sequences. He found that the simple genetic search (XO) performed the best out of all algorithms for the factorial function, but the more sophisticated EIHC and XO-EIHC algorithms outperformed the other algorithms definitively when evolving solutions to the more difficult Fibonacci series. Sample solutions from the XO-EIHC algorithm were then shown to produce general solutions to the

recursive problems through use of an infinite loop constructed from the branching functions. Our analysis of the recursive functions will examine the ability of coevolved mappings and genotypes to not only more efficiently learn recursive solutions than other competing algorithms tested, but to generate recursive solutions of generality and quality.

Huelsbergen's function set is designed to correspond to a virtual register machine (VRM), and is generic such that it consists only of instructions for primitive register manipulation, conditional and unconditional branching, arithmetic operators, and generation of an output stream. Each individual consists of a program with a number of external registers, and internal state trackers including a program counter (*PC*) and a flag (*Flag*). *Flag* corresponds to the last execution of a comparison instruction (*Cmp(R_{source}, R_{dest})*) that returns one of the values {*greater, less, equal*}; it serves as the basis on which to perform conditional branching. The program counter is an integer that points to the instruction to be currently executed; branching (jump) instructions cause the PC to point to their target, while remaining instructions cause PC to point to their following instruction. The *Output* function places an integer from a register on the output stream *Stdout*; if no output is generated by an individual the *Stdout* stream contains no values. We opt not to use Huelsbergen's NOP function, which has no effect. The function set is summarized in Figure 3 below, where R_s is the source register and R_d is the destination register, *PC* is the program counter, and *N* is the total number of instructions in the program. In total there are 16 different instructions, more than typically employed in a GP function set.

```
Out(Rs) {PC++; Write(Stdout, Rs);}
Neg(Rs) {PC++; Rs = 0 - Rs;}
Mov(Rd, Rs) {PC++; Rd = Rs;}
Set(Rs) {PC++; Rs = 1;}
Clear(Rs) {PC++; Rs = 0;}
Inc(Rs) {PC++; Rs = Rs + 1;}
Dec(Rs) {PC++; Rs = Rs - 1;}
Add(Rd, Rs) {PC++; Rd = Rd + Rs;}
Sub(Rd, Rs) {PC++; Rd = Rd - Rs;}
Mul(Rd, Rs) {PC++; Rd = Rd * Rs;}
Div(Rd, Rs) {PC++; Rd = Rd / Rs;}
Cmp(Rd, Rs) {
      PC++; If (Rd < Rs) Flag = less;
      Else If (Rd > Rs) Flag = greater;
      Else Flag = equal;          }
J(offset) {
       PC = min(max(0, PC + offset), N);}
Jl(offset)* {
      If (Flag == less)
          PC = min(max(0, PC + offset), N);
      Else PC++;              }
```

**Jg(offset) and Je(offset) are equivalent to Jl(offset), only substituting "greater" and "equal" for "less," respectively.*

Figure 3. Machine language-based function set.

In [4], Huelsbergen investigates four integer sequence problems: a sequence of squared numbers, cubed numbers, and the factorial and Fibonacci sequences. We focus on the more difficult and naturally recursive Factorial (*fact*) and Fibonacci (*fib*) sequences, and add the more difficult third order Fibonacci sequence (*fib3*). The function definitions, including base cases, are

$$fact(x) \equiv \begin{cases} 1 & \text{if } x = 0 \\ x \cdot fact(x-1) & \text{otherwise} \end{cases} \quad (3)$$

$$fib(x) \equiv \begin{cases} 1 & \text{if } x = 0 \text{ or } x = 1 \\ fib(x-2) + fib(x-1) & \text{otherwise} \end{cases} \quad (4)$$

$$fib3(x) \equiv \begin{cases} 1 & \text{if } x = 0, x = 1, \text{or } x = 2 \\ fib3(x-3) + fib3(x-2) + fib3(x-1) & \text{otherwise} \end{cases} \quad (5)$$

The fitness evaluation scheme is reproduced from [4] in which the first ten values of the *Stdout* stream, as generated by individuals using the `OUT` instruction, are matched against the ten values of the test case using the following fitness function:

$$fitness(p) \equiv \sum_{i=0}^{l-1} \left| s_i - f(i) \right| \cdot scale(i) \quad (6)$$

where *p* is the program in the form a binary string, *l* is the length of the recursive sequence (10 in these experiments), *f(i)* is the value of the recursive function for integer *i*, and *scale(i)* is defined

$$scale(i) \equiv \begin{cases} S_{\max} & \text{if } f(i) = 0 \\ S_{\max} / f(i) & \text{otherwise} \end{cases} \quad (7)$$

where $S_{max} = \max\{f(0), \ldots, f(l-1)\}$ for the recursive sequence defined by *f*. The sequence $\{s_0, \ldots, s_{l-1}\}$ is the first *l* values of *Stdout*, if the output contains at least *l* values. If it does not, the $j < l$ values *Stdout* contains (that is, $\{s_j, \ldots, s_{l-1}\}$) are set to S_{max}. The fitness function measures summed scaled error (Equation 6), thus lower fitness is better and the objective is fitness = 0.

Since Huelsbergen's results indicated that a larger number of tournament rounds would likely be necessary to generate recursive solutions compared to non-recursive problems, each of our 50 trials consisted of a steady state tournament of 500 000 rounds (4 individuals per round) with a population of 25 genotypes and 25 mappings (50 individuals for Traditional GP). Each genotype consists of 320 bits and 4 subresult registers, and each mapping consists of 160 bits (10 bits for each of 16 required encodings for a function set of size 16). Genotypes and mappings were randomly initialized, with registers initialized to 1. XOR mutation on a (uniform) randomly chosen instruction was used on genotypes, with less disruptive point mutation used on mappings to provide a more stable context against which the genotype could evolve. Both mutations used a rate of 0.5. Crossover occurred between equal-sized segments of individuals at a rate of 0.9. PAM DGP used a conservative learning rate of 0.1 and noise threshold of 0.8 to prevent premature convergence.

As was the case in [4], the program in each genotype individual terminates after running all instructions (PC = *n-1* for *n* instructions with indices 0 to *n-1*) or after the execution of 100 steps. Instructions are decoded from a genotype binary sequence under the guidance of the mapping, either Huffman or Redundant. In each case a number of bits define the instruction type (variable for Huffman, fixed for Redundant), two bits define register references, and five bits define the offset in branch instructions.

In the case of the five offset bits, bit one defines the direction of the jump, and four bits declare the (absolute) offset corresponding to integers over the interval [0, …,15].

4. RESULTS

In this section we compare the efficiency, solution content, and solution quality of Traditional (linear) GP (*Traditional*), the original adaptive mapping of Margetts and Jones (*Standard*), PAM DGP with Huffman encodings (*Huffman*), and PAM DGP with redundant encodings (*Redundant*). Some discussion of the recursive solutions produced by the algorithms covered in this work is in order before proceeding with the analysis of the results. Following Huelsbergen [4], a *solution* is said to have been located when the output stream of an individual's program produces the first ten digits of the required sequence that serve as the test case. Given this definition of *solution*, if the program produces incorrect digits or no digits after getting the initial ten digits correct, it is still technically a solution.

A program is considered a *general solution* if and only if *both* all the members of the sequence of length $l \geq 10$ generated by the output are correct *and* if the program were permitted to run beyond the maximum number of steps (100 in [4] and these experiments), then the program would continue to correctly generate the correct members of the sequence. All solutions (programs that generated the first ten members of the recursive sequence correctly) in these experiments were inspected by hand for generality. In practice, given the function set for these problems, a solution could only be general if it included an appropriate instruction sequence using a reverse branch (jump instruction with negative offset) at the end of the sequence. Furthermore, the repeated sequence would have to include appropriate manipulation of register contents and an output to the *Stdout* stream in its body such that the correct output was produced. The results focus on the ability of the algorithms to produce not just solutions, but *general* solutions.

4.1 The Factorial Function

The first recursive function we examined was the factorial sequence (Equation 3), which is a first order recursive function. That is, each iteration of the recursive function only references the value produced by the previous recursive step. In this respect, the Factorial problem is the simplest of the recursive functions considered. As mentioned earlier, Huelsbergen found that it was most efficiently solved by simple genetic search using only two-point crossover rather than his more sophisticated search techniques [4]. We similarly found that the less complex algorithms generated more solutions: given 50 independent trials, all trials for Traditional, Standard, and Huffman PAM DGP solve the factorial problem, as does 33 trials of Redundant PAM DGP. In the case of the factorial problem, every solution for all algorithms was general. The tournament round when a solution was located for each solution in 50 independent trials is given in Figure 4. Each box indicates the lower quartile, median, and upper quartile values. Notches indicate the 0.95 confidence interval, with points representing outliers to whiskers of 1.5 times the interquartile range. Given the overlap of the notches for the boxplots, there is actually no statistical difference at the 0.95 confidence interval in the median round at which a solution is found for any of the four algorithms. Huelsbergen's hybrid algorithm had a mean of 5.55 x 10^6 evaluations required per solution (over 9 solutions) for the factorial function, while Redundant PAM DGP had a mean of only 2.72 x 10^5 evaluations (4 evalutions per round) required per solution (over 33 solutions).

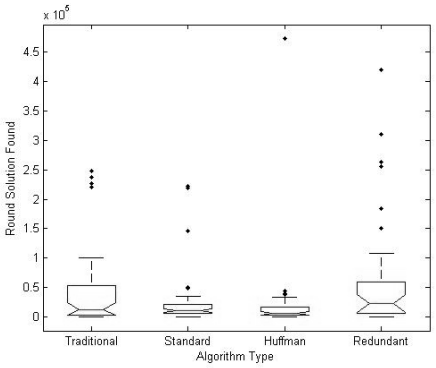

Figure 4. Tournament round at which a solution to the factorial problem was located over 50 independent trials.

While Redundant PAM DGP did not produce as many solutions as the other GP algorithms for this simple recursive function, it outperforms the other algorithms on solution quality. The programs that are of interest are those that have truly discovered recursive solutions, and are thus *general*. One way to measure the quality of general solutions is to examine how many members of the function's sequence the solution can produce before it reaches the program step limit. That is, efficiency of the program at generating the sequence is measured. The efficiency of sequence generation is an important measure: If the body of the loop(s) that produce the sequence contain junk code (introns), program steps will be (at best) wasted if the junk code is innocuous in so far as it does not disrupt the production of the sequence. A loop with innocuous junk code will produce a less lengthy sequence. In fact, introns must be innocuous in general solutions or the solutions would not be able to generate the repeated sequence indefinitely. Efficiency also reflects that the algorithm may be generating multiple outputs per iteration to avoid wasting steps on the jump instructions. Thus, the higher the value of the correct number of sequence members generated, the lower the content of junk code within the program loop(s) and/or the more efficient the loop(s) contents. The number of sequence members produced is thus a simple and informative measure of the quality of general recursive solutions. The number of sequence members produced by the *general* solutions of each algorithm is shown in Figure 5.

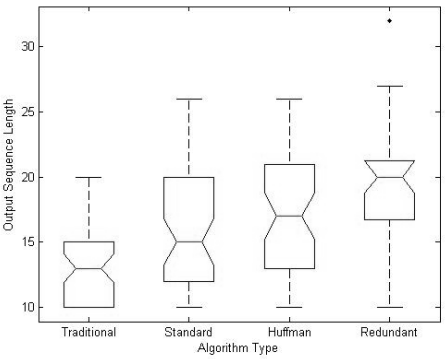

Figure 5. Number of sequence members output by the general solutions to the factorial problem over 50 independent trials.

It is evident that the Redundant PAM DGP algorithm produces the longest sequences among its general solutions at the 0.95 confidence interval compared to all other algorithms tested. Given the aim of discovering a program to produce quality general recursive solutions, rather than sheer quantity of solutions regardless of quality or even generality, Redundant PAM DGP clearly provides the best results on the factorial problem. The best general solution produced the first 32 members of the factorial sequence, and can be seen as the upper outlier for Redundant PAM DGP in Figure 5. The program code for the individual is given in Figure 6. This solution contained no introns. The loop responsible for the indefinite repeated production of the series is italicized. Any instructions that are not reached by the program counter (instructions that are never read by the hypothetical interpreter) are not displayed. Instruction addresses are enumerated on the left of each instruction to help the reader better interpret branching commands.

```
0 INC Reg2; 1 OUT Reg0; 2 OUT Reg1;
3 OUT Reg2; 4 INC Reg1; 5 INC Reg2;
6 MUL Reg1 Reg2; 7 OUT Reg1; 8 INC Reg2
9 MUL Reg1 Reg2; 10 OUT Reg1; 11 INC Reg2;
12 MUL Reg1 Reg2; 13 OUT Reg 1;
14 INC Reg2; 15 MUL Reg1 Reg2; 16 OUT Reg1;
17 J(to 8) using offset -9
```

Figure 6. Program code for the individual that produced the longest factorial sequence. Instructions that constitute the recursive loop are italicized.

In the solution above, instructions 1 to 7 generate, via sequential non-looping instructions, the first four values of the factorial series (1, 1, 2, 6) and thus set up the base case (first value) prior to entering the loop. Instruction 8 begins the loop body that contains three consecutive INC, MUL, OUT sequences that maintain the function's *x* and *x-1* values in registers 2 and 1, respectively. The loop efficiently uses all instructions in its body to output three members of the factorial solution with each iteration. This solution demonstrates the nature of the efficiency and generality of the solutions produced by Redundant PAM DGP, as quantified in Figure 5.

4.2 The Fibonacci Series

We now move to measuring the capability of the algorithms on a more challenging recursive problem: the Fibonacci series as defined in Equation 4. The Fibonacci series uses, by definition, second order recursion. In other words, the current value of the function (with the exception of the base cases, of course) depends on the values of the two previous recursive steps. Huelsbergen found that only his more sophisticated algorithms (EIHC and XO-EIHC) were able to produce solutions to the Fibonacci series; the other algorithms (XO and Random) produced no solutions given a limit of 5 x 10^7 evaluations.

Redundant PAM DGP produces the largest number of solutions (46), with Standard and Huffman PAM DGP producing comparable numbers of solutions (45 and 44, respectively). Traditional GP produced the least number of solutions (42). The boxplot for the tournament rounds at which a solution was located over 50 independent trials appears in Figure 7.

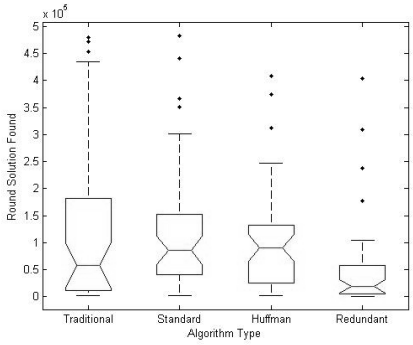

Figure 7. Tournament round at which a solution to the Fibonacci problem was located over 50 independent trials.

Redundant PAM DGP finds the Fibonacci series within fewer rounds than the Standard Adaptive Mapping and Huffman PAM DGP at the 0.95 confidence interval. Redundant PAM DGP also has a lower median than Traditional GP, but due to the large error level in the Traditional GP boxplot, the difference is not statistically significant. The spread of the Redundant PAM DGP boxplot also indicates that it solves the problem more consistently than any other algorithm. Huelsbergen's hybrid algorithm in [4] had a mean of 1.02 x 10^6 evaluations required per solution (over 10 solutions), while Redundant PAM DGP had a mean of only 2.12 x 10^5 evaluations required per solution (over 46 solutions) .

Considering the raw number of general solutions found, all algorithms actually generated comparable results. Redundant PAM DGP had 38 general solutions, Huffman PAM DGP had 42, Standard Adaptive Mapping found 43, and Traditional GP located 41. Despite having the lowest (but competitive) raw number of general solutions, Redundant PAM DGP definitively generated the highest quality (most general) solutions. The sequence length of the solutions generated by each algorithm over 50 independent trials is shown in Figure 8.

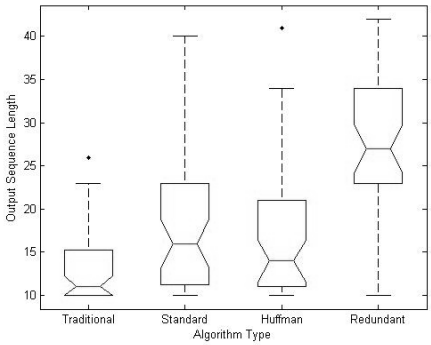

Figure 8. Number of sequence members output by general solutions to the Fibonacci sequence over 50 independent trials.

Redundant PAM DGP, as was the case for the factorial problem, outperforms all other algorithms in terms of efficiency of solutions in generating the series. For the Fibonacci series, however, the degree to which Redundant PAM DGP outperforms the other algorithms is more considerable: The lower end of the interquartile range for Redundant PAM DGP's output length is above the top of the interquartile range for all other algorithms. It was noted that almost all of the solutions found by Traditional GP were general solutions (41 of 42 solutions); however, we can see

in Figure 8 that Traditional GP achieved a median of only 11 sequence members. This means that Traditional GP was typically barely able to generate its minimum output length within its solutions—its solutions are thus not efficient despite their generality. The median performance of the Standard Adaptive Mapping and Huffman PAM DGP were also significantly lower than Redundant PAM DGP, indicating that despite generating more general solutions, their solutions were also not as efficient. The longest solution Redundant PAM DGP generated was 42 members of the Fibonacci series, of which there were two distinct instances. The programs that produced these solutions are given in Figure 9. As before, any instructions of the individual's program that were never reached by the program counter (never interpreted or executed) are not displayed.

```
Solution 1: 0 OUT Reg3; 1 ADD Reg0 Reg3;
2 OUT Reg2; 3 ADD Reg2 Reg0; 4 OUT Reg0;
5 OUT Reg2; 6 ADD Reg0 Reg2;
7 J(to 3) using offset -4

Solution 2: 0 ADD Reg2 Reg1; 1 OUT Reg3;
2 OUT Reg0; 3 ADD Reg1 Reg2; 4 OUT Reg2;
5 OUT Reg1; 6 ADD Reg2 Reg1;
7 J(to 3) using offset -4
```

Figure 9. Program code for the two individuals tied for producing the longest Fibonacci sequence. Instructions that constitute the loop are italicized.

In both of these solutions, there is a similar structure and neither solution includes any intron code in the body of the loop or otherwise. PAM DGP thus produces intron-free solutions to factorial and Fibonacci, whereas Huelsbergen's featured solutions for both functions in [4] contained introns. Both cases represent succinct, general recursive programs for generation of the Fibonacci series. Two of the first three instructions in each of the solutions establish the two required base case values, and the third performs a constructive addition instruction. Instructions 3 to 7 in both solutions comprise the loop that would indefinitely generate the Fibonacci series (in the absence of an upper limit of execution steps). Both loops generate two consecutive members of the series per iteration through a pair of addition and output instructions. Both of these solutions represent very efficient use of the available execution steps.

4.3 The Third Order Fibonacci Series

The final function we consider is the third order Fibonacci series as defined in Equation 5. The equation simply involves summing the results of the previous three values in the series as opposed to the classic, second order, Fibonacci series where the previous two values in the series are summed to determine the current value. Over 50 trials, neither the Standard Adaptive Mapping nor Huffman-encoded PAM DGP produced any solutions. Traditional GP produced only one solution after 460 181 rounds, and Redundant PAM DGP produced 14 solutions with mean time of 174 156 rounds (standard deviation of 24 946) and median of 173 320 rounds. The longest tournament for Redundant PAM DGP even took less time than Traditional GP (325 4440 rounds). Redundant PAM DGP appears to generate more solutions to a recursive problem of this order with a higher degree of reliability than any other algorithm tested. Huelsbergen did not attempt recursion of this order. Only one general solution was found by Redundant PAM DGP and generated 25 members of the third order Fibonacci series within the 100 instruction execution limit. The program expressing that general solution is given in Figure 10. Only instructions that were executed are displayed, and the solution contained no introns.

```
0 OUT Reg0; 1 OUT Reg2; 2 OUT Reg0;
3 ADD Reg3 Reg2; 4 ADD Reg2 Reg0; 5 ADD
Reg0 Reg3; 6 ADD Reg0 Reg3; 7 INC Reg2;
8 OUT Reg2; 9 ADD Reg3 Reg2; 10 OUT Reg3;
11 ADD Reg3 Reg2; 12 J(to 4) using offset-8
```

Figure 10. Program code for the individual that produced the longest 3rd order Fibonacci sequence in a general solution. Instructions that constitute the loop are italicized.

The methodology used by this solution is actually an interesting, less direct approach than simply adding the previous three values to generate the value for the current time step. The first four instructions generate the three required base cases by placing three 1.0s in the sequence and placing an initial value in Register 3. The loop actually causes repeated pairwise output of the values in Register 2 and 3 to produce all values following the base cases. Register 2, in addition to holding values to be output, helps Register 3 to generate the its next sequence member two values in advance. That is, if Register 3 has output sequence member n_t (instruction 10), Register 2 adds the last member it output (n_{t-1}) to Register 3 in instruction 11, and then Register 2 adds the necessary difference to generate n_{t+2} (instruction 9) in the following iteration of the loop just prior to Register 3's output. Register 2 generates its next value following output in instruction 8 by having the correct difference to its next value added to it (instruction 4) from a subresult in Register 0 defined at a previous iteration of the loop (instructions 5 and 6), along with an increment in the current iteration (instruction 7). There is an indirect interwoven relationship among the instructions to create an innovative solution to the harder regression problem.

Table 1 summarizes results from Sections 4.1 to 4.3. It is evident that Redundant PAM DGP generates more solutions for the higher order (2nd and 3rd) recursion problems (total solutions) with less computation effort than other algorithms (mean evaluations). Redundant PAM DGP also generates the most efficient general solutions across all orders of recursion (sequence length).

Table 1. Summary of results over 50 trials for each algorithm.

Algorithm	Factorial	Fibonacci	Fib3
General / Total Solutions, Mean General Sequence Length			
Traditional	50/50, 13.1	41/42, 13.2	0/1, N/A
Standard	50/50, 16.0	43/45, 17.8	0/0, N/A
Huffman	50/50, 16.9	42/44, 17.0	0/0, N/A
Redundant	33/33, 19.5	38/46, 27.9	1/14, 25.0
Mean Evaluations to Solution			
Traditional	1.67×10^5	4.98×10^5	1.84×10^6
Standard	9.59×10^4	4.89×10^5	N/A
Huffman	8.00×10^4	4.23×10^5	N/A
Redundant	2.72×10^5	2.12×10^5	6.97×10^5

5. FUNCTION SET ANALYSIS

It has been demonstrated empirically in Section 4 that Redundant PAM DGP produces the most efficient general solutions over the factorial, Fibonacci, and third order Fibonacci recursive functions out of the four GP algorithms benchmarked in this work. This section investigates whether there was an underlying trimming of the function set to contribute to these quality solutions.

Figure 11 shows the mean distribution of operators within factorial function solutions for Traditional GP and the two mapping types in PAM DGP (Standard is dropped for clarity since it also uses Huffman encoding for function emphasis). It is statistically significant at the 0.95 confidence interval that Redundant PAM DGP avoids move, set, negate, subtract, and divide to a greater degree than all the other algorithms. All of those operators could be disruptive to the production of the factorial series which requires repeated multiplication and addition. Also significant at the 0.95 confidence interval is Redundant PAM DGP's emphasis on addition. Moreover, the five operators most frequently emphasized by Redundant PAM DGP are all explicitly appropriate for generating the factorial sequence (multiplication, increment, addition, jump, and output).

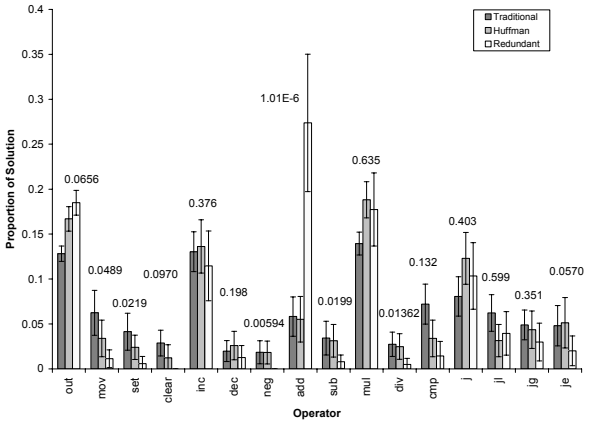

Figure 11. Mean operators as a proportion of total solution for the factorial sequence over 50 trials. Error bars reflect two-tailed t-distribution for the 0.95 confidence interval. P-values correspond to Huffman and Redundant mappings.

The Fibonacci series represented a more difficult (second order) recursive function, and required only three operators in its natural recursive form (Equation 4): addition, output, and jump. The Fibonacci series solutions' allocation of operators over 50 independent trials is shown below in Figure 12. Redundant PAM DGP placed a much higher level of emphasis on addition, output, and increment than the other algorithms (all very useful instructions for generating the Fibonacci series, and significant at the 0.95 confidence interval). The fourth most emphasized operator was the unconditional jump (with other jump variants close behind), allowing Redundant PAM DGP's top four operator choices to include the three required functions for the natural recursive solution of the Fibonacci series. The other algorithms failed to create the degree of preferential function emphasis exhibited by Redundant PAM DGP.

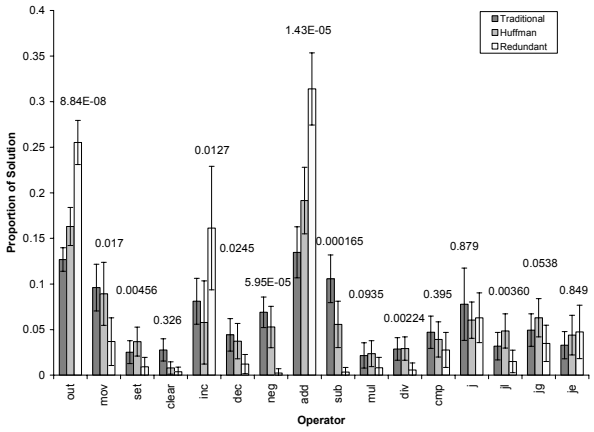

Figure 12. Mean operators as a proportion of total solution for the Fibonacci series over 50 trials. Error bars reflect two-tailed t-distribution for the 0.95 confidence interval. P-values correspond to Huffman and Redundant mappings.

The third order Fibonacci series represents the highest order of recursion investigated in this work. As was the case for the regular (second order) Fibonacci series, the operators used in the natural recursive solution are jump, addition, and output. The allocation of operators over 50 independent trials for the third order Fibonacci series is shown in Figure 13.

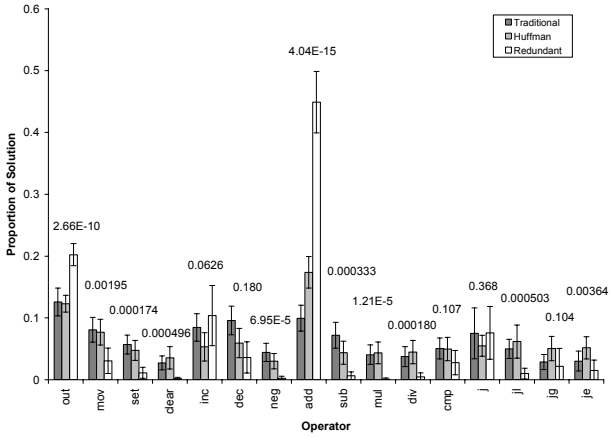

Figure 13. Mean operators as a proportion of total solution for the 3rd order Fibonacci series over 50 trials. Error bars reflect two-tailed t-distribution for the 0.95 confidence interval. P-values correspond to Huffman and Redundant mappings.

Figure 13 clearly shows that Redundant PAM DGP correctly emphasizes the addition and output operators in its solutions to a much greater degree than Traditional GP and Huffman PAM DGP (significant at the 0.99 confidence interval). It also has a healthy emphasis of the unconditional jump function (as well as emphasizing increment, which can also be useful in solution construction and was actually incorporated in Redundant PAM DGP's general solution in the previous section). Traditional GP and Huffman PAM DGP have a comparatively even distribution of functions across their solutions. For this problem, where Redundant PAM DGP produced considerably more solutions than

the other algorithms as shown in the last section, the beneficial effect of appropriate function emphasis is the most salient.

6. CONCLUSIONS AND FUTURE WORK

We have demonstrated that PAM DGP, through cooperative coevolution of redundant genetic code mapping and genotype, produces the most efficient general solutions (longest sequences) over the factorial, Fibonacci, and third order Fibonacci recursive functions among all algorithms considered. Furthermore, its best (most general) solutions for each problem were shown to be entirely intron-free. Given higher order recursion problems (2nd and 3rd order Fibonacci), PAM DGP generated the largest number of solutions and did so more efficiently than any other algorithm tested. Redundant PAM DGP was also shown to evolve its genetic code mappings so as to emphasize the operators useful for general recursive solutions to each function's sequence.

In future work, potentially more complex recursive problems from real world domains could be attempted with function sets consisting of higher level mathematical operators (such as square root, log, *et cetera*). Also, these studies relied on a preset limit of execution steps to terminate recursive loops. In fact, this limit was indirectly used as a means of roughly measuring semantics: semantically better solutions generated more correct values prior to forced termination (and were even found to be intron-free). Thus, in our search for a semantically good solution from a generic function set, we turned concern for termination on it head. Future work could continue to involve measuring semantic goodness in the same way in a first stage of an algorithm, followed by providing the semantically best solution's loop contents to a higher order function (as described by Yu in [17]) with ensured termination.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge the support of an Izaak Walton Killam scholarship (G.W.) and CFI New Opportunities and NSERC research grants (M.H.).

8. REFERENCES

[1] Banzhaf, W. Genotype-Phenotype Mapping and Neutral Variation. In *Parallel Problem Solving from Nature III*, (Jerusalem, Israel, Oct. 9-14, 1994), Springer-Verlag, Berlin, 1994, 322-332.

[2] Brave, S. Evolving recursive programs for tree search. In *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA, 1996, 203-219.

[3] Handley, S. A new class of function sets for solving sequence problems. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, (Stanford, California, July 18-31, 1996), MIT Press, Cambridge, MA, 1996, 301-308.

[4] Huelsbergen, L. Learning Recursive Sequences via Evolution of Machine-Language Programs. In *Genetic Programming 1997: Proceedings of the Second International Conference*, (Stanford, California, July 13-16, 1997), Morgan Kaufman, San Francisco, CA, 1997, 186-194.

[5] Keller, R. and Banzhaf, W. Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes in Linear Phenotypes. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, (Stanford, California, July 18-31, 1996), MIT Press, Cambridge, MA, 1996, 116-122.

[6] Keller, R. and Banzhaf, W. The Evolution of Genetic Code in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, (Orlando, Florida, July 13-17, 1999), Morgan Kaufman, San Francisco, CA, 1999, 1077-1082.

[7] Keller, R. and Banzhaf, W. Evolution of Genetic Code on a Hard Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, (San Francisco, California, July 7-11, 2001), Morgan Kaufman, San Francisco, CA, 2001, 50-56.

[8] Koza, J. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[9] Koza, J. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic, Norwell, MA, 2003.

[10] Margetts, S. and Jones, A. An Adaptive Mapping for Developmental Genetic Programming. In *Proceedings of the Fourth European Conference on Genetic Programming (EuroGP 2001)* (Lake Como, Italy, April 18-20, 2001), Springer Verlag, Berlin, 2001, 97-107.

[11] O'Neill, M. and Ryan, C. Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code. In *Proceedings of the Seventh European Conference on Genetic Programming (EuroGP 2004)*, (Coimbra, Portugal, April 5-7, 2004), Springer, Berlin, 2004, 138-149.

[12] Whigham, P. *Grammatical Bias for Evolutionary Learning*. Ph.D. Thesis, University of New South Wales, Sydney, Australia, 1996.

[13] Wilson, G. and Heywood, M. Probabilistic Adaptive Mapping Developmental Genetic Programming (PAM DGP): A New Developmental Approach. In *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN IX)*, (Reykjavik, Iceland, Sept. 9-13, 2006), Springer-Verlag, Berlin, 2006, 751-760.

[14] Wilson, G. and Heywood, M. Introducing Probabilistic Adaptive Mapping Developmental Genetic Programming with Redundant Mappings. *Genetic Programming and Evolvable Machines (Special Issue on Developmental Systems)*, 2007, to appear.

[15] Wong, M. and Leung, K. Evolving recursive functions for the even-parity problem using genetic programming. In *Advances in Genetic Programming II*, MIT Press, Cambridge, MA, 1996, 222-240.

[16] Wong, M. and Mun, T. Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming. *Genetic Programming and Evolvable Machines, 6*, *4* (Dec. 1995) 421-455.

[17] Yu, T. Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lamda Abstraction. *Genetic Programming and Evolvable Machines, 2, 4* (Dec. 2001) 345-380.

[18] Yu, T., Polymorphism and Genetic Programming. In *Proceedings of the Fourth European Conference on Genetic Programming*, (Lake Como, Italy, April 18-20, 2001), Springer-Verlag, Berlin, 2001, 218-231.