Developmental Plasticity in Linear Genetic Programming

Nicholas Freitag McPhee
Division of Science and Mathematics
U. of Minnesota, Morris; USA
mcphee@morris.umn.edu

Ellery Crane
Division of Science and Mathematics
U. of Minnesota, Morris; USA
ellerycrane@gmail.com

Sara E. Lahr
Division of Science and Mathematics
U. of Minnesota, Morris; USA
lahrx019@morris.umn.edu

Riccardo Poli
School of Computer Science and Electronic Engineering
University of Essex, UK
rpoli@essex.ac.uk

ABSTRACT

Biological organisms exhibit numerous types of plasticity, where they respond both developmentally and behaviorally to environmental factors. In some organisms, for example, environmental conditions can lead to the developmental expression of genes that would otherwise remain dormant, leading to significant phenotypic variation and allowing selection to act on these otherwise "invisible" genes. In contrast to biological plasticity, the vast majority of evolutionary computation systems, including genetic programming, are rigid and can only adapt to very limited external changes. In this paper we extend the N-gram GP system, a recently introduced estimation of distribution algorithm for program evolution, using Incremental Fitness-based Development (IFD), a novel technique which allows for developmental plasticity in the generation of linear-GP style programs. Tests with a large set of problems show that the new system outperforms the original N-gram GP system and is competitive with standard GP. Analysis of the evolved programs indicates that IFD allows for the generation of more complex programs than standard N-gram GP, with the generated programs often containing several separate sequences of instructions that are reused multiple times, often with variations.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Genetic Programming, N-grams, Plasticity, Development

1. INTRODUCTION

Biological organisms exhibit numerous types of plasticity, where they respond both developmentally and behaviorally to environmental factors. Recent work, for example, describes climate (temperature) induced color changes in certain hornworm caterpillars [16]. If the larvae develop in cool temperatures, they grow into dark caterpillars (better to absorb solar energy and stay warm), but if they develop in warmer temperatures, they become green caterpillars (better to hide in the foliage). In both cases there is a single set of genes being expressed in different ways, with environmental conditions driving or enabling the change.

In contrast to biological plasticity, the vast majority of evolutionary computation systems are entirely rigid. An individual has a fixed "chromosome" (e.g., a bit string or a syntax tree), which maps directly to a fixed "phenotype". There is no opportunity for plasticity in the developmental process (if there even is such a process). Also, since the resulting phenotype is typically evaluated in a static fashion, there is usually no opportunity for behavioral plasticity, where its actions could vary in response to its environment. Even evolved solutions to problems with dynamic environments, such as food gathering [4], Robocup Soccer [5], and board games [14], typically have no or little internal state and can only adapt to limited environmental changes.

Developmental plasticity is also important for the development of complex modular organisms. In mammals, for example, the same genotype is used to generate skin cells in one place, while nerve cells are being generated nearby. This modularity is crucial to the evolution of complex organisms and continues to be difficult to obtain in most evolutionary computation systems.

In this paper we explore one possible way of allowing developmental plasticity in Genetic Programming (GP) [4, 11]. In particular, we introduce *Incremental Fitness-based Development (IFD)*, an extension to the recent N-gram GP system [12]. It allows for developmental plasticity in the generation of linear-GP style programs. N-gram GP is an Estimation of Distribution Algorithm (EDA) for the evolution of linear computer programs which has been shown to perform well on a number of symbolic regression problems [12]. We chose it as the starting point for this research because its program-generation process readily lends itself to modeling developmental plasticity. N-gram GP has also shown an ability to generate significantly modular solutions, re-using code segments numerous times in the generation of solutions.

In the IFD system, linear programs are incrementally constructed by repeatedly appending blocks of additional instructions. The blocks are generated using the standard probabilistic N-gram GP system, but blocks are only accepted and added to the program if doing so improves the fitness of the program. IFD could therefore be seen as a form of (parallel) local search in the space of instruction sequences. The process is somewhat similar to certain aspects

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'09, July 8–12, 2009, Montréal, Québec, Canada.
Copyright 2009 ACM 978-1-60558-325-9/09/07 ...$5.00.

of plant growth; root tendrils are sent out in many different directions, but growth is focused on those in areas richer in resources.

The key results from this study are that the addition of IFD is able to improve the performance of N-gram GP on a variety of problems and make N-gram GP more competitive with standard genetic programming. We also find that IFD allows for the generation of substantially more modular and complex programs than are typically created by N-gram GP.

In the next section, we will briefly review the N-gram GP system. In Section 3 we discuss how such a system can be reinterpreted and modified to model biological phenotypic plasticity. In Section 4 we will introduce our IFD mechanism. Section 5 lays out our experimental settings and our test problems, and Section 6 presents a summary of the experimental results. We discuss the results in Section 7 and then wrap up in Section 8.

2. N-GRAM GP

An n-gram is a group of n consecutive items from a longer sequence. For example, `a b`, `b c`, and `c d` are all 2-grams from the sequence `a b c d`, while `a b c` and `b c d` are 3-grams. The items in the sequence can be anything (words, DNA base pairs, phonemes, etc.) although n-grams are most often used for the purpose of modeling the statistical properties of natural language [15, 6]. In particular, an n-gram model assumes that the probability of a particular symbol appearing in a sequence depends only on what appeared before that symbol and its vicinity in the sequence. More formally, we assume that each particular sequence, $x_1, x_2, \ldots$, is an instantiation of a family of stochastic variables $X_1, X_2, \ldots$, where $\Pr\{X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_{i-n+1} = x_{i-n+1}\}$ is independent of i and is sufficient to correctly capture the probability of X_i taking the value x_i in a particular sequence.

N-gram GP uses an n-gram distribution to generate linear computer programs as summarized in Alg. 1. In this context, n-grams are sequences of instructions from the assembly language of a register-based CPU, similar to linear GP [8]. In this research, only 3-grams are used. If $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ is the primitive set, our model of the language can be represented by a three dimensional matrix $M = (m_{l,m,n})$ with elements $m_{l,m,n} = \Pr\{X_i = p_n, X_{i-1} = p_m, X_{i-2} = p_l\}$, where $l, m, n \in \{1, \ldots, N\}$. For any sequence of two instructions (say l and m), the matrix M provides the probabilities for a third instruction. If, for example, $M[l,m,n]$ is twice $M[l,m,n']$, then n is twice as likely to be chosen to follow l, m than n' is. Because this lookup requires that two instructions have already been selected, in [12] two additional matrices were derived to select the first two instructions in a new program. We instead start every program with two no operation (`NO_OP`) instructions. This obviates the need for the derived matrices, and our findings suggest that the performance of the system is not altered significantly.

Assuming that the matrix M is available, one can use it to generate sequences of instructions with the same statistical properties as the language the n-gram model represents. The process starts by sampling the primitive set $\mathcal{P}$ using probability distribution $M[\texttt{NO\_OP}, \texttt{NO\_OP}]$ to obtain the first instruction l of a program. The second instruction is drawn by using row $M[\texttt{NO\_OP}, l]$ of the probability matrix. The third and all subsequent instructions are then drawn using the appropriate entries from M. There are various strategies to determine how the process terminates. In [12], a program length distribution, P_L, was explicitly represented and updated. When creating a new program, a length was first drawn from the length distribution, and a program with that length was then generated (see lines 6-7 of Alg. 2). In this paper we use a similar strategy for program construction, but we intermix program generation from an N-gram model with fitness evaluation (see Section 4). Also, in some cases we terminate the construction of programs before reaching the length drawn from the length distribution.

Algorithm 1 Program generation algorithm for N-gram GP.

```
genProgram( $M$, $\ell$ )
1: Initialize the first two instructions, $x_1$ and $x_2$, as NO_OPs.
2: for $i = 3$ to $\ell$ do
3:     Select $x_i$ based on $M_{x_{i-2},x_{i-1}}$ {via roulette wheel} {$M_{x_{i-2},x_{i-1}}$
       is the $x_{i-2}$-th row in the $x_{i-1}$-th page of $M$}
4: end for
5: return $x_1, x_2, \cdots, x_\ell$
```

Algorithm 2 N-gram GP main loop.

```
N-gram-GP
1: Initialize the distributions $P_L$ and $M$
2: repeat
3:     for $i = 1 \ldots$ popsize do
4:         With probability 1/popsize, pop[$i$]= best individual
           found so far
5:         With probability 1 − 1/popsize:
6:             Select program length, $\ell > 2$ based on the
               probabilities stored in $P_L$ {Perform a roulette
               wheel selection on the entries of $P_L$}
7:             pop[$i$] = mutate(genProgram($M$,$\ell$))
8:     end for
9:     elite = truncationSelection( pop )
10:    updateProbabilities( $P_L$, $M$, elite )
11: until Solution found or max number of instruction evaluations
    reached
12: return best individual found
```

So far we have assumed that the distributions M and P_L were available. N-gram GP constructs such models using a standard EDA approach with minor modifications as shown in Alg. 2. The process starts with the uniform initialization of the distributions P_L and M. That is, all entries of P_L are initialized to $1/\ell_{max}$, where ℓ_{max} represents the maximum program size we are interested in, and all the entries of M are initialized to $1/N^3$, where N is the size of the primitive set. N-gram GP then constructs a new population, most of which is created by sampling from the distributions M and P_L. However, occasionally (on average once per generation) the best individual seen so far in the run is reintroduced to guarantee stability in the estimated distribution. To maintain diversity and ensure that the search continues even after many entries of M converge to 0, we follow the standard EDA practice of performing point mutation at a (low) per-locus rate (p_m) on the individuals returned by the `genProgram` routine in Alg. 1. The population then undergoes truncation selection, where the best individuals are stored in a set, `elite`, which is used to update the program distribution; in this work we used the top 1/5 of the population for `elite`.

The update of the probabilities in P_L and M is performed using the additive update rules shown in Alg. 3. Note that the arrays are not explicitly zeroed before they are updated. In this way the model used to produce individuals at one particular generation can depend also on successful individuals discovered in previous generations in the run. How much the current `elite` influences the model depends on two learning rates: the per pool learning rate η_M and the length learning rate η_L. With large enough values for η_L and η_M, P_L and M are almost entirely determined by the current elite, making them effectively independent of the previous history of the run.

3. N-GRAM GP AND PHENOTYPIC PLASTICITY

One of the primary biological phenomena inspiring this research is the notion of developmental plasticity, where the phenotypic de-

Algorithm 3 Learning in N-gram GP.

```
updateProbabilities( $P_L$, $M$, elite )
1: for all $x$ in elite do
2:     $\ell = \texttt{length( x )}$
3:     $P_{L,\ell} = P_{L,\ell} + \eta_L/(\ell_{max} * \texttt{length(elite)})$
4:     for $j = 3 \ldots \ell$ do
5:         $M_{x_{j-2},x_{j-1},x_j} = M_{x_{j-2},x_{j-1},x_j} + \eta_M/(N^3 * \texttt{length(elite)})$
6:     end for
7: end for
8: $M = M/\sum_{l,m,n} M_{l,m,n}$
9: $P_L = P_L/\sum_l P_{L,l}$.
```

velopment of an organism can change as a result of changes in its environment [13]. Because an organism's phenotype is a product of both its genotype and environment, the same genotype can lead to the development of two organisms with distinct phenotypes.

While some GP systems have a developmental component (e.g., cellular encoding and its extensions [1, 3], grammatical evolution [9], and DTAG3P+ [2]), most GP systems have no developmental process which can be altered to incorporate plasticity. The developmental process in N-gram GP, however, makes it possible to introduce a form of developmental plasticity, where the N-gram probability matrices can be thought of as a genotype which can be used to develop a program (the phenotype) in a way that is sensitive to environmental feedback.

What form, however, will this environmental feedback take? In traditional GP and N-gram GP, the environment has no impact on the development of programs since fitness is only applied after the programs are fully formed. As we will show in Section 4, however, the developmental process in N-gram GP permits us to model developmental plasticity by altering the phenotype (instructions) of a program in response to stimuli from its environment. This allows a single genotype (N-gram GP's probability matrices) to generate a variety of different phenotypes (programs) in a manner that is responsive to environmental conditions.

4. INCREMENTAL FITNESS-BASED DEVELOPMENT

In incremental fitness-based development, a program is constructed through the standard process of drawing instructions from the N-gram matrix, as detailed in Alg. 1. Rather than drawing instructions until the appropriate program length has been reached, however, IFD generates only a short sequence of instructions (referred to as *blocks*) before evaluating the program. The fitness of the program is noted, and another block of instructions is drawn from the N-gram matrix using the last instructions in the previous block to seed the process of selecting instructions from the matrix. At this point, the program is evaluated again. If the fitness of the program was improved by the addition of the new block, the process described above continues. If, however, the fitness of the program did not improve or was actually made worse, the newly generated block of instructions is discarded and another candidate block is generated. The IFD operator, therefore, only allows blocks of instructions to be added to a program if they cause an improvement in the fitness of the program.[1] Rather than continuing the process of block generation ad infinitum until the program reaches the appropriate length, each program is allotted a total number of instruction evaluations for its construction. The generation process ends when either the program reaches the length chosen from the

[1]This can be seen as a form of greedy local search. We explored less greedy probabilistic block acceptance schemes, but we found that this greedy approach yielded the best or, in a very few cases, very near the best performance. Consequently all work reported here will use this greedy approach.

Algorithm 4 Program generation algorithm using IFD

```
genProgramWithIFD( $M$, $P_L$)
1: Select max program length, $\ell_{prog} > 2$ based on the probabilities
   stored in the distribution $P_L$ {Perform a roulette wheel selection
   on the entries of $P_L$}
2: Initialise $x_1, x_2, \cdots, x_{\ell_B}$ using genProgram($M$, $\ell_B$)
3: repeat
4:     $\ell = \texttt{length()}$ {Current length of the program}
5:     repeat
6:         $\beta = \texttt{genProgram}(M, \ell_B + 2)$ using $x_{\ell-1}$, $x_\ell$ as initial
           instructions instead of NO_OPs.
7:     until Evaluation of $\beta_3, \cdots, \beta_{\ell_B+2}$ improves overall fitness of
       program or number of instruction evaluations exceeds
       $\ell_{prog} * ICM$
8:     if $\beta$ increases overall fitness of program then
9:         Add $\beta_3, \beta_4, \cdots, \beta_{\ell_B+2}$ to program as $x_{\ell+1}, x_{\ell+2}, \cdots, x_{\ell+\ell_B}$
10:    end if
11: until $\ell$ reaches $\ell_{prog}$ or number of instruction evaluations
    exceeds $\ell_{prog} * ICM$
12: return $x_1, x_2, \cdots, x_\ell$
```

length distribution or when it uses all of the instruction evaluations alloted to it. See Alg. 4 for more details. [2]

The IFD operator introduces several new parameters to the system and, as is frequently the case, parameter choice can have a significant impact on the performance of the system. One new parameter is block size, ℓ_B. Because IFD requires the program's fitness to improve with the addition of each new block, block size can be very important. If the block size is too small, generating the right sequence of instructions to see an improvement in fitness becomes difficult. If it is too large, the potential for random noise interfering with an otherwise beneficial block grows too great. We tested a variety of block sizes in this work, and the results show that the optimal block size is highly problem dependent.

Another new parameter is the total number of instruction evaluations to use when generating each program. Once again, the ideal number here is likely tied directly to the problem being explored. A small number of instruction evaluations reduces the amount of local search and biases the programs towards shorter lengths which may be insufficient to solve the problem. Too many instruction evaluations, on the other hand, wastes processing time on programs that are impossible to improve. As shown in Alg. 4, we chose the maximum number of instruction evaluations to use when generating a program by multiplying the length chosen for the program by an instruction count multiplier (ICM). We tested a variety of values for ICM in this work and found that the value of ICM had a limited, but sometimes significant, impact on performance.

5. EXPERIMENTAL SETUP

5.1 Target functions, test points, and fitness

To assess the impact of IFD on N-gram GP, we tested it against standard GP and standard N-gram GP on 11 different symbolic regression problems. We did 100 independent runs of each system on each of the test problems. In this section we'll present the test problems we used, as well as the parameter settings for the various systems tested here.

The 11 test problems are listed in Table 1. For each of the polynomial targets we used 21 test points: $-1, -0.9, \ldots, 0.9, 1$. For the

[2]The IFD process does not require the entire program to be evaluated each time a block is created. The state of the virtual machine is saved after executing the previous successful block and reused each time a new block is created. Hence, evaluating a block only requires executing the new instructions starting from the saved state, rather than the complete evaluation of the whole program.

Table 1: **The target functions in our tests.**

Label	Function
P1	$x+x^2+x^3+x^4+x^5$
P2	$-x-2x^2+x^3$
P3	$1.009+1.419x+x^2$
P4	$6+x^2+3x^3+8x^5$
P5	6
P6	$6+x^2$
P7	$6+x^2+3x^3$
P8	$8x^5$
P9	$3x^3+8x^5$
P10	$x^2+3x^3+8x^5$
Sine	$sin(x)$

Sine target we used 63 test points: $0, 0.1, \ldots, 6.1, 6.2$. A program's fitness was the sum of its absolute error on each of the test points.

A program was defined as being successful if its average error across all the relevant test points was < 0.05. Thus the acceptable error for the polynomial targets was 1.05, while the acceptable error for the Sine problem was 3.15.

5.2 Parameter settings

See Table 2 for a listing of all the key parameters, most of which are either standard in the field or discussed above in the descriptions of N-gram GP and IFD. Table 3 lists all the linear GP instructions used in the N-gram GP runs, both with and without IFD. These instructions act on two general purpose registers, `R0` and `R1`, and a fixed input register `RIN`. The output is always read from `R0` at the end of the program's execution. Memory-with-memory [7] was available to both N-gram GP systems, with its use and strength determined by the assignment hardness parameter γ.

To enable better comparisons between the N-gram systems and standard GP, no constants, random or fixed, were used in any of these runs. One of the weaknesses of N-gram GP is that it doesn't scale well to large pools of constants since they necessitate numerous additional instructions. This causes the probability matrix M to quickly grow unmanageably large since its size is cubic in the number of instructions. There's no doubt that having appropriate constants would make several of these problems much easier to solve; standard GP, for example, does much better on several of these problems if it is allowed access to a large pool of constants.

Because IFD potentially evaluates many more instructions than those that are finally included in the generated program, we used instruction evaluations instead of generations as the "clock" used to limit the length of runs. All tested systems were limited to 50,000,000 evaluations.[3]

We used TinyGP as described in [11] and distributed on-line [10] with the handful of changes necessary to make the various systems more comparable:[4] (a) we allowed an average error of 0.05 per test case as discussed in Section 5.1; (b) we modified it to count instruction evaluations, and we limited runs to 50,000,000 evaluations; (c) we used a population size of 10,000; and (d) we modified protected division of $x\%y$ to return 1 if y was near 0 instead of returning x as in the original TinyGP.[5]

[3]This wasn't checked until after an entire pool or population was generated and evaluated, however, so the total number of instruction evaluations was typically slightly greater than 50,000,000.

[4]There have been a handful of bug fixes since the publication of [11]; the on-line version of the code [10] has these fixes applied.

[5]Since none of these systems used any constants, the ability to "construct" 1 by dividing by 0 was important to the success; without this change TinyGP with no constants did much worse.

Table 2: The parameter values used in these tests.

Shared	
Acceptable error per test case	0.05
Number of independent runs	100
Max instruction executions	50,000,000
N-gram GP & IFD	
Max program length (ℓ_{max})	100
Number of registers	2
Elite proportion	20%
Point mutation rate (p_m)	$0.25/\ell$
Pool sizes	100, 750, 1000
Per pool learning rate (η_M)	10, 10^3, 10^6
Length learning rate (η_L)	0.001, 0.01, 1
Assignment hardness (γ)	0.2, 0.3, 0.5, 1
IFD	
Block size (ℓ_B)	2, 3, 5, 7, 10
Instruction count multiplier (ICM)	1, 2, 3, 5, 7
TinyGP	
Population size	10,000
Initialization method	Grow
Max initial size	10,000
Max initial depth	5
Control strategy	Steady state
Selection	Tournament, size 2
Functions	$+, -, *, \%$
Per node mutation rate	0.05
Crossover probability	0.9

Table 3: The instructions used in the N-gram GP runs, both with and without IFD.

`NO_OP`	`R0 = R0 + R1`	`R0 = R0 - R1`
`R0 = RIN`	`R1 = R0 + R1`	`R1 = R0 - R1`
`R1 = RIN`	`R0 = R0 * R1`	`R0 = R0 % R1`
`Swap R0 R1`	`R1 = R0 * R1`	`R1 = R0 % R1`

No particular effort was made to tune any remaining TinyGP parameters such as crossover and mutation rates, although limited testing suggested that changing these parameters didn't have a major impact on the results.

6. RESULTS

6.1 Success vs. method and problem

Table 4 shows the number of successes, runs with a final total average error of less than 0.05 per test case, out of 100 runs using each of the three tested systems: Standard Genetic Programming (using TinyGP), N-gram GP, and N-gram GP with IFD. In each case the number reported was the best success rate from all the tested parameter settings for that problem. While a large number of parameter sets were tested, there is no guarantee that the reported results are globally optimal. These results do strongly suggest, however, that N-gram GP is still competitive with standard GP; it clearly does worse on P3, P6, P7, and P10, but does better on P9. IFD never hurts success rates when compared to standard N-gram GP, and frequently improves them. It also makes N-gram GP more competitive with standard GP; IFD ties on P3 (where non-

Table 4: Number of successful runs (out of 100 independent runs) using standard GP, standard N-gram GP, and N-gram GP with IFD for each of the 14 test problems used in this paper. All differences are statistically significant (with $p < 0.05$ using a pairwise test of proportions) except the difference between 0 and 1 for P7 and the difference between 0 and 1 for Sine.

	TinyGP	N-gram	IFD
P1	100	100	100
P2	100	100	100
P3	100	61	100
P4	0	0	0
P5	100	100	100
P6	100	10	94
P7	85	0	1
P8	100	100	100
P9	22	51	100
P10	100	7	80
Sine	0	1	63

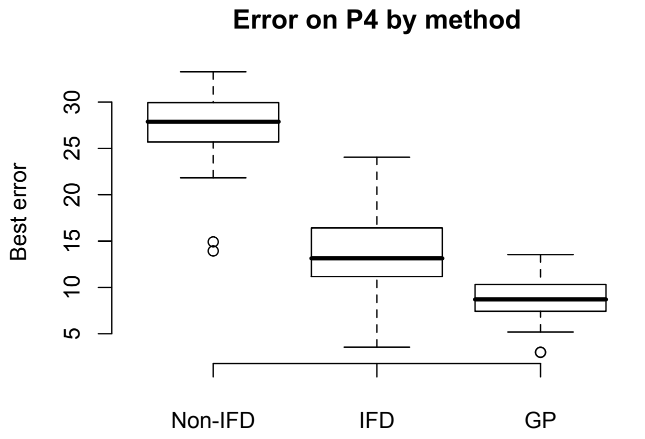

Figure 1: Best errors (over 100 runs) for P4 for each method. The most successful configurations for N-gram GP and IFD on P4 were used to generate those runs.

IFD lost), IFD is much more competitive on P6 and P10, and IFD strongly dominates on P9 and Sine.

Success rates don't provide a meaningful comparison on problems like P4, where no solutions were ever found. Looking at the best errors across 100 runs (Fig. 1), however, shows that (at least on P4), IFD yields significantly better errors than standard N-gram GP. The best errors for IFD are also much closer to the errors from standard GP; the variance with IFD is considerably greater, but the best (lowest) error values with IFD are nearly the same as those for standard GP. We also got similar error results (not reported here) for several other hard problems with success rates of 0.

6.2 Impact of IFD parameters

We did runs using a variety of values for the two new IFD parameters: block size and instruction count multiplier (ICM). Most problems were highly robust to changes in the block size and either succeeded (or failed) regardless of the value of the block size. There were four problems, however, where block size had a clear impact: P1, P2, P3, and Sine (see Fig. 2). P1 and P3 clearly preferred smaller block sizes, having the highest success rates with block size 2. P2, on the other hand, strongly preferred larger block

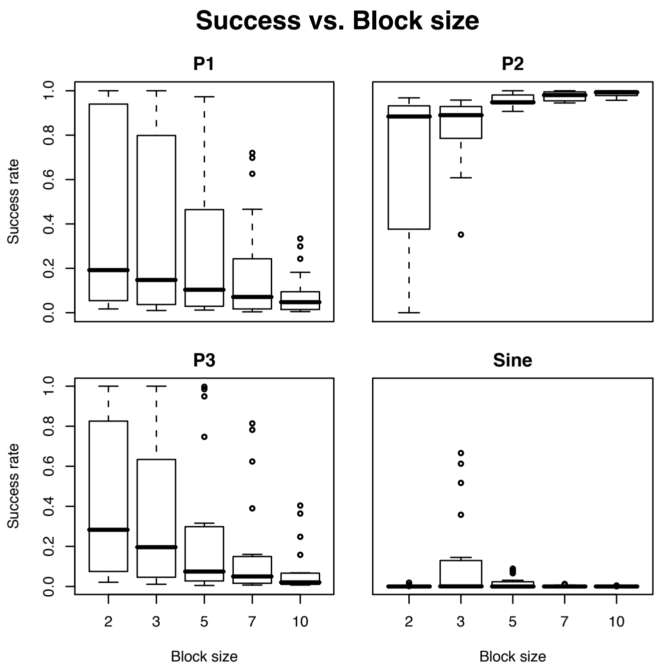

Figure 2: Success rates for P1, P2, P3, and Sine for different values of block size.

sizes, having the best results with block size 10. While block size appears to have a monotonic effect on success for P1, P2, and P3, for the Sine problem, block size 3 is clearly much better than any of the other block sizes we tried.[6]

The impact of ICM was generally smaller and less obvious, with most problems showing no clear relationship between ICM values and success rates or errors. The two exceptions were P1 and P3, both of which showed a strong preference for larger values of ICM. Fig. 3 shows the relationship for P3; P1 had very similar results.

It is unclear at this point why success for certain problems is dependent on the value of block size and ICM. Both of these parameters affect the amount and nature of IFD's local search when

[6]For the other problems, the only other bias of note was for P9 and P10, where their success rates were much lower for block size 2 than for larger block sizes.

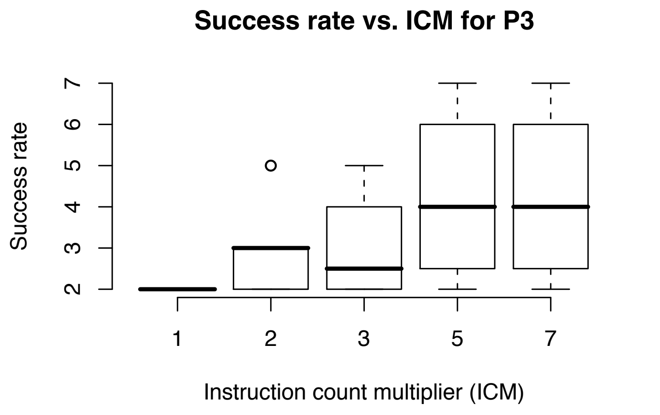

Figure 3: Success rates for P3 for different values of instruction count multiplier (ICM).

generating programs, and it's possible that the impact of these parameters on success rates reflects the varying role of local search on different problems.

7. DISCUSSION

7.1 Program efficiency with IFD

In contrast with standard N-gram GP, Incremental Fitness Development forces a program to make consistent improvements in fitness over the course of its execution. When examining programs generated by standard N-gram GP, we noticed that they would often 'reset' themselves several times during the course of execution, usually by overwriting both registers with new data. This is not only inefficient, but possibly detrimental because all n-grams in a program are used to update the M matrix, not just those which contribute to the final solution. As a result, everything prior to the final 'reset' within a program is just adding noise to the matrix. IFD, on the other hand, doesn't allow such resets unless they include instructions that can, on their own, generate a better overall fitness for the program than had been obtained before the reset. By preventing these resets, IFD ensures that only the n-grams relevant to the generation of the final program are written back into the matrix, eliminating the aforementioned noise.

Further, this likely enables the length distribution, P_L, to generate more meaningful lengths; when a program resets, its length is effectively only those instructions which came after the reset. In standard N-Gram GP, the length of the whole program is written into P_L, regardless of resets, which may skew it towards deceptively large program lengths. With IFD, however, the entire program is significant, so the length that is used to update P_L represents a "real" length.

7.2 IFD, looping, and modularity

N-gram GP frequently generates modular solutions, repeating significant sequences of instructions in the generation of solutions. Often these sequences are repeated verbatim, while in other cases the repetitions have small differences. A potential problem, however, is that N-gram GP solutions are often constrained to one or two sequences representing high probability loops in the N-gram matrix M. This work shows, however, that IFD allows for more complex modularity in the generated solutions, often involving multiple paths through M, some of which are repeated (perhaps with variation).

Fig. 4 illustrates the generation of a successful program for P3 that was developed using standard N-gram GP.[7] During the evolutionary process the probabilities within the matrix became increasingly fixed, leading to a small number of very high probability paths. The figure shows a few instances when lower probability paths are taken, but in general the program is defined by a very rigid set of high probability transitions.

The generation of a successful program with IFD typically has a more modular composition as seen in Fig. 6. This graph has a number of largely independent loops and more variety in the choice of paths. Lower probability options are taken in several cases (for example, the red path out of node M, the light blue path out of node EE), and we generally found that IFD matrices were much less converged than standard N-gram GP matrices. Presumably this is because IFD's guided local search allows the discovery and utilization of less likely paths.

[7]The runs illustrated in Figs. 4 and 6 were chosen because the corresponding graphs were the smallest and most manageable among those we generated. The IFD graphs were consistently larger and more complex than the standard N-gram GP graphs, however. One IFD run, for example, generated a graph with roughly six times as many edges as in Fig. 6.

```
NO_OP, NO_OP, DIV0:1:0, DIV0:1:0, R_IN0, R_IN0, SUB0:1:1, DIV0:1:0,
SUB0:1:0, SUB0:1:1, SWAP0:1, ADD0:1:0, R_IN0, ADD0:1:0, SWAP0:1,
MULT0:1:1, ADD0:1:1, SUB0:1:1, MULT0:1:1, MULT0:1:1, DIV0:1:1, SWAP0:1,
R_IN1, ADD0:1:0, ADD0:1:0, R_IN0, ADD0:1:0, SWAP0:1, MULT0:1:1,
ADD0:1:1, SUB0:1:1, MULT0:1:1, MULT0:1:1, DIV0:1:1, SWAP0:1, R_IN1,
ADD0:1:0, ADD0:1:0, R_IN0, ADD0:1:0, SWAP0:1, MULT0:1:1, ADD0:1:1,
DIV0:1:0, MULT0:1:1, MULT0:1:1, DIV0:1:1, SWAP0:1, R_IN1, ADD0:1:0,
ADD0:1:1, R_IN0, ADD0:1:0, SWAP0:1, SUB0:1:1, ADD0:1:1
```

Figure 5: Successful program on P3 generated using standard N-gram GP (i.e., without IFD). For labels ADD, SUB, MULT, and DIV, OpX:Y:Z is the instruction that applies operator Op to the value of register X and Y and places the result in register Z. R_INX reads the input into register X. SWAPX:Y swaps the values of registers X and Y. Assignment hardness $\gamma = 0.3$ was used in this run.

```
NO_OP, NO_OP, ADD0:1:1, DIV0:1:0, R_IN0, DIV0:1:0, SWAP0:1, DIV0:1:1,
SWAP0:1, DIV0:1:0, MULT0:1:0, R_IN1, ADD0:1:1, SWAP0:1, DIV0:1:1,
DIV0:1:0, MULT0:1:0, R_IN1, ADD0:1:1, SWAP0:1, DIV0:1:1, DIV0:1:1,
DIV0:1:1, DIV0:1:0, MULT0:1:0, R_IN1, ADD0:1:1, SWAP0:1, DIV0:1:1,
DIV0:1:0, MULT0:1:0, R_IN1, ADD0:1:1, SWAP0:1, DIV0:1:1, SWAP0:1, R_IN1,
R_IN1, ADD0:1:0, SUB0:1:1, MULT0:1:1, ADD0:1:0, ADD0:1:1, MULT0:1:0,
DIV0:1:0, R_IN1, SWAP0:1, ADD0:1:0, MULT0:1:0, ADD0:1:0, ADD0:1:1,
MULT0:1:0, DIV0:1:0, R_IN1, SWAP0:1, ADD0:1:0, MULT0:1:0, ADD0:1:1,
SWAP0:1, DIV0:1:1, SWAP0:1, DIV0:1:1, SWAP0:1
```

Figure 7: Successful program on P3 generated using IFD. See Fig. 5 caption for program interpretation. Assignment hardness $\gamma = 0.3$ was used in this run.

7.3 "Knocking out" IFD

To better understand the role of a gene, biologists often "knock out" (disable) the gene and see what impact this has on the development, phenotype, and behavior of the organism in question. We borrowed a variant of that idea to better understand the impact of IFD on both the evolutionary process and on the generation of programs from probability matrices M and P_L. We took the matrices from the end of a run **using IFD**, for example, and used them to generate 1,000 additional programs **without using IFD**, to see whether matrices evolved when using IFD could generate successful individuals without IFD. The Sine problem was used for these comparisons since IFD had a significant impact in that problem.

The results of generating programs both with and without IFD from matrices evolved using IFD can be seen in Fig. 8. The individuals with IFD are significantly more successful than those without it.[8] Because the matrices were developed incrementally, they would contain a larger number of low-probability paths, and presumably IFD's local search mechanism was important in navigating the numerous options.

Fig. 9 shows the results of generating programs both with and without IFD from matrices evolved using standard N-gram GP. Here the performance is much closer, but there is a slight but statistically significant advantage for IFD in this case.[9] Even though the matrices were created without IFD, the fitness of IFD individuals were equivalent to, *if not better than*, the non-IFD individuals. Despite the fact that the probability matrices were highly converged, IFD was still able to use local search to find low-probability paths leading to better individuals.

In both instances, IFD was able to succeed despite the predispositions of the matrices. The fact that the IFD individuals were able to make up for the weakness in the development of the matrices suggests a flexibility that is not available to standard N-gram GP.

[8]$p < 2.2 \times 10^{-16}$ using a Wilcoxon rank sum test.
[9]$p = 0.01014$ using a Wilcoxon rank sum test.

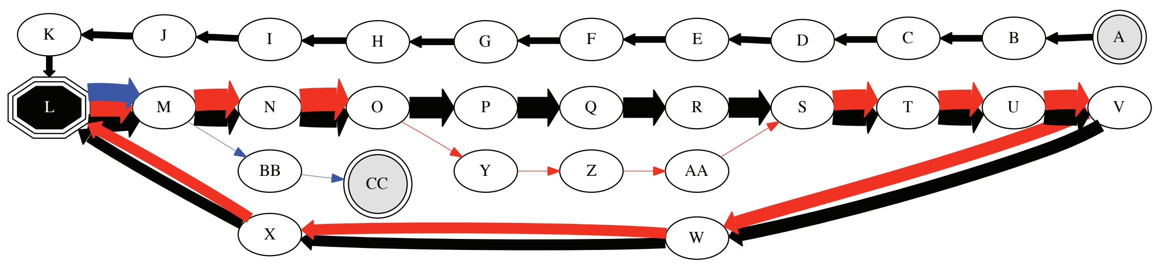

Figure 4: Illustration of the generation of a successful program (shown in Fig. 5) on P3 using standard N-gram GP. Each node represents an N-gram triple of instructions, indicating the generation of a single instruction (the third instruction in the triple). The thickness of arrows indicates the frequency of that transition in the elite pool at the end of the run. Thick arrows indicate widely used transitions coming from high probability entries in the N-gram matrix *M*. Thin lines represent lower probability transitions. The generation of this program begins with the upper right node A (marked with the double circle). After an initial sequence of 11 instructions (the top row of nodes, A-K), the generation process effectively does two large loops (the black and then the red arrows) generating nearly the same sequence of instructions twice in a row. The exception is that the red path makes a small low-probability detour through nodes Y, Z, and AA instead of taking the higher probability route through P, Q, and R. After the second pass through the loop (the red arrows), the loop is started again (the blue arrows), but a low-probability detour is taken almost immediately and the generation process terminates shortly thereafter (at node CC, also double circled).

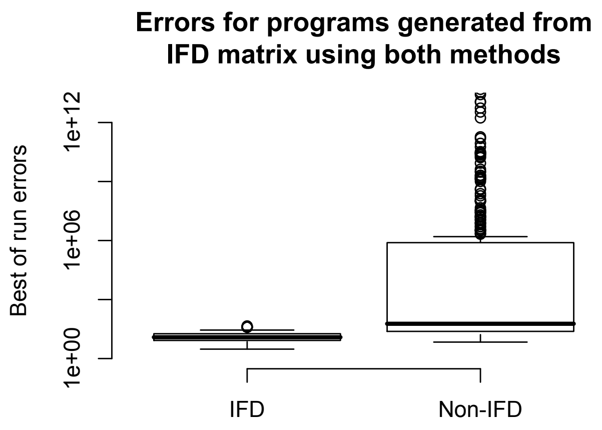

Figure 8: Distribution of errors for pools of 1,000 programs generated both with and without IFD using matrices *M* and P_L from a successful run on the Sine problem using IFD. Programs generated using IFD were clearly more successful than those generated without IFD. There are in fact outliers in the non-IFD data that extend up past 10^{306}, so it's obviously possible for the N-gram system without IFD to generate extremely unfit individuals from a matrix evolved using IFD.

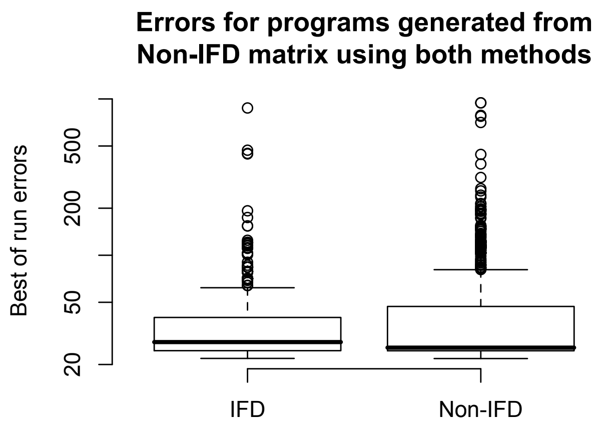

Figure 9: Distribution of errors for pools of 1,950 programs generated both with and without IFD using matrices *M* and P_L from a successful run on the Sine problem using IFD. Programs generated using IFD were clearly competitive with those generated without IFD, so it's obviously possible for IFD to generate successful individuals even using probably matrices evolved with standard N-gram GP without IFD.

8. CONCLUSIONS

Plasticity in biology allows the same genetic material to be expressed in a variety of ways depending on environmental conditions. Standard GP doesn't implement any form of plasticity in its development and evaluation mechanisms. N-gram GP provides a developmental process, but since this process is not impacted by external conditions, there is no developmental plasticity. In this paper we introduce an extension of the N-gram system, *Incremental Fitness Development* (IFD). This extension displays the plasticity that these other GP systems lack by incrementally constructing programs, by adding a series of instruction blocks, using environmental feedback to guide the construction process.

In this study we tested IFD on a number of symbolic regression problems and found that IFD was frequently better than the standard N-gram system, and never worse. The results of comparing standard GP to IFD were more mixed; on two of the tested problems IFD had higher success rates, while standard GP had higher success rates on two other problems.

Success rates in several problems depended on block size when using IFD. In the Sine case, one specific value of block size yielded significantly better results than any neighboring values. This indicates that for at least some problems, block size is important, and tuning could be difficult. One approach would be to have block sizes chosen probabilistically similar to how lengths are chosen in Alg. 2 and then allow the system to dynamically discover successful block sizes for the problem at hand.

N-gram solutions tend to depend on a small number of high probability loops (e.g., Fig. 4), while IFD allows more complex and modular solutions involving a larger number of lower probability structures (e.g., Fig. 6). A key factor in enabling the development of these more complex structures appears to be the ability

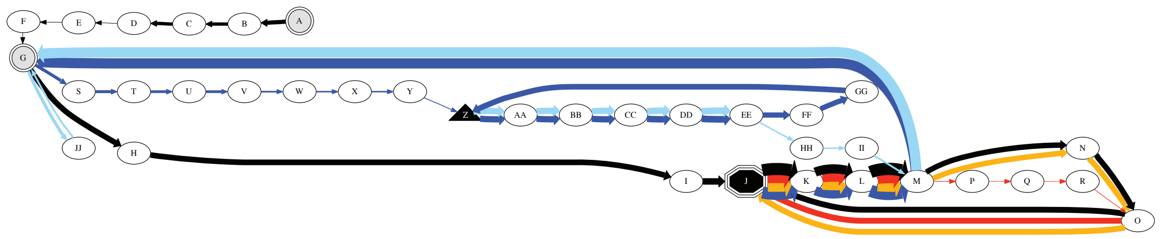

Figure 6: Illustration of the generation of a successful program (Fig. 7) on P3 using IFD. Each node represents an N-gram triple of instructions, indicating the generation of a single instruction (the third instruction in the triple). The thickness of arrows indicates the frequency of that transition in the elite pool at the end of the run. Thick arrows indicate widely used transitions coming from high probability entries in the N-gram matrix *M*. Thin lines represent lower probability transitions. Where standard N-gram tends to rely on a small number of very high probability loops (Fig. 4), IFD allows the generation process to be both more complex and more modular. Here, along with unrepeated sequences like A-J, the process contains two distinct loops (the loop of black, red, and orange arrows in the bottom right and the blue loop along the top), each of which has repetitions and exceptions. The first and third times through the bottom right loop (black and orange) are identical and so generate the same sequence of instructions. The second time (red), however, shares the first four instructions (J-M) but takes a lower probability route (P-R) instead of going through the single node (N). The fourth time into that loop (dark blue) takes a high probability alternative (at node M), entering into the second major phase of the generation process. This second phase also has an introductory sequence (nodes G and S-Z), followed by a repeated section (Z-EE) and loop (EE-Z). The generation process then ends with another exit sequence (EE-II, M, G, JJ, and back to G).

of IFD's local search to use a broader range of lower probability paths through the matrix *M*. Standard N-gram solutions tend to be similar to Fig. 4, relying on one or two heavily used loops, where IFD solutions typically have more loops and more sequences of instructions that are only used once.

Here IFD's local search is a key source of plasticity, allowing the same "genetic material" (the probabilities in the N-gram matrices) to generate a variety of different programs in a manner driven by environmental feedback. One indication of the value of this plasticity is the ability of IFD to improve performance even when applied to N-gram matrices from runs that did not use IFD in the evolutionary process (e.g., Fig. 9). This suggests that IFD might be particularly valuable in dynamic environments, an area we hope to explore in the future.

In summary, we have taken a general biological concept (plasticity) as the motivation for an extension (IFD) of the N-gram GP system, and found the extension to improve success rates and allow for the development of more complex and more modular programs. Future areas for exploration include the possibility of dynamic, probabilistically chosen block sizes for problems like Sine that are dependent on block sizes, and the possible application of IFD in dynamic environments.

9. ACKNOWLEDGEMENTS

The initial development and data collection for this project was supported by EPSRC grant EP/G000484/1. Lahr was supported by the Morris Academic Partners program at the University of Minnesota, Morris. The Brain-Computer Interfaces group at the University of Essex allowed us to do the bulk of our empirical runs on their cluster. Bill Langdon provided valuable ideas and feedback, especially in the early stages of the project. We gratefully acknowledge all these important sources of support.

10. REFERENCES

[1] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, *et al*, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[2] T.-H. Hoang, D. Essam, R. I. B. McKay, and X. H. Nguyen. Building on success in genetic programming: Adaptive variation & developmental evaluation. In *Proceedings of the 2007 International Symposium on Intelligent Computation and Applications (ISICA)*, Wuhan, China, Sept. 21-23 2007. China University of Geosciences Press.

[3] J. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. The design of analog circuits by means of genetic programming. In P. Bentley, editor, *Evolutionary Design by Computers*, chapter 16, pages 365–385. Morgan Kaufmann, San Francisco, USA, 1999.

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

[5] S. Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In J. R. Koza, *et al*, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[6] C. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, 1999.

[7] N. F. McPhee and R. Poli. Memory with memory: Soft assignment in genetic programming. In M. Keijzer, *et al*, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1235–1242, Atlanta, GA, USA, 2008.

[8] P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[9] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[10] R. Poli. Tinygp, 2009. Available at `http://cswww.essex.ac.uk/staff/rpoli/TinyGP/`.

[11] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[12] R. Poli and N. McPhee. A linear estimation-of-distribution GP system. In M. O'Neill, *et al*, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 206–217, Naples, 26-28 Mar. 2008. Springer.

[13] T. D. Price, A. QvarnstrŽm, and D. E. Irwin. The role of phenotypic plasticity in driving genetic evolution. *Proceedings of the Royal Society of Biological Sciences*, 270(1523):1433–1440, July 2003.

[14] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel. Designing an evolutionary strategizing machine for game playing and beyond. *Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(4):583–593, 2007.

[15] C. Y. Suen. *n*-gram statistics for natural language understanding and text processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):164–172, Apr. 1979.

[16] Y. Suzuki and F. H. Nijhout. Evolution of a polyphenism by genetic accommodation. *Science*, 311(5761):650–652, February 2006.