A Linear Estimation-of-Distribution GP System

Riccardo Poli[1] and Nicholas F. McPhee[2]

[1] Department of Computing and Electronic Systems, University of Essex, UK
rpoli@essex.ac.uk
[2] Division of Science and Mathematics, University of Minnesota, Morris, USA
mcphee@morris.umn.edu

Abstract. We present N-gram GP, an estimation of distribution algorithm for the evolution of linear computer programs. The algorithm learns and samples a joint probability distribution of triplets of instructions (or 3-grams) at the same time as it is learning and sampling a program length distribution. We have tested N-gram GP on symbolic regressions problems where the target function is a polynomial of up to degree 12 and lawn-mower problems with lawn sizes of up to 12×12. Results show that the algorithm is effective and scales better on these problems than either linear GP or simple stochastic hill-climbing.

1 Introduction

Estimation of distribution algorithms (EDAs) (see [4] for a review) are powerful population-based searchers where the variation operations traditionally implemented via crossover and mutation in evolutionary algorithms are replaced by the process of sampling from a distribution. For example, PBIL [2] and UMDA [6] assume that the distribution is a product of univariate marginals. EDAs modify the distribution generation after generation, using information obtained from the more fit individuals in the population. The objective of these changes is to increase the probability of generating individuals with high fitness. Different algorithms use different models for the probability distribution that controls the sampling.

There have been several applications of EDA-style probabilistic model-based evolution to tree-based GP; we review several below and provide a full literature review in [8]. In PIPE [11], the first EDA-type GP system, the population is replaced by a hierarchy of probability tables organised into a tree, where each table represents the probability that a particular instruction be at that particular location in a newly generated program tree. eCGP [12] assumes that all trees will be created by sampling within a maximal tree and partitions the nodes in this tree into groups. The co-occurrence of the nodes in each group is modelled by a full joint distribution table. EDP [16] uses a conditional probability table which can, in principle, capture more complex dependencies between nodes. As in eCGP and PIPE, programs are tree-like and are assumed to always fit within an ideal maximal full tree. A hybrid between EDP and GP was proposed in [17].

Various other systems have been proposed which combine the use of grammars and probabilities.[1] For example, [10] used a stochastic context free grammar to generate

[1] In fact, there is a fundamental equivalence between probabilistic grammars and other probabilistic approaches (see [14] for a detailed explanation).

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 206–217, 2008.

program trees. The probability of application of each rewrite rule was adapted using a standard EDA approach so as to increase the probability of application of successful rules. Slightly more general is the approach taken in PEEL (Program Evolution with Explicit Learning) [13], where a probabilistic L-system is used with rewrite rules that are depth- *and* location-dependent and have associated probabilities of application which are adapted by an Ant Colony Optimisation (ACO) algorithm. Another programming system based on a probabilistic grammar optimised via ant systems is ant-TAG [1].

While there is a significant amount of prior work, there has been no application of EDA-style ideas to linear GP. This paper starts filling this gap by proposing *N-gram GP*, an EDA-type GP system capable of evolving machine-language-type programs [7]. A further novelty is our use of n-grams, borrowed from the field of natural language processing, to represent regularities in the language necessary to solve a problem.

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 of a variety of types, including words from natural language, base pairs in a DNA fragment, and phonemes in a speech recording. Very often n-grams are used for the purpose of modelling the statistical properties of sequences, particularly natural language [15,9,5]. 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 in its vicinity in the sequence. More formally, if we imagine that a particular sequence $x_1, x_2, \ldots$ is an instantiation of a family of stochastic variables $X_1, X_2, \ldots$, the assumption is that, for any k-gram, $\Pr\{X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$ is independent of i and is sufficient to correctly capture the probability of X_i taking the value x_i in a particular sequence.

In this work we will use an n-gram distribution to generate linear computer programs, that is sequences of instructions from the assembly language of a register-based CPU, as in linear GP [7]. We will limit our attention to the case of 3-grams. So, if $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ is the primitive set, our model of the language can then be represented by a matrix $M^{(3)} = (m_{l,m,n})$ with elements $m_{l,m,n} = \Pr\{X_i = p_n, X_{i-1} = p_m, X_{i-2} = p_l\}$, where the indices l, m, and n range over the set $\{1, \ldots, N\}$. From this matrix we derive two further matrices, $M^{(2)} = (m_{l,m})$ and $M^{(1)} = (m_l)$, with elements $m_{l,m} = \sum_n m_{l,m,n}$ and $m_l = \sum_m m_{l,m}$, respectively. So, the elements of these matrices are marginals of the distribution $\Pr\{X_i = p_n, X_{i-1} = p_m, X_{i-2} = p_l\}$.

Note that, in general, by definition $\Pr\{X_i = x_i, X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$ $= \Pr\{X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\} \times \Pr\{X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$. So, the elements of $M^{(3)}$ are proportional to $\Pr\{X_i = p_n | X_{i-1} = p_m, X_{i-2} = p_l\}$, the elements of $M^{(2)}$ are proportional to $\Pr\{X_{i-1} = p_m | X_{i-2} = p_l\}$, and, finally, the elements of $M^{(1)}$ are proportional to the priors $\Pr\{X_{i-2} = p_l\}$.

Assuming that the matrices $M^{(1)}$, $M^{(2)}$ and $M^{(3)}$ are available, one can then use them to generate sequences of instructions with the same statistical properties as the language the n-gram model is supposed to represent. The process starts by sampling the primitive set $\mathcal{P}$ using probability distribution $M^{(1)}$ to obtain the first instruction of a program.

The second instruction is drawn by using the distribution prescribed by the row of $M^{(2)}$ corresponding to the first primitive. The third and all subsequent instructions are then drawn using the appropriate entries from $M^{(3)}$. (See Alg. 1 for more details.)

How does this process terminate? If one of the instructions in the primitive set $\mathcal{P}$ is an `EXIT` instruction of some type, the construction process can naturally terminate when such an instruction is drawn. The program length distribution is then determined by the probability of drawing `EXIT` instructions. An alternative, which gives more control on program length, is to dispense with the `EXIT` instruction and instead explicitly represent and update a program length distribution. In this case a length is first drawn from the length distribution, and programs are then grown up to the prescribed length. In this paper we take the latter approach.

One option when explicitly representing the length would be to add an extra dimension to our N-gram model, making it possible to learn different 3-gram distributions for different length classes. However, this is problematic. The matrix $M^{(3)}$ is of size N^3, where N is the size of the primitive set. Extending the model to 4 dimensions implies an increase in the number of parameters to be learnt by two or more orders of magnitude, which is a distinct disadvantage. So, we decided in this initial research to focus on a system where program length is independent from program primitives. That is, we assume that the joint distribution of 3-grams and length is a product of the form $M^{(3)} \times P_L$, where P_L is the length distribution. The construction of new programs, therefore, proceeds as shown in Alg. 1.

So far we have assumed that the distributions $M^{(3)}$ and P_L were somehow available. Now we look at how we construct such models. We do this using a standard EDA approach with minor modifications as shown in Alg. 2. We start by initialising the distributions P_L and $M^{(3)}$. If we have no prior information on the problem to be solved (as is assumed in all experiments reported in the paper), the most natural initialisation is the uniform distribution. If ℓ_{max} represents the maximum program size we are interested in, then all entries of P_L are initialised to $1/\ell_{max}$. Similarly, all the entries of $M^{(3)}$ are initialised to $1/N^3$. Then, after projecting $M^{(3)}$ to obtain its marginals, we proceed to construct a new population. This is, for the most part, created by sampling from our model (the distribution $M^{(3)} \times P_L$). 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 $M^{(3)}$ converge to 0, we follow standard EDA practice and perform point mutation (at a low per-locus rate) on the individuals returned by the `genProgram` routine. Like in many EDAs, the population then undergoes a step of truncation selection, where the best individuals are stored in a set `elite` which is then used to update the program distribution; in this work we used the top 1/5 of the population for `elite`.

The update of the distribution is performed independently for P_L and $M^{(3)}$ using an additive update rule as 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, η_M and η_L. If desired, these can be made arbitrarily big. When $\eta_M \gg 1$

Algorithm 1. Program generation algorithm for N-gram GP.

genProgram($M^{(1)}$**,** $M^{(2)}$**,** $M^{(3)}$**,** P_L**)**

1: Select program length, $\ell > 0$ based on the probabilities stored in the distribution P_L {Perform a roulette wheel selection on the entries of P_L}
2: Select the first instruction, x_1, based on the probabilities stored in $M^{(1)}$ {via roulette wheel}
3: If $\ell > 1$ select the second instruction, x_2, based on the probabilities stored in the x_1-th row of $M^{(2)}$, which we indicate with $M^{(2)}_{x_1}$ {Again this is done via roulette wheel selection on $M^{(2)}_{x_1}$}
4: **for** $i = 3$ **to** ℓ **do**
5: Select x_i based on $M^{(3)}_{x_{i-2},x_{i-1}}$ {$M^{(3)}_{x_{i-2},x_{i-1}}$ is the x_{i-2}-th row in the x_{i-1}-th page of $M^{(3)}$}
6: **end for**
7: **return** $x_1, x_2, \cdots, x_\ell$

Algorithm 2. N-gram GP main loop.

N-gram-GP

1: Initialise the distributions P_L and $M^{(3)}$
2: **repeat**
3: Compute marginals of $M^{(3)}$ to obtain $M^{(1)}$ and $M^{(2)}$
4: **for** $i = 1 \ldots$ `popsize` **do**
5: With probability $1/$`popsize`, `pop[`i`]` = best individual found so far
6: With probability $1-1/$`popsize`, `pop[`i`]=mutate(genProgram(`$M^{(1)}, M^{(2)}, M^{(3)}, P_L$`))`
7: **end for**
8: `elite = truncationSelection( pop )`
9: `updateProbabilities(` P_L, $M^{(3)}$, `elite )`
10: **until** Solution found or max number of iterations exhausted
11: **return** best individual found

and $\eta_L \gg 1$, P_L and $M^{(3)}$ are almost entirely determined by the current elite, effectively independent of the previous history of the run.

3 Experimental Results

3.1 Problems and Primitive Sets

We used two families of test problems: `Polynomial` and `Lawn-Mower`. `Polynomial` is a symbolic regression problem where the objective is to evolve a function which fits a polynomial of the form $x+x^2+\cdots+x^d$, where d is the degree of the polynomial, and x is in the range $[-1,1]$. In particular we considered degrees $d = 5,\ldots,12$, and we sampled the polynomials at the 21 equally spaced points $x \in \{-1.0,-0.9,\ldots,0.9,1.0\}$. Fitness (to be minimised) was the sum of the absolute differences between target polynomial and the output produced by the program under evaluation over these 21 fitness cases. Polynomials of this type have been widely used as benchmark problems in the GP literature. However, we are unaware of any experiments with degrees as high as the ones we consider here.

Algorithm 3. Learning in N-gram GP.

updateProbabilities(P_L, $M^{(3)}$, `elite` **)**

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

Table 1. Primitive sets used in our experiments (% represents protected division, which returns its first argument if the second argument is zero)

	Polynomial		Lawn
ID	PlusTimesSwapR1R2	AllOpsSwapR1R2	Mower
0	R1 = RIN	R1 = RIN	Mow
1	R2 = RIN	R2 = RIN	Left
2	R1 = R1 + R2	R1 = R1 + R2	Right
3	R2 = R1 + R2	R2 = R1 + R2	
4	R1 = R1 * R2	R1 = R1 * R2	
5	R2 = R1 * R2	R2 = R1 * R2	
6	Swap R1 R2	Swap R1 R2	
7		R1 = R1 - R2	
8		R2 = R1 - R2	
9		R1 = R1 % R2	
10		R2 = R1 % R2	

For these problems we considered two primitive sets: `PlusTimesSwapR1R2`, that is particularly suitable for the solution of this problem, and `AllOpsSwapR1R2` which is a superset of `PlusTimesSwapR1R2` containing two spurious primitives. These primitive sets are detailed in the first two columns of Table 1. The instructions refer to three registers: the input register `RIN` which is loaded with the value of x before a fitness case is evaluated and the two registers `R1` and `R2` which can be used for numerical calculations. `R1` and `R2` are initialised to x and 0, respectively. The output of the program is read from `R1` at the end of its execution.

`Lawn-Mower` is a variant of the classical Lawn Mower problem introduced by Koza in [3]. As in the original version of the problem, we are given a square lawn made up of grass tiles. In particular, we considered lawns of size $d \times d$ with $d = 5, \ldots, 12$. The objective is to evolve a program which allows a robotic lawnmower to mow all the grass. In our version of the problem, at each time step the robot can only perform one of three actions (see Table 1): move forward one step and mow the tile it lands on (`Mow`), turn left by 90 degrees (`Left`) or turn right by 90 degrees (`Right`). In the original problem fitness (to be minimised) was measured by the number of tiles left non-mowed at the

end of the execution of a program; whether or not the lawnmower kept visiting other tiles after finishing the job was not considered a relevant part of the problem. This makes the problem rather easy to solve and uninteresting if the GP system is allowed to grow large enough programs. So, to make the problem more difficult, we implemented two further constraints. First, we limited the number of instructions allowed in a program to a small multiple of the number of tiles available in the lawn (more precisely $4 \times d^2$). Second, we required the lawnmower to be energy efficient, so we added corrections to the fitness function which encouraged the evolution of rapidly-mowing programs and programs that stop immediately after having cut the last grass patch:

$$\text{fitness} = \begin{cases} 0.0001 \times \text{extraMoves} & \text{if all tiles mowed} \\ 0.1 \times \text{progLength} + \text{numUnmowedTiles} & \text{otherwise} \end{cases}$$

where *extraMoves* is the number of moves made after the last tile was mowed.

3.2 Other Algorithms Used for Comparison

In order to evaluate the strengths and weaknesses of N-gram GP, we tested it against two other techniques: simple stochastic hill climbing and a traditional linear GP system. All algorithms were given the same number of fitness evaluations (20,000 in all experiments reported in the paper) and were individually optimised (by doing a large sweep of their parameter space) to maximise their performance on our test problems. These parameters are detailed in Table 2.

The hill climber is initialised by choosing a random program length between 1 and ℓ_{max} and then generating a random program of that length. From then on, the algorithm repeatedly attempts to improve over the best individual seen so far by randomly mutating it. The algorithm has two equally probable mutation operations to choose from. The first is point mutation which is applied to the primitives of the best program with a mutation rate of p/ℓ where ℓ is the length of the current best program and p is a parameter ($p = 2$ in our experiments). This form of mutation cannot change program length. The second form of mutation is effectively subtree mutation applied to linear sequences. I.e., a random mutation point is chosen in the parent individual, all of the instructions following the mutation point are excised, and they are replaced by a newly generated random sequence. As a result, the offspring program can have a length which is different from the parental length.

Our linear GP system works as follows. It initialises the population by repeatedly creating random individuals using the same distribution as for the initialisation of the hill climber, and evaluating their fitness. Then a typical steady state evolutionary loop starts. At each iteration the algorithm decides whether to create a new individual via mutation or crossover. If mutation is chosen, a parent individual is selected via tournament selection (with tournament size 2), and an offspring is generated using the same algorithm as for the hill climber (i.e., we use point mutation 50% of the time, and subtree mutation the other 50% of the time). If crossover is chosen, we select two parents (again, via tournament selection) and apply homologous two-point crossover with 50% probability, and subtree crossover with 50% probability. Subtree crossover involves the selection of one crossover point in each parent, and the swap of the instructions following the crossover

Table 2. Parameter settings for hill-climber, linear GP and N-gram GP

Parameter	*Hill-Climber*	*Linear GP*	*N-gram GP*
Fitness evaluations	20,000	20,000	20,000
Independent runs	1,000	1,000	1,000
ℓ_{max} Polynomial Problem	100	100	100
Lawn-Mower	$4 \times d^2$	$4 \times d^2$	$4 \times d^2$
Point mutation rate (per primitive)	$\frac{2}{\ell}$	$\frac{1}{\ell}$	$\frac{0.25}{\ell}$
Population size	1	500	10
Generations	20,000	40	2,000
Crossover rate (per individual)	n/a	0.9	n/a
Mutation rate (per individual)	1	0.1	1
Tournament size	n/a	2	n/a
Truncation selection ratio	n/a	n/a	5
η_M learning rate	n/a	n/a	8
η_L learning rate	n/a	n/a	0.075

points. Homologous crossover requires choosing the same crossover points in both parents. Irrespective of the genetic operation chosen, the individual picked for replacement is selected via a negative tournament.

3.3 Performance Results

Fig. 1(a) shows a comparison of the success rate obtained by the hill climber, the N-gram GP and the linear GP on Polynomial problems of degrees from 5 to 12, when using the `PlusTimesSwapR1R2` primitive set. It is apparent how the use of a small set of suitable primitives makes the problem solvable for all techniques tested. Unsurprisingly, the simple hill-climber does well on the relatively easy instances, but its performance is unsatisfactory for d bigger than 9 or 10. On the contrary, the performance of the linear GP never really drops to unacceptable levels. Furthermore, both linear GP and the hill-climber do marginally better than the N-gram GP for small d. On the more difficult problems, however, N-gram GP shows much better performance. Furthermore, its performance drops more slowly than the other techniques as d increases, suggesting better scalability on these problems.

As illustrated in Fig. 1(b), the use of a larger-than-necessary primitive set (`AllOpsSwapR1R2`) makes the problem harder, because the size of the search space increases enormously without a corresponding increase in the size of the solution space. In these conditions, a searcher would need more time and resources to identify solutions. However, in this work, all problems, searchers and primitive sets are compared using the same number of fitness evaluations, leading to generally poorer performance. Despite this general trend, all observations we made for Fig. 1(a) appear to be valid for Fig. 1(b) as well. In particular the N-gram GP system still shows superior scalability and higher performance on the harder problems.

Finally, let us consider the `Lawn-Mower` problem. Again the simple hill-climber is the weakest of all searchers. Linear GP is marginally superior to N-gram GP for small problem sizes. However, its performance rapidly degrades, becoming unacceptable for

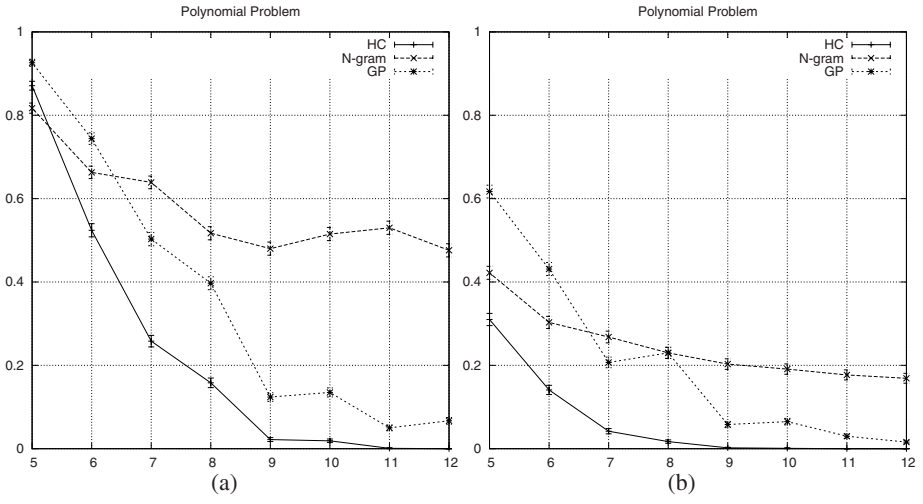

Fig. 1. Success rate of hill climber, N-gram GP and linear GP on Polynomial problems of degrees from 5 to 12, when using the `PlusTimesSwapR1R2` primitive set (a) and the `AllOpsSwapR1R2` primitive set (b). Parameters are as detailed in Table 2.

problems of size $d = 11$ or larger. Conversely, N-gram GP's performance degrades much more gracefully, leading it to being still able to solve problems of size $d = 12$ in about one third of runs.

4 Analysis and Discussion

The results from Sec. 3.3 clearly show that N-gram GP outperforms hill-climbing and often also linear GP, so the N-gram GP is obviously learning during a run. What is being learned, and how is that happening? Presumably, if there is learning going on, it is in the proportions being stored in the $M^{(i)}$ and P_L arrays. To better understand this learning, we will examine the kinds of bias that develops in these matrices in successful runs.

Of particular interest is whether (and how) N-gram GP learns longer patterns when limited to only keeping statistics on the occurrence of triplets. Even though N-gram GP can only learn triplets, there can obviously be correlations across multiple triplets; if *wxy* and *xyz* both have high probabilities, then the 4-tuple *wxyz* is a likely outcome if one starts with *wx*. To what degree does N-gram GP utilise this opportunity? How long are the patterns that it learns? What role does repetition play in those patterns?

To address these questions, we took successful runs, generated probability trees based on the distribution matrices, and catalogued programs that were generated with high probabilities. In every instance that we explored, there were clear patterns of instructions captured in the distribution matrices. In Sec. 3.3, the runs were halted once the goal was discovered. The distribution matrices often were not strongly converged when the goal was first found, so for this section we allowed runs to continue through the maximum allowed number of fitness evaluations, regardless of whether a solution was discovered.

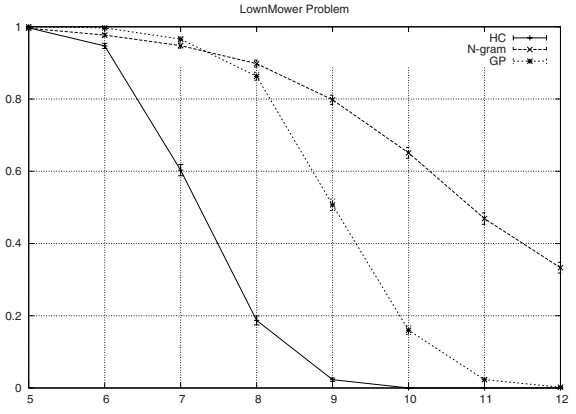

Fig. 2. Success rate of hill climber, N-gram GP and linear GP on Lawn-Mower problems with lawn sizes from 5×5 to 12×12. Parameters are as detailed in Table 2.

This gave the distribution matrices the opportunity to continue converging after finding the goal, making it easier to see what was being learned. To make it easier to present long sequences of instructions, and to see patterns in those sequences, we will present programs as sequences of the integer IDs of the instructions, using the IDs given in Table 1.

4.1 Polynomial

To simplify our analysis of the polynomial problems, we will consider each sequence of instructions as a mapping from one state of the system to another. Since the state in the regression problems is fully determined by the contents of the registers, we can represent that state as an ordered tuple. In the two register problems, for example, we can use an ordered pair (r_1, r_2) to represent the values of `R1` and `R2`, respectively. Consider, for example, the pair of instructions represented by the index sequence 53, namely instruction 5 (`R2 = R1 * R2`) followed by instruction 3 (`R2 = R1 + R2`). If we start the system in state (a,b) (where a and b represent arbitrary initial values) and execute this pair of instructions we end in the state $(a, a(b+1))$.

It turns out that polynomials of the form $x + x^2 + \cdots + x^d$ can be easily constructed in a highly patterned way using our simple instruction set. In one run, for example, with the degree 7 target $(x + x^2 + \cdots + x^7)$, the evolved distribution matrices have a high probability of generating sequences containing repetitions of the instruction pair 53, such as the solution 1535353535352. Here the initial instruction (1) loads x into R2, which, because `R1` is initialised to contain x, means that our state is (x,x) after that first instruction. The first pair 53 then maps this to $(x, x + x^2)$, the second to $(x, x + x^2 + x^3)$, and so on until the last 53 pair yields the state $(x, x + x^2 + x^3 + x^4 + x^5 + x^6)$. The final pair is 52 which has a similar effect, but leaves the result in `R1`, so we have the final state $(x + x^2 + \cdots + x^7, x^2 + x^3 + x^4 + x^5 + x^7)$, which has our target function in `R1`.

This solution is actually somewhat brittle because it depends crucially on getting the final 52 pair at the end, while having 53's everywhere else. This is reflected in the

probability matrices, where the probability of following 35 with a 3 is 66%, while the probability of following 35 with a 2 is only 1.3%. Contrast this with the probabilities following the pair 53, where the probability of a 5 is greater than 99.999%. Consequently the system has learned that the sequence 53 should *always* be followed by a 5, where the sequence 35 should *usually* be followed by 3, but occasionally by a 2 instead.

A somewhat more robust solution found on another run is 3412412412412412. Here the only "novelty" is the initial 3, which ensures that both registers have a copy of the argument x at the beginning. Then the pattern 412 generates the sequence of polynomials $x, x+x^2, x+x^2+x^3, \ldots, x+x^2+x^3+x^4+x^5+x^6+x^7$ in `R1`. This solution is also interesting in that it in fact generates *every* polynomial of the form $x+x^2+\cdots+x^d$, and all that is necessary to generate polynomials of other degrees is to increase or decrease the number of 412 triplets appropriately.

One might reasonably assume that the probability of generating the sequence 412 must be very high in this solution, perhaps approaching 1. It is, however, 83% which, while quite high, certainly is not high enough to ensure long repeated sequences of 412 triples. Looking at the length distribution, it turns out that the length 91 is far and away the most likely length in this run (an order of magnitude more likely than any other length in P_L). 91 instructions is obviously *far* more than necessary to solve this problem, so a successful 91 instruction program based on the 412 pattern is going to have to "get lucky" and have some instructions re-write `R1` to x towards the end, leaving just the right number of 412 triples at the end. This, however, requires that there is some reasonable chance of "escaping" runs of 412, which presumably accounts for the lower than expected probability for this key triplet. If this run had settled on a shorter length, one might expect the probability of generating the 412 sequence to be higher.

Tests with functions with more complex structure, e.g., the polynomials of the form $x^d+2x^{d-1}+3x^{d-2}+\cdots+d\,x$ (not reported due to space limitations), show that N-gram GP is capable of learning correlations that allow it to construct important sequences that are considerably longer than the triplets actually tracked in the distribution matrices. For example, in one case, N-gram GP solve a problem by learning a sequence of 9 instructions and setting triplets probabilities so that this sequence was generated with roughly 63% probability, which is over 500,000 times more likely than choosing it at random from a uniform distribution of sequences of seven instructions. N-gram GP is thus clearly capable of capturing and using long distance correlations despite only tracking the distribution of 3-grams.

4.2 Lawnmower

In contrast to the polynomial regression problems discussed above, the lawnmower problem requires much less precision in the sequence of instructions, and consequently the distribution matrices do not converge as strongly on a specific sequence.

One representative run, for example, is capable of generating a whole variety of sequences of instructions, with much less obvious patterning, including sequences such as 0000000001002…, 0000000020202…, and 2001122020020…. Still, while there are not clear sequences of instructions, there are definite trends in the distribution of instructions. This run has a high probability (74%) of starting with a `Mow` instruction, and given an initial `Mow` instruction, again a high probability (75%) of following that

with a second `Mow` instruction. In general `Mow` instructions are very likely throughout, as we would expect, while other combinations are quite uncommon. A `Mow-Right` pair has effectively no chance of being followed by a `Left`, which seems reasonable as a `Right-Left` sequence is effectively a NO-OP; in fact a `Mow-Right` pair is almost always followed by another `Mow`.

As an indication of the trends in the evolved distribution matrices, there are five initial sequences of length 8 that have a cumulative probability of at least 0.0001 of being generated: 00000000, 00000200, 00002000, 00020000 and 00200000. `Mow` instructions obviously dominate, with at most one other instruction in each case (which is in fact always a `Right`). So, in the lawnmower problem, instead of learning specific sequences of instructions to mow the lawn, N-gram GP develops stochastic space filling strategies.

5 Conclusions

We presented N-gram GP, an EDA for the evolution of linear computer programs. The algorithm learns and samples the joint probability of 3-grams of instructions at the same time as it is learning and sampling a program length distribution.

This work has several interesting features. Firstly, while several authors have extended EDAs to evolve computer programs, virtually all have done so for a tree representation. Here, for the first time, we explore the application of EDAs to a linear GP. The second distinctive feature of this work is that we explictly represent the program length distribution to be used during the search. With tree-based representations this is not used, since the primitive set always includes terminals, which, if drawn with sufficiently high frequency can terminate the construction of programs. A disadvantage of relying on terminal selection is that the probability of drawing terminals is totally under the control of evolution. Thus the user has no control over the size of the evolved programs, which, if unchecked, can easily become excessive. So, often hard limits on program depth or length must be artificially enforced, producing an undesirable bias. Here, by explicitly modelling the size distribution we instead have a natural way of limiting the search to programs of manageable size, without introducing any undesired bias. Finally, previous work has tended to use different probability distributions for different positions (or loci) in a tree, thereby expanding significantly the size of the parameter space in which the model lives. This is not a problem *per se*, but one must keep in mind that the more parameters a model has the more information must be collected from the search space to properly set those parameters, while still avoiding over-fitting. In N-gram GP we borrow from the field of natural language processing, using n-grams to represent regularities in the language necessary to solve a problem. This means that we use the same distribution for all loci in a program (except the first two, of course). This leads to a much smaller model space, where models can thus reliably be identified with less sampling, and with a higher degree of regularity in the evolved solutions.[2]

In our tests with two problem classes the N-gram GP system has been a very effective solver. Furthermore, its scalability has been significantly better than the simple

[2] Regular solutions to a problem are normally more compact and easier to interpret. However, whether or not highly regular solutions exist for a particular problem, or are easier to find than irregular ones, depends on the problem. If there is regularity, N-gram GP can exploit it.

hill-climber and linear GP, leading it to routinely solve problems of a difficulty which is way beyond what can be tackled by the other two algorithms tested.

References

1. Abbass, H., Hoai, N., McKay, R.: AntTAG: A new method to compose computer programs using colonies of ants. In: IEEE Congress on Evolutionary Computation (2002)
2. Baluja, S., Caruana, R.: Removing the genetics from the standard genetic algorithm. In: Prieditis, A., Russell, S. (eds.) Machine Learning: Proceedings of the Twelfth International Conference, pp. 38–46. Morgan Kaufmann Publishers, San Francisco (1995)
3. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
4. Larrañaga, P., Lozano, J.A.: Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation. Kluwer Academic Publishers, Dordrecht (2002)
5. Manning, C., Schütze, H.: Foundations of statistical natural language processing. MIT Press, Cambridge (1999)
6. Mühlenbein, H., Mahnig, T.: Convergence theory and application of the factorized distribution algorithm. Journal of Computing and Information Technology 7(1), 19–32 (1999)
7. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In: Kinnear Jr, K.E. (ed.) K, ch. 14, pp. 311–331. MIT Press, Cambridge (1994)
8. Poli, R., McPhee, N.F.: A linear estimation-of-distribution GP system. Tech. Report CES-479, Dept. of Computing and Electronic Systems, University of Essex (January 2008)
9. Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
10. Ratle, A., Sebag, M.: Avoiding the bloat with probabilistic grammar-guided genetic programming. In: Collet, P., Fonlupt, C., Hao, J.-K., Lutton, E., Schoenauer, M. (eds.) EA 2001. LNCS, vol. 2310, pp. 255–266. Springer, Heidelberg (2002)
11. Salustowicz, R.P., Schmidhuber, J.: Probabilistic incremental program evolution. Evolutionary Computation 5(2), 123–141 (1997)
12. Sastry, K., Goldberg, D.E.: Probabilistic model building and competent genetic programming. In: Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practise, ch. 13, pp. 205–220. Kluwer, Dordrecht (2003)
13. Shan, Y., McKay, R.I., Abbass, H.A., Essam, D.: Program evolution with explicit learning: a new framework for program automatic synthesis. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC 2003, Canberra, December 2003, pp. 1639–1646. IEEE Press, Los Alamitos (2003)
14. Shan, Y., McKay, R.I., Essam, D., Abbass, H.A.: A survey of probabilistic model building genetic programming. In: Pelikan, M., Sastry, K., Cantu-Paz, E. (eds.) Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications, Springer, Heidelberg (2006)
15. Suen, C.Y.: *n*-gram statistics for natural language understanding and text processing. IEEE Transactions on Pattern Analysis and Machine Intelligence 1(2), 164–172 (1979)
16. Yanai, K., Iba, H.: Estimation of distribution programming based on bayesian network. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC 2003, pp. 1618–1625. IEEE Press, Los Alamitos (2003)
17. Yanai, K., Iba, H.: Program evolution by integrating EDP and GP. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, pp. 774–785. Springer, Heidelberg (2004)