View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Essex Research Repository

# A Linear Estimation-of-Distribution GP System

Riccardo Poli
Department of Computing and Electronic Systems
University of Essex, UK
rpoli@essex.ac.uk

Nicholas F. McPhee
Division of Science and Mathematics
University of Minnesota, Morris, USA
mcphee@morris.umn.edu

Technical Report CES-479
ISSN: 1744-8050
January 2008

**Abstract**

We present N-gram GP, an estimation of distribution algorithm for the evolution of linear computer programs. The algorithm learns and samples the 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 $12 \times 12$. 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) 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. The distribution is modified generation after generation, by 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. This paper introduces *N-gram GP*, an EDA for the evolution of linear computer programs.

EDAs are reviewed in [12, 13]. Different algorithms use different models for the probability distribution that controls the sampling. For example, population-based incremental

learning (PBIL) [2] and the uniform multivariate distribution algorithm (UMDA) [17, 18] assume that the distribution is factorised into a product of univariate marginals, i.e., each variable in a chromosome is assumed to be independent of other variables. So, these algorithms need to store and adjust only a linear array of probabilities. This works well for problems without significant interactions between variables. Partial solutions of order greater than one are disrupted, however, making it difficult to solve problems where the fitness function has significant interactions between variables. Naturally, higher order models are possible. For example, the MIMIC algorithm of [6] uses second-order statistics. It is also possible to use flexible models where interactions of different orders are captured. For example, the Bayesian optimisation algorithm (BOA) [21] uses Bayesian networks to represent generic sampling distributions, while the extended compact genetic algorithm (eCGA) [9] clusters genes into groups where the genes in each group are assumed to be linked but groups are considered independent. The sampling distribution is then taken to be the product of the distributions modelling the groups.

There have been several prior applications of probabilistic model-based evolution (EDA-style) to tree-based GP. The first EDA-type GP system was inspired by PBIL [2] and was called probabilistic incremental program evolution (PIPE) [25]. In PIPE, the population is replaced by a hierarchy of probability tables organised into a tree. Each table represents the probability that a particular instruction be at that particular location in a newly generated program tree. At each generation a population of programs is generated. The generation of a program starts by choosing a root node based on the probabilities of the root table, and it then continues recursively down the hierarchy of probability tables until all branches of the tree are complete. The probability hierarchy is updated on the basis of the fitness of the programs in the population, so as to make the generation of above average fitness programs at the next generation more likely. A positive feature of PIPE is that the probabilities of choosing each primitive can vary with its depth (and, more generally, position) in the tree. This makes it possible, for example, for terminals to become more and more probable as a node's depth increases. A limit of PIPE, however, is that the primitives forming a tree are chosen independently from each other, so it is impossible for PIPE to capture dependencies among primitives. Another limitation is that trees are limited to a preset depth. In [26] an algorithm called extended compact GP (eCGP) was proposed which effectively extends the eCGA algorithm [9] to trees. The algorithm assumes that all trees to be created by sampling will fit within a maximal tree. It partitions the nodes in this maximal tree into groups. The nodes in a group are assumed to be linked and their co-occurrence is modelled by a full joint distribution table. Like for eCGA, the probability of generating a particular individual (a tree in the case of eCGP) is given by the product of the probabilities of generating each group of genes using the groups' joint distributions. An advantage of the system is that, unlike PIPE, it capture dependencies among primitives. In [32] an EDA called estimation of distribution programming (EDP) was proposed that, in principle, can capture complex dependencies between nodes in a program tree and nodes directly above it (or to its left), through the use of a conditional probability table. Like for eCGP and PIPE, programs are tree-like and are assumed to always fit within an ideal maximal full tree. A conditional probability table is necessary for each node in such a tree. So, to keep the size

of data structures under control, the system was only tested by modelling the dependency of each node on its parent node. I.e., the model captured pairwise dependencies. A hybrid between EDP and GP was proposed in [33]. A hierarchical BOA is used as the main mechanism to generate new deme members in a system called MOSES (Meta-Optimising Semantic Evolutionary Search) [14], which combines multiple strategies and uses semantics to restrict and direct the search. BOA was also used to evolve programs in [15] using a specialised tree representation.

Various other systems have been proposed which combine the use of grammars and probabilities.[1] We mention only a few here (an extended review is available in [30]) since this paper focuses on grammar-less GP approaches. For example, [23] used a stochastic context free grammar to generate 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. The system could also be run in a mode where rule probabilities are contextual to the depth of the non-terminal symbol to which a rewrite rule is applied, thereby providing a higher degree of flexibility. Slightly more general is the approach taken in PEEL (Program Evolution with Explicit Learning) [28], where a probabilistic L-system is used where rewrite rules are depth- *and* location-dependent and have associated probabilities of application which are adapted by an Ant Colony Optimisation (ACO) algorithm [7]. ACOs are characterised by pheromone update rules which strongly resemble the model update rules in EDAs. A feature of PEEL is that the L-system's rules can automatically be refined via splitting and specialisation. Other programming systems based on probabilistic grammars which are optimised via ant systems include ant-TAG [1, 27] (which uses a tree-adjunct grammar as its main representation) and generalised ant programming (GAP) [10] (which is based on a context free grammar). Other systems which learn and use probabilistic grammars include grammar model based program evolution (GMPE) [29], the system described in [3, 4] and Bayesian automatic programming (BAP) [24].

While there is a significant amount of work in this area, we are unaware of any applications of EDA-style ideas to linear GP. This paper starts filling this gap, by proposing an EDA-type GP system capable of evolving machine-language-type programs [19, 20, 5].

The paper is organised as follows. We present our method in Sec. 2. In Sec. 3 we thoroughly evaluate it against hill climbing and standard linear GP. In Sec. 4 we analyse how N-gram GP solves problems and what it learns during evolution. We provide our conclusions in Sec. 5.

# 2 N-gram GP

An *n*-gram is a group of *n* 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

[1] In fact, there is a fundamental equivalence between probabilistic grammars and other probabilistic approaches (see [30] for a detailed explanation).

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 [31, 22, 16]. 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, our sequences are sequences of instructions from the assembly language of a register-based CPU, as in linear GP [19, 20, 5, 8].

In this paper 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 before the process of constructing a program is started a program length is drawn from this distribution. Programs are then grown up to the prescribed length. In this paper we take this second 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. However, 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 somehow the distributions $M^{(3)}$ and $P_L$ were 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 (we assumed this in all experiments reported in the paper), the most natural initialisation is the uniform distribution. If $\ell_{max}$ 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. Note that, following standard EDA practice, to maintain diversity and ensure that the search continues even after many entries of $M^{(3)}$ converge to 0, we 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 $M^{(3)}$ as shown in Alg. 3. We use an additive update rule for updating $P_L$ and $M^{(3)}$. 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, $\eta_M$ and $\eta_L$. If desired, these can be made arbitrarily big. When $\eta_M \gg 1$ and $\eta_L \gg 1$ effectively $P_L$ and $M^{(3)}$ are entirely determined by the current elite and not 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`. In this section we briefly describe these problems and the primitive sets used to solve them.

`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, 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 $x = -1$, $x = -0.9$, ..., $x = 1.0$. Fitness (to

**Algorithm 1** Program generation algorithm of 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** $\ell$ **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

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.

For this problem 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.

**Algorithm 3** Learning in N-gram GP.

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

1: **for** $i = 1 \ldots |\texttt{elite}|$ **do**
2: $\quad x = \texttt{elite}[i]$
3: $\quad \ell = \texttt{length}(\ x\ )$
4: $\quad P_{L,\ell} = P_{L,\ell} + \eta_L/\ell_{max}$
5: $\quad$ **for** $j = 3 \ldots \ell$ **do**
6: $\qquad M^{(3)}_{x_{j-2},x_{j-1},,x_j} = M^{(3)}_{x_{j-2},x_{j-1},x_j} + \eta_M/N^3$
7: $\quad$ **end for**
8: **end for**
9: $M^{(3)} = M^{(3)}/\sum_{l,m,n} M^{(3)}_{l,m,n}$
10: $P_L = P_L/\sum_l P_{L,l}$.

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 | |

`Lawn-Mower` is a variant of the classical Lawn Mower problem introduced by Koza in [11]. 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 and, so, whether or not the lawnmower keeps 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, using the 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 $\ell_{max}$ 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 $p/\ell$ where $\ell$ 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, then, we 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 points. Homologous crossover requires choosing the same crossover points in both par-

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 |
| $\ell_{max}$ 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 |
| $\eta_M$ learning rate | n/a | n/a | 8 |
| $\eta_L$ learning rate | n/a | n/a | 0.075 |

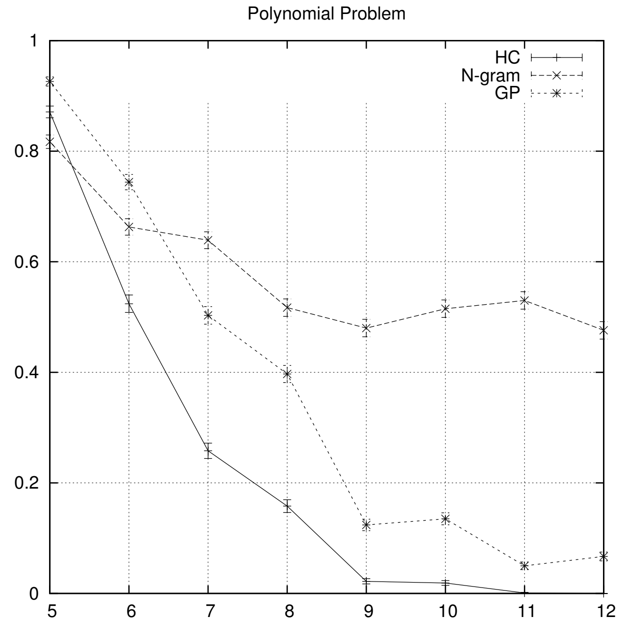
ents. Irrespective of the genetic operation chosen, the individual picked for replacement is selected via a negative tournament.

## 3.3 Performance Results

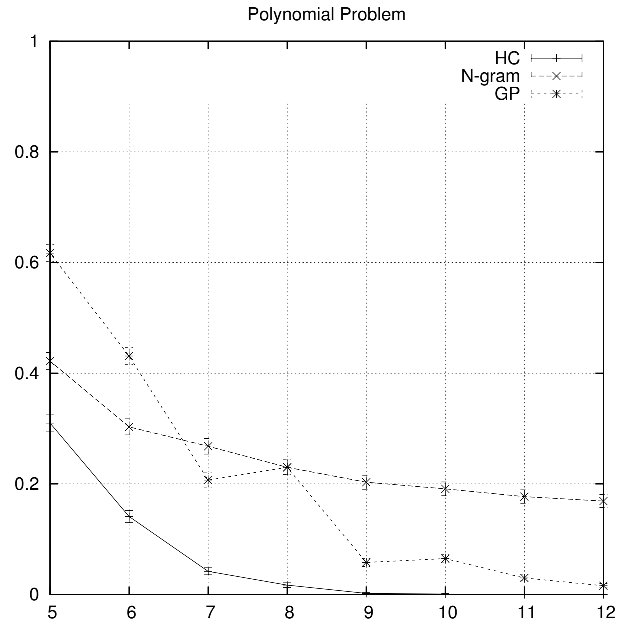
Fig. 1(a) shows a comparison of the success rate obtained by the hill climber, the N-gram GP and the linear GP on Polynomial problems of degrees from 5 to 12, when using the `PlusTimesSwapR1R2` primitive set. It is apparent how the use of a small set of suitable primitives makes the problem solvable 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 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 here. In particular, again, the N-gram GP system 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




(a)




(b)

Figure 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.

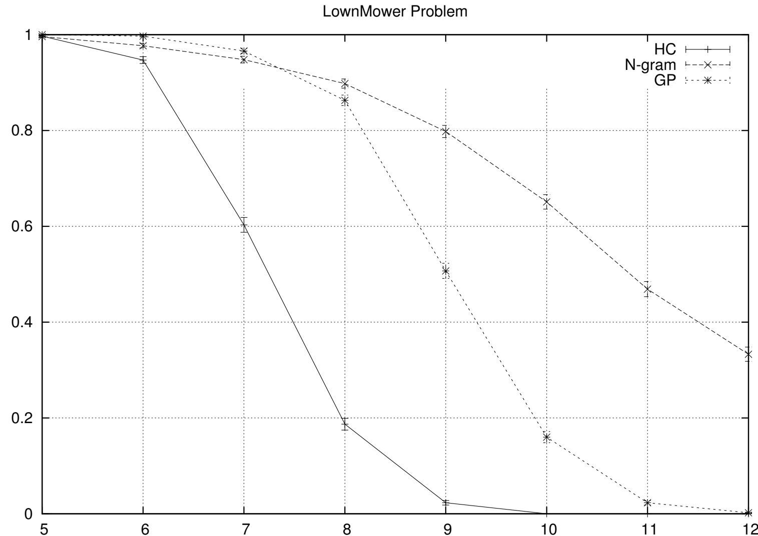



Figure 2: Success rate of hill climber, N-gram GP and linear GP on Lawn-Mower problems with lawn sizes from $5 \times 5$ to $12 \times 12$. Parameters are as detailed in Table 2.

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 problems of size $d = 11$ or bigger. On the contrary, 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 something during a run. The question, then, is 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, and generated probability trees based on the distribution matrices, cataloguing 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 for a fixed number of fitness evaluations, regardless of when the target was discovered. This gave the distribution matrices the opportunity to continue converging after finding the goal, making it clearer what was in fact 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 regression 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 53, namely `R2 = R1 * R2` followed by `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 3412412412412412412. 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 for this run, 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 in such a way to ensure 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. There is 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, it would appear the 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 have presented N-gram GP, an estimation of distribution algorithm for the evolution of linear computer programs. The algorithm learns and sample the joint probability of 3-grams of instructions at the same time as it is learning and sampling a program length distribution.

This work presents some interesting features. Firstly, although several authors have extended estimation of distribution algorithms to evolve computer programs, virtually all have done so for the tree representation. Here, for the first time, we explore the application of EDAs to linear-GP-type representations. 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 term 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 (except the first two, of course) in a program. This leads to a much smaller model space where models can thus reliably be identified with less sampling, and a higher degree of regularity in the evolved solutions.[2]

In our tests with two problem classes, polynomial regression of high degree and lawnmower problems with large lawn sizes, the N-gram GP system has been a very effective solver. Furthermore, its scalability has been significantly better than the simple 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.

[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.

# References


[1] H. Abbass, N. Hoai, and R. McKay. AntTAG: A new method to compose computer programs using colonies of ants. In *IEEE Congress on Evolutionary Computation, 2002.*, 2002.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 38–46. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[3] P. A. N. Bosman and E. D. de Jong. Grammar transformations in an EDA for genetic programming. In R. Poli, S. Cagnoni, M. Keijzer, E. Costa, F. Pereira, G. Raidl, S. C. Upton, D. Goldberg, H. Lipson, E. de Jong, J. Koza, H. Suzuki, H. Sawai, I. Parmee, M. Pelikan, K. Sastry, D. Thierens, W. Stolzmann, P. L. Lanzi, S. W. Wilson, M. O'Neill, C. Ryan, T. Yu, J. F. Miller, I. Garibay, G. Holifield, A. S. Wu, T. Riopka, M. M. Meysenburg, A. W. Wright, N. Richter, J. H. Moore, M. D. Ritchie, L. Davis, R. Roy, and M. Jakiela, editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004.

[4] P. A. N. Bosman and E. D. de Jong. Learning probabilistic tree grammars for genetic programming. In X. Yao, E. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. Rowe, P. T. A. Kabán, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 192–201, Birmingham, UK, 18-22 Sept. 2004. Springer-Verlag.

[5] R. L. Crepeau. Genetic evolution of machine language software. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 121–134, Tahoe City, California, USA, 9 July 1995.

[6] J. S. de Bonet, C. L. Isbell, Jr., and P. Viola. MIMIC: Finding optima by estimating probability densities. In M. C. M. et. al., editor, *Advances in Neural Information Processing Systems*, volume 9, page 424. MIT Press, 1997.

[7] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press (Bradford Books), 2004.

[8] S. E. Eklund. A massively parallel GP engine in VLSI. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 629–633. IEEE Press, 2002.

[9] G. Harik. Linkage learning via probabilistic modeling in the ECGA. IlliGAL Report 99010, University of Illinois at Urbana-Champaign, 1999.

[10] C. Keber and M. G. Schuster. Option valuation with generalized ant programming. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 74–81, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[12] P. Larrañaga. *A review on estimation of distribution algorithms*, chapter 3, pages 57–100. Kluwer Academic Publishers, 2002.

[13] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation.* Kluwer Academic Publishers, 2002.

[14] M. Looks. Scalable estimation-of-distribution program evolution. In H. Lipson, editor, *GECCO*, pages 539–546. ACM, 2007.

[15] M. Looks, B. Goertzel, and C. Pennachin. Learning computer programs with the bayesian optimization algorithm. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 747–748, Washington DC, USA, 25-29 June 2005. ACM Press.

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

[17] H. Mühlenbein and T. Mahnig. Convergence theory and application of the factorized distribution algorithm. *Journal of Computing and Information Technology*, 7(1):19–32, 1999.

[18] H. Mühlenbein and T. Mahnig. FDA – a scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary Computation*, 7(4):353–376, 1999.

[19] 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.

[20] P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[21] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532, Orlando, FL, 13-17 1999. Morgan Kaufmann Publishers, San Fransisco, CA.

[22] L. Rabiner. A tutorial on hidden Markov models and selected applications inspeech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[23] A. Ratle and M. Sebag. Avoiding the bloat with probabilistic grammar-guided genetic programming. In P. Collet, C. Fonlupt, J.-K. Hao, E. Lutton, and M. Schoenauer, editors, *Artificial Evolution 5th International Conference, Evolution Artificielle, EA 2001*, volume 2310 of *LNCS*, pages 255–266, Creusot, France, Oct. 29-31 2001. Springer Verlag.

[24] E. N. Regolin and A. T. R. Pozo. Bayesian automatic programming. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 38–49, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[25] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[26] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practise*, chapter 13, pages 205–220. Kluwer, 2003.

[27] Y. Shan, H. Abbass, R. I. McKay, and D. Essam. AntTAG: a further study. In R. Sarker and B. McKay, editors, *Proceedings of the Sixth Australia-Japan Joint Workshop on Intelligent and Evolutionary Systems*, Australian National University, Canberra, Australia, 30 Nov. 2002.

[28] Y. Shan, R. I. McKay, H. A. Abbass, and D. Essam. Program evolution with explicit learning: a new framework for program automatic synthesis. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1639–1646, Canberra, 8-12 Dec. 2003. IEEE Press.

[29] Y. Shan, R. I. McKay, R. Baxter, H. Abbass, D. Essam, and N. X. Hoai. Grammar model-based program evolution. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 478–485, Portland, Oregon, 20-23 June 2004. IEEE Press.

[30] Y. Shan, R. I. McKay, D. Essam, and H. A. Abbass. A survey of probabilistic model building genetic programming. In M. Pelikan, K. Sastry, and E. Cantu-Paz, editors,

*Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*. Springer, 2006.

[31] 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.

[32] K. Yanai and H. Iba. Estimation of distribution programming based on bayesian network. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1618–1625, Canberra, 8-12 Dec. 2003. IEEE Press.

[33] K. Yanai and H. Iba. Program evolution by integrating EDP and GP. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 774–785, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.