An Investigation of Local Patterns For Estimation of Distribution Genetic Programming

Erik Hemberg
School of Computer Science & Informatics
University College Dublin
erik.hemberg@ucd.ie

Kalyan Veeramachaneni
MIT CSAIL
kalyan@csail.mit.edu

James McDermott
MIT CSAIL
jmmcd@csail.mit.edu

Constantin Berzan
Department of Computer Science
Tufts University
cberzan@gmail.com

Una-May O'Reilly
MIT CSAIL
unamay@csail.mit.edu

ABSTRACT

We present an improved estimation of distribution (EDA) genetic programming (GP) algorithm which does not rely upon a prototype tree. Instead of using a prototype tree, Operator-Free Genetic Programming learns the distribution of ancestor node chains, "n-grams", in a fit fraction of each generation's population. It then uses this information, via sampling, to create trees for the next generation. Ancestral n-grams are used because an analysis of a GP run conducted by learning graphical models for each generation indicated their emergence as substructures of conditional dependence. We are able to show that our algorithm, without an operator and a prototype tree, achieves, on average, performance close to conventional tree based crossover GP on the problem we study. Our approach sets a direction for pattern-based EDA GP which offers better tractability and improvements over GP with operators or EDAs using prototype trees.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Algorithms

Keywords

genetic programming, estimation of distribution, representation

1. INTRODUCTION

In genetic programming (GP), the crossover and mutation operators tend to be highly destructive. Not only are newly-produced individuals often worse than their predecessors, as in many other forms of evolutionary computation (EC); they are also often *semantically dissimilar* to their predecessors to a high degree [7]. This is not surprising. Tree shape correlates with semantics at least weakly. Likely, more influentially, our intuition formed when programming "by hand" suggests that executable structures are highly context dependent. Part of a program pasted into a different context will, even if syntactically correct, result in very different behaviour, e.g. it makes a great difference whether a subtree is executed as the first or the second child of a non-commutative operator like division. GP crossover and mutation operators ensure syntactic correctness but make no attempt at semantic compatibility. (There are a few exceptions, such as [8]).

A very different approach to passing on useful content and structure is the *estimation of distribution algorithm* (EDA). Crossover and mutation are done away with, and replaced with statistical distributions which are estimated and then sampled from, from one generation to the next, from a high-fitness fraction of the population. An EDA's learning is explicit, in the distribution, as opposed to implicit in the population. EDAs have enjoyed significant success in numerical problems [10] and have also been used in GP (see Sect. 2).

In the case of optimization problems with binary or discrete search spaces, the decision variables $\{d_1, d_2..d_n\}$ are mapped to random variables $\{Y_1, Y_2, \ldots Y_n\}$ to estimate a multivariate distribution $P(Y_1, Y_2, \ldots Y_n\}$ with the discrete range of each decision variable mapping to support for its random variable. A Bayesian network with *binomial or multinomial* distributions at each node can be learned from a fit fraction of a population. A Bayesian network has both structure and parameters. A correlation or dependency structure can be imposed on the network or learned every generation. With adoption of a variety of estimation techniques from statistical machine learning EDAs such as the Bayesian Optimization Algorithm (BOA) have enjoyed significant success in numerical problems [10].

Exploiting this framework for genetic programming (GP) is less straightforward than with continuous or discrete optimization. This is because the representation or the number of nodes is not fixed. Additionally, GP's representation has no decision variables, per se. In some sense GP is looking for the "right" place to put a function or terminal and it can use a function or terminal multiple times in almost any

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

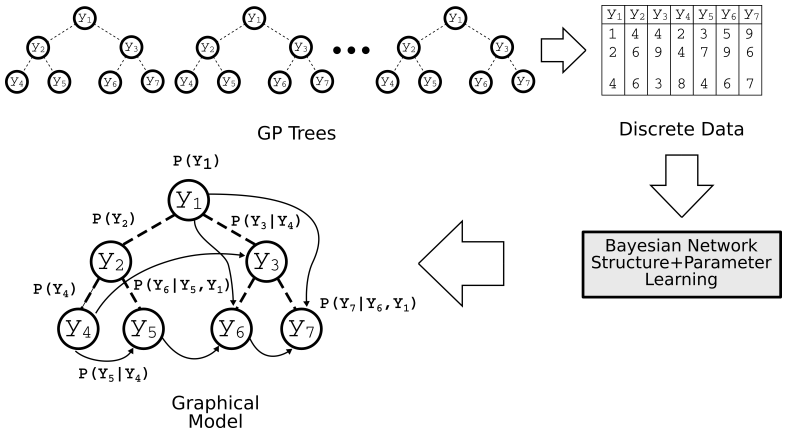


Figure 1: Converting a GP population into a statistical distribution. The nodes of the trees are the multiple random variables. The tree is parsed to extract the discrete choices made for each random variable. This multivariate dataset is then passed through structure and parameter learning routines to achieve a graphical model representation of the trees

place (subject to syntax restrictions). Therefore, in order to exploit the same techniques developed for EDA and BOA, EDA-GP references a so-called prototype tree.

A *prototype tree* is a single *complete* n-ary tree (where n is the maximum arity of the non-terminals). Each node in the tree is mapped to a *random variable* with its *support* corresponding to the discrete values for the choice set it can take. The choice set of the root is all functions, and the choice set of the leaves is only terminals. The choice set of all the nodes except the root and its two children (for binary tree) includes the value *null*. This enables an actual GP tree which could vary in size and structure to be generated by sampling from a multivariate distribution $P(Y_1, Y_2, \ldots Y_n\}$ defined on the prototype tree. However, it imposes the constraint that a program tree cannot exceed the height of the prototype tree. The process of transforming a population of GP trees into a multivariate distribution is shown in Figure 1.

A prototype tree large enough to "cover" all the possible tree sizes for candidate solutions must be used. This quickly makes the approach intractable, because increasing the depth of the prototype tree by one level doubles the number of nodes and hence the number of random variables whose joint distribution needs to be estimated. With the increased number of random variables estimating the structure and parameters for the multivariate distribution becomes computationally expensive. Additionally, the *support* of each random variable is large due to the typical number of functions and terminal sets in a GP problem. Thus the number of samples required for estimation also increases. The intractability has been circumvented by imposing constraints on the correlation structure of the multivariate distribution. We provide more background on different methods developed for EDA-GP in Section 2.

In this paper we are interested in overcoming the constraints and the computational expense induced by the prototype tree based EDA-GP by developing an alternative probabilistic model. It will reduce the computational expense since it does not rely upon estimating a (high dimensional, large *support*) distribution related to a prototype tree. Like conventional EDA-GP, OFGP iteratively estimates a distribution from a fit fraction of a generation then samples from it. However OFGP differs from convention by:

1. estimating the distribution related to a *pattern* that can occur throughout a tree, rather than a distribution where the random variables are related to the nodes of a (positionally static) prototype tree, and

2. referencing the pattern distribution with a tree creation algorithm/iterative sampling algorithm which generates trees which are biased by the distribution

We discuss how we determined the pattern OFGP uses in Section 3. Rather than relying upon intuition, we took a statistical approach to determine it. We learn the bayesian network for the population of trees from a standard GP run. We impose minimal constraints on the structure learning algorithm and are able to generate more insightful structure. What we learn from the analyses of these structures derived for population of a GP run, guides us in our definition of the local patterns whose distribution we will estimate and recursively sample from to generate trees.

We provide a detailed description of OFGP in Section 4 where we present an example of a pattern, explain how its distribution is learned and how it is used to generate the next iteration's trees.

In Sect. 5 we present the results achieved using our approach and evaluate OFGP in comparison to GP. We conclude and discuss future work in Sect. 6.

2. BACKGROUND

EDA-GP can be categorized based on whether or not an algorithm uses a *prototype tree* [14]. There are few approaches that do not use prototype tree. First is a *Grammar-based* method. A probability is attached to each production in each rule in a context-free grammar, so that the likelihood of achieving any given derivation is controlled. This area has been explored by Shan [15]. Additionally, *Generative* approaches have also been explored, e.g. [5, 16]. These approaches are based on different representations than the tree based representation we are interested in.

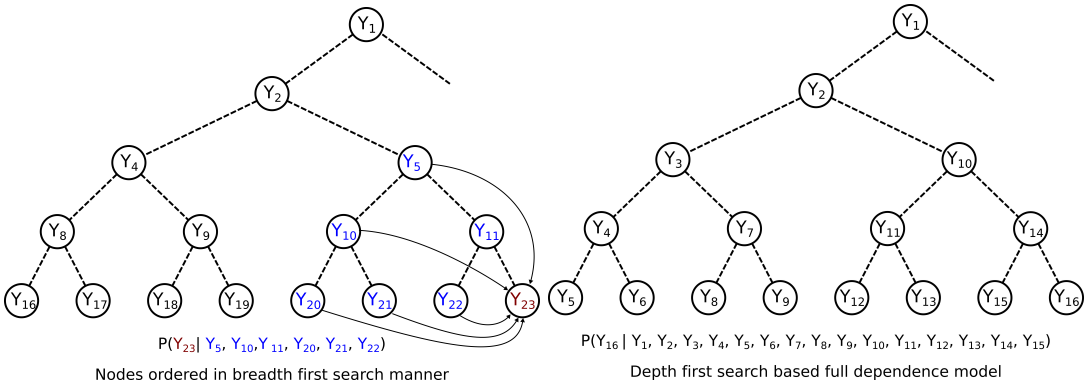


Figure 2: Dependencies modeled in different prototype tree based EDA-GP. The dependency structure for the last node is presented. On the right hand side the possible dependencies are shown as incoming arrows into the last node. This is model used in POLE. On the right hand side a dependency model is shown if a depth first search dependency model is used, i.e., the node depends on all the nodes preceding it.

Finally, a Linear GP program consists of a list of instructions for a register-based virtual machine. It does not use a tree representation however it has similarity with OFGP because it shares the use of n-grams. In Poli and McPhee [11] an n-gram approach is used where the length of programs was modeled separately, and updated according to the lengths of highly fit programs, alongside the n-gram model.

Prototype tree based methods have to estimate the joint multinomial multivariate distribution $P(Y_1 \dots Y_N)$. Two decisions guide the estimation of this distribution. First is the choice of its dependency structure: for a given random variable Y_i, what other random variable 's could it depend upon. This decision often written as $P(Y_1 \dots Y_N) = \Pi_{i=1}^{N}(Y_i|[Y_s])$ where $[Y_s]$ is a set of random variables on which Y_i depends. Second is the approach used for estimating the parameters of the multinomial distribution for each i, $\theta_i \leftarrow P(Y_i|[Y_s])$.

Multiple techniques designed can be distinguished based upon the dependencies incorporated in their probabilistic model, i.e. their model structure. The simplest of all is Probabilistic Incremental Program Evolution (PIPE) by Salustowicz and Schmidhuber [12]. Every random variable is assumed to be independent, hence the model simply decomposes into: $P(Y_1, Y_2, ...Y_N) = \prod_{i=1}^{N} P(Y_i)$. To assure a tree created by sampling the PIPE distribution is not too big, the probabilities of functions at a random variable deeper in the prototype tree are intentionally lowered.

The Extended Compact Genetic Programming (eCGP) [13] differs from PIPE in that it allows the probabilistic model to have dependencies between multiple random variables, i.e. multivariate interactions between GP nodes. The algorithm casts, finding optimal structure, as an optimization problem and uses minimum description length (MDL) as a cost function. This probabilistic model is called Marginal Product Model.

Estimation of Distribution Programming (EDP) by Yanai and Iba [17] learns the parameters for the Bayesian network that represents the multivariate distribution. The structure of the Bayesian network however, is fixed with parent node dependence, i.e., $P(Y_1, Y_2...Y_N) = P(Y_1)\prod_{i=2}^{N} P(Y_i|Y_i^{parent})$. Frequencies of samples are reweighted according to fitness and the conditional probability tables at each node are learnt using these weighted counts. Samples from the network are drawn to generate new individuals.

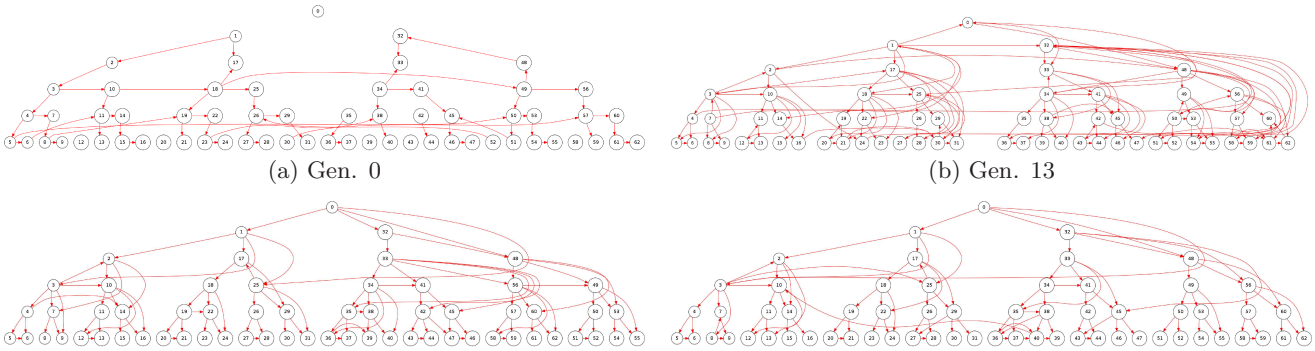
In Program Optimization with Linkage Estimation (POLE)[3, 4], again a Bayesian network is used to represent the multivariate distribution. However, in this approach the bayesian network structure is learnt every generation from the promising individuals. A heuristic structure learning algorithm called *K2* is used in which an additional constraint is placed such that each node can only depend on one another node among a set of choices available. These choices are determined for every node as follows. For any node i, let $U(Y_i, R_P)$ be the node positioned at R_P levels above Y_i. Directed dependencies are allowed between Y_i and nodes that belong to the subtree rooted at $U(Y_i, R_P)$ and nodes whose indices are smaller than i. Nodes are ordered in breadth-first traversal. Figure 2(left) shows the choices of parents for the node Y_{23} as per the POLE algorithm.This allows conditional dependence among left-hand uncles, siblings, parents and grandparents only. POLE's assumptions of dependency may seem obvious and intuitive, but there is no confirmation that they are appropriate.

3. GRAPHICAL MODELS FOR GP POPULATION

The population in standard GP forms an implicit probability distribution over trees. We want to explicitly model this distribution, and examine the dependency structure that emerges between nodes in the prototype tree. Our modeling tool of choice is the Bayesian network. A Bayesian network $\mathcal{B} = \langle \mathcal{G}, \theta \rangle$ is a probabilistic graphical model that represents a joint probability distribution over a set of random variables $Y_1, \dots, Y_n$. The Bayesian network representation has two components. A directed acyclic graph (DAG) $\mathcal{G}$ encodes independence relations between variables. Each variable is a node in this graph. A set of local probability models θ defines the conditional probability distribution of each node given its parents in the graph. Let Pa_Y denote the parents of node Y in $\mathcal{G}$. Then the network $\mathcal{B}$ encodes the following probability distribution:

$$P(Y_1, \dots, Y_n) \;=\; \prod_{i=1}^{n} P(Y_i \mid \mathrm{Pa}_{Y_i}).$$

Consult Koller and Friedman [6] for an introduction to Bayesian networks. In the rest of this section, we describe how we col-



(a) Gen. 0 (b) Gen. 13

(c) Gen. 25 (d) Gen. 40

Figure 3: The full graphical models for a GP population at generation 0, 13, 25 and 40

Table 1: Parameters for the GP in ECJ.

Parameter	Values
Language	Symbolic Regression
Population size	10,000
Generations	40
Crossover	0.9
Mutation	0.1
Selection	Tournament size 7
Initialization	Ramped Half-Half
Max Depth	5

lect data from GP, learn Bayesian networks from this data, and analyze the resulting network structures.

We first run GP on a symbolic-regression problem called Pagie-2D [9], which is to find an expression matching the target function $1/(1+x^{-4}) + 1/(1+y^{-4})$ over 676 fitness cases in $[-5,5]^2$. This seems to be a difficult problem [2], regardless of the available alphabet or functions. We use functions $\mathcal{F} = \{+, -, *, \%\}$ (where % is protected division) and terminals $\mathcal{T} = \{x, y, 0.1, 1.0\}$. We use a prototype tree of depth 5, which has 63 nodes. We run standard GP for 40 generations, using ECJ, with a population size of 10,000. The GP parameters are summarized in Table 1. After each generation, we store the entire population to disk. We repeat the process in 30 independent runs, thus obtaining 1,200 populations to analyze, the average best performance is in Fig. 7(a).

We convert each population into a data set $\mathcal{D}$ with one row for every tree. We represent each tree as a vector of 63 values, by traversing the prototype tree in depth-first order, and using *null* for any missing node. The root can take any value from the set of functions: $S_{\text{root}} = \{+, -, *, \%\}$. The leaves can take any value from the set of terminals, as well as *null*: $S_{\text{leaf}} = \{x, y, 0.1, 1.0, null\}$. Intermediary nodes can take any value from the set of functions and terminals, as well as *nil*: $S_{\text{intermediary}} = \{+, -, *, \%, x, y, 0.1, 1.0, null\}$. Thus, our data set $\mathcal{D}$ has 63 random variables $Y = \{Y_1, Y_2, \ldots Y_{63}\}$. Each variable is discrete, and its support is either S_{root}, S_{leaf}, or $S_{\text{intermediary}}$. We then learn a Bayesian network for the distribution $P(Y_1, Y_2 \ldots Y_{63} | \mathcal{D})$, as detailed in the next section.

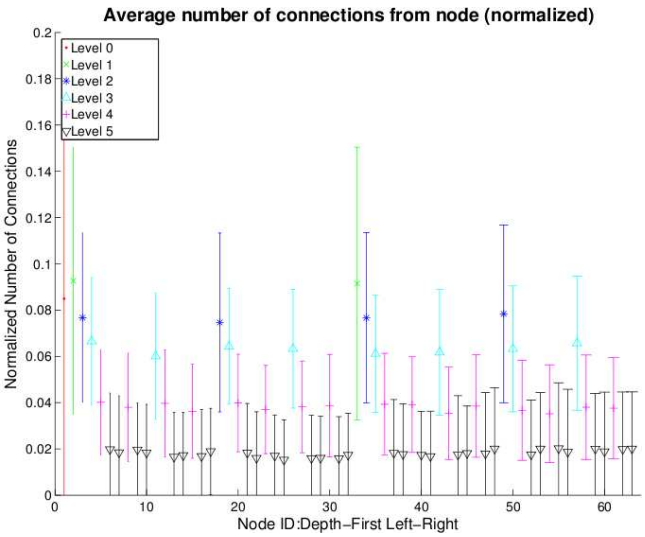
3.1 Learning the graphical model

Learning Bayesian networks from data has received much attention in the machine-learning community. The space of all possible DAGs for n nodes is super-exponential in n, and finding the optimal network for a given data set is in general NP-hard [6]. Learning algorithms for Bayesian networks loosely fit in two categories: constraint-based approaches, which use statistical tests to determine whether an edge is present or not, and search-and-score techniques, which define a scoring function and then search for a high-scoring network. The scoring function prefers network structures that model the data set well, while penalizing structures that are too complex.

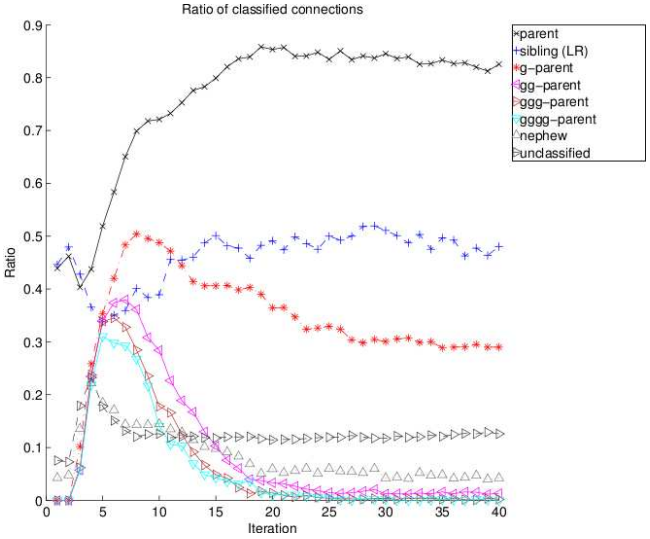
Some search-and-score techniques, such as the *K2* algorithm, require that we provide an order for the nodes. An edge can appear between nodes A and B only if A precedes B in the order. If we also limit the maximum number of parents a node can have to a constant k, finding the optimal network *consistent with the given order* becomes polynomial in n, and thus tractable if k is small. Note that providing a node order means that we are unable to find potentially better-scoring networks that are *inconsistent* with that order. In our case, specifying a good node order is not obvious. For example, if we impose a left-to-right depth-first-search order, then a node in the tree cannot have an edge from its right sibling or its right uncle. If we impose a left-to-right breadth-first-search order, a node in the tree cannot have an edge from any node to its right (at the same height) or below it.

We wanted to discover the dependency structure in our distributions, so we aimed to impose as few restrictions as possible on the types of networks we could learn. Thus, we opted for a search-and-score algorithm that did not require a node order as input [1]. This algorithm uses an evolutionary approach to search for good node orders. To evaluate a given order, the algorithm uses advanced caching techniques to compute the optimal network consistent with that order. The only restriction we impose is that a node can have at most 3 parents. This is reasonable, because data fragmentation prevents us from learning Bayesian networks with large parent sets anyway [6].

The complexity of structure learning increases with the number of nodes (63 in our case) and the size of their support (at most 9 in our case). This is a fundamental limitation in applying graphical-model-based estimation techniques in an EDA-GP with a prototype tree, because a new structure has to be learned in every iteration. For this reason, existing EDA-GP algorithms such as POLE place additional



(a) Connection ratios



(b) Connection ratios

Figure 4: Plot 4(a) with error bars of the average number of connections per node. The nodes are ordered in depth-first left-right. The color shows which level the node is on. Plot of the ratios of connections in different types in 4(b)

constraints on the types of structures that can be learned, thus radically reducing the search space.

3.2 Analyses of the graphical models

We ran the standard GP for 40 generations and ran 30 trials of this run. We built Bayesian network for the population from each generation and each run. In this section we present different summaries of the Bayesian networks achieved from a standard GP run for a Pagie-2D problem. This is possibly the first attempt to explain evolution statistically. Our hope is to uncover the correlation of statistical variations with fitness of the population and isolate the statistical patterns that are being caused by evolution.

In Fig. 3 we show the graphical models for a GP population at iteration 0, 13, 25 and 40 ($G_0, G_{13}, G_{25}, G_{40}$). The GMs all have different layouts, but as the iterations progress the variations in connections between the iterations are decreasing. The initial G_0 does find many connections, and none from the root. In contrast, G_{13} has more connections, especially from the root. The G_{25} has fewer connections and not as "long", i.e. spanning multiple levels. Finally, the G_{40} has stabilized in the number of connections, and connection type.

We also observed that the total number of connections are changing over the generations. Initially there are around 70 connections, while there is a peak of roughly 180 connections around iteration six. Then the number of connections tails of and stabilizes to around 130, which is roughly the same iteration when the fitness stops improving. Thus, implying that the population has started to converge.

In Fig. 4(b) the connections are classified according to a number of types: parent, sibling (a sibling is the rightmost child of a parent), grand-parent, grand-grand-parent, grand-grand-grand-parent, grand-grand-grand-grand-parent, uncle, and unclassified. The plot shows that after iteration 20 almost all nodes have a connection with their ancestor and half has to their left sibling. The ratio is calculated as the number of connections in the graph of the type and the possible number of connections of the type, e.g. for parent there are 62 possible connections, a ratio of 0.5 means that there are 31 connections in the Bayesian network. After 25 iterations the ratios of the other connections have somewhat stabilized. The figures show that ancestor and sibling relations are captured by the Bayesian network, as well as some grand-parent. This is an observation that could be taken into account when designing operators in GP, or reduced Bayesian network for EDA-style GP.

From Fig. 4(a) the average number of connections per node over the generations are shown. The number of connections are normalized according to the possible number of connections, 62. The nodes are in depth-first left-right order. It is possible to see a pattern over the number of connections given the position of the node in the tree. As expected the nodes at the lower level have fewer connections. The classifications of connections, Fig. 4(b), explains what type many of these connections are, e.g. parent and sibling dependencies.

In Fig. 5 a tree with the total connections from all runs and all iterations. We only present the connections that appeared at least 10% of the times in the 30 runs where each run was for 40 generations. The size and color of the edge denotes strength, red is strongest. The connections show the position of the connection types ratios displayed in Fig. 4(b) and reiterates the strong parent connections found.

4. SCALABLE OPERATOR FREE GENETIC PROGRAMMING

Having observed different dependence patterns that emerged during the graphical model analyses from 30 independent runs of GP we now focus on developing an EDA-GP that does not require a prototype-tree. We clearly observe that parent and grand-parent relations are the most important. We remind the reader that our goal is to use a smaller and

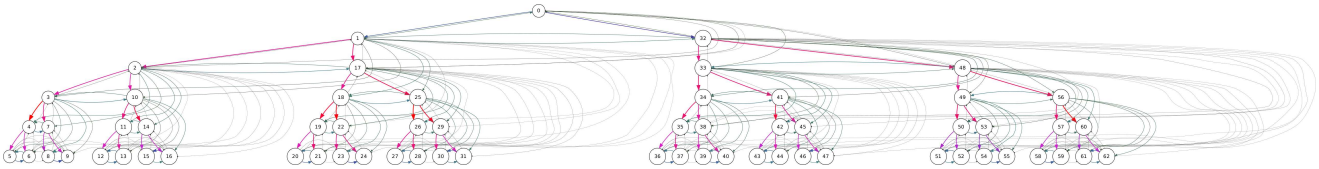


Figure 5: Tree with the total connections from all runs and all iterations (cut-off at 0.1). The size and color of the edge denotes strength, red is strongest.

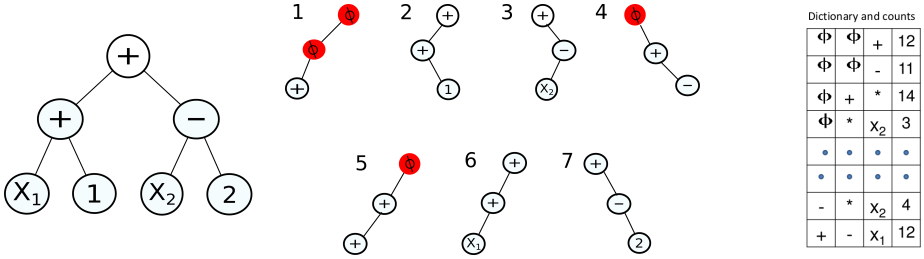


Figure 6: Extracting n-grams for n=3 from a GP tree.

scalable probabilistic model defined over a very small pattern and recursively sample from this model.

Hence we propose an n-gram model over ancestor chains since they represent the link between a node and an important aspect of its context i.e. *parent/ancestors*. n-grams are usually used in linear contexts and are touted for their relative simplicity. To be able to use an n-gram model in a tree-structured context and to be able to sample trees from them we need two components whose design we will describe in the following subsections, leading up to the pseudo code for our OFGP algorithm shown in Algorithm 2. These two components are:

1. Generation of n-gram counts from the GP trees (Sect. 4.1)
2. Tree creation via recursive sampling from the n-gram distribution (Sect. 4.2)

4.1 n-grams model over ancestor chains

n-grams model the probability distribution for the random variable Y_i for a node i (numbered in the depth first search manner) in the tree given the values for its parent Y_{i-1}, grandparent Y_{i-2}, and so on up to the $(n-1)$th ancestor:

$$P(Y_1 \dots Y_N) = \Pi_{i=1}^{N} P(Y_i | Y_{i-1} \dots Y_{i-n-1}) P(Y_{i-1} \dots Y_{i-n-1}) \quad (1)$$

For every node in a tree, the node and its $(n-1)$ ancestors constitute one n-gram observation. For example consider the GP tree shown in Figure 6. For $n = 3$, we extract 7 different trigrams as shown in the figure. Near the root, grandparent and/or parent nodes do not exist: in this case they are represented by the null node ϕ.

We enumerate the possible trigrams as a dictionary, Δ, in which each entry is a 3-tuple (grandparent (Y_{k-2}), parent (Y_{k-1}), node (Y_k)) and the final colum represents the observed counts of the tuple across the population of trees (see Figure 6 for an example dictionary. In the example we increment the counts for the 7 trigrams in our dictionary. We convert these counts into frequencies and rely on them when generating a trigram.

We use this model to fill up the content in an empty tree T of a prespecified size N, where N represents the number of nodes in the tree (for simplicity let us consider a binary tree).

Algorithm 1 Recursive sampling algorithm

```
1: function GENERATE(Δ, T, N)
2:     Evaluate the frequencies for different n-grams
3:     k = 1
4:     Choose root Y_k ~ P(Y_k | Y_{k-1} = φ, ... Y_{k-n-1} = φ)
5:     k ← k + 1
6:     while k < N do
7:         if Y_k = LEAF then
8:             Choose a T Y_k ~ P(Y_k | Y_{k-1}, ... Y_{k-n-1})
9:         else
10:            Choose a F Y_k ~ P(Y_k | Y_{k-1}, ... Y_{k-n-1})
11:        end if
12:        k ← k + 1
13:    end while
14:    return T
15: end function
```

4.2 Operator Free GP

The Operator Free GP algorithm is described in Algorithm 2. The algorithm takes the n-gram counts table Δ, initial preferred tree size, l, size of the population, S_Θ, truncation ratio α as inputs. A population of trees are generated of sizes, $l \in [2l/3, 4l/3]$ using a standard algorithm that generates randomly-shaped unlabelled binary trees of precise size, based on the idea of binary search trees[1]. It begins with an empty tree. At each step, a new node is created and associated with a random number r (e.g. chosen uniformly from $[0, 1]$). This node is "bubbled down" through the existing tree, moving *left* of those nodes which have $r' > r$ and moving *right* otherwise, eventually reaching a leaf position. This process is repeated until the tree has the desired size. The algorithm can also be adapted to produce trees with *labeled* nodes of *variable* arity in two steps.

[1] `http://en.wikipedia.org/wiki/Random_binary_tree#Binary_trees_from_random_permutations`

Algorithm 2 OFGP algorithm

```
1: Δ ← uniform_initialisation()
2: α ← 0.2                                   ▷ truncation ratio
3: for i < iter do
4:     Θ ← ∅
5:     while |Θ| < S_Θ do
6:         T ← BUBBLE-DOWN (l)
7:         Θ ← Θ∪ GENERATE (Δ,T, l)
8:     end while
9:     EVALUATE (Θ)
10:    SORT (Θ)
11:    Θ_s ← TRUNCATE (α, Θ)
12:    Δ, l ← UPDATE (Θ_s)
13: end for
```

An unlabeled tree is then passed to the GENERATE function presented in Algorithm 1. Each nodes label in this tree is determined according to its ancestors and the observed frequency of n-gram ancestor chains (available in Δ). The mean size s of high-fitness trees can also be observed, and new trees can be generated in a distribution centered around s.

The population is then evaluated and the best 20% are retained (truncation selection). Their n-grams are extracted as shown in Figure 6 and Δ is updated by updating the counts. Their sizes are also calculated. A new preferred size l is now set to the mean of this best 20%. A new population is generated using the same algorithm as before, with the only differences being the new value of l and the fact that the n-grams are used to fill-in the terminals' and non-terminals' labels after the tree-structure is determined.

5. EXPERIMENTS & RESULTS

We now compare the performance of the OFGP algorithm with standard GP. In both standard GP and OFGP algorithms, the population size was 2000 and the number of generations 40. Fitness was calculated as the root mean square error against the target function. In standard GP the maximum depth was 17 (no such parameter is needed for OFGP). In OFGP, three sets of runs using n-gram size $n = 3, 4, 5$ were performed.

Results are shown in Fig. 7, best of run values for OFGP are shown in the comparison with GP. It is clear that standard GP achieves more reliable results, getting a fitness of below 0.40 in the majority of the 30 runs. Standard GP almost always retains its best results from one generation to the next, even in the absence of elitism. By contrast, OFGP often finds a good result but then abandons it, leading to erratic-seeming behaviour over the generations. Nevertheless it is clear that OFGP achieves fitness better than 0.40 in relatively few of the 30 runs, for all values of n. The main advantage of OFGP is that what results it achieves are achieved with exceptionally small and readable trees, often of just 20-30 nodes (result not shown but data available). In contrast standard GP tends to produce best individuals of over 100 nodes. OFGP with n-gram size $n = 4, 5$ produce better results than $n = 3$. Between $n = 4$ and $n = 5$ the performance is similar.

Though this represents the feasibility of a scalable approach to EDA-GP. However, we hypothesize that the distribution used in OFGP (simple n-grams) fails to capture the most important aspects of the highly-fit trees, regardless of the n-gram size. We note that sibling relations also are important to form highly fit trees.

6. CONCLUSIONS & FUTURE WORK

Our goal through this paper was to develop a methodology in which an estimation of distribution of patterns (rather than a prototype tree) could be used to generate a GP tree. We wanted to use this to sample, select and re-sample forming an EDA-GP. The methodology is key to reducing complexity and intractability that a traditional EDA-GP encounters. The distribution of patterns have only few random variables, i.e. for n-grams it is only limited to either 4 or 5. These are easy to learn during a GP run and are easy to sample from. Although we are modestly successful, much of the work remains to identify patterns that OFGP should track.

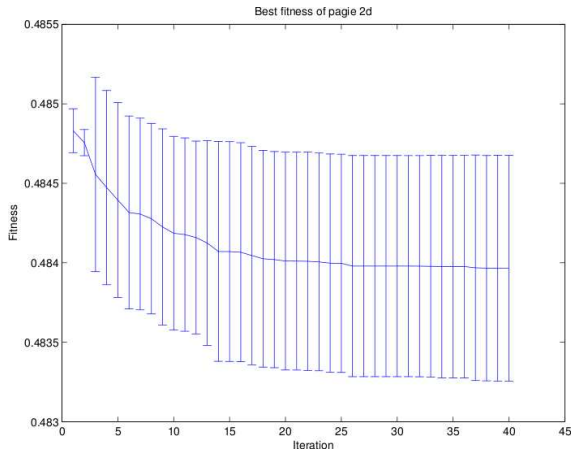
We can add patterns like subtree expression which will capture parent-child-sibling relationships. If we choose to track and estimate distributions for multiple patterns we then have to design a tree sampling algorithm that uses these distributions in some probabilistic fashion. We can bias the choice via the choices that had lead to higher fitness trees in the previous generations. From a probabilistic standpoint this could be a mixture model for multiple pattern distributions.

We would also like to investigate our intuition that patterns are spatially dependent (i.e. dependent on the level of the tree in which they appear). If they are spatially dependent, we will develop a tree creation algorithm that takes pattern depth into account. This would imply that we would estimate a different distribution for patterns at different levels. Much needs to be investigated in the statistical machine learning area to find equivalent probabilistic models that could be used. Foundationally we are driven by making the learning part within the GP generation as simple and tractable as possible.

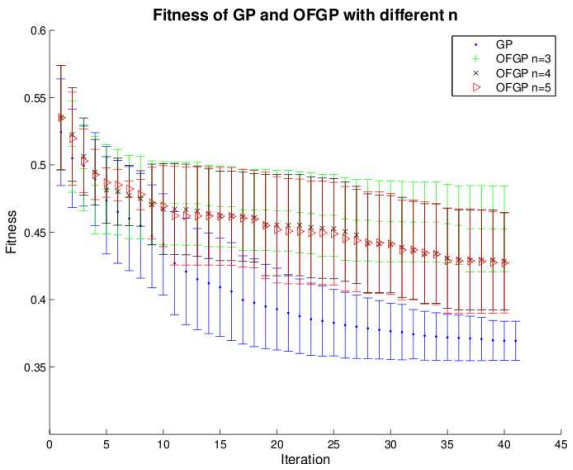
To motivate our quest for which patterns to choose we analyzed a traditional GP run and learnt a graphical model for the population that it generates. We gained many interesting insights. However, we have thus far only focused on analyzing, via a bayesian network the conditional dependence that arises for Pagie-2D. We intend to next consider other GP problems while using the same set of functions and terminals to see if the patterns differ. We will also try different operators with GP to see if other patterns emerge for the same problem. It may be that the patterns have a strong relation to the semantics of the functions and terminals in which case they'll be robust across problems. During these exercises, we intend to be mindful that running GP with variation operators to discover patterns out is nonetheless imperfect because GP's operators may induce inappropriate patterns.

Acknowledgment

Erik thanks the support by the Science Foundation Ireland under Grant No. 08/IN.1/I1868. James is funded by IRCSET, co-funded by Marie Curie. Kalyan and Una-May thank the support received from General Electric Global Research Center and Li Ka Shing Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessar-



(a) GP best fitness, depth=5



(b) GP vs OFGP

Figure 7: The Fig. 7(a) shows the best fitness over the generations for GP with depth=5. In Fig. 7(b) the performance of standard GP and OFGP on the Pagie 2D problem is shown. Values of n refer to n-gram size, i.e. the number of ancestors taken into account when generating a node. Best of run values for OFGP are shown in the comparison with GP

ily reflect the views of General Electric Company or Science Foundation Ireland or Li Ka Shing Foundation.

References

[1] Berzan, C.: An Exploration of Structure Learning in Bayesian Networks. Tufts University Senior Honors Thesis (2012)

[2] Harper, R.: Spatial co-evolution in age layered planes (SCALP). In: CEC. IEEE (2010)

[3] Hasegawa, Y., Iba, H.: Estimation of Bayesian network for program generation. In: Proc. 3rd Asian-Pacific Workshop on Genetic Programming. p. 35 (2006)

[4] Hasegawa, Y., Iba, H.: A Bayesian network approach to program generation. Evolutionary Computation, IEEE Transactions on 12(6), 750–764 (2008)

[5] Keller, R.E., Banzhaf, W.: The evolution of genetic code in genetic programming. In: Banzhaf, W., et al. (eds.) GECCO. pp. 1077–1082. Morgan Kaufmann, San Francisco, CA (1999)

[6] Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)

[7] McDermott, J., Galván-Lopéz, E., O'Neill, M.: A fine-grained view of phenotypes and locality in genetic programming. In: Riolo, R., Vladislavleva, K., Moore, J. (eds.) Genetic Programming Theory and Practice IX. Kluwer (2011)

[8] Nguyen, Q.U., Nguyen, X.H., O'Neill, M.: Semantic aware crossover for genetic programming: The case for real-valued function regression. In: EuroGP. pp. 292–302. Springer (2009)

[9] Pagie, L., Hogeweg, P.: Evolutionary Consequences of Coevolving Targets. Evolutionary Computation 5, 401–418 (1997)

[10] Pelikan, M.: Hierarchical Bayesian optimization algorithm: Toward a new generation of evolutionary algorithms. Springer (2005)

[11] Poli, R., McPhee, N.F.: A linear estimation-of-distribution GP system. In: EuroGP. pp. 206–217 (2008)

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

[13] Sastry, K., Goldberg, D.: Probabilistic model building and competent genetic programming. Genetic Programming Series 6, 205–220 (2003)

[14] Shan, Y., McKay, R., Essam, D., Abbass, H.: A survey of probabilistic model building genetic programming. Scalable Optimization via Probabilistic Modeling pp. 121–160 (2006)

[15] Shan, Y.: Program Distribution Estimation with Grammar Models. Ph.D. thesis, University of New South Wales (2005)

[16] Wilson, G., Heywood, M.: Introducing probabilistic adaptive mapping developmental genetic programming with redundant mappings. Genetic Programming and Evolvable Machines 8(2), 187–220 (2007)

[17] Yanai, K., Iba, H.: Estimation of distribution programming based on Bayesian network. In: CEC 2003. vol. 3, pp. 1618–1625. IEEE (2003)