Introducing Graphical Models to Analyze Genetic Programming Dynamics

Erik Hemberg
MIT CSAIL
hembergerik@csail.mit.edu

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

Kalyan Veeramachaneni
MIT CSAIL
kalyan@csail.mit.edu

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

ABSTRACT

We propose graphical models as a new means of understanding genetic programming dynamics. Herein, we describe how to build an unbiased graphical model from a population of genetic programming trees. Graphical models both express information about the conditional dependency relations among a set of random variables and they support probabilistic inference regarding the likelihood of a random variable's outcome. We focus on the former information: by their structure, graphical models reveal structural dependencies between the nodes of genetic programming trees. We identify graphical model properties of potential interest in this regard – edge quantity and dependency among nodes expressed in terms of family relations. Using a simple symbolic regression problem we generate a graphical model of the population each generation. Then we interpret the graphical models with respect to conventional knowledge about the influence of subtree crossover and mutation upon tree structure.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

genetic programming, graphical models, Bayesian network

1. INTRODUCTION

We begin by noting that the population of a GP run can be regarded as observed stochastic samples of a set of random variables where the stochasticity injected into the samples changes over time (generations) due to the evolutionary operators of selection, mutation and crossover. Alternatively, but equivalently, the GP population each generation can be regarded as observed samples of a different statistical population. Generally, a GP run is the outcomes from a set of random variables, spaced in time at the interval of generations.

The population dynamics of genetic programming are still not completely understood (15), i.e. how the dependencies between the solutions vary in the population and how they changes over time (generations). Using the insight that each genetic population of a GP generation is a set of samples, our aim is to model the statistical population from which they are drawn as a distribution of random variables. We will then analyze the changes in each distribution as the generations proceed. Previously, analyzing a GP run's dynamics has focused on studying the distribution of tree size, tree depth or solution fitness as probability distributions or simply calculating the moments of this distribution like mean and standard deviation (17). In contrast, our method is a probabilistic statistical approach.

We examine how to build a multivariate distribution Π which is represented by a set of samples (a generation) of GP trees. We use a prototype tree which is the size and shape of the largest possible GP tree for the problem. Each node of the prototype tree represents a random variable. GP trees are mapped onto the prototype tree to become samples of the multivariate distribution, see Fig. 1. We use a probabilistic graphical model to represent this multivariate distribution. (In a graphical model a directed acyclical graph denotes the conditional dependences between random variables.) To efficiently build Π requires accommodating GP trees which are of variable sizes and shape, identifying the best possible representation of the graphical model (GM) given a set of trees, and devising computational efficiencies which allow the building of the graphical model to be tractable.

In the last two decades building graphical models and associated inference engines has been addressed by numerous researchers (7; 11). We explored a variety of techniques (which lead to different graphical models) from this body of literature to address the challenges graphical model building in the GP context specifically poses. We present one technique in Section 3. It is chosen because it is unbiased with respect to any assumptions about variable dependency.

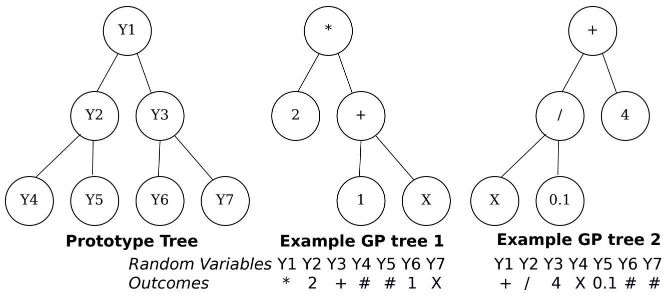


Figure 1: A prototype tree of depth d = 2, two example GP trees, and, as a result of alignment with the prototype tree, their random variable outcomes. Note that we do not show # *(null)* nodes.

Once we describe computationally efficient means to build a probabilistic graphical model we proceed to analyze how the generational graphical models transition over the course of a run as a result of selection and crossover. Specifically, we set up a very simple GP symbolic regression problem (Pagie-2D (16)) to execute over a fixed number of generations. Then we execute experiments composed of GP runs which vary the presence or absence of different operators. For each set of runs with a particular variation operator (only crossover or only mutation), we then aggregate all the trees of a generation and treat this as a set of samples from which we build a graphical model. We analyze the changes in the resulting graphical models as we process one generation after another.

The paper's contributions are:

1. Use of probabilistic graphical models for modeling structural dynamics of a population of GP trees

2. Description of an efficient technique to build graphical models. The technique needs to be fast because our aim is to to build graphical models for the population from every generation. The technique further needs to be unbiased to permit the data to be as accurately reflected as possible.

3. Analyzes of GP tree dynamics in different variational operator scenarios with respect to structural dependencies mapped from graphical models.

We proceed in the following manner: In Section 2 we present an overview of current techniques that analyze GP populations. In Section 3 we present how we resolve the issues arising in how to build a graphical model for a GP population. We also present how we have used a genetic algorithm to efficiently learn the structure of a Bayesian network from a fully observed data set. In Section 4 we present the results for a first set of statistical analyzes that reference generational graphical models. We present our conclusions and future work in Section 5. The Appendix describes the building of the graphical model in more detail.

2. BACKGROUND

Many different statistics have been used to study GP. Widely used and reported, "conventional" analyzes include a time series plot, per generation, of mean, standard deviation, maximum and minimum size of structure (e.g. tree, graph or other executable representation) or population fitness. Tomassini et al. (21) study fitness distance correlation as a difficulty measure in genetic programming. Langdon and Poli (12) analyze the behavior of GP on a wide range of problems, e.g. artificial ant and the Max Problem using schema analysis and Price's covariance and selection theorem.

Nearly all visualizations of structure have been for illustrative purposes according to Daida et al. (5), who visualized tree structures in GP. This new way of "seeing" can afford a potentially rich way of understanding dynamics that underpin GP. Almal et al. (2) perform a population based study of evolutionary dynamics in GP and show that heat maps are a useful tool for exploring the dynamics of genetic programming. The visualization allows the user to draw conclusions from data. We expand on this by inducing a graphical model to both visualize and explain the structure and content of a GP tree.

Daida et al. (6) study phase transitions in GP search to support a statistical mechanics approach where GP would be quantitatively compared to a broad range of other systems rigorously described by phase transitions. In order to identify subtrees that carry out similar semantic tasks, Wolfson et al. (22) perform post-evolutionary analysis of GP by defining a functionality-based similarity score between expressions. Similar sub-expressions are clustered from a number of independently-evolved fit solutions, thus identifying important semantic building blocks lodged within the hard-to-read GP trees.

Raidl and Gottlieb (18) empirically investigate locality, heritability and heuristic bias in Evolutionary Algorithms by generating individuals with static measurements based on randomly created solutions. This is done in order to quantify and visualize specific properties, and shed light onto the complex behavior of the system. Here, we take a different approach and we investigate the structure and contents of a GP tree by creating a graphical model from a population of GP trees.

In a paper by McPhee and Hopper (13) they analyze genetic diversity through population history by size and showed that the genetic diversity was low. They measure the number of different nodes in the population based on if the nodes begin at the root or non-root. Our study differs by investigating the dependencies between the nodes in the solutions based on a graphical model from the solution. Thus, we do not need to identify any relationships between individual solutions in the population, instead we take a statistical approach and learn a model of the population.

3. GRAPHICAL MODELS FOR GP

Our goal is to model the multivariate distribution of the trees in the GP population in generation t. We denote this by Π_t. The following issues are resolved in order to build a multivariate distribution for the population:

GP trees are of variable size: Because a population's trees vary in size and structure and GP's representation is not homologous, the question of defining the random variables is open to question. We choose a prototype tree based approach in which each node in a tree with a maximum size is a random variable and its support is all possible functions and terminals that could be at its position. We present more details in Section 3.1.

Defining the random variables in this manner provides us with a fixed set of variables for the multivariate distribution.

Modeling the multivariate distribution: Our representation of the multivariate distribution needs to allow us to gain insight from it. We choose a Bayesian network based representation which allows us to identify directed dependencies between the nodes of GP trees. We provide more details in Section 3.2.

Efficient building of the distribution: Given our choices of a prototype tree for the random variable set and a Bayesian network model, a further challenge is to build the distribution efficiently.

3.1 GP trees to random variables

To define the random variables for the modeling distribution we adopt a prototype tree based approach, see Fig. 1. Prototype trees have been used to represent GP trees when developing estimation of distribution based GP (EDA-GP), e.g. Salustowicz and Schmidhuber (19) and Hasegawa and Iba (8). A prototype tree is the size and shape of the largest (i.e. completely full) tree in the GP experiment. Each node in the prototype tree designates a random variable whose support (possible outcomes) depends on its position in the tree. That is, the root of the prototype tree, which can assume any function, is a random variable with its support being the problem's function set because each function is a possible outcome for the root's random variable. Each leaf of the prototype tree, which can only assume a terminal, is a random variable with support comprising the problem's terminal set (because each terminal is a possible outcome of a leaf's random variable). For each interior node, support is the set of all functions and terminals because each terminal or function is a possible outcome of an interior node's random variable. Because any GP tree may be smaller than the experiment's prototype tree, an extra outcome, *null*, is added to the support at every node except the root.

For example, for a symbolic regression problem with function set $\{+, -, /, *\}$ and terminal set $\{x_1, x_2\}$, prototype tree leaves have a support of two terminals and *null*, the root has a support of four functions, and all the intermediate nodes have a support of six outcomes (function set, terminals and *null*). The number of random variables in the multivariate distribution at time t, Π_t depends on the size of the prototype tree. The fundamental challenge in using a prototype tree based representation is supporting a large prototype tree so that every possible size or shape of a GP tree within a run can align with the prototype tree.

The GP tree population at a generation is a set of samples. To transform a tree to a sample, it is aligned with the prototype tree from the root downwards. Each function or terminal of the sample is counted as an outcome of the random variable onto which it maps in the prototype tree. Where the sample tree has no nodes, *null* outcomes are assumed. Figure 1 illustrates the transformation of a set of GP trees to outcome statistics of the prototype tree's random variables. As a graphical model for Π_t is built for each generation, the population of GP trees is transformed to samples by alignment with the prototype tree, and the outcomes of the random variables are statistically accumulated from each sample.

We choose a Bayesian network representation for the graphical model because it allows us to analyze the changes in the multivariate distribution via its graphical properties. Next, we describe this representation.

3.2 Bayesian networks as a graphical model

We can associate the population in standard GP with an implicit probability distribution over the random variables in the prototype tree. 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, consult (11) for an introduction to Bayesian networks. 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 $Y_1, \ldots, Y_n$. The Bayesian network representation has two components. A directed acyclic graph (DAG) $\mathcal{G}$ encodes (conditional) independence relations between random variables. More precisely, a missing edge encodes an independence relationship. Each random 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:

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

Learning Bayesian networks from data has received much attention in the machine learning community (11). 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 (11). 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.

3.3 Learning network structure

Some search-and-score techniques, such as the *K2* algorithm (4), require that we prepare an edge order for learning the parameters of the nodes. Given an order, an edge can appear between nodes A and B only if A precedes B in the order. In our case of prototype trees for GP, specifying an accurate node order is not obvious. For example, if we impose a left-to-right depth-first search order, then a node in the graphical model cannot have an edge from its right sibling or its right uncle. In GP terms, this implies we would not sample relationships among the functions and terminals which are parameters of a parent function, in a right to left order. In Fig. 2 the possible dependencies in left-to-right depth-first order is shown, the dashed arrows are impossible dependencies given the ordering. If we impose a left-to-right breadth-first search order, a node in the graphical model cannot have an edge from any node to its right (at the same height) or below it. *K2* is commonly used in EDA-GP (8) where the node order is assumed to be depth-first search yet its assumptions are not necessarily valid.

Because we wanted to discover the true dependency structure in a population's distribution, we aimed to impose as

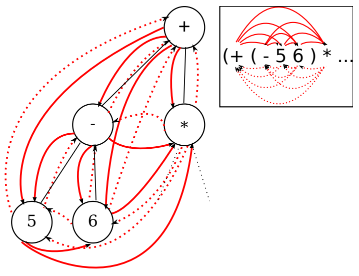


Figure 2: Shows order bias, with possible dependencies in left-to-right depth-first order depicted by the arrows, the dashed arrows are impossible dependencies given the ordering. In the square the string representation of the tree, with the same possible and impossible dependencies, is shown.

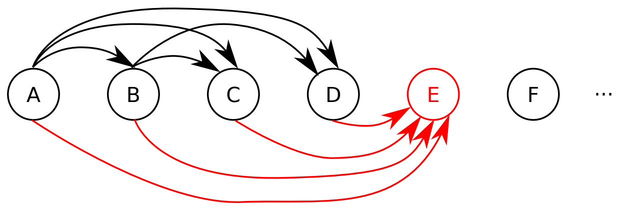


Figure 3: Possible structure with known node order. Suppose the node order is $A, B, C, D, E, F, \ldots$. This means that every edge in the network must go from left to right. Suppose we have already found the best parent sets for nodes A, B, C, and D, and are now considering node E. Each of E's parents must precede E in the order. We can add any of the edges shown in red without introducing a cycle. Furthermore, the parents we choose for E will not limit the choice of parents for nodes F and beyond.

few restrictions as possible on the type of graphical model structure 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 algorithm in a search-and-score approach to propose and search for node orders leading to an optimal network structure. For each order the EA proposes, the algorithm then identifies an optimal network structure. This requires evaluating every structure that is consistent with that order. Since in our approach each individual is an order, this identification occurs during the evaluation of an individual's fitness. While the identification is computationally expensive, the number of structures (DAGs) that are consistent with the order is finite. In Section A, we present a scalable approach to systematically overcome the computational burden.

When learning the order of a DAG, parents of any node Y must precede Y in the ordering. In other words, given a node ordering R, such that for any nodes X and Y, if $X \overset{R}{<} Y$ (X precedes Y in R), then any edge between X and Y must be directed from X to Y. The ordering constraint R allows the choice of parents for one node to be independent of the other nodes' parents. This is because any combination of edges consistent with the ordering R produces an acyclic graph. Figure 3 illustrates this observation.

With a decomposable score this makes the choice of parents for any node X independent of the choice of parents for any other node Y. A decomposable score allows us to evaluate parent sets for each node independently and select the best one for each node and total score of the entire network is the total sum of all the scores for each node. It is apparent that the minimum across each node's family score minimizes the overall score as well. This means that now we can search for each node's parent set independently. If we limit the indegree (number of parents) of every node to a constant $k < n/2$, then there are at most

$$1+\binom{n-1}{1}+\ldots+\binom{n-1}{k} = O\left(k\cdot\binom{n-1}{k}\right) = O(n^k)$$

possible parent sets for each node, where n is the number of nodes in the network. Finding the optimal parent set for all nodes thus requires at most

$$n \cdot O(n^k) = O(n^{k+1})$$

parent sets to be evaluated, which is polynomial in n, for a fixed k. This means that we can exhaustively search over the space of networks that are consistent with an ordering R and have an indegree of at most k.

Of course, an $O(n^{k+1})$-time exhaustive search is only reasonable if k is small (up to about 3 or 4). This restriction is not as bad as it seems, because we cannot learn Bayesian networks with large parent sets anyway, unless we have an extremely large training set. To see why, consider a Bayesian network where all variables are binary. If a variable X has no parents, then to compute the Bayesian score we only need two counts from the training set: $M[X = 0]$ and $M[X = 1]$. If X has one parent Y, then we need four counts: $M[X = 0, Y = 0]$, $M[X = 0, Y = 1]$, $M[X = 1, Y = 0]$, and $M[X = 1, Y = 1]$. If X has two parents, we need eight counts, and so on. In general, if X has k parents, we need a total of 2^{k+1} counts. As the number of parents Pa_X increases, it becomes less and less likely to find representative counts for all possible assignments $M[\mathrm{Pa}_X = \mathrm{pa}_X, X = x]$ in the training set. This phenomenon of *data fragmentation* means that we can only learn networks where each node has a small numbers of parents. Thus, setting the maximum indegree k to a small value is justified.

4. EXPERIMENTS

In this section we start by describing our choice of demonstration problem (Section 4.1), our procedure for transforming population data into graphical models (Section 4.2) and how we set up a number of experiments (Section 4.3) for which we build a graphical model every at generation. By their structure, graphical models reveal structural dependencies between the nodes of genetic programming trees. In Section 4.2 we identify graphical model properties of potential interest in this regard – edge quantity and dependency among nodes expressed in terms of family relations. We then interpret each experiment in light of the operator it has used and the information the graphical model conveys. This section contains an extended analysis of experiments initially done in (9) and (10).

4.1 Demonstration Problem

Because our goal is to figure out the nature of insights a graphical model offers in terms of its dependency struc-

Table 1: Parameters for the GP runs in ECJ. *XO* is crossover, *Mut* is mutation, *No ops* is selection without variational operators and *No Sel* is crossover with random selection.

Parameter	*XO*	*Mut*	*No ops*	*No Sel*
Crossover	0.9	0.0	0.0	0.9
Mutation	0.0	1.0	0.0	0.0
Selection	Tournament size 7			Random
Population size	10,000			
Generations	40			
Initialization	Ramped Half-Half			
Max Depth	5			
Language	Symbolic Regression			
Functions ($\mathcal{F}$)	$\{+, -, *, \%\}$			
Terminal ($\mathcal{T}$)	{x,y, 0.1, 1.0}			

ture, for our experiments we choose Pagie-2D (16) which is a simple but sufficiently realistic symbolic regression problem with a modestly sized set of functions and terminals. The problem is to regress an expression matching the target function $1/(1 + x^{-4}) + 1/(1 + y^{-4})$ over 676 fitness cases in $[-5, 5]^2$, and fitness is calculated as the mean squared error. The GP parameters are summarized in Table 1, and % is protected division. We run standard GP on the Pagie-2D problem using tournament selection with a tournament size of 7 and ramped half-half initialization. We employ a prototype tree of depth 5, which has 63 nodes implying an equivalent maximum tree size for the Pagie-2D runs. Our runs execute for 40 generations, using ECJ[1], with a population size of 10,000.

4.2 Graphical Model Building Procedure

After each generation, we store the entire population to disk so we can build its corresponding graphical model. We start by converting each population into a data set $\mathcal{D}$ with one row for every tree and 63 columns, one per outcome of the 63 random variables. We represent each tree as a row of 63 outcomes, by aligning it to the prototype tree, starting from the root, with a depth-first order traversal and substituting *null* for any node of the tree not in the prototype tree.

The root can take any outcome from the set of functions: $S_{\text{root}} = \{+, -, *, \%\}$. The leaves can take any outcome from the set of terminals, as well as *null*: $S_{\text{leaf}} = \{x, y, 0.1, 1.0, \mathit{null}\}$. Intermediary nodes can take any outcome from the set of functions and terminals, as well as *null*: $S_{\text{intermediary}} = \{+, -, *, \%, x, y, 0.1, 1.0, \mathit{null}\}$. Thus, our data set $\mathcal{D}$'s columns are 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, as described in Section 3, a graphical model for the distribution $P(Y_1, Y_2 \ldots Y_{63}|\mathcal{D})$.

4.3 Experiment Definition and Method

We define three experiments which vary in terms of GP operator in order to detect differences in this respect:

No variational operators Neither crossover nor mutation is used, selection only acts on the population.

Crossover Single point subtree crossover is the only variational operator and it is used in combination with selection.

Mutation Subtree mutation is the only variational operator and it is used in combination with selection.

No selection Single point subtree crossover is the only variational operator and selection is random.

Each experiment consists of 30 independent runs. For each run we build a graphical model each generation. This results in 1200 graphical models per experiment and 4800 graphical models overall.

4.4 Study of Graphical Model Properties

This section outlines the properties of the graphical model which is studied as well as the results from the different experimental setups. The aim is to identify properties which can explain the dynamics of GP populations.

4.4.1 Edge Quantity

Fig. 4 shows an example graphical model for a GP population at generation 25 (G_{25}), taken from a run using crossover. In addition to studying graphical models at different generations directly, a graphical model structural property worthy of attention is the number of edges in the graph. In graphical model terms, because an edge from Node A to Node B implies that the likelihood of random variable B depends on random variable A, edge quantity reveals the degree of variable inter-dependence. Edge quantity conveys different information from previous work in GP where, by simple counting, the frequency of symbols is determined or even the frequency of symbols in specific tree locations is determined. Simple counting does not reflect dependencies which the graphical modeling mines from the population by treating it as samples and using max-likelihood estimation. Given that a graphical model supplies dependency information, these dependencies should then be mapped back to the prototype tree to provide interpretation in a GP context. That is, each random variable of a graphical model maps to a node of a prototype tree which in turn maps to the symbol set, i.e. the function and terminal sets, of a GP experiment.

We can formulate one simple hypothesis: a random population of GP trees would result in a graphical model where there are arbitrary dependencies (i.e. edges) between nodes compared to a population evolved later in a run. The later population has been "shaped" by selection and variation so it will be reflected by a structure in the dependencies. These dependencies will reflect the complex mapping between genotype and phenotype, i.e. the direct representation of the solution and its behavior. Fig. 5(a), with data aggregated over each of the four experiments, shows the change in edge quantity over generations. It provides one unit of evidence for the hypothesis. In the initial populations' graphical model there are few edges, then, due to selection, the number sharply rises regardless of whether variation is employed. When a variational operator is employed, subsequently the edge quantity diminishes and eventually tapers. This can also be seen for the run without selection, where the rise is significantly smaller. We speculate that this dynamic will generally remain the same with other problems due to the macro-influences of the operators but will vary in detail due to the unique fitness landscape of every problem.

[1]`http://cs.gmu.edu/~eclab/projects/ecj/`

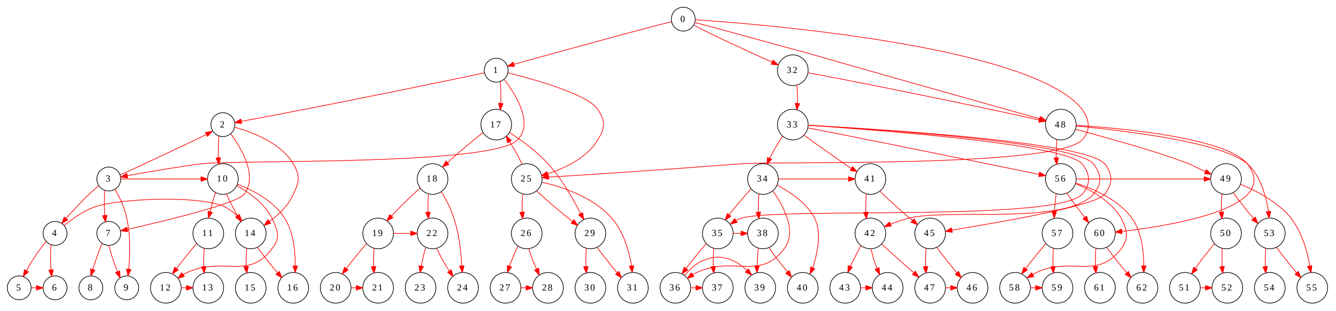
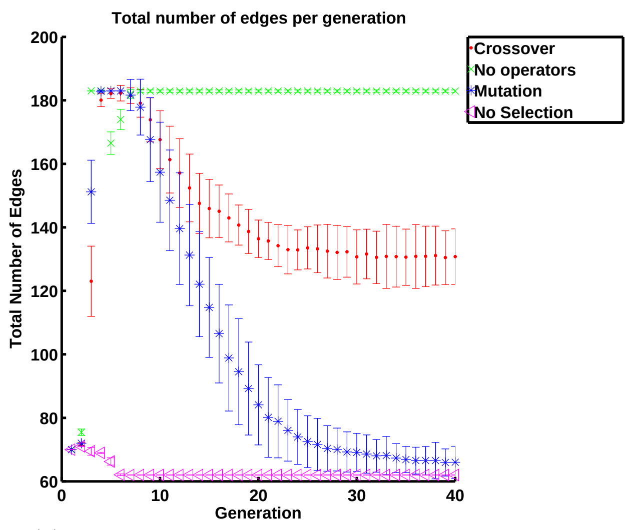
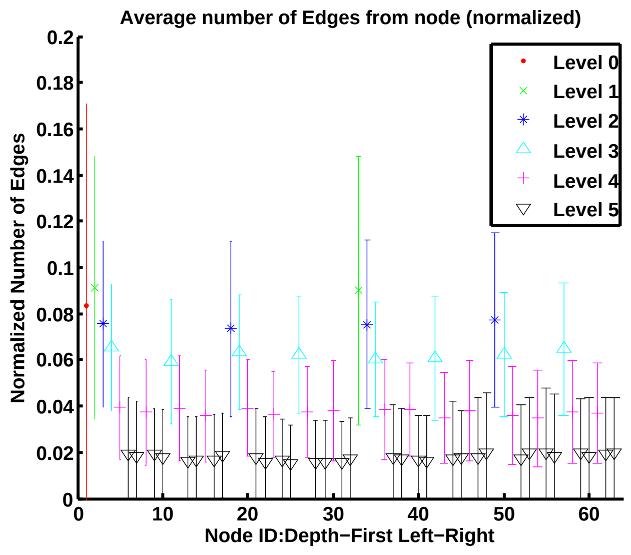


Figure 4: A graphical model for generation 25 from a run of the crossover only experiment.



(a) Edge quantity per generation, average of 30 runs.



(b) Average edges per node, color sorted by depth level. Nodes are ordered in depth-first left-right.

Figure 5: Edge quantity plots

Edge quantity can be studied further. Fig. 5(b) indicates how many edges there are at each depth level of the prototype tree by showing the average number of edges per node, with error bars, for a run which used crossover. The nodes are ordered depth-first left-right. The coloring shows the depth level of the node and the number of edges are normalized according to the possible number of edges, 62. We observe more edges at lower depth levels.

4.4.2 Edge Type

While a random population's graphical model edges will generally span, in GP terms, between a node and its parent, we would hypothesize that an evolved population's edges would span between a node and its sibling and even, to higher ancestral generations (in tree structure terms), to its uncles and grandparents or farther. This leads us to investigate the ancestral property of edges.

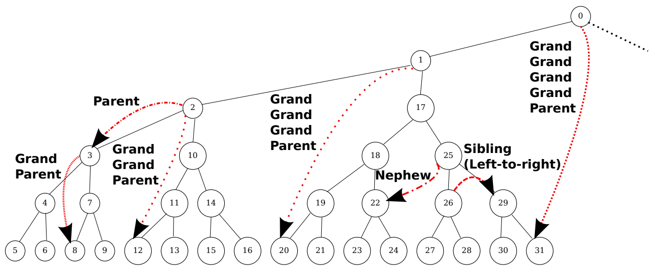
We therefore also perform a basic enumeration of different types of edges. We define parent, sibling, grand-parent, grand-grand-parent, grand-grand-grand-parent, grand-grand-grand-grand-parent, uncle, and unclassified types. Sibling edges are one direction only: a sibling is the rightmost child of a parent but not the leftmost child of a parent. The edge types are illustrated in Fig. 6(a) using different dashes to denote each type. In Fig. 6(b) an example tree with a count of parent, left sibling, nephew and grandparent edge types is shown.

The interpretation of edges according to this structural ancestry typing is interesting. To see this, it is necessary to conceptually map between a prototype tree's (or graphical model's) edges to GP context in the following manner: A GP tree is a representation for an (executable) expression that is a nested set of functions where arguments of a function are recursively themselves functions until the recursion "bottoms out" where arguments are non-functions (variables or constants, a.k.a terminals) represented as leaves. A GP tree is evaluated (a.k.a executed or interpreted) by a preorder parse where successive arguments are recursively evaluated in preorder. The dependency edges to a leaf from nodes above it in the tree indicate which functions in the expression depend on that argument's value. If the dependency edge is reversed in direction, i.e. from a leaf to a node above it, it indicates that a function elsewhere in the tree (coming either before or after the evaluation of the terminal when the nested expression is evaluated) influences the value of the terminal (e.g. whether it is 0.1 or 1.0 in Pagie-2D).

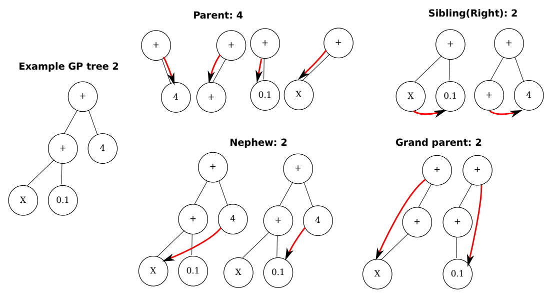
We now proceed to analyze each experiment wherein variation differs. We start with a baseline where the GP run uses selection but no crossover or mutation.

4.4.3 No Variational Operators Experiment

When GP is run without any variation operators, selection

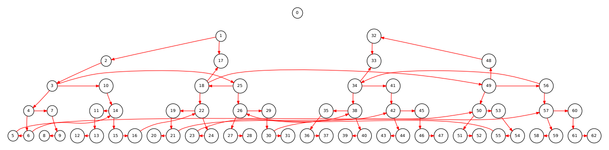


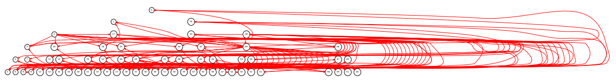
(a) Edge types



(b) Number of edge types in the example tree

Figure 6: Possible edge types by ancestry. In Fig. 6(a) the edges (dependencies) in the GM which are studied are shown. Fig. 6(b) there is an example of how the edge types are counted.


(a) Generation 0, no variational operators


(b) Generation 13, no variational operators

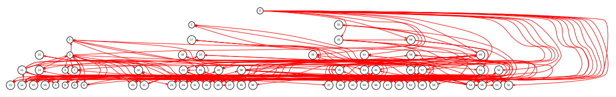

(c) Generation 40, no variational operators

Figure 7: Selection Only Experiment: graphical models for one randomly selected run at generation 0, 13 and 40.

eventually hones in on a completely homogeneous population. We find in our runs that this usually happens by generation ten. For one run, Fig. 7(a) shows a graphical model of the random population at generation 0. It has relatively fewer edges than subsequent generations. Fig. 7(b) shows a graphical model of generation 13 when the number of edges has increased. After the generation when the population has converged to a single point, any variation we observe in generational graphical models is due to the stochasticity of the EA we use to build the model.

Recall that our model building algorithm limits the number of dependency edges from a node to three. This implies that the maximum number of edges, given a prototype tree of size 63, equals 183. Which is the case at generation 40, when the population has completely converged to one GP solution, per Fig. 7(c) the graphical model has the maximum number of edges.

Next we consider edge types. The prevailing edge type is parent, as shown in Fig. 8(a). This figure is a time series plot of edge type quantity expressed as the ratio of the number of edges in the graph of the type to the total possible number of edges of the type. For example, for an edge of type parent there are 62 possible edges, a ratio of 0.5 means that there are 31 edges in the graphical model of this type. There are many different types of edges in a homogeneous population and they vary more between each generation compared to the experiments using operators. Half of the edges are unclassified, the other half is almost uniformly distributed between the edge types, except for grand grand parent. Moreover, the sibling relations are not as frequent compared to the experiments using operators. This is again likely due to the fact that the population is homogeneous and the structure finding algorithm is stochastic.

4.4.4 Mutation Only Experiment

In the experiment where GP was run only with a subtree mutation operator for variation, we examine three resulting graphical models at different generations, the quantity of edges and their ancestry type.

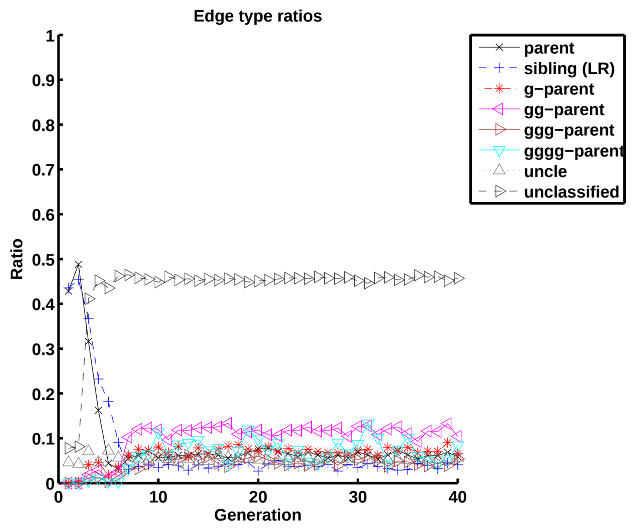
Mutation randomizes the population, but selection promotes the edge types associated with fit solutions. This makes later generation graphical models distinguishable from the initial population. The graphical models in Fig. 9 show a selective pruning of edges over generations.

Consulting Fig. 5(a) as to the number of edges, we observe the quantity rapidly increasing (in approximately the first 7 generations) to reach close to the maximum number of possible edges in the graphical model, before decreasing to almost the same number of edges as nodes. The edge ratios shown in Fig. 8(b) indicate the most frequent type of edge is from a parent to a child. As the number of edges decrease, the most frequent edge type is parent, then sibling and some uncles, with hardly any grand-parent dependencies. Our interpretation is that by adding new random subtrees, mutation creates new local dependence which is confined to the subtree. It disrupts any dependencies extending out of the subtree which it excises. This would imply that mutation runs have more parent edges.

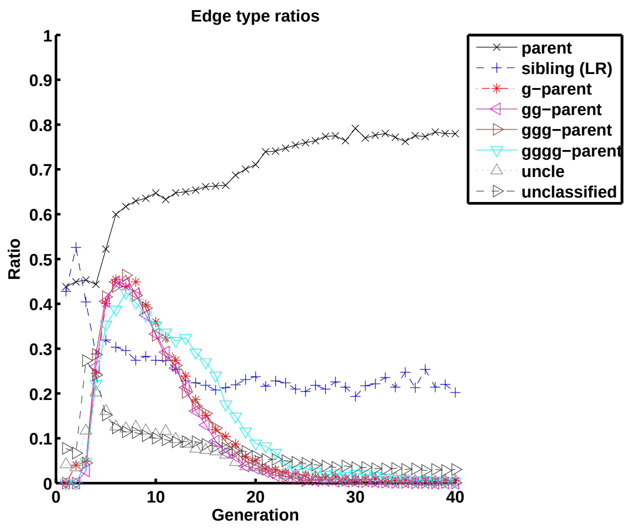
4.4.5 Crossover Only Experiment

Considering the experiment where GP was run only with the subtree crossover operator for variation, we examine three resulting graphical models from different generations, the quantity of edges and their ancestry type.

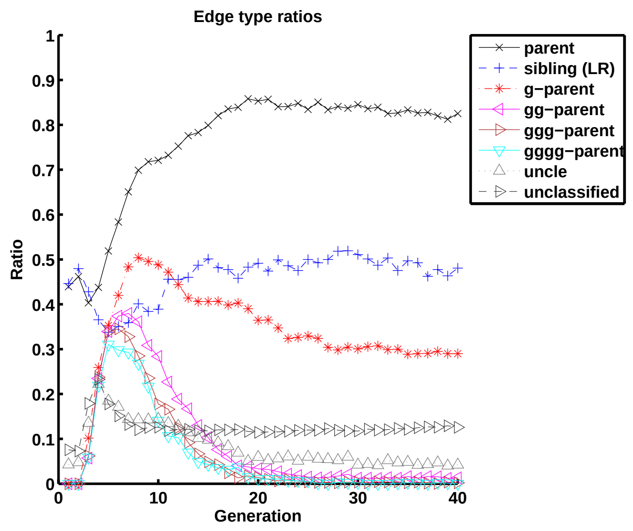
The graphical models in Fig. 8(c) differ by generation and, in unshown detailed graphical model data (animated generation by generation), we observe that the change from one generation to the next is very gradual, more so than with mutation. Consulting Fig. 5(a), we observe that initially there are approximately 70 edges, while there is a peak of approximately 180 edges around generation six. Then the



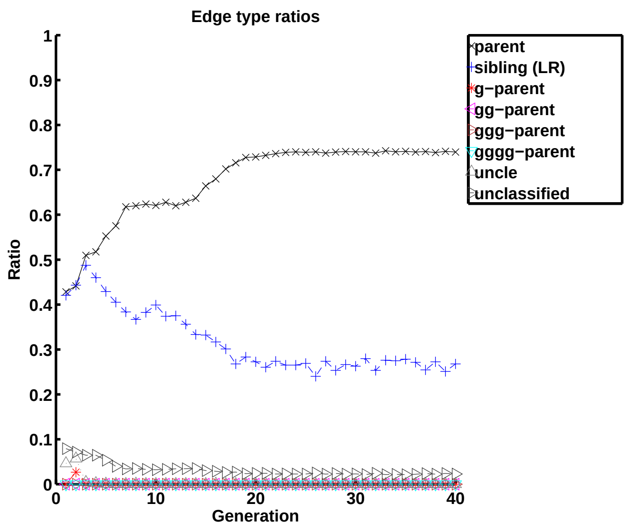
(a) Edge type ratios, no operators



(b) Edge type ratios, mutation



(c) Edge type ratios, crossover



(d) Edge type ratios, no selection

Figure 8: Plot of the ratios of edges in different types. For Pagie 2D without crossover and mutation 8(a), mutation 8(b), crossover 8(c) and no selection 8(d).

number of edges tails off and stabilizes at approximately 130. When we align our animation with the fitness time series we see the tail off synchronizes with approximately the same generation when fitness stops improving and the population has started to converge.

Considering the type of edges, per Fig. 8(b), we observe that, after generation 20, almost all nodes in the graphical model have an edge with their ancestor and half have edges to their sibling. After 25 generations the ratios of the other edges have somewhat stabilized. Fig. 5(b) is an example of crossover only experiment. We see a consistency in the number of edges given the depth level of the node in the tree. As expected, the nodes at the lower depth levels have fewer edges.

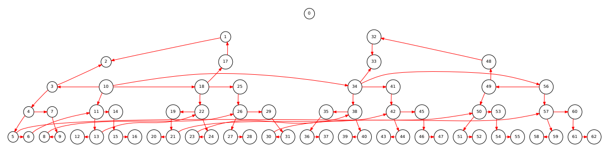
Crossover is a subtree swap. This implies that it preserves more short and long range, i.e. ancestral dependencies such as parent, sibling plus grandparent and uncle. Because of the exchange, rather than the new material, it results in only a minor net change in dependency, *at the structural level*. Of course, the issue with crossover in GP is its semantic consequences while its structural behavior directly follows from its definition.

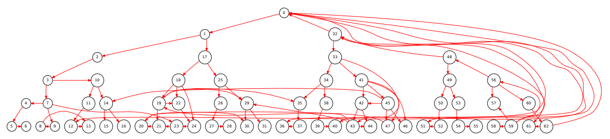
4.4.6 No Selection Experiment

For the experiment where GP was run with random selection and only the subtree crossover operator for variation, we examine the quantity of edges and their ancestry type.

The graphical models for some generation are not shown, but they have very few edges and the connections are mainly between parent and sibling, we verify this by consulting Fig. 8(d). In addition, in Fig. 5(a), we observe that initially there are approximately 70 edges and within 6 generations the number drops to the the minimum number of edges. This effect is most likely due to the effect of initialization.

The next section will further discuss the results from the different experiments.


(a) Generation 0, mutation


(b) Generation 13, mutation

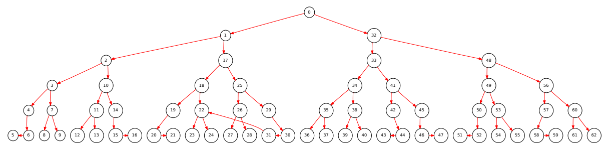
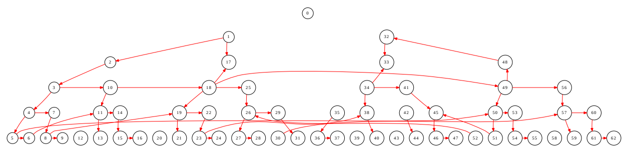
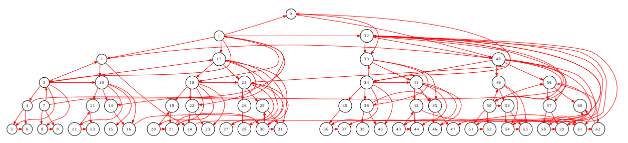

(c) Generation 40, mutation

Figure 9: Mutation Only Experiment: graphical models for one randomly selected run at generation 0, 13 and 40.


(a) Generation 0, crossover


(b) Generation 13, crossover

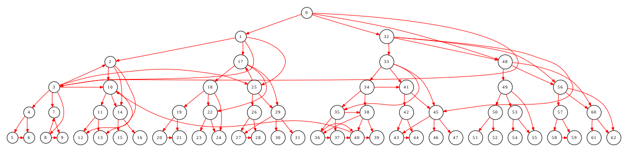

(c) Generation 40, crossover

Figure 10: Crossover Only Experiment: graphical models for one randomly selected run at at generation 0, 13, and 40

4.5 Cross Experiment Remarks

Our understanding of mutation and crossover would lead us to hypothesize that the dependencies being studied, due to their structural nature, would be different. Comparing Fig. 9 to Fig. 10 bears this out. In the crossover only experiment we can observe that crossover's graphical model in the final generation has different dependencies and more dependencies than mutation. This is interpretable: crossover is an exchange with less net disruption while mutation introduces new material while excising existing material. Both operators culminate with approximately the same quantity of parent edge types but reach this endpoint with different trajectories (compare Fig. 8(b) to Fig. 8(c)). Crossover has more siblings, and longer range dependencies with uncles and grand-parents. This is consistent with the difference between the two operators in terms of different extents of dependence preservation and disruption.

From the different experiments which differ by variation operator, we are able to distinguish distinctly different graphical models, both regarding number of edges and type of edges. For all runs, across the variation operator experiments (i.e. excluding selection only), when inspecting the fitness, the best fitness is only slightly improving during the runs. The operators manipulate the population in such a way that the dependencies between experiments differ. When comparing the number of edges and the type of edges from each experiment it can be seen that the number of edges in the crossover only experiment is higher than for a random population. The crossover operator helps to create a population that has many parent dependencies, sibling dependencies and some longer grand-parent dependencies. In contrast, for the experiments with the mutation operator the number of edges is the same as for a initial random population, but the structure is quite different from initial random. The types of edges are not the same as when crossover is used either, there are hardly any longer dependencies, such as grand-parents edges. When random selection is used there are no long distance dependencies and the only the minimum number of edges after a the effects of initialization have disappeared. Finally, the number of edges in a homogeneous setup is very high, reaching the maximum limit of edges and the variation in edges is most likely due to the stochastic learning of the graphical model.

A graphical model also provides a way to generate a GP population from its parsimonious representation. It is an approximate form of compression where a very important property of the original population is preserved - variable dependency. Further information from the graphical model could be taken into account when designing operators in GP, or reduced Bayesian network for EDA-style GP, as in Hemberg et al (9).

5. CONCLUSIONS & FUTURE WORK

We have introduced graphical models as a new means of investigating the dynamics of genetic programming populations in general and,in this work specifically, GP's operators. Graphical model based analysis is complementary to conventional statistics that are tracked across generations. The graphical model shows the dependencies between nodes in GP trees when treated as a multivariate distribution represented as prototype trees. We presented an unbiased means of deriving a graphical model and introduced an EA based algorithm for efficiently finding graphical models. Moreover, algorithm wise we only search for order of the nodes in a graphical model, unlike K2 which searches for a graphical model given an order. We exhaustively search for best graphical model for each order. For the purpose to overcome the cost of exhaustive search, we initially build a cache and ADTrees. These techniques speed up the fitness evaluation in the creation of the graphical model.

It is important to recognize that alternate kinds of graphical models could be built. Graphical models different from the one we present have the potential to emphasize different properties about GP's population distribution. We envision future investigations that would answer which are most appropriate and insightful for modeling GP dynamics. Our motivation herein was to investigate whether insight into a run's dynamics at the population, tree structure level, could be gained by observing the dependency structure graphical models reveal about the multivariate distribution associated with functions and terminals in GP trees. The graphical model information is in this form structural rather than syntax oriented.

Using Pagie-2D problems as exemplars, we take first steps in analyzing generation and inter-generation dynamics of GP runs in terms of changing graphical model structure. The experiments show that there is a distinct difference in the graphical model of a GP population when different operators are used, both in the number of edges in the graphical model and the types of edges.

For future work further problems and operators will be investigated. Other work can be combinations of edge types as well as doing inference of the outcome of the random variables from the generated graphical models. Moreover, if we correlate fitness with the graphical model we can:

- Pool all samples and take top fitness slice and learn a graphical model.
- Use fitness value as a conditional value for the graphical model.
- Align the data based on fitness to get the dependency structure of populations with similar fitness distribution

Acknowledgment

We thank the support received from General Electric Global Research Center and Li Ka Shing Foundation.

APPENDIX

A. FAST LEARNING OF THE MODEL

We use an evolutionary algorithm for learning the structure of a Bayesian network from a fully observed data set. This algorithm searches over the space of node *orders*, rather than searching over the space of DAGs. The genetic algorithm proposes an order and an optimal DAG is identified by exhaustively searching through different possible DAGs given this order. Each DAG is scored using a Bayesian score function.

A.1 Representation and Operators

In our GA over orders, we use the node order R as the genotype of an individual. We use swap mutation and cycle crossover as our operators (3). There are many networks consistent with the order R. We define the phenotype of an individual as the *best* network consistent with R, subject to a maximum indegree k, which is fixed at the start of the algorithm. This follows the setup that Teyssier and Koller (20) used for their hill-climbing algorithm over orders. The GA needs to find the best DAG for R and k.

A.1.1 The Bayesian Score

We want our scoring function to measure how well a structure fits the data. We also want it to penalize complex structures, because a simpler model makes inference more tractable. The Bayesian score is one of several scoring functions satisfying these criteria. We define a prior $P(\mathcal{G})$ over graph structures, and another prior $P(\boldsymbol{\theta}_\mathcal{G} \mid \mathcal{G})$ over parameters given a graph. The posterior over structures for a data set $\mathcal{D}$ is given by Bayes' rule:

$$P(\mathcal{G} \mid \mathcal{D}) \;=\; \frac{P(\mathcal{D} \mid \mathcal{G})\; P(\mathcal{G})}{P(\mathcal{D})}$$

where the denominator is a normalizing factor that does not depend on the structure we are trying to evaluate. The Bayesian score is then given by:

$$\text{score}_B(\mathcal{G} : \mathcal{D}) \;=\; \log P(\mathcal{D} \mid \mathcal{G}) + \log P(\mathcal{G}). \qquad (1)$$

The second term is usually less important, because it does not grow with the size of $\mathcal{D}$. The first term is the logarithm of the marginal likelihood of the data given the structure, which averages over the choices of parameters for $\mathcal{G}$:

$$P(\mathcal{D} \mid \mathcal{G}) \;=\; \int_{\boldsymbol{\theta}_\mathcal{G}} P(\mathcal{D} \mid \boldsymbol{\theta}_\mathcal{G}, \mathcal{G})\; P(\boldsymbol{\theta}_\mathcal{G} \mid \mathcal{G})\; d\boldsymbol{\theta}_\mathcal{G},$$

where $P(\mathcal{D} \mid \boldsymbol{\theta}_\mathcal{G}, \mathcal{G})$ is the likelihood of the data set given the network $\langle \mathcal{G}, \boldsymbol{\theta} \rangle$, and $P(\boldsymbol{\theta}_\mathcal{G} \mid \mathcal{G})$ is our prior over parameters given a structure. The marginal likelihood is also known as the evidence function.

If the prior over parameters is a Dirichlet distribution where $P(\boldsymbol{\theta}_{Y_i \mid \text{pa}_{Y_i}} \mid \mathcal{G})$ has hyperparameters $\{\alpha^{\mathcal{G}}_{y_i^j \mid \boldsymbol{u}_i}\}$ for j from 1 to $|Y_i|$, which satisfies global and local parameter independence. With $\alpha^{\mathcal{G}}_{Y_i \mid \boldsymbol{u}_i} = \sum_j \alpha^{\mathcal{G}}_{y_i^j \mid \boldsymbol{u}_i}$, then the marginal likelihood can be written as follows:

$$P(\mathcal{D} \mid \mathcal{G}) \;=\; \prod_i \prod_{\boldsymbol{u}_i \in Val(\text{Pa}^{\mathcal{G}}_{Y_i})} \left[\frac{\Gamma(\alpha^{\mathcal{G}}_{Y_i \mid \boldsymbol{u}_i})}{\Gamma(\alpha^{\mathcal{G}}_{Y_i \mid \boldsymbol{u}_i} \;+\; M[\boldsymbol{u}_i])} \cdot \prod_{y_i^j \in Val(Y_i)} \frac{\Gamma(\alpha^{\mathcal{G}}_{y_i^j \mid \boldsymbol{u}_i} \;+\; M[y_i^j, \boldsymbol{u}_i])}{\Gamma(\alpha^{\mathcal{G}}_{y_i^j \mid \boldsymbol{u}_i})} \right] \qquad (2)$$

This formula is simplified by choosing a reasonable prior.

Priors and Decomposability

It is desirable to have a scoring function that decomposes:

$$\text{score}(\mathcal{G} \;:\; \mathcal{D}) \;=\; \sum_i \text{FamScore}(Y_i \mid \text{Pa}^{\mathcal{G}}_i \;:\; \mathcal{D})$$

where the family score $\text{FamScore}(Y_i \mid \boldsymbol{U} \;:\; \mathcal{D})$ measures how well the variables $\boldsymbol{U}$ serve as parents of Y. With a decomposable score, local changes to the structure affect only a small number of families, and so we do not have to recompute the score of the entire network. For example, if we change the order such that we only swap the last two positions in a node ordering R we only have to search for the best parent sets for the last two nodes and only evaluate the family scores for these two nodes. Additionally, we can search for best parent sets of individual nodes independently.

To ensure that the Bayesian score decomposes, our structure and parameter priors should satisfy some conditions. The structure prior should satisfy structure modularity:

$$P(\mathcal{G}) \;\propto\; \prod_i P(\text{Pa}_{Y_i} = \text{Pa}^{\mathcal{G}}_{Y_i}),$$

where $P(\text{Pa}_{Y_i} = \text{Pa}^{\mathcal{G}}_{Y_i})$ is the probability of choosing that specific set of parents for X_i. The parameter prior should satisfy global parameter independence and parameter modularity:

$$P(\boldsymbol{\theta}_{Y_i|\boldsymbol{U}} \mid \mathcal{G}) \;=\; P(\boldsymbol{\theta}_{Y_i|\boldsymbol{U}} \mid \mathcal{G}')$$

for any pair of structures $\mathcal{G}$, $\mathcal{G}'$.

When we have no prior knowledge about the structure we are trying to learn, we can use a structure prior that assigns equal probability to every possible structure. In this case, we can ignore the second term ($logP(\mathcal{G})$) in the Bayesian score (equation 1), and evaluate structures solely by their marginal log-likelihood $\log P(\mathcal{D} \mid \mathcal{G})$.

In the case when we have no prior knowledge about the parameters, we can use the BDeu prior. (BDeu stands for uniform Bayesian Dirichlet prior satisfying likelihood equivalence (11).) For this prior, we only need to specify an equivalent sample size (or strength) α. Given a node Y_i and its parents Pa_{Y_i}, the BDeu prior assigns equal probabilities to all values of the node and parents:

$$\alpha_{y_i|\mathrm{pa}_{Y_i}} \;=\; \alpha \cdot \frac{1}{|\,Val(Y_i, \mathrm{Pa}_{Y_i})|}, \qquad (3)$$

where $|\,Val(Y_i, \mathrm{Pa}_{Y_i})|$ is the number of possible assignments to X_i and Pa_{Y_i}. Let $\gamma(\boldsymbol{Y}) = \alpha \cdot \frac{1}{|Val(\boldsymbol{Y})|}$ for any set of nodes $\boldsymbol{Y}$. Then the marginal log-likelihood score decomposes into a sum of terms for each family:

$$\mathrm{FamScore}(Y_i, \mathrm{Pa}_{Y_i}) = \sum_{\boldsymbol{u}_i \in Val(\mathrm{Pa}^{\mathcal{G}}_{Y_i})} \Bigg[\qquad (4)$$

$$\sum_{y_i^j \in Val(y_i)} \ln \frac{\Gamma(\gamma(Y_i, \mathrm{Pa}_{Y_i}) + M[y_i^j, \boldsymbol{u}_i])}{\Gamma(\gamma(Y_i, \mathrm{Pa}_{Y_i}))}$$

$$\ln \frac{\Gamma(\gamma(\mathrm{Pa}^{\mathcal{G}}_{Y_i}))}{\Gamma(\gamma(\mathrm{Pa}^{\mathcal{G}}_{Y_i})) + M[\boldsymbol{u}_i])} +$$

$$\sum_{y_i^j \in Val(y_i)} \ln \frac{\Gamma(\gamma(Y_i, \mathrm{Pa}_{Y_i}) + M[y_i^j, \boldsymbol{u}_i])}{\Gamma(\gamma(Y_i, \mathrm{Pa}_{Y_i}))} \Bigg]$$

A.2 Caching for Efficient Fitness Evaluation

Evaluating the fitness of an individual in our GA involves computing the Bayesian score of a network, which is an expensive operation. We use two levels of caching to speed up this computation. The first level caches sufficient statistics (counts) of the data set. The second level caches set scores.

Suppose that we had a *contingency table* for a node Y_i and its parents Pa_{Y_i}. This table contains one row for each possible assignment to Y_i and Pa_{Y_i}, and the number of times that assignment occurs in the training set. If we have this contingency table, then we can evaluate $FamScore(Y_i, \mathrm{Pa}_{Y_i})$ in time linear to the size of the contingency table, simply by performing the summation in Equation (4). We can compute contingency tables naively as follows:

- We start with a table with all zeros.
- From each row in the data set, we extract the values of Y_i and Pa_{Y_i}, and denote them y_i and $\boldsymbol{u}_i$ respectively.
- We then increment the row corresponding to $y_i, \boldsymbol{u}_i$ in the contingency table.

With this approach, computing each contingency table requires one pass over the entire data set.

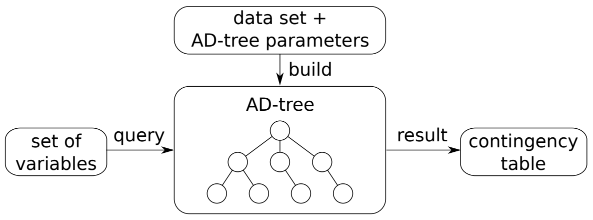


Figure 11: An AD-tree is built once and queried many times.

Our first level of caching uses the AD-tree data structure of Moore and Lee (14) to speed up the computation of contingency tables. The AD-tree saves time at the expense of using more memory. It is a tree data structure, where each node "splits" the data set according to all possible assignments to a variable. Contingency tables can be computed from the AD-tree in time that is independent of the total size of the data set. Thus, we no longer require one full pass through the data set to compute a contingency table. Figure 11 illustrates the usage of the AD-tree. For details see Moore and Lee (14).

Traditional hill-climbing methods cache family scores so that they can be retrieved again and again without being recomputed. We improve on this slightly by noticing that with the BDeu prior, the family score (equation (4)) decomposes into two set scores:

$$\mathrm{FamScore}(Y_i, \mathrm{Pa}_{Y_i}) \;=\; \mathrm{SetScore}(\mathrm{Pa}_{Y_i} \cup \{Y_i\}) - \mathrm{SetScore}(\mathrm{Pa}_{Y_i})$$

where

$$\mathrm{SetScore}(\boldsymbol{Y}) \;=\; \sum_{\boldsymbol{v}_i \in Val(\boldsymbol{Y})} \ln \frac{\Gamma(\alpha_{\boldsymbol{Y}})}{\Gamma(\alpha_{\boldsymbol{Y}} + M[\boldsymbol{v}_i])}$$

for any set of variables $\boldsymbol{Y}$, and $\alpha_{\boldsymbol{Y}} = \alpha \cdot \frac{1}{|Val(\boldsymbol{Y})|}$. Computing a family score seems to require the contingency tables of $\mathrm{Pa}_{Y_i} \cup \{Y_i\}$ and Pa_{Y_i}. But in fact, the latter contingency table can be obtained from the former by marginalizing the column Y_i. This means that only one AD-tree query is required for computing the score of a family.

Our second level of caching stores set scores instead of family scores. This results in more cache hits, because multiple families during the lifetime of the algorithm may have the same set of parents.

Additional improvements We cache sufficient statistics and set scores, as we did for our GA over graphs (section A.2). Note that we no longer need to build the full AD-tree. If the maximum indegree is k, we will never need contingency tables for more than $k+1$ variables, so we only need to build the AD-tree up to depth $k+1$.

From Teyssier and Koller (20) we use the following. Suppose our maximum indegree is k. For each node X, we compute the score of all possible parent sets of size k or less. We rank these parent sets in decreasing order by score. We then prune the set of possible parent sets as follows: if $\boldsymbol{U}' \subset \boldsymbol{U}$ and $\mathrm{FamScore}(X \mid \boldsymbol{U}') > \mathrm{FamScore}(X \mid \boldsymbol{U})$, then we remove the parent set $\boldsymbol{U}$ from consideration. For any order R, if $\boldsymbol{U}$ is a valid choice of parents for X, then $\boldsymbol{U}'$ is also valid, and has a better score. Therefore, we can safely eliminate $\boldsymbol{U}$, because we would never prefer it over $\boldsymbol{U}'$. We say that $\boldsymbol{U}$ is *dominated* by $\boldsymbol{U}'$.

To evaluate an order R, we go through every node from first to last. For node X, we go through the ranked list of its possible parent sets, stopping at the first parent set consistent with the order R. This is the optimal parent set for X, subject to the order R and the indegree constraint. We continue until we have found the optimal parent set for each node. At this point, as we previously discussed, we have found the optimal network for the order R and the given maximum indegree.

References

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

[2] Almal, A., MacLean, C., Worzel, W.: A population based study of evolutionary dynamics in genetic programming. Rick Riolo, Terence Soule and Bill Worzel (Eds.) (2009)

[3] Arthur Carvalho.: A cooperative coevolutionary genetic algorithm for learning bayesian network structures. GECCO 2011, 1131–1138

[4] Cooper, G. F., Herskovits, E.: A Bayesian Method for the Induction of Probabilistic Networks from Data Machine Learning, 1992, 9, 309-347

[5] Daida, J., Hilss, A., Ward, D., Long, S.: Visualizing tree structures in genetic programming. Genetic Programming and Evolvable Machines 6(1), 79–110 (2005)

[6] Daida, J., Tang, R., Samples, M., Byom, M.: Phase transitions in genetic programming search. Genetic Programming Theory and Practice IV, 237–256 (2007)

[7] Nir Friedman, Iftach Nachman, and Dana Peér.: Learning bayesian network structure from massive datasets: The "sparse candidate" algorithm. In *Uncertainty in Artificial Intelligence (UAI-99)*, 206–215, San Francisco, CA, 1999. Morgan Kaufmann.

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

[9] Erik Hemberg, Kalyan Veeramachaneni, James McDermott, Constantin Berzan, Una-May O'Reilly: An investigation of local patterns for estimation of distribution genetic programming. GECCO 2012: 767-774

[10] Erik Hemberg, Kalyan Veeramachaneni, Una-May O'Reilly: Graphical models and what they reveal about GP when it solves a symbolic regression problem. GECCO(Companion) 2012: 493-494

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

[12] Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002),

[13] McPhee, N., Hopper, N.: Analysis of genetic diversity through population history GECCO, 1999, 2, 1112-1120

[14] Andrew Moore and Mary Soon Lee.: Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, March 1998.

[15] O'Neill, Michael and Vanneschi, Leonardo and Gustafson, Steven and Banzhaf, Wolfgang.: Open issues in genetic programming *Genetic Programming and Evolvable Machines*, 11, 3:339–363, 2010

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

[17] Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming `http://lulu.com` (2008)

[18] Raidl, G., Gottlieb, J.: Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem. Evolutionary Computation 13(4), 441–475 (2005)

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

[20] M. Teyssier and D. Koller. Ordering-based search: A simple and effective algorithm for learning bayesian networks. In *Proceedings of the Twenty-first Conference on Uncertainty in AI (UAI)*, 584–590, Edinburgh, Scotland, UK, July 2005.

[21] Tomassini, M., Vanneschi, L., Collard, P., Clergue, M.: A study of fitness distance correlation as a difficulty measure in genetic programming. Evolutionary Computation 13(2), 213–239 (2005)

[22] Wolfson, K., Zakov, S., Sipper, M., Ziv-Ukelson, M.: Have your spaghetti and eat it too: evolutionary algorithmics and post-evolutionary analysis. Genetic Programming and Evolvable Machines 12(2), 121–160 (2011)