

# Automatic Creation of Taxonomies of Genetic Programming Systems

Mario Graff and Riccardo Poli

School of Computer Science and Electronic Engineering, University of Essex, UK  
{mgraft, rpoli}@essex.ac.uk

**Abstract.** A few attempts to create taxonomies in evolutionary computation have been made. These either group algorithms or group problems on the basis of their similarities. Similarity is typically evaluated by manually analysing algorithms/problems to identify key characteristics that are then used as a basis to form the groups of a taxonomy. This task is not only very tedious but it is also rather subjective. As a consequence the resulting taxonomies lack universality and are sometimes even questionable. In this paper we present a new and powerful approach to the construction of taxonomies and we apply it to Genetic Programming (GP). Only one manually constructed taxonomy of problems has been proposed in GP before, while no GP algorithm taxonomy has ever been suggested. Our approach is entirely automated and objective. We apply it to the problem of grouping GP systems with their associated parameter settings. We do this on the basis of performance signatures which represent the behaviour of each system across a class of problems. These signatures are obtained through a process which involves the instantiation of models of GP's performance. We test the method on a large class of Boolean induction problems.

## 1 Introduction

A *taxonomy* is the coherent arrangement of elements into groups. For many sciences, the construction of a taxonomy has been an important step towards maturity. Taxonomies have many applications. In biology, for example, taxonomies of animals and plants are the starting point in understanding the biodiversity of a region. Also, taxonomies help to model a group of individuals as a single entity, thereby removing the need to analyse each member of the group separately.

The importance of taxonomies is also clear for Evolutionary Algorithms (EAs). For example, from a practitioner's point of view, an *algorithm taxonomy* may reduce the time-consuming task of finding the most suitable algorithm to solve a problem at hand. This is generally done by selecting an algorithm that has already been applied to a "similar" problem. If this algorithm fails to return a satisfactory result, one would want to look for an algorithm that, due to its characteristics, is sufficiently different from the first, e.g., an algorithm that belonged to a different group of the taxonomy. *Taxonomies of problems* are also useful since it is often possible to associate algorithms to problem classes. Knowing which taxonomic class a problem is in can then guide a practitioner towards good algorithms for solving it.

So, what taxonomies are available for EAs? There are some general taxonomies. In fact, every author of a book on EAs is forced to come up with some structure within which to

present the material in an orderly fashion; for example see [3,14]. This structure can be seen as a taxonomy of algorithms. Typically these taxonomies are constructed on the basis of a set of characteristics which are deemed to best represent such algorithms. These often focus on each of the main components of an algorithm, e.g., the representation, the genetic operators, the number of parents used to produce an offspring, the number of offspring, the way the offspring is included in the population, etc. That is, they take a reductionist approach, which looks at similarities and differences among the components of an algorithm, *as perceived by the user*. Also, often the topmost levels in the taxonomy are determined by historical reasons.

While these taxonomies are useful, they are, rather naturally, subjective. Furthermore, while some of the most useful taxonomies in other disciplines are *based on behaviour*, not just form (think, for example, of the periodic table), these types of EA taxonomies are typically only based on “form”, i.e., the components of an algorithm. So, they aren’t designed to give answers to practitioners, who are really interested in determining whether an algorithm will solve their problems.

Of course, there are other EA taxonomies which are much more precise and useful. For example [4] proposed to use a tabular representation to describe genetic algorithms, scatter search and ant systems. In [13] a taxonomy of parallel genetic algorithms was proposed, which was based on the different ways in which a genetic algorithm can be parallelised. [15] categorises various EAs and other mathematical and AI search algorithms. In [7] a taxonomy of crossover operators for real-coded genetic algorithms was presented. The taxonomy consisted in 18 different crossover operators grouped into 3 classes. There have also been a very small number of taxonomic efforts of this kind in GP, e.g., [18]. Also a GP problem taxonomy was proposed in [12].

One feature of these taxonomies is that they capture only specific aspects of an algorithm or they are very specific to a special class of algorithms. The reason is combinatorial: not only there are many different EAs to consider, but also each may have from several to tens of parameters (e.g., 17 parameters were used in the GP system described in [10]) which can alter its behaviour, in some cases very significantly. If one considers the combinations of these parameters, then the set of algorithms with their different configurations immediately becomes astronomically large. Another feature of these taxonomies is that, like the ones described above, they are *manually built* and so they are subjective and are normally based on form rather than behaviour of algorithms.

Modelling the behaviour of algorithms is the realm of EA theory. There are a variety of tools that have been used to understand and categorise EAs. These range from Markov chains [16] to schema theories [11] to computational complexity [8]. These have produced important results which go some way in the direction of a taxonomy based on behaviours. It is possible, for example, to determine whether a particular EA will solve a particular problem in polynomial vs. non-polynomial time or whether a particular EA is a global optimiser (given enough time). However, results are either specific to a problem or of purely mathematical interest. Furthermore, they don’t really provide a detailed EA taxonomy, in the sense that typically they only include two classes. So, mathematical models, too, struggle to give taxonomies that meet the needs of practitioners.

A step in the direction of automating the creation of taxonomies as well as turning them into practical tools has been made in [1,2] where several tens of problems where

automatically grouped into a *problem taxonomy*. The idea was to run a particular EA where mating is controlled via a neighbourhood structure recording the average number of fitness evaluations required to solve each problem using each of a small number of neighbourhood structures. The resulting performance signatures (vectors) were then clustered hierarchically, thereby producing a taxonomy.

Here, we present a method to create taxonomies which has some similarity with the one proposed by Ashlock *et al.*, but with also some differences. Firstly, we focus on GP. Secondly, we are interested in producing a practical *algorithm taxonomy*, which we think may be more useful than a problem taxonomy. Like Ashlock's work, our approach is automated and objective. We apply it to the problem of grouping GP systems with their associated parameter settings. Like [1,2] we do this on the basis of the performance signatures and a hierarchical clustering algorithm. However, our performance signatures represent the behaviour of each system across a class of problems, not the behaviour of multiple versions of an algorithm over one problem. Also, our signatures are not the measurement of performance over multiple cases: they are obtained thorough a process which involves the instantiation of a mathematical models of GP's performance recently introduced in [6]. It is then this model which provides the signature for each algorithm.

The paper is organised as follows. In Sec. 2 we describe the clustering algorithm used to create our GP taxonomy. In Sec. 3 we review the models of GP's performance proposed in [6]. Sec. 4 presents a new procedure to instantiate such models. Alternative similarity measures for comparing different GP performance models and systems are presented in Sec. 5. In Sec. 6 we describe the parameters setting and the different GP systems used to test our approach. This is followed (in Sec. 7) by a description of the system taxonomies created when evaluating performance over a large class of Boolean induction problems. Some conclusions are given in Sec. 8.

## 2 Creating GP's Taxonomies

As we already mentioned, our technique to create taxonomies for GP is based on a *hierarchical clustering algorithm* (see [9]). This works as follows. Let us assume we are given a set of objects (performance signatures, in our case) which we wish to group. Let  $\mathcal{M}$  be a square matrix where the element  $(i, j)$  represents the distance (or similarity) between the  $i$ -th and  $j$ -th objects. We derive from it a similarity measure  $s(\mathcal{X}, \mathcal{Y})$  between pairs of clusters,  $\mathcal{X}$  and  $\mathcal{Y}$  (more on this below). Then the clustering algorithm of [9] involves the following three steps: (a) First, each object is assigned to a separate cluster; (b) A new cluster is created by merging the two clusters which are closest based on the similarity measure  $s(\mathcal{X}, \mathcal{Y})$ , thereby reducing the number of clusters by one; (c) Repeat step 2 until there is only one cluster left.

Naturally the behaviour of the algorithm is influenced by the similarity measure  $s(\mathcal{X}, \mathcal{Y})$  used. In this paper, we used the *average linkage* as similarity measure. That is

$$s(\mathcal{X}, \mathcal{Y}) = \frac{\sum_{i \in \mathcal{X}} \sum_{j \in \mathcal{Y}} \mathcal{M}(i, j)}{|\mathcal{X}| |\mathcal{Y}|} \quad (1)$$

where  $|\cdot|$  denotes the number of elements in a cluster.

This algorithm has now one key component left to specify: the distance matrix  $\mathcal{M}$ . A meaningful  $\mathcal{M}$  can be computed using different procedures. In this paper we associate to each GP system a vector  $\mathbf{c}$  (a performance signature) and build  $\mathcal{M}$  by computing the distance (or similarity) between all the possible pairs of vectors  $\mathbf{c}$  (more on this later).

Of course, then the question becomes how to choose the size and components of  $\mathbf{c}$ . Again, there are different ways to define a mapping between GP systems (and parameter settings) and  $\mathbf{c}$  vectors. For example, taking a dual approach to Ashlock's [1,2], we could select a group of problems of interest, and associate a different component of  $\mathbf{c}$  to each problem. The component could just represent the performance of a particular GP system on the problem associated to that component. Each  $\mathbf{c}$  vector would then represent the behaviour of a GP system over all problems in our set.

While this is a simple and effective way of proceeding when the set of problems we want to use to build a taxonomy of algorithms is small, it is not viable when one wants to consider large problem classes. This is unfortunate, since to draw general taxonomic conclusions about different GP systems we need to test them over a large set of problems. So, we decided against this approach and instead adopted a more complex, but also more general technique which consists in creating the  $\mathbf{c}$  vectors using the coefficients of a model of GP behaviour. In particular, we will use a version of the performance model originally proposed in [6]. We review it in the next section.

### 3 Performance Model

The model proposed in [6] is a practical model of GP which has been specifically designed to focus on performance. Its applicability was tested on the class of rational symbolic regression problems with excellent results using as a performance measure the best fitness recorded in a run.

The model is based on the idea that GP's performance can be modelled using a re-representation of the fitness function. The simplest re-representation is as follows. Let  $\Omega = \{p_1, p_2, \dots\}$  be the ordered set of all programs obtained by recursively composing primitives from the primitive set and let  $f$  be a function over  $\Omega$ . Then, one can represent the fitness function as the ordered set  $\mathcal{F}(\Omega) = \{f_{p_1}, f_{p_2}, \dots\}$  where  $f_{p_i} = f(p_i)$ . Using this tabular re-representation of the fitness function, in principle, one could compute the following linear approximation (model) of GP's performance  $P(f) \approx a_0 + \sum_{p \in \Omega} a_p f_p$  where the  $a_p$  are coefficients, which, once again, in principle, could be obtained running a least square method on a suitable training set of  $(P, f)$  pairs. In reality, however, this approach is not viable because one needs to identify  $|\Omega| + 1$  parameters to create the model. So, if the search space is large (and it typically is), very large training sets would be needed. It is easy, however, to fix this problem: instead of re-representing  $f$  using all the programs in  $\Omega$ , we can use a subset  $\mathcal{S} \subseteq \Omega$ . We can then use the approximation

$$P(f) \approx a_0 + \sum_{p \in \mathcal{S}} a_p f_p \quad (2)$$

where we can control the trade-off between model accuracy and training set size by varying the cardinality of  $\mathcal{S}$ .

In GP, the fitness function  $f(p)$  typically states how similar the functionality of a program  $p$  is to a target functionality  $t$ . Often  $t$  is represented via a finite set of fitness cases. So, we

can think of both  $t$  and  $p$  as being  $\ell$ -dimensional vectors,  $\mathbf{t}$  and  $\mathbf{p}$ , respectively,  $\ell$  being the number of fitness cases. We can then define  $f(p) = d(\mathbf{p}, \mathbf{t})$  where  $d$  is a similarity measure between vectors.<sup>1</sup>

Under these conditions, (2) transforms into:

$$P(\mathbf{t}) \approx a_0 + \sum_{\mathbf{p} \in S} a_{\mathbf{p}} \cdot d(\mathbf{p}, \mathbf{t}) \quad (3)$$

In order to instantiate this model, one needs: a training set of problems  $T$ , a validation set  $V$  (to test the generality of the model), a closeness measure  $d : \mathbb{R}^\ell \times \mathbb{R}^\ell \mapsto \mathbb{R}$  and a set of program behaviours from which elements of  $S$  are drawn.  $T$  and  $V$  are composed by pairs  $(P(\mathbf{t}), \mathbf{t})$  where  $\mathbf{t}$  is a problem (represented by its target vector) and  $P(\mathbf{t})$  is the performance of GP on problem  $\mathbf{t}$ . To obtain  $P(\mathbf{t})$  a GP system is run multiple times and its performance is estimated by averaging. Note that while (3) was derived from (2) under the assumption that  $f(p) = d(\mathbf{p}, \mathbf{t})$ , we are not forced to use the same similarity measure  $d$  in both: in (3) we can choose a different  $d$  if this, for example, improves the model's accuracy.

A model selection algorithm is used to obtain  $S$  and the coefficients  $a_{\mathbf{p}}$  from the training set  $T$ . This is a point where our approach deviates from [6] where a GA and ordinary least squares were used for this purposes, while here we use a much simpler, yet effective approach. In the next section we provide details on the model identification techniques we used.

## 4 Model Optimisation

We used the Least Angle Regression (LAR) algorithm (see [5] for details) as a procedure to decide which elements should be included in  $S$  given a larger set of prospective target behaviours  $\Sigma$ . We stop LAR after  $m$  steps, where  $m$  is the desired size for the set  $S$ , and we pick as elements of  $S$  the  $m$  elements from  $\Sigma$  chosen by the algorithm so far. In this way we are certain to retain in  $S$  elements of  $\Sigma$  with a high correlation with GP's performance, thereby increasing the accuracy of the model over alternative ways of choosing  $S$ .

Of course, now we are faced with the question of how to choose the elements of  $\Sigma$ . Since problems  $\mathbf{t}$  and program behaviours  $\mathbf{p}$  are discretised as vectors of size  $\ell$ , we could pick the elements of  $\Sigma$  from the problem class itself. For small, discrete domains, one could use all the possible functions in the class of problems. If the number of problems is too big or infinite, one could draw a representative set of samples from the problem class. In this work we effectively used the latter approach. More specifically, we took  $\Sigma = T$ .

Although this procedure automates the process of selecting  $S$ , it still requires the user to specify the size of  $S$ . To free the user from this delicate duty, we decided to use a cross-validation technique on the training set to determine the best size for  $S$ . This works as follows. The training set is split into five sets of equal size. Four sets are joined together and are used to produce a model (i.e., these sets are now the training set) while the remaining set is used to assess its generalisation. That is, we use the created model to predict the performance of the GP system in the problems of the remaining set. The process is repeated 5 times, each time leaving out a different fifth of the training set  $T$ . As a result, we get a performance predictions for the problems in the training set.

<sup>1</sup> Typically,  $d(\mathbf{p}, \mathbf{t}) = \sum_i |p_i - t_i|^k$ , with  $k = 1$  (sum of absolute errors) or  $k = 2$  (RMS error).

We use these predictions and the actual GP's performance to measure the quality of models using the Relative Squared Error (RSE). The RSE is defined as  $rse = \frac{\sum_i (P_i - \tilde{P}_i)^2}{\sum_i (P_i - \bar{P})^2}$  where  $i$  ranges over the set of test problems used to evaluate the accuracy of a model,  $P_i$  is the average performance recorded for problem  $i$ ,  $\tilde{P}_i$  is the performance predicted by the model, and  $\bar{P}$  is the average performance over all runs. The objective is to obtain values of  $rse$  as close as possible to zero.<sup>2</sup>

Cross-validation was applied to models produced by LAR with  $m = 1, 2, \dots, |T|$  with the aim of identifying the value of  $m$  which provided the best generalisation. The process ends when the best generalisation on the training set has been reached. Note that while the process of identifying  $\mathcal{S}$  from  $T$  with a known  $m$  involves first running LAR and then doing least squares, to save computation during the determination of the optimal  $m$  we did not performed the least square method after LAR. Instead we simply used the models produced by LAR.

To compute the performance model we need to define a closeness measure  $d$ . Obviously, the accuracy of the model depends on the closeness measure used. In preliminary work we tested different closeness measures. Here we adopt the one that gave the model with best quality, namely  $d = (\mathbf{p} \cdot \mathbf{t})^2$  where  $\cdot$  is the scalar product.

To sum up, we create a performance model of a GP system by using the LAR algorithm to select the set  $\mathcal{S}$ , a cross-validation technique to set the size of  $\mathcal{S}$ , and, finally, a least square method to compute the coefficients  $a_p$ .

## 5 Similarity Matrices

We are now in the position to compute the matrix  $\mathcal{M}$  required to apply (1). We present three different methods to compute  $\mathcal{M}$ . Two of them use the coefficients  $a_p$  of the model obtained using two different similarity measures  $d$ , namely the angle and the Euclidean distance between vectors of coefficients  $a_p$ . The third method uses the mean and the standard deviation GP's performance on a training set to associate a pair  $(\mu, \sigma)$  to each GP system. The matrix  $\mathcal{M}$  is then constructed using the Euclidean distance between pairs of  $(\mu, \sigma)$  tuples. These methods are described in detail below.

It is possible to compute the angle between two GP systems by first noticing that (3) represents a hyper-plane. To see this, we rewrite (3) in normal form using the scalar product between two vectors, namely  $\mathbf{c} \cdot ((P(\mathbf{t}), d(\mathbf{p}_1, \mathbf{t}), \dots, d(\mathbf{p}_{|\mathcal{S}|}, \mathbf{t})) - \mathbf{x}_0) = 0$  where  $\mathbf{x}_0 = (a_0, 0, \dots, 0)$  is a particular point on the hyper-plane,  $\mathbf{c} = (-1, a_{\mathbf{p}_1}, \dots, a_{\mathbf{p}_{|\mathcal{S}|}})$ , and  $a_p$  are the coefficients of the performance model. Then, we can measure the angle (or dissimilarity) between two GP systems as the angle between the hyper-planes representing these systems. If the systems are characterised by the vectors  $\mathbf{c}' = (-1, a'_{\mathbf{p}_1}, \dots, a'_{\mathbf{p}_{|\mathcal{S}|}})$  and  $\mathbf{c}'' = (-1, a''_{\mathbf{p}_1}, \dots, a''_{\mathbf{p}_{|\mathcal{S}|}})$ , the angle between them is  $\alpha = \arccos\left(\frac{\mathbf{c}' \cdot \mathbf{c}''}{\|\mathbf{c}'\| \|\mathbf{c}''\|}\right)$ .

Naturally, for this idea to work, one needs to make sure that the coefficients of the two models are associated to the same programs  $\mathbf{t}$ . Because different models are built independently (using the procedure described in Sec. 4), this may not be the case. So, to avoid

<sup>2</sup> A value of  $rse$  close to 1 means that the model is as good (or bad) at predicting performance differences as the mean. A value of  $rse$  less than 1 means that the model predicts better than the mean, while  $rse > 1$  implies worse predictions than the mean.

problems, we used the following strategy. Let  $S'$  and  $S''$  be two sets associated to the performance models of two different GP systems,  $a$  and  $b$ . We start by instantiating two models, represented by the vectors  $c'_a = (-1, a'_{\mathbf{p}_1}, \dots, a'_{\mathbf{p}_{|S'|}})$  and  $c'_b = (-1, b'_{\mathbf{p}_1}, \dots, b'_{\mathbf{p}_{|S'|}})$ , which were obtained by creating performance models of  $a$  and  $b$  using set  $S'$ . We repeat the procedure, obtaining vectors  $c''_a = (-1, a''_{\mathbf{p}_1}, \dots, a''_{\mathbf{p}_{|S''|}})$  and  $c''_b = (-1, b''_{\mathbf{p}_1}, \dots, b''_{\mathbf{p}_{|S''|}})$ , respectively, but this time using  $S''$ . We then compute two angles for each pair of GP systems  $\alpha_1 = \arccos(\mathbf{c}'_a \cdot \mathbf{c}'_b / \|\mathbf{c}'_a\| \|\mathbf{c}'_b\|)$  and  $\alpha_2 = \arccos(\mathbf{c}''_a \cdot \mathbf{c}''_b / \|\mathbf{c}''_a\| \|\mathbf{c}''_b\|)$ . From these we finally compute the *angle between two GP systems* as  $\alpha = \frac{\alpha_1 + \alpha_2}{2}$ . By applying the procedure described above to all pairs of GP systems under study we obtain the elements of matrix  $\mathcal{M}$  one by one.

In the second method to compute  $\mathcal{M}$  we first re-write (3) as a scalar product between two vectors:

$$P(\mathbf{t}) \approx (a_0, \dots, a_{|S|}) \cdot (1, d(\mathbf{p}_1, \mathbf{t}), \dots, d(\mathbf{p}_{|S|}, \mathbf{t})). \quad (4)$$

Clearly, the performance model in (4) depends only on a vector  $\mathbf{c} = (a_0, \dots, a_{|S|})$ . We can therefore calculate the similarity between GP systems by measuring the Euclidean distance between their associated  $\mathbf{c}$  vectors.

As for the case discussed in Sec. 5, again for this to work we must make sure corresponding components of the  $\mathbf{c}$  vectors being compared are associated to the same elements of  $\Sigma$ . Here again we used the same approach, i.e., compute two distance measures and then average them.

The third distance matrix  $\mathcal{M}$  is computed using the statistics of GP's performance on a set of test problems. This set is the union of the training set  $T$  and validation set  $V$ . We define  $\mathbf{c} = (\mu, \sigma)$  where  $\mu$  is the mean and  $\sigma$  is the standard deviation of GP's performance.

These statistics were collected by running each GP system in the conditions described in the following section.

## 6 Test Problems, GP Systems and Parameter Settings

To test our approach, we used two, fairly standard GP systems. Both systems use subtree crossover and subtree mutation. One system is essentially identical to the one used by Koza [10] the only significant difference being that we select crossover and mutation points uniformly at random, while [10] used a non-uniform distribution. The other system is TinyGP (see [14, Appendix]). The main difference between the two systems is that the Koza-style system is generational, while TinyGP uses a steady-state strategy.

Tab.1 shows the parameters for the GP systems. We only use combinations of crossover and mutation rates such that  $p_{xo} + p_m \leq 100\%$ <sup>3</sup> because in GP crossover and mutation are mutually exclusive operators. For the Koza-style GP system, besides the traditional roulette-wheel selection, we also tested tournament selection with a tournament size of 2. When using the terminal set  $\mathcal{T}_1$ , we considered all possible combinations of the parameters presented in Tab. 1 for all three types of GP system (Koza-style system with fitness proportionate and tournament selection and TinyGP). We, therefore, obtained 36 different GP systems. Besides these systems, we included another set of systems using as terminal

<sup>3</sup> The systems with a crossover rate of 90% use a 10% rate of reproduction.

**Table 1.** Parameters used in the GP experiments

Function set	$\{AND, OR, NAND, NOR\}$
Terminal set	$\mathcal{T}_1 = \{x_1, x_2, x_3, x_4\}, \mathcal{T}_2 = \mathcal{T}_1 \cup \{\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4\}$
Crossover rate $p_{xo}$	100%, 90%, 50%, and 0%
Mutation rate $p_m$	100%, 50%, and 0%
Maximum tree depth used in mutation	4
Selection Mechanism	Tournament (size 2) and Roulette-wheel
Population size	5000, 1000, and 500
Number of generations	50
Number of independent runs	100

set  $\mathcal{T}_2$  and restricting the population size to be 1000. Therefore, in total we analysed 48 different GP systems. While this number is not very large, the variety of systems considered is sufficiently representative for the purpose of exemplifying our method for the automated creation of algorithm taxonomies in GP.

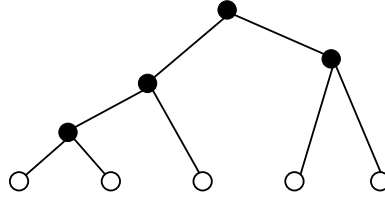
The training set  $T$  and validation set  $V$  are composed of Boolean induction problems with 4 inputs. We randomly selected 1,100 different Boolean functions from this set and for each one of them we counted the number of times the GP algorithm found a program that encoded the target functionality in 100 independent runs. We took as our performance measure the fraction of successful runs out of the total number of runs. Then these 1,100 randomly generated problems were divided into two sets: a training set  $T$  composed of 500 functions and the validation set  $V$  comprising the remaining 600 functions.

## 7 Taxonomies

Once the matrix  $\mathcal{M}$  is available, we can apply the hierarchical clustering algorithm described in Sec. 2 to progressively group our 48 GP systems into larger and larger clusters. These clusters form a hierarchy including 48 levels (one for each cycle of the clustering algorithm). The hierarchy can naturally be represented using a *dendrogram* such as the one shown in Fig. 1.

While dendrograms are a useful tool, they are not a taxonomy. Firstly, there are far too many classes for a taxonomy (the dendrogram for  $n$  objects includes  $n - 1$  classes). Secondly, the dendrograms produced in the presence of many objects (such as our 48 GP systems) are complex and difficult to digest. For example, it is difficult to appreciate the relative distance between clusters. So, how can we convert a dendrogram of (models of) GP systems into a taxonomy?

To transform dendrograms into taxonomies we first perform a data reduction exercise. We focus only on the 10 topmost clusters in the dendrogram starting from its root node. We then imagine that each of these 10 clusters is a node in a fully connected graph. We associate to each edge a weight defined by  $s(\mathcal{X}, \mathcal{Y})$ . We add to the resulting graph a further set of 48 nodes, one for each GP system, and we connect each of them to the node representing the (lowest level) cluster containing the corresponding GP system. We also associate a weight  $s(\mathcal{X}, \mathcal{Y})$  to these new edges. This new weight was defined to be the result of  $s(\mathcal{X}, \mathcal{X})$  when the GP system belongs to the class  $\mathcal{X}$ , thereby we can interpret it as the distance within cluster  $\mathcal{X}$ . This graph is our GP taxonomy.



**Fig. 1.** Example dendrogram: hollow circles represent class members, solid ones represent clusters

Of course, an important objective is to be able to visualise this taxonomy, to see what we can learn from it. There are number of ways one can draw graphs. In this paper, we decided to use the `neato` package, which is part of the `GraphViz` library.

`Neato` uses the following strategy to position nodes. Nodes are interpreted as physical bodies connected by springs (which correspond to the edges between nodes in the graph). The user can set both the spring stiffness and the rest length. To produce a layout, the (virtual) physical system is initialised in some suboptimal configuration, which is then iteratively modified until the system relaxes into a state of minimal energy. Often this relaxation process leads to nicely drawn graphs where groups of nodes with either strong connections or many connections tend to be placed next to each other (since this effectively minimises the associated elastic energy).

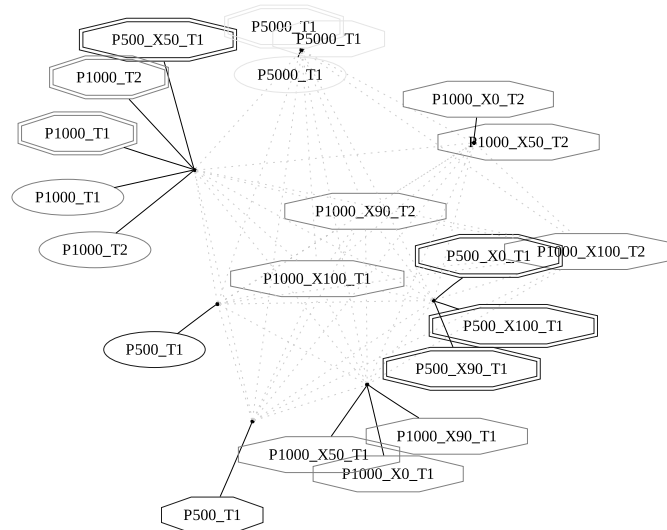
To produce graphical representations of our taxonomies we gave `neato` the weighted graphs described above, setting each spring's rest-length to 10 times the weight of the corresponding edge. The stiffness of springs was set to be approximately inversely proportional to weights using the formula  $\frac{1}{0.01+s(x,y)}$ .

Below we will show figures illustrating the results produced by `neato`. In each figure, edges between clusters are represented using dotted lines, while edges connecting GP systems to their mother cluster are drawn with solid lines. To label nodes we use the following nomenclature. Each system is represented by three terms: a) a term of the form `P####`, which represents the size of the population; b) a term of the form `X####`, which represents the crossover rate (as a percentage);<sup>4</sup> c) a term which indicates which terminal set was used. So, a node labelled `P500_X50_T1` represents a system with a population of size 500,  $p_{xo} = p_m = 0.5$  and terminal set  $\mathcal{T}_1$ . To distinguish different forms of selection and reproduction we use different graphical symbols to represent the corresponding nodes. Namely, the `TinyGP` systems are represented with ellipses, while the `Koza-style GP` systems with roulette-wheel selection and tournament selection are represented using octagons and double octagons, respectively. Finally, since in many cases a cluster contains all the GP systems with a particular population size and terminal set, we represent such systems with a single node. In these cases, we drop element (b) from the node label.

Fig. 2 shows the taxonomy obtained when  $\mathcal{M}$  is computed using the angles between GP performance models (interpreted as hyper-planes, as explained in Sec. 5). As one can see, the taxonomy includes a central cluster that is mainly populated by generational systems with small populations and tournament selection, which is surrounded by several satellite

<sup>4</sup> We did not include in the nomenclature the mutation rate because this can be inferred using the following rule: mutation is not used if  $p_{xo} = 90\%$ , otherwise its rate is given by  $p_m = 1 - p_{xo}$ .



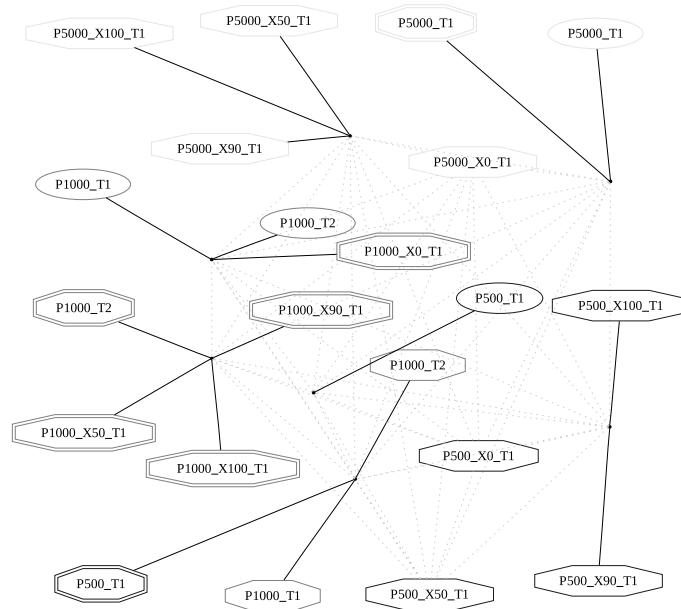


**Fig. 3.** Taxonomy created using the Euclidean distance between performance models. Octagon=Koza's GP w/ roulette selection, double octagon=Koza's GP w/ tournaments, ellipse=TinyGP.

taxonomies and it is of harder interpretation. One of the most notable differences is that this taxonomy does not group the systems with a population of 5000 individuals. Furthermore, both Fig. 2 and Fig. 3 indicated that systems with a population size of 5000 and the Koza-style systems with a population size of 500 are very distant performance-wise, while this information is not made explicit in the taxonomy shown in Fig. 4. This suggests that models of GP performance capture more information than simple raw statistics. Raw statistics (particularly mean performance) are constrained by the limits imposed by the no-free lunch [17]. They are unable to discover if two systems have distinct specialisation niches or whether they show similar average performance because they succeed and fail to the same degree on the same problems. The performance models presented above can. These are, therefore, better suited for the construction of algorithm taxonomies.

If we look at the taxonomies in Figs. 2 and 3 as a whole, we can see a fairly clear overall picture of the behaviour of GP systems over the class of Boolean problems.<sup>6</sup> In particular we find that behaviours tend to be relatively little influenced by the choice of crossover and mutation rates, while changes in the selection and reproduction schemes can give substantial performance differences. This in turn suggests that adjusting crossover and mutation rates is particularly useful if one has already found a good GP system for a problem (or group of problems) and now wants to get the maximum out of that system. Changing selection strategy, instead, may give better chances of success if the user is not satisfied

<sup>6</sup> While our taxonomies were built using a subset of all Boolean functions with 4 inputs, the accuracy of our performance model on the validation set  $V$  suggests that the model is in fact general. So, taxonomies based on it should represent reasonably well the behaviour of the 48 GP systems studied across the whole class of Boolean functions.



**Fig. 4.** Taxonomy created using Euclidean distances between vectors of empirical statistics of GP performance. Octagon=Koza's GP w/ roulette selection, double octagon=Koza's GP w/ tournaments, ellipse=TinyGP.

with a particular GP system and wants to find a better alternative. Furthermore, it is clear from the taxonomies that as the population size increases, also the choice of selection and reproduction scheme become less important in determining GP's behaviour. This suggests that another important taxonomic factor in relation to performance on Boolean induction problems is population size.

Before concluding this section we would like to mention that although we used figures to explain the structure of the taxonomies produced with our method and the relations between different GP systems, all our observations are also supported by the raw distance between the clusters formed. Unfortunately, due to lack of space we cannot present tables containing the similarities between the clusters and within each cluster.

## 8 Conclusions

In this paper we have presented an automatic procedure to produce taxonomies in GP. The procedure is based on performance models of GP systems. We use the models' parameters as signatures which are then clustered via a hierarchical algorithm to create dendrograms. The dendrograms are further processed to identify the key groups and their relationships. These are formalised into a weighted graph, which really represents our taxonomies. Plotting the graph with standard tools produces a highly informative representation of taxonomies from which a variety of lessons can be learnt. Since our taxonomies are based on performance, these lessons are particularly useful to guide practitioners.

Two lessons were particularly obvious in all taxonomies: population size and selection scheme are more important factors than crossover and mutation rates in determining performance on Boolean problems. So, the former should be changed first if one is not happy with the performance provided by a GP system, while the latter could be changed later to finely optimise an already satisfactory system.

Although in this contribution we have only presented taxonomies in relation to Boolean induction problems, preliminary results show that this procedure produces meaningful taxonomies of algorithms also for the class of rational symbolic regression problems (the class of functions used originally in [6]). In the future, we will explore this in depth. In addition, we hope to apply our taxonomic approach to some classes of real-world applications. Finally, while in this paper we explicitly focused on GP, nothing prevents the application of the method to other EAs. In the future, we intend to explore this avenue.

## References

1. Ashlock, D., Bryden, K., Corns, S.: On taxonomy of evolutionary computation problems. In: Proceedings of the 2004 IEEE Congress on Evolutionary Computation, Portland, Oregon, pp. 1713–1719. IEEE Press, Los Alamitos (2004)
2. Ashlock, D.A., Bryden, K.M., Corns, S., Schonfeld, J.: An updated taxonomy of evolutionary computation problems using graph-based evolutionary algorithms. In: Yen, G.G., Wang, L., Bonissone, P., Lucas, S.M. (eds.) Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pp. 403–410. IEEE Press, Los Alamitos (2006)
3. Back, T., Fogel, D.B., Whitley, D., Angeline, P.J.: Mutation operators. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 1 Basic Algorithms and Operators, ch. 32, pp. 237–255. Institute of Physics Publishing, Bristol (2000)
4. Calégari, P., Coray, G., Hertz, A., Kobler, D., Kuonen, P.: A taxonomy of evolutionary algorithms in combinatorial optimization. *J. Heuristics* 5(2), 145–158 (1999)
5. Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. *Annals of Statistics* (2004)
6. Graff, M., Poli, R.: Practical model of genetic programming’s performance on rational symbolic regression problems. In: O’Neill, M., Vanneschi, L., Gustafson, S., Esparcia-Alcázar, A., Falco, I.D., Cioppa, A.D., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 122–133. Springer, Heidelberg (2008)
7. Herrera, F., Lozano, M., Sánchez, A.M.: A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. *Int. J. Intell. Syst.* 18(3), 309–338 (2003)
8. Jansen, T., Wegener, I.: The analysis of evolutionary algorithms - a proof that crossover really can help. *Algorithmica* 34(1), 47–66 (2002)
9. Johnson, S.: Hierarchical clustering schemes. *Psychometrika* 32(3), 241–254 (1967)
10. Koza, J.R.: Genetic Programming. The MIT Press, Cambridge (1992)
11. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
12. Martin, P.: Building a taxonomy of genetic programming. In: Spector, L., Goodman, E.D., Wu, A., Langdon, W.B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), p. 182. Morgan Kaufmann, San Francisco (2001)
13. Nowostawski, M., Poli, R.: Parallel genetic algorithm taxonomy. In: Jain, L.C. (ed.) Proceedings of the Third International conference on knowledge-based intelligent information engineering systems (KES 1999), pp. 88–92. IEEE, Los Alamitos (1999)
14. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008), <http://lulu.com> <http://www.gp-field-guide.org.uk> (With contributions by Koza, J.R)

15. Poli, R., Logan, B.: The evolutionary computation cookbook: Recipes for designing new algorithms. In: Proceedings of the Second Online Workshop on Evolutionary Computation, Nagoya, Japan (March 1996)
16. Vose, M.D.: The simple genetic algorithm: Foundations and theory. MIT Press, Cambridge (1999)
17. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1(1), 67–82 (1997)
18. Zhang, B.-T.: A taxonomy of control schemes for genetic code growth. In: The Workshop on Evolutionary Computation with Variable Size Representation at ICGA 1997, July 20 (1997)