A Simple Approach to Lifetime Learning in Genetic Programming-Based Symbolic Regression

Raja Muhammad Atif Azad
CSIS Department, University of Limerick, Ireland
atif.azad@ul.ie

Conor Ryan
CSIS Department, University of Limerick, Ireland
conor.ryan@ul.ie

doi:10.1162/EVCO_a_00111

Abstract
Genetic programming (GP) coarsely models natural evolution to evolve computer programs. Unlike in nature, where individuals can often improve their fitness through lifetime experience, the fitness of GP individuals generally does not change during their lifetime, and there is usually no opportunity to pass on acquired knowledge. This paper introduces the Chameleon system to address this discrepancy and augment GP with lifetime learning by adding a simple local search that operates by tuning the internal nodes of individuals. Although not the first attempt to combine local search with GP, its simplicity means that it is easy to understand and cheap to implement. A simple cache is added which leverages the local search to reduce the tuning cost to a small fraction of the expected cost, and we provide a theoretical upper limit on the maximum tuning expense given the average tree size of the population and show that this limit grows very conservatively as the average tree size of the population increases. We show that Chameleon uses available genetic material more efficiently by exploring more actively than with standard GP, and demonstrate that not only does Chameleon outperform standard GP (on both training and test data) over a number of symbolic regression type problems, it does so by producing smaller individuals and it works harmoniously with two other well-known extensions to GP, namely, linear scaling and a diversity-promoting tournament selection method.

Keywords
Genetic programming, memetic algorithms, lifetime learning, symbolic regression, local search, hybrid genetic algorithms.

1 Introduction
The success of the so-called memetic algorithms (MAs; Norman et al., 1991; Moscato, 2003) has underlined the importance of local search in augmenting the global search of evolutionary algorithms (EAs; Goldberg, 1989; Holland, 1975). Unlike traditional EAs, MAs intrinsically exploit problem specific information to fine-tune the evolving solutions, giving them the opportunity to improve beyond their genetic makeup at birth. Thus, MAs go a step further to mimic natural evolution: not only does information spread through genes across generations, it can also spread within a generation, through imitating ideas, catch phrases, and fashion (Dawkins, 1990). Moreover, the particular experiences of an evolving entity may add to its survivability.

Manuscript received: May 22, 2012; revised: December 19, 2012 and July 26, 2013; accepted: September 19, 2013.
© 2014 by the Massachusetts Institute of Technology
Evolutionary Computation 22(2): 287–317
While there has been quite some work conducted on attaching such lifetime learning to EAs (see, e.g., Moscato, 1989; Östermark, 1999; Ozcan and Mohan, 1998; M.J. Bayley and Williamson, 1998; Dandekar and Argos, 1996; de Souza et al., 1998), genetic programming (GP; Koza, 1992; Banzhaf et al., 1998), a type of EA that evolves computer programs, reports a disproportionally smaller number of examples.

However, GP faces issues that suggest that it can benefit from lifetime learning with local search. For example, the tension between structure and content in GP is well documented (Ryan and Keijzer, 2003; Korns, 2009; Daida, 2004). That is, the eventual shape that the structures of the individuals take depends upon the availability of basic building blocks (functions, terminals, and/or subtrees) in the first few generations. If something is not available, or disappears (Hu et al., 2005; e.g., if a particular constant is absent) then evolution may work around its absence and produce a subtree to overcome the handicap.

There are costs to this, of course, such as extra time spent evolving the missing parts, a larger memory requirement due to larger trees, a tendency to grow too complex to generalise well across unseen data, and an increased danger of bloating because of the larger trees (Blickle and Thiele, 1994; Soule and Foster, 1998; Langdon, 2000; Poli, 2003).

Another issue with GP is that the population quickly becomes homogeneous; McPhee and Hopper demonstrated that up to 70% of the final population shared the top four levels with a single ancestor (McPhee and Hopper, 1999). If these levels are not optimal, or even not great, consequences for the rest of the run can be grave: already faced with evolving solutions to the problem at hand, GP now also has to work around the major handicap of the top levels interfering with the work of the lower ones.

Furthermore, in common practice of GP, the population learns at the expense of the individual: bad individuals go out of the population while the good individuals stay. To decide whether an individual is good or bad, typically, it is evaluated only once. Thereafter, if the individual is good enough, it can only breed offspring or spread multiple copies of itself. This is unlike the case in nature, where, in addition to breeding, the organisms test themselves multiple times against the environment and improve their behaviour through experience.

To address these issues, this paper introduces Chameleon, essentially a hill climbing GP algorithm that permits individuals, through a form of hypermutation, to change their internal function nodes during their lifetime. Tuning the leaves to optimise the coefficients or variables deserves a separate investigation; that is beyond the scope of this paper.

A question then arises as to whether such additional learning encourages overfitting the training data even more than in standard GP, and how expensive it is. The results demonstrate that this form of learning improves the best fit individuals on both training and test data while also decreasing the average tree size. In fact, we show that we can encourage smaller sized trees in the population as a very consequence of lifetime learning. To do this, instead of uniformly tuning every tree, we reward the shorter-than-average sized trees by giving them more than the average number of chances of tuning their internal nodes. Correspondingly, larger-than-average sized trees get less than the average number of tuning opportunities per node. Such a focus on smaller trees not only decreases tuning expense but also promotes smaller trees in the population.

Exhaustive tuning can appear expensive, particularly when the cardinality of the functions set increases, or as the trees grow larger; however, we demonstrate that with our choice of algorithm, the former is not necessarily true, whereas the latter (growth...
in tree size) can be curtailed by preferentially tuning smaller trees. Also, in Section 5 we formally quantify the maximum tuning expense.

The paper is laid out as follows: Section 2 gives the background to this work; Section 3 describes the proposed method, how it can be made computationally more efficient and how it can be used to induce parsimony pressure in the population; Section 4 discusses the experimental setup, enlists the results, and then comments on their significance; Section 5 theoretically analyses our method for the maximum tuning expense, gives an upper bound on it, and shows that even when used sparingly (as against using it on every individual), Chameleon can still outperform standard GP; and finally, Section 6 concludes the paper.

2 Background

Approaches to lifetime learning in GP can coarsely be grouped as external, internal, or cultural. The key difference between the first two approaches is the type of individuals they work with. Typical GP individuals are improved by the external approaches, that is, some form of expression trees, and improving these individuals with local search methods such as hill climbing. The internal approaches, however, work with individuals that incorporate an internal mechanism of learning by design, for example, when the individuals are neural networks or support vector machines. The third approach uses a notion of culture to share the learning across the population through some sort of implicit communication. Although Chameleon is very much in the first category, we briefly discuss each category in this section.

Although not as common as with non-GP EAs (see Moscato, 2003, for a detailed review of local search with non-GP EAs), examples of external approaches applying hill climbing to GP date from relatively early (Harries and Smith, 1997; Iba et al., 1994b; O’Reilly and Oppacher, 1994, 1996) to the more recent (Topchy and Punch, 2001; Zhang and Smart, 2004; Krawiec, 2001; Krasnogor, 2004; Nel, 2004; Majeed and Ryan, 2006; Wang et al., 2011). We now briefly review these studies.

The tuning of numeric coefficients has been a popular research topic in GP (Keijzer, 2004b; Iba et al., 1994a; Iba and Nikolaev, 2000; Nikolaev and Iba, 2001; McKay et al., 1999; Hiden et al., 1999; Hunter, 2002), and some studies have also employed local search for this purpose (Topchy and Punch, 2001; Krawiec, 2001; Zhang and Smart, 2004, 2005). Among these approaches, linear scaling (Keijzer, 2004b) has gained some popularity (Raja et al., 2008, 2007; Majeed and Ryan, 2006; Archetti et al., 2006) because it is a deterministic and computationally cheap technique. As we adopt it later in this paper, we explain it further, below.

Linear scaling deterministically optimises two linear parameters to minimise the sum of squared errors between target values, \( t(x) \), and approximate values, \( y(x) \), where \( x \) represents a vector of independent variables. The linearly scaled mean squared error (MSE) is calculated as:

\[
MSE(t(x), a + by(x)) = \frac{1}{N} \sum_{i=1}^{N} (t(x_i) - (a + by(x_i)))^2
\]

where

\[
b = \frac{\sum_{i=1}^{N} (t(x_i) - \bar{t})(y(x_i) - \bar{y})}{\sum_{i=1}^{N} (y(x_i) - \bar{y})^2}, a = \bar{t} - b\bar{y}.
\]

Keijzer (2003) has shown that linear scaling significantly boosts the performance of GP on training cases; results on test cases were not presented. Due to the simplicity and
widespread usage of linear scaling, we consider GP with linear scaling as a benchmark GP setup in Section 4. We verify whether linear scaling and Chameleon can combine harmoniously to produce an even better GP system.

As with tuning the external nodes (such as numeric leaves) GP literature also has some examples of tuning the internal nodes. Among these examples, O’Reilly and Oppacher (1994, 1996) used hill climbing and simulated annealing to improve a certain percentage of the evolving population with repeated mutation or crossover. Harries and Smith (1997) also tuned the evolving individuals, in their case by crossing them over with themselves and by mutating them; both crossover and mutation were applied several times in a hill climbing fashion. Krasnogor (2004) co-evolved local search heuristics and applied them to the evolving individuals successfully to compare protein structures. Nel (2004) optimised the threshold values in decision trees with directed increasing incremental local search (DILL); much like gradient descent methods use momentum to speed up convergence when the gradient is unchanging (Mitchell, 1996, p. 100), DILL increases the magnitude of change if fitness continues to improve with each iteration of local tuning. Majeed and Ryan (2006) and Majeed (2007) improved crossover by finding the best crossover point for the incoming subtree by trying every possible crossover site. The best crossover point was the one that maximised fitness after crossover. Zhang et al. (2007) repeatedly crossed over two parents in a hill climbing fashion to produce an offspring better than the parents. For the offspring thus produced, the corresponding crossover points in the parents were assigned a weight. This weight was used to protect the underlying subtrees from subsequent crossovers to promote larger building blocks in GP. As with Nel (2004), Wang et al. (2011) tuned the thresholds in decision trees to restrict the decision boundaries. Moreover, they also used a splitting operator to subdivide the decision subspaces and further tune the decisions.

In general, all these approaches combine standard GP with some sort of local (often greedy) search. Many of these methods have enjoyed success by combining the global search abilities of GP with local power of hill climbing type methods.

Another recent effort is that of Korns (2009), in which the system uses abstract grammars. Unlike standard grammar-based systems such as grammatical evolution (Ryan et al., 1998; O’Neill and Ryan, 2003), in which, once an individual maps an expression from a grammar it is ready to be evaluated, Korns would generate expressions containing placeholders, before using an external tuning process to decide upon the contents of these placeholders. Depending upon whether it requires a function or a terminal to replace it, the placeholder is called an abstract function or an abstract terminal. For example, an individual could be of the following form: \((f_1 \times (f_2 \times (f_3 \times f_4 \times f_5)))\) which has a mixture of abstract functions \(f_i\), concrete functions (e.g., \(\times\)), and constants. Next, a vector of functions is produced by a method of choice such as particle swarm optimisation, differential evolution, or even a GA from which each of the \(f_1..f_5\) are chosen. Thus, unlike the previous work, the tuning algorithm is not necessarily greedy. Next, the same or a different algorithm also tunes variables and constants in the expression.

This system can be very expensive: many parts of an expression are affected in each tuning iteration, thereby reducing the ability to cache the intermediate results, and the tuning algorithms can be as expensive as the user can afford. However, Korns has enjoyed some success with it, at least partly because it uses related methods for constant discovery and variable selection.

As we discuss in the following section, Chameleon is considerably simpler, partly because it only considers a single node at a time, which allows the results of the rest of the tree to be cached, but also because the lifetime learning it performs is passed on
to offspring as in Lamarckian evolution (i.e., changes made to an individual are passed on to offspring, e.g., Downing, 2001). Moreover, we show that it is possible to perform the learning in a manner that generates parsimony pressure in the GP population, thus further decreasing the computational expense.

The second approach to lifetime learning is the internal approach, that is, to let individuals control their own learning; although, clearly, this requires individuals to produce structures that are capable of doing so. Usually (Khan and Miller, 2009; Curran and O’Riordan, 2006), these methods generate a neural network of some description or a support vector machine (Howley and Madden, 2005) that can be further trained on the problem at hand.

While the spirit and intention of the second approach is essentially the same as what we are attempting to achieve here, the tuning method in Chameleon is so designed that a standard GP run can benefit from lifetime learning, regardless of whether or not the individual agents are active learning structures (e.g., a neural network).

The third approach to lifetime learning involves sharing information across the population or imitating other individuals to learn from their experience in addition to genetic inheritance (Spector and Luke, 1996; Zannoni and Reynolds, 1997; Eskridge and Hougen, 2004; Meuth, 2010). Such a culture of sharing information can occur within a generation (intra-generational), across generations (inter-generational), or across runs (inter-run).

Spector and Luke (1996) implement intra-generational learning with a shared memory which each individual can write to and read from. Thus, the end results of various computations during fitness evaluation remain in this memory, and other individuals can read these results. Spector and Luke hypothesise that such a culture of sharing information improves the overall performance of GP.

Among the other two cultural approaches, inter-generational learning attempts to encapsulate potentially useful genetic material into a single node during a run so that, when crossover occurs, the material is less likely to be broken up (Angeline and Pollack, 1992; Rosca and Ballard, 1996). The inter-run methods use a form of cascading runs, in which potentially useful genetic material discovered in one run is immediately available at the start of the following one (Ryan et al., 2004; Murphy et al., 2007; Keijzer et al., 2005; Meuth, 2010). However, the cultural learning algorithms are beyond the scope of this paper; we do not discuss them any further.

In spite of all these advances, however, the overwhelming majority of GP users still use standard GP without any hill climbing additions. Why is this? We believe it is because of fear of expense. Expense not only in terms of implementing the algorithm, but also in terms of how quickly it will run: after all, almost by definition, each iteration of hill climbing per individual adds to fitness evaluation count. For example, Krawiec (2001) limited local learning to just a single individual per generation to minimise the runtime costs.

In this paper, we introduce a system that improves performance, reduces cost (smaller size of individuals), is trivial to implement, and can have an upper bound of cost imposed upon it.

3 Lifetime Learning with Chameleon

We propose to locally tune each internal node in a GP tree, one node at a time, in a top-down fashion. First, we evaluate the tree in the standard fashion to establish a baseline fitness value. Then, starting with the root node, at each internal node we iterate through the entire set of applicable functions, that is, we respect the original arity of
each node, which means that all the changes are point changes, rather than structural changes. As with Korns' approach, the node undergoing tuning acts as a placeholder or as an abstract function; however, the rest of the tree remains fully specified. The whole process is akin to mutating a node exhaustively.

This exhaustive mutation iteratively attempts to improve the fitness of an individual. After each mutation, we evaluate the tree for fitness; if the new fitness is higher than the best result so far, the change is accepted into the tree, otherwise, it is discarded. While in this case we traverse the tree in a depth-first and top-down fashion, other approaches (e.g., breadth-first, bottom-up) may also be useful.

The bigger the tree, the more expensive it is to tune. However, the same is not necessarily true if the functions set increases in size. Let \( F = F_1 \cup F_2 \cup \cdots \cup F_j \) be the functions set such that \( F_j \) is the set of functions with arity \( j \); let \( C_{\text{max}} = \max_j |F_j| \) such that \( |F_j| \) is the cardinality of \( F_j \) and \( C_{\text{max}} \) is the highest cardinality among all \( |F_j| \), and let \( i \) be the number of internal nodes of the tree under consideration. The additional number of fitness evaluations carried out during one complete sweep of tuning \( n \leq i(C_{\text{max}} - 1) \). Let \( s \) be the tree size, and \( N_1 \) be the number of nodes processed during \( n \) fitness evaluations then \( N_1 \leq i(C_{\text{max}} - 1)s \). Note, though, that this upper limit on \( N_1 \) (or \( n \)) increases with \( C_{\text{max}} \); however, it remains unaffected if we add some \( F_k \) to \( F \) so that \( |F_k| \leq C_{\text{max}} \) even though the overall functions set still gets larger. For example, if we add unary functions to a functions set that previously only contained binary functions, then, because we iterate only over the type of functions that an individual already has in it, the tuning expense does not increase for an individual that only has binary functions in its internal nodes.

### 3.1 Cost Reduction

Before we compare tuning costs against benefits, we employ a simple technique to avoid repetitive evaluations in a tree, that is, caching the intermediate results. Keijzer (Keijzer, 2004a) has proposed vectorised evaluation of input data points and caching of the corresponding outputs of subtrees in a GP population. While Keijzer and others (Downey and Zhang, 2011; Wong and Zhang, 2007) use a population-wide cache, here we only consider a cache local to the tree under consideration.\(^1\) This cache is very efficient, as the same subtrees will be evaluated several times for a particular individual. We implement it as a two-dimensional floating point array of size \( \text{tree\_size} \times |\text{data\_set}| \). To further aid efficiency, we use a large cache for each tree; that is, the cache is cleared out at the start of each individual’s evaluation. Occasionally, when encountering exceptionally large trees, the cache can be resized. However, this is less frequent and, as we discuss in Section 3.2, can be discouraged by containing growth in tree size.

We now quantify how much the cache saves us when evaluating the same tree multiple times while tuning it. Note that the output of every node is cached when the tree is evaluated for the first time. During the tuning performed by Chameleon, this output is retrieved from the cache and only the nodes on the path to the root node are reevaluated; Figure 1 exemplifies this. However, the number of nodes on the path to the root node is simply one more than the depth of the tuning site: the depth of the root node is 0. Let \( d_k \) be the depth of the \( k \)th internal node, and for simplicity let \( |F_j| = C_{\text{max}} \) for all \( j \) so that the number of nodes processed without the cache \( N_1 = i(C_{\text{max}} - 1)s \).

\(^1\)When implementing it, we require two caches: the additional cache holds the results from the last iteration of tuning. We swap these results back into the main cache if the current iteration does not improve fitness.
Figure 1: (Left): Local caching is demonstrated: (a) during normal evaluation, the results of every node are placed in the cache; (b) during tuning only, the nodes on the path to the root node are evaluated afresh while the cache is used for the rest. (Right): The cache can significantly decrease the cost, that is, the number of nodes processed while tuning. The figure plots percentage decrease for skinny (with only unary functions) and maximally grown binary trees.

Then the number of nodes processed with the cache is:

\[ N_2 = (C_{\text{max}} - 1) \sum_{k=1}^{i} (d_k + 1) = i(C_{\text{max}} - 1)(\bar{d} + 1) = \frac{\bar{d} + 1}{s} N_1 \]

where \( \bar{d} \) is the average depth of the internal nodes of the tree under consideration. This average depth is strictly smaller than the tree size, that is, \( s \); therefore, \( N_2 \) is only a fraction of \( N_1 \). Appendix A shows that the same result holds for the general case where \( |F_j| \neq |C_{\text{max}}| \) for at least some \( j \), that is, the number of functions in each subset of arity \( j \) is not necessarily the same. This confirms that using the cache decreases the tuning cost. The exact value of this decrease depends upon the size and shape of the tree under consideration; however, Figure 1 plots the percentage decrease for skinny trees that only use unary functions, and maximally grown binary trees. For the skinny trees the cache constantly decreases the tuning cost by 50%; however, for the maximal binary trees the decrease asymptotically approaches 100%.

### 3.2 Probabilistic Tuning and Parsimony Pressure

Even with caching, tuning incurs an additional computational cost after a standard fitness evaluation. What can exacerbate the cost further is that the larger the tree, the more tuning sites it has. Thus, larger trees have more opportunities to improve their fitness than smaller ones, which may encourage growth in tree size in the population. This is somewhat similar to standard GP, where the (generally) inferior average fitness of smaller trees helps increase the average tree size of the population (Poli, 2003) as larger trees tend to have a higher relative fitness. To avoid promoting larger trees with Chameleon (and, in turn, incurring an even greater tuning expense), we propose rewarding smaller trees with more chances of tuning than the larger ones. To achieve this, we make the probability of tuning each node of a tree a function of the corresponding tree size (\( s \)). Let \( p(s) \) be such a probability, and let \( \bar{s} \) be the average tree size of the
Figure 2: Function $p(s)$ from Equation (1) is depicted. $\bar{s}$ is the average tree size, $p(\bar{s} = 0.5)$, $p(s \leq 0.5\bar{s}) = 1$ and $p(s \geq 1.5\bar{s}) = 0$. 

population, then,

$$p(s) = \begin{cases} 
  y = c - \frac{s}{\bar{s}} & \text{if } 0 \leq y \leq 1 \\
  1 & \text{if } y > 1 \\
  0 & \text{otherwise.}
\end{cases} \quad (1)$$

Figure 2 shows a plot of Equation (1). In this study we use $c = 1.5$. Thus, an average sized tree has a per node probability $p(\bar{s}) = 0.5$. When $s \leq 0.5\bar{s}$, the probability rises to 1.0, whereas when $s \geq 1.5\bar{s}$, it drops to 0.0. This means that trees smaller than the average sized trees in the population are more likely to be tuned. If we increase the value of $c$ in Equation (1), the graph in Figure 2 shifts right, thus increasing the tuning opportunities for the larger trees. Decreasing $c$ has the opposite effect.

Note that Equation (1) is neither the only, nor necessarily the optimal, way of modelling $p(s)$; infinitely many functions may model $p(s)$. However, in using Equation (1), we present one such approach that targets the twin objective of keeping tuning expense low and discouraging growth in tree size. Later, Section 5 formally shows that Equation (1) conservatively increases the tuning expense as the average tree size increases.

Of course, irrespective of whether or not they get tuned, all the trees undergo normal evaluation. Therefore, if a large tree can outperform its smaller counterparts in spite of their increased tuning opportunities, it will survive. This makes our strategy different from those methods using parsimony pressure that subjectively decrease the fitness of a large tree (Poli and McPhee, 2008), or probabilistically kill the individuals with larger than the average size (Poli, 2003).

However, as in Poli (2003), our strategy also relies on a theoretical prediction that, if the average fitness of smaller individuals is greater than that of the larger ones, the average size in the next generation will be smaller than that in the current generation. Still, to contain the growth in tree sizes in this study, we do not lower the fitness values of the larger trees; instead, we offer a greater chance for the smaller individuals to prosper. Using a carrot and stick terminology, we rely on carrots alone.

With probabilistic tuning, the number of nodes undergoing tuning is likely to be substantially smaller than that with unconditional tuning. Thus, the training performance with the former may suffer; however, correspondingly, the tree sizes may
also be smaller if tuning rewards smaller trees in an effective manner. The results in Section 4.3 show that this is indeed the case.

4 Experiments

To estimate the effectiveness of the proposed method, we use five different GP approaches. These are \textit{(corresponding legends in parentheses)}: standard genetic programming (GP), that is, the state of the art; GP with Chameleon function tuning (tune); GP with probabilistic use of Chameleon function tuning (ptune); GP with linear scaling (Keijzer, 2003; to improve the numeric coefficients), and a tournament selection scheme to prevent mating between parents with identical fitness values (that is, no same mates; Gustafson et al., 2005) \((sc1 + nsm)\); and probabilistic function tuning combined with the previous setup \((ptune + sc1 + nsm)\). GP enhanced with linear scaling and no same mates (NSM) tournament selection offers a tougher benchmark: when the two techniques combine with GP they perform significantly better than standard GP on both training and the unseen (test) data (Costelloe and Ryan, 2009). Section 2 described linear scaling; we describe NSM below.

Gustafson et al. (2005) showed that NSM tournament selection reduces the probability of producing offspring identical to their parents, and delays stagnation in evolutionary search. To achieve this goal, NSM selection chooses genetically distinct parents. Since different fitness values guarantee a distinct genetic makeup, standard tournament selection is repeated until two parents with different fitness values are obtained. As with linear scaling (Keijzer, 2003), Gustafson et al. only showed training results where GP gained significantly from the use of this technique.

Alone, neither linear scaling nor NSM tournament outperforms standard GP on test cases on a selection of symbolic regression problems (Costelloe and Ryan, 2009); however, when these two methods combine, then together they do. Therefore, in this study if results still improve even more with a triple combination also involving Chameleon, it would be a significant step forward.

4.1 Test Suite and GP Parameters

We use eight problems for this study: four low dimensional problems (one or two input variables), and four high dimensional problems (8–241 input variables). The low dimensional problems are listed in Table 1. Among these, problems 1 and 2 are taken from Keijzer (2003); problem 3 is a bi-variate version of problem 2, and problem 4 is a bi-variate version of a polynomial used in Gustafson et al. (2005). 50 and 200 data points are used for training and testing, respectively, in each of these problems. As these problems are less challenging than their higher dimensional counterparts, we use test sets larger than the training sets to focus on how well a particular set up generalises. The high dimensional problems are listed in Table 2. We describe each of them separately in the following paragraphs; however, for all of these problems, we use 70\% of the randomly chosen data points for training during evolutionary runs, and the remaining 30\% for testing.

In the first high dimensional problem, Dow Chemical, the objective is to predict a real valued chemical composition from 57 process measuring input variables such as temperature, pressures, and flows. This problem was posed as a challenge for GP based
Table 1: Low dimensional problems used in this study.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Training set [min : step]</th>
<th>Test set [min : step]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 arcsinh(x)</td>
<td>[0.0 : 1.0]</td>
<td>50 points</td>
</tr>
<tr>
<td>2 $x^3e^{-x}\cos(x)\sin(x)(\sin^2(x)\cos(x) - 1)$</td>
<td>[0.0 : 0.2]</td>
<td>200 points</td>
</tr>
<tr>
<td>3 $y^3e^{-x}\cos(y)\sin(x)(\sin^2(y)\cos(x) - 1)$</td>
<td>$x [0.0 : 0.2], y = x + 0.03$</td>
<td></td>
</tr>
<tr>
<td>4 $y^2x^6 - 2.13y^4x^4 + y^6x^2$</td>
<td>$x [-1.9 : 0.075], y = x + 0.015$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: High dimensional problems used in this study.

<table>
<thead>
<tr>
<th>Problem Label</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dow Chemical</td>
<td>57</td>
<td>1,066</td>
</tr>
<tr>
<td>Boston Housing</td>
<td>13</td>
<td>506</td>
</tr>
<tr>
<td>Concrete Strength</td>
<td>8</td>
<td>1,030</td>
</tr>
<tr>
<td>Bioavailability</td>
<td>241</td>
<td>359</td>
</tr>
</tbody>
</table>

In the second problem, Boston Housing, the objective is to model house prices by using 12 real valued input variables and one binary valued input variable. The source of data is UCI Machine Learning Repository (Frank and Asuncion, 2010). Originally, Harrison and Rubinfeld (1978) reported on this problem; it uses data collected from the suburbs of Boston, Massachusetts.

In the third problem, Concrete Strength, the objective is to predict a quantitative value of compressive strength of concrete. This strength is a highly nonlinear function of eight input variables. Again, the data source is UCI Machine Learning Repository and the problem itself is detailed in Yeh (1998).

In the last problem, Bioavailability, the objective is to predict the percentage of an orally submitted dose of a drug that effectively reaches the systemic blood circulation from 241 input variables. This problem was first tackled with GP in Archetti et al. (2006).

In the second problem, Boston Housing, the objective is to model house prices by using 12 real valued input variables and one binary valued input variable. The source of data is UCI Machine Learning Repository (Frank and Asuncion, 2010). Originally, Harrison and Rubinfeld (1978) reported on this problem; it uses data collected from the suburbs of Boston, Massachusetts.

In the third problem, Concrete Strength, the objective is to predict a quantitative value of compressive strength of concrete. This strength is a highly nonlinear function of eight input variables. Again, the data source is UCI Machine Learning Repository and the problem itself is detailed in Yeh (1998).

In the last problem, Bioavailability, the objective is to predict the percentage of an orally submitted dose of a drug that effectively reaches the systemic blood circulation from 241 input variables. This problem was first tackled with GP in Archetti et al. (2006).

Table 3 details experimental parameters for this study. We did not optimise these parameters, except the population size. Initially, we tried population sizes of 120, 500, and 1,000 for GP. GP, both with and without linear scaling and NSM selection, performed the best with a population size of 500; therefore, we use this size for all the experiments reported here. A question arises as to whether a tournament size of 2 is too small for such a population size, that is, the sampling probability of each individual may diminish with implications towards population diversity. However, Xie et al. (2007a, 2007b) show

symbolic regression, and the data originates from a real industrial application at Dow Chemical.

In the second problem, Boston Housing, the objective is to model house prices by using 12 real valued input variables and one binary valued input variable. The source of data is UCI Machine Learning Repository (Frank and Asuncion, 2010). Originally, Harrison and Rubinfeld (1978) reported on this problem; it uses data collected from the suburbs of Boston, Massachusetts.

In the third problem, Concrete Strength, the objective is to predict a quantitative value of compressive strength of concrete. This strength is a highly nonlinear function of eight input variables. Again, the data source is UCI Machine Learning Repository and the problem itself is detailed in Yeh (1998).

In the last problem, Bioavailability, the objective is to predict the percentage of an orally submitted dose of a drug that effectively reaches the systemic blood circulation from 241 input variables. This problem was first tackled with GP in Archetti et al. (2006).

Table 3 details experimental parameters for this study. We did not optimise these parameters, except the population size. Initially, we tried population sizes of 120, 500, and 1,000 for GP. GP, both with and without linear scaling and NSM selection, performed the best with a population size of 500; therefore, we use this size for all the experiments reported here. A question arises as to whether a tournament size of 2 is too small for such a population size, that is, the sampling probability of each individual may diminish with implications towards population diversity. However, Xie et al. (2007a, 2007b) show

2A. Kordon presented this challenge at EvoStar 2010. For details see: http://casnew.iti.upv.es/index.php/evocompetitions/105-symregcompetition
Table 3: Configuration parameters for the runs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>500</td>
</tr>
<tr>
<td>Run Terminates at</td>
<td>exhausting $1.2 \times 10^6$ nodes or after 75,000 fitness evaluations</td>
</tr>
<tr>
<td>Operator probabilities</td>
<td>crossover: 0.9; Point mutation: 0.1</td>
</tr>
<tr>
<td>Tournament size</td>
<td>2</td>
</tr>
<tr>
<td>Replacement</td>
<td>Steady state, inverse tournament</td>
</tr>
<tr>
<td>Functions set</td>
<td>${+,-,\times,/,}$</td>
</tr>
<tr>
<td>Terminal set</td>
<td>${\text{Input variables}} \cup \text{ERC}$</td>
</tr>
<tr>
<td>ERC</td>
<td>${c</td>
</tr>
<tr>
<td>Normalised Fitness</td>
<td>$\frac{1}{1 + MSE}$</td>
</tr>
<tr>
<td>Initialisation</td>
<td>Ramped half and half (max. initial depth = 4)</td>
</tr>
</tbody>
</table>

that this probability remains constant if the number of tournaments conducted every generation is equal to the population size. We adhere to this formula.

To ascertain overfitting, the end-of-run results in Section 4.3 may not be enough. Therefore, we also provide the test set results over the entire runs in Appendix C.

The choice of functions set deserves attention. Often, parsimony pressure in GP focuses just on the size of the evolved expression (e.g., Poli, 2003; Luke and Panait, 2002). This ignores the underlying complexity of the expression and the fact that complex models can overfit the training data. For example, $(x + x + x)$ has a larger tree size than $\sin(x)$, yet the latter has a far more complex behaviour. In order to maintain a more reasonable size-complexity relationship, we have omitted transcendental functions in this study.

Since the ratio between the number of ephemeral random constants (ERCs) and input variables varies from problem to problem, the probability of their selection also varies. To keep a uniform probability of selecting ERCs across different problems, first we decide between a constant and a variable with a probability of 50%. Then, among themselves, both variables and constants are selected randomly.

4.2 How To Compare Performance?

To fairly compare the performances of different evolutionary algorithms, they should be allowed the same computational expense. Often this means identical number of generations or (by extension) fitness evaluations for every run. However, in GP, the same number of fitness evaluations can incur very dissimilar computational costs. For example, a fitness evaluation in the later generations may consume much more CPU time than that in the initial generations when the trees are much smaller in size. Therefore, another measure of computational effort is the amount of genetic material (or simply the number of nodes in GP trees) processed, for example, as in Silva and Costa (2005).

During each iteration of cached tuning, Chameleon does not process the same amount of genetic material as in a normal fitness evaluation with standard GP. In standard GP, each fitness evaluation fully traverses a tree, whereas during a Chameleon tuning evaluation, only the nodes on the path joining the root node to the node subjected
to tuning are evaluated afresh (see Figure 1). Moreover, a standard fitness evaluation can, potentially, sample an entirely different tree each time; however, during a Chameleon tuning evaluation, only a point change is made to an otherwise identical tree. Therefore, it makes more sense to compare standard GP with Chameleon on the basis of the number of processed nodes. A counterargument, then, is that in this way Chameleon avails itself of additional decision making opportunities (fitness evaluations) per unit genetic material processed, which standard GP cannot. However, that is precisely the point: with Chameleon, we propose to use the genetic material more efficiently by actively exploring it.

Still, we present results in terms of both processed nodes and fitness evaluations. Since Chameleon is at a disadvantage with the latter, the performance over the training sets can suffer. Runs terminate after processing $1.2 \times 10^6$ GP tree nodes, or after 75,000 fitness evaluations, depending upon the preselected criterion.

4.3 Results

Figures 3–4 show the results for the problems considered here. For each run we record the best training score (also referred to as fitness), the corresponding test score, and the average tree size of the population. The figures then show the mean of the training score (fitness value) of the best individual in each of the runs, the mean of their corresponding test scores, and the mean of the average tree size in each run.

The rationale for noting test scores is not to decide the best method for avoiding over-fitting; instead, it is to investigate whether tuning with Chameleon over-trains the evolving individuals so they over-fit even more than with standard GP. While, to aid readability, we only present the end-of-run results in this section, Appendix C gives mean test scores for the best individual throughout the evolution. Therefore, the reader interested in the entire trend of test set performance should refer to Appendix C.

The training and test scores are normalised between 0.00 and 1.00 (1.00 being the ideal score). Each sampled point in the plots depicts an average over 500 independent runs. As in Costelloe and Ryan (2009), the 95% confidence limits of the error bars at each point are computed as follows:

$$\bar{x} \pm 1.96 \frac{\sigma}{\sqrt{n}}$$

where $\bar{x}$ and $\sigma$ are the mean and standard deviation of $n$ observations; $n = 500$ represents the number of runs in this case. We can be 95% confident that the statistical population lies within these limits, and that a lack of overlap with another error bar means that the corresponding populations are different.

We also validate the significance of the differences in performance with the Mann-Whitney-Wilcoxon test (Anderson et al., 2001); we test at $p = 0.05$. It is a nonparametric test that does not assume normality of the sample populations. However, for the results presented in this section, this test agreed consistently with the graphical test described earlier that checks for an overlap of the error bars.

While the results for all the setups are plotted together, GP is the performance benchmark for tune and ptune as is scl + nsm for ptune + scl + nsm. This separation of benchmarks is due to the advantage available to GP with the additional computational expense of applying linear scaling and NSM tournament. Thus, separating benchmarks in this fashion helps compare like with like.

Note that in Figures 3 and 4, for the last three problems, the training and test results of all the GP setups were out of scale with respect to the results on the other five problems. As a result, we partition the results for the last three problems separately.
Figure 3: Normalised mean best training (top), and the corresponding test (bottom) scores at the end of the runs are plotted. For each of the last three problems, the figure rescales y-axis because the results are out of scale with respect to those for every other problem. For these problems the mean values are (specified as $[\text{min}, \text{max}]$ in the order: B-Housing, Concrete and Bioavail): (training) $[0.0763001, 0.118518]$, $[0.00757136, 0.0165866]$ and $[0.00117998, 0.00138766]$; (test) $[0.0101093, 0.0166162]$, $[0.00539519, 0.00830962]$ and $[0.000677607, 0.00105967]$. Runs end after processing $1.2 \times 10^6$ GP tree nodes.

For each of these three problems, the y-axis is rescaled every time; therefore, the plots in Figures 3 and 4 only show the relative performances of the GP setups. However, the figure captions describe the magnitudes of these results by specifying the minimum and maximum values for the three problems. Also, note that although the scales of figures go beyond the ideal value of 1.00, the actual results never exceed that value.

Figure 3 shows the scores on training and test sets after processing a fixed number of nodes. Training results show that on six out of eight problems, ptune + scl + nsm
Figure 4: Normalised mean best training (top), and the corresponding test (bottom) scores at the end of the runs are plotted. As in Figure 3, for each of the last three problems the figure rescales the y-axis. For these problems, the mean values are (specified as $[\text{min}, \text{max}]$ in the order: B-Housing, Concrete, and Bioavail): (training) $[0.074728, 0.135479]$, $[0.00613116, 0.0215204]$ and $[0.00114992, 0.00160564]$; (test) $[0.00866396, 0.0160436]$, $[0.00414232, 0.00869288]$ and $[0.000862427, 0.000827737]$. Runs end after processing 75,000 fitness evaluations.

performs the best; on the other two problems ($\text{arcsinh}(x)$ and $\text{sextic}_2d$) its performance is indistinct from the best. The corresponding test results show that, except on one problem (Bioavailability), the same setup, $p_{\text{tune}} + \text{scl} + \text{nsm}$, is at least as good as the best and, on five problems, is strictly better than its counterpart, $\text{scl} + \text{nsm}$. The figure also shows that GP is the worst performer on all the problems on both training and test sets except on Bioavailability where it performs the best on the test set.

Figure 4 shows the scores on training and test sets after a fixed number of fitness evaluations. Training results show a role reversal: $p_{\text{tune}} + \text{scl} + \text{nsm}$ lags behind
Figure 5: Mean of average tree size at the end of the runs is plotted: (top) runs end after processing $1.2 \times 10^6$ GP tree nodes; (bottom) runs end after 75,000 fitness evaluations (note the log scale in this case). The error bars are very small in the figures because the tree sizes are very consistent across different runs for each setup. Unlike in Figures 3 and 4, the $y$ axis was not rescaled.

$scl + nsm$ on six out of eight problems; $scl + nsm$ is indistinct from the best performers on the other two problems. Similarly, on training scores, GP outperforms $tune$ and $ptune$ on six problems. However, the situation again reverses on the all-important test results: on all the problems $ptune + scl + nsm$ is at least as good as $scl + nsm$ and strictly better on five of them; similarly, the two remaining Chameleon setups outperform GP on all the problems.

Figure 5 shows the average tree sizes for both criteria for run termination, that is, maximum allowable nodes and fitness evaluations. Clearly, in both the cases, the average tree size with all the variants of Chameleon is much smaller than that with the two variants of standard GP. The difference in size is even larger with maximum fitness evaluations: note the log scale. Among the variants of Chameleon, $ptune$ either has the shortest tree size, or it is indistinct with those with the shortest tree size.
4.4 Discussion

The results clearly show that GP, both in standard (GP) and enhanced (scl + nsm) versions, performs better with Chameleon. Even when Chameleon fails to improve results on the training data when fitness evaluations are constant, it compensates for that with a superior performance on the test data. Also, with Chameleon, the average tree size is significantly smaller than without it, even though we have taken no additional bloat controlling measure.

Maximally tuning the internal nodes of trees, tune performs consistently better than GP (except on training results with fixed fitness evaluations), and often performs better than its probabilistic counterpart, ptune. tune also maintains an average tree size smaller than that with GP, scl + nsm, and, rather surprisingly, ptune + scl + nsm. This is encouraging as it dispels the concern that local tuning in such an exhaustive fashion could be prohibitively expensive. It is also surprising that, despite being potentially favourable to larger trees, tune generates trees smaller than those even with ptune + scl + nsm which favours smaller trees. The question arises of whether rewarding the smaller trees with lifetime learning (as opposed to indiscriminately using it) has any effect at all in reducing the average tree size of the population.

To understand the apparently anomalous results in terms of average tree sizes, we compare the average number of generations each GP setup takes until the end of the run; Figure 6 presents this comparison. By measuring the number of generations, we compare the effect of pressure of selection and replacement on the population members. Typically, the higher the number of generations, the larger the trees are in the population. In a steady state setup (as employed in these experiments) with standard GP, the number of generations is simply the number of fitness evaluations divided by the population size. However, this formula does not hold for Chameleon. This is because during tuning an individual undergoes several fitness evaluations before it faces selection pressure. Also, note that while tuning, the size of a tree does not change; therefore, even with the same number of fitness evaluations, the number of generations elapsed with Chameleon can be very different from that with standard GP. Similarly, this number can be different between different variants of Chameleon or even between different runs with the same Chameleon setup. Therefore, we measure the number of generations explicitly.

Figure 6 tells us why tune can have surprisingly small trees. Both with a fixed number of nodes and fitness evaluations, it takes the least number of generations. While so, it exhaustively explores the internal nodes of earlier generations (often under 10 generations) and outperforms GP. While that is remarkable, it also clarifies that in this case the population does not face as much selection pressure over time as the rest of the setups. Earlier work (Azad and Ryan, 2010) also showed that if we increase selection pressure for tune by reducing the population size to 50, the average tree size becomes indistinct with that of standard GP; however, ptune still manages smaller tree sizes despite using a smaller population size than GP. Also, Figure 6 only reports the results at the end of the runs for consistency and due to space restrictions; however, the detailed results show that given the same number of generations the average tree size with ptune is significantly smaller than that with tune.

Figure 6 also shows that when the runs terminate after processing a fixed number of nodes, the number of generations taken by each of ptune and GP are very similar. Thus, that probabilistic tuning outperforms GP while keeping the trees smaller shows that probabilistic tuning successfully exploits the potential of smaller trees over the problems examined. Moreover, it answers the question raised earlier in this section:
Figure 6: Mean of number of generations elapsed at the end of the runs is plotted: (top) runs end after processing $1.2 \times 10^6$ GP tree nodes; (bottom) runs end after 75,000 fitness evaluations. The number of elapsed generations helps compare the selection pressure over time; thus, when $p_{tune}$ consumes the same number of generations as GP (top), the smaller tree sizes with the former directly result from the efficacy of Chameleon tuning as a reward for smaller trees. Unlike in Figures 3 and 4, the $y$-axis was not rescaled.

Lifetime learning is indeed an incentive attractive enough to promote smaller trees that benefit from tuning. This is also significant because the fitness is not decreased owing to size of individuals: if a large individual performs better than the smaller and tuned individuals, then it deserves to retain its competitive edge and so it does. However, we do not suggest that this is the best way of controlling bloat in GP; comparing bloat control techniques is beyond the scope of this paper.

Figure 6 also shows that with a fixed number of fitness evaluations, the number of generations that the two standard GP setups take far exceeds that taken by all the variants of Chameleon. For GP, the number of generations both with and without linear scaling are identical; hence, the corresponding legends are exactly on top of each other.
This also explains why the tree size for standard GP with fixed evaluations is much greater than that with a fixed number of nodes. The number of generations with the former is much higher than that with the latter.

Also, significantly, \( ptune + scl + nsm \) outperforms \( scl + nsm \). Thus, probabilistic tuning improves over a known improvement to GP and not just over standard GP.

Although the results show that tuning with Chameleon is both feasible and beneficial, questions remain about its expense. From the above discussion, we recommend that \( tune \) should always reasonably budget the fitness evaluations or nodes. Thus, when the trees grow larger, the budget also exhausts faster during each generation; this keeps the total number of generations low. If, instead, we fix the total number of generations as is common with GP, then because \( tune \) potentially favours large trees (as they have more tuning sites), the computational cost can be high. However, \( ptune \) is robust, as it successfully contains tree growth despite going through the same number of generations as GP. Another question then is: can we ascertain the maximum number of tuning events in a population, given the average tree size? Clearly, with maximal tuning (\( tune \)), we cannot do that since this number depends upon the evolutionary dynamics of every individual run: the maximum number of tuning events in a population is equal to the number of internal nodes in the largest tree in that population. However, we show in Section 5 that with probabilistic tuning setups, we can give a more reliable bound on the maximum number of tuning events. This analysis is also useful as a template for analysing another function that replaces Equation (1).

### 5 How Expensive Can Probabilistic Tuning Get?

The previous section raised the question of how expensive tuning can get. Here we answer this by identifying the tree size that gets the maximum number of tuning events (i.e., when a node is selected for tuning) and then, for that size, calculate that number. Note, by counting the tuning events instead of the iterations (through the functions set) while tuning a node, we abstract out of the exact implementation details. For example, instead of exhaustively iterating over the functions set at a selected node, we can probabilistically iterate over a subset and still the analysis in this section holds.

For this analysis, we consider the average size of the population at a given time, the possible shapes that the trees can take, and Equation (1), which calculates the probability that each node in a tree of size \( s \) will be tuned. To aid readability, we repeat Equation (1) with \( c = 1.5 \) substituted; for the rest of the section, we assume this substitution whenever we refer to Equation (1):

\[
p(s) = \begin{cases} 
  y = 1.5 - \frac{s}{\bar{s}} & \text{if } 0 \leq y \leq 1 \\
  1 & \text{if } y > 1 \\
  0 & \text{otherwise},
\end{cases}
\]

where \( p(s) \) (shown in Figure 2) is the per node probability of tuning a tree of size \( s \), and \( \bar{s} \) is the average tree size of the population; \( 0 \leq y \leq 1 \) when \( 1.5\bar{s} \geq s \geq 0.5\bar{s} \).

Let \( T(s) \) be the number of nodes tuned (or the number of tuning events) in a tree of size \( s \), and \( s_{\text{max}} \) be the tree size with the highest number of nodes tuned in the population.

---

3Some exploratory runs showed that maximal tuning can indeed be very expensive if we force the evolution through a fixed number of generations instead of terminating the runs after a fixed number of nodes or fitness evaluations. We consistently recorded individuals with over 20,000 nodes, which is an order of magnitude larger than the largest tree size reported in this study.
under consideration. Thus, \( T(s_{\text{max}}) \) represents the maximum tuning expense in a given population. With the subsequent analysis, we want to characterise \( T(s_{\text{max}}) \) as a function of the average tree size.

Despite not tuning leaves in this study, we analyse this case first as it is easier to do so. Next, in Section 5.2, we analyse the present case where leaves are not tuned.

### 5.1 If Leaves are Also Tuned

If the leaves are also tuned then the number of tuning events are: \( T(s) = s \times p(s) \); note that \( \times \) represents ordinary multiplication.

\[
\Rightarrow T(s) = \begin{cases} 
1.5s - \frac{s^2}{\bar{s}} & \text{if } 0 \leq y \leq 1 \\
 s & \text{if } y > 1 \\
0 & \text{otherwise}
\end{cases} 
\]

(2)

Figure 7 exemplifies Equation (2) when \( \bar{s} = 10 \); the maximum of Equation (2) is \( s_{\text{max}} \). To quantify \( s_{\text{max}} \), we find the maxima for both the linear and the nonlinear (quadratic) regions in Figure 7 and then compare \( T(s) \) on these two maxima. When \( T(s) \) is linear, \( y > 1 \) and \( s \leq 0.5\bar{s} \). Thus, the maximum of the linear region is at its endpoint, that is, at \( 0.5\bar{s} \). The maximum of the nonlinear region \( (0 \leq y \leq 1) \) is where \( T' = 0 \). Thus,

\[
s_{\text{max}} = 0.75\bar{s} 
\]

(3)

because the maximum number of tuning events for the nonlinear region, that is, \( T(0.75\bar{s}) = 0.5625\bar{s} \), is greater than that for the linear region, that is, \( T(0.5\bar{s}) = 0.5\bar{s} \). Therefore, even if we also tune the leaves, the maximum number of tuning events is only a fraction of the average size of the population, that is, \( 0.5625\bar{s} \). This is useful in two ways. First, a smaller than average sized tree gets the most out of tuning; this can promote smaller than average sized trees. Second, the maximum tuning expense is a function of the average size of the population instead of the maximum size as in maximal tuning. Clearly, the average size increases slower than the maximum size in a population and can even be regulated (Poli and McPhee, 2008). In the following section, we show that these benefits also hold for when the leaves are not tuned.
5.2 If Leaves Are Not Tuned

Typically, we only tune the internal nodes with Chameleon. Let \( i(s) \) be the number of internal nodes in \( t \), so that:

\[
T(s) = i(s) \times p(s).
\]

(4)

The crucial question then is: \( i(s) = f(s) \)? In other words, can we compute the number of internal nodes from just the tree size? To answer this, first, we consider full binary trees\(^4\) such that each node has exactly zero or two children. Then, in Section 5.2.2 we also consider the general case where the arity of the nodes varies. We show that the results for the general case are bounded by those for full binary trees.

For full binary trees, clearly, we have two cases: (1) when the tree is perfect\(^5\) (that is, all leaf nodes are at the same depth) or can be restructured into a perfect binary tree, and (2) otherwise. Next, we show that we can tackle both these cases together.

5.2.1 Full Binary Trees

The analysis in this section applies to both cases (1) and (2) because we show that the same relationship holds for \( i(s) \); however, we begin with case (1). We define a perfect convertible tree to be such that can be restructured into a perfect tree without changing the number of internal or external nodes. During this restructuring the semantics of the tree may change; however, that does not concern this analysis because for a full binary tree the number of internal nodes \( i(s) \) is only a function of the tree size \( s \). Thus, \( i(s) \) is the same for both the perfect and the perfect convertible trees. To compute \( i(s) \) for these two types of trees, we consider that 

\[
s = 2d_{\text{max}} + 1 \Rightarrow i(s) = 2^{d_{\text{max}} - 1} - 1.
\]

(5)

Appendix B shows that Equation (5) also holds for those full binary trees which cannot be restructured into a perfect tree; thus, the following analysis also applies to such trees.

From Equations (4) and (1) we get

\[
T(s) = \frac{1}{2} \begin{cases} 
1.5(s - 1) + \frac{s - 1}{\bar{s}} & \text{if } 0 \leq y \leq 1 \\
S - 1 & \text{if } y > 1 \\
0 & \text{otherwise.}
\end{cases}
\]

(6)

Again, the maximum of the linear part is \( T(0.5\bar{s}) = (0.5\bar{s} - 1)/2 \). To find the maximum for the nonlinear part, we set \( T'(s) = 0 \) and solve for \( s \) so that

\[
s_{\text{max}} = 0.5 + 0.75\bar{s}
\]

(7)

and correspondingly, \( T(s_{\text{max}}) = (0.5625\bar{s}^2 - 0.75\bar{s} + 0.25)/(2\bar{s}) \).

We can confirm that \( s_{\text{max}} \) indeed corresponds to the maximum by verifying whether \( T(s_{\text{max}}) - T(0.5\bar{s}) > 0 \).

\[
T(s_{\text{max}}) - T(0.5\bar{s}) = (0.0625\bar{s}^2 + 0.25\bar{s} + 0.25)/(2\bar{s})
\]

> 0 since \( \bar{s} > 0 \).

---

\(^4\)http://www.itl.nist.gov/div897/sqg/dads/HTML/fullBinaryTree.html

\(^5\)http://www.itl.nist.gov/div897/sqg/dads/HTML/perfectBinaryTree.html
Thus, as in Section 5.1, $s_{\text{max}}$ is a fraction of the average tree size. Also, the maximum tuning expense is a function of the average tree size. Therefore, again the smaller trees get the most out of the tuning; this also keeps the tuning expense manageable. Furthermore, Figure 8 plots $T(s_{\text{max}})$ and its rate of change with average tree size ($\bar{s}$) both when the leaves are tuned and when they are not. The figure shows that when leaves are not tuned, the increase in the tuning expense with an increasing average size ($\frac{dT(s_{\text{max}})}{ds}$) is asymptotic: $T(s_{\text{max}})$ grows very slowly with $\bar{s}$. When the leaves are also tuned, the growth rate is constant. This is unlike maximal tuning which can grow prohibitively expensive with increasing size.

5.2.2 General Case: Variable Arity
When the nodes have variable arity, we cannot exactly formulate $i(s)$ as in Equation (5); however, we know that at most $i(s) = (s - 1)$ (when arity = 1). Thus, we have an upper limit on $i(s)$. Substituting $i(s) = (s - 1)$ in Equation (4), again we get $s_{\text{max}}$ as in Equation (7). Therefore, the results for the binary trees also provide an upper limit for the general case when the arity varies.

In summary, we quantify the tuning expense with probabilistic tuning given Equation (1) and show that the expense increases conservatively as a function of the average size of the population. The average size of the population not only grows slower than the maximum size but is also more manageable.

6 Conclusions and Future Work
In this paper we describe a simple approach to lifetime learning in GP and apply it to some difficult, well-known problems from the symbolic regression domain. We show that it provides better performance on unseen test data and produces smaller individuals than the current state of the art of GP. The results also show that our hill climbing approach improves not only over standard GP but also over some known improvements to GP in the symbolic regression domain, that is, linear scaling and NSM tournament selection.

We dispel the fear of computational expense with our proposed method by taking a three pronged approach. First, we show that adding to the functions set does not
necessarily increase the number of tuning steps. Next, we show (and quantify) that, with the use of a simple cache, the computational costs associated with the proposed hill climbing approach substantially decreases and analytically show that the cost decreases by 50% for skinny trees that use only unary functions, and (asymptotically) up to 100% for maximally grown binary trees. Finally, we propose to tune probabilistically by preferring smaller than average sized individuals. Thus, the number of tuning opportunities for an individual is a function of its size. The results show that by preferring smaller trees like that, Chameleon generates trees smaller than those in standard GP; keeping trees small further reduces the computational cost.

While exhaustively using Chameleon also produces competitive results, such a use potentially promotes larger trees in the population. Also, an exhaustive approach cannot guarantee an upper limit on the tuning expense. Probabilistic Chameleon, however, contains tree growth and can guarantee an upper limit on the tuning expense as the average tree size in the population grows.

In future, we want to explore the effect of varying GP parameters on the performance of Chameleon. In particular, we aim to ascertain the effects of different probability models in probabilistic Chameleon, because although the selected model is an intuitive choice, it is by no means the optimal choice.

References


### Appendix A

**Savings with a Cache when Arity of the Functions Set Varies**

We define $F$, $C_{\text{max}}$, $F_j$, $N_1$, $N_2$, $i$, $s$, and $d_k$ as in Section 3. Let $Q(k) = |F_j|$ if the $k$th node contains a function with arity $j$ and $\exists j$ such that $|F_j| \neq C_{\text{max}}$. Then,

$$N_1 = s \sum_{k=1}^{i} (Q(k) - 1) = s(Q(k) - 1)i$$

where $Q(k) - 1 = Q(k) - 1$ represents the average of $(Q(k) - 1)$ over $i$ internal nodes. Similarly,

$$N_2 = (Q(k) - 1) \sum_{k=1}^{i} (d_k + 1) = (Q(k) - 1)(d + 1)i = \frac{d + 1}{s} N_1.$$
Therefore, even when arity of the functions in the functions set varies, $N_2$ is still only a fraction of $N_1$.

**Appendix B**

*Non Perfect Convertible Full Binary Trees*

Consider a tree which is perfect only until a depth level $d_f$ and only contains a few leaves at the level $d_f + 1$, for example, $\,(++(xy)1)\,$. This is an example of a tree which cannot be converted into a perfect binary tree. Such a tree is also called a complete binary tree. The trees which cannot be converted into a perfect tree can be converted into a complete binary tree such that every level, except the deepest, is completely filled.

Let $d_f$ be the deepest level at which the tree is completely filled and $i_f$ be number of internal nodes for $0 \leq d_k < d_f$ so that:

$$i_f = 2^{(d_f-1)+1} - 1 = 2^{d_f} - 1.$$  

Let $s_f$ be the size of the tree for $0 \leq d_k \leq d_f$ (i.e., the fraction of the tree that is completely filled) and $s$ be the total size of the tree. Thus, the number of nodes at depth $d_f + 1$ is:

$$s - s_f = s - 2^{d_f + 1} + 1$$

and (because we consider only full binary trees) the number of internal nodes at depth $d_f$ is $(s - s_f)/2$. Thus,

$$i(s) = i_f + \frac{s - s_f}{2}$$

$$= 2^{d_f} - 1 + \frac{s - 2^{d_f + 1} + 1}{2}$$

$$= \frac{s - 1}{2}$$

which is the same result as in Equation (5). Hence, the properties of $T(s)$ remain the same as with perfect or perfect-convertible trees.

**Appendix C**

*Test Set Results for Complete Runs*

This section presents the mean of test set performance of the best individual (best on training score) throughout the evolutionary run. Note that the best individual may vary during the run; the result at the end of the run corresponds to the best individual that appeared during the entire run.

As highlighted in Section 4.3, we investigate the test set results as to whether Chameleon over-trains the evolving individuals, that is, whether their test set performance degrades over time in a manner even worse than that with standard GP. The end of the run results in Section 4.3 dismissed such a notion. However, the end of the run samples the overall trend only at a single point in time. Since, unlike training score and tree sizes that, typically, either only increase or stay the same over time, test set results may fluctuate, a question arises as to whether the end of run was just a lucky point in time for Chameleon. To address this question, we present the test set results from the entire runs in Figures 9 and 10.

---

Figure 9: Normalised mean test score for the best fit individual is plotted at various stages of evolution. Runs end after processing $1.2 \times 10^6$ GP tree nodes.
Figure 10: Normalised mean test score for the best fit individual is plotted at various stages of evolution. Runs end after processing 75,000 fitness evaluations.
Figure 9 shows the results for when the runs terminated after traversing $1.6 \times 10^6$ nodes; Figure 10 shows the same results for when the runs terminated after exhausting 75,000 fitness evaluations. The trends in the figures show that:

- Except for bioavailability in Figure 9, Chameleon setups are never consistently inferior to their standard GP counterparts throughout the runs; and

- On some problems, when training begins, standard GP setups are indeed better than Chameleon, for example, as in Boston Housing, Concrete Strength, and Dow Chemical in Figure 10. However, while performance of standard GP setups degrades later, the performance of Chameleon setups improves.

Hence, except in one case, for the problems studied in this paper, Chameleon tuning does not harm the test set performance as might be expected.