A Hyper-heuristic Approach to Automated Generation of Mutation Operators for Evolutionary Programming

Libin Hong[a], John H. Drake[b], John R. Woodward[c], Ender Özcan[a]

[a]*School of Computer Science, University of Nottingham*
[b]*Operational Research Group, School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS, UK*
[c]*Computing Science and Mathematics, University of Stirling*

Abstract

Evolutionary programming can solve black-box function optimisation problems by evolving a population of numerical vectors. The variation component in the evolutionary process is supplied by a mutation operator, which is typically a Gaussian, Cauchy, or Lévy probability distribution. In this paper, we use genetic programming to automatically generate mutation operators for an evolutionary programming system, testing the proposed approach over a set of function classes, which represent a source of functions. The empirical results over a set of benchmark function classes illustrate that genetic programming can evolve mutation operators which generalise well from the training set to the test set on each function class. The proposed method is able to outperform existing human designed mutation operators with statistical significance in most cases, with competitive results observed for the rest.

Keywords: Evolutionary Programming; Genetic Programming; Automatic Design; Hyper-heuristics; Continuous Optimization

1. Introduction

Black-box function optimisation is the task of finding the optima of an objective function for which we do not have access to an analytical form. A generate-and-test approach can be used to sample the domain of the function in order to identify potential optima. Different black-box optimisation techniques have been proposed; the majority of which are the result of much manual effort. In addition, each optimisation algorithm is designed in isolation from a problem environment. A proposed algorithm is usually tested on a number of benchmark functions to demonstrate its ability to identify the optima of a function. Unlike an individual function, a function class represents a parameterised function in which the parameters have a certain range of values. Thus a function class represents an infinite source of functions, from which it is possible to draw sets of sample functions from the same distribution.

This paper is concerned with evolutionary programming (EP) [1], which evolves a population of real-valued input vectors for a function, a technique widely applied to real-world problems [2, 3, 4]. As EP has an evolutionary basis, each vector undergoes selection, evaluation, and mutation, with the expectation that fitter and fitter vectors are obtained. Here we focus on the mutation component of EP, which in the past has been designed manually. Real-valued optimisation is an active research topic and several population-based metaheuristics have been applied to function optimisation, including differential evolution and its variants [5, 6] , particle swarm optimisation [7], covariance matrix adaptation evolution strategy (CMA-ES) [8] and hybrid methods [9]. We acknowledge the existence of these algorithms. However, as this study is concerned specifically with the modification of EP, a full review of all of these algorithms is beyond the scope of this paper.

A hyper-heuristic is a search method or learning mechanism for selecting or generating heuristics to solve computational search problems [10]. In addition to the broad distinction between selection and generation hyper-heuristics, it is also possible to classify hyper-heuristics according to the source of feedback during learning. *Online* hyper-heuristics learn while solving a given instance of a problem, whilst hyper-heuristics which learn in an *offline* manner gather knowledge in the form of rules or programs, from a set of training instances, that would hopefully generalise to the process of solving unseen instances. genetic programming (GP) [11] is a population-based evolutionary computation method for evolving program trees, that has frequently been used as a generation hyper-heuristic in the literature [12].

In this paper, we use GP as an offline generation hyper-heuristic to automatically create mutation operators for EP operating on function classes. A mutation operator in EP is a probability distribution, represented as a random number generator. We present an algorithmic framework which can not only express a number of currently existing EP mutation operators, but also generate novel variants of EP mutation operators. Using a train-and-test approach, GP is used to evolve mutation operators for EP, using a training set drawn from a class of functions which is then validated on a larger set of unseen instances taken from the same class. We use the term *automatically designed mutation operators* (ADMs) to describe the mutation operators generated by GP. We demonstrate that the ADMs for EP are capable of comparable, and often superior, performance to existing human designed operators. An additional set of experiments that takes ADMs that are trained on one function class, but then tested on a different function class is also conducted to further examine the performance of the evolved ADMs.

The outline of the remainder of this paper is as follows. In Section 2, we give the background to the proposed approach of automatically designing algorithms using GP-based hyper-heuristics. In Section 3 we consider the task of function optimisation and introduce the notion of a function class. Section 4 presents our experimental results, which are analysed in Section 5. In Section 6 we discuss the research presented and in Section 7 we summarise the article and outline potential further research directions.

2. Automated Design Using Hyper-Heuristics

The key distinction between metaheuristics and hyper-heuristics is that the former operate directly on the solution search space, while the latter operate indirectly on the solution search space, working with a set of low-level heuristics or heuristic components. Hyper-heuristics come in two main types: *heuristics to choose heuristics* and *heuristics to generate heuristics* [10]. In this paper we are concerned with the second of these two categories, heuristics to generate heuristics [12].

The automated generation of heuristics has received much attention in the last few years. Woodward et al. [13] automatically search the space of genetic algorithm (GA) selection operators, which contain *fitness proportional* and *rank* selection, where bitstrings in the population are chosen in proportion to their fitness value or indexed position in the sorted population respectively. In a later paper by the same authors, novel mutation operators were automatically constructed using random search and multiple-restart hill-climbing to search the space of mutation operators [14]. The system was capable of expressing two well-known mutation operators: *one-point* and *uniform.* While random search and hill-climbing may not be considered to be particularly ‘sophisticated’, they were sufficient to discover new selection and mutation operators which outperformed their human designed counterparts. Diosan and Oltean [15] evolved crossover operators for genetic algorithms outperforming existing crossover operators on some function optimisation problems.

As one of the main applications areas of metaheuristics is combinatorial optimisation problems, it is not surprising that this type of problem has attracted the attention of automated design. GP has been widely adopted to generate heuristics for a variety of problems. Nguyen et al. [16]

used GP to automatically design algorithms for job-shop scheduling, Keller et al. [17] tackled the travelling salesman problem, while Bader El Den et al. [18, 19] evolved timetabling heuristics. Drake et al. [20] used GP to evolve a scoring mechanism to determine the order in which knapsack items should be considered by a constructive heuristic for the multidimensional knapsack problem. Hong et al. [21] used GP to automate the design of probability distributions as mutation operators for evolutionary programming. GP was used to generate new data mining algorithms which were tested on well-known machine learning benchmark datasets by Freitas and Pappa [22]. Their paper showed that GP could outperform random search in searching the space of rules for data mining. The rules evolved by GP were observed to be at least as good as human designed rules in terms of classification. GP has also been used to automatically design schedule policies for dynamic multi-objective job shop scheduling [23], to evolve ensembles of dispatching rules for the job shop scheduling problem [24], and to automate the design of production scheduling heuristics [25]. Both online and offline bin packing have attracted attention within the heuristic generation research community [26]. Heuristic functions have been evolved to determine in which bin to place a given item [27]. In this case, the evolved heuristic functions have been shown to perform well on problem instances drawn from the same problem class used in the training phase, while a degradation in performance is witnessed when heuristics are applied to problem instances drawn from different problem classes. Heuristics have been evolved on problem instances containing a small number of items, then applied to much larger problem instances containing many more items [28]. In these approaches, a heuristic function is evolved as a GP syntax tree. However, more recently, both a look-up table (referred to as a "matrix") [29, 30] and function interpolation [31] have also been used to represent a heuristic function. Other metaheuristics have been automatically designed using hyper-heuristics, such as particle swarm optimisation [32] and variable neighbourhood search [33].

This current paper builds on previous papers. Hong et al. [21] first demonstrated that GP could automatically construct random number generators which are typically used in EP. In a second paper, it was shown that ADMs could be trained on collections of functions classes, showing good performance across a broader range of functions [34], however a tradeoff between general training and specific performance was observed. This paper presents a study of the design of 23 ADMs, for 23 functions classes, and then tests each of the 23 ADMs on each of the function classes.

3. Optimisation and Function Classes

In this section we first discuss optimisation with EP, and then introduce function classes as probability distributions over functions. Function classes are central to this paper, differentiating our approach from the standard convention of benchmarking on arbitrary functions. Rather than demonstrating the utility of an optimisation algorithm for specific arbitrary functions, we demonstrate the utility of an ADM on a set of functions are drawn from a fixed probability distribution (i.e. a function class).

3.1. Evolutionary Programming and Optimisation

We follow the formulation of optimisation as stated by Yao et al. [1, 35], which we repeat here. Global minimisation can be formalised as a pair (X, f), where $X \in \mathbb{R}^n$ is a bounded set in $\mathbb{R}^n$ and $f : X \rightarrow \mathbb{R}$ is an n-dimensional real-valued function. The objective is to find a point $x_{min} \in X$ such that $f(x_{min})$ is a global minimum in X. More specifically, it is required to find an x_{min} such that

$$\forall x \in X : f(x_{min}) \leq f(x).$$

Here, f does not need to be continuous or differentiable. While the aim of optimisation is to identify global optima of the function, in practice we often settle for methods, such as EP, which identify near-optima. EP is a widely used evolutionary algorithm for continuous optimisation introduced by Bäck and Schwefel [36] as follows:

1. *Generate the initial population of μ individuals. Each individual is taken as a pair of real-valued vectors, (x_i, η_i), $\forall i \in \{1, \dots, \mu\}$, where x_i's are objective variables and η_i's are standard deviations for Gaussian mutations.*

2. *Evaluate the fitness value for each (x_i, η_i), $\forall i \in \{1, \dots, \mu\}$, of the population based on the objective function, $f(x_i)$.*

3. *Each parent (x_i, η_i), $\forall i \in \{1, \dots, \mu\}$, creates λ / μ offspring on average, so that a total of λ offspring are generated. Offspring are generated by: for $i \in \{1, \dots, \mu\}$, $j \in \{1, \dots, n\}$ and $p \in \{1, \dots, \lambda\}$,*

$$\eta_i'(j) = \eta_i(j) exp(\gamma' N(0,1) + \gamma N_j(0,1)) \tag{1}$$

$$x_p'(j) = x_i(j) + \eta_p'(j) D_j \tag{2}$$

where $x_i(j)$, $x_p'(j)$, $\eta_i(j)$ and $\eta_p'(j)$ denote the j-th component of the vectors $\boldsymbol{x}_i$, $\boldsymbol{x}_p'$, η_i and η_p' respectively. $N(0,1)$ denotes a normally distributed one-dimensional random number with mean 0 *and standard deviation 1. $N_j(0,1)$ indicates that the random number is generated anew for each value of j. The factors γ and γ' are set to $(\sqrt{2\sqrt{n}})^{-1}$ and $(\sqrt{2n})^{-1}$ [36].*

4. *Calculate the fitness of each offspring (x'_p, η'_p), $\forall p \in \{1, \ldots, \lambda\}$, according to $f(x'_p)$.*

5. *Conduct pairwise comparison over the union of parents (x_i, η_i) and offspring (x'_p, η'_p), $\forall i \in \{1, \ldots, \mu\}$, $\forall p \in \{1, \ldots, \lambda\}$. For each individual, q opponents are selected randomly from the parents and offspring. For each comparison, if the individuals' fitness is no smaller than the opponent's it receives a 'win'.*

6. *Select the μ individuals out of the parents and offspring ((x_i, η_i) and (x'_p, η'_p), $\forall i \in \{1, \ldots, \mu\}$, $\forall p \in \{1, \ldots, \lambda\}$) that have the most wins to be the parents of the next generation.*

7. *Stop if the halting criterion is satisfied; otherwise return to Step 3.*

Different variants of EP can be obtained by using different probability distributions D_j in Step 3 above. If Gaussian distribution is used, then the algorithm is classical evolutionary programming (CEP) [1], the Cauchy distribution is used in Fast EP (FEP) [1] whereas the Lévy distribution is used in Lévy evolutionary programming (LEP) [37]. The Lévy distribution is parameterised with a single parameter α, and corresponds to the Cauchy distribution when α=1.0 and the Gaussian distribution when α=2.0. Where LEP is used in this paper, the Lévy $L_{\alpha,\gamma}(y)$ distribution is implemented according to Mantegna [38] as given by Lee and Yao [37], with γ fixed at 1 (note that γ here is independent from step 3 above):

$$L_{\alpha,\gamma}(y) = \frac{1}{\pi} \int_0^\infty e^{-\gamma q^\alpha} \cos(qy) dq, y \in \mathbb{R}. \tag{3}$$

In this paper we will use genetic programming to evolve distributions to replace D_j as a mutation operator in the EP system described above. In each case, we use the same parameters for EP as previous publications [1, 35], using tournament selection with tournament size 10 on a population of 100 individuals. The initial value of the strategy parameter η is set to 3.0.The number of generations is different for each function and is specified in Table 1 below.

3.2. Functions and Function Classes

In previous papers [35, 37, 39, 40], methods have been compared on functions from an arbitrary set. Each method is executed multiple times on a *single* function to provide a statistical comparison. Our method deviates from this approach in that we evolve a mutation operator for a *function class*, representing a probability distribution over a set functions. An example of a function class is $y = ax^2$ where a is uniformly distributed in the range [1, 2]. In this case, $y = 1.3x^2$ is a function from this function class, while $y = 0.3x^2$ is not from this function class. For each of the 23 standard functions often used in EP research [1, 35], we have constructed a corresponding function class. The 23 benchmark function classes are given in Table 1. These can be classified as follows: f_1–f_7 are unimodal functions, f_8–f_{13} are multimodal functions with many local optima, and f_{14}–f_{23} are multimodal function with few local optima [35].

Table 1: Function classes with number of dimensions n, domain S, and number of generations Gen. The parameters a, b and c are uniformly distributed in [1 ,2], [-1, 1] and [-1, 1], respectively. For the values of w in f_{19} to f_{23}, please see [35]

Function Class	n	S	Generations
$f_1(x) = \sum_{i=1}^{n}[(ax_i - b)^2 + c]$	30	$[-100, 100]^n$	1500
$f_2(x) = a\sum_{i=1}^{n} \mid x_i \mid + b\prod_{i=1}^{n} \mid x_i \mid$	30	$[-10, 10]^n$	2000
$f_3(x) = \sum_{i=1}^{n}[\sum_{j=1}^{i}(ax_j + b)]^2$	30	$[-100, 100]^n$	5000
$f_4(x) = \max_i\{a \mid x_i \mid, 1 \le i \le n\}$	30	$[-100, 100]^n$	5000
$f_5(x) = \sum_{i=1}^{n}[a(x_{i+1} - x_i^2)^2 + b(x_i - 1)^2 + c]$	30	$[-30, 30]^n$	1500
$f_6(x) = \sum_{i=1}^{n}(\lfloor ax_i + 0.5\rfloor)^2$	30	$[-100, 100]^n$	1500
$f_7(x) = a\sum_{i=1}^{n} ix_i^4 + random[0, 1)$	30	$[-1.28, 1.28]^n$	3000
$f_8(x) = \sum_{i=1}^{n} -(x_i \sin(\sqrt{\lvert x_i\rvert}) + a)$	30	$[-500, 500]^n$	1500
$f_9(x) = \sum_{i=1}^{n}[ax_i^2 + b(1 - cos(2\pi x_i))]$	30	$[-5.12, 5.12]^n$	5000
$f_{10}(x) = -a\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - b\exp(\frac{1}{n}\sum_{i=1}^{n}\cos 2\pi x_i) + e + c$	30	$[-32, 32]^n$	1500
$f_{11}(x) = \frac{a}{4000}\sum_{i=1}^{n} x_i^2 - b\prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) + c$	30	$[-600, 600]^n$	1500
$f_{12}(x) = \frac{a\pi}{n}\{10sin^2(\pi y_i) + \sum_{i=1}^{n-1}(y_i - 1)^2[1 + 10sin^2(\pi y_{i+1})$ $+ (y_n - 1)^2]\} + \sum_{i=1}^{n} u(x_i, 10, 100, 4),$ $y_i = 1 + \frac{1}{4}(x_i + 1)$ $u(x_i, w, k, m) = \begin{cases} k(x_i - w)^m, & x_i > w, \\ 0, & -w \le x_i \le w, \\ k(-x_i - w)^m, & x_i < -w. \end{cases}$	30	$[-50, 50]^n$	1500
$f_{13}(x) = 0.1a\{sin^2(3\pi x_1) + \sum_{i=1}^{n-1}(x_i - 1)^2[1 + sin^2(3\pi x_{i+1})] +$ $(x_n - 1)[1 + sin^2(2\pi x_n)]\} + \sum_{i=1}^{n} u(x_i, 5, 100, 4)$	30	$[-50, 50]^n$	1500
$f_{14}(x) = [\frac{1}{500} + a\sum_{i=1}^{25}\frac{1}{j+\sum_{i=1}^{2}(x_i - w_{ij})^6}]^{-1}$	2	$[-65.536, 65.536]^n$	100
$f_{15}(x) = \sum_{i=1}^{11}[w_i - \frac{ax_1(y_i^2+y_ix_2)}{b(y_i^2+y_ix_3+x_4)}]^2$	4	$[-5, 5]^n$	4000
$f_{16}(x) = a(4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4) + b$	2	$[-5, 5]^n$	100
$f_{17}(x) = a(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6)^2 + 10b(1 - \frac{1}{8\pi})cosx_1 + 10$	2	$[-5, 10] \times [0, 15]$	100
$f_{18}(x) = a[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2$ $+ 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2(18 - 32x_1$ $+ 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)] + b$	2	$[-2, 2]^n$	100
$f_{19}(x) = -\sum_{i=1}^{4} y_i exp[-\sum_{j=1}^{4} aw_{ij}(x_j - p_{ij})^2 + b]$	3	$[0, 1]^n$	100
$f_{20}(x) = -\sum_{i=1}^{4} y_i exp[-\sum_{j=1}^{6} aw_{ij}(x_j - p_{ij})^2 + b]$	6	$[0, 1]^n$	200
$f_{21}(x) = -\sum_{i=1}^{5}[(x - w_i)^T(x - w_i) + y_i]^{-1} + a$	4	$[0, 10]^n$	100
$f_{22}(x) = -\sum_{i=1}^{7}[a(x - w_i)^T(x - w_i) + y_i + b]^{-1}$	4	$[0, 10]^n$	100
$f_{23}(x) = -\sum_{i=1}^{10}[a(x - w_i)^T(x - w_i) + y_i + b]^{-1}$ $where$ $y_i = 0.1$	4	$[0, 10]^n$	100

4. Experimental Design

In this section we describe the experimental set-up of GP and EP. With hyper-heuristic approaches, it is important to identify the two levels at which the heuristics operate. In a typical

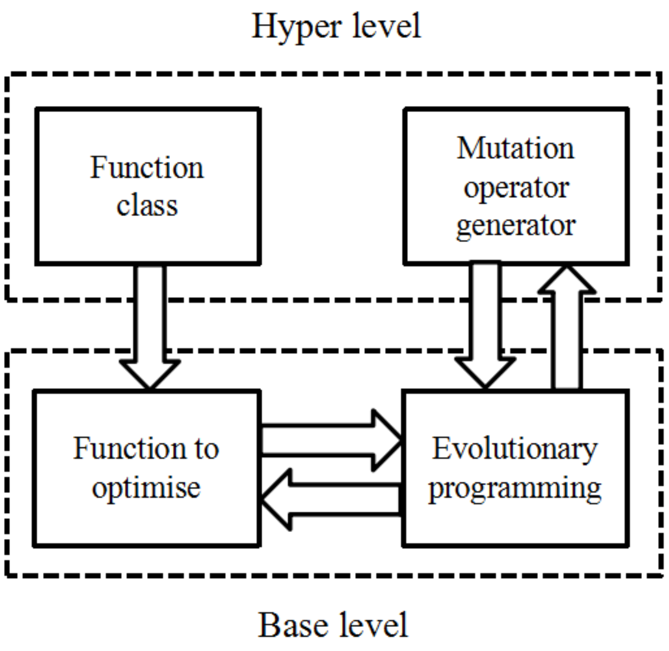

Figure 1: Overview of the hyper-heuristic framework used

hyper-heuristic, a metaheuristic operates on a space of (meta) heuristics, which operate directly on the space of solutions. Here we use GP as a mutation operator generator at the hyper-level to manipulate the mutation operators within a population of EP algorithms working at the base level. The overall framework used is given in Figure 1.

Previous approaches manually build optimisation methods and test them on benchmark functions [1, 35]. Here we employ a train-and-test approach, in which a first set of functions is used to train a method before it is tested on a second independent set of functions drawn from the same probability distribution to analyse performance.

4.1. The Training Phase

We call a program generated by GP an 'automatically designed mutation operator' (ADM), which is in effect a random number generator. Each ADM is used as an EP mutation operator on 5 training functions drawn from a given function class. The fitness of an ADM is the average of the best values obtained in each of the individual 5 EP runs on a given function class. We use the same 5 functions from each function class for the entire run of GP on a given function class. For each function class, 10 functions are taken for training, 5 of which are used to calculate the fitness, and 5 of which are used to monitor overfitting.

4.2. The Testing Phase

When the training phase of EP is complete for a given function class, the output is an ADM intended *solely* for the function class on which it was trained. We then draw 50 new functions from the given function class and test the ADM, comparing against 6 existing EP variants (CEP, FEP and EP with 4 settings for the α parameter of the Lévy distribution). Note that training is expensive, as many ADMs are evaluated, so a small set of functions is used when training. In contrast, testing involves only a single ADM so is less costly, and a larger number of testing samples allows a better comparison through statistical tests.

4.3. Parameter Settings for Genetic Programming

The GP implementation used in this paper is the genetic programming toolbox for Matlab (GPLAB) [41]. The parameters for GP are given in Table 2. We use subtree crossover, in which a node in each of the parent programs is chosen uniformly at random, and the respective subtrees are swapped, creating two new offspring. One point mutation is also used, where a node is chosen in the parent tree and substituted for a new tree created with the Grow initialisation method [11], obeying the size and depth restrictions imposed by the GP parameters. The fitness of each GP individual is calculated as the average best fitness values of the EP runs on each of the 5 training functions, as described in Section 4.1 above. At each generation, the best individual from both parents and offspring, along with the best offspring created during that generation are retained in the population. The selection method used is 'lexictour', which uses lexicographic parsimony pressure when two individuals are compared [42], if two individuals are of equal fitness, the tree with a smaller number of nodes is chosen. As the evolutionary process is incredibly expensive in computational terms (each GP individual has to perform 5 EP runs), the maximum number of generations is initially set at a low value to reduce the amount of computational effort spent. The maximum number of generations is set dynamically to try to ensure that a sufficient number of evaluations are made to achieve convergence, whilst minimising the time spent on evaluating poor quality runs. If the best individual in the population is found within the final three generations of a run, the maximum number of generations is increased by 5 in order to allow the evolutionary process to continue. The maximum number of generations is capped at 100, with the evolutionary process terminated regardless of when the best individual in the population was found.

Table 2: Parameter settings for GP

Parameter	Value
Population Size	10
Initial Number of Generations	5
Upper Bound of Number of Generations	100
Crossover Proportion	45%
Mutation Proportion	45%
Reproduction Proportion	10%
Selection Method	lexictour [42]
Tournament Size	2
Maximum Initial Size of Tree	28
Maximum Size of Tree	512

Table 3: Function set for GP

Symbol	Function	Arity
$+$	addition	2
$-$	subtraction	2
$\times$	multiplication	2
$\div$	protected division	2
pow	power	2
log	logarithm (base e)	1
sin	sin	1
cos	cos	1
sqr	square root	1

Table 4: Terminal set for GP

Symbol	Terminal
U	$\sim$[0, 1]
$N(\mu, \sigma^2)$	Normal Distribution
chy	Cauchy Distribution

The function and terminal sets for GP are given in Tables 3 and 4, respectively. U is the uniform distribution on [0, 1]. $N(\mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2. Here we clarify that this μ is independent to that used in the EP descriptions in Section 3 In our experiments, μ lies within the range [-2, 2] and σ^2 is in [0, 10]. chy is the Cauchy distribution. '$\div$' is protected divide: if the numerator is divided by a zero denominator, then the numerator is returned. The square root function is also the protected variant, taking the square root of the absolute value of a single argument to ensure that no negative input is used. As the Lévy distribution can be constructed from the normal distribution and arithmetic operators [38], it is also within the search space that GP is operating in. In the terminal set there are no input variables, here we are using GP to construct mutation operators which are effectively random number generators so do not require any input variable.

5. Analysis of the Performance of the Automatically Designed Mutation Operators

Table 5 reports the average best values obtained over 50 EP runs of each of the 23 function classes using a number of different mutation operators. The corresponding standard deviations are shown underneath each mean value in parentheses. These values are displayed for Cauchy (FEP, Lévy with $\alpha = 1.0$), Lévy with $\alpha = 1.2$, $\alpha = 1.4$, $\alpha = 1.6$ and $\alpha = 1.8$, and Gaussian (CEP, Lévy with $\alpha = 2.0$), as well as the best ADM evolved by GP for that function class. The best values (lowest, as we are minimising) are in boldface.

Table 5: ADMs compared to human designed mutation operators, means and (standard deviations), averaged over 50 runs. The best fitness values are in bold.

	Cauchy	$\alpha = 1.2$	$\alpha = 1.4$	$\alpha = 1.6$	$\alpha = 1.8$	*Gaussian*	ADM
f_1	6.012303	6.011758	6.011538	6.011426	6.011369	6.011267	**6.011234**
	(15.5123)	(15.5123)	(15.5123)	(15.5123)	(15.5123)	(15.5123)	(15.5123)
f_2	0.140	0.102	0.084	0.073	0.065	0.043	**0.017**
	(2.8E-02)	(2.0E-02)	(1.7E-02)	(1.5E-02)	(1.3E-02)	(8.6E-03)	(3.5E-03)
f_3	0.028	0.018	0.014	0.015	0.018	0.018	**0.008**
	(2.0E-02)	(2.2E-02)	(1.8E-02)	(3.4E-02)	(5.3E-02)	(2.6E-02)	(1.4E-02)
f_4	1.88	5.30	9.10	10.58	13.78	17.31	**0.13**
	(1.87)	(3.49)	(4.36)	(4.43)	(6.56)	(6.92)	(0.21)
f_5	-19.94	-19.87	-20.42	-20.21	-19.61	-20.22	**-20.63**
	(26.13)	(26.51)	(26.94)	(25.71)	(26.44)	(26.35)	(27.15)
f_6	0.0336	0.0134	0.0076	0.0724	0.9058	322.7146	**0.0074**
	(3.0E-02)	(1.3E-02)	(8.2E-03)	(2.4E-01)	(2.87)	(820.2)	(9.4E-03)
f_7	0.0586	0.0530	0.0506	0.0524	0.0554	0.0609	**0.0486**
	(9.3E-03)	(8.6E-03)	(7.9E-03)	(6.9E-03)	(9.9E-03)	(1.1E-02)	(6.3E-03)
f_8	-11058.28	-10642.83	-10009.65	-9530.88	-8818.26	-8053.96	**-12469.12**
	(397.2)	(507.5)	(483.6)	(586.8)	(645.2)	(603.5)	(109.0)
f_9	-10.953	-10.955	-10.956	-10.956	-10.357	-8.269	**-10.958**
	(15.82)	(15.82)	(15.82)	(15.82)	(16.21)	(16.29)	(15.82)
f_{10}	-27.8428	-27.8496	-27.8529	-27.8549	-27.8562	-27.7300	**-27.8634**
	(6.86)	(6.87)	(6.87)	(6.87)	(6.87)	(6.93)	(6.87)
f_{11}	-0.4963	-0.4839	-0.4704	-0.4534	-0.4554	-0.4465	**-0.5030**
	(6.4E-01)	(6.4E-01)	(6.5E-01)	(6.7E-01)	(6.4E-01)	(6.3E-01)	(6.4E-01)
f_{12}	1.72E-05	2.13E-02	1.82E-01	1.77E-01	1.01	2.37	**3.57E-06**
	(8.8E-06)	(5.9E-02)	(4.7E-01)	(3.7E-01)	(2.0)	(2.9)	(2.5E-06)
f_{13}	2.20E-04	5.20E-04	2.87E-01	5.67E-01	2.23	7.04	**6.46E-05**
	(5.5E-05)	(2.7E-03)	(1.5)	(1.6)	(6.9)	(13.0)	(2.3E-05)
f_{14}	1.32	0.98	1.24	1.08	1.11	0.90	**0.78**
	(1.1)	(6.9E-01)	(8.1E-01)	(7.1E-01)	(6.0E-01)	(5.1E-01)	(2.9E-01)
f_{15}	5.68E-04	5.13E-04	5.90E-04	6.03E-04	6.34E-04	5.39E-04	**4.66E-04**
	(3.6E-04)	(3.2E-04)	(3.8E-04)	(3.5E-04)	(3.9E-04)	(3.3E-04)	(3.1E-04)
f_{16}	-1.522775	-1.522775	-1.522776	-1.522776	-1.522776	-1.522777	**-1.522779**
	(0.6049959)	(0.6049966)	(0.6049960)	(0.6049964)	(0.6049962)	(0.6049962)	(0.6049964)
f_{17}	5.8792588	5.8792572	5.8792571	5.8792674	5.8792569	5.8792566	**5.8792563**
	(2.9140509)	(2.9140486)	(2.9140487)	(2.9140637)	(2.9140487)	(2.9140488)	(2.9140489)
f_{18}	4.615353	4.615353	4.615322	4.615324	4.615301	4.615228	**4.615192**
	(0.9234)	(0.9235)	(0.9234)	(0.9234)	(0.9234)	(0.9234)	(0.9234)
f_{19}	-3.353969	-3.354014	-3.354021	-3.354022	-3.354025	-3.354040	**-3.354058**
	(1.7371)	(1.7370)	(1.7370)	(1.7370)	(1.7370)	(1.7370)	(1.7371)
f_{20}	-3.67	-3.73	-3.83	-3.56	-3.76	-3.74	**-3.86**
	(2.15)	(2.30)	(2.24)	(2.13)	(2.25)	(2.18)	(2.25)
f_{21}	-4.33	-5.99	-5.59	-5.84	-6.44	-6.96	**-7.26**
	(2.0)	(2.7)	(2.7)	(2.8)	(2.5)	(2.4)	(2.2)
f_{22}	-11910.70	-23968.23	-20517.80	-20734.62	-12662.55	-68694.69	**-85515.41**
	(25106.9)	(70957.4)	(36414.1)	(35143.6)	(20568.8)	(193492.5)	(195260.3)
f_{23}	-20808.83	-19096.31	-14155.95	-18304.26	-20858.36	-27047.33	**-111864.57**
	(86762.9)	(39304.9)	(23208.8)	(46037.7)	(36873.8)	(40003.3)	(178141.4)

For all 23 of the function classes, the GP designed ADM outperform the 6 human designed mutation operators. In Table 5 the evolved ADMs show the best performance on both unimodal and multimodal functions generated by all listed function classes in Table 1. We retain 2, 3 or 4 digits after the decimal point for most of the results. To distinguish the difference of testing results on f_1, f_{16} and f_{18}, we retain 6 digits after the decimal point. We retain 7 digits after the decimal point for testing results on f_{17}.

To determine which of these performances differ with statistical significance, we perform a Wilcoxon signed-rank test, the results of which are presented in Table 6. Shown are the results of the Wilcoxon signed-rank test within a 95% confidence interval of an ADM compared with other mutation operators. In this table, '$\geq$' indicates that the ADM performs better than another mutation operator on average. In the cases where this difference is statistically significant, '$>$' is used. In the majority of the cases, the ADMs outperform human designed mutation operators including Gaussian, Cauchy and Lévy, and this performance difference is statistically significant.

Table 6: Wilcoxon Signed-Rank Test of ADMs versus Gaussian, Cauchy and Lévy (with α = 1.2, 1.4, 1.6, 1.8) on f_1–f_{23}.

Function Class	Cauchy	$\alpha = 1.2$	$\alpha = 1.4$	$\alpha = 1.6$	$\alpha = 1.8$	Gaussian
f_1	$>$	$>$	$>$	$>$	$>$	$>$
f_2	$>$	$>$	$>$	$>$	$>$	$>$
f_3	$>$	$>$	$>$	$\geq$	$\geq$	$>$
f_4	$>$	$>$	$>$	$>$	$>$	$>$
f_5	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$
f_6	$>$	$>$	$\geq$	$>$	$\geq$	$>$
f_7	$>$	$>$	$>$	$\geq$	$>$	$>$
f_8	$>$	$>$	$>$	$>$	$>$	$>$
f_9	$>$	$>$	$>$	$>$	$>$	$>$
f_{10}	$>$	$>$	$>$	$>$	$>$	$>$
f_{11}	$>$	$>$	$>$	$>$	$>$	$>$
f_{12}	$>$	$>$	$>$	$>$	$>$	$>$
f_{13}	$>$	$>$	$>$	$>$	$>$	$>$
f_{14}	$\geq$	$>$	$>$	$>$	$>$	$>$
f_{15}	$>$	$>$	$>$	$>$	$>$	$>$
f_{16}	$>$	$>$	$>$	$>$	$>$	$>$
f_{17}	$>$	$>$	$>$	$>$	$>$	$>$
f_{18}	$>$	$>$	$>$	$>$	$>$	$>$
f_{19}	$>$	$>$	$>$	$>$	$>$	$>$
f_{20}	$>$	$>$	$>$	$\geq$	$>$	$>$
f_{21}	$>$	$>$	$>$	$>$	$>$	$\geq$
f_{22}	$>$	$>$	$>$	$>$	$>$	$>$
f_{23}	$>$	$>$	$>$	$>$	$>$	$>$

There are only 2 (f_5, f_{14}), 1 (f_5), 2 (f_5, f_6), 4 (f_3, f_5, f_7, f_{20}), 3 (f_3, f_5, f_6) and 1 (f_5) function classes for which evolved ADMs perform slightly better than Cauchy, Levy$_{\alpha=1.2}$, Levy$_{\alpha=1.4}$, Levy$_{\alpha=1.6}$, Levy$_{\alpha=1.8}$ and Gaussian respectively, with no statistical significance. This could be for a number of reasons. It is possible that EP has been run for so many iterations that it does not matter which mutation operator is used, and any difference in performance is negligible. It also may be the case that GP would be able to find better mutation operators if it were allowed to run for a longer period of time. One reason that GP can consistently find ADMs which perform at least as well as human designed mutation operators is that they are easily expressed within the function and terminal set used.

The best ADMs obtained with GP for each function class are listed in Table 7. Figure 2 provides histograms for a subset of ADMs, showing 3000 samples taken from the ADM to give an indication of the underlying probability distribution they represent.

Table 7: Evolved ADMs for all 23 function classes. These have been algebraically simplified where possible.

ADMs	Automatically designed mutation operators
ADM1	$(\sin(+(chy\ chy)))$
ADM2	$(\sin(-(U\ U)))$
ADM3	$(\log(chy))$
ADM4	$(\div(-(chy\ chy)\ +(-(0\ N(-1.8493, 2.288))\ \cos(\log(N(-1.6403, 4.4607))))))$
ADM5	$(\times(\cos(+(\div(1\ \times(\div(-(U\ U)\ N(0.94108, 7.9111))\ N(-0.5776, 0.10706)))$ $N(-1.0504, 5.9002)))\ N(-0.89638, 7.9277)))$
ADM6	$(N(-0.11984, 5.631))$
ADM7	$(N(-0.058664, 3.2512))$
ADM8	$(+(\times(\times(chy\ \times(\times(chy\ chy)\ chy))\ \times(chy\ +(chy\ U)))\ +(chy\ U)))$
ADM9	$(\div(\sin(N(0.0078838, 0.17049))\ -(\cos(\div(chy\ chy))\ chy)))$
ADM10	$(\times(\times(U\ U)\ \times(U\ chy)))$
ADM11	$(\div(\times(U\ \div(chy\ \log(U)))\ U))$
ADM12	$(\div(chy\ -(N(-0.62528, 8.6422)\ \sin(N(-1.3941, 5.1622)))))$
ADM13	$(\div(U\ chy))$
ADM14	$(\div(-(chy\ N(-0.77005, 1.7459))\ +(chy\ N(0.7052, 4.8637))))$
ADM15	$(\sin(N(-0.039909, 2.854)))$
ADM16	$(\times(\cos(\log(U))\ \sin(\sin(\log(pow(\cos(sqr(\log(U)))\ \times(\cos(\log(U))$ $\sin(\sin(\log(pow(\cos(\sin(\log(pow(\cos(sqr(\log(U)))\ U))))\ U)))))))))))$
ADM17	$(\times(\log(N(0.29948, 0.99092))\ U))$
ADM18	$(\cos(N(1.6565, 0.8667)))$
ADM19	$(\log(\cos(\div(\log(\cos(\div(\times(\sin(U)\ U)\ pow((-(sqr(\cos(chy))\ \cos(\sin(N(1.904, 2.002)$ $))))\ \sin(chy)))))\ pow((-(sqr(\cos(chy))\ \cos(\sin(N(1.3206, 2.6021)))))\ \sin(chy))))))$
ADM20	$(\div(U\ +(N(1.1209, 9.3713)\ N(-1.1291, 6.3921))))$
ADM21	$(\log(sqr(N(0.41597, 1.3872))))$
ADM22	$(\times(\log(chy)\ U))$
ADM23	$(\div(+(N(-0.041901, 0.11743)\ \cos(N(1.5605, 0.044548)))\ pow(sqr(U)\ chy)))$

One might expect that for unimodal functions (e.g. f_1 - f_8) unimodal distributions make good mutation operators. The intuition being, as one moves around the domain of the function, there are corresponding changes in the objective value which can clearly guide the search. Similarly, one might expect, multimodal probability distributions to be more suitable to search multimodal functions than unimodal probability distributions. However, our results do not support this. For example, in Figure 2, ADM1 clearly shows peaks at +1 and -1, which is due to the final application of the

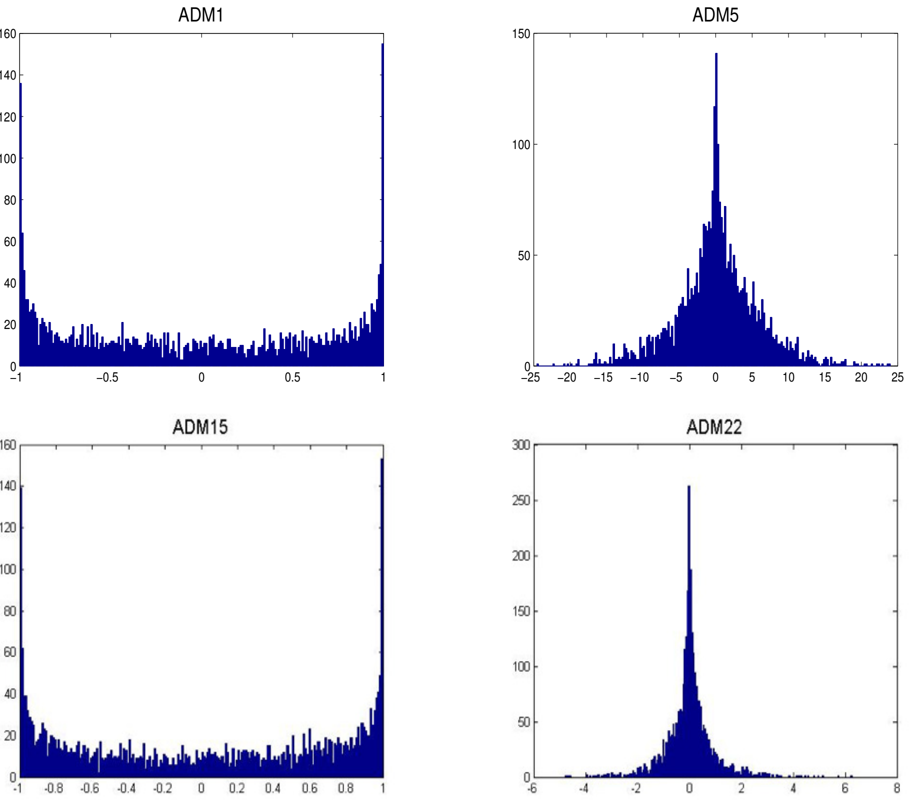

Figure 2: Histograms of the ADM1, ADM5, ADM15 and ADM22 for 3000 samples

trigonometric function. In contrast, ADM5 is unimodal. Conversely, some multimodal functions have resulted in symmetric unimodal probability distributions. For example, ADM15 is U-shaped and not the traditional bell-shape typically used with EP. Not all of the ADMs are symmetric, for example, ADM22, where the log function introduces asymmetry into the mutation operator.

In all but two cases (ADM6 and ADM7), the ADM is not a standard probability distribution (normal, Cauchy, Lévy) but something more complex. It is worth noting that these standard probability distributions are within the GP terminal set, however more 'complex' ADMs are the best found by GP for each function class. This supports the case for the automatic design of mutation operators. An alternative would be to automatically tune the numerical parameters of a Lévy distribution (the α parameter), or normal distribution (i.e. the mean and standard deviation), but this would only ever result in normal distributions which are a linearly scaled version of $N(0, 1)$. However, the automatic design process starts by defining a much broader search space than can be done with numerical parameters alone. This allows GP to find new probability distributions which perform better than the standard probability distributions as mutation operators for EP.

Figure 3 shows the performance of different mutation operators during an EP run, using a single function for eight of the function classes tested.

The ADMs show significant improvement on f_2, f_3, f_4, f_{20}, f_{22}, f_{23}. The figures of f_4, f_{19}, f_{20} and f_{23} shows that the ADMs not only perform well in early generations of the evolutionary process, but also in later generations. The figure for f_2 shows that ADM2 has good performance in the early generations, poor performance towards the middle, but outperforms all other methods by the end the run. This phenomenon was also found and discussed in previous references combining Cauchy and Gaussian mutation operators together. This suggests that, as the performance of a mutation operators varies at different stages of the search, using dynamic or multiple mutation operators may be preferable to using a single operator.

5.1. Performance of Evolved Mutation Operators on Other Function Classes

As a result of the train-and-test approach used, each ADM evolved by GP is designed for a specific function class. For example, ADM1 was designed for functions drawn from the function class f_1. Specifically, it was trained on 10 functions (although only 5 were used for evaluation) drawn from f_1 and tested on a further 50 functions also taken from f_1. This presents the following question: what happens when an ADM designed for one function class is used to optimise functions

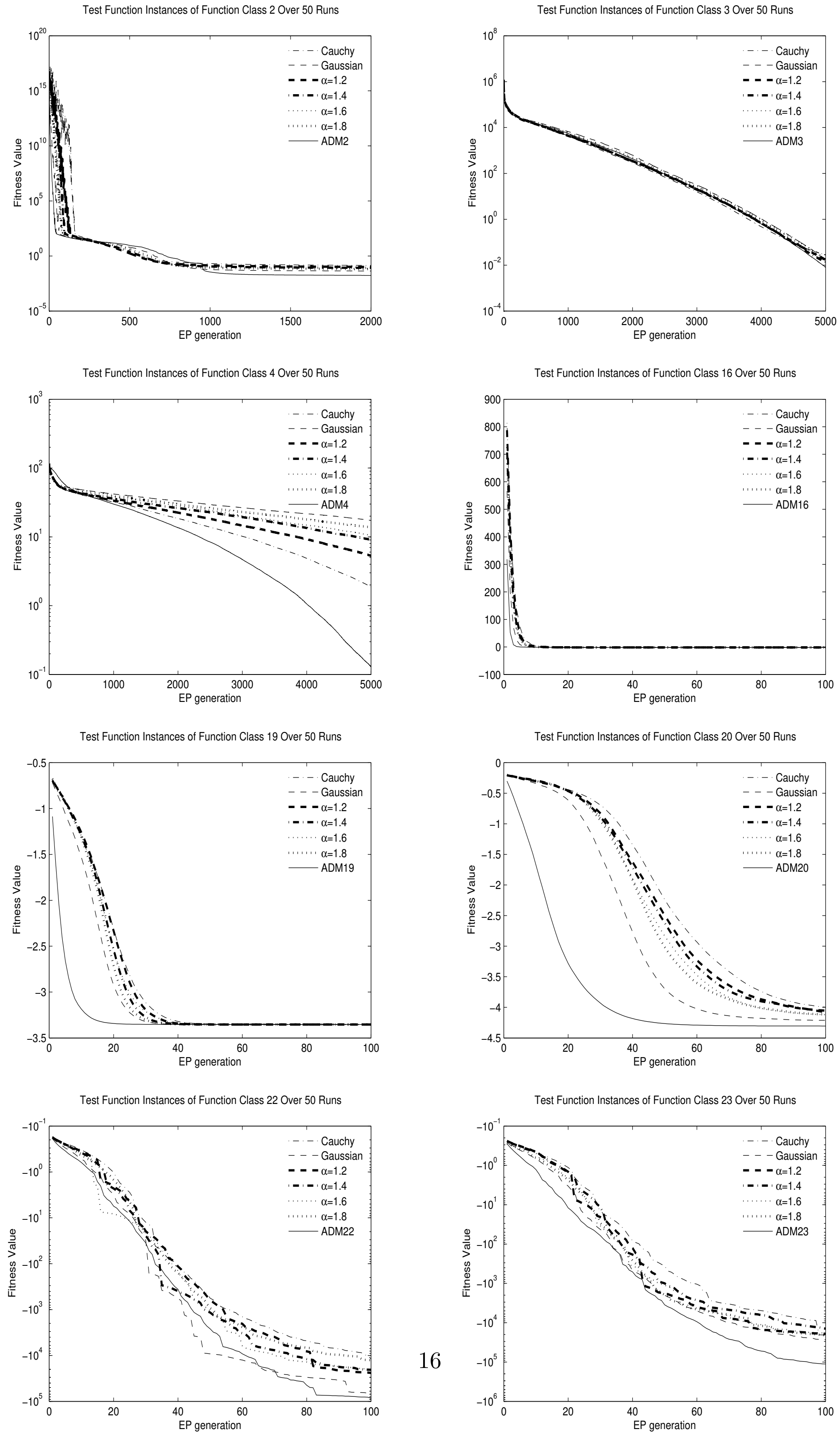

Figure 3: EP evolutionary process for ADM, Gaussian, Cauchy, and Lévy distribution with different value of α, the X axis represents EP generation, the Y axis represents fitness value of EP.

from a different function class? In other words, is the tailored ADM better than an arbitrary ADM?

We will now compare the performance of an ADM tailored to one function class with the other 22 ADMs which are intended for use on different function classes. The mean values and standard deviations achieved by each ADM are presented in Tables 8 and 9. The diagonal is in boldface and represents the performance of ADMi on the function class f_i. For a given row (f_i), the values in boldface indicate the ADMs that have beaten ADMi on f_i. For example, for f_1, ADM1 is outperformed only by ADM10. From both tables we can see that ADM8, ADM11, ADM12 and ADM16 have the best performance on f_8, f_{11}, f_{12} and f_{16}, respectively. ADM1, ADM4, ADM7, ADM9, ADM13, ADM19 and ADM20 have the second best performance on f_1, f_4, f_7, f_9, f_{13}, f_{19} and f_{20}, respectively. The worst performance of the tailored ADMs are ADM15, ADM18 and ADM22: their ranks are 11th, 11th, and 12th, on f_{15}, f_{18} and f_{22}, respectively. Overall, as expected an ADM tailored to a function class has better performance than the non-tailored ADMs. Table 10 shows the results of a Wilcoxon signed-rank test within a 95% confidence interval of a tailored ADM (TADM) compared with non-tailored ADMs (with statistically significant differences in boldface). The tailored ADM is an ADM trained for that specific function class. For example, ADM1 is the TADM for function class 1, but ADM1 is a non-tailored ADM for function class 2. In this table, '$\geq$' and '$\leq$' indicate that an ADM performs better or worse than the other ADMs. In the case that this difference is statistically significant, '$>$' and '$<$' are used. Although ADM16 shows the best performance on f_{16}, in Table 10 '$=$' indicates that the performances of ADM16 and ADM19 are equal on f_{16}. For f_8, '$\geq$' indicates that the performance of ADM8 is better than that of ADM23, but that this is not significant. As can be seen from the last column in Table 10, in the majority of cases, the tailored ADM outperforms all of the other ADMs.

Table 8: Comparison (averaged over 20 runs) of each of the 23 ADMs on each of the 23 function classes (f_1–f_{19}). The fitness value of the TADM is in bold, and those other fitness values that are better than the fitness value of the TADM are also in bold.

	ADM1	ADM2	ADM3	ADM4	ADM5	ADM6	ADM7	ADM8	ADM9	ADM10	ADM11	ADM12	ADM13	ADM14	ADM15	ADM16	ADM17	ADM18	ADM19	ADM20	ADM21	ADM22	ADM23
f_1	**3.79604**	3.79679	3.79617	3.80010	3.79822	3.79843	3.79682	2531.21	3.79606	**3.79599**	3.84599	3.79619	3.79628	3.79665	3.79612	1655.31	16670.68	3.80282	4.26045	3.79609	44.19	3.79607	3.81683
	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(11290.26)	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(15.53)	(2395.68)	(7873.98)	(15.53)	(15.51)	(15.53)	(49.93)	(15.53)	(15.53)
f_2	3.27E-02	**1.73E-02**	6.41E-02	2.44E-01	2.27E-01	2.42E-01	1.41E-01	10.89	**1.26E-02**	**1.56E-02**	8.78E-01	4.83E-02	6.82E-02	1.02E-01	3.30E-02	5.65	62.09	2.94E-02	**3.08E-03**	**8.14E-03**	2.69E-01	3.31E-02	3.96E-02
	(6.2E-03)	(3.2E-03)	(1.2E-02)	(5.3E-02)	(4.2E-02)	(4.5E-02)	(2.7E-02)	(29.6)	(2.4E-03)	(3.2E-03)	(2.2E-01)	(9.0E-03)	(1.3E-02)	(2.1E-02)	(6.5E-03)	(6.57)	(15.77)	(5.4E-03)	(8.5E-04)	(1.5E-03)	(1.5E-01)	(6.5E-03)	(3.9E-02)
f_3	0.0591	0.5426	**0.0186**	0.1031	**0.0139**	**0.0142**	**0.0059**	33.78	2.67	2.04	0.9379	0.2866	0.0641	0.0970	0.0849	3433.71	7087.02	0.1574	50.23	5.58	11.02	0.1992	182.15
	(0.1276)	(0.9071)	(0.0388)	(0.0522)	(0.0062)	(0.0059)	(0.0031)	(25.61)	(3.03)	(2.68)	(0.4713)	(0.4207)	(0.1000)	(0.1135)	(0.1233)	(3872.31)	(4298.40)	(0.1946)	(47.12)	(6.89)	(7.78)	(0.3809)	(157.13)
f_4	17.26	19.42	16.15	**0.29**	9.65	14.05	16.79	28.13	10.99	14.20	**0.16**	2.63	4.40	1.33	21.58	68.40	74.66	40.88	37.07	16.70	61.05	15.81	8.93
	(4.97)	(7.25)	(4.89)	(0.50)	(5.40)	(7.01)	(7.26)	(43.09)	(5.90)	(6.77)	(0.13)	(2.04)	(2.85)	(1.70)	(7.95)	(7.99)	(7.26)	(9.61)	(14.01)	(5.75)	(8.86)	(4.98)	(5.60)
f_5	-12.5987	-11.3929	-**13.3862**	-12.1777	-**12.9862**	-**13.2639**	-**13.4344**	4522608.39	-12.8065	-12.3842	-10.2108	-12.6180	-12.0733	-12.3406	-11.3735	768.5564	189434.0968	-11.6838	-10.2007	-10.9133	6.0180	-12.7928	-7.3536
	(19.08)	(16.78)	(19.29)	(20.18)	(18.94)	(19.31)	(19.83)	(4559249.52)	(19.39)	(17.83)	(18.26)	(18.51)	(17.93)	(17.08)	(18.02)	(1491.14)	(135909.77)	(18.82)	(17.38)	(18.05)	(25.80)	(19.30)	(20.10)
f_6	422.03	1143.00	12.17	0.135	0.058	**0.054**	0.11	9.298	0.107	**0.015**	0.326	**0.039**	**0.035**	**0.032**	195.78	17987.88	37431.70	424.44	375.22	**0.017**	4999.89	185.43	83.99
	(1649.47)	(2074.28)	(38.35)	(0.27)	(0.22)	(0.22)	(0.30)	(6.86)	(0.25)	(0.02)	(0.16)	(0.06)	(0.07)	(0.02)	(490.12)	(13480.82)	(19447.52)	(752.32)	(888.00)	(0.03)	(5268.86)	(446.24)	(123.80)
f_7	0.0655	0.0783	0.0562	0.0841	**0.0470**	0.0488	**0.0487**	2.8072	0.0635	0.0586	0.1949	0.0619	0.0537	0.0623	0.0702	0.5328	45.5222	0.0915	0.0870	0.0659	0.4786	0.0629	0.1865
	(0.0108)	(0.0243)	(0.0116)	(0.0196)	(0.0061)	(0.0065)	(0.0077)	(1.2139)	(0.0128)	(0.0093)	(0.0418)	(0.0107)	(0.0081)	(0.0107)	(0.0105)	(0.4611)	(19.5016)	(0.0228)	(0.0255)	(0.0123)	(0.1751)	(0.0157)	(0.1112)
f_8	-7476.80	-7873.51	-8371.09	-11531.08	-8638.19	-8567.79	-8461.85	-**12471.54**	-10836.76	-10864.24	-11921.11	-11261.25	-11017.46	-11363.52	-7712.38	-9103.97	-7289.48	-7716.48	-11715.12	-10802.04	-7922.38	-8271.94	-12347.43
	(648.37)	(503.11)	(779.63)	(307.06)	(533.70)	(489.80)	(682.14)	(158.44)	(498.97)	(528.49)	(319.68)	(332.98)	(352.28)	(396.68)	(673.42)	(629.07)	(588.01)	(519.33)	(392.40)	(383.83)	(534.77)	(686.29)	(208.99)
f_9	-6.3643	-6.6138	-7.2742	-8.8411	-6.5252	-6.6248	-6.1178	-7.4905	-**8.8540**	-8.8540	-8.7243	-8.8536	-8.8532	-8.8521	-5.7131	-8.4279	11.8005	-6.4276	-8.7550	-**8.8541**	-5.8654	-7.2526	-8.8323
	(13.55)	(14.20)	(13.11)	(11.28)	(13.91)	(14.12)	(14.37)	(11.30)	(11.28)	(11.28)	(11.25)	(11.28)	(11.28)	(11.28)	(15.25)	(11.22)	(19.67)	(13.48)	(11.37)	(11.28)	(13.78)	(13.43)	(11.26)
f_{10}	-26.5488	-27.6246	-27.8734	-28.5297	-28.4378	-27.4749	-28.1247	-7.2537	-**28.5828**	-**28.5820**	-23.8488	-28.5745	-28.5713	-28.5635	-27.3001	-16.8669	-4.3441	-26.5893	-28.4834	-**28.5836**	-23.4200	-27.4131	-28.5401
	(8.64)	(7.69)	(7.11)	(6.28)	(6.45)	(8.12)	(6.63)	(14.71)	(6.29)	(6.29)	(13.26)	(6.29)	(6.29)	(6.29)	(7.78)	(9.53)	(2.23)	(8.95)	(6.39)	(6.29)	(9.54)	(7.76)	(6.29)
f_{11}	-0.6511	-0.4304	-0.6994	-0.7322	-0.6968	-0.7229	-0.6814	-0.7159	-0.6402	-0.6742	-**0.7384**	-0.7172	-0.7353	-0.7251	-0.6194	72.6735	245.0964	-0.3781	-0.3158	-0.5856	5.0014	-0.5955	-0.7322
	(0.55)	(0.59)	(0.53)	(0.55)	(0.57)	(0.54)	(0.55)	(0.55)	(0.56)	(0.55)	(0.55)	(0.56)	(0.54)	(0.54)	(0.53)	(74.08)	(108.70)	(0.52)	(0.64)	(0.54)	(4.93)	(0.54)	(0.55)
f_{12}	2.38	1.52	8.18E-01	4.99E-05	1.14	1.08	2.61	2.95E+08	8.63E-02	4.90E-02	7.61E-04	**2.74E-06**	2.39E-02	9.89E-06	1.50	34.73	1.5E+07	2.15	2.01E-01	1.21E-01	18.16	1.13	4.26E+04
	(2.85)	(2.15)	(1.30)	(1.32E-05)	(1.20)	(2.17)	(3.63)	(5.62E+08)	(2.99E-01)	(1.77E-01)	(2.36E-04)	(1.09E-06)	(5.98E-02)	(3.88E-06)	(1.33)	(35.79)	(2.46E+07)	(1.76)	(4.23E-01)	(3.22E-01)	(10.06)	(1.06)	(1.90E+05)
f_{13}	7.53	11.54	5.38	6.69E-04	3.18	5.34	10.81	4.51E+08	3.80E-03	1.87E-03	1.01E-02	**4.50E-05**	**6.35E-05**	1.26E-04	3.66	333.02	7.36E+06	18.23	7.15E-01	2.86E-02	257.29	5.85	527.92
	(11.5)	(12.6)	(10.3)	(1.4E-04)	(5.0)	(5.6)	(19.7)	(611339756.7)	(7.1E-03)	(4.6E-03)	(2.3E-03)	(2.0E-05)	(1.7E-05)	(2.9E-05)	(4.6)	(304.3)	(4.79E+06)	(21.7)	(1.6)	(7.2E-02)	(118.6)	(7.5)	(2359.9)
f_{14}	1.73	1.77	1.23	1.07	1.69	1.81	1.17	**0.75**	1.29	1.05	**0.78**	**0.83**	1.24	**0.84**	1.46	2.63	0.99	1.35	1.25	1.65	1.54	1.05	0.87
	(1.39)	(1.52)	(0.56)	(0.66)	(1.66)	(1.35)	(0.60)	(0.17)	(1.08)	(0.57)	(0.17)	(0.31)	(1.22)	(0.34)	(0.86)	(3.71)	(0.55)	(0.87)	(1.99)	(1.30)	(1.42)	(0.68)	(0.35)
f_{15}	**6.43E-04**	1.55E-03	7.85E-04	7.66E-04	2.25E-03	1.92E-03	1.20E-03	1.64E-03	**5.36E-04**	**4.18E-04**	1.10E-03	**5.80E-04**	**5.80E-04**	**6.34E-04**	**7.07E-04**	1.55E-03	1.23E-03	1.41E-03	**4.96E-04**	**4.42E-04**	**6.56E-04**	**4.79E-04**	8.71E-04
	(5.4E-04)	(4.4E-03)	(8.6E-04)	(8.6E-04)	(5.8E-03)	(3.5E-03)	(1.8E-03)	(2.0E-03)	(3.7E-04)	(2.9E-04)	(1.2E-03)	(4.0E-04)	(3.6E-04)	(3.9E-04)	(5.4E-04)	(4.5E-03)	(6.9E-06)	(3.9E-03)	(3.9E-04)	(3.5E-04)	(4.3E-04)	(3.1E-04)	(4.5E-04)
f_{16}	-1.92447213	-1.924473263	-1.924470875	-1.924468207	-1.924439114	-1.924429001	-1.924455664	-1.924280681	-1.924473516	-1.924473529	-1.924454275	-1.924473348	-1.924472904	-1.924472224	-1.924472113	-**1.924473543**	-1.92447336	-1.924472505	-1.924473543	-1.924473528	-1.924473215	-1.924473016	-1.92447352
	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)	(0.58)
f_{17}	3.786952682	3.786952414	3.786953089	3.786953821	3.786960153	3.786965855	3.786956682	3.787007025	**3.786952351**	**3.786952347**	3.78696211	3.786952403	3.78695256	3.786952596	3.786952688	**3.786952342**	**3.786952395**	3.78695256	3.786953207	**3.786952346**	3.786952426	3.786952433	3.786952348
	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)	(2.37)
f_{18}	4.575031533	**4.574971819**	4.575098811	4.575282035	4.5763558	4.577539016	4.575858015	4.588715133	**4.574958404**	**4.574958205**	4.576371722	**4.574967813**	4.575001205	4.575011001	4.575028689	6.005761337	**4.574969199**	**4.575008958**	**4.57495705**	**4.574957813**	**4.574974777**	**4.574979692**	**4.57495876**
	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.11)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(6.14)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)	(1.12)
f_{19}	-3.552067105	-3.552080968	-3.552050408	-3.551996944	-3.551673503	-3.538670434	-3.551882996	-3.55009538	-3.552083616	-3.552083593	-3.551749707	-3.55208093	-3.552074315	-3.552064298	-3.552065788	-**3.552084117**	-3.552080969	-3.552072265	-**3.552084001**	-3.552083898	-3.552079811	-3.552078513	-3.552083673
	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.92)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)	(1.90)

Table 9: Comparison (averaged over 20 runs) of each of the 23 ADMs on each of the 23 function classes (f_{20}–f_{23}).

	ADM1	ADM2	ADM3	ADM4	ADM5	ADM6	ADM7	ADM8	ADM9	ADM10	ADM11	ADM12	ADM13	ADM14	ADM15	ADM16	ADM17	ADM18	ADM19	ADM20	ADM21	ADM22	ADM23
f_{20}	-4.0774	-4.0748	-3.8878	-4.0932	-4.0110	-3.4181	-4.0560	-3.9522	-4.1095	**-4.1141**	-3.8083	-4.1051	-4.1100	-3.8682	-4.0393	-4.0958	-4.0869	-4.0615	-4.1102	**-4.1107**	-4.0990	-4.0948	-4.1099
	(2.02)	(1.98)	(1.95)	(2.02)	(2.10)	(1.85)	(2.03)	(2.02)	(2.04)	(2.04)	(1.97)	(2.05)	(2.04)	(1.85)	(2.07)	(2.05)	(1.99)	(1.98)	(2.05)	(2.04)	(2.05)	(2.02)	(2.04)
f_{21}	-7.50	**-8.26**	-6.75	-3.69	-4.44	-4.58	-4.06	-3.64	-7.63	-7.12	-3.55	-7.50	-7.12	-5.96	-7.50	-3.99	**-8.24**	-6.87	-3.97	**-8.14**	**-7.63**	-7.24	-6.99
	(2.4)	(1.6)	(2.9)	(1.5)	(1.9)	(2.6)	(1.7)	(0.3)	(2.3)	(2.7)	(0.7)	(2.3)	(2.7)	(2.6)	(2.1)	(3.3)	(1.5)	(2.7)	(2.7)	(1.9)	(2.4)	(2.3)	(2.5)
f_{22}	-22690.01	**-138764.16**	-16262.26	-7564.46	-4081.24	-7710.71	-8391.89	-661.13	**-138490.10**	**-156034.53**	-2163.99	**-121815.79**	**-75620.18**	-21671.31	**-184449.89**	**-20512221.26**	-71248.07	-53103.24	**-3872071.64**	**-266869.50**	**-88299.30**	**-72258.79**	**-156292.70**
	(30957.5)	(240207.8)	(22759.7)	(14484.9)	(7046.2)	(20971.9)	(15850.7)	(1197.4)	(193997.9)	(256068.3)	(4065.4)	(272880.3)	(142754.5)	(34334.5)	(574931.9)	(81568165.2)	(109512.0)	(85281.3)	(7903902.6)	(408822.5)	(146589.9)	(79909.2)	(227462.4)
f_{23}	-32152.70	-52174.33	-26607.57	-21922.82	-6153.03	-5790.76	-9617.22	-380.46	-103530.92	**-132236.20**	-2388.34	-50209.17	-29264.20	-14424.39	-24997.68	**-3511793.67**	-98707.40	**-211246.70**	**-1024445.84**	**-290292.30**	-107802.76	-36810.44	**-111941.27**
	(61337.3)	(93572.1)	(68340.3)	(54964.3)	(11938.5)	(11424.6)	(19983.7)	(657.0)	(206684.0)	(244626.7)	(4202.0)	(100548.3)	(59321.4)	(31387.8)	(51026.8)	(8105197.0)	(235422.7)	(642660.0)	(2471291.7)	(839727.1)	(284455.3)	(74288.9)	(233075.7)

Table 10: Comparison of ADMs on different function classes.

FC	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM	TADM WIN
	-ADM1	-ADM2	-ADM3	-ADM4	-ADM5	-ADM6	-ADM7	-ADM8	-ADM9	-ADM10	-ADM11	-ADM12	-ADM13	-ADM14	-ADM15	-ADM16	-ADM17	-ADM18	-ADM19	-ADM20	-ADM21	-ADM22	-ADM23	TIMES
f_1	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	$<$	$>$	$>$	$>$	$>$	$\geq$	$>$	$>$	$>$	$>$	$\geq$	$>$	$\geq$	$>$	21
f_2	$>$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$<$	$>$	$>$	$>$	18
f_3	$\geq$	$>$	N/A	$>$	$\leq$	$\leq$	$\leq$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	19
f_4	$>$	$>$	$>$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$\leq$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	21
f_5	$\geq$	$\geq$	$\leq$	$\geq$	N/A	$\leq$	$\leq$	$>$	$\geq$	$\geq$	$>$	$\geq$	$\geq$	$\geq$	$>$	$>$	$>$	$\geq$	$>$	$>$	$>$	$\geq$	$>$	19
f_6	$>$	$>$	$>$	$>$	$\geq$	N/A	$\geq$	$>$	$>$	$\leq$	$>$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$\leq$	$>$	$>$	$>$	17
f_7	$>$	$>$	$>$	$>$	$\leq$	$\geq$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	21
f_8	$>$	$>$	$>$	$>$	$>$	$>$	$>$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	22
f_9	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	$>$	$>$	$>$	$<$	$>$	$>$	$>$	21
f_{10}	$>$	$\geq$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$	$>$	20
f_{11}	$>$	$>$	$>$	$\geq$	$>$	$\geq$	$>$	$>$	$>$	$>$	N/A	$>$	$\geq$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	22
f_{12}	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	$>$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	22
f_{13}	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	$>$	$<$	N/A	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	21
f_{14}	$>$	$>$	$>$	$\geq$	$>$	$>$	$\geq$	$\leq$	$\geq$	$\geq$	$\leq$	$\leq$	$\geq$	N/A	$>$	$>$	$\geq$	$>$	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	19
f_{15}	$\leq$	$\geq$	$\geq$	$\geq$	$\geq$	$>$	$>$	$>$	$<$	$<$	$>$	$\leq$	$\geq$	$\leq$	N/A	$>$	$>$	$\geq$	$<$	$<$	$\leq$	$<$	$\geq$	12
f_{16}	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	N/A	$>$	$>$	$=$	$>$	$>$	$>$	$>$	22
f_{17}	$>$	$\geq$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$<$	$>$	$\geq$	$>$	$>$	$>$	$<$	N/A	$>$	$>$	$<$	$>$	$\geq$	$>$	18
f_{18}	$>$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$<$	$>$	$<$	$\geq$	$\geq$	$>$	$>$	$<$	N/A	$<$	$<$	$<$	$<$	$<$	12
f_{19}	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\leq$	$>$	$>$	N/A	$>$	$>$	$>$	$>$	21
f_{20}	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$	$>$	$>$	$>$	$\geq$	$>$	$>$	$\geq$	N/A	$>$	$>$	$>$	21
f_{21}	$\geq$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$\geq$	$\geq$	$>$	$\geq$	$\geq$	$>$	$\geq$	$>$	$\leq$	$\geq$	$>$	$\leq$	N/A	$\geq$	$\geq$	19
f_{22}	$>$	$\leq$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$\leq$	$>$	$\leq$	$\leq$	$>$	$\leq$	$\leq$	$\geq$	$\geq$	$\leq$	$<$	$\leq$	N/A	$\leq$	11
f_{23}	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	$>$	$>$	$>$	$\geq$	$\leq$	$>$	$\geq$	$>$	$\geq$	$\geq$	$\leq$	$\geq$	$\leq$	$\leq$	$\leq$	$\geq$	$\geq$	N/A	17

6. Discussion

One of the advantages of the new method presented here is that it eliminates the need for human researchers to continually propose new distributions for use as mutation operators in EP. Instead, we have a search space which contains a rich set of mutation operators, and we can let a metaheuristic, such as GP, sample this space and select a suitable choice for the sample of functions at hand. In addition, it designs an ADM within the *context* of a function class. In other words, it tailors a mutation operator (random number generator) to a function class (probability distribution over functions). The suitability of mutation operator depends on the function class. Rather than tuning a *numerical parameter* to a function class, it tunes a *program* that generates random numbers to a function class.

One of the apparent disadvantages of the proposed system is the time needed to evolve the ADMs. This is because we have an EP algorithm at the base level, the mutation operator of which is being evolved by a GP algorithm at the hyper-level. While this may appear to be a superficial disadvantage, there are other advantages. Firstly, it is difficult to measure the amount of human effort required in designing a new mutation operator, and therefore it is difficult to directly compare the design phases of human and machine (GP in this case). We can only sensibly compare the performance of two mutation operators at the testing phase. Secondly, the system can be used to automatically generate new ADMs as and when needed to the demands of a new function class, whereas the human designer would have to start the whole process over again.

It is important to note that the fact that the training and testing are drawn from the same distribution is central to the train-and-test approach. In our case, this means that an ADM is developed to be used as a mutation operator *for a given function class*, but also, importantly, *within a given EP algorithm*, which includes a fixed EP population size and number of generations.

One of the current limitations of the proposed method is that not only must the training set of functions be representative of the testing set of functions, but also the conditions under which they are sampled must be similar. For example, when testing a function from function class f_1, we used the same population size and number of generations of EP as in the training phase. This is a limitation which needs to be addressed. One possibility would be to use GP to train EP using a *different* number of generations and population sizes. However, this would only partially address the issue, since if we attempted to use an ADM with a population size and number of generations outside the ranges seen in the training phase, there would be no guarantee of performance.

7. Summary and Future Work

In this paper we have used genetic programming (GP) as an offline hyper-heuristic to automatically evolve probability distributions, to use as mutation operators in evolutionary programming (EP). This is in contrast to existing operators in the literature which are human designed. The function and terminal set for GP was chosen to be able to express a number of currently existing human designed mutation operators, namely Cauchy, Gaussian and Lévy, and also express novel automatically designed mutation operators (ADMs). Each ADM is constructed from a function set including arithmetic and trigonometric functions, and a terminal set of probability distributions included as standard in many programming libraries and mathematical packages. Using a train-and-test approach, where two independent sets of functions are drawn from the same *function class* for training and testing, it is shown that GP is capable of generating ADMs which outperform existing EP variants over a number of different function classes. As an additional validation exercise, we have also presented experiments to show that the ADM tailored to a given function class performs better that ADMs tailored to different function classes.

There are a number of possible directions for future work. As GP has been able to evolve good variation operators for EP, further work will explore the ability of GP to generate variation operators for other real-valued optimisation methods such as differential evolution (DE) and particle swarm optimisation (PSO), particularly for the case of function classes where a train-and-test approach can be used. Another direction within EP is to extend our hyper-heuristic approach beyond simple static mutation operators. It has been observed previously that both Cauchy and Gaussian mutation are effective in EP at different points of a search [35]. As both of these operators can be defined as a parameterised version of the Lévy distribution, through the use of different α values, we will evolve the value of α as a function over time, subsequently defining a family of adaptive mutation operators which can be trained to specialise in solving different classes of functions.

References

[1] X. Yao, Y. Liu, Fast evolutionary programming, in: Proceedings of the Fifth Annual Conference on Evolutionary Programming, MIT Press, 1996, pp. 451–460.

[2] N. Sinha, R. Chakrabarti, P. Chattopadhyay, Evolutionary programming techniques for economic load dispatch, IEEE Transactions on Evolutionary Computation 7 (1) (2003) 83–94.

[3] P. Shelokar, A. Quirin, O. Cordón, A multiobjective evolutionary programming framework for graph-based data mining, Information Sciences 237 (2013) 118–136.

[4] M. Mutyalarao, A. Sabarinath, M. Xavier James Raj, Taboo evolutionary programming approach to optimal transfer from earth to mars, in: Proceedings of Swarm, Evolutionary, and Memetic Computing (SEMCCO 2011) - Part II, Vol. 7077 of LNCS, Springer, Visakhapatnam, Andhra Pradesh, India, 2011, pp. 122–131.

[5] A. K. Qin, V. L. Huang, P. N. Suganthan, Differential evolution algorithm with strategy adaptation for global numerical optimization, IEEE transactions on Evolutionary Computation 13 (2) (2009) 398–417.

[6] R. Mallipeddi, P. N. Suganthan, Q.-K. Pan, M. F. Tasgetiren, Differential evolution algorithm with ensemble of parameters and mutation strategies, Applied Soft Computing 11 (2) (2011) 1679–1696.

[7] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of the IEEE International Conference on Neural Networks, Vol. 4, Perth, Australia, 1995, pp. 1942–1948.

[8] N. Hansen, The cma evolution strategy: A comparing review, in: Towards a New Evolutionary Computation, Vol. 192 of Studies in Fuzziness and Soft Computing, Springer Berlin Heidelberg, 2006, pp. 75–102.

[9] S. M. Elsayed, R. A. Sarker, D. L. Essam, Adaptive configuration of evolutionary algorithms for constrained optimization, Applied Mathematics and Computation 222 (2013) 680–711.

[10] E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. Woodward, A classification of hyper-heuristic approaches, in: M. Gendreau, J.-Y. Potvin (Eds.), Handbook of Metaheuristics, Vol. 146 of International Series in Operations Research and Management Science, Springer US, 2010, pp. 449–468.

[11] R. Poli, W. B. Langdon, N. F. McPhee, A field guide to genetic programming, Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008, (With contributions by J. R. Koza).

[12] E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. Woodward, Exploring hyper-heuristic methodologies with genetic programming, in: C. Mumford, L. Jain (Eds.), Computational

Intelligence, Vol. 1 of Intelligent Systems Reference Library, Springer Berlin Heidelberg, 2009, pp. 177–201.

[13] J. R. Woodward, J. Swan, Automatically designing selection heuristics, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2011), ACM, Dublin, Ireland, 2011, pp. 583–590.

[14] J. R. Woodward, J. Swan, The automatic generation of mutation operators for genetic algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2012), ACM, Philadelphia, Pennsylvania, USA, 2012, pp. 67–74.

[15] L. Dioşan, M. Oltean, Evolving crossover operators for function optimization, in: P. Collet, M. Tomassini, M. Ebner, S. Gustafson, A. Ekárt (Eds.), Proceedings of the European Conference on Genetic Programming (EuroGP 2006), Vol. 3905 of LNCS, Springer, Budapest, Hungary, 2006, pp. 97–108.

[16] S. Nguyen, M. Zhang, M. Johnston, K. Tan, Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming, IEEE Transactions on Evolutionary Computation 18 (2) (2013) 193–208.

[17] R. E. Keller, R. Poli, Linear genetic programming of parsimonious metaheuristics, in: D. Srinivasan, L. Wang (Eds.), Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007), IEEE Press, Singapore, 2007, pp. 4508–4515.

[18] M. Bader El Den, R. Poli, Grammar-based genetic programming for timetabling, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2009), IEEE Press, Piscataway, NJ, USA, 2009, pp. 2532–2539.

[19] M. Bader El Den, R. Poli, S. Fatima, Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework, Memetic Computing 1 (3) (2009) 205–219.

[20] J. H. Drake, M. Hyde, K. Ibrahim, E. Özcan, A genetic programming hyper-heuristic for the multidimensional knapsack problem, Kybernetes 43 (9-10) (2014) 1500–1511.

[21] L. Hong, J. Woodward, J. Li, E. Özcan, Automated design of probability distributions as mutation operators for evolutionary programming using genetic programming, in: Genetic Programming, Vol. 7831 of LNCS, Springer Berlin Heidelberg, 2013, pp. 85–96.

[22] G. L. Pappa, A. A. Freitas, Discovering new rule induction algorithms with grammar-based genetic programming, in: Soft Computing for Knowledge Discovery and Data Mining, Springer, 2008, pp. 133–152.

[23] S. Nguyen, M. Zhang, M. Johnston, K. C. Tan, Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming, IEEE Transactions on Evolutionary Computation 18 (2) (2014) 193–208.

[24] J. Park, S. Nguyen, M. Zhang, M. Johnston, Evolving ensembles of dispatching rules using genetic programming for job shop scheduling, in: Genetic Programming, Vol. 9025 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 92–104.

[25] J. Branke, S. Nguyen, C. W. Pickardt, M. Zhang, Automated design of production scheduling heuristics: A review, IEEE Transactions on Evolutionary Computation 20 (1) (2016) 110–124.

[26] R. Poli, J. Woodward, E. K. Burke, A histogram-matching approach to the evolution of bin-packing strategies., in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007), IEEE, 2007, pp. 3500–3507.

[27] E. K. Burke, M. R. Hyde, G. Kendall, J. Woodward, Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2007), ACM, London, England, 2007, pp. 1559–1565.

[28] E. K. Burke, M. R. Hyde, G. Kendall, J. R. Woodward, The scalability of evolved on line bin packing heuristics, in: D. Srinivasan, L. Wang (Eds.), Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007), IEEE Press, Singapore, 2007, pp. 2530–2537.

[29] A. Parkes, E. Özcan, M. Hyde, Matrix analysis of genetic programming mutation, in: A. Moraglio, S. Silva, K. Krawiec, P. Machado, C. Cotta (Eds.), Genetic Programming, Vol. 7244 of LNCS, Springer Berlin Heidelberg, 2012, pp. 158–169.

[30] E. Özcan, A. J. Parkes, Policy matrix evolution for generation of heuristics, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2011), ACM, Dublin, Ireland, 2011, pp. 2011–2018.

[31] J. H. Drake, J. Swan, G. Neumann, E. Özcan, Sparse, continuous policy representations for uniform online bin packing via regression of interpolants, in: European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2017), Vol. 10197 of LNCS, Springer, 2017, pp. 189–200.

[32] R. Poli, W. Langdon, O. Holland, Extending particle swarm optimisation via genetic programming, in: M. Keijzer, A. Tettamanzi, P. Collet, J. Hemert, M. Tomassini (Eds.), Genetic Programming, Vol. 3447 of LNCS, Springer Berlin Heidelberg, 2005, pp. 291–300.

[33] J. Drake, N. Kililis, E. Özcan, Generation of vns components with grammatical evolution for vehicle routing, in: Genetic Programming, Vol. 7831 of LNCS, Springer Berlin Heidelberg, 2013, pp. 25–36.

[34] L. Hong, J. H. Drake, J. R. Woodward, E. Özcan, Automatically designing more general mutation operators of evolutionary programming for groups of function classes using a hyper-heuristic, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2016), ACM, New York, NY, USA, 2016, pp. 725–732.

[35] X. Yao, Y. Liu, G. Lin, Evolutionary programming made faster, IEEE Transactions on Evolutionary Computation 3 (1999) 82–102.

[36] T. Bäck, H.-P. Schwefel, An overview of evolutionary algorithms for parameter optimization, Evolutionary Computation 1 (1) (1993) 1–23.

[37] C.-Y. Lee, X. Yao, Evolutionary programming using mutations based on the lévy probability distribution, IEEE Transactions on Evolutionary Computation 8 (1) (2004) 1–13.

[38] R. N. Mantegna, Fast, accurate algorithm for numerical simulation of lévy stable stochastic processes, Physical Review E 49 (1994) 4677–4683.

[39] H. Dong, J. He, H. Huang, W. Hou, Evolutionary programming using a mixed mutation strategy, Information Sciences 177 (1) (2007) 312 – 327.

[40] R. Mallipeddi, S. Mallipeddi, P. Suganthan, Ensemble strategies with adaptive evolutionary programming, Information Sciences 180 (9) (2010) 1571–1581.

[41] S. Silva, J. Almeida, Gplab-a genetic programming toolbox for matlab, in: Proceedings of the Nordic MATLAB conference, 2003, pp. 273–278.

[42] S. Luke, L. Panait, Lexicographic parsimony pressure, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), Morgan Kaufmann Publishers, New York, NY, USA, 2002, pp. 829–836.