The Automatic Generation of Mutation Operators for Genetic Algorithms

[Workshop on Evolutionary Computation for the Automated Design of Algorithms]

John Woodward
The University of Nottingham, China
199 Taikang East Road
Ningbo, Zhejiang, 315100, P.R.C.
john.woodward
@nottingham.edu.cn

Jerry Swan
Department of Computing Science and Mathematics
School of Natural Sciences, University of Stirling,
Stirling FK9 4LA, SCOTLAND.
jerry.swan@cs.stir.ac.uk

ABSTRACT

We automatically generate mutation operators for Genetic Algorithms (GA) and tune them to problem instances drawn from a given problem class. By so doing, we perform meta-learning in which the base-level contains GAs (which learn about problem instances), and the meta-level contains GA-mutation operators (which learn about problem classes). We use Register Machines to explore a constrained design space for mutation operators. We show how two commonly used mutation operators (viz. one-point and uniform mutation) can be expressed in this framework. Iterated local search is used to search the space of mutation operators, and on a test-bed of 7 problem classes we identify machine-designed mutation operators which outperform their human counterparts.

Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming—*Program synthesis*

Keywords

Genetic Programming, Genetic Algorithms, Hyper-Heuristics, Automatic Design

1. INTRODUCTION

Metaheuristics are practical solution methods to intractable problems [13], Genetic Algorithms (GAs) being one such method [3].

An important issue in determining the utility of a metaheuristic is the notion of *problem classes*, i.e. a probability distribution over a set of problem instances. The notion of a 'best-performing' metaheuristic must be considered as a function of the problem instances to which it will be exposed. For a distribution of problem instances p_1, metaheuristic m_1 may outperform metaheuristic m_2, while for distribution p_2, the converse may be the case, i.e. best depends on the context provided by the problem class.

In the approach of automatic design, we let the search process determine which metaheuristic is better for the problem class at hand simply by training it on problem instances sampled from that problem class. The output of the automated design process is then a GA-mutation operator designed for the specific problem class which was used during the training process.

In the experimental set up used in this paper, a problem instance is never encountered twice. For the training and testing phases, new instances are produced, as is the case for the comparison of human-designed and machine-designed mutation operators. In other words, we cannot meaningfully compare algorithms on a fixed set of instances, but only on sets of instances drawn from a given problem class.

Engineering design is an intrinsically multi-dimensional activity, where the dimensions might be cost, energy efficiency, durability or maintainability etc. These goals often conflict, resulting in a trade-off. Similarly Software Engineers design to a specification, and nature evolves organisms in response to contextual environmental feedback (i.e. an organism "fits" its environment). [2] argues that there is a trade-off with metaheuristics which is often referred to as the No Free Lunch theorems, and therefore supports the meta-learning approach, i.e. it is an argument that we should automate the design process for problem classes in general. [10] has proved that for one problem class and algorithm, there exists another problem class and algorithm, where the performance is identical. This implies that we should target algorithms at problem classes. These two papers together imply that for a given problem class, a meta-learning technique can be employed to automatically design a meta heuristic (or a component thereof) for that problem class. Indeed, the classification of metaheuristics and the problem instances to which they should be applied are intrinsically linked [**?**].

Genetic Programming (GP) [4] is a standard method for automatically generating algorithms. With the current level of maturity of GP, its naïve application of GP is insufficient to generate a metaheuristic (for example a GA) from scratch (i.e. from a set of simple general purpose computational primitives such as INC and DEC (see Table 1). The breakthrough is not in using a more sophisticated search

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

process, but rather providing a framework or template into which GP can insert candidate programs. As evidence that the framework is an essential part of this approach, the results obtained in this article required nothing more sophisticated than hill-climbing to search the space of GP-generated designed. The framework can be viewed as constraining the space of algorithms (which includes e.g. sorting, prime number factorization etc.) to a more tractible subset that are be interpreted as GA-mutation operators.

Register machines are used to implement GA-mutation operators, and define a framework in which currently-existing mutation operators can be expressed. Generated mutation operators are embedded into a GA and run on a number of problem instances to assess their utility. Mutation operators are continually created in a repeated hill-climbing algorithm. We go to lengths to avoid infinite loops. For example, we can only move forward in a program (i.e. we cannot jump backwards), and the number of times instructions are iterated over is fixed at the start of any iteration process. Thus, the RM, rather than being Turing-complete can only express the primitive recursive functions. However, we do not view this as a limitation as the primitive recursive functions suffice to express (for example) all mappings from $\mathbb{B}^n \rightarrow \mathbb{B}^n$, which is what is required for mutating the representations used in this paper.

1.1 Motivation

There are a number of motives for the automated design of mutation operators (and in the wider scheme of complete metaheuristics). GAs are a generate-and-test approach and therefore it makes sense to employ a generate-and-test approach to design them, i.e. to use a closed feedback loop to contain the item being generated and repeatedly expose it to the kind of instances it will be tested against.

With this approach, we are searching at a higher level of abstraction, i.e. we are not searching the space of solutions (which is what a GA does) but instead searching the space of solution methodologies (in this case mutation operators which form part of a GA). The key motive here is if a scalable mutation operator is discovered, it can be applied to large problem instances and perhaps outperform a standard mutation operator.

An interpretation of this proposed methodology is that it allows researchers to propose a family of algorithms (in this case a large set of mutation operators) rather than proposing just a single algorithm which is what currently happens in many research articles. While the human designer still controls the design space of mutation operators, we can let the machine search this design space for an appropriate mutation operator for the given problem class.

The final point is that machines can outperform humans (higher quality and at lower cost) as we demonstrate in this paper. In addition Moore's law states that processor speed is increasing exponentially (which is good news for automated design), while the cost of human labor increases in-line with inflation (which is bad news for manual design).

1.2 Outline

The outline of this paper is as follows. In Section 2 we give the background. In Section 3 we give the experimental methodology. In Section 4 we give the results In Section 5 we give the further work. In Section 6 we give the summary and conclusions.

2. BACKGROUND

2.1 Meta-learning and learning to learn

Learning is defined as the improvement at a task according to some measure [5]. This is often achieved by sampling a search space of learners and maintaining the best. Often *the way* learning occurs with many learning algorithms does not change however. This can be achieved employing a meta-learning approach.

The key to meta-learning is to have two levels, a *base level* and a *meta level* [15, **?**]. At the base level, learning about a problem instance takes place. At the meta level learning about the problem class takes place. We can use any machine learning algorithms at the base and meta levels. Consider (as in this paper) the domain of generating function optimizers, where we use a GA as a function optimizer in the base level and an iterative hill-climber to learn about GA mutation operators in the meta level. At the base level a GA samples a function f at points x_1 and x_2, and perhaps learns $f(x_1) < f(x_2)$ (this is all that can be learned!).

Similarly, at the meta level a hill-climber samples the space of mutation operators expressed as register-machines and learns RM_1 outperforms RM_2 on the given problem class (again, this is all that can be learned).

2.2 Genetic programming as a hyper-heuristic

In GP, functions are created from a function and terminal set. It can be difficult for GP to evolve an entire algorithm (involving branching, iteration, and reading and writing to memory) from nothing more than a set of primitives. Hence, we adopt an approach called *Genetic Programming as a Hyper-heuristic* [1] in which currently-existing heuristics for the problem domain are examined and any commonality is extracted in the form of a template. This template provides a framework in which the generative components of GP are expressed as extension points (c.f. the *template method pattern* of [**?**]). In other words, any algorithmic invariant is identified and fixed, and GP can then be used to discover new functions which can be plugged into the variable parts of the framework. Such a framework provides a means by which the human designer can support and constrain what can be evolved. Thus the final algorithm consists of a human-designed template which is filled-in with functions produced by GP.

GP as a Hyper-heuristic can be considered as a special case of meta-learning, where the representation employed is the language (terminals and nonterminals) of a genetic program. This is an important point as the human designer must be able to confirm that the language of the GP can express pre-existing heuristics of interest. We do this in Section 3 where we show how two popular mutation operators can be expressed with a RM instruction set.

GP cannot currently be applied directly to many combinatorial problems and a bridge has to be built which allows a GP program to be able to be applied to a problem instance. The bridge takes the form of the framework which defines how any GP program is to be utilized to solve the problem (see [16] for some problems with GP). For example, it is not immediately obvious how GP could be applied to e.g. the knapsack problem, whereas it is easier to see how GA can be applied directly. This difficult arises because GP is working at a higher level of abstraction (i.e. the search space contains algorithms which must somehow be applied),

whereas GA works directly on a space of solutions (which are often easy to encode as e.g. a bit-string) In other words, G-P produces a completely machine-made algorithm, whereas Genetic Programming as a Hyper-heuristic produces a part man-made (the template) part machine-made (methods in the template) algorithm.

2.3 Literature review

2.3.1 Applications of GP as a HH

In recent years there have been a number of isolated examples of this approach: [14] evolved dispatching rules for solving multi-objective flexible job-shop problems; [8] tackled image recognition problems and [7] data mining algorithms.

It is interesting to note that human-designed heuristic are often rediscovered by these systems as they can be expressed as the composition of a few primitives. It is often in the first generation that these heuristics are found (i.e. effectively by random search, as no evolution has yet taken place). This raises in interesting question of whether or not to seed the initial population with human-designed heuristics (as expressed in the language).

2.3.2 Generating search algorithms

There have been a few attempts at generating search algorithms themselves. [9] decomposes search algorithms into four components (select, replace, growth, variation) and defines a grammar. Interestingly this framework can express all of random search, hill-climbing, annealing, and GAs. However, one major criticism of this work is that as the grammar is not recursive — it is effectively only generating combinations of components of known search algorithms. [6] creates a system which can express a steady-state GA, a standard GA and evolutionary programming. This system can express more complex combinations than [9].[12, 11] takes an interesting approach — the idea here is that by using a representation which is Turing-complete, it can express any learning algorithm. This method also incorporates the potential, not only for meta-learning, but recursively includes meta meta learning and so on. There are however a number of issues with this approach.

- While it can express any learning algorithm, it can also express irrelevant algorithms.
- It may fall foul of the halting problem.
- Algorithms are difficult to evolve, and evolving a learning algorithm from scratch is difficult (which is why we seed our hill-climber with a known learning algorithm).
- Although his system cleverly allows recursive meta-learning, we show in this paper that a two-layer learning mechanism is sufficient to produce novel algorithms.

3. EXPERIMENTAL METHODOLOGY

Optimization typically has a single phase, however in this paper we are learning to optimize, hence (as with all machine learning approaches) there are two phases. First, there is a training (or learning) phase where the RMs can be adjusted (generated and tested). Second, there is a testing (or validation) phase where the RMs are demonstrated to perform well on new problem instances drawn from the same probability distribution as the training phase. The fact we are designing optimizers is reflected in a methodology that differs from that which is typical in the optimization and machine-learning research communities.

The conventional approach to manually developing a metaheuristic is to implement and test on a set of benchmark instances taken from the research literature or e.g. from some online repository. A metaheuristic may be derived from theory, be inspired by 'natural computing' metaphors etc. The success of a metaheuristic is determined by its performance on a set of benchmark instances.

Our methodology differs in two fundamental respects. Firstly, the algorithms are automatically generated by machine rather than being human-designed. Secondly, the algorithms are tested on a problem class rather than 'disconnected' problem instances.

3.1 Function signatures

Given a function $f : \mathbb{B}^n \rightarrow \mathbb{R}^+$ (i.e. a non-negative real-valued function defined over Boolean vectors of length n), a conventional binary GA returns the Boolean vector $\bar{b}^*$ that yields the best value for $f(\bar{b})$ encountered by the GA. Ignoring parameters such as population size (which we keep fixed in this paper), such a GA can be considered as a function mapping from a function $f : \mathbb{B}^n \rightarrow \mathbb{R}$ to a real value $\mathbb{R}^+$ (via the application of f to $\bar{b}^*$. The function signature for a standard GA is therefore:

$$GA : (\mathbb{B}^n \rightarrow \mathbb{R}^+) \rightarrow \mathbb{R}^+$$

In this paper we are developing mutation operators as part of an overall GA, having signature:

$$mutation : \mathbb{B}^n \rightarrow \mathbb{B}^n$$

The input to this system is a list of functions $[f : \mathbb{B}^n \rightarrow \mathbb{R}^+]$ (where square brackets denote a list), and the output is a GA. Therefore the signature of the whole system is $[\mathbb{B}^n \rightarrow \mathbb{R}^+] \rightarrow ((\mathbb{B}^n \rightarrow \mathbb{R}^+) \rightarrow \mathbb{R}^+)$, i.e. a list of functions maps to a GA.

Comparing this signature with that of a standard GA shows that we are working at a higher level of abstraction. The output of this system is a GA in which the mutation operator has been generated to be tuned to a specific problem class. More concretely, a GA returns the solution to a problem instance, while in this paper, we are generating a system which returns a solution method (i.e. algorithm) to a problem class (i.e. a set of problem instance samples).

3.2 Framework

We describe a RM-framework for GA mutation operators and show how two of the most well-known operators (namely one-point and uniform mutation) can be expressed in this framework (see Table 3).

3.2.1 Register machines

Here we describe a RM and show how its input-output behavior can be interpreted as a mutation operator. A RM is a list of primitive instructions (program) and a list of addressable registers (memory) which hold data (providing a mechanism to map input to output) [17].

Our RM formulation maintains one set of floating point registers for input-output and one for working calculations.

The input-output register is initialised with the bit-string to be mutated and is placed on registers R_1 to R_n where n is the length of the bit-string, and $R_0 = 0$. We encode the bit-string as $\{0 \mapsto -1, 1 \mapsto +1\}$. There is a working register (of size 5) which can be used to calculate intermediate values. All values in the working register are initialised to zero. Note that the command *Set* stores a value which is written to a target register, so providing a mechanism for the program to store constants. After the execution of the RM the output (i.e. mutated bit-string) is read from R_1 to R_n. A positive value is mapped to true, else is mapped to false.

If the register-argument to an instruction is non-negative, it is the index of a working register. If the register-argument to an instruction is negative, the negated argument is used as the index of a working register. The contents of this register is then used as the index of an input-output register. In other words, instructions can only affect the input-output registers by indirection via the working registers. For example the instruction *inc* 1 increments the working register R_1 by 1. While the instruction *inc* -1 increments the input-output register pointer to by working register R_1 (i.e. if working register $R_1 = 4$ then the input-output register 4 is incremented). All register indices are modulo the size of the associated register set.

A program counter (*pc*) stores an integer which points to the next instruction to be executed (modulo program size). After "arithmetic" instructions are executed (see table 1), the next instruction in the list is executed (i.e. $pc = pc + 1$). Some instructions $\{If, IfRand, Rpt, Stp\}$ alter the flow of control by changing *pc* (see Table 2). They operate as follows: If take 3 arguments; if the contents of the register pointed to by the first argument is greater than the contents of the register pointed to by the second argument then *pc* jumps by the value pointed to by the third argument (the absolute value is used).
With $IfRand$ if the contents of the register pointed to by the first argument is less than 100 times a dynamically generated random number in the range [0, +1] then *pc* is incremented by the value pointed to by the second argument (the absolute value is used). Note that any changes to *pc* are positive so no infinite loops are possible (i.e. we cannot jump backwards in the program repeatedly).
The *Rpt* instruction executes the next R_j instructions R_i times. While the values of R_i and R_j may change during this execution, their initial values are stored locally in the *Rpt* instruction so we can guarantee termination. In other words, a *Rpt* instruction will terminate after $(contentof R_i) * (contentsof R_j)$ instructions (assuming there are no other instructions that affect *pc* in the body of the loop).

We do allow nested *Rpt* loops, and even though these are guaranteed to terminate, they can still take a long time to execute. Hence, for each execution of a program we terminate it after 64*20 instructions have been executed (allowing the program chance to iterate over the genotype). This limit is set high enough to allow both hand-coded RM which express one-point and uniform mutation to terminate naturally.

3.2.2 One-point and Uniform mutation

Here we describe how one-point and uniform mutation can be described in the RM-framework (see Table 3). One-point mutation flips at most a single bit in the bit-string. Uniform mutation flips all bits with equal probability.

Table 1: Instructions which do not affect the program counter

Instruction	Action	Params
Add	$R_i \leftarrow R_j + R_k$	3
Inc	$R_i \leftarrow R_i + 1$	1
Dec	$R_i \leftarrow R_i - 1$	1
Ivt	$R_i \leftarrow -1 * R_i$	1
Clr	$R_i \leftarrow 0 * R_i$	1
Rnd	$R_i \leftarrow Random([-1, +1])$	1
Set	$R_i \leftarrow value$	2

Table 2: Instructions which affect the program counter

Instruction	Action	Params
If	$if(R_0 > R_1)$ $pc = pc + R_2$	3
IfRand	if(arg1 < 100 * random[0,+1]) pc = pc + arg2	2
Rpt	Repeat R_j times next R_i instruction	2
Stp	$terminate$	0

These mutation operators have parts in common. Let us first describe uniform mutation The first instruction $Rpt, L, 18$ repeats L times the following 18 instructions (where L is 33 or 65 in these problems). The $4th$ instruction $Ivt, 3$ inverts (i.e. multiplies by -1) the value in the $3rd$ working register. The $IfRand$ instruction takes two arguments, the first is the mutation probability (which is $1/(bit - stringlength)$) and the second is the number of instructions to jump

The only line number where the two RM-programs differ is line 13. If a bit is flipped then the one-point mutation RM-program executes the *Stp* command (i.e. termination).

Note that the instruction *Nop* (no operation) is included in the RM instruction set but is not included in the function set of GP. This was done to encourage the search for new programs. As observed above, the presence of these *Nop* instructions allows mutation to make changes with less chance of a discontinuous effect on the original program. An important point to note is that while these mutation operators can be expressed as RMs consisting of a few lines, in the experiments we padded-out each program with a number of NOPs: the rationale being that if an important instruction is deleted, it may completely destroy any useful behavior of the RM. However, instructions can replace the initial NOPs without such deleterious effect.

3.3 Problem Classes

We generate problem instances for each of our 7 problem classes in three stages (see Table 4).

1. We generate a Normally-distributed value $t = -0.7 + 0.5N(0, 1))$ in the range [-1, +1]. If the number, t for target, falls outside this range it is regenerated. The constants -0.7 and 0.5 provide an arbitrary offset and scale which are fixed for all experiments.

2. We linearly interpolate the value t from the range [-1,

Table 3: Human designed RM programs

Line	Uniform	One-point
0	$Rpt, 33, 18$	$Rpt, 33, 18$
1	Nop	Nop
2	Nop	Nop
3	Nop	Nop
4	$Inc, 3$	$Inc, 3$
5	Nop	Nop
6	Nop	Nop
7	Nop	Nop
8	$IfRand, 3, 6$	$IfRand, 3, 6$
9	Nop	Nop
10	Nop	Nop
11	Nop	Nop
12	$Ivt, -3$	$Ivt, -3$
13	Nop	Stp
14	Nop	Nop
15	Nop	Nop
16	Nop	Nop

$+1]$ into an integer in the range $[0, 2^{num-bits} - 1]$, and convert this into a bit-string $t^{'}$ (using a Gray coding).

3. To calculate the fitness of an arbitrary bit-string x, the hamming distance between x and the target bit-string $t^{'}$ is calculated (giving a value in the range [0,num-bits]). This value is then fed into one of the 7 functions.

Table 4: 7 function classes

no.	function
p1	x
p2	$sin^2(x/4 - 16)$
p3	$(x - 4) * (x - 12)$
p4	$(value * value - 10 * cos(value))$
p5	$sin(\pi * value/64 - 4) * cos(\pi * value/64 - 12)$
p6	$sin(\pi * cos(\pi * value/64 - 12)/4)$
p7	$1/(1 + value/64)$

Table 5: Parameter settings for the RM search

Parameter	Value
restart hill-climbing	100
hill-climbing iterations	5
mutation rate	3
RM program length	17
Input-output register size	33 or 65
working register size	5
seeded	uniform-RM
fitness	best, averaged over 20

3.4 Experiments

Our aim is to generate a GA-mutation operator for each problem class. Each time a GA is run (either with a human-designed operator or a machine generated operator) a new problem instance is generated.

Table 6: Parameter settings for the GA

Parameter	Value
Population size	100
Iterations	1000
bit-string length	32 or 64
generational model	steady-state
selection method	fitness proportional
fitness	see table 4

At the meta-level, we are searching the space of mutation operators (i.e. RMs) for a given problem class. At the base-level the space of bit-strings for a given problem instance is being searched. Base and meta-level parameters are as given in Tables 6 and 5 respectively. Hence a single GA run consists of 100 fitness evaluations to initialize the population, and 1000 fitness evaluations to conduct the evolution. To search the space of RMs, we use hill-climbing, a single-point search methodology. We initialize the search with the human-designed RM representing uniform mutation (see column 2 in Table 3). We mutate RMs uniformly at random such that on average 3 instructions are mutated in a program. We allow 5 hill-climbing iterations so it is possible that 15 out of 16 of the instructions are mutated. This process is repeated 100 times. Thus 500 RMs will have been evaluated. At the end of each GA run, the best individual bit-string is recorded. The GA is run 20 times, and the fitness of the RM is the best of each GA run, averaged over the 20 runs. The fitness of a RM is thus an indication of what value it is expected to obtain on new problem instances.

4. RESULTS

To make a comparison between human automatically-designed mutation operators, batches of 20 GA runs are conducted and the best-of-run is recorded for each GA execution. These 20 best-of-run fitnesses are then averaged over the 20 runs. This is repeated 30 times to give 30 sets of data to compare. Two tailed t-tests are conducted using two-sample unequal variance (heteroscedastic).

The means and standard deviations can be found in tables 7 and 8 for 32 and 64 bit versions of the problem classes respectively.

The p-values can be found in table 9. These are arrived at by comparing the automatically generated RM with uniform and one-point mutation operators.

5. FURTHER WORK

The methodology described in this paper has achieved superior results to those of the most popular human-dsesigned mutation operators and is therefore worthy of further study. There are a number of interesting directions in which this work can proceed.

While we have obtained statistically significant results with this method, the automatically-designed operators afford little insight into their effectiveness. We could take the stance that the machine-generated operators outperform their human-designed counterparts and that is how meta-

Table 7: Statistics for 32-bit functions on the 7 problem classes

	Uniform	One-point	RM-mutation
p1 mean	30.82	30.96	31.11
p1 std-dev	0.17	0.14	0.16
p2 mean	951.0	959.7	984.9
p2 std-dev	9.3	10.7	10.8
p3 mean	506.7	512.2	528.9
p3 std-dev	7.5	6.2	6.4
p4 mean	945.8	954.9	978.0
p4 std-dev	8.1	8.1	7.2
p5 mean	0.262	0.260	0.298
p5 std-dev	0.009	0.013	0.012
p6 mean	0.432	0.434	0.462
p6 std-dev	0.006	0.006	0.004
p7 mean	0.889	0.890	0.901
p7 std-dev	0.002	0.003	0.002

Table 8: Statistics for 64-bit functions on the 7 problem classes

	Uniform	One-point	RM-mutation
p1 mean	55.31	56.08	56.47
p1 std-dev	0.33	0.29	0.33
p2 mean	3064	3141	3168
p2 std-dev	33	35	33
p3 mean	2229	2294	2314
p3 std-dev	31	28	27
p4 mean	3065	3130	3193
p4 std-dev	36	24	28
p5 mean	0.839	0.846	0.861
p5 std-dev	0.012	0.010	0.012
p6 mean	0.643	0.643	0.663
p6 std-dev	0.004	0.004	0.003
p7 mean	0.7520	0.7529	0.7684
p7 std-dev	0.0028	0.0040	0.0031

Table 9: p-values for 32 and 64-bit functions on the 7 problem classes

	32 bit	32 bit	64 bit	64 bit
class	Uniform	One-point	Uniform	One-point
p1	1.98E-08	0.0005683	1.64E-19	1.02E-05
p2	1.21E-18	1.0848E-12	1.63E-17	0.00353
p3	1.57E-17	1.649E-14	3.49E-16	0.00722
p4	4.74E-23	1.219E-16	2.35E-21	9.008E-13
p5	9.62E-17	1.667E-15	4.80E-09	4.23E-06
p6	2.54E-27	4.14E-24	3.31E-24	3.64E-28
p7	1.34E-24	3.00344E-18	1.45E-28	5.14E-23

heuristics are compared. However design is an iterative process, and understanding how these mutation operators work can feedback into the manual part of the design process (i.e. the design of the framework). It should also be noted that the design of mutation operators should not be studied in isolation, and that it is the interplay of variation (mutation), selection and fitness evaluation (i.e. the complete process of evolution) that needs to be understood if we are to successful automate the design of evolutionary search algorithms.

In this paper we created a system which easily expresses two human designed mutation operators. There are myriad mutation operators reported in the literature, and further work could include confirming that these are also readily expressible within the current RM framework. If they are not, then the framework could be extended to include them. However, we believe we have laid the foundations for a framework that can express a large family of mutation operators (those which are primitive recursive).

Further work includes extending the current unary operator architecture to a binary operator architecture. This would allow operators such as binary crossover to be expressed. This could be done by having two input-output registers which take two bit-strings as input and using instructions like swap(i,j) which would swop the ith bit of the first string with the jth bit of the second string. As in this paper, a framework could be built which can express two commonly used crossover operators, for example one-point and uniform crossover.

We have used a single-point search operator (repeated hill-climbing) to search the space of RMs and was seeded with a RM implementing uniform mutation. We found that one-point mutation often outperformed uniform mutation, and perhaps using one-point mutation as the initial seed for hill-climbing would have allowed us to either find good RMs faster, or find better RMs than we did using uniform mutation. However this adds weight to the case that automated design can find mutation operators which are comparable with, or better than human designed operators. Further work would include using a more sophisticated population-based search method such as GP, which would allow multiple seeds (i.e. we could seed the initial population with a variety of human-designed mutation operators but also with some randomly generated RMs).

By inspecting the two human-designed mutation operators presented in table 3, we can see striking similarities between these two operators. The only difference is a single instruction (line 13). This suggests that this may be a rich part of the search space to concentrate on. For example, it may be that it is beneficial to have *Rpt* instructions which allow the program to iterate over the bit-string, along with some conditional instructions. Indeed many of the manufactured RMs do contain these features, however it is not know whether these are due to the search being initialized with similar programs or these features are actually advantageous.

A hard upper limit is set on the total number of instructions which can be executed in a single execution of a program. In order to investigate the scalability of the mutation operators we obtain with this approach, this limit needs to be removed. If the mutation operators are used on problem instances where the size is greater than this limit currently allows and the program is terminated before it has run to completion, the mutation will fail to produce anything of use.

6. SUMMARY AND CONCLUSIONS

We have defined a framework in which two of the most commonly used GA-mutation operators can be expressed.

We have automatically generated register machines and used them to implement mutation operators for a GA. These mutation operators are then tested on a set of function optimization problem instances.

We demonstrated that the automatic generation of mutation operators for well-defined problem classes yields results that are human-competitive in respect of the two most popular mutation operators (vizòne-point and uniform mutation). This methodology is also cooperative in that it can incorporate human-designed operators into the framework, and indeed this is an essential part of the methodology.

In addition, mutation operators are designed automatically in the context of a problem environment, rather than being designed manually and in isolation from the intended problem class.

It is important to recall our methodology at this point: we are not investigating the performance of a mutation operator on a set of benchmark problem instances, nor are we generating a single mutation operator. We are generating specific mutation operator for specific problem classes (in this case function optimization over functions defined on Boolean domains). To emphasize the point, we are tailoring mutation operators to particular problem classes, and a mutation operator designed for one problem class will not necessarily perform well on a different problem class.

The automatic design of heuristic algorithms by a generate-and-test approach has proved itself to be successful in a number of different domains including job-shop scheduling, the traveling salesman problem, timetabling, Boolean satisfiability, bin packing, and image recognition. We have added mutation for GAs to this list.

On of the main objections to this approach may be the long training time. However, what is not reported in the literature is how much personnel time was spend developing the algorithms in the first place. We also think that this methodology works best by starting with existing heuristics, rather than evolving them from scratch. Therefore this methodology cannot be compared directly with manual design as they are two different approaches. This method could be considered as "algorithmically polishing" a heuristic which has already undergone years of manual design.

This approach will bear fruit if is adopted by the research community over the manual-design approach. This will manifest itself in two ways. Firstly we will see fewer papers that compare algorithms, but instead papers that compare frameworks. For example another template (framework) for the design of mutation operators could be proposed and compared to the register-machine framework described in this paper. Secondly, researchers will test their algorithms (or the algorithms produced by their frameworks) on a class of problem, rather than on unrelated problem instances.

7. REFERENCES

[1] Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In Christine L. Mumford and Lakhmi C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, chapter 6, pages 177–201. Springer, 2009.

[2] C. Giraud-Carrier and F. Provost. Toward a Justification of Meta-learning: Is the No Free Lunch Theorem a Show-stopper? In *Proceedings of the ICML-2005 Workshop on Meta-learning*, pages 12–19, 2005.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[5] Tom M. Mitchell. The need for biases in learning generalizations. Technical report, Rutgers University, New Brunswick, NJ, 1980.

[6] Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[7] Gisele L. Pappa and Alex A. Freitas. *Automatically Evolving Data Mining Algorithms*, volume XIII of *Natural Computing Series*. Springer, 2010.

[8] Mark E. Roberts and Ela Claridge. A multistage approach to cooperatively coevolving feature construction and object detection. In Rothlauf et al., editor, *Applications of Evolutionary Computing, EvoWorkshops2005*, volume 3449 of *LNCS*, pages 396–406, Lausanne, Switzerland, 30 March-1 April 2005. Springer Verlag.

[9] Brian J. Ross. Searching for search algorithms: Experiments in meta-search. Technical report, Brock University, 2002.

[10] Jonathan E. Rowe and Michael D. Vose. Unbiased black box search algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 2035–2042, New York, NY, USA, 2011. ACM.

[11] Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

[12] Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Simple principles of metalearning. Technical report, SEE, 1996.

[13] El-Ghazali Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009.

[14] Joc Cing Tay and Nhu Binh Ho. Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers & Industrial Engineering*, 54(3):453–473, 2008.

[15] Sebastian Thrun and Lorien Pratt, editors. *Learning to learn*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[16] John R. Woodward and Ruibin Bai. Why evolution is

not a good paradigm for program induction: a critique of genetic programming. In Lihong Xu, Erik D. Goodman, Guoliang Chen, Darrell Whitley, and Yongsheng Ding, editors, *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600, Shanghai, China, June 12-14 2009. ACM.

[17] J.R. Woodward and J. Swan. Automatically designing selection heuristics. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 583–590. ACM, 2011.