Automatically Designing Selection Heuristics

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

Jerry Swan
Automated Scheduling, Optimisation and
Planning (ASAP) Research Group,
School of Computer Science, University of
Nottingham,
Jubilee Campus, Wollaton Road, Nottingham
NG8 1BB, UK.
jerry.swan@nottingham.ac.uk

ABSTRACT

In a standard evolutionary algorithm such as genetic algorithms (GAs), a selection mechanism is used to decide which individuals are to be chosen for subsequent mutation. Examples of selection mechanisms are fitness-proportional selection, in which individuals are chosen with a probability in proportion to their fitness value, and rank selection, in which individuals are selected with a probability in proportion to their ordinal ranking by fitness. These two human-designed selection heuristics implicitly assume that fitter individuals produce fitter offspring. Whilst one might invest human ingenuity in the construction of alternative selection heuristics, the approach adopted in this paper is to represent a generic family of selection heuristics which are applied via an algorithmic framework. We then generate instances of selection heuristics and test their performance in an evolutionary algorithm (which in this paper tackles a variety of bitstring optimization problems). The representation we use for the program space is a *register machine* (a set of real-valued registers on which a program is executed). Fitness-proportional and rank selection can be expressed as one-line programs, and more sophisticated selection heuristics may also be expressed. The result is a system which produces selection heuristics that outperform either of the original selection heuristics.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence: Automatic Programming**]: Miscellaneous

General Terms

Algorithmic Tuning, Automatic Design, Selection Heuristics

1. INTRODUCTION

1.1 Heuristics and metaheuristics

Problems which cannot be solved exactly within practical time and memory constraints require the use of *heuristics* — elements of the solution strategy that are intended to promote the discovery of good solutions without requiring the exploration of the entire problem space. Metaheuristics may be considered to be abstract control strategies that have been found to achieve good results across different problem domains. Metaheuristics include simulated annealing, hill-climbing, GAs and tabu-search
[4, 6, 8].

Human-designed metaheuristics represent only a small fraction of the space of possible metaheuristics. In addition, it is known that no single metaheuristic can perform well across all possible problem instances [17]. Just as a species is fit for a particular niche in an environment, so a *problem class* (i.e. a probability distribution over a set of problems, see sections 7.1, 1.4) defines a niche which is appropriate for a particular metaheuristic. There is therefore a compelling requirement for the automatic design of metaheuristics. This paper addresses some of the issues involved in automatically designing one component of a metaheuristic for a particular problem class, namely the selection procedure.

There is a distinction between search algorithms and metaheuristics. A search algorithm (e.g. binary search) which is used under certain circumstances (e.g. finding an item in a sorted list) or searching a list for the first occurrence of a required item. Search algorithms are typically, not only tractable, but provably very efficient. In contrast, a metaheuristic performs the function of sampling a space, but in a heuristic fashion as the space is intractably large (e.g. the space of routes is $\mathcal{O}(n!)$ in a TSP problem) and therefore only a tiny subset of the space can be sampled in the hope of discovering a good enough solution within a time limit. In other words, metaheuristics belong to the family of generate-and-test algorithms. In this paper we will be generating and testing generate-and-test algorithms.

1.2 Generic Algorithms

Machine learning is a branch of Artificial Intelligence concerned with the induction of general rules from training data. Evolutionary Computation can be considered as a branch of machine learning which generates solutions to a target problem class. For example, if we are tackling a Traveling Salesman Problem (TSP) problem with GAs, the solution may be represented as a permutation of the cities in the order in which they are to be visited. If we encounter a new problem instance, we have to execute the GA again to generate another solution.

A more generic approach would be to produce an *algorithm* which can solve many instances of TSP, rather than producing a *single solution* each time. This distinction is well-made in [12] and captured by the phrase: "*Give a man a fish and he will eat for a day, teach a man to fish and he will eat for a lifetime.*" A discussion of some of the differences between GA and GP can be found in[19].

One popular approach to generating algorithms in this fashion is GP [9, 1]. GP is a method of producing functions (expressed as programs) which map inputs to outputs. An immediate issue of using GP in its conventional form as *Koza trees* [9] is how to utilize the output of such a syntax tree within a TSP setting. While is it very natural to use a permutation of cities within a GA representation, it is not obvious how to apply a GP tree to an instance of the TSP problem in order to produce solutions which are permutations of the cities.

We use the term *Generic Algorithms* to denote the use of GP within a algorithmic framework which defines the context for the mapping from the problem domain to\from the GP components. Together these two parts constitute a Generic Algorithm which can be used for multiple problem instances. This is summarized in the following equation.

$$\text{Generic Algorithms} = \text{Genetic Programming} + \text{Application Framework.}$$

Genetic Programming in this equation, could be replaced by any other metaheuristic (e.g. random search which we do in this paper) which is used to search a space of programs (e.g. register machines) so this equation maybe rewritten as "Generic Algorithms = (Metaheuristic + Space of Programs) + Application Framework". We prefer Generic Algorithms = Genetic Programming + Application as GP is in particular the metaheuristic used for searching a space of programs.

Our approach effectively automatically designs a generic algorithm for a problem class. If the probability distribution of the problem class changes, a human designer must revisit the heuristic design process. The method proposed in this paper automates this process: a new heuristic can be automatically produced for a new problem class.

Similar to the induction process in machine learning, the development of generic algorithms has a training phase followed by an independent testing phase (see section 4.1). During the training phase, the system is exposed to different scenarios and allowed to alter its heuristic. The testing phase then employs the most general heuristic obtained from the learning phase.

1.3 Automatic design of selection heuristics

One component of an Evolutionary Algorithm is the selection heuristic. Two ubiquitous selection policies are fitness-proportional selection and rank selection. In this paper, we represent selection heuristics with Register Machines (RMs) equipped a small instruction set. There are two inputs (the fitness of a bitstring and the rank of a bitstring in the sorted population). The output value is then used for selection purposes in place of rank or fitness. Random search is used to sample the space of RMs (i.e. the space of possible selection heuristics). The best resulting RM is then used as a selection mechanism in a GA.

We could have used other ways to represent selection heuristics such as syntax trees [9], and we made this choice arbitrarily. In order to search the space of RMs, we used random search as this was sufficient in this case. Again other methods could have been used.

1.4 Problem classes and Problem Instances

Central to our approach is the concept of a problem class for which the heuristic is designed. For this paper, we fix the domain to be that of functions which map bitstrings of a fixed length to a floating point value. A problem class is then a probability distribution over the set of all such functions and a problem instance is simply one such function.

It is important to note that we are not developing a general selection heuristic which will perform well for all functions (which is prohibited by the NFL theorems [17]). The NFL theorems prohibit any gain over function spaces [17], but NFL theorems are not valid over program spaces [22, 19]. We are automatically designing selection heuristics for a specific problem class. In other words, if we were presented with instances from a different problem class (e.g. deceptive functions as opposed to mimicry problems), we make no guarantees about performance.

1.5 Contribution of this paper

The contribution of this paper is a simple framework for the generation of novel selection heuristics for use in Evolutionary Algorithms. The selection heuristics expressible by this framework are not just a linear weighted sum of the component heuristics. Both rank and fitness-proportional selection are easily expressed in this framework. We also demonstrate that this framework produces selection heuristics which outperform either of the component heuristics on the problem class on which we train. Since we are producing new selection heuristics, we introduce the term "algorithmic tuning" as opposed to parameter tuning to differentiate this work from other literature (see section 3.3).

The research question being addressed in this paper is "*can we produce a system which can automatically generate effective selection heuristics*"? The conclusion of this paper is that we can reply in the affirmative. The tangible deliverable of this work is a method for producing appropriate selection heuristics for any problem domain presented to the framework. The purpose of this paper is thus not simply to propose another selection operator: the literature is already flooded with operators *without a cause*, i.e. constructed via ad hoc methods. The approach of tuning an algorithm to a problem class solves a number of issues:

1. It will generate new selection heuristics which are statistically guaranteed to perform no worse than either of the human-designed heuristics on which the system is built.

2. If a new heuristic is developed (either by human or machine), it can be incorporated seamlessly into the current framework. This avoids having to make an *a priori* choice of selection heuristic, as the system will generate one at least as good as the supplied set of heuristics.

3. The resulting selection heuristic is tailored to the problem class it is trained on, and thus will perform better on the target problem class than an "all purpose" se-

lection heuristic such as rank or fitness proportional selection.

This paper only provides "proof of concept" on a simple problem class. Future work will expand this with more difficult problem classes and more sophisticated search mechanisms (e.g. GP) to discover novel heuristics.

1.6 Outline of remainder of paper

As motivation is so important, we devote the whole of section 2 to it. In section 3 we review the current literature. The proposed methodology is described in section 4, and experimental studies and results are presented in section 5. In section 6 we discuss the philosophy of the proposed method, and finally the paper draws to a close with a conclusions section 7.

2. MOTIVATION

2.1 Automatically designing metaheuristics

The use of GP to produce human-competitive heuristics has already been demonstrated in a number of application areas including SAT, TSP, on-line bin-packing and data-mining [12, 3, 2, 13]. It therefore worth considering the automated design of metaheuristics themselves, and this paper is a step in that direction.

It is our claim that results given in this paper support the automation of the design process and show that there is little point in hand-coding new selection heuristics. This is particularly true given the lack of theoretical results to guide us. The design process proposed in this paper is a generate-and-test approach (as is the metaheuristic approach itself), and so it makes sense to hand-over as much of what can be automated as possible to the machine. The framework approach we propose is a step in the direction of full automation.

2.2 Reducing development costs

Heuristics save time and money. What is often ignored is the cost of developing these heuristics in the first place (rather than the savings they produce after development).

In most endeavors, there is generally a tradeoff between cost and quality, with lower cost implying lower quality, however this is *not* the case here. Mass-producing heuristics with the framework we propose will result in better-quality and lower-cost heuristics as compared to human-designed heuristics.

High-quality implies that the heuristics are tailored to the specific problem class, as they cannot be universally high-quality for all problem instances [17, 20]. Low-cost implies the heuristics are designed by computer and not by costly error-prone humans.

3. LITERATURE REVIEW

We begin by looking at some of the problems of evolving algorithms in general. We then look at some success-stories similar to the approach proposed in this paper. Finally we look at parameter tuning, which is also expressible within the currently proposed framework.

3.1 Evolving Algorithms

Previous work has been concerned with evolution of algorithms, which include loops, conditionals and access to memory i.e. the Turing-computable functions [18]. Algorithms have been automatically generated which perform functions of multiplication or listing the Fibonacci sequence [7]. However, the automatic generation of algorithms is not without huge practical difficulties, not least of which is difficulty associated with non-terminating programs and the highly rugged landscape associated with programs [16, 21]. The vast majority of Turing-complete programs do not halt, and while this problem is formally semi-decidable most papers take the simplistic approach of imposing an upper limit on the number of execution steps or execution time. We may therefore conclude that evolving algorithms capable of universal computation is a difficult goal, and there is the need for a syntactic framework to constrain what may potentially evolve.

3.2 Genetic Programming Hyperheuristics

Work has been done using GP to evolve programs that operate in a framework. Applications include on-line and off-line bin-packing, SAT, TSP and data-mining [12, 3]. In each of these cases, GP is used to evolve functions that are applied to the problem domain within the context of a framework.

The basic approach, common to all these works, consists of a number of stages (see [3] for further details);

1. Examine currently-existing heuristics for any common components. Define a framework which can express currently existing heuristics and that provides a context for newly-generated heuristics to operate in.

2. Define a GP system in terms of fitness-function, terminal and function sets, and run the GP system.

The contrast between evolving algorithms capable of universal computation, and evolving functions in a framework means that we restrict the possible interpretations of the evolved programs. That is, evolving algorithms *without* a framework can produce *any* computable function, while evolving algorithms *with* a framework is restricted to *only* functions with a framework-mandated signature.

3.3 Parameter tuning and self-adaptation

A trivial way of hybridizing two methods is to parameterize them over a probability distribution. This is a limited approach as it only allows a linear combination of the component parts. The framework we are proposing can certainly express this approach of a linear weighted sum. One example of how we can combine two operators with a parameter a (where $0.0 \leq a \leq 1.0$) is

```
if rand < a then
        operator1
else
        operator2
```

where rand is a random number in the range $[0.0, 1.0]$. If we set $a = 0$, we get operator1, and if we set $a = 1$, we get operator2.

Instead of having search operators act directly on the representation, the methodology of *self-adaptation* employs operators that act on a numerical parameter which in turn affects how operators affect the representation [10]. For example, a common parameter of many evolutionary systems is the mutation rate — if it is set too high, it will perform a very random search, conversely set too low it will not make sufficient progress through the search space. This sort of parameter is critical to the performance of the algorithm, and is therefore an ideal candidate to undergo self-adaption, with the mutation rate being governed by the evolutionary process itself. Instead of deciding upon a fixed initial value for a parameter, the responsibility of altering the parameter is made online by the system itself (using feedback from the evolutionary system). Thus parameter tuning, self-adaption and hybridization are connected. It is important to note that self-adaption occurs online during a single evolutionary run. This is in contrast to our approach where the selection heuristic is fixed for a single run, and only allowed to after from one run to the next run. It should be noted also that self-adaptation is not without its associated problems [11].

4. PROPOSED APPROACH

We begin by describing the stages of development of a generic algorithm. We describe the signature for our selection heuristics, and how a RM can be instantiated to do this. Finally we explain how current selection heuristics can be expressed as RM programs.

4.1 Stages of development

There are 3 stages in the development of the selection heuristics (as with all generic algorithms, see page 21 of [12].

1. the training or learning phase.
2. the testing or validation phase.
3. the real-world application or execution phase.

In the training phase, RMs are given the task of learning a selection heuristic, on sets of instances drawn from a given problem class. In the validation phase, we measure the performance of a RM as a selection heuristic on a set of independent problem instances drawn from the same problem class. At this stage, if a heuristic fails (i.e. is not deemed good enough), we can return to the first stage. In the real-world application phase, the selection heuristic are past the development stage and are therefore fixed.

When we compare our generic algorithms expressed as RMs with human-designed heuristics, we ignore any cost associated with the training and testing phases, i.e. we only consider the performance of heuristics after they have been designed for an application. In these experiments, we are therefore testing the performance of the heuristics, *not* the system that produced the heuristics. It is our contention that this is a fair approach because the cost of human-designed heuristics (however that might be determined) is not traditionally included in such comparisons.

4.2 Selection heuristic signature

Our framework defines the signature of the selection function to be: $Selection : (2^N, \mathbb{R})^P \to (2^N)^P$ where P is the population-size and N is the length of the bitstrings. Fitness-proportional and rank selection are clearly possible implementations of this signature.

The *selection policy* that we employ within our framework to implement this signature makes use of a register machine to generate a function

$$sel : (fitness(x), rank(x)) \mapsto r \in \mathbb{R}^+$$

where $fitness(x)$ and $rank(x)$ are functions that respectively determine the fitness and rank of an individual x. The output value of this generated function is determined for each individual in the population and individuals are then selected with a probability in proportion to this output value.

4.3 A selection framework

Here we consider two selection heuristics, and how they fit into the same framework (i.e. they are two specific instances of our generalization). In general, the selection mechanism in an Evolutionary Algorithm is a two stage process. Firstly, a value is assigned to each individual (its index in the case of rank selection, and its "normalized" fitness in the case of fitness-proportional selection). Secondly, individuals are selected in proportion to the value assigned to them (see section 5.2). For the purposes of this paper, we are concerned with how to assign values to each individual in the population and keep the second stage fixed. In this paper we are assuming we are maximizing a function. If we were minimizing a function then it would have to be transformed into a maximization problem.

In the case of *fitness* proportional selection, individuals are selected in proportion to their fitness.

$$p_i = fitness_i / (\textstyle\sum_{j=1}^{j=P} \text{fitness})$$

where fitness is normalized in the range [0,1] and p_i is the probability that the ith individual will be selected for the next generation and P is the population size. In the case of *rank* selection, individuals are selected in proportion to their rank (or index) in the sorted population (sorted by fitness).

$$p_i = i / (\textstyle\sum_{j=1}^{j=P} rank(j))$$

where p_i is the probability that the ith individual will be selected for the next generation and P is the population size. Normalized fitness means that fitness are in the range 0.0 to 1.0 (and so can be interpreted as probabilities).

In the second stage of selection, individuals are conditionally promoted to the next generation on the basis of the value assigned to each one.

```
for all individuals p in population
select p in proportion to value( p );
```

where *value* is one of rank or fitness and since we wish to interpret the output value as a probability for subsequent use in proportional selection, we normalize all such values accordingly. Within a proportional-framework, we may therefore use either the individual's rank or fitness and have two different selection heuristics in the same proportional-framework.

4.4 Searching for Register Machines

In order to represent selection heuristics, we elected to use a RM (we might have alternatively decided to use Koza-style GP syntax trees). A RM is a model of computation consisting of a program (a list of instructions see section 5.2) and a list of registers which act as memory and are updated according to the program's instructions [7].

A program counter controls which instruction in the program is to be executed next. For the experiments in this paper, we employed only arithmetic and copy operators, and instructions are therefore executed in a sequential fashion.

The program is terminated when the final instruction is executed: programs of n instructions therefore terminate after n steps.

To search the space of RMs we use random search. As this application is simple, we do not need to over complicate the matter using more sophisticated methods or representations.

4.5 Expressing current selection heuristics

It is desirable that our system can at least express the two human designed heuristics. In our framework, this is trivial, since as these heuristics are fed directly into the system as inputs on the registers. Each RM is initialized with the value of the fitness of a bitstring on R_0 and the rank of the individual in the population on R_1. Rank selection is expressed by the RM program as copy R1 R0 and fitness-proportional selection is expressed by the RM program as NOP since the fitness is already sitting on the output register. The decision regarding the registers to which fitness and rank are assigned is effectively arbitrary. If fitness was placed on a register other than R_0, it would only be a case of copying the value to the output register R_0. Both human-designed heuristics can therefore easily be expressed within this RM framework. The consequences of this are:

1. Selection heuristics equivalent to rank and fitness proportional are readily expressible in the framework and can almost trivially be discovered by searching the space of RM programs.
2. Programs that perform as well as the two human-designed heuristics are simple and therefore likely to be generate early in the search process, providing a baseline against which better-performing programs can be discovered.

5. EXPERIMENTS

We compare a set of automatically-generated RM selection heuristics against the two human designed heuristics; rank selection and fitness-proportional selection. These comparisons on conducted on a set of test cases, drawn independently from the training cases but with the same underlying probability distribution.

5.1 The Mimicry Problem Class

An example of a discrete optimization is the *mimicry problem* [5], a generalization of the well-known *onemax* problem [15]. The goal is to generate a bitstring identical to a fixed target bitstring, the objective value $e(x)$ being given by maximizing the ratio of the number of correct bits in the source to the total number of bits (i.e. the hamming distance divided by the length of the bitstring). Thus $0.0 \leq e(x) \leq 1.0$.

Our problem class for the Mimicry problem is parameterized by a Gaussian distribution. An instance of the problem class is obtained by:

1. Filtering values from $N(0, 1)$ such that they were restricted to the closed interval $[-1, 1]$.
2. Linearly interpolating the value into the range $[0, 2^{num-bits} - 1]$.
3. The target bitstring is then given by the Gray coding [14] of the linearly interpolated value. The rationale for using Gray coding is that it makes it easier to generalize across problem instances than if a direct binary encoding of the linearly-interpolated value were used.

5.2 Parameter settings

Parameter	Value
num-bits	64
metaheuristic-num-runs	50
metaheuristic-population-size	30
metaheuristic-num-generations	50
metaheuristic-mutation-probability	0.1

Table 1: Parameter settings for the metaheuristic

The parameters used for the bitstring evolution are given in Table 1, The length of bitstrings is num-bits. The population size of bitstrings is metaheuristic-population-size. Each bit in the bitstring is flipped with a probability of metaheuristic-mutation-probability. The number of generations the evolutionary metaheuristic is run for is metaheuristic-num-generations. The number of independent instances drawn from the problem class for which the evolutionary algorithm is run is metaheuristic-num-runs.

Parameter	Value
random-search-iterations	100
RM program length	2
register size	3
output register	R_0
contents of R_0	fitness
contents of R_1	rank
contents of R_2	0 (working register)

Table 2: Parameter settings for the RM framework

The parameters used for the RM generation are given in Table 2. Fitness and rank are placed on R_0 and R_1 respectively and R_2 is left free for the RMs to use as a temporary working register. The output is taken from R_0. For all experiments, the nonterminal function set (i.e. instruction set) for the RM is given in section 3. The search space is thus relatively small, as we only have to find two instructions and appropriate arguments for each instruction.

Instruction	Action	Arguments
Inc	$R_i \leftarrow R_i + 1$	1
Dec	$R_i \leftarrow R_i - 1$	1
Add	$R_k \leftarrow R_i + R_j$	3
Sub	$R_k \leftarrow R_i - R_j$	3
Mul	$R_k \leftarrow R_i * R_j$	3
Div	$R_k \leftarrow R_i / R_j$ if $R_j \neq 0, 0$	3
Set	$R_i \leftarrow x \in \mathbb{R}$	2
Copy	$R_i \leftarrow R_j$	2
Clear	$R_i \leftarrow 0$	1
Swap	$R_i \leftrightarrow R_j$	2

Table 3: Instruction names, with their action and number of arguments. Note, all instructions take register indexes as arguments ($i, j, k \in \{0, 1, 2\}$) except set which also take a value [-1,1] $\in \mathbb{R}$

5.3 Metaheuristic generational model

We use the generational model in which a new population is generated from an old population, individuals are selected from the current population, undergo mutation and fitness evaluation, and are inserted into the next population. This is repeated until the next population is full, and this entire process is repeated each generation. This is in contrast to the steady-state approach where there is only a single population and individuals are placed directly into the current population.

5.4 Results

	Fit Prop	Rank	RM-select
mean	0.8315281	0.9078094	0.9160875
std dev	0.003094834	0.002517495	0.006958214
min	0.824375	0.9028125	0.9025
max	0.8384375	0.9146875	0.9290625

Table 4: Results

After running the system, we obtained the statistics in table 4. Performing t-test comparisons of fitness-proportional selection and rank selection against RM-selection resulted in a p-value of better than 10^{-15} in both cases. In both of these cases the RM outperforms the standard selection operators.

6. DISCUSSION

6.1 Why do we need a framework at all?

One may ask "why do we need a framework?" when we might alternatively evolve a selection algorithm using otherwise unconstrained GP. Our response is that evolving algorithms in such an unconstrained fashion is likely to fail [21]. For example, some RMs may not assign values to some of the individuals in the population, whereas a framework approach forces RMs to iterate over the population, assigning a value to each individual. Secondly if we allow arbitrary algorithms to evolve, we may be producing RMs which perform unwanted functions such as sorting, listing the Fibonacci sequence, checking to see if the input is prime, or (more likely) performing some arbitrary function of their input. Having a framework constrains the GP program to performs a certain task. The GP program is then responsible for improving the quality of that task, within the context of the supplied framework.

6.2 Random search of register machines

We claim that the fact that we need use no more than random search to successfully explore our constrained space of selection heuristics is illustrative of the power of this method. It is part of our intended program of future work to use GP to search the space of such heuristics at the topmost level, but our results show that random search suffices for proof of concept.

6.3 Guaranteeing human competitiveness

In this paper we have produced a method of generating selection heuristics that perform better on the target problem class than either of the two human-designed heuristics that were incorporated at the design stage.

In one sense, the positive result regarding performance in this paper is a foregone conclusion. It would have been very unlikely that rank or fitness-proportional selection were the best selection heuristics for the given problem class. We therefore had every reason to be optimistic about the potential success of the system in advance of its implementation.

One might compare this method with a self-adaptive approach, but it is worth noting that if the initial framework is capable of expressing a self-adaptive approach within the space of possible designs, then there is every reason to be confident that our approach will find it and outperform it.

We should emphasize that we are a long way from our ultimate goal of fully-automated design and we consider that the current method can best be described as human-assisted design, since it is the human who ultimately defined the framework and the associated search-space of possible heuristic designs.

6.4 Scaling of fitness values

Scaling the fitness of an individual by a constant factor would not change the relative fitness of two individuals and therefore the performance of fitness-proportional selection would not change. In other words, fitness-proportional selection is invariant under proportional scaling.

If we were to use a strictly-increasing monotonic scaling of fitness value, the rank of an individual in the population would be preserved e.g. if $f(x)$ if we monotonically scaled $m(f(x))$ (by the definition of a strictly-increasing monotonic function for such a function m) and therefore rank selection would perform the same (rank selection is invariant under monotonic scaling).

6.5 Other information

In this paper we have only used two terminals (rank and fitness). In our "learning to walk before we can run" approach to research, we can now ask the question "what other information may be of use to our automatic design method which can be used as terminals in the GP system?" An obvious addition would be the generation number or the number of individuals process, as the selection heuristic could then effectively be time varying, in an analogous way to simulated annealing where the acceptance probability decays exponentially. Another useful terminal would be a constant and perhaps a random number generator (possibly time varying).

6.6 Freeing ourselves of human assumptions

Rank and fitness-proportional selection are both selection heuristics which have in common that fitter individuals are more likely it is to be selected. On the face of it, this seems reasonable, as expresses by the evolutionary sound-bite "survival of the fittest", but this is not always the best policy. By automatically generating selection heuristics, we can free ourselves of this central, sometimes limiting, assumption and any other implicit assumptions we might subconsciously make when hand-crafting heuristic approaches.

7. CONCLUSIONS

7.1 Link between problem classes and automatic design of metaheuristics

Often, when a new heuristic is proposed in the literature, it is tested on a set of benchmark problem instances. It is usually reported that it performs better on some problem instances and worse on others. This observation supports the case for the automatic design of heuristics to target a problem class. In this paper we have presented such a method where a heuristic is trained for a particular problem class, and is therefore automatically trained for exposure to subsequent instances of that problem class.

7.2 Framework vs. algorithm level design

The work contained in this paper represents a paradigm shift from thinking of heuristics at the algorithmic level (i.e. the implementation level of a single heuristic), to the framework level (i.e. a set of heuristics). In other words, researchers should not be occupied with the design of heuristics, but should work at the level of a framework (i.e. a space of possible algorithms) and employ a generate-and-test approach to road-test candidate algorithms. Put differently, the generation of heuristics is cheap (when done automatically), and the automatic generation of heuristics can only be achieved by having a framework in place in which different heuristics can be expressed.

Put differently, instead of researchers proposing heuristics, they can propose a whole family of heuristics A generate-and-test method can process the family deciding which one is best for the problem class at hand.

We are not saying this is "the" framework for selection. As we said in section 6, other information (i.e. terminals and functions) can be added easily into this framework, or indeed other representation of functions (e.g. Koza style GP trees, rather than RMs). We encourage research at the framework level, rather than at the algorithm level, the advantage being that a computer can search the space of algorithms defined by the framework.

The design of a good framework is still something of an art, rather than a science, just as the design of a heuristic used to be an art before the introduction of this method. A framework should be expressive enough to describe a few currently existing heuristics. (i.e. have the potential of rediscovering these heuristics). It should also be able to *simply* express (i.e. in a few lines of code) a number of currently existing heuristics (i.e. they have the potential of being found *easily* by GP, or even generated at random in the initial population). The framework should not be too restrictive that the set of heuristics expressible is too limited (e.g. a weighted sum of two currently existing heuristics). Nor should the framework should be too expressive (e.g. expressing functions which do not perform the sort of function required).

7.3 Raising the level of generality

It is known that metaheuristics must be tuned to a specific problem class [20]: when presented with a different problem class, there are no guarantees regarding the performance of a metaheuristic on this class (it could perform better, worse or be statistically indistinguishable). What our method affords is a robust approach for automatically configuring a metaheuristic component (in the case of this paper, a new selection heuristic) for instances drawn from an arbitrary problem class.

8. REFERENCES

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[2] E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. The scalability of evolved on line bin packing heuristics. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages 2530–2537, Singapore, 25-28 September 2007. IEEE Computational Intelligence Society, IEEE Press.

[3] 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.

[4] Fred Glover. Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[5] Michael Herdy. Application of the 'evolutionsstrategie' to discrete optimization problems. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, pages 188–192, London, UK, 1991. Springer-Verlag.

[6] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[7] Lorenz Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 186–194, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[9] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection.* The MIT Press, Boston, Massachusetts, 1992.

[10] Chang-Yong Lee and Xin Yao. Evolutionary programming using mutations based on the levy probability distribution. *IEEE Trans. Evolutionary Computation*, 8(1):1–13, 2004.

[11] Ko-Hsin Liang, Xin Yao, Yong Liu, Charles S. Newton, and David Hoffman. An experimental investigation of self-adaptation in evolutionary programming. In *Proceedings of the 7th International Conference on Evolutionary Programming VII*, EP '98, pages 291–300, London, UK, 1998. Springer-Verlag.

[12] Gisele L. Pappa and Alex A. Freitas. *Automating the Design of Data Mining Algorithms: An Evolutionary*

Computation Approach, volume XIII of *Natural Computing Series*. Springer, 2010.

[13] Riccardo Poli, John Woodward, and Edmund K. Burke. A histogram-matching approach to the evolution of bin-packing strategies. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages 3500–3507, Singapore, 25-28 September 2007. IEEE Computational Intelligence Society, IEEE Press.

[14] Carla Savage. A survey of combinatorial gray codes. *SIAM Rev.*, 39:605–629, December 1997.

[15] J.D. Schaffer and L.J. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R.K. Belew and L.B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[16] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

[17] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 4:67–82, 1997.

[18] John Woodward. Evolving turing complete representations. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 830–837, Canberra, 8-12 December 2003. IEEE Press.

[19] John Woodward. GA or GP? that is not the question. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 1056–1063, Canberra, 8-12 December 2003. IEEE Press.

[20] John R. Woodward. The necessity of meta bias in search algorithms. In *Computational Intelligence and Software Engineering (CiSE)*, Wuhan, 2010.

[21] 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 2009 Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600, Shanghai, China, 2009. ACM.

[22] John R. Woodward and James R. Neil. No free lunch, program induction and combinatorial problems. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 475–484, Essex, 14-16 April 2003. Springer-Verlag.