Pattern-Guided Genetic Programming

Krzysztof Krawiec
Institute of Computing Science
Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland
kkrawiec@cs.put.poznan.pl

Jerry Swan
Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland
jsw@cs.stir.ac.uk

ABSTRACT

Online progress in search and optimization is often hindered by neutrality in the fitness landscape, when many genotypes map to the same fitness value. We propose a method for imposing a gradient on the fitness function of a metaheuristic (in this case, Genetic Programming) via a metric (Minimum Description Length) induced from patterns detected in the trajectory of program execution. These patterns are induced via a decision tree classifier. We apply this method to a range of integer and boolean-valued problems, significantly outperforming the standard approach. The method is conceptually straightforward and applicable to virtually any metaheuristic that can be appropriately instrumented.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms, Design, Experimentation

Keywords

genetic programming, neutrality, program trace, MDL, Push

1. INTRODUCTION AND MOTIVATIONS

A Genetic Programming (GP) task is defined by a set of input data (fitness cases) and the desired program output for each of them. A GP algorithm is expected to infer the program only from these training examples. This can be very hard. When the task is challenging, it may be difficult for the search process to 'lift off', i.e. to make any progress. For instance, if exhibiting *any* nontrivial (non-constant) behavior leads to fitness deterioration, then it will locally pay off for a program in a population to return a constant output regardless of the input data.

This problem is not as severe in domains in which program performance on each test is assessed *quantitatively.* An example of such a domain is symbolic regression, where the error committed by a program on each fitness case is a continuous value, and thus can change gradually. Unfortunately, for more typical programing tasks, tests are *qualitative*: a program either passes a test or not. Synthesizing Boolean functions, sorting programs etc., belong to this category.

The key observation that motivates this study is that even programs which yield a completely useless output (meaning: output that does not match in any way the desired output) may produce some intermediate outcomes that can be relevant for solving the task. Take for instance the n-bit parity task. An initial part of program that correctly calculates the number of ones in half of the inputs (bits) provides a potentially useful capability. However, this partial outcome may be lost in the course of subsequent calculations carried out by the program, so that the ultimate output of the program is (for instance) the constant *true.* As a result, the program receives low fitness, and may not pass the selection process, so that its useful partial capability will be lost.

In standard GP, such intermediate results are not taken into account, because there is no singular means for assessing the quality of partial (intermediate) results produced by a program, given only the specification of the entire task. This is similar to a reinforcement learning setting, in which the agents are usually rewarded only *after* the entire task has been solved (i.e. once the goal state has been reached). For instance, an agent learning to play a game receives rewards only for complete games — there is no direct feedback about the quality of individual moves.

We postulate here that evaluation of partial program outcomes may yield better insight and is technically feasible. Since these partial results have ultimately been calculated from input data that came with the task and using an instruction set that also belongs to task formulation, they can represent useful pieces of knowledge of potential value in assembling a complete solution. In particular, the outcomes calculated for multiple inputs (fitness cases) can form some *patterns* that can be exploited to solve the task.

A skilled human programmer or mathematician is capable of discovering such patters and exploiting them to reach the goal, i.e. to design an algorithm that meets a prescribed specification. Moreover, humans are capable of defining what patterns are *desired* as an intermediate result. Consider designing an algorithm that calculates the median of a vector of numbers. A reasonable first stage of solving this task is sorting the elements of the input vector. Therefore, an intermediate memory state (pattern) containing sorted elements of the input vector is desired for this task (and anticipated by a human programmer).

Given that the distribution of problems considered in practice (including the problems approached with GP) is highly nonuniform, one might anticipate that some such patterns occur more frequently than others. In theory, it should be then possible to build an algorithm (system) which mimics the human programmer in that respect, i.e. in being capable of detecting the potentially useful patterns and rewarding the programs that produce them at intermediate execution stages. Hereafter, we use the term PANGEA (PAtterN Guided Evolutionary Algorithms) to describe this conceptual framework (as will be seen, there are no specific requirements that this approach be restricted to GP).

The specific approach proposed in this paper belongs to this framework and is based on the observation that, as a matter of fact, computational intelligence already provides automated pattern discovery tools that can capture the regularities in data. What we mean by this are the various machine learning and knowledge discovery algorithms. Thus, the key idea of the approach is to use a machine learning algorithm to search for such patterns by training it on the partial outcomes of program execution. Information on the resulting trained classifier (in particular its complexity and accuracy) is then used to augment the fitness function. If this approach is able to reveal meaningful dependencies between partial outcomes and the desired output, we may hope to thereby promote programs with the potential to produce good results in future, even if at the moment of evaluation the output they produce is incorrect.

2. THE BACKGROUND

We define a GP task as a set of n *fitness cases* (*tests*). A test is a pair (x_i, y_i), $i = 1, \ldots, n$, where x_i is the input to be fed into program, and y_i is the desired output. In general, x_is and y_is can be arbitrary objects.

Evaluation of an individual-program p assesses the quality of its mapping from inputs to desired outputs by applying it to all tests in turn. For each test, p is provided with input x_i and executed, returning output which we denote as $p(x_i)$. We say that *p solves test* (x_i, y_i) if it terminates and $p(x_i) = y_i$. The fitness of a program is simply the fraction of tests it does not solve, i.e.:

$$f(p) = \frac{1}{n}\left[n - |\{(x_i, y_i) : p(x_i) = y_i\}|\right]. \qquad (1)$$

Note that for the sake of simplicity, in this study we assume that only *perfect* matches between the desired output and the actual output are of interest. Therefore, only the equality relation in the space of program outputs needs to be defined (rather than more sophisticated relations like similarity). The fitness defined in this way is obviously minimized fitness, and a program p solves a task if $f(p) = 0$.

3. THE METHOD

The method proposed in this paper differs from standard GP only in the manner in which fitness is assigned to individuals. Therefore, given the formalization of the conventional fitness assessment in the previous section, we show here how we diverge from it. This process, illustrated in Fig. 1, can be split into four stages detailed in the following subsections.

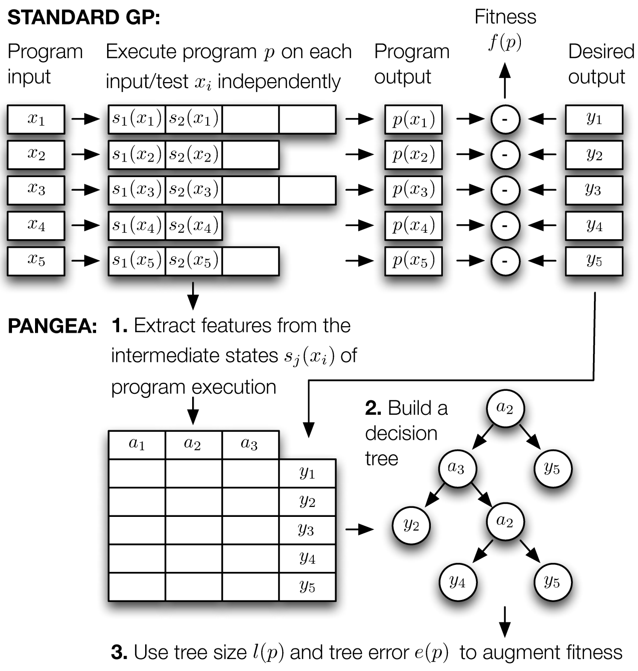
STANDARD GP:
Program input
Execute program p on each input/test x_i independently
Program output
Fitness $f(p)$
Desired output
x_1 x_2 x_3 x_4 x_5
$s_1(x_1)$ $s_2(x_1)$
$p(x_1)$ $p(x_2)$ $p(x_3)$ $p(x_4)$ $p(x_5)$
y_1 y_2 y_3 y_4 y_5
PANGEA: 1. Extract features from the intermediate states $s_j(x_i)$ of program execution
a_1 a_2 a_3
2. Build a decision tree
3. Use tree size $l(p)$ and tree error $e(p)$ to augment fitness

Figure 1: The outline of the proposed method. The top part illustrates the conventional GP evaluation procedure that results with fitness $f(p)$. The bottom part shows the components added by PANGEA.

3.1 Program trace acquisition

Our method works by gathering and exploiting information resulting from the intermediate stages of program execution. For this to be possible, we have to assume that (i) program execution is a stepwise process that can be paused, and that (ii) at such pauses, some information can be obtained from the execution environment. In virtually all genres of GP, these requirements are met by default. Concerning for instance the former, program is a discrete structure composed of symbols interpreted in certain order. These assumptions allow us to formalize certain notions, which we do below, abstracting from any specific form of GP.

Formally, let $s_j(p, x_i)$ denote the *state* of the execution environment when applying program p to test x_i after j steps have been executed. In side-effect free programming, relatively typical of tree-based GP, this would embrace a partial program (a subtree) and its outcome (the value returned by this subtree). For stateful execution environments, typical of programming languages with side-effects, this would be the state of the interpreter (its memory, instruction pointer, etc.). In other words, state is a 'snapshot' of the concrete process of program execution (i.e. pertaining to specific x_i). The sequence of states $(s_j(p, x_i) : j = 1, \ldots, l)$ resulting from entire program run forms its *trace*, where l is the number of steps, e.g., the number of instructions executed by a program, whether it terminated on its own or was forced to do so. The last state in a trace is the state of execution environment after program completion. We used a similar definition in our previous work on semantic analysis of program behavior [5].

In the proposed approach, the fitness assessment presented in Section 2 is accompanied by trace registration. As a result, we obtain a list of traces $(s_j(p, x_i) : j = 1, \ldots, l_i)$ for our n fitness cases (x_i, y_i). Let us note that, because in general program termination depends on its input, traces can vary in length, hence the index i in l_i.

3.2 Extracting trace features

The trace can be considered as a way of capturing program behavior. The key idea of PANGEA consists in analyzing the traces in search for patterns/regularities that reveal hidden qualities in the program and its partial outcomes. Various tools can be potentially used for this purpose, which we elaborate on in Section 7. In this paper, for the sake of simplicity, we aim at employing conventional machine learning algorithms that implement the paradigm of learning from examples and attribute-value representation [8].

To match the requirements of conventional machine learning algorithms with respect to input data, we transform the list of traces into a conventional machine learning dataset, where each row (example) corresponds to GP test, and every column is a feature derived in certain way from the traces. Each feature reflects certain syntactic or semantic information of all program traces at certain stage of program execution. Because the way the features are calculated from the states depends on state internals, this (and only this) stage of our approach varies depending on GP genre. In the subsequent section and experimental part, we detail it for Push [12], but in our pursuit of generality, we abstract from this technical detail in this section.

Crucially, we make this dataset define a *supervised* learning task. To this aim, we equip it with an extra column that defines a decision attribute. The value of this attribute is based on the desired output y_i of the corresponding test (x_i, y_i). In this paper we focus on GP tasks in which y_i is a single discrete value, in which case the decision attribute simply *is* y_i. This attribute will allow us to detect and capture patterns that are relevant for solving the task at hand.

3.3 Capturing patterns in features

The outcome of the above stage is a dataset of n examples, each described by the same number of features and labeled with a decision attribute that identifies the desired output. The purpose of the next step is to assess how useful these features are in predicting the desired output of the program. We achieve this by training a standard machine learning classifier, using the entire dataset as the training set. Specifically, we employ C4.5, a popular decision tree induction algorithm [9]. We anticipate that this choice is not critical; any inductive learning method could be applied here, as long as its outcomes can be analyzed with respect to the properties required in the next step.

We are interested in two characteristics of the induced classifier. Firstly, it has certain inherent *complexity*, which in the case of decision trees can be conveniently expressed as the number of tree nodes. Secondly, it commits a certain *classification error* on the training set, i.e. it erroneously classifies some of the examples. According to the *Minimum Description Length* principle (MDL [10]), the complexity of the mapping from the space of features onto the decision attribute may be expressed by summing the encoding length of the classifier (the 'rule') and the number of erroneously classified examples (the exceptions from the rule). In our context, this quantity tells us how easy it is to come up with the correct output of the program given the partial results gathered from its traces. This is the core element of the approach we propose here.

From another perspective, the role of the classifier is to complement the program's capability of solving the problem (i.e. producing the desired output value). If the features collected from the program trace form meaningful patterns, i.e. capture regularities in input data that are relevant to program output (decision class), then the induction algorithm should be able to build a compact tree that commits few classification errors (and thus has short description length).

To illustrate this, let us consider an extreme case of an ideal program p that solves the task, i.e. produces the desired output y_i for all tests ($y_i \equiv p(x_i)$). Since the final (post-execution) state is the last element of each trace, and the features are collected from trace elements, then one or more of the features in the training set will highly correlate with it. Assume for simplicity that the feature is perfectly correlated with y_i. In such a case, the induction algorithm will produce a decision tree composed of a single decision node (using that particular feature), and d leaves that correspond to the d decision classes. This is the smallest decision tree that can be built for such problem. This tree commits no classification errors, so the total length of the classifier encoding and the 'exceptions' will be minimal. In this scenario, the classifier does not augment the capabilities of the program.

If the program does not produce the desired output, the tree induction algorithm will be forced to make use of other features collected from the traces. The resulting tree will usually be larger than the minimal tree and/or commit some classification errors. In general, the less useful the features collected from the trace, the longer will be the total description length of the mapping. This shows that the total length of encoding expresses the usefulness of the intermediate results produced by the program (not only explicit results, reflecting, e.g. memory states, but also side-effects such as total number of execution steps). Most importantly, this length provides a meaningful way of assessing a program's 'prospective' capabilities, *even if the actual output of the program is completely useless from the point of view of the task being solved* (i.e. has nothing in common with the desired output).

3.4 Fitness calculation

Based on the above rationale, we define program fitness as follows. Let $l(p)$ denote the total number of tree nodes, and $e(p)$ the number of examples that are erroneously classified by the tree induced from the traces of individual-program p, $e(p) \in [0, n]$. In PANGEA, the fitness of p is the product of the standard fitness $f(p)$ ($0 \leq f(p) \leq 1$, Formula 1) and two terms that penalize the individual for, respectively, the complexity of mapping implemented by the induced classifier, and for the number of classification errors (exceptions from that mapping):

$$f_P(p) = f(p) \times \log_2(l(p) + 1) \times \frac{e(p) + 1}{n + 1}. \qquad (2)$$

The particular form of this equation results from preliminary experiments. The logarithm of the tree size is used so that model complexity is proportional to tree depth rather than to tree size. The $+1$ term in the second component (responsible for model complexity) prevents it from sinking to zero (the tree always has at least one node). The $+1$

term in the nominator of the third component (encoding length of exceptions) plays an analogous role. Therefore, the MDL-related components of fitness have an impact on program's fitness, but will never render it perfect. This can be achieved only by bringing the actual error committed by p on the tests, i.e., $f(p)$, to zero. Thus, a solution is optimal in PANGEA if and only if it is optimal w.r.t. the standard fitness definition (Formula 1).

4. RELATED WORK

There are numerous occasions in which the MDL principle has been used in GP. In most such cases, it has played a similar role to other machine learning techniques, i.e. as a means of controlling the trade-off between model complexity (sometimes referred to as *parsimony* in a ML context) and accuracy. In this spirit, Iba *et al.* [3] used it to prevent bloat in GP by designing an MDL-based fitness function that took into account the error committed by an individual as well as the size of the program. A few later studies followed this research direction (see, e.g. [14]).

By focusing mostly on the effects of program execution (the partial outcomes reflected in trace features) rather than on syntax, PANGEA can be seen as following the recent trend of semantic GP, initiated in [7]. Interestingly, it also resembles evolutionary synthesis of features for machine learning and pattern/image analysis tasks [6]. However, here the ML part serves only as a scaffolding for evolution; it is supposed to provide 'gradient' for evolution when the sole program output is not able to do so. The classifier is not the part of a solution.

5. IMPLEMENTING PANGEA WITH PUSH

In this section we explain how the process of feature extraction can be implemented for the programming language Push [12], which is also used in the experimental part of this paper. However, this choice is rather incidental, since program traces and their features can be easily acquired for other programming environments common in GP.

5.1 The fundamentals of Push

Push is a stack-based language, and its interpreter is equipped with a stack that holds the program to be executed, and a separate stack for each data type. For simplicity, we assume here that computation takes place in the Boolean and integer domains only, so only these three stacks (called EXEC, BOOLEAN and INTEGER in the following) will be of interest to us.

Push programs are lists of instructions that can be nested. To run a program, it is pushed onto the EXEC stack. Instructions are then popped from this stack and executed in turn. Every Push instruction manipulates one or more stacks. For instance, *integer.=* instruction pops two elements from the integer stack and pushes the result of their comparison atop the boolean stack. An instruction has no effect if the stack is too shallow. For details on this and other features of this programming framework, see [12].

When matching the notation introduced in Section 2 against Push, a task is a list of tests (x_i, y_i), where x_i determines the state of the execution environment (the stacks) before program execution, and y_i determines the analogous desired state after program completion. For inputs, 'determines' means in practice placing the elements of x_i onto the stacks of appropriate types. For outputs, y_i, usually a scalar value, specifies the desired value on the top of the stack of appropriate type.

For simplicity, we consider here only tasks with desired outputs y_i specified as scalars (atoms) of either integer or boolean type (depending on task type). We also assume that x_i is an arbitrary-length vector of integers that determines the initial state the INTEGER stack, and the output is the contents it leaves on the top the stack (INTEGER or BOOLEAN, depending on the type of desired output) after completion. We assume that a test has not been solved if the output stack is empty.

By way of example consider the task of verifying whether the input is a list of integers sorted ascedingly. In such task, each x_i would be a list of integers that would be placed on the INTEGER stack prior to program execution, while each y_i would be a Boolean value to be expected on the BOOLEAN stack.

5.2 Trace acquisition and feature extraction in Push

For Push, we implement the trace acquisition and feature extraction process introduced in general in Sections 3.2 and 3.1 as follows. We track program execution by pausing it after each instruction and storing the state of the interpreter. We do so until the program terminates or reaches a prescribed maximum number of steps. A state comprises all stacks that are relevant for the tasks considered in this paper, i.e., BOOLEAN, INTEGER, and EXEC. The list of states collected in this way for a specific test (x_i, y_i) forms the trace. Let us reiterate that, depending on input x_i, the number of executed instructions may vary, and hence the length of the resulting trace (see Fig. 1).

After tracing execution of the program on all n tests, we build a machine learning dataset by extracting features from selected elements of the history of program execution gathered in traces. Each trace gives rise to one example in the set. Starting from the last state of the trace, we iterate backwards over k last states, and from each of these states s_j we extract the following features:

- The sizes of all three stacks under consideration.
- The top elements of these stacks (if a stack is empty, we assume default values: zero for INTEGER stack, and a special value *Null* for the BOOLEAN and EXEC stacks).

In this process, we maintain the types of data: the integer-valued components of interpreter state (the sizes of all stacks and the top element of the INTEGER stack) translate into ordinal features, and the remaining components (the top elements of BOOLEAN and EXEC stacks) give rise to nominal (categorical) features.

Let us illustrate this process with a simple example, in which we focus exclusively on the INTEGER stack (feature extraction for the other stacks proceeds analogously). Assume we have three fitness cases and $k = 2$. A program was applied to these cases, and the top elements on the INTEGER in the corresponding traces were as follows:

- For test #1: `1 2 3 4`
- For test #2: `5 6 7`
- For test #3: `8 9`

The lengths of traces range from four to two, which tells us that the program terminated after executing different number of instructions for each fitness case. This is quite common for Push programs and may result from, e.g., the presence of loops. Because $k = 2$, two features (apart from the features reflecting the sizes of stacks) will be extracted from these traces: $a_1 = [3, 6, 8]$ and $a_2 = [4, 7, 9]$. As shown, they are 'aligned' with the ends of traces, because, in the end, a program may be expected to produce meaningful output in the final steps of its execution.

Nevertheless, it may be the case that meaningful patterns emerge also when stack contents are traced and converted into features by aligning them with respect to *beginning* of traces. This leads us to introduce also other features, which we build by merging into the feature vectors the top stack elements from the states that have the same index when counted from the beginning of programs. In our example, where $k = 2$, such forward-aligned features are: $a_3 = [3, 7, 0]$ and $a_4 = [4, 0, 0]$, where the zeroes are the default values.

In general, three stacks (each giving rise to two features, the top element and the size), and two methods of feature building total to $12 \times k$ features for horizon of length k. This set is supplemented with another feature: the number of execution steps t_i carried out by the interpreter for the ith test. Thus, the complete dataset forms a table of n rows (examples) and $12k + 1$ columns (features), $8k$ of which are ordinal and $4k+1$ nominal. Finally, we provide each example with a class label, which is simply the actual output of the program, i.e., z_i. Therefore, the number of distinct output values in the set of fitness cases is also the number of decision classes in the extracted dataset.

The dataset constructed in this way forms the result of feature extraction process, and is subject to further processing presented in Sections 3.3 and 3.4.

6. EXPERIMENT

The main objective of the experiment is to determine if pattern-based evaluation of individuals can be beneficial for evolutionary search. To this end, we compare the PANGEA approach to the control method of standard PushGP. More specifically, we compare GP running two fitness definitions, given by Formula (1) and Formula (2).

In all the test problems we consider, listed in Table 1, the input data is a table (one or more elements placed on the integer stack prior to program execution), and the desired output is a scalar value. In most problems (AEQ, CNF, ISO, MAJ), the task is to verify certain properties of the input (the output type is boolean). Two problems (COZ, MAX) are of a more arithmetic nature (the output type is integer).

The choice of this particular suite of tasks deserves justification. The tool we use for capturing patterns in traces and assessing their relevance is a decision tree inducer. As any machine learning algorithm, it brings with it certain *biases*, which in practice means that it prefers discovering certain types of patterns to others. Decision tree inducers like C4.5 are particularly good at discovering nontrivial patterns that engage *multiple* features. Therefore, it is suitable mostly for problems for which the traces are likely to give rise to multiple noncorrelated features, which, when used in combination, allow the tree to capture valuable input-output dependencies (patterns).

To illustrate this characteristic, let us shortly mention that in a preliminary experiment, we tested this approach on the problem of evolving a factorial function. In this problem, the tests (x_i, y_i) take the form $(m, m!)$, $m = 1, \ldots, n$. Because the input includes only one scalar datum m, many features collected from traces will be highly correlated with it (not mentioning the fact that most likely m itself will become one of the features). Secondly, because factorial is an injective function, there is one-to-one correspondence between the inputs x_i and the desired outputs y_i. As a result, each input (m) corresponds to a unique output $(m!)$, and in such a case building a tree that perfectly maps inputs to outputs becomes trivial. The classification error $e(p)$ is zero for most individuals in the run, and the MDL-related components of the fitness function cease to drive evolution in a useful direction.

For all tasks, the training set of cases contains all possible tables of lengths $2 \ldots m$, filled with all possible combinations of integers from the interval $[0, m-1]$. Thus, for the two values of n considered in this experiment, i.e., $m = 3$ and $m = 4$, the numbers of tests are $2^2 + 3^3 = 31$ and $2^2 + 3^3 + 4^4 = 286$, respectively. The only exception is the MAJ task, in which the input tables can contain only zeroes and ones; for this problem, there are $2^2 + 2^3 = 12$ and $2^2 + 2^3 + 2^4 = 28$ tests, respectively. We denote problem size by suffixing its name, e.g., MAX3.

Let us note two facts for which these tasks should be considered nontrivial. Firstly, the input tables are of variable size, so the evolved programs have to be general in this respect. In other words, evolving a simple sequence of instructions that individually and in turn pop the elements from the stack may be insufficient to solve a task, as such a program may reach the bottom of the stack too early for short tables, or ignore some elements on the stack for long tables. Secondly, because of that varying input size, we are not using the *type.input.k* instruction, which fetches the kth element from the input vector of the corresponding type (e.g., *integer.input.*2 fetches the third element of the input data). This instruction is present in the original Push to provide *permanent* access to particular input data for the entire program runtime; without it, the initial instructions of a program can quickly empty the stacks. In absence of the *input* instruction, programs cannot access input data directly, and must therefore solve the tasks only by stack manipulation.

We evolve a population of 1000 individuals for 100 generations, using evolutionary parameters typical for Push: viz. mutation probability 0.2, crossover probability 0.7, probability of reproduction 0.1, and a tournament size of 7. The Instruction set, presented in Table 2, consists of control structures, integer arithmetic, and boolean operations. The set of constants is limited to *true*, *false*, 0, 1, 2. To avoid interference of other factors, we do not use any additional mechanisms (*input* instruction, code stack), nor do we simplify programs (in contrast to many other studies on Push). Our implementation is based on a Push interpreter *PshGP* written by Jon Klein [4].

The maximal number of instructions in a newly-generated program is 20. However, the programs can grow much larger as a result of mutation and recombination.

The only difference between the standard Push (STD) and the proposed approach (PANGEA) is the manner in which individuals are evaluated. Fitness is minimized in both cases. In STD, it is Formula 1, and in PANGEA it is calculated according to Formula 2. PANGEA gathers fea-

Table 1: The definitions of tasks.

Symbol	Short explanation	Output type	Desired output
AEQ	All Equal	Boolean	*true* iff all elements on INTEGER stack are equal
CNF	Contains First	Boolean	*true* iff the element on top of INTEGER stack occurs also
COZ	Count Zeroes	Integer	The number of zeroes on the INTEGER stack
ISO	Is Ordered	Boolean	*true* iff the elements on the INTEGER stack are ordered non-descendingly
MAJ	Majority	Boolean	*true* iff more than half of the elements on the INTEGER stack are ones
MAX	Maximum	Integer	The maximum element on the INTEGER stack

Table 3: Success rate of compared methods (100 runs of each method and task).

Method	AEQ3	CNF3	COZ3	ISO3	MAJ3	MAX3	AEQ4	CNF4	COZ4	ISO4	MAJ4	MAX4	Rank
STD	0.93	0.17	0.47	0.20	0.46	0.07	0.32	0.00	0.02	0.00	0.20	0.00	3.67
PANGEA	**1.00**	**0.61**	**0.76**	**0.76**	0.97	**0.22**	**1.00**	0.00	0.04	0.00	**0.70**	0.00	**1.92**
PANGEA-E	**1.00**	0.18	0.42	0.46	**1.00**	0.00	**1.00**	0.00	0.02	0.00	0.67	0.00	2.50
PANGEA-M	**1.00**	0.32	0.40	0.67	**1.00**	0.22	0.99	0.00	**0.07**	0.00	0.50	0.00	4.04
PANGEA-X	0.97	0.12	0.46	0.03	0.46	0.01	0.42	0.00	0.00	0.00	0.09	0.00	2.88

Table 2: The instruction set.

boolean.and boolean.or boolean.xor boolean.= boolean.not boolean.dup boolean.flush boolean.pop boolean.rot boolean.stackdepth boolean.swap
integer.+ integer.- integer.* integer./ integer.% integer.< integer.= integer.> integer.dup integer.flush integer.pop integer.rot integer.stackdepth integer.swap
exec.= exec.dup exec.flush exec.pop exec.rot exec.stackdepth exec.swap exec.if exec.do*count exec.do*range exec.do*times
boolean.frominteger integer.fromboolean integer.erc true false

tures from the last three states of the interpreter ($k = 3$) before program termination (the upper limit on executed instructions is 150 instructions per test). This particular value of k was determined by preliminary experiments. The number of features then totals to $12k + 1 = 37$ (see Section 5). However, the tree does not have to use all of them (as a matter of fact, it is often the case that many of those features are constant). The decision tree is induced using an improved implementation of the C4.5 algorithm, known as J48 in WEKA package [2]. The algorithm builds an unprunned tree, i.e. no postpruning is employed.

We remind the reader that the decision trees are induced as part of an individual's evaluation process and do not become part of the resulting solutions. A tree is built to assess the quality of an individual, and is destroyed as soon as that assessment is completed.

6.1 The performance of methods

Given the nature of the benchmarks considered, we are interested only in perfect solutions, i.e. programs that solve *all* training tests. For this reason, success rate (i.e. the fraction of runs that yield perfect solutions), is our primary performance indicator.

The upper part of Table 3 reports the success rate of the two compared algorithms on all problems, based on 100 runs per each setup and benchmark. Overall, the proposed approach has a much higher success ratio, often several times greater than standard Push. The AEQ3 problem turns out to be very easy for pure Push, so it is unsurprising that PANGEA cannot do much better. As expected, the 'big' problems with four-element tables are substantially harder, and three of them (CNF4, ISO4, MAX4) have not been solved even once by either of the compared methods. Nevertheless, PANGEA outperforms STD on the remaining big problems.

One of the factors that can affect these results is the number of distinct desired outputs. For the AEQ, CNF, ISO, and MAJ, there are two desired values only: *true* and *false*. However, for the COZ and MAX tasks, there are, respectively, $m + 1$ and m distinct output values, where m is the size of the input table. This property of the problem determines the number of decision classes in the dataset used for tree induction.

6.2 The role of fitness components

The interplay of the components of the fitness function defined by Formula (2) can be expected to be very complex, so it is worth investigating the extent to which they all contribute to the performance of PANGEA. In other words: could it be that PANGEA fares better only because the MDL-related components introduce pseudo-random noise into the evaluation criterion and so help the evolution to escape stagnation?

To verify this, we ran a series of experiments with fitness based on Formula (2), but devoid of single components:

– PANGEA-E: formula (2) devoid of the program error (first) component,

– PANGEA-M: formula (2) devoid of the model length (second) component,

– PANGEA-X: formula (2) devoid of the classification error (third) component.

The performance of these methods is shown in the lower part of Table 3. As it turns out, all three components of Formula (2) are vital. Removal of any of them typically results in a lower success ratio than PANGEA. The only exceptions are AEQ3 and AEQ4, which apparently can be solved with near certainty even with such 'crippled' fitness functions. This result shows that for the fitness assessment scheme followed by PANGEA, it is important to take into account not only the complexity of the model (tree), but also the classification errors it commits.

To provide an overall perspective on these results, in the last column of Table 3 we report the average rank for each of the five setups. PANGEA clearly ranks as best. We validate the statistical significance of ranks using the Friedman test, the outcome of which is positive ($p \approx 0.0005$), i.e. at least one of the has success ratio significantly different from the others. Shaffer's post-hoc analysis procedure allows us to determine that PANGEA is significantly better ($p < 0.05$) than STD and PANGEA-M.

6.3 Dynamics of the search process

To illustrate the working principle of PANGEA, in this section we shortly analyze an exemplary evolutionary run for MAJ3 problem. That particular run lasted for 20 generations, in which an ideal solution has been found.

The chart in Fig. 2 summarizes the dynamics of that search process by plotting the important parameters of the best-of-generation individuals. For each such individual p, we show in parallel the essential components of PANGEA fitness (Formula 2): program fitness $f_P(p)$, the actual error $f(p)$ committed by the program on the fitness cases, the size $l(p)$ (the number of nodes) of the associated decision tree induced from trace features, and the classification error $e(p)$ committed by the tree. Because of different magnitudes, the former two are plotted against the left-hand ordinate axis, and the latter against the right-hand ordinate axis.

In the first five generations of the run evolution does not seem to make progress and all the parameters of the best-of-generation solutions remain unchanged. In this period, the decision trees comprise only three nodes and commit 8.4 classification errors (in general, the classification error of C4.5 is defined on a continuous scale), i.e. roughly eight of 12 fitness cases that constitute the MAJ3 task get misclassified. Let us emphasize that the fixing of this parameters may be purely incidental, i.e. it does not imply that the trees corresponding to these five best-of-generation individuals use the same features. The very small tree size in connection with relatively large number of classification errors suggests that the traces are scant in useful features, and that decision trees that accurately and succinctly map them onto the desired output cannot be built.

In generation 6 evolution manages to make progress and the best-of-generation fitness $f_P(p)$ substantially improves. The reason for this is however not a better match of program output with the desired output; just the opposite, $f(p)$ actually *deteriorates*. The cause for fitness improvement lies elsewhere: what actually happened is that the classification error committed by the tree $e(p)$ dropped from 8.4 to 3.5, while the size of the tree $l(p)$ increased from 3 to 5. Apparently, at some stage of execution of the best-of-generation program, a new pattern emerged in its traces (new w.r.t. to the best programs from previous generations). That feature (one or more) allowed building a more accurate decision tree that has only two more nodes. As a result, the product of the two MDL-related terms in Formula 2 decreased, and brought down the overall fitness $f_P(p)$ despite actual deterioration of the final output produced by the program as measured by the standard fitness $f(p)$ (Formula 1).

In the subsequent 7th generation the induction algorithm manages to build an even smaller tree, bringing its size back to three nodes, while maintaining the same classification error $e(p) = 3.5$. This results in further improvement of overall fitness, even given that $f(p)$ remains unchanged.

Generation 8 is the first time when program trace allows building a perfect decision tree ($e(p) = 0$). This is also the generation in which the mismatch between the program's output and the desired output ($e(p)$) is maximal in this run. This however does not prevent this program to outperform the best-of-run of the previous generation in terms of fitness $f(p)$.

The tree found in generation 8 is large: it comprises 21 nodes. Nevertheless, the very next generation sees a large decrease of this value: the tree has five nodes again, and classifies all examples perfectly. With minor fluctuations, this state is maintained in the remaining part of the run, while the evolution clearly works on the output of the program, gradually bringing $f(p)$ to zero.

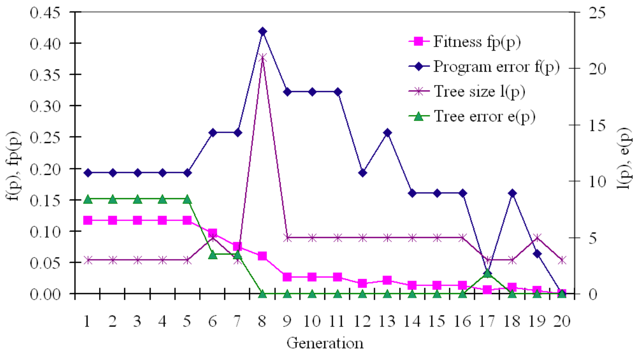

Figure 2: The changes of parameters of best-of-genration programs for an exemplary evolutionary run solving MAJ3 task.

This illustration confirms our rationale behind PANGEA's design, which we expressed in Section 3.3: the MDL-related components in fitness definition protect the individuals that discover meaningful patterns in the task and display those patterns in their behavior (written down in program traces). Without this mechanism, the individual like the that became best-of-generation 6 would most probably get lost in the selection process, as its error $f(p)$ is noticeably worse than the error of the best individual from the previous generation.

PANGEA enables even more insightful analysis, which we are forced to omit in this short contribution. For instance, it would be fascinating to see *what* are the features that led to the major transitions observable in runs like the one here, and also the code that generated it. Such investigations could help us for instance to determine whether, in the example above, evolution, after discovering the meaningful patterns around generations 5 to 8, kept the initial parts of programs roughly unchanged (to preserve the patterns), while working mostly on the final instructions (responsible for producing the final program output).

7. DISCUSSION

Our approach relies on a three-stage process: the recognition of patterns, the explicit description of such patterns in 'working memory', and the exploitation of these patterns to guide the search process. In the approach described here, the patterns are written down as decision trees, which in turn augment the fitness function. Of course, such classifiers might be incapable of capturing the specific form of patterns that are relevant for solving a particular task.

We have adopted the intentionally evocative term 'pattern' rather than feature in order to emphasize the potential for algorithmic sophistication and nontrivial computational effort in detecting and exploiting any aspects of the search process. We might therefore hope to mine for such patterns in problem description, genotype-to-phenotype mapping, solution-state trajectory, algorithm-state trajectory, or operator-sequence trajectory. It is clearly possible to combine these modalities, e.g. as in the current article which works with the combined trajectories of algorithm state, solution state and operator sequence.

We give here some possible 'use cases' for pattern detection that illustrate the potential for future work within this abstract framework:

1. One of the well-known properties of GP is that it is a 'model-agnostic' approach - assuming that we apply the function set {+,-,*,/,exp, log, sin, cos, tan} to a function that is actually linear, then we expect the transcendental non-terminals to be dropped and the resulting best program to be a linear expression.

However, the detection of linearity is merely implicit in GP - this activity is not explicitly directed by any features of the problem. GP may find out at some point of evolution that using a linear model is profitable, but does not 'sanction, mandate or suggest' it. The detection of linear relationships between features is of course a relatively trivial computational task, but the ability to explicitly recognize such patterns and inject the corresponding (sub)programs into the population can be viewed as a special case of a more general recognition-exploitation strategy [13].

2. Consider a training set T over a function $f(x, y)$, in which $f(x, y) = f(y, x) \forall f \in T$. Even in the absence of prior domain knowledge, a human presented with sufficient samples from the raw training data for this task would in all probability notice that the function is symmetric. Algorithms for online induction of symmetries in the context of search and optimization is given in [1, 11]. Such symmetry is of course a very specific case of (what is provisionally conjectured to be) an invariant of the function. Presented with such a regression task, a 'pattern-sensitive' human being would likely proceed to synthesize a function that respects this invariant. A partial synthesis might of course break the invariant, but activity is subject to the implicit understanding that ultimately restoring it is imperative. There are a number of ways in which one might attempt to incorporate the invariant into the subsequent search process, perhaps the most obvious of which is to add it as a soft constraint of the heuristic function.

8. CONCLUSIONS

PANGEA imposes a gradient on the fitness function of a metaheuristic (in this case, Genetic Programming) via a metric (Minimum Description Length) obtained from patterns detected in the trajectory of program execution (trace). The success of this method was demonstrated on a range of integer and Boolean-valued problems. The method is conceptually straightforward and applicable to virtually any genre of GP. The computational overhead it imposes in comparison to conventional fitness assessment is reasonable, because trace acquisition is done on the fly during program execution (which has to be carried out in regular GP as well). The only substantial additional cost results from training the classifier, however simple symbolic classifiers like C4.5 learn quickly.

It is increasingly being acknowledged that EC approaches are not a "one size fits all" approach. In particular, it is now understood that the initially popularized crossover and mutation operators do not enjoy the universality that were once ascribed to them. There is therefore in general a requirement in current EC practice for human ingenuity in the design and application of domain-specific operators, penalty functions etc. This work is an initial attempt to circumvent this requirement via the exploitation of regularity. In this case, regularity was determined by a statistical machine learning approach, i.e. the induction of a decision tree. In future work, we seek to extend the both the pattern detection methodology and the means by which it influences the subsequent search process.

Acknowledgments

K. Krawiec acknowledges financial support from grant no. 91507. J. Swan acknowledges support from grant EP/J017515/1.

9. REFERENCES

[1] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.

[2] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[3] H. Iba, T. Sato, and H. de Garis. System identification approach to genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 401–406, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[4] J. Klein. Psh - java implementation of the push programming language. Avalable from https://github.com/jonklein/Psh, 2010.

[5] K. Krawiec. On relationships between semantic diversity, complexity and modularity of programming tasks. In T. Soule et al., editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 783–790, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[6] K. Krawiec and B. Bhanu. Visual learning by evolutionary and coevolutionary feature synthesis. *IEEE Transactions on Evolutionary Computation*, 11(5):635–650, Oct. 2007.

[7] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic building blocks in genetic programming. In M. O'Neill et al., editors, *Genetic Programming*, volume 4971 of *LNCS*, pages 134–145. Springer, 2008.

[8] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[9] J. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, San Mateo, 1992.

[10] J. Rissanen. Modeling By Shortest Data Description. *Automatica*, 14:465–471, 1978.

[11] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 155(12):1539–1548, June 2007.

[12] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[13] J. Swan, J. Woodward, E. Özcan, G. Kendall, and E. Burke. Searching the hyper-heuristic design space. *Cognitive Computation*, February 2013.

[14] B.-T. Zhang and H. Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.