Population-Based Automatic Programming — Part 2: Performance On Some Example Problems

J.A. Schoonees[1] and E. Jakoet[2]

[1]Department of Electrical and Electronic Engineering, Manukau Institute of Technology, Auckland, New Zealand
jschoone@manukau.ac.nz
[2]Department of Electrical Engineering, University of Cape Town, Rondebosch, 7700 South Africa
ejakoet@eleceng.uct.ac.za

Abstract

A new automatic programming paradigm, called population-based automatic programming (PBAP) was proposed in Part 1 of this paper. The algorithm is in the style of genetic programming, but uses population-based incremental learning (PBIL) instead of the genetic algorithm. It produces a computer program by intelligent stochastic search of the space of all possible programs with a given set of functions and terminals. The new method was tested on four example problems from the original work on genetic programming by Koza: from optimal control, robotic planning, symbolic regression, and the Boolean multiplexer problem respectively.

1 Introduction

The genetic algorithm [Goldberg, 1993] forms the basis of two successful computational intelligence paradigms: Genetic programming [Koza, 1992] invented by Koza [1990], and population-based incremental learning (PBIL) [Baluja and Caruana, 1995] introduced by Baluja [1994]. Part 1 of this paper [Schoonees and Jakoet, 1997] proposed a new algorithm which is a combination of these two paradigms, called population-based automatic programming (PBAP).

PBAP is an experiment in automatic programming, using the program structures of genetic programming with the stochastic search engine of PBIL. It is an attempt to do 'genetic programming without the genetics'. Koza [1992, ch.7] demonstrated the power and efficiency of the genetic programming method with four introductory example problems, and we took these as benchmarks against which to compare our method.

2 The PBAP Algorithm

The PBAP algorithm is summarised here, and the interested reader is referred to Part 1 [Schoonees and Jakoet, 1997] for a more detailed description.

2.1 Program Representation

The generated program is represented as a dynamically linked tree of nodes. The nodes are labelled with function and terminal names represented by binary coded numbers, and are connected by branches. The first, topmost node is called the root. The number of branches originating from any function node depends on the number of inputs to that function (one for unary operators such as absolute value or square root, two for binary operators such as addition or multiplication, three for if-then-else-type structures, and so on). The function output is passed up to the input of the node immediately above it via a branch. Terminal nodes have no branches, and act as inputs to the program. A terminal may be a constant or an input variable which may change during execution of the program. The output of the root node is the output of the program as a whole.

2.2 Preparation of a Run

Before the algorithm can be run, a few decisions have to be made by the programmer. The functions and terminals of the program have to be selected according to a more or less subjective appraisal of the problem domain. This is a representation problem, and is generally problem-specific. Care must be taken that all function outputs return legal function input values, that is, that closure is achieved in the function set. For example, the protected-division function `DIV` should return an arbitrary value such as 0 if its divisor is 0.

Furthermore, a performance measure must be defined, which is a number or score assigned to a run of a trial program to reflect its performance. It may be thought of as the value of a cost function or objective function which takes as its input a program and produces a single number reflecting the performance of the program.

Some parameters for controlling the run have to be set, namely the maximum number of iterations, the number of trials in each iteration, the learning rate (and negative learning rate if used), the mutation rate or forgetting factor, and the maximum depth of the tree (an implementation limit). These parameters are described in more detail in Part 1. Unless otherwise stated, the internal parameters of the algorithm were set to the values shown in Table 1 for all the problems.

Table 1. Settings used for the various internal parameters.

Parameter	Value
Total iterations	100
Trials per iteration	500
Maximum tree depth	8
Positive learning rate	0.075
Negative learning rate	0.05
Forgetting factor	0.005

2.3 The Probability Vector

In PBIL, accumulated knowledge about the search space is encapsulated in the so-called probability vector (PV). It is a vector of real-valued elements PV_i, $i = 1, 2, \ldots, N$ in the range $0 \leq \mathrm{PV}_i \leq 1$, where each element represents a probability. Associated with the PV is a sample vector (SV) of the same length, or rather, a population of several such SVs. Each SV represents a trial program, and is the concatenation of all the node labels in the program. The SV elements are therefore binary digits (bits), in the range $\mathrm{SV}_i \in \{0, 1\}$, $i = 1, 2, \ldots, N$. The length of the PV is set by the size of the program: it is the total number of node labels times the number of bits per label.

There is a one-to-one correspondence between each SV bit and its associated PV element: SVs are generated randomly by assigning a 0 or a 1 to each bit position according to the probability in the corresponding PV element. More precisely, the i-th element of the PV is the probability that the i-th bit in any SV of the current iteration will be a 1. Initially, the PV elements are all set to 0.5, so that the first iteration of SVs is generated with an equal probability of a 0 or a 1 in each bit: the run starts with a population of random programs.

Each SV is evaluated by testing it in some way (typically by running it as a trial program and observing its performance) and assigning to it a number which is its performance measure. After all the SVs have each been assigned a performance measure, the best one is used to update the PV by biasing its elements slightly in favour of the best SV. Other operations may also be performed on the PV, such as negative learning and mutation.

The updated PV is used to generate the next iteration of SVs. After several iterations (anything from a few tens to a few thousands), the PV's elements converge to near either 0 or 1, and the PV as a whole becomes, after saturating its elements to the nearest 0 or 1, the found solution.

3 Testing the Algorithm

An attempt was made to evaluate the PBAP algorithm by running it on the example problems used by Koza [1992]. We implemented:

1. the cart-centring problem;
2. the artificial ant;
3. simple symbolic regression; and
4. the 6-multiplexer.

3.1 Optimal Control

As an example of optimal control, Koza [1992] evolved a control strategy for the cart-centring or isotropic rocket problem. The control program must switch a force of fixed magnitude (bang-bang control) so as to bring a cart moving along a frictionless track to rest at a designated target point in minimal time. The cart starts from a random position on the track and with random velocity. The inputs to the control program are the state variables: position $x(t)$ and velocity $v(t)$, and its output is the direction in which the force is to be applied. The set of terminals therefore comprises `X` and `V`, the components of the instantaneous state vector.

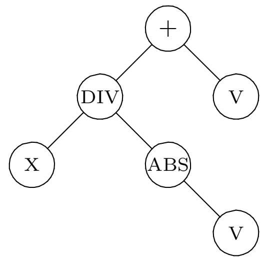

Fig. 1. The best-of-run solution to the cart-centring problem.

The set of functions were chosen by Koza to be `+` (the addition operator), `-` (the subtraction operator), `*` (the multiplication operator), `DIV` (the protected division operator), `GT` (the signum function), and `ABS` (the absolute value operator). All these functions take two arguments, except `ABS`, which takes one.

A known optimal solution to the control problem, expressed as a LISP program, is (GT (* -1 X) (* V (ABS V))). Koza [1992, p.122], using the genetic algorithm, obtained the following optimal solution in 33 iterations: (- (- (* (+ (* (ABS V) -1) (* (ABS V) -1)) V) X) X). A typical run of PBAP obtained an optimal solution in 31 iterations, which was also the final solution of the converged PV: (+ (DIV X (ABS V)) V). This solution is illustrated in fig. 1.

Fig. 2 shows the performance curves of a typical run of PBAP. Also shown, in fig. 3 is the convergence curve of the same run, which shows the state of convergence of the PV at each iteration. Convergence is defined as

$$C = \frac{2}{N} \sum_i \left| \mathrm{PV}_i - \frac{1}{2} \right| \qquad (1)$$

which is the mean difference between 0.5 and the PV elements. These curves may be compared with the corresponding genetic programming fitness curves in [Koza, 1992].

3.2 Robotic Planning

Koza's second example of genetic programming [Koza, 1992] involves programming an artificial ant to navigate a marked path by finding a trail of food pellets on a toroidal

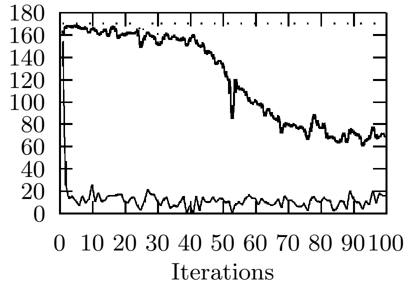

Fig. 2. The performance curves of a typical run of the cart-centring problem. From the top, they show the worst, average, and best normalised performance measures obtained for each iteration. A low normalised performance measure corresponds to better performance.

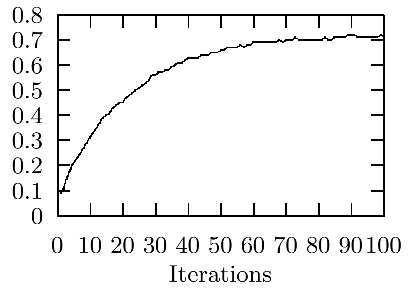

Fig. 3. The convergence curve of a typical run of the cart-centring problem.

grid. The pellets are placed along the so-called "Santa Fe trail", leaving small gaps and introducing sideways steps that become bigger the further the ant progresses along the trail. The ant is placed at the start of the trial and can only 'see' what is directly in front of it: food, or no food. The ant is capable of three primitive movements:

1. Turn right through 90°.
2. Turn left through 90°.
3. Move forward one square (and eat whatever food may be there).

The ant's goal is to traverse the whole length of the trail, and eat all the food pellets, in a reasonable amount of time.

The problem was originally cast as a finite-state automaton design, but was reformulated by Koza [1992] as a genetic programming problem. The three primitive movements are effected as side effects of three corresponding terminals `RIGHT`, `LEFT` and `MOVE`.

We adopted Koza's function set which comprised three functions, `IF-FOOD-AHEAD`, `PROGN2` and `PROGN3`. The `IF-FOOD-AHEAD` function is a conditional branching operator and has two arguments. It evaluates the left-hand argument if there is food directly in front of the ant, else it evaluates the right-hand argument. The `PROGN2` and `PROGN3` functions are connectives with two and three arguments respectively: they evaluate their arguments from left to right as unconditional sequences of steps. Note that none of these functions explicitly return an output value: their only outputs are in the side effects (changes in the location or orientation of the ant, and presence of pellets on the trail) of the evaluation of terminals in their arguments.

Koza [1992, p.147] obtained a perfect run of the

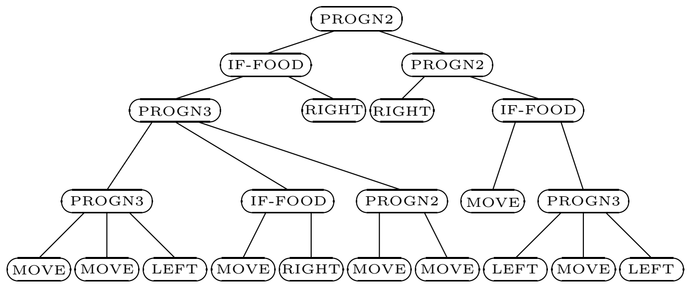

Fig. 4. The best-of-run solution to the artificial ant problem, obtained after 98 iterations.

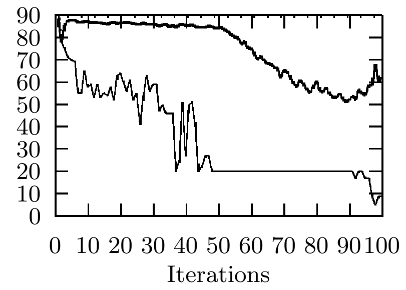

Fig. 5. The performance curves of a typical run of the artificial ant problem.

artificial ant on generation 21. PBAP could not find a perfect solution, but found a suboptimal solution (fig. 4) with a performance measure of 80, 9 short of the optimum value of 89. The performance curves of a typical run of the artificial ant problem are shown in fig. 5. The convergence curve is similar to that of fig. 3.

3.3 Symbolic Regression

In symbolic regression, a sampling of numeric points on a curve is provided, and the goal is to identify the function which most closely matches the samples. The process may also be called symbolic function identification. The sample curve used was $f(x) = x^4 + x^3 + x^2 + x$, which is illustrated in fig. 7. Twenty points x were chosen at random from the real interval $[-1, +1]$ to provide twenty pairs of coordinates $(x, f(x))$. The only terminal was the independent variable x, and the set of functions comprised `+`, `-`, `*`, `DIV`, SIN, COS, EXP, and RLOG (the protected log function).

Koza [1992, p.162] found a solution which was equivalent to the optimal $f(x)$ in generation 34 of his genetic programming run of the problem: (+ X (* (+ X (* (* (+ X (- (COS (- X X)) (- X X))) X) X)) X)). PBAP did not discover the exact solution, but found a close fit after 44 iterations: (* (X EXP (DIV X (COS (COS (COS (SIN (X)))))))) The function is illustrated in fig. 7. The PBAP solution is also plotted, but is indistinguishable from the ideal curve on this scale of the graph. The set of performance curves for the run are shown in fig. 8.

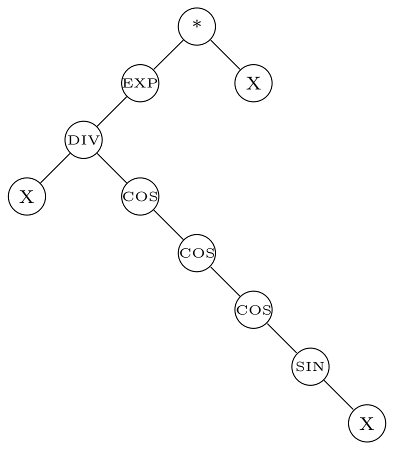

Fig. 6. The best-of-run solution to the symbolic regression problem.

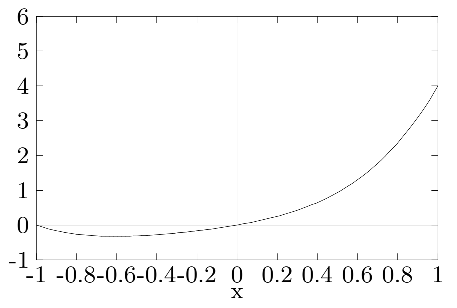

Fig. 7. The curve used in the symbolic regression problem, with the indistinguishable PBAP solution superimposed.

3.4 Boolean 6-Multiplexer

The Boolean 6-multiplexer is familiar to computer engineers as the function in which one of four binary-valued 'data' inputs is selected according to the value of two binary 'address' inputs. The aim is to derive the 6-multiplexer function using the four data inputs D0, D1, D2, and D3, and the two address inputs A0 and A1, as the terminals. The function set comprises the primitive Boolean operators NOT, AND and OR, as well as the IF-THEN-ELSE operator with three arguments.

A known solution to the 6-multiplexer problem is: (IF A1 (IF AO D3 D2) (IF A0 D1 D0)). PBAP found an equivalent correct solution, using 2000 trials per iteration: (IF (A0 (IF A1 D3 D1) (IF (A1 D2 D0))).

The performance curves are shown in fig. 9. A correct solution was found on the 43rd iteration on this run, and most runs seem able to find a correct solution. The convergence curve is similar to the cart-centring problem's curve.

4 Discussion

The artificial ant was the only problem of the ones tested which did not yield a good solution to PBAP. We believe that the characteristics of the problem may indicate a weakness in the way PBAP searches the space of possible solutions. Possible reasons for this weakness were proposed in Part 1. It appears that the artificial ant problem requires a larger tree than what suffices for easier problems. If the function and terminal sets have relatively few functions with many branches, and relatively more terminals, the average generated tree size may be too small to explore the space of more structurally complex solutions. A quick calculation of the average number of

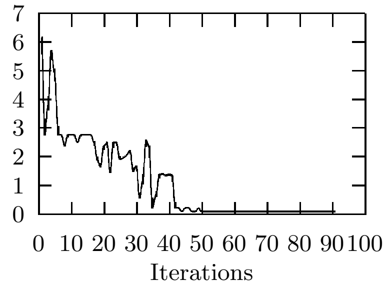

Fig. 8. The performance curves of a typical run of the symbolic regression problem.

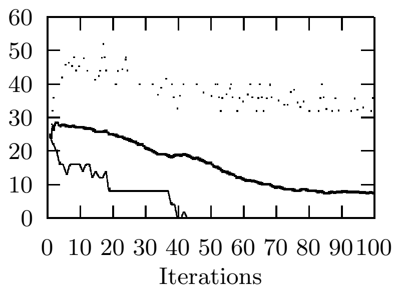

Fig. 9. The performance curves of a typical run of the 6-multiplexer problem.

branches per node seems to support the suggestion that the artificial ant problem could do with a larger average tree size. This problem was neglected in this work, and requires further work.

For the other problems, PBAP consistently found good or optimal solutions in a few tens of iterations. It seems to have inherited most of the power of genetic programming and the simplicity of PBIL. The algorithm tested here is a first attempt. Further investigation may well improve its performance.

References

[Baluja and Caruana, 1995] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," in *Proceedings of the Twelfth International Conference on Machine Learning*. 1995, Morgan Kaufmann Publishers.

[Baluja, 1994] S. Baluja, "Population-based incremental learning," Technical report CMU-CS-94-163, School of Computer Science, Carnegie Mellon University, June 1994.

[Goldberg, 1993] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1993.

[Koza, 1990] J.R. Koza, "Genetic programming," Technical report STAN-CS-90-1314, Computer Science Department, Stanford University, 1990.

[Koza, 1992] J.R. Koza, *Genetic Programming*, MIT Press, 1992.

[Schoonees and Jakoet, 1997] J.A. Schoonees and E. Jakoet, "Population-based automatic programming — Part 1: Description of the algorithm," in *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97)*. 1997, IEEE Computer Society Press.

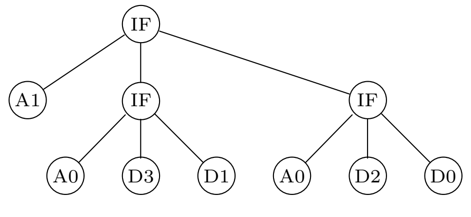

Fig. 10. The best-of-run solution to the Boolean 6-multiplexer problem.