Population-Based Automatic Programming — Part 1: Description of the Algorithm

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 is proposed, in the style of genetic programming, but using population-based incremental learning (PBIL) instead of the genetic algorithm. A computer program, which solves some given problem, is generated by intelligent search of the space of all possible programs with a given set of functions and terminals. The method starts with a set of random programs which are evaluated for their performance. The performance measure of the best (and optionally the worst) performing program is fed back to the stochastic program generator until the system converges on a program which is either a correct solution to the problem or a good suboptimal solution. The implementation uses a variable-length probability vector which is distributed across a dynamically linked program tree. The new method provides interesting insights into the working of PBIL itself. Some shortcomings are identified and improvements are proposed.

1 Introduction

The population-based incremental learning (PBIL) paradigm was introduced by Baluja [1994], as a powerful new 'soft computing' optimisation method based on the genetic algorithm [Goldberg, 1993] and hill-climbing optimisation. PBIL retains most if not all of the genetic algorithm's power [Baluja, 1995], yet is much simpler to implement. It is well suited as a domain-independent 'weak' method because its internal parameters are few and of low sensitivity.

This paper describes an experiment in automatic programming, and was inspired by the genetic programming paradigm of Koza [1992]. Koza uses the genetic algorithm to 'breed' computer programs from a random start. We attempted the same thing as Koza, but by adapting PBIL to the task instead of the genetic algorithm.

A crucial step in the adaptation of PBIL to automatic programming was extending PBIL to variable-length 'chromosomes' or sample vectors. In genetic programming, this is less of a problem because the crossover operation of the genetic algorithm lends itself naturally to generating individual chromosomes which are not of any fixed length. PBIL sample vectors are generally of fixed length.

2 Program Representation

The result of a run of the PBAP algorithm is a computer program. The program is represented as a tree formed from nodes and branches, as illustrated in fig. 1. Each node represents either a function with a fixed number of arguments, or a terminal which has no arguments. The arguments to a function are supplied through branches from nodes lower down in the tree, and the value of the function is passed upward through the branch connected at the top of the node. The topmost node is (paradoxically) called the root of the tree and returns the output of the program as a whole. The terminals form the end nodes of the lowest branches and provide the program with either constants or variable inputs which may change during the execution of the program. Many, if not most, procedural programs and functions can be represented in this form.

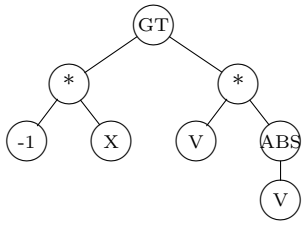

Fig. 1. Example of a program represented as a tree of nodes joined by branches.

Fig. 1, for example, evaluates the function

$$f(x, v) = \text{sgn}(x + v|v|) \tag{1}$$

which is the solution to a simple problem in optimal control [Schoonees and Jakoet, 1997]. This program can be regarded as a sample from the search space of all possible programs.

A program is represented by a variable-length vector of binary digits (bits). It is called a sample vector (SV), because it represents a single sample from the search space. The SV is a concatenation of a number of subvectors, each subvector encoding the label of a function or a terminal. In practice, each subvector is stored in the node it labels. The concept of a single composite SV is useful, even if it is actually stored in a distributed fashion throughout the program tree.

Table 1 shows the labels of the set of functions and terminals from which the program of fig. 1 was generated. In this version of the algorithm, the binary encoding of the labels is arbitrary (except that one should perhaps

Table 1. An example of functions and terminals that may be used in a particular run of the algorithm, and their binary encoded labels.

Name	Type	Description	Label
NEG1	terminal	Constant -1	0000
X	terminal	Position	0001
V	terminal	Velocity	0010
ABS	1-input function	Absolute value	1000
PLUS	2-input function	Addition	1001
MINUS	2-input function	Subtraction	1010
TIMES	2-input function	Multiplication	1011
DIV	2-input function	Protected division	1100
GT	2-input function	Greater-than	1101

not use more bits than is necessary). The order of the subvectors within the SV is also arbitrary, although the relative position of each subvector is recorded in the program tree. As the program tree grows larger, more subvectors are appended to the SV for the additional functions and terminals needed.

The data structure of a node, shown in fig. 2, comprises a few subvectors, which are the local fragments of some distributed main vectors. It has already been mentioned that the SV is one such main vector that is stored in a distributed way: when a new node is created it effectively extends the SV by one subvector. Each subvector is as long as the number of bits in a node label. The most important other subvector is the subvector of the probability vector (PV). In addition, some auxiliary vectors are needed to store intermediate results, such as a 'best' subvector, a 'worst' subvector, and an optional 'best-ever' subvector. Each new node also contains a list of branches to nodes one level lower down in the tree: in a new node this branch list is cleared.

3 The Algorithm

The PBAP algorithm is summarised in fig. 3. It runs through a number of iterations before producing a result; from about 50 to 500 iterations may be needed. The search space is at first explored randomly, by randomly sampling the space and evaluating the performance of each resulting SV. This provides information about the likely location of optima, which is collected in a probability vector (PV). The PV is a real-valued vector of the same length as the SV, and its elements have a one-to-one correspondence with those of the SV. Each element of the PV represents a probability $0 \leq \mathrm{PV}_i \leq 1$, $i = 1, 2, \ldots, N$ where N is the current (variable) length of the SV. All its elements are initialised to 0.5. As

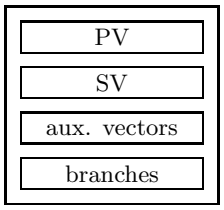

Fig. 2. Schematic diagram of the data structure of a node.

```
best_ever_measure  <-  -infinity
Repeat for the desired number of iterations:
    best_measure  <-  -infinity
    worst_measure  <-  +infinity
    Repeat for the desired number of trials:
        Generate a new sample vector (SV).
        measure  <-  SV perf. evaluation
        If measure  >  best_measure:
            Record SV as best of iteration.
            best_measure  <-  measure
        If measure  <  worst_measure:
            Record SV as worst of iteration.
            worst_measure  <-  measure
    If best_measure > best_ever_measure:
        Record SV as the best found so far.
        best_ever_measure <- best_measure
    Update the probability vector (PV).
    Output the best of the iteration.
Output the best ever SV.
Output final PV saturated to nearest 0 or 1.
```

Fig. 3. A summary of the PBAP algorithm. It is assumed that higher performance measures indicate better performance.

the run progresses, the accumulated information in the PV is increasingly exploited to focus the search on the most promising areas, until the elements of the PV have converged on asymptotic values near either 0 or 1.

Every time a new trial is needed, a new SV is generated from the PV as follows:

$$\mathrm{PV}_i \overset{\mathrm{SV}_i=1}{\underset{\mathrm{SV}_i=0}{>}} R_i, \quad R_i \in [0,1), \;\; i = 1, 2, \ldots, N \qquad (2)$$

where R_i is a random number uniformly distributed in the range $[0, 1)$. A new random number R_i is generated for each bit SV_i. If the corresponding element PV_i is greater than R_i, then $\mathrm{SV}_i = 1$, else $\mathrm{SV}_i = 0$. The SV therefore samples the PV such that the probability of generating a 1 in a given element of the SV is given by the corresponding element of the PV.

In practice, the SV is generated by recursively generating its subvectors from the root downward. As each subvector is created, the number of input branches of its node becomes known. If some of these branches and their associated nodes do not yet exist, they are now added to the tree 'on the fly'.

Within each iteration, many SVs are produced; a number of about 200 to 2000 is typical. The SVs are also called trials or trial solutions, because each one is evaluated by executing the program it represents to produce a performance measure. The performance measure 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 its performance.

Each SV is evaluated by passing it to an interpreter which runs the program. The interpreter is problem-specific and has to provide the necessary inputs to the program and monitor its output. Depending on the problem, a single run of the program may be sufficient to evaluate its performance, or several runs with sampled

or random input data may be needed, or an extended simulation of some system could be executed. The interpreter is typically encapsulated in a cost function which uses the program output (or other side effects) to generate a performance measure. In the majority of cases, performance evaluation consumes most of the processing time of a PBAP run.

4 Updating the Probability Vector

In PBIL, and PBAP, accumulated knowledge about the search space is encapsulated in the PV. Initially, the PV elements are all set to 0.5, so that the first iteration of trials is generated with an equal probability of a 0 or a 1 in each bit of the SV.

The best-performing SV of the iteration is used to implement so-called positive or reinforcement learning. All the elements of the PV are slightly biased in favour of the best SV:

$$\mathrm{PV}_i \leftarrow (1-\mathrm{LR})\cdot\mathrm{PV}_i+\mathrm{LR}\cdot\mathrm{SV}_i^B, \quad i=1,2,\ldots,N \quad (3)$$

where PV_i is the i-th element of the PV, LR is the learning rate, and SV^B is the best performing trial of the current iteration.

Similarly, the worst-performing trial can optionally be used to move the PV slightly away from that SV, a process called negative learning or punishment. Negative learning is implemented only in those elements PV_i in which the corresponding elements of the best and worst trials differ. In those positions i where the best and worst trials agree, this step is skipped:

$$\mathrm{PV}_i \leftarrow (1-\mathrm{NLR})\cdot\mathrm{PV}_i+\mathrm{NLR}\cdot\overline{\mathrm{SV}_i^W} \quad (4)$$

where NLR is the negative learning rate, SV^W is the worst-performing trial of the current iteration, and $\overline{\mathrm{SV}_i^W}$ is the Boolean inverse of the i-th element of SV^W.

Finally, an operation corresponding to mutation in the genetic algorithm is performed, by implementing a slight 'forgetting' in the PV: all its elements are biased very slightly towards 0.5:

$$\mathrm{PV}_i \leftarrow \mathrm{PV}_i-\mathrm{FF}\cdot\left(\mathrm{PV}_i-\frac{1}{2}\right), \quad i=1,2,\ldots,N \quad (5)$$

where FF is a 'forgetting factor' [Greene, 1996]. This has the effect of centring 'undecided' PV elements until such time as a strong bias is provided by either positive or negative learning. Baluja [1994] uses the same mutation operator in PBIL as is used in the genetic algorithm for maintaining genetic diversity, but we agree with Greene [1996] that the 'forgetting' operator achieves the same goal in a way that belongs more naturally in the philosophy of PBIL.

The updated PV is used to generate the trials in the next iteration of the run. 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. This converged solution may sometimes not be the same as the best-performing trial found in the course of the run. We regard such an event as a sign that either the PBAP algorithm is not sufficiently powerful to find a global optimum in the search space, or that poor settings of the various parameters were used in the run.

Table 2. Typical settings for the internal parameters.

Parameter	Value
Total iterations	100
Trials per iteration	1000
Maximum tree depth	10
Positive learning rate	0.1
Negative learning rate	0.05
Forgetting factor	0.005

5 Evaluation of the Algorithm

The PBAP algorithm was tested by running it on the four introductory example problems used by Koza [1992, ch. 7]. We implemented:

1. the cart-centring problem (optimal control);
2. the artificial ant (robotic planning);
3. simple symbolic regression; and
4. the Boolean 6-multiplexer.

A companion paper to this one [Schoonees and Jakoet, 1997] gives the detailed results of these tests. We are limited here to some general observations:

In conventional genetic programming, the initial population of program trees is important because it contains the raw material from which all subsequent individuals are formed. Koza [1992] goes to considerable lengths to ensure that the first generation of program trees has a large variety of different sizes and shapes of trees.

In contrast, individual trials in PBAP are generated stochastically from the PV. There is no direct control over the size and shape of the tree, except in the probabilities of generating terminals and functions with various numbers of arguments. The relative numbers of each type of node (terminal, 1-input, 2-input, etc) determine their relative abundance and hence indirectly the size and shape of the tree. This fact has been somewhat overlooked in this work so far, in that we have blindly adopted Koza's sets of functions and terminals. It is clear that the algorithm described here does not do an unbiased search of the search space, but is influenced by whatever the average size of the tree happens to be.

For some problems this does not seem to matter, but in others the average tree size seems to be too small to give consistently good results. A rough calculation of the average number of branches per node (the average branching factor) seems to support the suggestion that such problems could do with a larger average tree size. To achieve this, one might for example prime the initial values of the PV in such a way that functions are favoured over terminals; or perhaps make the learning rate an increasing function of depth, to compensate for the lower probability of using nodes at deeper levels.

6 The Rate of Convergence

The underlying program structure may be thought of as a root node with an infinite number of branches (function

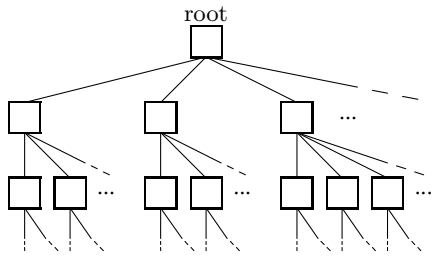

Fig. 4. The underlying infinite program tree. Every node has an infinite number of branches, each of which leads to another node one level down. In practice, the maximum depth is limited, and nodes are only created when needed.

arguments), each branch leading down to a node which may itself have an infinite number of branches. Fig. 4 shows a schematic representation of this idea. The PV is therefore, conceptually at least, of infinite length.

This conceptual structure is easy to implement, because it is only necessary to create (dynamically, as instances of programmatic data structures) the parts of the tree which are needed. Subsequent trials may or may not use the newly created branches and nodes, depending on the number of input arguments required by functions higher up in the tree. Any trial program fills only a small part of the space of all possible programs.

Two questions now arise in connection with the practical implementation of this concept:

1. In a newly created node, what should its fragment of PV be initialised to?

2. If an existing node is not used by a trial program, should its PV subvector be updated (learning) along with the rest of the PV in the currently used part of the tree?

For the moment, our algorithm initialises new subvectors of the PV to 0.5 when new nodes are created, because it is felt that this best reflects the fact that no information is available as yet about that part of the search space. Alternatively, one might randomly update the new subvector as many times as the number of iterations so far, in order to simulate the existence of the node since the beginning of the run.

Our algorithm answers the second question by updating the PV across the entire existing tree, whether all of it is occupied by a program or not. This assumes that every SV is also generated across the entire existing tree, and that no attempt is made to save on computation by only generating SV subvectors in those nodes that are needed as input arguments to nodes higher up.

This decision was significant, because when the PV subvectors in currently unused subtrees were left static, PBAP showed a strong bias toward converging on smaller 'solutions', even when it had discovered a better, but larger, sample along the way. When the whole tree is updated on every iteration, irrespective of which portions are in use or not, convergence is less biased toward small solutions, and there seems to be a better probability of discovering and converging on the optimal solution.

This effect may be explained by regarding any particular element of the PV as executing a random walk, starting at 0.5 and ending near one of the asymptotic convergence values, under the influence of the autoregressive update rule. In the absence of consistent feedback from the cost function with regard to this particular element, it nonetheless gets updated along with the rest of the PV. Even the slightest earlier learning will tend to 'grow' (if the forgetting factor is not set too high). This helps to drive the search in the direction of better solutions. If, instead, unused elements of the PV are not updated but are simply left at their current values, the rate of convergence becomes uneven across the tree, and convergence tends to be slower across the whole PV.

7 Conclusion

In the light of the discussion above, PBAP may provide an interesting insight into how PBIL's probability vector converges. PV elements that are not currently receiving feedback from the cost function, because of the nature of the search space in the area that is currently being sampled, still retain whatever little knowledge they had gained and are ready to reinforce any new impetus in the same direction, or resist change in the opposite direction.

PBAP is also an interesting experiment in 'genetic programming without the genetics'. An apparent weakness has been found in the way the algorithm converges on solutions in the search space, and as a result some modifications were proposed. Further investigation may give better results. The current PBAP algorithm already seems to inherit much of the power of genetic programming.

References

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

[Baluja, 1995] S. Baluja, "An empirical comparison of seven iterative and evolutionary function optimization heuristics," Technical report CMU-CS-95-193, School of Computer Science, Carnegie Mellon University, September 1995.

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

[Greene, 1996] J.R. Greene, "Population-based incremental learning as a simple versatile tool for engineering optimisation," in *Proceedings of the Evolutionary Computing and Applications Conference (EvCA96)*. Moscow, 1996.

[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 2: Performance on some example problems," in *Proceedings of the Fourth International Conference on Neural Information Processing (ICONIP'97)*. 1997, IEEE Computer Society Press.