Optimisation Using Population Based Incremental Learning (PBIL)

Evan J Hughes*

1 Introduction

The Population Based Incremental Learning (PBIL) algorithm is a simple stochastic optimisation technique that can be applied quickly to a wide range of problems. The technique's main area of application is for problems that are too multi-modal or discontinuous for gradient or simplex methods, but don't warrant a full evolutionary algorithm solution. PBIL has been shown to outperform conventional deterministic and stochastic optimisation techniques on a wide range of problems [1, 2] and yet is simple to code.

This paper describes a practical approach to applying the PBIL algorithm to optimisation problems. First the operation of the algorithm is described and then guidelines for tuning the algorithm are presented. An example implementation is given and a method for reducing the processing burden of the algorithm is detailed. An example *MATLAB* routine is included to demonstrate the simplicity of the algorithm.

2 Algorithm operation

The PBIL algorithm is a stochastic guided search process that obtains its directional information from the previous best solutions. The algorithm was first described in 1994 [1] and has been improved recently [3]. The PBIL algorithm presented here has three control parameters; *population size* (p), *learning rate* (l), and *search rate* (s). Unlike many other stochastic optimisation algorithms, this algorithm terminates automatically when the process has converged on a single solution. The problem parameters are represented as a binary chromosome of total length b bits. Each variable is coded in a binary form and then concatenated to any previous parameters to form a single chromosome.

A *prototype vector* ($\mathbf{P}$) is used to bias the generation of bits in a population of chromosomes. The prototype vector has b elements, one for each bit location. At each location, the prototype vector holds the probability that the corresponding bit is a '1'. Each location is initially set to 0.5 which corresponds to unbiased bit generation. A population of candidate solutions is generated using the prototype vector to bias the generation of bits. For each chromosome in the population, the bits are selected by generating a uniformly distributed random number in the range [0,1] for each bit. The chromosome bit is set to one if the random number is less than the corresponding prototype vector element, zero otherwise. All the chromosomes are then evaluated by the objective function and the best identified.

Equation 1 is then applied to the prototype vector to incorporate the directional information of the best chromosome. This equation is a variant of the process described in [3].

$$\begin{aligned} \mathbf{P}_{n+1} &= ((1-l)\mathbf{P}_n + l \cdot \mathbf{C}_B)(1-f) + \frac{f}{2} \quad (1) \\ f &= \frac{2sl}{1-2s(1-l)} \end{aligned}$$

Where $\mathbf{C}_B$ is the best chromosome and consists of a pattern of ones and zeros, l is the learning rate, and s is the search rate.

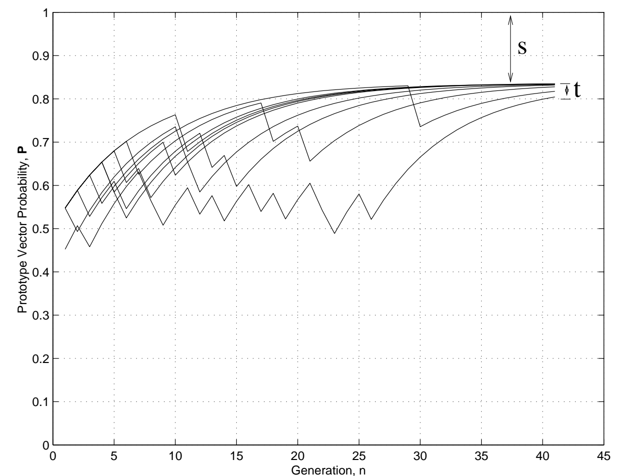

Figure 1: Typical Prototype vector plot (9 bits in $\mathbf{P}$)

Figure 1 shows the changes in the prototype vector during a typical optimisation run. The learning mechanism in equation 1 leads to a change in each prototype vector element level, that follows an exponential profile.

The search rate, s, is the distance that the final convergence level is offset by and is shown graphically on figure 1. The search rate may also be considered as the probability of selecting a one instead of a zero element after an infinite number of generations.

If the search rate is set to zero, the elements of the prototype vector can converge to either 0 or 1. Once this terminal value has been reached, if the element has converged in the wrong direction, there is no way for it to be corrected. Increasing the search rate prevents the prototype vector elements converging exactly to 0 or 1. The search rate is analogous to

*Department of Aerospace, Power and Sensors, Royal Military College of Science, Cranfield University, Shrivenham, England. E-mail: e.j.hughes@rmcs.cranfield.ac.uk .

mutation rate other evolutionary algorithms. The higher the value of s, the less likely the algorithm will get stuck in local optima.

The algorithm is allowed to run until all the elements are within a bound t of the final convergence level of s or $1-s$ as appropriate. The bound t is shown graphically in figure 1. The bound t is defined as shown in equation 2.

$$t = \frac{(0.5 - s)}{10} \tag{2}$$

The termination condition may be summarised as shown in equation 3.

$$\max(0.5 - |\mathbf{P} - 0.5|) < s + \frac{(0.5 - s)}{10} \tag{3}$$

3 Guidelines for tuning

Three parameters are used to control the algorithm. The first, *learning rate*, lies in the range zero to one and determines the final accuracy of the solution. The lower the learning rate, the less likely it is that the algorithm will converge on a local optimum. With high learning rates, the algorithm will be less likely to do a comprehensive search of the optimisation surface. Baluja [1, Page 17] observed that:

If the learning rate is high, the initial populations generated will largely determine the focus of the search, without enabling the algorithm to explore the function space. If the function space does not contain local optima, a high learning rate may work well. However, if local minima could be a problem, lower learning rates allow greater exploration.

As l is increased, the number of function evaluations reduces but the probability of premature algorithm convergence increases. An empirical range of $0.1 \leq l \leq 0.4$ has been found satisfactory for most problems. The higher learning rates are better for problems with few local optima.

The second, and most important parameter, is *population size*. Population size determines the probability that the algorithm will find the global optimum. Increasing the population size will increase the chance of finding the global optimum solution, though it will also increase the number of function evaluations required.

Small populations yield rapid but crude results, large populations will give more accurate results but at a processing cost. The PBIL algorithm will operate with population sizes as low as two, but the probability of finding the global solution over local optima is low.

The third parameter is *search rate*. We may operate the algorithm by choosing a population size and then calculating the maximum value of s that minimises the number of function evaluations. Equation 4 gives an empirical approximation to the optimum value of s.

$$s_{opt} \approx 1 - (2/\sqrt{p})^{1/b} \tag{4}$$

A population size of 5 with an approximate optimum value for s ($0 < s < 0.5$) is often a good starting point for tuning the algorithm. It is wise to start with low population sizes to assess how many function evaluations are required and then increase p to achieve sufficiently accurate results.

If a number of optimisation runs are performed, the rate of occurrence of the best solutions can be monitored. If the probability of finding a good solution is low, the population size should be increased. This will improve the probability of finding the global solution too. The repeatability of the solutions can be traded against execution time in this manner.

It must be noted that for the PBIL algorithm to converge on a final solution, each gene should influence the objective function. If inverting a gene value has no effect, the associated prototype vector element will drift randomly around the 0.5 average value. It has been demonstrated that given the condition of every gene having influence, the algorithm will eventually converge on a solution [4].

4 Example implementation

Section 7 shows the MATLAB source code for a problem where the function being optimised is the sum of the bits in the chromosome. This function is discontinuous in nature. The whole programme consists of about 20 lines without the comments. A learning rate of $l = 0.1$ was chosen for convenience.

The following table shows the steps taken to tune the algorithm (averaged over 100 runs). At each step, a new value of s was calculated. The trials have shown that with a population of 20 in the example, the correct answer can be obtained about 93% of the time and should take approximately 106 generations to complete. The *Expected Number of Evaluations to Success* is on average 2280 objective calculations. The population size was chosen to give better than 90% chance of finding the best value.

p	Gens.	Calcs.	P_{hit}	σ_{Gens}	ENES
5	167	835	3%	21	2783
10	132	1320	57%	14	2316
15	117	1755	83%	13	2115
20	106	2120	93%	10	2280
25	100	2500	98%	9	2551

Increasing either the learning rate or calculated search rate slightly may reduce the number of function evaluations needed, but the variability of the run lengths will increase.

5 Reduction of processing overhead

By their very design, evolutionary algorithms can be inefficient with objective function calculations. In the first few generations of the algorithm, all of the chromosomes evaluated are likely to be different. As the population of chromosomes converge toward a solution, a small set of chromosomes may be evaluated repeatedly. If the objective function has heavy processing requirements, much processor time can be wasted. A binary search tree described in [5, Pages 543–563] may be used to reduce the number of wasted calculations. The following approach may be applied to most evolutionary algorithm techniques.

The tree is used to store chromosome patterns and their corresponding objective values. The tree is generated by inserting each new chromosome as its objective value is required by the evolutionary algorithm. Chromosomes are added to the tree only if they are not already present. The new chromosome is compared to the first chromosome in the tree. If the new chromosome is smaller, the left branch of the tree is investigated. If it is greater, the right branch is chosen.

The tree is descended in a recursive fashion until either a matching chromosome is found or a chromosome with no sub-tree to follow is encountered. If a match is found, the previously recorded objective value is returned to the evolutionary algorithm. If no match is found before the tree ends, an objective value is calculated for the chromosome. The objective is then inserted into the tree along with the chromosome. If the same chromosome is generated by the evolutionary algorithm again, it can be retrieved quickly from the tree.

The tree structure is suited to both binary and integer chromosomes. Real valued chromosomes are difficult to store effectively as a tiny deviation in one gene is enough to prevent the chromosome being matched. If the objective is quick to calculate, it may be better not to use the tree. If the objective is computationally expensive or requires heavy disk usage, in a typical evolutionary algorithm, one quarter of the objective values may be returned from the tree.

6 Conclusions

In most cases, only the population size and search rate need to be adjusted to trade repeatability against number of function evaluations. The simplicity of the algorithm allows it to be applied to new problems rapidly and can give excellent results with little or no tuning.

7 *MATLAB* example

```
% Population Based Incremental Learning
% E.J.Hughes  14/11/97
%% user control parameters
  maxgen=3000;     % maximum no. of gens.
  b=101;           % no. of bits in chrom.
  l=0.1;           % learning rate
  p=20;            % population size
%% set other control parameters
  s=1-(2/sqrt(p))^(1/b);   % search rate
  f=2*s*l/(1-2*s*(1-l));   % f
%% initialise
  pv=0.5*ones(1,b);
  pvx=zeros(maxgen,b);
%% main loop
  for n=1:maxgen
%% generate population
    chrom=rand(p,b)<(ones(p,1)*pv);
%% put objective here
    obj=sum(chrom')';  % calc. sum of bits
    [a,i]=max(obj);    % i = best chrom.
%% update proto. vec. and stop if converged
    pv=((1-l)*pv+l*chrom(i,:))*(1-f)+f/2;
    pvx(n,:)=pv;
    if max(0.5-abs(pv-0.5))<(s+(0.5-s)/10)
       break; end
  end
%% output results
  [(pv>0.5) a]
  plot(1:n,pvx(1:n,:)); % plot proto. vec.
```

References

[1] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-95-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[2] Ignatious Thithi. Control system parameter identification using the population based incremental learning (PBIL). In *IEE Conference Publication No. 427, UKACC International Conference on Control (CONTROL '96)*, pages 1309–1314, London, 2–5 September 1996.

[3] J. R. Greene. A role for simple, robust 'black-box' optimisers in the evolution of engineering systems and artifacts. In *Genetic Algorithms in Engineering Systems: Innovations and Applications. (GALESIA '97)*, pages 427–432, Glasgow, 2–4 September 1997. IEE Conference Publication No. 446.

[4] Markus Höhfeld and Günter Rudolph. Towards a theory of population-based incremental learning. In *1997 IEEE International Conference on Evolutionary Computation*, Ch. 127, pages 1–5, Indianapolis, 13–16 April 1997. IEEE.

[5] Mark Allen Weiss. *Algorithms, data structures, and problem solving with C++*. Addison–Wesley Publishing Company, Inc., 1996.