Predicting Ten Thousand Bits from Ten Thousand Inputs

W. B. Langdon

Department of Computer Science, University of Essex, UK
10 August 2006
Technical Report CSM-457
ISSN 1744-8050

Abstract

Submachine code genetic programming and multiple classification GP runs are used to winow the predictive inputs and then evolve an entry to the WCCI-2007 binary time series prediction task held at the Congress on Evolutionary Computation (CEC-2006) held in Vancouver.

1 Introduction

As part of the World Congress on Computational Intelligence, CEC-2006 held a number of competitions. Entrants to the binary time series prediction could down load the first 10 000 bits of a 20 000 bit time series on which to train their prediction algorithm before the start of the conference.

1.1 Shift Registers for Time Series Prediction

A common technique in predicting time series is to treat the complete series as a shift register. At each time step the current value becomes available and all previous values are shifted into the past one step. Values from a range of previous times are extracted at certain "tap points" (see Figure 1). The learning algorithm combines these to give its prediction for the next time step.

Similar circuits were widely used in analogue signal processing circuits where the shift register was implemented as continuous time delay, e.g. via a transmission delay line.

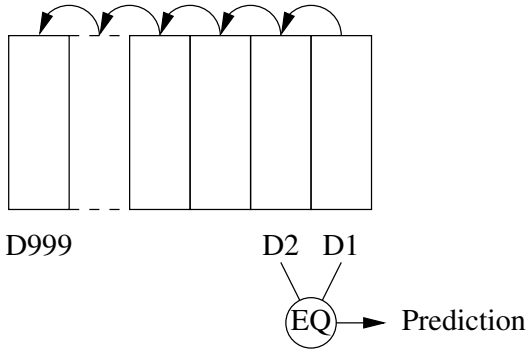

Figure 1: Shift Register. Each time step, data are moved one cell to the left. Two inputs (delay by one time unit and two time units) are fed into an small circuit (EQ) to predict the input on the next time step. It is correct about two thirds of the time.

Here the "tap points" where connections to physical parts of the line corresponding to delays of the required number of milliseconds.

For our learning system we used genetic programming. The GP's goal is to use the first ten thousand bits as a training set with which to automatically evolve a digital circuit whose inputs are taken from fixed tap points (i.e. historical values) and whose output would predict the next bit in the time series.

For simplicity we chose to make the circuit from two input gates. Since we use the C++ implementation of GProc we chose the six binary bit operations of C. I.e. AND, NAND, OR, NOR, EQ and XOR. Identity and not can be readily fabricated by including the two binary constants (0 and 1). Thus 10 of the 16 binary Boolean functions are readily available to GP however the remaining 6 require more complex circuits.

On other Boolean problems we found that good results can be obtained when evolution can also act at the individual gate level [1]. It would be possible to extend our implementation to include all 16 possible functions. Indeed sub-machine code GP readily allows manipulating data both vertically and horizontally across 32 bits. This might have been beneficial but we restrict our operators to making the GP run faster by performing 32 conventional Boolean operations in parallel.

1.2 Deciding Which Tap Points to Use

It is far from obvious which of the 9999 possible tap points to use. Therefore we decided to use them all. We adopted a technique we had previously successfully used when mining DNA chip datasets containing more than 12 000 analogue data signals.

Essentially the technique is to divide the problem into a number of passes. In the initial passes the GP is given the task as usual and all the available data. However it is not expected to solve the problem. Instead GP is run many times and we analyse the evolved solutions to find which inputs are used more often than the others. In [2] we used two initial passes to progressively reduce the number of inputs from tens of thousands to several hundred and then to ten or so. (In one cancer prediction study, despite the inevitable noise, GP selected two predictive genes from more than more than 12 000.)

On average the initial population contains 15.3 nodes per tree. Slightly more than half of these are leafs (remember the trees are binary) and half of these are signals (the other half are constants). Thus, on average, each tree contains 4 inputs. The expected number of random trees we need to create to ensure we have all 9999 inputs is about $\frac{1}{4}n \log n = 23\,023$ (with variance $\approx \frac{1}{4^2}n\pi^2/6$, i.e. standard deviation of 32). Of course even ensuring every tap point is included in the initial population does not give it a fair chance, since it may be included with poor signals or be badly placed in the randomly generated tree. As a compromise between bigger populations being fairer and excessive runtime we chose a GP population size of 25 000 programs. We ran GP 1000 times with different initials seed for the Park-Miller pseudo random number generator. The distribution of frequency with which the 9999 tap points occurred in the best of the last generation programs is shown in Figure 2.

2 Experiments

As Figure 2 shows some tap points (D3, D5 and D8) occur in about 60% of best of run programs but the fraction falls down to zero. We chose to set the cut point at 10 or

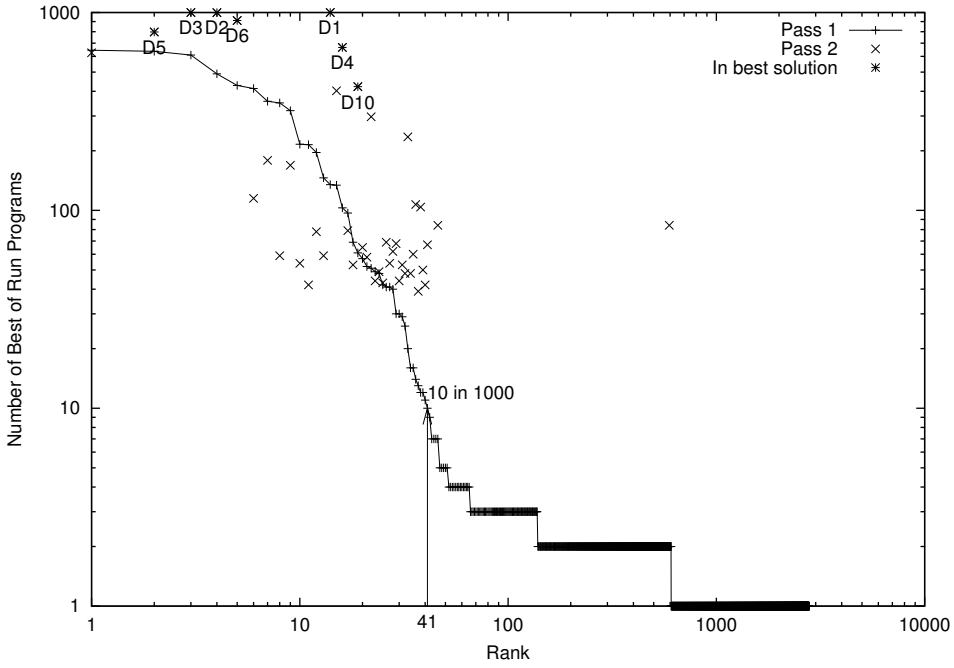

Figure 2: Frequency (+) with which each tap point occurred in 1000 best of run programs. Note 7215 taps are not used in any best of run program. The corresponding frequencies for best of run programs in second pass are plotted with × and the seven tap points used in the overall best program are labelled.

more programs. This included 41 tap points. See Table 1. However noticing that, except for the seventeenth and eighteenth, all of the first twenty tap points were included, we decided to include D17 and D18 regardless. (However despite this they are were not used by the best program. Indeed they are included in only 8% of best of run second pass programs.)

If we now look at the programs evolved at the end of identical runs when restricted to 43 tap points (rather than 9999) Figure 3 shows GP does much better. Returning to Figure 2 we see the inputs are more uniformly used in the second pass. About half of them are included in between 4-7% of best of run programs, with the other half being included in more than 7%. Note all of the tap points used in the overall best program are included in about half or more of the second pass runs.

Figure 4 shows evolution tended not to place taps points with longer delays (i.e. more than 11) in the best of run programs and where it did use them, the evolved program tended to be of mediocre performance. This suggests evolution agrees with our expectation that longer delays are less useful.

We used Sub-machine-code Genetic Programming [4] to run 32 test cases simultaneously. Effectively reducing the 10 000 test cases to 313. In initial runs we noted that in two thirds of cases the best of the initial population was equivalent to (EQ D1 D2). It appeared that evolution was struggling to escape from this trap. Therefore we introduced the pragmatic step, of initially (i.e. during the first ten generations), not testing programs on the 6 783 test cases which this simple rule correctly predicts.

I suspect that further modest improvements might be made by dynamically changing which subset of the training data to use during the run. Gathercole's limited error fitness (LEF) and dynamic subset selection (DSS) [5] and coevolution [6] come to mind as possible ways to implement this.

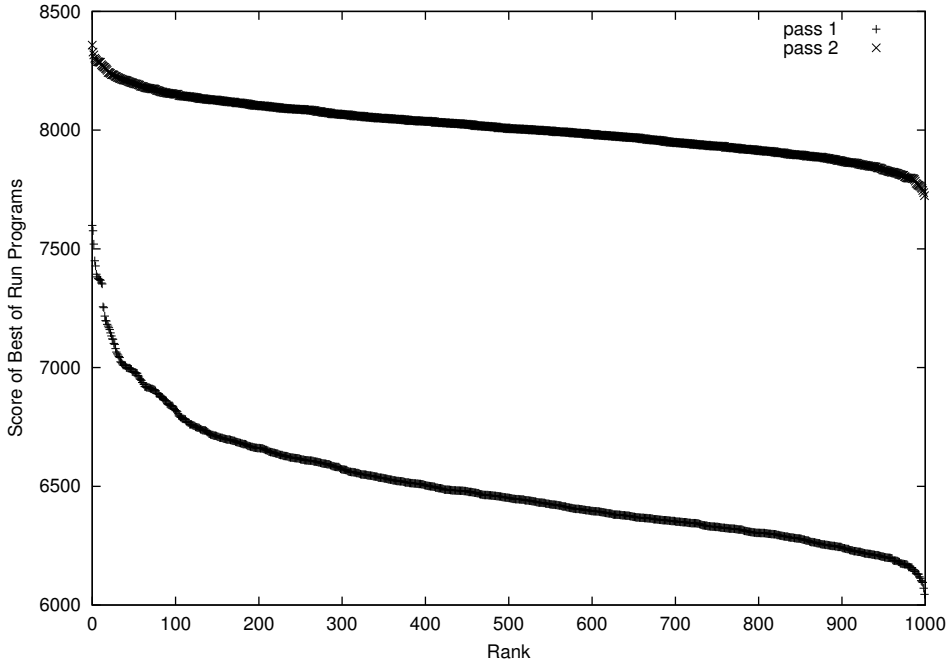

Figure 3: Number of correct predictions in first (lower curve) and second passes (upper curve). Pass 2 run using exactly the same settings as pass 1 but with 43 inputs rather than 9999.

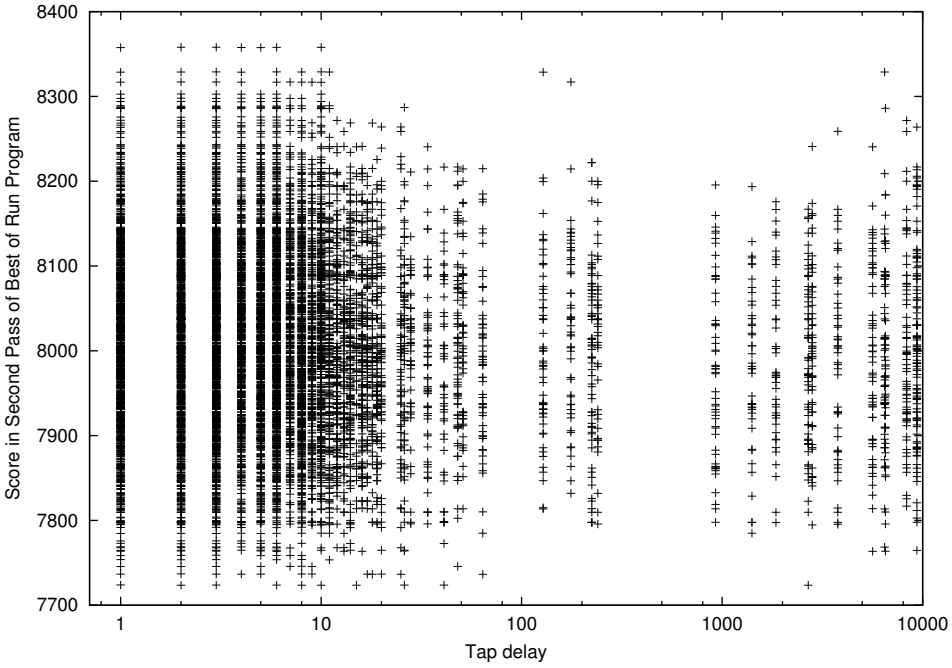

Figure 4: Score of 1000 best of run programs from the second pass containing each tap point. Note most taps up to 11 are included in the highest scoring program and a large number of best of run programs contain them. Taps with longer programs are included in fewer such programs and are not included in the best of them.

Table 1: GP Parameters to Evolve a Binary Time Series Predictor

Objective:	Predict the next bit in the CEC-2006 binary time series prediction competition
Function set:	AND OR NAND NOR XOR EQ
Terminal set:	1 0 D8 D5 D3 D2 D6 D16 D14 D5632 D12 D64 D6528 D19 D25 D1 D7 D4 D20 D2816 D10 D26 D34 D9 D1856 D3776 D1408 D8320 D128 D224 D48 D28 D928 D51 D11 D41 D2688 D13 D6464 D9360 D176 D240 D15 *Plus* D17 D18
Fitness:	Number of correct predictions. Note for the first ten generations test cases correctly predicted by the rule (EQ D1 D2) are not used.
Selection:	generational (non elitist), tournament size 7
Pop Size:	25 000
No size or depth limits. (Largest evolved tree 363, average 36.)	
Initial pop:	Each individual comprises a tree created by ramped half-and-half (2:6). Approximately half terminals are constants, 0 and 1.
Parameters:	50% size fair crossover, crossover fragments $\leq$ 30 [3] 50% mutation (point 22.5%, constants 22.5%, shrink 2.5% subtree 2.5%)
Termination:	generation 50

For convenience the parameters used in the first pass where also used in the second pass. The best of 1000 runs scores 8 342 on the 10 000 bits of the training set.

A small gawk script was used to convert the lisp s-expression style program tree generated by GProc into compilable C code. A wrapper was coded in C to enable the conference organisers to invoke the evolved code on up to 20 000 bits, without running the whole GP system.

3 Discussion

It is encouraging that the input features found by this technique are mostly those which we would have anticipated.

While the use of Sub-machine code GP made runs faster the conditional compilation switch SMC_GP in GProc-1.8.tar.gz is not compatible with others, leading to errors. As usual GP initially, evolved code which worked around horrendous implementation errors and produced credible performance. Naturally this conceals errors and impedes debugging. Nonetheless the supplied code could be speedily adapted for a totally new application.

On hard problems previously [7] we had had success by dividing a large population into a fine mesh of overlapping demes which restrict selection and crossover to a small geographic neighbourhood. However in this case, a few initial runs with large populations were disappointing. We also had success with multiobjective approaches [7] but did not try them on this problem. Another alternative would be to try small populations [8, 9].

The use of fast C++ implementation and sub-machine code genetic programming enabled the equivalent of approximately 31.6 10^{12} program runs to be executed from 9^{th} to 11^{th} July at the University of Essex on five PCs running Linux.

4 Conclusions

The technique we proposed for finding relevant genes for tens of thousands using multi-run multi-pass evolutionary approach with about a hundred noisy training examples [2] successfully identified all relevant genes when adapted for use with a single (albeit noise free) training example.

It is unclear what is special about this time series prediction task which makes it so hard even when what appears to be the relevant data is winowed from the chaff. The competition organiser hints that a 100% correct solution is possible, even easy, if the relevant mathematical knowledge is used. Which in turn suggests the conventional delay line metaphor is the wrong approach and that we are left with the usual AI dilemma that the problem must be correctly represent before it can be readily solved.

References

[1] Riccardo Poli, Jonathan Page, and W. B. Langdon. Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In Wolfgang Banzhaf *et al.*, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1162–1169, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[2] W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257, September 2004.

[3] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000.

[4] Riccardo Poli and William B. Langdon. Sub-machine-code genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA, June 1999.

[5] Chris Gathercole. *An Investigation of Supervised Learning in Genetic Programming*. PhD thesis, University of Edinburgh, 1998.

[6] Ludo Pagie and Paulien Hogeweg. Ludo Pagie and Paulien Hogeweg. Evolutionary consequences of coevolving targets. *Evolutionary Computation*, 5(4):401–418, 1997.

[7] William B. Langdon. William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, 1998.

[8] Chris Gathercole and Peter Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In John R. Koza, *et al.*, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 111–118, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[9] Wendy Ashlock. Using very small population sizes in genetic programming. In *2006 IEEE World Congress on Computational Intelligence, 2006 IEEE Congress on Evolutionary Computation*, pages 1023–1030, Vancouver, 16-21 July 2006.