

Local Search in Parallel Linear Genetic Programming for Multiclass Classification

Aaron Scoble¹, Mark Johnston², and Mengjie Zhang¹

¹ School of Engineering and Computer Science

² School of Mathematics, Statistics and Operations Research

Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand

{aaron.scoble,mengjie.zhang}@ecs.vuw.ac.nz, mark.johnston@msor.vuw.ac.nz

Abstract. Parallel Linear Genetic Programming (PLGP) is an architecture that addresses instruction dependencies in Linear Genetic Programming (LGP). The Co-operative Coevolution (CC) methodology has previously been applied to PLGP but implementations have not been able to improve performance over vanilla PLGP. In this paper we present Hill Climbing Parallel Linear Genetic Programming (HC-PLGP) which uses a local search to discover effective combinations (blueprints) of partial solutions that are evolved in subpopulations. By introducing a new caching technique we can efficiently search over the subpopulations, and our improved fitness function combined with normalisation and blueprint elitism address some of the weaknesses of the previous approaches. HC-PLGP is compared to three PLGP architectures over six datasets, and significantly outperforms them on two datasets, is comparable on three, and is slightly worse on one dataset.

1 Introduction

Classification tasks involve classifying a set of instances as one of a number of potential classes. Each instance is typically represented by a set of features, and a machine learning classifier will use the features as input and attempt to learn some relationship between them that allows it to correctly classify each instance. The *classifier* is first fed training data in which each item is already labeled with the correct class, and creates models that can then be used to classify similar, previously unseen, data.

Genetic Programming (GP) seeks to evolve useful programs in a *population* which evolves over a number of *generations* by applying evolutionary operations analogous to those found in nature [12][13]. Individual programs have their *fitness* measured to determine their suitability as donors for the next generation. *Mutation* and *crossover* operations are performed, and *elitism* preserves the best solutions. The original and most widely used form of GP is Tree Based GP (TGP) [12] where operations are performed on subtrees of operations.

Linear Genetic Programming (LGP) uses a linear sequence of instructions using a set of registers that are re-used during execution of the program and give multiple outputs upon completion [1]. Similar to assembly language, each

instruction may use up to two registers as input, and a single register as output. The strength of LGP is that values that have been computed by one instruction can be used by another instruction later in the program execution. This makes LGP a powerful problem solving technique that has been shown to outperform TGP in multiclass classification [15][6]. However, every time a result is reused an *instruction dependency* is introduced, which means that a relatively minor modification to one instruction may then cascade throughout the program by altering the value in a register that is used by multiple subsequent instructions. As large and random changes are known to disrupt the performance of GP, the strength of LGP is also a potential shortcoming. To reduce the effect of instruction dependencies, Downey and Zhang [4] proposed the Parallel Linear Genetic Programming (PLGP) architecture, in which a program is separated into independent sub-programs, or *factors*. Each factor is executed with its own set of registers, and the register values from all factors are summed to give an overall *result vector*.

LGP can easily be applied to classification problems by providing an output register for each of the class labels. When execution is complete, the register with the highest value is selected as the result of classification. For example, the highest value in a result vector of $[-1, 4, 0]$ for a three class problem is that of the second register, so the instance is classified as class two.

Co-operative Coevolution (CC) considers several subpopulations, each of which evolves in parallel and optimises one aspect of the final solution [17][16][18]. This technique is applied in Downey and Zhang's Co-operative Coevolution Parallel Linear Genetic Programming (CC-PLGP) and Blueprint Search Parallel Linear Genetic Programming (BS-PLGP) which introduce a two level solution into the PLGP architecture. The CC architecture is implemented by first evolving *individual* factors in subpopulations, and then searching for the best *combinations* of those factors [3].

CC-PLGP has not so far proven to be as effective as PLGP or the state-of-the-art BS-PLGP. The goal of this paper is to refine CC-PLGP to further exploit caching techniques, as well as introducing a new approach to blueprint evolution using local search. We discuss some of the obstacles to effective evolution when using a CC design, and propose solutions to them.

The rest of the paper is organised as follows. Section 2 provides some background, and Section 3 describes our approach to addressing some of the shortcomings of current solutions. Section 4 describes our experimental setup, and Section 5 provides results and analysis. Conclusions are presented in Section 6.

2 Background

2.1 Classification

Supervised learning methods, such as GP, require a labelled data set, in which each *instance* consists of a set of features that describe the real-world object as well as a class label which denotes the desired output from the classifier for that

instance [12]. The data set is divided into three subsets: training, validation, and testing.

A GP program iterates through each instance in the training set, using features as input, and produces an output which represents the classification that has been made. The *fitness* of that program is typically calculated from the number of correct classifications. Programs are checked periodically (every few generations) by executing them on the validation set so that the problem of *overfitting* is addressed, i.e., where programs fail to generalise by memorising the training set rather than learning how to identify them. Finally, after a predefined number of generations, the best program that has been evolved is executed on the test set, which is an entirely unseen set of instances, and the true effectiveness of the best program is evaluated.

2.2 Linear Genetic Programming

The use of registers is the key motivation behind LGP, as it allows the reuse of computational results by instruction sequences later in the program [1]. This makes it possible to express complicated solutions, but a genetic operation may be propagated throughout the program by making even a small modification. If an instruction is modified and that changes the value in a register that is subsequently reused many times, the result may be much larger and less controlled than was intended. The value in the register r that holds the output from an instruction i may be modified if instruction i is altered by genetic operations, and any subsequent instruction that uses register r as input is said to have an *instruction dependency* on instruction i .

line	instruction	depends on
1	$r[1] = 3.1 + f1;$	–
2	$r[3] = f2/r[1];$	1
3	$r[2] = r[1] * r[1];$	1
4	$r[1] = f1 - f1;$	–
5	$r[1] = r[1] - 1.5;$	4
6	$r[2] = r[2] + r[1];$	3,5

Fig. 1. An example of instruction dependency, adapted from [3]

An example of instruction dependencies is given in Figure 1. The registers are denoted as $r[i]$, and the features are denoted as fi . Here, e.g., instructions 2 and 3 will both read from $r[1]$, so they have a dependency on instruction 1.

2.3 Parallel Linear Genetic Programming

Downey and Zhang [4] developed PLGP to limit instruction dependencies between instructions by controlling the interaction between them. Here, n instruction sequences, called *factors*, are evaluated independently to give n register

vectors which are then summed to give a single result vector. Each program factor has its own set of registers which are initialised to zero before execution and are not passed to the following factor. This prevents instruction dependencies between factors, and limits the potential number of instruction dependencies to the number of instructions in each factor. Each generation, evolutionary operations are applied to a single program factor, and mutation is restricted to operate on instructions *within a single factor*. By regarding each program as an *ordered list* of factors, crossover is limited to *equivalent factors*, i.e., those that occupy the same list position in both programs. Crossover is performed by exchanging contiguous blocks of instructions between equivalent factors. This creates Enforced Sub-Populations (ESP) [8] where genetic material may flow within subpopulations but not between them.

A PLGP program *topology* specifies the number of factors in a program and the number of instructions in each of those factors. A large number of small factors will allow few dependencies and produce simple solutions, while a small number of large factors may produce a large number of dependencies, although allow for greater complexity.

The structure of PLGP makes it possible to exploit caching of registers, and *difference caching* [5][3] significantly reduces execution time. The evolutionary process is faster when three or more factors are used, and efficiency improves as the number of factors increases. In PLGP, program result vectors are retained from one generation to the next, and when a factor is selected for modification, the difference in output for that factor before and after modification is simply added to the overall evaluation.

2.4 Co-operative Coevolution

A CC framework consists of n subpopulations where each individual is a *partial solution*. A *complete solution* will consist of n partial solutions, with one candidate from each subpopulation. Each subpopulation performs evolutionary operations in isolation and components are encouraged to specialise and evolve within a *niche* consisting of similar individuals.

Symbiotic Adaptive Neuroevolution (SANE) applies this approach to evolving neural networks [14] and describes complete solutions as *blueprints*, which contain pointers to neurons (partial solutions) and act as a mechanism for discovering and remembering high quality *combinations* of neurons. In CC-PLGP, the principles of SANE are applied to the PLGP architecture [3][14]. The notion of equivalent factors is extended to the creation of subpopulations, and a PLGP program is replaced by a list of pointers, or blueprint. After the factor subpopulations perform their evolutionary operations, the blueprints undergo a further evolutionary process to create new combinations of factors.

In the first phase of a generation, factors are executed against all instances and their register values are cached. A blueprint is evaluated by summing the register values for each factor that contributes to that blueprint, relative to each instance. The index of the register with the highest value for each instance is compared to the class label for that instance, and the number of incorrect

classifications is assigned as that blueprint's fitness value. The fitness measure of a factor is calculated as the average fitness of all the blueprints that factor contributes to. If a factor does not contribute to any blueprint, it is assigned a *No Fitness Information (NFI)* fitness value which is the average fitness of all blueprints. Blueprint evolution in CC-PLGP is performed by applying a set of evolutionary operations to the blueprints. Mutation changes a pointer into a new random pointer, and crossover exchanges randomly selected pointers between two blueprints.

Using outdated factor information when selecting combinations for blueprints is a flawed approach to fitness evaluation [3]. In CC-PLGPs factor evolution phase, factors may be modified by evolutionary operations regardless of whether they contributed to blueprints. Any assumptions made in finding effective blueprints based on improving existing blueprints are therefore undermined. A blueprint that had high fitness in generation g could conceivably have every factor modified in generation $g + 1$, making the search for solutions through gradual modification more difficult. The ability of a blueprint to make changes to several pointers in the same generation may also have presented problems, as blueprints quickly attained a reasonable fitness in early generations, but seemed unable to be refined in sufficiently accurate steps during later generations.

BS-PLGP is another approach to CC, but with a different approach to blueprint evolution. Particle Swarm Optimisation (PSO) [11] is used to select combinations of factors [3].

The improved accuracy attained by BS-PLGP suggests that the CC concept has its merits in PLGP, but the evolutionary operations CC-PLGP applies to blueprints are ineffective. Although BS-PLGP and CC-PLGP have not provided significant improvement over PLGP with regard to classification accuracy, it is suggested that a more thorough investigation of the interaction and cooperation between subpopulations could enable further improvements.

3 Hill Climbing Parallel Linear Genetic Programming

We propose Hill Climbing Parallel Linear Genetic Programming (HC-PLGP) as a CC framework where factors are evolved in subpopulations and hill climbing search is used for blueprint evolution. We extend the use of register caches to enable a complete search through the factor combinations, allowing us to develop a fitness function that is not dependent on the NFI values. The hill climbing blueprint search searches for a better candidate from each subpopulation in turn while the rest of the blueprint remains unchanged, to find the best candidate factor to cooperate with the existing factor combination, if one can be found in the current generation. The goal is to develop a method that is comparable to PLGP in accuracy and efficiency, while strengthening the CC architecture.

Overview of HC-PLGP

A *Register Vector* rv_i is a set of registers that are used to store the output for a factor after it has been executed on an instance i . A register vector will contain c

registers, where c is the number of classes in the classification problem. A *Register Collection* rc_j is a set of k register vectors, one for each of the k instances in the dataset. Each factor j will have a register collection that stores the register vector after execution on each instance, i.e., rc_j will contain $rv_0, rv_1, \dots, rv_k$ which are the results for executing factor j on instances $0, 1, \dots, k$.

In the same manner as the previous CC designs, n subpopulations are evolved in isolation from each other. In addition to this, a blueprint population undergoes a separate process in which combinations of factors are selected and evaluated.

At the beginning of each generation, factors are evolved within their subpopulations using standard evolutionary operations. Any factors that contribute to blueprints will be retained in the child population, and the remaining child factors are generated. Once factor evolution is complete, the factors are executed on all training examples. The Training Register Cache (TRC) is filled with a set of register collections, one for each factor. Factor combinations are then searched using the TRC, giving a value for classification fitness for every combination of factors, which are stored in the Fitness Cache (FC). Once the classifications have been made, the hill climbing blueprint search will determine whether the replacement of any single factor can improve that blueprint's fitness by searching through the FC. The algorithm is divided into four main stages as follows.

Execution. Here, factor output is calculated, similar to the execution stage in CC-PLGP, i.e., every instance is presented in turn and executed by all factors. Register vectors for each instance are stored as the register collection of each factor.

Classification. For each training instance i , we consider every blueprint b in turn. We sum the register vectors rv_i for each factor that contributes to this blueprint b , giving the *reference vector* for b . Classification accuracy can be observed here by comparing the index of the highest register value against the class label for i . Iterate through each subpopulation s to cache the classification accuracy for each factor in the subpopulation, with the other factors in b remaining unchanged. To achieve this, subtract the output vector rv_i of the factor that currently represents subpopulation s in blueprint b to obtain a *base vector*, or a result vector for b that does not include a factor from s . Then iterate through s and test the other factors in turn by adding their register vector rv_i to the *base vector*, and comparing the index of the highest register value against the class label for instance i . An incorrect classification will result in the FC for that combination being incremented, and after all instances have been presented the FC will represent the classification accuracy for that combination over all instances. Of note here is that only the factors being searched are assigned classification accuracies in the FC, not the 'locked' factors that the search aims to compliment when searching for replacement factors. Also of importance is that the order of execution specifies that all factor combinations are checked against each instance before moving on to the next.

Hill Climbing Blueprint Search. The hill climbing blueprint search is performed by searching through the FC to find the best single modification for each blueprint, if an improvement is possible. The FC is used for these calculations, and by

adding the errors for all the factors that contribute to b , we can search each subpopulation for a viable replacement for any of them. We record the best factor for each subpopulation with the other pointers locked, and after all subpopulations have been investigated the best overall improvement is selected. If that will give an improvement over the original combination, a single factor is exchanged.

To help prevent the search from becoming trapped in local optima, *Blueprint mutation* provides a stochastic element to HC-PLGP. With a probability p one of the following types of mutation are applied. *Micro mutation* chooses a subpopulation at random and then searches it in the normal way, and *macro mutation* selects a random subpopulation and then a random factor from that subpopulation.

Conditional Validation. The final step is to execute the newly selected blueprints on the validation dataset. By performing this step after blueprint combinations have been finalised, we avoid executing redundant factors, or those that will not contribute to any blueprints. If 5 blueprints are used with a subpopulation size of 100 (10×10 topology), at least 95 factors per subpopulation will not need to be validated.

Factor	r1	r2	r3	r4	Factor	r1	r2	r3	r4
1	0.3	0.5	-1.2	3.4	1	0.23	0.33	-0.54	0.77
2	-0.8	-2.4	1.1	2.2	2	-0.44	-0.70	0.52	0.68
3	1.2	40	-5.2	-0.3	3	0.54	0.97	-0.83	-0.23
4	4.6	0.8	-1.2	0.1	4	0.82	0.44	-0.54	0.09
Total	5.3	[38.9]	-6.5	5.4	Total	1.15	1.04	-1.39	[1.31]

(a) non-normalised registers (b) $h(x)$ normalised registers

Fig. 2. Normalisation of registers

Register Normalisation. Figure 3 represents a blueprint that consists of four factors, each with four registers. In Figure 3(a) the registers have not been normalised, and in (b) they have been normalised with the function $h(x) = \frac{x}{1+|x|}$. It can clearly be seen that in (a) factor 3 has greatly influenced the classification from the blueprint with a single large value for register $r2$. In (b), we can see that normalisation has mitigated this effect by reducing the range of register values to $[-1, 1]$. This normalisation may encourage factor specialisation by preventing a single large register value from dominating factors that may otherwise have combined to form an effective classifier.

Improved Fitness Function. A new fitness function is derived from the values retained in the FC and does not distinguish between factors that contribute to a blueprint and those that do not. As each factor is checked once for every blueprint during the classification phase, the FC will hold a number of classification values for that factor. The average of those classification values is assigned as the fitness for that factor, providing a function with a much better ability to discern between factors than one that assigns an NFI fitness value.

Factor Elitism. A critical shortcoming of CC-PLGP was the problem of factor evolution undermining blueprint composition by modifying the components that had been selected during the blueprint evolution phase. To address this, any factors that are selected to contribute to a blueprint are added to a set of *elite* factors that are preserved during the evolutionary process, where they are eligible as parents but will not be replaced.

4 Experimental Setup

A series of experiments were performed and results were averaged over 30 replications. Execution time and classification accuracy of HC-PLGP were compared with PLGP, CC-PLGP, and BS-PLGP using six datasets. Eight program topologies were used, although space limitations have necessitated that comparisons from only one topology are presented here. PLGP, CC-PLGP, and BS-PLGP were run with 500 blueprints, and HC-PLGP was run with 5. Preliminary experimentation was performed with the number of HC-PLGP blueprints ranging from 1 to 40, and although reducing the number of blueprints to less than 5 had a negative effect on accuracy, increasing to more than 5 did not improve accuracy sufficiently to justify the extended execution time.

Datasets. Six data sets are used that provide classification problems of varying complexity. Artificial Characters, Hand Written Digits, and Yeast were used in order to compare directly with [4][3], and the others were chosen so that their characteristics would give a good range of observations. The large number of classes was the selection criterion for Letters, and USPS was introduced due to the large number of features. The datasets are summarised in Table 1, and were divided equally into training, validation, and test sets.

The following datasets are from the UCI machine learning repository [7]. *Hand written digits* consists of 3750 hand written digits with added noise. There are 10 classes of 64 attributes. *Artificial characters* consists of vector representations of the English letters {A,C,D,E,F,G,H,L,P,R}. There are 5000 instances, 10 classes, and 56 features. The *Yeast* dataset represents protein localisation sites and has 1484 instances, 10 classes and 8 attributes. *USPS* contains alphanumeric characters and codes extracted from handwritten addresses, gathered as part of a research project sponsored by the United States Postal Service [10]. It has 4863 instances, 10 classes, and 256 features. *Cardiotocography* contains 2126 instances where diagnostic features processed from foetal cardiograms. There are 10 classes, and 22 features. It was obtained from [2]. *Letters* consists of all 26 English capital letters using different fonts on black and white rectangular displays. There are 2126 instances, 26 classes, and 16 features, also obtained from [2].

GP Parameters. The GP parameters are constant across all experiments. We use terminal constants in the range $[-1, 1]$, and a function set that consists of addition, subtraction, multiplication, protected division, and *if* < 0 . Protected division returns a value of one if the denominator is zero. *if* < 0 takes three arguments and returns the second if the first is negative, otherwise it returns the third. Population size is 1000 and evolution is performed over 400 generations.

Table 1. Datasets

Dataset	type	instances	features	classes
(a) Artificial Characters	real	5000	56	10
(b) Cardiotocography	real	2126	22	10
(c) Letters	real	2226	16	26
(d) Hand Written Digits	real	3750	64	10
(e) USPS	real	4863	256	10
(f) Yeast	real,binary	1484	8	10

The parameters for factor evolution are as follows: mutation 30% crossover 60%, elitism 10%. Tournament size is 5. Of note here is the elitism parameter, which in the case of HC-PLGP, is equal to the number the blueprints, and is therefore set at a constant 5 for all experiments. For CC-PLGP, blueprint evolution is performed with equal weighting to the operators mutation, crossover, and new random blueprint. HC-PLGP blueprint mutation probability p was set to 0.30.

Table 2. Program topologies

	2x5	4x5	5x7	5x10	10x10	10x20	20x20	30x20
Instructions	10	20	35	50	100	200	400	600
Factors	2	4	5	5	10	10	20	30
Instructions per factor	5	5	7	10	10	20	20	20

PLGP Program Topologies. Program topologies are selected to give a reasonable diversity of factor sizes and number of factors. Table 2 shows the eight program topologies that were used to evaluate the methods. Due to space constraints, we have only included results for the 10x10 program topology. This particular topology was overall the most effective for all methods and has therefore been selected as the best representative topology.

5 Results and Analysis

Table 3 shows the classification accuracy of the four methods for each problem. The results are averaged over 30 replications and are as follows: *Fac* and *Ins* are the number of factors in a program and the number of instructions in a factor, respectively; *Time* is the total execution time, in seconds, for 400 generations; *BstGen* is the generation at which the best validation accuracy was achieved, and at which point the best program was selected to perform the test; *Train* and *Test* represent the classification accuracies for the different procedures; and standard deviations of training and test accuracy are given in *TrainSD* and *TestSD*.

In general, PLGP and BS-PLGP have similar test accuracy over all the problems, and CC-PLGP does not perform as well. Although the execution time of HC-PLGP is longer than that of CC-PLGP, it is much faster than BS-PLGP and comparable to that of PLGP, being faster on (c),(e), and (f) but slower in (a),(b), and (d). The Test SD for HC-PLGP is the lowest of all the architectures in (a),(b),(d),and (e). Performance in (f) is much worse for HC-PLGP, and although the SD is high for both training and testing, it has been consistently

Table 3. Experiment Results for 10x10 topology

(a) Artificial Characters								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	6148.22	322.43	95.30	94.60	2.71	2.64
CC-PLGP	10	10	983.17	333.67	85.44	84.54	5.81	5.93
PLGP	10	10	1348.15	311.10	95.46	94.56	2.89	2.90
HC-PLGP	10	10	1556.97	260.430	96.74	97.16	4.20	1.76

(b) Cardiotocography								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	2137.54	265.30	58.54	40.12	4.49	9.07
CC-PLGP	10	10	491.40	253.13	53.39	38.82	5.27	9.61
PLGP	10	10	713.16	276.07	62.94	46.05	4.99	6.96
HC-PLGP	10	10	781.67	276.74	60.11	37.46	8.27	4.80

(c) Letters								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	3800.41	359.53	36.59	30.72	2.67	2.35
CC-PLGP	10	10	1440.35	326.63	31.19	26.27	2.25	2.13
PLGP	10	10	1578.88	353.00	40.44	31.72	2.77	3.10
HC-PLGP	10	10	1312.09	347.20	41.30	34.66	3.57	2.93

(d) Hand Written Digits								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	3928.20	372.60	82.40	80.07	3.56	3.87
CC-PLGP	10	10	783.03	359.37	71.04	69.33	2.89	2.98
PLGP	10	10	1155.97	374.70	85.68	81.61	2.55	3.00
HC-PLGP	10	10	1499.29	355.55	84.22	82.17	1.87	1.79

(e) USPS								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	3777.47	374.43	80.09	78.49	3.78	3.93
CC-PLGP	10	10	982.70	356.93	69.01	68.03	6.47	6.40
PLGP	10	10	2131.17	346.93	82.59	79.20	3.81	3.77
HC-PLGP	10	10	1958.34	357.40	84.52	78.10	2.10	2.15

(f) Yeast								
Method	Fac	Ins	Time	BstGen	Train	Test	TrainSD	TestSD
BS-PLGP	10	10	1277.93	300.20	56.03	56.07	2.38	2.87
CC-PLGP	10	10	371.01	268.27	50.45	50.82	4.71	4.74
PLGP	10	10	591.68	242.40	58.77	55.61	2.96	3.54
HC-PLGP	10	10	521.65	287.10	52.64	48.57	5.46	4.14

outperformed across all experiment replications. Of note with this dataset is that it has the least number of both instances and classes. There is very pronounced overfitting apparent on (b) for all architectures, and the SD for HC-PLGP here is much higher for training than the other methods, yet lower for testing.

Test accuracy of HC-PLGP was compared to that of all other methods. Significance testing was performed using two-sided t-tests and summarised in Table 4.

Table 4. Significance of results : HC-PLGP vs PLGP, CC-PLGP, and BS-PLGP

	PGLP		CC-PLGP		BS-PLGP	
(a) Artificial Characters	0.00002	$\triangle$	0.00000	$\triangle$	0.00004	$\triangle$
(b) Cardiotocography	0.00000	$\bullet$	0.49458		0.13854	
(c) Letters	0.00318	$\triangle$	0.00000	$\triangle$	0.00002	$\triangle$
(d) Hand Written Digits	0.37313		0.00000	$\triangle$	0.02032	$\triangle$
(e) USPS	0.16684		0.00000	$\triangle$	0.68146	
(f) Yeast	0.00000	$\bullet$	0.07793		0.00000	$\bullet$

The columns represent the p -values of the comparison between HC-PLGP and each method in turn, where the p -values are corrected using Holm's method with an overall significance level of 0.05 for each row [9]. Here, $\triangle$ denotes that the performance of HC-PLGP is significantly better, while $\bullet$ indicates that it is significantly worse. The results show that HC-PLGP significantly outperforms all methods on (a) and (c), and outperforms CC-PLGP and BS-PLGP on (d). However, it is outperformed by PLGP on (b), and by both BS-PLGP and PLGP on (f).

6 Conclusions

Hill Climbing Parallel Linear Genetic Programming (HC-PLGP) has been implemented as a complete search through the subpopulations where all factors are examined, and the best improvement is selected. Blueprint mutation provides a stochastic element to help prevent the search from becoming trapped in local optima. Five blueprints was found to be effective, providing a good balance between execution time and classification accuracy. This number was small enough to retain factor diversity in the subpopulations once elitism had prevented selected factors from being modified by evolution. Output registers were normalised and a new fitness function was used that exploited existing classification information from the Fitness Cache (FC). Classification accuracy of HC-PLGP compares well against the existing Parallel Linear Genetic Programming (PLGP) architectures, significantly outperforming them all on a number of datasets, with a comparable execution time.

Future work. In general, all implementations of the PLGP architecture performed best with the 10x10 topology. It is not clear whether the difference in performance has been affected directly by the structure of the complete program, or the effect on diversity by a changing subpopulation size. A set of experiments where the subpopulation size remains constant and the number of factors and instructions changes will be performed to investigate this further.

There are three datasets, Cardiotocography, Letters, and Yeast, where none of the PLGP methods were very effective and some overfitting was apparent. As these are the datasets with the least number of instances, cross-validation will be used to re-evaluate performance of these methods. The vast difference in training and test accuracy in the Cardiotocography dataset for all architectures is certainly a subject for closer investigation. Simple metaheuristics, e.g., Iterated

Local Search, may be used to prevent becoming stuck in local optimum, so HC-PLGP is potentially very promising as an architecture.

References

1. Brameier, M., Banzhaf, W.: A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation* 5(1), 17–26 (2001)
2. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2, 27:1–27:27 (2011), software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
3. Downey, C.: Explorations in Parallel Linear Genetic Programming. Master's thesis, Victoria University of Wellington, New Zealand (2011)
4. Downey, C., Zhang, M.: Parallel Linear Genetic Programming. In: Silva, S., Foster, J.A., Nicolau, M., Machado, P., Giacobini, M. (eds.) *EuroGP 2011*. LNCS, vol. 6621, pp. 178–189. Springer, Heidelberg (2011)
5. Downey, C., Zhang, M., Liu, J.: Parallel linear genetic programming for multi-class classification. *Genetic Programming and Evolvable Machines* 13(3), 275–304 (2012)
6. Fogelberg, C., Zhang, M.: Linear Genetic Programming for Multi-class Object Classification. In: Zhang, S., Jarvis, R.A. (eds.) *AI 2005*. LNCS (LNAI), vol. 3809, pp. 369–379. Springer, Heidelberg (2005)
7. Frank, A., Asuncion, A.: UCI machine learning repository (2010), <http://archive.ics.uci.edu/ml>
8. Gomez, F., Mikkulainen, R.: 2-d pole balancing with recurrent evolutionary networks. In: *Proceedings of the International Conference on Artificial Neural Networks*, pp. 425–430 (1998)
9. Holm, S.: A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 65–70 (1979)
10. Hull, J.J.: A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16, 550–554 (1994)
11. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, pp. 1942–1948 (1995)
12. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge (1992)
13. Koza, J.R., Streeter, M.J., Keane, M.A.: Routine high-return human-competitive automated problem-solving by means of genetic programming. *Information Sciences* 178(23), 4434–4452 (2008)
14. Moriarty, D.E., Mikkulainen, R.: Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation* 5, 373–399 (1997)
15. Olague, G., Romero, E., Trujillo, L., Bhanu, B.: Multiclass Object Recognition Based on Texture Linear Genetic Programming. In: Giacobini, M. (ed.) *EvoWorkshops 2007*. LNCS, vol. 4448, pp. 291–300. Springer, Heidelberg (2007)
16. Potter, M., Jong, K.: Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation* 8(1), 1–29 (2000)
17. Potter, M., De Jong, K.: A Cooperative Coevolutionary Approach to Function Optimization. In: Davidor, Y., Schwefel, H.P., Milnner, R. (eds.) *PPSN 1994*. LNCS, vol. 866, pp. 249–257. Springer, Heidelberg (1994)
18. Yang, Z., Tang, K., Yao, X.: Large scale evolutionary optimization using cooperative coevolution. *Information Sciences* 178(15), 2985–2999 (2008)