Parallel Linear Genetic Programming

Carlton Downey and Mengjie Zhang

School of Engineering and Computer Science
Victoria University of Wellington, Wellington, New Zealand
Carlton.Downey@ecs.vuw.ac.nz,
Mengjie.Zhang@ecs.vuw.ac.nz

Abstract. Motivated by biological inspiration and the issue of code disruption, we develop a new form of LGP called *Parallel LGP (PLGP)*. PLGP programs consists of n lists of instructions. These lists are executed in parallel, after which the resulting vectors are combined to produce program output. PGLP limits the disruptive effects of crossover and mutation, which allows PLGP to significantly outperform regular LGP.

1 Introduction

Derived from genetic algorithms [5], Genetic Programming (GP) [2, 6] is a promising and nature inspired approach to constructing reliable solutions to a range of problems quickly and automatically, given only a set of human labeled instances on which an evolved program can be evaluated. GP uses ideas analogous to biological evolution to search the space of possible programs to evolve a good program for a particular task. Since the 1990s, GP has been successful for solving many machine learning problems [7, 9, 11–13].

In conventional GP, programs are trees of operators. This form of GP is known as Tree based GP (TGP). Linear Genetic Programming (LGP) is an alternative form of GP where individuals in the population are sequences of instructions. Structuring programs as sequences of instructions has many advantages over structuring them as trees. LGP has been shown to significantly outperform TGP on machine learning tasks such as multiclass classification [4, 8].

LGP performs well on multiclass classification problems because of the power of its flexible program structure. LGP programs consist of a sequence of instructions which operate on a list of registers. This allows multiple outputs and permits results computed early in program execution to be reused later. These two properties make LGP a powerful problem sovling technique.

Unfortunately LGP has a significant weakness inherent to its program structure, which severely limit its effectiveness. The execution of an instruction is heavily influenced by the execution of all previous instructions. This means that modifying an early instruction can be highly disruptive because it has the potential to upset the execution of all subsequent instructions.

LGP programs can be viewed as a tangled web of dependencies. Each instruction depends on the output of several other instructions in order to produce a

result. GP operators can completely destroy this fragile balance by modifying one or more instructions critical to effective execution. Disrupting the program to a large extent results in useless output, and wastes the time required to evaluate the program, greatly slowing down convergence.

Clearly the structure of LGP programs is fatally flawed, and a new program structure is required. In this paper we describe a new form of LGP which overcomes the problems of code disruption while still possessing all the power of conventional LGP.

1.1 Objectives

In this paper, we aim to develop a new form of LGP which removes, or at least limits the disruptive effects of mutation and crossover. This new form of LGP should posses all the power of conventional LGP, but should posses far fewer dependencies. Specifically, this paper has the following research objectives:

- To isolate and identify the features of conventional LGP program structure which significantly compromise performance.
- To develop a new program structure for LGP which limits these problems while preserving the power of conventional LGP.
- To compare the performance of our new program structure to a conventional program structure over a range of parameter settings on several classification problems.

1.2 Organization

The rest of the paper is organized as follows. Section 2 describes some necessary background on LGP and Classification, as well as covering GP related work to classification. Section 3 discusses the problem of code disruption, and describes our new program structure which prevents code disruption. The experiment design and configurations are provided in Section 4 and the results are presented in Section 5 with discussion. Section 6 concludes the paper and gives future work directions.

2 Background

2.1 LGP

In LGP the individuals in the population are programs in some imperative programming language. Each program consists of a number of lines of code, to be executed in sequence. The LGP used in this paper follows the ideas of register machine LGP [2]. In register machine LGP each individual program is represented by a sequence of register machine instructions, typically expressed in human-readable form as C-style code. Each instruction has three components: an operator, 2 arguments and a destination register. To execute the instruction,

the operator is applied to the two arguments and the resulting value is stored in the destination register. The operators can be simple standard arithmetic operators or complex specific functions predefined for a particular task. The arguments can be constants, registers, or features from the current instance. An example LGP program is shown in figure 1.

```
r[1] = 3.1 + f1;
r[3] = f2  / r[1];
r[2] = r[1] * r[1];
r[1] = f1  - f1;
r[1] = r[1] - 1.5;
r[2] = r[2] + r[1];
```

Fig. 1: An example LGP program.

After an LGP program has been executed the registers will each hold a real valued number. For presentation convenience, the state of the registers after execution is represented by a floating point (`double`) vector `r`. These numbers are the outputs of the LGP program and can be interpreted appropriately depending on the problem at hand. A step by step example of LGP program execution can be found in figure 2.

Program Inputs

f1	f2	f3
0.1	3.0	1.0

(a) Feature Values

Program		Registers		
index	Instruction	r[1]	r[2]	r[3]
0	-	0	0	0
1	r[1] = 3.1 + f1;	3.2	0	0
2	r[3] = f2 / r[1];	3.2	0	0.94
3	r[2] = r[1] * r[1];	3.2	10.24	0.94
4	r[1] = f1 - f1;	0	10.24	0.94
5	r[1] = r[1] - 1.5;	-1.5	10.24	0.94
6	r[2] = r[2] + r[1];	-1.5	8.74	0.94

(b) Program Execution

Program Outputs

r[1]	r[2]	r[3]
-1.5	8.74	0.94

(c) Final Register Values

Fig. 2: Example of LGP program execution

2.2 Classification

Classification problems involve determining the type or *class* of an object instance based on some limited information or *features*. Solving classification problems involves learning a classifier, a program which can automatically perform classification on an object with unknown class.

Classification problems can be loosely separated into two types, based on the number of possible object classes. Problems that require distinguishing between objects of only two classes are called *binary classification* tasks. Problems where

there are more than two potential classes for each object are known as multi-class classification problems. Many important classification tasks such as digit recognition are examples of multiclass classification problems.

Classification problems form the basis of empirical testing in this paper.

2.3 Classification using LGP

LGP is particularly well suited to solving multiclass classification problems. The number of outputs from an LGP program is determined by the number of registers, and the number of registers can be arbitrarily large. Hence we can map each class to a particular output in the form of a single register. Classification then proceeds by selecting the register with the largest final value and classifying the instance as the associated class. For example if registers (r1, r2, r3) held final register values (-1.5, 8.74, 0.94) then the object would be classified as class 2, since register 2 has the largest final value (8.74).

3 Parallel LGP

In this section we introduce a new form of LGP called parallel LGP (PLGP)[1] based on the concept of dividing programs into multiple parts which can be executed in parallel.

3.1 Motivation

While LGP is an effective technique for solving problems across many problem domains, it is hamstrung by several key weaknesses. We motivate our new form of LGP by discussing two of these weaknesses in some depth.

Disruption One weakness is the disruption of program structure caused by the application of genetic operators. Genetic operators work by taking an existing LGP program and producing a new LGP program by modifying of one or more instructions. This modification is what causes the formation of good solutions, but it can equally result in the formation of poor solutions. Poor solutions are usually the result of modifications disrupting a key part of the program. In standard LGP, disruption of program structure occurs when modification of one instruction has a negative impact on the execution of other instructions. Because LGP programs are sequences of instructions, modification of any one instruction has the potential to disrupt *every single subsequent instruction in the program.*

Large LGP programs with many instructions are more likely to experience code disruption. Every single instruction after the modification point is vulnerable to code disruption. Longer programs generally have more instructions after

[1] There is an existing technique called Parallel GP (PGP) based on executing distinct GP programs in parallel[1]. Namewise, PGP is similar to PLGP, however the two techniques are unrelated

the modification point. This means they have more instructions vulnerable to code disruption, and hence a higher chance of being disrupted. Unfortunately we often desire large LGP programs. Large programs have more power as they are able to explore a larger solution space. To solve complicated problems we often require complex solutions which can only be expressed by large programs.

Structural Introns A second weakness is that LGP programs have significant amounts of non-effective code, otherwise known as structural introns [3]. Structural introns are instructions which have no effect on program output due to their position in the instruction sequence. An instruction is *not* a structural intron if no subsequent instruction overwrites its value. Hence instructions near the start of a program are more likely to be structural introns, and larger LGP programs have a higher proportion of structural introns.

While it has been argued that such structural introns may be desirable on a small scale [3], they are certainly undesirable on a large scale. As LGP program size increases it becomes overwhelmingly likely that the changes caused by GP operators will be opaque to the fitness function. This in turn will greatly slow convergence resulting in solutions with poor fitness.

Solution We desire a form of LGP where both the disruptive effects of the genetic operators, and the amount of non-effective code are independent of program size and program fitness. This is in contrast to conventional LGP where the negative influence of both *increases* with program size and fitness.

The genetic operators in LGP are most effective when applied to short sequences of instructions. By strictly limiting the maximum length of instruction sequences we can achieve the most benefit from the application of genetic operators, and hence achieve the best performance. However, many interesting problems require a large number of instructions in order to evolve an effective solution. Therefore the only way to increase the number of instructions *without* increasing the length of instruction sequences is to *allow each LGP program to consist of multiple short instruction sequences*. This idea forms the core of our new program representation.

3.2 Program Structure

Parallel LGP (PLGP) is an LGP system where each LGP program consists of n dissociated pieces. A PLGP program consists of n LGP programs which are evaluated independently, to give n sets of final register values. These final register values are then combined by summing the corresponding components to produce a single set of final register values. Formally let V_i be the ith set of final register values and let S be the summed final register values. Then $S = \sum_{i=0}^{n} V_i$.

An example of PLGP program execution is shown in figure 3 and contrasted to an example of LGP program execution. In LGP all instructions are executed in sequence using a single set of registers, to produce a single set of register values as output. In PLGP the instructions in *each program piece* are executed

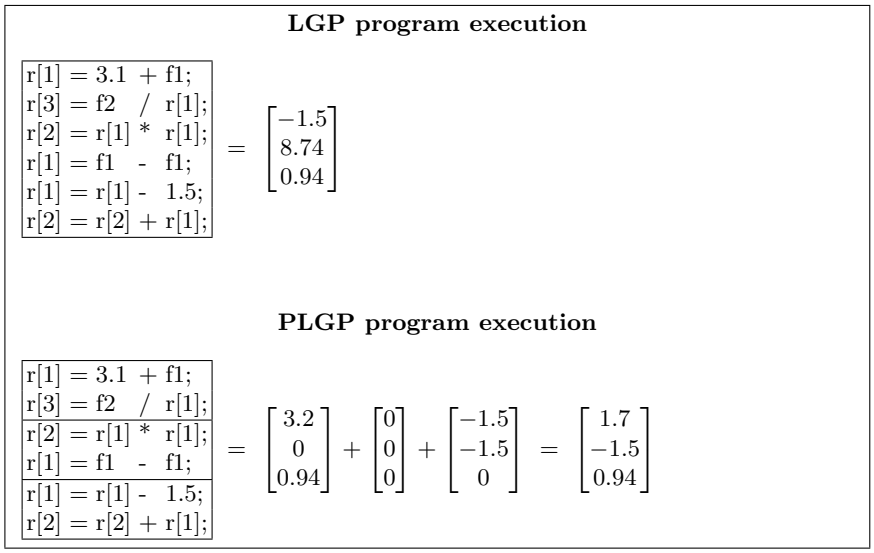

Fig. 3: Contrasting PLGP program execution to LGP program execution

on *their own set of registers* to produce n (in this case 3) sets of register values. In our example the program pieces are separated by horizontal lines, so our PLGP program consists of 3 pieces. These vectors of register values are then summed to produce the final program output. Notice how our LGP program and our PLGP program have *the same instructions* but produce *different final register values*. This is because in our LGP program the results of earlier computations are stored in the registers and can be reused by later computations, while in PLGP each piece begins execution with all register values set to zero.

By separating the program code into several dissociated pieces and summing the resulting vectors we obtain independence between instructions in different pieces. Modifications to one part of the program have no influence on the output of other parts of the program. Hence programs may consist of a large number of instructions, but each instruction is executed as if it was part of a very short program.

3.3 Crossover for PLGP

An important issue to consider for our new LGP system is how crossover will be performed. More specifically, which components of the program we will allow code exchange to occur between.

One option is to allow free exchange of code between any two pieces of any two programs. In other words the crossover operator would select one piece from each program at random and perform normal crossover between these two pieces. In this view the pieces which comprise the program have no ordering, and we can view a program as a *set* of pieces.

The second option is to strictly limit code exchange to occur between equivalent pieces. In other words we impose an ordering on the pieces of each program, so that each program has a first piece, a second piece, etc. In this view the pieces of the program are strictly ordered, and hence a program can be viewed as a *ordered list* of pieces.

Which option we choose determines how we view the population as a whole. If we allow free crossover then we can view all genetic code as belonging to a single 'genetic population' where any instruction can exchange genetic material with any other instruction. If we restrict crossover based on piece number then we are effectively creating a number of genetic sub-populations where there is no interpopulation genetic flow. In other words genetic material can be exchanged within each sub population, but there is no transfer of genetic material *between* sub populations. We borrow some terminology from the area of Cooperative Coevolution and term this kind of PLGP as PLGP with *enforced sub populations (ESP)*.

Using ESP has been shown to give improved classification accuracy when applied in the area of cooperative coevolution. The theory is that by limiting the exchange of genetic material to within sub populations we encourage speciation. This in turn increases the likelihood of crossover between compatible segments, and hence improves the likelihood of a favorable crossover outcome. Unpublished work by the authors demonstrates that PLGP using ESP significantly outperforms constraint free crossover. Unfortunately this paper lacks adequate space to discuss these results in depth.

4 Experimental Setup

In order to compare the effectiveness of different GP methods as techniques for performing multiclass classification, a series of experiments was conducted in the important problem domain of object recognition.

4.1 Data Sets

We used three image data sets providing object classification problems of varying difficulty in the experiments. These data sets were chosen from the UCI machine learning repository [10].

The first data set is *Hand Written Digits*, which consists of 3750 hand written digits with added noise. It has 10 classes and 564 attributes. The second set is *Artificial Characters*. This data set consists of vector representations of 10 capital letters from the English language. It has a high number of classes (10), a high number of attributes/features (56), and 5000 instances in total. The third set is *Yeast*. In this data set the aim is to predict the protein localization sites based on the results of various tests. It has 10 classes, 8 attributes and 1484 instances. These tasks are highly challenging due to a high number of attributes, a high number of classes and noise in some sets.

4.2 Parameter Configurations

The parameters in table 5a are the *constant parameters*. These are the parameters which will remain constant throughout all experiments. These parameters are either experimentally determined optima, or common values whose reasonableness is well established in literature [6].

The parameters in table 5b are the *experiment specific parameters*. Each column of the table corresponds to the parameter settings for a specific experiment. Each experiment has two components, an LGP stage and a PLGP stage. In the LGP stage we determine the classification accuracy of an LGP system using programs of the specified length. In the PLGP stage we repeat our experiment but we use PLGP programs of equivalent length.

Fig. 4: Experimental Parameters

Parameter	Value
Population	500
Max Gens	400
Normal Mutation	45%
Elitism	10%
Crossover	45%
Tournament Size	5

(a) Constant

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Total Instructions	10	20	35	50	100	400
# PLGP Pieces	2	4	5	5	10	20
PLGP Piece Size	5	5	7	10	10	20

(b) Experiment Specific

We allow terminal constants in the range [-1,1], and a function set containing Addition, Subtraction, Multiplication, Protected Division, and If. The data set is divided equally into a training set, validation set, and test set, and results are averaged over 30 runs. All reported results are for performance on the test set. The fitness function is simply the number of missclassified training examples. Finally all initial programs in the population consist of randomly chosen instructions.

PLGP Program Topologies It is important to note that there are many different ways of arranging the instructions in a PLGP program. For instance a PLGP program with 10 instructions could consist of either 2 pieces of 5 instructions, or 5 pieces of 2 instructions. We refer to these different arrangements as *Program Topologies*. We lack the room to give an in depth analysis in this paper, however unpublished work by the authors shows that a large number of reasonable topologies give near optimal results. The program topologies used in our experiments have been previously determined to be optimal to within some small tolerance.

5 Results

The following graphs compare the performance of LGP with PLGP as classification techniques on the three data sets described earlier. Figure 5 compares

performance on the Hand Written Digits data set, Figure 6 compares performance on the Artificial Characters data set and Figure 7 compares performance on the Yeast data set. Each line corresponds to an experiment with programs of a certain fixed length. Program lengths vary from very short (10 instructions) to very long (400 instructions).

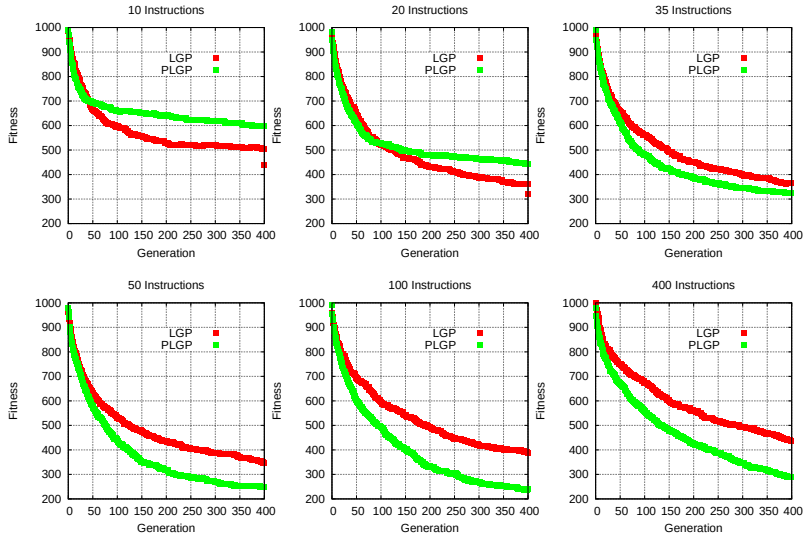

Fig. 5: LGP vs. PLGP on the *Hard Written Digits* data set

5.1 Discussion

Program Size *LGP performs at least as well as PLGP when small programs are used.* In the Hand Written Digits data set LGP significantly outperforms PLGP for programs of length 10-20. In the Artificial Characters data set the performance of LGP and PLGP is comparable for programs of length 10. In the Yeast data set the performance of LGP and PLGP is comparable for programs of length 10-35.

Short programs will not suffer from code disruption or non-effective code in the way larger programs do. This means our motivation for developing the PLGP technique does not hold for short programs. Also, dividing short programs into many pieces strips them of much of their usefulness. The power of LGP programs lies in their ability to reuse results calculated earlier in the program and stored in the registers. Dividing very short programs into many pieces means each piece will only be a handful of instructions long, rendering very short PLGP programs relatively powerless. However the flip side of this result is that short LGP programs are not what we are really interested in. Typically short programs are not powerful enough for any sort of interesting application. On difficult problems neither LGP or PLGP can achieve good performance using short programs.

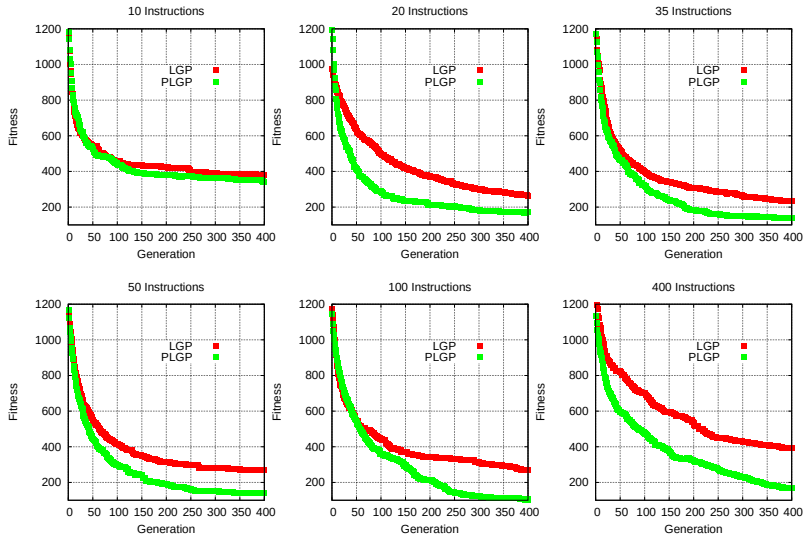

Fig. 6: LGP vs. PLGP on the *Artificial Characters* data set

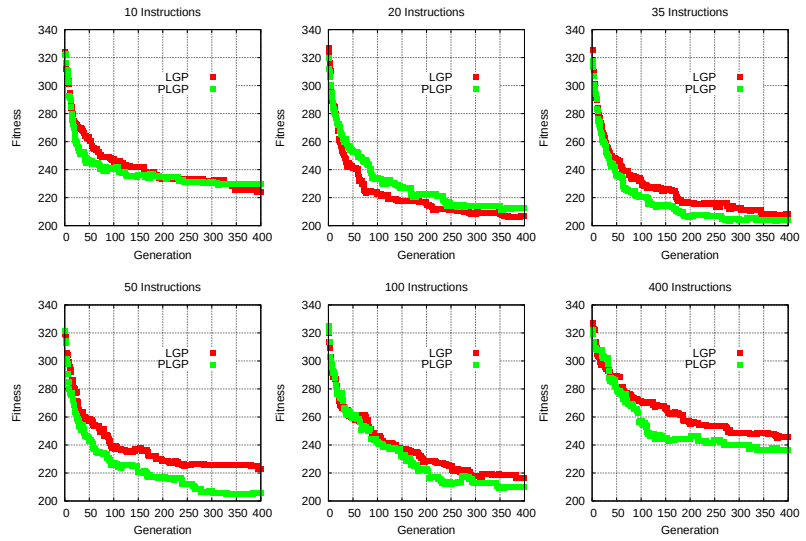

Fig. 7: LGP vs. PLGP on the *Yeast* data set

PLGP significantly outperforms LGP for longer programs. In all three data sets PLGP significantly outperforms LGP whenever program length exceeds some minimum. For the Hand Written Digits this holds for all programs larger than 20 instructions. For the Artificial Characters data set this holds for all programs larger than 10 instructions. For the yeast data set this holds for all programs larger than 35 instructions.

Longer LGP programs are prone to problems such as code disruption and non-effective code. By dividing the program into parallel pieces we can alleviate these problems. Hence it is expected that PLGP will outperform LGP when program size is large. In addition, the severity of these problems is proportional to the length of instruction sequence. Hence larger LGP programs will suffer these effects to a greater degree.

Optimal Size There is an optimal size for programs. Short programs are insufficiently expressive: it is not possible to easily *express* good solutions using only a very small number of instructions. Long programs are overly expressive: while it is possible to express good solutions the search space is too large, making it difficult to *find* good solutions. Hence there is an optimal size for LGP programs, a balance between expressive power and search space complexity.

The optimal size for PLGP programs is significantly larger that the optimal size for LGP programs. For LGP the optimal program size lies between 20-50 instructions. For PLGP the optimal program size lies between 50-400 instructions.

Optimal size is also influenced by code disruption and non effective code. These problems exacerbate the problem of search space exploration. This means that exploring large search spaces using LGP is difficult, preventing large programs from being used effectively. Conversely PLGP can more easily explore a large search space since it avoids both code disruption and non-effective code.

Optimal Performance It is clear that PLGP gives rise to better fitness solutions than LGP. This is intrinsically linked to the optimal program size for each of these two methods: PLGP has a higher optimal program size than LGP. Both program representations are equally able to *express* powerful solutions, however only PLGP is able to actually *find* these powerful solutions. By avoiding code disruption and non effective code PLGP allows us to exploit a range of powerful solutions not available to us with LGP.

6 Conclusions and Future Work

PLGP is a technique designed to minimize building block and program disruption by the addition of parallelism into the standard LGP technique. Longer LGP programs are easily disrupted since modifications to early instructions result in massive changes to program output. By executing many program parts in parallel, PLGP prevents code disruption from occurring. This allows PLGP to effectively exploit larger programs for significantly superior results. Our empirical tests support this: long PLGP programs significantly outperform long LGP programs on all data sets. In addition our results show that by exploiting the ability of PLGP to utilize large programs it is possible to obtain a significant overall improvement in performance. Both theory and results clearly demonstrate the benefits of this new parallel architecture.

PLGP offers many exciting opportunities for future work. PLGP programs are naturally suited to caching. They consist of several dissociated pieces and at each generation only a single piece is modified. Therefore by caching the output of each piece it should be possible to cut execution time by an order of magnitude.

When summing the pieces of a PLGP program it should be possible to introduce piece weights. By replacing the sum with a *weighted sum*, it is possible to deterministically improve performance by optimizing these weights. This weighted PLGP could either be used during evolution, or it could be applied post evolution to the best individual as a final optimization procedure.

References

1. Andre, D., Koza, J.R.: A parallel implementation of genetic programming that achieves super-linear performance. Information Sciences 106(3-4), 201–218 (1998)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications. San Francisco, Calif. : Morgan Kaufmann Publishers; Heidelburg : Dpunktverlag (1998)
3. Brameier, M., Banzhaf, W.: Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation, Springer (2007)
4. Fogelberg, C., Zhang, M.: Linear genetic programming for multi-class object classification. In: AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Proceedings. vol. 3809, pp. 369–379 (2005)
5. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Ann Arbor : University of Michigan Press; Cambridge, Mass. : MIT Press (1975)
6. Koza, J.R.: Genetic programming : on the programming of computers by means of natural selection. Cambridge, Mass. : MIT Press, London, England (1992)
7. Krawiec, K., Bhanu, B.: Visual learning by evolutionary and coevolutionary feature synthesis. IEEE Transactions on Evolutionary Computation 11(5), 635–650 (Oct 2007)
8. Olague Caballero, G., Romero, E., Trujillo, L., Bhanu, B.: Multiclass object recognition based on texture linear genetic programming. In: Applications of Evolutionary Computing, EvoWorkshops2007. LNCS, vol. 4448, pp. 291–300 (11-13 Apr 2007)
9. Olaguea, G., Cagnoni, S., Lutton, E.: (eds.) special issue on evolutionary computer vision and image understanding, pattern recognition letters.**27**(11) (2006)
10. S. Hettich, C.B., Merz, C.: UCI repository of machine learning databases (1998), http://www.ics.uci.edu/∼mlearn/MLRepository.html
11. Zhang, M., Ciesielski, V.B., Andreae, P.: A domain-independent window approach to multiclass object detection using genetic programming. EURASIP Journal on Applied Signal Processing 2003(8), 841–859 (Jul 2003), special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis
12. Zhang, M., Gao, X., Lou, W.: A new crossover operator in genetic programming for object classification. IEEE Transactions on Systems, Man and Cybernetics, Part B 37(5), 1332–1343 (Oct 2007)

13. Zhang, M., Smart, W.: Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. Pattern Recognition Letters 27(11), 1266–1274 (Aug 2006), evolutionary Computer Vision and Image Understanding