Proceedings of the First On-line Workshop on Soft Computing, August 1996 (revised version).

Some Steps Towards a Form of Parallel Distributed Genetic Programming

Riccardo Poli
School of Computer Science
The University of Birmingham
Birmingham B15 2TT, UK
R.Poli@cs.bham.ac.uk

***Abstract*—Genetic Programming is a method of program discovery consisting of a special kind of genetic algorithm capable of operating on non-linear chromosomes (parse trees) representing programs and an interpreter which can run the programs being optimised. This paper describes PDGP (Parallel Distributed Genetic Programming), a new form of genetic programming which is suitable for the development of fine-grained parallel programs. PDGP is based on a graph-like representation for parallel programs which is manipulated by crossover and mutation operators which guarantee the syntactic correctness of the offspring. The paper describes these operators and reports some preliminary results obtained with this paradigm.**

I. Introduction

Genetic Programming (GP) is an extension of Genetic Algorithms (GAs) in which the structures that make up the population to be optimised are not fixed-length character strings that encode possible solutions to a problem, but *programs* that, when executed, *are* the candidate solutions to the problem [11; 12].

Programs are expressed in GP as parse trees, rather than as lines of code. For example, the simple expression `max(x * y, 3 + x * y)` would be represented as shown in Figure 1. The basic search algorithm used in GP is a classical GA with mutation and crossover specifically designed to handle parse trees.

The set of possible internal (non-leaf) nodes used in GP parse trees is called *function set*, $\mathcal{F} = \{f_1, \cdots, f_{N_F}\}$. All functions of $\mathcal{F}$ have *arity* (the number of arguments) greater than one. The set of terminal (leaf) nodes in the parse trees is called *terminal set* $\mathcal{T} = \{t_1, \cdots, t_{N_T}\}$. Table 1 shows some typical functions and terminals used in GP.

This form of GP has been applied successfully to a large number of difficult problems like automated design, pattern recognition, robot control, symbolic regression, music generation, image compression, image analysis, etc. [11; 12; 9; 10; 1; 17].

When appropriate terminals, functions and/or interpreters are defined, standard GP can go beyond the production of sequential tree-like programs. For example using cellular encoding GP can be used to develop (grow) structures, like neural nets [5; 6] or electronic circuits [14; 13], which can be thought of as performing some form of parallel computation. Also, in conjunction with an interpreter implementing a parallel virtual-machine, GP can be used to develop certain kinds of parallel programs [3] or to translate sequential programs into parallel ones [19]. (The development of parallel programs should not be confused with the parallel implementations of GP, which are essentially methods of speeding-up the genetic search of standard tree-like programs [2; 4; 16; 8; 18]. These methods are usually based on the use of multiple processors, each one handling a separate population, a subset of fitness evaluations or a subset of fitness cases.)

This paper describes PDGP (Parallel Distributed Genetic Programming), a new form of genetic programming which is suitable for the development of fine-grained or medium-grained parallel programs. PDGP is based on a graph-like representation for parallel programs and genetic operators which guarantee the syntactic correctness of the offspring. In the following sections the representation and operators used in PDGP are described and some preliminary results obtained with this paradigm are reported.

II. Representation

Taking inspiration from the parallel distributed processing performed in neural nets, we represent fine-grained parallel programs as graphs with labelled nodes and oriented links. The nodes are the functions and terminals used in the program while the links determine which arguments are used by each function-node when it is next evaluated. Figure 2 shows an example of

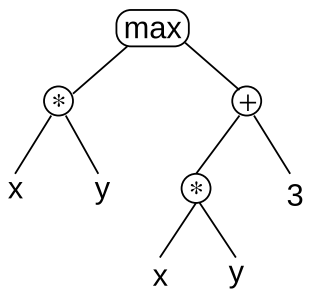

Fig. 1. Parse-tree representation of the expression `max(x * y, 3 + x * y)`.

Functions	
Kind	*Examples*
Arithmetic	+, *, /
Mathematical	sin, cos, exp
Boolean	AND, OR, NOT
Conditional	IF-THEN-ELSE, IFLTE
Looping	FOR, REPEAT

Terminals	
Kind	*Examples*
Variables	x, y
Constant values	3, 0.45
0-arity functions	rand, go_left
Random constants	random

Table 1. Typical functions and terminals used in genetic programming.

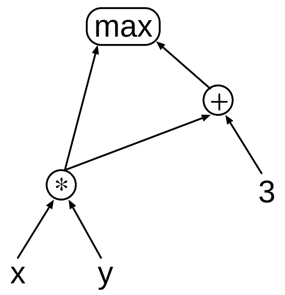

Fig. 2. Graph-like representation of the expression `max(x * y, 3 + x * y)`.

a parallel distributed program represented as a graph. The program implements the same function as the one shown in Figure 1, i.e. `max(x * y, 3 + x * y)`. Its execution should be imagined as a "wave of computations" starting from the terminals and propagating upwards along the graph.

This tiny-scale example shows that graph-like representations of programs can be more compact (in term of number of nodes) and more efficient (the sub-expression `x * y` is be computed only once instead of twice) than tree-like representations. However, the direct handling of graphs within a genetic algorithm presents some problems.

Several direct representations for graphs exist in graph theory. For each of them one could imagine operators that select a random sub-graph in one parent and then swap it with a "properly" selected sub-graph in the other parent or a "properly" generated random sub-graph (by "proper" sub-graph we mean a sub-graph with the correct number of input and output links). However, as shown by the considerable work done in the field of neural networks, it is not easy to produce good genetic operators for direct graph encodings. In particular it is hard to produce a crossover operator such that: a) when parents share a common characteristic their offspring inherit such a characteristic, b) when parents have different characteristic their offspring can inherit both such characteristics, c) every offspring is a valid solution, d) crossover is efficient.

Indirect graph representations, like cellular encoding [5; 6; 14; 13] or edge encoding [15], do not suffer from this problem as the standard GP operators can be used on them. However, such representations require an additional genotype-to-phenotype decoding step before the interpretation of the graphs being optimised can start, as the search is not performed in the space of possible graphs, but in the space of sequential programs that produce graphs. When the fitness function involves complex calculations with many fitness cases the decoding step can have a limited relative effect on the evaluation of each individual in terms of computational cost. However, the meta-encoding of the graphs seems to make the search more costly by increasing the total number of individuals that must be evaluated (see for example the comparison between cellular and direct encodings of neural nets in [7]).

PDGP uses a direct representation of graphs which, although not general, allows the definition of crossover operators which respect all the criteria listed above (in particular efficiency and offspring validity).

The representation is based on the idea of assigning each

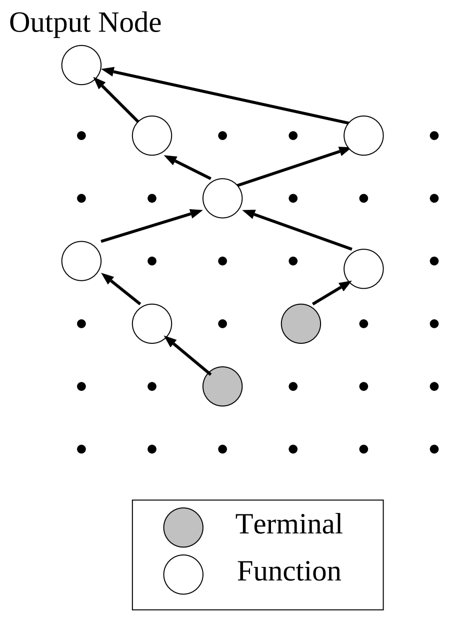

Fig. 3. Grid-based representation of graphs representing programs in PDGP.

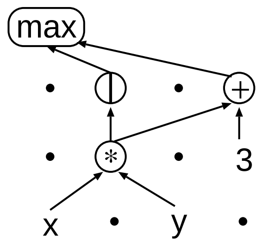

Fig. 4. Grid-like representation of the expression `max(x * y, 3 + x * y)`.

node in the graph to a physical location in a multi-dimensional (evenly spaced) grid with a pre-fixed (regular or irregular) shape and limiting the connections between nodes to be upwards. Also, connections can only be established between nodes belonging to adjacent rows, like the connections in a standard feed-forward multi-layer neural network. This representation for parallel distributed programs is illustrated in Figure 3, where we assumed that the program has a single output at coordinates (0,0) (the y axis is pointing downwards) and that the grid is two-dimensional and includes $6 \times 6 + 1$ cells.[1]

By adding the identity function (i.e. a wire or pass-through node) to the function set, any parallel distributed program (i.e. any directed acyclic graph) can be rearranged so that it can be described with this grid-like graph representation. For example, the program shown in Figure 2 can be transformed into the layered network in Figure 4.

In this representation it is possible to describe any program by listing the following parameters for each node: 1) label, 2) the coordinates of the node, 3) the horizontal displacement of the nodes in the previous layer (if any) whose value is used as arguments to compute the value of the node. For example, the program in Figure 4 could be described by the following list:

[1] In this work we have adopted the convention that the first row of the grid includes as many cells as the number of outputs of the program.

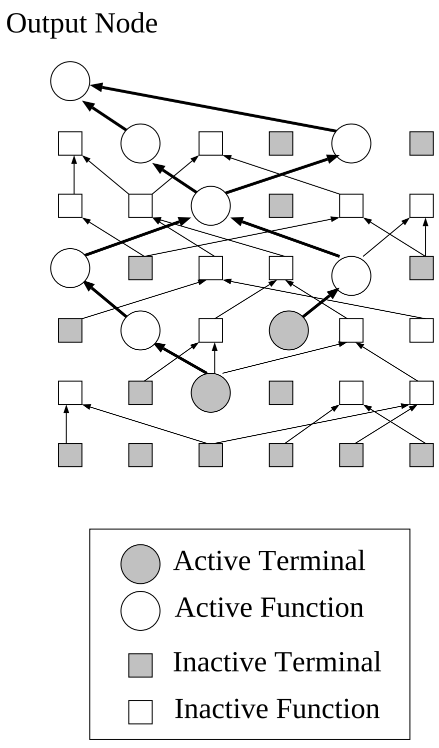

Fig. 5. Intron-augmented representation of programs in PDGP.

```
max (0,0) +1 +3
I   (0,1)  0
+   (3,1) -2  0
*   (1,2) -1 +1
3   (3,2)
x   (0,3)
y   (2,3)
```

The representation described above will allow us to define very efficient forms of crossover and mutation. However, in order to study all the possibilities offered by a our network-based representation of programs, we decided to expand the representation to explicitly include introns ("unexpressed" parts of code). In particular we assumed that once the size and shape of the grid is fixed, a function or a terminal is associated to *every* node in the grid, i.e. also to the nodes that are not directly or indirectly connected with the output. For example, the program shown in Figure 3 could have an expanded representation like the one in Figure 5.

It should be noted that inactive nodes can receive as input the results computed by both active and inactive nodes. It should be also noted that by using an array with the same topology as the grid, it is possible to omit the coordinates from the representation of each node and store in each cell of the array only the label and the input connections. This can lead to more efficient implementations of the genetic operators described in the next section.

III. Genetic Operators

With the representations described in the previous section, several kinds of crossover, mutation and initialisation strategy can be defined.

A. Random Program Generation

In standard GP, it is possible to obtain balanced or unbalanced random initial trees with the "full" method and the "grow" method, respectively. Similarly, in PDGP it is possible to obtain "balanced" or "unbalanced" random graphs depending on whether we allow terminals to be at a pre-fixed maximum distance from the output node(s) only or whether they can occur anywhere in the initial programs.

In PDGP the generation of random programs can proceed in several ways, depending whether introns are used or not. If introns are used, then the grid can be filled with random functions and terminals. When functions are inserted, a corresponding number of random input-links are also generated. Alternatively, it is possible to build random graphs recursively (like in standard GP) starting from the output nodes and selecting random functions (with their input links) or terminals depending on kind of graphs we want to build (balanced or not) and on the depth reached with recursion.

B. Crossover

The first crossover operator, which we call *Sub-graph Active-Active Node (SAAN) crossover*, works as follows:

1. A random active node is selected in the first parent.
2. The sub-graph including all the active nodes which are used to compute the output value of the selected node is extracted and its hight h and width w is determined.
3. An active node in the second parent is selected such that its y coordinate is compatible with the hight h of the sub-graph, i,e. $y_{max} - y > h$.
4. The sub-graph is inserted in the second parent to generate the offspring. If the coordinate x of the insertion node in the second parent is not compatible with the width of the sub-graph, the sub-graph is wrapped around.

An example of SAAN crossover is shown in Figure 6.

The idea behind this form of crossover is that connected sub-graphs are functional units whose output is used by other functional units. Therefore, by replacing a sub-graph with another sub-graph, we tend to explore different ways of combining the functional units discovered during evolution. So, sub-graphs act as building blocks. It should be noted that in SAAN crossover inactive nodes play no role (for this reason they are not shown in the figure).

Several different forms of crossover can be defined by modifying SAAN crossover. Here is a list of the operators we have tried (more could be defined):

- *Sub-Sub-graph Active-Active Node (SSAAN) Crossover* uses an incomplete sub-graph which includes only part of the active nodes used to compute the output value of the crossover point selected in the first parent. This form of crossover requires the presence of introns to avoid the generation of invalid programs (in which some functions have

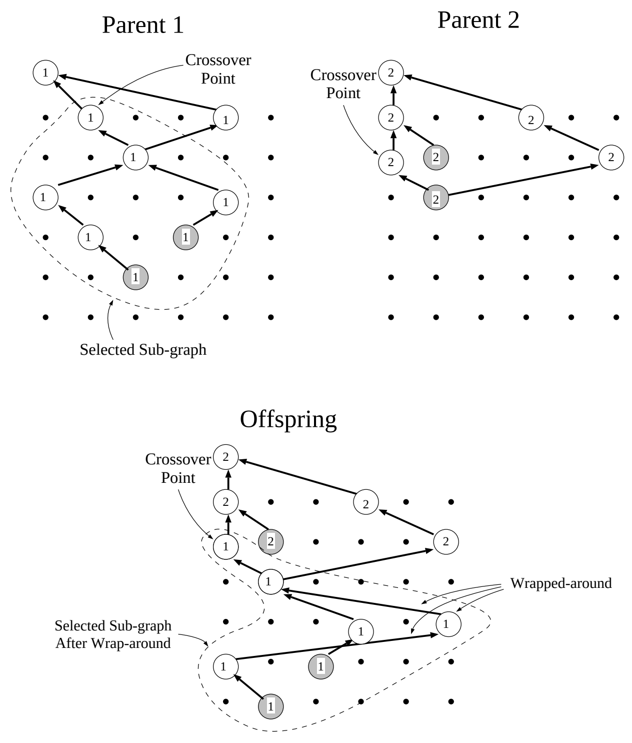

Fig. 6. Sub-graph active-node crossover (inactive nodes are not shown).

not enough arguments).

- *Sub-graph Inactive-Active Node (SIAN) Crossover* selects the first crossover point among both the active and the inactive nodes.
- *Sub-sub-graph Inactive-Active Node (SSIAN) Crossover* is like SIAN crossover but uses (possibly) incomplete sub-graphs.
- *Sub-graph Active-Inactive Node (SAIN) Crossover* selects the second crossover point among both the active and the inactive nodes.
- *Sub-graph Inactive-Inactive Node (SIIN) Crossover* selects both crossover points among all nodes (active or inactive).

C. Mutation

The standard GP technique of defining mutation as the swapping of a randomly selected sub-tree in an individual with a new randomly generated tree can be naturally applied in PDGP as well. In the current implementation of PDGP we use a less efficient mutation operator which is however equivalent to this (we define mutation as the crossover of an individual with a randomly generated new individual, which is then discarded). We call this form of mutation, *global mutation*.

We have also used another form of mutation, which we call *link mutation*, which makes local modifications to the connection topology of the graph. Link mutation selects a random function node and then a random input link of such a node and alters the offset associated to the link, i.e. it changes a connection in the graph. Other forms of mutation can easily be defined (e.g. node mutation).

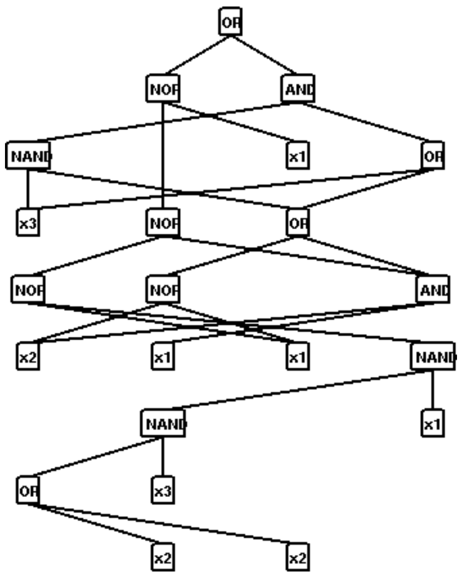

Fig. 7. Parallel distributed program implementing the even-3 parity function.

IV. Experimental Results

In this section we report on some preliminary experimental results obtained by applying PDGP to the even-3 parity problem. The problem consists of finding the Boolean function of three arguments, x1, x2, and x3, whose output is 1 if an even number of arguments is set to 1, 0 otherwise.

It should be noted that this is a relatively simple problem which has been used extensively to test standard GP and that can be solved in a relatively short time. These two properties have allowed us to study the behaviour of PDGP in thousands of runs and to make a comparison with the results of standard GP reported in the literature.

In all our experiments we used the function set $\mathcal{F}$={AND, OR, NAND, NOR, I} (where I is the identity function) and the terminal set $\mathcal{T}$={x1, x2, x3}. The population size was P=1,000 individuals, the maximum, number of generation was G=50, and the crossover probability was 0.7. The GA used tournament selection with tournament size 7.

Figure 7 shows a typical solution to the even-3 parity problem obtained in one run of PDGP. The parameters of the run were: grid with 4 columns and 8 rows, “grow” initialisation method, link mutation with probability 0.02 and SIAN crossover. The figure shows the actual output produced by our Pop-11 implementation of PDGP.[2] Figure 8 shows the expanded representation of the program which contains also some introns. (It should be noted that in both figures the output node, which, having coordinates (0,0), should be in the top-left corner, is actually centred horizontally for displaying purposes.)

In order to assess the behaviour of PDGP on this problem we tested it with different choices of parameters and operators, performing 20 runs for each setting (with different initialisations of the random number generator), 140 experiments in total (2,800 runs). The parameters we varied (in all possible combinations) were: 1) the probability of mutation $pmut$ (0, 0.02 and 0.04), 2)

[2] Pop-11 is an AI language with features similar to Lisp, which is quite widespread in the UK where it was originally developed.

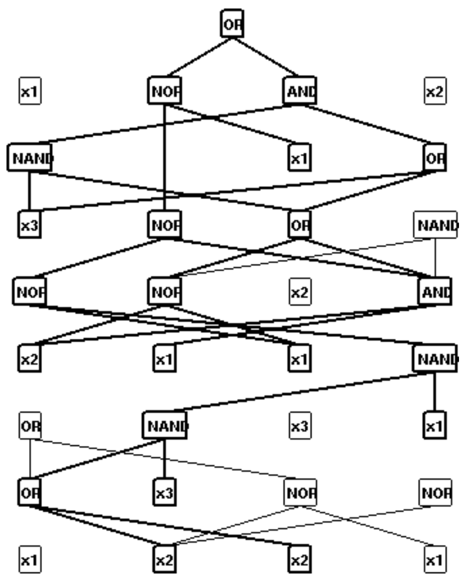

Fig. 8. Complete (intron-augmented) representation of the program in Figure 7.

the initialisation procedure ("full" vs. "grow"), 3) the crossover operator (SAAN vs. SIAN), 4) the mutation operator (link vs. global) and 5) the number of rows and columns in the grid (4×2, 4×4, 6×4, 6×6, 8×4, 8×6 and 8×8).

One of the criteria we used to assess the performance of PDGP was the computational effort E used in the GP literature (E is the number of fitness evaluations necessary to get a correct program, in multiple runs, with probability 99%). As the effort of evaluating each individual (at least on a sequential machine) depends on the number of nodes it includes, we also used as a criterion the total number of nodes N to be evaluated in order to get a solution with 99% probability. As in these experiments we used introns which, for simplicity of implementation, were evaluated even if they did not contribute to the output of the program, the parameter N was evaluated as E times the number of nodes in the grid.

The best results in terms of fitness evaluations were obtained using a grid with 8 rows and 8 columns with global mutation (pmut=0.02), "full" initialisation, SAAN crossover. With these settings E=25,000 and N=1,625,000. This compares extremely favourably with the results described by Koza in [11; 12] in terms of fitness evaluations (Koza obtained E=96,000 for P=16,000 and E=80,000 for P=4,000). This result is particularly interesting considering the small size of the populations used in this work (P=1,000).

It should be noted that this level of performance is not a statistical accident. For example, 15 experiments out of 140 showed values of $E < 50,000$. It is interesting to note that in most of such experiments PDGP was using 8 rows and 6 or 8 columns. This seems to suggest that (relatively) large grids make the search easier. This would seem logical considering that PDGP tend to behave more and more like standard GP, i.e. to develop trees, as the size of the grid increases and that crossover using sub-graphs can be more disruptive than crossover using trees.

However, if we take the best results in terms of total number of nodes to be evaluated the picture changes dramatically. The best result, N=702,000 with E=78,000, was obtained using a 4×2 grid (this is confirmed by the fact that the nine best experiments had all the same grid configuration). The result is surprising as the effort N of finding more constrained (i.e. more parsimonious) solutions is a fraction of that of finding less constrained (tree-like) ones.

In comparative terms in this experiment PDGP has solved the even-3 parity problem with approximately the same number of fitness evaluations as standard GP while producing solutions with 9 or fewer nodes. As the average complexity of the solutions described in [12] was 44.6 nodes, this seems to be a considerable saving.

V. Conclusions

In this paper we have presented PDGP, a new form of genetic programming which is suitable for the automatic discovery of parallel network-like programs.

The grid-like representation of programs used in PDGP allowed us to develop efficient forms of crossover and mutation. By changing the size, shape and dimensionality of the grid, this representation allows a fine control on the size and shape of the programs being developed. For example it is possible to control the degree of parallelism by changing the number of columns.

The parallel programs developed by PDGP are fine-grained, but the representation used is also suitable for the development of medium-grained parallel programs via the use of Automatically Defined Functions (the results, not reported, of preliminary experiments with ADFs are quite promising).

In this paper we have used a very simple form of program interpretation similar to the propagation of activation in the neurons of feed-forward neural nets. This is quite suitable for programs which do not include functions or terminals with side-effects. If this is the case, our grid-based representation of programs can be directly mapped into the nodes of some kinds of fine-grained parallel machines. This could lead to a very efficient evaluation of fitness cases as PDGP programs could produce a new result at every clock tick. PDGP programs (possibly developed with some additional constraints) could also be used to define the functions of field programmable gate arrays. This, for example, could lead to new ways of doing research in the field of evolvable hardware.

Some preliminary work (not reported here) has been done to address the problems generated by the use of terminals and functions with side effects. At the moment, the most promising idea seems to be the use of two implementations for each different kind of node (macro, function or terminal) in conjunction with a top-down flow of control signals and a bottom-up flow of results. In the first implementation a node, if not yet evaluated, requests the evaluation of its arguments, waits until the results are available, computes and stores its value and returns it. If the node had been evaluated already, the stored value is returned immediately and no computation is performed. In the second implementation a node would always request the re-evaluations of its arguments.

The results shown by PDGP on the even-3 parity problem are promising. However, in general no final conclusion should be drawn from the performance obtained by a machine learning paradigm on a single problem. More research will be necessary to understand if PDGP really outperforms standard GP and if a set of parameters which work for large classes of problems exists.

Finally, it should be noted that PDGP has quite a broad applicability which goes beyond programming. PDGP can be seen as a paradigm to optimise directed acyclic graphs which need not be interpreted as programs: they can be interpreted as designs, semantic nets, neural network topologies (links can be labelled with weights), etc. It is also possible to imagine ways in which some of the constraints (like acyclicity) imposed on the graphs produced by PDGP could be removed without reducing the search efficiency significantly.

Acknowledgements

The author wishes to thank Aaron Sloman, Brian Logan and all the members of the EEBIC (Evolutionary and Emergent Behaviour Intelligence and Computation) group for useful discussions and comments. This research is partially supported by a grant under the British Council-MURST/CRUI agreement.

References

[1] *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford University, July 1996. Stanford Bookstore.

[2] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17. MIT Press, Cambridge, MA, USA, 1996.

[3] Forrest H. Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 30, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[4] Dimitris C. Dracopoulos and Simon Kent. Speeding up genetic programming: A parallel BSP implementation. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 421, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[5] F Gruau and D. Whitley. Adding learning to the cellular development process: a comparative study. *Evolutionary Computation*, 1(3):213–233, 1993.

[6] Frederic Gruau. Genetic micro programming of neural networks. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 24. MIT Press, 1994.

[7] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 81, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[8] Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17. MIT Press, Cambridge, MA, USA, 1996.

[9] K. E. Kinnear, Jr., editor. *Advances in Genetic Programming*. MIT Press, 1994.

[10] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Proceedings of the First International Conference on Genetic Programming*, Stenford University, July 1996. MIT Press.

[11] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[12] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Pres, Cambridge, Massachusetts, 1994.

[13] John R. Koza, David Andre, Forrest H. Bennett III, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 132, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[14] John R. Koza, Forrest H. Bennett III David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 123, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[15] S. Like and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 117–124, July 1996.

[16] Mouloud Oussaidene, Bastien Chopard, Olivier V. Pictet, and Marco Tomassini. Parallel genetic programming: An application to trading models evolution. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 357, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[17] Riccardo Poli. Genetic programming for image analysis. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 363, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[18] Kilian Stoffel and Lee Spector. High-performance, parallel, stack-based genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 224, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[19] Paul Walsh and Conor Ryan. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 406, Stanford University, CA, USA, 28–31 July 1996. MIT Press.