Discovery of Symbolic, Neuro-Symbolic and Neural Networks with Parallel Distributed Genetic Programming

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

Abstract

Parallel Distributed Genetic Programming (PDGP) is a new form of genetic programming suitable for the development of parallel programs in which symbolic and neural processing elements can be combined in a free and natural way. This paper describes the representation for programs and the genetic operators on which PDGP is based. Experimental results on the XOR problem are also reported.

1 Introduction

Genetic Programming (GP) is a method of program discovery consisting of a special kind of genetic algorithm (GA) capable of handling programs and an interpreter to run them [3].

Programs are expressed in GP as parse trees. For example, the expression `max(x*y,3+x*y)` would be represented as shown in Figure 1(a). The set of possible internal nodes used in GP parse trees is called *function set*, $\mathcal{F}$. $\mathcal{F}$ can include arithmetic operators, mathematical and Boolean functions, etc. The set of terminal nodes in the parse trees is called *terminal set* $\mathcal{T}$. $\mathcal{T}$ can include variables, constants, etc. The basic search algorithm used in GP is a GA with mutation and crossover specifically designed to operate on parse trees.

GP has been applied successfully to induce *sequential programs* and solve a large number of difficult problems (see [5] for an extensive bibliography). However, only a very small number of results have been reported where GP, appropriately modified, has gone beyond the production of sequential tree-like programs. For example, using cellular encoding GP has been used to develop neural nets [2] and electronic analogue circuits [4], while using interpreters implementing parallel virtual machines it has been used to develop special kinds of parallel programs [1, 7].

This paper describes Parallel Distributed Genetic Programming (PDGP), a new form of GP which is specialised in the development of parallel programs in which symbolic, numeric and neural processing elements can be combined in a totally free and natural way. 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 PDGP is described and some results on the XOR problem are reported.

2 Representation

In PDGP we represent (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.

Figure 1(b) shows how `max(x*y,3+x*y)` can be represented in a parallel distributed form as a graph. The execution of the program should be imagined as a "wave of computations" starting from the terminals and propagating upwards along the graph, like the updating of the activations of the neurons in a multi-layer perceptron.

Graph-like representations of programs can be more compact (in term of number of nodes) and more efficient than tree-like representations (e.g. in Figure 1(b) the subexpression `x*y` is computed only once). However, the direct handling of graphs within a GA presents some problems.

PDGP uses a direct representation of graphs which allows the definition of crossover operators which always produces valid offspring in a very efficient way. The representation is based on the idea of assigning each node in the graph to a physical location in a grid of pre-fixed shape and limiting the connections between nodes to be forward and between adjacent layers only.

By adding the identity function (i.e. a 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 in Figure 1(b) can be transformed into the network in Figure 1(c).

In order to study all the possibilities offered by our representation, we decided to expand it to explicitly include introns ("unexpressed" parts of code) by associating a function or a terminal to *every* node in the grid, i.e. also to the nodes not directly or indirectly connected with

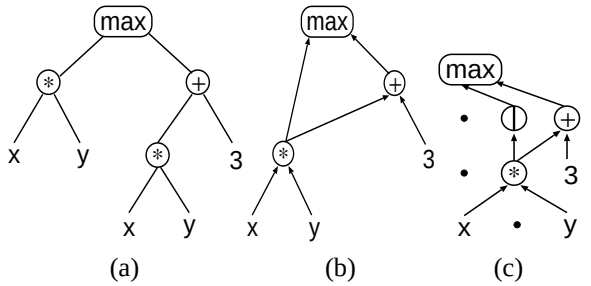


Figure 1: Parse-tree representation of the expression `max(x*y,3+x*y)` (a), the corresponding graph-like representation (b), and its PDGP grid-based representation (c).

the output. In some experiments we also added numeric labels to links to allow the direct development of neural networks.

3 Genetic Operators

Several kinds of crossover and mutation can be defined in PDGP. The crossover operator most similar to the one used in GP is called *Sub-graph Active-Active Node (SAAN) crossover*. It works as follows: a) a random active node is selected in the first parent, b) the sub-graph including all the active nodes which are used to compute the output of the selected node is extracted and its height and width is determined, c) an active node in the second parent is selected such that its vertical position is compatible with the height of the sub-graph, d) the sub-graph is inserted in the second parent to generate the offspring. In this last phase, if the horizontal position 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 2.

The idea behind the SAAN 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.

Other forms of crossover can be defined by modifying SAAN crossover. In this paper we have adopted the *Sub-graph Inactive-Active Node (SIAN) crossover* in which the crossover point in the second parent is randomly selected among the active nodes, while the crossover point in the first parent is randomly chosen among all nodes in the grid.

In standard GP, mutation consists of swapping a random sub-tree in an individual with a new randomly generated tree. This technique, which we call *global mutation*, can easily be extended to PDGP. We also use another form

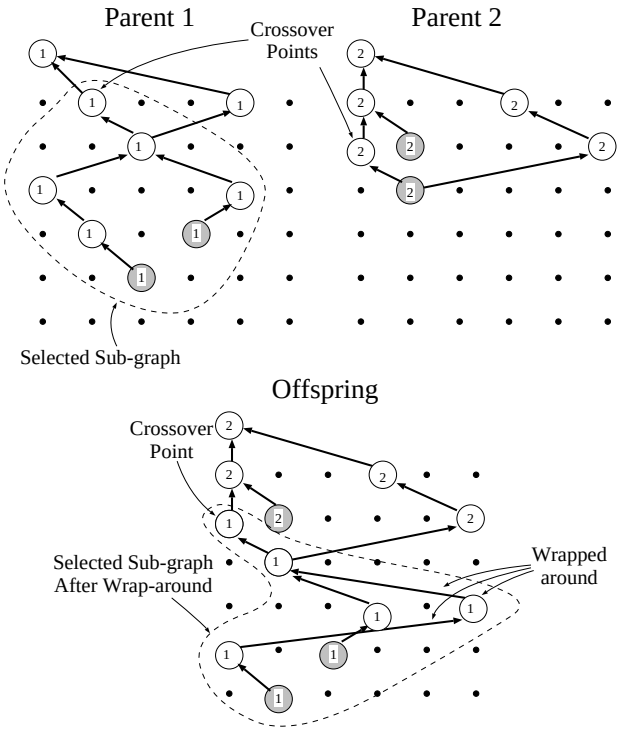


Figure 2: Sub-graph active-active node (SAAN) crossover.

of mutation, *link mutation*, which makes local modifications to the connection topology of the graph (see [6] for more details). Mutation operators are applied after recombination and cloning.

4 Experimental Results

In this section we report on some experimental results obtained by applying PDGP to the problem of finding parallel distributed programs implementing the XOR function.

In the experiments, the population included 200 individuals, the maximum number of generation was 20, the crossover probability was 0.7, the global mutation probability was 0.25 and the link mutation probability was 0.25. The GA used tournament selection with tournament size 7. The other parameters were: "grow" initialisation method and SIAN crossover. The fitness of a solution was the number of entries in the XOR truth-table it correctly predicted.

Logic solutions In these experiments we used the function set $\mathcal{F}$={AND, OR, NAND, NOR, I} (I is the identity function) and the terminal set $\mathcal{T}$={x1, x2}.

Figure 3 shows two typical solutions obtained by PDGP. In the figure the active nodes and the active links have been drawn with thick lines and the output node has been centred horizontally for displaying purposes.

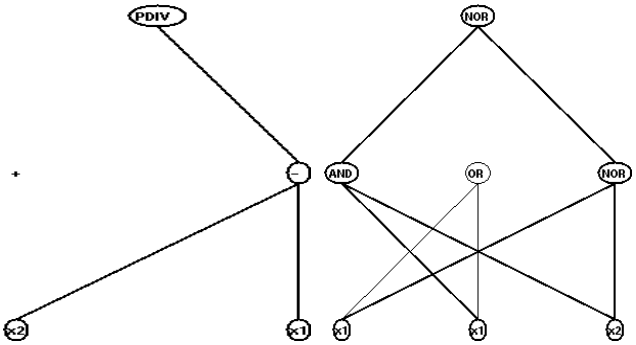


Figure 3: Symbolic networks implementing the exclusive-or function with Boolean processing elements.

In order to study the behaviour of PDGP on this problem we performed 20 runs (with different seeds for the random number generator) with three different grid sizes: 2×2, 2×3 and 3×4.

To assess the performance of PDGP we used two criteria: 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%) and the total number of nodes N to be evaluated in order to get a solution with 99% probability. We (over)estimated N by multiplying E by the number of nodes in the grid. The results are summarised in the first column of Table 1.

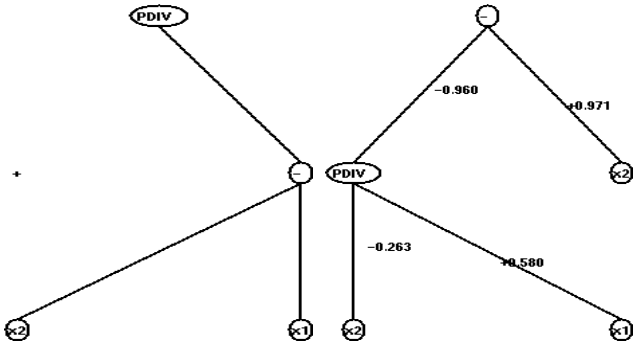
These results indicate that increasing the size of the grid reduces considerably the number of fitness evaluations necessary to get a solution. This seems reasonable considering that smaller grids impose harder constraints on the search. However, if we look at the values of N the advantage of larger grids is not so clear. In fact, the slightly greater number of node evaluations required by smaller grids is balanced by the fact that they produce better solution in terms of size, execution speed and generalisation.

Algebraic solutions In these experiments we used $\mathcal{F}$={+, -, *, PDIV, I} (PDIV is the protected division, which returns its first argument if the second is 0) and $\mathcal{T}$={x1, x2}. In these and the following experiments the output is considered to be 1 if it is greater than 0.5, 0 otherwise.

Figure 4(a) shows a typical solution to the XOR problem obtained using algebraic operators.

In 20 runs with three different grid sizes we obtained the results in the second column of Table 1 which indicate that algebraic operators make the search easier and that larger grids reduce the number of fitness evaluations but not the number of node evaluations.

Neuro-Algebraic solutions In these experiments we used the same function and terminal sets as in the previous section, but we added random weights in the range $[-1, 1]$ to



(a) (b)

Figure 4: Algebraic (a) and neuro-algebraic (b) realisations of the XOR function.

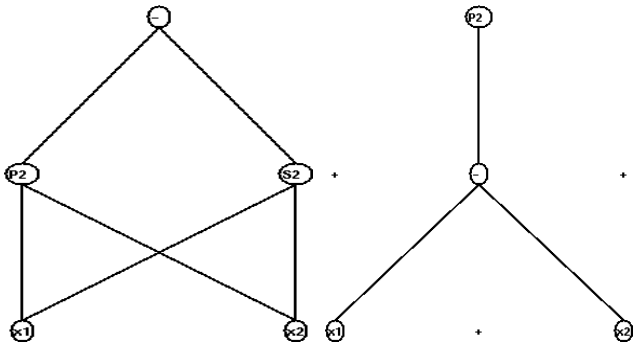


Figure 5: Weight-less neural networks implementing the XOR function.

the links. The weights act as pre-multipliers for the arguments of the functions in $\mathcal{F}$.

Figure 4(b) shows a typical solution to the XOR problem obtained using neuro-algebraic operators.

In 60 runs with this setting we obtained the results in the third column of Table 1 which indicate that the use of random weights makes the search much harder, at least without specialised weight-altering operators. Also, again increasing the size of the grid reduces the number of fitness evaluations but not the number of node evaluations.

Weight-less Neural Solutions In these experiments we used the function set $\mathcal{F}$={+, −, S2, S3, P2, P3, I} where + and − are introduced to simulate linear neurons, S2 and S3 are neurons whose inputs are added and then passed through a sigmoid activation function, and P2 and P3 are Π neurons which compute the product of their inputs. The terminal set included also a random constant generator, to create biases in the range [-1.0,+1.0]. The links had no weights.

Figure 5 shows two XOR implementation obtained by PDGP, while the fourth column of Table 1 reports the results obtained in 60 runs.

These results suggest that the use of neurons instead of Boolean or algebraic nodes makes the search harder. The

	Logic		Algebraic		Neuro-algebraic		Weight-less neural		Neural	
Grid size	E	N	E	N	E	N	E	N	E	N
2×2	5,200	26,000	2,400	12,000	9,600	48,000	10,200	54,000	342,000	1,710,000
2×3	4,200	29,400	2,800	19,600	12,000	84,000	6,800	47,600	378,000	2,646,000
3×4	1,600	20,800	1,600	20,800	7,000	91,000	6,000	78,000	46,200	600,600

Table 1: Computational effort necessary to solve the XOR problem with PDGP using different representations.

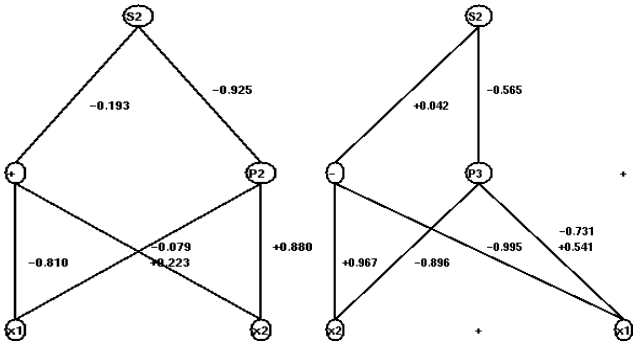


Figure 6: Neural XOR realisations.

reason for this might be the limited expressive power of neurons with respect to other classes of functions, at least for Boolean classification problems.

Neural Solutions In these experiments we used the same function and terminal sets as in the previous section but we added weights to the links.

Figure 6 shows two typical solutions to the XOR problem obtained with this setting. The results obtained in 60 runs are summarised in the last column of Table 1.

As expected, the combination of the negative effects of weights and neural processing elements produces a considerable degradation of the performance of PDGP. However, in this case a large grid is a big relative advantage.

5 Conclusions

PDGP is a new form of GP which is suitable for the automatic discovery of parallel network-like programs in which symbolic and sub-symbolic primitives can be combined in a free and natural way. In this paper we have presented PDGP and studied its representational capabilities using a very simple problem: learning the XOR function.

The results described here, along with recent results obtained on a variety of more complex problems [6], are very promising as they clearly show how PDGP can explore entirely new spaces of programs in which neural nets and classical tree-like programs are just special cases.

Acknowledgements

The author thanks 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] F. H. Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In *Proc. of Genetic Programming'96*, pages 30–38, Stanford University, July 1996. MIT Press.

[2] F. Gruau. Genetic micro programming of neural networks. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 24, pages 495–518. MIT Press, 1994.

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

[4] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In *Proc. of Genetic Programming'96*, pages 123–131, Stanford University, July 1996. MIT Press.

[5] W. B. Langdon. A bibliography for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter B, pages 507–532. MIT Press, Cambridge, MA, USA, 1996.

[6] R. Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, The University of Birmingham, September 1996.

[7] A. Teller and M. Veloso. Neural programming and an internal reinforcement policy. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 186–192, Stanford University, July 1996. Stanford Bookstore.

[8] P. Walsh and C. Ryan. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In *Proc. of Genetic Programming'96*, pages 406–409, Stanford University, July 1996. MIT Press.