

---

# Evolution of Graph-like Programs with Parallel Distributed Genetic Programming

---

**Riccardo Poli\***

School of Computer Science  
The University of Birmingham  
Birmingham B15 2TT, UK

## Abstract

Parallel Distributed Genetic Programming (PDGP) is a new form of Genetic Programming (GP) suitable for the development of programs with a high degree of parallelism. Programs are represented in PDGP as graphs with nodes representing functions and terminals, and links representing the flow of control and results. The paper presents the representations, the operators and the interpreters used in PDGP, and describes experiments in which PDGP has been compared to standard GP.

## 1 Introduction

In Genetic Programming (GP) [Koza, 1992, Koza, 1994] programs are expressed as parse trees to be executed sequentially. This form of GP has been applied successfully to a number of difficult problems like automated design, pattern recognition, robot control, image analysis, etc. [Koza, 1992, Koza, 1994, K. E. Kinneer, Jr., 1994].

When appropriate terminals, functions and interpreters are defined, standard GP can go beyond the production of sequential programs. For example using cellular encoding GP can be used to develop structures, like neural nets [Gruau, 1994] or electronic circuits [Koza et al., 1996], which perform some form of parallel analogue computation. Also, in conjunction with an interpreter implementing a parallel virtual machine, GP can be used to translate sequential programs into parallel ones [Walsh and Ryan, 1996] or to develop some kinds of parallel programs. For example, Bennett [Bennett III, 1996] used a simulated parallel virtual machine in which several standard tree-like programs were executed in parallel, node by node. In [Andre et al., 1996]

GP was used to discover rules for cellular automata (CAs) which could solve majority-classification problems. In a system called PADO [Teller and Veloso, 1995], a combination of GP and linear discriminators was used to obtain a kind of parallel classification programs for signals and images.

All these methods are limited to very special kinds of parallelism (e.g. PADO's nodes include a stack and a branching condition, CAs use purely functional nodes, and cellular encoding has not been used to evolve parallel programs) and cannot be considered a natural generalisation of GP. This paper describes Parallel Distributed Genetic Programming (PDGP), a form of GP which is suitable for the development of programs with a high degree of parallelism and distributedness. Programs are represented in PDGP as graphs with nodes representing functions and terminals, and links representing the flow of control and results. In PDGP, programs are manipulated by special crossover and mutation operators which guarantee the syntactic correctness of the offspring.

In the simplest form of PDGP, links are directed and unlabelled, in which case PDGP can be considered a generalisation of standard GP as it uses the same kind of nodes as GP. However, PDGP can use more complex representations and evolve neural networks, finite state automata, recursive transition nets, etc. In previous work we have studied some of these new representational capabilities [Poli, 1997a, Poli, 1997b] but we have presented limited evidence to show when and if PDGP can perform better than standard GP [Poli, 1996b]. The main objective of this paper is to investigate this issue.

The paper is organised as follows. Firstly, in Section 2, we describe the representation, the genetic operators and the interpreter used in PDGP. Then, we illustrate the behaviour of PDGP on two classes of problems (Section 3). We discuss our results in Section 4 and we draw some conclusions in Section 5.

---

\* E-mail: R.Poli@cs.bham.ac.uk

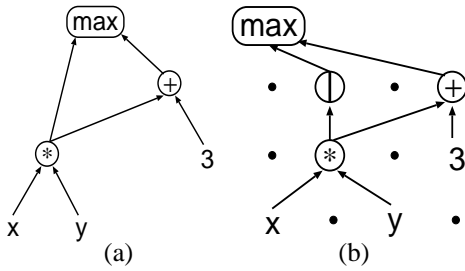


Figure 1: (a) Graph-based representation of the expression  $\max(x*y, 3+x*y)$ ; (b) Grid-based representation of the graph in (a).

## 2 PDGP

### 2.1 Representation

Standard “tree-like” programs can often be represented as graphs with labelled nodes (the functions and terminals used in the program) and oriented links which prescribe which arguments to use for each node. For example, Figure 1(a) shows the function  $\max(x*y, 3+x*y)$  represented as a graph. Its execution should be imagined as an upwards propagation of results.

This representation is in general more compact (there are fewer nodes) and efficient (partial results are reused). Also, it can be used to express a much bigger class of programs (trees are special kinds of graphs) which are parallel in nature: we call them *parallel distributed programs*.

PDGP can evolve parallel distributed programs using a direct graph representation which allows the definition of efficient crossover operators. The basic representation is based on the idea of assigning each node in the graph to a physical location in a multi-dimensional grid with a prefixed shape.

This grid-based representation is illustrated in Figure 1(b). In this example the program’s output is at coordinates (0,0) and the grid includes  $3 \times 4 + 1$  cells. Connections are directed upwards and are allowed only between nodes belonging to adjacent rows, like in a multi-layer perceptron. This is not a limitation as, by adding the identity function to the function set (see the pass-through node in the figure), any parallel distributed program can be described.

From the implementation point of view PDGP represents programs as arrays with the same topology as the grid, in which each cell contains a function label and the horizontal displacement of the nodes in the previous layer used as arguments for the function. Functions or terminals are associated to *every* node in the grid, i.e. also to the nodes that are not directly or indirectly connected to the output. We call them *inactive nodes* or *introns*. The others are called *active nodes*.

This basic representation has been extended in various directions.<sup>1</sup> The first extension was to introduce vertical displacements to allow feed-forward connections between non-adjacent layers. With this extension any directed acyclic graph can be naturally represented within the grid of nodes, without requiring the presence of pass-through nodes. The second extension has been to allow non-positive vertical displacements to represent graphs with backward connections (e.g. with cycles). The third extension has been to add labels to links. This extension allows the direct development of neural networks, finite state automata, transition nets, semantic nets, etc.

### 2.2 Genetic Operators

Several kinds of crossover, mutation and initialisation strategies can be defined for the basic representation used in PDGP and for its extensions. In the following we will describe three forms of crossover and two mutation operators (more details can be found in [Poli, 1996a]).

The basic crossover operator of PDGP, which we call *Sub-graph Active-Active Node (SAAN) crossover*, is a generalisation to graphs of the crossover used in GP to recombine trees. SAAN crossover works as follows: 1) a random active node is selected in each parent (crossover point); 2) a sub-graph including all the active nodes which are used to compute the output value of the crossover point in the first parent is extracted; 3) the sub-graph is inserted in the second parent to generate the offspring (if the  $x$  coordinate 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(a). Obviously, for SAAN crossover to work properly some care has to be taken to ensure that the depth of the sub-graph being inserted in the second parent is compatible with the maximum allowed depth, i.e. the number of rows in the grid. A simple way to do this is to select one of the two crossover points at random and choose the other with the coordinates of the first crossover point and the depth of the sub-graph in mind.

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.

Several different forms of crossover can be defined by modifying SAAN crossover. One that has given good results is *Sub-Sub-graph Active-Active Node (SSAAN) Crossover*.

<sup>1</sup>All these extensions require changes to the operators and interpreter used in PDGP which will not be described as they have not been used in the experiments reported in the paper.

In SSAAN crossover one crossover point is selected at random among the active nodes of each parent. Then, a complete sub-graph is extracted from the first parent (like in SAAN crossover) disregarding the problems possibly caused by its depth. If the depth of the sub-graph is too big for it to be copied into the second parent, the lowest nodes of the sub-graph are pruned to make it fit. Figure 2(b) illustrates this process. Another good form of crossover is the *Sub-Sub-graph Inactive-Active Node (SSIAN) Crossover* which selects the first crossover point among both the active and the inactive nodes. Obviously, introns are essential in order for these types of crossover to work properly.

Two forms of mutation are used in PDGP. *Global mutation* consists of inserting a randomly generated sub-graph into an existing program. *Link mutation* changes a random connection in a graph by firstly selecting a random function-node, then a random input-link of such a node and finally altering the offset associated to the link.

### 2.3 Interpreters

If no functions or terminals with side effects are used, it is possible to evaluate a PDGP program just like a feed-forward neural net, starting from the input-layer, which always contains only terminals, and proceeding layer after layer upwards until the output nodes are reached. However, the current PDGP interpreter does this differently, by using recursion and a hash table to store partial results and control information.

The interpreter starts the evaluation of a program from the output nodes and calls recursively the procedure `microeval(node)` to perform the evaluation of each of them. `microeval` checks to see if the value of a node is already known. If this is the case it returns the value stored in the hash table, otherwise it executes the corresponding terminal or function (calling itself recursively to get the values for the arguments, if any).

This approach has the advantage of allowing the use of nodes with side effects (in this case `microeval` forces the evaluation of a node even if its value is already known) (see [Poli, 1996a] for more details). It also allows a total freedom in the connection topology of PDGP programs.

It is worth noting that although the interpreter described above performs a sequential evaluation of programs, this process is inherently parallel. In fact, if we imagine each node to be a processor (with some memory to store its state and current value) and each link to be a communication channel, the evaluation of a program is equivalent to the parallel downward propagation of control messages requesting a value followed by an upward propagation of return values. In this sense our interpreter can be seen as a parallel virtual machine capable of running data-

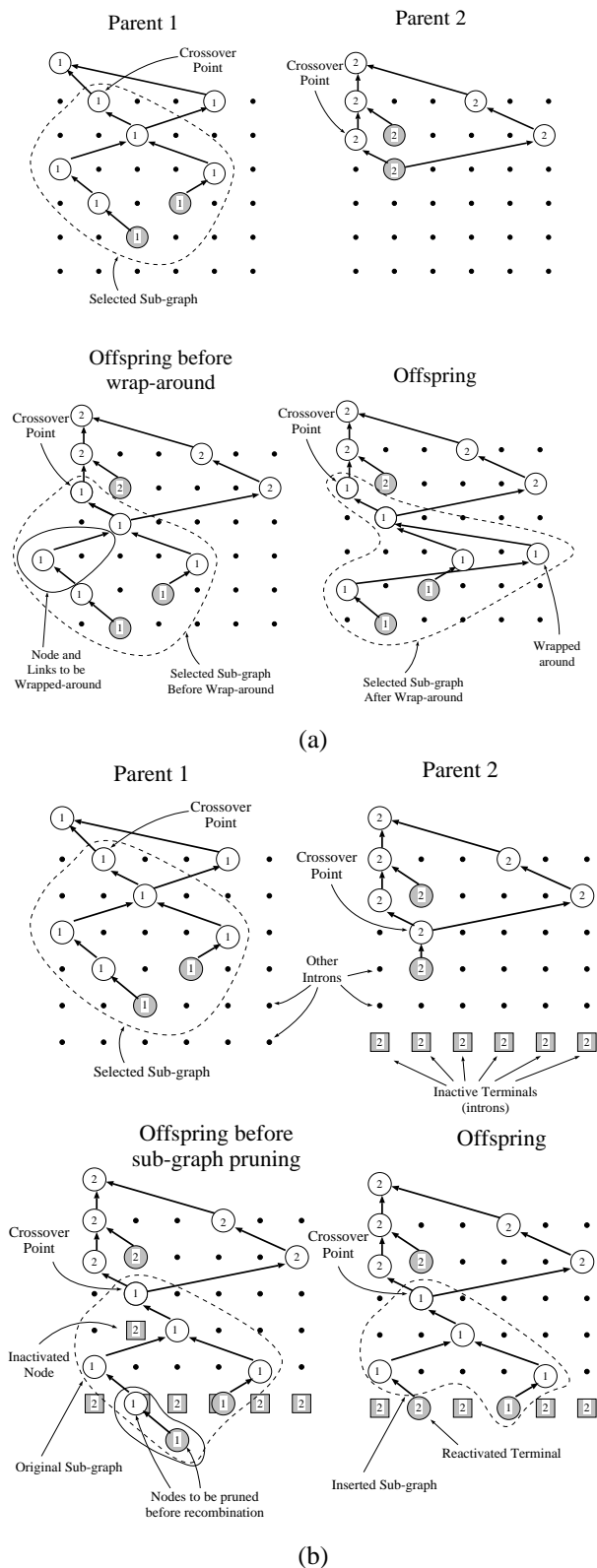


Figure 2: Sub-graph active-active node (SAAN) crossover (a) and sub-sub-graph active-active node (SSAAN) crossover (b).

driven dataflow programs [Jagannathan, 1995] and PDGP as a method to evolve them.

### 3 Experimental Results

In this section we will report on experiments which, while emphasising some of the features of PDGP, allow a fair performance comparison with standard GP. As a performance criterion we used the computational effort  $E$  used in the GP literature ( $E$  is the minimum number of fitness evaluations necessary to get a correct program, in multiple runs, with probability 99%). The test problems used for the comparison were the lawnmower problem and the MAX problems.

#### 3.1 Lawnmower problem

The lawnmower problem consists of finding a program which can control the movements of a lawnmower so that it cuts all the grass in a rectangular lawn [Koza, 1994, pages 225–273]. This problem presents very interesting features: a) its difficulty can be varied indefinitely by changing the size of the lawn, b) although it can be very hard to solve for GP, the evaluation of the fitness of each program requires running the program only once, c) it is a problem with a lot of regularities which GP without Automatically Defined Functions (ADFs) seems unable to exploit, d) GP with ADFs does very well on this problem.

In the lawnmower problem the lawn is represented by an array of grass tiles and the lawnmower can only move in four different directions (0, 90, 180 and 270 degrees). The mower can only be rotated towards left, moved forward one step (in its current direction) or lifted and repositioned on another tile. To do this we used the terminal set  $\mathcal{T}=\{\text{MOW}, \text{LEFT}, \text{random}\}$  and the function set  $\mathcal{F}=\{\text{V8A}, \text{FROG}, \text{PROG2}, \text{PROG3}\}$ . The function MOW performs a step forward, mows the new grass patch and returns the pair of coordinates (0, 0). LEFT performs a 90 degree left rotation of the mower and returns (0, 0). random returns a random pair of coordinates  $(x, y)$  with  $x, y \in \{0, \dots, 7\}$ . V8A performs the addition modulo 8 of two pairs of coordinates. FROG lifts and moves the mower to a new cell whose displacement in horizontal and vertical direction is indicated by its (single) argument, mows the new grass patch and returns its argument. PROG2 (PROG3) is the usual LISP programming construct which executes its two (three) arguments in their order and then returns the value returned by the second (third) argument. These functions and terminals allow a direct comparison of the results described in this section with those obtained with standard GP [Koza, 1994].

The other parameters of PDGP were: SSIAN crossover with probability 0.7, link mutation with probability 0.009, global mutation with probability 0.001, population size

$P=1000$ , maximum number of generations  $G=50$ , grid with 8 rows and 2 columns and “grow” initialisation method.

Figure 3 shows a typical program obtained in our experiments with an  $8 \times 8$  lawn. The numeric labels near the links represent the order in which the arguments of each node are evaluated. For example they show that the function PROG3 in the output node first invokes two times the subgraph on its left and then executes once the sub-graph of the right.

The program is extremely parsimonious (it includes only 11 nodes out of the possible 17) and shows one of the features of PDGP: the extensive reuse of sub-graphs. In fact, each sub-graph of this program is called over and over again by the nodes above it and the program is therefore organised as a complex hierarchy of nine non-parametrised subroutines (the non-terminal nodes) calling each other multiple times. This tends to be a common feature in PDGP programs: sub-graphs act as building blocks for other sub-graphs.

Despite its small size, the depth of the nested function calls makes this program execute hundreds of actions during the interpretation. Figure 4 shows the behaviour of the program. The squared grid represents the lawn, while the numbers inside each cell represent the order in which it was mowed (the program was stopped as soon as it mowed the entire lawn, after 112 mowing steps).

In order to assess the performance of PDGP on the lawnmower problem, we varied the width of the lawn from 4 to 16, while keeping its height constant (8 cells), thus varying the lawn size from 32 to 128 grass tiles. For each lawn size we performed 20 runs with different random seeds. Figure 5 shows a plot of the computational effort  $E$  necessary to solve the lawnmower problem as a function of the complexity of the problem (i.e. the number of cells in the lawn).

The results shown in the plot are amazing. The computation effort reported in [Koza, 1994] for standard GP without ADFs is  $E=19,000$  for a lawn size of 32,  $E=56,000$  for a size of 48,  $E=100,000$  for a size of 64,  $E=561,000$  for a size of 80, and  $E=4,692,000$  for a size of 96 cells, with an exponential grow in the effort as the difficulty of the problem increases. On the contrary, PDGP is solving the same problems with an effort ranging from 4,000 to 6,000, which seems to grow linearly even beyond the sizes on which standard GP was tried. Actually, a comparison between the slopes of the linear regression plotted in the figure and the linear regression for standard GP [Koza, 1994, page 266] shows that *PDGP scales up about 2,300 times better*. Also, as shown in Figure 6, the structural complexity (i.e. the number of nodes) of the solutions does not vary significantly (it really could not) as the complexity of the problem changes. As the average structural complexity of

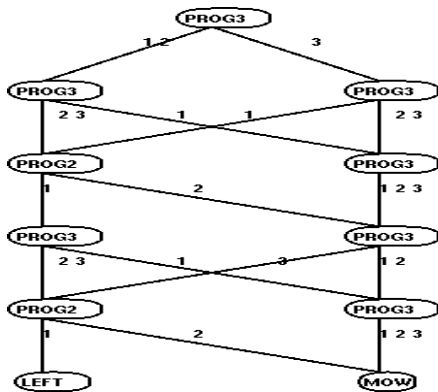


Figure 3: Parallel distributed program solving the lawnmower problem with a lawn of  $8 \times 8 = 64$  cells.

100	107	106 72 34	105 65 37	104 56 4	103 13	102	101
93 9	8	71 33 7	66 38 6	55 5	12	11	92 10
94 22	23	70 32	67 39	54	19	20	91 21
95 25	24	69 31	68 40 30	53 29	28 18	27	90 26
96	111	76	77 41	78 52 0	79 17	80	89
97 87	110 86	85 75	84 42	83 51 1	82 16	81	88
98 46	109 45	74 44	43	50 2	49 15	48	47
99 61	108 62	73 63 35	64 36	57 3	58 14	59	60

Figure 4: Behaviour of the program in Figure 3.

standard GP programs ranges from 145.0 for the 32-cell lawn to 408.8 for the 96-cell one, *PDGP is between 10.5 and 29.2 times more parsimonious than GP* on this problem.

PDGP clearly outperforms GP without ADFs. The situation for GP improves somehow when ADFs are added. For example, the computational effort drops to values from 5,000 (for the 32-cell lawn) to 20,000 (for the 96-cell lawn) while the average structural complexities range from 66.3 to 84.9. However, the analysis of the linear regression equations reveals that *PDGP still scales up about 9 times better than GP with ADFs* and that *the structural complexity of PDGP programs is 4.7 to 6.1 times smaller*.

In order to understand whether these levels of performance were due to the features of the SSAN crossover or to the particularly narrow grid used, which enforces strongly graph-like (as opposed to tree-like) solutions, we decided to study the behaviour of PDGP on the 64-cell problem as the width of the grid was varied from 2 to 64 (with intermediate values 4, 6, 8, 16, 32). Again we did 20 different

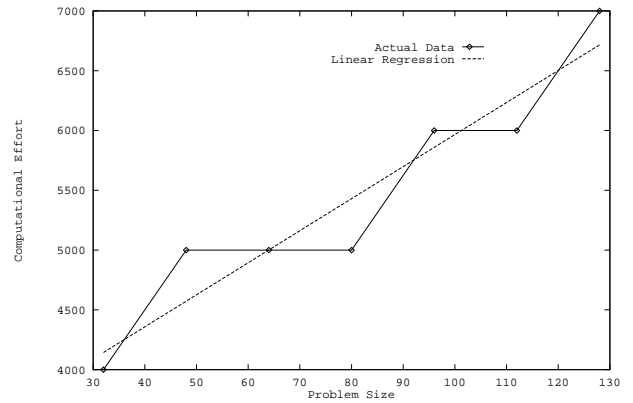


Figure 5: Computational effort  $E$  necessary to solve the lawnmower problem as a function of the complexity of the problem.

runs for each configuration.

Figure 7(a) shows a plot of the computational effort  $E$  necessary to solve the 64-cell lawnmower problem as a function of number of nodes in the grid. The inspection of the resulting programs, also corroborated by the plot of the average structural complexity of the PDGP programs reported in Figure 7(b), shows that by increasing the width of the grid, the “tree-ness” of the programs increases (when the grid is very large PDGP really tends to behave like standard GP with a special form of crossover). As somehow expected given the results mentioned above, PDGP performance decreased as the size of the grid grew. However, it never became as bad as the one shown by standard GP ( $E$  was always less than a third).

### 3.2 MAX Problems

The MAX problems are a class of problems introduced in [Gathercole and Ross, 1996] to study the adverse interaction between crossover and the size limits imposed on the trees developed by standard GP. They consist of finding a program which returns the largest possible value for a given function and terminal set with the constraint that the maximum tree-depth cannot exceed a pre-fixed value  $D$ . For example, one class of MAX problems, the *MAX-depth- $D\{*,+\}\{0.5\}$* , consists of finding the combination of multiplications, additions and constants (0.5) such that the output of the program is the theoretical maximum  $4^{2^{D-3}}$  (for a prefixed maximum depth  $D \geq 3$ ).

Gathercole and Ross showed that the seemingly simple class of problems *MAX-depth- $D\{*,+\}\{c\}$*  can be actually very difficult to solve for GP if the mutation probability is set to zero (as usual) and the constant  $c$  in the terminal set is smaller than 1. In particular the smaller  $c$  and the larger  $D$  the harder the problem. For example, with populations of 200 individuals, in 200 generations GP could solve approx-

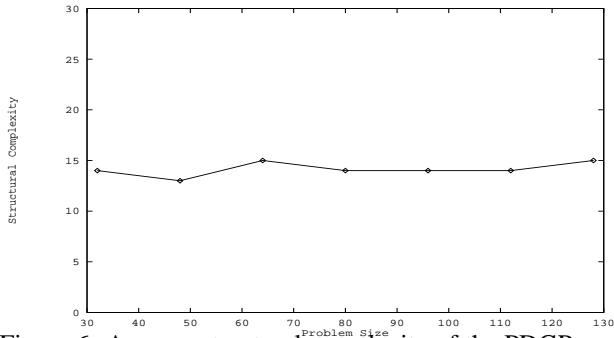


Figure 6: Average structural complexity of the PDGP programs which solve the lawnmower problem as a function of the complexity of the problem.

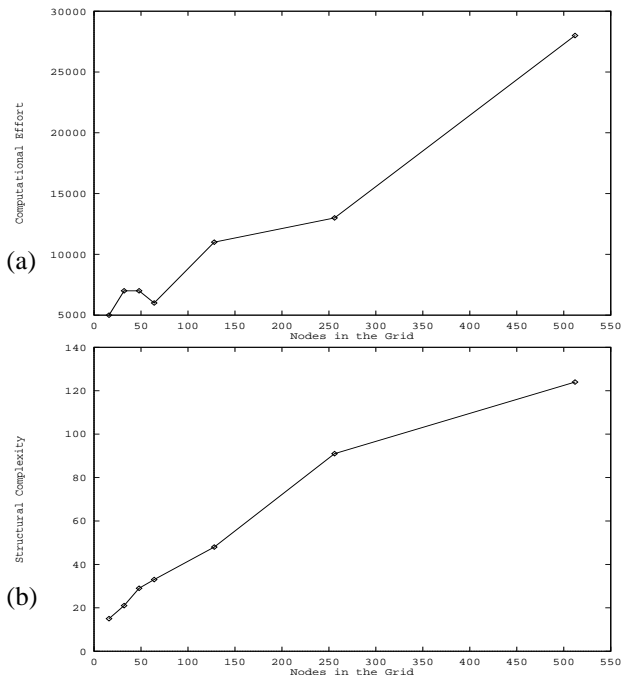


Figure 7: Computational effort  $E$  to solve the 64-cell lawnmower problem (a) and average structural complexity of the resulting PDGP programs (b) as a function of number of nodes in the PDGP grid.

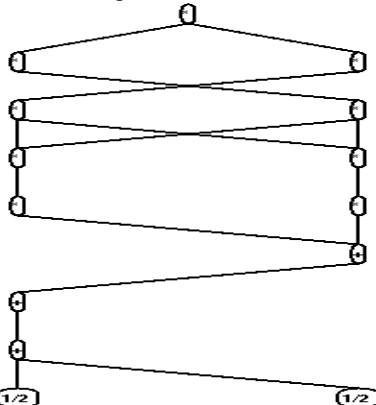


Figure 8: Typical solution to the  $MAX\text{-depth-}8\{*,+\}\{0.5\}$  problem evolved by PDGP.

imately 30%  $MAX\text{-depth-}6\{*,+\}\{0.5\}$  problems and basically no  $MAX\text{-depth-}D\{*,+\}\{0.25\}$  problem with  $D \geq 6$ . The difficulty of the MAX problems has been recently confirmed in [Langdon and Poli, 1997] where a success probability of 52% for  $MAX\text{-depth-}6\{*,+\}\{0.5\}$  and 42% for  $MAX\text{-depth-}8\{*,+\}\{0.5\}$  was reported for runs of 500 generations.

One of the difficulties of the MAX problems is that if  $c < 1$  in order to get large output values first GP has to use only additions to get constants bigger than 1 and then it has to use only multiplications to get the maximum output. Therefore, during a first stage of evolution multiplication operations will tend to survive better in the upper nodes in the tree (where they can act on numbers bigger than one). Later, when they should also be used at lower levels in the tree, crossover is unable to move them given the depth limit. As discussed in [Langdon and Poli, 1997] another reason for the observed adverse interaction is a reduced probability of selecting, as crossover points, nodes near the root node when all trees in the population are very large. This is a situation to which virtually all GP populations tend (bloating) and which has been experimentally observed in GP populations solving the MAX and lawnmower problems.

Given that in PDGP we have added an additional constraint on the structures being evolved, namely the limited width of the grid, we thought the class of MAX problems could provide a good test for our representation and crossover operators. In particular we have studied the behaviour of PDGP with populations of 200 individuals, SSAAN crossover and no mutation on the  $MAX\text{-depth-}D\{*,+\}\{0.5\}$  and  $MAX\text{-depth-}D\{*,+\}\{0.25\}$  problems. We chose to use the SSAAN crossover as it is the one that uses fewer introns (only the inactive terminals in the last row of each grid) and, therefore, makes the comparison with GP fairer. For the same reason in our experiments the number of rows in the grid was set to  $D$ . The number of columns was 2, 4, 8, 16, 32, 64 and 128. For each combination of  $D$ ,  $c$ , and number of columns we repeated 10 experiments (with different random seeds).

Figure 8 shows a typical solution to the  $MAX\text{-depth-}8\{*,+\}\{0.5\}$  problem evolved by PDGP. This program is really parsimonious as it contains only 14 nodes: about 36 times fewer than the 511 necessary to build 100% correct solutions with GP. Again this is achieved by extensively reusing the partial results computed by other nodes. Tables 1 and 2 summarise the results of our experiments. The last row of Table 1 reports the computational effort for standard GP obtained from the experiments in [Langdon and Poli, 1997].

The situation depicted by these tables is quite clear: SSAAN crossover in conjunction with narrow grids trans-

PDGP	Maximum Depth ( $D$ )					
Cols	3	4	5	6	7	8
2	0.4	1.0	2.0	2.8	3.2	3.8
4	0.6	7.2	8.4	3.8	5.6	17.4
8	1.0	4.8	12.0	16.8	21.0	60.0
16	1.4	10.8	19.2	8.4	29.6	176.4
32	1.0	18.2	27.6	56.4	114.4	N/A
64	1.4	30.0	56.0	57.4	369.6	N/A
128	1.0	18.2	64.0	193.6	N/A	N/A
GP	0.4	7.3	30.8	102.5	236.1	682.1

Table 1: Computational effort  $E$  (thousands of fitness evaluations) necessary to solve  $MAX\text{-depth-}D\{*,+\}\{0.5\}$ .

	Maximum Depth ( $D$ )					
Cols	3	4	5	6	7	8
2	0.6	1.0	4.0	2.4	3.6	5.8
4	0.8	1.2	23.4	28.0	16.2	18.0
8	0.8	1.4	26.0	41.6	39.2	88.2
16	1.0	2.0	58.8	57.2	172.2	167.2
32	1.2	2.4	79.2	147.0	N/A	255.2
64	1.4	2.6	N/A	N/A	N/A	N/A
128	1.0	2.8	123.2	N/A	N/A	N/A

Table 2: Computational effort  $E$  (thousands of fitness evaluations) necessary to solve  $MAX\text{-depth-}D\{*,+\}\{0.25\}$ .

forms a problem which can be very hard to solve for standard GP into a very easy problem. Actually PDGP with a two-column grid is able to solve it with a computational effort  $E$  which comparable with the one necessary to find solutions to XOR problems [Poli, 1996a] or lawnmower problems without showing any difficulty for any values of  $D$ . A comparison between the linear regression equation of PDGP and the linear regression equation for standard GP shows that *PDGP scales up about 170 times better than GP and, again, it does it linearly rather than exponentially*. For relatively hard problems (with  $D \geq 6$ ), performance seems to degrade quite quickly as the number of columns in the grid is increased.

## 4 Discussion

In the experiments with the lawnmower problem PDGP has outperformed GP without ADFs. This might seem counter-intuitive given that we impose strong constraints on the size and shape of the grid and, therefore, on the space of programs that PDGP can explore. On the contrary, no practical constraints on the size and shape of programs are imposed in GP. The only explanation for the effectiveness of PDGP in solving the lawnmower problem is its ability to discover parallel distributed programs which reuse partial solutions.

This ability is maximum when relatively narrow grids are used and decreases as the width of the grid is increased. This indicates that enlarging the space of possible programs by widening the grid does not mean making the search easier. Indeed what is important for the efficiency a search

algorithm is the frequency of solutions in the search space. For problems with a lot of regularities like the lawnmower problem, small grids seem to create a search space with a high density of solutions. Small grids might not be optimum for problems where irregular solutions are needed. For example, in [Poli, 1996b] we found that PDGP solved the even-3 parity problem more easily with relatively large grids. So, the optimum grid size might be problem specific.

However, somehow surprisingly, on the same problem, PDGP also outperformed GP with ADFs, which have been proven to help discovering regularities and to reuse partial results [Koza, 1994]. Certainly in the lawnmower problem the use of parametrised ADFs allows GP to explore a much larger space of possible programs in which the frequency of solutions is significantly higher than in the original space. However, the non-parametrised reuse of partial results available in standard PDGP seems to generate a search space with an even higher density of solutions. This can be explained by considering the relative difficulty of discovering good ADFs: ADFs behave differently in different parts of a program when they have different arguments, so in order to discover if an ADF is good, GP has also to “understand” with which arguments the ADF can be used properly.

The analysis of the results obtained in the MAX problems reveals another factor contributing to the good performance of PDGP: its ability to overcome the adverse interaction between the maximum program depth and crossover reported by Gathercole and Ross. Part of this ability derives from the fact that, although SSAAN crossover works very similarly to standard GP crossover when the structures in the grid tend to be trees, it presents a crucial difference: by design it can move any node (with the attached sub-graph) to any position in the grid at any time. However, a larger contributions to PDGP ability to solve MAX problems is probably the fact that, in PDGP with narrow grids, the probability of selecting crossover points near the output layer is not significantly different from that of the other layers.

## 5 Conclusions

In this paper we have presented PDGP, a form of genetic programming which is suitable for the development of programs with a high degree of parallelism and an efficient and effective reuse of partial results.

The grid-based 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 degree of parallelism and of reuse of partial results in the programs being developed.

Although in all our experiments we have run PDGP pro-

grams sequentially or simulating a parallel virtual machine, they could really be executed on demand-driven dataflow computers. This would lead to a very efficient evaluation of fitness cases as purely functional PDGP programs could produce a new result at every clock tick.

The programs developed by PDGP are fine-grained, but the representation used is also suitable for the development of medium-grained programs via the use of ADFs and Automatically Defined Links (ADLs), a technique which has given very promising preliminary results [Poli, 1997b].

ADLs are just one of the new representational possibilities opened by PDGP. Indeed, PDGP is a paradigm to evolve general graphs (with or without cycles, possibly recursively nested via ADFs and ADLs, with or without labelled links, with or without directions, etc.) which need not be interpreted as programs: they can be interpreted as designs, semantic nets, neural networks, finite state automata, etc. Future work will be devoted to explore the ability of PDGP to evolve other such structures.

## Acknowledgements

The author wishes to thank W. B. Langdon and the other 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

- [Andre et al., 1996] Andre, D., Bennett III, F. H., and Koza, J. R. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. *Proceedings of Genetic Programming'96*, p. 3–11, Stanford. MIT Press.
- [Bennett III, 1996] Bennett III, F. H. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. *Proceedings of Genetic Programming'96*, p. 30–38, Stanford. MIT Press.
- [Gathercole and Ross, 1996] Gathercole, C. and Ross, P. An adverse interaction between crossover and restricted tree depth in genetic programming. *Proceedings of Genetic Programming'96*, p. 291–296, Stanford. MIT Press.
- [Gruau, 1994] Gruau, F. Genetic micro programming of neural networks. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 24, p. 495–518. MIT Press.
- [Jagannathan, 1995] Jagannathan, R. Dataflow models. In Zomaya, E. Y., editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill.
- [K. E. Kinnear, Jr., 1994] K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*. MIT Press.
- [Koza, 1992] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.
- [Koza, 1994] Koza, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts.
- [Koza et al., 1996] Koza, J. R., Andre, D., Bennett III, F. H., and Keane, M. A. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. *Proceedings of Genetic Programming'96*, p. 132–149, Stanford. MIT Press.
- [Langdon and Poli, 1997] Langdon, W. B. and Poli, R. Price's theorem and the MAX problem. Technical Report CSRP-97-4, School of Computer Science, University of Birmingham.
- [Poli, 1996a] Poli, R. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, The University of Birmingham.
- [Poli, 1996b] Poli, R. Some steps towards a form of parallel distributed genetic programming. In *Proceedings of the First On-line Workshop on Soft Computing*.
- [Poli, 1997a] Poli, R. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In *3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97*.
- [Poli, 1997b] Poli, R. Evolution of recursive transistion networks for natural language recognition with parallel distributed genetic programming. *Proceedings of AISB-97 Workshop on Evolutionary Computing*, Manchester, (in press).
- [Teller and Veloso, 1995] Teller, A. and Veloso, M. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Department of Computer Science, Carnegie Mellon University.
- [Walsh and Ryan, 1996] Walsh, P. and Ryan, C. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. *Proceedings of Genetic Programming'96*, p. 406–409, Stanford. MIT Press.