Parallel Distributed Genetic Programming

Riccardo Poli
School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom

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

Technical Report: CSRP-96-15
September 1996

Abstract

This paper describes Parallel Distributed Genetic Programming (PDGP), a new form of Genetic Programming (GP) which is suitable for the development of programs with a high degree of parallelism and an efficient and effective reuse of partial results. Programs are represented in PDGP as graphs with nodes representing functions and terminals, and links representing the flow of control and results. In the simplest form of PDGP links are directed and unlabelled, in which case PDGP can be considered a generalisation of standard GP. However, more complex (direct) representations can be used, which allow the exploration of a large space of possible programs including standard tree-like programs, logic networks, neural networks, recurrent transition networks, finite state automata, etc. In PDGP, programs are manipulated by special crossover and mutation operators which guarantee the syntactic correctness of the offspring. For this reason PDGP search is very efficient. PDGP programs can be executed in different ways, depending on whether nodes with side effects are used or not. The paper describes the representations, the operators and the interpreters used in PDGP, and illustrates its behaviour on a large number of problems.

1 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 [16, 17].

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 set of possible internal (non-leaf) nodes used in GP parse trees is called *function set*, $\mathcal{F} = \{f_1, \cdots, f_{N_F}\}$. $\mathcal{F}$ can include almost any kind of programming construct: arithmetic operators, mathematical and Boolean functions, conditionals, looping constructs, procedures with side effects, etc. The set of terminal (leaf) nodes in the parse trees is called *terminal set* $\mathcal{T} = \{t_1, \cdots, t_{N_T}\}$. $\mathcal{T}$ can include: variables, constants, 0-arity functions with or without side effects, random constants, etc. The basic search algorithm used in GP is a classical GA with mutation and crossover specifically designed to handle parse trees.

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. [16, 17, 14, 15, 1, 22].

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 [9, 10] or electronic circuits [19, 18], which can be thought

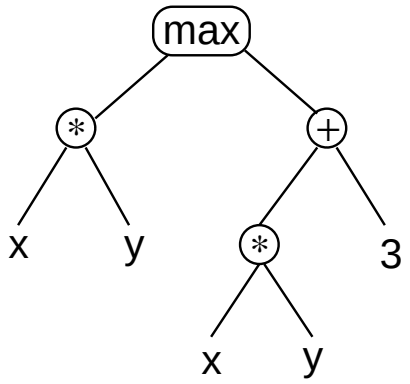

Figure 1: Parse-tree representation of the expression `max(x * y, 3 + x * y)`.

of as performing 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 [31] or to develop some kinds of parallel programs.[1]

For example, Bennett, in [4], used a parallel virtual machine in which several standard tree-like programs (called "agents") would have their nodes executed in parallel with a two stage mechanism simulating parallelism of sensing actions and simple conflict resolution (prioritisation) for actions with side effects. Andre, Bennet and Koza [2] used GP to discover rules for cellular automata which could solve large majority-classification problems. In a system called PADO (Parallel Algorithm Discovery and Orchestration), Teller and Veloso [28] used a combination of GP and linear discrimination to obtain classification programs for signals and images. Given the slight similarity between the representation for programs used in PADO and the one used in PDGP we will give more details on such a system.

PADO programs are actually made up of several groups of programs (called systems) each group being used to recognise the instances of a given class. The system with the highest output determines the output of PADO. The output of a system is a linear combination of the output of the programs included in the system. Initially PADO programs seemed to use the standard tree-like representation used in GP (see for example [28, page 30]). However, more recent work [27, 26] has clarified that systems are represented as directed graphs in which arcs represent possible flows of control and nodes perform actions and take decisions on which of the possible control flows should be followed. Each node includes an action (a terminal, a function, an automatically define function (ADF) or library call), a branching condition, and a stack. The interpretation of PADO programs is performed through a post-order chronological (not structural) graph traversal (see [26] for more details). A form of program representation similar to the one used in PADO (but using nodes without branching conditions) has been recently termed by Teller [29] "neural programming".

Other work which has some relation with ours is based on the idea, firstly proposed by Handley [12] and later improved by Ehrenburg [7], of storing the population of parse trees as a single directed acyclic graph, rather than as a forest of trees. This leads to considerable savings of memory (structurally identical subtrees are not duplicated) and computation (the value computed by each subtree for each fitness case is cached). However, it is important to emphasise that this technique can only be applied when using *primitives with no side effects* and that it is conceptually equivalent to the evolution of tree-like programs as standard GP, i.e. it is not about the evolution of parallel programs.

This paper describes Parallel Distributed Genetic Programming (PDGP), a new form of Genetic Programming (GP) which is suitable for the development of programs with a high degree of parallelism and distributedness (some early explorations with PDGP have been described in [23]). Programs are represented in PDGP as graphs with nodes representing functions and terminals and links representing

[1] 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 [3, 6, 21, 13, 25]. 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.

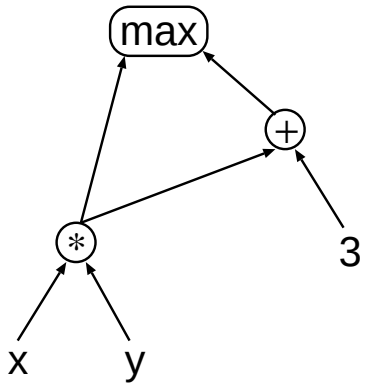

Figure 2: Graph-like representation of the expression `max(x * y, 3 + x * y)`.

the flow of control and results. In the simplest form of PDGP, links are directed and unlabelled, in which case PDGP can be considered a generalisation of standard GP as PDGP (unlike PADO) uses the same kind of nodes as GP, and trees are a special kind of graphs. However, PDGP can use more complex (direct) representations, which allow the development of symbolic, neuro-symbolic and neural networks, recurrent transition networks, finite state automata, etc.

Like GP, PDGP allows the development of programs of any size and shape (within predefined limits). However, it also allows the user to control the degree of parallelism of the programs to be developed. In PDGP, programs are manipulated by special crossover and mutation operators which, like the ones used in GP, guarantee the syntactic correctness of the offspring. This leads to a very efficient search of the space of possible parallel distributed programs. PDGP programs can be executed in different ways, depending on whether nodes with side effects are used or not.

The paper is organised as follows. Firstly, in Section 2, we describe the basic representation used in PDGP. Then, we report on genetic operators and initialisation strategies (Section 3) and we describe the basic interpreters used in PDGP (Section 4). Then, we describe some extensions to the basic representation and operators used in PDGP (Section 5) and we illustrate the behaviour of PDGP on a large number of problems (Section 6). Finally, we draw some conclusions and describe promising directions for future work in Section 7.

2 Representation

As mentioned above, usually programs in GP are represented as parse trees. Taking inspiration from the parallel distributed processing performed in neural nets [24], we decided to explore the possibility of representing programs as graphs with labelled nodes and oriented links. The idea was that nodes could be the functions and terminals used in a program while the links would determine which arguments to use in each function-node when it is next evaluated. We hoped in this way to obtain a form of parallel distributed programming with some of the useful features shown by artificial neural nets.

Figure 2 shows an example of 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)`. For now, its execution should be imagined as a "wave of computations" starting from the terminals and propagating upwards along the graph, more or less like the updating of the activations of the neurons in a multi-layer feed-forward neural net.

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 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-

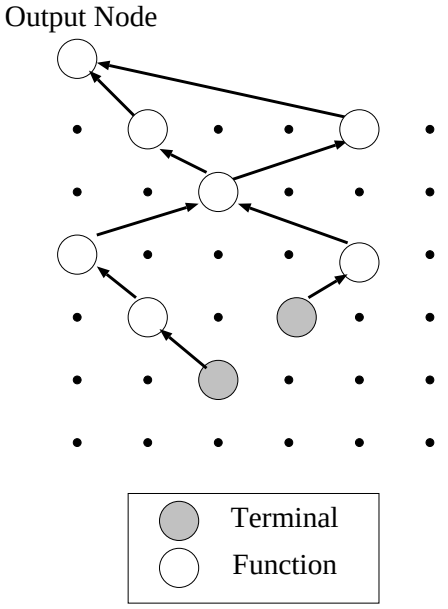

Figure 3: Grid-based representation of graphs representing programs in PDGP.

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 [9, 10, 19, 18] or edge encoding [20], 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 [11]).

PDGP uses a direct representation of graphs which allows the definition of crossover operators which respect all the criteria listed above (in particular efficiency and offspring validity). The basic representation is based on the idea of assigning each 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, we allow connections to exist only 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.[2]

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-based graph representation. For example, the program in Figure 2 can be transformed into the

[2] In this paper (if not otherwise stated) we have adopted the convention that the first row of the grid includes as many cells as the number of outputs of the program.

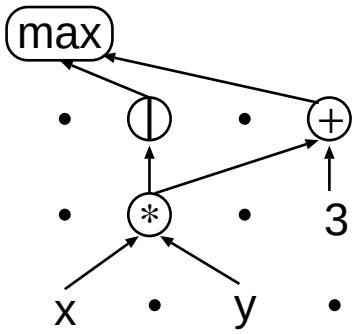

Figure 4: Grid-based representation of the expression `max(x * y, 3 + x * y)`.

layered network in Figure 4.

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

```
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 basic 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 be associated to *every* node in the grid, i.e. also to the nodes that are not directly or indirectly connected to 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 also be noted that by using an array with the same topology as the grid, it is possible to store in each cell of the array only the label and the input connections of each node and omit the coordinates from the representation. This can lead to more efficient implementations.

This basic representation can and has been extended very in various directions. We will briefly describe these extensions after we have introduced the standard genetic operators and interpreters of PDGP so as to give a clearer explanation of the changes the different representations impose in such components. We will also present some examples of programs using these representations.

3 Genetic Operators

Several kinds of crossover, mutation and initialisation strategies can be defined for the basic representation used in PDGP.

3.1 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.

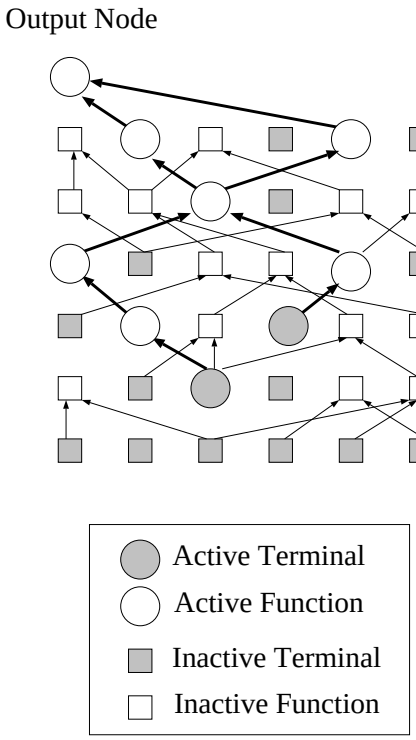

Figure 5: Intron-augmented representation of programs in PDGP.

In PDGP the generation of random programs can proceed in several ways, depending on 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 the kind of graphs we want to build (balanced or not) and on the depth reached with recursion.

In most of the examples described in this paper we have used relatively "shallow" grids (i.e. grids with a small number of rows). In our experience for such grids maintaining a distinction between the maximum depth of graphs (the number of rows in the grid) and the maximum depth for the initial graphs makes little difference. For this reason we have always given the same value to the two parameters.

3.2 Crossover

The basic crossover operator of PDGP, which we call *Sub-graph Active-Active Node (SAAN) crossover*, is basically 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 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.[3] The idea behind this form of crossover is that connected sub-graphs are functional units whose output is used by other functional units. Therefore,

[3] In the "vanilla" SAAN crossover inactive nodes play no role: for this reason they are not shown in the figure.

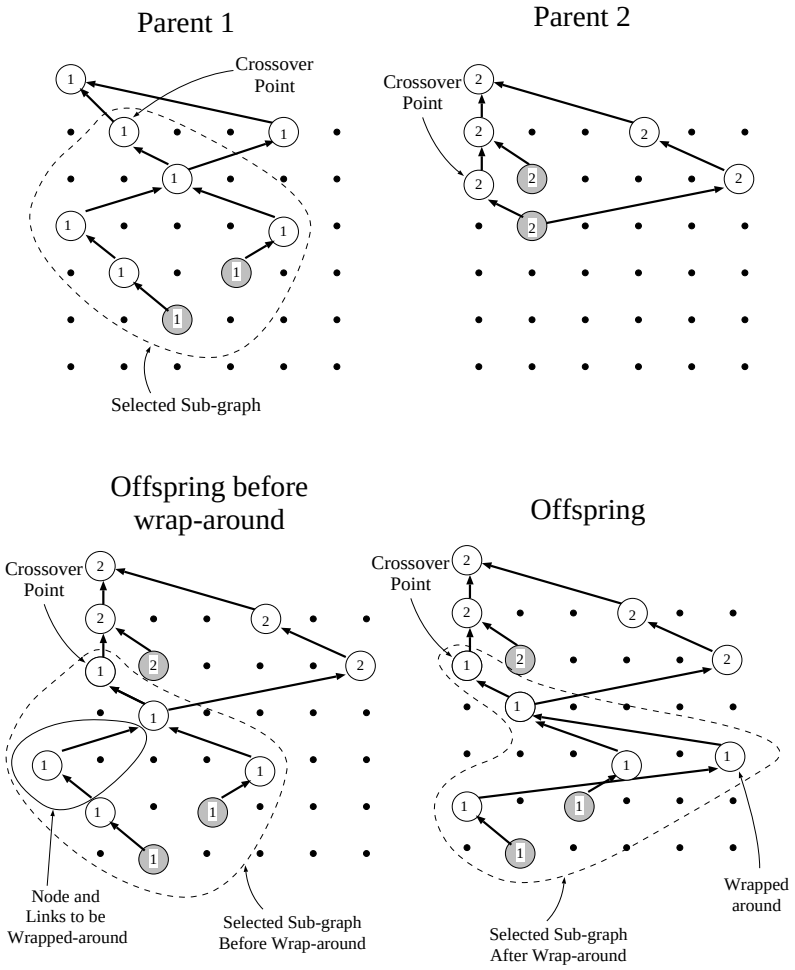

Figure 6: Sub-graph active-active node (SAAN) crossover.

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.

Obviously, for the 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, for example, 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.

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 a connected sub-set of the active nodes used to compute the output value of the crossover point selected in the first parent. This form of crossover may require the presence of introns to avoid the generation of invalid programs (in which some functions have 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).

Each of the above mentioned forms of crossover can offer some useful feature. For example, SSAAN crossover allows the insertion of any parts of a parent programs into any other locations of the other parent, while the operators using introns can be useful in the presence of dynamic fitness functions or epistatic problems.

At present a complete study of such operators has only been performed for a small set of problems from which it is difficult to draw any final conclusions on the superiority of one form of crossover over the others. However, the evidence collected until now has led us to use two hybrid forms of crossover which are a mixture of the ones described above.

The first one is a form of SSAAN crossover where both crossover points are selected at random among the active nodes. Conceptually, the operator works in two stages. Firstly, it extracts a complete sub-graph from the first parent (like the 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, in a second phase the lowest nodes of the sub-graph are pruned to make it fit. In doing so, care is taken to preserve terminals which are exactly at the maximum allowed sub-graph depth. Figure 7 illustrates this process. In order for this type of crossover to work properly introns are essential, in particular the inactive terminals in the lowest row of the second parent (unlike other introns, they have been explicitly shown in the figure to clarify the interaction between them and the functions and terminals in the sub-graph).

The second hybrid crossover is a form of SSIAN crossover which works exactly as the SSAAN crossover just described except that the crossover point in the first parent can be chosen also among the inactive nodes.

All the experiments reported in the following sections have been carried out with one of the two afore-mentioned operators.

3.3 Mutation

The standard GP technique of defining mutation as the swapping of a randomly selected sub-tree of an individual with a new randomly generated tree can be naturally applied in PDGP as well. It is sufficient to select a random active node in a program, generate a sub-graph which can be inserted into it, and perform the insertion. We call this form of mutation, *global mutation.*[4]

We have also found quite useful another form of mutation, *link mutation*, which makes local modifications to the connection topology of the graph. Link mutation selects a random function-node, 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 for the basic PDGP representation (e.g. node mutation), but we have not done extensive experiments with them, yet.

4 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 (the lowest row of nodes), which always contains only terminals, and proceeding layer after layer upwards until the output nodes are reached.

The output of each node could be stored in the representation of a node itself (e.g. as an additional slot in the structure representing a node). However, this would be a waste of resources as the output values are used only during the evaluation of the fitness of PDGP programs. A better alternative would be to use a single array to store temporarily the output values during the evaluation of each individual.

[4] In the current implementation of PDGP we use a less efficient mutation operator which is however equivalent to global mutation: we perform the crossover of an individual with a randomly generated new individual, which is then discarded.

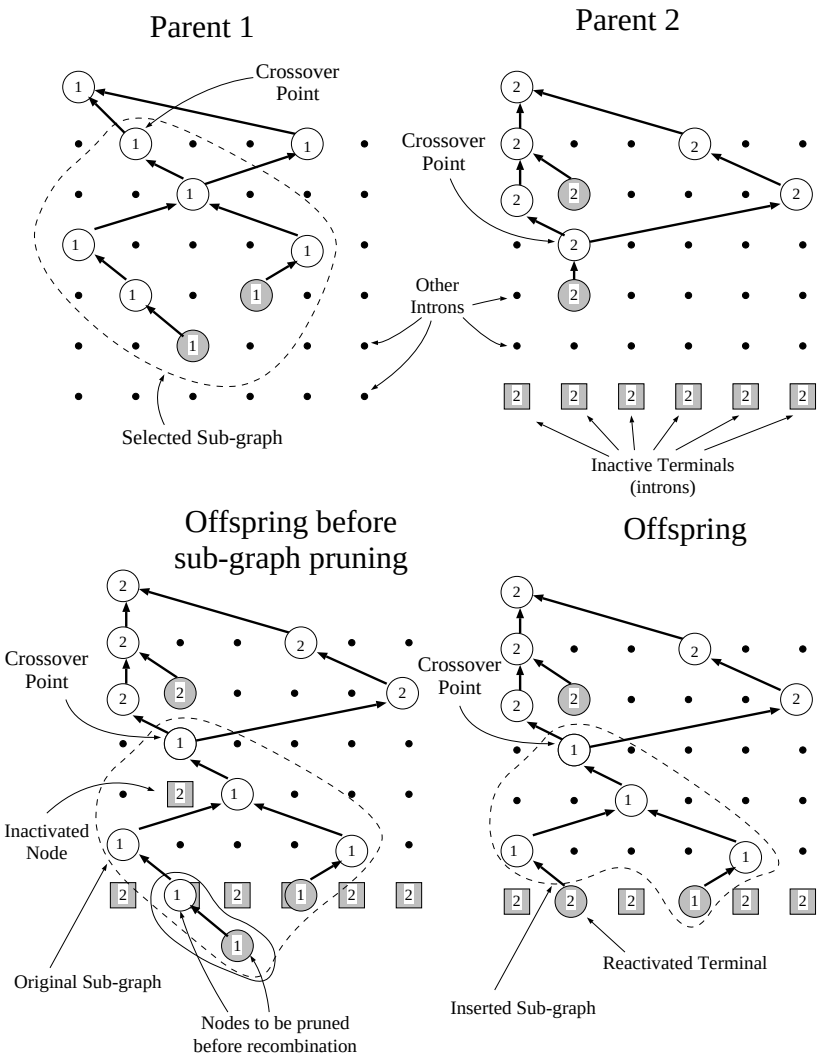

Figure 7: Sub-sub-graph active-active node (SSAAN) crossover with random crossover points.

```
eval(program):
begin
    Reset the hash table NodeValue.
    For each node N in the  first layer (the output nodes) do
        Call the procedure microeval(N)
        Store the results in a temporary output vector Out
    Return(Out)
end
```

```
microeval(N):
begin
    If NodeValue(N) is "unknown" then
        If N is a function then
            For each node M connected (as an argument) to node N do
                Call the procedure microeval(M)
                Store the results in a temporary output vector Out
            Call the procedure N(Out)
            Store the result R in NodeVal(N)
            Return(R)
        elseif N is a macro then
            Call the procedure N(M1,...Mn) where M1...Mn  are the
                 nodes connected (as  arguments) to node N
            Store the result R in NodeVal(N)
            Return(R)
        else /* N is a variable or a constant */
            Return(valof(N))
        endif
    else /* N has been already evaluated */
       Return(NodeVal(N))
    endif
end
```

Figure 8: Pseudo-code for the interpreter of PDGP (in the absence of nodes with side effects).

Given that the output of the nodes of each layer can only depend on the value of the nodes in the previous layer, an even more efficient alternative is to use a vector to temporarily store such values. This was the approach to evaluation we was used in our initial experiments where we did not use nodes with side effects and connections were limited to being between nodes in adjacent layers.

The current PDGP interpreter overcomes these limitations by using a totally different approach based on recursion and a hash table which stores partial results and control information. Let us first consider the case of nodes with no side effects. Then, the interpreter, the procedure `eval(program)`, can be thought of as having the structure outlined in Figure 8.

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 not it executes the corresponding terminal or function (calling itself recursively to get the values for the arguments, if any), otherwise it returns the value stored in a hash table.

It should be noted that the use of a hash table allows a total freedom in the connection topology of PDGP programs. In the absence of nodes with side effects accessing the hash table is, however, slightly slower than accessing a vector of values (not much slower as we hash on an integer key obtained by combining the coordinates of the node, and hashing on integers is very fast). This overhead could be removed by using an array with the same topology as the grid instead of the hash table.

It is also worth noting that although the interpreter described in Figure 8 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 order to imagine ways in which to add the handling of nodes with side effects it is now sufficient to consider what would happen if such nodes were used in conjunction with this simple interpreter. Obviously, the nodes in the program would work on a "read-many-execute-once" (RMEO) basis: they would perform their action (if any) only the first time they are called (i.e. once for each fitness case) as their value would then be cached and used thereafter. In some cases this might be a desirable property, but in general we want more freedom. Also, if the RMEO policy may work with functions, it can produce some really strange behaviours with macros. For example, the standard `IFLTE(arg1,arg2,arg3)` macro (which executes `arg2` if `eval(arg1)` is less than or equal to zero, `arg3` otherwise) would never check again the value of `arg1` to see which branch to execute and/or which value to return (the one returned by `arg2` or the one returned by `arg3`?).

Total freedom can be obtained by using terminals, functions and macros with different re-execution policies and let evolution choose which combinations are best for a particular application. For example, in addition to nodes with the standard RMEO policy we could define: nodes with an "execute-always" (EA) policy, nodes with a combination of EA and RMEO (for example `arg1` in `IFLTE` could be of type EA), nodes which are executed for a finite number of times or when certain conditions are satisfied (this would allow for example the handling of conflicts if PDGP programs with side effects were executed on a parallel machine), etc.

Introducing non-RMEO execution policies in the PDGP interpreter is trivial. It is sufficient to introduce more conditions in the procedure `microeval`. For example, the only change required by EA macros and functions is to copy the pseudo-code for standard RMEO nodes and remove the lines which store the result `R` in the hash table `NodeVal(N)`.

5 Extended Representations

It is possible to imagine many ways in which some of the constraints imposed on the graphs produced by PDGP could be removed without reducing the search efficiency significantly. In the following we briefly describe the extended representations and operators we have implemented and started testing.

A first obvious extension of the basic representation is allowing feed-forward connections between non-adjacent layers. With this representation any directed acyclic graph can be naturally represented within the grid of nodes, without requiring the presence of pass-through nodes. In order to implement this it is sufficient to store within each node also the vertical displacements of the nodes whose value is used as an argument by such a node. The link mutation operator has to consider these additional parameters when deciding where to reconnect a mutated link.

Once the previous extension is implemented, it becomes trivial to represent graphs with backward connections as well, for example with cycles. It is sufficient to accept negative or 0 vertical displacements. Obviously, the presence of recurrent connections requires some changes to the interpreter and to the crossover operator. The interpreter must be able to detect infinite loops and to terminate them somehow "nicely", for example by returning a default value after a given number of evaluations of the same node. With the present interpreter this is trivial to implement as it is sufficient to store in the hash table how many times a node has been evaluated in addition to its value. Also the change to the crossover operator is surprisingly simple: it is sufficient to perform a wrap-around of nodes and connection in the vertical direction before performing the wrap around in the horizontal one.

Adding labels to links is another extension of the basic representation which we have started exploring. If the labels are real numbers this technique allows the direct development of neural networks. The interpreter requires only minor changes: it has to multiply the value obtained via a link by the corresponding weight before using it as an argument for a node, and it has to pass the value of the label as an additional argument to macros.

However, the labels for the links can really be anything: if they are symbols of a language the representation can be used to develop finite state automata, transition nets, semantic nets, etc. In these cases the semantics of the evaluation is application specific. However, no change to the interpreter is necessary once the link labels are implemented, as virtually any semantics can be obtained by using macros.

The addition of labels to links allows the definition of additional operators, like *label mutation* which substitutes the label of a link with a randomly selected label from a label set $\mathcal{L}$. If $\mathcal{L}$ contains the random constant generator `random`, label mutation gives a mechanism to alter the weights of PDGP programs representing neural nets.

Examples in which some of the above-mentioned extensions are used and more details on the interpretation process will be given in the next section.

6 Experimental Results

In this section we will report on experiments which emphasise some of the features of PDGP. Some experiments involved large number of runs which have been used to study the performance of PDGP in various conditions (e.g. with different parameter settings). Other experiments are only meant to show the representational power of PDGP or its behaviour on relatively hard problems.

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 minimum 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 several experiments we used introns which, in our initial iterative interpreter 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. This is a very crude overestimate of the actual number of nodes to be evaluated with the current interpreter (about N/2) but guarantees a fair comparison with earlier results.

6.1 Exclusive-Or Problem

In order to study the representational capabilities of PDGP, in this section, we report on some experimental results obtained by applying PDGP to the exclusive-or problem, using different function and terminal sets. The problem is finding a parallel distributed program that implements the function

$$XOR(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{otherwise.} \end{cases}$$

Given the simplicity of the problem, in all the experiments reported in this section, the population size was very small: P=200 individuals. The maximum number of generation was G=20, the crossover probability was 0.7, the global mutation probability was 0.25 while the link mutation probability was 0.25. The GA used tournament selection with tournament size 7. The other parameters were "grow" initialisation method and SSIAN crossover. The fitness of a solution was the number of correct predictions of the entries in the XOR truth-table. With each function and terminal set we performed 20 runs (with different seeds for the random number generator) with three different grid sizes: 2×2, 2×3 and 3×4.

In a first set of experiments we tried to evolve *logic solutions* to the problem by using the function set $\mathcal{F}$={AND, OR, NAND, NOR, I} and the terminal set $\mathcal{T}$={x1, x2}. Apart from the use of the identity function I, these are the functions and terminals normally used in GP to solve Boolean classification problems.

Figure 9 shows three typical solutions to the XOR problem obtained by PDGP. The figure shows the actual output produced by our Pop-11 implementation of PDGP.[5] Active nodes and links are drawn

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

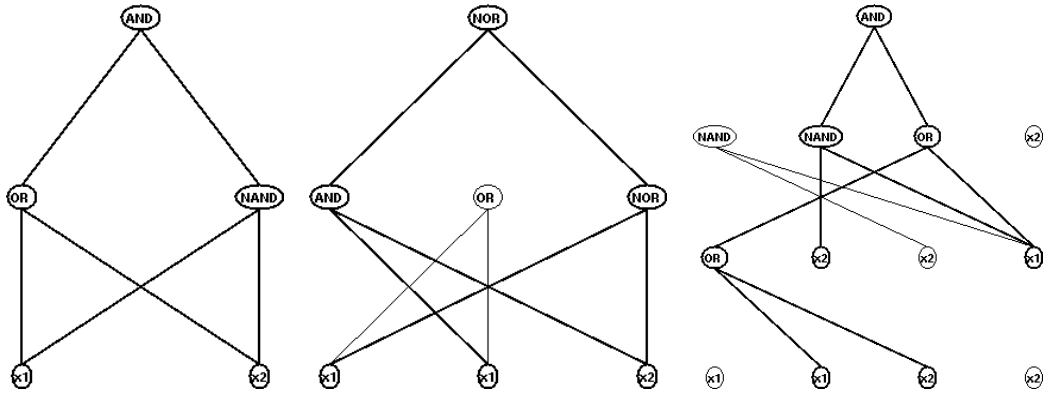

Figure 9: Symbolic networks implementing the exclusive-or function with Boolean processing elements (introns are drawn with thin lines).

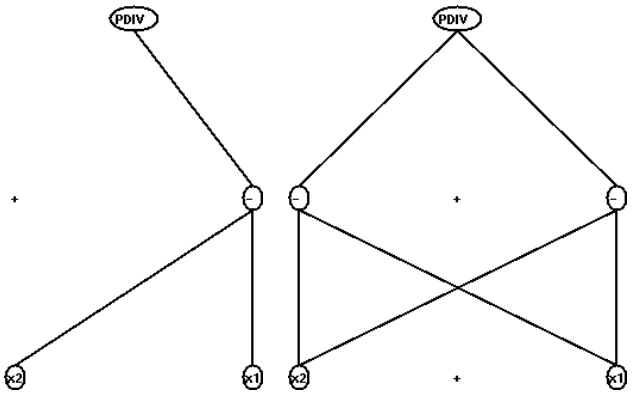

Figure 10: Algebraic network-like realisations of the exclusive-or function.

with thick lines, all the rest are introns.[6] These examples show how PDGP tends to reuse some nodes (in this case only terminals given the limited size of the grids used) and therefore to be parsimonious.

In other experiments we used the function set $\mathcal{F}$={+, -, *, PDIV, I} (PDIV is the protected division, which returns its first argument if the second is 0) and the terminal set $\mathcal{T}$={x1, x2} to evolve *algebraic solutions* to the XOR problem. In these and the other experiments involving analogue functions we used a "wrapper" that transformed the output of the program into `true` if it was grater than 0.5, `false` otherwise.

Figure 10 shows two typical solutions to the XOR problem obtained using algebraic operators.[7] Both networks represent the same quite interesting solution $XOR(x_1, x_2) = (x_1 - x_2)/(x_1 - x_2)$ which returns 0 if the divisor is 0 by exploiting the protection in PDIV. However, in the first network this is achieved more efficiently by reusing the value $(x_1 - x_2)$ without recomputing it twice.

Inspired by these "creative" analogue solutions we tried to develop *neuro-algebraic solutions* by using the same function and terminal set but adding random weights in the range $[-1, 1]$ to the links. As explained in Section 5, the weights acted as multipliers for the arguments of the functions in $\mathcal{F}$.

Figure 11 shows two typical solutions to the XOR problem obtained with PDGP using neuro-algebraic

[6] It should be noted that in the figure the output node, which, having coordinates (0,0), should be in the top-left corner, is actually centred horizontally for displaying purposes.

[7] For the sake of clarity, in this and the following figures introns are not drawn (they are simply represented by small crosses). It should also be noted that in general the interpretation of the non-commutative nodes (like "−" and PDIV) requires the knowledge of the order of evaluation of their arguments (the incoming links). However, for clarity reasons, we have preferred not to add this information to the figures in this section, as the order of evaluation could easily be inferred given the simplicity of the examples reported.

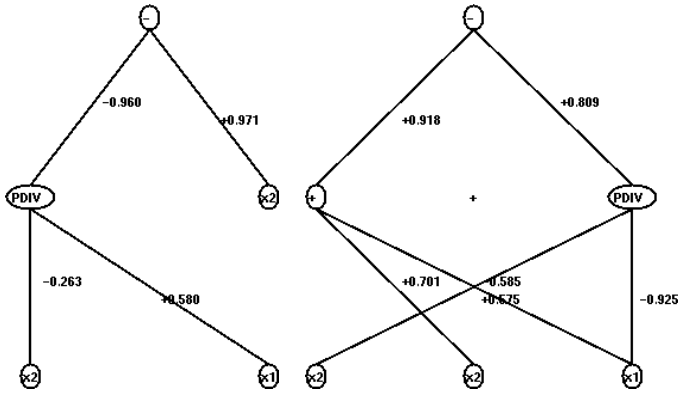

Figure 11: Exclusive-or implementations based on neuro-algebraic parallel distributed programs.

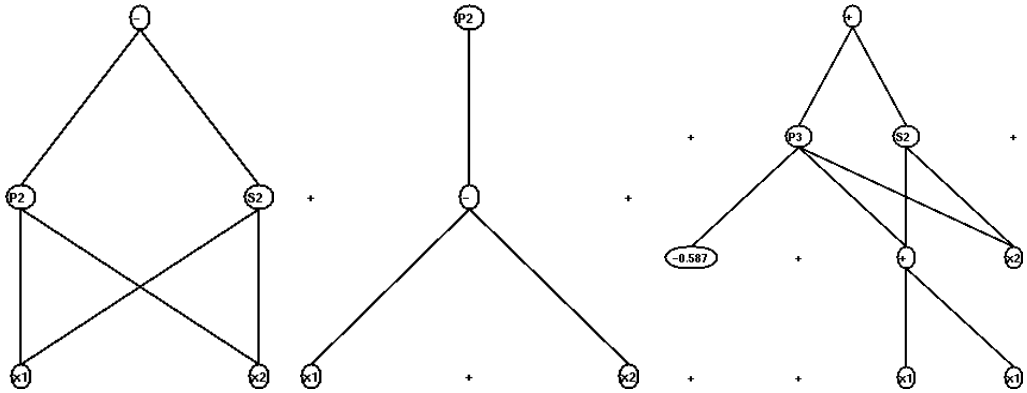

Figure 12: Weight-less neural networks implementing the exclusive-or function.

operators. Judging from these examples the addition of weights seems to enlarge the space of solutions available to PDGP. However, it is clear that neuro-algebraic solutions present the same kind of "black-boxness" typical of neural nets: they are hard to understand.

In part, this negative feature may also be present in the *weight-less neural solutions* obtained by using the function set $\mathcal{F}$={+, −, S2, S3, P2, P3, I} where S2 and S3 are neurons with sigmoid activation function, P2 and P3 are Π neurons (which compute the product of their inputs) and "+" and "−" simulate linear neurons. 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 12 shows three typical solutions to the XOR problem obtained with these operators. It is interesting to note that the solutions developed with smaller grids are again understandable quite easy as they do not use numeric coefficients. Actually the solution in the center of the figure, $XOR(x_1, x_2) = (x_1 - x_2)^2$, is quite "clever" and efficient. As soon as biases are used solutions become much harder to understand.

More standard forms of *neural solutions* can also be obtained with PDGP by using the same function set and terminal set as above but adding weights (in the range [-1,1]) to the links. For example, Figure 13 shows two typical solutions to the XOR problem obtained with these choices. Unfortunately, although perfectly valid, these solutions can only be understood with pen, paper and a pocket calculator.

In terms of computational effort different representations show quite different behaviours as detailed in Table 1. In the table, the columns reporting the computational effort E indicate that increasing the size of the grid tends to reduce 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 results in terms of total number of nodes to be evaluated N, the advantage of larger grids is not clear at all. In fact, in most cases N is smaller for smaller grids and, in general, it seems anyway worth

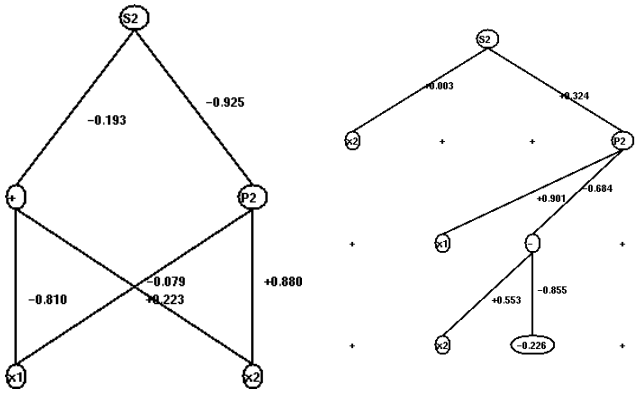

Figure 13: Neural realisations of the exclusive-or function.

	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 for the XOR problem for different PDGP representations

spending a few percent more evaluations of nodes and get a better solution in terms of size, execution speed and (possibly) generalisation.

In the case of neuro-algebraic solutions, the table indicates that the use of random weights makes the search much harder and that again increasing the size of the grid reduces the number of fitness evaluations but not necessarily the number of nodes to be evaluated. This might seem surprising at first as in theory by adding weights to the connections we can explore a much large space of programs. However, enlarging the search space does not necessarily mean to increase the frequency of solutions. As PDGP, like any other incomplete search method, can only explore part of the search space, it is quite likely that, at least for this problem, by adding weights we make a less efficient use of the available resources (the fitness evaluations).

A similarly increased computational effort is necessary to get weight-less neural solutions. This is probably due to the limited expressive power of neurons with respect to other classes of functions, at least for Boolean classification problems.

The combination of the negative effects of weights and neural processing elements leads to a considerable degradation of the performance of PDGP when searching the space of possible neural nets. In this case a large grid seems to be a relative advantage.

In general the increased computational effort required to develop programs with weighted links seems to indicate that the operators used by PDGP to optimise the topology and the processing elements of parallel distributed programs are probably ineffective in optimising the connection weights. Specialised operators, like weight mutation or gradient-based tuning, could probably solve this problem.

In summary, the results with the XOR problem show how PDGP can explore (although with different degrees of efficiency) a large space of programs which ranges from non-recurrent neural nets to classical tree-like programs without discontinuities. They also suggest that, when looking for logic and algebraic solutions, PDGP can compete, in terms of computational effort, with other well established machine learning techniques, like the standard back-propagation algorithm which requires from several hundreds to a few thousands training epochs to solve the XOR problem [24].

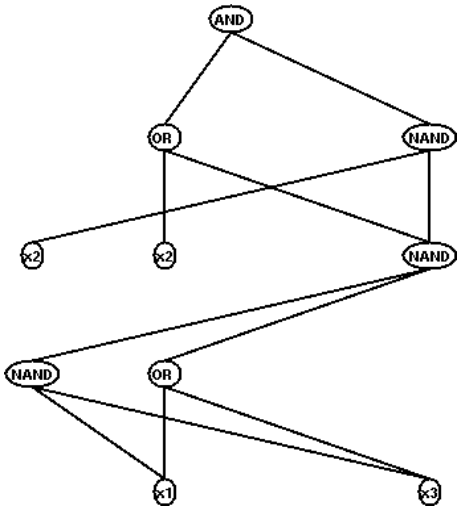

Figure 14: Parallel distributed program implementing the even-3 parity function.

6.2 Even-3 Parity Problem

In this section we report on some 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 1, 0 otherwise.

Although this is a simple problem which can be solved in a relatively short time, it is quite harder than the XOR problem. Also the problem has been used extensively to test standard GP. This led us to use the even-3 parity to study the behaviour of PDGP with different parameter settings and to compare its performance 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 14 shows a typical solution to the even-3 parity problem obtained in one run of PDGP using a grid with 4 columns and 4 rows.

In order to assess the influence of various parameters and operators on the behaviour of PDGP we tested it with different settings, performing 20 runs for each setting (with different initialisations of the random number generator) for a total of 140 experiments (2,800 runs). The parameters we varied (in all possible combinations) were: 1) the probability of mutation (0, 0.02 and 0.04), 2) the initialisation procedure ("full" vs. "grow"), 3) the crossover operator (SSAAN vs. SSIAN), 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).

The interpretation of the results of our tests (which are represented by a table with 140 rows and 11 columns) is not easy as projecting the data along one or two dimensions makes it difficult to see the correlations between groups of settings.

The best results in terms of number of fitness evaluations were obtained using a grid with 8 rows and 8 columns, global mutation (with probability 0.02), "full" initialisation, and SSAAN crossover. With these settings E=25,000 and N=1,625,000. This compares extremely favourably with the results described by Koza [16, 17] who 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 these experiments (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 $E < 50,000$. Interestingly in most of such experiments PDGP was using 8 rows and 6 or 8 columns. This suggests that (relatively) large grids make the search easier, like in the XOR problem. This would seem logical considering that PDGP tends to behave more and more like standard GP, i.e. to develop trees, as the width of the grid increases and that crossover using

sub-graphs can be more disruptive than crossover using trees.

However, again, 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. The superiority of this grid size is confirmed by the fact that the nine experiments with the best N had all the same grid configuration. The result confirms the observation we made in the previous section: the effort of finding more constrained (i.e. more parsimonious) solutions is a fraction of that of finding less constrained, tree-like ones. In comparative terms, with this grid size, PDGP solves 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 [17] was 44.6 nodes, this seems to be a considerable saving.

Obviously, no final conclusion can be drawn from the performance obtained with the best set of parameters for a single problem: after all it would be possible to optimise the parameters of standard GP for the even-3 parity problem and probably get results comparable to ours. The objective of this set of experiments was instead to indicate what kind of operators, grid configurations etc. to use (or not to use) to get successful runs with a relatively high probability.

Unfortunately, the parameters that gave the best results with small grids were sub-optimal for larger ones (in particular small grids seem to need higher mutation probability). Fortunately, the analysis of the data showed that, as long as the "full" initialisation method is used, for any set of parameters, there is at least a grid size for which performance is very good. This can readily be seen from the plots, shown in Figures 15 and 16, of the computational effort E and the number of nodes N vs. the size of the grid for the "full" initialisation method.

The basic lesson learnt from these experiments is that if the objective is to minimise the wall-clock time of each run (on a sequential computer) and get small solutions, then one should try and use a small grid. The risk in doing so is obviously that if the grid is too small there might be no solution at all in the search space (for example it is impossible to solve the even-3 parity problem with a 2×2 grid).

However, this is not an uncommon and unsolvable situation in many other machine learning paradigms. For example in neural nets, if the number of hidden neurons is insufficient for a particular problem, no satisfactory solution can be found. For this reason neural network designers usually try different topologies of increasing complexity until acceptable solutions are found.

In our experience it has always been quite easy to understand when the grid was too small. For example, a clear indication of this was the fact that, in repeated runs, PDGP got stuck at the same non-optimum fitness value.

If one is using a form of parallel computer which can evaluate several nodes, if not the whole program, in parallel, the importance of parsimony is reduced. In these cases larger grids might become advantageous as they guarantee a smaller number of fitness evaluations and present fewer risks of absence of solutions in the search space. Overfitting is, however, always a risk with large grids (except for problems, like the even-3 parity problem, where the entire space of possible inputs can be used for training).

6.3 Lawnmower problem

In this section we report on a number of experiments in which nodes with side effects were used. For our experiments we selected a well-known, extensively studied problem: the lawnmower problem. The problem consists of finding a program which can control the movements of a lawnmower so that it cuts all the grass in the lawn [17, pages 225–273]. The interesting facts about this problem are: a) its difficulty can be varied indefinitely just by changing the size of the lawn, b) although it can become very hard to solve for GP, the evaluation of the fitness of each individual requires only one interpretation of the program (the program must contain all the operations needed to solve the problem), c) it is a problem with a lot of regularities which GP without ADFs seems unable to exploit, d) GP with ADFs does very well on this problem.

In the lawnmower problem the lawn is a rectangular array of grass tiles and the lawnmower has a direction of movement which can take only four different values (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.

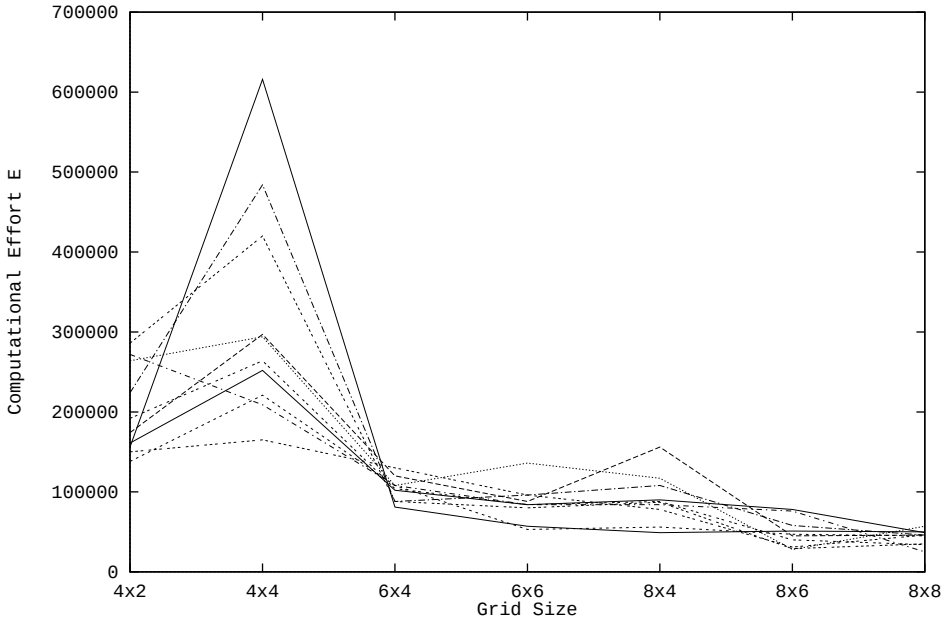

Figure 15: Computational effort E for the even-3 parity problem as a function of the grid size ("full" initialisation method). Different plots refer to different combinations of parameters and operators.

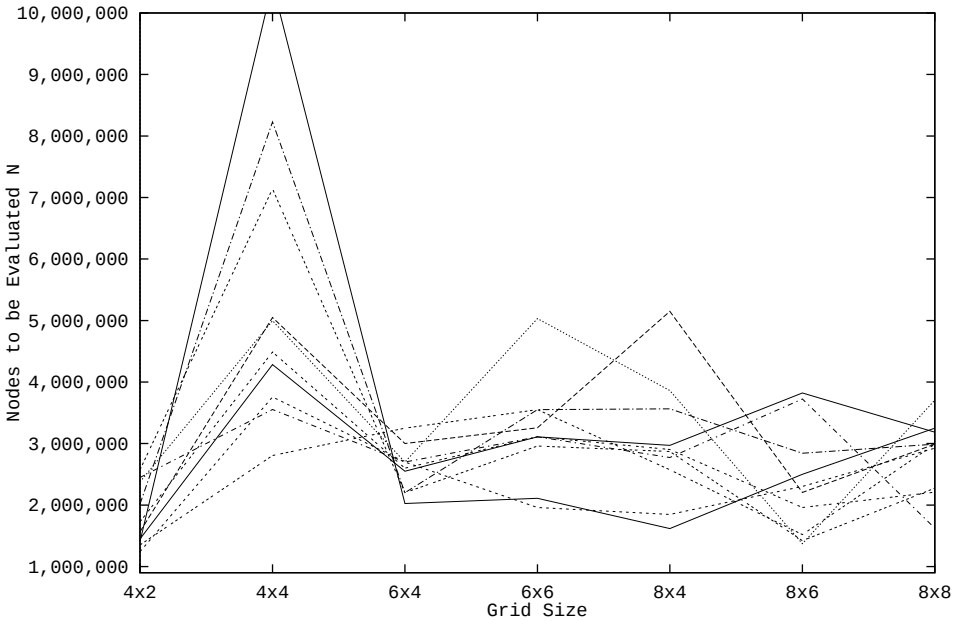

Figure 16: Nodes to be evaluated N for the even-3 parity problem as a function of the grid size ("full" initialisation method). Different plots refer to different combinations of parameters and operators.

In order to make the comparison with the results described in [17] possible, we used the terminal set $\mathcal{T}$={`MOW, LEFT, random`} and the function set $\mathcal{F}$={`V8A, FROG, PROG2, PROG3`}. The function `MOW` performs a step forward, mows the new grass patch and returns the pair of coordinates $(0, 0)$. `LEFT` performs a left rotation of the mower and returns (0,0). `random` returns a random pair of coordinates (x, y) with $x, y \in \{0, \cdots, 7\}$. `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 the single argument of `FROG`, 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.[8]

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 with size P=1000, a maximum number of generations G=50, a grid with 8 rows and 2 columns and the "grow" initialisation method. All functions with side effects used an "execute-always" policy.

Figure 17 shows a typical program obtained in our experiments with an 8×8 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 (note the prevalence of `PROG3` nodes). 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 18 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 19 shows a plot of the computational effort E necessary to solve the lawnmower problem as a function of the complexity of the problem (quantified by the number of cells in the lawn).

The results shown in the plot are amazing. Standard GP (without ADFs) requires 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 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 linear regression plotted in the figure, which has equation $3286 + 26.8 \times L$, L being the size of the lawn (regression coefficient r=0.949), and the linear regression for standard GP [17, page 266] shows that PDGP scales up about **2,300 times better**. Also, as shown in Figure 20, 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. If we compare the number of nodes in our programs with the average structural complexity of standard GP programs (which ranges from 145.0 for the 32-cell lawn to 408.8 for the 96-cell one) we observe an improvement of 10.5 to 29.2 times.

PDGP clearly outperforms GP without ADFs. The situation for GP improves somehow when as 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 ability of the SSIAN crossover or to the particularly narrow grid used, which enforces strongly graph-like (as opposed to tree-

[8] Unlike in [17], we did not use `PROGN` for efficiency reasons.

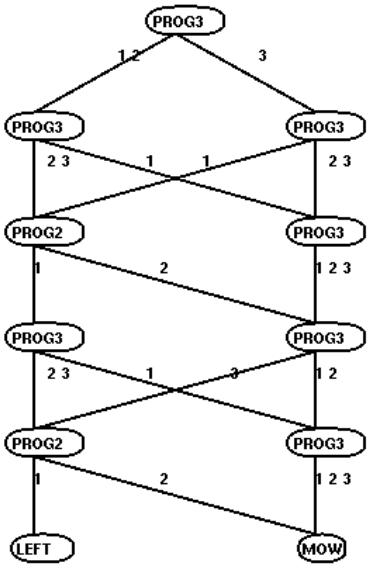

Figure 17: Parallel distributed program solving the lawnmower problem with a lawn of $8 \times 8 = 64$ cells.

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 18: Behaviour of the program in Figure 17.

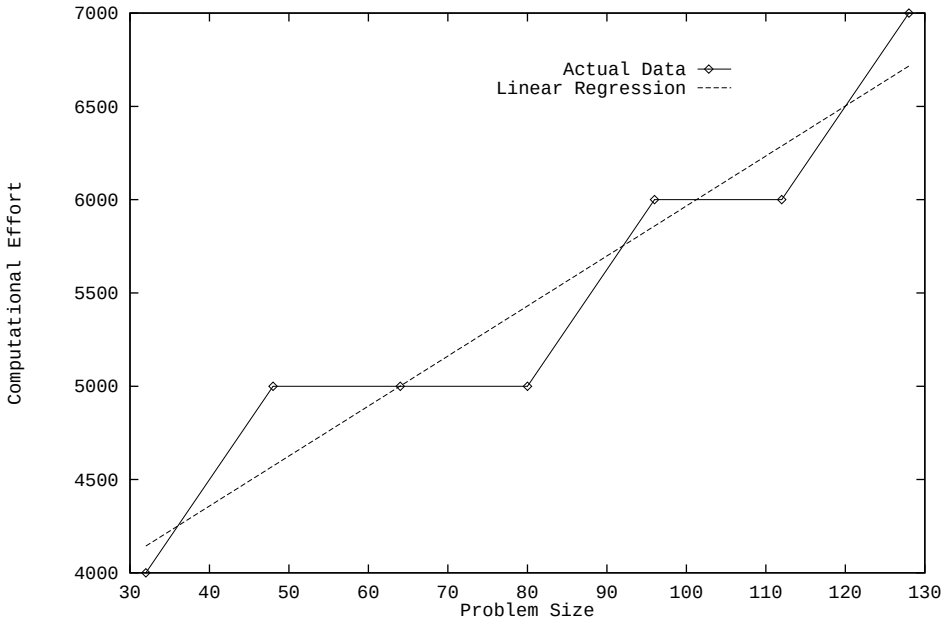

Figure 19: Computational effort E necessary to solve the lawnmower problem as a function of the complexity of the problem (number of cells in the lawn).

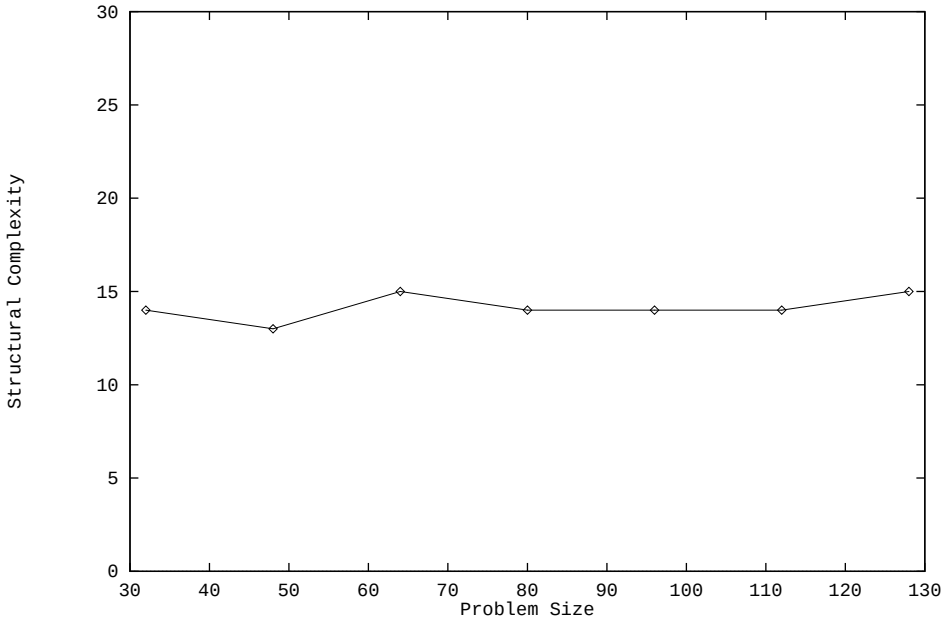

Figure 20: Average structural complexity of the PDGP programs which solve the lawnmower problem as a function of the complexity of the problem.

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 runs for each configuration.

Figure 21 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 22, shows that by increasing the width of the grid, the "treeness" of the programs increases (when the grid is very large PDGP really tends to behave like a kind of standard GP with a special form of crossover). As somehow expected by the results mentioned above, PDGP performance decreased quite dramatically 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) despite the relatively shallow trees PDGP could develop (this is probably the reason why the structural complexity was always less than a half with respect to standard GP). In Section 6.5 we will further investigate the behaviour of PDGP crossover when the width of the grid is changed and we will compare it with the behaviour of standard GP crossover on a set of problems in which the same maximum tree/graph depth was used.

Really, however, the power of PDGP on this problem derives from its ability to discover and reuse building blocks. The fact that it outperforms GP with ADFs suggests that parametrised reuse is not necessarily an advantage with respect to the non-parametrised reuse of standard PDGP (i.e. PDGP without ADFs).[9]

6.4 Symbolic Regression

To investigate further the matter of parametrised vs. non-parametrised reuse, we applied PDGP to a symbolic regression problem on which GP performance where approximately the same with (E=1,440,000) or without (E=1,176,000) ADFs with a population of P=4,000 individuals (see [17, pages 110–122]).

The problem is to find a function (i.e. a program) which fits 50 data samples obtained from the sextic polynomial $p(x) = x^6 - 2x^4 + x^2$. The samples are obtained by selecting 50 random values x_i in the range [-1,1] and associating them with the corresponding value of $p_i = p(x_i)$. The fitness function is $\sum_i |p_i - \texttt{eval(prog,}x_i\texttt{)}|$. The stopping criterion for a run is the discovery of a program that fits all the datapoints with an absolute error smaller than 0.01.

The parameters we used for PDGP were: population size P=1000, maximum number of generations G=100, SSIAN crossover, "grow" initialisation method, no mutation, grid with 3 columns and 6 rows. The terminal set was $\mathcal{T}$=`[x,random]` (where `random` is a constant random generator with values in the range [-1,1]) while the functions set was the standard $\mathcal{F}$={+, -, *, PDIV, I}.

Figure 23 shows a typical program discovered by PDGP which fully solves the problem. With a very clever use of "pieces of wire" (the identity function I), the program is actually computing the expression $(x - x^3)(1 - x^2)x$ which is equivalent to $p(x)$.

The computational effort shown in 20 runs with different random seeds was surprisingly small: E=91,000. While we expected somehow better performance than standard GP (in fact we obtained a nearly 16-fold reduction of E), we did not expect such a large difference (about 13 times) between PDGP and GP with ADFs. A possible explanation for this phenomenon is that in some problems (like the one described in this section) the use of parametrised ADFs allows GP to explore a much larger space of possible programs in which, however, the frequency of solutions is not significantly higher than in the original space. This is quite easy to understand if we consider, for example, that ADFs show somehow different behaviours in different parts of a program (because they receive different arguments). This means that in order to discover if an ADF is good or not, GP has also to "understand" with which kind of arguments the ADF can be used properly. On the contrary the non-parametrised reuse available in standard PDGP seems to increase the size of the space of possible programs (it allows the development of graph-like in addition to tree-like programs) in a direction in which the frequency of solutions increases as well.

[9] Also PDGP can use ADFs, even if in this paper we will not present any examples of their use.

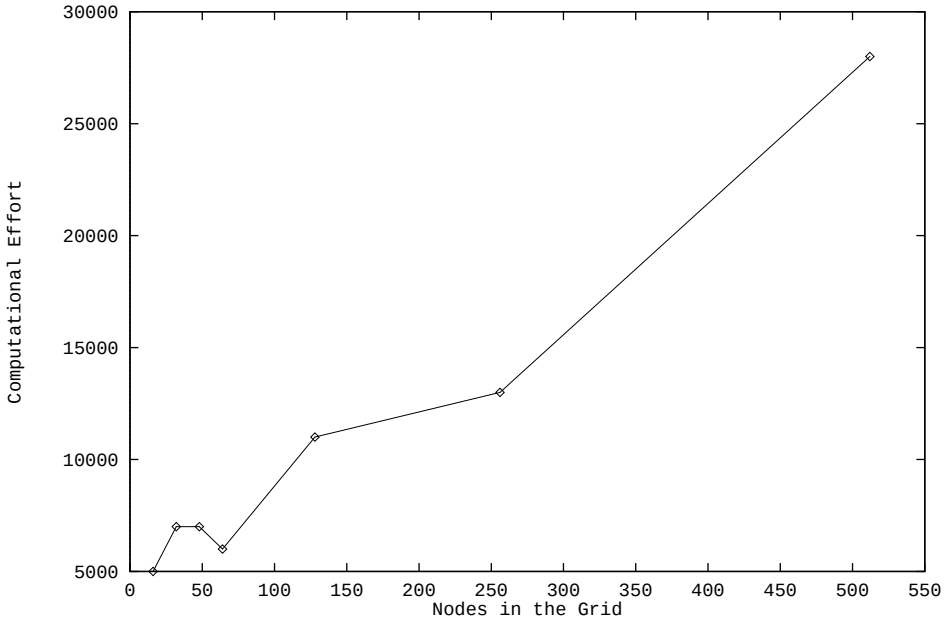

Figure 21: Computational effort E necessary to solve the 64-cell lawnmower problem as a function of number of nodes in the PDGP grid.

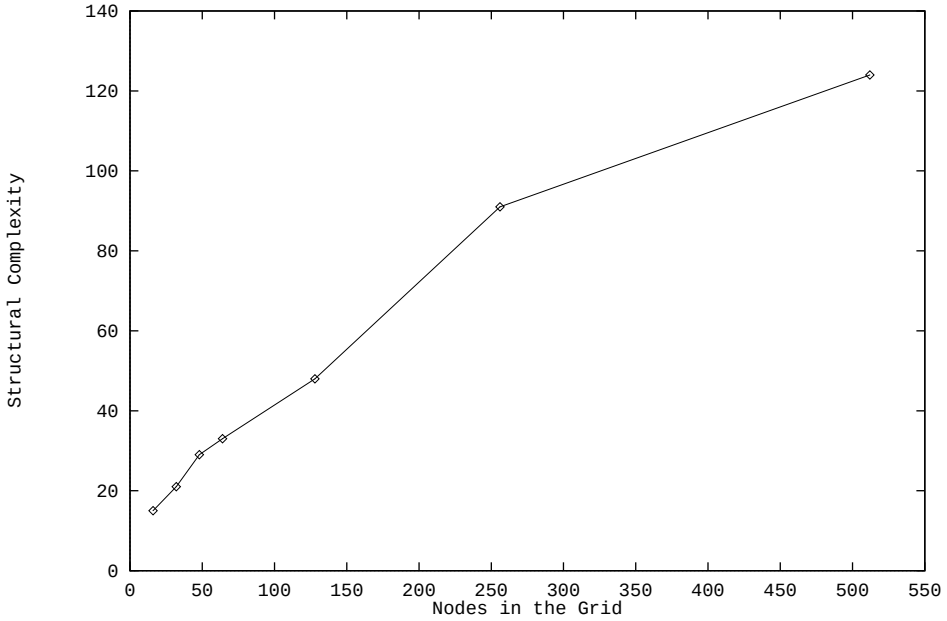

Figure 22: Average structural complexity of the PDGP programs which solve the 64-cell lawnmower problem as a function of number of nodes in the PDGP grid.

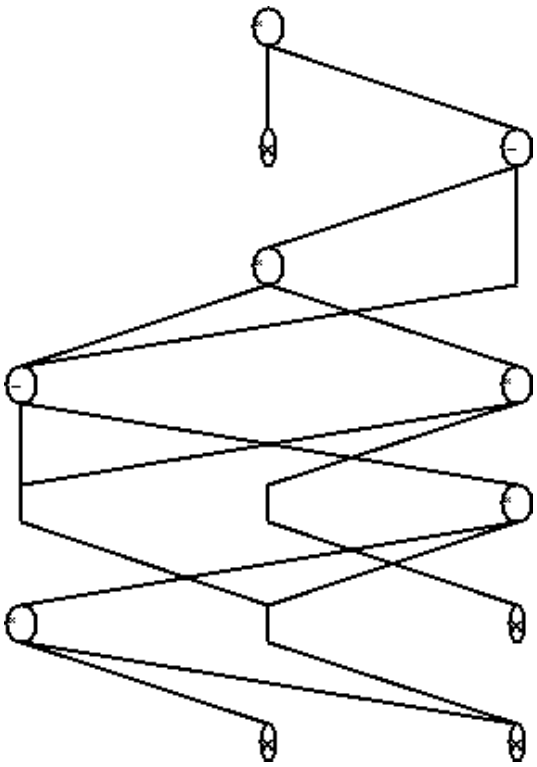

Figure 23: Parallel distributed program implementing the sextic polynomial $x^6 - 2x^4 + x^2$.

	Maximum Depth (D)					
Columns	3	4	5	6	7	8
2	400	1000	2000	2800	3200	3800
4	600	7200	8400	3800	5600	17400
8	1000	4800	12000	16800	21000	60000
16	1400	10800	19200	8400	29600	176400
32	1000	18200	27600	56400	114400	N/A
64	1400	30000	56000	57400	369600	N/A
128	1000	18200	64000	193600	N/A	N/A

Table 2: Computational effort E necessary to solve the **MAX-depth-D{*,+}{0.5}** problems using SSAAN crossover.

6.5 MAX Problems

The MAX problems are a class of problems introduced by Gathercole and Ross [8] 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 values 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}** are 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 approximately 30% **MAX-depth-6{*,+}{0.5}** problems and basically no **MAX-depth-D{*,+}{0.25}** problem with $D \geq 6$.

The reason for this 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. The difficulty resides in the fact that during the first stage of evolution multiplication operations will only tend to survive in the upper nodes in the tree (where they can act on numbers bigger than one), and later, when they should also be used at lower levels in the tree, crossover is unable to move them given the depth limit.

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 on 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-depth-D{*,+}{0.5}** and **MAX-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. We repeated 10 experiments (with different random seeds) for each combination of D, c, and number of columns. The results are shown in Table 2 and 3.

Despite the slight inaccuracies in some data due to the relatively small number of runs, the situation depicted by these tables is quite clear. SSAAN crossover in conjunction with narrow grids transforms a problem which can be nearly impossible to solve for standard GP into a very very easy problem. Actually PDGP is able to solve it with a computational effort E comparable with the one necessary to find solutions to the XOR problem or the lawnmower problems without showing any problems for any values of D. But again, for relatively hard problems (with $D >= 6$), performance seems to degrade quite quickly (super-linearly) as the number of columns in the grid is increased.

The width-scaling results observed in the MAX problems and the lawnmower problems seem to

	Maximum Depth (D)					
Columns	3	4	5	6	7	8
2	600	1000	4000	2400	3600	5800
4	800	1200	23400	28000	16200	18000
8	800	1400	26000	41600	39200	88200
16	1000	2000	58800	57200	172200	167200
32	1200	2400	79200	147000	N/A	255200
64	1400	2600	N/A	N/A	N/A	N/A
128	1000	2800	123200	N/A	N/A	N/A

Table 3: Computational effort E necessary to solve the **MAX-depth-D{*,+}{0.25}** problems using SSAAN crossover.

suggest that the adverse interaction between the maximum tree depth and crossover hypothesised by Gathercole and Ross might be present in PDGP as well. If we assume this, then the fact that PDGP crossover does not have any problems when the width of the grid is relatively small can only be explained by imagining that PDGP, thanks to the extensive reuse of unparametrised sub-programs, discovers quickly, before diversity is lost, how to use the two kinds of useful building blocks for this problem (addition and product).

However the hypothesis that PDGP crossover can adversely interact with the depth of the grid seems very unlikely. In fact, 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. This seems to suggest that, more likely, both in PDGP and in GP the adverse interaction is between crossover and some feature of the problem other than the maximum depth.

In PDGP this feature must obviously be related to the maximum width of the programs. In problems where solutions with a very high degree of regularity are better (like the MAX and lawnmower problems) it is possible to imagine that crossover, after an initial constructive phase, might have a disruptive effect: it might tend to break the local or global symmetries build in the previous phase. In PDGP by increasing the width of the grid we require the algorithm to produce larger and larger, very symmetrical solutions which, due to the hypothesised disruptive effect, would become more and more unlikely. More research will be needed to investigate this topic.

6.6 Encoder-Decoder Problem

In all the work described in this paper we have used two-dimensional grids with a rectangular shape. This is not a limitation of PDGP: it just seemed to be the most natural choice for the particular experiments we performed. For other applications it might make sense to use other shapes. For example, it would make sense to use triangular grids with layers of increasing width if one wanted to develop classification programs or to develop tree-like structures. In this section we present an interesting application of irregular grids to the 4-bit binary encoder-decoder problem.

The problem is to find a program capable of first encoding (with a 2 bit binary code) and then decoding binary input-vectors of the form {1,0,0,0}, {0,1,0,0}, etc. The program had four output nodes and four input variables. In order to force PDGP to develop programs that could encode and decode the input patterns we used a sand-glass-like grid obtained by creating a narrowing in the middle of a rectangular grid. In particular, the grid had nine rows with the following widths: 4, 4, 4, 4, 2, 4, 4, 4 and 4. The objective of the experiment was to discover a program whose outputs reproduced the inputs and in which no terminals were above the narrowing in the grid. Such a program would guarantee that the sub-graph in the lower part of the grid would perform some form of encoding of the input patterns while the upper sub-graph would decode them.

The function set was $\mathcal{F}$=[AND, OR, NOR, NAND, I] while the terminal set was $\mathcal{T}$=[x1, x2, x3, x4].

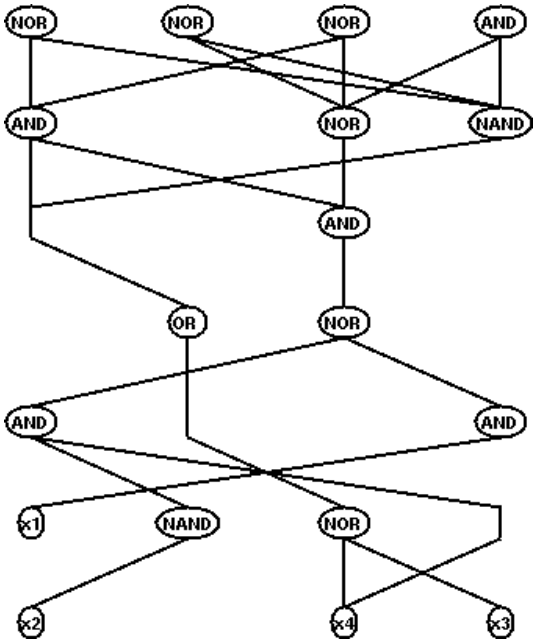

Figure 24: Solution to the 4-bit binary encoder-decoder problem produced by PDGP.

We used "full" initialisation, a relatively large population with P=4,000 individuals, a maximum number of generations G=200, a link mutation probability of 0.045 and a global mutation probability of 0.005. As usual, the crossover probability was 0.7.

In preliminary tests we immediately realised that, with the normal crossover and mutation operators, PDGP tended to discover the invalid solution in which the terminals x1, x2, x3 and x4 were positioned in the first row of the grid (the output layer). For this reason we used a specialised form of crossover and mutation which simply kept trying generating new offspring until one was produced in which all the terminals were not above the narrowing.[10] Figure 24 shows a 100% correct program produced in the second run of PDGP with these operators. The binary code used in the nodes in the narrowing is the somehow unusual code:

Input	Code
1000	01
0100	11
0010	10
0001	00

It is interesting to note that exactly the same idea used here to discover binary encoder-decoders could be used to discover compression and decompression algorithms, dimensionality-reduction algorithms, visualisation algorithms, tools for statistical analysis, non-linear principal component algorithms, clustering algorithms, etc.

6.7 Finite State Automata Induction Problem

Up until now we have presented results involving the basic PDGP representation for directed acyclic graphs without and with labelled links. In this section we want to show how, by using all the extensions presented in Section 5, the PDGP can actually search very efficiently a complete space of directed cyclic and acyclic graphs.

[10] Much more efficient forms of crossover and mutation could have been used if this one did not work. For example, a SSAAN crossover which does not move terminals would have guaranteed validity of all offspring.

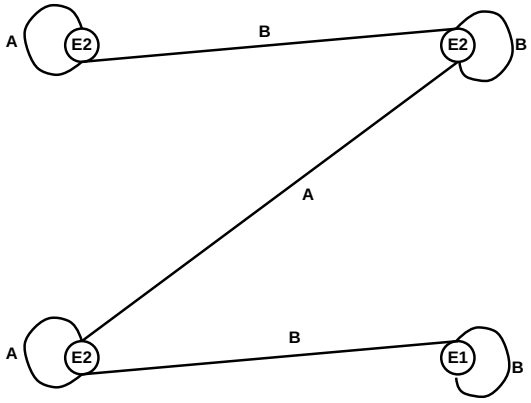

Figure 25: Finite state automaton capable of recognising the language `L=a*b*a*b*`.

The problem on which we applied the full PDGP representation was inducing, from positive and negative example, a deterministic finite state automaton capable of recognising a prefixed language (for more details on the problem and on past and recent results see [5]). The language we chose for our tests was the regular language `L=a*b*a*b*` consisting of all sentences with 0 or more `a` symbols followed by 0 or more `b` symbols followed by 0 or more `a`'s followed by 0 or more `b`'s. For example the sentence `abbbbab` is in `L` while the sentence `babab` is not.

For these experiments we use a population with P=2,000 individuals, a maximum number of generations G=200, a regular 4×4 grid in which the node in the top left corner was considered to be the start node, and the "grow" initialisation method. The crossover operator was a form of SAAN crossover with wrap around of nodes and links in both horizontal and vertical direction. The probability of crossover was 0.7, the probability of link mutation was 0.09, and the probability of global mutation was 0.01.

The link set included the labels [A B] which represented the two different kinds of input symbols. The function set included the macros [N1 N2 E1 E2]. The macros N1 and N2 represent non-stop states in the finite state machine with one and two outgoing links. respectively. When called they just check to see if the current symbol in the input sentence is present on one of their links. If this is the case they remove the symbol from the input stream and pass the control to (i.e. evaluate) the node connected to the link labelled with the matching symbol. If no link has the correct label, `false` is returned to the calling node. The same happens if no more symbols are available in the input stream. The nodes E1 and E2 represent terminal states. They have exactly the same behaviour as the `N` nodes except that if the end of the sentence is reached they return `true`. We also used a non-empty terminal set T=[DONE] where the node DONE behaves like a terminal state without output links. With these macros (whose execution policy is of type "always execute") no change is required to the standard PDGP interpreter. It is only necessary to reset the symbol stream before the `eval` procedure is called by the fitness function.

In the experiments we used 165 positive and 485 negative example sentences including up to 20 symbols. In 10 runs, all successful in developing a general automaton (not just a 100% fit automaton), we measured a computational effort E=22,000 and an average structural complexity of the 4.2 nodes (in 2 runs out of 10 there was a trailing DONE in the automaton) which compare very favourably with the results reported by others. Figure 25 shows one of the solutions found by PDGP (this is the only program we have redrawn due to the inability of the graphic routines in our current implementation to draw self connections with labels properly). The execution of the automaton starts from the node in the upper left corner of the figure and proceeds following the outgoing links (which can be recognised as they depart from the bottom of the nodes like in all the other programs shown in this paper).

6.8 MONK's Problems

The problems described in the previous sections were all relatively simple. One might wonder whether PDGP would really be able to solve harder problems. Although our current implementation is certainly

inefficient[11] and, therefore, unsuitable for that, we have started applying PDGP to some harder problems involving large sets of fitness cases and several input variables. In this section we report on a set of experiments in which PDGP was used to tackle the so-called MONK's problems.

The MONK's problems are a collection of binary classification problems over a six-attribute discrete domain which have been used extensively to compare different machine learning techniques (the results of a contest between 25 different techniques in described in detail in [30]). The ranges for the attributes A_i are: $A_1 \in \{1,2,3\}$, $A_2 \in \{1,2,3\}$, $A_3 \in \{1,2\}$, $A_4 \in \{1,2,3\}$, $A_5 \in \{1,2,3,4\}$ and $A_6 \in \{1,2\}$. By varying the values of the attributes 432 different examples can be constructed. The output value is either 0 or 1, depending on the class a particular tuple of attributes belongs. In the so-called MONK-1 problem, the corresponding output value is computed using the logic formula: $(A_1 = A_2) \vee (A_5 = 1)$. In the MONK-2 problem the output is given by the formula: $A_n = 1$ for exactly two values of n (with $n \in \{1,2,\cdots,6\}$). In the MONK-3 problem the desired output is given by $(A_5 = 3 \wedge A_4 = 1) \vee (A_5 \neq 4 \wedge A_2 \neq 3)$. In all problems the training set is a fraction of the complete set of examples: 124 examples for MONK-1, 169 for MONK-2, and 122 for MONK-3. The training set for MONK-3 contains 5% misclassified examples (6 out of 122).

We have applied PDGP to the three MONK's problems by devoting to them three runs each. The parameters for the runs were: a population size P=1000, a maximum number of generations G=200, the usual crossover probability 0.7, link mutation with probability 0.045, global mutation with probability 0.005, the "grow" initialisation method, and a grid with 9 rows and 4 columns.

In order to be able to evolve a logic solution to the problems we used the function set $\mathcal{F}$=[AND, OR, NAND, NOR, EQ, NEQ, I], where EQ (NEQ) is a function with two arguments which is true (false) if the arguments are equal. We also had to "binarise" the attributes A_i. For example, the attribute A_2 which can take three different values was encoded using three binary variables x21, x22 and x23 which are set according to the rule `x2j=`$\begin{cases} 1 & \text{if } A_2 = j \\ 0 & \text{otherwise} \end{cases}$. As a result we obtained a terminal set $\mathcal{T}$=[x11, x12, x13, x21, x22, x23, x31, x32, x41, x42, x43, x51, x52, x53, x54, x61, x62] including 17 binary variables.[12]

Within three attempts PDGP was able to solve MONK-1 in one run (with 100% correct classification of the test set), and to get between 6 and 12 misclassification errors on the training set for MONK-3. Of the three end-of-run programs for the MONK-3 problem, the one which made 6 errors on the training set correctly classified 100% of the test set (which is noise free). Figures 26 and 27 show the two 100% correct solutions found for MONK-1 and MONK-3, respectively.

Unfortunately, we were unable to solve the MONK-2 problem. We believe the reason for the luck of success on this problem is the fact that the rule "$A_n = 1$ for exactly two values of n" expands to an extremely large Boolean expression with little reuse of sub-expression, in the binarised version of the problem, which could not possibly fit into the small grid used.

However, the overall results shown by PDGP on the MONK's problems are aligned to those produced by the best machine learning techniques: in [30] no technique was able to perform 100% correctly on all three MONK's problems. For example neural nets could not avoid over-fitting the noise in the MONK-3 problem. Actually, only five techniques out of 25 could perform 100% correctly on the MONK-3 test set. In this respect, the fact that PDGP, in all three runs of the MONK-3 problem, avoided over-fitting seems very promising.

7 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 in which symbolic and sub-symbolic (neural, numeric, etc) primitives can be combined in a free and natural way as can primitives with and without side effects.

The grid-based representation of programs used in PDGP allowed us to develop efficient forms of

[11] Pop-11 is extremely good for fast prototyping and for exploring alternative algorithms, but it is also much slower than C/C++.

[12] The same form of binarisation of the attribute of the MONK's problems was used in [30] to train multi-layer neural nets.

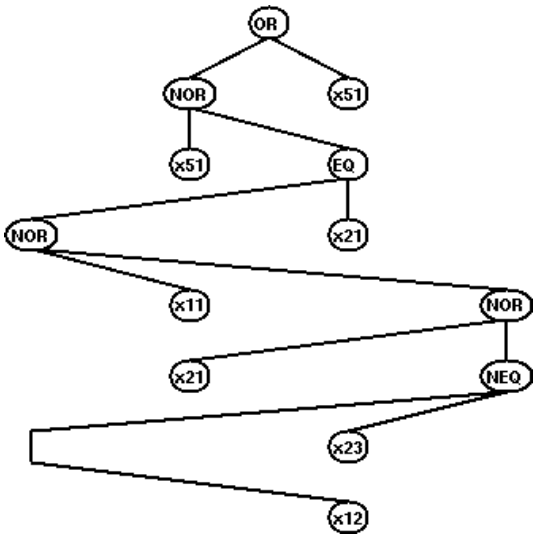

Figure 26: Solution to the MONK-1 problem produced by PDGP.

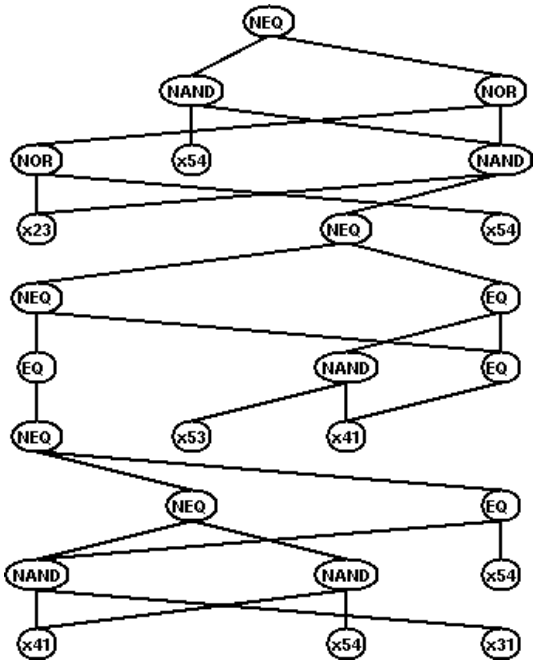

Figure 27: Solution to the MONK-3 problem produced by PDGP.

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 and the reuse of building blocks by changing the number of columns in the grid. Future research will be devoted to the possibility of evolving the shape and size of the grid during PDGP runs.

When programs do not include functions or terminals with side-effects, our grid-based representation of programs can be directly mapped onto the nodes of some kinds of fine-grained parallel machines. This can 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.

In this paper we have gone beyond our initial simple form of program interpretation inspired to the propagation of activation in the neurons of feed-forward neural nets. By using a more sophisticated interpreter we have been able to solve the problems of executing general graph-like programs, possibly including recurrent connections and nodes with side effects, on a sequential computer. We also think that, through the use of macros, functions and terminals that use special execution policies, like "execute-if-no-clash" or "lock-execute-unlock", PDGP could naturally be used to develop parallel distributed programs for any kind of parallel machine, too. More research will be done to corroborate this belief.

The 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 and Automatically Defined Links (ADLs) (the results, not reported, of experiments with ADLs are particularly promising).

It should be noted that ADLs are just one of the new representational possibilities opened by PDGP and its powerful interpreter. PDGP is an efficient paradigm to optimise 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. Also, cellular encoding can naturally be extended to PDGP, thus creating nearly infinite new possibilities, the weirdest of all possibly being using the direct graph representation of PDGP to develop GP trees.

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, Forrest H. Bennett III, and John R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. 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 3, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[3] 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.

[4] 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.

[5] Scott Brave. Evolving deterministic finite automata using cellular encoding. 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 39, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[6] 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.

[7] Herman Ehrenburg. Improved direct acyclic graph handling and the combine operator in 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 285, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[8] Chris Gathercole and Peter Ross. An adverse interaction between crossover and restricted tree depth in 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 291, Stanford University, CA, USA, 28–31 July 1996. MIT Press. to appear in.

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

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

[11] 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.

[12] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Riccardo Poli. Some steps towards a form of parallel distributed genetic programming. In *Proceedings of the First On-line Workshop on Soft Computing*, August 1996.

[24] D.E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1-2.* MIT Press, Cambridge, MA, 1986.

[25] 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.

[26] Astro Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3. MIT Press, Cambridge, MA, USA, 1996.

[27] Astro Teller and Manuela Veloso. A controlled experiment: Evolution for learning difficult image classification. In *Seventh Portuguese Conference On Artificial Intelligence*. Springer-Verlag, 1995.

[28] Astro Teller and Manuela Veloso. 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, Pittsburgh, PA, USA, 1995.

[29] Astro Teller and Manuela Veloso. Neural programming and an internal reinforcement policy. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 186–192, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[30] S. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The monk's problems – a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, School of Computer Science, Carnegie Mellon University, 1991. Available via anonymous ftp from archive.cis.ohio-state.edu, file /pub/neuroprose/thrun.comparison.ps.Z.

[31] 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.