

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

Abstract

This chapter describes Parallel Distributed Genetic Programming (PDGP), a 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 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 chapter describes the representations, the operators and the interpreters used in PDGP, and illustrates its behaviour on a 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 (Koza, 1992). 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(a). The set of possible internal (non-leaf) nodes used in GP parse trees is called *function set*, $\mathcal{F} = \{f_1, \dots, f_{N_F}\}$. \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* $\mathcal{T} = \{t_1, \dots, t_{N_T}\}$. \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. For example, in their simplest form, crossover works by replacing a random subtree in one parent program with another randomly selected subtree taken from another parent program, while mutation replaces a random subtree with a new randomly generated subtree. To increase the modularity of the programs discovered by GP and the efficiency of the algorithm, it is possible to use a technique known as Automatically Defined Functions (ADFs) (Koza, 1992; Koza, 1994). When ADFs are used one branch of the root node is interpreted as the main program, while the other branches are interpreted as subroutines which the main program can call (ADFs are part of the function set of the main program). ADFs can be seen as individual-specific parametrised building blocks. Sometimes ADFs are part of the function set of other ADFs, which allows the evolution of complex hierarchies of function calls. When ADFs are used

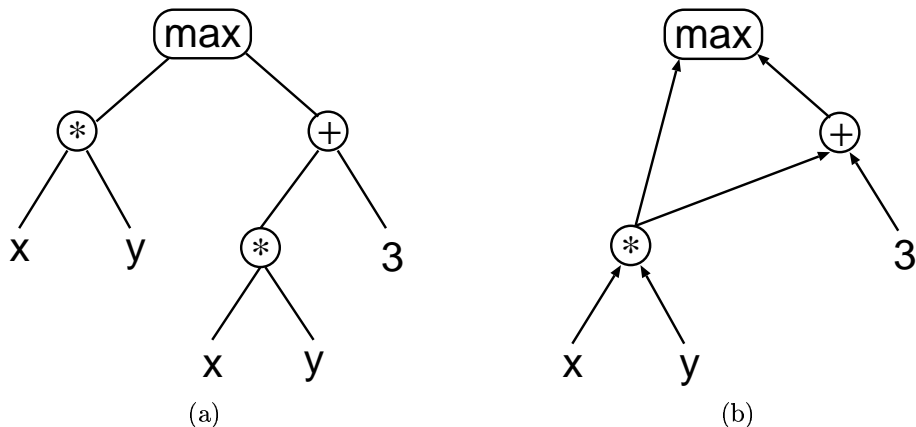


Figure 1: (a) Parse-tree representation of the expression $\max(x * y, 3 + x * y)$ and graph-like representation of the same expression.

crossover and mutation are modified to make sure that the nodes in the main program are never mixed with those of the subroutines and viceversa (this is necessary since ADFs normally use a terminal set including only dummy arguments). For a more detailed introduction to standard GP see (Koza, 1992; Koza, 1994; Banzhaf et al., 1998a; Langdon, 1998).

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. (Koza, 1992; Koza, 1994; Kenneth E. Kinneer, Jr., 1994; Koza et al., 1996c; Koza et al., 1997b; Koza et al., 1998a; Banzhaf et al., 1998b). However, tree-based GP is not very well suited to the evolution of parallel computational structures such as parallel programs, neural networks, logic circuits, etc.

This chapter describes Parallel Distributed Genetic Programming (PDGP), a new form of GP which is suitable for the development of programs with a high degree of parallelism and distributedness.¹ 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 since PDGP 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 chapter 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 number of problems (Section 6). In Section 8 we describe work related to PDGP. Finally, we draw some conclusions and describe promising directions for future work in Section 9.

¹The work described in this chapter has been previously reported in more extensive form in (Poli, 1996d; Poli, 1996a; Poli, 1996c; Poli, 1996b; Poli, 1997a; Poli, 1997b) and has been extended in (Pujol and Poli, 1998a; Pujol and Poli, 1998b; Pujol and Poli, 1998c).

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 (Rumelhart and McClelland, 1986), 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 1(b) 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(a), 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. Also, it can be used to express a much bigger class of programs (trees are special kinds of graphs) which are parallel in nature: we call them *parallel distributed programs*. However, the direct handling of graphs within a genetic algorithm presents some problems.

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

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 2(a), 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 (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 1(b) can be transformed into the layered network in Figure 3(a).

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 3(a) could be described by the list in Figure 3(b).

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. We call them *inactive nodes* or *introns*. The others are called *active nodes*.

²In this chapter (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.

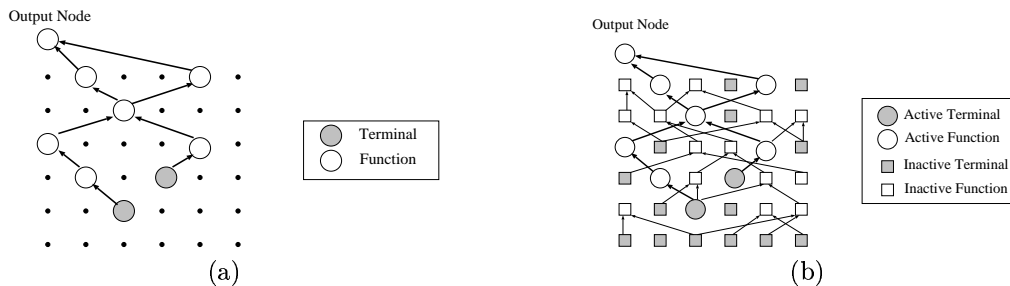


Figure 2: (a) Grid-based representation of graphs representing programs in PDGP and (b) intron-augmented representation of programs in PDGP.

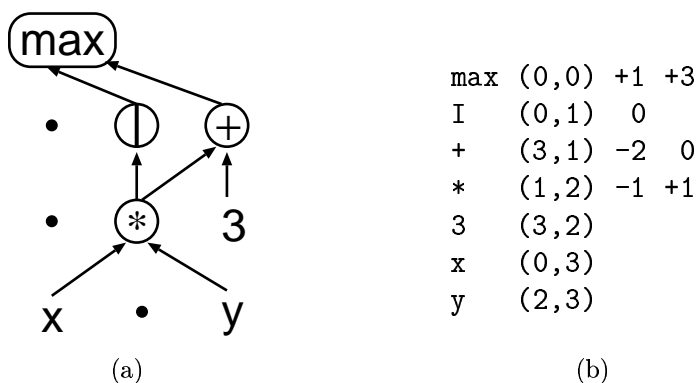


Figure 3: (a) Grid-based representation of the expression $\max(x * y, 3 + x * y)$ and (b) an equivalent description using labels, coordinates and displacements.

For example, the program shown in Figure 2(a) could have an expanded representation like the one in Figure 2(b).

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 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. We describe some of them in the following subsections.

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 (Koza, 1992). In the “grow” method trees are build recursively (starting from the root node) by randomly selecting each node either from the function set

or the terminal set. When a node is drawn from the function set the procedure is recursively applied to its arguments. When the desired maximum depth for the initial trees has been reached, only nodes from the terminal set are selected. The “full” method works in the same way, except that until the maximum depth has been reached nodes are selected only from the function set.

Similarly, in PDGP it is possible to obtain “balanced” or “unbalanced” random graphs depending on whether we allow terminals to be at a pre-fixed maximum distance from the output node(s) only or whether they can occur anywhere in the initial programs.

In PDGP the generation of random programs can proceed in several ways, depending 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 standard GP, during a run trees can grow bigger than the initial maximum depth. Normally the growth is limited by a second depth threshold. In most of the examples described in this chapter 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 x coordinate of the insertion node in the second parent is not compatible with the width of the sub-graph, the sub-graph is wrapped around.

An example of SAAN crossover is shown in Figure 4.³ The idea behind this form of crossover is that connected sub-graphs are functional units whose output is used by other functional units. Therefore, by replacing a sub-graph with another sub-graph, we tend to explore different ways of combining the functional units discovered during evolution.

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 (a number of them have been described and tested in (Poli, 1996c)). In some operators incomplete subgraphs or *sub-sub-graphs* are transferred to form the offspring. By sub-sub-graph we mean a graph fragment 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 behaviour allows the insertion of any parts of a parent programs into any other locations of the other parent. In other operators the selection of the crossover points is not limited to the active nodes. This can be useful in the presence of dynamic fitness functions or epistatic problems. The best operators in our experience are two in which the abovementioned feature are somehow combined:

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

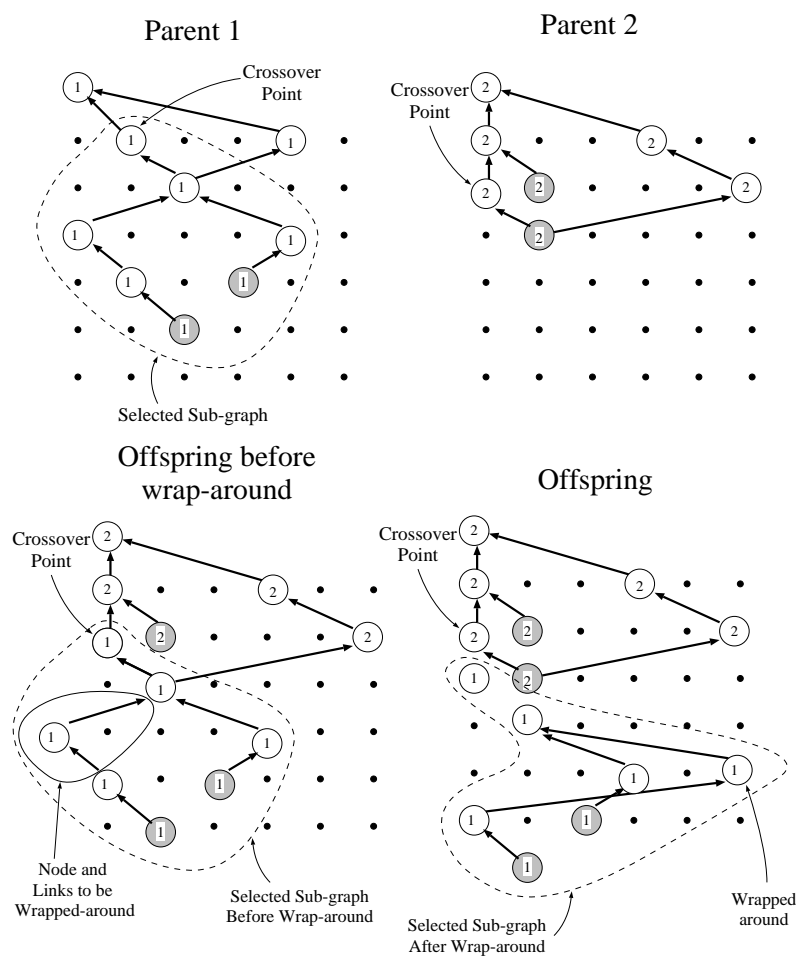


Figure 4: Sub-graph active-active node (SAAN) crossover.

- *Sub-Sub-graph Active-Active Node (SSAAN) Crossover*. This is a form of 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 5 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).
- *Sub-sub-graph Inactive-Active Node (SSIAN) Crossover* is like the SSAAN crossover but selects the first crossover point among both the active and 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 mutation can be naturally extended to PDGP. 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

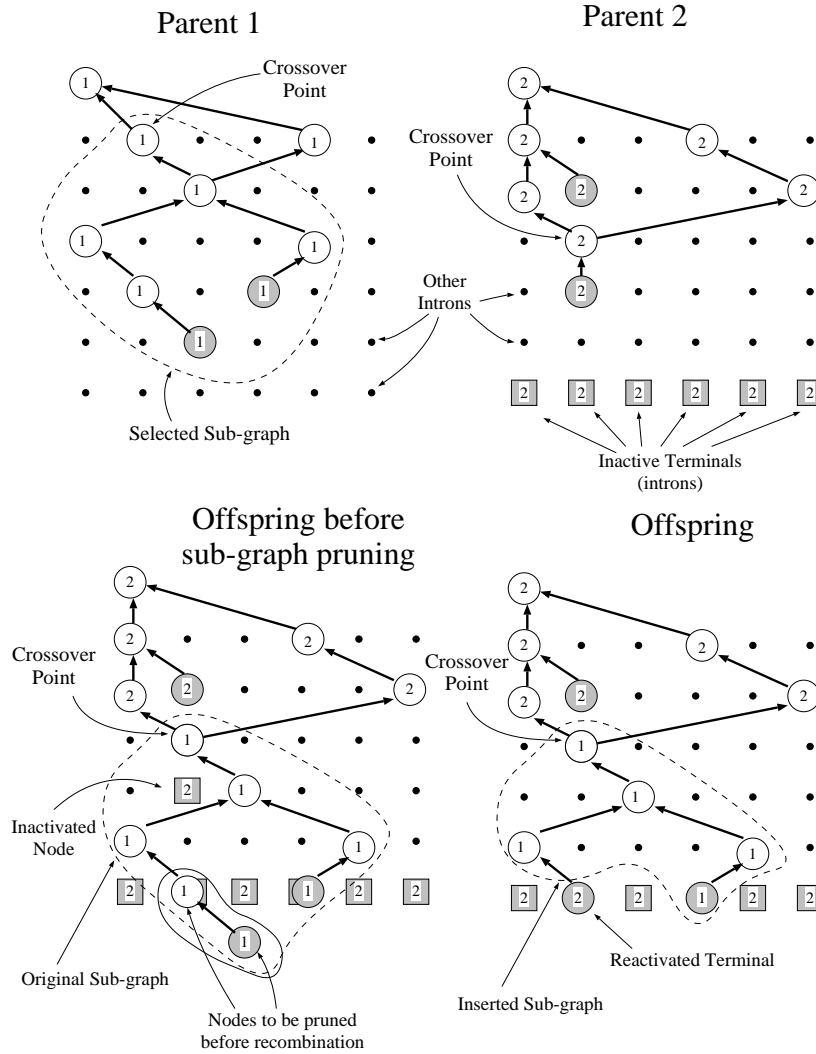


Figure 5: Sub-sub-graph active-active node (SSAAN) crossover with random crossover points.

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).

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. However, this approach present some limitations (Poli, 1996c).

⁴In 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.

```

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

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 6.

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. This is illustrated in Figure 7.

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 6 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

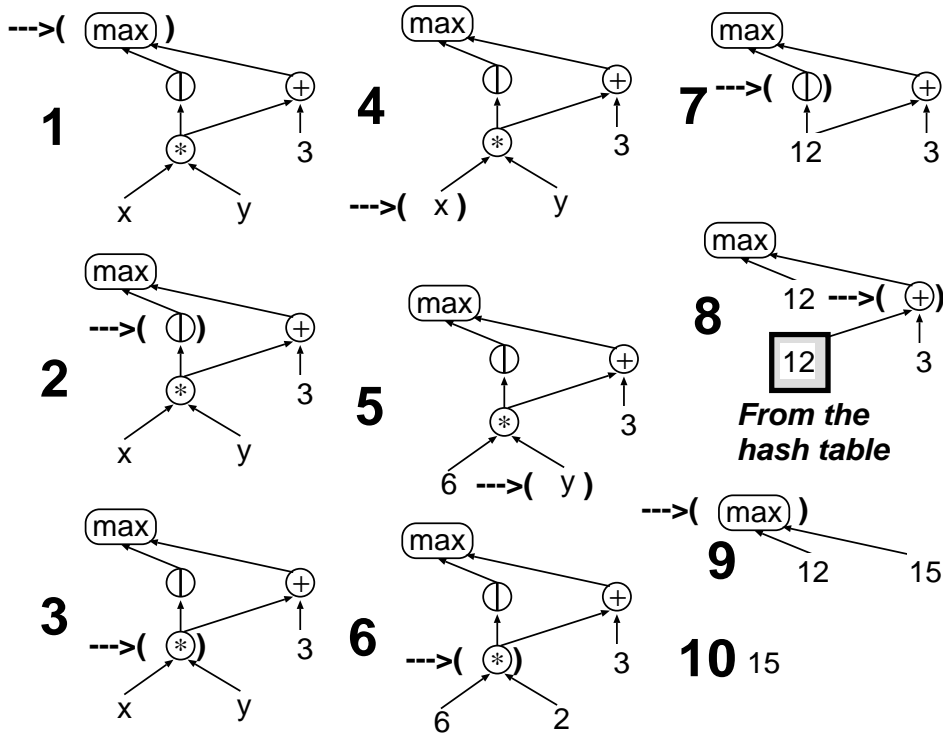


Figure 7: Sample evaluation of the program in Figure 2(a). The horizontal arrow indicates which node is evaluated by the `microeval` function at different time steps (indicated by the number, 1 through 10, in boldface).

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 per 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 letting 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.

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 gracefully, 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 explored. The labels for the links are contained in a *link set* \mathcal{L} which is used in the initialisation phase and also during mutation. If \mathcal{L} contains a random constant generator `random` only, 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 link set can really contain anything: if the labels 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 \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 Examples

In this section we will describe some experiments which show the representational power and efficiency of PDGP. Other examples can be found in (Poli, 1996c).

6.1 Exclusive-Or Problem

In order to show the representational capabilities of PDGP, in this section, we report on some experimental results obtained by applying PDGP to the exclusive-or problem (a sort of benchmark problem in the neural networks community), 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

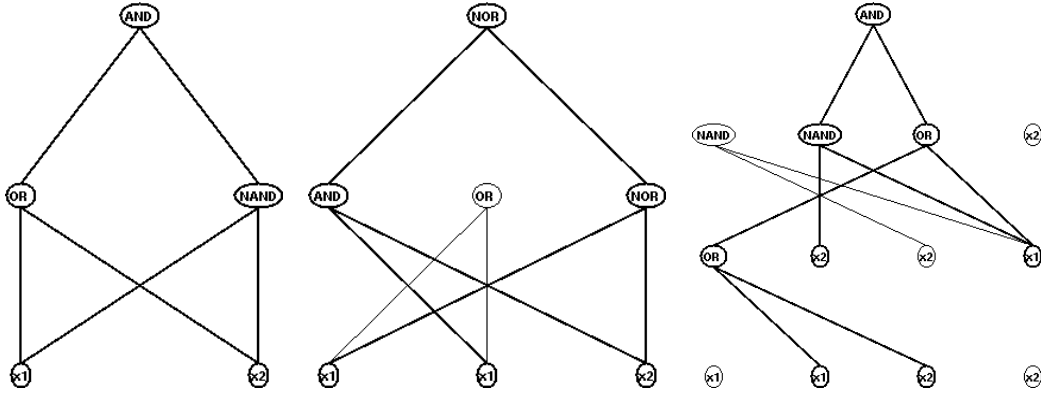


Figure 8: Symbolic networks implementing the exclusive-or function with Boolean processing elements (introns are drawn with thin lines).

probability was 0.7, the global mutation probability was 0.25 while the link mutation probability was 0.25 (we used a generational GA in which mutation was applied to all individuals produced by crossover and reproduction). 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 $\mathcal{F}=\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}, \text{I}\}$ and the terminal set $\mathcal{T}=\{x_1, x_2\}$. 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 8 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 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 $\mathcal{F}=\{+, -, *, \text{PDIV}, \text{I}\}$ (PDIV is the protected division, which returns its first argument if the second is 0) and the terminal set $\mathcal{T}=\{x_1, x_2\}$ 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 greater than 0.5, `false` otherwise. Figure 9 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 10 shows two typical solutions to the XOR problem obtained with PDGP using neuro-algebraic 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”

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

⁶It 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.

⁷For 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.

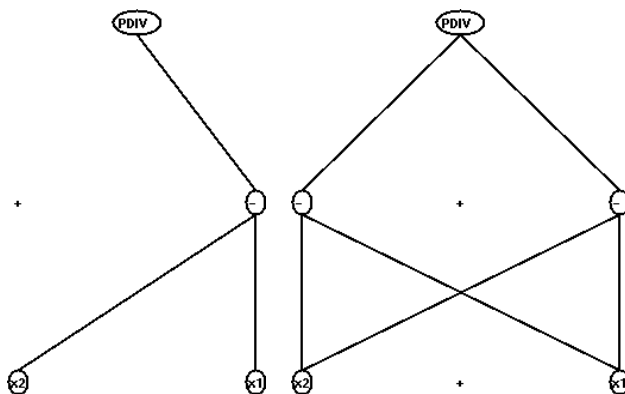


Figure 9: Algebraic network-like realisations of the exclusive-or function.

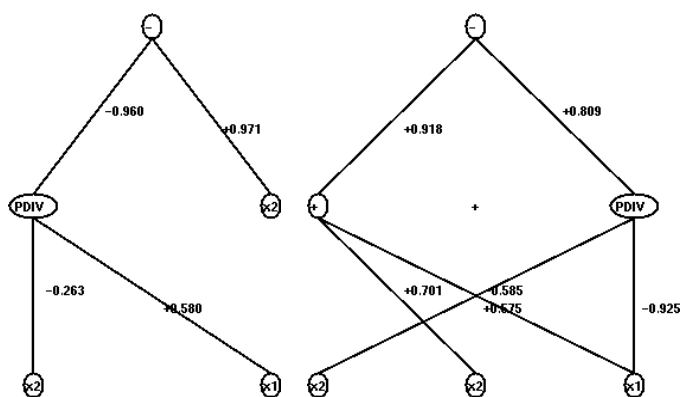


Figure 10: Exclusive-or implementations based on neuro-algebraic parallel distributed programs.

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 11 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 easily 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 12 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. 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. 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

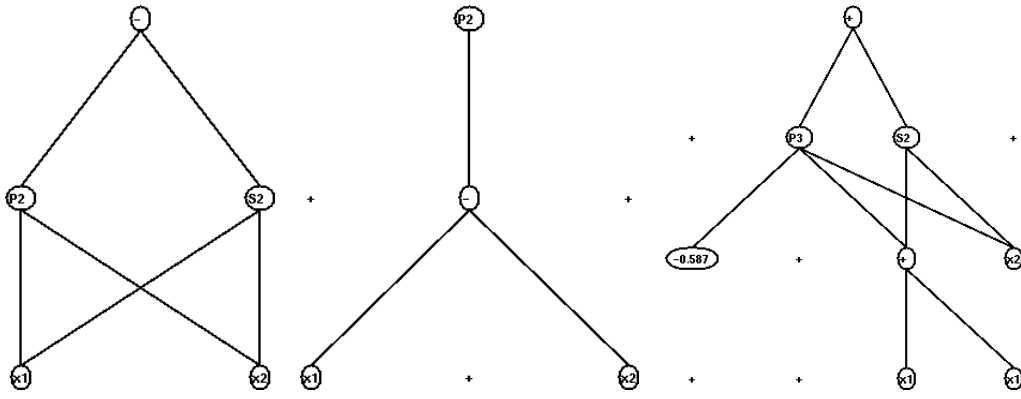


Figure 11: Weight-less neural networks implementing the exclusive-or function.

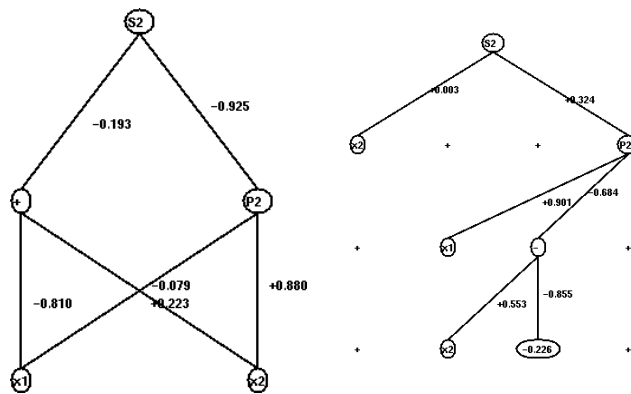


Figure 12: Neural realisations of the exclusive-or function.

<i>Grid size</i>	Logic		Algebraic		Neuro-algebraic		Weight-less neural		Neural	
	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

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 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 can solve this problem (Pujol and Poli, 1998a; Pujol and Poli, 1998b; Pujol and Poli, 1998c).

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 (Rumelhart and McClelland, 1986).

6.2 Lawnmower problem

In this section we describe some 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 a lawn (Koza, 1994, pages 225–273). In the lawnmower problem the lawn is a rectangular array of grass tiles. Figure 13(a) shows the initial configuration for the problem. 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 13(b) shows the effect of a sequences of forward moves. The lawn is considered to be toroidal, as indicated in Figure 13(c) which depicts the situation after two additional forward moves, a turn left and a sequence of forward moves.

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

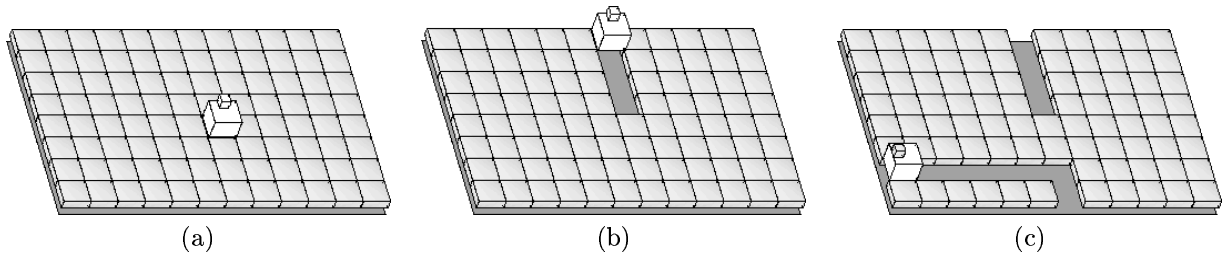


Figure 13: Lawnmower problem

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 order to make the comparison with the results described in (Koza, 1994) possible, we used the terminal set $\mathcal{T}=\{\text{MOW}, \text{LEFT}, \text{random}\}$ and the function set $\mathcal{F}=\{\text{V8A}, \text{FROG}, \text{PROG2}, \text{PROG3}\}$. The function **MOW** performs a step forward, mows the new grass patch and returns the pair of coordinates $(0,0)$. **LEFT** performs a left rotation of the mower and returns $(0,0)$. **random** returns a random pair of coordinates (x,y) with $x,y \in \{0, \dots, 7\}$. **V8A** performs the addition modulo 8 of two pairs of coordinates. **FROG** lifts and moves the mower to a new cell whose displacement in horizontal and vertical direction is indicated by 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. The other parameters of PDGP were: SSIA 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 14 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 sub-graph 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.

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. The results were 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. The linear regression equation for PDGP is $E = 3286 + 26.8 \times L$, L being the size of the lawn (regression coefficient $r=0.949$). A comparison between the linear regression for PDGP and the linear regression for standard GP (Koza, 1994, page 266) shows that PDGP scales up about *2,300 times better*. The average structural complexity of PDGP solutions is approximately 15 nodes, and it does not vary significantly (it really could not) as the complexity of the problem changes. If we compare this 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 ADFs

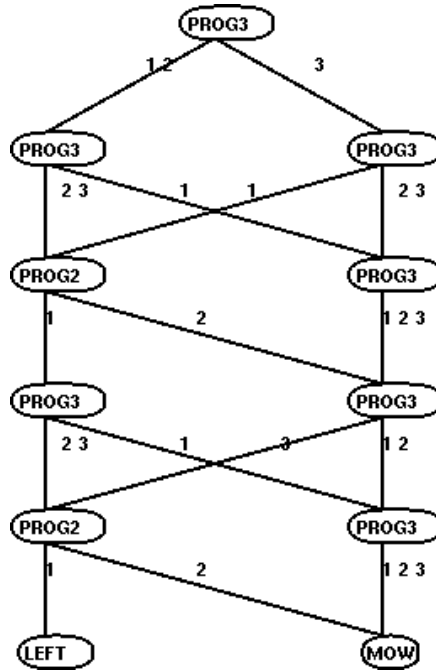


Figure 14: Parallel distributed program solving the lawnmower problem with a lawn of $8 \times 8 = 64$ cells.

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. We believe that 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).

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-like) solutions, in (Poli, 1997a) we studied the behaviour of PDGP as the width of the grid was varied from 2 to 64. The analysis showed 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. This suggests that favouring reuse through small grids may make the search in the space of programs much easier.

More details on the experiments described in this section can be found in (Poli, 1996c; Poli, 1997a).

6.3 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 were approximately the same with ($E=1,440,000$) or without ($E=1,176,000$) ADFs with a population of $P=4,000$ individuals (see (Koza, 1994, 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

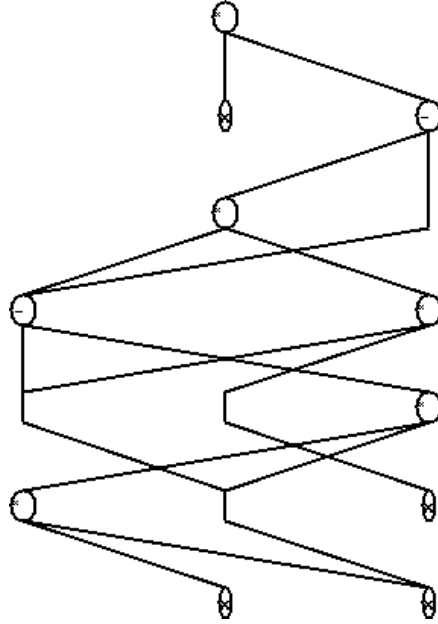


Figure 15: Parallel distributed program implementing the sextic polynomial $x^6 - 2x^4 + x^2$.

$\sum_i |p_i - \text{eval}(\text{prog}, x_i)|$. 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 $\mathcal{T}=[x, \text{random}]$ (where **random** is a constant random generator with values in the range $[-1,1]$) while the functions set was the standard $\mathcal{F}=\{+, -, *, \text{PDIV}, \text{I}\}$.

Figure 15 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.

6.4 Encoder-Decoder Problem

In all the work described in this chapter 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 programs to be evolved have 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 $\mathcal{F}=[\text{AND}, \text{OR}, \text{NOR}, \text{NAND}, \text{I}]$ while the terminal set was $\mathcal{T}=[x_1, x_2, x_3, x_4]$. 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 x_1, x_2, x_3 and x_4 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.⁸ Figure 16 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.

6.5 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, PDGP can actually search very efficiently the space of directed (cyclic and acyclic) graphs.

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 (Brave, 1996)). 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.

⁸Much 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.

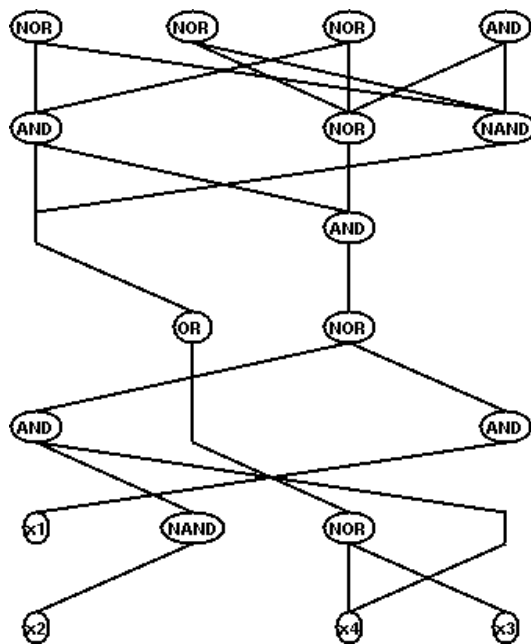


Figure 16: Solution to the 4-bit binary encoder-decoder problem produced by PDGP.

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 $T=[\text{DONE}]$ 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 17 shows one of the solutions found by PDGP (this has been hand drawn for displaying purposes). The execution of the automaton starts from the node in the upper left corner of the figure and proceeds following the outgoing links (which depart from the bottom of the nodes).

7 Natural Language Recognition Problem

In this section we will show how the representations, operators and interpreters used in PDGP can be tailored to solve a much harder problem: the induction of a recogniser for natural language from positive and negative examples. A recogniser for natural language is a program that given a sentence (say in English) returns `true` if the sentence is grammatical, `false` otherwise. The problem of inducing recognisers (and parsers) from actual sentences of a language, also known as language acquisition, is a very hard machine learning problem (see (Shapiro, 1992, pages 443–451) for a survey on the topic).

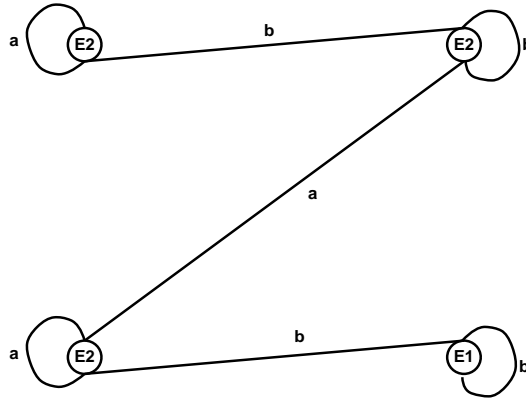


Figure 17: Finite state automaton capable of recognising the language $L=a*b*a*b^*$.

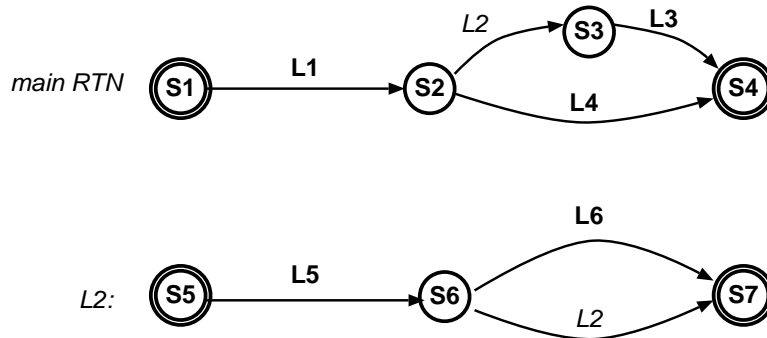


Figure 18: Sample recursive transition network.

Deterministic finite state automata are well suited to build recognisers for simple regular languages. However, they are not particularly suited to represent the natural recursivity of natural-language grammars. Indeed, the language used in the previous section has nothing to do with the complexity of natural language. For this reason we have decided to use PDGP to evolve recognisers based on Recursive Transition Networks (RTNs). RTNs are extensions of finite state automata, in which the label associated to a link can represent either symbols of the language like in standard finite state automata or other RTNs possibly including the RTN containing the link.

To understand how RTNs work let us consider the example in Figure 18. In the figure the double circles represent initial and end states. The labels L1, L3, L4, L5, L6 are symbols of the language while L2 represents another RTN. In the presence of the input sequence [L1 L5 L6 L3] the following happens. Starting from S1, L1 is read and S2 is reached. Since S2 has no outgoing link labelled L5, the sub-RTN L2 is invoked to check whether the subsequence starting from L5 can be parsed. So, control passes to state S5. Since S5 has an outgoing link labelled L5, S6 is reached, then L6 is read and S7 is reached. Since S7 is an end state, control is returned to S3, which causes L3 to be read and the end state S4 to be reached. If instead of a single L5, the sequence contained several L5 in a row, the sub-RTN L2 would invoke itself several times before returning control to the main RTN.

While standard PDGP can evolve networks of any topology, in order to evolve RTNs for natural language processing we decided to use a technique unique to PDGP, called Automatically Defined Links (ADLs). ADLs can be seen as the dual of ADFs. ADLs work in a very similar way to ADFs. Basically, they are ADFs which, instead of being part of the function set of the main program, are part of the link set, and instead of being tree-like are graph-like subprograms. For example, if the label of a link is the symbol ADL0, the interpreter will “invoke” the parallel distributed program (in this case an RTN) corresponding to the first ADL. It should be noted that PDGP allows the use of both ADFs and ADLs.

For our experiments we generated 67 grammatical sentences and 181 ungrammatical ones, all with different syntactic structures, using the following grammar:

```

S := NP VP
NP := SNP | SNP PP
SNP := DET NOUN | DET ADJ NOUN | DET ADJ ADJ NOUN
PP := PREP SNP
VP := TVERB NP | IVERB

```

where S stands for sentence, NP for noun-phrase, VP for verb-phrase, SNP for simple NP, PP for prepositional phrase, DET for determiner, ADJ for adjective, TVERB for transitive verb, and IVERB for intransitive verb. The words in the examples were assigned a lexical category (like “noun”, “verb”, “adjective”, etc.) so that the actual examples were sequences of lexical categories like DET ADJ NOUN TVERB DET ADJ NOUN PREP DET ADJ NOUN rather than “the little kitten frightened the big mouse with a sudden jump”.

In the experiments we used a population with $P=4,000$ individuals, a maximum number of generations $G=200$, two ADLs, a regular 4×7 grid for the main RTN and two 4×5 grids for the ADLs. The recombination operator was a form of SSAAN crossover. The probability of crossover was 0.7, the probability of mutation was 0.1. The node in the top left corner of the main grid was considered to be the start node.

The following link set was used for both the main RTN and the two ADLs: [NOUN, TVERB, IVERB, PREP, DET, ADJ, ADL0, ADL1]. It includes the different kinds of lexical categories and the names of the two ADLs. The function set included the macros [N1 N2] which represent non-stop states in the RTN, 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 the node connected to the link labelled with the matching symbol. If the label is an ADL, then control is passed to the corresponding RTN. 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 terminal set for the main RTN included only nodes of type DONE which behave like N1 nodes without output link except that if the end of the sentence is reached they return `true`. The terminal set for the ADLs included only nodes of type RETURN, which simply return `true`.

The fitness of an RTN was $f = 67 - n_s - 10^{-5} \times n_w$, where n_s is the number of sentences incorrectly classified and n_w is the number of words not read by the RTN in the sentences incorrectly classified. The value 67 corresponds to the number of grammatical sentences in the training set. The term $10^{-5} \times n_w$ has the function of smoothing the fitness landscape created by the term n_s . The coefficient 10^{-5} has been chosen so that $10^{-5} \times n_w < 1$. This ensures that evolution will always favour solutions with smaller n_s whatever the value of n_w .

Figure 19 shows the 100%-correct solution found by PDGP after two unsuccessful runs. The performance of this RTN has been tested with 156 different grammatical sentences (all with different structure), which the system classified correctly, and 12066 different random ungrammatical sentence-types, of which the system correctly classified 12008.⁹ This would suggest that the system has a sensitivity of 100% and a specificity of 99.5%. However, tests with 156 different slightly ungrammatical sentence-types (obtained by modifying one lexical category in a valid sentence) have revealed a lower specificity, 68.6%, which seems still quite good for a system 100% sensitive.

The recogniser works in a very complicated and unusual way, by using ADL0 to move from node to node without changing the symbol to be parsed, and ADL1 mainly to parse prepositional phrases or fragments of them. For example, the output produced by the Pop-11 tracer when recognising the sentence “the kitten on the small mat frightened a mouse” is shown in Figure 20. The parse tree for this sentence is shown in Figure 21 together with the parts of the RTN responsible for the recognition of each word.

⁹The disparity between the number of grammatical and ungrammatical sentences is due to the impossibility of generating more grammatical sentence types with the grammar described above.

```

> DET DET          !!!> ADJ ADJ          !< PREP <false>          !< ADLO <true>
< DET <true>      !!!< ADJ <true>      !> ADLO END             !> TVERB END
> ADJ NOUN        !!!> ADL1 NOUN       !< ADLO <true>          !< TVERB <false>
< ADJ <false>     !!!> NOUN NOUN       !> ADJ END              !> ADLO END
> NOUN NOUN       !!!< NOUN <true>     !< ADJ <false>         !< ADLO <true>
< NOUN <true>     !!!> PREP TVERB       !> ADLO END             < ADL1 <true>
> DET PREP        !!!< PREP <false>    !< ADLO <true>         > DET END
< DET <false>     !!!> ADLO TVERB      !> TVERB END           < DET <false>
> ADL1 PREP       !!!< ADLO <true>     !< TVERB <false>     > ADL1 END
!> NOUN PREP      !!!> ADJ TVERB        !> ADLO END            !> NOUN END
!< NOUN <false>   !!!< ADJ <false>         !< ADLO <true>       !< NOUN <false>
!> ADLO PREP      !!!> ADLO TVERB      < ADL1 <true>        !> ADLO END
!< ADLO <true>    !!!< ADLO <true>     > DET END            !< ADLO <true>
!> PREP PREP      !!!> TVERB TVERB          < DET <false>       !> PREP END
!< PREP <true>   !!!< TVERB <true>     > ADL1 END           !< PREP <false>
!> DET DET        !!!< ADL1 <true>     !> NOUN END           !> ADLO END
!< DET <true>    !!!< ADL1 <true>     !< NOUN <false>     !< ADLO <true>
!> ADL1 ADJ       < ADL1 <true>         !> ADLO END          !> ADJ END
!!!> NOUN ADJ    > DET DET             !< ADLO <true>       !< ADJ <false>
!!!< NOUN <false> < DET <true>         !> PREP END          !> ADLO END
!!!> ADLO ADJ   > DET NOUN          !< PREP <false>     !< ADLO <true>
!!!< ADLO <true> < DET <false>    !> ADLO END          !> TVERB END
!!!> PREP ADJ   > ADL1 NOUN        !< ADLO <true>       !< TVERB <false>
!!!< PREP <false> !> NOUN NOUN          !> ADJ END            !> ADLO END
!!!> ADLO ADJ   !< NOUN <true>     !< ADJ <false>       !< ADLO <true>
!!!< ADLO <true> !> PREP END          !> ADLO END          < ADL1 <true>

```

Figure 20: Functions invoked by the parsing of the sentence "the kitten on the small mat frightened a mouse".

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 (Gruau et al., 1996)).

In conjunction with an interpreter implementing a parallel virtual machine, GP can be used to translate sequential programs into parallel ones (Walsh and Ryan, 1996) or to develop some kinds of parallel programs.¹⁰ Bennett, in (Bennett III, 1996), 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 (Andre et al., 1996) 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 (Teller and Veloso, 1995b; Teller and Veloso, 1995a; Teller, 1996; Teller and Veloso, 1996) used a combination of GP and linear discrimination to obtain classification programs for signals and images.

Other work which has some relation with ours is based on the idea, firstly proposed by Handley (Handley, 1994) and later improved by Ehrenburg (Ehrenburg, 1996), 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

¹⁰The 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 (Andre and Koza, 1996; Dracopoulos and Kent, 1996; Oussaidene et al., 1996; Juille and Pollack, 1996; Stoffel and Spector, 1996). 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.

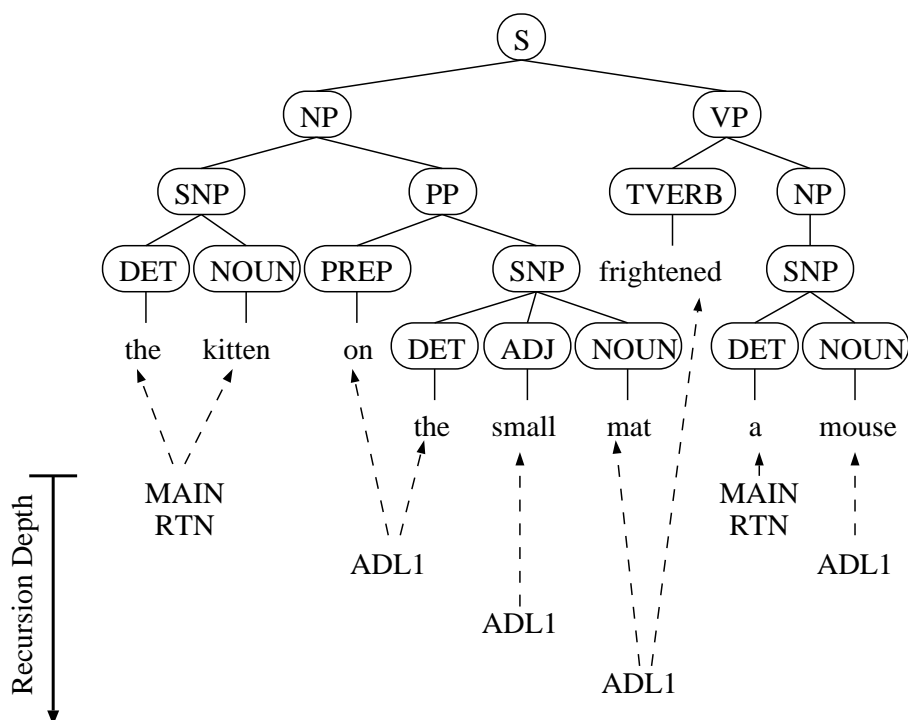


Figure 21: Parse tree for the sentence "the kitten on the small mat frightened a mouse" and parts of the RTN responsible for its recognition (calls to ADL0 are not shown).

the evolution of tree-like programs as standard GP, i.e. it is not about the evolution of parallel programs.

9 Conclusions

In this chapter 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 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. In the future it would be interesting to study 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 chapter 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.

PDGP uses a grid-based representation of programs which allowed us to develop efficient genetic operators. 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: a new technique unique to PDGP.

The experimental results with the evolution of recursive transition nets for natural language recognition obtained using this technique are very promising and show how evolutionary computation with the use of little prior knowledge can now tackle higher and higher level problems usually considered to require special AI techniques.

It should be noted that ADLs are just one of the new representational possibilities opened by PDGP. 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.). These graphs need not be interpreted as programs. They can be interpreted as engineering designs, semantic nets, neural networks, etc. Also, cellular encoding can naturally be extended to PDGP, thus creating a very large set of 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 Bill Langdon and the other members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group for useful discussions and comments.

References

- Andre, D., Bennett III, F. H., and Koza, J. R. (1996). Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 3–11, Stanford University, CA, USA. MIT Press.
- Andre, D. and Koza, J. R. (1996). Parallel genetic programming: A scalable implementation using the transputer network architecture. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA.
- Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998a). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag.
- Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors (1998b). *Genetic Programming*, volume 1391 of *LNCS*, Paris. Springer-Verlag.
- Bennett III, F. H. (1996). Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 30–38, Stanford University, CA, USA. MIT Press.
- Brave, S. (1996). Evolving deterministic finite automata using cellular encoding. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 39–44, Stanford University, CA, USA. MIT Press.
- Dracopoulos, D. C. and Kent, S. (1996). Speeding up genetic programming: A parallel BSP implementation. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 421, Stanford University, CA, USA. MIT Press.

- Ehrenburg, H. (1996). Improved direct acyclic graph handling and the combine operator in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 285–291, Stanford University, CA, USA. MIT Press.
- Gruau, F. (1994). Genetic micro programming of neural networks. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 24, pages 495–518. MIT Press.
- Gruau, F. and Whitley, D. (1993). Adding learning to the cellular development process: a comparative study. *Evolutionary Computation*, 1(3):213–233.
- Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA. MIT Press.
- Handley, S. (1994). 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. IEEE Press.
- Juille, H. and Pollack, J. B. (1996). Massively parallel genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, Cambridge, MA, USA.
- Kenneth E. Kinnear, Jr., editor (1994). *Advances in Genetic Programming*. MIT Press.
- Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.
- Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts.
- Koza, J. R., Andre, D., Bennett III, F. H., and Keane, M. A. (1996a). Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 132–149, Stanford University, CA, USA. MIT Press.
- Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors (1998a). *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, WI, USA. Morgan Kaufmann.
- Koza, J. R., Bennett III, F. H., Andre, D., and Keane, M. A. (1996b). Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Stanford University, CA, USA. MIT Press.
- Koza, J. R., Bennett III, F. H., Lohn, J., Dunlap, F., Keane, M. A., and Andre, D. (1997a). Use of architecture-altering operations to dynamically adapt a three-way analog source identification circuit to accommodate a new source. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 213–221, Stanford University, CA, USA. Morgan Kaufmann.
- Koza, J. R., David Andre, Bennett III, F. H., and Keane, M. (1998b). *Genetic Programming 3*. MIT Press, Cambridge, MA, USA. Forthcoming.

- Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors (1997b). *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA. Morgan Kaufmann.
- Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors (1996c). *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA. MIT Press.
- Langdon, W. B. (1998). *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!* Kluwer, Boston.
- Like, S. and Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 117–124.
- Oussaidene, M., Chopard, B., Pictet, O. V., and Tomassini, M. (1996). Parallel genetic programming: An application to trading models evolution. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 357–380, Stanford University, CA, USA. MIT Press.
- Poli, R. (1996a). Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Technical Report CSRP-96-14, University of Birmingham, School of Computer Science.
- Poli, R. (1996b). Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming. Technical Report CSRP-96-19, School of Computer Science, University of Birmingham, B15 2TT, UK. Presented at AISB-97 workshop on Evolutionary Computation.
- Poli, R. (1996c). Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK.
- Poli, R. (1996d). Some steps towards a form of parallel distributed genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*, <http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/>. Nagoya University, Japan.
- Poli, R. (1997a). Evolution of graph-like programs with parallel distributed genetic programming. In Back, T., editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA. Morgan Kaufmann.
- Poli, R. (1997b). Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming. In Corne, D. and Shapiro, J. L., editors, *Evolutionary Computing*, volume 1305 of *Lecture Notes in Computer Science*, pages 163–177, Manchester, UK. Springer-Verlag.
- Pujol, J. C. F. and Poli, R. (1998a). Efficient evolution of asymmetric recurrent neural networks using a PDGP-inspired two-dimensional representation. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 130–141, Paris. Springer-Verlag.
- Pujol, J. C. F. and Poli, R. (1998b). Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. *Applied Intelligence*, 8:73–84.
- Pujol, J. C. F. and Poli, R. (1998c). Evolving neural networks using a dual representation with a combined crossover operator. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 416–421, Anchorage, Alaska, USA. IEEE Press.
- Rumelhart, D. and McClelland, J., editors (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1-2*. MIT Press, Cambridge, MA.

- Shapiro, S. C. (1992). *Encyclopedia of Artificial Intelligence*. Wiley, New York, second edition.
- Stoffel, K. and Spector, L. (1996). High-performance, parallel, stack-based genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA. MIT Press.
- Teller, A. (1996). Evolving programmers: The co-evolution of intelligent recombination operators. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, Cambridge, MA, USA.
- Teller, A. and Veloso, M. (1995a). A controlled experiment: Evolution for learning difficult image classification. In *Seventh Portuguese Conference On Artificial Intelligence*, volume 990 of *Lecture Notes in Computer Science*, pages 165–176, Funchal, Madeira Island, Portugal. Springer-Verlag.
- Teller, A. and Veloso, M. (1995b). 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.
- Teller, A. and Veloso, M. (1996). Neural programming and an internal reinforcement policy. In Koza, J. R., editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 186–192, Stanford University, CA, USA. Stanford Bookstore.
- Walsh, P. and Ryan, C. (1996). Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 406–409, Stanford University, CA, USA. MIT Press.