

Parallel Distributed Genetic Programming Applied to the Evolution of Natural Language Recognisers

Riccardo Poli

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

Abstract. This paper describes an application of Parallel Distributed Genetic Programming (PDGP) to the problem of inducing recognisers for natural language from positive and negative examples. PDGP is 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. PDGP allows the exploration of a large space of possible programs including standard tree-like programs, logic networks, neural networks, finite state automata, Recursive Transition Networks (RTNs), etc. The paper describes the representations, the operators and the interpreters used in PDGP, and describes how these can be tailored to evolve RTN-based recognisers.

1 Introduction

In Genetic Programming [11, 12] programs are expressed as parse trees to be executed sequentially in the standard depth-first evaluation order. This form of GP has been applied successfully to a large number of difficult problems like automated design, pattern recognition, robot control, symbolic regression, music generation, image compression, image analysis, etc. [11, 12, 9, 10, 1, 18].

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 [6, 7] or electronic circuits [15, 13], which can be thought 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 [24] or to develop some kinds of parallel programs [3, 2, 23]. However, all these methods either are limited to very special kinds of parallelism or use indirect representations which require computationally expensive genotype-to-phenotype mappings.

This paper describes an application of Parallel Distributed Genetic Programming, a new form of GP which is suitable for the development of programs with

a high degree of parallelism and distributedness. Programs are represented in PDGP as graphs with nodes representing functions and terminals, and links representing the flow of control and results. In the simplest form of PDGP, links are directed and unlabelled, in which case PDGP can be considered a generalisation of standard GP (trees are special kinds of graphs). However, PDGP can use more complex (direct) representations, which allow the development of symbolic, neuro-symbolic and neural networks, recursive 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.

In previous work [19, 20, 17] we have studied some of the representation capabilities of PDGP and compared PDGP with standard GP on simple problems. In this paper 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 [21, pages 443–451] for a survey on the topic).

A limited amount of work has been done on evolutionary algorithms applied to the problems of natural language processing. A very recent review of these is presented in [5]. To the author's knowledge only two applications of GP in this area have been published to date [22, 16], none dealing with the problem of natural language recognition. Somehow more closely related to this topic is the work described in [4, 17] on inducing, from positive and negative example, deterministic finite state automata capable of recognising simple regular languages.

Deterministic finite state automata are well suited to build recognisers for this kind of languages. However, they are not particularly suited to represent the natural recursivity of natural-language grammars. Indeed, the work in [4, 17] considered very simple non-recursive languages, like $L = a^*b^*a^*b^*$ (consisting of all sentences with 0 or more a's followed by 0 or more b's, etc), which have 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).

The paper is organised as follows. Firstly, in Section 2, we describe the representation, the genetic operators and the interpreter used in PDGP. Then, we illustrate how PDGP can be used to solve the problem of inducing recognisers

for natural language from positive and negative examples (Section 3). Finally, we draw some conclusions in Section 4.

2 PDGP

2.1 Representation

Standard “tree-like” programs can often be represented as graphs with labelled nodes (the functions and terminals used in the program) and oriented links which prescribe which arguments to use for each node when it is next evaluated. Figure 1(a) shows an example of a program represented as a graph. The program implements the function $\max(x * y, 3 + x * y)$. Its execution should be imagined as a “wave of computations” starting from the terminals and propagating upwards like in a multi-layer perceptron.

This form of representation is in general more compact and efficient than a tree-like one and can be used to express a much bigger class of programs which are parallel in nature: we call them *parallel distributed programs*.

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

This representation for parallel distributed programs is illustrated in Figure 1(b), where we assumed that the program has a single output at coordinates (0,0) (the y axis is pointing downwards) and that the grid includes $6 \times 6 + 1$ cells. We have also assumed that connections between nodes are upwards and are allowed only between nodes belonging to adjacent rows, like the connections in a standard multi-layer perceptron. This is not a limitation as, by adding the identity function to the function set, any parallel distributed program can be described with this representation. For example, the program in Figure 1(a) can be transformed into the layered network in Figure 1(c).

From the implementation point of view in PDGP programs are represented as arrays with the same topology as the grid, in which each cell contains a function label and the horizontal displacement of the nodes in the previous layer used as arguments for the function. It should be noted that, in order to exploit the possibilities offered by the use of “unexpressed” parts of code, functions or terminals are associated to *every* node in the grid, i.e. also to the nodes that are not directly or indirectly connected to the output. We call them *inactive nodes* or *introns*; the others are called *active nodes*.

This basic representation has been extended in various directions.¹ The first extension was to introduce vertical displacements to allow feed-forward connections between non-adjacent layers. With this extension any directed acyclic graph can be naturally represented within the grid of nodes, without requiring

¹ All these extensions require changes to the operators and interpreter used in PDGP. For the sake of brevity we do not discuss these changes in detail here.

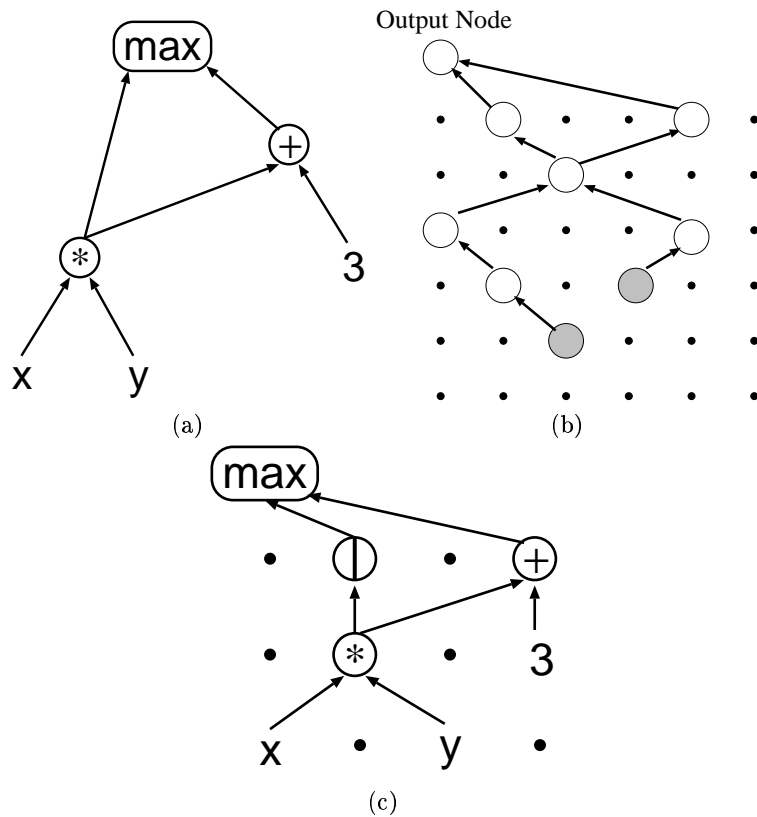


Fig. 1. (a) Graph-based representation of the expression $\max(x * y, 3 + x * y)$; (b) Grid-based representation of graphs representing programs in PDGP (hollow circles represent functions, grey circles represent terminals); (c) Grid-based representation of the expression in (a).

the presence of pass-through nodes. The second extension has been to allow non-positive vertical displacements to represent graphs with backward connections (e.g. with cycles). The third extension has been to add labels to links. The link set can contain constants, variables, functions and macros. If the labels are numbers (e.g. ephemeral random numbers) this extension allows the direct development of neural networks. If the symbols of a language are used as labels for the links, it is possible to evolve finite state automata, transition nets, semantic nets, etc. If the links are functions or macros, then very complex behaviours can be realised, like the ones required to implement RTNs (see Section 3).

2.2 Genetic Operators

Several kinds of crossover, mutation and initialisation strategies can be defined for the basic representation used in PDGP and for its extensions. In the following we will only describe two forms of crossover (more details can be found in [17]).

The basic crossover operator of PDGP, which we call *Sub-graph Active-Active Node (SAAN) crossover*, is a generalisation to graphs of the crossover used in GP to recombine trees. SAAN crossover works as follows: 1) a random active node is selected in each parent (crossover point); 2) a sub-graph including all the active nodes which are used to compute the output value of the crossover point in the first parent is extracted; 3) the sub-graph is inserted in the second parent to generate the offspring (if the x coordinate of the insertion node in the second parent is not compatible with the width of the sub-graph, the sub-graph is wrapped around).²

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. An example of SAAN crossover is shown in Figure 2.

Obviously, for SAAN crossover to work properly some care has to be taken to ensure that the depth of the sub-graph being inserted in the second parent is compatible with the maximum allowed depth, i.e. the number of rows in the grid. A simple way to do this is, 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. The one which has given the best results until now is *Sub-Sub-graph Active-Active Node (SSAAN) Crossover*. In SSAAN crossover one crossover point is selected at random among the active nodes of each parent. Then, a complete sub-graph is extracted from the first parent (like in SAAN crossover) disregarding the problems possibly caused by its depth. If the depth of the sub-graph is too big for it to be copied into the second parent, the lowest nodes of the sub-graph are pruned to make it fit.³ Figure 3 illustrates this process. In order for this type of crossover to work properly introns are essential, in particular the inactive terminals in the last row of the second parent (they have been explicitly shown in the figure to clarify the interaction between them and the functions and terminals in the sub-graph).

² The term “active-active” in the name of SAAN crossover derives from the fact that the crossover points selected in step 1 are active nodes for both parents. The term “sub-graph” refers to the fact that the nodes transferred from one parent to the other form a sub-graph.

³ The term “sub-sub-graph” in the name of SSAAN crossover refers to the fact that, unlike in SAAN crossover, due to this pruning process the sub-graph actually transferred from one parent to the other may be a sub-graph of the sub-graph extracted from the first parent.

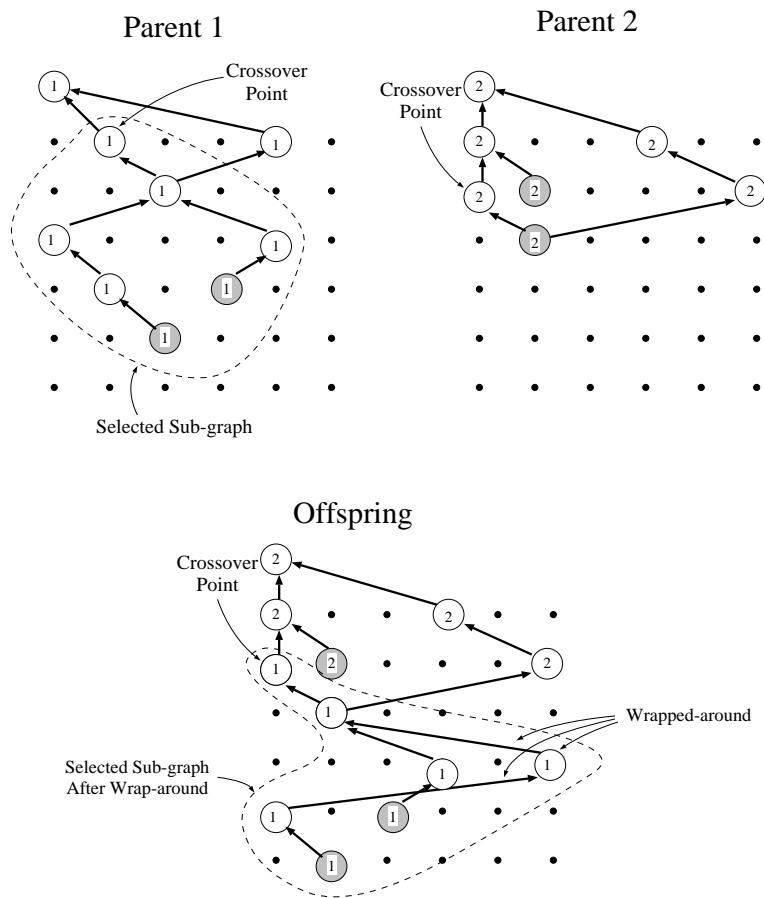


Fig. 2. Sub-graph active-active node (SAAN) crossover.

2.3 Interpreters

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

The interpreter, the procedure `eval(program)`, can be thought of as having the structure outlined in Figure 4. `eval` starts the evaluation of a program from the output nodes and calls recursively the procedure `microeval(node)` to perform the evaluation of each of them. `microeval` checks to see if the value of a node is already known. If this is the case it returns the value stored in a hash table, otherwise it executes the corresponding terminal or function (calling itself

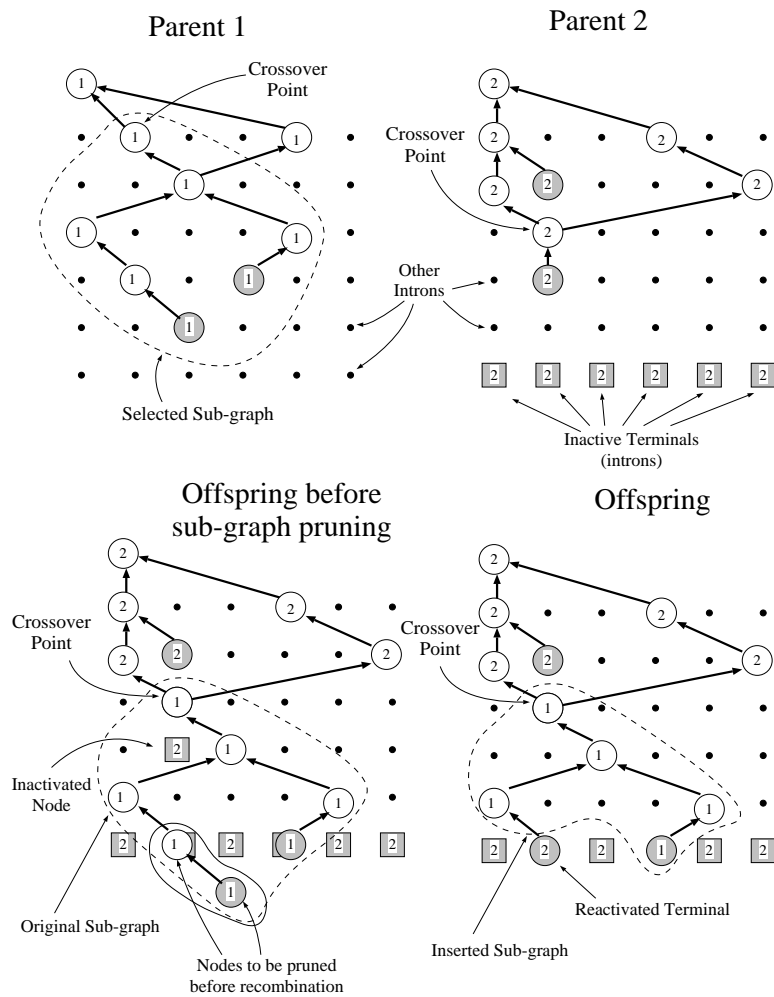


Fig. 3. Sub-sub-graph active-active node (SSAAN) crossover.

recursively to get the values for the arguments, if any).

This approach has the advantage of allowing the use of nodes with side effects⁴ and of offering a total freedom in the connection topology of PDGP programs. Although the interpreter described above performs a sequential evaluation of programs, this process is inherently parallel. In fact, if we imagine each node to be a processor (with some memory to store its state and current value) and each link to be a communication channel, the evaluation of a program is equivalent to the parallel downward propagation of control messages requesting

⁴ For brevity we do not describe here the changes needed to evaluate correctly nodes with side effects. More information on this is given in [17].

```

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

```

Fig. 4. Pseudo-code for the interpreter of PDGP (in the absence of nodes with side effects).

a value followed by an upward propagation of return values. In this sense our interpreter can be seen as a parallel virtual machine capable of running data-driven dataflow programs [8] and PDGP as a method to evolve them.

3 Evolution of Natural Language Recognisers

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 Automatically Defined Functions (ADFs), which are sometimes used to evolve higher-level problem-specific primitives

(parametrised subroutines) which improve the problem solving capabilities of GP [12]. ADFs are usually implemented using a special root node with multiple branches. One of the branches is interpreted as the value-returning (main) program, the others are interpreted as function definitions (the ADFs). The ADFs are part of the function set of the main program, which can therefore use them as individual-specific parametrised building blocks. Sometimes ADFs are part of the function set of other ADFs, too.

ADLs work in a very similar way. 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.

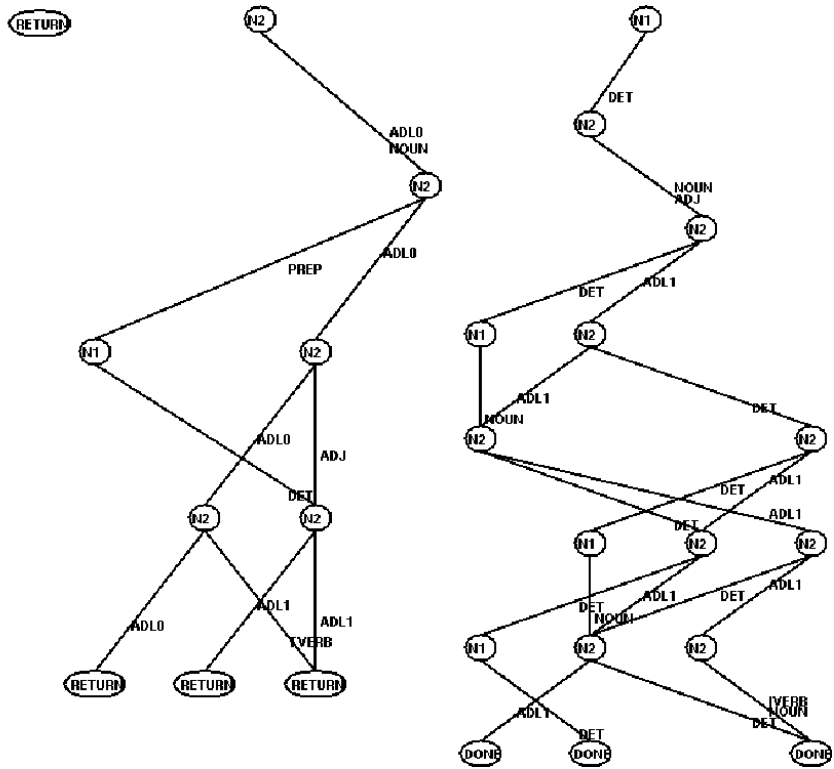


Fig. 5. Recursive transition network for recognising grammatical English sentences evolved by PDGP. The main RTN is shown on the right, ADL0 on the left and ADL1 in the centre.

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 .

```

> 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>

```

Fig. 6. Functions invoked by the parsing of the sentence "the kitten on the small mat frightened a mouse".

Figure 5 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 12008.⁵ 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.

⁵ The 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.

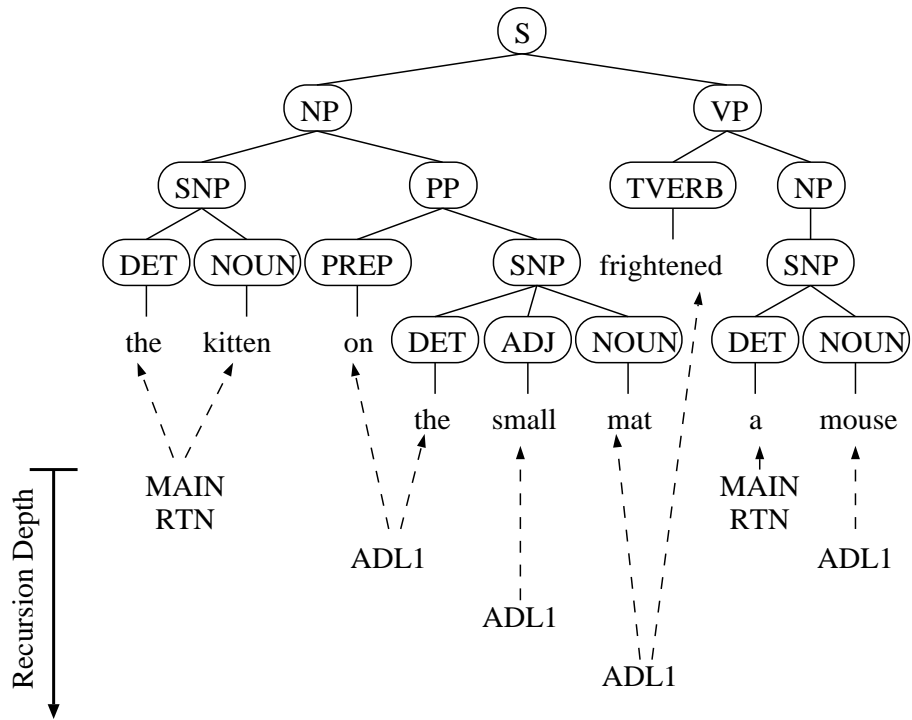


Fig. 7. Parse tree for the sentence "the kitten on the small mat frightened a mouse" and parts of the RTN responsible for its recognition (calls to ADL0 are not shown).

The 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 propositional 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 6. The parse tree for this sentence is shown in Figure 7 together with the parts of the RTN responsible for the recognition of each word.

It should be noted that for very difficult problems which have never been solved before, at least with evolutionary techniques, such as the one considered in this paper or the evolution of electronic circuits (see for example [14]), it is usually impossible to perform more than a few runs (in this case 3). This is normally considered more than acceptable because, in these cases, the research focus is not assessing the performance of GP through the usual statistics used in the GP literature (like the computational effort, the percentage of successful runs, etc.), but rather showing how GP can reach new horizons and studying the characteristics of the solutions evolved.

4 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, and applied it to the problem of inducing recognisers for natural language.

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.

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. The anonymous reviewers are also thanked for their invaluable help in improving this paper.

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*, pages 3–11, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
3. Forrest H. Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 30–38, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

4. 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*, pages 39–44, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
5. Ted E. Dunning and Mark W. Davis. Evolutionary algorithms for natural language processing. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 16–23, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.
6. F. Gruau and D. Whitley. Adding learning to the cellular development process: a comparative study. *Evolutionary Computation*, 1(3):213–233, 1993.
7. Frederic Gruau. Genetic micro programming of neural networks. In Kenneth E. Kinneer, Jr., editor, *Advances in Genetic Programming*, chapter 24, pages 495–518. MIT Press, 1994.
8. R. Jagannathan. Dataflow models. In E. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.
9. K. E. Kinneer, Jr., editor. *Advances in Genetic Programming*. MIT Press, 1994.
10. J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Proceedings of the First International Conference on Genetic Programming*, Stanford University, July 1996. MIT Press.
11. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
12. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1994.
13. John R. Koza, David Andre, Forrest H. Bennett III, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 132–149, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
14. John R. Koza, Forrest H. Bennett III, Jason Lohn, Frank Dunlap, Martin A. Keane, and David Andre. Use of architecture-altering operations to dynamically adapt a three-way analog source identification circuit to accommodate a new source. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
15. 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*, pages 123–131, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
16. Thomas R. Osborn, Adib Charif, Ricardo Lamas, and Eugene Dubossarsky. Genetic logic programming. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, page 728, Perth, Australia, 29 November - 1 December 1995. IEEE Press.
17. R. Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, The University of Birmingham, September 1996.
18. 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*, pages 363–368, Stanford

- University, CA, USA, 28–31 July 1996. MIT Press.
19. 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.
 20. Riccardo Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In *3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97*, 1997.
 21. Stuart C. Shapiro. *Encyclopedia of Artificial Intelligence*. Wiley, New York, second edition, 1992.
 22. Eric V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19, pages 409–423. MIT Press, 1994.
 23. 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.
 24. 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*, pages 406–409, Stanford University, CA, USA, 28–31 July 1996. MIT Press.