Linear-Tree GP and Its Comparison with Other GP Structures

Wolfgang Kantschik[1] and Wolfgang Banzhaf[1,2]

[1] Dept. of Computer Science, University of Dortmund, Dortmund, Germany
[2] Informatik Centrum Dortmund (ICD), Dortmund, Germany

Abstract. In recent years different genetic programming (GP) structures have emerged. Today, the basic forms of representation for genetic programs are tree, linear and graph structures. In this contribution we introduce a new kind of GP structure which we call Linear-tree. We describe the linear-tree-structure, as well as crossover and mutation for this new GP structure in detail. We compare linear-tree programs with linear and tree programs by analyzing their structure and results on different test problems.

1 Introduction of Linear-Tree GP

The representations of programs used in Genetic Programming can be classified by their underlying structure into three major groups: (1) tree-based [Koz92,Koz94], (2) linear [Nor94,BNKF98], and (3) graph-based [TV96] representations.

This paper introduces a new representation for GP programs. This new representation, named linear-tree, has been developed with the goal to give a program the flexibility to choose different execution paths for different inputs. For tree or linear based programs the interpreter usually executes the same nodes (functions) for each input. However, a program may contain many decisions and each decision may call another part of the program code. So the program flow of the linear-tree-program is more natural than linear or tree GP-programs, similar to the program flow of hand written programs.

In *linear-tree* GP each program $\mathcal{P}$ is represented as a tree. Each node in the tree has two parts, a *linear program* and a *branching node* (see Figure 1). The *linear program* will be executed when the node is reached during the interpretation of the program. After the linear program of a node is executed, a child node is selected according to the branching function of this node. If the node has only one child, this child will be executed. If the node has no child the execution of the program stops. During the interpretation only the nodes of one path through the tree, from the root node to a leaf will be executed.

The implementation of linear GP in our system represents a linear program as a variable length list of C instructions that operate on (indexed) variables or constants (see [BDKB98]). In linear GP all operations, e.g. $a = b + 1.2$, implicitly include an assignment of a variable. After a program has been executed its output

J. Miller et al. (Eds.): EuroGP 2001, LNCS 2038, pp. 302–312, 2001.

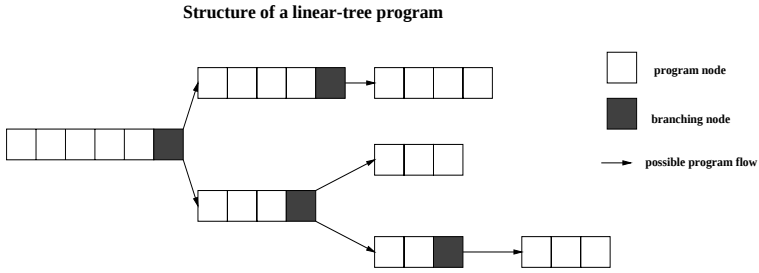

Fig. 1. Individual structure of a linear-tree representation.

value(s) are stored in designated variables. The *branching function* is also a C instruction that operates on the same variables as the linear program, but this function only reads these variables. Table 1 contains a collection of all branching functions. Figure 2 shows an example of a short linear program and a branching function for one node in a linear-tree.

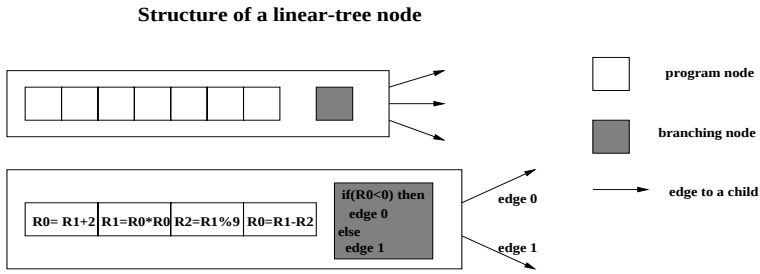

Fig. 2. The structure of a node in a linear-tree GP program (top) and an example node (bottom).

1.1 Recombination of Linear-Tree Programs

A crossover operation combines the genetic material of two parent programs by swapping certain program parts. The crossover for a linear-tree program can be realized in two ways. The first possibility is to perform the crossover like it is done in tree-based GP by exchanging subtrees (see [Ang97]). Figure 3 illustrates this tree-based recombination method. In each parent individual the crossover operator chooses a node randomly and exchanges the two corresponding subtrees. In our system, the crossover points are directed to inner nodes with a higher probability than to terminal nodes. We choose an inner node with a probability of 80 % and a terminal node with 20 % of the crossover operation. This values are also taken for the tree-based crossover in our tests.

Table 1. All the branching operators used in the runs described here.

branching operator	description of the operator
result < 0	If the result register is less than zero the left child is choosen else the right child.
result > 0	If the result register is greater than zero the left child is choosen else the right child.
result $== 0$	If the result register is equal zero the left child is choosen else the right child.
register $x <$ result	If the result register is less than register x the left child is choosen else the right child.
register $x >$ result	If the result register is greater than register x the left child else the right child.
register $x ==$ result	If the result register is equal register x the left child is choosen else the right child.

The second possibility is to perform linear GP crossover. Figure 4 illustrates the linear recombination method. A segment of random position and length is selected in each of the two parents for exchange. If one of the children exceeds the maximum length, crossover with equally sized segments will be performed.

For linear-tree programs we use both methods but only one at a time. The following algorithm for the recombination of linear-tree programs is applied for recombination:

```
procedure crossover ( ind1, ind2)
1    p1 = a crossover point of  ind1;
2    p2 = a crossover point of  ind1;
3    randProb = random value between 0 and 1.;
4    if (randProb < probxover)
5         depth1 = depth of  ind1 after potential crossover;
6         depth2 = depth of  ind2 after potential crossover;
7         if (depth1 < depthMax and depth2 < depthMax)
8              perform a tree-based crossover with the given crossover points.;
9         else
10             perform linear crossover between the nodes p1 and p2;
11        endif
12   else
13        perform linear crossover between the nodes p1 and p2;
14   endif
end
```

In our tests the parameter $prob_{xover}$, which defines the probability whether the tree-based or linear crossover method is used, was set to the 50 %. We also tested the crossover operation with a probability of 10 % and 90 % (the results see Section 3.3).

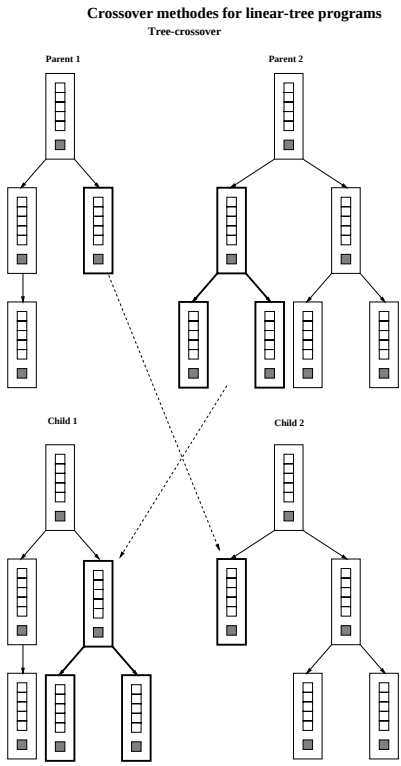

Fig. 3. Crossover-operation of two linear-tree programs using the tree-based crossover method. This crossover method exchanges two subtrees of the programs.

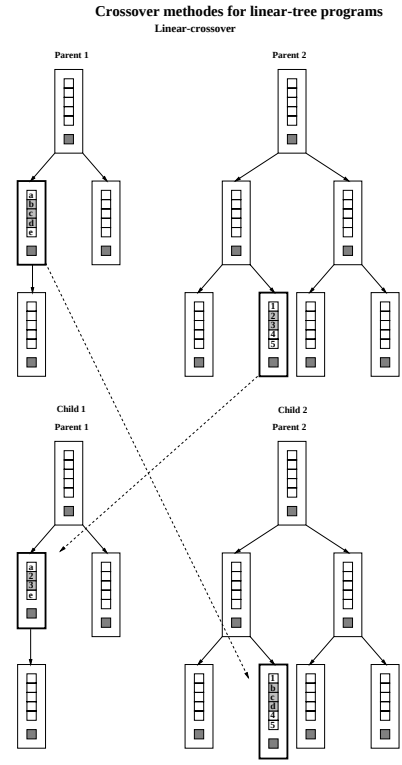

Fig. 4. Crossover-operation of two linear-tree programs using the linear-based crossover method. This crossover method is a two-point crossover, which exchanges a part of the linear-code between the nodes.

1.2 Mutation

The difference between crossover and mutation is that mutation operates on a single program only. After the recombination of the population a program is chosen with a given probability for mutation. The random mutation operator selects a subset of nodes randomly and changes either a node of a linear program, a branching function, or the number of outgoing edges. This version of the mutation operator does not generate new linear sequences. The altered program is then placed back into the population.

2 Test Problems

As test problems we use four different symbolic regression problems, the parity problem, and a classification problem to determine the effect of the linear-tree

structure compared to other representation forms in GP. In Section 3 the results are presented. They show that the new representation performs better than both the linear and tree representations in most test cases.

2.1 Symbolic Regression

We use four different regression problems to test the linear-tree structure. In general, symbolic regression problems deal with the approximation of a set of n numeric input-output relations (x, y) by a symbolic function. The programs should produce an output as close as possible to y if the input is x. The fitness of an individual program p is defined here as the sum of the errors between all given outputs y (here one of the given functions $f(x)$) and the predicted outputs $p(x)$:

$$\text{fitness}(p) = \sum_{i=1}^{n} |p(x_i) - f(x_i)|.$$

The following functions are used as test problems:

- Sine with an input range $[0, 2\pi]$ and 20 fitness cases, chosen uniformly and including both endpoints.
- Sine with an input range $[0, 4\pi]$ and 40 fitness cases, chosen uniformly and including both endpoints.
- Rastrigin, $f(x) = x^2 - 5 * cos(2\pi * x)$, with an input range $[-4, 4]$ and 40 fitness cases.
- $f(x) = \frac{1}{2x} * sin(2x)$ with an input range $[-4, 4]$ and 40 fitness cases, the fitness case $f(0)$ is excluded from the sets of fitness cases.

All variants of GP have been configured with population size of 500 individuals, a maximum crossover and mutation rate of 100 %, and without adf's. This means that in one generation each individual is selected for a crossover and after the crossover each individual will be mutated by the mutation operator. All variants use the arithmetic operations $(+, -, *, /)$.

2.2 Non-regression Problems

For this paper we use the following two non-regression test problems:

- The 4-parity problem, a Boolean problem.
- The chain problem, a classification problem[BB01] Figure 5 visualises the two classes of the problem.

The task of the GP program for both problems is to find a relation that connects a given input x to the correct class, here $c \in 0, 1$, so the fitness cases are also an input-output tuple (x, c). The quality of the programs depends on its ability to find a generalised mapping from the input-output pairs (x, c) of

the n fitness cases. The parity problem is a boolean problem though it can also interpreted as a classification problem.

All variants of GP have been configured with population size of 500 individuals, a maximum crossover and mutation rate of 100 %, and without adf's. For each problem we use different sets of operators; for the chain problem we use arithmetic operations $(+, -, *, /, \sin, \cos)$, for the 4-parity problem we use the following Boolean functions (not, or, and, nand, nor).

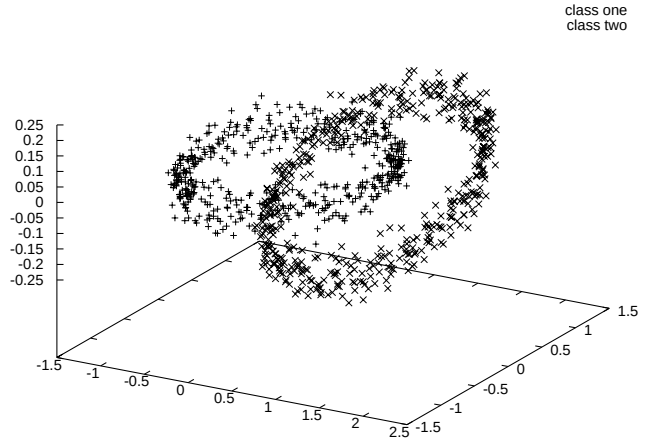

Fig. 5. This figure shows the both links of the chain, which represents the two classes of the problem [BB01]

3 Experimental Results

In this Section we describe the results of using the different GP structures on the six test problems from Section 2. All plots show the average fitness of the best individual. The average is calculated over 30 runs with the same parameter set. In all runs we used tournament selection with a tournament size of two and a population size of 500.

3.1 Difference between the Representations

Figure 6 and 7 show the development of the fitness values using the different structures for the sine problem with one and two periods respectively. Both plots exhibit the same ranking of the different GP structures. In both test cases

the linear-structure is superior to the tree-structure and the new linear-tree-structure is superior to both structures. In Figure 7 we can see that the linear-tree-structure has not reached a stagnation phase even though the tree-structure has virtually reached a stagnation. The plot of Figure 8 shows the trend of the fitness values for the function $\frac{1}{2x} * sin(2x)$. Even for this function we get the same ranking for the three structures. The fitness differences are not as high as for the second sine problem, but this function can be approximated more easily by a polynomial than the sine problem with two periods.

The Rastrigin function is the first and only of our test problems that breaks the established ranking (see Figure 9). Here, a tree-structure is superior to the other program structures, but even here the linear-tree-structure shows better performance than the linear-structure.

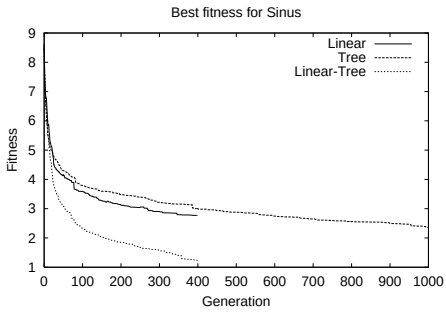

Fig. 6. The curves show the average fitness value of the sine function form 0 to 2π. Each curve is an average over 30 runs. Zero is the best fitness for a individual.

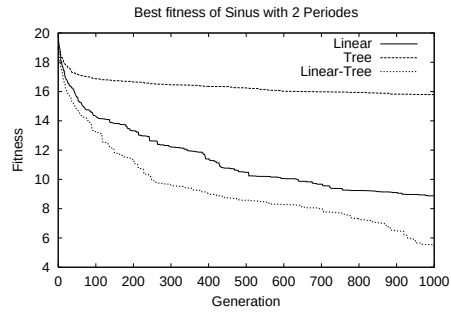

Fig. 7. The curves show the average fitness value of the sine function form 0 to 4π. Each curve is an average over 30 runs. Zero is the best fitness for a individual.

For the non-regression problems we obtain results similar to the regression problems. For both problems GP evolves individuals with linear-tree-structure which have better fitness values than with the other structures. Figure 10 shows the evolution of fitness values for the parity-4 problem. Only a few of the runs with the linear-tree-structure do not reach the optimal solution. The plot in Figure 11 shows the results for the chain problem (of Figure 5). For this problem the linear-tree-structure reaches the best results, but for the second time the tree-structure evolves individuals with a better fitness than the linear-structure.

3.2 Analysis of the Linear-Tree-Structure

After analyzing the individuals with the linear-tree-structure for the parity problem we saw that most programs did not make use of branching and executing different code for different inputs. Some individuals have copies of the same

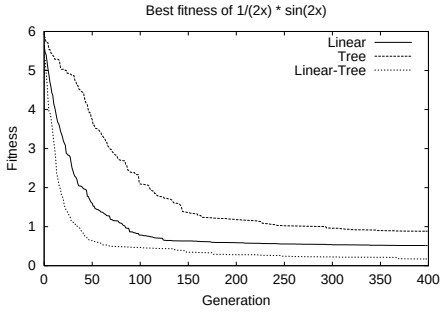

Fig. 8. The curves show the average fitness value of $\frac{1}{2x} * sin(2x)$ function. Each curve is an average over 30 runs. Zero is the best fitness for a individual.

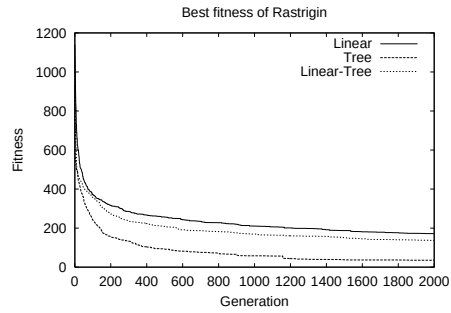

Fig. 9. The curves show the average fitness value of the Rastrigin function. Each curve is an average over 30 runs.

code at each node on the same level in the tree, so that each path through the linear-tree interprets the same code. Other individuals set the parameters for the branching functions in such a way that the result is always the same, and so the path through the individual is always identical. Whether the behavior of the structure during the evolution is a *bloat* phenomena [MM95,LP98] or not will be examined in future work.

Although there is no difference between a linear individual and a linear-tree individual in this case, evolution creates better individuals with the linear-tree-structure for this problem than with the other structures. The answer to this riddle seems to be the existence of *building blocks* [Hol75]. The structure of the linear-tree creates *natural blocks* of code, the nodes of a tree. The chance, then, for a good block or a sequence of blocks to survive a crossover is higher than in other structures. During around 50 % of crossover only subtrees will be exchanged, so *building blocks* can be passed on to other individuals.

Program analysis for the other problems reveal that in these problems individuals use the ability to branch through the code. Obviously, for some problems there is no need to exploit this ability and evolution circumvents the branching ability.

3.3 The Analysis of the Crossover-Operator

The crossover operator for linear-tree programs has two modes, in which genetic material can be exchanged (see Section 1.1). The first mode is the exchange of subtrees and the second mode is the exchange of linear sequences of nodes. During our tests we have set the probability to choose one of the modes to 50 % (parameter $prob_{xover}$). This means that 50 % of the crossover operations are linear crossover operations. The question is whether this parameter has a serious effect on the crossover performance and on evolution. We tested this question with values of 10 % and 90 % for $prob_{xover}$. We also tried changing the crossover

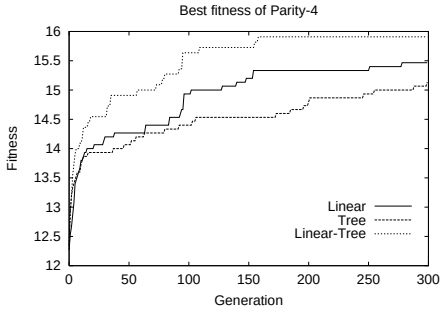

Fig. 10. Average fitness value of the parity 4 problem. Each curve is an average over 30 runs. The best fitness for this problem is 16. The linear-tree-structure shows the best results of the tree structures. In most runs it finds the optimal solution after 150 generations.

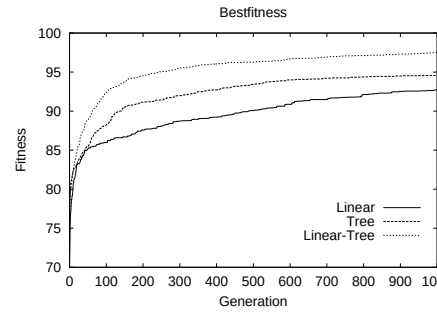

Fig. 11. Average fitness value of the chains classification problem. Each curve is an average over 30 runs. The best fitness is 100 for this problems, the fitness is the classification rate in percent.

operation so that line 10 is not executed in the crossover procedure if the tree-based crossover was not allowed. This means if no tree-based crossover can take place no crossover will be performed.

For our first test to analyze the crossover-operator, we use the standard setting of our system, and we find no significant difference and this parameter has no real effect to the crossover operation. Because of our standard parameter setting after each crossover a mutation is performed. Therefore we performed a test series where mutation after the crossover operation was turned off. The results are shown in Figure 12. The test demonstrates that the gain of linear-based crossover is greater than that of the tree-based crossover for the linear-tree-programs. This plot shows also the result for the evolution without crossover, this curve shows how important mutation is for the evolution. Tree-based crossover makes big changes in the individual code compared to linear crossover and mutation. After a few hundred generations runs with a higher probability for linear-based crossover or mutation yield better results than runs with a high probability for tree based crossover. The linear crossover has the ability to make changes near the root node of the tree easily. Standard tree-based GP finds this very difficult[MH99] so this could also be a reason why linear crossover leads to better results.

4 Summary and Outlook

We tested the performance of linear-tree against other structures of GP programs. In all test cases the linear-tree-structure performed better than the linear-structure, and only in one test case the linear-tree-structure did perform worse

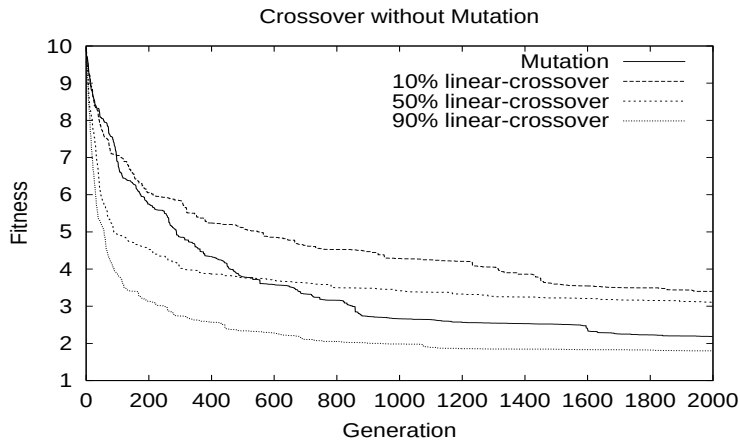

Fig. 12. Average fitness value of the sine function form 0 to 2π for the crossover operator with different values for the value $prob_{xover}$. For these results we use crossover without mutation. The curve 'mutation' is the result without a crossover operator. Each curve is an average over 30 runs.

than the tree-structure. From these tests we can say that the new structure has shown its advantage.

We have observed that the structure of a GP individual makes a significant difference in the evolutionary process and the expressiveness of code. Good performance of a structure may be caused by the effect of building blocks [Hol75], which are the nodes in our linear-tree-structure. In order to clarify whether the good performance of the linear-tree-structure is a general phenomenon for genetic programming, more experiments have to be run on a variety of test problems, but the results achieved so far are strong evidence that the new structure may lead to better result for a range of problems. In future work we plan to extend tests to other problems and check different branching methods. We also have to examine in detail why the new structure leads to better results.

Acknowledgement

Support has been provided by the DFG (Deutsche Forschungsgemeinschaft), under grant Ba 1042/5-2.

References

Ang97. P.J. Angeline. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, San Francisco, CA, 1997. Morgan Kaufmann.

BB01. M. Brameier and W. Banzhaf. Evolving teams of mutiple predictors with Genetic Programming. Technical report, Universitt Dortmund SFB 531, 2001. Data available from the authors.

BDKB98. M. Brameier, P. Dittrich, W. Kantschik, and W. Banzhaf. SYSGP - A C++ library of different GP variants. Technical Report Internal Report of SFB 531,ISSN 1433-3325, Fachbereich Informatik, Universität Dortmund, 1998.

BNKF98. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco und dpunkt verlag, Heidelberg, 1998.

Hol75. J. Holland. *Adaption in Natural and Artifical Systems*. MI: The University of Michigan Press, 1975.

Koz92. J. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

Koz94. J. Koza. *Genetic Programming II*. MIT Press, Cambridge, MA, 1994.

LP98. W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris, 14-15 April 1998. Springer-Verlag.

MH99. Nicholas Freitag McPhee and Nicholas J. Hopper. Analysis of genetic diversity through population history. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

MM95. Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

Nor94. J. P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machine code*. MIT Press, Cambridge, 1994.

TV96. A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81 –116. Oxford University Press, 1996.