Linear-Graph GP – A New GP Structure

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-graph. This is a further development to the linear-tree structure that we developed earlier. We describe the linear-graph structure, as well as crossover and mutation for this new GP structure in detail. We compare linear-graph programs with linear and tree programs by analyzing their structure and results on different test problems.

1 Introduction of Linear-Graph GP

This paper introduces a new representation for GP programs. This new representation, named linear-graph, has been developed with the goal of giving a program the flexibility to choose different execution paths for different inputs. The hope is to create programs of higher complexity, so that we can evolve programs that can compete with the complexity and possibilities of hand-written programs.

Linear-graph is the logical next step after the introduction of linear-tree structure. We have shown the power of the linear-tree structure in [7], but trees are not really the structure of a complex hand written program. Graphs come one step nearer to the control flow of a hand written program, though there is still a long way until we can evolve programs of the complexity of hand-written. Our efforts are devoted is to create a GP-structure able to solve tasks, which cannot be completed with current structures. It is possible for the current structures like,

- (1) tree-based GP [8,9],
- (2) linear-based GP [10,3], or
- (3) graph-based GP[12,2,11],

to create more complex programs and hence solve more complex problems. However we think, that this structures need more time and resources to evolve such programs.

Let us look how the program flow of a hand-coded program could look like. Many programs contain decisions where another part of the program code will be called. After different program parts have been executed they flow together again. If one draws the possible program flows normally it will become a graph.

J.A. Foster et al. (Eds.): EuroGP 2002, LNCS 2278, pp. 83–92, 2002.
© Springer-Verlag Berlin Heidelberg 2002

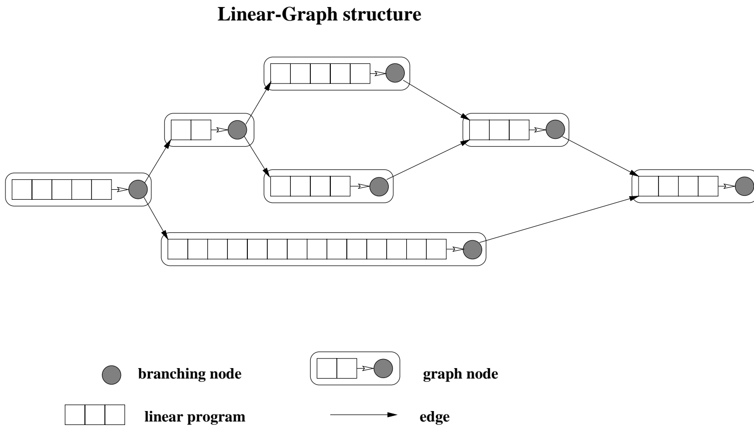

Fig. 1. Individual structure of a linear-graph representation.

So program flow of a linear-graph program is more natural than linear or tree GP-programs and similar to program flow of hand written programs.

In *linear-graph* GP each program $\mathcal{P}$ is represented as a graph. Each node in the graph 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 at all execution of the program stops. During the interpretation only the nodes of one path through the graph, from the root node to a leaf will be executed.

The implementation of linear substructures in our uses a variable length list of C instructions that operate on (indexed) variables or constants (see [5]). 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 values 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 we used in our runs. Figure 2 shows an example of a short linear program and a branching function for one node in a linear-graph.

1.1 Recombination of Linear-Graph Programs

A crossover operation combines the genetic material from two parent programs by swapping certain program parts. The crossover for a linear-graph program can be realized in two ways. The first possibility is to perform the crossover similar to crossover in tree-based GP by exchanging subtrees (see [1]). Here we would exchange subgraphs instead of subtrees, Figure 3 illustrates the recombination method by exchanging a subgraph. In each parent individual the

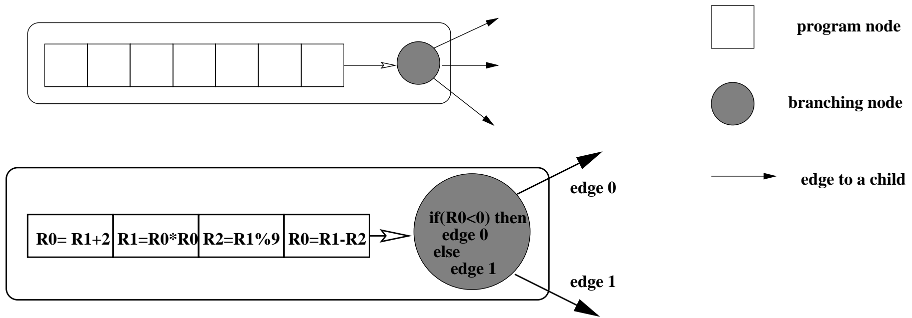

Fig. 2. The structure of a node in a linear-graph GP program (top) and an example node (bottom).

Table 1. All the branching operators used in the runs described here. The data register holds the input data of an individual and is only readable. The result register is the register which is used as output.

branching operator	description of the operator
result register < 0	If result register is less than zero the left child is chosen else the right child.
result register > 0	If result register is greater than zero the left child is chosen else the right child.
result register $<$ operand	If result register is less than the value of an operand the left child is chosen else the right child.
result register $>$ operand	If result register is greater than the value of an operand the left child else the right child.
data register < 0	If data register is less than zero the left child is chosen else the right child.
data register > 0	If data register is greater than zero the left child is chosen else the right child.
data register $<$ operand	If data register is less than the value of an operand the left child is chosen else the right child.
data register $>$ operand	If data register is greater than the value of an operand the left child else the right child.

crossover operator chooses a set of contiguous nodes randomly and exchanges the two subgraphs.

The second possibility is to perform linear 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. The linear crossover is performed for a given percentage of nodes of the graph, we performed this crossover for 10% of the graph nodes.

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

1. Choose the crossover points p_1, p_2 in both individuals.
2. Choose with a given probability $prob_{gx}$ the graph-based crossover method (go to step 3), and with the probability $1 - prob_{gx}$ the linear-based crossover method (go to step 4).
3. If the depth of one of the children does not exceed the maximum depth perform crossover, else go to step 4.
4. Perform linear-based crossover.

In our tests the parameter $prob_{gx}$, which defines the probability whether the graph-based or linear crossover method is used, was set to the 20 %.

1.2 Mutation

The difference between crossover and mutation is that mutation operates on a single program only. After applying recombination to 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. In other words, 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 two symbolic regression problems, a sine wave and the Rastrigin function and a classification problem the two chains, see below. The linear-graph structure is compared to a linear GP structure. In Section 3 the results for the regression and classifications problems are presented.

The fitness measure for the regression problem with program p is defined as mean squared error between all given outputs y (here one of the given functions $f(x)$) and the predicted outputs $p(x)$:

$$\text{fitness}(p) = \frac{\sum_{i=1}^{n} (p(x_i) - f(x_i))^2}{n}.$$

We chose 20 fitness cases for the sine function in the range from $[0, 2\pi]$ uniformly and including both endpoints and the Rastrigin function in the range of $[-2, 2]$, with 40 fitness cases.

For classification we used the chain problem [4], Figure 5 visualizes the two classes this problem. Fitness measured is the number of misclassifications. The task of the GP program is to find a relation that connects a given input x to its correct class, here $c \in \{0, 1\}$, so fitness cases can be written as input-output tuples (x, c). The quality of a program depends on its ability to find a generalized mapping deduced from the input-output pairs (x, c) of n fitness cases.

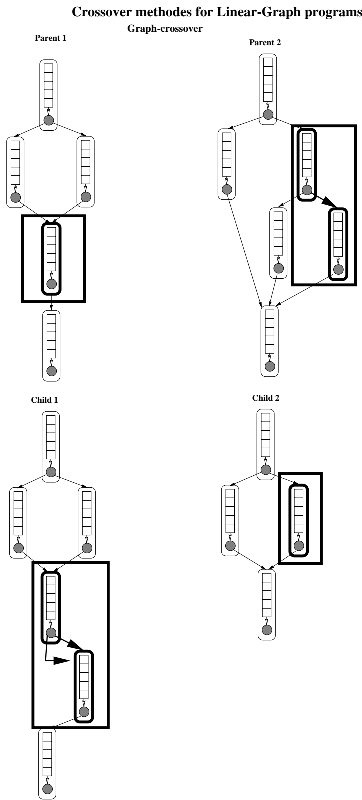

Fig. 3. Crossover-operation of two linear-graph programs using the graph-based crossover method. This crossover method exchanges two subgraphs of the programs.

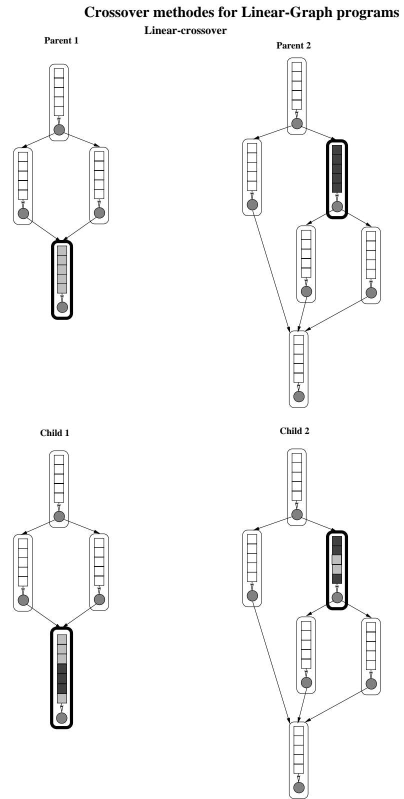

Fig. 4. Crossover-operation of two linear-graph programs using the linear-based crossover method. This crossover method is a two-point crossover, which exchanges a part of the linear-code between the nodes.

All variants of GP have been configured with population size of 100 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 $(+, -, *, /)$. For the chain problem we use arithmetic operations $(\sin, \cos)$ additionally. For all test problems we allow jumps and an if-then-else function.

3 Experimental Results

In this section we describe the performance of the different GP structures with different population sizes on the three test problems from Section 2. All plots

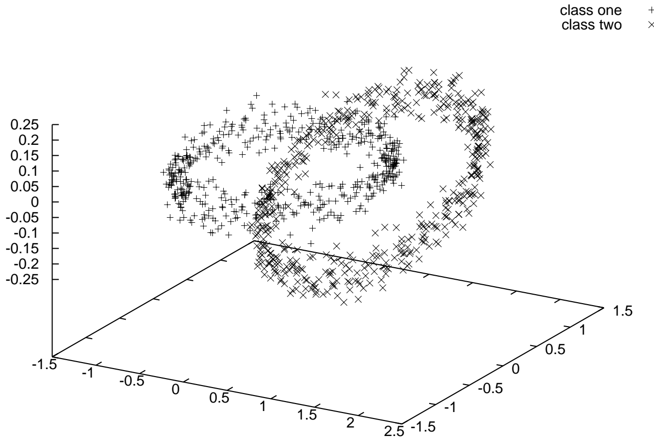

Fig. 5. This figure shows both links of a chain these represent the two classes of the chain problem [4]

show the average fitness of the best individual in different runs. The average is calculated over 50 runs with the same parameter set. In all runs we used tournament selection with a tournament size of 2. We have done two sets of runs, one with a population size of 10 and one with a population size of 100. In all runs we compare on the basis of number of nodes evaluated. So the difference in results cannot be interpreted by the fact that one structure has more or less resources to develop a good solution.

Figure 6 shows development of fitness values using the linear and linear-graph structure for the sine problem. We can see that for a population size of 100 individuals the linear-graph structure reaches a better fitness than the linear structure and the improvement of the fitness value for the linear-graph structure is faster than for the linear-structure. After 20.000 fitness evaluations the linear-graph structure reaches a fitness value which the linear structure reaches after 200.000 fitness evaluations. Even at this point we can see that the new structure supports the evolutionary process. Another very interesting result is the behavior of the linear-graph individuals during an evolution with small populations. Figure 6 also shows the development of fitness values for a population size of 10 individuals. With linear structures we observe the expected result, the performance is inferior to the result with 100 individuals. Even with the same number of fitness evaluations we can not reach the same result on average. The error bars also show the large variance in runs with a population size of 10. The linear-graph structure on the other hand, obtains the same fitness values as with 100 individuals, and error bars shows that these runs have a small variance.

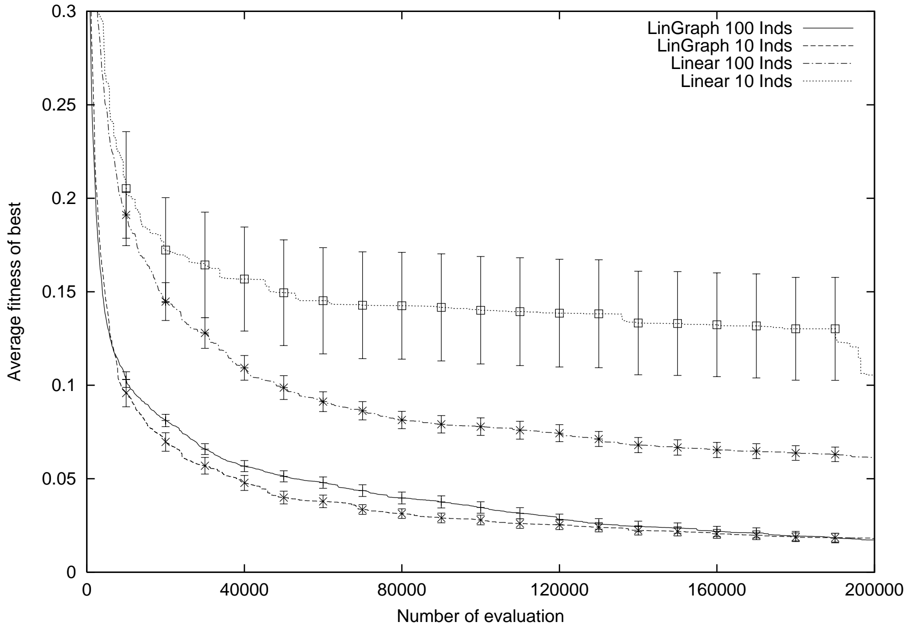

Fig. 6. The curves show the average fitness values for the sine regression problem with data form 0 to 2π. Each curve is an average of 50 runs. Zero is the best fitness for a individual.

Figure 7 shows the development of fitness values using the linear and linear-graph structure for the classification problem. The linear-graph structure reaches a better fitness than the linear structure with a population size of 100 individuals. Even the fitness development during the first 40.000 fitness evaluations is faster for the linear-graph structure. About 20.000 fitness evaluations the linear-graph structure reaches a fitness value which the linear structure reaches after 200.000 fitness evaluations. The behavior of linear-graph individuals during the evolution with small populations is similar to their behavior for the sine problem. The result for 10 individuals is not as good as with 100 individuals for the linear-graph structure, but it still outperforms the linear structure. The interesting result here is the performance of the linear-structure with a population size of 10 for this problem. We expected a similar result as for the sine problem. The error bars shows that there was a high variance for the different runs, however on average runs reach the same fitness as with 100 individuals.

The result for the Rastrigin function is shown in figure 8. The development of fitness values is similar to the case of the sine problem. Linear-graph structures outperform the linear structures with both population sizes. This result shows also the large error bars.The plot also shows that there is almost no difference between runs with population size of 10 or 100 for the linear-graph structure. The runs with the linear-graph structures are the only runs for the Rastrigin problem

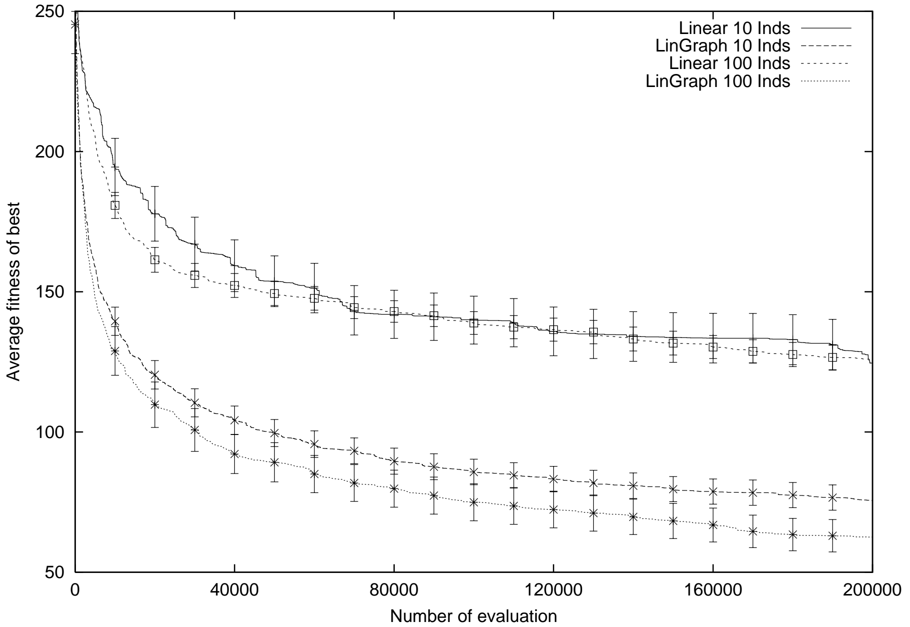

Fig. 7. The curves show the average fitness value of the chain problem. Each curve is an average over 50 runs. Zero is the best fitness for a individual. The x axis is the number of fitness evaluations and the y axis is the number of miss classifications. The chain problem contain 1000 data points, so that 100 means a classification error of 10 %.

where an individual could reach a fitness smaller than 0.2. For the linear structure best fitness is 2.9. This shows that behavior of the linear-graph structure is not only improving average behavior but also improving overall behavior.

4 Summary and Outlook

In conclusion we have observed that linear-graph structures outperform a linear structure significantly. We have also seen that even with a population size of 10 individuals only evolution reaches fitness values which are better than the fitness values of linear structures with 100 individuals. This allows GP to evolve individuals for problems with high cost in fitness evolution.

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 [6], which could be identified with the nodes in our linear-graph structure. In order to clarify whether this good performance of the linear-graph structure is a general phenomenon more experiments need to be run on a variety of test problems, but the results achieved so far are strong evidence that the new structure may lead

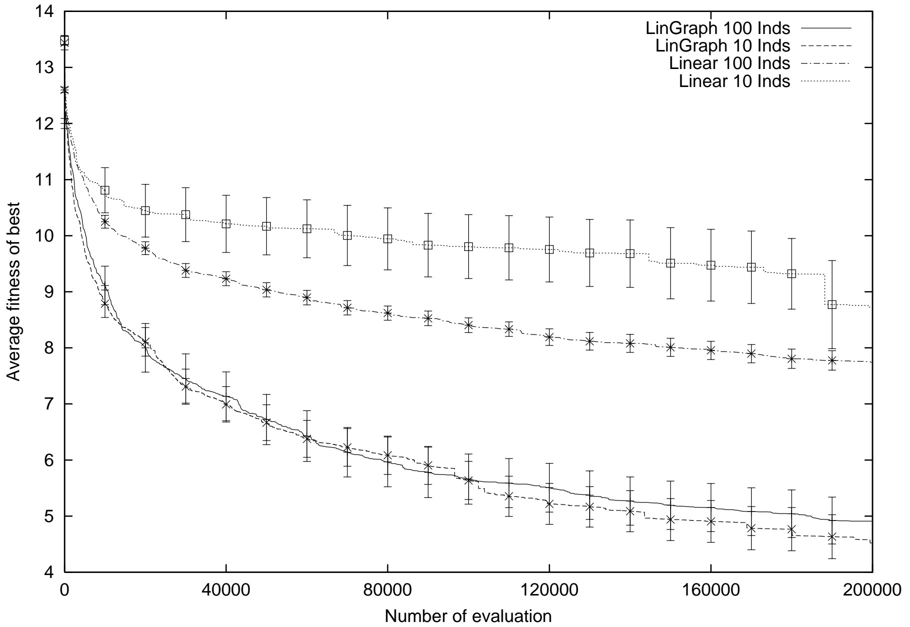

Fig. 8. The curves show the average fitness value of the Rastrigin problem. Each curve is an average over 50 runs. Zero is the best fitness for a individual.The x axis is the number of fitness evaluations.

to better result for a range of problems. But the astonishingly result was the behavior of the linear-graph structure during the evolution with small populations. A careful analysis is now needed to find out what reasons determine the improvements in performance of these new GP structures.

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

References

1. 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.
2. P.J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems (in press)*, 1998.
3. 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.
4. M. Brameier and W. Banzhaf. Evolving teams of mutiple predictors with Genetic Programming. *Genetic Programming and Evolvable Maschines*, 2(4):381–407, 2001.

5. 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.
6. J. Holland. *Adaption in Natural and Artifical Systems.* MI: The University of Michigan Press, 1975.
7. W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.
8. J. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.
9. J. Koza. *Genetic Programming II.* MIT Press, Cambridge, MA, 1994.
10. J. P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machine code.* MIT Press, Cambridge, 1994.
11. Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.
12. A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81 –116. Oxford University Press, 1996.