A New, Node-Focused Model for Genetic Programming

David Jackson

Department of Computer Science, University of Liverpool
Liverpool L69 3BX, United Kingdom
djackson@liverpool.ac.uk

Abstract. We introduce Single Node Genetic Programming (SNGP), a new graph-based model for genetic programming in which every individual in the population consists of a single program node. Function operands are other individuals, meaning that the graph structure is imposed externally on the population as a whole, rather than existing within its members. Evolution is via a hill-climbing mechanism using a single reversible operator. Experimental results indicate substantial improvements over conventional GP in terms of solution rates, efficiency and program sizes.

Keywords: Genetic programming, Graph-based representation.

1 Introduction

In genetic programming (GP), a variety of representations are available to encode the programs representing individuals of the population. Most commonly, programs are stored as tree structures [1], in which leaf nodes are taken from a set of terminals, and internal nodes are drawn from a set of functions appropriate to the problem. Evolutionary operators work on such representations by swapping subtrees or replacing them with new, randomly-generated subtrees.

In linear GP [2] the representation has no such structure imposed on it. Instead, programs are simply sequences of individual instructions. Whereas tree-based GP takes a functional view of programs, in which calculations are passed up a tree as it is evaluated, linear GP is more akin to conventional imperative programming, with intermediate and final results being stored in registers or memory variables. The representation language used can be either real or abstract: when machine code instructions are directly manipulated, the efficiency gains can be substantial [3].

A tree is merely one form of a graph, and so it is perhaps not surprising that it is not the only such graph structure that has been tried for GP. One of the first systems to explore this was PADO (Parallel Algorithm Discovery and Orchestration) [4]. PADO makes use of stack memory and indexed memory, and a graph may contain action nodes and branch-decision nodes. The system was used to evolve parallel programs for classifying images.

Taking inspiration from the parallel processing performed in neural networks, Poli's PDGP (Parallel Distributed GP) [5] uses a grid representation to hold graph-structured programs. Individuals are still subject to (suitably modified) crossover and

A. Moraglio et al. (Eds.): EuroGP 2012, LNCS 7244, pp. 49–60, 2012.

mutation, but programs are more compact than tree-based equivalents, and offer opportunities for concurrent execution. A similar grid-based approach is employed in Cartesian Genetic Programming (CGP) [6], in which the number of rows and columns, and the amount of feed-forward, are all parameters to the system. Originally developed to evolve digital logic designs, the approach made use exclusively of mutation to generate new candidates which took part in a (1+λ) evolutionary strategy, but more recent research has explored the advantages of a new crossover operator [7]. In the GRAPE (GRAph structured Program Evolution) approach [8], graphs contain arbitrarily directed links, and both calculations and node sequencing are determined by a separate data set.

Other researchers have taken conventional tree-based or linear GP and augmented them with additional structures. Often, this is done as a way of introducing modules or other forms of hierarchy into programs [1,9-11]. In linear-tree GP [12], each node of a tree consists of a linear program and a branching node which determines the next node in the tree to be executed. The idea was later extended to more general graph structures [13]. In the MIOST system [14], program trees may contain additional links both to provide more sophisticated interaction between nodes and also to allow multiple outputs from individuals.

In Multi-Expression Programming (MEP) [15,16], each individual has a structure similar to that of single-row CGP, with each node of the graph having links to operands further back in the graph. The main difference is that execution results are computed not only for a program graph as a whole, but also for each of its sub-graphs. The overall fitness of the individual is defined to be the fitness of the best sub-expression. Mutation and crossover are the primary evolutionary operators.

In our own proposed approach, which we call Single Node Genetic Programming (SNGP) we take this idea of associating a fitness with every node a stage further. In our model, every individual in the population has just one fitness value, rather than several as in MEP, but that is because each individual consists of only one node. Another key difference between our approach and others is that individuals are not entirely distinct: they are interlinked in a graph structure similar to that of MEP or CGP, with population members acting as operands of other members. In a sense, then, an entire SNGP population can be viewed as being very similar to a single MEP individual, although, as we shall see, the mechanics of evolution are very different.

2 The SNGP Model

An SNGP population is a set of N members

$$M = \{m_0, m_1, \ldots, m_{N-1}\}.$$

Each member is a tuple of the form:

$$m_i = < u_i, r_i, S_i, P_i, O_i >$$

where:

$u_i \in \{T \cup F\}$ is a single graph node taken from either the function set F or the terminal set T of the problem;

r_i is the rating of fitness for the individual;

S_i is a set of successors of this node;
P_i is a set of predecessors of the node;
O_i is a vector of outputs generated when this node is evaluated.

During initialisation, the population is partitioned in such a way that:

$u_i \in T$ if $i < TNUM$
$u_i \in F$ otherwise

where $TNUM$ is the number of terminals in the terminal set. Moreover, for any u_i, u_j such that $i, j < TNUM$ and $i \neq j$, we have $u_i \neq u_j$.

In other words, the first $TNUM$ members of the population are initialised to represent the members of the terminal set, with each terminal appearing exactly once. All other members contain nodes drawn from the function set. These are allocated at random, and so may be replicated in the population.

For a population member which represents a function, the operands of that function are drawn from other members of the population. The successor set of the node is a list of the population members acting as operands, represented by their position in the population. We make the restriction that for each $s \in S_i$ we have $0 \leq s < i$, i.e. the operands of a function must be 'lower down' in the population (towards position zero).

Similarly, the predecessors of an individual are those population members for which the individual is used directly as an operand, i.e. they take us to the next higher expression level. This means that for each $p \in P_i$ we have $i < p < N$.

Note that for terminal nodes the successor sets are empty. Moreover, as these nodes cannot change during evolution (see later), their predecessor sets are not needed and are also left empty.

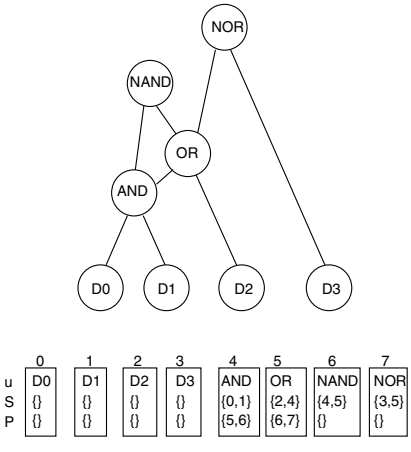

Fig. 1. Small (8-member) SNGP population and corresponding graph structure

Fig. 1 shows how a population of just 8 members might be initialised, together with the corresponding graph. The first four positions in the population are occupied

by terminals, the remainder by functions. For ease of explanation the functions shown here are all different, although in reality functions could be replicated, and certainly will be with larger population sizes. Note that the AND node and the OR node both have two predecessors, i.e. they appear as immediate operands of two other function nodes. This form of reuse is characteristic of SNGP programs, and therefore differs from conventional tree-based GP.

The graph shown here contains eight different expressions, one per node. The simplest expressions are the single-node terminals: D0, D1, D2 and D3. The other expressions are those rooted at the remaining nodes:

AND(D0, D1)
OR(AND(D0, D1), D2)
NAND(AND(D0, D1), OR(AND(D0, D1), D2))
NOR(OR(AND(D0, D1), D2), D3)

It can be seen that, even with only eight nodes, a range of reasonably complex expressions can be encoded. This complexity can rise dramatically when hundreds of nodes are used.

Key to the efficiency of SNGP is the way in which the fitness of each individual is calculated. This is achieved using a form of dynamic programming. During initialisation, each terminal is evaluated across all test input cases, and the outputs generated are stored in O_i. These outputs are used to calculate the fitness values r_i. As initialisation continues, and each randomly selected function is inserted into the population, outputs and fitnesses continue to be computed, but making use of the values already stored for the operands forming the successor set. In this way, the fitness calculation for an individual is highly efficient, involving the application of only one operator or function per test case.

In SNGP there is only one evolutionary operator, called *smut* (successor mutate). The way that *smut* works is that a member of the population is chosen at random, and then one of its operands (i.e. a member of its successor set) is replaced by a reference to a different member of the population (but still lower down in the position order). Figure 2 shows how this might work for the small program graph that was given previously in Figure 1.

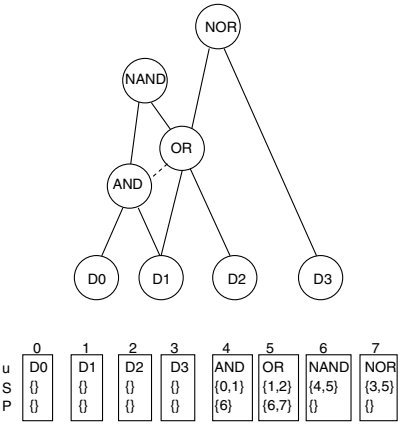

Fig. 2. Effects of the *smut* operator on the population

Here, the first operand of the OR node is being changed from population member number 4 (the AND node) to member number 1 (the terminal D1). Hence, the successor set of node 5 must be changed to reflect this, and node 5 must therefore be deleted from the predecessor set of the AND node. In this example, the new operand is a terminal, and so nothing more needs to be done to the graph structure; when the new operand is a function, its predecessor set must also be updated to add in the new parent.

A modification such as this means that the individual which has been changed must be re-evaluated to determine its new outputs and fitness rating. In our example, the expression OR(D1, D2) must be computed for all test cases. However, this will also have an effect on individuals higher up in the population. Exactly which individuals are affected is determined by the predecessor sets. In Figure 2, the predecessors of the OR node are the NAND node and the NOR node, and so these must be re-executed. In larger graphs, it may be necessary to continue this chain of execution by pursuing the predecessor references until all affected individuals have been re-assessed.

The order in which evaluations proceed up the population can have a great impact on efficiency. Returning to Figure 1, a change to the operands of the AND node might cause the immediate predecessors NAND and OR to be evaluated next. Then, because the OR outputs have changed, the NAND node might be invoked once again. In general, there may be many unnecessary evaluations that take place before the population eventually settles to its final values. To circumvent this, we implement a mechanism in which the predecessor sets are followed to build an ordered 'update list' of all affected individuals. We then execute each member of the list in turn, from the lowest to the highest position in the population, thus ensuring that no node is visited more than once.

Evolution of an SNGP population is driven using a hill-climbing approach. This is based on fitness measurements across the whole population, rather than on single individuals. More formally, and assuming that lower fitness values are better, the aim is to minimize Σr_i. Whenever the *smut* operator is applied, this summation is computed. If the result is worse than the value of the summation before *smut* was invoked, and if no solution has been found, then the modifications made by *smut* are reversed. To make this more efficient, the old outputs and fitness values of each member of the update list are recorded by *smut*, so that they can be put back in place if necessary by a single *restore* operation. Note that worse overall fitness will cause the reversal to be activated even if it means discarding individuals that are fitter than any of those previously present in the population.

3 Experimentation

In evaluating SNGP against conventional GP we have made use of three standard benchmark problems: 6-multiplexer, even-parity, and symbolic regression.

In the 6-mux problem, the aim is to evolve a program that interprets the binary value on two address inputs (A0 and A1) in order to select which of the four data inputs (D0-D3) to pass onto the output. The function set is {AND, OR, NOT, IF}. Fitness evaluation is exhaustive over all 64 combinations of input values, with an

individual's fitness being given in terms of the number of mismatches with expected outputs.

In the even parity problem we search for a program which returns TRUE if the number of inputs set to logic 1 is even, and FALSE otherwise. The function set is {AND, OR, NAND, NOR}. In this paper, we will first consider the 4-input version of the problem, with fitness being given in the range 0-16; later, we will discuss higher level parity problems.

Our final problem is symbolic regression of a polynomial. In our version of this, the polynomial we attempt to match through evolution is $4x^4 - 3x^3 + 2x^2 - x$. The only terminal is x, and the function set is {+, -, *, /}, with the division operator being protected to ensure that divide-by-zero does not occur. The fitness cases consist of 32 x-values in the range [0,1), starting at 0.0 and increasing in steps of 1/32, plus the corresponding y-values. Fitness is calculated as the sum of absolute errors in the y-values computed by an individual, whilst success is measured in terms of the number of 'hits' – a hit being a y-value that differs from the expected output by no more than 0.01 in magnitude.

The problem parameters as they apply to the use of standard GP in all these problems is given in Table 1. For SNGP there are really only two parameters. The first is the population size (number of nodes), which we have arbitrarily set to 100, although later we will discuss the effects of altering this. The second parameter is the 'length' of a run, which we will refer to as *L*. SNGP does not have generations as such; we can think instead in terms of the number of evolutionary operations performed. Since standard GP with a population size of 500 running over 50 generations generates 25,000 individuals via crossover or reproduction, we will set the upper limit on the number of *smut* applications to 25,000. Again, however, we will examine the effects of changing this parameter.

Table 1. GP system parameters common to all experiments

Population size	500
Initialisation method	Ramped half-and-half
Evolutionary process	Steady state
Selection	5-candidate tournament
No. generations	51 generational equivalents (initial+50)
No. runs	100
Prob. crossover	0.9
Mutation	None
Prob. internal node used as crossover point	0.9

In comparing SNGP with conventional GP we consider three factors: solution rate, efficiency, and solution size. The solution rate is given simply in terms of the number of solutions to the problem found in 100 runs. Comparisons of efficiency can be a little more difficult to make fair. For example, it would be possible to compare the number of fitness evaluations performed in each case. However, the nature of a fitness

evaluation performed in standard GP is so different from that of SNGP that comparing the two becomes meaningless. Instead, we use the more uniform measure of standard wall-clock time. The timings we give are for a PC with an Intel Core i7 quad-core processor running at 2.8GHz. The GP systems are compiled using Microsoft Visual Studio as single-threaded processes executing under identical load conditions. Each timing figure is for the number of seconds required to perform 100 runs. Table 2 summarises the results for our three problems.

Table 2. Comparison of SNGP with standard GP

	Even-4		**6-mux**		**Regression**	
	GP	**SNGP**	**GP**	**SNGP**	**GP**	**SNGP**
Soln. rate (%)	14	95	66	100	11	43
Time 100 runs (secs)	16	7	15	7	59	21
Av. soln size	278	38	83	31	223	24
Max soln size	709	56	449	54	1011	48
Min soln size	59	22	10	15	29	12

We can see from this table that SNGP substantially out-performs conventional GP. For the even-4 parity problem, SNGP finds almost seven times as many solutions as standard GP, and in less than half the time. Standard GP improves on the multiplexer problem, but now SNGP finds a solution on every run, again in less than half the time. For the symbolic regression problem, SNGP discovers four times as many solutions in one third of the time of standard GP. It should be pointed out that all the comparative performance figures in this paper have been established as statistically significant using a t-test at the 95% confidence level.

Turning our attention to the program sizes in Table 2, we see that SNGP again comes off best. The explanation for this is of course that, with the parameters we have used, it is impossible for the SNGP programs to grow beyond 100 nodes, whereas standard GP is effectively unbounded. Although it could be argued that this frees standard GP to explore a larger search space, it is clear that it does not hamper SNGP's ability to find solutions. Conversely, standard GP would find it very difficult to find solutions when using such small population sizes.

The only figure that could be regarded as a victory for conventional GP in Table 2 is that of the minimum solution size found for the multiplexer problem. Standard GP finds the following 10-node solution:

IF (A1 IF (A0 D3 IF (A0 D1 D2)) (IF (A0 D1 D0)))

whilst the best that SNGP can find is the following 15-node solution:

26: IF 5 19 7
19: IF 4 3 18
18: AND 2 16
16: IF 6 14 10
14: OR 7 5
10: OR 9 4

9: OR 0 5
7: IF 4 1 0
6: NOT 5
5: A1
4: A0
3: D3
2: D2
1: D1
0: D0

In this linearised form of the program graph, each line represents a node, and the number at the beginning of the line is the index number of the node in the population. As can be seen, population members 0 to 5 represent the terminals of the problem. The way to interpret the graph as a program is to read it in a top-down fashion. The first node (node 26 in this example) is the one at which the solution is rooted. Hence, the top-level function in this program is an IF construct which will test the value of A1 (node 5); if A1 is true, execution will be directed to node 19 (another IF statement), otherwise it will go to node 7 (also an IF statement, corresponding to 'IF A0 then D1 else D0').

That said, the sizes of the programs evolved by SNGP are constrained by the population size. This suggests that reducing the population size might have the effect of encouraging the evolution of smaller programs. However, the question is by how much we can do this while still providing the evolutionary process with enough genetic material to discover solutions. The graph of Figure 3 charts what happens when the population size is altered for the 6-mux problem. Similar graphs are obtained for the symbolic regression and parity problems.

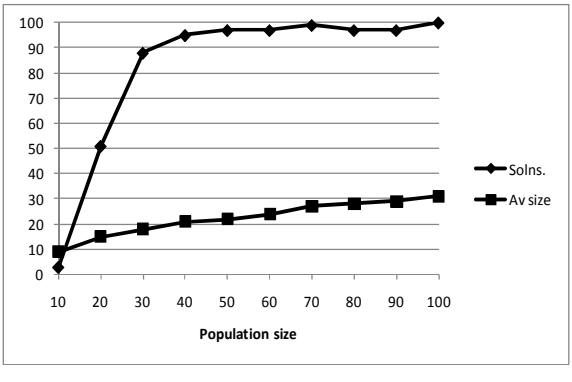

Fig. 3. 6-mux soln rate and av. soln size for varying population sizes

We can see that, as the population size is gradually reduced, we do indeed see the desired reduction in the average solution size. As might be expected, the number of solutions obtained also decreases, but what is surprising is that solutions continue to be found even when the population size becomes quite small. For the 6-multiplexer problem, we require only 10 individuals (and therefore just 10 nodes across the whole

population) to discover solutions, whilst anything greater than 40 individuals is enough to give us solution rates close to the 100% level. Similarly, 20 individuals are sufficient to evolve solutions for the symbolic regression problem, and 30 individuals for the even-4 parity problem.

The population size parameter therefore acts as a useful mechanism for tuning the results we wish to obtain from evolution. Higher values give us lots of solutions; lower values provide fewer solutions, but they are smaller in size and can be generated much quicker. When just 10 individuals are used in the 6-mux problem, we get a solution rate of just 3%. However, the solutions we obtain can be no bigger than 10 nodes (because they are bounded by the population size), and the complete set of 100 runs executes in only 2 seconds.

Hence, the inability of SNGP to match standard GP on minimum program size for 6-mux in our earlier experiments is easily remedied. With a population size of 10, we obtain two solutions of size 9, and one of size 10. In general, the extensive re-use of nodes via multiple pointers in SNGP enables much more compact solutions than the conventional GP equivalents. For example, one 14-node SNGP solution to the 6-mux problem takes up 59 nodes when written out as a standard expression tree such as would be used in standard GP.

The other SNGP parameter – the maximum number of operations per run (L) – can, of course, also be altered. In conventional GP, increasing the duration of runs generally has little effect: populations which converge without finding a solution usually do not recover no matter how much extra evolutionary time they are granted. In contrast, our observations of SNGP runs suggest that many runs are still continuing to evolve when they are terminated. The parameters we have used thus far for SNGP have been sufficient to obtain a 100% solution rate for the 6-mux problem. Table 3 shows what happens in the even-parity and regression problems if the maximum length of each run is increased from 25,000 to 100,000 operations.

Table 3. Effects on even-4 parity and symbolic regression problems when SNGP max run length increased to 100,000 ops

	Even-4	**Regression**
Soln. Rate (%)	99	54
Time 100 runs (secs)	10	61

The results we have presented above are encouraging for these simple benchmark problems, but suppose we increase the problem difficulty? A simple and effective way of investigating this is to apply SNGP to the solution of higher-order parity problems, which are known to be difficult for conventional GP to solve. Table 4 presents the results of these experiments, with N kept at 100 and L maintained at 25,000. By way of contrast, conventional GP is unable to find any solutions to the even-5 parity problem, even when the population size is increased to 4,000 (equivalent to 200,000 evolutionary operations over 50 generations) [1].

Table 4. Performance of SNGP for higher-order even parity problems (N=100, L=25,000)

	Even-5	Even-6	Even-7
Soln rate (%)	58	14	2
Time 100 runs (secs)	47	67	82
Av. soln size	48	52	58
Max. soln size	67	64	59
Min. soln size	31	40	58

The results presented here for even parity also compare well against those reported for Multi-Expression Programming (MEP) [16]. For MEP populations up to size 500, each member having 200 genes, the best success rate for even-4 parity is less than 50% (cf. SNGP's rate of 95-99%). Similarly, MEP with a population of 1000 individuals having 600 genes each has a success rate of just 16.66% for even-5 parity.

It should also be borne in mind that the SNGP results have been obtained using our standard parameter values. As before, these can be manipulated to tune the solution rate, solution sizes, and speed of solution discovery. For example, halving the population size to 50 for the even-5 parity problem provides us with only 19 solutions in 100 runs, but execution of this set of runs is completed in just 13 seconds. At the other end of the scale, increasing L to 100,000 operations whilst keeping N at 100 leads to a 12% solution rate for the even-7 parity problem, found in just over 11 minutes. Moving up to even-8 parity, the parameter values N=200 and L=200,000 gave us 2 solutions in the 5 runs we attempted, one with 100 nodes, the other with 110. In the other 3 runs, the best programs found had fitness values of either 2 or 3.

4 Conclusions

In this paper we have introduced a new graph-based model for genetic programming. It has several key differences from existing representations, perhaps the most striking being that each individual in the population consists of just a single program node. Moreover, the graph structure to which we refer is not contained within individuals, as it is in most other representations, but is external to them: it is imposed on the population as a whole, linking them into a network of functions and their operands. Indeed, it could be argued that an SNGP population is not a collection of members at all, but merely a single individual. However, the complexity of each node, comprising not only the function to be applied, but also its predecessor and successor sets, its output values, and in particular its own identifiable fitness value, makes it worthy of consideration as a distinct individual. This is really just a question of semantics; what really matters is whether the approach has any merit.

Another important difference is the way in which evolution is carried out. We do not use crossover, and the form of mutation we employ is non-standard in the sense that the function or terminal held at a particular node never alters: once a population has been randomly initialized, each node will retain its given operation for the life-time of the run. What does change are the references to other individuals acting as operands of a given function. In this way, we view the dynamics of SNGP evolution

as a search for an optimal set of connections between functions and terminals which are present in sufficient number to solve the problem at hand.

Evolution in SNGP is a lot more collaborative and altruistic than it is in other approaches. The hill climbing approach we use is based on the good of the whole population, not just individuals. If a given operation does not lead to better fitness across the whole population, then its actions are reversed, even if there exist particular individuals which would have benefited greatly from the change. The only exception to this is when the operation leads to a solution being discovered.

Despite the simplicity of its representation, SNGP copes extraordinarily well with the benchmark problems we have thrown at it. Its solution-finding performance is superior in terms of both the numbers of solutions obtained and in the times taken to discover them. It readily finds solutions to higher-order parity problems that are beyond the reach of conventional GP and many other approaches, and it does so using populations that are comparatively minute.

A further advantage to be gained is that these solutions are significantly smaller than those evolved in other approaches. A key factor in this compactness is the ability to re-use nodes as operands of numerous individuals simultaneously, effectively converting them into program modules.

A serious problem in conventional GP is that of bloat – the rapid explosion in program sizes as evolution proceeds. SNGP does not have this problem. This is partly due, of course, to the restrictions on program growth imposed by a fixed population size. But it is also because SNGP does not have the equivalent notion of introns, which are often a major contributing factor in bloat. In SNGP, every node is evaluated, and therefore has its own intrinsic worth in addition to the value it may offer in its role as an operand of other individuals.

All of these findings are highly encouraging. There is, however, still much research to be done on SNGP. Some of the questions which immediately jump to mind include: What are the dynamics of SNGP, such that it is able to find solutions so readily with such small populations? What is the potential for exploiting the parallelism inherent in both the SNGP system and in the programs it evolves? Are we using the best evolutionary operator, or should it be modified or others introduced? Are we making too much of randomness in the initialization phase and in the way changes are made during evolution? How well does SNGP deal with problems in which functions have side-effects and which are therefore not amenable to dynamic programming?

We hope to address these questions and others in future work.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
2. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, Heidelberg (2007)
3. Nordin, P., Banzhaf, W., Francone, F.D.: Efficient Evolution of Machine Code for CISC Architectures Using Instruction Blocks and Homologous Crossover. In: Spector, L., et al. (eds.) Advances in Genetic Programming, vol. 3, pp. 275–299. MIT Press, Cambridge (1999)

4. Teller, A., Veloso, M.: PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System. Technical Report CS-95-101, Department of Computer Science, Carnegie-Mellon University, USA (1995)
5. Poli, R.: Parallel Distributed Genetic Programming. In: Corne, D., et al. (eds.) New Ideas in Optimization, pp. 779–805. McGraw-Hill Ltd., UK (1999)
6. Miller, J.F., Thomson, P.: Cartesian Genetic Programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000)
7. Clegg, J., Walker, J.A., Miller, J.F.: A New Crossover Technique for Cartesian Genetic Programming. In: Thierens, D., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf (GECCO 2007), London, England, UK, pp. 1580–1587 (2007)
8. Shirakawa, S., Ogino, S., Nagao, T.: Graph Structured Program Evolution. In: Thierens, D., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf. (GECCO 2007), London, England, UK, pp. 1686–1693 (2007)
9. Angeline, P.J., Pollack, J.: Evolutionary Module Acquisition. In: Proc. 2nd Annual Conf. on Evolutionary Programming, La Jolla, CA, pp. 154–163 (1993)
10. Jackson, D.: The Performance of a Selection Architecture for Genetic Programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 170–181. Springer, Heidelberg (2008)
11. Rosca, J.P., Ballard, D.H.: Discovery of Subroutines in Genetic Programming. In: Angeline, P., Kinnear Jr., K.E. (eds.) Advances in Genetic Programming, ch. 9, pp. 177–202. MIT Press, Cambridge (1996)
12. Kantschik, W., Banzhaf, W.: Linear-Tree GP and Its Comparison with Other GP Structures. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 302–312. Springer, Heidelberg (2001)
13. Kantschik, W., Banzhaf, W.: Linear-Graph GP - A New GP Structure. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 83–92. Springer, Heidelberg (2002)
14. Galvan-Lopez, E.: Efficient Graph-Based Genetic Programming Representation with Multiple Outputs. International Journal of Automation and Computing 5(1), 81–89 (2008)
15. Oltean, M.: Evolving Digital Circuits using Multi-Expression Programming. In: Zebulum, R.S., et al. (eds.) Proc. 2004 NASA/DoD Conf. on Evolvable Hardware, Seattle, USA, pp. 87–97 (2004)
16. Oltean, M.: Solving Even-Parity Problems using Multi-Expression Programming. In: Chen, C., et al. (eds.) Proc. 7th Joint Conf. on Information Sciences, North Carolina, USA, vol. 1, pp. 295–298 (2003)