Evolution and Acquisition of Modules in Cartesian Genetic Programming

James Alfred Walker[1], Julian Francis Miller[2]

[1] School of Computer Science, University of Birmingham,
Edgbaston, Birmingham, UK B15 2TT.
jaw@cs.bham,.ac.uk
http://www.cs.bham.ac.uk/~jaw
[2] Department of Electronics, University of York,
Heslington, York, UK, YO10 5DD.
jfm@ohm.york.ac.uk
http://www.elec.york.ac.uk/staff/jfmhome.htm

Abstract. The paper presents for the first time automatic module acquisition and evolution within the graph based Cartesian Genetic Programming method. The method has been tested on a set of even parity problems and compared with Cartesian Genetic Programming without modules. Results are given that show that the new modular method evolves solutions up to 20 times quicker than the original non-modular method and that the speedup is more pronounced on larger problems. Analysis of some of the evolved modules shows that often they are lower order parity functions. Prospects for further improvement of the method are discussed.

1 Introduction

Since the introduction of Genetic Programming (GP) by John Koza [3] researchers have been trying to improve the performance of GP and to develop new techniques for reducing the time taken to find an optimal solution to many different types of evolutionary problems. One such approach, called Evolutionary Module Acquisition is capable of finding and preserving problem specific partial solutions in the genotype of an individual [1]. Since then researchers have been interested in the potential and power that this feature brings to the evolutionary process and have built on this work or taken ideas from it for their own specific situations, to re-use these partial solutions as functions elsewhere in the genotype [2] [7][8][13].

Recently another form of GP called Cartesian Genetic Programming (CGP), has been devised that uses directed graphs to represent programs rather than a tree based representation like that of GP. Even though CGP did not have Automatically Defined Functions (ADFs) it was shown that CGP performed better than GP with ADFs over a series of problems [5][6]. The work reported in this paper implements for the first time in CGP a form of ADFs by automatically acquiring and evolving modules. We call it Embedded CGP (ECGP) as it is a representation that uses CGP to construct modules that can be called by the main CGP code. The number of inputs and outputs

to a module are not explicitly defined but result from the application of module encapsulation and evolution.

The problem of evolving even parity functions using GP with the primitive Boolean operations of AND, OR, NAND, NOR has been shown to be very difficult and has been adopted by the GP research community as good benchmark problems for testing the efficacy of new GP techniques. It also is particularly appropriate for testing module acquisition techniques as even-parity functions are more compactly represented using XOR and EXNOR functions. Also smaller parity functions can help build larger parity functions. Thus parity functions are naturally modular and it is to be expected that they will be evolved more when such modules are provided. It is therefore of great interest to see whether modules that represent such functions are constructed automatically. We show that the new technique evolves solutions to these problems up to 20 times quicker than the original. It also scales much better with problem size. The plan of the paper is as follows. In section 2 we describe CGP. Section 3 is an overview of related work. In section 4 we explain the method of module acquisition and evolution. Our experimental results and comparisons with CGP are presented in section 5. Section 6 gives conclusions and some suggestions for further work.

2 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming was developed from methods developed for the automatic evolution of digital circuits [5][6]. CGP represents a program as a directed graph (that for feed-forward functions is acyclic). The genotype is a list of integers that encode the connections and functions. It is a fixed length representation in which the number of nodes in the graph is bounded. However it uses a genotype-phenotype mapping that does not require all nodes to be connected to each other. This results in a bounded variable length phenotype. Each of the nodes represents a particular function and the number of inputs and outputs that each node has, is dictated by the arity of function. The nodes take their inputs in a feed forward manner from either the output of a previous node or from one of the initial program inputs (terminals). The initial inputs are numbered from 0 to n-1 where n is the number of initial inputs. The nodes in the genotype are then also numbered sequentially starting from n to m+n-1 where m is the user-determined upper bound in the number of nodes. These numbers are used for referencing the outputs of the nodes and the initial inputs of the program. If the problem requires k outputs then these will be taken from the outputs of the last k nodes in the chain of nodes. In Fig. 1 a genotype is shown and how it is decoded (an even 4-parity circuit). Although each node must have a function and a set of inputs for that function, the output of a node does not have to be used by the inputs of later nodes. This is shown in Fig. 1, where the output of nodes 8, 11 and 13 are not used (shown in grey). This causes areas of the genotype to lie dormant, leading to a neutral effect on genotype fitness (neutrality). When point mutations are carried out on genes representing connections (the mutation is constrained to respect the directed and acyclic nature of the graphs) these dormant genes can be activated or active genes can be made dormant. This unique type of neutrality has been investigated in detail

[6][11][14] and found to be extremely beneficial in the evolutionary process for the problems studied.

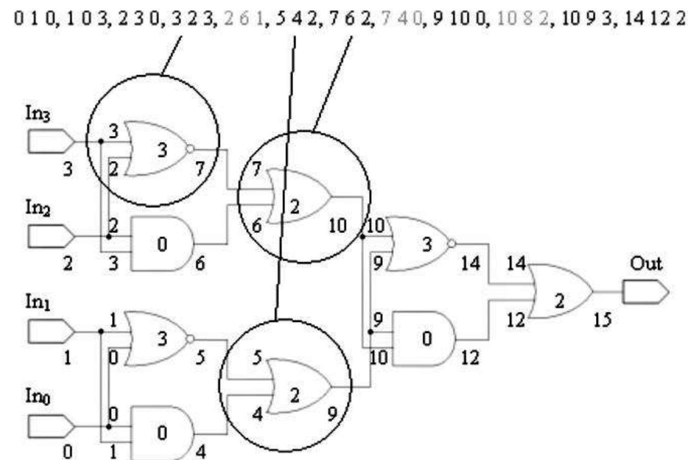

Fig. 1. Cartesian genotype and corresponding phenotype for an even 4-parity program

The evolutionary algorithm used for the experiments is a form of 1+λ evolutionary strategy, where λ=4, i.e. one parent with 4 offspring (population size 5). The algorithm is as follows:

1. Randomly generate an initial population of 5 genotypes and select the fittest;
2. Carry out point-wise mutation on the winning parent to generate 4 offspring;
3. Construct a new generation with the winner and its offspring;
4. Select a winner from the current population using the following rules:
   - If any offspring has a better fitness, the best becomes the winner.
   - Otherwise, an offspring with the same fitness as the best is randomly selected.
   - Otherwise, the parent remains as the winner.
5. Go to step 2 unless the maximum number of generations is reached or a solution is found.

3 Related work on Module Acquisition

The original idea of Module acquisition [1] was to try and find a way of protecting desirable partial solutions contained in the genotype, in the hope that it might be beneficial in finding a solution. This is because in practice you may find a desirable partial solution in the genotype, but due to the nature of evolution, an operator could modify the partial solution therefore causing the program to take longer to find a solution. Module acquisition does this by introducing another two operators to the evolutionary process, *compress* that selects a section of the genotype to make it immune to manipu-

lation from operators (the module) and *expand* that decompresses a module in the genotype therefore allowing this section of the genotype to be manipulated once more. The fitness of a genotype is unaffected by these operators. However they affect the possible offspring that might be generated using evolutionary operators. Atomisation [1] not only makes sections of the genotype immune from manipulation by operators but also represents the module as a new component in the genotype therefore allowing the module to be manipulated further by additional compress operators. This allows the possibility of having modules within modules therefore creating a hierarchy organisation of modules. These techniques have been shown to decrease the time taken to find a solution by reducing the amount of manipulations that can take place in the genotype. Rosca's method of Adaptive Representation through Learning (ARL) [7] also extracted program segments that were encapsulated and used to augment the GP function set. The system employed heuristics that tried to measure from population fitness statistics good program code and also methods to detect when search had reached local optima. In the latter case the extracted functions could be modified. More recently Dessi et al [2] showed that *random* selection of program sub-code for re-use is more effective than other heuristics across a range of problems. Also they concluded that, in practice, ARL does not produce highly modular solutions. Once the contents of modules are themselves allowed to evolve (as in this paper) they become a form of automatically defined function (ADF), however in contradistinction to Koza's form of ADFs [4] and Spector's automatically defined macros [8], there is no explicit specification of the number or internal structure of such modules. This freedom also exists in Spector's more recent PushGP [9].

In addition to decreasing computational effort and making more modular code van Belle and Ackley have shown that ADFs can increase the evolvability of populations of programs over time [10]. They investigated the role of ADFs in evolving programs with a time dependent fitness function and found that not only do populations recover more quickly from periodic changes in the fitness function but the recovery rate increases in time as the solutions become more modular.

Woodward [12] showed that the size of a solution is independent of the primitive function set used when modularity is permitted, thus including modules can remove any bias caused by the chosen primitive function set.

4 Algorithm details and mutation operators

The performance of CGP and ECGP were tested on even 4 through to even 8 parity. The output of even parity is one if an even number of inputs are one and zero otherwise. Initially all genotypes are randomly initialized with fifty nodes (150 genes). The fitness is defined as the number of phenotype output bits that differ from the perfect parity function. A perfect solution has score zero. Every generation, any modules in the function list are removed and any modules present in the fittest genotype of the generation are added to the function list. This allows the 1-point mutation operator to randomly choose from any of the modules found in the fittest genotype and the primitive functions to insert into the genotype of the offspring for that generation. The creation of modules allows new functions to be defined from a combination of primitive

functions that can then be re-used in other areas of the genotype. For all the experiments, both versions of the program: CGP and ECGP were averaged over fifty independent runs.

4.1 Modules

In this paper we only allowed modules to contain nodes rather than other modules. Also we only allowed the number of nodes in a module to be not greater than a certain user defined value. Modules were required also to have greater than one node for obvious reasons. The modules were required to have a minimum of two inputs and a maximum number of inputs equal to twice the number of nodes contained in the module. This is so that a module has at least the first node, and at most all the nodes, in the module connected to the outputs of earlier nodes or modules or the initial inputs outside the module. It must also have a minimum of one output and a maximum number of outputs equal to the number of nodes contained in the module so that there is at least one output from a node, and at most every output from a node in the module available for connection to the later nodes or modules outside of the module. Modules with no outputs are not allowed, as they would simply contain "junk code" which is not in use and could never be connected, therefore increasing the complexity and size of the genotype. The number of inputs and outputs that a module initially has is determined by the connections between the nodes and modules when a module is created. The nodes within the module whose inputs were connected to the outputs of earlier nodes and modules or the initial inputs when modularisation takes place remain connected to the outputs of the same nodes, modules or initial inputs via the inputs of the module. The later nodes or modules whose inputs were connected to the outputs of the nodes contained in the module before modularisation took place remain connected to the same outputs of the nodes contained within the modules via the module outputs. The module inputs are the initial inputs to the CGP program in the module, therefore they act as pointers to the output of the previous node or module in the genotype which represents their value as an initial input to the CGP program in the module. This means that each module input now has a number and the nodes in the module are now numbered starting from the "number of inputs", as they would be in a CGP program. The outputs of the module are also numbered starting from the "number of inputs + number of nodes" as this allows them to be treated as part of the genotype in the module.

4.2 Operators

ECGP uses four main evolutionary operators: a standard point mutation operator, a compress operator, an expand operator and a module mutation operator.

Mutation. The 1-point mutation operator used for ECGP is the same as the mutation operator used in standard CGP. It selects a node or module from the genotype at random and then chooses randomly one of the inputs or the function of the selected node or module to mutate. If an input of a node or module is chosen for mutation then the new value for the input is chosen at random from the outputs of any of the previous nodes or modules in the genotype so that it preserves the directedness of the graph. If a function of a node is chosen for mutation, then the new value for the function of the node is chosen at random from any of the pre-defined primitive functions or any of the modules contained in the module list. However, if the function of a module is chosen for mutation, there are certain conditions that must be met.

If the function chosen for mutation belongs to a module that was introduced to the genotype by the 1-point mutation operator then the new value for the function of the module is chosen in the same way as the function of a node. This is because these modules are treated just like the primitive functions, they represent a copy of a section of the genotype that has been reused in another area of the genotype. We found that this also helps to stop "bloat", as the total length of the genotype can be made shorter at any point by changing a module to a primitive function. On the other hand, if the function chosen for mutation belongs to a module that was introduced to the genotype by the compress operator (rather than the 1-point mutation operator), the function of the module cannot be changed, as the module is immune from function mutation. This is because it is an original section of the genotype encapsulated in a module, which can only be altered once the module has been decompressed by the expand operator.

Whenever a module in the genotype (with arity m) is mutated to a primitive function (with arity n), the new function uses the first n inputs from the module so that it keeps the number of changes in the genotype to a minimum. The same goes for when a primitive function is mutated to a module, the first "k" inputs of the module use the values of the inputs of the primitive function and the rest of the inputs are randomly generated as required. The new value for either an input or function of a node or module is chosen at random with an equal probability. The 1-point mutation operator has a probability of 0.6 of being used on the fittest parent of the population in each generation and a probability of 0.3 of being used in conjunction with either the compress or expand operator on the fittest parent of the population in each generation. The remaining probability of 0.1 is the chance of a module mutation being used on the fittest parent of the population. This is because every offspring is to be mutated in some way to minimise the chance of two parents in the population having the same genotype.

Compress. The compress operator randomly selects two points in the genotype of the fittest parent, a minimum of two nodes apart and a maximum of the pre-defined maximum module size, and creates a new module containing the nodes between these two selected random points. In the work of this paper it was chosen to disallow modules being called within modules - so the modules can only contain nodes. The module then replaces the sequence of nodes between the two randomly selected points in the genotype but is not added to the module list. The only time modules are added to the module list is in the selection process of the fittest parent of a generation. The compress operator has the effect of making the contents of the module immune from the 1-point mutation operator and also shortening the genotype of the fittest parent but

does not affect the fitness of the parent. The compress operator has a probability of 0.1 of being used on the fittest parent in the production of each new member of the population for each generation. This value was chosen because in tests it proved to be optimal when compared with higher and lower values. If the encapsulation process is too frequent too many modules are introduced and they don't have enough time to replicate through the genotype if they are associated with higher fitness. For lower values not enough modules are produced.

Expand. The expand operator randomly selects a module from the genotype of the fittest parent and replaces the module with the nodes contained inside. This operation can only be used on modules that have been made by the compress operator, as the module contents were nodes in the original genotype. We did investigate the possibility of allowing modules created by point mutation to be expanded but found the genotype code grew uncontrollably in length. This operator therefore has a lengthening effect on the genotype. The expand operator has a probability of 0.2 of being applied to the fittest parent of each generation when creating each new member of the population. This value proved to be optimal in tests as it means that modules can only survive if they exist in the genotype more than once. This is because there is a greater chance of a module being destroyed by the expand operator than created by the compress operator, so only the good modules can survive by being replicated (by 1-point mutation) in genotypes with improved fitness.

Module mutation. The module mutation operator consists of five different mutations which all affect the contents or the structure of a module. The operator works by firstly selecting a module at random from the module list and then applying one of the following mutations (at random). Note that the changes only apply to all occurrences of the mutated module in a *single offspring* and not to any occurrences of the module in the whole population.

Add input. The "add input" mutation randomly selects an output of a previous node or module in the genotype and creates a new module input to act as a pointer to the selected node or module output. Once the new input has been created, the operator randomly selects an input of a node contained inside the module and reassigns it to the new module input. An illustration of this process is shown overleaf in Fig. 2.

Remove input. This operator reduces the number of inputs by one each time it is applied but only if there are more than two inputs. This is because the module must have a minimum of two inputs to connect the first node in the module to the previous nodes and modules in the genotype. First the operator randomly selects the module input that it is going to remove. Then it checks through the nodes contained in the module to see if any node inputs are connected to the selected module input. If any inputs are found, they are randomly reassigned to one of the other module inputs or to the output of any previous nodes contained in the module. Once nothing is connected to the module input it is deleted from the module. An example of the remove input operator is the reverse of the example shown above in Fig.2.

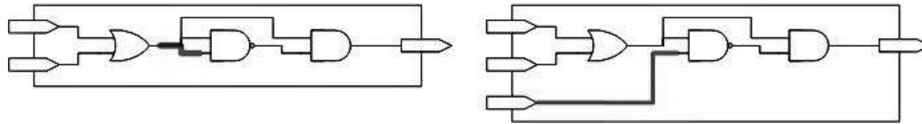

Fig. 2. An illustration of the phenotype of a module before (left) and after (right) the application of the add input operator

Add output. The add output operator increases the number of outputs that a module has by one each time it is applied to a module in the fittest parent, providing that there are fewer outputs than nodes in the module. The add output procedure is started by randomly selecting a node output contained in the module. A module output that points to the chosen node output is then added to the module. The addition of another module output allows later nodes or modules outside the chosen module greater connectivity to the module, but nothing is connected to the modules new output and the fitness of the parent hasn't changed from that of the fittest parent. Therefore the second step of the procedure is to apply the standard mutation operator to the parent, as this will mutate the parent, hence altering its fitness and maybe even creating a connection to the mutated module via its new output.

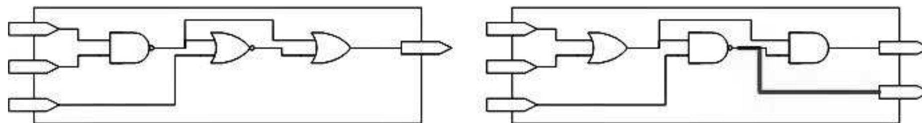

Fig. 3. An illustration of the phenotype of a module before (left) and after (right) the application of the add output operator

Remove output. The remove output operator reduces the number of outputs a module has by one each time it is applied to the fittest parent unless the module only has one output. This is because if the only output of a module were removed it would have no way of allowing latter nodes or modules in the genotype to connect, thus making the module "junk" which can never be used until an add output operator is applied to it. This operator has the effect of limiting the number of connections to nodes contained in the module. The first step of the operator is to randomly select an output of the module that is going to be removed. Before removing the module output however, it is possible that the selected output of the module is in use by later nodes or modules in the genotype. Therefore all the inputs of the nodes or modules that follow the mutated module in the genotype are checked and if an input of a later node or module is using the selected module output, then it is randomly reassigned to one of the other module outputs. The selected module output is deleted once it is no longer in use.

1-point mutation. This mutation operator is essentially the same as the standard mutation operator in CGP but with some limitations. The operator starts by selecting either a node contained in the module or a module output at random. If a node is selected, then it randomly mutates either an input or the function of the node. If an input is selected for mutation then the new value for the input can be any of the module inputs or any output of the previous nodes in the module and is chosen at random. If a function is selected for mutation then the new value can be any of the primitive functions (AND, NAND, OR, NOR) and is chosen at random. No modules from the module list can be used, as this would allow modules inside other modules. If a module output is selected for mutation then its new value can be the output of any of the nodes contained inside the module and is chosen at random. The new value of an output cannot be any of the module inputs as this allows a connection that completely bypasses the contents of the module and is the same as connecting to the output of the node or module previous to the module in the genotype.

5 Results

5.1 What is the optimum maximum module size?

We investigated the variation in average evolution time for the even 4 parity problem as a function of maximum module size (varying module size from 3 to 11 primitives). If the maximum module size is too small then it may not be possible to create a good module. If it were possible, it might take a long time to find, as the limited number of nodes would mean exploration would be slow, as there would be very few, if any, unused nodes. If the maximum module size is set too large then the complexity of the modules could be too high. We found a marked improvement in performance for a maximum module size between three and five but performance flattened off for larger modules (with size 8 performing best). This could be due to the fact that one requires a minimum of three primitives to construct either the XOR or EXNOR function.

5.2 Performance comparison of CGP versus ECGP

The next experiment was a direct comparison between ECGP and CGP to see the differences in speed of solving even-parity functions. The maximum module size was chosen to be five for ECGP. The results are shown in Table 1.

Over all five of the even parity problems tested ECGP varies between 1.25 and 20.27 times faster than standard CGP. Notice that the speedup factor grows with problem size, indicating that ECGP may perform substantially better on even larger problems. It was observed that as the fitness of genotypes improved so the proportion of modules to primitives grew.

Table 1. Average number of generations required to solve even parity problems for CGP and ECGP (to calculate the average number of individuals processed multiply by 4)

	CGP	ECGP	Speedup of ECGP vs. CGP
Even 4-parity	20,432	16,324	1.25
Even 5-parity	73,393	45,480	1.61
Even 6-parity	243,105	71,941	3.38
Even 7-parity	874,883	77,985	11.22
Even 8-parity	2,737,314	135,056	20.27

To give the CGP program a fairer chance against ECGP we looked at how many nodes were contained in the genotype of the final parent when a solution was found. In ECGP the genotype can have grown significantly when all the modules are expanded to nodes. We found that when we ran CGP program using the average number of nodes calculated from ECGP for the corresponding parity problem we found it to be even slower. This suggests strongly that it was not the program size of the phenotypes evolved using ECGP that provided its advantage, but rather, it was the nature of the modules that made the difference. However further investigation is required as it might have been that the mutation rate was too low for the number of nodes so the mutations were being wasted on the junk code in the genotype.

5.3 What kinds of modules evolve?

We examined genotypes that solved the parity problems. In the majority of cases, the genotype consisted mainly of modules and a few primitive functions, which shows that the modules were more desirable in terms of improving fitness. Occasionally we observed modules with few active nodes (sometimes only a single primitive was active).

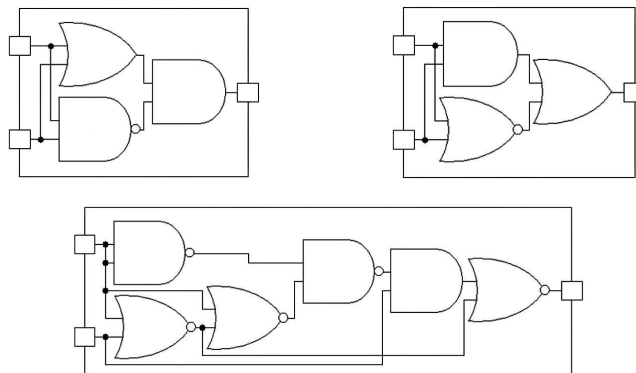

Fig. 4. Evolved modules producing XOR (top left and bottom) and EXNOR.

However, in most cases there were only approximately three to six different modules present, with some modules being used at least ten times in the genotype. The pheno-

type of modules that were being frequently used almost always constructed the XOR or EXNOR functions. These functions were not always made in the same way, some were made compactly out of three nodes, while others were made in a much more complex way, as shown in Fig. 4.

The average number of modules that were available for re-use per generation was approximately five but this could vary depending on the size of module chosen and length of genotype used, as both of these factors could allow a greater or fewer number of modules to be created respectively.

7 Conclusion

We have presented a form of module acquisition and evolution called ECGP. The new method is able to evolve solutions to even parity problems much quicker than the original non-modular form of CGP. Furthermore the speedup grows with problem difficulty, and we found that ECGP was able to evolve solutions to even 8 parity about 20 times quicker on average. It would be interesting to see if ECGP performs better than CGP and GP on other problems.

Other types of problem that could benefit from this approach are the design of adder and multiplier digital circuits as these are also modular like the even-n-parity problems. Many other problems that are modular could benefit as well. Problems where this approach wouldn't be beneficial are quite simple problems, whether modular or not, as the overhead of having to compress, evolve and expand modules might make this approach slower when compared to a non-modular approach such as CGP.

Currently ECGP does not allow modules within modules. We intend to allow this in future investigations. Our results indicate that success is associated with modules building smaller parity functions, thus we can hope that we might improve performance even further.

References

1. Angeline, P. J. Pollack, J. (1993) Evolutionary Module Acquisition, Proceedings of the 2nd Annual Conference on Evolutionary Programming, pp. 154-163, MIT Press, Cambridge.
2. Dessi, A. Giani, A. Starita, A. (1999) An Analysis of Automatic Subroutine Discovery in Genetic Programming, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 996-1001, Morgan-Kaufmann, San Francisco.
3. Koza, J. R. (1993) Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, London.
4. Koza, J. R. (1994) Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press, London.
5. Miller, J. F. (1999) An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, pp 1135-1142, Morgan Kaufmann, San Francisco.

6. Miller, J. F. Thomson, P. (2000) Cartesian Genetic Programming, Proceedings of the 3rd European Conference on Genetic Programming, Edinburgh, Lecture Notes in Computer Science, Vol. 1802, pp 121-132, Springer-Verlag, Berlin.
7. Rosca, J. P. (1995) Genetic Programming Exploratory Power and the Discovery of Functions, Proceedings of the 4th Annual Conference of Evolutionary Programming, San Diego, pp 719-736, MIT Press, Cambridge.
8. Spector, L. (1996) Simultaneous evolution of programs and their control structures, Advances in Genetic Programming II, pp. 137-154, MIT Press, Cambridge.
9. Spector, L. (2001) Autoconstructive Evolution: Push, PushGP, and Pushpop, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, pp. 137-146. San Francisco, CA: Morgan Kaufmann Publishers
10. Van Belle, T, and Ackley, D.H. (2001) Code Factoring and the Evolution of Evolvability,. Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, pp. 1383--1390. San Francisco, CA: Morgan Kaufmann Publishers
11. Vassilev, V. K. and Miller J. F. (2000) The Advantages of Landscape Neutrality in Digital Circuit Evolution, Proceedings of the 3rd International Conference on Evolvable Systems: From Biology to Hardware (ICES2000), Lecture Notes in Computer Science, Vol. 1801, 252-263. Springer, Berlin.
12. Woodward, J. R. (2003) Modularity in Genetic Programming, Proceedings of the Fifth European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 2610, pp. 258--267, Springer-Verlag, Berlin.
13. Yu, T. Clack, C. (1998) Recursion, Lambda Abstractions and Genetic Programming, Proceedings of the 3rd Annual Conference on Genetic Programming, pp. 422-431, Morgan Kaufmann, San Francisco.
14. Yu, T. and Miller, J. F. (2001) Neutrality and the evolvability of Boolean function landscape, Proceedings of the 4th European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 2038, pp. 204-217, Springer-Verlag, Berlin.