A Divide & Conquer Strategy for Improving Efficiency and Probability of Success in Genetic Programming

Cyril Fillon and Alberto Bartoli

University of Trieste, P.le Europa, 1, 34127 Trieste, Italy
{cfillon, bartolia}@univ.trieste.it

Abstract. A common method for improving a genetic programming search on difficult problems is either multiplying the number of runs or increasing the population size.

In this paper we propose a new search strategy which attempts to obtain a higher probability of success with smaller amounts of computational resources. We call this model Divide & Conquer since our algorithm initially partitions the search space in smaller regions that are explored independently of each other. Then, our algorithm collects the most competitive individuals found in each partition and exploits them in order to get a solution. We benchmarked our proposal on three problem domains widely used in the literature. Our results show a significant improvement of the likelihood of success while requiring less computational resources than the standard algorithm.

1 Introduction

When one uses Genetic Programming to solve a problem, he has two expectancies: on the one hand, maximize the probability to obtain a solution, and on the other hand, minimize the amount of computational resources to get this solution.

Unfortunately performance on a given problem may be strongly dependent on a broad range of parameters, including the choice of the functions and terminals set, size and composition of the initial population, maximum number of generations and so on.

To overcome the difficulties of such problems a traditional method was to use a larger population and to increase the maximum number of generations [1][6]. Large populations were considered beneficial because they maintain diversity and may avoid premature convergence. However, more recent works advocate for different approaches, using either populations of moderate or variable size (e.g. [5][9]), or using multiple independent short runs (e.g. [8][2]) in order to outperform a long run.

Recent research [4] confirms that the composition of an initial population has a crucial influence on the probability of success for a problem. It is suggested that the initial population must contain a sufficient quantity of useful building blocks and that these blocks must be part of the fittest individuals. Moreover it was shown in [3] that the building blocks used during the GP process are not dispersed throughout in the initial population, but is instead concentrated in a subset of individuals.

Thus the effectiveness of GP to solve a problem is conditioned by its ability to create potential good individuals in the initial population and identify the individuals which are most likely to provide building blocks useful to find the solution.

In this paper we build on these existing results and propose a new model attempting to meet these requirements. Our model works in two phases. During the first phase we partition the search space in smaller regions that are explored independently of each other. We do so by generating initial populations carefully tailored to maximize the coverage of each region. Then, in the second phase, we collect the most competitive individuals found in each region and explore the resulting search space. In other words, we attempt to generate individuals which maximize the coverage of the search space and then we attempt to exploit the most promising individuals. Our strategy aims to obtain a solution with a better probability of success for a lower computational cost.

The outline of the paper is as follows. In Section 2 we describe our strategy in detail. Section 3 presents the experimental procedure used to study the behavior of the new algorithm with various metrics. Section 4 discusses the results of our experiments. Section 5 concludes and anticipates on further evolutions of our strategy.

2 Divide & Conquer Strategy

Our proposal, that we call *Divide & Conquer*, is inspired by earlier models for coarse-grained parallelization of the genetic programming process [10][11]. These models build a net of subpopulations called "demes". Each deme evolves independently of one another during a sequence of consecutive generations. Then, demes may exchange information by migrating individuals between each other according to a predefined pattern. Our proposal uses the concept of demes but works differently.

2.1 Model Description

First of all, we apply a reduction and differentiation function on the functions set F_S, as follows. Let n denote the cardinality of F_S. We build all possible subsets of F_S composed of exactly p elements, where p is a parameter of the algorithm such that $p < n$. We may apply the same procedure also on the terminals set T_S, or on the union of both $T_S \cup F_S$. If the reduction and differentiation is applied on F_S then the elements of T_S are added to the new subsets, if it is applied on T_S we add the elements of F_S. We denote by f_{RD} the reduction and differentiation function and by RD^i_{SS} ($i \in [1, n]$) the subsets generated.

Each deme operates on one of the subsets generated by f_{RD}. It follows that each deme operates on a subset of functions and terminals different from the subset of any other deme. Subsets obtained from T_S must contain at least one element and those obtained from F_S must contain at least two elements or, two elements and one element respectively from T_S and F_S. Otherwise, the use of an evolutionary approach would be meaningless.

The number of demes must be the same as the number of the subsets generated by f_{RD}, that is $numOfDemes = C_n^p$. The algorithm analyzed in this paper use $p = n - 1$. It follows that, in our case it will be $numOfDemes = C_n^{n-1}$, hence $numOfDemes = n$.

At this point, the initial population of each deme is constructed based on the subset associated with that deme. Each deme evolves independently of each other deme, until either the problem is solved or a maximum number of generations $maxDemeGenNumber$ is reached.

If no solution is found, i.e., all demes reach the maximum number of generations, we merge all the demes and keep only the best individuals based on a ranking selection procedure. This new population then evolves as usual, i.e., either until the problem is solved or a maximum number of generation is reached.

Full details about our algorithm are given in Figure 2.

2.2 Model's Dynamics

An example of the algorithm's dynamics is shown in Figure 1 where the final population size is 500, the maximum number of generations is 95 and the maximum number of generations for a deme is 5. For this example we have chosen to reduce only the functions set. The algorithm initiates by creating 4 demes which evolve independently for 5 generations. Then, the individuals of the demes are ranked according to their fitness. The best 500 individuals are used to build up a new population. This population evolves until the problem is solved or until the maximum number of generations is reached.

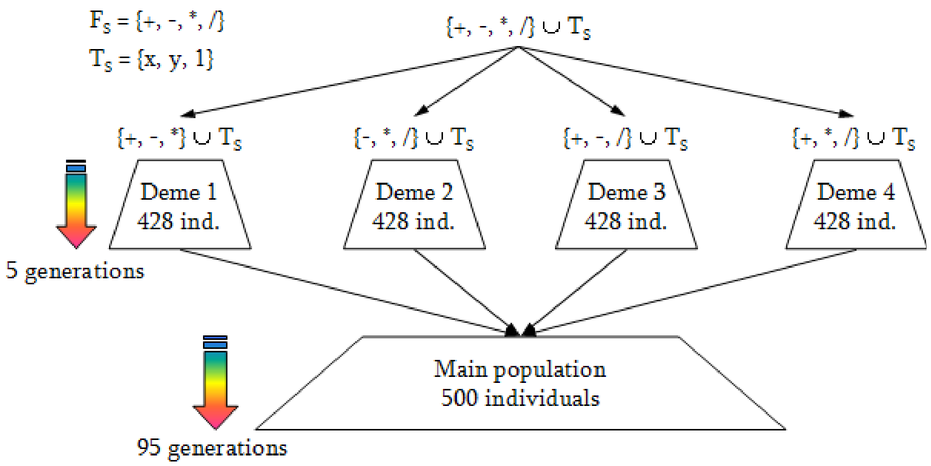

Fig. 1. Example of the Divide & Conquer strategy

```
$divideAndConquer(T_S, F_S, C_S, populationSize, maxGenNumber, maxDemeGenNumber)$

$T_S$                  Terminals set
$F_S$                  Functions set
$C_S$                  the set chosen by the user where $f_{RD}$ is applied either $T_S$, or $F_S$, or $T_S \cup F_S$
$populationSize$       Population size
$maxGenNumber$         Maximum number of generations
$maxDemeGenNumber$     Maximum number of generations for each deme

1. $n = |C_S|$
2. $numOfDemes = C_n^{n-1} = n$ // Number of demes
3. Generate subsets $RD_{SS}^1, RD_{SS}^2, \ldots, RD_{SS}^n$ by applying $f_{RD}$ on $C_S$
4. if $C_S == F_S$ then
       Foreach $i \in [1, n]$, $RD_{SS}^i = T_S \cup RD_{SS}^i$
5. else if $C_S == T_S$ then
       Foreach $i \in [1, n]$, $RD_{SS}^i = F_S \cup RD_{SS}^i$
6. $demePopulationSize = floor\left(populationSize \frac{|T_S \cup F_S| - 1}{|T_S \cup F_S|}\right)$ // Population size for each deme
7. $j = 1$ // Deme counter
8. while problem is not solved or $j \leq numOfDemes$
   (a) Randomly initialize a deme population $D_j$ from elements of $RD_{SS}^j$
   (b) $D_j = evolve(D_j, demePopulationSize, maxDemeGenNumber)$
   (c) Next deme: $j = j + 1$

9. Create a new population $P$ composed of the best ranked individuals in $\bigcup_{k=1}^{numOfDemes} D_k$
10. $P = evolve(P, populationSize, maxGenNumber)$
11. Get the best individual from $P$

$evolve(population, populationSize, maxGenNumber) : finalPopulation$

$population$         Population to evolve
$populationSize$     Population size
$maxGenNumber$       Maximum number of generations
$finalPopulation$    the population returned after evolution

1. $i = 0$ // Initial Generation
2. $P_i = population$ // Population at a given generation $i$
3. while problem is not solved or $i < maxGenNumber$
   (a) Evaluate population $P_i$
   (b) Generate a new population $P_{i+1}$ by reproduction, crossover, mutation of individuals
         i. Select genetic operation $O(O_r, O_c, O_m)$
        ii. Select best individuals $B_S$ from current population ($P_i$)
       iii. Generate offspring $(O, B_S, P_{i+1})$
   (c) Next generation: $i = i + 1$

4. return $P_i$
```

Fig. 2. The Divide & Conquer algorithm

We used a little number of generations to evolve the demes since we do not expect to find a solution in a deme but only to explore different regions of the search space and then to make emerge potential good individuals. In a second step, the final population is evolved on a longer period of time in order to exploit the building blocks included in the best individuals discovered during the exploration step. Note that we construct demes in such a way that the population size of a deme is smaller than the size of the final population. The reason for this choice is twofold. On the one hand, we believe we can reduce the population size since the search space is substantially smaller. On the other hand, we increase the probability that the final population will include individuals from different demes, thereby increasing its diversity.

3 Experimental Procedure

We benchmarked our proposal on a range of standard test problems used in genetic programming research. We present them briefly here but a more detailed description can be found in [6].

Multiplexer 6 bits. The goal is to determine a boolean function which decodes a binary address and restores the value contained in the corresponding register of data. The training set consists of the 64 possible combinations of inputs-outputs. The fitness function evaluates the number of correct answers provided by the program considered on the whole set of training.

Santa Fe ant. An artificial ant tries to find some pieces of food which are arranged along a path on a two dimensional grid. The "Santa Fe Trail" is an irregular trail composed of 89 food pellets. The fitness function counts the number of pieces of food picked up by the ant.

Symbolic regression. The goal is to find a mathematical expression, in symbolic form, that fits a given sample of data points. Our training set is composed of 81 elements corresponding to all combinations of the integer values taken in the interval $[-4, 4]$ assigned to each combination of input's variables. The fitness function computes the sum of the errors on the training set i.e. the sum of the distances between the desired values and those obtained with the program considered.

We used a second degree polynomial function with two parameters:

$$2x^2 - 3y^2 + 5xy - 7x + 11y - 13$$

We indicate functions and terminals set for each problem in Table 1

Table 1. Terminals and functions set

	Multiplexer 6 bits	**Santa Fe ant**	**Symbolic regression**
Terminals set	$A0, A1, D0, D1, D2, D3$	Left, Right, Move	$1, x, y$
Functions set	And, Or, Not, If	IfFoodAhead, Progn2, Progn3	$+, -, /, \times$

For our experiments we used Sean Lukes Evolutionary Computation and Genetic Programming Research System (ECJ13) which is freely available on the web at `http://cs.gmu.edu/ eclab/projects/ecj/`. We developed a companion package which implements our Divide & Conquer model without any modification to the original API.

The first two problems are provided with the API, for the third we slightly modified the code of the multivalued regression example in order to implement our own function.

For each problem, we executed the following algorithms.

Classical GP. For comparison purpose, we used the standard GP algorithm with populations size 500, 1000, 1500 and 2000.

Divide & Conquer. For each problem we apply the Divide & Conquer strategy by reducing either the terminals set, or the functions set, or the union of both. The size of the final population, as resulting from merging the demes, was set to 500. We refer to these as follows:
D&C (F_S) when the reduction and differentiation function f_{RD} is applied only on F_S
D&C (T_S) when f_{RD} is applied only on T_S
D&C ($F_S \cup T_S$) when f_{RD} is applied on $F_S \cup T_S$

Divide & Conquer naive. In order to evaluate the effectiveness of our reduction and differentiation procedure, we repeated the very same tests as those of the previous suite, but without applying f_{RD}. For example, we repeated the test *D&C* (F_S) with the same number of demes but without applying f_{RD} on F_S.

It can be seen that we executed 10 tests for each problem. Each test is the result of 100 independent executions. Each execution starts with a different seed for the random number generator. Moreover, we used the same seeds for each test.

We allocate the same maximum number of fitness evaluations for each test. That is, the generation at which that number of fitness evaluation is reached (200000 in our experiments) is the last generation. The parameters common to all tests are summarized in the Table 2.

Table 2. Parameter settings

Parameter	**Setting**
Selection	Tournament of size 7
Initialization method	Ramped Half-and-Half
Initialization depths	2-6 levels
Maximum depth	17
Internal node bias	90% internals, 10% terminals
Crossover rate	90%
Reproduction rate	10%
Number of runs	100
Max number of generations for a deme	5

We focused on the following metrics:

Percentage of success, a run is considered successful if the algorithm finds an optimal solution.

Number of fitness evaluations performed on successful runs.

Time spent on successful runs. This index captures the fact that each evaluation has its own cost, depending for instance on the number of nodes or the complexity of each node in that evaluation. The previous index, in contrast, treats all evaluations as having the same cost.

Obviously, the absolute value of the "time" performance index is not very meaningful: while probability of success and number of evaluations describe properties that are intrinsic to the genetic programming process, time is related to the specific hardware and software platform used. However, as we shall see, the *normalized* value of the "time" performance index does provide important insights into the behavior of the algorithms.

4 Results

Figure 3 shows the percentage of success achieved by each test. We do not report the percentages of success for the *Multiplexer 6 bits* because almost all tests reach 100% (only the test based on the standard algorithm with a population size of 500 give a value sligtly lower with 97% of success).

We note that our proposal exhibits the best probability of success, provided the differentiation and reduction function is applied either on the function set F_S or on the union $F_S \cup T_S$. In particular, for the symbolic regression problem, the improvement with respect to the best result with the classical GP algorithm is 27% and 22%, respectively. For the ant problem the improvement is 20% and 19%, respectively. It can be seen that the Santa Fe ant problem benefits by our strategy whereas it is considered as a deceptive problem[7].

We also note that, with the classical GP algorithm and for a limited number of evaluations, use of a larger population may improve the probability of success but up to a certain upper bound. For example, in the ant problem, the performance

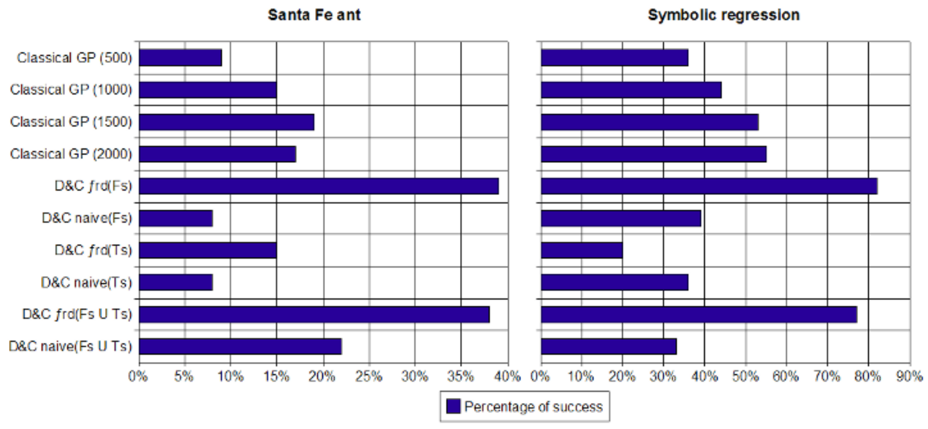

Fig. 3. Percentage of success (for the classical algorithm we indicate in parenthesis the population size)

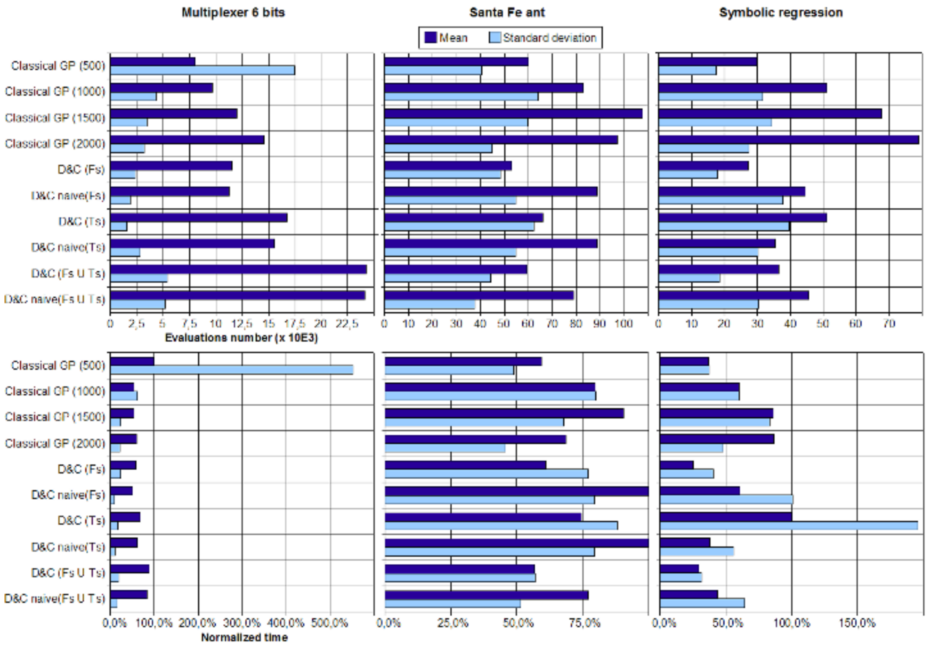

Fig. 4. The upper row of charts presents the average and standard deviation number of fitness evaluations and the lower row, the average and standard deviation of the normalized time

with initial population 2000 is worse than with initial population 1500. In the symbolic regression problem we have not reached the upper bound, however, the improvement from 1500 to 2000 is quite small.

The next suite of experiments is meant to assess the efficiency of the methods (cf. Figure 4). So we counted for each test the average number of fitness evaluations for all runs which achieve a success. We also measured the average time consumed Te for the same runs, and normalized it versus the slowest one $\frac{Te}{Te_{slow}}$. For example, the standard algorithm applied to the ant problem with a population size of 500 obtains a success after 60000 fitness evaluations or after 22% of the computing time used by the slowest test ($D\&C$ *naive*(F_S)).

Once again, the Divide & Conquer strategy using f_{RD} applied on F_S or on $T_S \cup F_S$ achieves the best results for the ant and symbolic regression problem and that, independently of the metric used. For the multiplexer problem, there is neither improvement, nor a significant computational overload for our model. As our approach uses several demes, we did not expect to give any benefits to problems which can be solved in a small number of generations with small populations.

As an aside, the results in Figure 4 confirm that measuring the (normalized) time required for each test does provide important insights into the cost of each algorithm. For example, in the ant problem, the $D\&C$ *naive*(F_S) and $D\&C$ *naive*(T_S) are by far the most expensive tests, but this fact would be hidden if one assumed that all evaluations have the same cost.

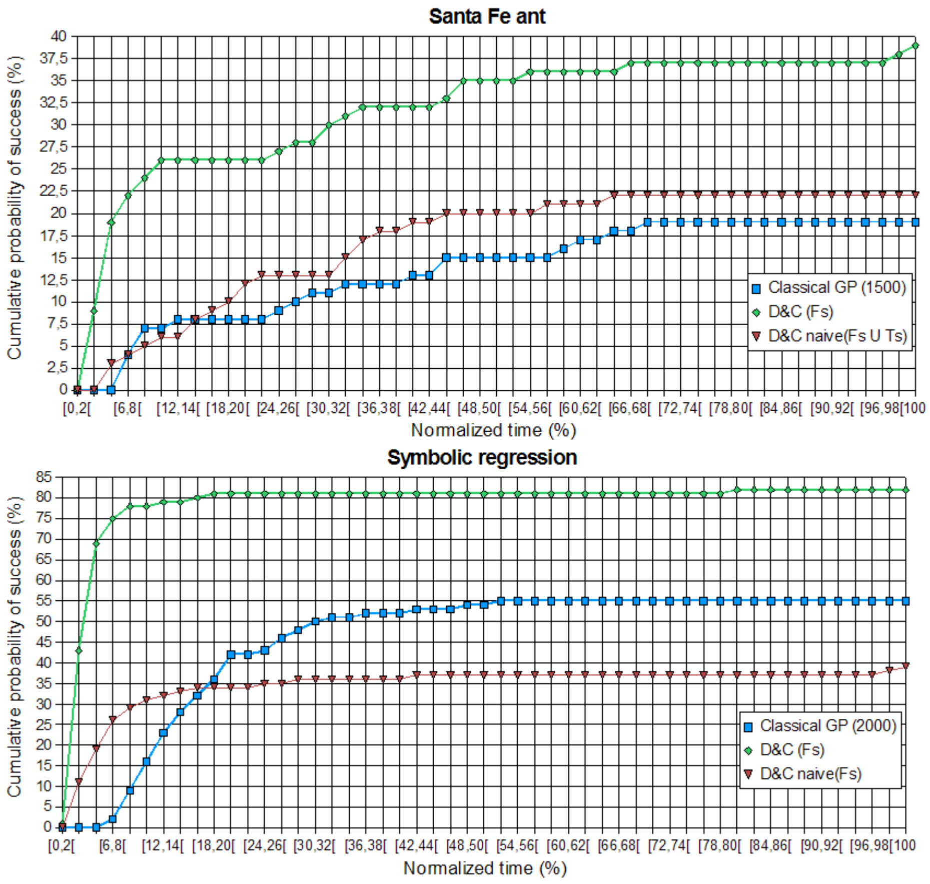

Fig. 5. Cumulative probability of success versus normalized time for the ant and symbolic regression problem

Finally, we evaluate the cumulative probability of success as a function of the computational cost (measured by the normalized time). For example, the standard algorithm applied to the ant problem gives a likelihood of success of 15% for a computing time ranging between 48 and 50% of the slowest test. For the sake of readability, we plotted only the tests that give the best probability of success for each model (*Classical GP*, *Divide & Conquer*, *Divide & Conquer naive*).

It appears clearly that *D&C* (F_S) maximizes the probability of success for a given computational cost. It is interesting to note that, in the symbolic regression problem, *D&C* (F_S) reaches its upper bound very quickly, that is, not only this method provides the best probability of success, it also reaches its best performance much faster (more than 10 times) than the two other best methods.

One can also note that for all the methods, increasing the number of generations and thus the computing time does not bring a significant improvement. Thus, for the problems considered our experiments confirm the fact that long runs are not useful.

5 Conclusions and Future Work

In this paper we introduced a new search strategy for the genetic programming in order to maximize the probability of success with a smaller amount of computational resources. The Divide & Conquer strategy offers a new way to manage the evolutionary process through two keys ideas.

Firstly, we use the concept of demes built on subsets of functions and terminals in order to maximize the coverage of the search space. In this way, each deme works on a region of the original search space.

Secondly, we decompose the evolutionary process in two distinct levels of research. With this approach, the higher level combines the best partial solutions found by the lower level. In doing so, our model has clearly demonstrated its efficiency on the proposed problems, and the results show that the probability of success is improved for a reduced computational cost.

Future works for the Divide & Conquer strategy will investigate the efficiency of the model on others problems in order to determine automatically whether the reduction and differentiation function should be applied either on the functions set, or on the terminals set, or on both, according to the characteristics of the problem.

We work also on an extended version of the algorithm described in Section 2 which might be applied recursively within each deme. In this case, the reduction and differentiation function will operate on the subset RD_{SS} associated with each deme.

Acknowledgments

The authors are grateful to C. Poloni and L. Onesti for their insightful comments. This work was supported by the Marie-Curie RTD network AI4IA, EU contract MEST-CT-2004-514510 (December 14th 2004).

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming - An Introduction; On the Automatic Evolution of Computer Programs and its Applications. (1998), Morgan Kaufmann, dpunkt.verlag
2. Cantu-Paz, E., Goldberg, D.E.: Are Multiple Runs of Genetic Algorithms Better than One? In Proceedings of the Genetic and Evolutionary Computation Conference GECCO'03, (2003), LNCS 2723, Springer-Verlag, 801-812
3. Daida, J.M., Samples, M.E., Byom, M.J.: Probing for Limits to Building Block Mixing with a Tunably-Difficult Problem for Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference GECCO'05, June 25-29, (2005)
4. Daida, J.M.: Towards Identifying Populations that Increase the Likelihood of Success in Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference GECCO'05, June 25-29, (2005)

5. Gathercole, C., Ross, P.: Small Populations Over Many Generations Can Beat Large Populations Over Few Generations in GP. In Koza, J.R. et al. eds GP, (1997), Morgan Kaufmann, 111–118
6. John R. Koza.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
7. Langdon, W.B., Poli, R.: Why Ants are Hard. In Genetic Programming 1998: Proceedings of the Third Annual Conference (1998), Morgan Kaufmann publishers, 193–201
8. Luke, S.: When Short Runs Beat Long Runs. In Proceedings of the Genetic and Evolutionary Computation Conference GECCO'01, (2001), Morgan Kaufmann publishers, 74–80
9. Luke, S., Balan, G.C., Panait, L.: Population Implosion in Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference GECCO'03, (2003), LNCS 2724, Springer-Verlag, 1729–1739
10. Alba, E.; Tomassini, M.: Parallelism and Evolutionary Algorithms. Evolutionary Computation, IEEE Transactions on Volume **6**, Issue 5, (October 2002) 443–462
11. Van Veldhuizen, D.A., Zydallis, J.B., Lamont, G.B.: Considerations in Engineering Parallel Multiobjective Evolutionary Algorithms. Evolutionary Computation, IEEE Transactions on Volume **7**, Issue 2 (April 2003) 144–173