Computing and Informatics, Vol. 23, 2004, 377–394

THE EFFECTS OF TRANSFER OF GLOBAL IMPROVEMENTS IN GENETIC PROGRAMMING

Ricardo Aler, David Camacho

Univ. Carlos III de Madrid
Avda. Universidad
30 28911 Leganés (Madrid), Spain
e-mail: aler@inf.uc3m.es, dcamacho@ia.uc3m.es

Alfredo Moscardini

School of Computing
University of Sunderland
St. Peter Campus
Sunderland, UK
e-mail: a.moscardini@sunderland.ac.uk

Manuscript received 22 February 2002; revised 12 November 2004
Communicated by Burkhard Monien

Abstract. Koza has shown how Automatically Defined Functions (ADFs) can reduce computational effort in the genetic programming paradigm. In Koza's Automatically Defined Functions, as well as in standard genetic programming, an improvement in a part of a program (an ADF or a main body) can only be transferred to other individuals in the population via crossover. In this article, we consider whether it is a good idea to transfer immediately improvements found by a single individual to other individuals in the population. A system that implements this idea has been proposed and tested for the even-5-parity, even-6-parity, and even-10-parity problems. Results are very encouraging: computational effort is reduced (compared to Koza's ADFs) and the system seems to be less prone to early stagnation. Also, as evolution occurs in separate populations, our approach permits to parallelize genetic programming in another different way.

Keywords: Automatically defined functions, cultural evolution, co-evolution, genetic programming

1 INTRODUCTION

In [10], Koza showed how "automatically defined functions enable genetic programming to solve a variety of problems in a way that can be interpreted as a decomposition of a problem into subproblems, a solving of the subproblems, and an assembly of the solutions to the subproblems into a solution to the overall problem". Also, he showed that "For a variety of problems, genetic programming requires less computational effort to solve a problem with automatically defined functions than without them, provided the difficulty of the problem is above a certain relatively low problem-specific breakeven point for computational effort."

With Koza's ADFs, each individual consists of both the main body of the program and of all its subroutines. An improvement in a subroutine (or a main body) of an individual can only be transferred to another individual via crossover between both individuals. It would seem that if an individual can immediately use improvements obtained by other individuals of the population, then the rate of discovery would be faster. An example of this is science itself, where discoveries by one scientist are quickly transferred to other scientists, leads to fast advance. If discoveries were transferred only by word of mouth, advance would be much slower. However, in the case of genetic programming, it is not obvious that transfer of discoveries (pieces of code, or subtrees) will be beneficial. For instance, it might happen that an individual cannot use another individual "discovery" (for instance, a subtree) because their structures are too different. Actually, this is true even for crossover. It is well known that exchange of subtrees by crossover is frequently harmful [6]. In the more extreme case where new discoveries are supplied to all the individuals in the population, something could be an improvement for an individual but a hindrance to another and therefore, former good individuals would become instantly bad individuals. What would be the net effect of both tendencies is unclear, although one would expect that immediate transfer of global improvements would be generally harmful. Thus, the aim of this article is to start exploring empirically this matter: what would happen if improvements in a subroutine (or a program's main body) would be transferred immediately to all individuals of the population? The reader should be aware that by *improvements* we mean *global* improvements, that is, improvements that lead to a change in the best of population.

This article has been structured as follows. Section 2 gives a background on genetic programming and automatically defined functions, and motivates the need for evolving subroutine automatically. Section 3 describes our approach to transferring discoveries between populations. Section 4 contains the results obtained for the well-known even-n-parity family of problems and Section 5 shows experimentally

what happens during a transfer. Finally, Sections 6 and 7 discuss the related work, summarize the conclusions, and describe the future lines of work.

2 GENETIC PROGRAMMING AND AUTOMATICALLY DEFINED FUNCTIONS

In this section, we will introduce the field of Genetic Programming (GP) [8], and motivate the need for Automatic Defined Functions [9].

Genetic programming is an evolutionary technique designed to generate programs automatically. It has three main elements:

- A population of individuals. In this case, the individuals are computer programs. They are usually represented as parse trees, made of functions (with arguments), and terminals (with no arguments: constants). The initial population is made of individuals generated randomly.
- A fitness function. It is used to measure the goodness of the computer program represented by the individual. Usually, this is done by running the individual many times, using many (input, output) cases, also known as fitness cases. The fitness of the individual is then computed by taking into account how many times the output of the program guesses the expected output.
- A set of genetic operators. In GP, the basic operators are reproduction, mutation, and crossover. Reproduction does not change the individual. Mutation changes a function, a terminal, or a subtree; and crossover exchanges two subtrees from two parent individuals, thereby combining characterictics from both of them into the offspring.

The GP algorithm enters into a cycle of fitness evaluation and genetic operator application, producing consecutive generations of populations of computer programs, until a good enough individual is found. In terms of classical search, GP is a kind of bema search, the heuristic being the fitness function. GP has many parameters, the most important ones being the size of the population (M) and the maximum number of generatiosn (G). Also, every genetic operator has a probability of being applied. The crossover operator is usually the most likely one to be used.

When building computer programs, programmers rarely write code by using only the primitive functions and terminals the language provides. Rather, they write parameterized chunks of code, called functions, procedures, or subroutines, that are likely to be used in different parts of the program, and that simplify their structure. Basically, these functions and subroutines act as the building blocks that facilitate programming in a particular domain. Based on this insight, Koza devised a new GP paradigm called Automatically Defined Functions (ADF) [9]. Within this paradigm, every individual has several parts: one main program and several subroutines or ADFs. The crossover operator has been modified, so that an ADF from an individual can be recombined with another ADF from another individual (and not with a main program, for instance). It was found experimentally that,

for complex problems, the computational effort required to evolve correct computer programs is reduced by using ADFs. This is because GP finds the proper low-level building blocks (the subroutines) appropriate to solve the problem at hand. Also, the structural complexity of the final individuals (their size) is also smaller. The reason is that ADFs allow to reuse code in different parts of the program, just as programmers do.

In the same line, Spector experimented with Automatically Defined Macros with similar results [13]. In the ADF and ADM approach, each individual has its own ADFs. The only way a useful subroutine discovered by one individual can be transmitted to other individuals is by means of the crossover operator: from parents to offspring, under several generations.

Other ways of creating ADFs have also been researched. In [5], pieces of code are extracted from individuals and stored in a global library, called GLIB, and become primitives that can be used by any individual. The only way these new primitives can be used by other individuals is by crossover and mutation. In [16] a similar approach is followed, although in this case subroutines compete according to their fitness. ARL (Adaptive Representations Learning) is another approach that selects small building blocks from frequently used good individuals [12]. These subroutines are parameterized and stored in a global library. From time to time, a percentage of individuals is killed and new ones are generated, so that functions in the library have a chance of being used. Subroutines are deleted from the library if their fitness is too low. However, once code is put in the library, it stops evolving, even though it might not contain optimized code.

3 TRANSFERRING IMPROVEMENTS BETWEEN POPULATIONS

In this section, we will describe our approach, which consists in having separate populations for the main programs (or main program bodies) and the subroutines, instead of complete individuals carrying their own subroutines. A population will transfer a piece of code to another when it discovers something having a higher fitness than the previous best piece of code. This scheme allows to transfer improvements to all individuals in the other populations as soon as they are discovered.

Let us suppose that our individuals consist of just two parts: a main body and one ADF (ADF0). As shown in Figure 1, our implementation divides the population of individuals into two separated populations: one for program's main bodies (called main population) and the other one for ADFs (named ADF population). Both populations will evolve independently. Each population will supply the best individual obtained so far to the other population so that individuals of the other population can be evaluated. More specifically, the main population will supply the best main body obtained so far to the ADF population. Likewise, the ADF population will supply the best ADF0 obtained so far to the main population.

In order to evaluate a member (a main body or main program) of the main population, it will be coupled with the best ADF0 supplied from the ADF population,

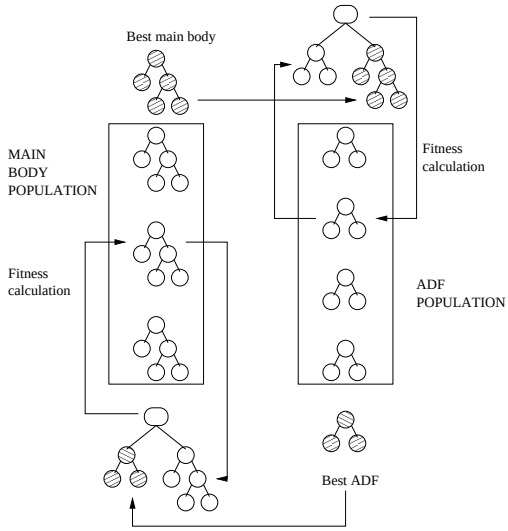


Fig. 1. Inter-relations between the main body population and the ADF population

a whole individual will be built, and the fitness obtained by that individual will be assigned to the main body being evaluated. Similarly, in order to evaluate an ADF in the ADF population, the ADF will be coupled with the best main body supplied by the main population and the resulting individual will be evaluated. Of course, it is impossible to evaluate an individual in a population until a best individual has been obtained in the other population. But this is also true for individuals in the other population. As the process must start somewhere, at the beginning of the run a randomly chosen individual from each population is designated as the best of that population.

Therefore, all individuals in the main population will be coupled and evaluated with the same ADF (the best one obtained so far), and vice versa. Thus, an improvement found by the main population will be immediately transferred to all ADFs in the ADF population (and vice versa).

Populations are evaluated sequentially: if we let an n population system run for 150 generations, P_0 will be run at generation 0, P_1 will be run at generation 1 and so on. P_0 will be run again at generation n, P_1 at generation $n+1$ and so forth. Therefore, each population P_i will run for $150/n$ generations, interleaved with the rest of the populations. Table 1 shows the algorithm we have used for this article.

4 EXPERIMENTAL RESULTS

The approach shown in Figure 1 has been tested with the even-5-parity, even-6-parity, and even-10-parity problems, described in [10]. Table 2 shows the tableau for the even-5-parity problem problem with ADFs (the 6 and 10 EVEN-N-PARITY

1. For each i, create population P_i and choose randomly a best-of-run B_i.
2. Do until the number of generations is exhausted or success.
   (a) run GP on P_0 for 1 generation and update B_0. Stop if success.
   (b) run GP on P_1 for 1 generation and update B_1. Stop if success.
   (c) ...
   (d) run GP on P_{n-1} for 1 generation and update B_{n-1}. Stop if success.

Table 1. Basic algorithm

problems are similar). In this case, we have used two ADFs with three arguments and one main body.

Objective	Find a program that produces the value for the Boolean even 5-parity function as its output when given the values of the tree independent Boolean variables as its input.
Architecture	One result-producing branch and two two-argument functions-defining branches, with ADF1 hierarchically referring to ADF0.
Parameters	Branch typing.
Terminal set for the result-producing branch:	D0, D1, D2, D3, D4
Function set for the result-producing branch:	ADF0, ADF1, AND, OR, NAND, and NOR
Terminal set for the function-defining branch ADF0:	ARG0, ARG1, and ARG2
Function set for the function-defining branch ADF0:	AND, OR, NAND, and NOR.
Terminal set for the function-defining branch ADF1:	ARG0, ARG1, and ARG2
Function set for the function-defining branch ADF1:	AND, OR, NAND, NOR, and ADF0 (hierarchical reference to ADF0 by ADF1).
Fitness cases	All $2^5 = 32$ combinations of the five Boolean arguments D0, D1, D2, D3, D4.
Raw fitness	The number of fitness cases for which the value returned by the program equals the correct value of the even-5-parity function.
Standardized fitness	The standardized fitness of a program is the sum, over the 32 fitness cases, of the Hamming distance (error) between the value returned by the program and the correct value of the Boolean even-5-parity function.
Hits	Same as raw fitness.
Wrapper	None.
Parameters	$M = 200$, $G = 150$
Success predicate	A program scores the maximum number of hits

Table 2. Tableau with ADFs for the even-5-parity problem

This tableau is similar to Koza's, but for M and G. M is the size of the population and G is the number of generations. Koza's M is 16000 whereas we

use a much smaller population size of 200 for even-5-parity, 400 for even-6-parity, and 1000 for even-10-parity. We have done so, so that many more experiments can be run in order to get better statistical estimates. Each configuration was run 200 times. Besides, as we wanted to explore the behaviour of the system for long runs, our G has been extended to 150 (being Koza's $G = 51$). As we use different parameters, we performed a series of experiments for Koza's ADFs as well, so that it can be compared to our system. From now on, Koza's ADF results will be referred to as *Kadf* and our system results as *Iadf* ("Independent ADFs").

As Koza states in [10], a good way to determine how well an adaptive system performs (for a given problem and chosen parameters) is to obtain the computational effort (E) for that problem. In order to estimate computational effort, it is necessary to estimate first the probability that a run with population M yields, at least, a solution by evaluation i or before. This is called the cumulative probability $P(M, i)$. From this, the number of runs required to obtain a solution by generation i (or before) with probability z $R(M, i, z)$ can be calculated. The computational effort E is then equal to the number of individuals that have to be evaluated in order to obtain a solution by generation i, or before.

$$I(M, i, z) = R(M, i, z) \times M \times i \tag{1}$$

As M and z remain fixed, I is a function of i. In order to estimate how difficult it is for a system to obtain a solution, the minimum I value is obtained. Therefore, the computational effort a system needs to solve a problem (with probability z) is:

$$E = \min_{i=0,\ldots,G}(I(M, i, z)). \tag{2}$$

The generation at which this minimum computational effort is realized is called i^*.

Computational effort and related data for both *Kadf* and *Iadf* are shown in Table 3. Graphs displaying computational effort per generation are shown in Figures 2 and 3. Figures 4 and 5 show the cumulative probabilities of solving even-5-parity and even-6-parity, respectively.

	EVEN-5-PARITY		EVEN-6-PARITY	
	Kadf	*Iadf*	*Kadf*	*Iadf*
Population size	200	200	400	400
Computational Effort	428400	365800	952000	704000
Best generation i^*	33	30	34	54
E_{Kadf}/E_{Iadf}	1.17		1.35	

Table 3. Computational effort results (and related data) for *Kadf* and *Iadf*

It turns out that *Iadf* performs slightly better than *Kadf* (see Table 3) for the even-5-parity problem (being the effort ratio $E = 1.17$) and somewhat better for

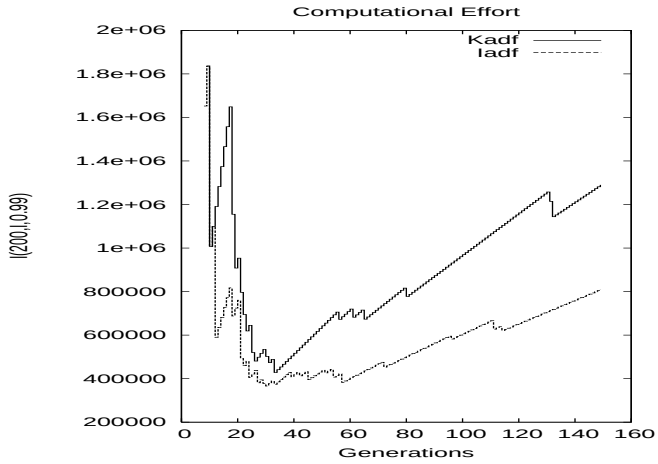


Fig. 2. Computational effort (I) for both *Kadf* and *Iadf*, given that even-5-parity should be solved by generation i with probability $z = 0.99$

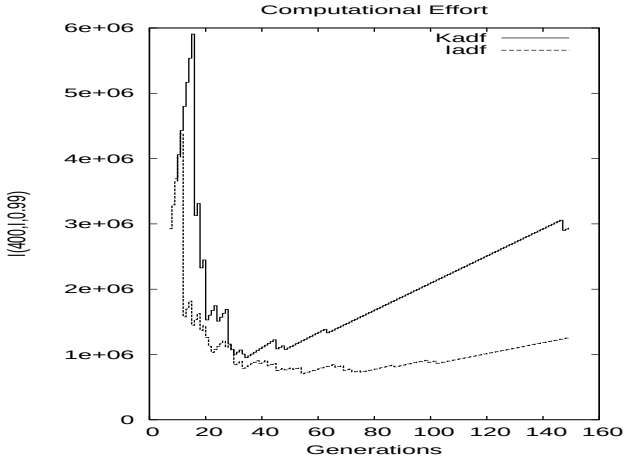


Fig. 3. Computational effort(I) for both *Kadf* and *Iadf*, given that even-6-parity should be solved by generation i with probability $z = 0.99$

the more complex even-6-parity problem ($E = 1.35$).[1] However, that is the effort that the system would have spent had we chosen $G = i^*$. But i^* is not a datum we can know a priori. Had we started our runs without this knowledge, we could have chosen any other G and spent a different computational effort I. In order to have a better picture of what happens for different values of G, graphs displaying computational effort are shown in Figures 2 and 3. Also, Figures 4 and 5 show

[1] Koza, using a population of 16000 individuals, reported a computational effort of 464000 for even-5-parity and of 1344000 for even-6-parity (with ADFs) [9]

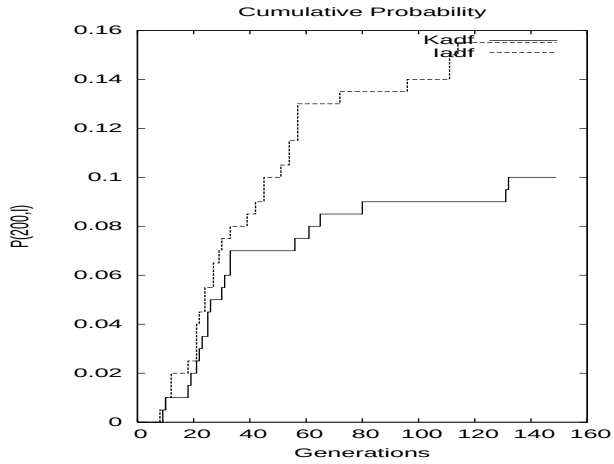


Fig. 4. Cumulative probability of solving even-5-parity by generation i with $M = 200$ for both *Kadf* and *Iadf*

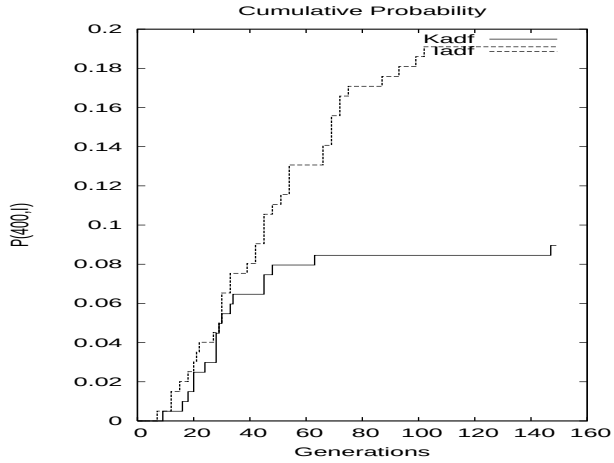


Fig. 5. Cumulative probability of solving even-6-parity by generation i with $M = 400$ for both *Kadf* and *Iadf*

the cumulative probabilities of solving even-5-parity and even-6-parity, respectively. Results in these graphs can be easily summarized:

- *Iadf* has a smaller computational effort than *Kadf* for all generations and especially for the latest generations (see Figure 2). This fact is even more noticeable for the more difficult problem (even-6-parity) (see Figure 3).
- *Iadf* manages to keep a steady rate of improvement (in terms of cumulative probability of success) for longer than *Kadf* (see Figure 4). *Kadf*'s rate diminishes by generation 60 whilst *Iadf* continues improving at a good pace for much

longer. Again, this is even more noticeable in the even-6-parity problem (see Figure 5).

In order to test how our system behaves for higher order parity functions, we used the even-10-parity problem. We ran this problem 200 times with a population size of 1000 individuals. *Kadf* did not find any solution, and *Iadf* found just one, However, given that so few solutions were found, it is not possible to compute the computational effort, because the probability of success cannot be estimated with accuracy. In [9], Koza reported solving the even-10-parity problem with ADF's with a population of 16000 individuals, but he was unable to estimate the computational effort for the same reason. In order to compare *Kadf* and *Iadf* in this problem, we will follow a different approach that uses the fitness of every best individual obtained in the last generation (the 50th) for every one of the 200 runs. Given these 200 runs, Figure 6 displays the probability of obtaining an individual with fitness x, or better, after 50 generations. It has to be taken into account that in standard GP, the goal is to minimize the fitness value. So, small fitness means a better individual. That is why smaller fitness values are less probable in Figure 6: it is more difficult to find good individuals (small fitness) than bad ones (large fitness). We can observe that the probabilities for *Iadf* are always larger than for *Kadf*, which means that *Iadf* will find better individuals more likely. The median value for *Kadf* is 488.0, whereas for *Iadf* is 416.0.

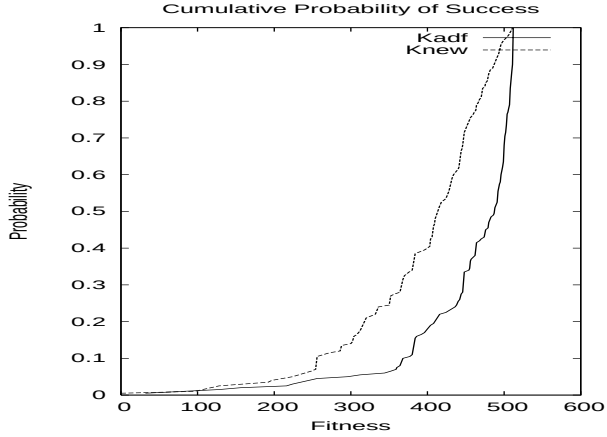


Fig. 6. Probability of obtaining an individual with fitness x, or better, after 50 generations, for the even-10-parity problem

Results seem positive, but it is not clear why transfer of the best individual of a population does not usually harm individuals of the other populations. Next subsection tries to deepen our understanding of what is going on during transfers.

5 BEFORE AND AFTER THE TRANSFER

In order to understand better what happens when a population transfers its best individual to another, we have re-executed all the successful runs for the even-5-parity configuration. It is in successful runs where the effects of transfer, if any, will be seen more clearly. In order to observe the changes, we measured the fitness of the populations just before the transfer happened, and just after the transfer. Let us remember that a transfer occurs when one population finds a better individual than its best so far. All re-runs followed the same pattern. A sample run can be seen in Table 4 where transfers from population ADF0 are observed.

Generation	% Worse	% Equal	% Better	Average Improvement	Gain in Best
1	9	43	48	0.455	2.0
4	31	16	54	0.445	-1.0
10	53	7	40	0.33	2.0
13	26	24	50	1.78	0.0
16	21	44	35	1.23	0.0
19	27	4	70	2.87	4.0

Table 4. Sequence of transfers from ADF0 in a sample successful run

The first column of Table 4 is the generation at which the trasnfer happened. It is noticeable that transfer happens only a few times. This is very usual in the other runs as well.

Colums "Worse", "Equal" and "Better" of Table 4 display the percentage of individuals that worsened, remained equal, or improved their fitness after the transfer. We can see that although many individuals improve after the transfer, many are worse off as well. Usually, in most of the transfers, more individuals improve their fitness, compared to those that worsen it. In order to know what the net effect is, the "Average Improvement" for all individuals in the population is also displayed. It can be observed that it is usually positive but small. This is true of most of the runs we sampled. Therefore, it seems that, in general, transfer do have a small positive net effect on the whole population.

However, the most interesting result is displayed in the last column "Gain in Best". This is the increase (or decrease) in fitness for the best individuals found before and after the transfer. Here, a positive amount means that the population contains a better individual after the transfer than the previous best. That is, a real advance happened. A negative value means that the best individual was lost and replaced by a worse one: knowledge was lost. It can be seen that most values are positive, some are 0, and one is negative. One of the positive values is quite large (the 4.0 increase, for instance. The maximum improvement for even-5-parity is 32.0 points). This pattern is followed, to some extent, in all the successful runs. As this is the most important measure (it measures genuine advances due to transfers), we will quantify it further.

Figure 7 displays the frequency of fitness improvements over all actual transfers of all runs from the ADF0, ADF1, and Main Program populations, respectively. For instance, the point at (1.0, 0.36) means that 36 % of transfers from population Main Program improve the best individual in the population by 1.0 points of fitness.

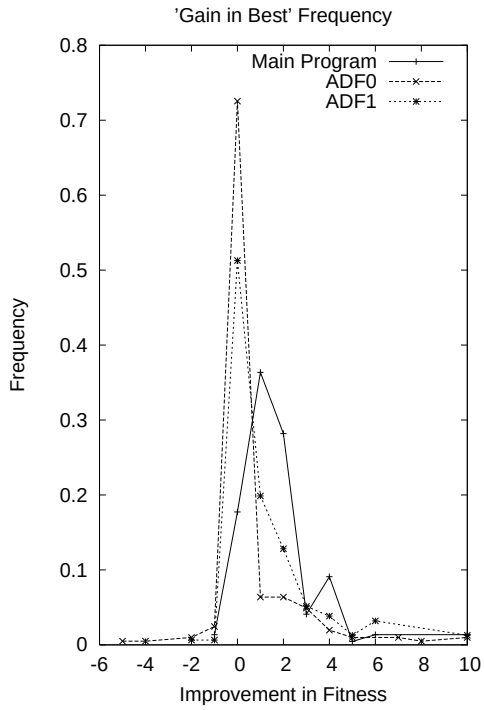


Fig. 7. Frequency (y-axis) of fitness improvements (x-axis) from transfers from the ADF0, ADF1, and Main Program populations, respectively

The first remark about Figure 7 is that very few of the transfers worsen the best individual (i.e. negative values in the x-axis). For instance, the probability of worsening one point of fitness because of transfers from the Main Program population is 2 %. The probability of worsening 6 points is 0.5 %. The second remark is that for transfers from populations ADF0 and ADF1, the most frequent outcome is no change (0 fitness points) in the best individual. For transfers from Main Program, the most likely outcome (36 %) is improving the best individual by 1 point. Yet, there is a good probability that large changes (greater or equal to 2 fitness points) will happen. Even very large changes of 10 fitness points can happen (about 1 % probability in the three cases).

In summary, very few transfers cause harm (to the best individual found so far), most do nothing, and a fair amount of them are genuine discoveries.

We now will show a couple of instances where an improvement is transferred. The first one is a transfer from the ADF0 population. The discovery by the ADF0 population was:

```
(OR (AND ARG1 ARG2)
    (NOR ARG2
         (AND (OR (NOR ARG0 ARG2)
              (OR ARG2 ARG1))
         (NAND (NAND ARG2 ARG1)
               (NOR ARG2 ARG1)))))
```

which after simplification is logically equivalent to:

```
(NOT (XOR ARG1 ARG2))
```

It makes a lot of sense that this subroutine was considered a discovery worth transferring, because this ADF0 actually solves the even-2-parity problem, and can be used to solve the larger even-5-parity problem. This ADF0 is true when an even number of bits in ARG1 and ARG2 are 1 (ARG0 is not used). Table 5 shows the sequence of transfers from fhe ADF0 population. The transfer at generation 13 is the individual just reported. It is remarkable that, although at generation 13 the "gain in best" after the transfer is 0, many individuals benefit from it (58 %). Also, the "average improvement" among the individuals of the population is quite large (5.0 fitness points). So this ADF0 was a worthy discovery.

Generation	% Worse	% Equal	% Better	Average Improvement	Gain in Best
1	16	66	18	0.02	1.0
4	18	54	28	0.2	0.0
13	22	20	58	5.0	0.0

Table 5. Transfers from ADF0 in a successful run

In this same run, another very interesting subroutine was found by population ADF1. Table 6 shows the sequence of ADF1 transfers. The one worth mentioning is that of generation 11, that has a "gain in best" of 10 fitness points, a very large amount, that benefited 58 % of the receiving individuals.

Generation	% Worse	% Equal	% Better	Average Improvement	Gain in Best
2	13	72	14	0.015	2.0
11	28	14	58	1.15	10.0

Table 6. Transfers from ADF1 in a successful run

The ADF1 individual being transferred was:

```
(ADF0 (ADF0 (ADF0 ARG0 (OR ARG0 ARG2)
                  (OR ARG0 ARG2))
            (NOR (AND (OR (NAND ARG2 ARG1) ARG1) ARG1)
                 (ADF0 ARG1 ARG2 ARG1))
            (NAND (AND ARG2 ARG0)
                  (ADF0 ARG2 ARG2 ARG1)))
      (NOR (NAND (NOR ARG1 ARG1) (OR ARG0 ARG2))
           (NAND (NOR ARG0 ARG1) (NAND ARG2 ARG2)))
      (NAND (NAND (AND ARG2 ARG2) (ADF0 ARG0 ARG1 ARG0))
            (ADF0 (NOR (OR ARG0 ARG2) ARG2)
                  (NOR ARG0 ARG0)
                  (ADF0 ARG1 ARG2 ARG1))))
```

This seemingly complex individual, when simplified, reduces to:

```
(NOT (XOR ARG0 ARG1 ARG2))
```

which solves the even-3-parity problem. Hence, it is a good subroutine to be used by the main program, in order to solve the even-5-parity problem. Actually, the program that uses this ADF1 is a perfect individual.

We have checked other transfers at different runs. Not always subroutines can be understood so easily, but in all cases, the transferring individual implied some sort of improvement for the receiving population.

6 RELATED WORK

Iadf originated in an idea that emerged from previous research. In [4], it was indirectly shown how fixing part of a program and letting the rest evolve could be an interesting way for a programmer to introduce background knowledge into GP and to reduce the search space. This article is an offshoot of that idea, although it is an evolving population best individual (instead of the programmer) which fixes part of the program for the rest of the evolving populations. This idea was initially explored in [2] and has been extended for the present article.

The system we have studied in this paper can be considered as an extreme case of co-evolution [7], albeit a strange one, because interaction between populations happens only through the best individual of each population. Co-evolution of a main program and several independent ADF populations has already been dealt with in [1]. Both approaches differ in perspective, though: we are more interested in the simultaneous transfer of information from one population to all individuals in the other populations than in studying general ADF co-evolution. In their approach, in order to evaluate a main program, ADF individuals are selected from the ADF sub-populations. They tested several selection policies, being "the best individual" policy very similar to our own approach. However, in their work this policy does

not fare well compared to GP+ADF, which is the opposite of the results obtained in our paper. Other differences are that we use a generational model for all evolving populations instead of a steady-state model and that we favor program-level fitness evaluation instead of evaluating directly the individuals in the ADF sub-populations. Finally, our results are in terms of computational effort to solve the problem rather than of average results per generation, as in their case [11] proposes a scheme consisting of co-evolving populations of subroutines, but structured in a hierarchy. No complete experimental study has been carried out, but there are some initial results in [3]. The idea of transfer is not explicitly studied in any of these coevolutionary approaches.

A related work reports the use of shared memory (initially proposed in [15]) between all the members of a population [14]. That is, they use a global memory as a form of culture, to transfer data to all individuals in a population, with positive results. Although the aim is similar to ours, the mechanism used in both cases is different. In our case, what is transferred is not data but actual pieces of code.

7 CONCLUSIONS

This paper started by posing the question of whether it would be useful that improvements in a part of an individual (a subroutine, for instance) would be transferred to all members of the population as soon as they were found. We then proposed a system to test this idea and utilized it for the even-5, 6, 10-parity problems. A comparison of our results with Koza's ADF applied to the same problem shows that performance (in terms of minimum computational effort) is better. Thus, there seems to be an advantage by immediately transferring improvements to all individuals, at least in this case. A possible explanation of this behaviour is that populations get specialised in some task, and therefore, it is easier for the rest of the populations to accept their findings than if they were arbitrary improvements. We have also analysed and quantified the effect of transfers, and found out that transfers usually help the individuals of the receiving population on average, and that in some cases, the receiving population discovers better individuals than the best so far. That is, a real advance was achieved. It must be stressed that this happens just because of the transfer, and that no genetic operators are involved.

Our system shows a curious effect: the cumulative probability of success keeps increasing at a good rate for longer than *Kadf*. That is, it doesn't seem to stagnate as soon as GP (or GP+ADF) does. The approach scales well, obtaining better results for the more complex problem than for the simpler one.

Our approach is another way to parallelize GP, with the advantage that communication between populations happens at a very small rate: all the information populations need to exchange is the best individual obtained so far, which changes rather slowly. This kind of parallelism would be useful for problems requiring many different ADFs.

Finally, we would like to determine under what conditions and for what problems our "improvement transfer" approach works. Usually, new learning algorithms are tested in two or three problems, but this often gives no prediction about what kind of problems the algorithm is actually able to solve. After all, we already know that no learning algorithm can do well in all situations (the so-called No Free Lunch theorems) and therefore what should be done is to determine the subset of problems the algorithm biases are appropriate for [19, 18]. This is very rarely attempted and usually considered a difficult task (but see [17]). At least, the empirical study of our paper shows that, contrary to our initial intuitions, improvement transfers do not harm most of the individuals of the receiving population, as it has been shown in the even-n-parity problems.

REFERENCES

[1] Ahluwalia, M.—Bull, L.—Fogarty, T. C.: Co-Evolving Functions in Genetic Programming: A Comparison in ADF Selection Strategies. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, Genetic Programming 1997: Proceedings of the Second Annual Conference, pp. 3–8, Stanford University, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[2] Aler, R.: Immediate Transference of Global Improvements to All Individuals in a Population in Genetic Programming Compared to Automatically Defined Functions for the Even-5-Parity Problem. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, Proceedings of the First European Workshop on Genetic Programming, Vol. 1391 of LNCS, pp. 60–70, Paris, 14–15 April 1998. Springer-Verlag.

[3] Aler, R.—Blázquez, F.—Camacho, D.: Experimentación en Programación Multinivel. Revista Iberoamericana de Inteligencia Artificial, Vol. 13, 2001, pp. 10–22.

[4] Aler, R.—Borrajo, D.—Isasi, P.: Evolving Heuristics for Planning. In V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, Proceedings of the Seventh Annual Conference on Evolutionary Programming, Lecture Notes in Computer Science 1447, pp. 745–754, San Diego, CA, March 1998. Springer-Verlag.

[5] Angeline, P. J.—Pollack, J. B.: The Evolutionary Induction of Subroutines. In Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[6] Banzhaf, W.—Nordin, P.—Keller, R. E.—Francone, F. D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag, January 1998.

[7] Hillis, W. D.: Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, Artificial Life II, Volume X of Santa Fe Institute Studies in the Sciences of Complexity, pp. 313–324. Addison-Wesley, Santa Fe Institute, New Mexico, USA, February 1990.

[8] Koza, J. R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, 1992.

[9] Koza, J. R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts, May 1994.

[10] Koza, J. R.: Genetic Programming II. The MIT Press, 1994.

[11] Racine, A.—Schoenauer, M.—Dague, Ph.: A Dynamic Lattice to Evolve Hierarchically Shared Subroutines: DL'GP. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, Proceedings of the First European Workshop on Genetic Programming, Vol. 1391 of LNCS, pp. 220–232, Paris, 14–15 April 1998. Springer-Verlag.

[12] Rosca, J.: Towards Automatic Discovery of Building Blocks in Genetic Programming. In E. V. Siegel and J. R. Koza, editors, Working Notes for the AAAI Symposium on Genetic Programming, pp. 78–85, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[13] Spector, L.: Evolving Control Structures with Automatically Defined Macros. In E. V. Siegel and J. R. Koza, editors, Working Notes for the AAAI Symposium on Genetic Programming, pp. 99–105, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[14] Spector, L.—Luke, S.: Cultural Transmission of Information in Genetic Programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pp. 209–214, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[15] Teller, A.: Turing Completeness in the Language of Genetic Programming with Indexed Memory. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Vol. 1, pp. 136–141, Orlando, Florida, USA, 27–29 June 1994. IEEE Press.

[16] Teller, A.—Veloso, M.: PADO: A New Learning Architecture for Object Recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, Symbolic Visual Learning, pp. 81–116. Oxford University Press, 1996.

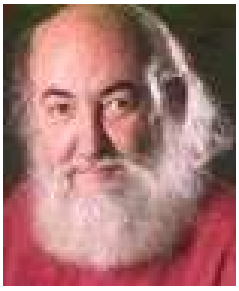
[17] Thornton, Ch.: There is no Free Lunch but the Starter is Cheap: Generalisation from First Principles. Technical Report CSRP 505, School of Cognitive and Computing Sciences. University of Sussex, 1999.

[18] Wolpert, D. H.: The Existence of a Priori Distinctions Between Learning Algorithms. Neural Computation, Vol. 8, 1996, No. 7, pp. 1391–1420.

[19] Wolpert, D. H.: The Lack of a Priori Distinctions Between Learning Algorithms. Neural Computation, David H. Vol. 8, 1996, No. 7, pp. 1341–1390.

Ricardo Aler is associate professor at Universidad Carlos III Computer Science Department. He has researched in several areas, including automatic control knowledge learning, genetic programming, and machine learning. He has also participated in international projects about automatic machine translation and optimising industry processes. He holds a PhD in computer science from Universidad Politécnica de Madrid (Spain) and a MSc in decision support systems for industry from Sunderland University (UK). He graduated in computer science at Universidad Politcnica de Madrid.

David Camacho is an associate professor in the Department of Computer Science at Universidad Carlos III de Madrid. He is member of the Planning and Learning Group (PLG). He holds a PhD in computer science from Universidad Carlos III de Madrid (Spain) in 2001. He has a B.S. in physics science (1994) from the Universidad Complutense de Madrid. He has researched in several areas, including planning, inductive logic programming, and multiagent systems. He has also participated in international projects about automatic machine translation and optimising industry processes.



Alfredo Moscardini has been a professor of mathematical modelling at the University of Sunderland, U.K. for 10 years. His research includes cybernetics, system dynamics and their application to economics. He is also leader of the research group in Decision Support Systems Group at the University. Recently, he has been working with colleagues in the area of neural networks. He is currently leading a computational intelligence team and is responsible for the creation of a new masters course in this area. He has been working for 15 years with universities in Eastern Europe and is responsible for introducing many of these ideas in Bulgaria, Estonia, and the Ukraine.