Evolving Reusable Subroutines for Genetic Programming

Oliver Brock
Computer Science Department
Stanford University, CA 94305
oli@cs.stanford.edu

Abstract

Although automatically defined functions (ADFs) are able to significantly reduce the computational effort required in genetic programming, reasonably difficult problems still require large amounts of computation time. However, every time genetic programming evolves a program to solve a problem those ADFs have to be rediscovered from scratch. If the ADFs of a correct program contain partial solutions that are generally useful, they can be used to solve similar problems. This paper proposes a technique to make the information of successful ADFs accessible to genetic programming in order to reduce the computational costs of solving related problems with less computational effort and demonstrates its utility using the example of the even n-parity function.

1 Introduction

In the paradigm of genetic programming populations of computer programs are altered by operators that resemble genetic operations, like sexual crossover, mutation, inversion, deletion, etc., in order to evolve a program that solves a given task. Programs are much more likely to partake in this evolution if they solve the task better than most of the other programs in the population, thereby imitating the the Darwinian principle of survival of the fittest. This procedure starts out with a set of randomly generated computer programs that consist of *function symbols* chosen form the *function set*, representing the commands of the computer language, and *terminal symbols* chosen from the *terminal set*., containing, for example, constants and the arguments of the program.

Genetic programming has been successfully applied to many different domains (Kinnear 1994, Koza 1992), but more difficult problems require a huge amount of computation time. In (Koza 1994) an attempt was made to overcome this drawback by introducing automatically defined functions (ADFs). This concept allows the programs to use subroutines that themselves are dynamically created by the above described genetic operations. ADFs have been shown to speed up the process of genetic programming considerably, given that the problem at hand is above a certain complexity threshold. Furthermore it decreases the structural complexity of the programs and thereby increases their parsimony. If ADFs are used the programs consists of one or more multiple *result producing branches (RPBs)* and a set of ADFs. The RPBs can call the ADFs as subroutines and ADFs can call each other, provided that the calling scheme is acyclic to avoid infinite recursion. The RPBs and the ADFs are considered to be the *branches* of the program.

As RPBs and ADFs are initially randomly generated, they cannot be expected to represent valuable programs with reguard to the problem. They both have to be reevolved by genetic programming in each run, thereby wasting computational resources. The question arises as to whether these subroutines that are evolved by genetic programming only contain valuable techniques to solve the specific problem at hand in collaboration with a specific RPB or if they can be used advantageously for similar problems. If the latter is true, they could be compiled into a library that would help make the solution of related problems computationally less expensive by allowing evolving programs to access these subroutines. The experiments presented in this paper strongly suggest that this is indeed a practical and fruitful approach.

2 The Setting

In order to verify whether ADFs of a correct program evolved by genetic programming can be used advantageously for related problems, they are included into the function set of a similar, but potentially more difficult problem. If ADFs indeed contain general procedures for problem solving, a significant speedup can be expected. As the exemplary problem in this paper the even n-parity function has been chosen. It has the advantage of easy scaleability and, as shown in (Koza 1994), its complexity increases considerably as n increases. Furthermore, the resulting programs and ADFs can be interpreted conveniently, since the parameter domain is limited.
The conducted experiment consists of two parts:

1. Genetic Programming is applied to the even 4-parity and 5-parity problem. This process is stopped as the first individual solves the parity problem completely and the ADFs of this individual are stored.

2. In the second part of the experiment the ADFs obtained previously are included into the functional sets of both RPBs and ADFs of another attempt to solve the even 4, 5, or 6-parity problem with genetic programming. Those pre-learned ADFs from the first part of the experiment are referred to as preADFs.

For all experiments one RPB with the n arguments `D1, D2, ..., Dn` and a set of two ADFs (`ADF1, ADF2`), with the arguments `ARG1` and `ARG2` were used. In case of the 5-parity and higher order parity functions an additional argument `ARG3` was used for the ADFs. The choice of the number of arguments is motivated by the fact that the 4-parity problem is easily decomposable into 2-parity functions, whereas the 5-parity problem has a simpler

	Experiment 1	**Experiment 2**
Objective	Evolve a program that solves the even n-parity function	Evolve a program that solves the even n-parity function using ADFs from a successful run of Experiment 1
Architecture	One RPB and two ADFs with two arguments each, ADF1 can hierarchically refer to ADF2	One RPB and two ADFs with two arguments each, ADF1 can hierarchically refer to ADF2
RPB: Terminal set	`D1, D2, ..., Dn`	`D1, D2, ..., Dn`
RPB: Function set	`ADF1, ADF2, AND, OR, NAND, NOR`	`ADF1, ADF2, AND, OR, NAND, NOR`
ADF1: Terminal set	`ARG1, ARG2, (ARG3)`	`ARG1, ARG2, (ARG3)`
ADF1: Function set	`ADF2, AND, OR, NAND, NOR`	`ADF2, PREADF1, PREADF2, AND, OR, NAND, NOR`
ADF2: Terminal set	`ARG1, ARG2, (ARG3)`	`ARG1, ARG2, (ARG3)`
ADF2: Function set	`ADF2, AND, OR, NAND, NOR`	`PREADF1, PREADF2, AND, OR, NAND, NOR`

Table 1: The tableau for the experiments.

solution with 3-parity functions. Of course, the 2-parity and 3-parity ADFs – if they ever evolve – would represent the ideal case for our purposes.

Parameter	Setting
Size of population	10,000
Selection	Tournament selection of size 8
$P_{Crossover}$ for leaves	0.1
$P_{Crossover}$ for nodes	0.8
$P_{Mutation}$ for nodes	0.1

Table 2: The parameters for genetic programming.

The logical two-arity functions `AND`, `OR`, `NAND`, and `NOR` were also included into the functional set of all branches. They represent the programming language we allow genetic programming to use. Table 1 summarizes the architecture of the individual programs in both experiments. The parameters for genetic programming are shown in table 2. All experiments presented were done multiple times with different seeds for a Park-Miller randomizer and their results were averaged.

3 Results

3.1 Experiment 1

This experiment was conducted in order to obtain ADFs that will be used for the second experiment, and to provide data to compare the computational effort of genetic programming with automatically defined functions to genetic programming with automatically defined functions augmented by pre-learned ADFs.

In all attempts to learn the 4-parity function genetic programming evolved four different kinds of ADFs, covering all possible boolean functions with two arguments:

1. In almost all of the runs one of the ADFs of the first correct individual turned out to be the odd or even 2-parity function.

2. Another type of ADF, frequently accompanying the first type, was the constant function that for all argument values returned either one or zero. This kind of ADF obviously is not very useful. It was found in most of the programs that contained a 2-parity ADF.

3. A compromise between the two previous types is the mixed 2-parity function, which corresponds to the even 2-parity if one of the arguments is of value zero and to the odd 2-parity if it is of value one.

4. The last variety of ADF that occurred was the half 2-parity function that gave the correct result only if one of the arguments had a specific value and was constant otherwise.

The frequency of occurrence based on over twenty runs is summarized in table 3. The percentages add up to 200%, because each program contains two ADFs. For the eight runs that were conducted for the 5-parity problem with 2-arity ADFs approximately the same distribution could be observed.

	2-parity	constant	mixed 2-parity	half 2-parity
Occurrence in %	85	85	10	20

Table 3: The occurrence of the different types of ADFs.

Based on five runs of the 5-parity problem with 3-arity ADFs more than half of the ADFs were parity functions of order two or three. Various other boolean functions were evolved that also could be classified into mixed parities, as with the 2-arity ADFs, and other boolean functions. The functional diversity of 3-arity boolean function, however, does not make it appear very fruitful to examine their functionality in great detail. As we will see in the following sections, it suffices to differentiate between parity and non-parity functions. Therefore we omit a more detailed discussion.

The amount of computation time that was required to solve a parity problem is measured in fitness evaluations. The figure for the 6-parity problem is a very low estimate based on the rate of convergence of incomplete runs and on results from (Koza 1994). The value for the 4-parity is based on 14 runs and the one for the 5-parity on eight runs.

	4-parity	5-parity	6-parity
Fitness evaluations without preADFs	80,000	450,000	1,500,000

Table 4: The number of fitness evaluations required to solve the different parity problems.

3.2 Experiment 2

The ADFs evolved by genetic programming in the first part of the experiment are now included into the functional set of both the RPB and the ADFs (see table 2). Since the domain of the arguments for this particular problem was very restricted, all preADFs could be compiled into a table of function values. This has a great computational advantage over reevaluating the preADFs for each call. Using the same settings otherwise, the 4-parity, the 5-parity, and the 6-parity problem are solved. In a first set of runs the 2-arity ADFs of a previously evolved 4-parity program were used as preADFs. The results are based on twelve, eight, and three runs for the 4-, 5-, and 6-parity function, respectively. Note that in all runs one of the preADFs was the even or odd 2-parity function and the second one was either a constant function, a mixed, or a half parity function.

	4-parity	5-parity	6-parity
Fitness evaluations without preADFs	80,000	450,000	1,500,000
Fitness evaluations with 2-parity preADF	12,000	85,000	700,000
Fitness evaluations with 3-parity preADF	5,000	40,000	400,000

Table 5: Comparison of the number of fitness evaluations.

In a second set of runs 3-arity ADFs from previously evolved 5-parity programs were used as preADFs for the 4-, 5-, and 6-parity problem. Table 5 lists the results for this experiment that were based on eight, five, and two runs for the different parity problems, respectively. Note that in all of these runs one of the preADFs was a 3-parity function. Whereas in the runs to solve the 4-parity problem the rate of occurrence of 2-parity ADFs was reasonably high to justify this, in runs to solve the 5-parity function lower order parity functions evolve in only about 50% of all individual programs.

Table 5 summarizes the results of those two set of runs and compares the number of fitness evaluations for these experiment to the ones described in the previous section.

Figure 1 presents the results graphically. The 4-parity problem is learned approximately six times faster if a 2-parity function is provided as a preADF. A 3-parity preADF speeds up genetic programming by a factor of sixteen, when solving the 4-parity problem. As expected those factors decrease as the problem becomes increasingly difficult. For the 5-parity problem speedups of factor five and ten were observed (with 2-parity preADF and 3-parity preADF, respectively) and for the 6-parity problem the corresponding values were two and four. For the 4-, 5-, and 6-parity problems an impressive decrease in the computational costs could be observed.

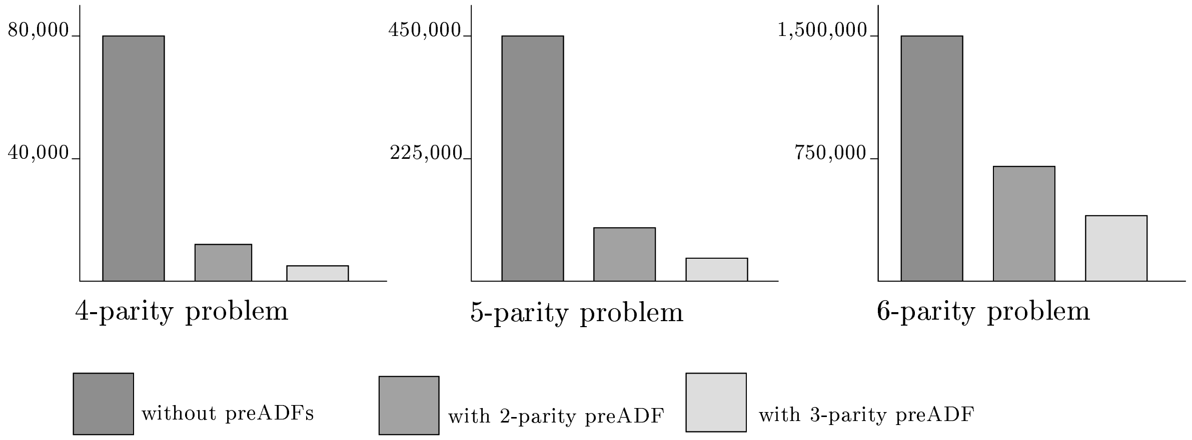

Figure 1: Comparison of the number of fitness evaluations.

Runs in which only preADFs were used that did not correspond to any parity functions mostly converged significantly slower than those that did not use preADFs at all (experiment 1). To save computational resources those runs were interrupted when they reached the maximal value of fitness evaluations for a comparable run without preADFs. At that point the fittest program in the population generally still represented a fairly bad approximation to the problem.

4 Discussion

The results presented in the previous section clearly show that ADFs evolved by genetic programming contain algorithmic structures that can advantageously applied for similar problems. In what follows we will analyze what factors are important to guarantee prelearned ADFs to be beneficial.

The experiments imply that for this particular problem only parity functions are useful as preADFs. Other boolean functions, even those that represent partial solutions to the parity problem cause genetic programming to slow down, if not to fail. The reason for this appears to be that the lower order parity function is the "obvious" solution for genetic programming. If a parity function is evolved as an ADF, this function causes a particular program to have an enormous competitive advantage over others, therefore it is likely to dominate the population. Since it is very probable that the parity function is evolved in a large population, it is expected that in most of the runs parity-ADFs will arise that increase the fitness of its host-RPB. This was confirmed by the experiments in section 3.1.

For a program using a 3-arity preADF representing a 2-parity function by just neglecting one of the arguments, the required computational effort to actually solve the 5-parity problem without taking advantage of this preADF is smaller than the effort to determine which of the arguments to the preADF is unimportant, as can be concluded from the fact that runs with non-parity preADFs have higher computational costs than runs without preADFs (compare section 3.1 and section 3.2). The same holds for all other boolean functions that were evolved during the runs. They are only useful to those RPBs that were evolved to take advantage of their special properties. In general it requires a larger effort to decipher there "hidden wisdom" than to solve the problem from scratch.

It is worth noticing that by using preADFs that did not represent a parity function in programs *without* ADFs, the computational effort over several runs was comparable to the experiments that are from section 3.1 that are summarized in table 4.

The observation that 2-arity ADFs are much more likely to evolve a 2-parity function than 3-arity ADFs suggests that as the size of subproblem solved by an ADF decreases, the probability that this ADF can be used beneficially as a preADF increases. However, the results also imply that with increasing size of the subproblem the required computational effort decreases.

All results discussed so far seem to support the fact that ADFs in order to be beneficial as preADFs have to accommodate or support the most general way of solving the particular problem at hand. Since the number programs that can be evolved by genetic programming to solve a problem is enormous, it would be very useful to find techniques that enforce such generality. The results in section 3.1 imply that this is of crucial importance as the problems get more complicated and the number of arguments to the ADFs increases.

Motivated by this observation experiments with multiple RPBs were conducted in order to ensure the usefulness for higher-arity ADFs. The first RBP had to solve the 4-parity and the second one the 5-parity problem. The rationale was that now the ADFs had to accommodate both RPBs and were therefore forced to be more general in the way they solve the problem. Unfortunately in repeated runs with this architecture genetic programming was not able to evolve a correct program. This can be explained with the results of section 3.1 and 3.2: during the run the ADFs are randomly better suited for one of the RPBs. This will give this particular program an advantage over others, since this part of there result producing branch will have a high fitness. In the later stages of the experiment the second result producing branch will either have to adopt to these ADFs, which is – as discussed above – computationally more expensive than solving the original problem, or will succeed by being provided with new ADFs as a result of a crossover operation, in which case the first RPB would be highly unfit. Weighting the RPBs differently causes the same problem.

By examining a representative program shown in figure 2 that solves the 4-parity problem using two 2-arity ADFs and two preADFs, one of which being the 2-parity function

RPB :

```
(preADF2
  (preADF2
    (preADF2
      (D1)
      (D3))
    (NAND
      (D2)
      (D4)))
  (NOR
    (D4)
    (D2)))
```

ADF1 :

```
(preADF1
  (NOR
    (ARG1)
    1)
  (NAND
    (ARG2)
    (ARG2)))
```

ADF2 :

```
1
```

Figure 2: A solution to the 4-parity problem with an even 2-parity preADFs.

and the other one a constant function, it can be observed that the ADFs are not used at all. This was generally observed during those experiments in which preADFs were included. And indeed, the computational effort was approximately the same when these runs were repeated with programs that had no ADFs but could still access the same preADFs.

In conclusion one can say that the proposed technique is of great benefit if the preADFs are generally useful, as pointed out earlier. This requirement leads to goals for the continuation of this research. Some insight was gained concerning the difficulties for genetic programming to recognize the way preADFs that are not generally useful can be incorporated into a program.

5 Future Work

The experiments in section 3.2 have shown that not all the ADFs can be used profitably as preADFs. Although enough evidence was presented to prove that the concept in general is useful, it is not acceptable that only certain ADFs are chosen, as it was done for the experiments in section 3.2, where at least one parity function was included as a preADF. In order to guarantee that beneficial preADFs are included in each run, several individuals have to be evolved in the first phase that solve the problem and all of their ADFs need to be included as preADFs into the second phase.

There are two diffent methods to accomplish this integration. The first one is simply to make all preADFs accessible to all inidviduals in a population. This method is likely to be less efficient than what was presented in the previous sections. It appears to be better to allow each program access to only a few preADFs. This requires to permanently assign those functions to specific programs. However, if a crossover is performed on two especially fit individuals with different preADFs the result is not likely to be comparably fit. The crossover operation should favor programs with a compatible set of preADFs, but still allow crossover between all programs with lower probability to warrant diversity. This corresponds to subpopulations within the population.

Another area for future work is the development of learning techniques that cause ADFs to be generally useful with a higher probability. However, the experiments conducted with multiple RPBs did not seem very promising.

Furthermore, the technique of using preADFS could be applied to other and potentially more difficult problems, as for example the artificial ant problem (Koza 1992, Koza 1994). This problem appears to be even better suited, since it does not involve the issue of choosing the number of arguments for the ADFs and is more likely to evolve ADFs that will be useful as preADFs.

Applying this technique to problems that only require approximate solutions, as for example function regression, also seems promising, because the ADFs of programs that do not solve the problem perfectly could still represent valuable preADFs.

Not in all problems the entire preADF can be precomputed in order to save computational resources, as done for the parity problem in section 3.2. Therefore it will be unavoidable to simplify preADFs in order to reduce computational costs of their evaluation. There are various methods for this, based on simplification rules depending on the used function set. Another fruitful approach might be to recursively prune subtrees according to their correlation to the final output of the entire function.

6 Conclusion

This paper presented a method to integrate automatically defined functions evolved by genetic programming into other experiments in order to solve closely related problems. The results show that genetic programming is able to discover subproblems of the original problem and to solve those subproblems by evolving appropriate functions. The inclusion of those previously learned subproblems into the function set of a similar problem greatly reduces the computational effort required to find a solution to this new problem.

Acknowledgements

For the experiments presented in this paper a modified version of David Andre's DGPC (Dave's Genetic Programming Code) was used. We would like to thank Dave for making it available: Thanks, Dave!

Bibliography

Kinnear, K.E., Jr. (editor) 1994. *Advances in Genetic Programming.* Cambridge: The MIT Press.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge: The MIT Press.

Koza, J.R. 1994. *Genetic Programming II – Automatic Discovery of Reusable Programs.* Cambridge: The MIT Press.