External Concepts Reuse in Genetic Programming

Grégory Seront
gseront@ulb.ac.be
Département d'Informatique
Université Libre de Bruxelles
CP 212
50, av Fr. Roosevelt
1050 Bruxelles
BELGIUM

Abstract

In this paper we show how concepts synthesised by Genetic Programming to solve a problem can be reused to solve other ones. This aim is achieved by the creation of a concepts library. These concepts can then be injected in a new population in order to solve a problem that needs them. We explore the performance of this approach against the case where the search starts from a totally random population.

1. Introduction

In its latest book, Koza introduced the concept of Automatically Defined Function (ADF) [5]. The goal of ADF is to improve the ability of GP to reuse already discovered features. The properties of ADF and the gains it brings during the search have been studied extensively[3, 5]. But surprisingly, these studies were limited to the effects of internal reuse of the Defined Functions. The reuse of the Defined Functions is termed here as internal, because it is limited to the course of the run. The Functions are reused from generation to generation into different individuals but within the same population. This can be confronted to external reuse where the Functions are reused in another run to solve another problem.

The external reuse might be a good idea. If two problems are close enough, the concepts evolved to solve one can be useful to solve the other one.

For example if we try to evolve creatures able to walk toward some food source, we might first try to evolve a creature able to walk. And then, restart the evolution with the last generation population to make them take the food. This way, the walking concept will not have to be re-evolved.

In this example, the first problem is included in the second since a creature able to walk toward food is also able to walk.

In this paper we will also deal with non included problems. For example the problem of creature walking toward food and the problem of creature walking away from enemies. These problems are not included in each other since neither of them is the solution of the other.

In this paper we show how concepts synthesised by Genetic Programming to solve a problem can be reused to solve other ones. This aim is achieved by the creation of a concepts library. These concepts can then be injected in a new population in order to solve a problem that needs them as it is summarised in figure 1.

In the next section we will explain what we mean by 'concept' and how they can be reused. This will be illustrated by an example where two non included problems lead to the evolution of similar concepts. Next, we will prospect how the information contained in a population can be retrieved for another use and how libraries of concepts are created.

Section 4 explains the experiments that have been conducted.

2. Concepts reuse

2.1 Concepts versus Trees

In this paper, we are talking about concepts reuse rather than functions, individuals or sub-trees reuse. The reason for this is that what we are trying to reuse is not a whole individual or a sub-tree. By 'concept' we mean a collection of informations that are useful for the resolution of a problem.

This has not to be mistaken with the usual meaning of concept in machine learning.

A function or a sub-tree alone may not be enough to form a concept in that sense.

For example, if we look at the concept of Stack. We note that it is constituted by the PUSH, POP and INIT functions. The POP function by itself is not sufficient to form the concept. It is the co-operative use of these three functions that forms the concept of stack.

2.2 Concepts versus Primitives

One of the main criticism toward GP, is that usually the set of primitives contains too much knowledge about the problems to be solved. This way, claim the detractors of GP, the power is not in the search algorithm, but rather in the representation that is too close to the solution.

In addition, the use of too high level primitives may prevent the discovery of the solution, because the level of granularity is not sufficient.

In this paper, we will use minimal primitives set. By solving problems with a minimal set of primitives one can demonstrate the true power of GP.

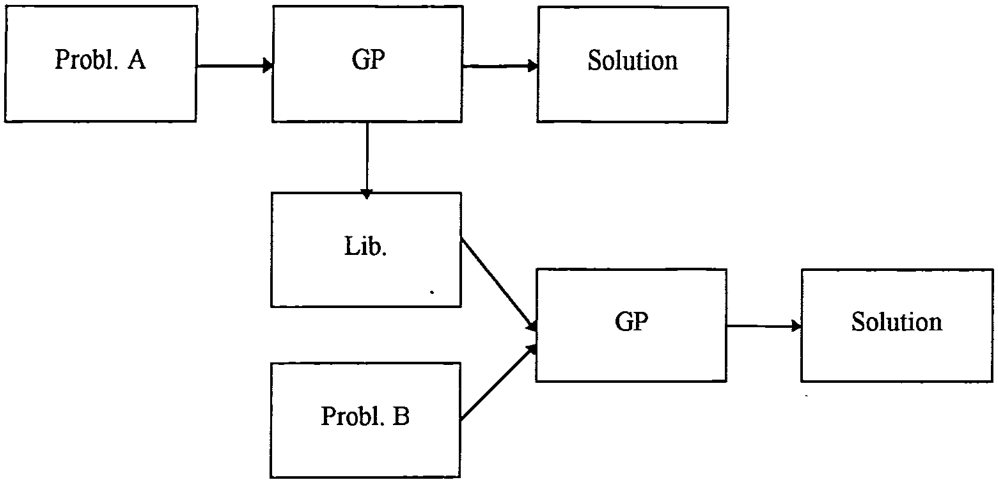

Figure 1 Outline of the Concept Library System

2.3 Independent discovery of the array index concept

In this section, we describe two problems leading to the independent evolution of the same concept: an array index.

2.3.1 Common context

The two problems share the same context: an array of integer of a given size (SizeMem) that must be partially filled with a certain type of information. This vector is initially filled with all '1'.

1	1	1	1	...	1

fig. 2 Initial configuration of the array

The two problems share the same terminals and functions set. This set is minimal in the sense that it is impossible to remove a terminal or a function without making the problem impossible to solve.

+, -	classical arithmetic operators.
while(arg1, arg2)	evaluate arg1 then arg2 while arg1 is > 0
mem(arg1)	returns the integer at the position (|arg1| modulo SizeMem) of the array
assign(arg1, arg2)	assign the value of arg2 to the position (|arg1| modulo SizeMem) in the vector.

fig 3. Common functions set

[-5, ,5]	Integer constants
Length	the number of positions to fill

fig. 4 Common terminals set

To be complete, we must add that 'while' and 'assign' return the value of arg2.

As you can see, the terminals set does not include the notion of variable nor index. These notions will thus have to be evolved.

Those two problems were run with 3 ADF's, with respectively 0, 1 and 2 arguments.

2.3.2 Problem 1: Zeroes

The first problem called 'Zeroes' is the following: to find a program that fills the array, with 'Length' consecutive zeroes, starting from the first position in the vector.

The solution must be general for all possible value of Length.

The fitness is defined by the following formula:

$$Fitness = NbAssign + 3.NbCorrect + \frac{10}{TreeSize} - NbDestroyed$$

Where,

NbAssign is the number of positions different from '1' in the area [0, Length-1]

TreeSize is the size of the Tree.

NbDestroyed is the number of positions different from '1' in the area [Length, SizeMem].

NbCorrect is the number of '0' in the wanted area.

NbCorrect for the 'Grow' problem is the number of positions in the wanted area that are greater than the preceding one (ex: 4, 8).

The figure 5 shows a solution for this problem.

2.3.3 Problem 2: Grow

The second problem called 'Grow' is very close: fill the array of integers with an increasing suite of consecutive numbers of a given 'Length' (ex: 4, 7, 10, 45, 48, ...).
The fitness is the same as it is for 'Zeroes' except that NbCorrect is the number of positions in the wanted area that are greater than the preceding one (ex: 4, 8, 3, 5 gives NbCorrect = 2).
The figure 6 shows a solution for this problem.

2.3.4 Discussion

We see that we have two common concepts evolved: the decrease and the consultation of an array index. This concept is build upon very low level operators.

```
Consultation           Increase
 (mem length)          (= length ADF0)
                       ADF0:
                       (+ (mem length) 1)
```

The usual solution for this problem would have been to put an 'inc' and 'dec' operator.
Teller showed in [6, 7], that the power of GP is greatly improved by the use of indexed memory and that its mastering is of great importance for the future of GP.
By this example, we show that GP is able to evolve the needed concepts for indexed memory treatment from very low level operators.
It is noticeable that the notion of index have been discovered and expressed in both cases in a very similar form.
The fact that close problems often involve the evolution of concepts close syntactically grounds our work on External concepts reuse. If these concepts could be transferred from one problem to another, without having to be rediscovered, considerable time may be gained.

```
RPB:
 (while (ADF2 ADF0 -1)
      (= (mem length) 0))
ADF0:
 (+ (mem length) 1)
ADF1:
 -1
ADF2:
 (= length ADF0)
```

fig.5 a solution for 'Zeroes'

```
RPB:
 (= (ADF1 (while (= -3 ADF0)
             (= ADF0
                (mem 3)))) 3)
ADF0:
 (- (mem (= (mem 3)
         (mem 3))) -1)
ADF1:
 3
ADF2:
 ARG1
```

fig. 6 a solution for 'Grow'

2.4 Concepts library and concepts storing

As mentioned in the previous section, close problems may involve the evolution of similar concepts.
The idea here, is to create a library that would contain those evolved concepts.
In order to create a real library, we must know at which level of granularity we have to pick the concepts up. Obviously, they are included in the population and since in the traditional GP model, there is no co-operative behaviour among the individuals, we can assume that an individual able to solve a problem contains the concepts needed.
As it is impossible to know which parts of the individual contain interesting concepts, the proper granularity for concept storing seems to be the individual.
For the sake of diversity, our library of concepts will be constituted by a collection of individuals.

2.5 Concept sampling

Now that we know how to store them, we have to decide when to pick them up. Should we take them when a totally fit individual has been found, or before?
Intuition tells us that concepts held in a totally fit individual might be overadapted to the problem they are helping to solve, and thus lack some generality.
The question of when and which individual to take remains an open question that will be explored in next papers.
For the moment we will build our library by taking a copy of the whole population when some statistical indexes are reached (see section 4.2).

3. Library Exploitation System

Now that we have our library, we must know how to use it in order to solve other problems.

Various options present themselves. The most obvious one is to use the library to create the initial population 'seeded' with the concepts and thereafter to perform a normal GP run.
We present here two seeding methods (Inject and Shake), and another one that inject the concepts during the run.

3.1. Inject

This method generates the new population by choosing randomly a given number of individuals from the library, and by filling the rest of the population with totally random individuals.
The only parameter for this method is the number of individuals to inject.

3.2 Shake

This method is less direct. The new population is created by performing totally random cross-overs and mutations on the library for a given number of generations. By totally random we mean that the candidate individuals for genetic operations are chosen with uniform probability.
The parameters for this method are the number of generation, and the probability of the different operators.
The rationale behind this methods is that the individuals taken from the library are often more fit to the new problem than the totally random one. So after a few generations, they might overcrowded the random ones, which would lead to a loss of diversity.

3.3. Mutation

Another method would be to start from a random population, and to inject the concepts during the course of the run. The injection could be materialised by a special mutation operator replacing subtrees in the population by subtrees taken from the library.
This method is not tested in this paper.

4. The experiments

4.1 Performance Index

The measure of performance used is the Effort as defined by Koza in [5]. The Effort is the expected number of individuals to evaluate, to have a probability z to find the solution. This effort was computed for z =0.99 by performing 20 runs of the same problem with different random seeds.
Here we consider that we have a solution if the fitness is superior to 580 on values of Length different from the training set. Since the solution is tested on out of sample data, this will insure the generality.

4.2 The problems set

The problems set for the experiments is the one described in section 2.3. Here, we tried to evolve solution for the problem 'Grow' by reusing the concepts evolved during the run of the problem 'Zeroes'.
The number of operations that the program is allowed to execute before the fitness evaluation, is 10*Length.
The raw fitness is the sum of 5 fitnesses obtained by the formula in section 2.3 for 5 different values of Length.

4.3 Results

The experiments were conducted with a population of 500 individuals. The selection method was a tournament of size 7.
The figure 7 shows the results summarized in a bar graph. The first two bars represent respectively the the effort for the problem 'Grow' and 'Zeroes'. The other bars are the efforts for the different the 'Shake' and 'Inject' Library Exploitation Systems with different parameters.
For the 'Shake' method, the parameter is the number of generations of random application of the genetic operators. Shake = 3 means that the genetic operators are applied totally randomly for 3 generations.
For the 'Inject' method, the parameter is the percentage of individuals coming from the library in the initial population.
The first conclusion we can draw from these results is that the « Concept Reuse » works. The effort needed in the worst case to find a solution has been divided by five.
The effect of the different parameters are not clear yet. We can just notice that if the number of generations for 'Shake' is too high, the results become close to those obtained with a totally random initial population.
Further experiments will have to be conducted to determine the most efficient method and to measure the effects of the different parameters. (see Section 6: Future work).

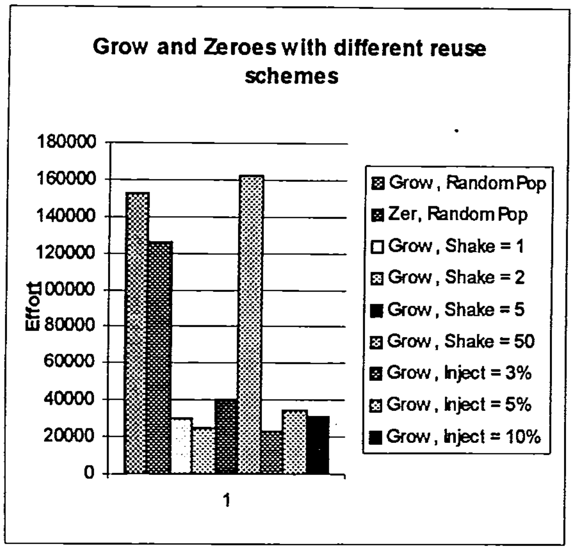

fig. 7 Experiments results

5. Conclusions

In this paper we introduced a new way of using the GP paradigm. In our approach, the problem solving does not have to start from scratch each time. Libraries of concepts can be created and reused just as it is done in the classical programmatic paradigm. In this way, the computation time usually wasted to re-discover concepts previously evolved, can be used to solve more complex problems. Successive layers of increasingly complex concepts libraries can be build from low level operators.
We showed that concepts reuse works and saves a considerable amount of time.

6. Future Work

Several points must still be explored:
a) What is the comportment of concepts reuse on more complicated concepts involving larger trees?
b) Which are the relative performances of the Shake and Inject methods?
c) Which are the effects of the different parameters of these methods?
Those points are under study and will be the subject of further reports.

7. Acknowledgments

I would like to thank Stefan Langerman for those valuable hours of discussion, and Anna Shotton for the spelling corrections.

8. References

[1] Angeline P.J., and Pollack J.B. (1993) « Evolutionnay Module Acquisition », in *Proceeedings of the second Annual Conference on Evolutionnary Programming*, La Jolla, CA: Evolutionnary Programming Society.
[2] Collins R.J., Jefferson D.R.(1992) Antfarm: Toward simulated evolution, in *Artificial Life II, Proceedings of the Workshop on Artificial Life* Addison-Wesly, Reading, MA
[3] Kinear K.E. Jr. (1994) Alternative in Automatic Function Definition in *Advance in Genetic Programming* Cambridge MA: The MIT Press.
[4] Kinear K.E. Jr. (1993) Generality and difficulty in Genetic Programming: Evolving a Sort in *Proceedings of the fifth international conference on Genetic Algorithms*, S. Forest, Ed. San Mateo, CA Morgan Kaufmann.
[5] Koza, J. R. (1994) Genetic Programming II. Cambridge MA: The MIT Press.
[6] Teller A. (1994), The Evolution of Mental Models, in *Advance in Genetic Programming* Cambridge MA: The MIT Press.
[7] Teller A. (1994), Turing Completeness in the Language of Genetic Programming with Indexed Memory, in *Proceedings of The First IEEE Conference On Evolutionary Computation*