Speeding up Genetic Programming

Penousal Machado[a], Amílcar Cardoso[b]

[a]*Instituto Superior de Engenharia de Coimbra*
E-mail: machado@dei.uc.pt
[a,b]*Centro de Informática e Sistemas da Universidade de Coimbra*
Polo II da Universidade de Coimbra, Departamento de Engenharia Informática
3030 Coimbra, Portugal
E-mail: amilcar@dei.uc.pt

One of the major drawbacks of Evolutionary Computation is the need for great computational power. The set of problems that can be solved, in practice, by evolutionary approaches is highly connected with the efficiency of the algorithm. In most Genetic Programming applications the majority of time is spent on the evaluation of the individuals. Accordingly, it is desirable to optimise this step of the process. In this paper we present two approaches through which significant speed improvements can be achieved. The first approach, *T-functions*, is effective in tasks, such as symbolic regression, that require repeated evaluation of the individuals. The second approach, *caching*, resorts to the storage of the execution results of individuals' sub-trees, thus avoiding the recalculation of these sub-programs. *Caching* finds its application when the function set includes complex, time-consuming functions.

Keywords: Genetic Programming, Implementation Issues, Symbolic Regression.

1. Introduction

Genetic Programming (GP) is an Evolutionary Computation (EC) technique; i.e. it uses the mechanisms behind natural selection to evolve computer programs. In theory, like other EC approaches, GP is specially suited to domains with weak theories or when the human resources are insufficient to allow human programming [3][5]. GP is a computationally demanding task. Accordingly, the set of practically solvable problems is deeply connected to the efficiency of the algorithm and of the implementation [5]. It is vital to minimize the overhead involved in the evaluation of the individuals, since in a large number of applications the majority of time is spent on evaluation [1][5][6].

In most GP approaches the individuals are programs written in a problem specific-language, and assume the form of a tree. The evaluation of the individuals involves their execution by an interpreter. Thus, they are decoded at runtime by a virtual machine [5], this involves transversing the tree and calling, for each node, the appropriate function. Usually, most nodes could be evaluated by a single machine-code instruction. Therefore, the majority of time is spent in the overhead caused by "unnecessary" pushing and popping and function calling [1][5][6]. To cope with this problem two approaches where proposed:

- Evolving, directly, machine-code programs [5][6].
- Online compilation of the individuals [1].

The advantages of the first approach are obvious: the individuals are machine-code programs that can be directly executed. The advantages of the second are less apparent. To compile the individuals we need to transverse the tree. Furthermore, compilation requires performing some "extra" operations. However, in tasks such as symbolic regression, the evaluation of an individual involves executing it several times. Thus, if we use a standard approach each individual will be repeatedly interpreted. If we use online compilation, the individual is compiled once and the resulting machine-code executed several times.

The speed enhancements achieved by these approaches are very high, up to 100 times faster than a standard C implementation [1][5][6]. Yet, they have some limitations: lack of portability and high development time. Furthermore, they don't deal with overheads caused by the existence of complex, time-consuming functions in the function set.

In this paper, we propose two approaches through which significant speed improvements can be achieved.

The first, *T-functions*, deals with the problem of repeated execution. The second, *Caching*, deals with overheads caused by the existence of complex functions. These approaches maintain portability and are of easy implementation. Although these algorithms are still under development, the preliminary results are promising, showing speed enhancements of up to 20 times over a standard C implementation.

The paper as the following structure: In Section 2 we present the *T-functions* approach. Next, in Section 3, we present two caching algorithms. The first one was designed for a specific application, with slow program execution and big memory requirements. The second one is simpler and of wider applicability. Section 4 comprises the experimental results and its analysis. Finally, in Section 5, we draw some conclusions and suggest directions for future work.

2. The T-functions Approach

In symbolic regression, the goal is approximating a target function, f_{target}. As a result, each individual, implicitly, defines a function, f_{ind}. To evaluate an individual we compare f_{target} with f_{ind} for a given number of points, fitness cases, $\{x_1, x_2,\ldots, x_n\}$. The fitness is, usually, the error between f_{target} and f_{ind} according to some metric. Considering that f_{target} depends on a single variable and the root mean square error as fitness we will have:

$$fitness = \sqrt{\sum_{i=1}^{n} (f_{ind}(x_i) - f_{target}(x_i))^2}$$

To calculate it the individual must be execute n times. Thus, the individual's tree will be transversed n times. Furthermore, each node's corresponding function will be called n times. This process is, of course, extremely ineffective.

Our approach consists in considering that the functions of the function set operate on tuples, instead of single arguments. E.g. instead of using:

add(a, b)

we use a T-function

add($[a_1, a_2, \ldots, a_n]$,$[b_1, b_2, \ldots, b_n]$), that adds a_1 to b_1, a_2 to b_2, etc.

By doing this, we only have to execute the individual once. The number of operations (e.g. additions) remains the same, however the tree will only be transversed once and there is only one function call per node.

It would be time consuming to pass the tuples as arguments. This can be avoided through the implementation of a stack machine. To achieve this we resort to a global variable, ReturnPos, and to a bi-dimensional global array, ReturnTuple[Tsize][n]. The results of the *T-functions* are written in the ReturnTuple. ReturnPos indicates the number of results stored in the ReturnTuple, and thus, the first free position. A small example may prove useful:

Consider the individual: add(x, sub(y, z))

The first node to be evaluated is x, its result will be written in:

ReturnTuple[0][0…n]

Next, the node y is evaluated and the result stored in:

ReturnTuple[1][0…n]

The result of z in is stored:

ReturnTuple[2][0…n]

The node sub(y, z) is calculated, using ReturnTuple[1] and ReturnTuple[2], and the result is written in:

ReturnTuple[1][0…n]

Finally add(x, sub(y, z)) is calculated and the result stored in:

ReturnTuple[0][0….n]

The variable ReturnPos controls the position in which the results are stored. It is incremented when a result is stored and, decreased when a result is used (and hence no longer needed).

As usual there is a tradeoff between speed and space, fortunately the memory requirements are not too severe. Considering *MaxDepth* as the maximum depth of the individuals and *MaxArity* as the maximum number of arguments of the functions, *MaxDepth*(MaxArity-1)* gives an upper bound for *TSize*.

3. The Caching Approach

In a GP algorithm each population is generated from the previous one. This means that a large amount of the current population's genetic code was already present in the previous population. In other words, most of the sub-programs (sub-trees) present in the current population were already executed. Nearly all GP packages store the fitness of individuals, to avoiding recalculation when they pass, unchanged, to the next generation. Our algorithm can be viewed as a generalization of this method. In addition to storing the fitness of the individuals, we also store the results of the execution of individuals' sub-trees.

As in any caching algorithm the major difficulties to handle are, deciding which results should be stored, and how to retrieve them in an efficient way.

We are going to present two different approaches. The first was designed for a specific application, the evolutionary art tool NEvAr [4]. This application evolves large sized images (from 128*128 pixels to 512*512). In this task, program execution is slow and memory requirements are big. We used a small cache and concentrated in deciding which sub-programs' results should be stored. Due to its specificity the applicability of

this algorithm to tasks with different characteristics may be small. Furthermore, its implementation is quite complex. Considering this, we developed an alternative algorithm with wider applicability and of simple implementation. We will start by describing NEvAr's caching algorithm and then we will describe the simpler one.

3.1. Caching in NEvAr

Caching, in NEvAr, involves changes to the representation of the individuals. In standard GP each individual is represented by an independent tree [3] and the populations are lists of individuals (Figure 1).

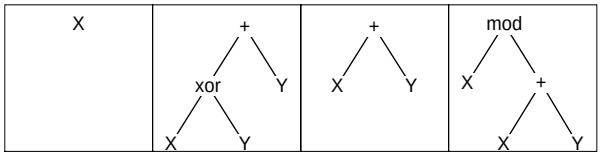

Figure 1. Four individuals represented by trees.

In our case, individuals are also trees. However, they are no longer independent, they are merged together in a way that no sub-tree is repeated; the population is a list of pointers to the roots of each individual (Figure 2).

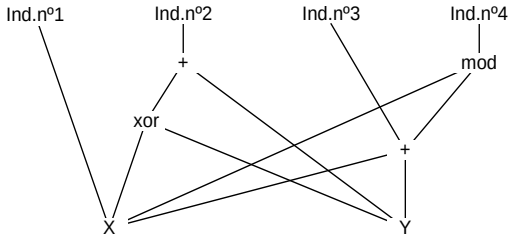

Figure 2. The four individuals of Figure 1, represented in the new format.

To each node we add two fields: *result* – to store the result of the corresponding sub-program; *time_est* - to store the time (or time estimate) of the sub-program's execution.

We also use two lists of pointers to sub-programs: *l_by_name* - sorted by name; *l_by_time* - sorted by time gain, *time_est* multiplied by *n_times*: the number of times that the sub-program must be executed when the result is already present and *n_times*-1if it hasn't yet been calculated. In our application the execution results are images, and thus use lots of memory. This implies that we have to consider a limited cache size. When we don't have this constraint, *l_by_time* isn't necessary and the algorithm is simplified.

The algorithm, with limited cache size, has the following steps:

1. Generate an initial random population of computer programs.
2. For each node of the population:
   a) Add a pointer to the node to *l_by_name*.
   b) Actualise *time_est*.
   c) Add a pointer to the node to *l_by_time*.

l_by_name	Time_est	n_times	Order in l_by time
(+ X Y)	3	2	1st
(+ (xor X Y) Y)	5	1	4th
(mod X (+ X Y))	5	1	5th
(xor X Y)	3	1	6th
X	1	4	2nd
Y	1	3	3rd

Table 1. *l_by_name* and *l_by_time* for the population of Figure 2 after step 2, considering that all functions take the same amount of time.

3. Repeat the following substeps until the termination criteria has been satisfied:
   d) Program execution and fitness assignment:
      - For each individual:
        – Transverse the tree:
          - If *result* is present, return *result* else
            – calculate result.
            – *time_est* = time taken.
          - move the corresponding *l_by_name* to its new *position* using time_est * *n_times.*
          - *n_times*=*n_times*-1
          - If *position* < cache size, store[1] the *result* else delete[1] the *result.*
   e) Generate a new population:
      - Repeat until population size is met
        – Select two individuals A and B
        – For all crossover points a' and b' and mutation points c'
          - Copy the ancestors of a' and b' or c' to a new independent location, and swap a' and b' or mutate c'
        – To all the ancestors of the a', b' and c' points:
          - If the corresponding sub-program already exists in *l_by_name*:
            – Redirect the pointer to node to the position indicated in *l_by_name*. Delete the node and corresponding sub-tree.
          - else perform the steps a) ,b),c)
      - Delete the individuals of the previous population, including unnecessary *l_by_name* and *l_by_time* entries.
      - Sort *l_by_time*.

[1] Of course that we only store the result if it isn't already present and we only delete it if it exists.

4. The best computer program is the result of the GP algorithm.

This algorithm may seem a little complex, but it is certainly worth while. Using a conventional GP algorithm, the execution step for the presented initial population would require 14*(xsize)*(ysize) operations. This number is reduced to 6*(xsize)*(ysize) with a cache of size 3.

3.2. Caching

NEvAr's caching algorithm poses several drawbacks, its complex implementation and the extra work required for the maintenance of the ordered lists. Considering this, we developed a simpler algorithm.

In this approach we give preference to storing the results of small sub-programs. The idea is that small sub-programs are more likely to appear often.

Each terminal returns a value, identifying it and the position of its result in the cache. All terminal-variables, e.g. ***x*** are stored; terminal-constants aren't actually stored, instead they return a negative number and its constant value. When a sub-tree's result is not present in the cache, it is calculated, and stored in the cache or in the Return-Tuple, depending on the availability of space. The sub-tree returns the position of its result (in the Cache or in the Return-Tuple) and a value that both, identifies it, and shows if it is cached.

To avoid search, we must be able to determine the position of a sub-tree's result in the cache-array, from its root node and the identifying numbers returned by its descendents. The need of identifying a sub-tree, by a single number, poses limits to the depth of the cached sub-programs. This problem becomes bigger as the number of variable-terminals and functions increases.

To cope with this limitation we propose the following heuristics:

- Don't cache sub-trees unless all its arguments are non-constant.
- If the function set functions have different complexities, give preference to caching the most complex ones.
- Give preference to the caching of commutative functions.
- In general the bigger the sub-tree is, the less probable it is to appear. Store sub-trees of depth N only when you have "space" to store all sub-trees of depth N-1.

4. Experimental Results

To test our algorithms we used lil-gp 1.1 [8], a widely used GP shell, which was also used by Fukunaga [1] to implement its genome compiler (GC) approach. we implemented the *T-functions* approach and the simple *caching* mechanism were implemented. The caching algorithm also uses the T-function approach. This implementation process was conservative, we made as little alterations to lil-gp as possible. In figure 3 we show the evaluation times achieved by: a standard lil-gp implementation, the *T-functions* approach, and the *T-functions* plus *caching*. The selected task was the symbolic regression of the function ***x^9***. This function was chosen because it was also used in [1][5] and thus allows comparison of results. The parameters used were the same as in [1]: population=500, generations=30, f-set = {+,-,*,%} (% is the protected division operator [3]), terminal-set={x}, tournament selection (size=5), 90% crossover, 10% reproduction, no mutation, depth limit=5.

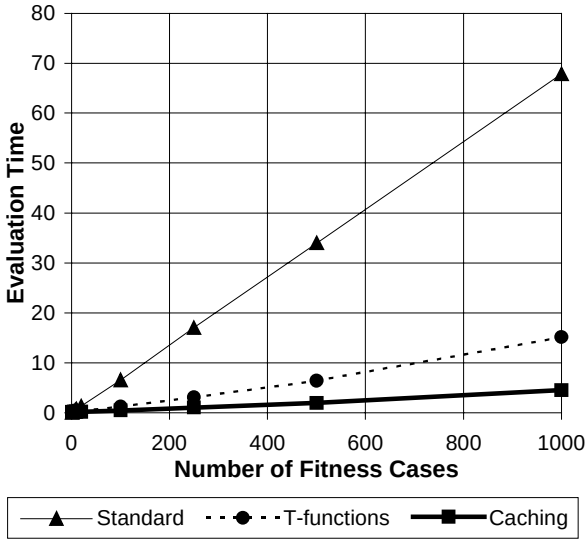

Figure 3. Evaluation time vs. number of fitness cases. The caching algorithm uses *T-functions*.

The function and terminal sets are very small, enabling the caching algorithm to store bigger sub-trees than what is generally possible. To cope with this "unfair" advantage, we limit the depth of the cached sub-trees to two, i.e., we only cache X, +(X,X), -(X,X), *(X,X), %(X,X). We also decided to cache them as variables, i.e., don't take advantage of the fact that -(X,X) and %(X,X), are constants.

	Number of Fitness Cases								
	1	**2**	**5**	**8**	**10**	**20**	**100**	**250**	**500**
GC	1.0	1.7	2.4	4.0	5.5	10.3	30.1	44.0	53.3
T-functions	0.7	1.4	2.3	2.9	3.2	4.1	5.3	5.5	5.3
Caching	0.4	1.0	2.3	3.3	4.1	6.7	14.0	16.1	17.2

Table 2. Speed enhancements when compared with standard lil-gp, a value of 2 means two times faster. Results averaged over series of 100 runs. The results of the genome compiler [1] are approximations.

Table 2 shows the speed enhancements over a standard lil-gp implementation. Even for a small number of fitness cases, significant speed improvements can be achieved, e.g., for 2 fitness cases the *T-functions* method is already 1.4 times faster. We only benefit from caching when the number of fitness cases is greater than 5. This result was expected, since the time needed to check if the sub-tree is in the cache must be smaller than the time needed to its calculation. It is interesting to notice that using an approach as simple as *T-functions*, can yield a speed enhancement of 5.5 over a standard implementation. The combination of *T-functions* with caching further enhances the results giving a 17.2 speed improvement for 500 fitness cases.

For comparison we also present the results achieved by the Fukunaga's genome compiler [1], this results were performed in a different platform. Nevertheless it is only fair to say that the GC approach clearly outperforms the presented ones. The same would be true for approaches like the ones presented in [5][6] that evolve, directly, programs in machine-code. However, since they only attack the overheads caused "unnecessary" function calls, it is doubtful that these approaches can achieve significant speed improvements, in situations where the execution time is largely influenced by the existence of complex function in the function set.

We have also applied our algorithms to programmatic compression of images (size 16*16 and 32*32). In this task, the terminal set includes X, Y and ephemeral random constants [8], the function set is the one described above. Combining our methods we got speed improvements from 15 to 25, depending on the depth limit of the individuals (9-20) and on the number of generations. Since the terminal set includes random constants, the caching algorithm was not artificially limited. To cope with the appearance of new constants, the cache is periodically restarted.

5. Conclusions and Further Work

We presented two approaches that proved to be effective in speeding up the evaluation step of GP, in tasks such as symbolic regression and programmatic compression. These algorithms have some limitations: the *T-function* method is only effective when the evaluation of the programs involves repeated execution; the caching algorithm requires that the functions have a high degree of complexity. Fortunately, these are also the cases in which speed enhancements are most needed.

Although systems like [1][5] outperform the ones presented were, our approaches have the advantages of straightforward implementation and maintenance of portability. Furthermore, the caching method seems to be the only way to get significant speed improvement when overhead in evaluation is due to the use of complex functions.

The caching algorithms are under development. In NEvAr we use an algorithm that performs a lot of calculations to decide which sub-trees should be stored. Here we also presented a simpler approach that showed to be effective.

We want to merge these ideas and develop general algorithm applicable to a wider number of tasks. The combination of caching with a genome compiler or with a Java byte-code system (to maintain portability) seems very promising.

Acknowledgements

This work was partially funded by the Portuguese Ministry of Science and Technology, under Program PRAXIS XXI.

References

[1] Fukunaga, A. Stechert, A. Mutz, D. *A Genome Compiler for High Performance Genetic Programming*, Genetic Programming Conference, GP'98, 1998.

[2] Keith, M. Martin, M. *Genetic programming in C++: Implementation Issues*. Advances in Genetic Programming, 1994.

[3] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. 1992.

[4] Machado, P.;Cardoso, *A. Model Proposal for a Constructed Artist*, World Multiconference on Systemics, Cybernetics and Informatics: SCI '97/ISAS '97, Caracas, Venezuela, 1997.

[5] Nordin, P. *A compiling genetic programming system that directly manipulates the machine-code.* Advances in Genetic Programming, 1994.

[6] Nordin, P., Banzhaf, W. *Evolving Turing-complete programs for a register machine with self-modifying code*. International Conference on Evolutionary Computation, 1995.

[7] Stoffel, K., Spector, L. *High-performance parallel, stack-based genetic programming*, Genetic Programming Conference, GP'96, 1996.

[8] Zongker, D. Punch, B. *lil-gp1.01 User's Manual*, 1996.