GenCo: A project report

Penousal Machado
ISEC – Instituto Superior de Engenharia de Coimbra
CISUC – Centro de Informática e Sistemas da Universidade de Coimbra
Dep. de Engenharia Informática
3030 Coimbra, Portugal
machado@dei.uc.pt

André Dias
CISUC - Centro de Informática e Sistemas da Universidade de Coimbra
adias@student.dei.uc.pt

Amílcar Cardoso
CISUC - Centro de Informática e Sistemas da Universidade de Coimbra
DEI - Polo II da Universidade de Coimbra, Dep. de Engenharia Informática
3030 Coimbra, Portugal
amilcar@dei.uc.pt

Abstract

Genetic Programming involves the evolution of computer programs, which are usually represented by trees composed by functions and terminals. In order to assign fitness, one must evaluate the programs, which is the most time demanding step of GP. In nowadays standard approaches, the evaluation involves an interpretation step. To avoid this step, which significantly slows the algorithm, some researchers evolve, directly, machine code programs. An alternative approach is to build a Genome Compiler, i.e. a system that transforms the individual's trees in machine-code programs and executes this code. Both techniques can bring huge speed improvements. However, these approaches have some shortcomings. In this paper we present GenCo: a research project whose main goal is development of a Genetic Programming Genome Compiler system, that overcomes some of the drawbacks of current approaches, enabling high speed improvements in a wider range of domains. We will also present experimental results in a programmatic compression task, in which GenCo was, on average, 80 times faster than a standard C based GP system.

1. Introduction

GP is one of the most recent Evolutionary Computation techniques. Its goal is to evolve populations of computer programs, which improve automatically as evolution progresses [Banzhaf 98]. Due to the outstanding influence of Koza's seminal book, "Genetic Programming: On the Programming of Computers by Means of Natural Selection" [Koza 92], it is common, within the Machine Learning community, to associate the term GP to the evolution of tree structures (even when the trees are not interpreted as computer programs). In this paper we are going to follow this "classical" definition. Therefore, when we talk about GP we are talking about the evolution of tree structures, which are built from a set of functions (f-set) and terminals (t-set). The internal nodes of the tree are members of the f-set, and the leafs are members of the t-set.

The interest in GP is growing rapidly, which can be easily explained, if we take into account that automatic programming is expected to be one of the most important tasks in computer science research over the next twenty years [Banzhaf 98]. The increase of speed in computer hardware and capability increased exponentially. However, software development was unable to keep up with this growth, and the gap is still increasing. Additionally, the demand for new software is also growing exponentially, but there isn't enough humanpower to respond to this demand. The process of writing code is simply to slow.

GP has already achieved human-competitive results in a wide variety of fields. However, the applicability of GP to complex problems and real life situations is, still, undermined by the computational complexity of the GP process. To overcome this problem, researchers have, frequently, resorted to the use of massively parallel computers, the problem is that most researchers cannot afford one. According to [Nordin 94] about 99% of the time is spend on the individuals' evaluation. In problems such as symbolic regression, in which the individuals must be evaluated for a set of fitness cases, this number becomes even higher. Thus, if we can achieve significant speed improvements in the evaluation step we will also get significant overall speed improvements.

The first GP systems where implemented in LISP, over the time researchers gradually moved to compiled languages. Nowadays, C and C++ based systems are the most popular ones, and are, roughly, 25 times faster than LISP base ones.

In 1994 Nordin proposed the direct evolution of machine code programs. His system achieved unprecedented speeds, being, approximately, 60 times faster than standard C based GP. Fukunaga [98] proposes an alternative way of getting significant speed improvements, namely: compile the individuals online and execute the resulting code. This kind of system was named a Genome Compiler, and is about 50 times faster than standard C based GP [Fukunaga 98]. The main advantage of this kind of system over Nordin's approach, is the ease of implementation, as one only needs to replace the interpretation step of an existing GP shell by a compiling and execution step. The disadvantage being that it is only effective when the individuals are evaluated several times.

In this paper we present and make a status report of the research project GenCo. The main goal of this project is to build a Genome Compiler (GC), and test its performance in a wide set of tasks. Our proposal may be considered as an extension to the current GCs, intended to minimize the drawbacks of the current GC systems and to further extent their capabilities.

The paper has the following structure: In Section 2 we make a description of the state of the art in this field, which includes an analysis of the shortcomings of current GC systems; in Section 3 we will make a brief overview of the project; Section 4 comprises an in-depth description of our project; in Section 5 we will present some experimental results; finally, in section 6 we will draw some conclusions.

2. State of the Art

GP is a computationally demanding task, so its application to complex problems is limited by the available computational power. It is, therefore, imperative to increase the efficiency of GP. In this section we will make an overview of techniques that significantly improve the speed of GP. We will focus on approaches that have a neutral effect on GP, i.e. that increase the speed without changing, in anyway, the GP results. Therefore, techniques such as partial evaluation (see, e.g., [Ochoa 97]) won't be analyzed.

The most well known methods for improving the speed of GP systems are:

- Directly evolving machine-code programs [Nordin 94], and
- Online compilation of the individuals [Fukunaga 98].

The speed enhancements brought by these approaches are huge, yet, they still haven't become popular. This can, probably, be explained by the difficulty of implementing these approaches, which requires machine-code programming. The release of Discipulus, a commercial implementation of Nordin's approach, will certainly change this picture and bring efficient GP to a broader number of researchers.

How do these systems manage to get such huge speed improvements? In standard GP approaches, the evaluation of an individual implies the transversal of its tree. For each node, one must call the corresponding function [Koza 92]. The functions in the f-set are usually very simple, e.g. +, -, *, etc., and can be evaluated by a simple machine-code instruction. Therefore, the majority of time is spent on: identifying the appropriate function to call, function calling, and pushing and popping of arguments [Nordin 94][Fukunaga 98][Banzhaf 98].

Thus, the speed improvements do not result from a faster execution of the functions, but from the elimination of the above mentioned steps.

Nordin's system was used in a wide variety of domains and one can expect speeds 60 times faster than standard C approaches [Nordin 94]. Fukunaga's Genome Compiler system wasn't, at least apparently, as thoroughly tested as Nordin's. Nevertheless, in the symbolic regression of $f(x)=x^8$ the system was 50 times faster than standard C approaches, when the individuals where evaluated for 100 fitness cases [Fukunaga 98]. It is important to notice, that this kind of approach implies the transversal of the individual's tree and its compilation. Although the compilation step can be quite efficient, this implies that this system will only be useful when the individual is evaluated several times.

It is unquestionable that these systems can bring significant speed improvements, nevertheless they have some shortcomings:

- If the function set includes complex time consuming functions the speed improvements will be small.
- It's hard to incorporate new functions, at least in a way that doesn't hinder the performance of the algorithm.

It's easy to understand why the inclusion of complex function decreases the speed improvement, if one remembers that this improvement is mainly due to the removal of the steps associated with the interpretation of the tree. If the time spent on, function identification, calling and argument passing, is small when compared with the time spent on the computation of the function, the speed improvement will be marginal.

The second drawback is also easy to comprehend, it's possible to incorporate new functions, however if you want the same type of speed improvement you must implement them in machine-code.

These systems have an additional shortcoming which is more subtle and, perhaps, more important.

One of the most interesting aspects of GP is the emergence of introns, i.e. pieces of code which are not expressed in the phenotype. The role of introns during the evolutionary process is still a source of debate (see, e.g., [Banzhaf 97, 98] for a good analysis of this issue), and, probably this role changes as the evolutionary process progresses. The emergence of introns seems to be linked with destructive crossover and mutation. It is nowadays commonly accepted that introns work as a protection against destructive genetic operations.

As the evolutionary process progresses and the number of populations increases, it becomes harder to improve the individuals, eventually the percentage of destructive crossover will also increase. As a result, individuals which are more resistant to crossover and mutation, i.e. those with a higher number of introns, will tend to dominate the population. After a while, exponential growth of introns will occur, leading to the stagnation of the run. This problem is usually named bloat problem, and is one of the central research issues in GP [Banzhaf 98].

In domains with a complex fitness landscape bloat is almost bound to happen, therefore program size and, consequently, time needed to evaluated the programs will grow exponentially. Thus, even fast systems like the ones mentioned above will become slow.

3. Project Overview

In this section we will make a brief overview of our research project, GenCo. The project is under development and the expected date of its conclusion is December 2001. As mentioned before it involves the construction of a Genome Compiler system. In the current state of development, our system is similar to the one presented by Fukunaga [98]. Thus, the individuals are compiled online, and the resulting machine code is executed.

One of the main goals of our project is to overcome or minimize the above mentioned drawbacks and shortcomings of current GCs. The basic idea to accomplish this goal is to decrease the number of operations required to evaluate the individual. To do so, we will resort to a collection of techniques, namely:

- Individual code optimization – Nowadays compilers resort to several optimization techniques in order to produce faster code. We want to incorporate these techniques in our system, which implies adapting them to the requirements of online compilation and to particularities of GP.
- Intron detection and removal – As mentioned before the number of introns tends to grow exponentially. Through the use of intron detection and removal techniques, we can reduce the number of operations needed to evaluate the individual;
- Sub-trees caching – presented in [Machado 99a], it proved to be useful when the function set includes complex primitives.
- T-functions – it is effective when the individuals are evaluated for several fitness cases[Machado 99a].

Techniques for code optimization and intron detection are widely known. However they have never been applied in this context. In a previous research project [Machado 99a], we incorporated sub-tree caching and t-functions in a well known GP shell (lil-gp). As a result, the system became 15 times faster in problems such as symbolic regression and programmatic compression. Due to their ease of implementation these techniques can be seen as a way to get significant speed improvements, without going through all the trouble required to implement a GC system. Moreover, we have good reason to believe that these techniques can be applied (with some adaptation) in a GC system, bringing further speed improvements.

Next we make a brief synthesis of GenCo’s main goals:

1. Develop a GC of high usability and of easy set-up.
2. Enhance the compilation methods of current GC's by incorporating speed oriented optimizing techniques and methods of intron detection and removal.
3. Adapt and incorporate the techniques proposed in [Machado 99a].
4. Perform a comparative study of the performance of the system in a variety of domains in order to determine its strengths and shortcomings.
5. Make a restricted release of the GC system, in order to further test its robustness and usability when used by third-parties.

The system will be tested in several domains, to evaluate it in relation to other approaches. In a later phase, it will be used in concrete situations and will become the engine of at least one application - NEvAr [Machado 99b].

To conclude, the motivation for this project is twofold:

a) It is, itself, an interesting field of research;

b) The performance improvement open new areas of application to GP, where high computational power needs inhibited its use.

4. Project Description

In this section we make a in depth description of our project. In order to make this explanation clearer, we divided the description in a set of tasks.

Task 1 Development of a Basic GC System
Expected outcome: A solid prototype of high usability and easy set-up

The development of the genome compiler system was already initiated. At this point, we have already built a basic prototype, which will be used as a development basis. In its present form, this prototype is equivalent to the system presented by Fukunaga [98]. Therefore the speed improvements should be similar (see Section 5).

The current prototype uses a "static" function set. To include new functions in the f-set the user needs to write them in assembly language.

The next steps in the development of the prototype are:

- Expanding the number of predefined f-set functions.
- Enable the inclusion of user defined functions in the f-set. The user should be able to write functions in C and integrate them in the f-set in an almost effortless way.

Additionally, the prototype shall be tested in a vast array of problems.

Task 2 Inclusion of optimizing and intron detection techniques

Expected outcome: Improved version of the previous prototype.

Speed oriented optimization techniques are well know and have became common in nowadays compilers.

In our system compilation is performed online and, therefore, the speed of the compilation is an important factor. This may decrease the range of optimization techniques that can be applied. On the other hand, evolved programs usually include redundant statements, therefore, there are a lot of simplifications that can be performed.

The detection and removal of introns is deeply linked with code optimization. introns are pieces of genetic code that don't influence the phenotype, these pieces of code can, consequently, be removed without affecting the execution results.

The removal of introns may bring important speed improvements, especially when bloat occurs, since introns become a considerable percentage of the code and delay significantly its execution.

It is virtually impossible to remove all the introns, however a large part can be deleted, through the use of simple detection techniques, without bringing significant speed overheads to the compiling stage.

We are also considering the possibility of using more complex intron detection and techniques. These techniques can detect a higher number of introns, the drawback is that they are computationally expensive. Nevertheless, they still can become handy, specially if one considers the possibility of attacking the bloat problem by periodically "cleaning" the introns in the population. Additionally, this type of technique can be used to simplify the final outcome of the GP algorithm, resulting in a faster and cleaner (and consequently easier to understand) final solution.

Task 3 Study and adaptation of other speed enhancement technique

Expected outcome: A set of speed enhancement techniques ready to implement

In a previous study [Machado 99a] we have proposed two techniques, which bring significant speed improvements to GP systems: sub-tree caching and t-functions.

The goal of this task is to adapt these techniques so that they can be incorporated in the Genome Compiler system. The adaptation of the t-function approach doesn't pose many problems and should be almost straightforward. The adaptation of the caching technique is not so obvious.

The idea behind our caching algorithm is to store the execution results of the individuals' sub-trees.

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. Therefore, most of the sub-programs (sub-trees) present in the current population were already executed.

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.

In [Machado 99a] we proposed feasible solutions for both problems. It is our belief that these solutions can be improved and adapted for the requirements of our GC system.

We will also make a survey of the current state of the art, in order to determine the existence of other speed enhancement techniques that may contribute to the improvement of the GenCo. This task was already initiated and is expected to be concluded at the same time that task 2.

Task 4 Inclusion of other speed enhancement techniques in the GC compile

Expected Outcome: An improved version of the previous prototype

The set of techniques to be implemented is going to be determined in Task 3, which is a prerequisite for this task. Some of the researchers involved in the previous task will also be involved in this one, in order to provide the necessary details. We expect, therefore a swift implementation.

Task 5 Testing and collection of performance results, comparison with other approaches

Expected outcome: Successful application of the Genome Compiler System to several domains. Assessment of the system's performance in a set of benchmark problems.

This task will run in parallel with the previous ones. We want to perform tests of different versions of the system (outputs from tasks 1,2 and 4), in order to know how the inclusion of new techniques affects performance.

The preferential domains of application are:

- Packing
- Symbolic Regression
- Programmatic Compression

The implementation of these approaches doesn't pose difficult problems since they have already been tackled by members of our team.

Additionally, GenCo will replace the current GP engine of NEvAr [Machado 99b]. NEvAr is an Evolutionary Art Tool developed by Penousal Machado. The substitution of the current engine by GenCo will result in a faster version of the tool. Moreover, intron detection, optimization and caching techniques can help the implementation of a set of features related with machine learning.

Task 6 Restricted Release and testing of the Beta Version

Expected Outcome: Evaluation of the system by third parties

This task involves making adjustments to the developed software so that it can be used by third-parties.

During the development we will keep in mind that one of our goals is to develop an "user-friendly" system. It is, however, unrealistic to think that we will arrive to this stage of development with such a system.

We will make a release of the Beta version, which will be available to a restricted and reliable group of researchers. While waiting for feedback, we will conduct our own set of tests and provide the assistance to the groups using our system.

Task 7 Project conclusion

Expected Outcome: Final version of the program

We will use the feedback from other research groups and the results of our own testing to make minor adjustments in the beta version, and to correct bugs which are bound to appear.

5. Experimental results

In this section we present some preliminary experimental results achieved with GenCo. As mentioned before, in the current state of development GenCo is roughly equivalent to the system proposed by Fukunaga [98]. Therefore the results should be roughly the same. To implement GenCo we used lil-gp [Zongker 96] and replaced the interpretation step of the algorithm by a compilation step.

We tested our system in a programmatic image compression task [Nordin 95]. We used a 32x32 pixel version of the well-known know "Lena" image (see Fig. 1), which became popular in the image compression field.

Figure 1. The *Lena* image

The experimental settings were the following: population size=500; generations=1500; f-set={+, -, *, %} (where % is the protected division operator [Koza 92]); terminal set = {X, Y, ephemeral random constants}; tournament selection of size 5; 70% crossover, 20% mutation, 10% reproduction; depth limit=25; initialization method = ramp-half-and-half. The experiments were repeat 50 times, in order to guarantee the statistical significance of the results.

The chart in Fig. 2 shows the speed improvements achieved by GenCo, when compared with the standard lil-gp implementation, and relates to population 500 to 1500. During the first 500 generations GenCo was, on average, 120 times faster than the standard lil-gp approach. The overall speed improvement between population 500-1500 is of 81.95.

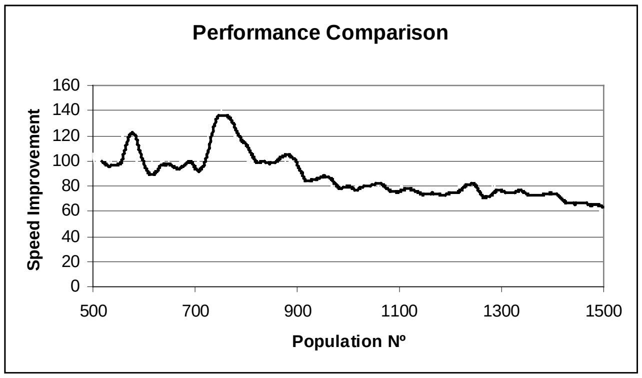

Figure 2. Speed improvements achieved with GenCo between populations 500 and 1500.

As the number of populations increases the speed improvement gets smaller, eventually stabilizing in 60 times faster. Next we will try to explain why this occurs.

In the tests conducted, as the evolution progresses, the size of the individuals tends to grow. When GenCo is dealing with small individuals, it can store most of the arguments in registers. As the individuals become larger, the percentage of arguments stored in registers becomes small when compared with the percentage stored on the stack. This implies a overhead in the retrieval of the arguments, which explains the decrease of GenCo's performance.

This decrease was expected, and that was the main reason for allowing a depth limit of 25 and concentrating our analysis on populations 500 to 1500.

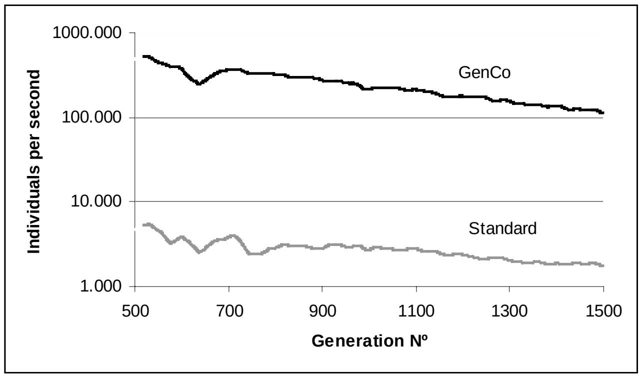

Figure 3. Number of individuals evaluated per second.

The chart in Fig. 3 shows the number of individuals evaluated per second using GenCo and standard lil-gp, the difference may seem small, however this is only apparent, the y scale is logarithmic.

In Fig. 4 we show the percentage of time spent on the individuals' evaluation and on the breeding stage, for GenCo and standard lil-gp. In a standard approach 99.472% of the time is spent on the evaluation step, in GenCo only 69.53% of the time is spent in evaluation.

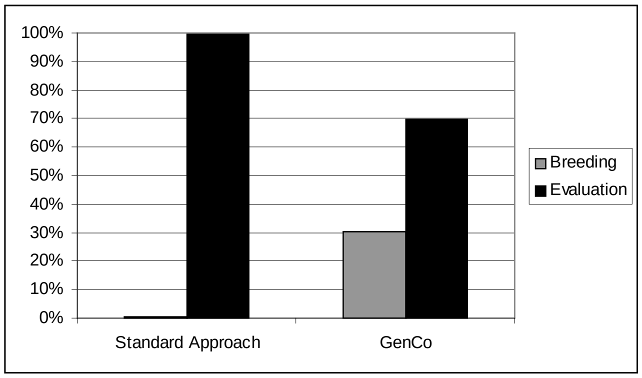

Figure 4. Percentage of time spent on evaluation and breeding between populations 500 and 1500.

We consider these experimental results to be extremely promising, specially when we take into consideration that the compiling stage can be further optimized, and that our management of registers is, still, a little rudimentary.

6. Conclusions

In this paper we made a report of the research project GenCo. We started by describing its relation with the state of the art. Then, we made a brief overview of the project describing its main ideas, the motivation and the goals. Next we made a detailed description of the project, resorting to a list of tasks. Finally, we presented some experimental results, which prove the soundness of our approach.

Our system is still under development, and there is still much work to be done. In its present form GenCo achieves results similar to the system presented in Fukunaga [98]. We consider, however, that our system as potential to outperform previous GC approaches (e.g. Nordin's and Fukunaga's) in the previously mentioned situations, namely:

- When the function set includes time consuming functions.
- When the number of introns grows exponentially.

The incorporation of techniques such as: code optimization, intron detection, sub-tree caching and t-functions, is bound to bring further speed enhancements.

We have already started to implement code optimization and expression simplification techniques, and we expect to present the results achieved with these methods soon.

Acknowledgments

This research project was approved by FCT (Fundação para a Ciência e Tecnologia) and POSI (Programa Operacional "Sociedade da Informação"), and is partially funded by FEDER, project n. POSI/34756/SRI/2000.

References

[Banzhaf 97] Banzhaf, W., Nordin, P. and Francone, F. D. *Why introns in genetic programming grow exponentially*, ICGA, East Lansing, MI, USA,1997.

[Banzhaf 98] Banzhaf, W., Nordin, P., Keller, E. and Francone, F. D. *Genetic Programming – An Introduction*, Morgan Kaufman, 1998.

[Cardoso 00] A. Cardoso, E. Costa, P. Machado, F. C. Pereira and P. Gomes, *An Architecture for Hybrid Creative Reasoning*. In: S. K. Pal, T. Dillon and D. Yeung (Eds.), Soft Computing in Case Based Reasoning, Springer-Verlag, 2000.

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

[Keith 94] Keith, M. Martin, M. *Genetic programming in C++: Implementation Issues*. Advances in Genetic Programming, 1994.

[Koza 92] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. 1992.

[Machado 99a] P. Machado and A. Cardoso, *Speeding up Genetic Programming*. In: A. O. Rodriguez, M. S. Ortiz and R. S. Hermida, Procs. 2nd Int. Symp. AI and Adaptive Systems, CIMAF'99, La Havana, Cuba, pp. 217-222, March, 1999.

[Machado 99b] Machado, P. and Cardoso, A. *NEvAr – The Assessment of an Evolutionary Art Tool*. In: Wiggins, G. (Ed.). Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science, Birmingham, UK, 2000.

[Nordin 94] Nordin, P. *A compiling genetic programming system that directly manipulates the machine-code*. Advances in Genetic Programming, 1994.

[Nordin 95] Nordin, P., Banzhaf, W. *Complexity Compression and Evolution*. International Conference on Evolutionary Computation, 1995.

[Ochoa 97] A. Ochoa. *Partial Evaluation of Genetic Algorithms*, Proc. 1st Sysposium on Artificial Intelligence, CIMAF'97, Havana, Cuba, pp. 29-36, 1997.

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

[Zongker 96] Zongker, D. Punch, B. *lil-gp1.01 User's Manual*, 1996.