An Analysis of Automatic Subroutine Discovery in Genetic Programming

Antonello Dessì
Dip. di Informatica
Corso Italia, 40 I-56125 Pisa
dessi@mclink.it

Antonella Giani
Dip. di Informatica
Corso Italia, 40 I-56125 Pisa
giani@di.unipi.it

Antonina Starita
Dip. di Informatica
Corso Italia, 40 I-56125 Pisa
starita@di.unipi.it

Abstract

This paper analyses Rosca's ARL as a general framework for automatic subroutine discovery. We review and compare a number of heuristics for code selection, and experimentally test their effectiveness in the ARL framework. We also propose and analyse a new heuristic, the *Saliency*, and two extensions to ARL: diffusion of the new subroutines through *mutation* and the *MaxFit* technique to adaptively change the length of an epoch. In spite of the effectiveness of the proposed extensions, the main result is that any attempt to improve the selection criterion seems not able to produce better results than a simple near-random heuristic.

1 INTRODUCTION

To deal with problems of increasing complexity, Genetic Programming (GP) [5] has often been extended with mechanisms for automatic subroutine discovery, to promote the emergence of modular and hierarchically structured solutions: these approaches can be classified depending on how they identify and manipulate subroutines. Koza's ADF [6] and its extension Architecture Altering [7] belong to the *evolutionary selection* approach, as subroutines evolve concurrently with the main program. GLiB [2] uses the *random selection* method, where pieces of code are randomly chosen and frozen into subroutines. While each ADF is local to the program it belongs to, the subroutines defined in GLiB can be called by any program. *Heuristic selection* is similar to random selection, but it exploits heuristics to choose the pieces of code. The best known model based on this method is ARL [9]. Evolutionary selection explicitly requires synchronization between the evolution of the main body of the program and the evolution of its subroutines, but the models proposed cannot guarantee it. Furthermore, this approach requires substantial changes to the standard genetic operators. On the other hand, random selection may be not efficient [9]. ARL does not require the definition of new genetic operators, and it can exploit the available theory on GP, where subroutines can be identified with "building blocks" [9]. In addition, ARL can be viewed as a general framework, where alternative strategies can be experimented simply by changing the selection heuristic.

This paper focuses on the ARL algorithm, to analyse its effectiveness as a general model for automatic subroutine discovery. In the next section, we investigate several aspects of ARL and present a critical review of the model and of a number of heuristics proposed in the literature to identify useful subroutines. We also propose a new heuristic, the *Saliency*, as well as two substantial changes to the basic ARL model: the diffusion of new subroutines through *mutation*, and the *MaxFit* technique to determine the length of a dynamic epoch. In Sect. 3, we develop an experimental analysis to compare the heuristics that were considered and to evaluate the effectiveness of our proposals. Finally, Sect. 4 outlines some conclusions and proposes possible developments of this work.

2 ANALYSIS OF THE ARL MODEL

Viewed as a general framework, implementing ARL implies deciding on several issues, which we separately analyse in the following.

2.1 WHEN TO CREATE SUBROUTINES

A straightforward way to determine when to create new subroutines is to fix a priori a number of generations (epoch) between two successive creations. This method, however, does not take into account the search dynamics. Furthermore, determining the length of the epoch requires some task-specific knowledge that may not be available. A more interesting solution is to let the length of the epoch change adaptively according to population diversity, whose decrease indicates that the search process is stuck in a local optimum. Under the assumption that programs with similar fitness exhibit a similar behavior, we can efficiently monitor fitness diversity. Rosca's basic ARL exploits population entropy as a measure of fitness diversity [9]. The appropriate time to introduce new subroutines is indicated by a long-term monotonic entropy decrease, which suggests that the search process is approaching a local optimum. This method has some drawbacks: first, entropy requires the set of fitness values to be partitioned into a number of classes, whose a priori definition is not obvious; second, since the entropy measure is often noisy, deducing its trend is not trivial. As Rosca points out, a monotonic entropy decrease is correlated to a plateau in the best-of-generation fitness. To simplify the definition of the dynamic epoch, we propose the *MaxFit* strategy, which directly monitors the best-of-generation fitness value in the population: only when this value does not improve over a fixed number of generations, it is the appropriate time to introduce new subroutines. Note that entropy may decrease even when the current best fitness increases, due to an excessive selective pressure. With MaxFit, this can be taken into account by considering only substantial changes in the fitness measure. Although MaxFit requires a few a priori settings as well, its implementation is much simpler than monitoring entropy, and its computational cost is much lower.

2.2 SELECTION OF USEFUL BLOCKS

The selection of useful pieces of code is the crucial point of the model. In the following, we analyse a number of heuristics proposed in the literature, and we propose a new one. Note that we only consider whole subtrees of bounded depth.

The simplest heuristic is a random choice of a block within a program, where the program may be chosen randomly (**Rnd**) or through the GP selection algorithm, like in GLiB (**RndFit**). The computational cost is low, but the approach is likely to generate too many useless subroutines.

Frequency-based heuristics select the blocks that most often recur either in the population [11] (**Frq**), or within single programs [1] (**FrqPrg**). The known large diffusion of redundant code (introns) makes this approach quite questionable: a frequent block is certainly not harmful, but it is not necessarily useful [9]. When embedded in ARL, these heuristics could promote the proliferation of the already widespread useless code. A further drawback is the large computational cost of monitoring identical blocks.

The utility of a block of code can be evaluated through the fitness function (**Fit**), or through a specialized version of that function which only exploits a part of the training set (**FitBlk**) [9]. The risk of this approach is to force a predetermined behavior of the generated subroutines, thereby a priori excluding possibly better structured solutions. Furthermore, this approach assumes that the optimal solution of a given problem can be built from a complete solution of the same problem of a lower size. This is not true in general. A further drawback is the large cost of evaluating the fitness function on each block considered.

An alternative heuristic selects blocks according to the average fitness of the programs they belong to [11] (**Schema**). However, it is not guaranteed that a large fitness of a set of programs depends on the common block. In general, the large fitness is more likely due to the interactions between such a block and the rest of the code. Furthermore, the average fitness value may be not statistically significant if not coupled with a measure of variance. Like frequency-based heuristics, this approach is very expensive, as it requires keeping track of identical blocks.

A statistical correlation between the value computed by a block and the output of the program it belongs to indicates the relevance of the block with respect to the overall behavior of the program [4] (**Correl**). However, this approach is likely to generate subroutines that approximate the behavior of the whole program, discarding possibly better alternatives. Furthermore, computing statistical correlation is expensive.

In the original ARL definition, Rosca combines two heuristics (**Blkact**): *differential fitness* and *block activation*. The former selects a subtree inserted by a crossover only if the fitness of the resulting offspring improves over at least one parent. In general, the improvement may be not due to the block itself but only to the new semantics induced by the insertion of that block into the new context. The second heuristic counts the number of times the root node of the block is executed, in order to discard blocks that are never evaluated. This method cannot be applied to domains where the defined operators always evaluate their arguments. Furthermore, this heuristic cannot discard blocks that are always evaluated although being semantically irrelevant. In spite of these limitations, the combination of the two heuristics has proven effective [9]. A further advantage is the low computational cost.

We propose a new heuristic, the *Saliency*, which aims at identifying semantically relevant blocks by discarding both semantic and syntactic introns. This heuristic slightly modifies the value computed by the block, then it evaluates the variations induced into the program's behavior. The more semantically relevant is the block, the larger should be the observed variation. If such a salient block belongs to a highly fit program, it is likely to be an useful subroutine. We consider three alternative ways of computing Saliency: the first one only evaluates the variation induced into the program output (**SalOut**):

$$SalOut = \frac{1}{N} \cdot \sum_{i=1}^{N} \begin{cases} 0 & \text{if } out(i) = out'(i) \\ 1 & \text{otherwise} \end{cases}$$

where N is the size of the training set, $out(i)$ is the program output on the i-th input and $out'(i)$ is the program output after the block output has been altered. The second alternative only considers the variation induced into the fitness (**SalFit**):

$$SalFit = ABS(f - f')$$

where f (f') is the program fitness before (after) the alteration of the block output. The third heuristic (**Sal**) combines both measures to get a more reliable estimate:

$$Sal = SalOut \cdot SalFit.$$

The computational cost of this heuristics can be reduced by considering only a small fraction of the population of programs, chosen through the GP selection algorithm. As an alternative, differential fitness may be used to select the blocks to be evaluated.

2.3 INTRODUCTION OF ARGUMENTS

In the original ARL [9], the newly created subroutines are augmented with formal arguments, to increase the generality of the selected block and to promote modularity of solutions. This is done by replacing a random subset of terminals in the block with variables. When the new subroutine is inserted into the population, the replaced terminals are used as call parameters, to preserve the original semantic.

However, while the creation of new subroutines basically aims at preserving supposed useful blocks from disruption, using formal arguments actually allows their semantic to be altered. Furthermore, the random choice of arguments does not ensure an actual generalization of the subroutine behavior, which might be drastically altered instead. The above criticisms raise some doubts about the actual effectiveness of introducing arguments, as this technique may waste the effort of selecting useful blocks.

2.4 DIFFUSION OF SUBROUTINES

To be usefully exploited, the new subroutines have to spread across the population. GLiB delegates the diffusion task to the evolutionary process, but if the block belongs to a program whose fitness is below average, its chance of spreading is low, independently of its quality. Furthermore, this method does not guarantee a fast and effective diffusion. Using the function set extended with new subroutines, ARL creates a number of random programs which replace low-fitness individuals in the population. However, the probability of generating random programs with above average fitness decreases as long as the evolutionary process improves the average fitness of the population. As a consequence, randomly created programs participate in the reproductive process only in the very early phase, whereas they are almost irrelevant in later phases. To improve the diffusion of subroutines, we propose *mutation*: a fraction of the population is chosen through the GP selection algorithm, then a random branch of each selected program is replaced with a call to the new subroutine. The new programs replace low-fitness individuals in the population. Diffusion through mutation increases the probability of generating programs with high fitness, which are more likely to take part in the reproductive process, thereby spreading the new subroutines. In addition, differently from the creation approach, this method ensures that each new program contains the defined subroutine.

2.5 DELETION OF SUBROUTINES

Rosca's ARL evaluates the actual effectiveness of the new subroutines by monitoring statistics about the programs they belong to, in order to delete the subroutines that have proven useless. However, using both heuristic selection and utility evaluation appears inefficient and resource wasting: if one assumes that the heuristic used is able to identify useful code, a further selection level is not needed. On the other hand, if the heuristic is not reliable, one should better use a random selection coupled with utility monitoring.

3 EXPERIMENTAL ANALYSIS

Our experimental analysis investigates the dynamics of ARL through one experiment for three of the basic steps of the algorithm: selection of blocks (Sect. 3.2), introduction of arguments (Sect. 3.3), and decision on when to create new subroutines (Sect. 3.4). As discussed in Sect. 2.5, we did not consider subroutine deletion, as we preferred spending more computational resources for heuristic selection. Preliminary tests persuaded us to use the *mutation* technique to spread new subroutines across the population.

3.1 METHODOLOGY

We used an implementation of ARL extended with the heuristics and the techniques described in the previous sections. To select blocks, the heuristics only consider complete subtrees of depth between 2 and 4. This does not bound their expressiveness, as they can exploit subroutines already defined. Formal arguments are introduced only in the second experiment. The epoch is dynamic only in the third experiment, whereas in the others it is fixed to 3 generations. In all the experiments, a population of 500 programs evolves for at most 50 generations. The crossover rate is 90%, whereas mutation is not used. The selection strategy is tournament selection of size 7. Only one new subroutine per epoch is created. Each experiment includes a number of tests, each one devoted to the analysis of a given heuristic. 50 independent executions have been run for each test, where each execution starts with a different seed for the random number generator. The seeds used are the same for each test, so that heuristics are tested on the same initial populations.

To select blocks, **Frq**, **FrqPrg**, **Fit**, **FitBlk** and **Schema** examine the whole population. **FitBlk** exploits 60% of the training set, randomly chosen. **Correl** computes both direct and inverse statistical correlation on 10% of the population of programs, chosen through the GP selection algorithm. Three versions of saliency-based heuristics have been implemented: **Sal** examines 10% of the population, chosen through the GP selection algorithm, whereas **SalElt** only examines 1% of the population. This comparison aims at evaluating to which extent the number of programs examined is relevant to the performance. The third version, **SalXov**, exploits differential fitness to choose the blocks to examine. The other saliency-based heuristics, **SalOut** and **SalFit**, examine blocks like **SalElt**. To compute saliency, the small alteration of the output block is differently defined for each task domain, and it includes side effects when allowed in the domain.

The experiments have been run on three known domains of varying complexity: 6 bit boolean multiplexer (**M6**), symbolic regression (**SR**), and sorting (**Sort**) (see e.g. [8] and [9] for a detailed description). Symbolic regression is defined on a second degree polynomial function of 2 input variables. The primitive set contains the basic four arithmetic operators, the unitary constant, and the two independent input variables. For the sorting problem, the primitive set contains a loop with one local variable, basic arithmetic and conditional operators, the swap operator, and two constants denoting the initial and the final index of the input array. The array is not a primitive: it's located in RAM and it's modified through side effects only. Therefore, the output of a program is not the returned value but the final configuration of the array. The sorting problem has also been used to evaluate generalization capabilities of the optimal solutions found, by computing their fitness on a different test suit.

For each heuristic, we analysed the following aspects: (1) percentage of success (an execution is successful if it finds an optimal solution); (2) total number of programs generated, averaged on all and on successful runs; (3) average population fitness in the final generation, averaged on all, on successful and on unsuccessful runs; (4) percentage of programs that use subroutines out of the total number of programs generated, averaged on all, on successful and on unsuccessful runs; (5) percentage of optimal solutions found that use subroutines; (6) average fitness in the final generation of programs that use subroutines and of programs that do not use subroutines; (7) generalization capability of the optimal solution found; (8) structural complexity [9], modularity and hierarchical degree of the subroutines generated; (9) percentage of subroutines semantically distinct and maximum number of subroutines semantically equivalent. The experimental results not shown in this paper are illustrated in details in [3].

3.2 COMPARISON OF HEURISTICS

The first experiment compares heuristics performance against the canonical GP (**CGP**). The percentage of success on each domain is shown in Tab. 1. The last two columns evaluate the generalization capability for the sorting problem: it is shown the averaged normalized fitness on the test suit and its standard deviation.

Unlike results in [9], frequency-based heuristics achieve high, although episodic, performances. A further interesting result emerges by comparing the performances of the heuristics against those achieved with a completely random choice of blocks (**Rnd**): the more complex is the problem, the more effective is an informed choice with respect to a random choice. Saliency-based heuristics show good generalization capabilities, but they are not very effective in the most complex problem (SR). The basic idea behind saliency-based heuristics is to discard introns as plausible candidates to subroutines. The experimental results suggest that this approach is not appropriate, and underline the relevance of the role played by introns in the GP dynamics. The most relevant result is that spend-

Table 1: Comparison of Heuristics

	M6	Sort	SR	Gen(StdDev)
CGP	90%	62%	36%	0.931 (0.146)
Rnd	92%	56%	18%	0.955 (0.124)
RndFit	90%	66%	34%	0.885 (0.148)
Frq	90%	66%	26%	0.896 (0.145)
FrqPrg	86%	54%	32%	0.823 (0.216)
Fit	84%	40%	24%	0.669 (0.265)
FitBlk	80%	36%	22%	0.886 (0.173)
Schema	84%	46%	20%	0.879 (0.208)
Correl	82%	58%	36%	0.822 (0.198)
Blkact	82%	54%	32%	0.885 (0.195)
SalOut	90%	60%	24%	0.862 (0.200)
SalFit	96%	54%	18%	0.878 (0.180)
Sal	86%	54%	22%	0.924 (0.160)
SalElt	92%	56%	20%	0.806 (0.231)
SalXov	84%	54%	26%	0.871 (0.144)

ing more computational resources to make the selection strategy more intelligent seems not to produce the hoped results. In fact, **RndFit** is the only heuristic that emerges as widely applicable, as it shows high performances on every problem considered. **Fit**, **FitBlk**, and **Schema** are globally inefficient, whereas other heuristics exhibit domain dependent performances. A plausible reason for the effectiveness of **RndFit** is the close similarity between the way this heuristic operates and the crossover dynamics: they share the same selection strategy of programs as well as of blocks. This similarity is likely to induce a natural and effective cooperation between the two search levels of ARL. Further results, not illustrated here, show that the discovered subroutines often exhibit low degrees of modularity and hierarchical structure, although they can improve the search. These features emerge at their least degree in the most complex problem, where modularity and hierarchical structure would be more desirable. This experiment globally suggests that the ARL framework may not be sufficient to effectively promote modularity of solutions and hierarchical decomposition of complex problems into simpler subproblems.

3.3 INTRODUCTION OF ARGUMENTS

Replacing some random subset of terminals in the block with variables is the only heuristic proposed in the literature to automatically extend subroutines with formal arguments. Although this method may increase the generality of a subroutine's behavior, it is not obvious whether this new capability is correctly exploited by GP to discover more modular and hierarchically structured programs, or whether the overall performance is at least improved. This experiment aims at evaluating the effectiveness of this method. Only 6 heuristics are considered, chosen among the best performing in the previous experiment. Only one argument is added to each subroutine. The performances obtained are shown in Tab. 2.

Table 2: Introduction of Formal Arguments

	M6	Sort	SR	Gen(StdDev)
RndFit	82%	62%	28%	0.934 (0.141)
Frq	90%	66%	30%	0.891 (0.177)
Cor	82%	62%	32%	0.856 (0.170)
Blkact	90%	64%	22%	0.951 (0.113)
SalOut	92%	54%	18%	0.932 (0.141)
SalElt	90%	52%	24%	0.934 (0.130)

The results confirms our doubts, expressed in Sect. 2.3. This process does not improve performances significantly, and in some cases it leads to a performance decrease. The technique neither improves modularity nor hierarchical structure, which are substantially the same as in the previous experiment. The only positive result concern generalization capability of optimal solutions, which globally improves over the previous experiment, although the highest scores are comparable to that achieved by the canonical GP. These negative results may be due to the random choice of the insertion point of the argument. An alternative strategy could select a set of identical leaves or subtrees, in order to increase the reuse of code.

3.4 DYNAMIC EPOCH

The third experiment analyses when to introduce new subroutines. We compare the static epoch used in the first experiment with the dynamic one, using both MaxFit and population entropy. To evaluate the entropy trend, the implementation computes a linear approximation of the values on the last 5 generations. The MaxFit technique triggers the creation of a new subroutine when the best-of-generation fitness has not significantly improved on the last 5 generations. This experiment only considers two heuristics, **RndFit** and **FitBlk**, which have achieved opposite performances in the first experiment. The results are shown in Tab. 3.

The entropy technique is generally effective and it improves the performance with respect both to the canonical GP and to subroutine discovery with static epoch. The behavior of MaxFit is similar to or better than the entropy technique. In particular, it is better than entropy on the most complex problem. We can then conclude that MaxFit is a valid alternative to the

Table 3: Static and Dynamic Epoch

	Epoch	M6	Sort	SR
CGP	unused	90%	62%	36%
RndFit	static	90%	66%	34%
	Entropy	90%	66%	32%
	MaxFit	92%	64%	40%
FitBlk	static	80%	36%	22%
	Entropy	90%	64%	36%
	MaxFit	92%	62%	36%

entropy measure, as it has the further advantages of simplicity and a low cost. However, it is worth noting that the improvement on the canonical GP is not outstanding. This suggests that the techniques analysed are still not sufficient for an effective automatic subroutine discovery mechanism.

4 CONCLUSIONS AND FUTURE WORK

The conclusions of our analysis can be summarized as follows: (1) **RndFit** is the only heuristic which has proven general and effective, while the others show domain dependent performances; (2) the not-outstanding performance of saliency-based heuristics confirms the relevant role of introns in the GP dynamics, already recognized in the literature; (3) the *MaxFit* technique has turned out to be a valid alternative to population entropy, as it is simpler and it shows a comparable or better performance; (4) the introduction of formal arguments into subroutines seems to be not generally effective and sometimes it turns out to be harmful; (5) the diffusion of new subroutines through *mutation* has proven very effective; (6) even with the extensions and changes considered in this paper, the ARL algorithm does not show a real capability of hierarchical decomposition of problems: the solutions found always exhibit low degrees of modularity and hierarchical structure.

Surprisingly, any attempt to improve the selection criterion has not managed to produce better results than a simple near-random heuristics. This is probably due to the human point of view in the design of criteria: a reasonably 'good' subroutine for an hand-coded program may not properly match the GP dynamics. In order to devise heuristics closer to the GP dynamics, an important research line is a deep theoretical analysis of GP, as well as extensive experiments to gain an insight into how GP actually works. Current research is very active in this field, but its difficulty is widely recognized. An alternative approach is to devise ad hoc heuristics for the problem at hand, a common practice in evolutionary computation, where any task-specific knowledge can be fruitful exploited to speed up search. However, this means giving up the goal of universal automatic programming. A feasible and interesting approach is to develop hybrid methods with two search levels, where the highest level decomposes the problem into a hierarchy of subtasks and combines the simpler subsolutions found by the lowest level. In this case, the GP could operate at the lowest level, as it has proven very effective in solving simple tasks which do not require complex structures. The level of task decomposition could be managed by non-genetic techniques, such as Automatic Task Decomposition, the inductive method used in the PIPE model [10].

References

[1] L. Altenberg. The evolution of evolvability in genetic programming. In *Advances in Genetic Programming*. MIT Press, 1994.

[2] P. Angeline and J. Pollack. The evolutionary induction of subroutines. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum, 1992.

[3] A. Dessì. Scoperta automatica di subroutine in Programmazione Genetica. Master's thesis, Department of Computer Science, University of Pisa, Italy, 1998. In italian.

[4] H. Iba and H. de Garis. Extending GP with recombinative guidance. In *Advances in Genetic Programming 2*. MIT Press, 1996.

[5] J. Koza. *Genetic programming: On the programming of computers by the means of natural selection*. MIT Press, 1992.

[6] J. Koza. *Genetic programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[7] J. Koza. Evolving the architecture of a multi-part program in GP using architecture-altering operations. In *Proceedings of the 4th Annual Conference on Evolutionary Programming*. MIT Press, 1995.

[8] U.M. O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, 1995.

[9] J. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, University of Rochester, 1997.

[10] R. Salustowicz and J. Schmidhuber. Learning to predict through probabilistic incremental program evolution and automatic task decomposition. Technical Report IDSIA-11-98, IDSIA, 1998.

[11] W. Tackett. *Recombination, selection, and the genetic construction of computer programs*. PhD thesis, University of Southern California, 1994.