Towards Automatic Discovery of Building Blocks in Genetic Programming

Justinian P. Rosca
Computer Science Department
University of Rochester
Rochester NY 14627
rosca@cs.rochester.edu

Abstract

This paper presents an algorithm for the discovery of building blocks in genetic programming (GP) called *adaptive representation through learning* (ARL). The central idea of ARL is the adaptation of the problem representation, by extending the set of terminals and functions with a set of evolvable subroutines. The set of subroutines extracts common knowledge emerging during the evolutionary process and acquires the necessary structure for solving the problem. ARL supports subroutine creation and deletion. Subroutine creation or discovery is performed automatically based on the differential parent-offspring fitness and block activation. Subroutine deletion relies on a utility measure similar to schema fitness over a window of past generations. The technique described is tested on the problem of controlling an agent in a dynamic and non-deterministic environment. The automatic discovery of subroutines can help scale up the GP technique to complex problems.

1 Introduction

Holland hypothesized that genetic algorithms (GAs) achieve their search capabilities by means of "block" processing (see [Holland, 1975], [Goldberg, 1989]). Blocks are relevant pieces of a solution that can be assembled together, through crossover, in order to generate problem solutions. Goldberg argues in favor of the hypothesis of building block processing by looking also for arguments in nature:

> "...simple life forms gave way to more complex life forms, with the building blocks learned at earlier times used and reused to good effect along the way. If we require similar building block processing, perhaps we should take a page out of nature's play book, testing simple building blocks early in a run and using these to assemble more complex structures as the simulation progresses".

However, recent GA experimental work disputes the usefulness of crossover as a means of communication of building blocks between the individuals of a population [Jones, 1995]. The class of problems and problem representations for which block processing is useful remains an open research topic. This topic is even more challenging for genetic programming (GP,) which involves semantic evaluation of the structures evolved.

A more detailed perspective of GA block processing is offered by the Messy Genetic Algorithm (mGA) approach [Goldberg *et al.*, 1989], [Goldberg *et al.*, 1990]. The mGA attempts to solve the *linkage problem*, a problem of representations in which features are not tightly coded together. The messy genetic algorithm explicitly attempts to discover useful blocks of code, being guided by the string structure of individuals.

In contrast, we argue that GP should rely entirely on the function of blocks of code. A lesson is learned by contrasting the GP analogy to schemata theorem from [O'Reilly and Oppacher, 1994] and modularization approaches in GP.

Modularization approaches disregard the structure of manipulated subtrees. Among them, the adaptive representation (AR) GP extension [Rosca and Ballard, 1994a] points out the importance of considering fit blocks of code.

This paper discusses in detail the usefulness of block processing in GP and presents the main features of an improved AR algorithm called *Adaptive Representation through Learning* that has builtin features for the discovery and generalization of salient blocks of code.

The paper is organized as follows. Section 2 reviews the messy GA representation and block processing paradigm that it proposes. Section 3 considers two alternative views of building blocks in GP. The first is a direct descendant from schema theory and is based on the structure of individuals. The second is given by modularization approaches.

From a practical point of view it is important to be able to automatically detect building blocks. Section 4 discusses the drawbacks of various block selection approaches from the GP literature and presents a new approach based on block"activation." This method is used in an improved AR method, ARL. The ARL algorithm is described in section 5. Section 6 presents experimental results. Finally, section 7 summarizes the main ideas of this paper.

2 The Messy Genetic Algorithm

In standard GAs it is hard to get the right combination of alleles for blocks with high defining length or high order. Moreover it is hard to preserve them in the population in order to combine them with other building blocks. Last but not least, it is more difficult to combine building blocks whose bit positions are interleaved, based on fixed crossover operators and the mutate operator. Solving these problems would enable solving bounded deceptive problems [Goldberg, 1989], and, in general, the linkage problem defined above.

To solve such problems, an mGA encodes objects as variable length strings with position independent genes. Each gene is tagged with an index representing its original position. A basic assumption is that the mapping from tag sets to building blocks is unknown, i.e. a partitioning of the set of string bit positions that corresponds to building blocks is not given initially.

An mGA attempts the building block processing problem by decomposing the search problem into three steps. First, the initialization phase generates *all possible substrings* of length k. A better alternative to generating useful combinations of alleles is the *probabilistically complete initialization* discussed in [Goldberg *et al.*, 1993]. Second, proportions of good substrings are increased and new substrings are generated by gene deletion operations. In order to evaluate substrings, the missing positions of the substring are filled with the bit values taken from a *competitive template* (see figure 1). Third, a standard GA is run to juxtapose the already detected good building blocks. The GA uses two special genetic operators, cut and splice. Cut generates two individuals from a parent by cutting the parent at a randomly chosen position. Splice concatenates two strings into one. Besides, the mutation operator may also be used.

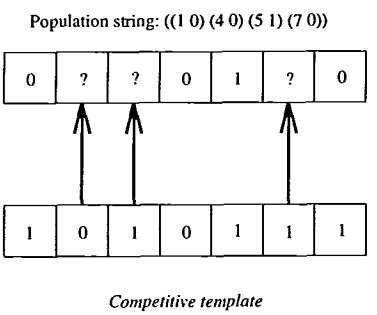

Figure 1: In order to evaluate a partially defined string (top), an mGA uses a template (bottom) that is a locally optimal structure and is named *competitive template*. The string elements are pairs of the form (index-tag, allele).

A string can be chosen as a competitive template for level k when it is optimal, i.e. no k bit changes can improve its fitness. Thus, a competitive template is a locally optimal structure. Only strings that achieve a fitness value better than that of the competitive template are candidate building blocks at that level. A greedy procedure is used to generate such templates. Prior domain knowledge may also be used in this process.

The mGA raises two problems and offers possible solutions to them: (1) evaluation of a block or partially defined string, and (2) population enrichment with salient blocks. Once salient blocks are separately created on the basis of procedures for (1) and (2), it is hoped that the mechanics of GA crossover eventually combines the locally optimal blocks and speeds up the genetic search process.

3 Building Blocks in GP

3.1 Structural Approach

The above discussion of the mGA representation points out the explicit role of the *structure* of an individual in a genotype encoding as well as the idea of block evaluation. The analysis of schemata in GA processing also relies on the structure of an individual. It is natural to question whether structure information could provide hints in the analysis of building blocks in genetic programming.

A GP analogy along the lines of GA schemata theory and GA building block hypothesis has been attempted in [O'Reilly and Oppacher, 1994]. The main goal was understanding if GP problems have building block structure and when GP is superior to other search techniques. Next, we overview this analysis.

A GP schema was defined to be a collection of tree fragments [Koza, 1992]. This intuitive definition was generalized to a collection of trees possibly having subtrees removed [O'Reilly and Oppacher, 1994]. An individual instantiates a schema in case it "covers" (matches) all the fragments. Overlappings between fragments are not allowed.

The probability of tree disruption by crossover was estimated based on this latter definition. A couple of problems specific to GP had to be overcome. First, subtrees are free to move anywhere in a tree structure as a result of crossover. Multiple copies of a subtree can appear within the same individual, so that instances of a tree fragment should be counted. A count value, representing the number of appearances of that fragment in a tree, is attached to each schema fragment. Second, the notion of schema order or specificity changes. Specificity is a relative measure, as the size of GP individuals is variable.

A characterization of schema processing was difficult in the structural approach offered by the GP schema definition. [O'Reilly and Oppacher, 1994] conclude that schema processing, as defined, does not offer an appropriate perspective for analyzing GP.

A structural approach is also at the basis of "constructional problems" [Tackett, 1995], i.e. problems in which the evolved trees are not semantically evaluated. Instead, tree fitness is based on the decomposi-

tion into target expressions, similar to the generalized GP schema, to which are assigned fitness values. By ignoring the semantic evaluation step, the analysis of constructional problems is not generalizable to GP in general.

3.2 Functional Approach

A GP *structural theory* analogous to GA schemata theory, as attempted in [O'Reilly and Oppacher, 1994] side-stepped the *functional* role of the GP representation. In contrast, modularization approaches take a functional perspective. Modularization addresses the problems of inefficiency and scaling in GP. Modularization approaches consider the effect of encapsulating and eventually generalizing blocks of code.

Three main approaches to modularization, discussed in the GP literature, are automatically defined functions (ADFs) [Koza, 1992], module acquisition (MA) [Angeline, 1994b] and adaptive representation (AR) [Rosca and Ballard, 1994a].

A first approach to modularization was the idea of *encapsulation* or *function definition*, introduced in [Koza, 1992]. The encapsulation operation, originally called "define building block" was viewed as a genetic operation that identifies a potential useful subtree and gives it a name so that it can be referenced and used later. Encapsulation is a particular form of function definition, with no arguments. In general, a function or subroutine is a piece of code that performs common calculations parameterized by one or more arguments.

[Koza, 1992] also introduced the idea of *automatic function definition*. In this approach each individual program has a dual structure. The structure is defined based on a fixed number of components or *branches* to be evolved: several function branches, also called automatically defined functions, and a main program branch. Each function branch (for instance ADF_0, ADF_1) has a fixed number of arguments. The main program branch (*Program-Body*) produces the result. Each branch is a piece of LISP code built out of specific primitive terminal and function sets, and is subject to genetic operations. The set of function-defining branches, the number of arguments that each of the function possesses and the "alphabet" (function and terminal sets) of each branch define the *architecture* of a program.

GP has to evolve the definitions of functions with a fixed number of arguments and a value-returning expression (main program) that combines calls to these functions. During evolution, only the fitness of the complete program is evaluated.

Genetic operations on ADFs are syntactically constrained by the components on which they can operate. For example, crossover can only be performed between subtrees of the same type, where subtree type depends on the function and terminal symbols used in the definition of that subtree. An example of a simple typing rule for an architecturally uniform population of programs is *branch typing*. Each branch of a program is designated as having a distinct type. In this case the crossover operator can only swap subtrees from analogous branches.

The second approach to modularization is called *module acquisition* ([Angeline, 1994b], [Angeline, 1994a]). A module is a function with a unique name defined by selecting and chopping off branches of a subtree selected randomly from an individual. The approach uses the *compression* operator to select blocks of code for creating new modules, which are introduced into a genetic library and may be invoked by other programs in the population. Two effects are achieved. First the expressiveness of the base language is increased. Second modules become frozen portions of genetic material, which are not subject to genetic operations unless they are subsequently decompressed.

The third approach is called *Adaptive Representation* (AR) [Rosca and Ballard, 1994a]. The basic idea is to automatically extract common knowledge in the form of subroutines that extend the problem representation. AR explicitly attempts to discover new good functions or subroutines, based on heuristic criteria in the form of domain knowledge. Good subroutines would represent building blocks in GP.

In order to control the process of subroutine discovery, AR keeps track of small blocks of code appearing in the population. A key idea is that although one might like to keep track of blocks of arbitrary size, only monitoring the merit of small blocks is feasible. Useful blocks tend to be small. Blocks are generalized to subroutines ([Rosca and Ballard, 1994b]) and the process can be applied recursively to discover more and more complex useful blocks and subroutines. Newly discovered subroutines dynamically extend the problem function set.

Consequently, AR takes a *bottom-up approach* to subroutine discovery and evolves a hierarchy of functions. At the basis of the function hierarchy lie the primitive functions from the initial function set. More complex subroutines are dynamically built based on the older functions, adapting the problem representation (see Figure 2).

Discovered functions or function-defining branches will be called subroutines in the rest of the paper.

4 Block Selection

4.1 Existent Approaches

MA randomly selects a subtree from an individual and randomly chops its branches to define a module and thus decide what part of genetic material gets frozen.

ADF samples the space of subroutines by modifying automatically defined functions at randomly chosen crossover points. This may not be a good strategy due to the the non-causality problem of ADF [Rosca and Ballard, 1995]. The causality perspective analyzes how natural or smooth perturbations of solutions can be generated through crossover and are advantageous. Non-causality relates small changes in the structure of

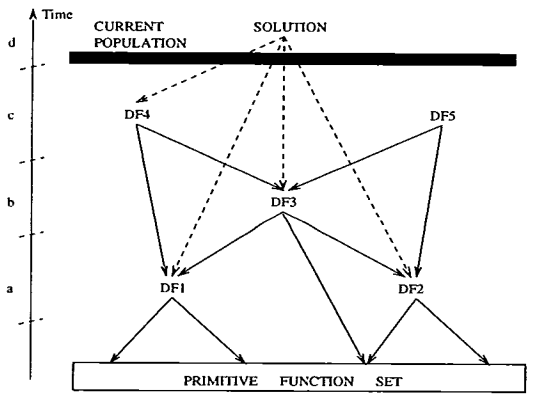

Figure 2: A hypothetical call graph of the extended function set in the AR method. The primitive function set is extended hierarchically with subroutines ($DF1, DF2$, etc.) discovered at generation numbers a, b, c.

a parse tree with drastic effects of such changes on the properties or behavior of the individual (such as the results or side effects of executing the function represented by the parse tree). Causality, or smooth behavior change in response to small genotypic changes (also called strong causality), would be needed in late stages of evolution for tuning evolved structures.

However, ADF does not appear to favor such fine adjustments. In most cases, crossover changes in either a subroutine or the result producing branch of an individual create an offspring with a totally changed behavior. For example, a change in a subroutine at the basis of the hierarchy of ADFs is amplified through the hierarchy of calls drastically changing the individual behavior. Similarly, a change in the result producing branch may also determine a drastical change in behavior. Due to lexical scoping for ADF calls, calls to ADFs from the other parent will now refer to local and potentially different subroutines, drastically changing the semantics of the individual program. Such non-causal changes may determine a loss of beneficial evolutionary changes. ADF counteracts the non-causality effect at some waste in computational effort by employing a bottom-up stabilization of subroutines. Also, the implicit biases of GP search (regarding expected height of crossover points, position of subtrees involved in crossover, effective code and structure exploitation in GP) alleviate the problem [Rosca and Ballard, 1995]. Early in the process changes are focused towards the evolution of more primitive functions. Later in the process the changes are focused towards the evolution of program control structures, i.e. at higher levels in the hierarchy.

The search control structure of AR explicitly favors the bottom-up definition of a hierarchy of subroutines. In contrast to uninformed or random manipulation of blocks of code as in ADF, AR takes an informed or heuristic approach to sift through all new blocks of code based on block fitness. Determining fit or salient blocks of code is a critical problem. What is a salient block of code and how can it be used to define new subroutines?

In AR, the elements of the hierarchy of subroutines are discovered by using either heuristic information as conveyed by the environment or statistical information extracted from the population. The heuristics are embedded in block fitness functions which are used to determine fit blocks of code. The hierarchy of subroutines evolves as a result of several steps:

1. Select candidate building blocks from fit small blocks appearing in the population
2. Generalize candidate blocks to create new subroutines
3. Extend the representation with the new subroutines.

The generation intervals with no function set changes represent evolutionary *epochs*. At the beginning of each new epoch, part of the population is extinguished and replaced with random individuals built using the extended function set [Rosca and Ballard, 1994a]. The extinction step was introduced in order to make use of the newly discovered subroutines.

Evaluation should be based on additional domain knowledge whenever such knowledge is available. However, domain-independent methods are more desirable for this goal. Unfortunately, simply considering the frequency of a block in an individual [Tackett, 1993], in the population [Rosca and Ballard, 1994c], the block's constructional fitness or schema fitness [Altenberg, 1994] is not sufficient.

Constructional fitness takes into account the proliferation rate of the block within the population. However, such a measure has inherent drawbacks. First, constructional fitness is biased due to the extremely small population sizes considered [O'Reilly and Oppacher, 1994]. Second, it is computationally expensive to keep track of all possible blocks of code and block frequency can be misleading [Rosca and Ballard, 1994a]. Third, a block of code may rarely have a stationary distribution in its effects on the fitness of programs, a necessary condition to make constructional fitness useful [Altenberg, 1994].

Schema fitness, also called conditional expected fitness of a block [Tackett, 1995] is the average fitness of all the members of the population which contain the block. Tackett performs an off-line analysis of conditional fitness. Problems encountered when considering block frequency appear again when trying to determine saliency based on conditional fitness: salient traits have high conditional fitness but high fitness does not necessarily imply salience.

ARL discovers potential salient blocks of code based on two notions, differential offspring-parent fitness and "active" blocks. These notions are presented next.

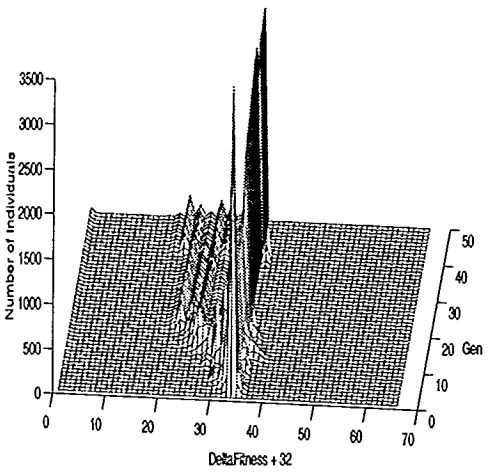

Figure 3: Differential fitness distributions over a run of GP.

4.2 Differential Fitness

Global measures such as the population diversity or local measures such as the differential fitness from parents to offspring can be used to guide the creation of new subroutines. ARL relies both on global and local information implicitly stored in the population to determine the utility of blocks of code.

First, blocks of code are selected from programs (i) with the highest difference in fitness:

$$StdFitness(i) - min_{p \in Parents(i)}\{StdFitness(p)\}$$

High differences in fitness are presumably created by useful combinations of pieces of code appearing in the structure of an individual. This is exactly what the algorithm should discover. Figure 3 shows a 3D histogram of the differential fitness defined above for a run of Even-5-Parity. Each slice of the 3D plot for a fixed generation represents the number of individuals in a population of size 4000 for which the differential fitness has a given value. The figure suggests that a small number of individuals improve on the fitness on their parents (the right of the "neutral wall" for $DeltaFitness = 0$). ARL will focus on such individuals in order to discover salient blocks of code.

4.3 Block Activation

Program fitness is calculated as a result of program evaluation. In some applications a program is evaluated on a set of fitness cases. In other applications the same program has to be evaluated a number of times on the same problem instance. During repeated program evaluation, some blocks of code are executed more often than others. The active blocks become candidate blocks. *Block activation* is defined as the number of times the root node of the block is executed. Salient blocks are active blocks of code that prove to be useful.

The method requires that each node have an associated counter recording its number of executions but does not necessitate additional effort in an interpreted GP system.

In contrast to [Tackett, 1995], salient blocks have to be detected efficiently, on-line. Consequently, candidate blocks are only searched for among the blocks of small height (between 3 and 5 in the current implementation) present in the population. This is done by using a record of the dynamics of the population. Only new blocks created through crossover are examined. All new blocks can be discovered in $O(M)$ time, where M is the population size, by marking the program-tree paths actually affected by GP crossover and by examining only those paths while searching for new blocks [Rosca and Ballard, 1994a].

Nodes with the highest activation value are considered as candidates. In addition, we require that all nodes of the subtree be activated at least once or a minimum percentage of the total number of activations of the root node. This condition is imposed in order to eliminate from consideration blocks containing introns (for a discussion of introns in GP see [Nordin *et al.*, 1995]) and hitch-hiking phenomena [Tackett, 1995].

5 Adapting Representation through Learning

5.1 ARL Strategy

The central idea of the ARL algorithm is the dynamic adaptation in the problem representation. The problem representation at generation t is given by the union of the terminal set $\mathcal{T}$, the function set $\mathcal{F}$, and a set of evolved subroutines $\mathcal{S}_t$. $\mathcal{T}$ and $\mathcal{F}$ represent the set of initial primitives and are fixed throughout the evolutionary process. In contrast, $\mathcal{S}_t$ is a set of subroutines whose composition may vary from one generation to another. The intuition is that an alteration of the composition of $\mathcal{S}_t$ may dramatically change the search behavior of the GP algorithm. For example, the inclusion of more complex subroutines, that turn out to be part of a final solution, will result in less computational effort spent during search for the creation of good candidate solutions and will speed up search. The ARL algorithm attempts to automatically discover useful subroutines and grow the set $\mathcal{S}_t$ by applying the heuristic *"pieces of useful code may be generalized and successfully applied in more general contexts."*

$\mathcal{S}_t$ may be viewed as a population of subroutines that extends the problem representation in an adaptive manner. Subroutines compete against one another but may also cooperate for survival. New subroutines are born and the least useful ones die out. $\mathcal{S}_t$ is used as a pool of additional problem primitives, besides $\mathcal{T}$ and $\mathcal{F}$ for randomly generating individuals in the next generation, $t+1$.

5.2 Discovery of Subroutines

New subroutines are created using blocks of genetic material from the pool given by the current population. The major problem here is the detection of what are salient, or useful, blocks of code. The notion of usefulness in the subroutine discovery heuristic relies on the idea of tracking active pieces of code, as described before. Useful active code is generalized and transformed into useful subroutines.

Each block of code finally selected from a population individual is transformed into a subroutine through inductive generalization. Generalization replaces some random subset of terminals in the block with variables. Variables become formal arguments of the subroutine created. This operation makes sense in the case when the closure condition is satisfied by the sets of terminals and functions. In typed GP [Montana, 1994], each terminal selected for generalization will have a certain type. The type is inherited by the variable introduced in the corresponding place. Thus, the type information of all the variables introduced and the type of the block root node will define the signature of the new subroutine.

Note that the method of discovering building blocks and creating new subroutines based on simple blocks of code is applied recursively so that subroutines of any complexity can be discovered.

New subroutines created in a generation are added to $\mathcal{S}_t$. $\mathcal{S}_t$ is used to randomly generate new individuals that enter the fitness proportionate selection competition in the current population. Each subroutine is assigned an utility value that averages the fitness of all individuals that invoke it. The subroutine utility is updated using a running average formula over successive generations. Low utility subroutines are deleted in order to make room to newly discovered subroutines. The code of deleted subroutines is substituted in all places where the subroutine is invoked.

6 Experimental Results

6.1 Test Case

We have tested the ARL algorithm on the problem of controlling an agent in a dynamic environment, similar to the Pac-Man problem described in [Koza, 1992]). The problem is to evolve a controller to drive the agent (Pac-Man) in a dynamic and non-deterministic world in order to acquire the maximum reward. A partial solution to the problem is a program involving decisions of what actions to take in every resulting world state.

Pac-Man is complex discrete-time, discrete-space problem. Koza describes a GP implementation to the Pac-Man problem, that involves a high enough problem representation so as to focus on a single game aspect, that of task prioritization [Koza, 1992]. Call this representation A.

We have extended and changed representation A in three ways:

- All the action primitives now return the distance from the corresponding element.
- We have introduced relational ($<, =, >=$), logical operators (and, or, not), and random integer constants representing distance and direction.
- The $iflte$ function was replaced by an if-then-else function, $ifte$, which tests its condition and executes either its *then* or its *else* branch.

The GP system uses point typing in the random generation of programs and in crossover operations under this new representation B (see Table 1). The goal was to evolve programs that express explicit conditions under which certain actions are prescribed, as opposed to the non-causal representation in [Koza, 1992] which creates programs that encode state information and rely on side-effects.

The typed representation takes into account the signature of each primitive, i.e. the return type of each function as well as the types of its arguments. It facilitates the evolution of explicit logical conditions under which actions can be executed.

Table 1: Signatures of functions (type of result and argument types) in the typed Pac-Man representation.

Function	Type	#Arg	Arg. Types	Description
ifb	act	2	act, act	if monsters blue
ifte	act	3	bool, act, act	if-then-else
rand-dis	dis	0		random dis.
rand-direct	direct	0		random direct.
and	bool	2	bool, bool	
not	bool	1	bool	
or	bool	2	bool, bool	
<	bool	2	dis, dis	
<	bool	2	direct, direct	
...				
sense-dis-food	dis	0		sense dis. to food
...				
act-a-pill	act	0		move to closest pill
...				

6.2 ARL Runs

We examined the ability of the ARL algorithm to discover useful subroutines and to use them in a beneficial way for generating problem solutions. We also compared ARL with standard GP using the problem representation from [Koza, 1992]. In all experiments the population size was 500 and the algorithm was run for 100 generations. The size of the set of subroutines was 10. Other GP parameters were chosen as in [Koza, 1994b].

Table 2 shows a sample of results for the best solutions obtained using standard GP, ARL and by hand coding.

The best-of-generation program evolved by ARL for run number 3 is extremely modular, relying on 6 useful subroutines. Only one of the subroutines is reused a second time. All the others are invoked once only. However, all six subroutines are effective in guiding the

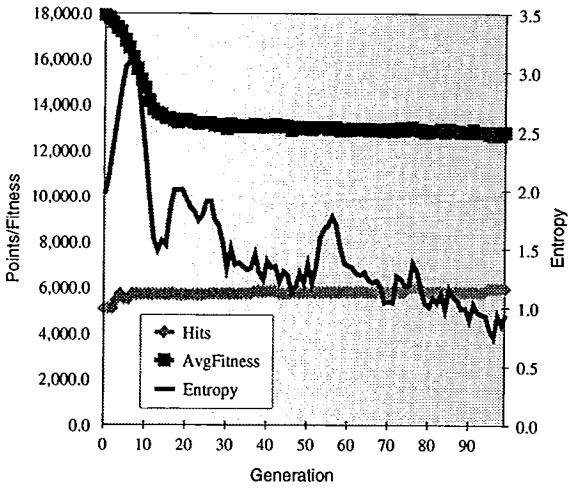

Figure 4: Variation of the number of hits, average fitness and entropy for solution 1 (GP).

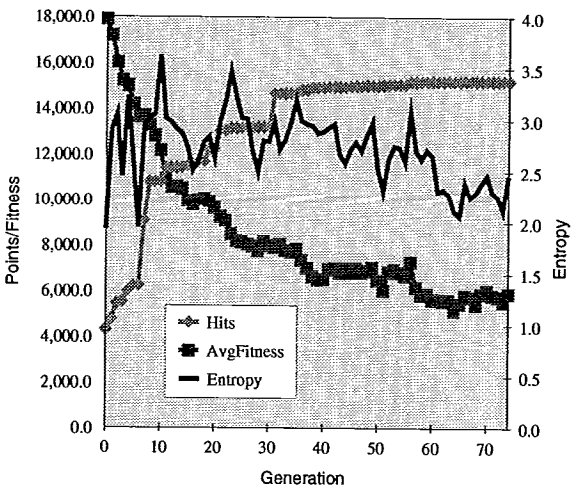

Figure 5: Variation of the number of hits, average fitness and entropy for solution 2 (ARL).

Pac-Man for certain periods of time. Among the six subroutines, two parameterless subroutines appear to be extremely interesting. One is successfully used for attracting monsters:

```
(ifte (< (sense-dis-fruit) 50)
      (act-a-fruit) (act-a-mon-1))
```

The second is used for carefully advancing to the pill. It's simplified code is:

```
(if (= (sense-direct-mon-1)
       (sense-direct-pill))
    (act-r-mon-1) (act-a-pill))
```

Figures 4 and 5 compare the evolutionary process for GP (solution 1) and ARL (solution 3). ARL maintains a high diversity (measured as entropy, see [Rosca, 1995]) due to the discovery and use of new subroutines and is able to discover better solutions much faster.

GP solutions have poor generalization capability but this may not be surprising taking into account that the environment is non-deterministic. However, this is also the case with human designed solutions which are written so that they apply in the most general case (see solution 4 in table 2). Solutions obtained with ARL appear to be very interesting. They perform much better than the ones evolved by means of GP or hand-coded, and they are also evolved faster. The GP traces clearly show the reach of local minima.

Table 2: Comparison between solutions obtained with standard GP and ARL and a carefully hand designed program. Each solution has been tested on 100 different random seeds simulating a non-deterministic environment. The table shows the maximum number of points, average, standard deviation, and number of cycles of program execution until the agent is eaten by Pac-Man.

Run	Algorithm	Rep	Max	Avg	Std.Dev.	Cycles
1	GP	A	5720	2400	150	121
2	ARL	A	14640	2568	308	1
3	ARL	B	9200	2930	290	136
4	Hand	B	8560	2736	323	151

7 Conclusions

We proposed an improved version of the AR approach called adaptive representation through learning (ARL). ARL combines competition between individuals in a population of programs with cooperation between individuals in a population of evolvable subroutines. ARL does not require explicit block fitness functions. It evaluates blocks of code from highly improved offspring based on block activation.

GP and ADF are based on a "blind" competition between individuals. In contrast, in ARL, the population of subroutines extracts common knowledge that emerges during the evolutionary process. Subroutines beneficial to some individuals could be invoked also by other individuals. The end result is the collection of subroutines that acquires the necessary structure for solving the problem. It takes the form of a hierarchy of subroutines as in figure 2. The population of subroutines evolves through creation and deletion operations.

The ARL extension to GP maintains a fixed size population of programs (partial solutions) and a fixed size population of discovered subroutines. Subroutines are cumulatively rewarded for each invocation from newly created programs or other subroutines. They also compete for existence in the function set. The population of subroutines evolves slowlier than the population of programs.

We plan to experiment with duplication and mutation operations on the population of subroutines. The

duplication operations are causal [Rosca and Ballard, 1995] and should have exploitative role, by increasing the potential for specialization or generalization of the behavior of programs, similarly to the creation and addition of ADF operations described in [Koza, 1994a]. The mutation operator will have an exploitative role and enable the evolution of the set of functions. It would determine the creation of new population individuals that would invoke the newly created functions.

A more thorough comparison among solutions obtained using the GP, ADF and ARL algorithms is undergoing.

References

[Altenberg, 1994] Lee Altenberg, "The Evolution of Evolvability in Genetic Programming," In Kim Kinnear, editor, *Advances in Genetic Programming.* MIT Press, 1994.

[Angeline, 1994a] Peter J. Angeline, *Evolutionary Algorithms and Emergent Intelligence,* PhD thesis, Computer Science Department, Ohio State University, 1994.

[Angeline, 1994b] Peter J. Angeline, "Genetic Programming and Emergent Intelligence," In Kim Kinnear, editor, *Advances in Genetic Programming.* MIT Press, 1994.

[Goldberg, 1989] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, 1989.

[Goldberg *et al.*, 1993] David E. Goldberg, Kalyanmoy Deb, and Bradley Korb, "Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms," In *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, 1993.

[Goldberg *et al.*, 1989] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb, "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex Systems*, 3:493–530, 1989.

[Goldberg *et al.*, 1990] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb, "Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale," *Complex Systems*, 4:415–444, 1990.

[Holland, 1975] John H. Holland, *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, The University of Michigan, 1st edition, 1975.

[Jones, 1995] Terry Jones, "Crossover, Macromutation and Population-Based Search," *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA95)*, pages 73–80, 1995.

[Koza, 1992] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[Koza, 1994a] John R. Koza, "Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming," Computer Science Department STAN-CS-TR-94-1528, Stanford University, 1994.

[Koza, 1994b] John R. Koza, *Genetic Programming II*, MIT Press, 1994.

[Montana, 1994] David J. Montana, "Strongly Typed Genetic Programming," Technical Report 7866, BBN, 1994.

[Nordin *et al.*, 1995] Peter Nordin, Frank Francone, and Wolfgang Banzhaf, "Explicitly defined introns and destructive crossover in genetic programming," Technical Report NRL2, Univ. of Rochester, June 1995.

[O'Reilly and Oppacher, 1994] Una-May O'Reilly and Franz Oppacher, "The troubling aspects of a building block hypothesis for genetic programming," In *Proceedings of the Third Workshop on Foundations of Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, 1994.

[Rosca, 1995] Justinian P. Rosca, "Entropy-Driven Adaptive Representation," In *Proceedings of the Workshop: Genetic Programming: From Theory to Real World Application, the Twelfth International Conference on Machine Learning*, pages 23–32. Univ. of Rochester, June 1995.

[Rosca and Ballard, 1994a] Justinian P. Rosca and Dana H. Ballard, "Genetic Programming with Adaptive Representations," Technical Report 489, University of Rochester, Computer Science Department, February 1994.

[Rosca and Ballard, 1994b] Justinian P. Rosca and Dana H. Ballard, "Hierarchical Self-Organization in Genetic Programming," In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 251–258. Morgan Kaufmann Publishers, Inc, 1994.

[Rosca and Ballard, 1995] Justinian P. Rosca and Dana H. Ballard, "Causality in Genetic Programming," *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA95)*, 1995.

[Rosca and Ballard, 1994c] Justinian P. Rosca and Dana H. Ballard, "Learning by Adapting Representations in Genetic Programming," In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 407–412. IEEE Press, Orlando, 1994.

[Tackett, 1993] Walter Alden Tackett, "Genetic Programming for Feature Discovery and Image Discrimination," In *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, 1993.

[Tackett, 1995] Walter Alden Tackett, "Mining the Genetic Program," *IEEE Expert Magazine*, June 1995.