Justinian P. Rosca and Dana H. Ballard

A fundamental problem in learning from observation and interaction with an environment is defining a good representation, that is a representation which captures the underlying structure and functionality of the domain. This chapter discusses an extension of the genetic programming (GP) paradigm based on the idea that subroutines obtained from blocks of good representations act as building blocks and may enable a faster evolution of even better representations. This GP extension algorithm is called *adaptive representation through learning* (ARL). It has built-in mechanisms for (1) creation of new subroutines through discovery and generalization of blocks of code; (2) deletion of subroutines. The set of evolved subroutines extracts common knowledge emerging during the evolutionary process and acquires the necessary structure for solving the problem. ARL was successfully tested on the problem of controlling an agent in a dynamic and non-deterministic environment. Results with the automatic discovery of subroutines show the potential to better scale up the GP technique to complex problems.

9.1 Introduction

Hierarchical Genetic Programming (HGP) extensions discover, modify, and exploit subroutines to accelerate the evolution of programs [Koza 1992, Rosca and Ballard 1994a] . The use of subroutines biases the search for good programs and offers the possibility to reuse code.

While HGP approaches improve the efficiency and scalability of genetic programming (GP) for many applications [Koza, 1994b], several issues remain unresolved. The scalability of HGP techniques could be further improved by solving two such issues. One is the characterization of the *value* of subroutines. Current methods for HGP do not attempt to decide *what* is relevant, i.e. which blocks of code or subroutines may be worth giving special attention, but employ genetic operations on subroutines at random points. The other issue is the time-course of the generation of new subroutines. Current HGP techniques do not make informed choices to automatically decide *when* creation or modification of subroutines is advantageous or necessary. The *Adaptive Representation through Learning* (ARL) algorithm copes with both of these problems. The "what" issue is addressed by relying on local measures such as parent-offspring *differential fitness* and *block activation* in order to discover useful subroutines and by learning which subroutines are useful. The "when" issue is addressed by relying on global population measures such as *population entropy* in order to predict when search reaches local optima and escape them. ARL co-evolves a set of subroutines which extends the set of problem primitives.

ARL points out an important distinction between representations of data and the underlying system that enables the creation of representations. Representations in GP, often referred to as trees or programs, are the structures evolved. The underlying system is called a *representation system*. In GP, the representation system or *vocabulary* is given by the primitive symbols (terminals and functions) and rules of combining these symbols. For strongly-typed GP [Montana, 1994] the rules of the representation system are equivalent to a grammar [Whigham, 1995]. ARL does not only adapt representations, which is what any learning algorithm attempts to do, but also adapts the representation system i.e. biases search in the language given by the problem primitives. ARL also adapts the vocabulary.

This chapter is organized as follows. Section 9.2 motivates the focus on subroutine discovery in GP. Section 9.3 overviews previous work. Section 9.4 describes the key concepts used by the ARL algorithm (block selection, subroutine discovery, evolution of the set of subroutines) and shows how they are merged within the GP framework. Section 9.5 discusses an answer to the "when" question based on population entropy. Next we present a complex test case, the problem of learning to control an agent in a dynamic environment (Section 9.6) and results obtained in experiments on this problem (Section 9.7). Section 9.8 presents the results obtained in the larger context of reinforcement learning problems. Finally, Section 9.9 concludes and considers directions for future work.

9.2 Background

The idea of using subroutines in genetic programming (GP) is drawn from the genetic algorithm (GA) building block hypothesis. Building blocks are relevant pieces of a partial solution that can be assembled together in order to generate better partial solutions to the problem at hand. Holland [1992] (see also [Goldberg, 1989]) hypothesized that GAs achieve their search capabilities by means of "block" processing. This lead to several attempts to explicitly identify and use blocks in GA algorithms. For example, the messy genetic algorithm (mGA) [Goldberg et al., 1989] explicitly attempted to discover useful blocks of code guided by the *string structure* of individuals. The structure is apparent in the mGA representation which takes the form of a string having each gene tagged with an index representing the gene's original position. After filtering useful blocks, mGA employs typical GA operations to combine those blocks. Perhaps owing to the purely structural nature of block definitions, the improvements of these experiments were somewhat modest and in fact the building block hypothesis has so far not gained conclusive support in GA literature. Nor is it clear how blocks can be best combined: 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].

In GP, O'Reilly and Oppacher [1995] attempted an analogy to the GA schemata theory. A main goal of that work was to understand whether GP problems have building block structure but the results were also inconclusive. 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, program fitness is determined by matching a set of patterns against the program and adding up the predefined fitness of each pattern. By ignoring the semantic evaluation step, the analysis of constructional problems is not generalizable to typical GP problems. GP presents a challenging picture due to the functional representation it generally uses.

An analysis of block processing in GP has to rely on the *function* of blocks of code. GP modularization approaches consider the effect of encapsulating and possibly generalizing blocks of code.

The first approach to modularization in GP was *encapsulation*[1], introduced in [Koza, 1992]. Refinements or extensions of the encapsulation concept have focused on different aspects of function definition. Three main approaches to modularization discussed in the GP literature are automatically defined functions (ADF) [Koza, 1994b], module acquisition (MA) [Angeline, 1994], and adaptive representation (AR) [Rosca and Ballard, 1994a]. In the ADF approach individuals are represented by a set of subroutines and a main function[2]. ADF co-evolves representations for the components of the hierarchy of subroutines implementing a program. In MA, pieces of code called *modules* are frozen from manipulation as a result of *compress operations* and are kept in a global genetic library[3]. AR [Rosca and Ballard, 1994a] explicitly attempts to discover and use new good functions or subroutines. Subroutines are created from blocks identified as good by user supplied *block fitness functions*.

In the above methods, selection of programs to participate in reproduction operations is fitness-proportional. However, in ADF and MA selection of blocks of code within programs is purely random or "uninformed." Uniform random changes at all levels may determine a loss of beneficial evolutionary changes. For instance, ADF samples the space of subroutines by modifying automatically defined functions at *randomly* chosen crossover points. Similarly, MA *randomly* selects a subtree from an individual and *randomly* chops its branches to define a module and thus decide which genetic material is frozen. All points of a tree, be they good or bad, active (i.e. effective during evaluation) or inactive (i.e. introns; for a discussion of introns in GP see [Nordin et al., 1995] or Chapter 6 in this book) are equally likely to be the source of a compress operation.

Random changes will not be an efficient strategy if the *bottom-up evolution hypothesis*

[1]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 (as a function with zero arguments) and used later.

[2]Each component of the representation of an individual has its own "vocabulary" (function and terminal sets). Subroutine terminal sets may also include formal argument symbols. Crossover is only performed between analogous components, constraint called *branch typing*.

[3]More precisely, a module is a piece of code obtained by randomly choosing a subtree and randomly chopping off its branches. A module can be *decompressed* by an additional genetic operation which has a complementary effect to compress. The genetic library has a passive role.

[Rosca and Ballard, 1995] holds. This theory conjectures that ADF subroutine representations become stable in a bottom-up fashion. Early in the process changes are focused towards the evolution of low level subroutines. Later in the process the changes are focused towards the evolution of program control structures, that is structures at higher levels in the hierarchy of subroutines [Rosca and Ballard, 1995]. For this reason we have focused on *informed* or adaptive measures to guide the choice of crossover points[4] and the creation and modification of subroutines.

9.3 Adaptive Representation

The central idea of an adaptive representation system is to find and use subroutines based on measures of their function. Reusing good building blocks has obvious advantages in terms of the economizing the search process. A larger set of functions positively affects the fitness distribution of programs created through initialization or genetic operators [Rosca, 1995c]. Thus the use of subroutines biases where genetic search focuses in the space of programs (see also [Whigham, 1995]). This idea is implemented in a simple form in the adaptive representation approach [Rosca and Ballard, 1994a].

AR extends GP search with three steps whose overall result is the creation of new subroutines that extend the problem representation:

- Identify useful blocks of code that appear as a result of genetic operations. An *informed*, or heuristic technique, was employed in the selection of what could be useful blocks of code. Usefulness is given by a user supplied block fitness function.

- Generalize the blocks that withstand the selection criterion above using inductive generalization [Michalski, 1983]. The result is a set of new subroutines which extend the current function set.

- Create a number of random individuals from the extended function set and replace low-fitness individuals (thus exploit newly created functions).

The critical problem in AR is the evaluation of the usefulness of a block of code (the first step above). 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, 1994a], the block's constructional or schema fitness [Altenberg, 1994], or conditional expected fitness [Tackett, 1995] is not sufficient because either the methods are not functionally motivated or they are impractical

[4]See also Chapters 3, 4, and 5 authored by and Teller, Angeline, Iba and de Garis in this volume.

for use in on-line computation. The second and third steps above are preserved in the ARL algorithm and will be described in detail later.

AR does not delete subroutines. Provided that the block selection heuristics are good, the procedure creates *stable*, useful subroutines. This focuses search towards desirable regions in the search space leading to improved overall efficiency and scalability [Rosca and Ballard, 1994b]. In general some subroutines may be bad guesses or may only present a temporary advantage. The algorithm should learn which subroutines to delete and which to keep around.

9.4 Learning Good Subroutines: ARL

This section answers the “what” question by showing how both local and global information implicitly stored in the population can be exploited. Local information is brought to bear based on the notions of differential fitness and block activation. Global information is used to define subroutine utility.

9.4.1 The ARL Strategy

The central idea of the ARL algorithm as well as of AR is the dynamic adaptation in the problem vocabulary. The vocabulary at generation t is given by the union of the terminal set $\mathcal{T}$, the function set $\mathcal{F}$[5], and a set of evolved subroutines $\mathcal{S}_t$.

$\mathcal{T} \cup \mathcal{F}$ represents the set of primitives which is fixed throughout the evolutionary process. In contrast, $\mathcal{S}_t$ is a set of subroutines whose composition may vary from one generation to another. $\mathcal{S}_t$ may be viewed as a population of subroutines that extends the representation vocabulary in an adaptive manner. Subroutines compete against one another but may also cooperate for survival, as will be described below. New subroutines are discovered 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 some individuals in the next generation, $t + 1$.

The ARL algorithm attempts to automatically discover useful subroutines and adapt the set $\mathcal{S}_t$ by applying the heuristic “*pieces of useful code may be generalized and successfully applied in more general contexts.*”

9.4.2 Discovery of Useful Subroutines

New subroutines are created using blocks of genetic material from the pool given by the current population. The major issue here is the detection of what are “useful” blocks of

[5]Function with zero arguments have been usually treated as terminals in the GP literature. In our implementation, the only terminals are variable symbols or numbers. Functions with zero or more arguments are part of the function set. This distinction is natural in LISP because any S-expression representing a function call is a (non-void) list.

```
Subroutine-Discovery($\mathcal{P}_t, \mathcal{S}^{new}, \mathcal{P}_t^{dup}$)
1. Initialize the set of new subroutines $\mathcal{S}^{new} = \emptyset$.
   Initialize the set of duplicate individuals $\mathcal{P}_t^{dup} = \emptyset$
2. Select a subset of promising individuals: $\mathcal{B} = max_i\{\text{DiffFitness}(i)\} > 0$
3. Label each node of program $i \in \mathcal{B}$ with the number of activations in the evaluation of $i$ on
   all fitness cases.
4. Create a set of candidate building blocks $\mathcal{BB}_t$ by selecting all blocks of small height and
   high activation($\mathcal{B}$);
5. Prune the set of candidates($\mathcal{BB}_t$) by eliminating all blocks having inactive nodes;
6. For each block in the candidate set, $b \in \mathcal{BB}_t$, repeat:
   (a) Let $b$ belong to program $i$. Generalize the code of block $b$:
       i. Determine the terminal subset $T_b$ used in the block($b$);
       ii. Create a new subroutine $s$ having as parameters a random subset of $T_b$ and as body
           the block($b, T_b$);
       iii. Create a new program $p^{dup}$ making a copy of $i$ having block $b$ replaced with an
            invocation of the new subroutine $s$. The actual parameters of the call to $s$ are given
            by the replaced terminals.
   (b) Update $\mathcal{S}^{new}$ and $\mathcal{P}_t^{dup}$:
       i. $\mathcal{S}^{new} = \mathcal{S}^{new} \cup \{s\}$
       ii. $\mathcal{P}_t^{dup} = \mathcal{P}_t^{dup} \cup \{p^{dup}\}$
7. Results $\mathcal{S}^{new}, \mathcal{P}_t^{dup}$
```

Figure 9.1
ARL extension to GP: the subroutine discovery algorithm for adapting the problem representation.

code. The notion of usefulness in the subroutine discovery heuristic is defined by two concepts, *differential fitness*, and *block activation*. The subroutine discovery algorithm is presented in Figure 9.1.

The nature of GP is that programs that contain useful code will tend to have a higher fitness and consequently their offspring will tend to dominate the population. The concept of differential fitness is a heuristic which anticipates this trend and focuses on blocks from such individuals. Thus blocks of code are selected from programs that have the biggest fitness improvement over their least fit parent, i.e. the highest *differential fitness*. Let i be a program in the population having raw fitness $Fitness(i)$. Its differential fitness is defined as:

$$\text{DiffFitness}(i) = \text{Fitness}(i) - min_{p \in \text{Parents}(i)}\{\text{Fitness}(p)\} \tag{9.1}$$

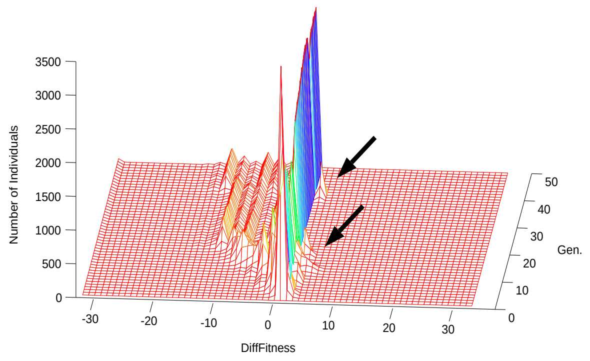

Figure 9.2
Differential fitness distributions over a run of GP. At each generation, only a small fraction of the population has DiffFitness > 0 and is indicated by arrows here.

We focus on the population program i having the following property:

$$max_i\{\text{DiffFitness}(i)\} > 0 \qquad (9.2)$$

Large 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 9.2 shows the histogram of the differential fitness defined above for a run of GP on the Even-5-Parity problem. Each slice of the plot for a fixed generation represents the number of individuals (in a population of size 4000) vs. differential fitness values. The figure suggests that a small number of individuals improve on the fitness on their parents (to the right of the "neutral wall" defined by DiffFitness $= 0$). ARL will focus on such individuals in order to discover salient blocks of code.

Once candidate parents have been selected, the next step is to identify useful blocks of code within those parents. During repeated program evaluation, some blocks of code are executed more often than others. The more 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 from individuals with the highest differential fitness. In contrast to [Tackett, 1995], salient blocks have to be detected efficiently, on-line. This is possible because candidate blocks are only searched for among the blocks of small height (between 3 and 5 in the current implementation) present in individuals with the highest differential fitness. 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 and hitch-hiking phenomena [Tackett, 1995]. It is represented by the pruning step (5) in Figure 9.1.

The final step is to formalize the block as a new subroutine and add it to the GP vocabulary. Blocks are generalized by replacing some random subset of terminals in the block with variables (see Step 6a in Figure 9.1). Variables become formal arguments of the subroutine created. The generalization operation makes sense in the case when the primitive symbols satisfy the closure condition [Koza, 1992], i.e. they can be functionally combined in every possible way. In strongly-typed GP [Montana, 1994] each variable or constant has a type, and each function has a *signature*. The function signature is defined by the type of the function result and by the formal argument types. Block generalization in typed GP additionally assigns a signature to each subroutine created. The subroutine signature is defined by the type of the function that labels the root of the block and the types of the terminals selected to be substituted by variables. Signatures of all primitives and new subroutines represent the new genetic composition constraints.

9.4.3 Subroutine Utility

ARL expands the set of subroutines $\mathcal{S}_t$ whenever it discovers new subroutine candidates. All subroutines in $\mathcal{S}_t$ are assigned utility values which are updated every generation. A subroutine's utility is estimated by observing the outcome of using it. This is done by accumulating, as *reward*, the average fitness values of all programs that have invoked s over the *past generations*, directly or indirectly (by calling other subroutines that call s directly or indirectly). The subroutine utility is analogous to schema fitness. However, reward accumulation[6] is done over a fixed time window of W generations. Thus for a subroutine s, its utility $U(s)$ is:

$$U(s) = K^{-1} \sum_{t-W}^{t} \sum_{j} F(j) \tag{9.3}$$

where j is a program that invokes s and K is a normalizing constant.

In a hierarchy of subroutines, good subroutines higher in the hierarchy may reinforce

[6] *Undiscounted past rewards* are currently used to estimate the fitness of subroutines. Reinforcement learning (RL) algorithms such as Q-learning [Watkins, 1989] uses temporal discounting of *future expected rewards*. Temporal discounting could be also used here. Undiscounted rewards have been used because the method is simpler, is analogous to schema fitness, and does not favor current mediocre programs that may invoke a subroutine. R-learning is an undiscounted RL algorithm [Schwartz, 1993] that outlines some of the advantages of undiscounted dynamic programming methods.

other subroutines lower in the hierarchy, so programs may “cooperate” for survival. If we define the raw utility of s, $\hat{U}(s)$, as the average fitness of all programs directly invoking it, the utility of s, $U(s)$, is equivalent to the following algebraic form:

$$U(s) = \lambda_0 \hat{U}(s) + \sum_j \lambda_j \cdot U(s_j) \tag{9.4}$$

where s_j is a subroutine that invokes s and λ_j is a subunitary weighting factor representing the fraction of all programs calling s indirectly through s_j ($j \geq 1$) or directly ($j = 0$). This formula shows that if s is a good subroutine and a particular subroutine s_j invokes s often, then its utility will also be higher.

The set of subroutines co-evolves with the main population of solutions through creation and deletion operations. New subroutines are automatically created based on active blocks as described before. Low utility subroutines are deleted in order to keep the total number of subroutines below a given number. In order to preserve the functionality of those programs invoking a deleted subroutine, calls to the deleted subroutine are substituted with the actual body of the subroutine as in an in-line substitution operation.

The general structure of the ARL algorithm is given in Figure 9.3. It extends a standard GP algorithm with one new step ($3b$) implementing the adaptation of the problem vocabulary.

9.5 When to Create Subroutines: Using Entropy

The use of functional subroutines in the GP population has the effect of preserving a higher population diversity over generations in comparison to standard GP [Rosca, 1995c]. An increased diversity may result in a more effective search, by escaping local optima. The notion of diversity in GP is involved in an answer to the “when” question. In general, there are several possible answers to the “when” question. New subroutines can be created:

1. Whenever the subroutine discovery algorithm suggests new subroutines.

2. At the end of epochs, i.e. periods of consecutive generations throughout which the GP system works like standard GP using a fixed representation system.

3. Adaptively, in response to long term decreases in population diversity.

There are several justifications for not attempting to adapt the problem representation all the time: (1) The computational overload introduced for tracking blocks of code; (2) The goal to exploit the current set of subroutines $\mathcal{S}_t$ (the vocabulary would be changed only if no progress is made using it); (3) The poor performance of candidate solutions in early generations (progress is probable in early generations anyway).

```
ARL-Skeleton
Define problem: terminal set $\mathcal{T}$, function set $\mathcal{F}$, fitness function $F$, set of training cases $\mathcal{E}$.
Denote the population at a given generation $t$ by $\mathcal{P}_t$.
1. Initial Generation: evolution time (generation) $t = 0$, discovered subroutines $\mathcal{S}_0 = \emptyset$
2. Randomly initialize population $\mathcal{P}_0(\mathcal{T} \bigcup \mathcal{F}, \mathcal{P}_0)$
3. Repeat until termination criterion is met:
    (a) Evaluate population($\mathcal{P}_t, \mathcal{E}, F$)
    (b) Adapt representation($\mathcal{P}_t, \mathcal{S}_t, \mathcal{S}_{t+1}, \mathcal{P}_t^{dup}, \mathcal{P}_t^{new}$)
        i. Discover new subroutines $\mathcal{S}^{new}$ and create duplicate individuals $\mathcal{P}_t^{dup}$ by calling
           Subroutine-Discovery($\mathcal{P}_t, \mathcal{S}^{new}, \mathcal{P}_t^{dup}$)
        ii. Update subroutine utilities ($\mathcal{S}_t, \mathcal{S}^{new}$)
        iii. Create the next generation subroutine set $\mathcal{S}_{t+1}$:
            A. Select subroutines of low utility to be deleted $\mathcal{S}^{old}(\mathcal{S}_t, \mathcal{S}^{new})$
            B. $\mathcal{S}_{t+1} = (\mathcal{S}_t - \mathcal{S}^{old}) \bigcup \mathcal{S}^{new}$
        iv. Randomly generate and evaluate newborns $\mathcal{P}_t^{new}$ ($\mathcal{T} \bigcup \mathcal{F} \bigcup \mathcal{S}_{t+1}, \mathcal{P}_t^{new}, \mathcal{E}, F$)
        v. Evaluate population of newborns($\mathcal{P}_t^{new}, \mathcal{E}, F$)
    (c) Generate a new population $\mathcal{P}_{t+1}$ by fitness proportionate reproduction, crossover, mutation of individuals($\mathcal{P}_t, \mathcal{P}_t^{dup}, \mathcal{P}_t^{new}, \mathcal{P}_{t+1}$)
        i. Create intermediate population $\mathcal{P}_t^{'} = \mathcal{P}_t \bigcup \mathcal{P}_t^{dup} \bigcup \mathcal{P}_t^{new}$
        ii. Select genetic operation $O$ ($p_r, p_c, p_m, O$)
        iii. Select winning individuals $\mathcal{W}$ from the intermediate population ($O, \mathcal{P}_t^{'}, \mathcal{W}$)
        iv. Generate offspring $\mathcal{P}_{t+1}(O, \mathcal{W}, \mathcal{P}_{t+1})$
    (d) Next generation: $t = t + 1$
```

Figure 9.3
The ARL algorithm extends the standard GP algorithm with Step $3b$ which adapts the problem representation system (vocabulary) by creating new subroutines, eventually deleting old ones and creating new individuals to be entered in the selection competition.

An appropriate measure of diversity is population entropy [Rosca, 1995b]. The entropy measure represents a global measure for describing the state of the dynamical system represented by the population in analogy to the state of a physical or informational system. Population entropy can be computed by grouping individuals into classes according to their behavior or phenotype and determining the number of individuals that belong to each class, according to Shannon's information entropy formula [Shannon, 1949]:

$$E(P) = -\sum_{k} p_k \cdot \log(p_k) \tag{9.5}$$

where p_k is the proportion of the population P occupied by population partition k at a given time. In GP a useful class of the partition is a fixed interval of fitness values. Individuals are regarded as equivalent if their fitness values lie in the same interval regardless of differences in their code.

Entropy provides a way to track diversity during a GP run. Decreases in population diversity can be correlated with a plateau in the best-of-generation fitness or a plateau in the average fitness over a fraction of the best individuals in the population. Such correlations suggest convergence towards a local optimum [Rosca, 1995b]. They can be used to decide when to create new subroutines.

9.6 Test Case: Controlling an Agent in a Dynamic Environment

We have tested the ARL algorithm on three problem domains: Boolean regression, symbolic regression, and controlling an agent in a dynamic environment (similar to the Pac-Man problem), all described in detail in [Koza, 1992]. In this chapter we will focus on the last of these problems.

9.6.1 The Pac-Man Problem

An agent, called Pac-Man, can be controlled to act in a maze of corridors. Up to four monsters chase Pac-Man most of the time. Food pellets, energizers and fruit objects result in rewards (game points) when reached by Pac-Man. After eating an energizer (also called "pill") Pac-Man can chase monsters in its turn, for a limited time. The problem is to evolve a controller to drive the Pac-Man agent in order to acquire as many points as possible. A snap-shot of the Pac-Man world is presented in Figure 9.4.

A solution (also called policy or agent function) to the problem is a program that controls Pac-Man movements based on current sensor readings, and possibly past sensor readings and internal state (memory). It maps states into actions or sequences of actions. Such a program is an implicit representation of the agent policy and can be evolved by means of

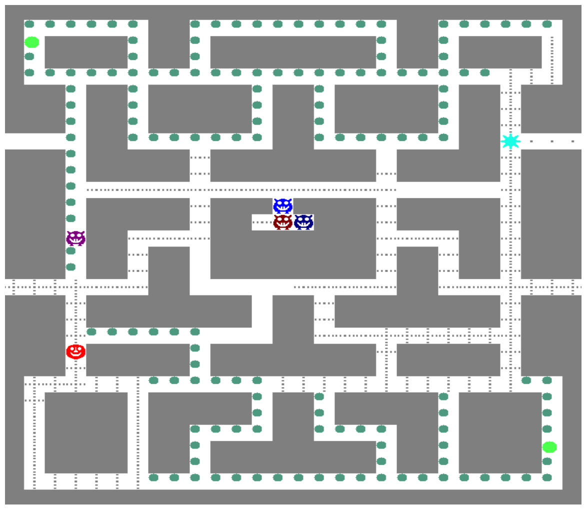

Figure 9.4
An example of the Pac-Man trajectory for an evolved program. The trace of Pac-Man is marked with vertical dots. The monster traces are marked with horizontal dots. Pac-Man started between the two upside-down T-shaped walls (bottom) while the four monsters were in the central hen. Pac-Man headed North-East, captured a fruit and the pill there, and then attracted the monsters in the South-West corner. There it ate the pill and just captured three of the monsters (to be reborn in the hen). Next it will closely chase the fourth monster.

GP. But how good can evolved solutions get?

9.6.2 Genetic Programming Implementations

The Pac-Man problem primitives chosen for the GP implementation in [Koza, 1992] are sufficiently high level in order to focus on a single game aspect, that of task prioritization. In this chapter we describe experiments with three representations of the Pac-Man problem. Problem representation A1 is from [Koza, 1992]. Representation A2 uses the same vocabulary as representation A1 but changes the result returned by some primitives. We refer to either A1 or A2 by A. Representation B uses a typed vocabulary by taking into account the *signature* of each primitive, i.e. the return type of each function (subroutine) as well as the types of its arguments. It introduces primitives for evolving explicit logical conditions under which actions can be executed.

9.6.2.1 Representation A1

The function set contains two conditional operators, five perception primitives, and eight action primitives. IFB (if-blue) senses if monsters are blue and when true executes its first argument otherwise executes its second argument. IFLTE (if-less-than-or-equal) compares its first argument to its second argument. For a "less-than" result the third argument is executed. For a "greater-or-equal" result the fourth argument is executed. The perception primitives return the Manhattan distance to the closest food pellet, pill, fruit and closest or second closest monster. They are, respectively: SENSE-DIS-FOOD, SENSE-DIS-PILL, SENSE-DIS-FRUIT, SENSE-DIS-MON1, SENSE-DIS-MON2. The terminal set has no elements.

The action primitives move the agent along maze corridors. All return a number encoding the direction faced by the agent. For instance, ACT-A-PILL advances the agent on the shortest path to the nearest uneaten energizer while ACT-R-PILL retreats the agent from the nearest uneaten energizer. The other actions have analogous functions with respect to the closest monster, the second closest monster, fruit, and food: ACT-A-MON-1, ACT-R-MON-1, ACT-A-MON-2, ACT-R-MON-2, ACT-A-FRUIT, ACT-A-FOOD. If the shortest path or closest monster or food are not uniquely defined, then a random choice from the valid ones is returned by the corresponding function. A program is evaluated based on the performance of the agent on an initial world configuration.

9.6.2.2 Representation A2

A2 is A1 modified by making all the action primitives return the distance from the corresponding element. This addresses the problem that distances and directions in A1 are mixed without making much resulting sense. GP using A2 cannot mix distances and directions. In contrast, GP using A1 is free to mix them if this provides an evolutionary advantage.

9.6.2.3 Representation B - Typed GP

In problem representations A1 and A2, actions may appear both in the condition and in the action part of an IFLTE expression. The evaluation of the condition changes the context where the action is executed. This makes it extremely difficult to understand what an evolved program really does without executing it.

For analyzing GP, it is desirable that evolved programs express explicit conditions under which certain actions are prescribed. To do this, we used a typed GP system [Montana, 1994, Johnson et al., 1994] based on an extended set of primitives obtained from A2. The problem representation A2 is extended with relational operators ($<, =, \geq, \neq$), logical operators (AND, OR, NOT), and generic functions returning random integer constants. IFLTE is generalized into an if-then-else function, IFTE, taking three arguments. IFTE evaluates its first argument (of BOOLEAN type) and, depending on the result, executes either its second or its third argument (of ACTION type). Conditions under which actions are executed can be evolved using the relational and boolean functions that have been added.

Table 9.1 presents a summary of the typed primitive vocabulary.

Table 9.1
Representative signatures (the return type of each function and the types of its arguments) in the typed Pac-Man vocabulary (B). The primitive vocabulary contains two control functions (IFB is also a perception function), two generic functions for creating random integers, three logic functions (AND, OR, and NOT), eight comparison functions (two instances from each of $<$, $\geq$, $\neq$ and $=$, operating on distances and directions respectively), nine perception functions (returning distances and directions to food, fruit, closest monster, second closest monster and sense distance to food) and eight action functions (advance to or retreat from closest pill, monsters, fruit, and food).

Function	Category	Type	No. Args	Argument Types	Description
IFB	Control	ACT	2	ACT, ACT	if monsters are blue
IFTE	Control	ACT	3	BOOL, ACT, ACT	if-then-else
RAND-DIS	Constant	DIS	0	–	random distance
RAND-DIRECT	Constant	DIRECT	0	–	random direction
AND	Logical	BOOL	2	BOOL, BOOL	logical and
$<$	Logical	BOOL	2	DIS, DIS	distance comparison
$<$	Logical	BOOL	2	DIRECT, DIRECT	direction comparison
SENSE-DIS-FOOD	Perception	DIS	0	–	sense distance to food
ACT-A-PILL	Action	ACT	0	–	advance to closest pill

9.6.3 Representation A versus Representation B

By default, any control decision (IFB, IFLTE, IFTE) or agent perceptual action ($sense$) takes zero time and any agent movement action (act) takes one time unit. Monsters and fruit move synchronously with the agent. A solution is interpreted repeatedly until Pac-Man is captured by a monster or eats all food pellets.

Two apparent advantages of vocabulary B over vocabulary A are comprehensibility and modularity. We present simple examples of evolved programs in order to outline the differences between A and B and suggest limitations of each alternative.

A very simple program evolved by GP using vocabulary A2 was:

```
(IFLTE (IFLTE (ACT-A-PILL) (SENSE-DIS-PILL)
              (ACT-A-PILL) (ACT-R-MON-1)) (ACT-A-FOOD)
    (SENSE-DIS-MON-1) (ACT-A-FOOD))
```

This program has a very non-intuitive behavior: "If the *direction* while advancing to the closest energizer is less than the *distance* to the closest energizer then advance to the energizer else retreat from the closest monster. Additionally, if the result of the advance or retreat action above is greater or equal than the distance to the closest monster then advance to the closest food." Note that the evaluation of the condition part of the IFLTE branches has side effects on the program's state, making hard to understand what the program does without actually executing it. This programs acquires just 516 points on average for 100 simulations (each simulation reevaluates the program 23 times on average until the agent is captured by monsters).

A second program, written in representation B, is easy to understand as it is memoryless. It decides to advance to the fruit if a fruit exists, retreat from closest monster or second closest monster (if close), otherwise advance to food. This program acquires 960 points on average under the same conditions (the program is evaluated 53 times on average).

```
(IFB (ACT-A-MON-1)
   (IFTE (< (SENSE-DIS-FRUIT) 100)
      (ACT-A-FRUIT)
      (IFTE (AND (<(SENSE-DIS-MON-1)5)(<(SENSE-DIS-MON-2)7))
        (ACT-R-MON-1) (ACT-A-FOOD))))
```

The power of the representation systems used is unequal. Programs written using one of the A1 or A2 vocabularies implicitly develop *memory* or *state*[7]. For instance, certain actions will be executed in sequence during the phase of testing conditions for an IFLTE operation. Thus, some action subsequences will only be executed if other actions are executed before, and this only happens in certain states memorized in the code itself. GP can develop very intricate solutions that can be hardly understood by humans.

In contrast, a solution expressed using vocabulary B is *memoryless*. It activates exactly one ACT leaf node based only on the current perception of the world. The inner nodes of type BOOL test applicability conditions for subsequent type ACT branches. Type ACT leaves are the agent's overt actions. Only one such leaf can be reached during the execution of a solution. Programs evolved are equivalent to decision trees and are easier to understand. Understandability is not a requirement in GP. However in this test case we want to understand the semantics of evolved subroutines.

A representation system using memory is theoretically more powerful than a memoryless one. We will explore whether GP can take advantage of this form of memory.

9.6.4 Problem Difficulty, Fitness Cases and the Fitness Measure

The Pac-Man problem has a number of major sources of difficulty. First, Pac-Man is an active agent in a dynamic environment, thus the number of world states that it can encounter during one simulation is huge. Second, the degree of perceptual aliasing (or hidden state) is high. Third, the problem has several sources of nondeterminism:

- monster moves are random 20% of the time.
- fruit moves are occasionally random.

[7]For a discussion of current research on state in GP see [Andre, 1995].

Finding an optimal control policy would be intractable even for a deterministic environment. Difficulty raises questions such as what are good solutions and how much training is needed to evolve good solutions. Training effort is measured by the number of simulations executed or the number of primitives executed.

Each simulation of an evolved program controller starts in the same initial world state. A number of training, or fitness cases is considered. Each training case corresponds to one simulation. Multiple simulations of the same program have different outcomes due to random events in the external environment. The actual sequence of random events for a simulation is controlled by a random number generator.

The standardized fitness measure of a program i is given by:

$$\text{StdFitness}(i) = \text{MAXPOINTS} - \text{Fitness}(i) \tag{9.6}$$

where Fitness(i) is the average number of points or "hits" (raw fitness) accumulated by the agent under the control of program i on the set of fitness cases and MAXPOINTS is the theoretical maximum number of points that can be obtained if the agent gathers all food pellets, all fruit, all energizers and the maximum number of monsters (four) in each of the four blue periods generated by eating all four energizers.

Random events in the simulation of a program result in huge variations in problem difficulty and thus determine huge variations in fitness. For example, for one sequence of random choices generated, the monsters may encircle Pac-Man making it impossible for him to escape. For another random sequence, all monsters may crowd on the same side of Pac-Man, making a full win from a simple strategy of attracting monsters towards an energizer and eating the energizer at the right time. In order to evolve general solutions, a large number of fitness cases should be considered. This implies an increased computational effort for each individual fitness evaluation. Reynolds [1992] discussed this typical speed-accuracy trade-off for the problem of evolving a "corridor following" control program. He defined $Fitness(i)$ as the *minimum* number of simulation steps, over several fitness (i.e. training) cases, taken before the first collision of the agent.

9.7 Results

The experiments performed examined the ability of the ARL algorithm to adapt its vocabulary, i.e. discover useful subroutines and to use them in a beneficial way for generating problem solutions. Also, we compared the performance of different GP algorithms (standard GP, ADF, ARL) on various problem representations. We also compared human designed solutions with the computer generated solutions.

In all experiments the population size was 500 and the algorithm was run for 50 or 100

Table 9.2
Features of Pac-Man solutions obtained with standard GP (SGP), ADF, ARL and carefully hand-coded programs using three different problem representations (A1, A2 and B). The main GP parameters are: Population size = 500; Number of generations = 100 (SGP, ADF) and 50 (ARL); Crossover rate = 90% (20% on leaf nodes); Reproduction rate = 10%; No mutation. During evolution, fitness is determined from one training case (columns marked '1') or by averaging simulation scores on three training cases (columns marked '3'). Each table entry shows the fitness of an evolved Pac-Man controller, the generation when that solution was discovered, and its structural complexity (/ expanded structural complexity, for programs with subroutines; see [Rosca 1995a]).

Representation		SGP		ADF		ARL		Hand-
		1	3	1	3	1	3	coded
A1	Fitness	6380	4293	6830	5743	9840	5793	-
	Generation	67	17	42	99	20	36	
	Complexity	149	21	3/125	123/7515	55/398	85/95	
A2	Fitness	7873	6353	13650	5070	12500	5840	7470
	Generation	99	97	38	9	22	29	
	Complexity	143	91	21/45	23/93	159/175	115/145	15
B	Fitness	6540	5427	6290	4653	6540	5840	5910
	Generation	10	80	33	9	33	10	
	Complexity	35	186	88/88	15/15	25/53	27/39	9/44

generations. Other GP parameters were chosen as in [Koza, 1994b] (see Table 9.2).

One or three training cases were considered for fitness computation. Distinct fitness cases are generated by controlling the periods when monsters or fruit move randomly. Generalization was tested by evaluating the evolved or hand-written programs on cases not used during training.

The additional parameters of the ARL algorithm were the time window for cumulating subroutine fitness ($W = 10$) and the size of the set of subroutines (10). No experiments were attempted to tune the values of these parameters. The ARL vocabulary was continuously updated. Also, entropy computation was based on a partitioning of the possible range of fitness values in 183 intervals of size 100.

Tables 9.2 and 9.3 show a sample of results for the best solutions evolved using standard GP, ADF, ARL (over four independent runs for each experiment) and hand coding. Several conclusions can be drawn from these two tables. Overall, ARL (which ran for 50 generations) obtained better solutions faster than SGP and ADF (which ran for 100 generations). More importantly, the ARL solutions are more general than all other solutions. Surprisingly, representation B (the least "powerful" but the most easy to understand) did best on this aspect. Also, the most general solutions obtained had a low structural complexity. Last, but not least, evolved solutions are more fit on average than the best hand-coded programs.

Table 9.3
Generalization performance of Pac-Man solutions from Table 9.2: average fitness of evolved solutions, trained on one or three cases, over 100 random test cases.

	SGP		ADF		ARL		Hand
Representation	1	3	1	3	1	3	coded
A1	2255	2212	1571	1005	1053	3378	-
A2	3095	2906	1011	1569	3594	3611	1798
B	595	3370	2696	3875	3706	4439	1009

The best ARL/ADF-evolved programs for representation A2 had 12500 and 13650 points respectively. The ADF solution had poor generality (see Table 9.3) as it became specialized on the deterministic environment used for training (one fitness case). The ARL solution performs much better on average over 100 random test cases. The subroutines evolved by ARL and ADF are very hard to comprehend.

Representation B enabled us to understand the effects of modularization in ARL. The best-of-generation program evolved by ARL for the corresponding run in Table 9.2 is extremely modular, relying on six useful subroutines that form a four-layer hierarchy on top of the primitive functions. Each of the six subroutines is effective in guiding Pac-Man for certain periods of time. Among the six subroutines, two subroutines define interesting “behaviors.” For example, S1749 is successfully used for attracting monsters. S1765, invoked with actual parameter values of 19 and 21, defines a fruit-chasing behavior for blue periods. The others are a predicate for testing if a fruit exists, a food-hunting behavior, a pill-hunting behavior, and a higher level behavior difficult to understand. The simplified code of the first two subroutines is:

```
[S1749]: (LAMBDA ()
             (IFTE (< (SENSE-DIS-FRUIT) 50)
                 (ACT-A-FRUIT) (ACT-A-MON-1)))

[S1765]: (LAMBDA (A0)
             (IFTE (< (SENSE-DIS-FRUIT) A0)
                (ACT-A-FRUIT) (ACT-R-PILL)))
```

Figures 9.5 and 9.6 show three quantities of interest, plotted against time (in generations) for two sample runs for GP and ARL. The three quantities are: (1) the fitness (number of points) of the best-of-generation individual; (2) the population average standardized fitness; (3) the population entropy. Entropy as well as the hit distributions for these runs (see Figures 9.7 and 9.8) were computed based on the partitioning of the range of fitness values as mentioned before.

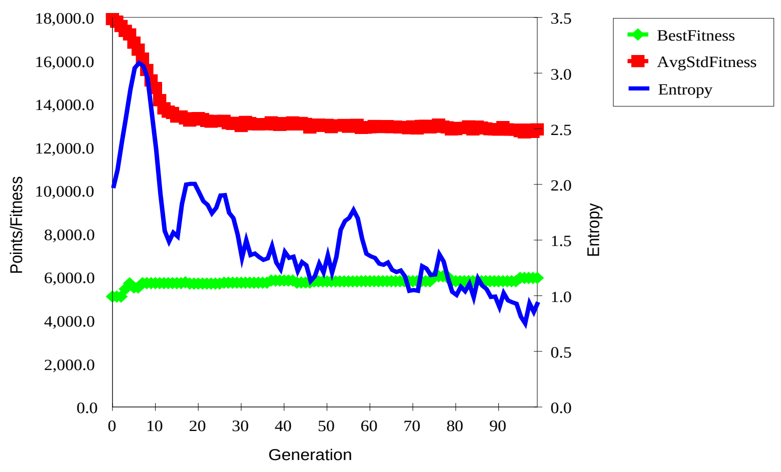

Figure 9.5
SGP learning curves: The fitness of the best-of-generation individual, average standardized fitness and entropy of the population for a SGP run. The decrease in entropy indicates the gradual loss in population diversity. Its correlation with a plateau in the best-of-generation fitness indicates a probable local optimum for genetic search.

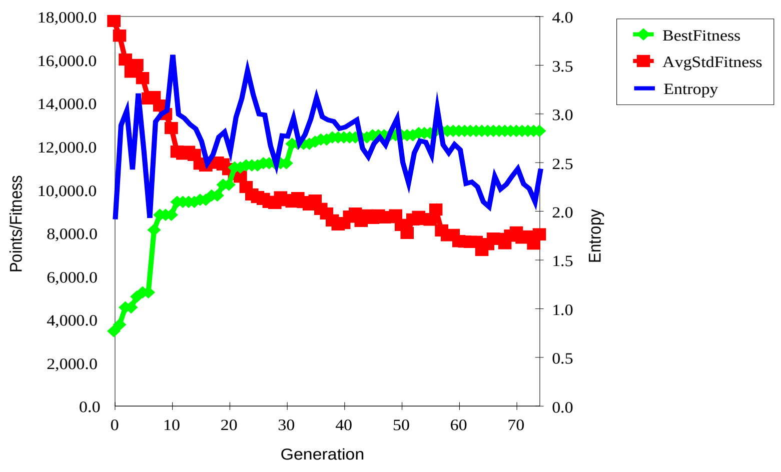

Figure 9.6
ARL learning curves: The fitness of the best-of-generation individual, average standardized fitness and entropy of the population for an ARL run. Discovery of subroutines preserves a high population diversity. The average population standardized fitness rapidly decreases.

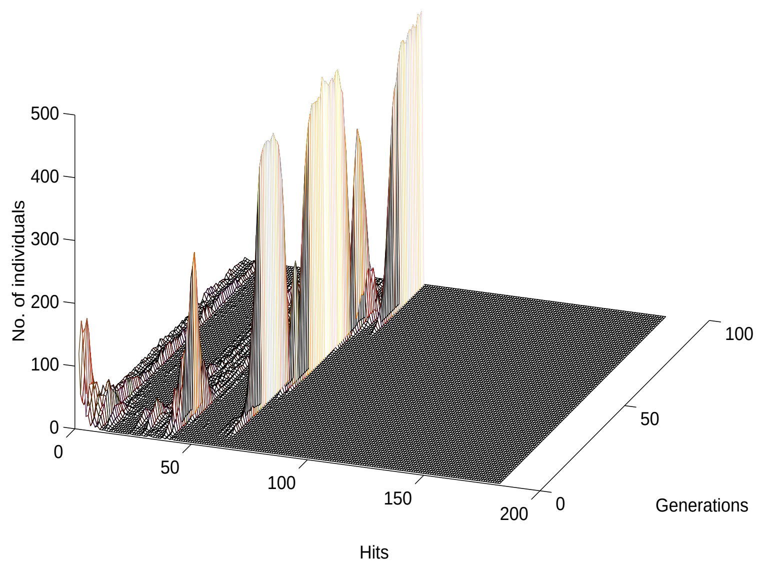

Figure 9.7
Hit distribution for the SGP solution from Figure 9.5. After generation 20, the population becomes dominated by a class of individuals for many generations.

A comparison of the corresponding graphs from Figures 9.5- 9.6 reveals the superiority of ARL over GP. Solutions obtained with ARL perform better than the ones evolved by means of GP or hand-coded, and they are also evolved faster. The GP traces show evidence of premature convergence to local rather than global optima.

The hit distributions and entropy variations are particularly interesting, showing the effects of the change in vocabulary. In the run from Figure 9.8: (1) better solutions are discovered much faster (see also Figure 9.6); (2) population diversity is much higher (see also the entropy plot in Figure 9.6); (3) several discovered subroutines offer just a short term advantage to population individuals and die out; (4) the number of individuals in certain classes grows exponentially fast over a number of generations, enabling the discovery of better subroutines.

Evolved programs outperformed hand-coded controllers. Some GP solutions have poor generalization capability but this may not be surprising taking into account the difficulty of the problem and the small number of fitness cases considered. However, this is also the case with human designed solutions which were written as carefully as possible to be general. Among hand-coded programs, the smallest (made up of only 15 nodes) was also the most general but its generality was behind that of the best evolved programs.

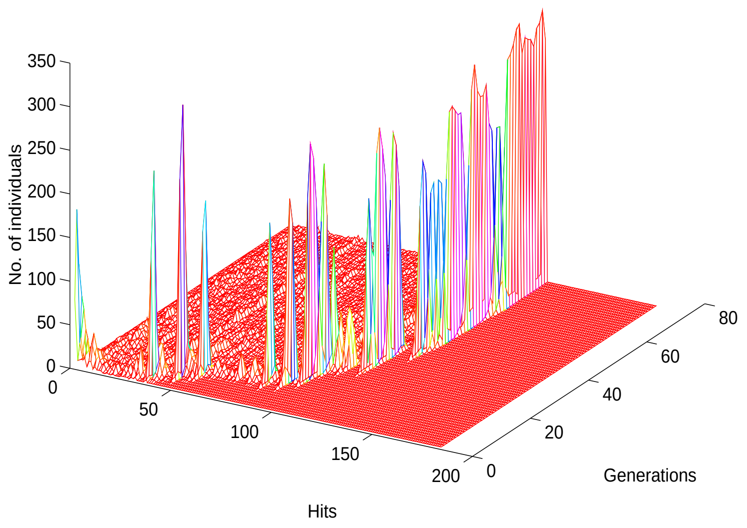

Figure 9.8
Hit distribution for the ARL solution from Figure 9.6. The number of individuals in certain classes grows exponentially. However discovery of subroutines regulates this growth and avoids a loss in diversity by facilitating the rapid discovery of even better individuals.

9.8 Discussion

Although the Pac-Man problem was solved using GP, it is a typical reinforcement learning (RL) task. The task in reinforcement learning is to learn a good satisficing policy mapping states and perceptions into actions.

In the idealized case of a Markovian decision task, the expected future sequence of states is determined by the current agent state and future actions. Q-learning [Watkins, 1989] is a RL algorithm that is guaranteed to converge towards the optimal policy. The agent space for Q-learning in representation A1 or A2 corresponds to 6 perceptions (5 distances of up to 40 units and blue periods) and 8 actions. This results in a huge state space, having a size bigger than 10^9. Q-learning has only been applied to considerably smaller problems.

An omniscient agent would have a complete model of the Pac-Man problem state space. The problem state space is defined by all possible world configurations specified in terms of the positions of all objects in the world (agent, monsters etc.). The world is represented as a graph with 308 vertices and 660 edges. The size obtained in this case is much bigger than the size of the Q-learning agent space. This difference outlines the *perceptual aliasing* (or hidden state) problem [Whitehead, 1991, McCallum, 1995] for the representation chosen. The agent perception does not give it complete information about the current state of the world. The agent does not know how many degrees of freedom it has in the current position and can not directly "reason" about the direction of move, for example.

Problem solutions specified using representation B are memoryless policies. An interesting question is whether satisficing memoryless policies, i.e. policies that can achieve the termination condition goal, exist. Littman [1994] discussed the theoretical limitations of such policies. The problem of finding a satisficing memoryless policy is intractable. The result carries over to more complex cases, such as when the agent has memory or uses state. Descriptions of practical algorithms for finding agent policies in RL, along the lines discussed here, can be found in [Littman, 1994] (the memoryless case) and [McCallum, 1995] (memory-based approaches). However such results can only be applied to small agent spaces.

A classifier system approach to reinforcement learning tasks is ZCS [Wilson, 1994]. The ZCS algorithm combines a sense-act cycle with a credit-assignment or reinforcement procedure and with a genetic algorithm that performs genetic search over the space of condition action rules (classifiers). The policy representations learned using a classifier system like ZCS are, similar to GP solutions, implicit representations. This may allow a higher degree of generalization and scale-up. In contrast, RL or dynamic programming approaches explicitly represent agent states.

GP can cope surprisingly well with the Pac-Man problem. [Koza, 1992] reported a solution that acquired 9420 points after training evolved solutions on only one fitness case. His solution was not tested in environments corresponding to different conditions, so it may not have generality. Although our representation A1 is equivalent to his problem representation, results from the two implementations have different scales due to the use of different Pac-Man simulators.

ARL memoryless solutions (representation B) are interesting and can be easily interpreted. Evolved subroutines form a hierarchy of behaviors. On average, these ARL programs perform better than the ones obtained using ARL on representations A1 or A2. In general, all algorithms evolved solutions of average or low generality. This may not be surprising taking into account that efficiency dictated the use of a few training cases. Evolved programs whose fitness is based on three fitness cases perform better on average than the ones tested on one fitness case. The latter learn more accurately the training case, on which they perform exceptionally for some runs, particularly in representations A1 and A2. Thus, generalization performance was traded for computational efficiency. A theoretical measure such as the VC-dimension [Vapnik, 1982] may tell more on the theoretical limitations of the standard GP or ARL approaches with respect to generalization. Surprisingly, human designed solutions possess even poorer generality.

9.9 Conclusions

"Evolution always finds a way." This truism is reflected in GP through GP's power to quickly make use of advantageous functions and subroutines, and especially to filter out useless functions or subroutines from the representation of population individuals.

The *adaptive representation through learning* (ARL) algorithm takes advantage of this simple but important remark. It extends the GP engine with machinery for adapting the problem vocabulary by creating and using new subroutines. Adaptation is guided by heuristics answering two questions, presented in the introduction as the "what" and "when" issues. The "what" issue is addressed by using *gradient* information that helps focus on beneficial genetic material. This information is represented by the differential fitness of individuals, the activation of blocks of code, and the utility of subroutines. All are domain independent methods. The "what" issue is analyzed using a global measure, population entropy. ARL operates on small blocks of code with three advantages. First, small blocks of code are more general, according to the principle of Ockham's razor. Second, online computation is very efficient. Third, subroutines of any complexity can be generated by iterative application over generations.

This chapter has presented in detail both the operating principles summarized above and the mechanics of the ARL implementation. Programs, as structures on which GP operates, are symbolically expressed as compositions of functions. By applying the above heuristics, the ARL algorithm manages to discover functions which increase the chances of creating better solutions. Further, this chapter has showed ARL at work on the complex problem of controlling an agent in a dynamic and nondeterministic environment abstracted by the Pac-Man game. We compared standard GP, ADF and ARL and also discussed three choices of domain representation. A more thorough comparison among solutions obtained using the GP, ADF and ARL algorithms with respect to search efficiency, scalability and generalization is ongoing.

AR demonstrated the potential to improve problem solving efficiency and to dynamically create hierarchies of subroutines. This chapter shows that it is possible to achieve the predicted effects of AR without relying on additional domain knowledge to estimate which blocks of code are useful. ARL uses more general heuristics to bias genetic search. Note that the block activation heuristic applies to all problems whose GP implementation employs functions evaluated in the *lazy* style, whereby a function's parameters are evaluated only if they are needed (such as IFTE and IFLTE here).

This work suggests a multitude of directions for future research. One idea is to consider other operations on the set of subroutines, in particular causal genetic operations, such as duplication [Koza, 1994a] or a carefully defined mutation. ARL suggests ways of improving the ADF technique. Gradient information about the use of ADFs can be used to control the application of the crossover operator (see also [Rosca and Ballard, 1995]).

From the machine learning perspective it is important to study the generalization of solutions evolved. A comparison with memoryless or memory-based reinforcement learning algorithms is also worthy of further investigation. Finally, an important question to answer is on which classes of problems ARL (and HGP techniques in general) work best.

Acknowledgments

Constructive criticism on previous versions of this chapter came from many people. We particularly thank Peter Angeline, Una-May O'Reiley, Peter Whigham, Hitoshi Iba, Bill Langdon, Astro Teller, Jonas Karlsson, Andrew McCallum, and Brij Massand. This research was sponsored by NIH/PHS research grant no. 1 R24 RR06853-02 and by the National Science Foundation under grant numbered IRI-9406481.

Bibliography

Altenberg, L. (1994). The evolution of evolvability in genetic programming. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, pages 47–74. MIT Press.

Andre, D. (1995). The evolution of agents that build mental models and create simple plans using genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 248–255, San Francisco, CA, USA. Morgan Kaufmann.

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, Cambridge, MA, USA.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goldberg, D. E., Korb, B., and Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA. Second edition (First edition, 1975).

Johnson, M. P., Maes, P., and Darrel, T. (1994). Evolving visual routines. *Artificial Life*, 1(4):373–389.

Jones, T. (1995). Crossover, macromutation and population-based search. *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA95)*, pages 73–80.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. (1994a). 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.

Koza, J. R. (1994b). *Genetic Programming II*. MIT Press.

Littman, M. L. (1994). Memoryless policies: theoretical limitations and practical results. In *Proceedings of the Third International Conference on Simulation and adaptive Behavior: From Animals to Animats*, pages 238–245.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning, An Artificial Intelligence Approach*, pages 83–129. Morgan Kaufmann Publishers, Inc.

Montana, D. J. (1994). Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., Cambridge, MA, USA.

Nordin, P., Francone, F., and Banzhaf, W. (1995). Explicitly defined introns and destructive crossover in genetic programming. In Rosca, J. P., editor, *Proceeedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (NRL TR 95.2, University of Rochester)*, pages 6–22.

O'Reilly, U.-M. and Oppacher, F. (1995). The troubling aspects of a building block hypothesis for genetic programming. In Whitley, L. D. and Vose, M. D., editors, *Foundations of Genetic Algorithms 3*, pages 73–88, San Mateo, CA, USA. Morgan Kaufmann.

Reynolds, C. W. (1994). Evolution of obstacle avoidance behaviour:using noise to promote robust solutions. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 10. MIT Press.

Rosca, J. P. (1995a). An analysis of hierarchical genetic programming. Technical Report 566, University of Rochester, Rochester, New York, USA.

Rosca, J. P. (1995b). Entropy-driven adaptive representation. In Rosca, J. P., editor, *Proceeedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (NRL TR 95.2, University of Rochester)*, pages 23–32.

Rosca, J. P. (1995c). Genetic programming exploratory power and the discovery of functions. In McDonnell, J. R., Reynolds, R. G., and Fogel, D. B., editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 719–736, San Diego, CA, USA. MIT Press.

Rosca, J. P. and Ballard, D. H. (1994a). Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department.

Rosca, J. P. and Ballard, D. H. (1994b). Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 251–258. Morgan Kaufmann.

Rosca, J. P. and Ballard, D. H. (1995). Causality in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 256–263, San Francisco, CA., USA. Morgan Kaufmann.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *The Proceedings of the Tenth International Machine Learning Conference*, pages 298–305. Morgan Kaufmann Publishers, Inc.

Shannon, C. E. (1964 [c1949]). *The mathematical theory of communication*. University of Illinois Press.

Tackett, W. A. (1993). Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann.

Tackett, W. A. (1995). Mining the genetic program. *IEEE Expert.* To appear in: IEEE Expert Special Track on Evolutionary Programming (P.J. Angeline editor).

Vapnik, V. N. (1982). *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, New-York.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University.

Whigham, P. A. (1995). Grammatically-based genetic programming. In Rosca, J. P., editor, *Proceeedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (NRL TR 95.2, University of Rochester)*, pages 33–41.

Whitehead, S. D. (1991). *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY.

Wilson, S. W. (1994). Zcs: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18.