Probabilistic Incremental Program Evolution

vorgelegt von
Diplom-Informatiker
Rafał Sałustowicz
aus Kraków

Von der Fakultät IV–Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 3. April 2003

Berlin 2003

D83

The work described in this thesis was performed at IDSIA: Istituto Dalle Molle di Studi sull'Intelligenza Artificiale in Lugano, Switzerland. It was made possible due to funding of IDSIA, and was supported in part by the Swiss National Fund (SNF) grant 2100-49'144.96 "Long Short-Term Memory".

To my family

Contents

Chapter 1

Introduction

Probabilistic Incremental Program Evolution (PIPE - Sałustowicz and Schmidhuber, 1997a,b,1999a) is a machine learning (ML) technique. Just like other ML techniques such as, e.g., neural networks (see, e.g., Hertz, Krogh, and Palmer (1991), or Bishop (1995) for a review), reinforcement learning (see, e.g., Wiering (1999) for a review), or evolutionary algorithms (see Chapter 3 for a review), PIPE tries to enable computers to solve problems automatically, i.e. to find solutions by "learning" from experience (examples), rather than being explicitly programmed to solve a task. PIPE is an evolutionary optimization algorithm, which employs stochastic models to search for computer programs that embody solutions to given problems.

1.1 Motivation

Solutions to a vast variety of problems can be represented by programs. Especially problems with many regularities in their solutions are interesting for program search. Many regularities allow for high compressibility. Compressible solutions allow for short algorithmic descriptions, where algorithmic description length is measured by Kolmogorov complexity (Solomonoff, 1964; Kolmogorov, 1965; Chaitin, 1969; Li and Vitányi, 1993), i.e. by the length of the shortest program, which produces the solution. In general, the shorter the description of a solution, the smaller the search space, and therefore the shorter the search for the solution. Provided that program runtimes remain short with respect to the overall search time, program search is efficient, if by mapping original solution space into program space the search space becomes smaller.

Additionally, unlike many other ML approaches, program search is also applicable when the size, and the shape of the solution are unknown in advance. Most neural networks (except for growing ones), e.g., require size and constraints (shape) of their weight matrices a priori. Reinforcement learning techniques usually need the approximate size of the solution, e.g., for setting the number of state/action values (Watkins, 1989; Peng and Williams, 1996). Most evolutionary algorithms optimize parameters of pre-shaped mathematical models. With program search, size and shape are part of the solution and not part of the problem description.

Program space, however, is generally a highly discontinuous space. Therefore, gradient descent based optimization methods will usually not be applicable. This leaves us with more general optimization algorithms — a choice of randomized trial and error methods, such as, e.g., random search, stochastic hillclimbing, simulated annealing, or evolutionary algorithms to search program space.

1.2 Origins

Probabilistic Incremental Program Evolution (PIPE) emerges at the intersection of two ML research directions: Probability-Based Program Search (PBPS – Schmidhuber, 1994; Wiering and Schmidhuber, 1996b; Schmidhuber, Zhao, and Schraudolph, 1997a; Schmidhuber, Zhao, and Wiering, 1997b), and Evolutionary Algorithms (EAs – Rechenberg, 1965, 1971; Schwefel, 1965, 1974; Fogel, 1962; Fogel, Owens, and Walsh, 1966; Holland, 1975; Baluja, 1994; Baluja and Caruana, 1995). PBPS algorithms search for programs that embody solutions to problems. They employ variable probability distributions over all possible programs to guide their searches. EAs are optimization algorithms that work with pools of solution candidates. They start with a pool of randomly created solution candidates. Letting solution candidates exchange information about the search space, EAs incrementally create successive pools of solution candidates that better and better solve a problem.

PIPE searches for programs and employs probability distributions to guide its search, just like other PBPS algorithms such as, e.g., Adaptive Levin Search (Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b), or Self-Modifying Probabilistic Learning Algorithms (Schmidhuber, 1994; Schmidhuber et al., 1997a,b). PIPE employs EA-based techniques for incrementally finding better solution candidates, just like Genetic Programming (Cramer, 1985; Dickmanns, Schmidhuber, and Winklhofer, 1987; Koza, 1992), another evolutionary program search algorithm.

Thus PIPE combines two features: It employs PBPS-like probabilistic models of program space and uses EA-based techniques to optimize the models.

1.3 Goals of the Thesis

The goals of the thesis are to present Probabilistic Incremental Program Evolution (PIPE), show that it works in practice, and to describe methods, which make PIPE applicable to a wide variety of problems. Therefore, the thesis focuses on the following areas of interest:

- **Probabilistic Incremental Program Evolution (PIPE).** First we present Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a). Then, to illustrate PIPE's performance capabilities, we benchmark PIPE on a variety of problems ranging from toy problems to complex multiagent tasks.

- **Structured Programs.** With "structured programs", i.e. programs that have constrained shapes, we investigate a way of incorporating a priori knowledge into PIPE. If the shape of the solution is partially known, we can use this information to reduce the size of the search space and thus speed up the search.

- **Memory.** Combining PIPE with memory is essential to make PIPE applicable to many problems. Without memory only problems for which the *Markov property* holds may be solved successfully. The Markov property requires input data to be non-ambiguous. I.e., for each input data point there is always the same output. For all problems, where an input data point may require different outputs depending on the data point's temporal context the Markov property does not hold. Thus memory is crucial to enlarge the scope of problems that can be solved by PIPE.

- **Automatic Task Decomposition.** Complex problems may be too hard to be solved by PIPE within acceptable time. There are essentially two ways of dealing with this problem. We can either enhance PIPE's performance, e.g., as discussed in the "Structured Programs" bullet point above, or we can make the problem simpler. We present an automatic task decomposition method called *filtering* (Sałustowicz and Schmidhuber, 1999b), which successively splits complex tasks into simpler subtasks that can be solved independently. Filtering also decomposes into solvable subtasks the possibly difficult task of integrating all the sub-solutions into a global one. Filtering is an optimization algorithm independent method. In combination with PIPE it makes PIPE applicable to more complex problems than PIPE could solve by itself.

Following the sequence of the above mentioned areas of interest, we hope to achieve our goal, to present PIPE to the reader, show him/her that PIPE is not just of theoretical interest, and thus stimulate further research in this direction.

1.4 Outline and Principal Contributions

In the first part of the dissertation Chapters 2 & 3 briefly review the two main research streams — probability-based program search and evolutionary algorithms — that engendered Probabilistic Incremental Program Evolution (PIPE).

Chapter 2 reviews the field of probability-based program search (PBPS). We start by presenting Levin search (LS – Levin, 1973, 1984). Although LS is not a PBPS algorithm, it can be regarded as a starting point for the development of PBPS algorithms. Then we present two PBPS algorithms — Adaptive Levin Search (Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b) and Self-Modifying Probabilistic Learning Algorithms (SMPLAs – Schmidhuber, 1994; Schmidhuber et al., 1997a,b). Both algorithms document the development of the field and especially SMPLAs build the basis for Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a).

Chapter 3 reviews the field of evolutionary algorithms (EAs). We show the development of the field from its early stages starting with Evolution Strategies (Rechenberg, 1965, 1971; Schwefel, 1965, 1974), and Evolutionary Programming (Fogel, 1962; Fogel et al., 1966). We then continue tracking the historic progress of EAs by reviewing Genetic Algorithms (GAs – Holland, 1975). In this context we also present Genetic Programming (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992), a family of GAs for evolving programs, which also stimulated the development of PIPE. Finally, we leave the area of traditional EAs to present another one of PIPE's origins

— Population-Based Incremental Learning (Baluja, 1994; Baluja and Caruana, 1995), a novel EA, which builds probabilistic models of its search space.

In the second part of the dissertation Chapters 4 & 5 describe PIPE and its applications respectively.

Chapter 4 presents PIPE. It defines PIPE programs, the probabilistic model of program space, and presents the cycles and model's update rules necessary for PIPE to find better and better solutions.

Chapter 5 firstly describes how to systematically setup PIPE to solve problems. Then it tests PIPE's performance first on toy problems and then on a complex multiagent task — a soccer simulation.

In the third part of the dissertation Chapter 6 investigates the dependencies between constrained program structures and PIPE's performance.

Chapter 6 deals with the evolution of structured programs. We present a way of incorporating a priori knowledge into PIPE by constraining program shapes with hierarchies of program instructions. We also show how "non-coding" program parts, i.e. instructions, which can be omitted without changing the functionality of the program, can facilitate the generation of structured programs.

In the fourth and last part of the dissertation Chapters 7 & 8 widen the scope and size of problems that PIPE can be applied to.

Chapter 7 firstly shows several ways how memory can be integrated into PIPE programs. In this way PIPE can solve problems for which the Markov property does not hold, i.e. where the program output does not only depend on the current program input, but also on the temporal context of the input. We show that PIPE programs can successfully utilize memory to solve a variety of problems. On problems with a so called "long time lag", i.e. with many time steps between a relevant input an the corresponding output, PIPE can outperform even the best neural network algorithms, such as, e.g., Long Short-Term Memory (Hochreiter and Schmidhuber, 1997a).

Chapter 8 presents how PIPE can be setup to solve difficult tasks within acceptable time. Unlike the solution presented in Chapter 6, where we boost PIPE's performance by incorporating a priori knowledge into the algorithm, Chapter 8 describes a novel, general automatic task decomposition method termed *filtering* (Sałustowicz and Schmidhuber, 1999b). Filtering not only decomposes the complex task to be solved into solvable subtasks, but it also decomposes into solvable subtasks the possibly also complex task of integrating the found sub-solutions into one final solution. Filtering is an an optimization algorithm independent task decomposition method. In conjunction with PIPE it allows PIPE to solve complex tasks not solvable by PIPE itself.

Finally, **Chapter 9** concludes this thesis.

Chapter 2

Probability-Based Program Search

The term *Probability-Based Program Search* (PBPS) describes a specific class of search algorithms. PBPS algorithms (PBPSAs) have two pronounced features in common that in combination distinguish them from other algorithms. Firstly, they search for programs, which constitute a solution to a given problem. Secondly, to guide their search they employ variable probability distributions over all possible programs with respect to a predefined instruction set.

A starting point for the development of PBPSAs is *Levin Search* (LS – Levin, 1973, 1984). LS is a program search algorithm, but not a PBPSA. LS does not use variable probability distributions to guide its search. For a wide variety of problems, however, such as time-limited optimization problems and inversion problems, LS is optimal with respect to total expected search time, leaving aside a problem size independent, constant factor (Levin, 1973; Levin, 1984; Li and Vitányi, 1993). LS is not necessarily optimal, if experiences collectible during a search can be used to speed up future searches (Schmidhuber, 1995, 1997; Wiering and Schmidhuber, 1996b). This is what PBPSAs are after. PBPSAs adapt their variable probability distributions using intermediate search results to speed-up their searches. Known PBPSAs are *Self-Modifying Probabilistic Learning Algorithms* (SMPLAs – Schmidhuber, 1994; Schmidhuber et al., 1997a,b) and adaptive versions of Levin search (Solomonoff, 1986; Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b) that employ a probability distribution on program space, such as *Adaptive Levin Search* (ALS – Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b; see also Solomonoff (1986) for related ideas). The difference between ALS and SMPLAs lies in the way the algorithms alter their probability distributions. ALS applies a fixed modification algorithm. SMPLAs try to find and optimize the modification algorithm itself, while solving a given task.

Chapter 4 will show how the concept of applying a probability distribution over all possible programs with respect to a predefined instruction set has been integrated into Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a). First, however, we will briefly review LS, ALS, and SMPLAs.

2.1 Levin Search

Levin Search (LS – Levin, 1973, 1984), also known as *universal search*, is an exhaustive search through program space. Starting with small programs of short runtime, where program size is defined by number of total program instructions and runtime by number of executed program instructions, LS evaluates successively larger, and runtime-wise longer programs till it finds a solution. More formally, LS traverses program space by evaluating programs in order of their outputs' Levin complexities (Levin, 1973; Levin, 1984; Li and Vitányi, 1993).

To better explain Levin search, we will first describe Kolmogorov complexity (Solomonoff, 1964; Kolmogorov, 1965; Chaitin, 1969; Li and Vitányi, 1993), and Levin complexity (Levin, 1973; Levin, 1984; Li and Vitányi, 1993). Then, we will show how Levin complexity can be rewritten using a Solomonoff-Levin distribution (Levin, 1974; Solomonoff, 1964; Gács, 1974; Chaitin, 1975; Li and Vitányi, 1993). Finally, we will present Levin's search algorithm (Levin, 1973; Levin, 1984). In this context we will also mention Hutter's search (Hutter, 2001).

Kolmogorov Complexity

Let's assume that the solution to a given problem can be written as string s. To find s we search for a program p that produces s as output. The Kolmogorov complexity $K(s) = |p^*|$ of string s is the length of shortest program p^*, as measured by number of program instructions, that produces s as output and halts. K is independent of p's programming language except for a problem size independent constant factor as shown by the invariance theorem (Solomonoff, 1964; Kolmogorov, 1965; Chaitin, 1969).

In general, however, we do not know how long it takes to compute K due to the halting problem. LS therefore uses an extension of K, termed Levin complexity, to guide its search.

Levin Complexity

Levin complexity Kt extends Kolmogorov complexity by a time factor. Apart from taking into account the length of a program $|p|$, Kt also considers the time $t(p, s)$ of executing program p to produce string s:

$$Kt(s) = \min_p \{|p| + \log_z t(p, s)\}$$

Time $t(p, s)$ is measured by number of executed instructions, $|p|$ denotes the length of program p as measured by number of program instructions, and z is the total number of instructions I_i in instruction set $S = \{I_1, I_2, \ldots, I_z\}$, from which programs can be formed.

We can rewrite the formula for Kt using a Solomonoff-Levin distribution (Li and Vitányi, 1993). The following exposition is based on (Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b).

Solomonoff-Levin Distribution

A Solomonoff-Levin distribution, also known as the *universal prior*, defines the a priori probability of a string s as the probability of guessing a program p, which outputs s and halts. The distribution can be used to assign a priori probabilities to programs in a general way. Given a sequential program $p = a_1 a_2 \ldots a_{|p|}$ ($|p| > 0$) that at each address a_j can have an instruction I_i from instruction set $S = \{I_1, I_2, \ldots, I_z\}$ with z possible instructions, let the probability of generating instruction I_i at program address a_j be $P(I_i, a_j)$. With a Solomonoff-Levin distribution each instruction $I_i \in S$ ($\forall i : 1 \leq i \leq z$) has the same probability of occurring at address a_j ($\forall j : 1 \leq j \leq |p|$). This yields a probability matrix M, where all $z \cdot |p|$ matrix entries $M_{ij} = P(I_i, a_j) = \frac{1}{z}$ are equal. Given such a probability matrix M the probability of generating program p is $P_M(p) = P(I(a_1), a_1) \cdot P(I(a_2), a_2) \cdot \ldots \cdot P(I(a_{|p|}), a_{|p|}) = (\frac{1}{z})^{|p|}$, where $I(a_j)$ denotes the instruction $I_i \in S$, which occurs at address a_j in program p. Therefore $|p| = -\log_z P_M(p)$, and we can write:

$$Kt(s) = \min_p \{-\log_z P_M(p) + \log_z t(p, s)\}$$

Levin's Search Algorithm

Levin's search algorithm (LS) successively generates programs p_x that output strings s_x, in order of the strings' growing Levin complexities $Kt(s_x)$ until a solution string s is found $s = s_x$. From a program search perspective this is equivalent to enumerating all programs in order of their increasing ratios of program runtime divided by program probability $\frac{t(p_x,s_x)}{P_M(p_x)}$ until a program generates s.

In their practical implementation of LS Wiering and Schmidhuber (1996b), and Schmidhuber et al. (1997b) have defined the following procedure:

Definitions
Let $\phi(T)$ denote the set of not yet executed programs p satisfying $P_M(p) \geq \frac{1}{T}$.

```
**Levin search (problem $N$, probability matrix $M$)**
$T := 1$
**repeat**
  **while** $\phi(T) \neq \{\}$ and no solution found **do**
    Generate a program $p \in \phi(T)$.
    Run $p$ until it halts or until it used up $\frac{P_M(p) \cdot T}{c}$ steps.
    If $p$ computed a solution for $N$, return $p$ and exit.
  $T := 2 \cdot T$
**until** $T \geq T_{MAX}$
return $\{\}$
```

T_{MAX} and c are user-defined constants. Programs $p = a_1 a_2 \dots a_{|p|}$ are generated incrementally by first selecting an instruction for address a_1, then for address a_2 etc. Program probabilities $P_M(p)$ are given by matrix M as described in the previous paragraph. Although the procedure above does not strictly follow LS' definition, it stays essentially equivalent to LS by keeping the same order of complexity.

Hutter's search (Hutter, 2001) is a recent program search algorithm, which also shares the same order of time complexity as Levin search. It reduces, however, Levin search's unknown multiplicative constant factor to 5, at the expense of introducing an unknown, problem class-specific, *additive* constant.

2.2 Adaptive Levin Search

Adaptive Levin Search (ALS – Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b) is a heuristic extension of LS. Initially ALS works just like LS. ALS generates and evaluates programs in order of their outputs' growing Levin complexities. It starts with the same probability distribution P_M over all possible programs as LS, where P_M is again determined by a matrix M as described in Section 2.1. Thus given a single problem N, LS and ALS do not differ in the way they solve it. Given, however, k successive problems $N_1, N_2, \ldots, N_k$ the behavior of LS and ALS is different. Being a non-incremental program search method, LS employs a *fixed* probability distribution to model program space. It does not alter its model to incorporate know-how gained from solving problems. ALS, on the other hand, adaptively modifies its *variable* probability distribution after having solved a problem. It makes the program, which constitutes the found solution, more probable. Given program $q = I(a_1)I(a_2)\ldots I(a_{|q|})$ with $|q|$ instructions $I(a_j)$ from instruction set $S = \{I_1, I_2, \ldots, I_z\}$ is a solution to problem N_x found by ALS, ALS will increase the probabilities of generating all $I(a_j)$'s at their corresponding addresses a_j, before searching for a solution to the next problem N_{x+1}. This way ALS quicker finds solutions to successive problems, if the solutions have a similar algorithmic description (program), where similarity between programs is measured by number of equal instructions at same addresses.

Wiering and Schmidhuber (1996b), and Schmidhuber et al. (1997b) use the following procedures to implement ALS:

```
**ALS (problems** ($N_1, N_2, \ldots, N_k$)**, variable matrix** $M$)
**for** $x := 1$ **to** $k$ **do**
   $q :=$ **Levin search** ($N_x, M$)**; Adapt** ($q, M$)
```

where procedure **Adapt** has been defined as follows:

```
**Adapt (program** $q$**, variable matrix** $M$)
**for** $j := 1$ **to** $|q|$, $i := 1$ **to** $z$ **do**
   **if** ($I(a_j) = I_i$) **then** $M_{ij} := M_{ij} + \gamma(1 - M_{ij})$
   **else** $M_{ij} + (1 - \gamma)M_{ij}$
```

Parameter γ ($0 < \gamma < 1$) is the "learning rate", which controls the size of each probability adjustment.

2.3 Self-Modifying Probabilistic Learning Algorithms

Self-Modifying Probabilistic Learning Algorithms (SMPLAs – Schmidhuber, 1994; Schmidhuber et al., 1997a,b) have been recently introduced in the field of *reinforcement learning* (RL). RL scenarios, where agents interact with their environment and receive sometimes positive or negative feedback depending on their actions, are especially well suited for the application of SMPLAs, since SMPLAs implement "lifelong learning" (LL). LL postulates that the entire "life" of a system is one *non-repeatable* training sequence. During system life SMPLAs search for solutions to problems $N_1, N_2, \ldots, N_k$ (compare to Section 2.2). In parallel SMPLAs try to improve their problem solving capabilities. This constitutes a significant difference to other program search algorithms such as LS, ALS, or Genetic Programming, which use fix solution search algorithms.

In what follows we will show how SMPLAs work and present the three most prominent features that make SMPLAs different: (1) SMPLAs do not *generate* whole programs (solution candidates) first and then *execute* them. They execute each instruction immediately after it has been generated. (2) SMPLAs do not use a fix program search algorithm. They use *self-modification* to change their search strategies. (3) SMPLAs implement lifelong learning. They employ *success story algorithm* (SSA) to measure a system's lifetime performance, rather than measuring the system's capability to solve single problems. SSA also ensures that only beneficial system adaptations are kept over time.

Program Generation and Execution

Like ALS, SMPLAs employ a variable probability distribution P_M over all possible programs to guide their search. Again P_M can be represented by a matrix M with entries M_{ij} holding a probability for each instruction $I_i \in S = \{I_1, I_2, \ldots, I_z\}$ to occur at program address a_j in a program $p = a_1 a_2 \ldots a_{|p|}$. Here z denotes the number of instructions in instruction set S, and $|p|$ the number of instructions in program p. ALS uses the probability distribution to generate a program q. After having generated q, ALS will execute it. SMPLAs work differently. SMPLAs employ an instruction pointer *IP*, which points to a program address a_{IP} (initially *IP=1*). SMPLAs generate instruction $I(a_{IP})$ for address a_{IP} randomly with respect to probabilities $M_{ia_{IP}}$ $(1 \leq i \leq z)$. After having generated instruction $I(a_{IP})$, SMPLAs also generate all arguments of $I(a_{IP})$ in the same manner using the n next addresses, where n is the number of $I(a_{IP})$'s parameters. After having generated the entire instruction with its parameters, SMPLAs will immediately execute the instruction, before shifting the *IP* to generate and evaluate further instructions. For sequential programs, where *IP* never revisits or skips an address it does not matter whether instructions are generated and executed online or whether all instructions are generated first (offline) and executed afterwards. If, however, instruction set S contains "jump" or "goto" instructions that allow for revisiting or skipping addresses the difference between online and offline generation and execution is significant. The offline mode ensures that instruction $I(a_x, t_1)$ executed at address a_x at timestep t_1 will remain unchanged when address a_x is revisited by *IP* at timestep t_2: $I(a_x, t_1) = I(a_x, t_2)$ $(t_1 \neq t_2)$. With online generation and execution of instructions the equation does not necessarily hold.

Self-Modifications

To change probability values stored in matrix M, SMPLAs employ special instructions I^{self}, which are part of instruction set S. All I^{self} instructions modify (increase or decrease) matrix entries M_{ij} (and renormalize matrix entries M_{yj} $(\forall y : y \neq i)$) thus changing the probability of generating instruction I_i at program address a_j. Since I^{self} instructions can also set probabilities of I^{self} instructions the self-modifications can change SMPLAs' probability update algorithm (search strategy). To ensure that self-modifications are beneficial for the overall performance of the system, SMPLAs employ the *success story algorithm.*

Success Story Algorithm

Success story algorithm (SSA) identifies a system's current performance and ensures that only those system modifications are kept, which led to an overall better performance. To do this SSA measures and compares reinforcement time ratios before and after system changes. This way SSA tracks the speed changes of reinforcement intake (reinforcement can be, e.g., given when a problem N_x has been solved by the system).

In case of SMPLAs a system's state $V(M, R, t)$ at time t is described by the current probability matrix $M(t)$, the sum of all achieved reinforcements (rewards) $R(t)$ from system start until t, and time t. Initially at $t_0 = 0$ all $M(t_0)$'s matrix entries $M_{ij}(t_0)$ are equal and the system has received no reward $R(t_0) = 0$. SSA saves this original state onto an initially empty stack. Then when SMPLAs start generating and executing instructions $M(t)$ changes whenever an I^{self} instruction is executed. Also $R(t)$ changes, whenever the system receives reinforcement for its actions. At some points in time SSA is called to check systems performance and undo changes to $M(t)$, if they have not been observed to enhance performance. SMPLAs control the intervals of measuring system's performance themselves by setting checkpoints on their own. To set checkpoints they use special instructions I^{check}, which are part of instruction set S. I^{check} instructions trigger SSA either immediately or time delayed.

When SSA is triggered it runs the following procedure: If the stack, onto which states are saved, contains only one entry $V_1(M, R, t_x)$, performance cannot be measured. Since SSA measures reinforcement intake acceleration, it needs at least three points in time. Thus SSA will simply save the current state V as the second stack entry V_2. If there is more than one state saved on the stack, SSA will firstly calculate the performance of the system. Let $M(t')$, $R(t')$ and t' be the probability distribution, reinforcement, and time (respectively) of the topmost (last) stack entry V'. Let $M(t'')$, $R(t'')$ and t'' denote the probability distribution, reinforcement, and time (respectively) of stack entry V'', which lies right below V'. System's performance is measured by comparing the speed differences of reinforcement intake between checkpoints. If the success story criterion $\frac{R(t)-R(t')}{t-t'} > \frac{R(t)-R(t'')}{t-t''}$, where t is current time, is met, the performance of the system improved, and the current state V is added to the stack. Otherwise M' is restored (the probability matrix M of current state V is replaced by M' yielding $V(M', R, t)$) and V' is popped off the stack. Then again system performance is measured as described above. The process of measuring performance and popping off stack entries is repeated until either the success story criterion is met, or the stack contains only a single entry. If the success story criterion is met, V is added to the stack. If the stack only contains a single entry, M' is restored, V' is popped off the stack, and the new current state $V(M', R, t)$ is added to the stack. In this case $M' = M(t_0)$.

2.4 Conclusion

This chapter gave an overview over the field of probability-based program search (PBPS). For many problems there exists an optimal (with respect to total expected search time), program search algorithm: Levin Search (LS –

Levin, 1973, 1984). LS is not a PBPS algorithm (PBPSA), since it does not model program space with variable probability distributions. It can, however, be seen as a starting point for the development of PBPSAs. LS does not use any experience collectible during its search to speed it up. PBPSAs such as Adaptive Levin Search (ALS – Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b), and Self-Modifying, Probabilistic Learning Algorithms (SMPLAs – Schmidhuber, 1994; Schmidhuber et al., 1997a,b) do. They apply variable probability distributions and incrementally adapt them by incorporating experiences gained during program search thus trying to focus on more promising regions of the search space.

Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a) is an alternative approach that uses *evolutionary algorithms* to optimize its probabilistic models of program space.

Chapter 3

Evolutionary Algorithms

Evolutionary Algorithms (EAs) is a comprehensive term for a specific class of optimization algorithms. In general, EAs start with randomly generated solution candidates and try to find better and better solution candidates during the "evolutionary" optimization run. The quality of solution candidates is measured with an objective function, also called the *fitness function.* A fitness function maps solution candidates to scalar fitness values that reflect the candidates' performances on a given tasks. From an EA point of view the fitness function defines the task to be solved.

EAs have two pronounced features in common that distinguish them from other optimization algorithms (Yao, 1999). They all work with pools (populations) of encoded solution candidates (individuals) and they all allow for global and/or local information exchange between those solution candidates. Global information exchange is achieved via the fitness function that influences the generation of further solution candidates. Local information exchange relies on swapping potential sub-solutions between individuals.

Traditional EAs are *Evolution Strategies* (ES – Rechenberg, 1965, 1971; Schwefel, 1965, 1974), *Evolutionary Programming* (EP – Fogel, 1962; Fogel et al., 1966), and *Genetic Algorithms* (GAs – Holland, 1975), although for ES populations with more than two individuals were introduced at a later stage (Schwefel, 1981). To encode solution candidates traditional EAs apply a reversible mapping function to map parameters of a problem (phenotype) onto fixed-length vectors of numbers or letters from a given alphabet (genotype). Given the encoded solution candidates (individuals), they then apply a common algorithmic framework to search solution space. The framework contains three, basic, repetitive steps: *selection*, *production*, and *reduction*,

```
Create an initial population Pop(i) of random individuals
Calculate each individual's solution quality (fitness)
REPEAT
  1. selection
     select individuals (parents) from Pop(i) for production
  2. production
     apply production operators (recombination and/or mutation)
     to parents to create new individuals (offspring)
  3. reduction
     (a) calculate each offspring's solution quality (fitness)
     (b) create a successive population Pop(i+1) from offspring
         and/or parents by discarding some individuals
UNTIL a desired solution has been found or a time limit has
been reached
```

Figure 3.1: General algorithmic framework of traditional evolutionary algorithms. Either the selection or the reduction step picks individuals based on their fitnesses. Fitness calculations are conducted on phenotypes and include therefore here the mapping step from genotype to phenotype.

as shown in Figure 3.1. Traditional EAs differ, however, in their mappings from phenotype to genotype and their implementation of the three basic steps of the algorithmic framework. Sections 3.1–3.3 will present details.

Recently a new EA termed *Population-Based Incremental Learning* (PBIL – Baluja, 1994; Baluja and Caruana, 1995) has been developed. PBIL is different from traditional EAs in the way it samples solution space. Traditional EAs generate a population of individuals and then successively modify those individuals until an adequate solution has been found. PBIL, however, generates successive populations from an adaptive probability distribution, which is refined using the best individuals of a current population. The probability distribution stores the knowledge gained during the evolutionary run and propagates it from population to population. Thus PBIL handles the aforementioned "information exchange between solution candidates" in a fundamentally different way than traditional EA approaches.

In Chapter 4 we will see how PBIL's evolution concept has been integrated into PIPE. First, however, we will briefly review ES, EP, GA, and PBIL. With GA we will also mention its applications to evolve programs termed *Genetic Programming* (GP – Cramer, 1985; Dickmanns et al., 1987; Koza, 1992).

3.1 Evolution Strategies

Evolution strategies (ES – Rechenberg, 1965, 1971; Schwefel, 1965, 1974) were originally developed as a method for numerical optimization. This coined ES' solution candidates' encoding and production operators (selection, production, and reduction). We will review the population-based versions (Schwefel, 1981).

Encoding

With ES solution candidates are encoded close to the problem's original representation. Parameters of a problem are randomly juxtaposed to form a vector-coded solution candidate (individual) $\vec{x} = (x_1, x_2, \ldots, x_n)$ with n vector components $x_i \in I\!R$ called object variables (Bäck and Schwefel, 1996). There is no emphasis neither on the position of any of the object variables in the vector, nor on the mappings between phenotype and genotype. Usually the identity mapping is used.

Additionally, ES also foresee the possibility of encoding so called "strategy parameters" next to the object variables (Schwefel, 1981). Strategy parameters are free parameters of the ES optimization algorithm. When added to the genotypic description they will be optimized during the evolutionary run along with the object variables. This technique of optimizing the search algorithm while solving the objective problem is termed *self-adaptation.* We will point out relevant strategy parameters in the forthcoming Section on production.

Selection

ES' selection strategies are determined by the reduction schemes (see Section on reduction below). There is no designated selection operation to define parents in a population for production. All individuals of a current population are used to generate offspring.

Production

ES use recombination and mutation to generate offspring from parents. Mutation is ES' main driving force and can be used without recombination. In case both are used, recombination is usually applied to parent individuals before mutation.

Recombination. There are two forms of recombination in ES: discrete and intermediary.

Discrete recombination generates an offspring $\vec{x}' = (x'_1, x'_2, \ldots, x'_n)$ by probabilistically taking some vector components from one parent $\vec{x} = (x_1, x_2, \ldots, x_n)$ and some from another parent $\vec{y} = (y_1, y_2, \ldots, y_n)$ as follows:

$$x'_i = \begin{cases} x_i & \text{with probability } P_{recombination} \\ y_i & \text{otherwise} \end{cases}$$

Here $P_{recombination}$ is a real valued number in (0,1). Offspring $\vec{y}'$ is then the complement of $\vec{x}'$ with respect to $\vec{x}$ and $\vec{y}$.

Intermediary recombination works by arithmetically averaging vector component values of parents. Given parents $\vec{x} = (x_1, x_2, \ldots, x_n)$ and $\vec{y} = (y_1, y_2, \ldots, y_n)$ offspring $\vec{x}'$ is calculated as follows:

$$x'_i = x_i + \alpha(y_i - x_i)$$

Here α is a weighting parameter in (0,1). Offspring $\vec{y}'$ is generated accordingly.

Mutation. Given individual $\vec{x} = (x_1, x_2, \ldots, x_n)$ with n vector components $x_i \in I\!R$, mutation is achieved by adding a Gaussian random number to each vector component x_i:

$$x'_i = x_i + N_i(0, \sigma_i)$$

Here $N_i(0, \sigma_i)$ is a normally distributed random number with mean 0 and standard deviation σ_i. All N_i are generated independently.

Parameters σ_i are strategy parameter that can be added to the vector-descriptions of individuals to enable self-adaptation. Strategy parameters are mutated right before mutating objective variables x_i. If a single σ is used for all vector components x_i the mutation rule for σ is:

$$\sigma' = \sigma \cdot \exp(\tau_0 \cdot N(0,1))$$

Here $\tau_0 \propto (\sqrt{n})^{-1}$ is the "learning rate". If a dedicated σ_i is used for each vector component x_i the mutation rule for σ_i is:

$$\sigma'_i = \sigma_i \cdot \exp(\tau' \cdot N(0,1) + \tau \cdot N_i(0,1))$$

Here the learning rates are $\tau' \propto (\sqrt{2n})^{-1}$ and $\tau \propto (\sqrt{2\sqrt{n}})^{-1}$. Since mutating vector components independently might be inappropriate, also a complete co-variance matrix can be made part of an individual (Bäck and Schwefel, 1996). The benefit of such self-adaptation, however, is unclear, as the search space is exponentially increased (Yao, 1999).

Reduction

ES use one of the two deterministic reduction schemes: $(\lambda + \mu)$ or (λ, μ), where μ is the population size and the number of parents and λ is the number of offspring. With $(\lambda + \mu)$, μ individuals with highest fitness will be selected from parents and offspring to form the successive population. With (λ, μ) the μ individuals will be selected from offspring only.

3.2 Evolutionary Programming

Evolutionary programming (EP – Fogel, 1962; Fogel et al., 1966) was originally developed to generate intelligent behavior by evolving finite state machines. Today, however, EP is mostly used for continuous parameter optimization problems.

From an algorithmic point of view current variants of EP for numerical optimization are very similar to ES, especially with respect to the representation of individuals, and the mutation operator including the self-adaptation of strategy parameters. A thorough analysis of the fine differences can be found in (Bäck, Rudolph, and Schwefel, 1993). The major differences between both algorithms are founded in the recombination and reduction parts, where reduction, as with ES, determines the selection strategy. EP does not apply any recombination and for reduction it uses a tournament selection scheme.

During reduction each individual of μ parents and λ offspring competes with q opponents that are chosen uniformly random from parents and offspring. The individual scores a point, each time its solution quality (fitness) is no worse than the fitness of its opponent. A total of μ individuals with highest scores form the population of the next generation, while the rest is discarded. With $q < (\lambda + \mu)$ this reduction scheme is probabilistic and constitutes therefore a major difference to ES' deterministic reduction schemes.

3.3 Genetic Algorithms

Genetic algorithms (GAs – Holland, 1975) differ from ES and EP in terms of solution candidates' encoding and production operators. GAs emphasize the use of elaborated reversible mappings between phenotypes and genotypes to transform original problem spaces to more suitable search spaces and use recombination instead of mutation as their main search operator.

Encoding

GAs encode parameters of problems (phenotypes) onto vectors of numbers or letters from a given alphabet (genotypes). The user-defined, problem-dependent mapping between phenotypes and genotypes, however, usually goes beyond the identity mapping. The choice of the mapping and therefore of the genetic representation (coding) is crucial to a GA's performance (Goldberg, 1989; Marti, 1992). There is no predefined way of finding efficiently searchable representations. There are, however, several criteria to guide the engineering of such a genetic coding (Gruau, 1994; Sałustowicz, 1995):

- **completeness**
  Every point in phenotypic solution space is representable in the genotypic search space.
- **closure**
  All individuals produced by applying production operators can be decoded into valid phenotypes (provided the individuals' parents have been valid individuals).
- **proximity**
  Small/large changes in the genotypic description result in small/large changes in phenotypic solutions.
- **short schemata**
  More correlated parameters have a smaller probability of being separated onto the two offspring during recombination than less correlated or independent parameters (Holland, 1975).
- **compactness**
  The length of the genotypic description, as given by number of vector components, is minimal.

- **non-isomorphism**
  Each genotype represents a single phenotype.

- **modularity**
  Partial solutions are encoded just once and a mechanism for referencing them at different places in the coding is provided to assure reusability of information.

Selection

GAs use *fitness proportionate selection.* Parent individuals are selected for production based on their solution quality. Given all μ individuals $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_\mu$ of the parent population with corresponding fitness values $FIT(\vec{x}_1)$, $FIT(\vec{x}_2)$,..., $FIT(\vec{x}_\mu)$, an individual i is selected for production with probability:

$$P(i) = \frac{FIT(\vec{x}_i)}{\sum_{j=1}^{\mu} FIT(\vec{x}_j)}$$

Altogether μ individuals are selected for production. A single individual can be selected multiple times.

Production

GAs apply two production steps to generate offspring from parents: recombination and mutation. Recombination is GAs' major search operator. Mutation is usually of smaller importance, although some recent studies show that mutation can speed up GAs and let the search converge more reliably (Bäck, 1993; Yanagiya, 1993; Bäck and Schütz, 1996).

Recombination. Recombination is achieved with the *crossover* operator. Crossover exchanges parts of two vector-coded individuals, thus forming two offspring. There are various implementations of crossover. The canonical GA introduced by Holland (1975) uses a one-point crossover. Given two individuals $\vec{x} = (x_1, x_2, \ldots, x_n)$ and $\vec{y} = (y_1, y_2, \ldots, y_n)$ with n vector components each, a single crossover point c is generated between vector component 1 and $n-1$ (inclusively). Then the first c vector components of $\vec{x}$ are exchanged for vector components of $\vec{y}$ and vice versa, thus creating two offspring $\vec{x}' = (y_1, y_2, \ldots, y_c, x_{c+1}, \ldots, x_n)$ and $\vec{y}' = (x_1, x_2, \ldots, x_c, y_{c+1}, \ldots, y_n)$. A two-point crossover will select two crossover points c_1 and c_2, where $1 \leq c_1 \leq c_2 \leq n-1$, and swap the vector components between those two points creating offspring $\vec{x}' = (x_1, x_2, \ldots, x_{c_1-1}, y_{c_1}, \ldots, y_{c_2}, x_{c_2+1}, \ldots, x_n)$

and $\vec{y}' = (y_1, y_2, \ldots, y_{c_1-1}, x_{c_1}, \ldots, x_{c_2}, y_{c_2+1}, \ldots, y_n)$. An N-point crossover will select N crossover points and work accordingly. Finally, a uniform crossover is comparable to ES' discrete recombination, where each vector component of an offspring is chosen randomly from one of the two parent individuals.

Crossover is typically executed with a certain probability $P_{crossover}$. Given $P_{crossover} < 1$ some individuals of the parent population might not undergo crossover. If not changed during the following, and also probabilistic mutation step, they may enter the successive population unaltered.

Mutation. Mutation alters vector components of an individual randomly. Each vector component is mutated with probability $P_{mutation}$, where $P_{mutation}$ is usually significantly smaller than $P_{crossover}$. With numerical vector components mutation will add/subtract a random number to/from the selected vector components. With alphabet-based vector components mutation will randomly exchange selected symbols for other symbols from the alphabet.

Reduction

During the reduction step usually all parents are discarded and offspring form the successive population. Another reduction strategy, termed *elitist strategy*, discards all parents but the best parent. The new population then consists of the best individual of the parent population and all but one (often the worst) offspring.

3.3.1 Genetic Programming

Genetic programming (GP – Cramer, 1985; Dickmanns et al., 1987; Koza, 1992) is a GA for evolving programs. There are two main variants of GP: "linear" GP (Cramer, 1985; Dickmanns et al., 1987), and "tree-based" GP (Cramer, 1985; Koza, 1992). Both variants differ in the genotypes they use, and as a consequence in the way they implement production operators.

Linear GP

Linear GP encodes programs onto variable-length vectors of numbers (Cramer, 1985) or symbols (Dickmanns et al., 1987) and can therefore implement production operators as described in Section 3.3. Neither Cramer's (1985), nor Dickmanns et al.'s (1987) linear GP variant, however, employs a genetic coding that meets the closure criterion. It seems to be difficult to find such a coding for linear GP. Tree-based GP overcomes this drawback.

Tree-Based GP

Tree-based GP (Cramer, 1985; Koza, 1992) encodes functional programs onto tree-structured genotypes. Programs consist of instructions that are either functions or terminals (e.g., input variables). Each argument of a function can either be a function or a terminal. This defines a tree structure where each nonleaf node contains a function and each leaf node a terminal (see Figure 3.2).

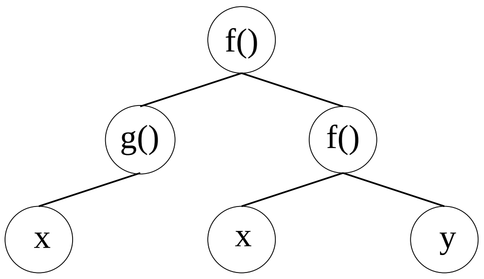

Figure 3.2: Sample GP program tree computing $f(g(x), f(x, y))$, where $f()$ and $g()$ are functions and x and y are terminals.

GP's main production operator is recombination. Recombination is achieved by swapping sub-trees between two parental individuals (see Figure 3.3). Closure is ensured, if all functions can accept as arguments all terminals and all results of function evaluations with respect to data type and value (Koza, 1992).

Further, optional GP production operators include mutation, inversion, permutation, editing, encapsulation, and decimation (see Cramer (1985) or Koza (1992) for implementation details).

3.4 Population-Based Incremental Learning

We have seen how traditional EAs sample search space by first generating a pool of random solution candidates and then exploring the neighboring regions of those solutions candidates with respect to the production operators. *Population-based incremental learning* (PBIL – Baluja, 1994; Baluja

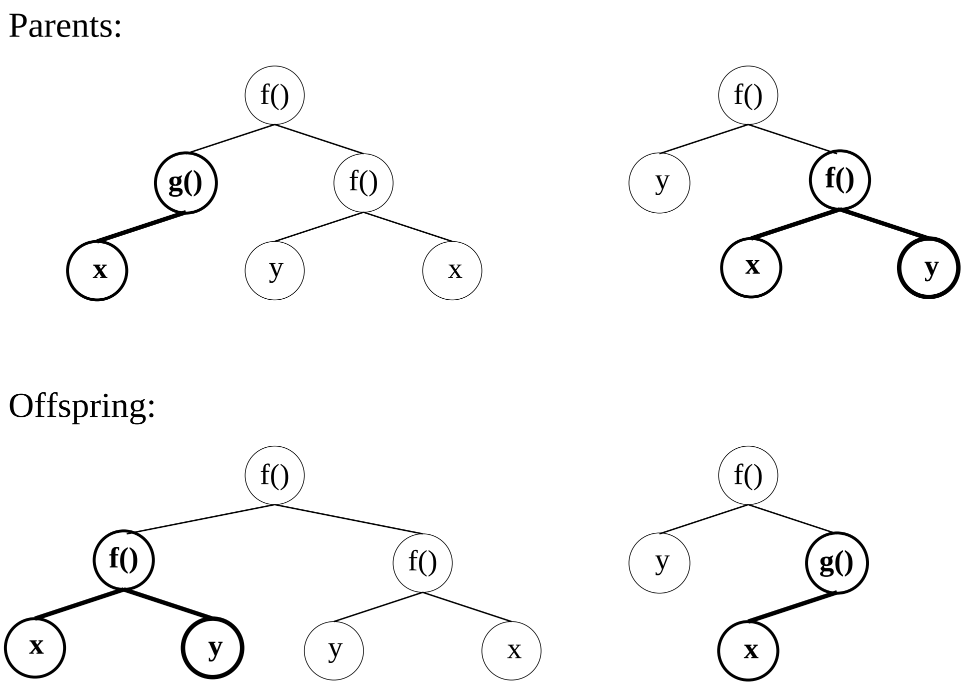

Figure 3.3: Example of recombination in tree-based GP. The two emphasized parts of parental programs have been swapped to form two offspring.

and Caruana, 1995) works differently. It builds probabilistic models of best solution candidates. Starting with a random model of the search space, PBIL generates a pool of solution candidates. It then refines the model to better represent the best of the solution candidates. New solution candidates are then generated from the updated model, the model is refined again, and so on. By applying a model PBIL samples the search space in a fundamentally different way than traditional EAs.

Encoding

PBIL encodes parameters of problems (phenotypes) onto bit-vectors (genotypes) $\vec{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in \{0, 1\}$, $\forall i : 1 \leq i \leq n$.

Model

PBIL uses a vector of real-valued numbers $\vec{p} = (p_1, p_2, \ldots, p_n)$ for a model. Each vector component p_i represents the probability of generating a "1" at position i of a genotype $\vec{x} = (x_1, x_2, \ldots, x_n)$.

Learning

Initially, PBIL starts out with a random model $\vec{p} = (0.5, 0.5, \ldots, 0.5)$. PBIL uses the model to generate a pool of μ solution candidates (population). A solution candidate (individual) $\vec{x}$ is generated by setting its ith component x_i to 1 with probability p_i ($\forall i : 1 \leq i \leq n$). The M best performing individuals $\vec{x}^1, \vec{x}^2, \ldots, \vec{x}^M$ of the population are then used to update model $\vec{p}$. Here $M \ll \mu$ is a user-defined constant. For each individual $\vec{x}^* \in \{\vec{x}^1, \vec{x}^2, \ldots, \vec{x}^M\}$ model $\vec{p}$ is updated as follows:

$$p_i := p_i \cdot (1 - LR) + LR \cdot x_i^*$$

Here LR is the learning rate, and x_i^* is the i'th component of solution candidate $\vec{x}^*$. After the model update all individuals of the population are discarded and a new population is generated using the updated model. This cycle of generating populations and updating the model is repeated until the model reflects a single solution with high probability. A final probability vector (model) might be, e.g., $\vec{p} = (0.99, 0.01, \ldots, 0.01)$.

3.5 Conclusion

This chapter reviewed the field of evolutionary algorithms. It has shown the development of the field, which started with evolution strategies (Rechenberg, 1965, 1971; Schwefel, 1965, 1974), and evolutionary programming (Fogel, 1962; Fogel et al., 1966), has developed further with genetic algorithms (Holland, 1975), and continues developing with population-based incremental learning (PBIL – Baluja, 1994; Baluja and Caruana, 1995). We have also mentioned the application of genetic algorithms to program evolution termed genetic programming (GP – Cramer, 1985; Dickmanns et al., 1987; Koza, 1992).

Chapter 4 will now present how the ideas of PBIL and the program representation of tree-based GP have been integrated into Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a).

Chapter 4

Probabilistic Incremental Program Evolution

Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a) is a new method for synthesizing programs. PIPE searches spaces of tree-structured, functional programs that can be constructed from predefined instruction sets. PIPE applies a generational model: Starting with a population of randomly generated programs, it iteratively generates successive program populations (generations). To create better and better programs PIPE uses an adaptive probability distribution over all possible programs with respect to a predefined instruction set. Initially the probability distribution is random. It is then successively adapted as follows: (1) Each generation, the probability of generating the best program in the current population is increased; (2) occasionally the probability of generating the best program found so far (elitist) is increased; (3) sometimes, single probabilities are mutated to better explore the search space.

PIPE emerged from three major sources of inspiration: (1) Probability-Based Program Search; (2) Genetic Programming; and (3) Population-Based Incremental Learning.

Probability-Based Program Search

Probability-Based Program Search Algorithms (PBPSAs) have been recently introduced in the field of reinforcement learning (Schmidhuber, 1994; Wiering and Schmidhuber, 1996b; Zhao and Schmidhuber, 1996; Schmidhuber et al., 1997a,b). With PBPSAs instruction sequences are generated and executed according to sets of variable, initially uniform probability distributions. The distributions are modified either by a fixed learning algorithm such as Adaptive Levin Search (ALS – Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b) or by an evolving, self-modifying, probabilistic learning algorithm (SMPLA) embedded within the distributions themselves (Schmidhuber et al., 1997a). ALS extends Levin Search (LS – Levin, 1973, 1984), a theoretically optimal algorithm for *non-incremental* search in program space, to the incremental case. SMPLAs go one step further – they try to improve even the way they learn by modifying their learning algorithm.

PIPE similarly encodes programs in variable probability distributions, but it is not an on-line reinforcement learning method and does not use SMPLAs, or ALS. It is an evolutionary PBPSA that bases its search on successive program generations comparable to those used by Genetic Programming (e.g., Cramer, 1985; Dickmanns et al., 1987; Koza, 1992).

Genetic Programming (GP)

GP is a Genetic Algorithm for evolving programs. It starts with a population of random programs. Each program's quality is evaluated on a given task. Selected programs may (1) immediately join the next generation, or (2) exchange code with other programs ("crossover"). Programs with high quality have higher probability of being selected than others. The procedure is repeated for a fixed number of generations or until a satisfactory solution has been found.

Koza's GP variant (Koza, 1992) encodes programs in parse trees. So does PIPE. Thus, both can be applied to the same problems. GP, however, stores domain knowledge in program populations, whereas PIPE captures this knowledge in a probability distribution. GP relies on crossover to generate better and better programs, whereas PIPE uses a learning method similar to Population-Based Incremental Learning (Baluja and Caruana, 1995).

Population-Based Incremental Learning (PBIL)

PBIL generates a population of fixed-length bitstrings (solution candidates for a given task) according to a vector of probabilities (initially 0.5). The probabilities are then adjusted to increase the probability of the current population's best individuals. This procedure is repeated until all probabilities are either 1.0 or 0.0. Thus, PBIL does not store domain knowledge in a population, but in a probability distribution.

PIPE follows PBIL's update algorithm but uses a different representation. PIBL stores the probability distribution in a fixed-length probability vector that encodes probabilities for bits in the solution representation being set. PIPE, however, needs to handle tree-coded programs of varying size and uses an incrementally growing and shrinking *Probabilistic Prototype Tree (PPT)* which contains the probability distribution over all possible programs with respect to a predefined instruction set. Furthermore, PIPE significantly extends PBIL's initialization and update rules to accommodate tree-coded programs.

Outline

The remainder of this chapter is organized as follows: Section 4.1 defines instructions – the basic program elements – and describes the tree-structure of programs. Section 4.2 is dedicated to the *Probabilistic Prototype Tree (PPT)* that stores the probability distribution. Section 4.3 presents PIPE's learning algorithm and explains all update rules. Section 4.4 shows how PIPE handles multiple outputs. Section 4.5 concludes this chapter.

4.1 Programs

This section defines a program's elementary parts (instruction set) and its representation (tree structure).

4.1.1 Instructions

Programs are made of instructions from an instruction set $S = \{I_1, I_2, \ldots, I_z\}$ with z instructions. Instructions are user-defined. Each instruction is either a *function* or a *terminal*. Instruction set S therefore consists of a function set $F = \{f_1, f_2, \ldots, f_k\}$ with k functions and a terminal set $T = \{t_1, t_2, \ldots, t_l\}$ with l terminals, where $z = k + l$ holds. Functions and terminals differ in that the former have one or more arguments and the latter have zero. For instance, to solve a one-dimensional function regression task one might use $F = \{+, -, *, \%, sin,\ cos, exp, rlog\}$ and $T = \{x, R\}$, where % denotes protected division ($\forall y, z \in I\!R, z \neq 0$: $y\%z = y/z$ and $y\%0 = 1$); *rlog* denotes protected logarithm ($\forall y \in I\!R, y \neq 0$: $rlog(y)$=log(abs(y)) and $rlog(0) = 0$); x is an input variable; and R is a *generic random constant*.

Generic Random Constants (GRCs)

GRCs are used by PIPE to allow for random constants in programs. A GRC (compare also "ephemeral random constant" – Koza, 1992) is a zero argument function (a terminal). When accessed during program creation, it is either instantiated to a random value from a predefined, problem-dependent set of constants or a value previously stored in the *PPT* (see Section 4.2.3).

Closure

The instruction set must comply with the *closure* principle (Koza, 1992). The closure principle ensures that all created programs are syntactically correct. To ensure closure for PIPE every terminal and every output of a function must be acceptable as another function's argument with respect to type and value.

4.1.2 Representation

Programs are encoded in n-ary trees, with n being the maximal number of function arguments. Each nonleaf node encodes a function from F and each leaf node a terminal from T. The number of subtrees each node has corresponds to the number of arguments of its function. Each argument is calculated by a subtree. The trees are parsed depth first from left to right. Sample program trees for the function regression task of Section 4.1.1 are shown in Figure 4.1.

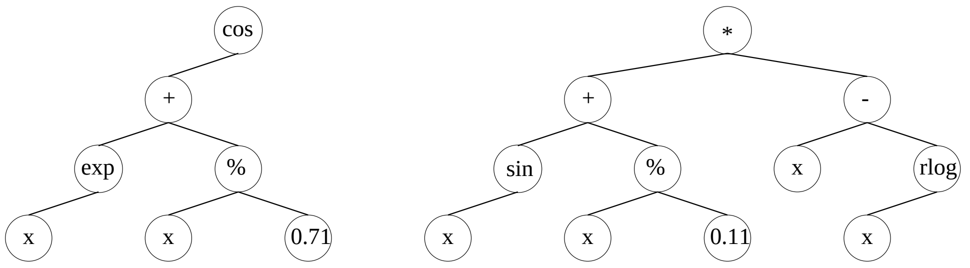

Figure 4.1: Sample program trees for function regression. (left) *f(x)= cos(exp(x)+(x%0.71))*; (right) *f(x)=((sin(x)+(x%0.11))*(x-rlog(x)).*

4.2 Probabilistic Prototype Tree

The *Probabilistic Prototype Tree* (*PPT*) stores the knowledge gained from experiences with programs and guides the evolutionary search. It holds random constants and the probability distribution over all possible programs that can be constructed from a predefined instruction set. The *PPT* is generally a complete n-ary tree with infinitely many nodes, where n is again the maximal number of function arguments.

4.2.1 Nodes

All *PPT* nodes are created equal. Each node N_j, with $j \leq 0$ contains a random constant R_j and a variable probability vector $\vec{P_j}$. Each $\vec{P_j}$ has z components, where z is the number of instructions in instruction set S. Each component $P_j(I)$ of $\vec{P_j}$ denotes the probability of choosing instruction $I \in S$ at node N_j. All components of vector $\vec{P_j}$ sum up to one: $\sum_{I \in S} P_j(I) = 1$.

4.2.2 Initialization

Each *PPT* node N_j requires an initial random constant R_j and an initial probability $P_j(I)$ for each instruction $I \in S$. A value for R_j is randomly taken from the same predefined, problem-dependent set of constants, from which also the GRC function draws its instantiations (see Section 4.1.1). To initialize instruction probabilities a predefined, constant probability P_T for selecting an instruction from T (the terminal set) and $(1 - P_T)$ for selecting an instruction from F (the function set) are used. Each vector $\vec{P_j}$ is then initialized as follows:

$$P_j(I) := \frac{P_T}{l}, \forall I : I \in T \qquad \text{and} \qquad P_j(I) := \frac{1 - P_T}{k}, \forall I : I \in F,$$

where l is the total number of terminals in T and k is the total number of functions in F. Figure 4.2 shows an initialized *PPT* node for the function regression task defined in Section 4.1.1 with $P_T = 0.6$ and R_j picked uniformly random from [0;1).

4.2.3 Program Generation

Programs are generated according to the probability distribution stored in the *PPT*. To generate a program PROG from *PPT*, an instruction $I \in S$ is selected with probability $P_j(I)$ for each accessed node N_j of *PPT*. This

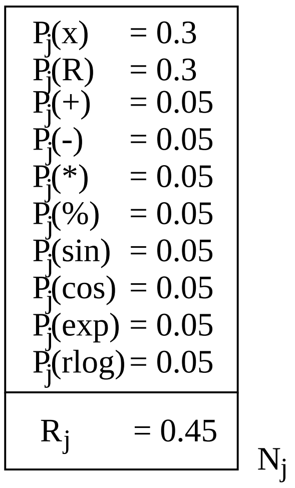

Figure 4.2: Initialized *PPT* node N_j with $P_T = 0.6$ and $R_j \in [0;1)$. N_j holds a probability $P_j(I)$ for each instruction $I \in S$ (see Section 4.1.1) and a random constant R_j to allow for GRCs.

instruction is denoted as I_j. Nodes are accessed in a depth-first way, starting at the root node and traversing *PPT* from left to right. Once $I_j \in F$ (a function) is selected, a subtree is created for each argument of I_j. If $I_j = R$ (the GRC), then an instance of R, called $V_j(R)$, replaces R in PROG. If $P_j(R)$ exceeds a threshold T_R, then $V_j(R) = R_j$ (the value stored in the *PPT*). Otherwise $V_j(R)$ is generated uniformly random from a problem-dependent set of constants. Starting with N_0 (root node) the program generation process can be recursively written as follows:

```
create_program_node_from_PPT_node(*ppt_node, *program_node) {
  probabilistically select instruction $I_j$ according to $\vec{P}_j$;
  /* special treatment, if instruction is a GRC */
  if $I_j$ = $R$ then {
   if $P_j(R) > T_R$ then $I_j := V_j(R) = R_j$;
   else $I_j := V_j(R)$ = ``random value from problem-dependent set'';
  }
  for ($i := 0$; $i <$ ``number of $I_j$'s arguments''; $i := i + 1$)
   create_program_node_from_PPT_node(ppt_node$\rightarrow$next[$i$],
                                   program_node$\rightarrow$next[$i$]);
}
```

Figure 4.3 illustrates the relation between a *PPT* and a possible program tree for the function regression example of Section 4.1.1.

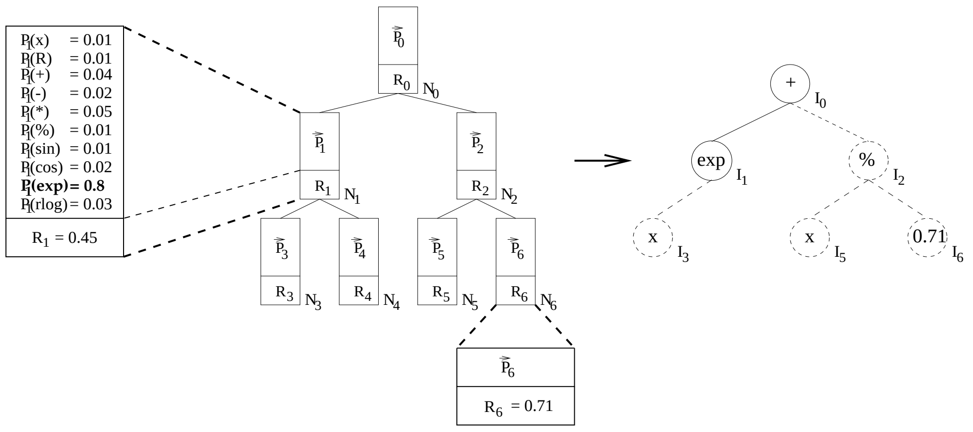

Figure 4.3: (left) Example of node N_1's instruction probability vector $\vec{P_1}$ and random constant R_1. (middle) Probabilistic prototype tree *PPT* with details of node N_6. (right) Possible extracted program PROG. At the time of creation of instruction I_1, the dashed part of PROG did not yet exist. $I_6 = R$ is instantiated to $I_6 := V_6(R) = R_6 = 0.71$ because probability $P_6(R)$ (not shown) exceeds the random constant threshold T_R.

4.2.4 Tree Shaping

A *complete PPT* is infinite. A “large” *PPT* is memory intensive. Recall that each *PPT* node holds a probability for each instruction, a random constant, and n pointers to following nodes, where n is *PPT*'s arity. Empirical evidence, however, indicates that it suffices to maintain a *PPT* with on average roughly two to three times as many nodes as in the current best solution (best program of generation). To reduce memory requirements, it is thus possible to incrementally grow and prune the *PPT*.

Growing

Initially, the *PPT* contains only the root node. Further nodes are created “on demand” whenever $I_j \in F$ is selected and the subtree for an argument of I_j is missing. Figure 4.4 shows how the *PPT* grows incrementally.

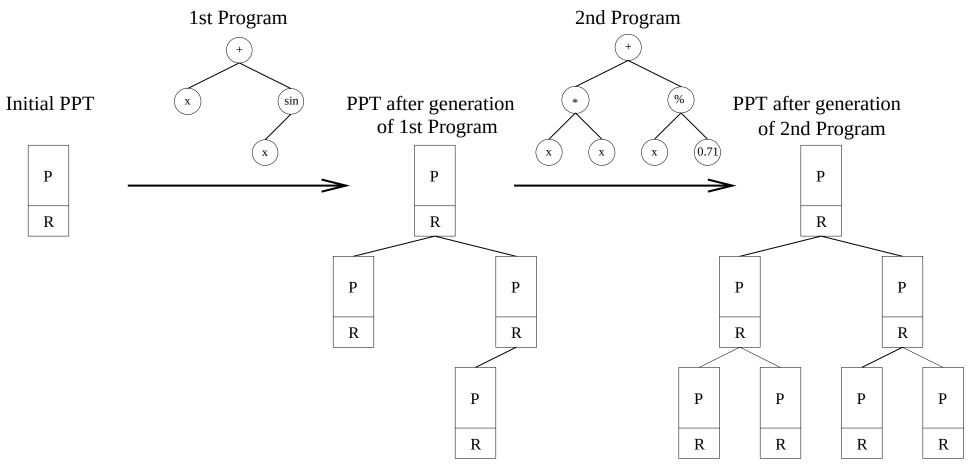
1st Program
2nd Program
Initial PPT
PPT after generation of 1st Program
PPT after generation of 2nd Program
+
x
sin
x
+
*
%
x
x
x
0.71
P
R

Figure 4.4: Growing the *PPT* "on demand". Initially the *PPT* contains only the root node (left). Additional nodes are created with each program that accesses non-existing nodes during its generation.

Pruning

PPT subtrees attached to nodes that contain at least one probability vector component above a threshold T_P can be pruned. If T_P is set to a sufficiently high value (e.g., $T_P = 0.99999$) only parts of the *PPT* will be pruned that have a very low probability of being accessed. In case of functions, only those subtrees should be pruned that are *not* required as function arguments (see Figure 4.5). Apart from reducing memory requirements, pruning also helps to discard elements of the probability distribution that have become irrelevant over time.

4.3 Learning

PIPE attempts to find better and better programs by biasing its search towards programs that are statistically similar to previous best solutions.

4.3.1 Fitness Functions

What makes a program better than another? In order to answer this question it is necessary to setup a quality measure for programs. PIPE uses *fitness functions.* A fitness function is problem-dependent and user-defined. It

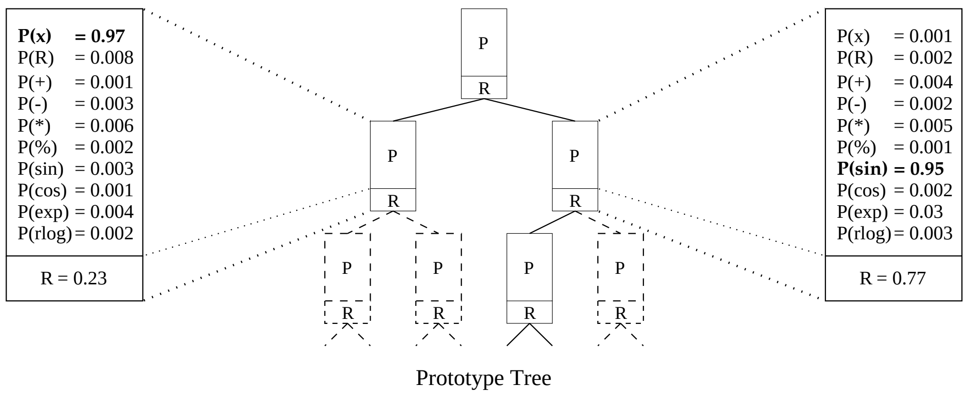

Figure 4.5: The dashed parts of the prototype tree can be pruned because the probabilities of the adjacent nodes exceed threshold value $T_P = 0.9$ and contain high probabilities for a terminal (left) and a single function with one argument (right).

defines the task to be solved. A fitness function maps programs to scalar, real-valued fitness values that reflect the programs' performances on a given task. For PIPE to work properly fitness functions need to comply with the following: (a) Fitness values must not be negative. (b) Programs embodying better solutions need to be mapped to smaller fitness values. Thus, PIPE seeks to minimize fitness and PIPE's fitness functions can therefore be seen as "error measures".

A secondary non-user-defined objective for which PIPE always optimizes programs is program size as measured by number of nodes. Among programs with equal fitness smaller ones are *always* preferred. This objective constitutes PIPE's built-in Occam's razor.

4.3.2 Learning Framework

PIPE combines two forms of learning: Generation-Based Learning (GBL) and Elitist Learning (EL). GBL is PIPE's main learning algorithm. EL's purpose is to make the best program found so far an attractor. PIPE executes:

```
GBL
REPEAT
    with probability $P_{el}$ DO EL
    otherwise DO GBL
UNTIL termination criterion is reached
```

Here P_{el} is a user-defined constant in [0;1].

4.3.3 Generation-Based Learning

PIPE learns in successive generations, each comprising five distinct phases: (1) creation of program population, (2) population evaluation, (3) learning from population, (4) mutation of prototype tree, and (5) prototype tree pruning.

(1) Creation of Program Population. A population of programs $\textsc{Prog}_j$ ($0 < j \leq PS$; PS is population size) is generated using the prototype tree PPT, as described in Section 4.2.3. The PPT is grown "on demand" (see Section 4.2.4).

(2) Population Evaluation. Each program $\textsc{Prog}_j$ of the current population is evaluated on the given task and assigned a fitness value $FIT(\textsc{Prog}_j)$ according to the predefined fitness function (see Section 4.3.1). The best program of the current population (the one with the smallest fitness value) is denoted $\textsc{Prog}_b$. The best program found so far (elitist) is preserved in $\textsc{Prog}^{el}$.

(3) Learning from Population. Prototype tree probabilities are modified such that the probability $P(\textsc{Prog}_b)$ of creating $\textsc{Prog}_b$ increases. This procedure is called `adapt_PPT_towards(`$\textsc{Prog}_b$`)`. It increases $P(\textsc{Prog}_b)$ independently of $\textsc{Prog}_b$'s length. This is important as otherwise a strong bias towards creating short programs is induced and hampers evolution. Procedure `adapt_PPT_towards(`$\textsc{Prog}_b$`)` works as follows:

First $P(\textsc{Prog}_b)$ is computed by looking at all PPT nodes N_j used to generate $\textsc{Prog}_b$:

$$P(\textsc{Prog}_b) = \prod_{j:N_j \text{ used to generate } \textsc{Prog}_b} P_j(I_j(\textsc{Prog}_b)) \tag{4.1}$$

where $I_j(\textsc{Prog}_b)$ denotes the instruction of program $\textsc{Prog}_b$ at node position j. Then a target probability P_{TARGET} for $\textsc{Prog}_b$ is calculated:

$$P_{TARGET} = P(\textsc{Prog}_b) + (1 - P(\textsc{Prog}_b)) \cdot lr \cdot \frac{\varepsilon + FIT(\textsc{Prog}^{el})}{\varepsilon + FIT(\textsc{Prog}_b)} \tag{4.2}$$

Here lr is a constant learning rate and ε a positive user-defined constant. Fraction $\frac{\varepsilon + FIT(\text{PROG}^{el})}{\varepsilon + FIT(\text{PROG}_b)}$ implements *fitness-dependent learning (fdl).* Larger steps are taken towards programs with higher quality (lower fitness) than towards programs with lower quality (higher fitness). Constant ε determines the degree of *fdl*'s influence. If $\forall$ $FIT(\text{PROG}^{el})$: $\varepsilon \ll FIT(\text{PROG}^{el})$, then PIPE can use small population sizes because generations containing only low-quality individuals do not affect the PPT much.

Given P_{TARGET}, *all* single node probabilities $P_j(I_j(\text{PROG}_b))$ are increased iteratively (in parallel):

$$\begin{aligned} &\text{REPEAT UNTIL } \; P(\text{PROG}_b) \geq P_{TARGET} : \\ &P_j(I_j(\text{PROG}_b)) := P_j(I_j(\text{PROG}_b)) + c^{lr} \cdot lr \cdot (1 - P_j(I_j(\text{PROG}_b))) \end{aligned} \tag{4.3}$$

Here c^{lr} is a constant influencing the number of iterations. The smaller c^{lr} the higher the approximation precision of P_{TARGET} and the number of required iterations. Setting $c^{lr} = 0.1$ turned out to be a good compromise between precision and speed. Then all adapted vectors $\vec{P}_j$ are renormalized by diminishing the values of all non-increased vector components proportionally to their current value:

$$P_j(I) := P_j(I) \cdot \left(1 - \frac{1 - \sum_{I^* \in S} P_j(I^*)}{P_j(I_j(\text{PROG}_b)) - \sum_{I^* \in S} P_j(I^*)} \right) \forall P_j(I) : I \neq I_j(\text{PROG}_b)$$

Finally, each random constant in PROG_b is copied to the appropriate node in the *PPT*: if $I_j(\text{PROG}_b) = R$ then $R_j := V_j(R)$.

(4) Mutation of Prototype Tree. Mutation is one of PIPE's major exploration mechanisms. Mutation of PPT probabilities is guided by the current best solution PROG_b. PIPE explores the area "around" PROG_b. All probabilities $P_j(I)$ stored in nodes N_j that were accessed to generate program PROG_b are mutated with probability P_{M_p}:

$$P_{M_p} = \frac{P_M}{z \cdot \sqrt{|\text{PROG}_b|}} \tag{4.4}$$

where the user-defined parameter P_M defines the overall mutation probability, z is the number of instructions in instruction set S (see Section 4.1.1) and $|\text{PROG}_b|$ denotes the number of nodes in program PROG_b. To prevent rapid growth of mutation probability P_{M_p} it is made dependent on PROG_b's size. The justification of the square root is empirical: Larger programs improve faster with higher mutation probability. Selected probability vector components are then mutated as follows:

$$P_j(I) := P_j(I) + mr \cdot (1 - P_j(I)) \tag{4.5}$$

where mr is the mutation rate, another user-defined parameter. All mutated vectors $\vec{P}_j$ are finally renormalized:

$$P_j(I) := \frac{P_j(I)}{\sum\limits_{I^* \in S} P_j(I^*)} \qquad \forall P_j(I) : I \in S$$

From Assignment 4.5 one can see that small probabilities (close to 0) are subject to stronger mutations than high probabilities. Otherwise, mutations would tend to have little effect on the next generation.

(5) Prototype Tree Pruning. At the end of each generation the prototype tree is pruned, as described in Section 4.2.4.

4.3.4 Elitist Learning

During elitist learning (EL), the *PPT* is adapted towards the elitist program PROG^{el} by calling `adapt_PPT_towards(`PROG^{el}`)`; then the *PPT* is pruned. However, neither is a population created and evaluated nor are the probabilities of the *PPT* mutated, making EL computationally cheap. EL focuses search on previously discovered promising parts of the search space. It is particularly useful with small population sizes and works efficiently in the case of noise-free problems.

4.3.5 Termination Criteria

PIPE is run either for a fixed number of program evaluations (PE) (time constraint) or until a solution with fitness better than FIT_s is found (quality constraint).

4.3.6 Summary of User-Defined Parameters.

The following above-mentioned parameters have to be set by the user:

P_T :	*Initial Terminal Probability.* The initial probability of selecting an instruction from terminal set T at each node N_j. A high P_T forces PIPE to start its search with small programs (containing few nodes) and prevents programs from growing rapidly.
P_{el} :	*Elitist Update Probability.* Probability of learning from the elitist program $\textsc{Prog}^{el}$ instead of a new generation of programs.
PS :	*Population Size.* The number of programs created and evaluated during one generation.
lr :	*Learning Rate.* Influences the step size for adapting the probabilities of PPT during each learning phase.
ε :	*Fitness Constant.* Determines the impact of fitness-dependent learning by introducing an absolute fitness scale.
P_M :	*Mutation Probability.* Probability of mutating probabilities in PPT. High P_M's stimulate exploration but may destabilize learning.
mr :	*Mutation Rate.* Strength of mutation of a single selected probability vector component of the PPT. A large mr ensures high impact of mutations on future generations. Many small mutations tend to make all PPT probability distributions uniform (see mutation probability above).
T_R :	*Random Constant Threshold.* Probability threshold that defines when to try new values for R_j's. A too high T_R tends to make PIPE forget previously discovered good random constants.
T_P :	*Prune Threshold.* Probability threshold used in the pruning procedure to reduce memory requirements.
PE :	*Program Evaluations.* Maximal number of programs tested during system life (time constraint).
FIT_s :	*Satisfactory Fitness.* Fitness of a satisfactory solution. Once a satisfactory solution is found, the search can be stopped (quality constraint).

4.4 Multiple Outputs

To accommodate for vector–valued outputs PIPE applies *multiple programs* (MPs). If n_O is the number of outputs then a "full" program will consist of n_O independent programs generated according to distinct probabilistic prototype trees. One program is generated for each output and the return value of each program is taken as an output value.

4.5 Conclusion

This chapter presented PIPE – a novel method for automatic program synthesis. It showed how PIPE searches program space by generating successive populations of programs according to a probability distribution over all possible programs with respect to a predefined instruction set. The probability distribution guides the search and is adapted according to the search results. Furthermore, all user–defined parameters were listed and summarized. Finally, a setup for PIPE with multiple outputs was presented.

Chapter 5

Applying PIPE

This chapter shows how PIPE can be applied to solve a variety of tasks. Section 5.1 contains a step by step description on how to setup PIPE for an application. Section 5.2 presents three applications: function regression, 6-bit parity, and 3+8-bit multiplexer. Section 5.3 contains a soccer case study. Section 5.4 concludes this chapter.

5.1 PIPE Setup

To apply PIPE the following steps need to be taken:

1. Training and Test Environment Setup
2. Fitness Function Definition
3. Instruction Set Selection
4. Output Interface Definition (optional)
5. Parameter Setup

5.1.1 Training and Test Environment Setup

A training and test environment that allows for establishing program quality needs to be derived from the problem definition. An environment in this context can be, e.g., a training and a test data set.

5.1.2 Fitness Function Definition

The fitness function must define the goal of PIPE's optimization process. It needs to be setup following the guidelines from Section 4.3.1.

5.1.3 Instruction Set Selection

The choice of the instruction set has great influence on PIPE's performance. In general there is no recipe which instructions to pick, since the choice of an appropriate instruction set is problem-dependent. Only a "weak" guideline can be given as to how to choose a terminal and function set.

Terminal Set Selection

The terminal set must at least include all relevant input variables. It may also include a GRC and further terminal instructions. The minimal constraint on the GRC is to insure closure with respect to type and value range.

Function Set Selection

To select an appropriate function set a priori knowledge about the problem is required. Choosing an inappropriate function set will prevent PIPE from finding a useful solution.

Many problems, however, can be solved using a single *basic function set (bfs)* as a basis and enriching it with further instructions when necessary. Establishing a *bfs* that serves well for a particular group of problems is non-trivial and beyond the scope of this thesis. A *bfs* that has been empirically shown to work well for a wide variety of problems is the function set presented in Section 4.1.1. It has been successfully applied to: function approximation, parity problems, learning in partially observable environments (Sałustowicz and Schmidhuber, 1997a), learning soccer strategies (Sałustowicz, Wiering, and Schmidhuber, 1998), and time series prediction (Sałustowicz and Schmidhuber, 1999b).

5.1.4 Output Interface Definition

The result of applying PROG to data x is denoted as PROG(x). For some problems PROG(x) needs to be transformed to a different output value and/or type to constitute a solution. This is what a predefined output interface does (compare also "wrappers" – Koza, 1992). If, e.g., the instruction set from Section 4.1.1 is used to evolve solutions for a Boolean problem requiring a "true" or "false" as an output, $\textsc{Prog}(x)$ can transformed to accommodate for the Boolean nature of the problem by using the following output interface:

```
if PROG(x) < 0 then ''false'' else ''true''
```

5.1.5 Parameter Setup

All user-defined parameters (see Section 4.3.6) need to be set. In general the optimal parameter setting is problem-dependent. There are, however, a few "rules of thumb" that have empirically been proven to work well:

- Initial terminal probability P_T is the most important parameter. It should be initially set to a high value (e.g., $P_T = 0.8$) to focus the search on small programs first, as smaller programs require less evaluation time. In case PIPE cannot improve its solutions and no larger programs are tired after some generations, PIPE needs to be restarted with a smaller value for P_T that favors larger programs.

- Elitist update probability P_{el} needs to be set to 0 for problems with a noisy program evaluation.

- Population size PS should be kept small (exceptions exist for obtaining speed-ups through parallelization – see Section 7.4) in favor of an increased number of generations.

- Apart from program evaluations PE and satisfactory fitness FIT_S which are the termination criteria, all remaining parameters tend to work well with "standard" values (see Section 5.2).

5.2 Applications

Three distinct problems have been selected to demonstrate how PIPE can be applied:

- function regression

- 6-bit parity

- 3+8-bit multiplexer

All selected problems verify empirically that the "rules of thumb" from Section 5.1.5 work. Each experiment on its own adds more insight into how PIPE works.

Function Regression. The task is to evolve a program constituting an approximation to a continuous, one-dimensional function. A non-trivial function is selected to prevent PIPE from simply guessing it. Since the function is continuous, infinite many fitness values exist and allow for a slow incremental adaptation.

6-Bit Parity. The 6-bit parity problem is a discrete task involving just 65 distinct fitness values. The limited number of fitness values allows for testing PIPE's built-in Occam's razor. Furthermore, 6-bit parity has been selected to show that the same basic function set as for the function regression problem can be applied. Finally, an output interface is presented.

3+8-Bit Multiplexer. The 3+8-bit multiplexer problem has been chosen to verify that PIPE works well with different function sets and program trees of various arities.

5.2.1 Function Regression

The function to be approximated is

$$f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$$

which is plotted in Figure 5.1.

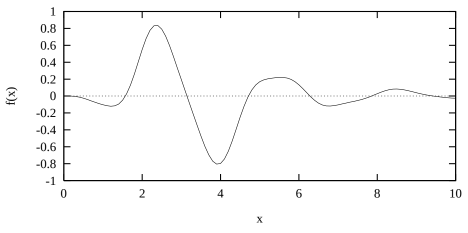

Figure 5.1: $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$

Training and Test Environment

The training data set D_{tr} samples f at 101 equidistant points in the interval [0;10]. The test data set D_{te} samples f at 101 equidistant points in the interval [0.05;10.05]. D_{tr} is used to calculate fitness values during program evolution, and D_{te} is used to test how well the best evolved programs generalize.

Fitness Function

The fitness value of each program PROG is $FIT(\text{PROG}) = \sum_{\forall x \in D_{tr}} |f(x) - \text{PROG}(x)|$. To verify how an evolved program generalizes its generalization performance is calculated: $GEN(\text{PROG}) = \sum_{\forall x \in D_{te}} |f(x) - \text{PROG}(x)|$. To obtain an idea how generalization performances relate to function approximation quality, consider Figure 5.2. The graphs show that with increasing $GEN(\text{PROG})$ approximation quality becomes worse.

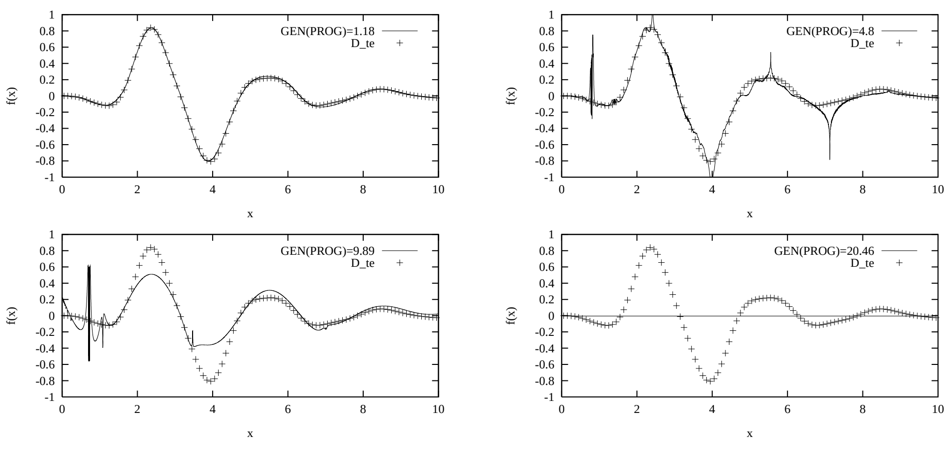

Figure 5.2: Test data set D_{te} and approximations with $GEN(\text{PROG}) = 1.18$ (upper left), $GEN(\text{PROG}) = 4.8$ (upper right), $GEN(\text{PROG}) = 9.89$ (lower left), and $GEN(\text{PROG}) = 20.46$ (lower right).

Instruction Set

Following function and terminal sets have been used: $F = \{+, -, *, \%, sin, cos, exp, rlog\}$ and $T = \{x, R\}$ (see Section 4.1.1). R denotes the generic random constant (GRC) in [0;1).

Output Interface

No specific output interface is required.

Parameter Setup

The termination criteria are set to: PE = 100,000 and FIT_s = 0.001. The initial terminal probability is P_T=0.8. The remaining parameters are set to "standard" values: ε = 0.000001, P_{el}=0.01, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999.

Results

Since PIPE is a stochastic learning algorithm it does not always deliver a solution of same quality. 200 independent runs have been conducted to obtain an overview over best solutions evolved by PIPE. PIPE's performance on training and test data sets (D_{tr} and D_{te}, respectively) is summarized in Figure 5.3. Performance v is plotted against percentage of programs with FIT(Prog) $\leq v$ and GEN(Prog) $\leq v$. PIPE's performance is very similar

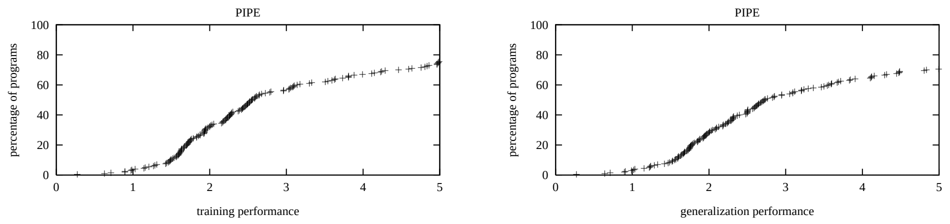

Figure 5.3: Cumulative histograms of PIPE's performance on the training (left) and test (right) data sets for the function regression problem. The plots show the range with the best performing programs. Programs with FIT(Prog) > 5 and GEN(Prog) > 5 are of little interest due to the bad solution quality they embody (see Figure 5.2 for reference).

on both data sets. The evolved programs generalize almost as well as they learn. Though not all evolved programs constitute a good solution, 50% of all runs achieved solutions with FIT(Prog) $<$ 2.6 and GEN(Prog) $<$ 2.8 (see Figure 5.2 for reference).

A program with generalization performance GEN(Prog) = 1.18 that was found by PIPE after 99,390 program evaluations contains 248 nodes and computes:

```
(sin(((x-((cos(((sin((sin(cos((rlog(sin(0.350466))*(((cos(
sin((cos((x-((rlog(cos(((0.359722+cos(x))+(x-0.082538))))*(x-
(0.039232-((x%0.440611)%0.499641))))*0.025812)))-(0.914140*
(x*(0.506207%0.379995))))))*(x-((x%sin(rlog(0.334052)))+rlog(((
x+x)*x)))))%exp(exp((0.743179-0.128703))))+x))))%(((0.507077*
((exp((x-x))-((cos(sin((x-(cos((0.915233%x))-exp(0.709387)))))-
0.492354)%0.840741))%cos((x*0.981004))))*((x%cos(x))*(0.091520*
(0.112682+sin(sin(x))))))+x)))%x)%(sin((((cos(sin(rlog((exp(x)%
cos(0.712427)))))+0.933998)%0.609029)-cos(0.936381)))%(((cos(
0.790039)-(x-0.069650))*sin(x))-x))))+sin(exp(rlog(x))))-((
sin(0.375208)*(exp(rlog(exp(0.697598)))%cos((cos(0.585192)-
0.095603))))+(((0.395458-(0.282354*sin(0.822447)))%(0.533448%
(0.785156*0.918876)))*cos((x-(0.639372%0.524799)))))))-sin((
sin(sin((sin(sin(sin(x)))%sin(cos(0.287498)))))+x))))*(cos(((
0.482642+((0.183318*(0.338145+0.069478))*cos((x+0.496698))))+
(cos((x*0.649953))%cos(0.151858))))%(sin(sin((0.470205+(x%
((exp(rlog(x))%(x+0.684205))+0.058088)))))+((exp(rlog((x%
0.994150)))%cos(0.178977))*(cos(0.785409)*0.700799)))))
```

Conclusion

PIPE exhibits a stochastic learning behavior. Solution quality varies for every evolutionary run. In 50% of all runs, however, PIPE finds programs embodying high quality solutions. Generalization performances are similar to training performances. Programs that perform well on the training set also generalize well. Similar to programs evolved by Koza's GP variant (Koza, 1992), PIPE's programs differ much from programs created by human programmers.

5.2.2 6-Bit Parity

The 6-bit parity function has six Boolean arguments represented by integers: 1 for true and 0 for false. It returns 1 if the number of nonzero arguments is odd and 0 otherwise.

Training and Test Environment

All 64 patterns are used for training.

Fitness Function

The fitness of a program is the number of patterns it classifies *incorrectly.* Best fitness for classifying all patterns correctly is 0 and worst fitness for classifying no patterns correctly is 64.

Instructions Set

The terminal set is set to $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$, where $x_0, x_1, x_2, x_3, x_4, x_5$ are input variables and R the GRC in [0;1). Function set F (and GRC interval) is identical to the function set used for the function regression task in Section 5.2.1.

Output Interface

To fit the Boolean nature of the problem the real-valued output of a program is mapped to 0 if negative and to 1 otherwise.

Parameter Setup

The termination criteria are set to: PE = 500,000 and FIT_s = 0.001. The initial terminal probability is P_T=0.6. The remaining parameters are set to "standard" values: ε = 0.000001, P_{el}=0.01, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999.

Results

100 independent test runs were conducted. Table 5.1 summarizes the results. In 71% of all runs PIPE found a perfect solution within the given time

Table 5.1: Summary of 6-bit parity results.

6-bit parity		
	Program Evaluations	Nodes
solved	min– med –max	min–med–max
71 %	8,100–**75,210**–483,790	25– **63** –231

frame. The median successful run took 75,210 program evaluations. The appearance of solutions differs wildly. Perfect solutions are made of 25 to 231 nodes, while 63 nodes are needed in the median.

Conclusion

Due to its build-in Occam's razor PIPE solves the 6-bit parity problem in 71% of all runs. Experiments without the Occam's razor (not presented) deliver significantly worse results. Furthermore, given an appropriate output interface, a basic function set (the same as for function regression) can be used.

5.2.3 3+8-Bit Multiplexer

The input of the Boolean x+y multiplexer function consists of x address bits a_i and $y = 2^x$ data bits d_j. The target is to output the data bit d_j addressed by the address bits $j_{10} = (a_0 \ldots a_{x-1})_2$. The 3+8-bit multiplexer function has therefore 11 Boolean arguments. Figure 5.4 shows a possible input/output configuration for the 3+8-bit multiplexer.

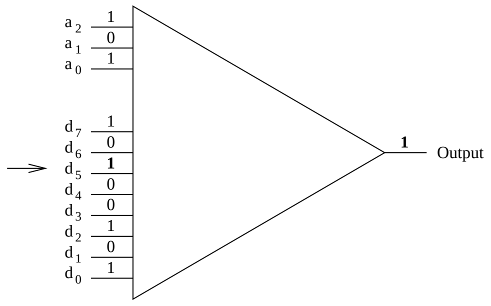

Figure 5.4: The 3+8-bit Boolean multiplexer function with address lines a_0, a_1, a_2 and data lines $d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7$. The output is the value of data bit d_5, which is singled out by address bits $a_1 a_2 a_3$ that are set to $101_2 = 5_{10}$ respectively.

Training and Test Environment

All 2048 possible patterns are used for training.

Fitness Function

The fitness of a program is the number of incorrect outputs when applying all input patterns. Best fitness, if a program evaluates to the correct output for each input pattern, is 0 and worst fitness for not achieving a single correct output is 2048.

Instruction Set

The terminal set is set to $T = \{a_0, a_1, a_2, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$, where $a_0, a_1, a_2, d_0, d_1, d_2, d_3,\ d_4, d_5, d_6, d_7$ are input variables. No distinction is made between address inputs and data inputs for PIPE. The following function set was used: $F = \{and, or, not, if\}$, where if is a three argument function that returns the second argument, if the first argument evaluates to 1, and the third argument otherwise.

Output Interface

An output mapping is not required. Only terminals and functions which output either 1 or 0 are used.

Parameter Setup

The termination criteria are set to: PE = 500,000 and FIT_s = 0.001. The initial terminal probability is P_T=0.8. The remaining parameters are set to "standard" values: ε = 0.000001, P_{el}=0.01, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999.

Results

100 independent test runs were conducted. Table 5.2 summarizes the results. In 93% of all runs PIPE found a perfect solution within the given time

Table 5.2: Summary of 3+8-bit multiplexer results.

	Program Evaluations	Nodes
solved	min– med –max	min–med–max
93 %	17,290–**130,210**–484,400	29– **53** –132

frame. The median successful run took 130,210 program evaluations. The appearance of solutions differs. Perfect solutions are made of 29 to 132 nodes, while 53 nodes are needed in the median.

Conclusion

PIPE is not bound to the function set used for the function regression and the 6-bit parity task. PIPE works well with a completely different, for Boolean problems more intuitive function set. The increased arity of program trees and *PPT* (recall that instruction *if* has three arguments) does not prevent PIPE from solving the 3+8-bit multiplexer problem.

5.2.4 Conclusion

This part of the chapter was dedicated to some basic applications: function approximation, 6-bit parity, and 3+8-bit multiplexer. PIPE found programs embodying good solution for all of them. Solution programs had a similar form to those evolved by Koza's GP variant (Koza, 1992) and differed much from programs created by a human. Comparing across applications shows that PIPE can use "standard" settings for most of its parameters, which facilitates setting it up for different problems. It also reveals that a basic function set can be applied to various problems such as function approximation and 6-bit parity. Not always, however, such a function set is useful. Thus to solve the 3+8-bit multiplexer problem a different, more suitable function set was used. This verified that PIPE is not limited to a single instruction set, but can successfully use different instruction sets and deal with program trees of various arities.

In a next step we will investigate how PIPE behaves in a complex multiagent environment.

5.3 Soccer Case Study

We use simulated soccer to study multiagent learning (Sałustowicz et al., 1998). Each team's players (agents) share action set and policy, but may behave differently due to position-dependent inputs. All agents making up a team are rewarded or punished collectively in case of goals. We conduct simulations with varying team sizes, and compare Probabilistic Incremental Program Evolution (PIPE), and a PIPE version that learns by coevolution (CO-PIPE) to TD-Q learning with linear neural networks (TD-Q). TD-Q is based on learning evaluation functions (EFs) mapping input/action pairs to expected reward. PIPE and CO-PIPE search policy space directly. They synthesize programs that calculate action probabilities from current inputs. The results show that linear TD-Q encounters several difficulties in learning appropriate shared EFs. PIPE and CO-PIPE, however, do not depend on EFs and find good policies faster and more reliably. This suggests that in some multiagent learning scenarios direct search in policy space can offer advantages over EF-based approaches.

5.3.1 Learning in Multiagent Environments

Policy-sharing

Multiagent learning tasks often require several agents to learn to cooperate. In general there may be quite different types of agents specialized in solving particular subtasks. Some cooperation tasks, however, can also be solved by teams of essentially identical agents whose behaviors differ only due to different, situation-specific inputs. Our case study will be limited to such teams of agents of identical type. Each agent's modifiable policy is given by a variable data structure: for each action in a given set of possible actions the current policy determines the conditional probability that the agent will execute this action, given its current input. Each team's members share both action set and adaptive policy. If some multiagent cooperation task indeed can be solved by homogeneous agents then policy-sharing is quite natural as it allows for greatly reducing the number of adaptive free parameters. This tends to reduce the number of required training examples (learning time) and increase generalization performance, e.g., (Nowlan and Hinton, 1992).

Challenges of Multiagent Learning

One challenge is the "partial observability problem" (POP): in general no learner's input will tell the learner everything about its environment (which includes other changing learners). This means that each learner's environment may change in an inherently unpredictable way. Also, in multiagent reinforcement learning (RL) scenarios delayed reward/punishment is typically given to an entire successful/failing team of agents. This provokes the "agent credit assignment problem" (ACAP): the problem of identifying those agents that were indeed responsible for the outcome (Weiss, 1996; Crites and Barto, 1996; Versino and Gambardella, 1997).

Evaluation Functions versus Search through Policy Space

There are two rather obvious classes of candidate algorithms for learning shared policies in multiagent RL. Class I includes traditional singleagent RL algorithms based on adaptive evaluation functions (EFs) (Watkins, 1989; Bertsekas and Tsitsiklis, 1996). Usually online variants of dynamic programming and function approximators are combined to learn EFs mapping input-action pairs to expected discounted future reward. The EFs are then exploited to generate rewarding action sequences.

Methods from class II do not require EFs. Their policy space consists of complete algorithms defining agent behaviors, and they search policy space directly. Members of this class are Levin search (Levin, 1973; Levin, 1984; Solomonoff, 1986; Li and Vitányi, 1993; Wiering and Schmidhuber, 1996b; Schmidhuber, 1997), Genetic Programming (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992) and Probabilistic Incremental Program Evolution (PIPE, Sałustowicz and Schmidhuber, 1997a)

Comparison

In our case study we compare two learning algorithms, each representative of its class: TD-Q learning (Lin, 1993; Peng and Williams, 1996; Wiering and Schmidhuber, 1997) with linear neural networks (TD-Q) and Probabilistic Incremental Program Evolution (PIPE, Sałustowicz and Schmidhuber, 1997a). We also report results for a PIPE variant based on coevolution (CO-PIPE, Sałustowicz, Wiering, and Schmidhuber, 1997a). TD-Q learning and PIPE have both already been successfully applied to interesting singleagent tasks (Lin, 1993; Sałustowicz and Schmidhuber, 1997a) (additionally TD learning (Sutton, 1988) is quite popular due to a successful application to backgammon (Tesauro, 1994)). Linear TD-Q selects actions according to linear neural networks trained with the delta rule (Widrow and Hoff, 1960) to map player inputs to evaluations of alternative actions. Linear networks keep simulation time comparable to that of PIPE and CO-PIPE — more complex approximators would require significantly more computational resources. PIPE and CO-PIPE are based on probability vector coding of program instructions (Schmidhuber, 1999), Population-Based Incremental Learning (Baluja, 1994; Baluja and Caruana, 1995) and tree coding of programs used in variants of Genetic Programming (Cramer, 1985; Koza, 1992). They synthesize programs that calculate action probabilities from inputs. Experiences with programs are stored in adaptive probability distributions over all possible programs. The probability distributions then guide program synthesis.

Soccer

To come up with a challenging scenario for our multiagent learning case study we decided on a non-trivial soccer simulation. Soccer recently received much attention by various multiagent researchers (Sahota, 1993; Asada, Uchibe, Noda, Tawaratsumida, and K. Hosoda, 1994; Littman, 1994, Stone and

Veloso, 1996a; Matsubara, Noda, and Hiraki, 1996). Most early research focused on physical coordination of soccer playing robots (Sahota, 1993; Asada et al., 1994). There also have been attempts at *learning* low-level cooperation tasks such as pass play (Stone and Veloso, 1996; Matsubara et al., 1996; Nadella and Sen, 1996). Littman (1994) used a tiny 5×4 grid world with two single opponent players to learn soccer strategies. Stone and Veloso (1996b) mentioned early that even team strategies might be learnable by TD(λ) or genetic methods. Learning entire *team* soccer strategies in more complex environments was then pursuit by (Sałustowicz, Wiering, and Schmidhuber, 1997a,b,1998; Luke, Hohn, Farris, Jackson, and Hendler, 1997; Stone and Veloso, 1998). Our case study involves simulations with varying sets of continuous-valued inputs and actions, simple physical laws to model ball bounces and friction, and up to 11 players (agents) on each team.

Results Overview

The results indicate: linear TD-Q has severe problems in learning and keeping appropriate shared EFs. It learns relatively slowly, and once it achieves fairly good performance it tends to break down. This effect becomes more pronounced as team size increases. PIPE and CO-PIPE learn faster than linear TD-Q and continuously increase their performance. This suggests that PIPE-like, EF-independent techniques can easily be applied to complex multiagent learning scenarios with policy-sharing agents, while more sophisticated and time consuming EF-based approaches may be necessary to overcome TD-Q's current problems.

Outline

Section 5.3.2 describes the soccer simulation. Section 5.3.4 describes PIPE and CO-PIPE. Section 5.3.5 describes TD-Q. Section 5.3.6 reports experimental results. Section 5.3.7 concludes.

5.3.2 Soccer Simulator

Our discrete-time simulations involve two teams. There are either 1, 3 or 11 players per team. Players can move or shoot the ball. Each player's abilities are limited (1) by the built-in power of its pre-wired action primitives and (2) by how informative its inputs are. We conduct two types of simulations. *"Simple"* simulations involve less informative inputs and less sophisticated actions than *"complex"* simulations.

Soccer Field

We use a two dimensional continuous Cartesian coordinate system. The field's southwest and northeast corners are at positions (0,0) and (4,2) respectively. As in indoor soccer the field is surrounded by impassable walls except for the two goals centered in the east and west walls (see Figure 5.5(left)). Only the ball or a player with ball can enter the goals. Goal width (y-extension) is 0.4, goal depth (x-extension beyond the field bounds) is 0.01. The east goal's "middle" is denoted $m_{ge} = (x_{ge}, y_g)$ with $x_{ge} = 4.01$ and $y_g = 1.0$ (see Figure 5.5(right)). The west goal's middle is at $m_{gw} = (x_{gw}, y_g)$ with $x_{gw} = -0.01$.

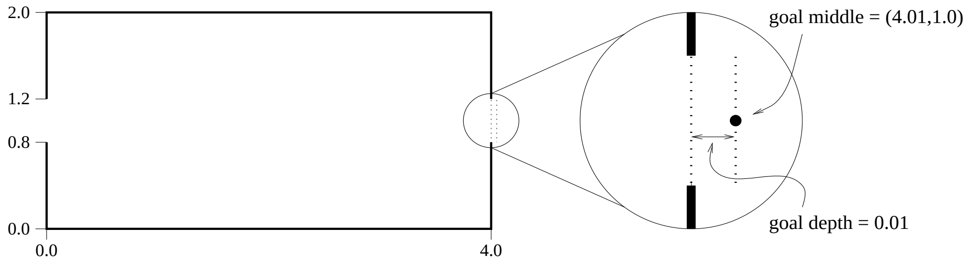

Figure 5.5: *Left:* Soccer field. *Right:* Depth and "middle" m_{ge} of east goal (enlarged).

Ball/Scoring

The ball is a circle with variable center coordinates $c_b = (x_b, y_b)$, variable direction $\vec{o}_b$ and fixed radius $r_b = 0.01$. Its speed at time t is denoted $v_b(t)$. After having been shot the ball's initial speed is v_b^{init} (max. 0.12 units per time step). Each following time step the ball slows down due to friction: $v_b(t+1) = v_b(t) - 0.005$ until $v_b(t) = 0$ or it is picked up by a player (see below). The ball bounces off walls obeying the law of equal reflection angles as depicted in Figure 5.6. Bouncing causes an additional slow-down: $v_b(t+1) = v_b(t) - 0.005 - 0.01$. A goal is scored whenever $0.8 < y_b < 1.2 \wedge (x_b < 0 \vee x_b > 4.0)$.

Players

There are two teams consisting of Z homogeneous players $T_{east} = \{pe_1, pe_2, \ldots, pe_Z\}$ and $T_{west} = \{pw_1, pw_2, \ldots, pw_Z\}$. We vary team size: Z can be

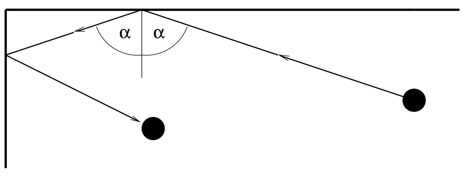

Figure 5.6: Ball “reflected” by wall.

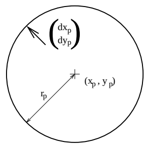

Figure 5.7: Player: center c_p = (x_p, y_p), radius r_p and orientation $\vec{o}_p$ = $\binom{dx_p}{dy_p}$.

1, 3 or 11. At a given time step each player $p \in T_{east} \cup T_{west}$ is represented by a circle with variable center $c_p = (x_p, y_p)$, fixed radius $r_p = 0.025$ and variable orientation $\vec{o}_p = \binom{dx_p}{dy_p}$ (see Figure 5.7). Players are “solid”. If player p, coming from a certain angle, attempts to traverse a wall then it “glides” on it, loosing only that component of its speed which corresponds to the movement direction hampered by the wall. Players p_i and p_j collide if $dist(c_{p_i}, c_{p_j}) < r_p$, where $dist(c_i, c_j)$ denotes Euclidean distance between points c_i and c_j. Collisions cause both players to bounce back to their positions at the previous time step. If one of them has owned the ball then the ball will change owners (see below).

Initial Setup

A game lasts from time $t = 0$ to time t_{end}. There are fixed initial positions for all players and the ball (see Figure 5.8). Initial orientations are $\vec{o}_p = \binom{-1}{0}$ $\forall p \in T_{east}$ and $\vec{o}_p = \binom{1}{0}$ $\forall p \in T_{west}$.

Action Framework/Cycles

Until one of the teams scores, at each discrete time step $0 \leq t < t_{end}$ each player executes a “cycle” (the temporal order of the $2 \cdot Z$ cycles is chosen randomly). A cycle consists of: (1) attempted ball collection, (2) input computation, (3) action selection, (4) action execution and (5) attempted ball collection. Once all $2 \cdot Z$ cycles have been executed we move the ball if $v_b > 0$. If a team scores or $t = t_{end}$ then all players and ball are reset to their initial positions.

(1) Attempted Ball Collection. A player p successfully collects ball b if its radius $r_p \leq dist(c_p, c_b)$. We then set $c_b := c_p, v_b := 0$. Now the ball will move with p and can be shot by p.

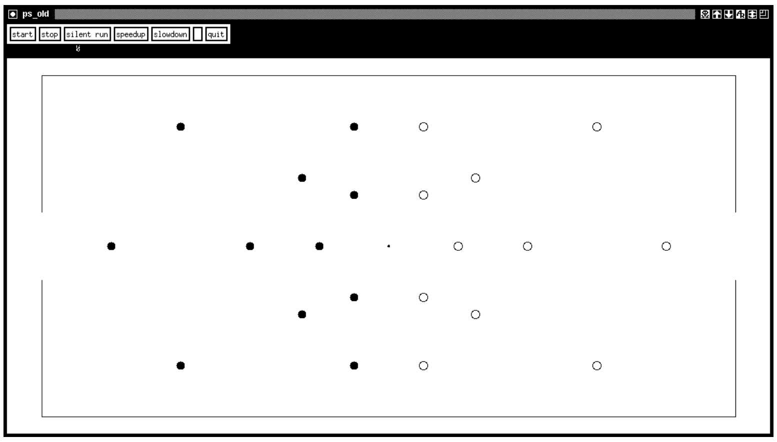

Figure 5.8: 22 players and ball in initial positions. Players of a 1 or 3 player team are those furthest in the back (defenders and/or goalkeeper).

(2) Input Computation. In *simple* simulations player p's input at a given time is a *simple* input vector $\vec{i}_s(p,t)$. In *complex* simulations it is a *complex* input vector $\vec{i}_c(p,t)$.

Simple vector $\vec{i}_s(p,t)$ has 14 components: (1) Three boolean inputs (coded with 1=true and -1=false) that tell whether player p/a team member/an opponent has the ball. (2) Polar coordinates (distance, angle) of both goals and the ball with respect to pole c_p and polar axis $\vec{o}_p$ (player-centered coordinate system). (3) Polar coordinates of both goals relative to a ball-centered coordinate system with pole c_b and polar axis $\vec{o}_b$ — if $v_b = 0$, then $\vec{o}_b = \vec{0}$ and the angle towards both goals is defined as 0. (4) Ball speed. Note that these inputs are not sufficient to make the environment fully observable — e.g, there is no information about positions of other players.

The 56-dimensional complex vector $\vec{i}_c(p,t)$ is a concatenation of $\vec{i}_s(p,t)$ and 21 $c_p/\vec{o}_p$-based polar coordinates of all other players ordered by (a) teams and (b) ascending distances to p. The environment still remains partially observable, however, since the player orientations and changing behaviors are not included in the inputs.

TD-Q's, PIPE's, and CO-PIPE's input representation of distance d (angle α) is $\frac{5-d}{5}$ $(e^{-20\cdot\alpha^2})$. This helps TD-Q since it makes close distances and small angles appear more important to TD-Q's linear networks.

(3) Action Selection. See Sections 5.3.4 and 5.3.5.

(4) Action Execution. Depending on the simulation type, player p may execute either *simple* actions from action set $ASET_S$ or *complex* actions from action set $ASET_C$. $ASET_S$ contains:

- *go_forward:* move player p 0.025 units in its current direction $\vec{o}_p$ if without ball and $0.8 \cdot 0.025$ units otherwise.
- *turn_to_ball:* change direction $\vec{o}_p$ of player p such that $\vec{o}_p := \binom{x_b - x_p}{y_b - y_p}$
- *turn_to_goal:* change direction $\vec{o}_p$ of player p such that $\vec{o}_p := \binom{x_{ge} - x_p}{y_g - y_p}$, if $p \in T_{west}$ and $\vec{o}_p := \binom{x_{gw} - x_p}{y_g - y_p}$, if $p \in T_{east}$.
- *shoot:* If p does not own the ball then do nothing. Otherwise, to allow for imperfect, noisy shots, execute $turn(\alpha_{noise})$ which sets $\vec{o}_p := \binom{cos(\alpha_{noise}) \cdot dx_p - sin(\alpha_{noise}) \cdot dy_p}{sin(\alpha_{noise}) \cdot dx_p + cos(\alpha_{noise}) \cdot dy_p}$, where α_{noise} is picked uniformly random from $-5° \leq \alpha_{noise} \leq 5°$. Then shoot ball in direction $\vec{o}_b := \vec{o}_p$. Initial ball speed is $v_b^{init} = 0.12$. Noise makes long shots less precise than close passes.

Complex actions in $ASET_C$ are parameterized. They allow for pre-wired cooperation but also increase action space. Parameter α stands for an angle, P/O stands for some teammate player's/opponent's index from $\{1..Z-1\}/\{1..Z\}$. Indices P and O are sorted by distances to the player currently executing an action, where closer teammate players/opponents have lower indices. For TD-Q α is either picked from $s_1 = \{0, \frac{\pi}{4}, \frac{\pi}{2}, -\frac{\pi}{4}, -\frac{\pi}{2}\}$ or from $s_2 = \{0, \frac{2}{5}\pi, \frac{4}{5}\pi, -\frac{2}{5}\pi, -\frac{4}{5}\pi\}$. PIPE uses continuous angles. Player p may execute the following complex actions from $ASET_C$:

- *goto_ball(α):* If p owns ball do nothing. Otherwise execute *turn_to_ball*, then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$) and finally *go_forward*,
- *goto_goal(α):* First execute *turn_to_goal*, then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$) and finally *go_forward*.
- *goto_own_goal(α):* First execute *turn(β)* such that $\vec{o}_p := \binom{x_{gw} - x_p}{y_g - y_p}$ (if $p \in T_{west}$) or $\vec{o}_p := \binom{x_{ge} - x_p}{y_g - y_p}$ (if $p \in T_{east}$); then $turn(\alpha)$ (TD-Q: $\alpha \in s_1$); finally *go_forward*.

- *goto_player(P,α):* First execute *turn(β)* such that $\vec{o}_p := \binom{x_P - x_p}{y_P - y_p}$, then *turn*($\alpha$) (TD-Q: $\alpha \in s_2$) and finally *go_forward*. Here $(P, p \in T_{east} \vee P, p \in T_{west}) \wedge P \neq p$.

- *goto_opponent(O,α):* First execute *turn(β)* such that $\vec{o}_p := \binom{x_O - x_p}{y_O - y_p}$, then *turn*($\alpha$) (TD-Q: $\alpha \in s_2$) and finally *go_forward*. Here $(p \in T_{east} \wedge O \in T_{west}) \vee (p \in T_{west} \wedge O \in T_{east})$.

- *pass_to_player(P):* First execute *turn(β)* such that $\vec{o}_p := \binom{x_P - x_p}{y_P - y_p}$, then *shoot*. Here $P, p \in T_{east} \vee P, p \in T_{west}$. Initial ball speed is set to $v_b^{init} = 0.005 + \sqrt{2 \cdot 0.005 \cdot dist(c_p, c_P)}$. If $v_b^{init} > 0.12$ then $v_b^{init} := 0.12$. This ensures that the ball will arrive at c_P at a slow speed, if the distance to the player is not larger than 1.5 (“maximal shooting distance”).

- *shoot_to_goal:* First execute *turn_to_goal*, then *shoot*, where initial ball speed is set to $v_b^{init} = 0.005 + \sqrt{2 \cdot 0.005 \cdot dist(c_p, m_g)}$, where $m_g = m_{ge}$ if $p \in T_{west}$ and $m_g = m_{gw}$ if $p \in T_{east}$. If $v_b^{init} > 0.12$ then $v_b^{init} := 0.12$.

5.3.3 Training and Test Environment

We conduct two different types of simulations – simple and complex. During simple simulations we use simple input vectors $\vec{i}_s(p,t)$ and simple actions from $ASET_S$. During complex simulations we use complex input vectors $\vec{i}_c(p,t)$ and complex actions from $ASET_C$. In simple simulations we compare TD-Q's, PIPE's and CO-PIPE's behavior as we vary team size. In complex simulations we study the algorithms' performances in case of more sophisticated action sets and more informative inputs. Informative inputs are meant to decrease POP's significance. On the other hand, they increase the number of adaptive parameters. For a statistical evaluation we perform 10 independent runs for each combination of simulation type, learning algorithm and team size.

Opponent and Competitor

PIPE and TD-Q are trained against a “biased random opponent” *BRO*, while CO-PIPE learns through coevolution.

BRO randomly executes simple actions from $ASET_S$. *BRO* is not a bad player due to the initial bias in the action set. For instance, *BRO* greatly prefers shooting at the opponent's goal over shooting at its own. If we let *BRO* play against a non-acting opponent *NO* (all *NO* can do is block) for twenty 5000 time step games then *BRO* wins against *NO* with on average 71.5 to 0.0 goals for team size 1, 44.5 to 0.1 goals for team size 3, 108.6 to 0.5 goals for team size 11.

We also designed a simple but good team *GO* by hand, which serves as a reference competitor. *GO* consists of players which move towards the ball as long as they do not have it, and shoot it at the opponent's goal otherwise. If we let *GO* play against *BRO* for twenty 5000 time step games then *GO* wins with on average 417 to 0 goals for team size 1, 481 to 0 goals for team size 3, and 367 to 3 goals for team size 11. Note that *GO* implements a non-cooperative (singleagent) strategy. Small *GO* teams perform extremely well — larger *GO* teams with many interacting agents, however, do not (see team size 11).

Simple Simulations

We play 3300 games of length $t_{end} = 5000$ for team sizes 1, 3 and 11.

Complex Simulations

In complex simulations we focus on team size 11. One run with complex actions and more informative inputs consists of 1200 games, each lasting for $t_{end} = 5000$ time steps.

Testing

Every 100 games we test current performance by playing 20 test games (no learning) against *BRO* and summing the score results. With PIPE and CO-PIPE we test the current best-of-generation program during performance evaluations (except for the first evaluation where we test a random program).

5.3.4 PIPE and CO-PIPE

We use PIPE as described in Chapter 4 except for "elitist learning" which we omit due to high environmental stochasticity. We also use PIPE to coevolve programs. There each population consists of only two programs with mutually dependent performance. Coevolutionary PIPE (CO-PIPE) works just like PIPE, except that: (1) To evaluate both programs of a population we let them play against each other. (2) The next generation consists of the winner and a new program generated according to the adapted *PPT*. (3) Among programs with equal fitness and length we prefer former winners. (4) We do not use *fitness dependent learning*, as the fitness function changes over time.

Fitness Function

PIPE. The fitness of a main Program PROGRAM is *FIT*(PROGRAM) = *100 - number of goals scored by* PROGRAM *+ number of goals scored by opponent.* Offset "100" ensures that fitness values remain non-negative in our experiments.

CO-PIPE. The fitness of each of the two main programs (PROGRAM$_1$ and PROGRAM$_2$) in the population depends on the other program's (opponent's) performance: *FIT*(PROGRAM$_i$) = *100 - number of goals scored by* PROGRAM$_i$ *+ number of goals scored by* PROGRAM$_j$, where $i, j \in \{1, 2\}$ and $i \neq j$. Again offset "100" ensures that fitness values remain non-negative in our experiments.

Instruction Set

We use $F = \{+, -, *, \%, sin,\ cos, exp, rlog\}$ (see Section 4.1.1) and $T = \{\vec{i}(p,t)_1, \ldots, \vec{i}(p,t)_v, R\}$, where R represents the *generic random constant* $\in$ [0;1) and $\vec{i}(p,t)_j$ $1 \leq j \leq v$ denotes component j of a vector $\vec{i}(p,t)$ with v components. For simple simulations we set $\vec{i}(p,t) := \vec{i}_s(p,t)$ and for complex simulations we set $\vec{i}(p,t) := \vec{i}_c(p,t)$.

Output Interface

PIPE synthesizes programs which, given player p's input vector $\vec{i}(p,t)$, select actions from $ASET$. In simple simulations we set $ASET := ASET_S$ and in complex simulations we set $ASET := ASET_C$.

Action Selection. Action selection depends on 5 (8) variables when simple (complex) actions are used: the "greediness" parameter $g \in I\!R$, and 4 (7) "action values" $A_a \in I\!R$, $\forall a \in ASET$. Action $a \in ASET$ is selected with probability P_{A_a} according to the Boltzmann-Gibbs distribution at temperature $\frac{1}{g}$:

$$P_{A_a} := \frac{e^{A_a \cdot g}}{\sum_{\forall j \in ASET} e^{A_j \cdot g}} \qquad \forall a \in ASET \tag{5.1}$$

All A_a and g are calculated by a program.

Programs. In simple simulations a main program PROGRAM consists of a program PROG^g which computes the greediness parameter g and 4 "action programs" PROG^a ($a \in ASET_S$). In complex simulations we need PROG^g, 7 action programs PROG^a ($a \in ASET_C$), programs $\text{PROG}^{a\alpha}$ for each angle parameter, programs PROG^{aP} for each player parameter and programs PROG^{aO} for each opponent parameter (for actions using these parameters). The result of applying PROG to data x is denoted $\text{PROG}(x)$. Given $\vec{i}(p,t)$, $\text{PROG}^a(\vec{i}(p,t))$ returns A_a and $g := |\text{PROG}^g(\vec{i}(p,t))|$. An action $a \in ASET$ is then selected according to the Boltzmann-Gibbs rule — see Assignment (5.1). In the case of complex actions programs $\text{PROG}^{a\alpha}$, PROG^{aP} and PROG^{aO} return values for all parameters of action a: $\alpha := \text{PROG}^{a\alpha}(\vec{i}(p,t))$, $P := 1 + (|round(\text{PROG}^{aP}(\vec{i}(p,t)))| \; mod \; (Z-1))$, $O := 1 + (|round(\text{PROG}^{aO}(\vec{i}(p,t)))| \; mod \; Z)$. Recall that Z is the number of players per team.

All programs PROG^a, $\text{PROG}^{a\alpha}$, PROG^{aP}, and PROG^{aO} are generated according to distinct *probabilistic prototype trees* PPT^a, $PPT^{a\alpha}$, PPT^{aP}, and PPT^{aO}, respectively.

Parameter Setup

Parameters for all PIPE and CO-PIPE runs are: P_T=0.8, ε=1, P_{el}=0 lr=0.2, P_M=0.1, mr=0.2, T_R=0.3, T_P=0.999999. For PIPE we use a population size of PS=10, while for CO-PIPE we use PS=2, as mentioned above.

5.3.5 TD-Q Learning

One of the most widely known and promising EF-based approaches to reinforcement learning is TD-Q learning (Sutton, 1988; Watkins, 1989; Peng and Williams, 1996; Wiering and Schmidhuber, 1997). Here Wiering uses an offline TD(λ) Q-variant (Lin, 1993), which he describes as follows (Sałustowicz et al., 1998):

For efficiency reasons our TD-Q version uses linear neural networks (networks with hidden units require too much simulation time). To implement policy-sharing we use the same networks for all players of a team. The goal of the networks is to map the player-specific input $\vec{i}(p,t)$ to action evaluations $Q(\vec{i}(p,t),a_1),\ldots,Q(\vec{i}(p,t),a_N)$, where N denotes the number of possible actions. We reward the players equally whenever a goal has been made or the game is over.

Simple Action Selection

In simple simulations we use a different network for each of the four actions $\{a_1, \ldots, a_4\}$. To select an action $a(p,t)$ at time t for player p we first calculate Q-values of all actions. The Q-value of action a_k, given input $\vec{i}(p,t)$ is

$$Q(\vec{i}(p,t), a_k) := \vec{w}^k \cdot \vec{i}(p,t) + b^k \tag{5.2}$$

where $\vec{w}^k$ is the weight vector for action network k and b^k is its bias strength. Once all Q-values have been calculated, a single action $a(p,t)$ is chosen according to the Boltzmann-Gibbs rule — see Assignment (5.1). Unlike PIPE, which evolves the greediness parameter, TD-Q needs an *a priori* value for g.

Complex Action Selection

Since complex actions may have 0, 1, or 2 parameters we use a natural, modular, tree-based architecture. Instead of using continuous angles we use discrete angles (see Section 5.3.2). The root node contains networks $N^{a_1}, \ldots, N^{a_7}$ for evaluating "abstract" complex actions neglecting the parameters, e.g., *pass_to_player*. Some specific root-network N^{a_k}'s "angle son networks" $N^{a_k}_{\alpha_1}, \ldots, N^{a_k}_{\alpha_5}$ are then used for selecting the angle parameter. Similarly, player and opponent parameters are selected using "player son networks" and "opponent son networks", respectively. For instance, if an action contains both player and angle parameters, then there are "son networks" for player-parameters and "son networks" for angle parameters. The complete tree contains 64 linear networks.

After computing the seven "abstract" complex action Q-values according to Equation (5.2), one of the seven is selected according to the Boltzmann-Gibbs rule — see Assignment (5.1). If the selected action requires parameters we use Equation (5.2) to compute the Q-values of all required parameters and select a value for each parameter according to the Boltzmann-Gibbs rule.

TD-Q Learning

For both simple and complex simulations we use an offline TD(λ) Q-variant similar to Lin's (1993). Each game consists of separate trials. At trial start we set time-pointer t to current game time t^c. We increment t after each cycle. The trial stops once one of the teams scores or the game is over.

Denote the final time-pointer by t^*. We want the Q-value $Q(\vec{i}(p,t), a_k)$ of selecting action a_k given input $\vec{i}(p,t)$ to approximate

$$Q(\vec{i}(p,t), a_k) \sim \mathcal{E}(\gamma^{t^*-t} R(t^*)),$$

where $\mathcal{E}$ denotes the expectation operator, $0 \leq \gamma \leq 1$ the discount factor which encourages quick goals (or a lasting defense against opponent goals), and $R(t^*)$ denotes the reinforcement at trial end (-1 if opponent team scores, 1 if own team scores, 0 otherwise).

To learn these Q-values we monitor player experiences in player-dependent history lists with maximum size H_{max}. At trial end player p's history list $H(p)$ is

$$H(p) := \{\{\vec{i}(p,t^1), a(p,t^1), V(\vec{i}(p,t^1))\}, \ldots, \{\vec{i}(p,t^*), a(p,t^*), V(\vec{i}(p,t^*))\}\}.$$

Here $V(\vec{i}(p,t)) := Max_k\{Q(\vec{i}(p,t), a_k)\}$, and t^1 denotes the start of the history list: $t^1 := t^c$, if $t^* < H_{max}$, and $t^1 := t^* - H_{max} + 1$ otherwise.

After each trial we calculate examples using offline TD-Q learning. For each player history list $H(p)$, we compute desired Q-values $Q^{new}(p,t)$ for selecting action $a(p,t)$, given $\vec{i}(p,t)$ $(t = t^1, \ldots, t^*)$ as follows:

$$Q^{new}(p,t^*) := R(t^*).$$
$$Q^{new}(p,t) := \gamma \cdot [\lambda \cdot Q^{new}(p,t+1) + (1-\lambda) \cdot V(\vec{i}(p,t+1))].$$

λ determines future experiences' degree of influence.

To evaluate the selected complex action parameters we store them in history lists as well. Their evaluations are updated on the Q^{new}-values of their (parent) "abstract" complex actions — Q-values of selected action parameters are not used for updates of other previously selected action parameters (or selected actions).

Once all players have created TD-Q training examples, we train the selected networks to minimize their TD-Q errors. All player history-lists are processed by dovetailing as follows: we train the networks starting with the first history list entry of player 1, then we take the first entry of player 2, etc. Once all fist entries have been processed we start processing the second entries, and so on. The networks are trained using the delta-rule (Widrow and Hoff, 1960) with learning rate lr_n.

Parameter Setup

After a coarse search through parameter space we used the following parameters for all TD-Q runs: γ=0.99, λ=0.9, H_{max}=100. All network weights are randomly initialized in $[-0.01, 0.01]$. During each run the Boltzmann-Gibbs rule's greediness parameter g is linearly increased from 0 to 60. For simple simulations we set lr_n=0.0001 and for complex simulations we set lr_n=0.001 (several other parameter values led to worse results).

5.3.6 Experimental Results

Simple Simulations

Results. We compare average score differences achieved during all test phases. Figure 5.9 shows results for PIPE, CO-PIPE, and TD-Q. It plots goals scored by learner and opponent (*BRO*) against number of games used for learning. Larger teams score more frequently because some of their players start out closer to the ball and the opponent's goal.

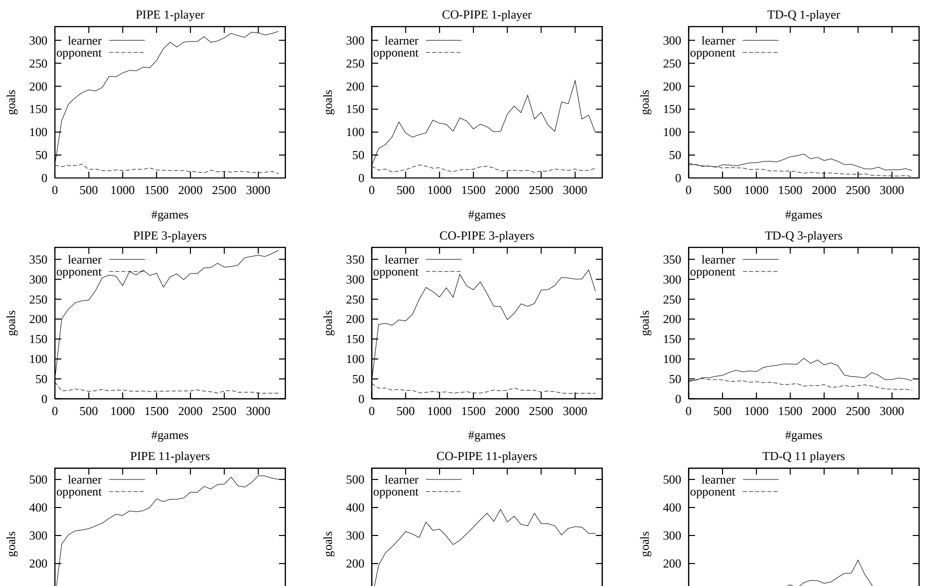

Figure 5.9: Average number of goals scored during all test phases, for team sizes 1, 3, 11.

PIPE learns fastest and always finds quickly an appropriate policy *regardless* of team size. Its score differences continually increase. CO-PIPE performs worse than PIPE, but is still able to find good policies with respect to *BRO*. Note, however, that CO-PIPE's task is more difficult than PIPE's or TD-Q's. It never "sees" *BRO* during training and therefore has no reason for optimizing its strategy against it. Stochastic fluctuations in CO-PIPE's performance tend to level out with increasing team size.

TD-Q also improves, but in a less spectacular way. It always learns more slowly than PIPE and CO-PIPE. It tends to increase score differences until it scores roughly twice as many goals as in the beginning (when actions are still random). Then, however, the score differences start declining. There are several reasons for TD-Q's slowness and breakdown: (1) POP makes learning appropriate EFs difficult. (2) The linear neural networks cannot keep useful EFs but tend to unlearn them instead. (3) Unlike PIPE, linear TD-Q suffers from ACAP: it needs to assign proper credit to individual player actions but fails to pick out the truly useful ones.

For each learning algorithm and *GO*, Table 5.3 lists results against *BRO* (averages over ten runs).

Table 5.3: Results of PIPE, CO-PIPE, TD-Q, and *GO* playing against *BRO*.

team size		*GO*	PIPE	CO-PIPE	TD-Q
1	**max. score difference**	**417**	**310**	**192**	**42**
	av. goals ± st.d.	417±6	320±42	212±97	52±14
	av. *BRO* goals ± st.d.	0±0	10±7	20±10	10±3
	achieved after games	n.a.	3300	3000	1700
3	**max. score difference**	**481**	**359**	**310**	**70**
	av. goals ± st.d.	481±8	373±86	324±62	102±14
	av. *BRO* goals ± st.d.	0±1	14±6	14±11	32±8
	achieved after games	n.a.	3300	3200	1700
11	**max. score difference**	**364**	**481**	**357**	**154**
	av. goals ± st.d.	367±18	512±129	393±53	212±84
	av. *BRO* goals ± st.d.	3±1	31±23	36±27	58±23
	achieved after games	n.a.	3100	1900	2500

The hand-made *GO* team outperforms (in terms of score difference) any of the 1 and 3 player teams. In the 11 player case, however, it plays worse than PIPE, while CO-PIPE's performance is comparable. This indicates that: (1) Successful singleagent strategies may not suit larger teams. (2) Useful strategies for large teams are learnable by direct policy search.

Conclusion. Comparing best programs of several successive generations revealed that PIPE and CO-PIPE are able to: (1) quickly identify the inputs that are relevant for selecting actions, and (2) find programs that compute useful action probabilities given the selected inputs. PIPE's and CO-PIPE's ability to set the greediness parameter helps to control exploration as it makes action selection more or less stochastic depending on the inputs.

Some linear TD-Q runs led to good performance. This implies clusters (or niches) in weight vector space that contain good solutions. TD-Q's dynamics, however, do not always lead towards such niches. Furthermore, sudden performance breakdowns hint at a lack of stability of good solutions. An indepth analysis of TD-Q's instability problems is in (Sałustowicz et al., 1998).

Complex Simulations

Results. Figure 5.10 shows the average number of goals scored by PIPE, CO-PIPE, and TD-Q (learners) in comparison to *BRO* (opponent) during all test phases.

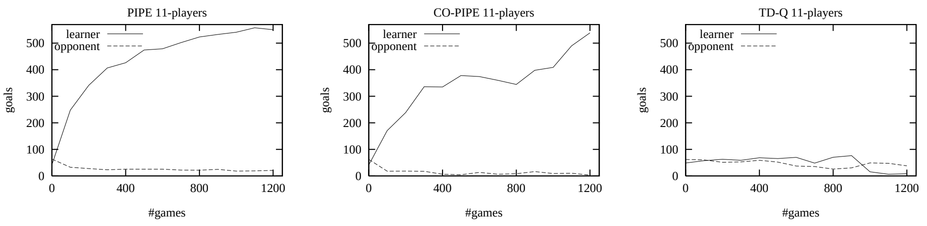

Figure 5.10: Average number of goals (means of 10 independent runs) for PIPE *(left)*, CO-PIPE *(middle)*, and TD-Q *(right)* vs. *BRO* using complex actions and inputs.

PIPE and CO-PIPE quickly find successful strategies. PIPE's performance steadily increases while CO-PIPE's is slightly more stochastic. In the long run (after 3300 games – not shown), however, both are very similar. Note again, though, that CO-PIPE solves a more difficult task — it is tested against an opponent that it never meets during training.

Linear TD-Q initially does worse than its opponent. It does learn to beat *BRO* by about 50 % but then breaks down completely. Examining all single runs revealed that TD-Q's average score results were strongly influenced by a single good run that scored up to 471 goals. Once this run's performance broke down after 1000 games the average declined to 16 goals.

We compare maximal average score differences in Table 5.4. PIPE and CO-PIPE both achieve score differences that are significantly better than

GO's. Linear TD-Q does not.

Table 5.4: Maximal average score differences against *BRO* for different learning methods and *GO*.

	GO	PIPE	CO-PIPE	TD-Q
max. score difference	**364**	**530**	**536**	**46**
av. goals ± st.d.	367±18	551±215	539±220	76±140
av. *BRO* goals ± st.d.	3±1	21±35	3±4	30±29
achieved after games	n.a.	1200	1200	900

Conclusion. Complex actions embody stronger initial bias and make cooperation easier, while more informative inputs make the POP less severe. In principle, this allows for better soccer strategies. PIPE and CO-PIPE are able to exploit this and perform better than with simple actions (compare Figures 5.9 and 5.10 and Tables 5.3 and 5.4). Linear TD-Q does not. It still suffers from the same problems as during simple simulations.

5.3.7 Conclusion

In a simulated soccer case study with policy-sharing agents we compared direct policy search methods (PIPE and coevolutionary CO-PIPE) and an EF-based one (linear TD-Q). All competed against a biased random opponent (*BRO*). PIPE and CO-PIPE always easily learned to beat this opponent regardless of team size, amount of information conveyed by the inputs, or complexity of actions. In particular, CO-PIPE outperformed *BRO* without ever meeting it during the training phase. TD-Q achieved performance improvements, too, but its results were less exciting, especially in case of several agents per team, more informative inputs, and more sophisticated actions.

PIPE and CO-PIPE found good strategies by simultaneously: (1) identifying relevant inputs, (2) making action probabilities depend on relevant inputs only, (3) evolving programs that calculate useful conditional action probabilities. Another important aspect is: unlike TD-Q, PIPE and CO-PIPE learn to map inputs to "greediness values" used in the (Boltzmann-Gibbs) exploration rule. This enables them to pick actions more or less stochastically and control their own exploration process.

Wiering identifies that TD-Q's problems are due to a combination of reasons (Sałustowicz et al., 1998): **(1)** *Linear networks.* Linear networks have limited expressive power. They seem unable to learn *and* keep appropriate evaluation functions (EFs). **(2)** *Partial observability.* Q-learning assumes that the environment is fully observable; otherwise it is not guaranteed to work. Still, Q-learning variants already have been successfully applied to partially observable environments, e.g., (Crites and Barto, 1996). Our soccer scenario's POP, however, seems harder to overcome than POPs of many scenarios studied in previous work. **(3)** *Agent credit assignment problem (ACAP)* (Weiss, 1996; Versino and Gambardella, 1997): how much did some agent contribute to team performance? ACAP is particularly difficult in the case of multiagent soccer. For instance, a particular agent may do something truly useful and score. Then all the other agents will receive reward, too. Now the TD networks will have to learn an evaluation function (EF) mapping input-action pairs to expected discounted rewards based on experiences with player actions that have little or nothing to do with the final reward signal. This problem is actually independent of whether policies are shared or not. **(4)** *Instability.* Using player-dependent history lists, each player learns to evaluate actions given inputs by computing updates based on its own TD return signal. The players collectively update their shared EF which can lead to significant "shifts in policy space" and to "unlearning" of previous knowledge. This may lead to performance breakdowns.

Our multiagent scenario seems complex enough to require more sophisticated and time-consuming EF-based approaches than TD-Q. In principle, however, EFs are not necessary for finding good or optimal policies. Sometimes, particularly in the presence of POPs and ACAPs, it can make more sense to search policy space directly. That is what PIPE and CO-PIPE do. Recently, however, a new promising EF-based approach termed *CMAC models* has been developed (Wiering, Sałustowicz, and Schmidhuber 1998, 1999, to appear). CMAC models combines world models (Moore and Atkeson, 1993; Wiering, 1999), cerebellar model articulation controllers (CMACs – Albus, 1975), and prioritized sweeping (Moore and Atkeson, 1993; Wiering and Schmidhuber, 1998). It is capable of finding soccer strategies with comparable or even better performances than PIPE.

5.4 Conclusion

This chapter described how to apply PIPE. First a step-by-step description on how to setup PIPE for an application was given.

Then PIPE was applied to three basic problems: function regression, 6-

bit parity, and 3+8-bit multiplexer. The experiments revealed that (1) PIPE was able to find suitable solutions to each of the problems, (2) PIPE's solution programs differ much from programs created by human programmers, (3) PIPE is rather easy to setup as it can use "standard" instruction sets and "standard" parameter settings to solve various problems, (4) PIPE can be used successfully with different instruction sets and deal with program trees of various arities.

Finally, PIPE was applied in a more complex soccer case study, where the objective was to learn complete soccer team strategies in a multiagent environment. PIPE's performance was compared to that of TD-Q learning with linear neural networks (TD-Q). Also a coevolutionary version of PIPE (CO-PIPE) was applied. PIPE and CO-PIPE compared favourably to TD-Q. Both PIPE variants found solutions that outperformed a strong engineered single-agent strategy, while TD-Q did not.

Chapter 6

Evolving Structured Programs

In previous chapters we have shown how PIPE evolves programs and how it can be applied to a wide variety of problems. Here we will focus on the evolution of *structured programs*. Structured programs are of interest as they allow for restricting the search space and thus speeding up evolution.

6.1 Introduction and Previous Work

To evolve structured programs we develop Hierarchical Probabilistic Incremental Program Evolution (H-PIPE – Sałustowicz and Schmidhuber, 1998), an hierarchical extension of Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a). H-PIPE uses "hierarchical instructions" (HIs) and "skip nodes" (SNs). HIs are limited to top-level, structuring program parts. They induce structure by combining lower-level program parts. SNs are inspired by biology's introns (non-coding segments). They function as gates that allow for keeping program parts dormant without losing them in the course of evolution. Thus SNs can be used to switch program parts on and off. In combination with HIs they enable H-PIPE to substitute program parts by superior partial solutions discovered at later evolutionary stages.

6.1.1 Structure

Early genetic programming (GP) work (Dickmanns et al., 1987) as well as Adaptive Levin Search (Schmidhuber, 1997; Schmidhuber et al., 1997b) allow for powerful programs with arbitrary loops etc. Sometimes, however, it is beneficial to introduce inductive bias by appropriately constraining the search space of possible programs. Except for programs evolved by tree-based GP (Cramer, 1985; Koza, 1992), however, not much work has been done on evolution of programs with significant *structural* constraints. There are two such GP variants.

The first reuses program parts, usually in a way less general than that achievable through arbitrary jumps. Typically subprograms are generated and/or extracted from evolved programs; they may then be called in a usually non-recursive fashion from different positions in the code. Examples are: "automatically defined functions" and encapsulation (Koza, 1992), module acquisition (Angeline and Pollack, 1992), adaptive representations through learning (Rosca and Ballard, 1996), automatically defined macros (Spector, 1996). Other approaches do not generate or extract subprograms but restrict GP's recombination operator such that it cannot destroy certain program parts to be reused in the future (e.g., Langdon, 1995; Pringle, 1995; Zannoni and Reynolds, 1997).

The second variant uses grammars to induce structure, constrain the search space, and provide initial bias to speed up evolution. Examples are context-free (Whigham, 1995; Gruau, 1996) or logic grammars (Wong and Leung, 1996).

Programs with hierarchical instructions (HIs) are special cases of programs constrained by context-free grammars: Higher-level instructions can be used to combine program parts made out of lower-level instructions, thus inducing structure.

6.1.2 Non-Coding Program Parts

Non-coding program parts ("introns") are those that do not affect the results the program calculates. E.g., in $f(x) = x * 1$, the "$*1$" part is non-coding. Most previous work on non-coding program parts focuses on genetic program synthesis (Blickle and Thiele, 1994; McPhee and Miller, 1995; Nordin et al., 1996; Haynes, 1996; Wineberg and Oppacher, 1996). Usually non-coding program parts evolve or can be inserted to protect coding program parts (parts that *do* affect results calculated by the program) from destructive genetic recombination operators (Blickle and Thiele, 1994; McPhee and Miller, 1995; Nordin et al., 1996; Haynes, 1996). Blickle and Thiele (1994), as well as McPhee and Miller (1995), however, point out that large blocks of non-coding segments in tree-based GP programs cause very slow convergence and difficulties in escaping from local minima. Haynes (1996), on the other hand, shows that artificial removal of non-coding segments from those programs leads to premature convergence. Nordin, Francone, and Banzhaf (1996) investigate the role of non-coding segments in a GP approach based on variable-length strings. They note that non-coding segments may play an important role in finding good solutions and speeding up convergence. Wineberg and Oppacher (1996) use *fixed*-length strings and find that non-coding segments reduce the search space and speed up evolution.

General Observation

The literature above suggests: in tree-based GP programs with little structure the effect of non-coding segments is twofold. On the one hand they seem necessary to protect blocks of coding segments, on the other hand they can hinder discovery of acceptable solutions. In the case of *structured* programs, however, non-coding program parts can both speed up convergence *and* aid in finding good solutions. Loosely speaking, the more structured the programs (e.g., the greater the restrictions on the coding strings), the higher the potential significance of non-coding segments. Our own experiments with skip nodes will add more empirical evidence in this direction.

Skip Nodes

Much like certain "jump" instructions, skip nodes (SNs) are instructions that allow for skipping program parts. In the context of tree-based functional programs, SNs are functions with n arguments, where n denotes the maximal number of arguments of functions in S. SNs return exactly one of their arguments and ignore the others, which thus represent non-coding program parts if $n > 1$. We will demonstrate the benefits of SNs in structuring parts of H-PIPE programs.

6.1.3 Results Overview

In our experiments H-PIPE outperforms PIPE, and SNs facilitate synthesis of certain structured programs but not unstructured ones. We conclude that introns can be particularly useful in the presence of structural bias.

6.1.4 Outline

Section 6.2 describes the H-PIPE approach. Section 6.3 compares the use of HIs and SNs to standard PIPE on function regression and 6-bit parity. Section 6.4 concludes this chapter.

6.2 Hierarchical Probabilistic Incremental Program Evolution

First we will describe how to extend PIPE to accommodate for hierarchical instructions (HIs). Then we will show how skip nodes (SNs) can be integrated into PIPE and H-PIPE.

6.2.1 Hierarchical Instructions

In this section we will first extend PIPE's elementary data structures (programs and probability distribution) to accommodate for HIs. Then we will describe H-PIPE's elementary procedures (program generation and "tree shaping") and update rules.

Program Instructions

H-PIPE's programs are composed from z instructions in the instruction set $S = \{I_1, I_2, \ldots, I_z\}$. Each node of the code tree contains an instruction I and can have several son nodes whose instructions are viewed as arguments of I. To allow for HIs we partition S into $m+1$ disjoint, non-empty instruction sets $S^0, S^1, \ldots, S^m$, and ensure that all "terminal instructions" — instructions with zero arguments — are in S^0. Hierarchical order arises as follows: Each argument of an instruction in S^v is in S^v or in the "lower level" set S^{v-1}. At least one argument must be in S^{v-1}, except when $v = 0$. To allow for enforcing descents in the instruction set hierarchy we add pointers to lower level instructions $\downarrow_i$ to all instruction sets S^v, $\forall v : 0 < v \leq m$, where $0 < i \leq l(v)$ is the argument index of an instruction $I \in S^v$ with $l(v)$ arguments from S^{v-1}. Although pointers to lower level instructions take a single argument and return it, they are treated as terminal symbols. Thus each instruction set S^v $(0 < v \leq m)$ can be written as $F^v \cup T^v$, where $F^v = \{f_1^v, f_2^v, \ldots, f_{k(v)}^v\}$ is a function set with $k(v)$ functions and $T^v = \{\downarrow_1, \downarrow_2, \ldots, \downarrow_{l(v)}\}$ is a terminal set containing $l(v)$ pointers to lower level instructions. We also have $S^0 = F^0 \cup T^0$, where $F^0 = \{f_1^0, f_2^0, \ldots, f_{k(0)}^0\}$ is a function set with $k(0)$ functions and $T^0 = T$ is a terminal set containing all terminals of S ($l(0) = l$).

For instance, to structure the function approximation task from Section 4.1.1 as a linear combination of non-linear parts we split the instruction set $S = \{+, -, *, \%, sin, cos, exp, rlog, x, R\}$ into $S^0 = \{*, \%, sin, cos, exp, rlog, x, R\}$ and $S^1 = \{+, -\}$. We then add a $\downarrow_1$ instruction to S^1 and obtain $S^1 = \{+, -, \downarrow_1\}$. Function and terminal sets for the lower and upper level then become $F^0 = \{*, \%, sin, cos, exp, rlog\}, T^0 = \{x, R\}$ and $F^1 = \{+, -\}, T^1 = \{\downarrow_1\}$, respectively. Figure 6.1 shows an example program.

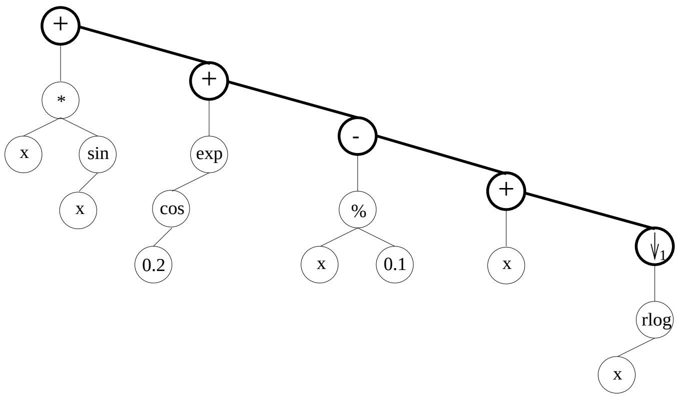

Figure 6.1: *f(x)=x*sin(x)+exp(cos(0.2))+x%0.1-(x+rlog(x)).* Exemplary program tree for function approximation constrained to a linear combination of non-linear parts. Top-level structuring instructions from S^1 appear in boldface.

Program Representation

With HIs the arity $n(v)$ of a program tree may vary depending on the hierarchical level v. On each level v, $n(v)$ is the maximal number of function arguments required by functions in S^v. For instance, in the function approximation example above, if we add to S^0 a three argument function, e.g $**$, where $**(a_1, a_2, a_3) = a_1 * a_2 * a_3$, then the lower-level part of the program tree will be 3-ary while the top-level part will remain 2-ary, as depicted in Figure 6.2.

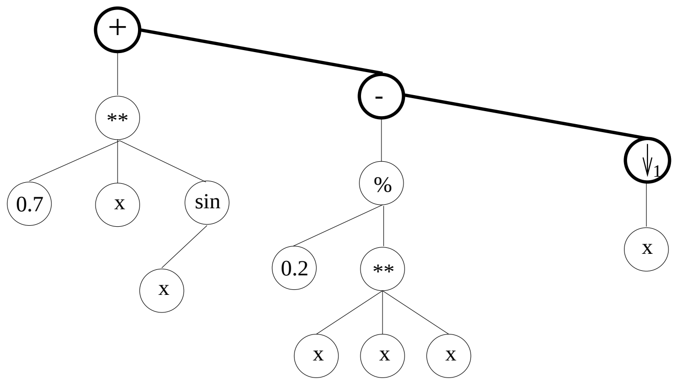

Figure 6.2: *f(x)=0.7*x*sin(x)+0.2%(x*x*x)-x.* Exemplary program tree for function approximation, with different level-dependent arities. Top-level program parts are 2-ary. Lower level program parts are 3-ary.

Hierarchical Probabilistic Prototype Tree

The probability distribution is stored in an *hierarchical probabilistic prototype tree* (*H-PPT*). The *H-PPT* is generally a complete $n(v)$-ary tree, where $n(v)$ is *H-PPT*'s arity at hierarchical level v.

Nodes

Each *H-PPT* node $N_j(v)$ contains a variable probability vector $\vec{P}_j(v)$, where list $j = (j_m, j_{m-1}, \ldots, j_1, j_0)$ with $m+1$ variable elements describes a unique absolute position in *H-PPT*. Each node $N_j(0)$ contains in addition a random constant $R_j(0)$. The probability vectors $\vec{P}_j(v)$, $\forall v : 0 \leq v \leq m$ have $k(v) + l(v)$ components. Each component $P_j(v, I)$, $\forall v : 0 \leq v \leq m$ denotes the probability of choosing instruction $I \in S^v$ at $N_j(v)$. We maintain $\sum_{I \in S^v} P_j(v, I) = 1$.

Initialization

Each *H-PPT* node $N_j(v)$ requires an initial probability $P_j(v, I)$ for each instruction $I \in S^v$. Furthermore, each bottom level ($v = 0$) node $N_j(0)$ requires an initial random constant $R_j(0)$. A value for $R_j(0)$ is randomly taken from a predefined, problem-dependent set of constants (compare with Section 4.2.2). To initialize instruction probabilities we use for each hierarchical level v a constant probability P_{T^v} for selecting an instruction from T^v

and $(1 - P_{T^v})$ for selecting an instruction from F^v. $\vec{P}_j(v)$ is then initialized as follows:

$$P_j(v, I) := \frac{P_{T^v}}{l(v)}, \forall I : I \in T^v \quad \text{and} \quad P_j(v, I) := \frac{1 - P_{T^v}}{k(v)}, \forall I : I \in F^v$$

Program Generation

Extracting programs from *H-PPT* is analogous to extracting programs from *PPT* (compare Section 4.2.3), except that instructions are selected from the appropriate S^v, depending on the hierarchical level. To generate a program PROG from *H-PPT*, an instruction $I \in S^v$ is selected with probability $P_j(v, I)$ for each accessed node $N_j(v)$ of *H-PPT*. This instruction is denoted by I_j. Nodes are accessed in a depth-first way, starting at the root node $N_{j^*}(m)$, where $j^* = (1, 0, \ldots, 0)$ and traversing *H-PPT* from left to right. Once $I_j \in T^v, \forall v : 0 < v \leq m$ has been selected the following instruction in this branch must be from S^{v-1}. Figure 6.3 shows a *H-PPT* and a corresponding possible program.

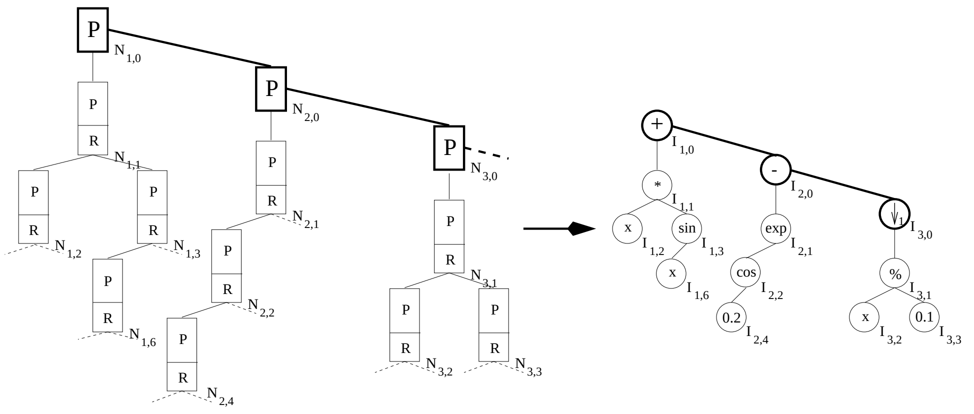

Figure 6.3: A *H-PPT* (left) and a corresponding possible program (right). The structuring parts of the program are highlighted.

Tree Shaping

H-PPT shaping is done just like *PPT* shaping (see Section 4.2.4).

Learning

H-PIPE's update rules are analogous to PIPE's. The only difference is the more sophisticated indexing method due to *H-PPT*'s hierarchical structure.

6.2.2 Skip Nodes

Skip nodes are inspired by biology's introns. They are functions that serve to switch code parts on and off. We will first define SNs for PIPE, then for H-PIPE. Finally we describe the necessary modifications of PIPE's and H-PIPE's update rules.

SNs for PIPE

Let n denote the maximal arity of the PTT (the maximal number of arguments of functions that are not SNs). There are at most n SNs. The i-th is denoted $\rightarrow_i$. It is a function with n arguments and returns the i-th. Its interpretation is: evaluate the i-th argument but ignore the others.

SNs are elements of the function set F. For instance, if we add SNs to the instruction set of the function approximation example from Section 4.1.1 we obtain: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_1, \rightarrow_2\}$ and $T = \{x, R\}$. Figure 6.4 shows a PIPE program with SNs. The dashed parts of the program can

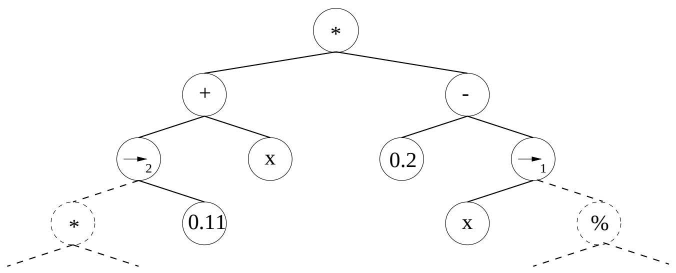

Figure 6.4: A PIPE program with SNs for function approximation: $f(x) = (0.11 + x) * (0.2 - x)$. The dashed parts of the program are non-coding segments.

be viewed as non-coding segments. Note that they need not even be created during program generation and are therefore computationally cheap.

SNs for H-PIPE

Let $h(v)$ denote the maximal number of arguments in S^v of non-SN functions in S^v.

At level v ($0 < v \leq m$) there are at most $h(v)$ SNs. The i-th is denoted $\rightarrow_i^v$. It is a function with $h(v)$ arguments in S^v and returns the i-th. Its interpretation is: evaluate the i-th argument but ignore the others. Arguments in S^{v-1} cannot be accessed by SNs. They are accessed by HIs. There are no SNs in S^0.

SNs are elements of the function set F^v. For instance, if we add SNs to the instruction set of the function approximation example from Section 6.2.1 we obtain: $F^0 = \{*, \%, sin, cos, exp, rlog\}, T^0 = \{x, R\}$ and $F^1 = \{+, -, \rightarrow_1^1\}, T^1 = \{\downarrow_1\}$. Figure 6.5 shows an H-PIPE program with SNs.

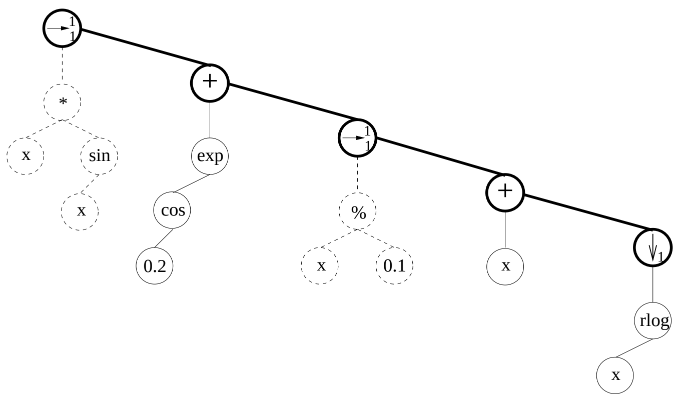

Figure 6.5: An H-PIPE program with SNs for function approximation: $f(x) = exp(cos(0.2)) + x + rlog(x)$. The dashed parts of the program are non-coding segments.

Changes to PIPE's and H-PIPE's Update Rules

Learning Update Modification. Parts of *PPT* or *H-PPT* corresponding to non-coding segments are *not* updated. They are *not* taken into account when (1) calculating the probability $P(\text{Prog}_b)$ of the best-of-generation program Prog_b (see Equation 4.1), nor when (2) calculating the probability $P(\text{Prog}^{el})$ of the elitist program Prog^{el} (see Equation 4.2), nor when (3) performing the learning update on the *H-PPT* (see Assignment 4.3).

Mutation. *PPT* or *H-PPT* parts that correspond to non-coding segments are also *not* accounted for during mutation. In Equation 4.4 $|\text{P}{\small ROG}_b|$ denotes the number of nodes in $\text{P}{\small ROG}_b$. With SNs, however, $|\text{P}{\small ROG}_b|$ denotes the number of nodes in program $\text{P}{\small ROG}_b$ *without* the non-coding segments created by SNs.

6.3 Experiments

To evaluate the impact of HIs and SNs we cross-compare: (1) PIPE, (2) H-PIPE without SNs (H-PIPE-NO-SN), (3) PIPE with SNs (PIPE-SN), (4) and H-PIPE (PIPE with HIs and SNs in the structuring program parts). To illustrate the significance of *appropriate* initial bias we also test H-PIPE with different structuring instructions (H-PIPE-DIFF). We consider the nontrivial continuous function regression problem from Section 5.2.1 and the 6-bit parity problem from Section 5.2.2. For each combination of learning algorithm and problem we conduct 50-200 independent runs to obtain statistically significant results.

6.3.1 Function Regression

We use the same function regression problem as in Section 5.2.1. Training and test environment, fitness function and output interface remain the same.

Instruction Set

We use the following instruction sets: (1) PIPE: $F = \{+, -, *, \%, sin, cos, exp, rlog\}$, $T = \{x, R\}$; (2) H-PIPE-NO-SN: $F^1 = \{+, -\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{*, \%, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$; (3) PIPE-SN: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_1, \rightarrow_2\}$, $T = \{x, R\}$; (4) H-PIPE: $F^1 = \{+, -, \rightarrow_1^1\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{*, \%, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$; (5) H-PIPE-DIFF: $F^1 = \{*, \%, \rightarrow_1^1\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x, R\}$. R is always picked uniformly random from the interval [0;1).

Parameter Setup

We time-constrain all runs to PE = 100,000 and use the following parameter setting: P_T=P_{T^0}=P_{T^1}=0.8, ε = 0.000001, P_{el}=0.01, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999, $FIT_s = 0$.

Results

Figure 6.6 summarizes the most interesting results in form of cumulative histograms. Note the different x-axis scaling for H-PIPE-DIFF. We plot

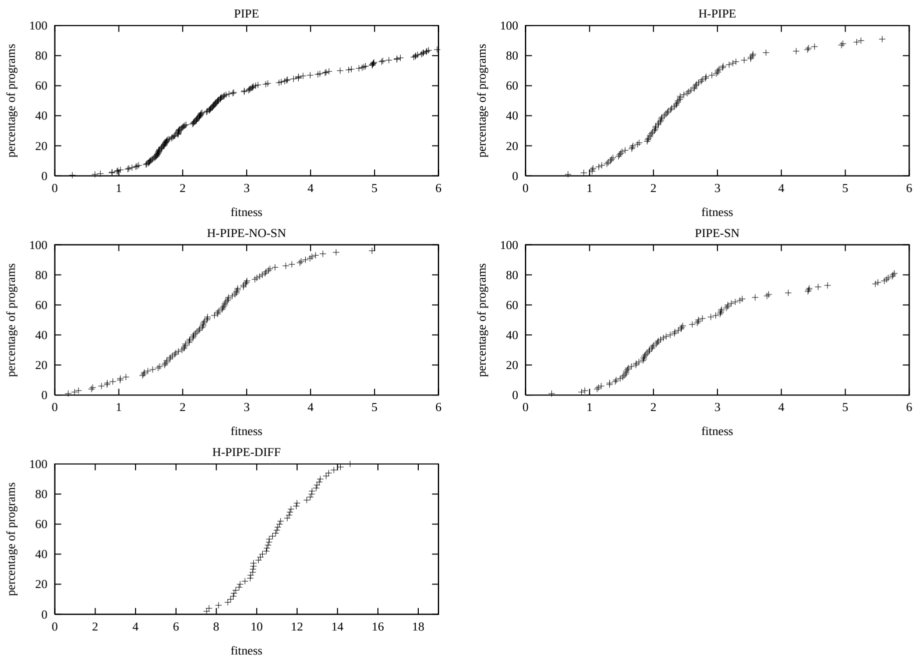

Figure 6.6: Results for the regression problem.

performance u against percentage of programs with $FIT(\text{ PROG}) \leq u$. Each point indicates the number of programs with $FIT(\text{PROG})$ equal to or better than its x-axis value: algorithms with better performance have more points with smaller x-values.

PIPE vs. H-PIPE. H-PIPE outperforms PIPE. H-PIPE's fitness in the median run is $FIT_{med} = 2.39$, slightly better than PIPE's with $FIT_{med} = 2.55$. In 82% of all runs H-PIPE finds programs with fitness below 4, while only 67% of all PIPE runs accomplish this. On the other hand, the worst 3% of all H-PIPE runs (not shown) resulted in programs worse than the best found by all PIPE runs. The median of H-PIPE's program size ($Node_{med} = 92$ nodes) is significantly smaller than PIPE's ($Node_{med} = 157$).

How much of the performance improvement can be attributed to HIs, how much to SNs? To study this question we now compare PIPE and H-PIPE to PIPE with SNs (PIPE-SN) and H-PIPE without SNs (H-PIPE-NO-SN).

PIPE and H-PIPE vs. PIPE-SN. PIPE-SN performs much like PIPE, and worse than H-PIPE. PIPE-SN's $FIT_{med} = 2.70$ is slightly higher than PIPE's ($FIT_{med} = 2.55$). Like PIPE, in 67% of all runs PIPE-SN found programs with fitness below 4. Its worst programs (not shown) are slightly better than the worst program among the best of the individual PIPE runs. PIPE-SN's programs ($Node_{med} = 117$) tend to be smaller than PIPE's ($Node_{med} = 157$), but larger than H-PIPE's ($Node_{med} = 92$).

We observe that SNs in unstructured PIPE programs are neither harmful nor beneficial.

PIPE and H-PIPE vs. H-PIPE-NO-SN. H-PIPE-NO-SN is the best competitor, slightly better than H-PIPE, much better than PIPE. H-PIPE-NO-SN's $FIT_{med} = 2.38$ is roughly as good as H-PIPE's $FIT_{med} = 2.39$. In 91% of all runs , however, H-PIPE-NO-SN found programs with fitness below 4, compared to H-PIPE's 82% and PIPE's 67%. Furthermore, unlike with H-PIPE and PIPE, no program found by H-PIPE-NO-SN has fitness above 7.39. The median size of H-PIPE-NO-SN programs, $Node_{med} = 96$, is roughly the same as H-PIPE's ($Node_{med} = 92$) and significantly smaller than PIPE's ($Node_{med} = 157$).

For this particular problem we observe that HIs by themselves increase PIPE's performance more than SNs. Later (in Section 6.3.2) we will see, however, that indeed both HIs *and* SNs are needed to solve certain tasks more efficiently. But first we will illustrate the importance of choosing the right HIs.

PIPE and H-PIPE vs. H-PIPE-DIFF. H-PIPE-DIFF performs significantly worse than H-PIPE and PIPE. The fitness of the best program found by H-PIPE-DIFF in 50 independent runs is only 7.52. H-PIPE-DIFF's median fitness is $FIT_{med} = 10.62$. Compare H-PIPE's and PIPE's, which are 2.39 and 2.55, respectively.

This demonstrates, not unexpectedly, that appropriate initial bias due to "good" HIs is crucial to H-PIPE's success.

Conclusion

HIs can increase PIPE's performance significantly. They need to be selected carefully, however. SNs do not contribute much to solving the function regression task. In case of PIPE they reduce program size without affecting solution quality. In case of H-PIPE they have a slightly detrimental effect on overall performance.

The next experiment, however, will show that for some tasks only the combination of HIs *and* SNs leads to significant performance improvement.

6.3.2 6-Bit Parity

We use the same 6-bit problem as in Section 5.2.2. Training and test environment, fitness function and output interface remain the same.

Instruction Set

We use the following instruction sets: (1) PIPE: $F = \{+, -, *, \%, sin, cos, exp, rlog\}$, $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (2) H-PIPE-NO-SN: $F^1 = \{*, \%\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (3) PIPE-SN: $F = \{+, -, *, \%, sin, cos, exp, rlog, \rightarrow_1, \rightarrow_2\}$, $T = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (4) H-PIPE: $F^1 = \{*, \%, \rightarrow_1^1\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{+, -, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$; (5) H-PIPE-DIFF: $F^1 = \{+, -, \rightarrow_1^1\}$, $T^1 = \{\downarrow_1\}$, $F^0 = \{*, \%, sin, cos, exp, rlog\}$, $T^0 = \{x_0, x_1, x_2, x_3, x_4, x_5, R\}$. R is always picked uniformly random from the interval [0;1).

Parameter Setup

We time-constrain all runs to PE = 500,000 and use the following parameter settings: P_T=P_{T^0}=P_{T^1}=0.6, ε = 0.000001, P_{el}=0.01, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.999999, FIT_s = 0. Note that, except for P_T, P_{T^0}, and P_{T^1}, all parameters are set to the same values as for the function regression task (see Section 6.3.1). Most of PIPE's and H-PIPE's parameters seem robust with respect to changing tasks.

Results

Table 6.1 summarizes all results. The first column displays for each algorithm the percentage of independent runs leading to perfect solutions within the given time frame (PE). The next three columns show the numbers of program evaluations necessary to find perfect solutions in the shortest, median, and longest run, respectively. The final three columns list the minimal, median, and maximal program sizes embodying perfect solutions.

Comparison

H-PIPE performs best. It solves the task more often and significantly faster (with less program evaluations) than PIPE, PIPE with SNs, and H-PIPE

Table 6.1: Summary of 6-bit parity results. Best values are in boldface.

	6-bit parity		
		Program Evaluations	Nodes
Algorithm	solved	min– med –max	min–med–max
H-PIPE	**94** %	5,700–**37,460**–397,000	23– 61 –96
PIPE	79 %	3,520– 79,950 –497,220	24– 64 –137
PIPE-SN	76 %	1,676– 73,720 –487,930	25– 58 –110
H-PIPE-NO-SN	66 %	3,720–166,740–468,950	21– **49** –85
H-PIPE-DIFF	28 %	38,300–216,570–457,330	24– 61 –94

without SNs. PIPE and PIPE-SN have roughly the same performance. PIPE-SN finds slightly fewer solutions, but is faster than PIPE in the median run. The median size of its solutions is also slightly smaller than PIPE's. Although its solution size is smallest in the median run, H-PIPE-NO-SN performs significantly worse than PIPE and PIPE-SN. It finds fewer solutions and requires more than twice as many program evaluations (in the median run). H-PIPE-DIFF with wrong initial bias is worst of all. It needs more than five times as many program evaluations as H-PIPE to find roughly three times fewer solutions.

Conclusion

With this particular task H-PIPE outperforms PIPE. Neither SNs by themselves nor HIs by themselves are able to improve PIPE's performance. In absence of structure SNs' effects are neither harmful nor beneficial, while HIs by themselves decrease PIPE's performance. The combination of both HIs (embodying the proper initial bias) and SNs in H-PIPE, however, allows for significant improvement.

6.4 Conclusion

H-PIPE, a novel method for synthesizing *structured* programs, uses hierarchical instructions (HIs) to structure programs and skip nodes (SNs) to facilitate their synthesis. HIs combine program parts, while SNs allow for introns (non-coding segments). In our experiments, SNs by themselves were useless for improving performance. Sometimes HIs by themselves worked extremely well, but not always. Then, however, SNs were crucial to achieve dramatic improvement.

Our review of previous work on non-coding segments suggests that non-coding segments seem to require *structured* code to unfold their benefits. Our own results add further empirical evidence in this vein.

Chapter 7

Memory

In previous chapters we have shown how some tasks can be solved by learning a simple mapping from inputs to outputs. There are, however, tasks that cannot be solved like that. This is because inputs may be ambiguous. A particular input may demand different output responses depending on the temporal context. Disambiguating inputs requires some sort of memory.

In this chapter we will present how to augment PIPE with memory. We will then investigate PIPE's behavior on tasks with a temporal context. We will apply PIPE to learning in partially observable environments and demonstrate that PIPE is able to find solutions to reinforcement learning problems. Then we will focus on the "long time lag challenge", where the task is to isolate and store relevant input information over long periods of time before relating it to an appropriate output. We will benchmark PIPE against "Long Short Term Memory" (LSTM), the to our knowledge currently most successful recurrent neural network approach to this kind of problems.

7.1 Memory Types

This section presents two types of memory that can be used by PIPE: recurrent output links, and memorizing cells.

7.1.1 Recurrent Output Links

To implement recurrent output links (ROLs) an instruction "o" is added to the terminal set. At each time step t, o contains the output of the program at time step $t-1$. For $t = 0$, o is set to 0. Figure 7.1 shows an example program containing a ROL.

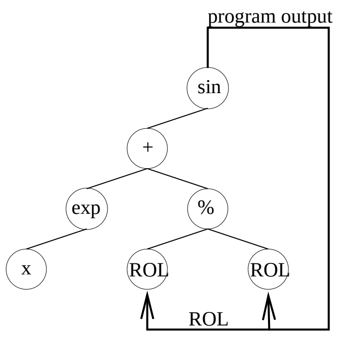

Figure 7.1: Example program with a recurrent output link (ROL).

7.1.2 Memorizing Cells

There are two kinds of memorizing cells: output cells (OCs) and memory cells (MCs). Their data structures are identical, their applications different. Figure 7.2 shows an example. OCs store the output of a program. The return value of the program is then ignored. MCs are only used as internal memory. OCs and MCs can be used together. There are n_{OC} OCs and n_{MC} MCs, where n_{OC} and n_{MC} are positive integer constants. OCs and MCs can be accessed and modified by programs during runtime. At any given time, OC_j and MC_i denote the current real-valued contents of the j-th and i-th output and memory cells, respectively ($j \in \{0..n_{OC}-1\}$, $i \in \{0..n_{MC}-1\}$). All OC_j and MC_i are initialized with 0. OCs and MCs are accessed by write and read functions that are added to the instruction set (see Section 7.2).

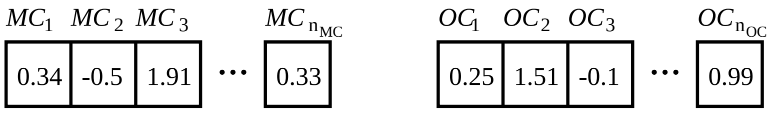

Figure 7.2: Array of n_{MC} and n_{OC} real–valued memory and output cells (MCs and OCs) respectively.

7.2 Memory Access Strategies

This section presents two kinds of memory access strategies: direct and indexed.

7.2.1 Direct Memory Access

With direct memory access (DMA) each OC_j and MC_i is associated with a distinct function for setting and reading it. Functions $set\_O_j(arg_1)$ ($set\_M_i(arg_1)$) set the j-th (i-th) output (memory) cell to arg_1 and return arg_1. Terminal instructions get_O_j (get_M_i) return the contents of OC_j (MC_i). The disadvantage of DMA is that the number of instructions in S (the instruction set) grows linearly with the number of output/memory cells. It is applicable only when few memory cells are used.

7.2.2 Indexed Memory Access

Indexed memory access (IMA) overcomes DMA's problem. With IMA only two functions need to be added to set and read arbitrary many output and memory cells. Function $set\_O(arg_1, arg_2)$ ($set\_M(arg_1, arg_2)$) sets $OC_{(|round(arg_1)|\ mod\ n_{OC})}$ ($MC_{(|round(arg_1)|\ mod\ n_{MC})}$) $:= arg_2$ and returns arg_2. Function $get\_O(arg_1)$ ($get\_M(arg_1)$) returns $OC_{(|round(arg_1)|\ mod\ n_{OC})}$ ($MC_{(|round(arg_1)|\ mod\ n_{MC})}$) (see, e.g., Teller, 1994).

7.3 Multiple Outputs

Section 4.4 described how *multiple programs* (MPs) can be used to accommodate for vector–valued outputs. Output cells (OCs) can also be used.

When OCs are used, their contents are treated as the output of a program, while the program's return value is ignored. As already hinted at in Section 7.1.2 a program can have multiple output cells ($n_{OC} > 1$) and in this way accommodate for multiple outputs.

7.4 Partially Observable Environments

In partially observable environments (POEs), a particular observation may demand different action responses depending on the temporal context. Disambiguating observations requires some sort of short-term memory (e.g., Schmidhuber, 1991; Littman, 1994a; Kaelbling, Littman, and Cassandra, 1995). POE tasks are generally considered difficult because of their particularly nasty temporal credit assignment problem: It is usually hard to figure out which observations are relevant and how they should affect short-term memory contents.

7.4.1 Short-Term Memorizing POE Algorithms

Apart from recent nontraditional methods (e.g., Zhao and Schmidhuber, 1996; Schmidhuber et al., 1997a,b) there are two classes of POE algorithms. Class I extends standard reinforcement learning (RL) algorithms based on adaptive evaluation functions (EFs) (Watkins, 1989; Bertsekas and Tsitsiklis, 1996). Usually, on-line variants of dynamic programming and some kind of function approximator with a short-term memory mechanism are combined to construct EFs mapping input/action histories to an expected discounted future reward. The EFs are exploited in an on-line fashion to learn rewarding action sequences (Whitehead and Ballard, 1990; Schmidhuber, 1991; Chrisman, 1992; Lin, 1993; Cliff and Ross, 1994; Ring, 1995; McCallum, 1996; Wiering and Schmidhuber, 1996a).

Methods from class II do not require EFs. Their policy space consists of complete algorithms allowing for temporary memory, and they search policy space directly. Members of this class are Levin Search (Levin, 1973, 1984; Solomonoff, 1986; Li and Vitányi, 1993; Schmidhuber, 1997a), Adaptive Levin Search (ALS) (Wiering and Schmidhuber, 1996b), GP with memory cells (e.g., Teller, 1994), and PIPE with memory. All those approaches generate and evaluate solution candidates in an off-line fashion. ALS, GP, and PIPE also update the generator on the basis of the evaluation results. Other, yet untried, class II methods include Simulated Annealing and Stochastic Iterated Hill Climbing for Program Discovery (O'Reilly, 1995) with memory.

7.4.2 Maze Tasks

PIPE with memory cells can solve the POE tasks shown in Figure 7.3. S denotes the start position and G the goal. The task is to find the shortest path from S to G. The gray fields result in ambiguous observations: For each gray field there is at least one other field on the shortest path from S to G at which the agent will make the same observation but will have to execute a different action. Memory of prior events is required to disambiguate those observations.

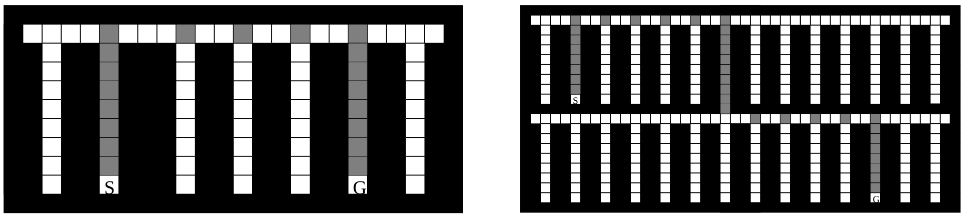

Figure 7.3: Two mazes with 78 fields (left) and 196 fields (right). The agent sees only whether its four adjacent states are blocked or free. This means there are $2^4 = 16$ possible observations, although only 6 (7) of them can occur in the left (right) maze. Fields producing ambiguous observations on the shortest path from S to G are colored gray; 19 (34) of the 29 (56) fields in the left (right) maze have this property.

Agent Inputs. The agent input is a vector $\vec{X} = (N, S, W, E, MC_0, MC_1, MC_2)$, where N (S, W, E) is one of four observation components and takes on value 1 if the field to the north (south, west, east) of the agent is blocked and 0 otherwise. The maze in Figure 7.3-left (-right) allows for only 6 (7) distinct observations. Fewer than $n_{MC} = 3$ memory cells did not lead to satisfactory results for both mazes.

Agent Actions. On any given field, the agent can execute one of four actions. Action GO_N (GO_S, GO_W, GO_E) moves the agent one field to the north (south, west, east) of its current position if this field is not blocked and has no effect otherwise.

Training and Test Environment

Initially an agent *AGENT* is placed onto the start field S. *AGENT* is controlled by a main program PROGRAM as follows:

```
executed_actions := 0
REPEAT
    Action := PROGRAM($\vec{X}$)
    EXECUTE Action
    executed_actions := executed_actions + 1
UNTIL (AGENT found goal G OR executed_actions = MAX_actions)
```

Since program runtimes are unknown, a time limit of *MAX_actions* executed actions is introduced. If no agent of the current population finds the goal within $MAX\_actions$ executed actions, then the generation is "ignored" by not learning from it, and not mutating the probability distributions.

Parallel Evaluation Saves Time. The current PIPE variant does not exploit the fact that programs in a generation may have many different fitness values. Instead, it considers just two fitness categories: "best" and "worse than best" – the adjustment of prototype tree probabilities depends on the best program only. Let us assume there are PS independent agents situated in PS equal mazes. Since program runtimes may vary wildly, we can save a lot of time by running all programs of each generation in parallel (or by interleaving them on a serial computer) until the first agent reaches the goal. Thus, we do not waste time on finishing executions of programs worse than the best.

Fitness Function

Fitness is defined by the number of actions the agent executes to reach goal G when starting from S.

Instruction Set

The function set is $F = \{+, -, *, \%, sin, cos, exp, rlog, set\_M, get\_M\}$. The terminal set is $T = \{N, S, W, E, R\}$. R denotes the generic random constant in [0;1).

Output Interface

PIPE synthesizes programs which, given $AGENT$'s input vector $\vec{X}$, select actions from $ASET = \{\text{Go\_N}, \text{Go\_S}, \text{Go\_W}, \text{Go\_E}\}$.

Action Selection. The same action selection scheme is used as in the soccer case study (see Section 5.3.4). Action selection depends on five variables: $g \in I\!R$, $A_i \in I\!R$, $\forall i \in ASET$. Action $i \in ASET$ is selected with probability P_{A_i} according to the Boltzmann-Gibbs distribution at temperature $\frac{1}{g}$:

$$P_{A_i} := \frac{e^{A_i \cdot g}}{\sum_{\forall j \in ASET} e^{A_j \cdot g}} \qquad \forall i \in ASET \tag{7.1}$$

All A_i and g are calculated by programs to be found by PIPE.

Programs. A main program PROGRAM consists of a program PROG^g, which computes the "greediness" parameter g, and four "action programs"

$\textsc{Prog}^i$ ($i \in ASET$). These programs are generated according to five distinct probabilistic prototype trees; they calculate

$$\begin{aligned}
A_{\text{Go\_N}} &:= \textsc{Prog}^{\text{Go\_N}}(\vec{X}) \\
A_{\text{Go\_S}} &:= \textsc{Prog}^{\text{Go\_S}}(\vec{X}) \\
A_{\text{Go\_W}} &:= \textsc{Prog}^{\text{Go\_W}}(\vec{X}) \\
A_{\text{Go\_E}} &:= \textsc{Prog}^{\text{Go\_E}}(\vec{X}) \\
g &:= |\textsc{Prog}^{g}(\vec{X})|
\end{aligned}$$

where $|\textsc{Prog}(x)|$ denotes the absolute value of $\textsc{Prog}(x)$. Given $\vec{X}$, $\textsc{Program}(\vec{X})$ finally returns an action $i \in ASET$ randomly selected according to Assignment 7.1. The evaluation order of all programs $\textsc{Prog}^i$ and $\textsc{Prog}^g$ is fixed. This is important because those programs access and modify the same memory cells. Calculating g enables PIPE to produce more or less probabilistic/deterministic programs (possibly depending on the input).

Parameter Setup

The following parameters worked well for both mazes in Figure 7.3: PS=100, PE=10,000,000, P_{el}=0, $\varepsilon = 1$, lr=0.2, P_M=0.1, mr=0.2, T_R=0.3, T_P= 0.999999, and $MAX\_actions = 10,000$. For experiments with the small maze (Figure 7.3, left), P_T was set to 0.9999 and FIT_s to 29. For the large maze (Figure 7.3, right), P_T was set to 0.999 and FIT_s to 56.

Since fitness evaluations are extremely noisy, P_{el} was set to 0. In the forthcoming experiments, PIPE worked better with smaller populations sizes PS and higher terminal probabilities P_T. A high terminal probability forces PIPE to search for small programs first. Smaller population sizes allow for more generations per time interval. Too small populations, however, slow PIPE down because it can take less advantage of time-saving parallel evaluations.

Results for Small Maze

11 independent runs were conducted. All runs found the optimal solution of 29 steps. The earliest (latest) discovery of an optimal solution took 663 (12,837) generations, or 66,300 (1,283,700) program evaluations. The median run took 2134 generations, or 213,400 program evaluations. In the median (fastest/slowest) run, 31,544,100 (7,496,500/ 189,601,700) agent actions were executed to find the shortest path. Recall that during each generation *only one* program is run to completion, because the evaluation of all programs is

stopped as soon as the first finds a solution. The program shown in Table

Table 7.1: A program embodying a partly stochastic, partly highly deterministic policy for solving the small maze (see Figure 7.3, left).

$\text{Prog}^{\text{Go\_N}}$:	`sin((((exp(N)+sin(0.286691))+0.970697)+get_M(get_M(cos(exp(set_M(((W*get_M((E*S)))%cos(get_M(0.126591))),exp(0.523777)))))))`
$\text{Prog}^{\text{Go\_S}}$:	`(rlog((cos(sin(E))*(get_M(((cos(((exp(set_M(E,S))+S)*N))%((N%(N+(N-E)))%E))+E))+cos((get_M(((N+S)*(exp(rlog(rlog(N)))-((S+W)-(set_M(S,(N+E))%N)))))-cos((W*set_M(0.758213,W))))))))+((W-S)-S))`
$\text{Prog}^{\text{Go\_W}}$:	`set_M((E-sin((S+S))),(get_M(rlog((sin((exp(sin(rlog(E)))+get_M((set_M(rlog(E),E)-(W%N)))))+(sin(set_M(E,0.061605))%cos(0.542749)))))-exp(sin(W))))`
$\text{Prog}^{\text{Go\_E}}$:	`set_M((S-set_M((S*exp(E)),exp(get_M(set_M(W,N))))),cos(exp(W)))`
Prog^{g}:	`(cos(0.666886)+((rlog((E*E))-((E-(sin((N%0.776983))+get_M(cos(W))))+(sin(E)-E)))%((S+0.860636)-W)))`

7.1 represents a partly stochastic, partly highly deterministic policy. Table 7.2 shows the corresponding probabilities of all actions during generation of a shortest path. Like the maze, Table 7.2 is divided into three parts. Starting with step 0 (= location S), the agent must go north for 8 steps to stay on the shortest path. In steps 8-20 it has to keep going east; in steps 21-28 it must go south. Table 7.2 shows that except for step 21, the optimal action at each step is the one with highest probability of being selected. The program uses memory cells to distinguish between ambiguous observations at steps 1-7 and 22-28 in an almost deterministic manner. In steps 12, 15, 18, and 21, however, the agent's policy involves a high degree of stochasticity (compare Jaakkola, Singh, and Jordan, 1995). Since successive populations generally contain multiple copies of such a program, the shortest path will be discovered within a reasonable time. It can then be stored separately. Note that the program sets the greediness parameter g by itself, thus controlling how stochastic/deterministic its policy is at any given step.

Table 7.2: Probability of each action in a given maze location as calculated by the example program in Table 7.1, for the shortest path from S to G. For each step, the probability of the optimal action appears in boldface.

	Action probabilities			
step	Go_N	Go_S	Go_W	Go_E
0	**0.417796**	0.057893	0.196644	0.327667
1	**0.965758**	0.034239	0.000000	0.000002
2	**0.879572**	0.120418	0.000001	0.000009
3	**0.879572**	0.120418	0.000001	0.000009
4	**0.879572**	0.120418	0.000001	0.000009
5	**0.879572**	0.120418	0.000001	0.000009
6	**0.879572**	0.120418	0.000001	0.000009
7	**0.879572**	0.120418	0.000001	0.000009
8	0.051486	0.047469	0.017957	**0.883088**
9	0.074389	0.016590	0.072058	**0.836963**
10	0.074389	0.016590	0.072058	**0.836963**
11	0.074389	0.016590	0.072058	**0.836963**
12	0.015821	0.229479	0.015041	**0.739660**
13	0.074389	0.016590	0.072058	**0.836963**
14	0.074389	0.016590	0.072058	**0.836963**
15	0.015821	0.229479	0.015041	**0.739660**
16	0.074389	0.016590	0.072058	**0.836963**
17	0.074389	0.016590	0.072058	**0.836963**
18	0.015821	0.229479	0.015041	**0.739660**
19	0.074389	0.016590	0.072058	**0.836963**
20	0.074389	0.016590	0.072058	**0.836963**
21	0.015821	**0.229479**	0.015041	0.739660
22	0.000273	**0.999727**	0.000000	0.000000
23	0.000000	**1.000000**	0.000000	0.000000
24	0.000000	**1.000000**	0.000000	0.000000
25	0.000000	**1.000000**	0.000000	0.000000
26	0.000000	**1.000000**	0.000000	0.000000
27	0.000000	**1.000000**	0.000000	0.000000
28	0.000000	**1.000000**	0.000000	0.000000

Stochasticity. If actions are selected uniformly random, then the probability of finding the shortest path for the small maze is $4^{-29} \approx 3.5 \cdot 10^{-18}$. If actions are chosen according to the example program in Table 7.1, this probability is approximately $2.15 \cdot 10^{-3}$. Although the program does not represent a deterministic solution, it will find the shortest path every 324th run on average. Compare this to $2.9 \cdot 10^{17}$ runs in case of uniformly random selected actions. Even if we knew (but we do not) that the shortest path requires 29 steps, we would have to process on average $8.4 \cdot 10^{18}$ agent moves to find it. PIPE, however, finds it within $3.15 \cdot 10^{7}$ agent moves in the median run.

Results for Large Maze

Again we conducted 11 independent runs. PIPE always found the optimal solution of 56 steps. The earliest (latest) discovery of an optimal solution took 1,782 (12,733) generations, or 178,200 (1,273,300) program evaluations. The median run took 2,717 generations, or 271,700 program evaluations. Recall that only the fastest program in each generation is run to completion. If actions are selected uniformly random, then the probability of finding the shortest path is 4^{-56}.

7.4.3 Conclusion

PIPE variants with memory-setting and memory-reading instructions are applicable to partially observable environments.

If the goal is to discover a program with minimal runtime, then PIPE's parallel population evaluation gains efficiency by stopping runs of programs slower than the best. This results in speed-ups even on serial computers.

7.5 Long Time Lag Challenge

Some pattern classification tasks with a simple sequential solution are hard to learn by static approaches such as, e.g., feedforward neural networks. For instance, the parity problem requires to separate bitstrings of length $n > 0$ (n integer) with an odd number of zeros from others. In principle the task is solvable by a 3-layer feedforward net with n input units. But learning the task from training exemplars is hard for $n > 20$, due to such a net's numerous free parameters. On the other hand, a very simple finite state automaton with just one bit of internal state can correctly classify arbitrary strings by sequentially processing them one bit at a time, and switching the internal state bit on or off depending on whether the current input is 1 or 0. Such simple observations make sequential, event-memorizing behavior interesting.

7.5.1 Analog vs. Discrete Methods

How to learn sequential, event-memorizing behavior from training examples? *Analog* methods typically use gradient-based methods to search continuous spaces of algorithms represented as sets of real numbers (such as weight matrices of recurrent neural nets). *Discrete* methods search spaces of enumerable algorithms composed from a finite set of primitive instructions. If there are long time lags between relevant events and later error signals, then most analog gradient-based recurrent net learning algorithms, such as "Back–Propagation Through Time" (BPTT, e.g., (Rumelhart, Hinton, and Williams, 1986; Werbos, 1988; Williams and Zipser, 1992)) or "Real-Time Recurrent Learning" (RTRL, e.g., (Robinson and Fallside, 1987)) (see overviews by Williams, 1989; Pearlmutter, 1995), will not work. Their main problem is that error signals "flowing backwards in time" tend to decay exponentially, as was shown first by Hochreiter (1991) and later by Bengio, Simard, and Frasconi (1994). A gradient-based method called "Long Short-Term Memory" (LSTM — Hochreiter and Schmidhuber, 1997a) eliminates some of gradient-based approaches' problems and can solve complex long time lag tasks involving distributed, high-precision, continuous-valued representations.

Even LSTM, however, does not fully eliminate the dependence on the time lag size. *Discrete* search methods, on the other hand, do not care for time lag size at all. They are of particular interest where the algorithmic complexity (AC) of a solution is low (i.e., the solution can be implemented by a short program in a given programming language representing initial bias). For instance, the few free parameters of a parity-recognizing *recurrent* net with a single input and hidden unit can be quickly and successfully *guessed* — few trials with random weight initializations and a small training set are sufficient to obtain solutions (weight matrices) with perfect generalization on a large test set (Schmidhuber and Hochreiter, 1996; Hochreiter and Schmidhuber, 1997b).

Weight guessing is one of the simplest discrete methods. It will not solve *non*-trivial tasks (requiring many or precise parameters) in reasonable time. More sophisticated discrete methods searching incrementally for better sequence-processing algorithms are needed. Such methods are Adaptive Levin Search (Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997b) based on Levin Search (Levin, 1973, 1984), Genetic Programming (Cramer, 1985; Dickmanns et al., 1987; Koza, 1992) with memory cells (e.g., Teller, 1994), and Probabilistic Incremental Program Evolution (PIPE) with memory cells (Sałustowicz and Schmidhuber, 1997a).

7.5.2 Long Short Term Memory

Long Short Term Memory (LSTM – Hochreiter and Schmidhuber, 1997a) is a recent, analog, gradient-based recurrent neural net approach for supervised learning of sequential processes. Unlike most alternative approaches it can learn from training sequences that do not exhibit any short time lags between relevant events. It does so by enforcing constant error flow through "constant error carrousels"(CEC) within special units, and applying multiplicative gate units that learn to open and close access to the constant error flow. LSTM combines CEC and multiplicative input and output gates to form memory cells that can store information over arbitrary periods of time. See Hochreiter and Schmidhuber (1997a), or Gers, Schmidhuber and Cummins (2000) for a detailed description of net structure and learning algorithm.

7.5.3 Time-Lag Size and Algorithmic Complexity

Tasks can be characterized by: (1) time lag size, and (2) an approximation of the algorithmic complexity of their solutions.

Time-Lag Size

Time-lag size measures the number of irrelevant time steps between some relevant event and a corresponding later error signal.

Algorithmic Complexity

Algorithmic complexity (AC) traditionally is used as a synonym for "Kolmogorov complexity", which refers to the length of the shortest program computing a solution (Kolmogorov, 1965; Solomonoff, 1964; Chaitin, 1987). Naturally the length of the program depends on the "programming language" representing the initial bias. Since Kolmogorov complexity is not computable in general, however, we roughly approximate it by simply counting the number of relevant event combinations that need to be distinguished by the learner. In general, the more event combinations, the longer the program that encodes them all.

7.5.4 Comparisons

PIPE and LSTM are compared on two problems involving both long minimal time lags and low algorithmic complexity (AC). So far both the "adding problem" and the "temporal order problem" (Hochreiter and Schmidhuber, 1997a) have been solved by only one single analog method (LSTM). (BPTT and RTRL failed.) The adding experiments show that LSTM's convergence speed depends on time lag size, while PIPE's does not. Sometimes simple ROLs suffice. The forthcoming temporal order experiments, however, will require memorizing (output) cells.

Two measures are used to compare the performances of LSTM and PIPE: (1) the probability of finding a solution within a given time frame, and (2) the average time needed to find a solution (with respect to the given time frame). Time is measured by the number of sequence presentations, to remain independent of implementation issues of LSTM and PIPE.

7.5.5 Adding Problem

The task is to identify two relevant, real-valued input components occurring in a long sequence and to output their sum at the end. The task's AC is low because a single combination of only two (although widely separated) past events is necessary for correct prediction (of the sum).

Task Definition

Training and test sequences have random lengths varying from minimal sequence length T to $T + \frac{T}{10}$. Each element of each input sequence is a pair of components. The first component is a real value randomly chosen from the interval $[-1, 1]$; the second is either 1.0, 0.0, or -1.0 for LSTM and 1.0, or 0.0 for PIPE. It is used as a marker: at the end of each sequence, the task is to output the sum of the first components of those pairs that are *marked* by second components equal to 1.0. In a given sequence exactly two pairs are marked as follows: first randomly select and mark one of the first ten pairs (whose first component is then called X_1). Then randomly select and mark one of the first $\frac{T}{2} - 1$ still unmarked pairs (whose first component is then called X_2). The second components of all remaining pairs are zero. Since LSTM needs a "trigger input" to mark the end of a sequence, the second component of the final pair is set to -1. An error signal is generated only at the sequence end: the target for LSTM is $0.5 + \frac{X_1+X_2}{4.0}$ (the sum $X_1 + X_2$ scaled to the interval $[0, 1]$) and for PIPE $X_1 + X_2$. A sequence is processed correctly if the absolute error at the sequence end is below 0.04.

LSTM Setup

LSTM's network topology and most important parameter values are listed here. A detailed description of all LSTM parameters is in (Hochreiter and Schmidhuber, 1997a). A 3-layer net with 2 input units, 1 output unit, and 1 cell block of size 2 is used. The output layer receives connections only from memory cells. Memory cells and gate units receive inputs from memory cells and input units. All non-input units have bias weights. All activation functions in the hidden layer are logistic with output range $[0, 1]$, except for h, whose range is $[-1, 1]$, and g, whose range is $[-1, 1]$. Output units do not apply a squashing function. The learning rate is set to 0.5. Online learning is used, generating sequences dynamically right before they are used for training. Training is stopped when the average training error is below 0.01, and the 2000 most recent sequences have been processed correctly or a total of 5,000,000 sequences has been processed.

PIPE Setup

Training and Test Environment. Each generation a new training data set is generated. It contains 100 randomly generated sequences. When all training sequences of a data set are processed correctly by a program, the program is manually verified whether it constitutes an analytically correct ("perfect") solution.
Fitness Function. The fitness of a program is the average absolute sum of differences of the program's output at sequence end and the $X_1 + X_2$ target (see above).
Instruction Set. PIPE uses recurrent output links (ROLs). The function set is $F = \{+, -, *, \%, nop, sin, cos, exp, rlog\}$, where nop is a single argument identity function (all other functions are defined in Section 4.1.1), and the terminal set is $T = \{x_0, x_1, o\}$, where x_0, x_1 are input variables and o is the ROL. At the beginning of each sequence o is set to 0.
Output Interface. No specific output interface is required.
Parameter Setup. Following parameters are used: PE = 20,000, P_T=0.9, ε = 0.000001, P_{el}=0.0, PS=10, lr=0.01, P_M=0.1, mr=0.1, T_R=0.3, T_P=

Table 7.3: Results for the adding problem. T is the minimal sequence length, $T/2$ the minimal time lag. LSTM's column "perfect solutions" gives the percentage of perfect solutions (all sequences in the test set correct) from a test set containing 2560 sequences. LSTM's "success after" column provides the number of training sequences required to achieve LSTM's stopping criterion. Values for T = 50, 100, 500, and 1000 are means of 10 trials; values for T = 1500, 2000, 2500, and 5000 are means of 6 trials. PIPE's "perfect solutions" column reports on how likely PIPE is to find perfectly predicting, algorithmically correct solutions. PIPE's "success after" column provides the number of training sequences required on average (means of 35, 33, 37, 39, 41, 37, 35, and 39 independent runs for T = 50, 100, 500, 1000, 1500, 2000, 2500, and 5000 respectively) to achieve a perfect solution (within the PE = 20,000 time limit).

		LSTM		PIPE	
T	minimal lag	perfect solutions	success after	perfect solutions	success after
50	25	100%.	127,000	70%	786,000
100	50	100%	172,000	66%	832,000
500	250	100%	253,000	74%	689,000
1000	500	100%	531,000	78%	832,000
1500	750	60%	526,000	82%	830,000
2000	1000	60%	1,007,000	74%	613,000
2500	1250	60%	1,266,000	70%	661,000
5000	2500	0%	n.a.	78%	675,000

0.999999, $FIT_s = 0$.

Results

The minimal time lag between the most recent occurrence of relevant information and the point of prediction varies from 25 to 2500 time steps. Table 7.3 summarizes all results. With increasing minimal time lag, LSTM needs more and more sequence presentations (from 127,000 up to 1,266,000) to solve the task. LSTM also finds fewer and fewer "perfect solutions" (from 100% down to 0%) with respect to a test set consisting of 2560 randomly chosen sequences. (10 independent trials were conducted for each time lag size from 25 to 500 and 6 trials for each larger time lag size.) With minimal time lag of 2500 LSTM did not find a single "perfect solution" within 5,000,000 sequence presentations. PIPE is able to find perfectly generalizing solutions (0 incorrectly processed sequences) in 66%-82% of all independent runs (50 for each time lag size). The number of required sequences varies, but seems independent of whether the minimal time lag is 25, 50, 250, 500, 750, 1000, 1250, or 2500 time steps. Although LSTM learns significantly faster than PIPE in case of smaller minimal time lags (25, 50, 250, 500 and 750), PIPE outperforms LSTM in case of very long ones (1000, 1250, and 2500).

Conclusion

LSTM's time lag dependence mostly stems from error signal interference that increases with sequence length (just like it is harder for a feedforward net to discover 1 relevant input unit among 100 irrelevant ones, than 1 among 10). The sample complexity (depending on the sequence length itself) may play a minor role, too. The task's AC remained constant, however, and the competing discrete method (PIPE) did not seem affected by the time lag increase.

7.5.6 Temporal Order Problem

The following task has been solved by only one analog method (LSTM). (BPTT and RTRL failed.) The goal is to classify sequences into four classes depending on the temporal order of two symbols in the sequence. Since there are only four relevant symbol combinations the task has a relatively low AC. It is sufficient, however, to prevent PIPE with ROLs from working efficiently. Memorizing cells are needed.

Task Definition

Inputs and targets of a sequence are represented locally (input vectors with only one non-zero bit). The sequence starts with an E, ends with a B (the "trigger symbol") and otherwise consists of randomly chosen symbols from the set $\{a, b, c, d\}$ except for two elements at positions t_1 and t_2 that are either X or Y. The sequence length is randomly chosen between 100 and 110, t_1 is randomly chosen between 10 and 20, and t_2 is randomly chosen between 50 and 60. There are 4 sequence classes Q, R, S, U which depend on the temporal order of X and Y. The rules are: $X, X \rightarrow Q$; $X, Y \rightarrow$

R; $Y, X \to S$; $Y, Y \to U$. There are as many outputs as there are classes. Each class is locally represented by a binary target vector with one non-zero component. Error signals occur only at the end of a sequence. A sequence is classified correctly by LSTM (PIPE) if the final absolute error of all LSTM outputs is below 0.3 (if only PIPE's Boolean output cell associated with the expected output class is set true).

LSTM Setup

Again LSTM's network topology and most important parameter values are listed here. A detailed description of all LSTM parameters is in (Hochreiter and Schmidhuber, 1997a). LSTM uses a 3-layer net with 8 input units, 2 cell blocks of size 2 and 4 output units. All non-input units have bias weights, and the output layer receives connections from memory cells only. Memory cells and gate units receive inputs from input units, memory cells and gate units. All activation functions are logistic with output range $[0, 1]$, except for h, whose range is $[-1, 1]$, and g, whose range is $[-2, 2]$. The learning rate is 0.5. Online learning is used, generating sequences dynamically right before they are used for training. Training is stopped once the average training error falls below 0.1 and the 2000 most recent sequences have been classified correctly. All weights are initialized in the range $[-0.1, 0.1]$. The first input gate bias is initialized with -2.0, and the second with -4.0. The test set consists of 2560 randomly chosen sequences.

PIPE Setup

Training and Test Environment. Each generation a new training data set is generated. It contains 100 randomly generated sequences. PIPE programs are applied to each sequence. In case the best program of a generation classifies 100% of the training data correctly, its performance is tested on 5000 randomly created test sequences. The run is stopped when a program classifies all training and test sequences correctly or the time constraint PE (see below) is exceeded.

Fitness Function. The fitness of a program is the number of training sequences the program misclassifies. A sequence is classified correctly if at the end of the sequence only the output cell associated with the expected output class is switched on.

Instruction Set. 4 MPs and 4 Boolean OCs with direct memory access are used. PIPE programs may set and reset output cells several times

Table 7.4: Results for the temporal order problem. "# wrong predictions" is the number of sequences incorrectly classified by LSTM (error > 0.3 for at least one output unit) from a test set containing 2560 sequences. For LSTM the "success after" column provides the number of training sequences required to achieve LSTM's stopping criterion. The results are means of 20 trials. PIPE's "solved" column reports how often PIPE was able to find solutions that correctly classify all sequences of the training data set (containing 100 sequences) and the test data set (containing 5000 sequences). PIPE's "success after" column displays how many sequence presentations were necessary on average (means of 46 runs).

LSTM		PIPE	
# wrong predictions	success after	solved	success after
1 out of 2560	31,390	92%	6,048,000

while processing a single data point of a sequence. Since OCs are used, the MPs merely impose an *a priori* structure on the full program. At the beginning of each sequence all OCs are set to false. Boolean values are represented by integers: 1 for true and 0 for false. The function set is $F = \{if\_set\_O_0\_else, if\_reset\_O_0\_else, if\_set\_O_1\_else$, $if\_reset\_O_1\_else$, $if\_set\_O_2\_else$, $if\_reset\_O_2\_else$, $if\_set\_O_3\_else$, $if\_reset\_O_3\_else\}$, where the two argument function $if\_set\_O_i\_else(arg_1, arg_2)$ $(if\_reset\_O_i\_else(arg_1$, $arg_2))$ $(0 \leq i \leq 3)$ sets the i-th output cell to true (false) and returns true if arg_1 evaluates to true. Otherwise arg_2 is returned. The terminal set is $T = \{E, B, a, b, c, d, X, Y\}$, where E, B, a, b, c, d, X, Y are Boolean input variables.

Parameter Setup. Following parameters are used: PE = 500,000, P_T=0.8, $\varepsilon = 0.000001$, P_{el}=0.0, PS=10, lr=0.1, P_M=0.2, mr=0.2, T_R=0.3, T_P=0.99, $FIT_s = 0$.

Results

Table 7.4 summarizes all results. LSTM results are taken from (Hochreiter and Schmidhuber, 1997a). LSTM finds almost perfect or perfect solutions after on average just 31,390 sequence presentations. PIPE is able to solve the problem in 92% of the time, but needs significantly more presentations.

Conclusion

A discrete method (PIPE) can employ memorizing cells to successfully solve a task that so far has been solved only by LSTM. (BPTT and RTRL failed.) LSTM, however, is much faster most likely because the time lags are not extremely long (see Section 7.5.5).

7.5.7 Conclusion

Benchmarking LSTM against PIPE showed that LSTM's requirements grow faster than PIPE's as time lag size grows. In case of high AC and not too long time lags, however, LSTM tends to be superior.

7.6 Conclusion

This chapter presented various ways how PIPE can be augmented with memory. Equipped with recurrent output links (ROLs) or memorizing cells (*MC*s or *OC*s) PIPE was able to solve tasks with a temporal context.

PIPE, e.g., found solutions to difficult reinforcement learning problems (maze tasks). This application also revealed that PIPE can achieve significant learning speedups (even on serial computers) by evaluating programs in parallel, if the goal is to find programs with minimal runtime. The learning speedup stems from PIPE's inherent learning strategy that makes PIPE just learn from the *best* program of a generation.

Furthermore, PIPE seems well suited for solving "long time lag" tasks, where the goal is to relate relevant inputs to error signals that only occur after a long minimal time lag. PIPE compared favorably to neural network approaches on a long time lag task with low algorithmic complexity (AC). On a task with shorter time lags and higher AC the currently most suited neural network algorithm (LSTM - "Long Short Term Memory", Hochreiter and Schmidhuber, 1997a) was able to outperform PIPE.

The next chapter will show how PIPE can be augmented with a novel automatic task decomposition method called *filtering* to solve tasks with high algorithmic complexity.

Chapter 8

Automatic Task Decomposition

We have seen how long time lag tasks with *low* algorithmic complexity (AC) can be solved by PIPE in conjunction with either ROLs or memorizing cells. Although memorizing cells offer advantages over ROLs (which only work for problems with extremely low AC), discrete methods such as PIPE fail to learn programs that memorize a vast number of independent relevant event combinations within acceptable time. For instance, when we tried PIPE on a high AC task (see Section 8.2) we obtained only partial solutions. Varying the number of memorizing cells did not help much: the problem is not the memory limitation but PIPE's limited ability to integrate complex information into a single program. To enable discrete methods such as PIPE to deal with high AC tasks we need to split them into subtasks that can be solved independently and then be assembled into an overall solution.

8.1 Filtering

To overcome AC-related drawbacks of discrete methods we develop *filtering* (Sałustowicz and Schmidhuber, 1999b), a novel, general divide-and-conquer method for automatic task decomposition. Unlike with certain previous approaches, e.g., (Angeline and Pollack, 1992; Koza, 1992; Spector, 1996), the decomposition is not "ad-hoc" but data-dependent. Filtering learns "experts" and special "gates" (filters) to decompose a task. Experts learn target values of data points, while filters learn which data points to assign to which experts. The first expert is taught to fit as many training data points as possible. After a while the training set is split into learned and yet unlearned data. The next expert then tries to fit just the unlearned data, etc. Once all data points have been fit by various experts, each expert also needs to learn which incoming (test) data to process and which to pass on to the next expert. This possibly complex decision task is also adaptively decomposed into subtasks learned by sequences of filters, each passing the current data to either its local expert or the next filter of the local expert or the first filter of the next expert.

Filtering is different from boosting (Schapire, 1990; Drucker, Cortes, Jackel, LeCun, and Vapnik, 1994) or "mixtures of experts" (Jordan and Jacobs, 1992). Training sets of different experts do not overlap, and there is no voting or adding mechanism. Instead the complex problem of assigning test data to trained experts is solved sequentially by chains of adaptive filters that learn to pass on data until an appropriate expert is found.

Filtering facilitates the task of the learning algorithm. Still, the discovery of algorithmic regularities allowing for good *generalization* on test data remains the burden of the learning algorithm itself. If it does not discover any then filtering will essentially yield a lookup table.

Filtering's basic idea is independent of a particular approach such as PIPE. It can be used in combination with PIPE, GP, neural networks and many other learning algorithms or a hybrid system of multiple learning algorithms. An important aspect of filtering is that it does *not* merely shift the problem without reducing its complexity: no single component (filter or expert) needs to be particularly powerful or significantly more potent than others.

We will see in Section 8.2 that a filtering variant that uses PIPE for learning both experts and filters achieves excellent generalization performance on a complex task unsolvable by PIPE itself.

8.1.1 Filtering Non–Temporal Data Sets

For clarity we will first show how filtering can help in learning static input patterns. It comprises two phases: (1) task decomposition (TD), and (2) task assembly (TA). During TD a task is automatically decomposed into subtasks that are then solved independently. During TA partial solutions are assembled into a final one. Either the same learning algorithm, or several different ones (hybrid system) can be used to perform TD and TA.

Task Decomposition

Given a learning algorithm ALG and a training set SET with n_{SET} data points, ALG is to output a desired target value for each data point in SET. Data points are treated in a discrete way: ALG is said to have learned a data point if the absolute difference between its target value and ALG's output falls below ϵ_d. ALG is trained on all data points in SET until either the task is solved (according to ALG's termination criterion), or until ALG has not been able to improve (learn more data points) for some prespecified interval Els_{max}. ALG, however, needs to learn at least $Ed_{min} \geq 1$ data points. If ALG stops and the task is not finished yet, (1) the learned partial solution (the first expert – E_1) is saved, (2) SET is split into SET_{E_1} containing the data correctly learned by expert E_1 and into SET_{rest}, a set containing the remaining data. Then ALG is applied to SET_{rest} and the procedure of saving experts and splitting the data set is repeated until all data points have been learned. This decomposes the task into subtasks in a way depending only on learning algorithm and data set. Note that E_i learns from a smaller data set than E_j for $i > j$.

Task decomposition by itself, however, is insufficient. After a task has been decomposed, the partial expert solutions need to be assembled in a way that allows for sensibly classifying new, previously unseen test data. The next subsection will address the question: which data points should be assigned to which expert?

Task Assembly

Filters. The task of assigning data to experts may be almost as difficult as the data fitting process itself, and may require similar decomposition. For this purpose chains of "filters" (sequentially invoked gates) are used and associated with each expert. Let $F^j_{E_i}$ denote the jth filter of the ith expert. See Figure 8.1 for an example architecture with experts E_1, E_2, E_3 and corresponding filter chains $F^1_{E_1}$, $F^2_{E_1}$, $F^3_{E_1}$, $F^4_{E_1}$, and $F^1_{E_2}$, $F^2_{E_2}$, $F^3_{E_2}$ respectively. Each incoming data point first moves to the first expert's first filter. Filters are either positive or negative: positive filters take a data point and decide whether to pass it on to their expert or not. Negative filters decide whether the data should definitely *not* be passed to their expert. In this case it is passed to the first filter of the next expert or directly to the next expert if it is the final one. Data points that cannot be decided upon are simply passed on to the next filter in the chain.

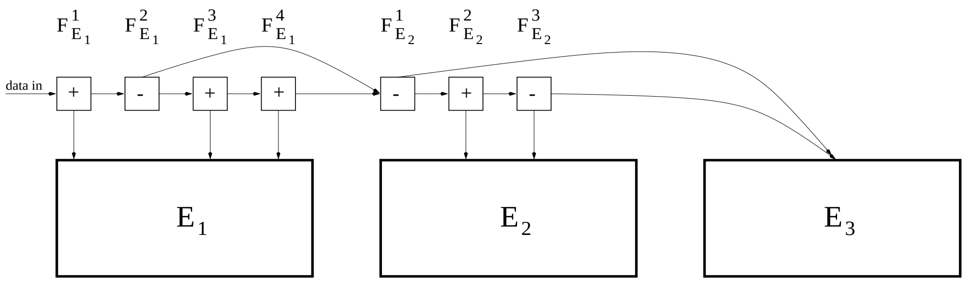

Figure 8.1: Three experts and their associated filters. Arrows indicate possible data flow. (Figure taken from (Sałustowicz and Schmidhuber, 1999b).)

Filter learning. Filters are learned sequentially in order of expert and filter numbers by dynamically relabeling the data in SET. To train $F^i_{E_j}$, all data points of SET that have not been learned by any previous filter $F^y_{E_x}$, for all $x < j$ and all $y < i$, if $x = j$, are labeled as belonging to class I if they are in SET_{E_j} and to class II otherwise. If there are more class I than class II data points then a positive filter will be learned, otherwise a negative one. A positive filter $F^i_{E_j}$ will learn to assign class I data points to E_j. A negative filter $F^i_{E_j}$ will learn to pass class II data points to the first filter of the next expert $F^1_{E_{j+1}}$, or to the next expert E_{j+1} in case E_{j+1} is the final one. No positive $F^i_{E_j}$ may pass any class II data points on to E_j and no negative $F^i_{E_j}$ may pass any class I data points on to $F^1_{E_{j+1}}$ or E_{j+1}. If a single filter has separated at least $Fd_{min} \geq 1$ data points, but not all of them, and was not able to improve its performance (by separating additional data points) for some prespecified interval Fls_{max}, (1) the filter ($F^i_{E_j}$) is preserved, (2) the data learned by $F^i_{E_j}$ is eliminated from class I or II, depending on filter type, and (3) the next filter $F^{i+1}_{E_j}$ is trained to separate the remaining data points. In this way filters are added incrementally until all class I and class II data points have been correctly classified. Then filters $F^i_{E_{j+1}}$ for the next expert are learned, and the entire procedure is repeated until all filters for all experts (except for the final one) have been learned. Note that the number of data points to be separated decreases with each learned filter.

8.1.2 Filtering Temporal Data Sets

With temporal data sets each training sequence may involve several intermediate target signals (e.g., each time step may require a new prediction).

Therefore sets of training *sequences* are split and grouped into learned/unlearned and class I/II sets in a slightly different way. Since temporal dependencies can occur among unlearned and already learned points, one cannot simply exclude the learned points from the training data set of an expert/filter: all experts and filters need to see all inputs of the *entire* data set *SET*. Data set splits during task decomposition and assembly are achieved by measuring an experts'/filter's performance only on data points that have not been already learned by a previous expert/filter.

Also during later processing of (previously unseen) test data, each data point is given to each filter and expert. Filter outputs are then processed sequentially (starting with $F^1_{E_1}$) to determine which expert's output is valid.

Filtering facilitates the decomposition of temporal tasks with many independent relevant event combinations. Detecting the relevant dependencies within a single event combination, however, remains the duty of the learning algorithm.

In Section 8.2 we will see that PIPE with memory cells plus filtering can extract the algorithmic regularities necessary for achieving perfect generalization.

8.1.3 Filtering with Few Data

Filtering can be applied even when few data is available. For instance, if there is a training set containing just two data points of the type (x, f(x)): f(1) = 10 and f(2) = 20, and if expert E_1 and E_2 are only able to assign a single output value to a single input value, then expert E_1 can learn f(1) = 10, expert E_2 f(2) = 20, and a filter $F^1_{E_1}$ can learn to split the data set and assign the data point with input 1 to expert E_1 and the data point with input 2 to expert E_2.

8.1.4 Relation to Boosting etc.

The well-known method of *boosting* essentially first trains a learning algorithm LA1 on a training subset T1, then creates a new training subset T2 for algorithm LA2 by filtering data through LA1 such that T2's distribution differs from T1's and includes elements of T1 misclassified by LA1, and so forth. After training, test data is classified by letting the LAs vote (see Schapire's theoretical result (Schapire, 1990)) or by adding their outputs (Drucker et al., 1994). A somewhat related approach called *mixtures of experts* (Jordan and Jacobs, 1992) uses gradient descent for learning to add outputs of different experts trained in parallel on all patterns.

Filtering is quite different from all these methods: (1) training sets of different experts do not overlap, (2) there is no need for a voting or adding mechanism, (3) the (in general) complex problem of assigning test data to trained experts is also decomposed into a sequence of subproblems; chains of adaptive filters learn to pass on data until an appropriate expert is found.

8.2 Embedded Reber Grammar

The task is to learn the "embedded Reber grammar", e.g. Smith and Zipser (1989), Cleeremans, Servan-Schreiber, and McClelland (1989), and Fahlman (1991). It allows for training sequences with very short time lags and can therefore be learned by many recurrent net algorithms. Its AC is rather high, though, since predictions are required at each time step, and numerous input combinations need to be learned. PIPE without filtering completely failed to solve this task. During its best runs PIPE was merely able to predict roughly 60% of all data points of a sequence correctly. Filtering, however, did enable PIPE to solve this popular recurrent net benchmark.

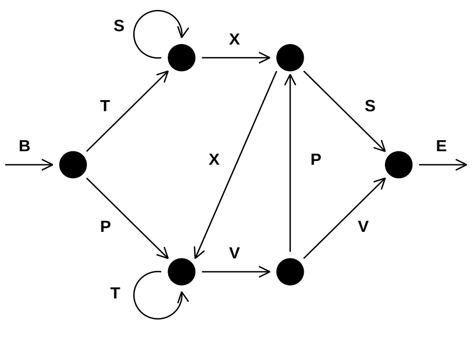

Figure 8.2: Transition diagram for the Reber grammar. (Figure taken from (Hochreiter and Schmidhuber, 1997a).)

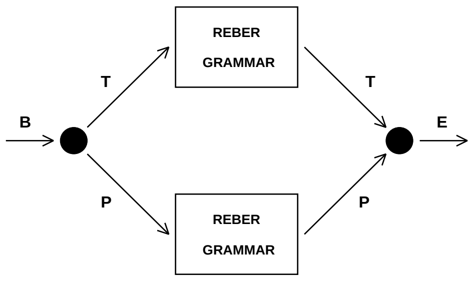

Figure 8.3: Transition diagram for the embedded Reber grammar. Each box represents a copy of the Reber grammar (see Figure 8.2). (Figure taken from (Hochreiter and Schmidhuber, 1997a).)

8.2.1 Task Definition

Starting at the leftmost node of the directed graph in Figure 8.3, symbol strings are generated sequentially (beginning with the empty string) by following edges — and appending the associated symbols to the current string — until the rightmost node is reached. Edges are chosen randomly if there is a choice (probability: 0.5). The task is to read strings, one symbol at a time, and to permanently predict the next symbol (error signals occur at every time step). To correctly predict the symbol before last, the second symbol has to be remembered.

8.2.2 Comparison

PIPE with *MC*s and filtering is compared to "Long Short Term Memory" (LSTM) (results taken from Hochreiter and Schmidhuber, 1997a), "Elman nets trained by Elman's training procedure" (ELM) (results taken from Cleeremans et al., 1989), Fahlman's "Recurrent Cascade-Correlation" (RCC) (results taken from Fahlman, 1991), and "Real Time Recurrent Learning" (RTRL) (results taken from Smith and Zipser, 1989), where only the few successful trials are listed. It should be mentioned that Smith and Zipser actually make the task easier by increasing the probability of short time lag exemplars.

8.2.3 Training and Test Environment

Local input/output representation (7 inputs, 7 outputs) is used. Following Fahlman, 256 training strings and 256 separate test strings are used. The training set is generated randomly. Test sequences are generated randomly, too, but sequences already used in the training set are not used for testing. For PIPE three pairs of training and test sets are generated. The first two pairs (1,2) have training sets that contain on average shorter sequences than their corresponding test sets. For the third pair (3) the opposite is true. A trial is considered successful if all symbols of all sequences in both test set and training set are predicted correctly — that is, if the output value(s) corresponding to the possible next symbol(s) is (are) always the largest ones. PIPE's test performance is measured on all three test sets.

8.2.4 Neural Network Setups

Architectures and parameter settings for LSTM, RTRL, ELM, and RCC are reported in the references listed in Section 8.2.2.

8.2.5 PIPE Setup

PIPE is used with *MC*s, MPs, and filtering. Each expert consists of 7 programs (one for each output) that share 10 *MC*s. Each filter consist of one program with 10 *MC*s. *MC*s can hold continuous values and are initialized to 0 before each sequence presentation.

Fitness Function

Expert fitness is the number of wrong predictions. Filter fitness for a positive (negative) filter is the number of incorrectly classified class I (II) points, if all class II (I) points have been classified correctly, and infinite otherwise.

Instruction Set

The function set is $F = \{+, -, *, \%, set\_M, get\_M, sin, cos, exp, rlog\}$ (see Sections 4.1.1 and 7.2 for function definitions), and the terminal set is $T = \{B, T, S, X, E, P, V, R\}$, where B, T, S, X, E, P, V are input variables and R is the GRC in [0;1).

Output Interface

No specific output interface is required for experts. For filters program output is mapped to class I if > 0 and to class II otherwise.

Parameter Setup

The following parameter setup was used: Els_{max}=1,000 program evaluations, Ed_{min}=1, Fls_{max}=10,000 program evaluations, Fd_{min}=1, $PE = 5,000,000$, $FIT_s = 0$, P_T=0.9, $\varepsilon = 0.000001$, P_{el}=0.1, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3, T_P=0.9.

8.2.6 Results

Table 8.1 shows all results for the analog methods. LSTM is the only one that always learns to solve the task. RTRL and RCC perform better than ELM, but worse than LSTM. Results for PIPE with filtering are shown in Table 8.2. Filtering enabled PIPE to always *learn* the task. PIPE programs consisted of on average 4 experts (min. 3, max. 7), a chain of 6 filters (min. 3, max. 12) for E_1, 3 filters (min. 1, max. 7) for E_2, and 2 filters (min. 1, max. 4) for E_3. Most likely due to presence of short time lags, however, LSTM

Table 8.1: Results of several analog approaches for the embedded Reber grammar: percentage of successful trials and number of sequence presentations until success for RTRL (results taken from Smith and Zipser, 1989), "Elman net trained by Elman's procedure" (results taken from Cleeremans et al., 1989), "Recurrent Cascade-Correlation" (results taken from Fahlman, 1991) and LSTM (results taken from Hochreiter and Schmidhuber, 1997). Only LSTM always learned to solve the task. It also needed least sequence presentations on average (mean of 30 trials).

Analog Approaches			
method	hidden units	% of success	success after
RTRL	12	"some fraction"	25,000
ELM	15	0	>200,000
RCC	7-9	50	182,000
LSTM	3 blocks, size 2	100	8,440

Table 8.2: PIPE's results for the embedded Reber grammar: The "tr. set" column shows which training set is used. The "av. tr. err. / max. err." column reports PIPE's average training error (fitness) and the maximal error (worst possible fitness) on the training set. The "success after" column reports on how many sequence presentations (averaged over 20 runs) are necessary to achieve perfect performance on the training set. The rightmost two columns report PIPE's average (vs. worst possible) test set performance on all three test sets and on how often PIPE discovered perfectly generalizing solutions.

PIPE				
tr. set	av. tr. err. / max. err.	success after	av. test err. / max. err.	perfect solutions
1	0 / 4018	24,880,896	0 / 13329	100%
2	0 / 3909	28,062,976	0 / 13329	100%
3	0 / 4717	16,006,400	4.7 / 13329	10%

learned significantly faster (see Section 7.5.5). With the first two training sets (containing short sequences) PIPE was always able to find perfectly generalizing solutions. Training set 3 it *learned* in roughly 2/3 of the time needed to learn training sets 1 and 2. When trained on longer sequences (training set 3), however, it rarely achieved perfect generalization. The imperfect solutions performed close to optimal (1–8 wrong predictions out of 4630–4705) on longer test sequences, but worse (9–21 wrong predictions out of 3994) on shorter ones (from test set 3).

8.2.7 Conclusion

Filtering enabled a discrete method (PIPE) to reliably *learn* the embedded Reber grammar task (when fed with appropriate training data) that PIPE by itself could not learn, and that has been reliably (always) solved by only one analog method (LSTM). PIPE's programs generalized extremely well, except for those learned from long sequences: one of the many non-minimal algorithmic representations of long sequences may be learned quickly but does not necessarily embody a small finite state automaton capable of generating both the long sequences and certain shorter ones outside the training set.

8.3 Conclusion

Filtering is a learning algorithm independent, automatic data and task decomposition method. It can enable a learning algorithm to solve algorithmically complex tasks, which the algorithm by itself cannot solve. Filtering not only splits complex tasks into several subtasks solvable by comparatively simple algorithms (experts) but also decomposes into manageable subtasks the hard problem of finding an appropriate expert for given data, thus going beyond boosting and mixture of expert approaches.

Chapter 9

Conclusion

This dissertation presented Probabilistic Incremental Program Evolution (PIPE). It started out with positioning PIPE at the intersection of two research directions — probability-based program search (PBPS) and evolutionary algorithms (EAs). After a brief review of both PBPS and EAs, it then focused on its main goals to describe PIPE, show that PIPE is not just of theoretical interest, and to elaborate on methods, which would make PIPE applicable to a wide variety of problems. To reach the goals the thesis dealt with the following four subjects:

- the basic PIPE algorithm
- structured programs
- memory
- automatic task decomposition

First we have described PIPE and demonstrated its practical applicability. With structured programs we have then elaborated on a way to enhance the performance of PIPE by providing a method for incorporating more a priori knowledge into the algorithm. Enhancing PIPE with memory made it applicable to the vast class of non-Markovian problems. Finally, automatic task decomposition simplified problems and thus allowed us to solve significantly more complex problems.

In the following sections we will now discuss in detail how and to what extent we have reached our goals, remark on open issues and possible future research. We will conclude this dissertation with final remarks on PIPE.

9.1 The basic PIPE algorithm

First we have described Probabilistic Incremental Program Evolution (PIPE – Sałustowicz and Schmidhuber, 1997a,b,1999a). PIPE searches for programs applying an evolution-based optimization algorithm and probabilistic models of program space. It generates successive populations of programs from an initially random model and adapts the model to make the best program of a current population more probable. In this way PIPE attempts to incrementally find programs that embody better and better solutions to the task at hand.

Applicability

After giving a step by step description on how to setup PIPE for applications, we conducted a first series of experiments.

Those simple experiments have empirically proven that PIPE can solve problems successfully. Furthermore, they have revealed the following: (1) PIPE's solution programs differ much from user-written programs. (2) We can apply PIPE successfully using different programming languages (instruction sets). (3) However, we can also use a single programming language to solve different problems. (4) We can use "standard" settings for most of PIPE's parameters. Although findings (3) and (4) do not make PIPE a parameter-free algorithm, they facilitate setting PIPE up for different problems.

In a second series of experiments we focused on soccer (Sałustowicz et al., 1997a,b,1998) — a complex multiagent task, where our objective was to develop team strategies. We benchmarked PIPE and CO-PIPE (Sałustowicz et al., 1997a), a coevolutionary version of PIPE that learns from playing against itself, against a well engineered single player algorithm (SPA), which did not take into account any team strategies and against TD-Q learning (Sutton, 1988; Watkins, 1989; Peng and Williams, 1996; Wiering and Schmidhuber, 1997), one of the most widely known and promising approaches to reinforcement learning. By varying team sizes we investigated how well PIPE and CO-PIPE were suited for learning cooperative team strategies in a competitive environment. We found that independent of team size both PIPE and CO-PIPE performed significantly better than TD-Q learning. Especially the partial observability problem (POP) and the agent credit assignment problem (ACAP – Weiss, 1996; Versino and Gambardella, 1997) had a negative influence on TD-Q learning's performance.

With POP the agent's input does not tell the agent everything about its environment. The environment is said to be only partially observable and may, from the agent's point of view, change in an inherently unpredictable way. ACAP is the problem of identifying those agents in a team that were indeed responsible for the outcome. PIPE and CO-PIPE did not seem to have problems with POP and ACAP. For small team sizes the SPA performed better than PIPE and CO-PIPE. For a larger team size CO-PIPE reached or exceeded the performance of SPA, while PIPE always exceeded it. We concluded from these comparisons that (1) pure singleagent strategies may not be optimal for larger teams of agents since they do not force cooperation and that (2) PIPE can be successfully applied for developing team strategies.

Concluding from these experiments we found that PIPE is practically applicable and for certain types of problems, such as, e.g., tasks with POP or ACAP, highly competitive.

Limitations

Naturally, like any other optimization algorithm, PIPE has some limitations. To stimulate further research on PIPE we would like to name a few: (1) Currently, we cannot make quantitative statements about PIPE's convergence rate, which strongly depends on terminal set, function set, and task. More experiments with varying instruction sets are needed to better analyze PIPE's adaptation dynamics. (2) Unlike methods by Zhao and Schmidhuber (1996) and Schmidhuber et al. (1997a,b), PIPE does not attempt to improve its own learning algorithm. (3) Currently, PIPE updates its probabilistic model using only one single individual per population. There may be ways of extracting additional information that is implicit in the population. (4) Unlike nonincremental Levin search (LS) (Levin, 1973, 1984), PIPE does not have an optimal way of allocating computation time to programs that do not halt or whose runtimes are unknown. (5) As with with most other comparable algorithms, several control parameters need to be set heuristically.

9.2 Structured Programs

We have developed a principal extension to PIPE: Hierarchical PIPE (H-PIPE — Sałustowicz and Schmidhuber, 1998). H-PIPE is based on restricting the program search space by context free grammars (Whigham, 1995;

Gruau, 1996). Programs are a priori structured by defining the hierarchical order of instructions. Certain instructions cannot contain certain other instructions in their argument subtrees. Additionally "skip nodes" are used to rapidly turn program parts on or off.

We have benchmarked H-PIPE against PIPE, against a PIPE version with just skip nodes, against one with just hierarchical instruction order and no skip nodes, and against a H-PIPE version with a different initial bias (different hierarchical order of instructions). The results show: (1) As expected setting the initial bias right (right hierarchical order of instructions) is crucial to H-PIPE's success. A wrong initial bias has a strong detrimental effect on performance. (2) Skip nodes by themselves do not have much influence on performance. (3) Given the right bias, the hierarchical order of instructions may enhance performance. Sometimes, however, skip nodes are necessary to achieve the performance increase.

With H-PIPE we have investigated a possible way of enhancing PIPE's performance by incorporation of a priori knowledge. We have found that performance can be increased significantly.

Future Work

There are also many yet untried PIPE variants, which may lead to performance enhancements. For instance, we may apply PIPE to programs with automatically defined functions (Koza, 1992) or to programs with even more general jump instructions (Dickmanns et al., 1987). Instead of coding programs by parse trees we may also use grids or directed acyclic graphs. It might also be possible to improve PIPE by updating its probabilistic model based on information conveyed by programs other than the best, and by incorporating second-order statistics similar to those used in string-based evolution (De Bonet, Isbell, and Viola, 1997; Baluja and Davies, 1997). Finally, we may plug PIPE into the on-line backtracking scheme proposed by Schmidhuber (Schmidhuber, 1994; Schmidhuber et al., 1997a,b) to undo probability modifications that have not triggered long-term reward speedups.

9.3 Memory

To make PIPE applicable to *non-Markovian* problems we need to evolve programs that use memory. Otherwise, all problems, where a program's output depends on a program's input *and* the input's temporal context, will remain unsolvable.

We have shown several ways of evolving programs with memory. First we have presented "recurrent output links", where the outputs of a program are directly fed into the program's next time step's inputs. Then we have described memorizing cells — arrays of numbers — and have shown how those cells can be set and read by programs.

We started evolving programs with memory for guiding agents through partially observable environments (POEs). Our POEs were mazes. The agents needed some sort of *short-term* memory to successfully navigate through them. Using memorizing cells we found stochastic solutions for all mazes. During those experiments we also discovered that when searching for programs with minimal runtime, we can boost PIPE's performance. We can evaluate all programs of a current population in parallel and stop the evaluation of all programs as soon as one has delivered a solution. This speedup is inherent to PIPE since PIPE only needs the currently best solution to update its probabilistic model of program space.

Problems with short time lags between relevant inputs and corresponding error signals can be solved by many different machine learning (ML) techniques, such as, e.g., various recurrent neural network approaches. Once, however, the data contains *long minimal time lags* between relevant inputs and corresponding error signals, the task of matching both becomes significantly more difficult. The currently most successful recurrent neural network approach for solving this kind of problems is "Long Short-Term Memory" (LSTM — Hochreiter and Schmidhuber, 1997a). We benchmarked PIPE against LSTM on two tasks with long minimal time lags that more traditional recurrent neural network approaches, such as "Back–Propagation Through Time" (e.g., Rumelhart et al., 1986; Werbos, 1988; Williams and Zipser, 1992) and "Real-Time Recurrent Learning" (e.g., Robinson and Fallside, 1987; see also overviews by Williams, 1989; Pearlmutter, 1995), could not solve. We found that with increasing minimal time lag size LSTM needs more and more time to find fewer and fewer solutions. PIPE's performance seems independent of the minimal time lag size. While for smaller minimal time lags LSTM's performance is better than PIPE's, PIPE outperforms LSTM, when minimal time lag sizes become larger. We also found that PIPE's performance depends on the *algorithmic complexity* (AC) of a task. AC is based on "Kolmogorov complexity", which refers to the length of the shortest program computing a solution (Kolmogorov, 1965; Solomonoff, 1964; Chaitin, 1987). While PIPE outperformed LSTM on a task with very long minimal time lags and low AC, LSTM was better suited for a task with smaller minimal time lags and higher AC.

Overall we conclude that PIPE can be used to evolve programs with memory and that it can be successfully applied to non-Markovian tasks. Especially for tasks with very long minimal time lags between relevant inputs and corresponding error signals, PIPE seems to be a highly competitive alternative to other ML techniques, such as, e.g., recurrent neural networks approaches. On a task with higher AC and shorter minimal time lag, however, PIPE became less competitive. To make PIPE more competitive on tasks with higher AC we can either enhance the basic algorithm, e.g., with structured programs, or find a way to automatically decompose the task into subtasks with lower AC.

9.4 Automatic Task Decomposition

We have developed *filtering* (Sałustowicz and Schmidhuber, 1999b), a learning algorithm independent, automatic task decomposition method. In connection with PIPE filtering first evolves programs, which constitute partial solutions by splitting the training data set into learnable chunks. Then programs are evolved, which assemble the learned partial solutions to a final solution. Thus filtering goes beyond boosting (Schapire, 1990; Drucker et al., 1994) and mixture of experts (Jordan and Jacobs, 1992) approaches. It not only splits complex tasks into several subtasks solvable by comparatively simple algorithms (experts), but also decomposes into manageable subtasks the hard problem of finding an appropriate expert for given data.

To test filtering's utility we have selected a task, which PIPE by itself could not solve. The embedded Reber grammar problem is a popular benchmark for recurrent neural network (RNN) algorithms. Till so far, it has only been reliably solved by LSTM. Other RNN algorithms, such as "Elman nets trained by Elman's training procedure" (Cleeremans et al., 1989), Fahlman's "Recurrent Cascade-Correlation" (Fahlman, 1991), and "Real Time Recurrent Learning" (Smith and Zipser, 1989) only found partial or no solutions. PIPE augmented with filtering was able to discover perfectly generalizing solutions to the problem.

Filtering makes PIPE applicable to a wider class of problems. We enhanced PIPE with filtering and were able to solve problems with higher algorithmic complexity than solvable by PIPE itself. Of course, in principle filtering could be used to augment other learning methods as well. Plugging alternative learning methods into filtering is in fact part of an ongoing research effort. Future progress in algorithm learning may involve combinations of different learning methods. Filtering is a technique sufficiently general to allow for building hybrid systems with several learning algorithms.

9.5 Final Remarks

We have presented Probabilistic Incremental Program Evolution (PIPE), a new, promising machine learning technique at the intersection of probability-based program search and evolutionary algorithms. We have elaborated on various methods to enhance PIPE and have shown that PIPE is applicable to a wide variety of problems. For specific problem groups it even seems to be highly competitive. Starting from this point, much can be done to further investigate PIPE.

From the basic research point of view two main research streams seem particularly promising — the probabilistic program space model and the optimization algorithm. We can elaborate on the basic idea of mapping discontinuous program space into continuous probability space by using more sophisticated and/or differently structured probabilistic program space models. A further investigation of PIPE's adaptation dynamics could lead to the design of new model optimization/update algorithms. In the long run we might even leave the field of *evolutionary* program search and start using other optimization methods, such as, e.g., gradient descent.

From the applied research point of view the following next two steps would constitute a valuable continuation. First, we could define classes of tasks for which PIPE is particularly well suited by establishing the tasks' distinctive features. Some distinctive features, such as long minimal time lags between relevant inputs and corresponding error signals, or the partial observability problem, or agent credit assignment problem in reinforcement learning tasks, have already been mentioned in this dissertation. In a second step, we could then develop new extensions to PIPE, like, e.g., structured programs and filtering — the two extensions described in this thesis — to enhance PIPE and make it quickly applicable to complex real world tasks.

Software

For educational purposes a free of charge software package containing the basic PIPE engine can be downloaded from:

ftp://ftp.idsia.ch/pub/rafal/PIPE_v1.0.tar.gz.

Bibliography

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Dynamic Systems, Measurement and Control*, 97:220–227.

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, pages 236–241, Hillsdale, NJ. Lawrence Erlbaum Associates.

Asada, M., Uchibe, E., Noda, S., Tawaratsumida, S., and Hosoda, K. (1994). A vision-based reinforcement learning for coordination of soccer playing behaviors. In *Proceedings of AAAI-94 Workshop on AI and A-life and Entertainment*, pages 16–21.

Bäck, T. (1993). Optimal mutation rates in genetic search. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2–8. Morgan Kaufmann, San Mateo, CA.

Bäck, T., Rudolph, G., and Schwefel, H.-P. (1993). Evolutionary programming and evolution strategies: Similarities and differences. In Fogel, D. and Atmar, W., editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 11–22. Evolutionary Programming Society, San Diego CA.

Bäck, T. and Schütz, M. (1996). Intelligent mutation rate control in canonical genetic algorithms. In Ras, W. and Michalewicz, M., editors, *Foundation of Intelligent Systems 9th International Symposium, ISMIS '96*, pages 158–167. Springer, Berlin.

Bäck, T. and Schwefel, H.-P. (1996). Evolutionary computation: An overview. In *Proceedings of the Third IEEE Conference on Evolutionary Computation*, pages 20–29. IEEE Press, Piscataway NJ.

Baluja, S. (1994). Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh.

Baluja, S. and Caruana, R. (1995). Removing the genetics from the standard genetic algorithm. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 38–46. Morgan Kaufmann Publishers, San Francisco, CA.

Baluja, S. and Davies, S. (1997). Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. Technical Report CMU-CS-97-107, Carnegie Mellon University.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Claredon-Press, Oxford.

Blickle, T. and Thiele, L. (1994). Genetic programming and redundancy. In Hopf, J., editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany. Max-Planck-Institut für Informatik (MPI-I-94-241).

Chaitin, G. (1969). On the length of programs for computing finite binary sequences: statistical considerations. *Journal of the ACM*, 16:145–159.

Chaitin, G. (1975). A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340.

Chaitin, G. (1987). *Algorithmic Information Theory.* Cambridge University Press, Cambridge.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 183–188. AAAI Press, San Jose, California.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

Cliff, D. and Ross, S. (1994). Adding temporary memory to ZCS. *Adaptive Behavior*, 3:101–150.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, Hillsdale NJ. Lawrence Erlbaum Associates.

Crites, R. and Barto, A. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA. MIT Press.

De Bonet, J. S., Isbell, Jr., C. L., and Viola, P. (1997). Mimic: Finding optima by estimating probability densities. In Jordan, M., Mozer, M., and Perrone, M., editors, *Advances in Neural Information Processing Systems*, volume 9, pages 424–430. MIT Press, Cambridge, MA.

Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1987). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

Drucker, H., Cortes, C., Jackel, L. D., LeCun, Y., and Vapnik, V. (1994). Boosting and other ensemble methods. *Neural Computation*, 6(6):1289–1301.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. San Mateo, CA: Morgan Kaufmann.

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence through Simulated Evolution.* Willey, New York.

Fogel, L. J. (1962). Autonomous automata. *Industrial Research*, 4:14–19.

Gács, P. (1974). On the symmetry of algorithmic information. *Soviet Math. Dokl.*, 15:1477–1480.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471.

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

Gruau, F. (1994). *Neural Networks Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Supérieure de Lyon.

Gruau, F. (1996). On using syntactic constraints with genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA.

Haynes, T. (1996). Duplication of coding segments in genetic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 344–349, Portland, OR.

Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München. See www7.informatik.tu-muenchen.de/˜hochreit.

Hochreiter, S. and Schmidhuber, J. (1997a). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hochreiter, S. and Schmidhuber, J. (1997b). LSTM can solve hard long time lag problems. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 473–479. MIT Press, Cambridge MA.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

Hutter, M. (2001). The fastest and shortest algorithm for all well-defined problems. Technical Report IDSIA-16-00, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Manno (Lugano), Switzerland. Submitted to the International Journal of Foundations of Computer Science.

Jaakkola, T., Singh, S. P., and Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, Cambridge MA. MIT Press.

Jordan, M. I. and Jacobs, R. A. (1992). Hierarchies of adaptive experts. In Moody, J., Hanson, S., and Lippmann, R., editors, *Advances in Neural Information Processing Systems - 4*, pages 985–993. Morgan Kaufmann, San Mateo, CA.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1995). Planning and acting in partially observable stochastic domains. Technical report, Brown University, Providence, RI.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11.

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

Langdon, W. B. (1995). Directed crossover within genetic programming. Research Note RN/95/71, University College London, Gower Street, London WC1E 6BT, UK.

Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

Levin, L. A. (1974). Laws of information (nongrowth) and aspects of the foundation of probability theory. *Problems of Information Transmission*, 10(3):206–210.

Levin, L. A. (1984). Randomness conservation inequalities: Information and independence in mathematical theories. *Information and Control*, 61:15–37.

Li, M. and Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, New York.

Lin, L. J. (1993). *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh.

Littman, M. (1994a). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J. A., and Wilson, S. W., editors, *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 3*, pages 297–305. MIT Press/Bradford Books.

Littman, M. L. (1994b). Markov games as a framework for multi-agent reinforcement learning. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 157–163, San Francisco, CA. Morgan Kaufmann Publishers.

Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. (1997). Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence (IJCAI-97).*

Marti, L. (1992). Genetically generated neural networks I: Representational effects. Technical Report CAS/CNS-TR-92-014, Boston University Center for Adaptive Systems.

Matsubara, H., Noda, I., and Hiraki, K. (1996). Learning of cooperative actions in multi-agent systems: a case study of pass play in soccer. In Sen, S., editor, *Working Notes for the AAAI-96 Spring Symposium on Adaptation, Coevolution and Learning in Multi-agent Systems*, pages 63–67, Menlo Park, CA. AAAI Press.

McCallum, R. A. (1996). Learning to use selective attention and short-term memory in sequential tasks. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324. MIT Press, Bradford Books.

McPhee, N. F. and Miller, J. D. (1995). Accurate replication in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA. Morgan Kaufmann.

Moore, A. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130.

Nadella, R. and Sen, S. (1996). Correlating internal parameters and external performance: learning soccer agents. In Weiss, G., editor, *Distributed Artificial Intelligence Meets Machine Learning. Learning in Multi-Agent Environments*, pages 137–150. Springer-Verlag, Berlin.

Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight sharing. *Neural Computation*, 4:173–193.

O'Reilly, U.-M. (1995). *An Analysis of Genetic Programming.* PhD thesis, Carleton University, Ottawa, Ontario.

Pearlmutter, B. A. (1995). Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228.

Peng, J. and Williams, R. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22:283–290.

Pringle, W. R. (1995). ESP: Evolutionary structured programming. Technical report, Penn State University, Great Valley Campus, PA, USA.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Royal Aircraft Establishment, Library translation No. 1122, Farnborough, Hants, UK.

Rechenberg, I. (1971). Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation. Published 1973 by Fromman-Holzboog.

Ring, M. B. (1995). *Continual Learning in Reinforcement Environments.* R. Oldenbourg Verlag, München, Wien.

Robinson, A. J. and Fallside, F. (1987). The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department.

Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming. In Angeline, P. and K. E. Kinnear, J., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–201. MIT Press, Cambridge, MA.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press.

Sahota, M. (1993). Real-time intelligent behaviour in dynamic environments: Soccer-playing robots. Master's thesis, University of British Columbia.

Sałustowicz, R. P. (1995). A genetic algorithm for the topological optimization of neural networks. Master's thesis, Technical University of Berlin, Germany.

Sałustowicz, R. P. and Schmidhuber, J. (1997a). Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141.

Sałustowicz, R. P. and Schmidhuber, J. (1997b). Probabilistic incremental program evolution: Stochastic search through program space. In van Someren, M. and Widmer, G., editors, *Machine Learning: ECML-97*, volume 1224 of *Lecture Notes in Artificial Intelligence*, pages 213–220. Springer-Verlag Berlin Heidelberg.

Sałustowicz, R. P. and Schmidhuber, J. (1998). Evolving structured programs with hierarchical instructions and skip nodes. In Shavlik, J., editor, *Machine Learning: Proceedings of the Fifteenth International Conference (ICML98)*, pages 488–496. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Sałustowicz, R. P. and Schmidhuber, J. (1999a). From probabilities to programs with probabilistic incremental program evolution. In Corne, D., Dorigo, M., and Glover, F., editors, *New Ideas in Optimization*, pages 433–450. McGraw-Hill, London.

Sałustowicz, R. P. and Schmidhuber, J. (1999b). Sequence learning through pipe and automatic task decomposition. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honovar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, pages 1184–1191. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. (1997a). Evolving soccer strategies. In Kasabov, N., Kozma, R., Ko, K., O'Shea, R., Coghill, G., and Gedeon, T., editors, *Progress in Connectionist-based Information Systems: Proceedings of the Fourth International Conference on Neural Information Processing ICONIP'97*, volume 1, pages 502–505. Springer-Verlag Singapore.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. (1997b). On learning soccer strategies. In Gerstner, W., Germond, A., Hasler, M., and Nicoud, J.-D., editors, *Proceedings of the Seventh International Conference on Artificial Neural Networks (ICANN'97)*, volume 1327 of *Lecture Notes in Computer Science*, pages 769–774. Springer-Verlag Berlin Heidelberg.

Sałustowicz, R. P., Wiering, M. A., and Schmidhuber, J. (1998). Learning team strategies: Soccer case studies. *Machine Learning*, 33:263–282.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5:197–227.

Schmidhuber, J. (1991). Reinforcement learning in Markovian and non-Markovian environments. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 500–506, San Mateo, CA. Morgan Kaufmann.

Schmidhuber, J. (1994). On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München. Revised January 1995.

Schmidhuber, J. (1995). Discovering solutions with low Kolmogorov complexity and high generalization capability. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 488–496, San Francisco, CA. Morgan Kaufmann Publishers.

Schmidhuber, J. (1997). Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873.

Schmidhuber, J. (1999). A general method for incremental self-improvement and multi-agent learning. In Yao, X., editor, *Evolutionary Computation: Theory and Applications*, pages 81–123. Scientific Publ. Co., Singapore.

Schmidhuber, J. and Hochreiter, S. (1996). Guessing can outperform many long time lag algorithms. Technical Report IDSIA-19-96, IDSIA.

Schmidhuber, J., Zhao, J., and Schraudolph, N. (1997a). Reinforcement learning with self-modifying policies. In Thrun, S. and Pratt, L., editors, *Learning to learn*, pages 293–309. Kluwer, Boston, MA.

Schmidhuber, J., Zhao, J., and Wiering, M. (1997b). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130.

Schwefel, H.-P. (1965). Kybernetische Evolution als Strategie der experimentellen Forschung aus der Strömungstechnik. Diplomarbeit, Technische Universität Berlin.

Schwefel, H.-P. (1974). Numerische Optimierung von Computer-Modellen. Dissertation. Published 1977 by Birkhäuser, Basel.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models.* John Wiley & Sons, Chichester.

Smith, A. W. and Zipser, D. (1989). Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems*, 1(2):125–131.

Solomonoff, R. (1964). A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22.

Solomonoff, R. (1986). An application of algorithmic probability to problems in artificial intelligence. In Kanal, L. N. and Lemmer, J. F., editors, *Uncertainty in Artificial Intelligence*, pages 473–491. Elsevier Science Publishers.

Spector, L. (1996). Simultaneous evolution of programs and their control structures. In Angeline, P. and K. E. Kinnear, J., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA.

Stone, P. and Veloso, M. (1996). Beating a defender in robotic soccer: Memory-based learning of a continuous function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 8*, pages 896–902, Cambridge MA. MIT Press.

Stone, P. and Veloso, M. (1998a). A layered approach to learning client behaviors in the robocup soccer server. In *Applied Artificial Intelligence (AAI)*, volume 12.

Stone, P. and Veloso, M. (1998b). Team-partitioned opaque-transition reinforcement learning. In Conference on automated learning and discovery (CONALD'98): Robot Exploration and Learning. Carnegie Mellon University, Pittsburgh.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Teller, A. (1994). The evolution of mental models. In Kenneth E. Kinnear, J., editor, *Advances in Genetic Programming*, pages 199–219. MIT Press.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

Versino, C. and Gambardella, L. M. (1997). Learning real team solutions. In Weiss, G., editor, *DAI Meets Machine Learning*, volume 1221 of *Lecture Notes in Artificial Intelligence*, pages 40–61. Springer-Verlag, Berlin.

Watkins, C. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge.

Weiss, G. (1996). Adaptation and learning in multi-agent systems: Some remarks and a bibliography. In Weiss, G. and Sen, S., editors, *Adaptation and Learning in Multi-Agent Systems*, volume 1042 of *Lecture Notes in Artificial Intelligence*, pages 1–21. Springer-Verlag, Berlin Heidelberg.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1.

Whigham, P. A. (1995). Grammatically-based genetic programming. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA.

Whitehead, S. and Ballard, D. H. (1990). Active perception and reinforcement learning. *Neural Computation*, 2(4):409–419.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. *1960 IRE WESCON Convention Record*, 4:96–104. New York: IRE. Reprinted in Anderson and Rosenfeld [1988].

Wiering, M. A. (1999). *Explorations in Efficient Reinforcement Learning.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Wiering, M. A., Sałustowicz, R. P., and Schmidhuber, J. (1998). Cmac models learn to play soccer. In Niklasson, L., Boden, M., and Ziemke, T., editors, *Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN'98)*, volume 1, pages 443–448. Springer-Verlag London.

Wiering, M. A., Sałustowicz, R. P., and Schmidhuber, J. (1999). Reinforcement learning soccer teams with incomplete world models. *Journal of Autonomous Robots*, 7(1):77–88.

Wiering, M. A., Sałustowicz, R. P., and Schmidhuber, J. (to appear). Model-based reinforcement learning for evolving soccer strategies. In *Soft Computing Techniques in Game Playing.* Springer-Verlag.

Wiering, M. A. and Schmidhuber, J. (1996a). HQ-Learning: Discovering Markovian subgoals for non-Markovian reinforcement learning. Technical Report IDSIA-96-96, IDSIA, Lugano, Switzerland.

Wiering, M. A. and Schmidhuber, J. (1996b). Solving POMDPs with Levin search and EIRA. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 534–542. Morgan Kaufmann Publishers, San Francisco, CA.

Wiering, M. A. and Schmidhuber, J. (1997). Fast online Q(λ). Technical Report IDSIA-21-97, IDSIA, Lugano, Switzerland.

Wiering, M. A. and Schmidhuber, J. (1998). Efficient model-based exploration. In Meyer, J. A. and Wilson, S. W., editors, *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 6*, pages 223–228. MIT Press/Bradford Books.

Williams, R. J. (1989). Complexity of exact gradient computation algorithms for recurrent neural networks. Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science.

Williams, R. J. and Zipser, D. (1992). Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications.* Hillsdale, NJ: Erlbaum.

Wineberg, M. and Oppacher, F. (1996). The benefits of computing with introns. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 410–415, Stanford University, CA, USA. MIT Press.

Wong, M. L. and Leung, K. S. (1996). Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA.

Yanagiya, M. (1993). A simple mutation-dependent genetic algorithm. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 659. Morgan Kaufmann, San Mateo, CA.

Yao, X. (1999). Introduction. In Yao, X., editor, *Evolutionary Computation: Theory and Applications*, pages 1–36. Scientific Publ. Co., Singapore.

Zannoni, E. and Reynolds, R. G. (1997). Learning to control the program evolution process with cultural algorithms. *Evolutionary Computation*, 5(2):181–211.

Zhao, J. and Schmidhuber, J. (1996). Incremental self-improvement for life-time multi-agent reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pages 516–525. MIT Press, Bradford Books.

Index

Acknowledgments

I would like to thank all the many persons, who supported me in creating this thesis, and who made my life joyful as it was.

First of all I would like to thank my beloved wife Małgosia for being a wonderful person, for caring, and for supporting me throughout this entire endeavor. Especially during hard working periods, she was always ready to sacrifice her personal pleasures, and push me to reach my goals.

Special thanks also to my parents for always supporting me, and thus making all of this possible in the first place, and to the rest of my family, who never doubted that it will come to an good end.

Jürgen Schmidhuber, my supervisor at IDSIA, did a lot to help me create this thesis. He taught me many valuable things about research, and especially machine learning. He put me through an excellent studying time challenging me, and pushing me to the limits, while at the same time giving me the freedom to do what really interested me. Thanks Jürgen!

I am also very thankful to Erhard Konrad, my supervisor at Technical University of Berlin, for his excellent scientific support, and his patience always giving me enough time and space to pursue my way.

Marco Wiering deserves special mentioning. It has been a great pleasure studying with him, discussing scientific issues, God, the world, and everything, partying with him, and having fun. Marco contributed much to both the scientific, and the fun part during those years.

Thanks to Nicol Schraudolph for excellent scientific discussions, and comments on my papers, as well as for his friendliness, helpfulness, originality – his friendship, and for sharing magnificent views onto research, life, and Lugano.

Thanks also to Felix Gers and Mara Kugler for all the fun we had together, for their kindness, their warm welcomes, their grand hospitality – never letting me down whenever I needed some place to finish my dissertation, and for the best parties I ever had. I am very happy to call them my friends.

Marcus Hoffmann made my life a little brighter every day. He was always in a good mood, provided me with interesting information from Internet, organized great happenings, and made the best coffee I ever had.

Monica Bancalà gave IDSIA the spirit. She took care of all the non-research matters from organization to administration in her very kind, cheerful, and friendly way, always ready to help. Without exaggerating, I can say she did a lot for making my stay at IDSIA a great pleasure. It would not have been the same without her.

With Jieyu Zhao you could always have an interesting chat about scientific and non-scientific issues. I very much enjoyed, and was often inspired by our conversations.

I would also like to thank Luca Gambardella, Cristina Versino, Marco Dorigo, and Richard Sutton for valuable discussions, comments, and suggestions on my scientific work.

Thanks to Andrea Rizzoli for his great sense of humor bringing sunshine even to rainy days, Eric Taillard for his delicious fondues, and Martin Eldracher for early insights into a PHD student's life. Thanks also to Paolo Cattaneo and Raffaello Giulietti for their excellent, and extremely responsive support as system administrators, and to Carlo Lepori, who took care of all the (sometimes quite challenging) official matters.

Of course, there were also many other persons not mentioned here, who supported the creation of this thesis, and made my life interesting, very joyful, and very much worth living. Many thanks to all of you!

Zusammenfassung

Das zentrale Thema der Dissertation ist “Probabilistic Incremental Program Evolution” (PIPE). PIPE ist ein neuer, evolutionärer Algorithmus, der stochastische Modelle verwendet um Computerprogramme zu finden, die eine Lösung zu gegebenen Problemen darstellen.

Insbesondere Probleme mit Regularitäten in ihren Lösungen sind für die Programmsuche interessant. Regularitäten ermöglichen kurze algorithmische Lösungsbeschreibungen. Kürzere Beschreibungen werden im Allgemeinen schneller gefunden. Programmsuche kann daher effizient sein, wenn die Abbildung des Lösungsraumes in den Programmraum den Suchraum verkleinert. Der Programmraum ist jedoch normalerweise ein diskontinuierlicher Raum. Gradientenabstiegsbasierte Optimierungsverfahren sind daher für die Programmsuche im Allgemeinen nicht anwendbar. Übrig bleiben verschiedene zufallsbasierte Verfahren, unter anderem auch evolutionäre Algorithmen.

Das Ziel dieser Arbeit ist es PIPE vorzustellen und Methoden zu definieren, die PIPE auf ein breites Spektrum von Problemen anwendbar machen.

Zuerst präsentieren wir PIPE und zeigen, dass PIPE für verschiedene Anwendungen eingesetzt werden kann, unter anderem auch für komplexe Anwendungen, wie z.B. das Lernen in Multiagentensystemen. Dann erhöhen wir mittels strukturierter Programme, wo die Programminstruktionsabfolge zum Teil fest vorgegeben ist, PIPE's Leistungsfähigkeiten. Programme ohne internen Speicher können keine Probleme lösen, die der Markov Eigenschaft nicht genügen, d.h. deren Output nicht nur vom Input abhängt, sondern auch vom zeitlichen Kontext des Inputs. Um das Anwendungsgebiet von PIPE zu erweitern, zeigen wir, wie PIPE Programme mit internem Speicher finden kann. Dabei scheint PIPE für Probleme mit sehr langen Zeitspannen zwischen relevanten Inputs und ihren korrespondierenden Outputs besonders gut geeignet zu sein. Mit der Lösung von hochkomplexen Aufgaben, d.h. wenn z.B. viele Datenabhängigkeiten in Programmen abgebildet werden müssen, kann der PIPE Algorithmus überfordert werden. Um PIPE auch für solche Probleme konkurrenzfähiger zu machen, haben wir *filtering* entwickelt. Filtering ist ein optimierungsalgorithmusunabhängiges, automatisches Aufgabenteilungsverfahren. Es teilt nicht nur die eigentliche Aufgabe in weniger komplexe Teilaufgaben, sondern zerlegt auch das Problem des Zusammenführens der Teillösungen in Teilaufgaben.