ON LEARNING HOW TO LEARN LEARNING STRATEGIES
Technical Report FKI-198-94 (revised)

Jürgen Schmidhuber
Fakultät für Informatik
Technische Universität München
80290 München, Germany

schmidhu@informatik.tu-muenchen.de
http://papa.informatik.tu-muenchen.de/mitarbeiter/schmidhu.html

Revised January 31, 1995

Abstract

This paper introduces the "incremental self-improvement paradigm". Unlike previous methods, incremental self-improvement encourages a reinforcement learning system to improve the way it learns, and to improve the way it improves the way it learns ..., without significant theoretical limitations — the system is able to "shift its inductive bias" in a universal way. Its major features are: (1) There is no explicit difference between "learning", "meta-learning", and other kinds of information processing. Using a Turing machine equivalent programming language, the system itself occasionally executes self-delimiting, initially highly random "self-modification programs" which modify the context-dependent probabilities of future action sequences (including future self-modification programs). (2) The system keeps only those probability modifications computed by "useful" self-modification programs: those which bring about more payoff (reward, reinforcement) *per time* than all previous self-modification programs. (3) The computation of payoff per time takes into account all the computation time required for learning — the *entire* system life is considered: boundaries between learning trials are ignored (if there are any). A particular implementation based on the novel paradigm is presented. It is designed to exploit what conventional digital machines are good at: fast storage addressing, arithmetic operations etc. Experiments illustrate the system's mode of operation.

Keywords: *Self-improvement, self-reference, introspection, machine-learning, reinforcement learning.*
Note: *This is the revised and extended version of an earlier report from November 24, 1994.*

1 Introduction

A recent debate in the machine-learning community highlighted a fact that appears discouraging at first glance: in general, generalization cannot be expected, inductive inference is impossible, and nothing can be learned. See, e.g., (Dietterich, 1989; Schaffer, 1993; Wolpert, 1993; Schmidhuber, 1994). Paraphrasing from a previous argument (Schmidhuber, 1994): let the task be to learn some relation between finite bitstrings and finite bitstrings. Somehow, a training set is chosen. In almost all cases, the shortest algorithm computing a (non-overlapping) test set essentially has the same size as the whole test set. This is because most computable objects are irregular and incompressible (Kolmogorov, 1965; Chaitin, 1969). The shortest algorithm computing the test set, given the training set, isn't any shorter. In other words, the relative algorithmic complexity of the test set, given the training set, is maximal, and the mutual algorithmic information between test set and training set is zero (ignoring an additive constant independent of the problem — see e.g. Kolmogorov, 1965; Chaitin, 1969; Solomonoff, 1964; Li and Vitányi, 1993). Therefore, in almost all cases, (1) knowledge of the training set does not provide any clues about the test set, (2) there is no hope for generalization, and (3) inductive inference does not make any sense.

Atypical real world / Previous learning algorithms. Apparently, however, generalization and inductive inference *do* make sense in the real world! One reason for this may be that the real world is run by a short algorithm. See (Schmidhuber, 1994). Anyway, problems that humans consider to be *typical* are *atypical* when compared to the general set of all well-defined problems. Otherwise, things like "learning by analogy", "learning by chunking", "incremental learning", "continual learning", "learning from invariances", "learning by knowledge transfer" etc. would not be possible, and experience with previous problems could not sensibly adjust the prior distribution of solution candidates in the search space for a new problem (shift of inductive bias, e.g. Utgoff, 1986). In fact, *all* previous learning systems are implicitly or explicitly designed to exploit task-specific regularities of some kind or another.

No previous learning system, however, is designed to make optimal use of its computational time/space resources, by exploiting *arbitrary*, task-specific regularities (if there are any). Such a system would have to be able (1) to develop *arbitrary* problem-specific representations, (2) to run *arbitrary* learning algorithms, and (3) to find the "good", problem-specific learning algorithms, as quickly as possible. In particular, it would have to be able to find algorithms for finding learning algorithms etc.

Self-improvement. Is it possible to build such a system? A system that can tailor its learning behavior to the requirements of a given environment with arbitrary, initially unknown, problem-specific regularities? A system that can learn to improve its own learning strategy in a universal way, without any significant theoretical limitations other than those imposed by the finiteness of the hardware? In principle, the answer is yes. The system described in this paper uses the novel *"incremental self-improvement paradigm"* to exploit "benign" environments in a more general way than previous systems. Some of its properties are: (1) Unlike e.g. hillclimbing/evolutionary/genetic/other algorithms, it potentially can evolve its own "smart" search strategies (as opposed to "dumb", non-adaptive strategies like the ones embodied by random mutation, "crossover" etc.). (2) Unlike with previous, less realistic approaches, each event in system life is viewed as a singular event — learning is inductive inference from non-repeatable experiences. (3) Unlike with previous approaches, the system's objective function takes into account the computation time required for learning.

Outline. Section 2 lists essential ingredients of the incremental self-improvement paradigm. Section 3 exemplifies the basic principles, by describing and justifying a concrete (working) implementation. Twenty comments on both general and implementation-specific properties of incremental self-improvement can be found in section 4. Illustrative applications to toy problems (including a simple "non-Markovian" maze task) will follow in section 5. Section 6 will then briefly describe the history of related ideas.

2 The Incremental Self-Improvement Paradigm

Incremental self-improvement is a machine-learning paradigm designed for a system executing a lifelong sequence of actions in an arbitrary environment. The system's goal is to maximize cumulative payoff (reinforcement, reward) to be obtained throughout its entire span of life. To achieve its goal, the system continually attempts to create action subsequences leading to faster and faster payoff intake. The central ideas are as follows:

1. **Computing self-modifications.** The initially highly random actions of the system actually are primitive instructions of a Turing machine equivalent programming language, which allows for implementing arbitrary learning algorithms. Action subsequences represent either (1) "normal" interactions with the environment, or (2) "self-modification programs". Self-modification programs can arbitrarily[1] modify the probabilities of future action subsequences, including future self-modification programs: the learning system is able to modify itself in a universal way. There is no explicit difference between "learning", "meta-learning", and other kinds of information processing.

2. **Life is one-way.** Each action of the learning system (including probability modifying actions executed by self-modification programs) is viewed as a singular event in the history of system life. Unrealistic concepts such as "exactly repeatable training iterations", "boundaries between trials", "epochs", etc. are thrown overboard. In general, the environment cannot be reset. Life is one-way. Learning is inductive inference from non-repeatable experiences.

3. **Evaluations of self-modification programs.** The system has a time-varying utility value, which is the average payoff per time since system start-up. Each self-modification program also has a time-varying utility value. This value is the average amount of payoff per time measured since the program began execution. Evaluations of utility take into account all the computation time required for learning, including the time required for evaluating utility.

4. **Useful self-modification programs accelerate payoff intake.** The system keeps track of probability modifications computed by self-modification programs that it considers *useful*. *Usefulness* is defined recursively. If there are no previous *useful* self-modification programs (e.g. at system start-up), a new self-modification program is considered *useful* only for as long as its utility value exceeds the system's utility value. More recent self-modification programs are considered *useful* for as long as they have higher utility values than all preceding self-modification programs currently considered *useful*.

   Essentially, the system only keeps modifications to its probability values that originated from *useful* self-modification programs. The result is that payoff intake is constantly accelerated. Over time, the system tends to make better and better use of its computational resources.

3 A Concrete, Working Implementation

This section presents one of many possible implementations of the incremental self-improvement paradigm. The implementation makes use of an integer-based programming language. The language is assembler-like and has primitive instructions designed to exploit what conventional digital machines are good at: fast storage addressing, jumping, basic arithmetic operations, etc. The language is universal (i.e., Turing machine equivalent). It is related to one previously published (Schmidhuber, 1994), but there are significant differences and extensions. In particular, this language is "self-referential" in a manner that will be described below.

[1] Throughout this paper, when referring to "arbitrary" modifications, functions, etc., there is only one essential requirement of these modifications or functions: they must be computable.

3.1 Overview

The system has a finite amount of addressable storage broken into two groups: *work cells* and *program cells*. The system exists in an *environment*, that occasionally writes inputs into certain work cells. No assumptions are made about the environment (it may or may not be non-deterministic, for example). There is a fixed set I of integer instruction values and integer argument values. There is also an internal variable (not stored in an addressable cell) called the *Instruction Pointer* (*IP*), which always points to one of the program cells (initially to the first one).

For each program cell and for every possible instruction and instruction argument, there is a probability value P_{ij}, where i is the index of a program cell, and $j \in I$. P_{ij} specifies for cell i the conditional probability that, when pointed to by *IP*, its contents will be j. If $IP = i$ and the contents of i is interpreted as an instruction, then the cells that immediately follow i will be interpreted as the instruction's arguments. If the instruction and its arguments pass a syntax check, the instruction is executed. This may result in modifications of *IP* and/or environment and/or storage. If the test is not passed, the *IP* is reset to the first program cell. If the instruction does not itself modify *IP* (with a "jump" instruction), then *IP* is incremented to point to the following cell. This *instruction cycle* is repeated over and over again and represents the basic operation of the system.

Some instructions are special *"self-referential"* instructions. Two of them can address and modify arbitrary P_{ij} values (and they are the only instructions that can do this). There is also another self-referential instruction that groups a sequence of probability-modifying instructions and other instructions into a *self-delimiting self-modification program* — so named because the end of such a program is computed by itself. Some of the (initially highly random) action subsequences executed during system life will indeed be self-delimiting self-modification programs. They can compute arbitrary sequences of modifications of P_{ij} values, resulting in arbitrary modifications of context-dependent probabilities of future action subsequences, including future self-modification programs.

Occasionally the environment may provide "payoff" — a real number indicating how well the system has done. The system's goal is to maximize the sum of all payoffs to be obtained throughout its entire (initially unknown) life span. This is done as follows:

The payoff/time ratio is constantly monitored and updated after every instruction. There is an unmodifiable *top-level credit assignment strategy*. It also executes after each instruction except when a self-modification program is running. The task of the top-level is to countermand the modifications to the P_{ij} values that were made by self-modification programs no longer considered *useful* (see recursive description in section 2). This countermanding is done by restoring "old" probability distributions saved onto a stack before the corresponding self-modification programs modified them. The computation time required for top-level management is taken into account when measuring *usefulness*.

It will be seen that this scheme favors sequences of *useful* self-modification programs leading to faster and faster payoff intake. In particular, self-modification programs can prove their long term usefulness by setting the stage for additional, useful self-modification programs, which potentially include programs executing known (and not yet known) learning algorithms. This encourages "learning how to learn".

3.2 Technical Details

Span of system life. For simplicity, we assume discrete time. System life begins at "birth," time step zero. It ends at "death," time step T. T is not necessarily known in advance.

Goal / Payoff. Occasionally, the environment provides "payoff". Payoff is an integer number depending on the tasks to be solved. The sum of *all* payoffs obtained between birth and time $t > 0$ is denoted by $R(t)$. Throughout its lifetime, the system's goal is to maximize $R(T)$, the cumulative payoff at "death". At a given time, the system can only maximize future payoff — the past is already gone.

Storage. The system's *storage* is a single array of cells. Each cell has an integer address in the interval $[Min, Max]$. Max is a positive integer. Min is a negative integer. a_i denotes the cell with address i. The variable contents of a_i are denoted by $c_i \in [-Maxint, Maxint]$, and are of type integer as well ($Maxint \geq Max$; $Maxint \geq abs(Min)$). Special addresses, $InputStart$, $InputEnd$, $RegisterStart$, and

Primitive	Semantics
Stop()	Halt current run
Jmp(a1)	$IP \leftarrow c_{a1}$
Jmpleq(a1, a2, a3)	If $c_{c_{a1}} < c_{c_{a2}}$ $IP \leftarrow c_{a3}$
Jmpeq(a1, a2, a3)	If $c_{c_{a1}} = c_{c_{a2}}$ $IP \leftarrow c_{a3}$
Add(a1, a2, a3)	$c_{c_{a3}} \leftarrow c_{c_{a1}} + c_{c_{a2}}$
Sub(a1, a2, a3)	$c_{c_{a3}} \leftarrow c_{c_{a1}} - c_{c_{a2}}$
Mul(a1, a2, a3)	$c_{c_{a3}} \leftarrow c_{c_{a1}} * c_{c_{a2}}$
Div(a1, a2, a3)	$c_{c_{a3}} \leftarrow c_{c_{a1}}/c_{c_{a2}}$ (integer division)
Rem(a1, a2, a3)	$c_{c_{a3}} \leftarrow$ remainder$(c_{c_{a1}}/c_{c_{a2}})$
Inc(a1)	$c_{c_{a1}} \leftarrow c_{c_{a1}} + 1$
Dec(a1)	$c_{c_{a1}} \leftarrow c_{c_{a1}} - 1$
Mov(a1, a2)	$c_{c_{a2}} \leftarrow c_{c_{a1}}$
Init(a1, a2)	$c_{a1-ProgramStart-2} \leftarrow a2$
Output$_i$(...)	i-th problem specific primitive for influencing the environment
Input$_i$(...)	i-th primitive for perceiving environmental input
GetP(a1, a2, a3)	$c_{c_{a3}} \leftarrow round(Maxint * P_{c_{a1},c_{a2}})$
IncP(a1, a2, a3)	$\forall k \neq c_{a2} : P_{c_{a1},k} \leftarrow 0.01 c_{c_{a3}} P_{c_{a1},k}$; $P_{c_{a1},c_{a2}} \leftarrow 1 - 0.01 c_{c_{a3}}(1 - P_{c_{a1},c_{a2}})$
DecP(a1, a2, a3)	$\forall k \neq c_{a2} : P_{c_{a1},k} \leftarrow \frac{1-0.01 c_{c_{a3}} P_{c_{a1},c_{a2}}}{1-P_{c_{a1},c_{a2}}} P_{c_{a1},k}$; $P_{c_{a1},c_{a2}} \leftarrow 0.01 c_{c_{a3}} P_{c_{a1},c_{a2}}$
EndSelfMod()	Enable top level strategy for restoring probabilities (see text)

Table 1: *Semantics of primitives and their parameters. The "normal" primitives are shown in the top block; the "self-referential" primitives are shown in the bottom block. Note the extensive use of double-indexed indirect addressing. Results of arithmetic operations leading to underflow/overflow are replaced by $-Maxint/Maxint$, respectively. The same holds for positive and negative divisions by zero.* DecP *and* IncP *have no effect if the indirectly addressed cell contents $c_{c_{a3}}$ are not an integer between 1 and 99, or if the corresponding probability modification would lead to at least one P value below* MinP. **Rules for syntactic correctness:** *IP may point to any program cell a_i, $i < Max - 3$ (enough space has to be left for arguments). Operations that read cell contents (such as* Add, Move, Jumpleq *etc.) may read only from existing addresses in storage. Operations that write cell contents (such as* Add, Move, GetP *etc.) may write only into work area addresses in $[Min, ProgramStart - 1]$.*

ProgramStart, are used to further divide storage into segments: $Min < InputStart \leq InputEnd < 0 = RegisterStart < ProgramStart < Max$. The *input area* is the set of *input cells* $\{a_i : InputStart \leq i \leq InputEnd\}$. The *register area* is the set of *register cells* $\{a_i : 0 \leq i < ProgramStart\}$. "Registers" are convenient for indirect-addressing purposes. The *program area* is the set of *program cells* $\{a_i : ProgramStart \leq i < Max\}$. Integer sequences in the program area are interpreted as executable code. The *work area* is the set of *work cells* $\{a_i : Min \leq i < ProgramStart\}$. Instructions executed in the program area may read from and write to the work area. Both register area and input area are subsets of the work area.

Environmental inputs. At every time step, new inputs from the environment may be written into the input cells.

Primitives. The number of instructions is n_{ops} ($n_{ops} << Maxint$). Each such "primitive" is represented by a unique number in the set $\{0, 1, \ldots, n_{ops} - 1\}$ (due to the code being written in C). The primitive with number j is denoted by p_j. Primitives may have from zero to three arguments, each of which has a value in $\{0, 1, \ldots, n_{ops} - 1\}$. The semantics of the primitives and their corresponding arguments are given in Table 1. The non-self-referential ("normal") primitives include actions for comparisons, and conditional jumps, for copying storage contents, for initializing certain storage cells with

small integers, and for adding, multiplying, dividing, and halting. They also include output actions for modifying the environment, and input actions for perceiving environmental states. The "self-referential" primitives will be described in detail below.

Primitive and argument probabilities. For each cell a_i in the program area, there is a discrete probability distribution P_i over the set of possible cell contents. The variable *InstructionPointer (IP)* always points to one of the program cells. If $IP = i$ and $i < Max - 3$, then P_{ij} denotes the probability of selecting primitive p_j as the next instruction. The restriction $i < Max - 3$ is needed to leave room for the instruction's possible arguments should it require any. Once p_j is selected: $c_i \leftarrow j$. If p_j has a first argument, then $P_{i+1,k}$ is the probability of k being chosen as its actual value, for $k \in \{0, 1, \ldots, n_{ops} - 1\}$. Once some k is selected: $c_{i+1} \leftarrow k$. Analoguously, if p_j has a second argument, then $P_{i+2,l}$ is the probability of l being chosen as its actual value, for $l \in \{0, 1, \ldots, n_{ops} - 1\}$. Once some l is selected: $c_{i+2} \leftarrow l$. And finally, if p_j has a third argument, then $P_{i+3,m}$ is the probability of m being chosen as its actual value, for $m \in \{0, 1, \ldots, n_{ops} - 1\}$. Once some m is selected: $c_{i+3} \leftarrow m$.

Arguments point to storage addresses. To reduce the number of probability values for each program cell, primitive arguments are restricted to only n_{ops} different values. Therefore, to allow all storage cells to be addressed, **double indexed indirect addressing** is used for most instructions. That is, (for most instructions) the arguments point to cells in the register area, which in turn point to cells in storage. Recall that the range of values available to registers (and other work cells) is far less restricted, which effectively allows all storage cells to be addressed.

Self-referential primitives. Two special primitives, *DecP* and *IncP*, may be used to address and modify the current probability distribution of any program cell (see Table 1). With the action *DecP*, the P_{ij} value for a particular cell/value pair (a_i, j) can be decreased by some factor in {0.01, 0.02, ..., 0.99}. The probabilities for that cell are then normalized. Likewise, with the action *IncP*, the *complement* $(1 - P_{ij})$ of the P_{ij} value for a particular cell a_i and value j can be decreased by a factor in $\{0.01, 0.02, \ldots, 0.99\}$ (and the cell probabilities are again renormalized). *DecP* and *IncP* have no effect if the indirectly addressed cell contents $c_{c_{a_3}}$ (see Table 1) are not an integer between 1 and 99, or if the corresponding probability modification would lead to at least one P value below *MinP* (a small positive constant).

The primitive *GetP* can be used to write scaled versions of current probability values into work cells. *GetP* is potentially useful for purposes of introspection.

Instruction cycle. A single step of the *interpreter* works as follows: if *IP* points to program cell a_i, a primitive and the corresponding arguments are chosen randomly according to the current probability distributions, as already described. They are sequentially written onto the program area, starting from a_i. Syntax checks are performed. Rules for syntactic correctness are given in the caption of Table 1. If syntactically correct, the instruction gets executed. Otherwise, the current "run" (see next paragraph) is halted. If the program did not halt nor change the value of *IP* (e.g. by causing a jump), *IP* is set to the address of the cell following the last argument of the current instruction.

Runs. In the beginning of a "run", *IP* is set equal to *ProgramStart*, and the instruction cycle is repeated until a halt situation (e.g. syntax error) is encountered. Due to *MinP* being positive, there is always a non-vanishing halting probability.

System life. At time step 0, storage is initialized with zeros. The probability distributions of all program cells are initialized with maximum entropy distributions (Shannon, 1948). That is, all P_{ij} values are initialized to the same value, so that there is no bias for a particular value in any cell. After initialization, **runs** are repeated over and over again until time T. Recall that T does not have to be known in advance.

Work area as part of the environment. Neither storage nor environment are re-initialized after each run. The system might use the environment to store representations of previous events, by executing actions that modify the environment[2]. Likewise, the program area may use the work area to store representations of previous events. Thus, the work area may be viewed as part of the total environment of the program area.

[2]Leslie Kaelbling sometimes refers to this as *"writing on the walls"* but says that the "real" name is *"stigmergy"* (personal communication, 1994/1995).

Programs. Each subsequence of primitives executed during system life is called a *program*. Primitives can be combined to form programs for performing arbitrary computations. The only limitations are those imposed by the necessarily finite hardware.

Self-delimiting self-modification programs. Occasionally, the system will modify one of its probability distributions, by using $IncP$ or $DecP$. Occasionally, it will execute the *EndSelfMod* primitive. The first probability modification after an *EndSelfMod* action or after system "birth" begins a self-modification program. The self-modification program ends itself by executing the *EndSelfMod* action. Due to the universality of the underlying programming language, self-modification programs may result in specific, arbitrary modifications of context-dependent probabilities of future programs. However, due to $MinP$ being positive, the probability of selecting and executing a particular instruction at a particular time cannot entirely vanish.

The remainder of this section is devoted to basic concepts required to ensure that the system keeps only probability modifications computed by "useful" self-modification programs: essentially those which bring about more payoff *per time* than all previous self-modification programs.

Payoff/time ratios. Suppose a self-modification program s started execution at time t_1 and completed itself at time t_2. For $t \geq t_2$ and $t \leq T$, the payoff/time ratio $Q(s,t)$ is defined as

$$Q(s,t) = \frac{R(t) - R(t_1)}{t - t_1}.$$

DEFINITION: useful self-modification programs. The *usefulness* of a self-modification program is defined recursively: at birth, there are no *useful* self-modification programs. At some later point t in system life, we consider two cases: a self-modification program s that ended itself at time t_2 is considered *useful* if

(1) (a) there are no previous self-modification programs that are considered *useful*, and

(b) for all $t_x \geq t_2, t_x \leq t$: $Q(s,t_x) > \frac{R(t_x)}{t_x}$ (the total payoff/time ratio at time t_x). Or

(2) (a) there *are* previous self-modification programs considered *useful*, and

(b) for all $t_x \geq t_2, t_x \leq t$: $Q(s,t_x) > Q(s',t_x)$, where s' is the most recent *useful* self-modification program preceding s.

The computation of payoff/time ratios always takes into account all computation time, including time required for learning.

> **FACT 1.** At a given time, to decide whether the most recent useful self-modification program remains useful, one needs to compare its current payoff/time ratio only to the current payoff/time ratio of the most recent *previous* useful self-modification program.
>
> **FACT 2.** A completed self-modification program is considered useful as long as average payoff intake since its beginning occurred faster than with *all* previous self-modification programs still considered useful.
>
> **Proof.** See definition of usefulness above.

Keeping track of useful self-modifications. In order to evaluate whether life after the beginning of some self-modification program was "better" or "worse" than before, and to be able to countermand probability modifications computed by self-modification programs that seem useless or even harmful, a stack, S, is introduced. The k-th entry of S, $k \in \{0, 1, \ldots, StackSize\}$, is denoted $S(k)$. $S(k)$ consists of the following variables: $S(k).t$, $S(k).R$, $S(k).address$, $S(k).first$, and $S(k).oldP$ (a vector of n_{ops} variables). The variable sp points to the current topmost stack entry. If $sp = 0$ (the value at system startup), the stack is empty. If the system modifies some probability distribution P_i at time

t (by using $IncP$ or $DecP$), sp is incremented, and the following values are pushed onto the stack: the current time, the total payoff so far, the address of the modified program cell, its old probability distribution right before the current modification (represented by n_{ops} real values), and a pointer to the stack entry corresponding to the first probability modification computed by the current self-modification program. More formally: $S(sp).t \leftarrow t$; $S(sp).R \leftarrow R(t)$; $S(sp).address \leftarrow i$; $S(sp).oldP \leftarrow P_i$ *before* the modification (represented by n_{ops} real values). If t marks the beginning of a self-modification program, then $S(sp).first \leftarrow sp$. Otherwise (in the middle of a self-modification program), $S(sp).first \leftarrow S(sp-1).first$. The zeroth stack entry, which cannot be popped, is initialized as follows: $S(0).t \leftarrow 0$; $S(0).R \leftarrow 0$; $S(0).first \leftarrow 0$. The remaining values are undefined.

UNMODIFIABLE TOP-LEVEL CREDIT ASSIGNMENT STRATEGY: inductive inference based on single experiences. It is very simple. If there is no self-modification program running, then after each completed instruction cycle, a decision is made as to whether probability distributions modified by previous self-modification programs should be kept or replaced by their old values from the stack. The process works as follows. After each completed instruction cycle, do:

(1) If there currently is an incomplete self-modification program (not yet ended by an *EndSelfMod* action), do nothing.

(2) Else successively pop and restore saved probability distributions, until the payoff/time ratio since the beginning of the most recent self-modification program that computed modifications of probability distributions currently saved on the stack exceeds the payoff/time ratio of the preceding such self-modification program (if there is any – otherwise compare with the total payoff/time ratio). More formally (t denotes the current time):

While $sp \neq 0$ **and**

$$\frac{R(t) - S(S(sp).first).R}{t - S(S(sp).first).t} \leq \frac{R(t) - S(S(S(sp).first - 1).first).R}{t - S(S(S(sp).first - 1).first).t}$$

do: $P_{S(sp).i} \leftarrow S(sp).oldP$; $sp \leftarrow sp - 1$.

FACT 1 above says that at a given time, the top-level strategy needs to consider only the two most recent self-modification programs whose direct effects have not yet been countermanded, in order to decide whether to pop the stack. **Then why do we need a while loop as above?** The reason is that popping and restoring probability distributions takes time (t increases during execution of the while loop), possibly causing utility values to drop. Therefore, in the process of popping and restoring distributions modified by one program, the payoff/time ratio of the preceding program may fall enough so that the distributions it modified must be restored, too. The process can potentially continue until the stack is completely empty.

FACT 3. After each instruction (except during the execution of a self-modification program), the top-level ensures that the beginning of each completed self-modification program that computed valid probability modifications has been followed by *faster* payoff intake than the beginnings of all previous such self-modification programs. *All currently valid probability modifications were computed by currently useful self-modification programs.* The nature of the environment does not matter.

Proof. See formal top-level description and FACT 2.

4 Twenty Comments

The experiments are described in section 5. If you are in a hurry, you can skip these (mostly rather obvious) comments.

1. **Why self-delimiting self-modification programs?** The *EndSelfMod* primitive allows the system to delay top-level evaluations of probability modifications arbitrarily. The expectation of the delay remains finite, however, due to $MinP$ being positive. The system's delaying capabilities are important, for two reasons: (1) In general, payoff events will be separated by long (unknown) time lags. Hence, novel probability modifications are not necessarily bad if they do not lead to immediate payoff. The system itself should be able to learn how much time to spend on waiting for first payoff events. (2) Two successive modifications of two particular probability distributions may turn out to be beneficial, while each by itself may be harmful. Therefore, the system should be able to compute arbitrary sequences of probability modifications, before facing top-level evaluations.

   Delaying top-level evaluations *does* cost time, though, which is taken into account when usefulness is measured. In the long run, the system is encouraged to create useful self-modification programs of the appropriate size.

2. **Non-decreasing search space.** Due to $MinP$ being positive, there will always be a non-vanishing (possibly tiny) probability of executing *any* program at *any* time. Thus, the space of possible action subsequences will never really decrease. Only the probability distribution on this space (the bias) can change. But there cannot be *total* determinism corresponding to total lack of exploration.

3. **Speeding up payoff intake / Learning how to learn.** The top-level takes the entire learning history into account: note that at time t, the value $t - S(S(sp).first).t$ stands for *all* the time since the beginning of the most recent self-modification program whose effects have not yet been countermanded. The utility value of a self-modification program is based on *total* elapsed time since the program began. This includes the computation time required for learning. Over time, the system tends to make better and better use of its limited temporal and spatial resources: due to FACT 3, self-modifications that speed up payoff intake in the long run are preferred. So are self-modifications speeding up the search for self-modifications speeding up payoff intake. This encourages "learning how to learn", and "learning how to learn how to learn"..., and represents an essential difference to previous approaches to continual learning, see (Ring, 1994).

4. **Directed mutations as opposed to random mutations.** Unlike evolutionary and genetic algorithms (Rechenberg, 1971; Schwefel, 1974; Holland, 1975; Hoffmeister and Bäck, 1991; Koza, 1992), self-modification programs may lead to very specific, *directed* sequences of strategy mutations, as opposed to *undirected*, totally random mutations. The system can arbitrarily modify its prior distribution on the space of solution candidates. Just as evolution "discovered" that having the "genetic crossover operator" was a "good thing", the system is potentially able to discover that various more directed search strategies are "good things".

5. **Life is one-way.** Note that only *direct* effects of self-modification programs on primitive probability distributions can be countermanded by the top-level strategy. In realistic environments, it is not possible to countermand all *indirect* effects and effects of the system behavior on the unknown environment — life is one-way. However, the top level may encourage the development of environment-specific strategies for countermanding certain indirect effects. Such strategies will be kept as long as they appear to be more useful than previous strategies. By focusing on the observation and control of changes of probability distributions (as opposed to general changes involving internal state and environment), the top level attempts to control a complex world by controlling a small part of it, namely, the variable probability distributions. The latter, however, may have an arbitrary influence on themselves and the rest of the world.

6. **"Usefulness" and "true usefulness".** Incremental self-improvement keeps *useful* self-modifications only in the sense that "useful" was defined above. However, the system will never have a proof that a particular self-modification program was the "true" reason for more payoff. In fact, what the system actually does is **inductive inference based on single experiences:** at one point in its life it did something, and at some later point it measures apparent overall effects on its performance. What appeared to be "useful" up until now is assumed to remain "useful" in the future, though it may have been just a fluke that might later turn out actually to have harmful consequences: "shifts of inductive bias" generated by the system itself may be evaluated as harmful in the eyes of a "god-like" external observer with additional prior knowledge. But without access to complete knowledge of the environment, the system is forced to rely upon its previous experience to decide what's harmful and what's not. This is what inductive inference is all about[3].

7. **What about self-modifications "useful just by chance"?** This question is related to the last comment. For the sake of the argument, suppose a self-modification program is considered useful by the system, but not by a god-like external observer. This does not at all imply a fatal catastrophy: typically, the reason for the apparent usefulness of the actually useless (or even harmful) self-modification program will be that its long-term effects were overcompensated (before the corresponding probability modifications were cancelled) by later, "truly" useful self-modification programs. And note that the system will always have a chance to undo previous probability modifications, by executing appropriate *additional* self-modifications.

8. **Universality / Learning to remember.** It is not difficult to show that the primitives in Table 1 form a universal set in the following sense: they can be composed to form programs writing any computable integer sequence onto the work area, within the hardwired size and range limitations. Note that the primitives make it easy to create action sequences for handling stacks, recursion, etc. The scheme allows for very general sequential interaction with the environment (given appropriate problem-specific actions that translate storage contents into output actions and environmental changes). The self-referential primitives are designed to allow for specific changes of probability distributions of all program cells (possibly done very quickly, making things like "one-shot learning" possible).

   Universality implies that the system is in principle capable of creating programs for storing representations of environmental events. Unlike with most previous reinforcement-learning algorithms, see e.g. (Barto, 1989; Watkins, 1989; Dayan and Sejnowski, 1994; Williams, 1992; Sutton, 1991), there is no need for a *Markovian interface* (Schmidhuber, 1991) between the environment and the learning system. Also, there is no need for a "discount factor" discounting the system's expectation of future payoff in case of potentially infinite life spans.

9. **Doesn't the system start with a huge disadvantage?** Conventional learning systems have a fixed learning strategy for selecting and testing solution candidates from some "non-universal" search space. Incremental self-improvement, however, does not only search for solutions to some specific task, but also for learning strategies for finding solutions. Doesn't the system's universality increase its search space? In general, it does. With many toy tasks, an external user will be able to provide a conventional learning algorithm with enough problem-specific bias to solve a certain task more quickly than (initially less informed) search based on incremental self-improvement. On the other hand, however, unlike previous learning methods, incremental self-improvement can use experience to modify its search in a universal way, by exploiting arbitrary task-specific regularities if there are any, and by creating its own problem-specific bias. In the long run, this advantage may outweigh initial disadvantages due to universality.

[3] Perhaps, tomorrow you will be punished for scratching your ear 10 years ago — maybe this is in the nature of the algorithm running the universe. There is no proof that this is not going to happen (though our environment appears to be somewhat more benign than this). In general, you would not have a chance to discover the "true" reason for the punishment.

10. **Inserting prior bias / Efficiency considerations.** Primitive instructions need not be low-level instructions like those in Table 1. They may correspond to complex submodules reflecting the user's prior knowledge. Informally, there is one general constraint to obey (Schmidhuber, 1994): whatever is computable on the used hardware, should be computable *just as efficiently* (up to a small constant factor) by a program written in the programming language. For instance, on a typical serial digital machine we would like to have instructions exploiting fast storage addressing mechanisms. We would not want to limit ourselves to the simulation of, say, a slow one tape Turing machine. Likewise, on a machine with many parallel processors we would like to use a set of instructions allowing for processes with maximal parallelism.

11. **Bias towards short runs.** Unlike Levin's universal search algorithm (which is optimal for a wide variety of *non*-incremental search problems based on trials with exactly repeatable initial conditions; see Levin, 1974), the system presented here has no explicit bias towards runs with low Kolmogorov complexity or low Levin complexity (Kolmogorov, 1965; Chaitin, 1969; Solomonoff, 1964; Levin, 1974) — e.g. runs based on only few instructions repeated over and over again. In principle, however, it may create/strengthen such a bias, and the bias will stick if it appears to be useful.

    Of course, *a priori* bias of this kind can be explicitly introduced by the programmer. One possibility is to reward low-complexity runs more than others (by providing more external payoff). Another possibility is this: instead of selecting primitives randomly (according to the current probability distributions) at each time step of each run, make random selections only if IP points to a program cell that has not yet been used during the current run. Otherwise use the instruction executed during the most recent visit of the program cell. This leads to an explicit bias towards low algorithmic probability (Solomonoff, 1964), and has been done previously in (Schmidhuber, 1994). Occasionally, this will lead to non-halting programs. For such cases, upper runtime bounds need to be introduced. In the spirit of the incremental self-improvement paradigm, such time bounds should be computed by the system itself (using appropriate special primitives). For an additional comment on inserting prior bias, see section 5.3.

12. **Exploration/exploitation tradeoff.** The system itself can decide how much time it wants to spend on exploring effects of new action sequences, and how much time it wants to spend on exploiting beneficial effects of action sequences that it tried before. In the long run, the system will prefer those strategies that led to the best (environment-specific) balance between exploration and exploitation.

13. **One task, many tasks.** An external user may choose a way of translating tasks and system performance into payoff. From the user's point of view, there may be many tasks, and the system itself may choose which to attack first. From the system's point of view, there is only one task, namely, to maximize cumulative payoff. Note that every task that requires the maximization of some kind of reward may be viewed as being decomposable into many tasks: the first task is to generate actions leading to a little bit of reward. The next task is to generate actions leading to more reward, etc.

14. **What if "the task changes?"** In the light of what has been said above, this is actually a misleading question. It is tinged by the idea of "exactly repeatable training events" suddenly being replaced by different "exactly repeatable training events". But, in this paper there is no unrealistic *a priori* assumption of exactly repeatable training events. From the system's point of view, there is only one task, namely, to maximize cumulative payoff (see previous comment). The system always tends to keep the strategy that led to the best overall results so far. Without additional prior knowledge, there are no alternatives: the system cannot know whether payoff changes are due to external "task changes", or whether they are due to long term effects of its own previous actions (as discussed in comments 6 and 7 above). Life is one-way, and change is in the nature of a dynamic environment. For the sake of the argument, however, suppose a

particular sequence of probability modifications appears justified at a certain point t_1, and the external observer decides that the "first task is solved". At time $t_2 > t_1$, however, the system fails to keep its old payoff/time ratio because the "task has changed" in the eyes of the external observer. Then, over time, direct effects of previously "useful" self-modification programs will tend to be countermanded in inverse order of their occurrence (unless they don't get protected by additional useful self-modification programs), until the current probability distributions reflect knowledge useful for solving *both* tasks. *All* probability modifications will be countermanded only if the initial strategies developed for solving the first task are useless for solving the second task. But without additional prior knowledge, this *does* make sense from the learning system's point of view.

15. **Teacher as part of the environment.** (1) Of course, an external teacher may provide task-specific inputs conveying information about task changes. But the system has first to learn to make use of these inputs. Analoguously, children first have to learn to interprete sound waves emitted by their parents as teacher signals. They will learn this if it turns out to be useful in the long run. (2) To achieve his teaching goals, the teacher may directly influence the way payoff is generated, thus influencing the context sensitivity of the reward. In both cases, it is natural to view the teacher as part of the environment.

16. **Direct teacher forcing.** The teacher may decide that the current strategy of the system (at time t_1) is actually a valuable one and should not be countermanded. Instead of influencing payoff generation (see previous comment), he may decide to influence the learning process directly, by preventing the top-level strategy from countermanding probability modifications generated by self-modification programs considered useful at time t_1. This would be one way to insert additional prior knowledge.

17. **Success history in stack.** At a given time, the current history of useful self-modifications is reflected by the current stack entries. Each self-modification program "on the stack" was followed by faster payoff intake than all previous self-modification programs "on the stack". This is true despite the fact that time for computing and testing later self-modification programs is taken into account.

18. **Useful self-modification programs are rare.** Each self-modification program undergoes a test which may last for the entire remaining system life, provided the program is considered useful for such a long time. Typically, only few self-modification programs will be followed by faster payoff intake than *all* previous useful self-modification programs. Therefore, the costs of saving "old" probability distributions in the stack typically will tend to remain comparatively small. This is borne out by the experiments in section 5.

19. **Limited stacksize — "circular" stack.** In practical applications, the stack will be finite. A circular stack that overwrites earlier stack entries (starting from the bottom entries) could keep track of self-modifications after stack overflow (circular stack). Only the *StackSize* most recent probability modifications could then be directly restored by popping. However, every probability distribution can be indirectly restored by *additional* self-modification programs executed by the system itself. In the experiments conducted so far, there never was a danger of stack overflow. See section 5.

    It is intended to introduce additional introspective primitives for addressing and examining stack entries (in the style of $GetP$). This is not yet implemented, however.

20. **When to apply incremental self-improvement?** It is always possible to construct "cruel" environments, where previous experiences are necessarily useless for future planning. Indeed, as can be seen from what has been said in the introduction, almost all thinkable environments are of this kind (except those which we generally are most interested in: those with regularities). The incremental self-improvement paradigm won't be of any help in the general case. *The same*

Primitive	Semantics
Write(a1, a2)	$C_{c_{a2}} \leftarrow c_{c_{a1}}$
Read(a1, a2)	$c_{c_{a1}} \leftarrow C_{c_{a2}}$

Table 2: *Semantics of problem specific primitives and their parameters. Again, double-indexed indirect addressing is employed. See text for rules for syntactic correctness. Compare with Table 1.*

holds for any other learning paradigm, though. However, if certain aspects of the environment "repeat themselves", if experiments conducted in the environment do not change it such that previous knowledge becomes totally useless, if the tasks to be solved do exhibit "regularities", then the incremental self-improvement paradigm appears to be a very general way of exploiting this. Incremental self-improvement should be of interest in cases where the user's bias is already captured by the choice of the initial programming language, and where the user expects additional (yet unknown) problem-specific regularities.

5 Illustrative Experiments

The following brief case studies are not designed to impress but to illustrate basic aspects of the system. The first task requires to compute regular integer strings. The second task is a maze task from (Sutton, 1991). With both tasks, the system uses low-level problem-specific primitives in addition to the general primitives from Table 1. The primitives reflect the system's initial (weak) bias. Of course, different problem-specific primitives lead to different initial bias and performance. A task that can be solved within a few minutes using one set of primitives may require a day of computation time using a different set of primitives. The purpose of this section, however, is *not* to perform a statistically significant experimental evaluation of the system's initial bias, or to study effects of introducing different kinds of initial bias, or to compare the system to other learning systems with different initial bias. Instead, this section's purpose is to describe typical aspects of system lives illustrating the system's basic (bias independent) mode of operation.

5.1 Writing Variable Sequences

Task. The external environment consists of an array of 30 variables $V_0, V_1, \ldots, V_{29}$. The i-th variable is denoted by V_i. Its current contents are denoted by $C_i \in [-Maxint, Maxint]$. Time is measured in discrete time steps. At time step 0, all variables are initialized with zeros. Every 1000 time steps, **the number of variables whose values equal their addresses** is written into a special input cell. This number is the current payoff. Then, all variables are re-initialized with zeros. The goal is to maximize cumulative payoff.

Details. In addition to the 17 general primitives from Table 1 (not counting input/output primitives), there are two problem-specific primitives. Each has two integer arguments: $Write(a1, a2)$ writes the contents of the storage cell indirectly addressed by the first argument into the variable indirectly addressed by the second argument. $Read(a1, a2)$ writes the contents of the variable indirectly addressed by the second argument into the work cell indirectly addressed by the second argument. See Table 2. Write and read operations outside the valid ranges halt the current run.

Since $n_{ops} = 17 + 2 = 19$, all initial probabilities of all possible contents of all program cells are equal to $\frac{1}{19}$. Parameters for storage size etc. are: $Min = -1000$, $Max = 100$, $ProgramStart = bottom(\frac{n_{ops}}{2})$, $MinP = 0.001$, $StackSize = 10,000$, $Maxint = 10,000$. To inform the system about what is going on, the following values are written into special input cells whenever they change: IP, sp, and the remainder of $t/Maxint$ (integer division, where t denotes the current time).

Measuring time. By definition, each computation that requires the consideration of all n_{ops} probabilities of some program cell (such as selecting an instruction, selecting a parameter, pushing or popping probability distributions during top-level management, etc.) costs one time step. Other computations do not cost anything. This ensures that measured time is of the order of total cpu-time. The somewhat unelegant way of measuring time was introduced because measuring cpu-time *directly* turned out to be somewhat unreliable.

How difficult is this task? For a number of reasons, the task is non-trivial — the system does not appear to have much built-in bias towards the task: (1) Only one of the 19 primitives ($Write$) may affect variable contents at all. But initially, the system does not even have such seemingly trivial knowledge — there is no built-in idea about which actions may lead to payoff. Therefore, it has to find out on its own. (2) The values referred to by the two arguments of $Write$ have to be identical and within the required ranges to lead to a useful result. (3) There are 30 *different* variables with 30 different values. Only one of them, namely V_0, is correctly re-initialized with its own address after each payoff event. (4) There is no explicit *a priori* bias towards *short* programs, such as the one in (Schmidhuber, 1994) for a related task. (5) Finally, recall that the work area is never re-initialized after system birth. Hence, as mentioned above, it may be viewed as part of the environment of the program area. The environment is changing quite unpredictably, due to actions executed by the system itself.

Performance was measured with and without self-modification capabilities. In the latter case, the primitives $IncP$ and $DecP$ had no effect. Both versions were run for 10^9 time steps, corresponding to 10^6 payoff events, or about half a day of computation time on a SUN SPARC 10. Note that the *optimal* cumulative payoff is $3.0 * 10^7$. This value can be achieved only by a system with "optimal" prior bias — starting at birth, such a system keeps executing optimal actions without having to learn anything.

5.1.1 Results Without Self-Modifications.

At system death, total payoff equaled about $1.41 * 10^6$. Average payoff per payoff event was about 1.41. Most of the total payoff (about $1.0 * 10^6$) could be attributed to the fact that V_0 was correctly re-initialized after each payoff event: the system received a little bit of payoff even in cases where it did not execute any *write* operations. As expected, average payoff intake did not significantly increase or decrease during the lifetime of the system. However, this was not *safely* predictable in advance, due to the changing environment.

5.1.2 Results With Self-Modifications.

At system death, total payoff was about $1.87 * 10^7$. To find out whether the incremental self-improvement paradigm did indeed lead to incremental self-improvement, let us have a look at the learning history.

Self-generated reduction of numbers of probability modifications. In the beginning, the system computed a lot of probability modifications but soon preferred to decrease the number of probability modifications per time interval. There were 36,729 probability modifications during the first 10^6 time steps. There were 9,045 probability modifications during the second 10^6 time steps. Almost all probability changes were countermanded by the top-level strategy; by this time, the stack had only 49 entries corresponding to 27 useful self-modification programs. Most of the useful self-modification programs computed either one or two probability modifications. After 10^7 time steps, there were only about 4,000 probability modifications per 10^6 time steps.

Speed-up of payoff intake. By then, the system behaved much more deterministically. Average payoff per payoff event had increased from 1.4 to 10.5 (the optimal value being 30.0, of course), and the stack had 66 entries. These entries corresponded to 66 modifications of single cell probability distributions, computed by 40 self-modification programs — each being more "useful" than all the previous ones. **Storage already looked very messy**. For instance, almost all cells in the work area were filled with (partly big) integers quite different from the initial values. Recall that the work area is never re-initialized and may be viewed as part of the environment of the program area.

First maximal payoff. After 286,918 payoff events, the system correctly had written *all* 30 variables for the first time, and received maximal payoff 30.0. Due to remaining non-determinism in the system,

the current *average* payoff per payoff event (measured shortly afterwards, at time step 300,000,000) was about 18.2.

After 400,000 payoff events, current average payoff per payoff event was 19.3. By now, the number of probability modifications per 10^6 time steps was up to 6600 again. But the stack had only 131 entries (corresponding to 90 "useful" self-modification programs). After 500,000 payoff events, current average payoff per payoff event was 20.7 (143 stack entries). After 1,000,000 payoff events (at "system death"), it was about 23.7, with tendency to increase. By then, there were 185 stack entries. They corresponded to 132 self-modification programs, each being more "useful" than all the previous ones.

Temporary speed-ups of performance improvement. Performance did *not* increase smoothly during the lifetime of the system. For instance, at time step 109,295,000, the system correctly had written more than 20 variables for the first time (namely 21). This record was not broken for a long time — for nearly 10^8 additional time steps. This time interval is comparable to the entire previous learning time. Then, an unexpected sequence of rather quick improvements began. At time 207,606,000, the new record was 23 correct variables. At time 217,961,000, the new record was 24. At time 225,123,000, the new record was 25. Nearly immediately afterwards, at time 225,254,000, the new record was 26. Thus, within less than $2 * 10^7$ time steps, the record was broken 4 times. Then progress slowed down again. Such temporary speed-ups of performance improvement indicate useful shifts of inductive bias, which may later be replaced by inductive bias created by the next "breakthrough".

Automatic fine-tuning of search space structure. A look at the stack entries revealed that many (but far from all) *useful* probability modifications focused on few program cells. Often, self-modification programs directly changing the probabilities of certain additional self-modification programs were considered useful. For instance, 5 of the 167 stack entries at time step $8 * 10^7$ corresponded to "useful" probability modifications of the (self-referential) *IncP* action of the program cell with address 14. 8 entries corresponded to "useful" modifications of the *EndSelfMod* probability of various cells. Such stack entries may be interpreted as results of "adjusting the prior on the space of solution candidates" or "fine-tuning search space structure" or "learning to create directed mutations" or "learning how to learn".

5.2 A Navigation Task

Task (following Sutton, 1991). The external environment consists of a two-dimensional grid with 9 by 6 fields. $F_{i,j}$ denotes the field in the i-th row and the j-th column. The following fields are blocked by obstacles: $F_{3,3}$, $F_{3,4}$, $F_{3,5}$, $F_{6,2}$, $F_{8,4}$, $F_{8,5}$, $F_{8,6}$. In the beginning, an artificial agent is placed on $F_{1,4}$ (the start field). In addition to the 17 general primitives from Table 1 (not counting input/output primitives), there are four problem-specific primitives with obvious meaning: *one-step-north()*, *one-step-south()*, *one-step-east()*, *one-step-west()*. The system cannot execute actions that would lead outside the grid or into an obstacle. Again, the following values are written into special cells in the input area whenever they change: *IP*, *sp*, $remainder(t/Maxint)$. Another input cell is filled with a 1 whenever the agent is on the goal field, otherwise it is filled with a 0. Four additional input cells are rewritten after each execution of some problem-specific primitive: the first (second, third, fourth) cell is filled with $Maxint$ if the field to the north (south, east, west) of the agent is blocked or does not exist, otherwise the cell is filled with $-Maxint$. *Whenever the agent reaches $F_{9,6}$ (the goal field), the system receives a constant payoff (100), and the agent is transferred back to $F_{1,4}$ (the start field).* Parameters for storage size etc. are the same as with the previous task, and time is measured the same way. Clearly, to maximize cumulative payoff, the system has to find short paths from start to goal.

How difficult is this task? Again, the system does not appear to have much built-in bias towards the task: (1) Unlike with previous reinforcement learning algorithms, the system does not have a smart initial strategy for temporal credit assignment — it has to develop its own such strategies. (2) Unlike with Sutton's original set-up (1991), the system does not see a built-in unique representation of its current position on the grid. From the system's point of view, *its interface to the environment is non-Markovian* (Schmidhuber, 1991): the current input does not provide all information about the agent's current position. (3) To make use of the few inputs it gets, the system first has to discover that certain

input cells may be relevant for solving its task. (4) The total environment (including the work area) is changing quite unpredictably, due to actions executed by the system itself.

5.2.1 Results without Self-Modifications.

As with the previous task, the system was first tested with self-referential primitives $IncP$ and $DecP$ being switched off. At system death at time 10^9, total payoff was about $0.79 * 10^8$. Average "trial length" (number of time steps required to move from start to goal) was 12,637. The shortest trial ever occurred around time step $8.5 * 10^8$ and took 168 time steps.

5.2.2 Results with Self-Modifications.

At system death (at time 10^9), total payoff was about $9.57 * 10^8$. By then, average trial length (including time required for top-level management, of course) was down to 79.7 time steps (as opposed to more than 12,000 time steps without self-modifications), with ongoing tendency to decrease. As with the previous task, performance did not improve smoothly. The history of broken records reflects the history of performance improvements:

First, there was a rather quick sequence of improvements which lasted until time $2.75 * 10^6$. By then (after 1951 payoff events), the shortest trial so far had taken 83 time steps. Then, the "current record" did not improve any more for a comparatively long time interval: $5.28 * 10^6$ time steps — the length of this "boring" time interval by far exceeded the entire previous learning time.

Sudden improvement speed-up. Then, quite unexpected to the observer, the system started to create a new sequence of additional improvements around time step $8 * 10^6$. At time $8.04 * 10^6$, the record was down to 73. At time $8.84 * 10^6$, the record was down to 68. At time $9.25 * 10^6$, the record was down to 63. At time $9.57 * 10^6$, the record was down to 57. At time $10.14 * 10^6$, the record was down to 50. At time $10.51 * 10^6$, the record was down to 32. Thus, within about $2.4 * 10^6$ time steps, the record was broken 6 times, sometimes dramatically. Then, performance improvement slowed down again.

Throughout this flurry of broken records starting at time $8.04 * 10^6$, the number of stack entries increased quite steadily from 25 (corresponding to 17 useful self-modification programs) to 32 (corresponding to 22 self-modification programs). Apparently, around time step $8 * 10^6$, the system made a "revolutionary" discovery that permitted a sequence of more "evolutionary" additional directed self-mutations.

At system death (time step 10^9), the record was down to 22. The system's average payoff intake per time interval still had a tendency to increase. In the end, there were 104 useful self-modification programs, each leading to "better" results than all previous ones. As with the previous task, many useful self-modification programs directly modified the probabilities of additional self-modification programs. Compare the paragraph entitled "automatic fine-tuning search space structure" in section 5.1.2.

Experiment 2: corrupted inputs. In another experiment, the system was applied to the same task, but inputs were corrupted and unreliable. In the beginning, it took the system much longer to come up with short trials. At time $3.83 * 10^8$, the current record was 51. Then, not much happened for a long time: there was only one minor improvement (50) during the next $5.33 * 10^8$ time steps. Again, the length of this "boring" time interval by far exceeded the entire previous learning time. Then, around time step $9.16 * 10^8$ (corresponding to half a day of computation time), a "revolution" occurred: within only about 10^8 additional time steps, the record was broken 13 times: at time $10.15 * 10^8$, the record was down to 20. Throughout this sudden flurry of broken records starting at time $9.16 * 10^8$, the number of stack entries increased quite steadily from 138 (corresponding to 96 useful self-modification programs) to 189 (corresponding to 137 self-modification programs). Then, performance improvement slowed down again. System life ended at time step $1.5 * 10^9$. By this time, the record was down to 18. The system's average payoff intake per time interval still had a tendency to increase.

5.3 Three Comments

1. **Stability of probability modifications.** With the experiments conducted so far, the top level hardly ever countermanded probability modifications other than those computed by the 10 most recent *useful* self-modification programs. For instance, once there were 120 stack entries, the 100 oldest stack entries appeared extremely safe and had a good chance to survive the entire system life. This empirically justifies the method suggested in the comment on limited, circular stacks in section 4.

2. **Revolutions and evolutions.** In the tasks above, unexpected temporary speed-ups of performance improvements were observed. Even if the system appears to be stuck for a long time, the external observer never can be sure that it will not suddenly discover a new, "revolutionary" shift of bias that builds the basis for additonal, smoother, "evolutionary" performance improvements. This is analoguous to the history of science itself. One nice thing about open-ended incremental self-improvement is that there is no significant theoretical limit to what the system may learn. This is, of course, due to the universal nature of the underlying programming language.

   Informally, a "revolution" corresponds to a self-improvement with high *"conceptual jump size"* (an expression coined by Solomonoff, 1990), while "evolution" corresponds to a sequence of self-improvements with low conceptual jump sizes.

3. **Inserting prior bias.** The experiments above certainly are not meant to convince the reader that from now on, he should combine the incremental self-improvement paradigm with the low-level programming language from section 3 and apply it to real world problems. Instead, the experiments are meant to illustrate basic principles of the paradigm. Of course, with large scale problems, it is desirable to **insert prior knowledge** into the system (if such knowledge is indeed available). With incremental self-improvement, *a priori* knowledge resides in the programmer's selection of primitives with problem-specific built-in bias (and in the payoff function he chooses). *There is no reason why certain primitives should not be complex, time consuming programs by themselves, such as statistic classifiers, neural net learning algorithms, logic programs, etc.* For instance, using different primitives for the navigation task from section 5.2 can greatly reduce the time required to achieve near-optimal trials. This paper, however, is not a study of the effects of different kinds of initial bias.

6 History of Ideas / Previous Work

In what follows, I will briefly describe earlier work and the train of thought leading to this paper.

Meta-evolution. My first attempts to come up with schemes for "true"[4] self-referential learning based on universal languages date back to 1986. They were partly inspired by a collaboration with Dickmanns and Winklhofer (1986). We used a genetic algorithm (GA) to evolve variable length Prolog programs for solving simple tasks[5]. Soon there was a desire to improve the trivial mutation and crossover strategies used to construct new programs from old ones. This led to an algorithmic scheme (called *"meta-evolution"*) for letting more sophisticated strategies be learned by a potentially infinite hierarchy

[4] I am not talking about fixed learning algorithms for adjusting the parameters of others. For instance, GAs are sometimes used to adjust learning rates of gradient based neural nets, etc. Or a neural net is used to compute the weights of another neural net. In the literature, one can find quite a few approaches of this kind (too many to cite them all — I settle by citing none, not even my own). Although such approaches sometimes may have their merits, they do not deserve the attribute "self-referential" — the additional level typically just defers the credit assignment problem.

There were a few apparently more general approaches. For instance, Lenat (1983) reports that his Eurisko system was able to discover certain heuristics for discovering heuristics. However, his approach, as well as all other previous approaches I am aware of, were either quite limited (many essential aspects of system behavior being unmodifiable), and/or lacked a convincing global credit assignment strategy (as embodied by the top-level strategy of the incremental self-improvement paradigm).

[5] Today, this approach would be classified as "Genetic Programming", e.g. (Koza, 1992).

of higher level GAs whose domains were to construct construction strategies (Schmidhuber, 1987). *Meta-evolution* recursively creates a growing hierarchy of pools of programs — higher-level pools containing program modifying programs being applied to lower-level programs and being rewarded based on lower-level performance.

Collapsing meta-levels. The explicit creation of "meta-levels" and "meta-meta-levels" seemed unnatural, however. For this reason, alternative systems based on "self-referential" languages were explored, the goal being to collapse all meta-levels into one (Schmidhuber, 1987). At that time, however, no convincing global credit assignment strategy was provided.

Self-referential neural nets. Later work presented a neural network with the potential to run its own weight change algorithm (Schmidhuber 1992, 1993a, 1993b). With this system, top-level credit assignment is performed by gradient descent. This is unsatisfactory, however, due to problems with local minima, and because repeatable training sequences are required. In general, this makes it impossible to take the entire learning history into account.

Algorithmic probability / Universal search. Levin's universal search algorithm is theoretically optimal for certain "non-incremental" search tasks with exactly repeatable initial conditions. See Levin (1974, 1984); see also Adleman (1979). There were a few attempts to extend universal search to incremental learning situations, where previous "trials" may provide information about how to speed up further learning, see e.g. (Solomonoff, 1990; Paul and Solomonoff, 1991; Schmidhuber, 1994). For instance, to improve future performance, Solomonoff (1964, 1990) describes more traditional (as opposed to self-improving) methods for assigning probabilities to successful "subprograms". Alternatively, one of the actually implemented systems in (Schmidhuber, 1994) simply keeps successful code in its program area. This system was a conceptual starting point for the one in the current paper. With first attempts (in September 1994), the probability distributions underlying the Turing machine equivalent language required for universal search were modified heuristically. One strategy was to slightly increase the context-dependent probabilities of program cell contents used in successful programs, and then continue universal search based on the new probability distributions. With a number of experiments, this actually led to good results (at first glance, more impressive results than those in the current paper, at least if one does not take the lack of bias into account, as one should always do). The system, however, was unsatisfactory, precisely because there was no principled way of adjusting probability distributions. This criticism led to the ideas expressed in the current paper.

Meta-version of universal search. Without going into details, Solomonoff (1990) mentions that self-improvement may be formulated as a time-limited optimization problem, thus being solvable by universal search. However, the straight-forward meta-version of universal search (generating and evaluating probability distributions in order of their Levin complexities — see Levin, 1974) just defers the credit assignment problem to the meta-level, and does *not* necessarily make optimal incremental use of computational resources and previous experience[6]. Note that incremental self-improvement is *not* a meta-version of universal search. In fact, incremental self-improvement does *not* make a difference between "search" and "meta-search".

Ongoing/future work. The concrete implementation described in section 3 represents only one out of many ways of implementing the incremental self-improvement paradigm. It is intended to apply incremental self-improvement to more complex tasks, including prediction and control tasks, using a variety of universal, "self-referential" sets of primitives, including sets designed to exploit the benefits of parallel, neural net-like hardware.

[6] Solomonoff appears to be well aware of problems with the meta-version: at the end of his 1990 paper, he refers to self-improvement as a "more distant goal": *"The kind of training needed involves more mathematics and work on various kinds of optimization problems — ultimately problems of improving computer programs."* Another "more distant goal" mentioned by Solomonoff is to let the system work *"on an unordered batch of problems — deciding itself which are the easiest, and solving them first"*. Note that the incremental self-improvement paradigm addresses both goals, without depending on a meta-version of universal search. See e.g. comment 13 in section 4.

7 Acknowledgements

I am grateful to Ray Solomonoff, Peter Dayan, Mike Mozer, Don Matthis, Clayton McMillan, and various NIPS*94 participants, for valuable comments/discussions on the first version of this report (from November 24, 1994). Many thanks to Sepp Hochreiter, Gerhard Weiß, Martin Eldracher, Margit Kinder, and Daniel Prelinger, for critical remarks on earlier drafts, and to Leslie Kaelbling, David Cohn, and Tommi Jaakkola, for useful comments on later versions. I am particularly indebted to Mark Ring for extensive and constructive criticism.

References

Adleman, L. (1979). Time, space, and randomness. Technical Report MIT/LCS/79/TM-131, Laboratory for Computer Science, MIT.

Barto, A. G. (1989). Connectionist approaches for control. Technical Report COINS Technical Report 89-89, University of Massachusetts, Amherst MA 01003.

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

Dayan, P. and Sejnowski, T. (1994). TD(λ): Convergence with probability 1. *Machine Learning*, 14:295–301.

Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1986). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

Dietterich, T. G. (1989). Limitations of inductive learning. In *Proceedings of the Sixth International Workshop on Machine Learning, Ithaca, NY*, pages 124–128. San Francisco, CA: Morgan Kaufmann.

Hoffmeister, F. and Bäck, T. (1991). Genetic algorithms and evolution strategies: Similarities and differences. In Männer, R. and Schwefel, H. P., editors, *Proc. of 1st International Conference on Parallel Problem Solving from Nature, Berlin*. Springer.

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

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

Koza, J. R. (1992). Genetic evolution and co-evolution of computer programs. In Langton, C., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 313–324. Addison Wesley Publishing Company.

Lenat, D. (1983). Theory formation by heuristic search. *Machine Learning*, 21.

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.

Paul, W. and Solomonoff, R. J. (1991). Autonomous theory building systems. Manuscript, revised 1994.

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

Ring, M. B. (1994). *Continual Learning in Reinforcement Environments.* PhD thesis, University of Texas at Austin, Austin, Texas 78712.

Schaffer, C. (1993). Overfitting avoidance as bias. *Machine Learning*, 10:153–178.

Schmidhuber, J. H. (1987). Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Institut für Informatik, Technische Universität München.

Schmidhuber, J. H. (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. H. (1992). Steps towards "self-referential" learning. Technical Report CU-CS-627-92, Dept. of Comp. Sci., University of Colorado at Boulder.

Schmidhuber, J. H. (1993a). A neural network that embeds its own meta-levels. In *Proc. of the International Conference on Neural Networks '93, San Francisco.* IEEE.

Schmidhuber, J. H. (1993b). A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer.

Schmidhuber, J. H. (1994). Discovering problem solutions with low Kolmogorov complexity and high generalization capability. Technical Report FKI-194-94, Fakultät für Informatik, Technische Universität München.

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

Shannon, C. E. (1948). A mathematical theory of communication (parts I and II). *Bell System Technical Journal*, XXVII:379–423.

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

Solomonoff, R. (1990). A system for incremental learning based on algorithmic probability. In Pednault, E. P. D., editor, *The Theory and Application of Minimal-Length Encoding (Preprint of Symposium papers of AAAI 1990 Spring Symposium).*

Sutton, R. S. (1991). Integrated modeling and control based on reinforcement learning and dynamic programming. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 471–478. San Mateo, CA: Morgan Kaufmann.

Utgoff, P. (1986). Shift of bias for inductive concept learning. In *Machine Learning*, volume 2. Morgan Kaufmann, Los Altos, CA.

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

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Wolpert, D. H. (1993). On overfitting avoidance as bias. Technical Report SFI TR 93-03-016, Santa Fe Institute, NM 87501.