SIMPLE PRINCIPLES OF METALEARNING

Technical Report IDSIA-69-96

Jürgen Schmidhuber & Jieyu Zhao & Marco Wiering
IDSIA, Corso Elvezia 36, CH-6900-Lugano, Switzerland

juergen,marco,jieyu@idsia.ch - http://www.idsia.ch

June 27, 1996

Abstract

The goal of metalearning is to generate useful shifts of inductive bias by adapting the current learning strategy in a "useful" way. Our learner leads a single life during which actions are continually executed according to the system's internal state and current *policy* (a modifiable, probabilistic algorithm mapping environmental inputs and internal states to outputs and new internal states). An action is considered a learning algorithm if it can modify the policy. Effects of learning processes on later learning processes are measured using reward/time ratios. Occasional backtracking enforces success histories of still valid policy modifications corresponding to histories of lifelong reward accelerations. The principle allows for plugging in a wide variety of learning algorithms. In particular, it allows for embedding the learner's policy modification strategy within the policy itself (self-reference). To demonstrate the principle's feasibility in cases where conventional reinforcement learning fails, we test it in complex, non-Markovian, changing environments ("POMDPs"). One of the tasks involves more than 10^{13} states, two learners that both cooperate and compete, and strongly delayed reinforcement signals (initially separated by more than 300,000 time steps).

The biggest difference between time and space is that you can't reuse time.
Merrick Furst

1 Introduction / Overview

In the spirit of the first author's earlier work (e.g., 1987, 1993, 1994), we will use the expressions "metalearning" and "learning to learn" to characterize learners that (1) can evaluate and compare learning methods, (2) measure the benefits of early learning on subsequent learning, (3) use such evaluations to reason about learning strategies and to select "useful" ones while discarding others. An algorithm is not considered to have learned to learn if it improves merely by luck, if it does not measure the effects of early learning on later learning, or if it has no explicit method designed to translate such measurements into useful learning strategies.

We focus on estimating the usefulness of each learning process or bias shift (Utgoff, 1986) and then exploiting it in later learning processes within a realistic, life-time reinforcement learning context. Applications will include "inductive transfer" across multiple tasks (e.g., Caruana et al., 1995).

Scenario. A reinforcement learner executes a lifelong action sequence in an unknown environment. Its single life lasts from birth at time 0 to death at unknown time T. Actions are selected according to its changing *policy*, a modifiable, probabilistic algorithm mapping environmental

inputs and internal states to outputs and new internal states. Different actions may consume different amounts of execution time — compare, e.g., (Russell and Wefald, 1991; Boddy and Dean, 1994), and references given therein. Occasionally the environment provides real-valued "reinforcement". The sum of *all* reinforcements obtained between time 0 and time $t > 0$ is denoted by $R(t)$ (where $R(0) = 0$). Because the learner cannot change the past, its goal at time t is to use previous experience to maximize $R(T) - R(t)$, the cumulative future reinforcement. Somewhat related, but more restricted, limited resource scenarios were also studied, e.g., by Berry and Fristedt (1985), Gittins (1989), Greiner (1996), and references therein.

Realistic environments. Convergence theorems for existing reinforcement learning algorithms require infinite sampling size as well as strong (often Markovian) assumptions about the environment, e.g., (Kumar and Varaiya, 1986; Sutton, 1988; Barto, 1989; Watkins and Dayan, 1992; Williams, 1992). They are of great theoretical interest but not extremely relevant for realistic environments where computational resources and sampling sizes are limited. "Learning" (modifying the policy) consumes part of the learner's limited life. So do policy tests. And a disappointing test outcome may imply that it is already too late for collecting much additional reinforcement. One cannot buy cheap shares of a company *after* the price jumped up (no repeatable trials guaranteed)! In general, at a given time in system life, we may assume only one single training example to estimate the long-term usefulness of any previous policy modification — namely the performance since then. This requires us to rethink a bit the conventional, multiple trial-based way we measure performance and generalize.

Basic ideas (see details in section 2). Meta-reinforcement learning (MRL) treats learning algorithms just like other actions. Their probabilities of being executed at a given time depend on the learner's current internal state and policy. Their only distinguishing feature is that they may also *modify* the policy. In case of policy changes, information necessary to restore the old policy is pushed on a stack.

The learner's life-time performance is occasionally evaluated by *backtracking processes*. At a given time, we assume there is only one single training example to evaluate the current long-term usefulness of any currently valid policy modification M, namely the average reinforcement per time since M occurred. This includes all reinforcement collected after later modifications for which M may have set the stage, thus providing a measure of how useful a learning process was for later learning processes, or how useful a shift of inductive bias was for further bias shifts. Using the stack, backtracking invalidates certain previous policy modifications such that the *remaining* modifications correspond to an (in the worst case empty) history of long-term reinforcement accelerations. Until the next backtracking process, the straight-forward generalization assumption is: each policy modification (or bias shift) that survived the most recent backtracking process was useful and will remain useful.

For instance, whenever the environment appears to change in the sense that the reward per time for the current behavior is observed to decrease, backtracking will selectively undo those previously learned policy modifications that do not appear useful any more (perhaps because they were too specifically tailored to previous tasks and are useless for "inductive transfer"). Backtracking will selectively keep those modifications that still appear useful because until now they were followed by long-term reinforcement accelerations, despite possible changes of the environment.

Due to unknown reinforcement delays, there is no *a priori* good way of starting backtracking processes. That's why MRL also allows for learning to trigger them. Since learning algorithms are actions and can be combined (according to the policy) to form more complex learning algorithms, MRL allows for embedding the learning strategy within the policy itself. There is no pre-wired difference between "learning", "metalearning", "metametalearning" etc. For alternative views of metalearning, see, e.g., Lenat (1983), Rosenbloom et al. (1993).

Disclaimer. In general, unknown environments, neither MRL nor any other scheme is guaranteed to continually increase reinforcement intake per *fixed* time interval, or to find the policy that will lead to maximal cumulative reinforcement. But at least MRL is guaranteed to selectively undo those policy modifications that were not empirically observed to be followed by an overall speed-up of average reinforcement intake (even in non-Markovian settings). This is more than can be said about interesting, previous reinforcement learning algorithms, e.g., (Kumar and Varaiya,

1986; Barto, 1989; Watkins and Dayan, 1992; Williams, 1992; Schmidhuber, 1991; Jaakkola et al., 1995; Kaelbling et al., 1995; Ring, 1994).

Outline of remainder. Section 2 will describe the learner's basic cycle of operations and clarify technical details of what has been said in paragraph "basic ideas". Section 3 will explain why the basic cycle enforces lifelong histories of reinforcement accelerations despite possible interference from parallel, internal or external processes. To demonstrate MRL's feasibility and generality, sections 4 and 5 will present two concrete implementations and experiments with complex, partially observable environments (POEs). They show that MRL makes sense especially in situations where previous algorithms fail because the environment does not satisfy preconditions that would make them sound. Some of our POEs are much bigger and more complex than POEs considered in previous POE work.

2 Basic MRL Cycle

At time 0 (system birth), we initialize the learner's variable internal state $\mathcal{I}$, a vector of variable, binary or real-valued components. Environmental inputs may be represented by certain components of $\mathcal{I}$. We also initialize the vector-valued policy Pol. Pol's i-th variable component is denoted Pol_i. There is a set of possible actions to be selected and executed according to current Pol and $\mathcal{I}$. For now, there is no need to specify Pol — this will be done in the experimental sections (typically, Pol_i will be a conditional probability distribution on the possible next actions, given current $\mathcal{I}$). We introduce an initially empty stack $\mathcal{S}$ that allows for stack entries with varying sizes, and the conventional *push* and *pop* operations. Until time T (system death), the system repeats the following basic MRL cycle over and over again (while time is continually increasing):

1. Execute actions selected according to Pol and $\mathcal{I}$ (this may change environment and $\mathcal{I}$), until a certain Evaluation Criterion is satisfied, or until an action is selected that will *modify* Pol.
2. **IF** the Evaluation Criterion is satisfied, **THEN** start the following backtracking procedure to undo certain previous Pol modifications if necessary (to ensure that the history of still valid modifications corresponds to a history of reinforcement accelerations):

   2.1. Set variable t equal to current time.

   IF there is no *"tag"* (a pair of time and cumulative reinforcement until then) stored somewhere in $\mathcal{S}$,

   THEN push the tag $(t, R(t))$ onto $\mathcal{S}$, and go to **3** (this ends the current backtracking process).

   ELSE denote the topmost tag in $\mathcal{S}$ by $(t', R(t'))$. **IF** there are no further tags, **THEN** set variable $t'' = 0$ (recall $R(t'') = R(0) = 0$). **ELSE** let $(t'', R(t''))$ denote the last but topmost tag in $\mathcal{S}$.

   2.2. IF

   $$\frac{R(t) - R(t')}{t - t'} > \frac{R(t) - R(t'')}{t - t''}$$

   THEN push tag $(t, R(t))$, and go to **3.** This ends the current backtracking process.

   ELSE pop off all stack entries above the one for tag $(t', R(t'))$ (these entries will be former policy components saved during earlier executions of step 3), and use them to restore Pol as it used to be before time t'. Then also pop off the tag $(t', R(t'))$. Go to **2.1.**

3. **IF** the most recent action selected in step **1** will modify Pol, **THEN** push copies of those Pol_i to be modified onto $\mathcal{S}$, and execute the action.
4. **IF** some Termination Criterion is satisfied, **THEN** die. **ELSE** go to step **1**.

Comment: each step above (including pushing and popping) will consume various amounts of system life-time.

3 The MRL Cycle Ensures Life-Time Success Stories

Lifelong reinforcement acceleration. At a given time in the learner's life, define the set of currently *valid* times as those previous times still stored in tags somewhere in $\mathcal{S}$. If this set is not empty right before tag $(t, R(t))$ is pushed in step **2.2** of the basic cycle, then let t_i $(i \in \{1, 2, \ldots, V(t)\})$ denote the i-th valid time, counted from the bottom of $\mathcal{S}$. It is easy to show (Schmidhuber, 1994, 1996) that the current backtracking process will have enforced the following, essential criterion which is central to MRL (t is the t in the most recent step **2.2**):

$$\frac{R(t)}{t} < \frac{R(t) - R(t_1)}{t - t_1} < \frac{R(t) - R(t_2)}{t - t_2} < \ldots < \frac{R(t) - R(t_{V(t)})}{t - t_{V(t)}}. \quad (1)$$

What does this mean? Each valid time marks the beginning of a long-term reinforcement acceleration (measured up until time t). Everything that happened after a valid time, every action and every backtracking process, is justified in the sense that it was observed to occur during a long-term speed-up. The only *still valid* policy modifications or bias shifts are those that occurred in between some valid time (or time 0) and the beginning of the next backtracking process following that time. Again, each such block of policy modifications will have its justification in the following sense: each block's "time marker" (the valid time preceding the block's first modification) was followed by faster average reinforcement intake than all previous such time markers. The *still valid* policy modifications are those that survived all backtracking processes until now. In this sense, the history of *still valid* bias shifts is **guaranteed** to be a life-time success story (in the worst case an empty one). No Markov-assumption is required.

MRL's generalization assumption. At the end of each backtracking process, until the beginning of the next one, MRL's simple, straight-forward generalization assumption for inductive inference is: policy modifications that survived the most recent backtracking will remain useful. In other words, until there is empirical evidence to the contrary, the assumption is: the still valid modifications *contributed* to the long-term speed-up, and will continue to contribute. In general, unknown environments, which other generalization assumption would make sense? Recall that since life is one-way (time is never reset), during each backtracking process the system has to generalize from a *single* experience concerning the usefulness of actions/policy modifications taken after any given previous point in time: the average reinforcement per time since then.

If we prevent modification probabilities from vanishing entirely then occasionally the system will execute policy modifications, and keep those consistent with inequality (1). In this sense, it cannot help getting better, if the environment does indeed provide a chance to improve performance, given the initial set of possible actions representing the system's initial bias. Essentially, the system keeps generating and undoing policy modifications until it discovers some that indeed fit its generalization assumption.

Greediness? MRL's strategy appears to be a greedy one. It always keeps the policy that was observed to outperform all previous policies in terms of long-term reward/time ratios. To deal with unknown reinforcement delays, however, the degree of greediness is learnable — backtracking processes may be triggered or delayed according to the modifiable policy itself.

Speed? Due to the generality of the approach, no reasonable statements can be made about improvement *speed*, which indeed highly depends on the nature of the environment and the choice of initial, "primitive" actions (including learning algorithms) to be combined according to the policy. This lack of quantitative convergence results is shared by almost all other, less general reinforcement learning schemes, though.

Actions can be almost anything. For instance, an action executed in step 3 may be a neural net algorithm. Or it may be a Bayesian analysis of previous events. While this analysis is running, *time is running*, too. Thus, the complexity of the Bayesian approach is automatically taken into account. Similarly, actions may be calls of a Q-learning variant (see experiment 4 in section 5.3). For instance, plugging Q-learning into MRL makes sense in situations where Q-learning by itself is questionable because the environment might not entirely satisfy the preconditions that would make Q-learning sound.

$\mathcal{I}$ as part of *Pol*'s environment. As the basic cycle is repeated again and again, neither the internal state nor the environment are assumed to be reset (real world set-up). Essentially, what each backtracking process attempts (and succeeds) to do is to make the history of *still valid* modifications a success story despite harmful (or beneficial) interference from parallel, external and internal processes. It is appropriate to view the internal state as part of the policy's changing environment.

Outline of remainder. Sections 4 and 5 will describe two concrete implementations of MRL. The first implementation's action set consists of a single but "strong", policy-modifying action (a call of a Levin search extension). The second implementation uses many different, less "powerful" actions. They resemble assembler-like instructions from which many different learning strategies can be built (the system's modifiable, "self-referential" learning strategy is able to modify itself). Both implementations are successfully tested in complex environments where standard reinforcement learning algorithms fail. In particular, the second, "self-referential" system is successfully applied to a non-Markovian task that involves more than 10^{13} states, two learners that both cooperate and compete, and strongly delayed reinforcement signals (initially separated by more than 300,000 time steps on average). Section 6 will conclude.

4 Implementation 1: Plugging Levin Search into MRL

Overview. In this section, we use an adaptive extension of Levin search (LS) (Levin, 1973; Levin, 1984) as only learning action to be plugged into MRL. We apply it to partially observable Markov decision problems (POMDPs), which recently received a lot of attention in the reinforcement learning community, e.g., (Jaakkola et al., 1995; Kaelbling et al., 1995; Ring, 1994; McCallum, 1995; Littman, 1994; Cliff and Ross, 1994; Schmidhuber, 1991). We first show that LS by itself can solve partially observable mazes (POMs) involving many more states and obstacles than those solved by various previous authors (we will also see that LS can easily outperform Q-learning). We then extend LS to plug it into MRL, and experimentally show dramatic search time reduction for sequences of more and more complex POMDPs ("inductive transfer").

4.1 Levin Search (LS)

Unbeknownst to many machine learning researchers, there exists a search algorithm with amazing theoretical properties: for a broad class of search problems, Levin search (LS) (Levin, 1973; Levin, 1984) has the optimal order of computational complexity. For instance, suppose there is an algorithm that solves a certain type of maze task in $O(n^3)$ steps, where n is a positive integer representing the problem size. Then universal LS will solve the same task in at most $O(n^3)$ steps. See (Li and Vitányi, 1993) for an overview. See (Schmidhuber, 1995) for recent implementations/applications.

Basic concepts. LS requires a set of n_{ops} primitive, prewired instructions $b_1, ..., b_{n_{ops}}$ that can be composed to form arbitrary sequential programs. Essentially, LS generates and tests solution candidates s (program outputs represented as strings over a finite alphabet) in order of their Levin complexities $Kt(s) = \min_q\{-logD_P(q) + log\ t(q,s)\}$, where q stands for a program that computes s in $t(q,s)$ time steps, and $D_P(q)$ is the probability of guessing q according to a *fixed* Solomonoff-Levin distribution (Li and Vitányi, 1993) on the set of possible programs (in section 4.2, however, we will make the distribution variable).

Optimality. Amazingly, given primitives representing a universal programming language, for a broad class of problems, including inversion problems and time-limited optimization problems, LS can be shown to be optimal with respect to total expected search time, leaving aside a constant factor independent of the problem size (Levin, 1973; Levin, 1984; Li and Vitányi, 1993). Still, until recently LS has not received much attention except in purely theoretical studies — see, e.g., (Watanabe, 1992).

Practical implementation. In our practical LS version, there is an upper bound m on program length (due to obvious storage limitations). a_i denotes the address of the i-th instruction.

Each program is generated incrementally: first we select an instruction for a_1, then for a_2, etc. D_P is given by a matrix P, where P_{ij} ($i \in 1, ..., m$, $j \in 1, ..., n_{ops}$) denotes the probability of selecting b_j as the instruction at address a_i, given that the first $i-1$ instructions have already been selected. The probability of a program is the product of the probabilities of its constituents.

LS' arguments are P and the representation of a problem denoted by N. LS' output is a program that computes a solution to the problem if it found any. In this section, all $P_{ij} = \frac{1}{n_{ops}}$ will remain fixed. LS is implemented as a sequence of longer and longer phases:

Levin search(problem N, probability matrix P)

(1) Set $Phase$, the number of the current phase, equal to 1. In what follows, let $\phi(Phase)$ denote the set of *not yet executed* programs q satisfying $D_P(q) \geq \frac{1}{Phase}$.

(2) **Repeat**

(2.1) **While** $\phi(Phase) \neq \{\}$ and no solution found **do**: Generate a program $q \in \phi(Phase)$, and run q until it either halts or until it used up $\frac{D_P(q) * Phase}{c}$ steps. If q computed a solution for N, return q and exit.

(2.2) Set $Phase := 2Phase$

until solution found or $Phase \geq Phase_{MAX}$.

Return empty program $\{\}$.

Here c and $Phase_{MAX}$ are prespecified constants. The procedure above is essentially the same (has the same order of complexity) as the one described in the second paragraph of this section — see, e.g., (Solomonoff, 1986; Li and Vitányi, 1993).

4.2 Adaptive Levin Search (ALS)

LS is not necessarily optimal for "incremental" learning problems where experience with previous problems may help to reduce future search costs. To make an incremental search method out of non-incremental LS, we introduce a simple, heuristic, adaptive LS extension (ALS) that uses experience with previous problems to adaptively modify LS' underlying probability distribution. ALS essentially works as follows: whenever LS found a program q that computed a solution for the current problem, the probabilities of q's instructions $q_1, q_2, \ldots, q_{l(q)}$ are increased (here $q_i \in \{b_1, \ldots, b_{n_{ops}}\}$ denotes q's i-th instruction, and $l(q)$ denotes q's length — if LS did not find a solution (q is the empty program), then $l(q)$ is defined to be 0). This will increase the probability of the entire program. The probability adjustment is controlled by a learning rate γ ($0 < \gamma < 1$). ALS is related to the linear reward-inaction algorithm, e.g., (Kaelbling, 1993) — the main difference is: ALS uses LS to search through *program space* as opposed to single action space. As in the previous section, the probability distribution D_P is determined by P. Initially, all $P_{ij} = \frac{1}{n_{ops}}$. However, given a sequence of problems $(N_1, N_2, ..., N_r)$, the P_{ij} may undergo changes caused by ALS:

ALS (problems $(N_1, N_2, ..., N_r)$, variable matrix P)

for $i := 1$ **to** r **do:**
  $q :=$ **Levin search**(N_i, P); **Adapt**(q, P).

where the procedure **Adapt** works as follows:

Adapt(program q, variable matrix P)

for $i := 1$ **to** $l(q)$, $j := 1$ **to** n_{ops} **do:**
  if $(q_i = b_j)$ **then** $P_{ij} := P_{ij} + \gamma(1 - P_{ij})$
  else $P_{ij} := (1 - \gamma)P_{ij}$

4.3 Plugging ALS into MRL

Critique of adaptive LS. Although ALS seems a reasonable first step towards making LS adaptive (and actually leads to very nice experimental results — see section 4.5), there is no theoretical proof that it will generate only probability modifications that will speed up the process of finding solutions to new tasks. Like *any* learning algorithm, ALS may sometimes produce harmful instead of beneficial bias shifts, depending on the environment. To address this issue, we simply plug ALS into MRL from section 2. MRL ensures that the system will keep only probability modifications representing a lifelong history of performance improvements.

ALS as primitive for MRL. At a given time, the learner's current policy is the variable matrix P above. To plug ALS into MRL, we simply replace steps 1 and 3 in section 2's MRL cycle by:

1. If the current MRL cycle's problem is N_i, then set $q :=$ **Levin search** (N_i, P). If a solution was found, generate reinforcement of $+1.0$. Set Evaluation Criterion $= TRUE$. The next action will be a call of **Adapt**, which will change the policy P.
3. Push copies of those P_i (the i-th column of matrix P) to be modified by **Adapt** onto $\mathcal{S}$, and call **Adapt**(q, P).

Each call of **Adapt** causes a bias shift for future learning. In between two calls of **Adapt**, a certain amount of time will be consumed by **Levin search** (details about how time is measured will follow in the section on experiments). As always, MRL's goal is to receive as much reward as quickly as possible, by generating policy changes that minimize the computation time required by *future* calls of **Levin search** and **Adapt**.

Partially Observable Maze Problems. The next subsections will describe experiments validating the usefulness of LS, ALS, and MRL. To begin with, in an illustrative application with a partially observable maze that has many more states and obstacles than those presented in other POMDP work (see, e.g., (Cliff and Ross, 1994)), we will show how LS by itself can solve POMDPs with huge state spaces but low-complexity solutions (Q-learning variants fail to solve these tasks). Then we will present experiments with multiple, more and more difficult tasks (inductive transfer). We will show that ALS can use previous experience to speed-up the process of finding new solutions, and that ALS plugged into MRL (MRL+ALS for short) always outperforms ALS by itself.

4.4 Experiment 1: A Big, Partially Observable Maze (POM)

The current section is a prelude to section 4.5 which will address inductive transfer issues. Here we will only show that LS by itself can be very useful for POMDP problems. See also (Wiering and Schmidhuber, 1996).

Task. Figure 1 shows a 39×38-maze with a single start position (S) and a single goal position (G). The maze has many more fields and obstacles than mazes used by previous authors working on POMDPs — for instance, McCallum's maze has only 23 free fields (McCallum, 1995). The goal is to find a program that makes an agent move from S to G.

Instructions. Programs can be composed from 9 primitive instructions. These instructions represent the *initial bias* provided by the programmer (in what follows, superscripts will indicate instruction numbers). The first 8 instructions have the following syntax : REPEAT step forward UNTIL condition *Cond*, THEN rotate towards direction *Dir*.
Instruction 1^1 : *Cond* = front is blocked, *Dir* = left.
Instruction 2^2 : *Cond* = front is blocked, *Dir* = right.
Instruction 3^3 : *Cond* = left field is free, *Dir* = left.
Instruction 4^4 : *Cond* = left field is free, *Dir* = right.
Instruction 5^5 : *Cond* = left field is free, *Dir* = none.
Instruction 6^6 : *Cond* = right field is free, *Dir* = left.
Instruction 7^7 : *Cond* = right field is free, *Dir* = right.
Instruction 8^8 : *Cond* = right field is free, *Dir* = none.

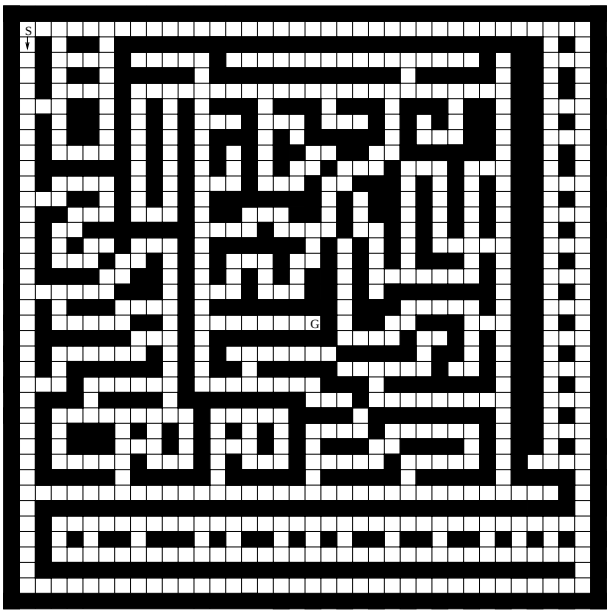

Figure 1: *An apparently complex, partially observable* 39×38*-maze with a low-complexity shortest path from start S to goal G involving 127 steps. Despite the relatively large state space, the agent can implicitly perceive only one of three highly ambiguous types of input, namely "front is blocked or not", "left field is free or not", "right field is free or not" (compare list of primitives). Hence, from the agent's perspective, the task is a difficult POMDP. The S and the arrow indicate the agent's initial position and rotation.*

Instruction 9 is: `Jump(address, nr-times)`. It has two parameters: `nr-times` $\in 1, 2, \ldots, MAXR$ (with the constant MAXR representing the maximum number of repetitions), and `address` $\in 1, 2, \ldots,$ *top*, where *top* is the highest address in the current program. `Jump` uses an additional hidden variable `nr-times-to-go` which is initially set to `nr-times`. The semantics are: If `nr-times-to-go` > 0, continue execution at address `address`. If $0 <$ `nr-times-to-go` $< MAXR$, decrement `nr-times-to-go`. If `nr-times-to-go` $= 0$, set `nr-times-to-go` to `nr-times`. Note that `nr-times` $= MAXR$ may cause an infinite loop. The `Jump` instruction is essential for exploiting the possibility that solutions may consist of *repeatable* action sequences and "subprograms", thus having low algorithmic complexity (Kolmogorov, 1965; Chaitin, 1969; Solomonoff, 1964). LS' incrementally growing time limit automatically deals with those programs that don't halt, by preventing them from consuming too much time.

As mentioned in section 4.1, the probability of a program is the product of the probabilities of its constituents. To deal with probabilities of the two `Jump` parameters, we introduce two additional variable matrices, $\bar{P}$ and $\hat{P}$. For a program with $l \leq k$ instructions, to specify the conditional probability $\bar{P}_{ij}$ of a jump to address a_j, given that the instruction at address a_i is `Jump` ($i \in 1, ..., l$, $j \in 1, ..., l$), we first normalize the entries $\bar{P}_{i1}$, $\bar{P}_{i2}$, ..., $\bar{P}_{il}$ (this ensures that the relevant entries sum up to 1). Provided the instruction at address a_i is `Jump`, for $i \in 1, ..., k$, $j \in 1, ..., MAXR$, $\hat{P}_{ij}$ specifies the probability of the `nr-times` parameter being set to j. Both $\bar{P}$ and $\hat{P}$ are initialized uniformly and are adapted by ALS just like P itself.

Restricted LS-variant. Note that the instructions above are not sufficient to build a universal programming language — the experiments in this paper are confined to a *restricted* version of LS. From the instructions above, however, one can build programs for solving any maze in which it is not necessary to completely reverse the direction of movement (rotation by 180 degrees) in a corridor. Note that it is mainly the `Jump` instruction that allows for composing low-complexity

solutions from "subprograms" (LS provides a sound way for dealing with infinite loops).

Rules. Before LS generates, runs and tests a new program, the agent is reset to its start position. *Collisions with walls halt the program — this makes the problem hard.* A path generated by a program that makes the agent hit the goal is called a solution (the agent is not required to stop at the goal — there are no explicit halt instructions).

Why is this a POMDP? Because the instructions above are not sufficient to tell the agent exactly where it is: at a given time, the agent can perceive only one of three highly ambiguous types of input (by executing the appropriate primitive): "front is blocked or not", "left field is free or not", "right field is free or not" (compare list of primitives). Some sort of memory is required to disambiguate apparently equal situations encountered on the way to the goal. Q-learning, for instance, is not guaranteed to solve POMDPs, e.g, (Watkins and Dayan, 1992). Our agent, however, can use memory implicit in the state of the execution of its current program to disambiguate ambiguous situations.

Measuring time. The computational cost of a single **Levin search** call in between two **Adapt** calls is essentially the sum of the costs of all the programs it tests. To measure the cost of a single program, we simply count the total number of forward steps and rotations during program execution (this number is of the order of total computation time). *Note that instructions often cost more than 1 step!* To detect infinite loops, LS also measures the time consumed by `Jump` instructions (one time step per executed `Jump`). In a realistic application, however, the time consumed by a robot move would by far exceed the time consumed by a `Jump` instruction — we omit this (negligible) cost in the experimental results.

Comparison. We compare LS to three variants of Q-learning (Watkins and Dayan, 1992) and random search. Random search repeatedly and randomly selects and executes one of the instructions (1-8) until the goal is hit (like with Levin search, the agent is reset to its start position whenever it hits the wall). Since random search (unlike LS) does not have a time limit for testing, it may not use the jump – this is to prevent it from wandering into infinite loops. The first Q-variant uses the same 8 instructions, but has the advantage that it can distinguish all possible states (952 possible inputs — but this actually makes the task much easier, because it is no POMDP any more). The first Q-variant was just tested to see how much more difficult the problem becomes in the POMDP setting. The second Q-variant can only observe whether the four surrounding fields are blocked or not (16 possible inputs), and the third Q-variant receives as input a unique representation of the five most recent executed instructions (37449 possible inputs — this requires a gigantic Q-table!). Actually, after a few initial experiments with the second Q-variant, we noticed that it could not use its input for preventing collisions (the agent always walks for a while and then rotates — in front of a wall, every instruction will cause a collision). To improve the second Q-variant's performance, we appropriately altered the instructions: each instruction consists of one of the 3 types of rotations followed by one of the 3 types of forward walks (thus the total number of instructions is 9 — for the same reason as with random search, the jump instruction cannot be used). The parameters of the Q-learning variants were first coarsely optimized on a number of smaller mazes which they were able to solve. We set $c = 0.005$, which means that in the first phase ($Phase = 1$ in the LS procedure), a program with probability 1 may execute up to 200 steps before being stopped. We set $MAXR = 6$.

Typical result. In the *easy, totally observable* case, Q-learning took on average 694,933 steps (10 simulations were conducted) to solve the maze in Figure 1. However, as expected, in the *difficult, partially observable* cases, neither the two Q-learning variants nor random search were ever able to solve the maze within 1,000,000,000 steps (5 simulations were conducted). In contrast, LS was indeed able to solve the POMDP: LS required 97,395,311 steps to find a program q computing a 127-step shortest path to the goal in Figure 1. LS' low-complexity solution q involves two nested loops:

```
1) REPEAT step forward UNTIL left field is free^5
2) Jump (1 , 3)^9
3) REPEAT step forward UNTIL left field is free, rotate left^3
4) Jump (1 , 5)^9
```

We have $D_P(q) = \frac{1}{9}\frac{1}{9}\frac{1}{4}\frac{1}{6}\frac{1}{9}\frac{1}{9}\frac{1}{4}\frac{1}{6} = 2.65 * 10^{-7}$.

Similar results were obtained with many other mazes having non-trivial solutions with low algorithmic complexity. Such experiments illustrate that smart search through program space can be beneficial in cases where the task appears complex but actually has low-complexity solutions. Since LS has a principled way of dealing with non-halting programs and time-limits (unlike, e.g., "Genetic Programming"(GP)), LS may also be of interest for researchers working in GP and related fields — among the first papers on using GP-like algorithms to evolve assembler-like computer programs are (Cramer, 1985; Dickmanns et al., 1987). See also (Koza, 1992) for later work.

ALS: single tasks versus multiple tasks. If we use the adaptive LS extension (ALS) for a single task as the one above (by repeatedly applying LS to the same problem and changing the underlying probability distribution in between successive calls according to section 4.2), then the probability matrix rapidly converges such that late LS calls find the solution almost immediately. This is not very interesting, however — once the solution to a single problem is found (and there are no additional problems), there is no point in investing additional efforts into probability updates (probability shifts). ALS is more interesting in cases where there are multiple tasks, and where the solution to one task conveys some but not all information helpful for solving additional tasks (inductive transfer). This is what the next section is about.

4.5 Experiment 2: Incremental Learning / Inductive Transfer

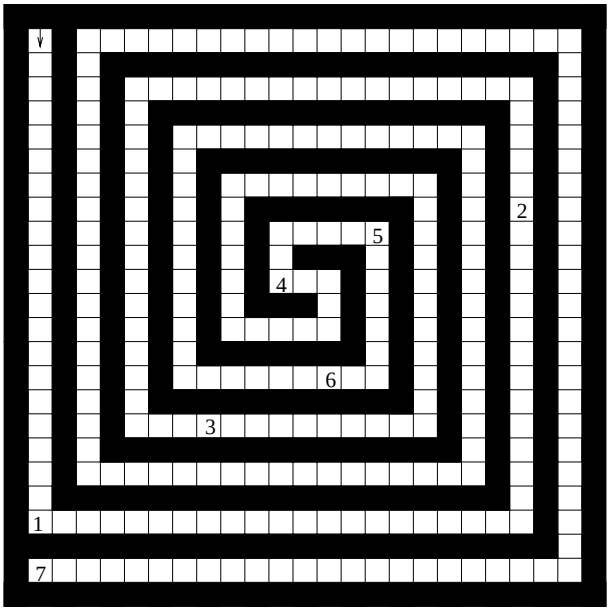

Figure 2: *A 23 × 23 labyrinth. The arrow indicates the agent's initial position and direction. Numbers indicate goal positions. The higher the number, the more difficult the goal. The agent's task is to find all goal positions in a given "goalset". Goalsets change over time.*

This section will show that ALS can use experience to significantly reduce average search time consumed by successive LS calls in cases where there are more and more complex tasks to solve (inductive transfer), and that ALS can be further improved by plugging it into MRL.

Task. Figure 2 shows a 23 × 23 maze and 7 different goal positions marked 1,2,...,7. With a given goal, the task is to reach it from the start state. Each goal is further away from start than goals with lower numbers. We create 4 different "goalsets" G_1, G_2, G_3, G_4. G_i contains goals 1, 2, ..., 3 + i. One simulation consists of 40 "epochs" E_1, E_2, ... E_{40}. During epochs $E_{10(i-1)+1}$ to E_{10i}, all goals in G_i $(i = 1, 2, 3, 4)$ have to be found in order of their distance to the start. During

Algorithm	METHOD	SET 1	SET 2	SET 3	SET 4
LS last goal		4.3	1,014	9,505	17,295
LS		8.7	1,024	10,530	27,820
ALS	1	12.9	382	553	650
ALS + MRL	1	12.2	237	331	405
ALS	2	13.0	487	192	289
ALS + MRL	2	11.5	345	85	230

Table 1: *For METHODs 1 and 2, we list the number of steps (in thousands) required by LS, ALS, MRL+ALS to find all goals in a specific goalset during the goalset's first epoch (for optimal learning rates). The probability matrices are adapted each time a goal is found. The first LS row refers only to the most difficult goals in each goalset (those with maximal numbers). ALS outperforms LS on all goalsets but the first, and MRL+ALS achieves additional speed-ups. MRL+ALS works well for all learning rates, ALS by itself does not. Also, all our incremental learning procedures dramatically outperform LS by itself.*

each epoch, we update the probability matrices P, $\bar{P}$ and $\hat{P}$ whenever a goal is found. For each epoch we store the total number of steps required to find all goals in the corresponding goalset. We compare two variants of incremental learning, METHOD 1 and METHOD 2:

METHOD 1 — inter-goalset resets. Whenever the goalset changes (at epochs E_{11}, E_{21}, E_{31}), we uniformly initialize probability matrices P, $\bar{P}$ and $\hat{P}$. Inductive transfer can occur only within goalsets. We compare METHOD 1 to simulations in which only the most difficult task of each epoch has to be solved.

METHOD 2 — no inter-goalset resets. We don't reset P, $\bar{P}$ and $\hat{P}$ in case of goalset changes. We have both intra-goalset and inter-goalset inductive transfer. We compare METHOD 2 to METHOD 1, to measure benefits of inter-goalset transfer for solving goalsets with an additional, more difficult goal.

Comparison. We compare LS by itself, ALS by itself, and MRL+ALS, for both METHODs 1 and 2.

LS results. Using $c = 0.005$ and $MAXR = 15$, LS needed $17.3 * 10^6$ time steps to find goal 7 (without any kind of incremental learning or inductive transfer).

Learning rate influence. To find optimal learning rates minimizing the total number of steps during simulations of ALS and MRL+ALS, we tried all learning rates γ in {0.01, 0.02,..., 0.95}. We found that MRL+ALS is fairly learning rate independent: it solves *all* tasks with *all* learning rates in acceptable time (10^8 time steps), whereas for ALS without MRL (and METHOD 2) only small learning rates are feasible – large learning rate subspaces do not work for many goals. Thus, the first type of MRL-generated speed-up lies in the lower expected search time for appropriate learning rates.

With METHOD 1, ALS performs best with a fixed learning rate $\gamma = 0.32$, and MRL+ALS performs best with $\gamma = 0.45$, with additional uniform noise in $[-0.05, 0.05]$ (noise tends to improve MRL+ALS's performance a little bit, but worsens ALS' performance). With METHOD 2, ALS performs best with $\gamma = 0.05$, and MRL+ALS performs best with $\gamma = 0.24$ and added noise in $[-0.05, 0.05]$.

For METHODs 1 and 2 and all goalsets G_i ($i = 1, 2, 3, 4$), Table 1 lists the numbers of steps required by LS, ALS, MRL+ALS to find all of G_i's goals during epoch $E_{(i-1)*10+1}$, in which the agent encounters the goal positions in the goalset for the first time.

ALS versus LS. ALS performs much better than LS on goalsets G_2, G_3, G_4. ALS does not help to to improve performance on G_1's goalset, though (epoch E_1), because there are many easily discoverable programs solving the first few goals.

MRL+ALS versus ALS. MRL+ALS always outperforms ALS by itself. For optimal learning rates, the speed-up factor for METHOD 1 ranges from 6 % to 67 %. The speed-up factor for METHOD 2 ranges from 13 % to 26 %. Recall, however, that *there are many learning rates where ALS by itself completely fails, while MRL+ALS does not.* This makes MRL+ALS much more

Algorithm	METHOD	SET 1	SET 2	SET 3	SET 4
ALS	2	675	9,442	10,220	9,321
ALS + MRL	2	442	1,431	3,321	4,728
ALS	1	379	1,125	2,050	3,356
ALS + MRL	1	379	1,125	2,050	2,673

Table 2: *For all goalsets, we list numbers of steps consumed by ALS and MRL+ALS to find all goals of goalset G_i during the final epoch E_{10i}.*

Algorithm	METHOD	TOTAL	TOTAL FIRST	TOTAL LAST
LS		39,385		
ALS	2	1,820	980	29.7
ALS	1	1,670	1,600	6.91
ALS + MRL	1	1,050	984	6.23
ALS + MRL	2	873	671	9.92

Table 3: *The total number of steps (in thousands) consumed by LS, ALS, MRL+ALS (1) during one entire simulation, (2) during all the first epochs of all goalsets, (3) during all the final epochs of all goalsets.*

robust.

For optimal learning rates, the biggest speed-up occurs for G_3. Here MRL decreases search costs dramatically, because after having found goal 5, it undoes apparently harmful bias shifts before searching for goal 6.

METHOD 1 versus METHOD 2. METHOD 2 works much better than METHOD 1 on G_3 and G_4, but not as well on G_2 (for G_1 both methods are equal — differences in performance can be explained by different learning rates which were optimized for the total task). Why? Optimizing a policy for goals 1—4 will not necessarily help to speed up discovery of goal 5, but instead cause a harmful bias shift by overtraining the probability matrices. METHOD 1, however, can extract enough useful knowledge from the first 4 goals to decrease search costs for goal 5.

More MRL benefits. Table 2 lists the number of steps consumed during the final epoch E_{10i} of each goalset G_i (the results of LS by itself are identical to those in table 1). Using MRL typically improves the final result, and never worsens it. Speed-up factors range from 0 to 560 %.

For all goalsets, Table 3 lists the total number of steps consumed during all epochs of one simulation, the total number of all steps for those epochs (E_1, E_{11}, E_{21}, E_{31}) in which new goalsets are introduced, and the total number of steps required for the final epochs (E_{10}, E_{20}, E_{30}, E_{40}). MRL always improves the results. For the total number of steps — which is an almost linear function of the time consumed during the simulation — the MRL-generated speed-up is 60% for METHOD 1 and 108 % for METHOD 2 (the "fully incremental" method). Although METHOD 2 speeds up performance during each goalset's first epoch (ignoring the costs that occurred before introduction of this goalset), final results are better without inter-goalset learning. This is not so surprising: by using policies *optimized* for previous goalsets, we generate bias shifts for speeding up discovery of new, acceptable solutions, without necessarily making *optimal* solutions of future tasks more likely (due to "evolutionary ballast" from previous solutions).

LS by itself needs $27.8 * 10^6$ steps for finding *all* goals in G_4. Recall that $17.3 * 10^6$ of them are spent for finding only goal 7. Using incremental learning, however, we obtain large speed-up factors. METHOD 1 with MRL+ALS improves performance by a factor in excess of 40 (see results of MRL+ALS on the first epoch of G_4). Figure 3(A) plots performance against epoch numbers. Each time the goalset changes, initial search costs are large (reflected by sharp peaks). Soon, however, both methods incorporate experience into the policy. We see that MRL keeps initial search costs significantly lower.

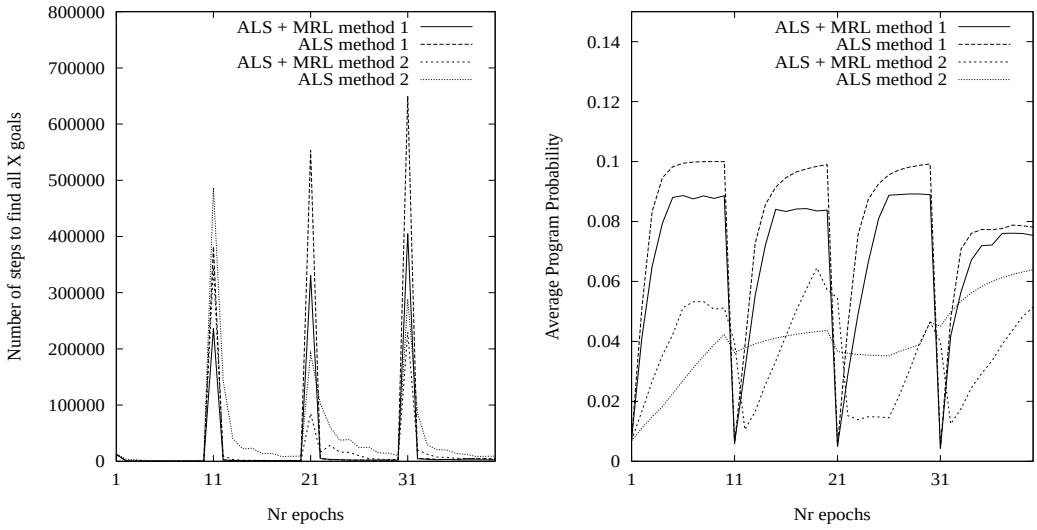

Figure 3: *(A) Average number of steps per epoch required to find all of the current goalset's goals, plotted against epoch numbers. Peaks reflect goalset changes. (B) Average probability of programs computing solutions (before solutions are actually found).*

The safety belt effect. Figure 3(B) plots epoch numbers against average probability of programs computing solutions. With METHOD 1, MRL+ALS tends to keep the probabilities lower than ALS by itself: high program probabilities are not always beneficial. With METHOD 2, MRL undoes many policy modifications when goalsets change, thus keeping the policy flexible and reducing initial search costs.

Effectively, MRL is controlling the prior on the search space such that overall average search time is reduced, given a particular task sequence. For METHOD 1, after E_{40} the number of still valid modifications of policy components (probability distributions) is 377 for ALS, but only 61 for MRL+ALS (therefore, 61 is MRL+ALS's total final stack size). For METHOD 2, the corresponding numbers are 996 and 63. We see that MRL keeps only about 16% respectively 6% of all modifications! The remaining modifications are deemed unworthy, because they were not observed to be followed by life-time reinforcement speed-ups. Clearly, MRL prevents ALS from overdoing its policy modifications ("safety belt effect"). This is MRL's simple, basic purpose: undo certain learning algorithms' policy changes and bias shifts once they start looking harmful in terms of long-term reinforcement/time ratios.

It should be clear that the MRL+ALS implementation is just one of many possible MRL applications — we may plug many alternative learning algorithms into MRL.

5 Implementation 2: Incremental Self-Improvement (IS)

The previous section used a single, complex, powerful, primitive learning action (adaptive Levin Search). The current section exploits the fact that it is also possible to use many, much simpler actions that can be combined to form more complex learning strategies, or metalearning strategies (Schmidhuber, 1994, 1996; Zhao and Schmidhuber, 1996).

Overview. We will use a simple, assembler-like programming language which allows for writing many kinds of (learning) algorithms. Effectively, *we embed the way the system modifies its policy and triggers backtracking within the "self-referential" policy itself.* MRL is used to keep only those "self-modifications" followed by reinforcement speed-ups, in particular those leading to "better" future self-modifications, recursively. We call this "incremental self-improvement" (IS).

Outline of section. Subsection 5.1 will describe how the policy is represented as a set of variable probability distributions on a set of assembler-like instructions, how the policy builds the basis for generating and executing a lifelong instruction sequence, how the system can modify itself executing special "self-referential instructions", and how MRL keeps only the "good" policy modifications. Subsection 5.2 will describe experiments. In the first experiment, MRL is applied to a sequence of more and more difficult function approximation tasks. The second task is our most challenging one: MRL solves a complex, huge state space POMDP which involves two interacting, changing, learning agents.

5.1 Policy and Program Execution

Storage / Instructions. The learner makes use of an assembler-like programming language similar to but not quite as general as the one in (Schmidhuber, 1995). It has n addressable *work cells* with addresses ranging from 0 to $n-1$. The variable, real-valued contents of the work cell with address k are denoted c_k. Processes in the external environment occasionally write inputs into certain work cells. There also are m addressable *program cells* with addresses ranging from 0 to $m-1$. The variable, integer-valued contents of the program cell with address i are denoted d_i. An internal variable *Instruction Pointer* (*IP*) with range $\{0, \ldots, m-1\}$ always points to one of the program cells (initially to the first one). There also is a fixed set I of n_{ops} integer values $\{0, \ldots, n_{ops}-1\}$, which sometimes represent instructions, and sometimes represent arguments, depending on the position of *IP*. *IP* and work cells together represent the system's internal state $\mathcal{I}$ (see section 2). For each value j in I, there is an assembler-like instruction b_j with n_j integer-valued parameters. See (Schmidhuber, 1996) for a related, illustrative figure. In the following (incomplete) list of instructions to be used in experiment 3, the symbols w_1, w_2, w_3 stand for parameters that may take on integer values between 0 and $n-1$ (later we will encounter additional instructions):

b_0: *Add*(w_1, w_2, w_3) : $c_{w_3} \leftarrow c_{w_1} + c_{w_2}$ (add the contents of work cell w_1 and work cell w_2, write the result into work cell w_3).

b_1: *Sub*(w_1, w_2, w_3) : $c_{w_3} \leftarrow c_{w_1} - c_{w_2}$.

b_2: *Mul*(w_1, w_2, w_3) : $c_{w_3} \leftarrow c_{w_1} * c_{w_2}$.

b_3: *Mov*(w_1, w_2) : $c_{w_2} \leftarrow c_{w_1}$.

b_4: *JumpHome: IP*$\leftarrow 0$ *(jump back to 1st program cell).*

Later (in the experimental subsections) we will encounter additional primitives allowing the learner (1) to move around in an environment, and (2) to perceive certain objects within a limited range.

Instruction probabilities / Current policy. For each program cell i there is a variable probability distribution P_i on I. For every possible $j \in I$, $(0 \leq j \leq n_{ops}-1)$, P_{ij} specifies for cell i the conditional probability that, when pointed to by *IP*, its contents will be set to j. The set of all current P_{ij}-values defines a probability matrix P with columns P_i $(0 \leq i \leq m-1)$. P is called the learner's *current policy*. In the beginning of the learner's life, all P_{ij} are equal (maximum entropy initialization). If $IP = i$, the contents of i, namely d_i, will be interpreted as instruction b_{d_i} (such as *Add* or *Mul*), and the contents of cells that immediately follow i will be interpreted as b_{d_i}'s arguments, to be selected according to the corresponding P-values. For example, the integer sequence 1 6 8 7 will be interpreted as *Sub(6, 8, 7)* — subtract the contents of cell 6 from the contents of cell 8 and put the result into cell 7.

"Self-reference". To obtain a learner that can explicitly modify its own policy (by running its own learning strategies), we introduce a special "self-referential" instruction *IncProb* not yet mentioned above:

b_5: *IncProb*(w_1, w_2, w_3) : Increase P_{ij} by γ percent, where $i = w_1 * n_{ops} + w_2$ and $j = w_3$ (this construction allows for addressing a broad range of program cells), and renormalize P_i (but prevent P-values from falling below a minimal value ϵ, to avoid near-determinism). Parameters w_1, w_2, w_3 may take on integer values between 0 and $n_{ops} - 1$. In the experiments, we will use $\gamma = 15, \epsilon = 0.001$.

In conjunction with other primitives, *IncProb* may be used in instruction sequences that compute directed policy modifications. Calls of *IncProb* represent the *only* way of modifying the policy.

Self-delimiting self-modification sequences (SMSs). SMSs are subsequences of the lifelong action sequence. The first *IncProb* after the learner's "birth" or after each MRL backtracking process (see section 2) begins an SMS. The SMS ends by executing another yet unmentioned "self-referential" primitive:

b_6: *EndSelfMod*(w_1). Temporarily disable *IncProb*, by preventing future *IncProb* instructions from causing any probability modifications, until w_1 ($1 \leq w_1 \leq n_{ops} - 1$) additional non-zero reinforcement signals have been received — this will satisfy the EVALUATION CRITERION in the basic MRL cycle (section 2).

Some of the (initially highly random) action subsequences executed during system life will indeed be SMSs. Depending on the nature of the other instructions, SMSs can compute almost arbitrary sequences of modifications of P_{ij} values. This may result in almost arbitrary modifications of context-dependent probabilities of future action subsequences, including future SMSs. *Policy changes can be generated only by SMSs.* SMSs build the basis for "metalearning": SMSs are generated according to the policy, and may change the policy. Hence, the policy can essentially change itself, and also the way it changes itself, etc.

SMSs can influence the timing of backtracking processes, because they can influence the times at which the EVALUATION CRITERION will be met. Thus SMSs can temporarily protect the learner from performance evaluations and policy restaurations.

Plugging SMSs into MRL. We replace step 1 in the basic MRL cycle (see section 2) by the following procedure:

1. **REPEAT** the following **UNTIL** the EVALUATION CRITERION is satisfied or the Boolean variable MODIFICATION-CRITERION (initially FALSE) is TRUE:

   1.1. Randomly generate an integer $j \in I$ according to matrix column P_{IP} (the distribution of the program cell pointed to by IP, initially 0 at system birth). Set program cell contents $d_{IP} := j$. Translate j into the corresponding current instruction b_j. Look up the number n_j of cells required to store b_j's parameters. If $IP > m - n_j - 2$, reset IP to 0, go to step 1. Otherwise generate instruction arguments for the n_j cells immediately following IP according to their probability distributions P_{IP+1}, ..., P_{IP+n_j}, and set IP to $IP + n_j + 1$.

   1.2. **IF** b_j is a learning instruction and not currently disabled by a previous *EndSelfMod* instruction, **THEN** set MODIFICATION-CRITERION $= TRUE$, exit the current REPEAT loop, and go to step **2** of the basic MRL cycle.

   1.3. Execute b_j. **IF** b_j is *EndSelfMod* and the topmost entry in the stack $\mathcal{S}$ is not a "tag", **THEN** set the integer variable n_{NZR} equal to the first parameter of b_j plus one (this will influence the time at which EVALUATION CRITERION will be reached).

   1.4. **IF** there is a new environmental input, **THEN** let it modify $\mathcal{I}$.

   1.5. **IF** $n_{NZR} > 0$ and non-zero reinforcement occurred during the current cycle, **THEN** decrement n_{NZR}. **IF** n_{NZR} is zero, **THEN** set EVALUATION CRITERION $= TRUE$.

We also change step 3 in the basic MRL cycle as follows:

3. **IF** MODIFICATION-CRITERION $= TRUE$, **THEN** push copies of those Pol_i to be modified by b_j (from step 1.2) onto $\mathcal{S}$, and execute b_j.

5.2 Experiment 3: Function Approximation / Inductive Transfer

Our first experiment does not yet involve multiple, interacting learners that are part of each other's changing environment. It just demonstrates that IS can successfully learn in a changing environment where the tasks to be solved become more and more difficult over time (inductive transfer). The experiment serves as an introduction to the next experiment (section 5.3), where task changes won't be easy to identify due to multiple, co-evolving learners with partly conflicting goals.

Task sequence. Our system is exposed to a sequence of more and more complex function approximation problems. The functions to be learned are $f_1(x,y) = x + y$; $f_2(x,y,z) = x + y - z$; $f_3(x,y,z) = (x + y - z)^2$; $f_4(x,y,z) = (x + y - z)^4$; $f_5(x,y,z) = (x + y - z)^8$.

Trials. The system's single life can be separated into n successive trials A_1, A_2, ..., A_n (but the learner has no *a priori* concept of a trial). The i-th trial lasts from discrete time step $t_i + 1$ until discrete time step t_{i+1}, where $t_1 = 0$ (system birth) and $t_{n+1} = T$ (system death). In a given trial A_i we first select a function $g_i \in \{f_1, \ldots, f_5\}$. As the trial number increases, so does the probability of selecting a more complex function. In early trials the focus is on f_1. In late trials the focus is on f_5. In between there is a gradual shift in task difficulty: using a function pointer ptr (initially 1) and an integer counter c (initially 100), in trial A_i we select $g_i := f_{ptr}$ with probability $\frac{c}{100}$, and $g_i := f_{ptr+1}$ with probability $1 - \frac{c}{100}$. If the reinforcement acceleration during the most recent two trials exceeds a certain threshold (0.05), then c is decreased by 1. If c becomes 0 then f_{ptr} is increased by 1, and c is reset to 100. This is repeated until $f_{ptr} := f_5$. From then on, f_5 is always selected.

Once g_i is selected, randomly generated real values x, y and z are put into work cells 0, 1, 2, respectively. The contents of an arbitrarily chosen work cell (we always use cell 6) are interpreted as the system's response. If c_6 fulfills the condition $|g_i(x,y,z) - c_6| < 0.0001$, then the trial ends and the current reward becomes 1.0; otherwise the current reward is 0.0.

Instructions. Instruction sequences can be composed from the following primitive instructions (compare section 5.1): *Add*(w_1, w_2, w_3), *Sub*(w_1, w_2, w_3), *Mul*(w_1, w_2, w_3), *Mov*(w_1, w_2), *IncProb*(w_1, w_2, w_3), *EndSelfMod*(w_1), *JumpHome*(). Each instruction occupies 4 successive program cells (some of them unused if the instruction has less than 3 parameters). We use $m = 50, n = 7$.

Evaluation Condition. Backtracking starts after each 5th consecutive non-zero reinforcement signal after the end of each SMS, i.e., we set $n_{NZR} = 5$.

Huge search space. Given the primitives above, random search would require about 10^{17} trials on average to find a solution for f_5 — the search space is huge. The gradual shift in task complexity, however, helps IS to learn f_5 much faster, as will be seen below.

Results. After about 9.4×10^8 instruction cycles (ca. 10^8 trials), the system is able to compute f_5 almost perfectly, given arbitrary real-valued inputs. The corresponding speed-up factor over (infeasible) random or exhaustive search is about 10^9 — compare paragraph "Huge search space" above. The solution (see Figure 4) involves 21 strongly modified probability distributions of the policy (after learning, the correct instructions had extreme probability values). At the end, the most probable code is given by the following integer sequence:

1 2 1 6 1 0 6 6 2 6 6 6 2 6 6 6 2 6 6 6 4 * * *...

The corresponding "program" and the (very high) probabilities of its instructions and parameters are shown in Table 4.

Evolution of self-modification frequencies. During its life the system generates a lot of self-modifications to compute the strongly modified policy. This includes changes of the probabilities of self-modifications. It is quite interesting (and also quite difficult) to find out to which extent the system uses self-modifying instructions to learn how to use self-modifying instructions. Figure 5 gives a vague idea of what's going on by showing a typical plot of the frequency of *IncProb* instructions during system life (sampled at intervals of 10^6 MRL cycles). Soon after its birth, the system found it useful to dramatically increase the frequency of *IncProb*; near its death (when there was nothing more to learn) it significantly reduced this frequency.

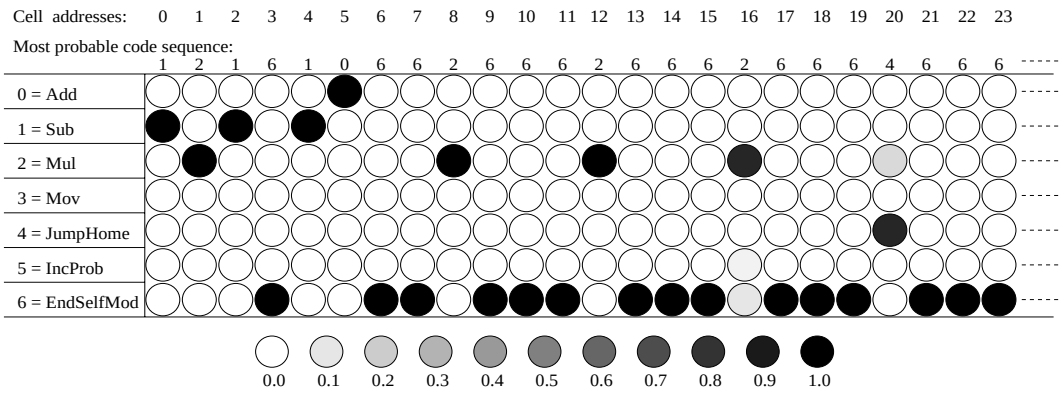

Figure 4: *The final state of the probability matrix for the function learning problem. Grey scales indicate the magnitude of probabilities of instructions and parameters. The matrix was computed by self-modification sequences generated according to the matrix itself (initially, all probability distributions were maximum entropy distributions).*

	Probabilities	Instruction	Parameters	Semantics
1.	(0.994, 0.975, 0.991, 0.994)	Sub	(2, 1, 6)	$(z - y) \Longrightarrow c_6$
2.	(0.994, 0.981, 0.994, 0.994)	Sub	(0, 6, 6)	$(x - (z - y)) \Longrightarrow c_6$
3.	(0.994, 0.994, 0.994, 0.994)	Mul	(6, 6, 6)	$(x + y - z)^2 \Longrightarrow c_6$
4.	(0.994, 0.994, 0.994, 0.994)	Mul	(6, 6, 6)	$(x + y - z)^4 \Longrightarrow c_6$
5.	(0.869, 0.976, 0.994, 0.994)	Mul	(6, 6, 6)	$(x + y - z)^8 \Longrightarrow c_6$
6.	(0.848, —, —, —)	JumpHome	(–, –, –,)	$0 \Longrightarrow IP$

Table 4: The final, most probable "program" and the corresponding probabilities.

Stack evolution. The temporary ups and downs of the stack reflect that as the tasks change, the system selectively keeps still useful old modifications (corresponding to information conveyed by previous tasks that is still valuable for solving the current task), but deletes modifications that are too specific for previous tasks. In the end, there were only about 200 stack entries corresponding to only 200 *valid* probability modifications – this is a small number compared to the about $5 * 10^5$ self-modifications executed during system life.

5.3 Experiment 4: A Hard POMDP

In the previous experiment, the learner's environment changed because of externally induced task changes. In the following experiment, it will change in a less predictable way because of another, changing learner. There won't be an obvious way of identifying task changes.

Environment. Figure 5.3 shows a partially observable environment (POE) with 600×800 fields (or pixels). The POE has many more fields and obstacles than POEs used by previous authors working on POMDPs. For instance, McCallum's maze has only 23 free fields (McCallum, 1995), and Littman et al.'s biggest problem (Littman, 1994) involves less than 1000 states. There are two IS-based agents A and B. Each has circular shape and a diameter of 30 pixel widths. At a given time, each is rotated in one of eight different directions. Total state space size exceeds 10^{13} by far, not even taking into account internal states (*IP* positions) of the agents.

There are also two keys, key A (only useful for agent A) and key B (only for agent B), and two locked doors, door A and door B, the only entries to room A and room B, respectively. Door A (B) can be opened only with key A (B). At the beginning of each "trial", both agents are randomly rotated and placed near the northwest corner, all doors are closed, key A is placed in the southeast corner, and key B is placed in room A (see Figure 5.3).

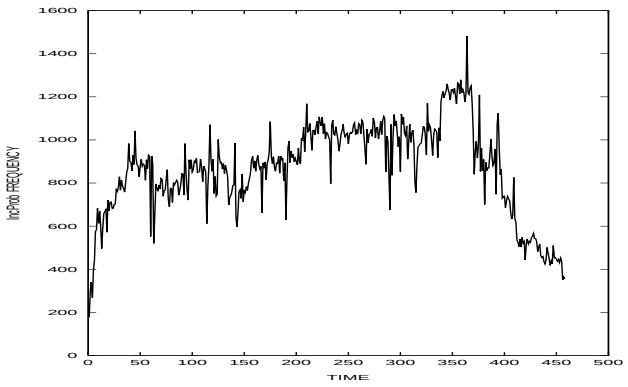

Figure 5: *Numbers of executed self-modifying instructions plotted against time, sampled at intervals of* 10^6 *instruction cycles. The graph reflects that the system soon uses self-modifying instructions to increase the frequency of self-modifying instructions. Near system death the system learns that there is not much to learn any more, and decreases this frequency.*

Task. The goal of each agent is to reach the goal in room B. This requires cooperation: (1) agent A must first find and take key A (by touching it); (2) then agent A must go to door A and open it (by touching it) for agent B; (3) then agent B must enter through door A, find and take key B; (4) then agent B must go to door B to open it (to free the way to the goal); (5) then at least one of the agents must reach the goal position. Only then a new trial starts (there is no maximal trial length, and the agents have no *a priori* concept of a trial). Reinforcement is generated *only* if one of the agents touches the goal. This agent's reinforcement is 5.0; the other agent's reinforcement is 3.0 (for its cooperation). Note that asymmetric reinforcement introduces competition.

Instruction set. Agents A and B are identical in design. Each is equipped with limited "active" sight: by executing certain instructions, it can sense obstacles, its own key, the corresponding door, or the goal, within up to 10 "steps" in front of it. The step size is 5 pixel widths. *Limited obstacle perception makes the problem a difficult POMDP.* The agent can also move forward, turn around, turn relative to its key or its door or the goal. Directions are represented as integers in $\{0, \ldots, 7\}$: 0 for north, 1 for northeast, 2 for east,... etc.. Each agent has got $m = 52$ program cells, and $n_{ops} = 13$ instructions (including *JumpHome/IncProb/EndSelfMod* instructions from section 5.1):

b_3: *Move*(w_1) — If $0 \leq w_1 \leq 9$, then move $w_1 - 3$ steps forward (if $w_1 \geq 3$) or $-w_1$ steps backward (if $w_1 < 3$) in the current direction.

b_4: *Turn*(w_1) — if $0 \leq w_1 \leq 7$ then change current direction D_c to $(D_c + w_1 + 4) mod 8$.

b_5: *TurnRelativeToKey*(w_1) — If $0 \leq w_1 \leq 7$ then first turn to the direction that best matches the line connecting the centers of agent and its key, then *Turn*(w_1).

b_6: *TurnRelativeToDoor*(w_1) — (analogous to b_5).

b_7: *TurnRelativeToGoal*(w_1) — (analogous to b_5).

b_8: *LookForKey*(w_1) — If $0 \leq w_1 \leq 9$, then if the agent's key is not within $w_1 + 1$ steps in front of the agent then increase *IP* by 4 (this is a limited kind of conditional jump). If $IP > m - 1$ then set $IP = 0$.

b_9: *LookForDoor*(w_1) — (analogous to b_8).

b_{10}: *LookForObstacle*(w_1) — (analogous to b_8).

b_{11}: *LookForGoal(w_1)* — (analogous to b_8).

b_{12}: *CondJumpBack(w_1)* — If $w_1 = 0$ and the agent does not hold its key, or if $w_1 = 1$ and the corresponding door is closed, or if $w_1 = 2$ and the agent does not touch the goal, then reset agent's IP to the instruction next to last executed *CondJumpBack* (if there is no such instruction, reset $IP = 0$).

Results without learning. If we switch off the system's self-modifying capabilities (*IncProb* has no effect), then the average reinforcement intake speed is about 1.2×10^{-5} per instruction cycle and agent. This corresponds to about 3.3×10^5 time steps per trial (random behavior).

Results with Q-Learning. Q-learning assumes that the environment is Markovian. Otherwise it is not guaranteed to work. Still, some authors occasionally apply Q-learning variants to non-Markovian tasks, sometimes even successfully. To test whether our problem is indeed too difficult for Q-learning, we tried to solve it using various $TD(\lambda)$ Q-variants. We first tried primitive actions and perceptions similar to IS's. There are 33 possible Q-actions. The first 32 are "turn to one of the 8 different directions relative to the agent's key/door/current direction/goal, and move 3 steps forward". The 33rd action is "turn 45 degrees to the right". These actions are more powerful than those used by IS (most combine two actions that are similar to IS's). There are $2 * 5 = 10$ possible, different inputs uniquely telling the agent whether it has/hasn't got its key, and whether the closest object (obstacle or key or door or goal) part of which is either 10 or 20 or 30 or 40 or 50 pixels in front of the agent is obstacle/key/door/goal/non-existent. All of this corresponds to 10 rows and 33 columns in the Q-tables. Q-learning's parameters are $\lambda = 0.9$, $\gamma = 1.0$, and learning rate 0.001 (these worked well for smaller problems Q-learning was able to solve).

This Q-learning variant *completely failed to achieve significant performance improvement on our complex task,* however. So we tried to make the problem easier (less non-Markovian) by extending the agent's sensing capabilities. Now, each possible input tells the agent uniquely whether it has/hasn't got the key, and whether the closest object (obstacle or key or door or goal) part of which is either 10 or 20 or 30 or 40 or 50 pixels away in front of/45 degrees to the right of/45 degrees to the left of the agent is obstacle/key/door/goal/non-existent, and if existing, whether it is 10/20/30/40/50 pixels away. All this can be efficiently coded by $21 * 21 * 21 * 2 = 18522$ different inputs corresponding to 18522 different rows in the Q-tables (with a total of 611226 entries). *This worked a bit better than the simpler Q-variant.* Still, we were not able to make Q-learning achieve very significant performance improvement. The experimental results are shown in Figure 5.3.

Results with IS. The system dies after 10^9 basic cycles (ca. 130,000 trials corresponding to only 130,000 reinforcement signals). By then, average reinforcement intake per agent is 7.5×10^{-4} per basic cycle (mean of 4 simulations). This is about 62 times faster than the initial speed (the one of random behavior), and is roughly $\frac{1}{4}$ to $\frac{1}{3}$ of the optimal speed (due to the POMDP setting and the random agent initializations at trial start, however, it is very hard to calculate optimal average speed). Results are shown in Figures 5.3 and 5.3.

Q-learning as an instruction for IS. The fact that Q-learning is not designed for POMDPs does not mean that Q-learning cannot be plugged into IS as a useful instruction. To examine this issue, we add Q-learning to the instruction list to be used by IS:

b_{13}: *Q-learning(w_1)* — with probability $\frac{w_1}{200 * n_{ops}}$, keep executing actions according to the Q-table until there is non-zero reinforcement, and update the Q-table according to standard Q-learning rules (Watkins and Dayan, 1992). Otherwise, execute only one single action according to the current Q-table.

Interestingly, this combination leads to even slightly better results near system death (see Figure 5.3). Essentially, the system learns *when* to trust the Q-table.

Stack size. Final stack size per agent was never higher than 250, corresponding to only about 250 still valid policy modifications. Space limitations prevent us from describing many interesting details such as the final, complex shape of the agent's probability matrices, and how the agents use memories (embodied by their *IP*s) to disambiguate ambiguous inputs, and how they use self-modifications to adapt the frequency of self-modifications.

6 Conclusion

During each backtracking process, MRL implicitly evaluates each still valid policy modification as to whether it belongs to a block of modifications whose beginning has been followed by long-term performance improvement. If there is empirical evidence to the contrary, then backtracking invalidates policy modifications until the history of valid modifications is again a success story (in the worst case an empty one — this will be reflected by an empty stack). The success of a policy modification or bias shift partly depends on the sucess of later bias shifts for which it set the stage ("metalearning"). MRL works no matter what the environment and the internal state are like. MRL is efficient in the sense that only the two most recent ("topmost") still valid modification blocks need to be considered at a given time in a backtracking process. A *single* backtracking process, however, may invalidate *many* modification blocks. MRL is general — you can plug in your favorite learning algorithm L as an action. This makes sense especially in situations where the applicability of L is questionable because the environment does not satisfy the preconditions that would make L sound. MRL can at least guarantee that those of L's policy modifications that appear to contribute to negative long-term effects are countermanded. This is more than can be said about previous reinforcement learning schemes.

We don't gain much by applying MRL to, say, simple "Markovian" mazes for which there already are efficient reinforcement learning methods based on dynamic programming. MRL is of interest, however, in more realistic situations where standard reinforcement learning methods fail (such as the 2-door/2-key problem from section 5.3).

We feel that we have barely scratched MRL's potential. Future work will focus on plugging a whole variety of well-known learning algorithms into MRL, and let it pick and combine the best, problem-specific ones.

7 Acknowledgments

Thanks for valuable discussions to Sepp Hochreiter, Marco Dorigo, Luca Gambardella, Rafał Sałustowicz. This work was supported by SNF grant 21-43'417.95 "incremental self-improvement".

References

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

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London.

Boddy, M. and Dean, T. L. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285.

Caruana, R., Silver, D. L., Baxter, J., Mitchell, T. M., Pratt, L. Y., and Thrun, S. (1995). Learning to learn: knowledge consolidation and transfer in inductive systems. Workshop held at NIPS-95, Vail, CO, see http://www.cs.cmu.edu/afs/user/caruana/pub/transfer.html.

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

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*, Hillsdale NJ. Lawrence Erlbaum Associates.

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.

Gittins, J. C. (1989). *Multi-armed Bandit Allocation Indices*. Wiley-Interscience series in systems and optimization. Wiley, Chichester, NY.

Greiner, R. (1996). PALO: A probabilistic hill-climbing algorithm. *Artificial Intelligence*, 83(2).

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*. MIT Press, Cambridge MA.

Kaelbling, L. (1993). *Learning in Embedded Systems*. MIT Press.

Kaelbling, L., Littman, M., and Cassandra, A. (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 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.

Kumar, P. R. and Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice Hall.

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

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

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.

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

McCallum, R. A. (1995). Instance-based utile distinctions for reinforcement learning with hidden state. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 387–395. Morgan Kaufmann Publishers, San Francisco, CA.

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

Rosenbloom, P. S., Laird, J. E., and Newell, A. (1993). *The SOAR Papers*. MIT Press.

Russell, S. and Wefald, E. (1991). Principles of Metareasoning. *Artificial Intelligence*, 49:361–395.

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

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. Morgan Kaufmann Publishers, San Francisco, CA.

Schmidhuber, J. (1996). A general method for incremental self-improvement and multi-agent learning in unrestricted environments. In Yao, X., editor, *Evolutionary Computation: Theory and Applications*. Scientific Publ. Co., Singapore.

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.

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

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

Watanabe, O. (1992). *Kolmogorov complexity and computational complexity*. EATCS Monographs on Theoretical Computer Science, Springer.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

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

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

Zhao, J. and Schmidhuber, J. (1996). Incremental self-improvement for life-time multi-agent reinforcement learning. In *Proc. SAB'96*. MIT Press, Cambridge MA. To appear.

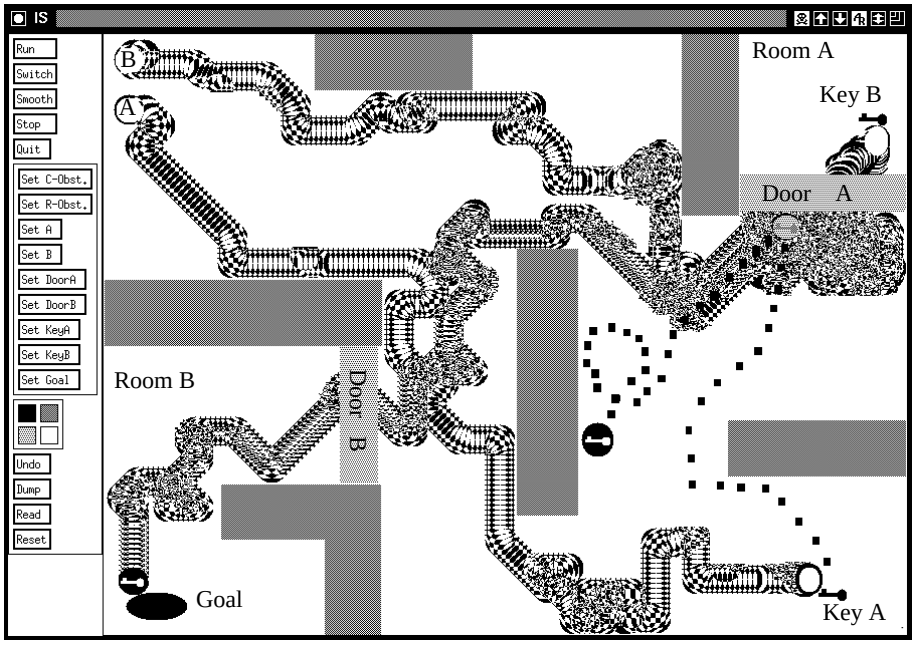

Figure 6: *Snapshot of traces of interacting, learning agents A and B in a partially observable environment with huge state space. Initially, both agents are randomly rotated and placed near the northwest corner. Agent A first moves towards the southeast corner to grab its key, then moves north (dotted trace) to open door A (grey). Simultaneously, agent B moves east to door A and waits for agent A to open it. Then B moves in, grabs key B, turns, and heads towards door B to open it, while agent A also heads southwest in direction of the goal (dotted trace). This time, however, agent B is the one who touches the goal first, because A fails to quickly circumvent the obstacle in the center. All this complex, partly stochastic behavior is learned solely by "self-referential" policy modifications (generated according to the "self-referential" policy itself via* IncProb *calls), although strongly delayed reinforcement is provided only if one of the agents touches the goal.*

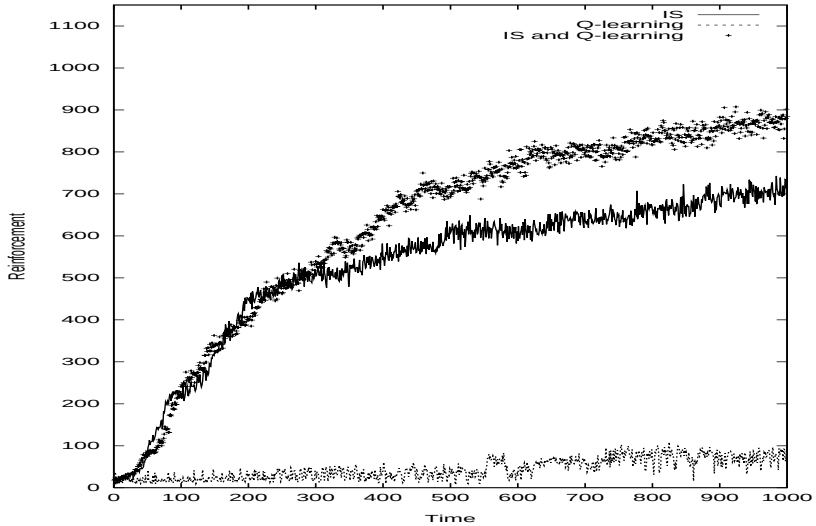

Figure 7: *Performance of incremental self-improvement (IS) compared to the one of the best Q-learning variant (reinforcement intake sampled at intervals of* 10^6 *instruction cycles, mean of 4 simulations). Q-learning hardly improves, while IS makes rather quick, substantial progress. Interestingly, adding Q-learning to IS's instruction set again tends to improve late-life performance a bit (trace of crosses) — essentially, the system learns when to trust/ignore the Q-table.*