Machine Learning, 28, 105–132 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Shifting Inductive Bias with Success-Story Algorithm, Adaptive Levin Search, and Incremental Self-Improvement

JÜRGEN SCHMIDHUBER juergen@idsia.ch
JIEYU ZHAO jieyu@idsia.ch
MARCO WIERING marco@idsia.ch
IDSIA, Corso Elvezia 36, CH-6900-Lugano, Switzerland

Editors: Lorien Pratt and Sebastian Thrun

Abstract. We study task sequences that allow for speeding up the learner's average reward intake through appropriate shifts of inductive bias (changes of the learner's policy). To evaluate long-term effects of bias shifts setting the stage for later bias shifts we use the "success-story algorithm" (SSA). SSA is occasionally called at times that may depend on the policy itself. It uses backtracking to undo those bias shifts that have not been empirically observed to trigger long-term reward accelerations (measured up until the current SSA call). Bias shifts that survive SSA represent a lifelong success history. Until the next SSA call, they are considered useful and build the basis for additional bias shifts. SSA allows for plugging in a wide variety of learning algorithms. We plug in (1) a novel, adaptive extension of Levin search and (2) a method for embedding the learner's policy modification strategy within the policy itself (incremental self-improvement). Our inductive transfer case studies involve complex, partially observable environments where traditional reinforcement learning fails.

Keywords: inductive bias, reinforcement learning, reward acceleration, Levin search, success-story algorithm, incremental self-improvement

1. Introduction / Overview

Fundamental transfer limitations. Inductive transfer of knowledge from one task solution to the next (e.g., Caruana et al. 1995, Pratt and Jennings 1996) requires the solutions to share mutual algorithmic information. Since almost all sequences of solutions to well-defined problems are incompressible and have maximal Kolmogorov complexity (Solomonoff, 1964, Kolmogorov, 1965, Chaitin, 1969, Li and Vitányi, 1993), arbitrary task solutions almost never share mutual information. This implies that inductive transfer and "generalization" are almost always impossible — see, e.g., Schmidhuber (1997a); for related results see Wolpert (1996). From a practical point of view, however, even the presence of mutual information is no guarantee of successful transfer. This is because concepts such as Kolmogorov complexity and algorithmic information do not take into account the time consumed by learning algorithms computing a new task's solution from previous ones. In typical machine learning applications, however, it is precisely the learning time that we want to minimize.

Reward acceleration. Given the observations above, all attempts at successful transfer must be limited to task sequences of a particularly friendly kind. In the context of reinforcement learning (RL) we will focus on task sequences that allow for speeding up the learner's long-term average reward intake. Fortunately, in our own highly atypical and regular universe such task sequences abound. For instance, often we encounter situations where high reward for some problem's solution can be achieved more quickly by first learning easier but related tasks yielding less reward.

Our learner's single life lasts from time 0 to time T (time is not reset in case of new learning trials). Each modification of its policy corresponds to a shift of inductive bias (Utgoff, 1986). By definition, "good" bias shifts are those that help to accelerate long-term average reward intake. The learner's method for generating good bias shifts must take into account: **(1)** Bias shifts occurring early in the learner's life generally influence the probabilities of later bias shifts. **(2)** "Learning" (modifying the policy) and policy tests will consume part of the learner's limited life-time[1].

Previous RL approaches. To deal with issues (1) and (2), what can we learn from traditional RL approaches? Convergence theorems for existing RL algorithms such as Q-learning (Watkins and Dayan, 1992) require infinite sampling size as well as strong (usually Markovian) assumptions about the environment, e.g., (Sutton, 1988, Watkins and Dayan, 1992, Williams, 1992). They are of great theoretical interest but not extremely relevant to our realistic limited life case. For instance, there is no proof that Q-learning will converge within finite given time, not even in Markovian environments. Also, previous RL approaches do not consider the computation time consumed by learning and policy tests in their objective function. And they do not explicitly measure long-term effects of early learning on later learning.

Basic ideas (see details in section 2). To address issues (1) and (2), we treat learning algorithms just like other time-consuming actions. Their probabilities of being executed at a given time may 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 or bias shifts, information necessary to restore the old policy is pushed on a stack. At any given time in system life there is only one single training example to estimate the long-term usefulness of any previous bias shift B — namely the reward per time since then. This includes all the reward collected after later bias shifts for which B may have set the stage, thus providing a simple measure of earlier learning's usefulness for later learning. Occasionally the "success-story algorithm" (SSA) uses backtracking to undo those policy modifications that have not been empirically observed[2] to trigger long-term reward accelerations (measured up until the current SSA call). For instance, certain bias shifts may have been too specifically tailored to previous tasks ("overfitting") and may be harmful for future inductive transfer. Those bias shifts that survive SSA represent a lifelong success history. Until the next SSA call, they will build the basis for additional bias shifts and get another chance to justify their existence.

Due to unknown reward delays, there is no *a priori* good way of triggering SSA calls. In principle, however, it is possible to build policies that can *learn* to trigger SSA calls. Since learning algorithms are actions and can be combined (according to the policy) to form more complex learning algorithms, SSA also allows for embedding the learning strategy within the policy itself. There is no pre-wired difference between "learning", "metalearning", "metametalearning" etc.[3]

Outline of remainder. Section 2 will describe the learner's basic cycle of operations and SSA details. It will explain how lifelong histories of reward accelerations can be enforced despite possible interference from parallel internal or external processes. Sections 3 and 4 will present two concrete implementations and inductive transfer experiments with complex, partially observable environments (POEs). Some of our POEs are bigger and more complex than POEs considered in most previous POE work.

2. Basic Set-Up and SSA

Reward/Goal. Occasionally E provides real-valued reward. $R(t)$ is the cumulative reward obtained between time 0 and time $t > 0$, where $R(0) = 0$. At time t the learner's goal is to accelerate long-term reward intake: it wants to let $\frac{R(T)-R(t)}{T-t}$ exceed the current average reward intake. To compute the "current average reward intake" a previous point $t' < t$ to compute $\frac{R(t)-R(t')}{t-t'}$ is required. How to specify t' in a general yet reasonable way? For instance, if life consists of many successive "trials" with non-deterministic outcome, how many trials must we look back in time? This question will be addressed by the success-story algorithm (SSA) below.

Initialization. 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.

Basic cycle. Between time 0 (system birth) and time T (system death) the following basic cycle is repeated over and over again:

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** call SSA, which backtracks and undoes certain previous Pol-modifications if necessary (to ensure that the history of still valid modifications corresponds to a history of reward accelerations):

   SSA.1. Set variable t equal to current time.

IF there is no *"tag"* (a pair of time and cumulative reward 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 SSA call).

ELSE denote the topmost tag in $\mathcal{S}$ by (t', $R(t')$). Denote the one below by (t'', $R(t'')$) (if there is not any tag below, set variable $t'' = 0$ — recall $R(t'') = R(0) = 0$).

SSA.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 SSA call.

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 was be before time t'. Then also pop off the tag (t', $R(t')$). Go to **SSA.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**.

SSA ensures life-time success stories. 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 **SSA.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 SSA call will have enforced the following "success-story criterion" SSC (t is the t in the most recent step **SSA.2**):

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

SSC demands that each still valid time marks the beginning of a long-term reward acceleration measured up to the current time t. Each Pol-modification that survived SSA represents a bias shift whose start marks a long-term reward speed-up. 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.

SSA's generalization assumption. Since life is one-way (time is never reset), during each SSA call the system has to generalize from a *single* experience concerning the usefulness of any previous policy modification: the average reward per time since then. At the end of each SSA call, until the beginning of the next one, the only temporary generalization assumption for inductive inference is: Pol-modifications that survived all previous SSA calls will remain useful. In absence of empirical evidence to the contrary, each still valid sequence of Pol-modifications is assumed to have successfully set the stage for later ones. What has appeared useful so far will get another chance to justify itself.

When will SSC be satisfiable in a non-trivial way? In irregular and random environments there is no way of justifying permanent policy modifications by SSC. Also, a trivial way of satisfying SSC is to never make a modification. Let us assume, however, that E , $\mathcal{I}$, and action set $\mathcal{A}$ (representing the system's initial bias) do indeed allow for *Pol*-modifications triggering long-term reward accelerations. This is an instruction set-dependent assumption much weaker than the typical Markovian assumptions made in previous RL work, e.g., (Kumar and Varaiya, 1986, Sutton, 1988, Watkins and Dayan, 1992, Williams, 1992). Now, if we prevent all instruction probabilities from vanishing (see concrete implementations in sections 3/4), then the system will execute *Pol*-modifications occasionally, and keep those consistent with SSC. In this sense, it cannot help getting better. Essentially, the system keeps generating and undoing policy modifications until it discovers some that indeed fit its generalization assumption.

Greediness? SSA's strategy appears greedy. 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 reward delays, however, the degree of greediness is learnable — SSA calls may be triggered or delayed according to the modifiable policy itself.

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. In section 3 we will actually plug in an adaptive Levin search extension. Similarly, actions may be calls of a Q-learning variant — see experiments in (Schmidhuber et al., 1996). Plugging Q into SSA makes sense in situations where Q by itself is questionable because the environment might not satisfy the preconditions that would make Q sound. SSA will ensure, however, that at least each policy change in the history of all still valid policy changes will represent a long-term improvement, even in non-Markovian settings.

Limitations. (1) In general environments neither SSA nor any other scheme is guaranteed to continually increase reward intake per *fixed* time interval, or to find the policy that will lead to maximal cumulative reward. (2) 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 many other, less general RL schemes though (recall that Q-learning is not guaranteed to converge in finite time).

Outline of remainder. Most of our paper will be about plugging various policy-modifying algorithms into the basic cycle. Despite possible implementation-specific complexities the overall concept is very simple. Sections 3 and 4 will describe two concrete implementations. 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 policies can be built (the system's modifiable learning strategy is able to modify itself). Experimental case studies will involve complex environments where standard RL algorithms fail. Section 5 will conclude.

3. Implementation 1: Plugging LS into SSA

Overview. In this section we introduce an adaptive extension of Levin search (LS) (Levin, 1973, Levin, 1984) as only learning action to be plugged into the basic cycle. We apply it to partially observable environments (POEs) which recently received a lot of attention in the RL community, e.g., (Whitehead and Ballard, 1990, Schmidhuber, 1991, Chrisman, 1992, Lin, 1993, Littman, 1994, Cliff and Ross, 1994, Ring, 1994, Jaakkola et al., 1995, Kaelbling et al., 1995, McCallum, 1995). 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 combine it with SSA. In an experimental case study we show dramatic search time reduction for sequences of more and more complex POEs ("inductive transfer").

3.1. 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. See (Li and Vitányi, 1993) for an overview. See (Schmidhuber 1995, 1997a) 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 $Kt(s) = \min_q\{-log D_P(q) + log\ t(q,s)\}$, 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 3.2, however, we will make the distribution variable).

Optimality. Given primitives representing a universal programming language, for a broad class of 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). More formally: a problem is a symbol string that conveys all information about another symbol string called its solution, where the solution can be extracted by some (search) algorithm, given the problem. Suppose there is an algorithm that solves certain time-limited optimization problems or inversion problems in $O(f(n))$ steps, where f is a total recursive function and n is a positive integer representing problem size. Then universal LS will solve the same problems in at most $O(f(n))$ steps (although a large constant may be buried in the $O()$ notation). Despite this strong result, until recently LS has not received much attention except in purely theoretical studies — see, e.g., (Watanabe, 1992).

Of course, LS and any other algorithm will fail to quickly solve problems whose solutions all have high algorithmic complexity. Unfortunately, almost all possible problems are of this kind (Kolmogorov, 1965, Chaitin, 1969, Solomonoff, 1964). In fact, the realm of practical computer science is limited to solving the comparatively few tasks with low-complexity solutions. Fortunately such tasks are rather common in our regular universe.

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).

3.2. 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., (Narendra and Thathatchar, 1974, 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 $P_{ij} := (1 - \gamma)P_{ij}$
```

3.3. Plugging ALS into the Basic SSA Cycle

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 3.5), there is no 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 the basic cycle from section 2. SSA ensures that the system will keep only probability modifications representing a lifelong history of performance improvements.

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

1. If the current basic cycle's problem is N_i, then set $q :=$ **Levin search** (N_i, P). If a solution was found, generate reward 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, the 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 SSA. To begin with, in an illustrative application with a partially observable maze that has more states and obstacles than those presented in other POE work (see, e.g., (Cliff and Ross, 1994)), we will show how LS by itself can solve POEs with large state spaces but low-complexity solutions (Q-learning variants fail to solve these tasks). Then we will present experimental case studies with multiple, more and more difficult tasks (inductive transfer). ALS can use previous experience to speed-up the process of finding new solutions, and ALS plugged into the SSA cycle (SSA+ALS for short) always outperforms ALS by itself.

3.4. Experiment 1: A Big Partially Observable Maze (POM)

The current section is a prelude to section 3.5 which will address inductive transfer issues. Here we will only show that LS by itself can be useful for POE problems. See also (Wiering and Schmidhuber, 1996).

Task. Figure 1 shows a 39×38-maze with 952 free fields, a single start position (S) and a single goal position (G). The maze has more fields and obstacles than mazes used by previous authors working on POMs — 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 : *Cond* = front is blocked, *Dir* = left.
Instruction 2 : *Cond* = front is blocked, *Dir* = right.
Instruction 3 : *Cond* = left field is free, *Dir* = left.
Instruction 4 : *Cond* = left field is free, *Dir* = right.
Instruction 5 : *Cond* = left field is free, *Dir* = none.
Instruction 6 : *Cond* = right field is free, *Dir* = left.
Instruction 7 : *Cond* = right field is free, *Dir* = right.
Instruction 8 : *Cond* = right field is free, *Dir* = none.
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`

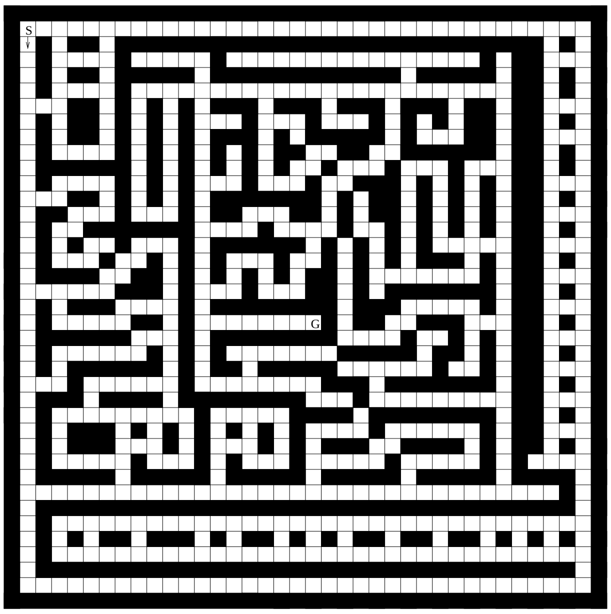


Figure 1. An apparently complex, partially observable 39×38*-maze with a low-complexity shortest path from start S to goal G involving 127 steps. Despite the relatively large state space, the agent can implicitly perceive only one of three highly ambiguous types of input, namely "front is blocked or 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 POE problem. The S and the arrow indicate the agent's initial position and rotation.*

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 do not halt, by preventing them from consuming too much time.

As mentioned in section 3.1, 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 POE problem? Because the instructions above are not sufficient to tell the agent exactly where it is: at any 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 at the beginning of this section). 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 POEs. 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 POE problem any more). The first Q-variant was just tested to see how much more difficult the problem becomes in the POE 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 — compare instruction list at the beginning of this section). 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). Q-learning's reward is 1.0 for finding the goal and -0.01 for each collision. 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 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 POE: 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)$^9$
```

In words: *Repeat the following action sequence 6 times: go forward until you see the fifth consecutive opening to the left; then rotate left.* 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 (Cramer, 1985, Dickmanns et al., 1987, Koza, 1992, Fogel et al., 1966).

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 3.2), 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 (unless such updates lead to an improved solution — this would be relevant in case we do not stop LS after the first solution has been found). 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.

3.5. Experiment 2: Incremental Learning / Inductive Transfer

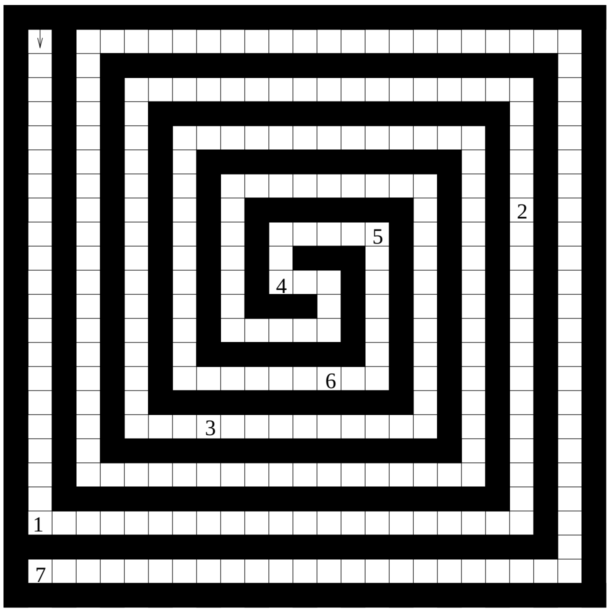


Figure 2. A 23 × 23 labyrinth. The arrow indicates the agent's initial position and direction. Numbers indicate goal positions. The higher the number, the more difficult the goal. The agent's task is to find all goal positions in a given "goalset". Goalsets change over time.

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 SSA.

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 distances to the start. Finding a goal yields reward 1.0 divided by solution path length (short paths preferred). There is no negative reward for collisions. During each epoch, we update the probability matrices P, $\bar{P}$ and $\hat{P}$ whenever a goal is found (for all epochs dealing with goalset G_n there are $n + 3$ updates). 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 do not 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 SSA+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 SSA+ALS, we tried all learning rates γ in {0.01, 0.02,..., 0.95}. We found that SSA+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 SSA (and METHOD 2) only small learning rates are feasible – large learning rate subspaces do not work for many goals. Thus, the first type of SSA-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 SSA+ALS performs best with $\gamma = 0.45$, with additional uniform noise in $[-0.05, 0.05]$ (noise tends to improve SSA+ALS's performance a little bit, but worsens ALS' performance). With METHOD 2, ALS performs best with $\gamma = 0.05$, and SSA+ALS performs best with $\gamma = 0.2$ 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, SSA+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.

SSA+ALS versus ALS. SSA+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,

Table 1. For METHODs 1 and 2, we list the number of steps (in thousands) required by LS, ALS, SSA+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 topmost 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 SSA+ALS achieves additional speed-ups. SSA+ALS works well for all learning rates, ALS by itself does not. Also, all our incremental learning procedures clearly outperform LS by itself.

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
SSA + ALS	1	12.2	237	331	405
ALS	2	13.0	487	192	289
SSA + ALS	2	11.5	345	85	230

however, that there are many learning rates where ALS by itself completely fails, while SSA+ALS does not. SSA+ALS is more robust.

Example of bias shifts undone. For optimal learning rates, the biggest speed-up occurs for G_3. Here SSA decreases search costs dramatically: after goal 5 is found, the policy "overfits" in the sense that it is too much biased towards problem 5's optimal (lowest complexity) solution: *(1) Repeat step forward until blocked, rotate left. (2) Jump (1,11). (3) Repeat step forward until blocked, rotate right. (4) Repeat step forward until blocked, rotate right.* Problem 6's optimal solution can be obtained from this by replacing the final instruction by *(4) Jump (3,3).* This represents a significant change though (3 probability distributions) and requires time. Problem 5, however, can also be solved by replacing its lowest complexity solution's final instruction by *(4) Jump (3,1).* This increases complexity but makes learning problem 6 easier, because less change is required. After problem 5 has been solved using the lowest complexity solution, SSA eventually suspects "overfitting" because too much computation time goes by without sufficient new rewards. Before discovering goal 6, SSA undoes apparently harmful probability shifts until SSC is satisfied again. This makes *Jump* instructions more likely and speeds up the search for a solution to problem 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 SSA 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

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

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

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

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
SSA + ALS	1	1,050	984	6.23
SSA + ALS	2	873	671	9.92

in table 1). Using SSA 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}). SSA always improves the results. For the total number of steps — which is an almost linear function of the time consumed during the simulation — the SSA-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 inductive transfer, however, we obtain large speed-up factors. METHOD 1 with SSA+ALS improves performance by a factor in excess of 40 (see results of SSA+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 SSA keeps initial search costs significantly lower.

The safety net effect. Figure 3(B) plots epoch numbers against average probability of programs computing solutions. With METHOD 1, SSA+ALS tends to keep the probabilities lower than ALS by itself: high program probabilities are not

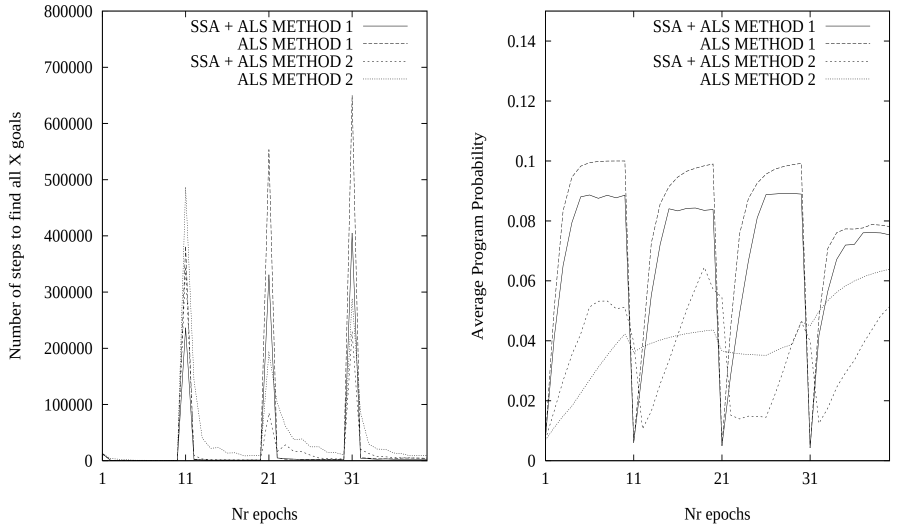


Figure 3. (A) Average number of steps per epoch required to find all of the current goalset's goals, plotted against epoch numbers. Peaks reflect goalset changes. (B) Average probability of programs computing solutions (before solutions are actually found).

always beneficial. With METHOD 2, SSA undoes many policy modifications when goalsets change, thus keeping the policy flexible and reducing initial search costs.

Effectively, SSA 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 SSA+ALS (therefore, 61 is SSA+ALS's total final stack size). For METHOD 2, the corresponding numbers are 996 and 63. We see that SSA keeps only about 16% respectively 6% of all modifications. The remaining modifications are deemed unworthy because they have not been observed to trigger lifelong reward speed-ups. Clearly, SSA prevents ALS from overdoing its policy modifications ("safety net effect"). This is SSA's simple, basic purpose: undo certain learning algorithms' policy changes and bias shifts once they start looking harmful in terms of long-term reward/time ratios.

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

4. 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, 1997b; 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-modifying policy itself. SSA is used to keep only those self-modifications followed by reward speed-ups, in particular those leading to "better" future self-modifications, recursively. We call this "incremental self-improvement" (IS).

Outline of section. Subsection 4.1 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-modification instructions, and how SSA keeps only the "good" policy modifications. Subsection 4.2 will describe an experimental inductive transfer case study where we apply IS to a sequence of more and more difficult function approximation tasks. Subsection 4.3 will mention additional IS experiments involving complex POEs and interacting learning agents that influence each other's task difficulties.

4.1. 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. In the following incomplete list of instructions $(b_0, \ldots, b_3)$ 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).

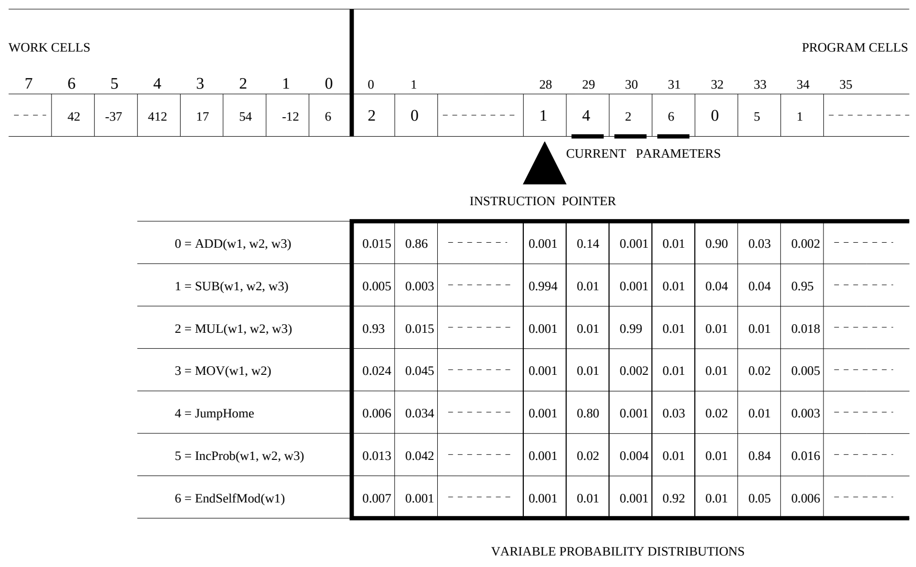
b_1: *Sub*$(w_1, w_2, w_3) : c_{w_3} \leftarrow c_{w_1} - c_{w_2}$.

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).*

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. See Figure 4.



7	6	5	4	3	2	1	0	0	1		28	29	30	31	32	33	34	35
- - - -	42	-37	412	17	54	-12	6	2	0	- - - - - - - -	1	4	2	6	0	5	1	- - - - - - - - -

0 = ADD(w1, w2, w3)	0.015	0.86	- - - - - - -	0.001	0.14	0.001	0.01	0.90	0.03	0.002	- - - - - - -
1 = SUB(w1, w2, w3)	0.005	0.003	- - - - - - -	0.994	0.01	0.001	0.01	0.04	0.04	0.95	- - - - - - -
2 = MUL(w1, w2, w3)	0.93	0.015	- - - - - - -	0.001	0.01	0.99	0.01	0.01	0.01	0.018	- - - - - - -
3 = MOV(w1, w2)	0.024	0.045	- - - - - - -	0.001	0.01	0.002	0.01	0.01	0.02	0.005	- - - - - - -
4 = JumpHome	0.006	0.034	- - - - - - -	0.001	0.80	0.001	0.03	0.02	0.01	0.003	- - - - - - -
5 = IncProb(w1, w2, w3)	0.013	0.042	- - - - - - -	0.001	0.02	0.004	0.01	0.01	0.84	0.016	- - - - - - -
6 = EndSelfMod(w1)	0.007	0.001	- - - - - - -	0.001	0.01	0.001	0.92	0.01	0.05	0.006	- - - - - - -

Figure 4. Snapshot of parts of policy and storage. IP currently points to program cell 28. The integer sequence 1 4 2 6 (generated according to the policy's current probability distributions) will be interpreted as Sub(4, 2, 6) *— subtract the contents of work cell 4 from the contents of work cell 4 and put the result into work cell 6.*

Self-modifications. To obtain a learner that can explicitly modify its own policy (by running its own learning strategies), we introduce a special self-modification 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 SSA call (see section 2) begins an SMS. The SMS ends by executing another yet unmentioned 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 reward signals have been received — this will satisfy the EVALUATION CRITERION in the basic 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 SSA. We replace step 1 in the basic 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 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 reward 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 SSA 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.

4.2. Experiment 3: Function Approximation / Inductive Transfer

This experimental case study will demonstrate that IS can successfully learn in a changing environment where the tasks to be solved become more and more difficult over time (inductive transfer).

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 is decomposable 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 reward 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 4.1):
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. SSA is called after each 5th consecutive non-zero reward 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 5) 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:

$\underline{1\ 2\ 1\ 6}\ \underline{1\ 0\ 6\ 6}\ \underline{2\ 6\ 6\ 6}\ \underline{2\ 6\ 6\ 6}\ \underline{2\ 6\ 6\ 6}\ \underline{4 * * *}\ldots$

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

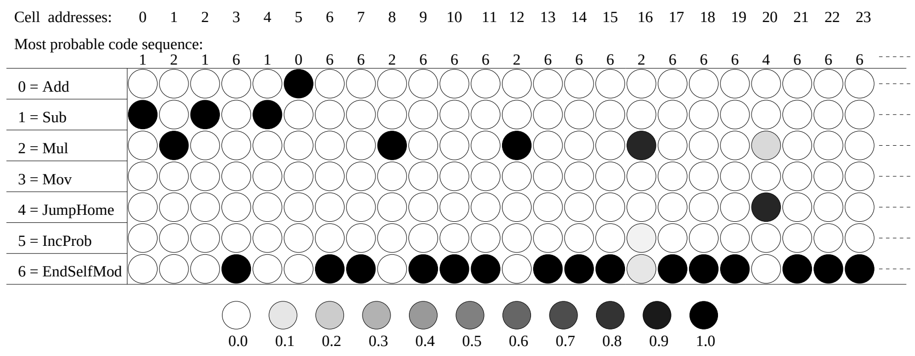


Figure 5. The final state of the probability matrix for the function learning problem. Grey scales indicate the magnitude of probabilities of instructions and parameters. The matrix was computed by self-modification sequences generated according to the matrix itself (initially, all probability distributions were maximum entropy distributions).

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 6 gives a vague

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

	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	(–, –, –,)	$0 \Longrightarrow IP$

idea of what is going on by showing a typical plot of the frequency of *IncProb* instructions during system life (sampled at intervals of 10^6 basic 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. This is reminiscent of Schwefel's work (1974) on self-adjusting mutation rates. One major novelty is the adaptive, highly non-uniform distribution of self-modifications on "promising" individual policy components.

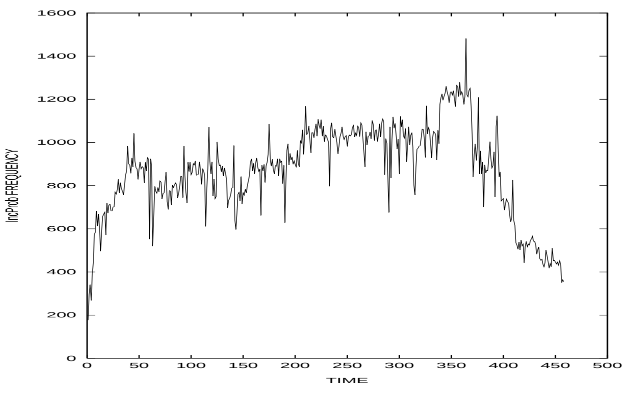


Figure 6. Numbers of executed self-modifying instructions plotted against time, sampled at intervals of 10^6 instruction cycles. The graph reflects that the system soon uses self-modifying instructions to increase the frequency of self-modifying instructions. Near system death the system learns that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

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 much tailored to previous tasks. In the end, there are 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.

4.3. Other Experiments with IS: Overview

In the previous experiments the learner's environment changed because of externally induced task changes. We also did experimental case studies where the tasks change in a less obvious way because of other changing learners.

A complex POE. Schmidhuber et al. (1996) describe two agents A and B living in a partially observable 600×500 pixel environment with obstacles. They learn to solve a complex task that could not be solved by various TD(λ) Q-learning variants (Lin, 1993). The task requires (1) agent A to find and take a key "key A"; (2) agent A go to a door "door A" and open it for agent B; (3) agent B to enter through "door A", find and take another key "key B"; (4) agent B to go to another door "door B" to open it (to free the way to the goal); (5) one of the agents to reach the goal. Both agents share the same 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 50 pixels in front of it. The agent can also move forward, turn around, turn relative to its key or its door or the goal. It can use memory (embodied by its *IP*) to disambiguate inputs (unlike Jaakkola et al.'s method (1995), ours is not limited to finding suboptimal stochastic policies for POEs with an optimal solution). Reward is provided only if one of the agents touches the goal. This agent's reward is 5.0; the other's is 3.0 (for its cooperation — note that asymmetric reward introduces competition).

In the beginning, the goal is found only every 300,000 basic cycles. Through self-modifications and SSA, however, within 130,000 trials (10^9 basic cycles) the average trial length decreases by a factor of 60 (mean of 4 simulations). Both agents learn to cooperate to accelerate reward intake. See (Schmidhuber et al., 1996) for details.

Zero sum games. Even certain zero sum reward tasks allow for achieving success stories. This has been shown in an experiment with three IS-based agents (Zhao and Schmidhuber, 1996): each agent is both predator and prey; it receives reward 1 for catching its prey and reward -1 for being caught. Since all agents learn each agent's task gets more and more difficult over time. How can it then create a non-trivial history of policy modifications, each corresponding to a lifelong reward acceleration? The answer is: each agent collects a lot of negative reward during its life, and actually comes up with a history of policy modifications causing less and less negative cumulative long-term rewards. The stacks of all agents tend to grow continually as they discover better and better pursuit-evasion strategies.

5. Conclusion

SSA collects more information than previous RL schemes about long-term effects of policy changes and shifts of inductive bias. In contrast to traditional RL approaches, time is not reset at trial boundaries. Instead we measure the total reward received and the total time consumed by learning and policy tests during all trials following some bias shift: bias shifts are evaluated by measuring their long-term effects on later learning. Bias shifts are undone once there is empirical evidence that they have not set the stage for long-term performance improvement. No bias shift is safe forever, but in many regular environments the survival probabilities of useful bias shifts will approach unity if they can justify themselves by contributing to long-term reward accelerations.

Limitations. (1) Like any approach to inductive transfer ours suffers from the fundamental limitations mentioned in the first paragraph of this paper. (2) Especially in the beginning of the training phase ALS may suffer from a possibly large constant buried in the $O()$ notation used to describe LS' optimal order of complexity. (3) We do not gain much by applying our methods to, say, simple "Markovian" mazes for which there already are efficient RL methods based on dynamic programming (our methods are of interest, however, in certain more realistic situations where standard RL methods fail). (4) SSA does not make much sense in "unfriendly" environments in which reward constantly decreases no matter what the learner does. In such environments SSC will be satisfiable only in a trivial way. True success stories will be possible only in "friendly", regular environments that do allow for long-term reward speed-ups (this does include certain zero sum reward games though).

Outlook. Despite these limitations we feel that we have barely scratched SSA's potential for solving realistic RL problems involving inductive transfer. In future work we intend to plug a whole variety of well-known algorithms into SSA, and let it pick and combine the best, problem-specific ones.

6. 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".

Notes

1. Most previous work on limited resource scenarios focuses on bandit problems, e.g., Berry and Fristedt (1985), Gittins (1989), and references therein: you have got a limited amount of money; how do you use it to figure out the expected return of certain simple gambling automata and exploit this knowledge to maximize your reward? See also (Russell and Wefald, 1991, Boddy and Dean, 1994, Greiner, 1996) for limited resource studies in planning contexts. Unfortunately the corresponding theorems are not applicable to our more general lifelong learning scenario.
2. This may be termed "metalearning" or "learning to learn". 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.

3. For alternative views of metalearning see, e.g., Lenat (1983), Rosenbloom et al. (1993). For instance, Lenat's approach requires occasional human interaction defining the "interestingness" of concepts to be explored. No previous approach, however, attempts to evaluate policy changes by measuring their long-term effects on later learning in terms of reward intake speed.

References

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.

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

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.

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

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*, pages 345–352. 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.

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

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.

Narendra, K. S. and Thathatchar, M. A. L. (1974). Learning automata - a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323–334.

Pratt, L. and Jennings, B. (1996). A survey of transfer between connectionist networks. *Connection Science*, 8(2):163–184.

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 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. (1997a). Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks.* In press.

Schmidhuber, J. (1997b). 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. In press.

Schmidhuber, J., Zhao, J., and Wiering, M. (1996). Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA.

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

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 Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning*, volume 2, pages 163–190. 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.

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

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*, pages 534–542. Morgan Kaufmann Publishers, San Francisco, CA.

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

Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390.

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

Received August 21, 1996
Accepted February 24, 1997
Final Manuscript March 1, 1997