REINFORCEMENT LEARNING WITH SELF-MODIFYING POLICIES

In S. Thrun and L. Pratt, eds., “Learning to learn”, pages 293–309, Kluwer, 1997

Jürgen Schmidhuber, Jieyu Zhao, Nicol N. Schraudolph
IDSIA, Corso Elvezia 36, CH-6900 Lugano, Switzerland

Abstract

A learner's modifiable components are called its policy. An algorithm that modifies the policy is a learning algorithm. If the learning algorithm has modifiable components represented as part of the policy, then we speak of a self-modifying policy (SMP). SMPs can modify the way they modify themselves etc. They are of interest in situations where the initial learning algorithm itself can be improved by experience — this is what we call “learning to learn”. How can we force some (stochastic) SMP to trigger better and better self-modifications? The *success-story algorithm* (SSA) addresses this question in a lifelong reinforcement learning context. During the learner's life-time, SSA is occasionally called at times computed according to SMP itself. SSA uses backtracking to undo those SMP-generated SMP-modifications that have not been empirically observed to trigger lifelong reward accelerations (measured up until the current SSA call — this evaluates the long-term effects of SMP-modifications setting the stage for later SMP-modifications). SMP-modifications that survive SSA represent a lifelong success history. Until the next SSA call, they build the basis for additional SMP-modifications. Solely by self-modifications our SMP/SSA-based learners solve a complex task in a partially observable environment (POE) whose state space is far bigger than most reported in the POE literature.

1 Introduction

Bias towards algorithmic regularity. There is no miraculous universal learning algorithm that will perform well in arbitrary environments [1, 2, 3]. Appropriate inductive bias [4] is essential. In unknown environments, however, we do not want too specific a bias. How unspecific may it be? The *algorithmic Occam's razor bias* (AORB) assumes that problem solutions are non-random but regular [5, 6, 7, 8], without specifying the way in which they are non-random. AORB is highly specific in the sense that it tends to be useless for solving an arbitrary problem from the set of all well-defined problems, almost all of which have irregular solutions [5, 6, 7, 8]. But AORB is highly unspecific in the sense that it can help to solve all kinds of “typical”, “regular”, interesting problems occurring in our atypical, regular universe. This paper, based on [1], goes beyond our other recent papers exploiting algorithmic regularities for practical machine learning purposes [9, 3, 10].

Levin search (LS). References [9, 3] show how a variant of Levin search (LS) [11, 12, 13, 8] can find problem solutions with low Kolmogorov complexity and high generalization ability in non-random but otherwise quite general settings. LS is of interest because it has the optimal order of computational complexity for a broad class of search problems. For instance, 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., [14].

Adaptive LS. References [15, 3, 10] note that LS is not necessarily optimal if algorithmic information (e.g., [8]) between solutions to successive problems can be exploited to reduce future search costs based on experience. Our adaptive LS extension (ALS) [15, 10] does use experience to modify LS' underlying probability distribution. ALS can dramatically reduce the search time consumed by successive LS calls in certain regular environments [10].

Exploiting arbitrary regularities? This paper goes one step further. Let us call a learner's modifiable components its policy. Policy-modifying algorithms are called learning algorithms. We observe that if our learner wants to exploit *arbitrary* algorithmic regularities then it must be able to execute arbitrary, problem-specific learning algorithms. In particular, this includes learning algorithms for creating better learning algorithms, given environmental conditions. In this sense we are interested in *learning to learn*.

Restricted notion of "learning to learn". In contrast to [16] and other chapters in the book at hand, but in the spirit of the first author's earlier work [17, 18, 1] we will reserve the expressions *metalearning* or *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.

Overview. To create a potential for learning learning algorithms, we construct a *self-modifying policy* (SMP) which incorporates modifiable aspects of its own learning algorithm. This is easily achieved by letting SMP influence the composition of complex (probabilistic) learning algorithms from SMP-modifying instructions called *primitive learning algorithms*. We then ask: how can we force SMP to trigger better and better self-modification methods? Addressing this question in a reinforcement learning (RL) context naturally leads to the *success-story algorithm* (SSA). SSA is a backtracking method that measures the long-term effects of learning on later learning using long-term reward/time ratios. SSA uses such measurements to decide which SMP-generated SMP-modifications to keep and which to undo.

Outline. The next section will introduce basic concepts. A concrete implementation will be presented in section 3. Section 4 will focus on an experiment involving a difficult partially observable environment (POE). POEs have received a lot of attention in recent years, e.g., [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 10, 29]. Our POE, however, is far larger than most of those purported to demonstrate the usefulness of previous POE algorithms, most of which appear to work only for tiny state spaces [23].

2 Basic Concepts

System resources. The learner lives from time 0 to unknown time T in an unknown environment E. It has an internal state S and a policy SMP. Both S and SMP are variable data structures influencing probabilities of instructions to be executed. Between time 0 and T, the learner repeats the following cycle over and over again ($\mathcal{A}$ denotes a set of possible instructions): select and execute $a \in \mathcal{A}$ with conditional probability $P(a \mid SMP, S, E)$. Instruction a will consume time [30, 31] and may change E, S, and even SMP. (Somewhat related, but more restricted limited resource scenarios were studied in [32, 33, 34] and references therein.)

Primitive learning algorithms (PLAs). Actions in $\mathcal{A}$ that modify SMP are called *primitive learning algorithms* (PLAs). To ensure non-vanishing exploration potential PLAs may not generate SMP-modifications that will let certain action probabilities vanish entirely. PLAs and other actions can be combined to form more complex (probabilistic) learning algorithms.

Checkpoints. The learner's entire life-time can be partitioned into time intervals separated by special times called *checkpoints*. Checkpoints are computed dynamically during the learner's life by certain actions in $\mathcal{A}$ executed according to SMP itself. The learner's k-th checkpoint is denoted s_k. Checkpoints obey the following rules: (1) $\forall k \;\; 0 < s_k < T$. (2) $\forall j < k \;\; s_j < s_k$. (3) Except for the first, checkpoints may not occur before at least one PLA executed at least one

SMP-modification since the previous checkpoint.

Sequences of SMP-modifications (SSMs). SSM_k denotes the sequence of SMP-modifications computed by PLAs between s_k and s_{k+1}. Since PLA execution probabilities depend on SMP, SMP can in fact modify the way in which it modifies itself.

Reward. Occasionally E provides real-valued reward. $R(t)$ is the cumulative reward obtained between time 0 and time $t > 0$, where $R(0) = 0$.

Goal. At some checkpoint t the learner's goal is to generate SMP-modifications that accelerate long-term reward intake: it wants to let $\frac{R(T)-R(t)}{T-t}$ exceed the current average reward intake. To determine this speed, a previous point $t' < t$ to compute $\frac{R(t)-R(t')}{t-t'}$ is required. How can t' be specified 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?

We address this question by maintaining a time-varying set V of past checkpoints that have led to long-term reward accelerations. Initially V is empty.

Success-story criterion (SSC). Let v_k denote the k-th element of V in ascending order. SSC is satisfied at time t if either V is empty (trivial case) or if

$$\frac{R(t)}{t} < \frac{R(t) - R(v_1)}{t - v_1} < \frac{R(t) - R(v_2)}{t - v_2} < \ldots < \frac{R(t) - R(v_{|V|})}{t - v_{|V|}}. \tag{1}$$

SSC demands that each checkpoint in V marks the beginning of a long-term reward acceleration measured up to the current time t.

Success-Story Algorithm (SSA). At every checkpoint we invoke the *success-story algorithm* (SSA):

1. **WHILE** SSC is not satisfied

   Undo all SMP modifications made since the most recent checkpoint in V.
   Remove that checkpoint from V.

2. Add the current checkpoint to V.

"Undoing" a modification means restoring the preceding SMP — this requires storing past values of SMP components on a stack prior to modification. (Components of SMP and elements of V can be stored on the same stack — see section 3.)

Thus each SMP modification that survived SSA is part of a bias shift generated after a checkpoint marking a lifelong reward speed-up.

Implementing SSA. Using stack-based backtracking methods such as those described in section 3, one can guarantee that SSC will be satisfied after each new SMS-start, despite interference from S and E. Although inequality (1) contains $|V|$ fractions, SSA can be implemented efficiently: only the two sequences of modifications on top of the stack need to be considered at a given time in an SSA call (see details in section 3). A *single* SSA call, however, may undo *many* sequences of self-modifications if necessary.

Timing SSA calls. Between two checkpoints SMP is temporarily protected from SSA evaluations. Since the way of setting checkpoints depends on SMP, SMP can learn *when* to evaluate itself. This evaluation-timing ability of SMP is important in dealing with unknown reward delays.

SSA's generalization assumption. At the end of each SSA call, until the beginning of the next one, the only temporary generalization assumption for inductive inference is: SMP-modifications that survived all previous SSA calls will remain useful. In absence of empirical evidence to the contrary, each surviving sequence of modifications SSM_k is assumed to have set the stage for later successful $\mathrm{SSM}_i, i > k$. In unknown environments, which other generalization assumption would make sense? 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 policy modifications executed after any given previous point in time: the average reward per time since then.

Satisfying SSC in a non-trivial way. In antagonistic environments with constantly decreasing reward there is no way of justifying *permanent* SMP-modifications by SSC. Let us assume, however, that E, S and $\mathcal{A}$ (representing the system's initial bias) do indeed allow for SMP-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., [35, 36, 37]. Now, since we prevent all instruction probabilities from vanishing (see implementation below), SMP cannot help but trigger occasional SMP-modifications, and keep those consistent with SSC: in this sense, the learner can only improve. Note that the older some SMP-modification, the more time will have passed to gather experience with its long-term consequences, and the more stable it will be if it is indeed useful and not just there by chance.

Measuring effects of learning on later learning. Since some SMP-modification's success recursively depends on the success of later SMP-modifications for which it sets the stage, SSA (unlike other RL methods) automatically provides a basis for *learning how to learn* or *metalearning*: SSA prefers SSMs making "good" future SSMs more likely. In other words, SSA prefers probabilistic learning algorithms that lead to better probabilistic learning algorithms.

SMP/SSA's limitations. Of course, in general environments neither SMP/SSA nor any other scheme is guaranteed to find "optimal" policies that will lead to maximal cumulative reward. SSA is only guaranteed to selectively undo those policy modifications that were not empirically observed to lead to an overall speed-up of average reward intake. Still, this is more than can be said about previous RL algorithms.

3 An Implementation

Overview. This section describes a concrete implementation of the principles above. SMP is a set of variable probability distributions over a set of assembler-like instructions including SMP-modifying instructions. The learner executes a lifelong instruction sequence generated according to its SMP. We describe details of its SSA implementation.

Architecture. There 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). *IP*'s current position represents the internal state S (and can be used to disambiguate inputs). Instructions and their parameters are encoded by a fixed set I of n_{ops} integer values $\{0, \ldots, n_{ops}-1\}$. For each value j in I, there is an instruction B_j with N_j integer-valued parameters.

Stochastic, self-modifying policy. For each program cell i there is a variable probability distribution SMP$_i$ over I. For every possible $j \in I$, $(0 \leq j \leq n_{ops}-1)$, SMP$_{ij}$ 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 SMP$_{ij}$-values defines a probability matrix SMP (the learner's *current stochastic policy*) with columns SMP$_i$ $(0 \leq i \leq m-1)$. If $IP = i$, then the contents of i, namely d_i, will be interpreted as instruction B_{d_i}, and the contents of cells immediately following i will be interpreted as B_{d_i}'s arguments, selected according to the corresponding SMP-values. See *Basic Cycle* below.

Normal instructions. Normal instructions are those that do not change SMP. An example is B_0 (later we will introduce many additional, problem-specific instructions for interaction with an environment):

B_0: *JumpHome — set* $IP = 0$ *(jump back to 1st program cell).*

Primitive learning algorithms (PLAs). By definition, a PLA is an instruction that modifies SMP. PLAs such as *IncProb* below and other instructions may be composed to form complex (probabilistic) learning algorithms. In what follows, the symbols $a1, a2, a3$ stand for instruction parameters in $\{0, 1, \ldots, n_{ops}-1\}$.

B_1: *IncProb(a1, a2, a3)* — Set $i := (a1 * n_{ops} + a2)/k$ (integer division); set $j := a3$. If $0 \leq i \leq m-1$, increase SMP$_{ij}$ by γ percent, and renormalize SMP$_i$ (but prevent SMP-components from falling below a minimal value ϵ, to avoid near-determinism). We will use $\gamma = 15, \epsilon = 0.001$, $k = 3$.

Although our PLAs do not allow for composing *arbitrary* probabilistic learning algorithms, they do allow for a wide variety of them (in [1] a universal programming language is used to create the potential for even more powerful learning algorithms). Later we will integrate Q-learning [37, 21] into SMP as a primitive instruction.

Evaluation instruction. By executing the instruction *"PrepareEvaluation"* the learner can influence the timing of the next checkpoint evaluating the learner's lifelong performance so far:

B_2: *PrepareEvaluation(a1)* — Temporarily disable PLAs (e.g., *IncProb*) by preventing them from executing further SMP-modifications, until $a1+1$ additional, positive reward signals have been received (this will define a checkpoint and trigger an SSA call — see basic cycle below).

Initialization. At time 0 (system birth) we initialize SMP with maximum entropy distributions (all SMP_{ij} equal), and set $IP = 0$. We introduce an initially empty stack $\mathcal{S}$ that allows for variable-sized stack entries, and the conventional *push* and *pop* operations. The integer variable $NumR$ (initially zero) is used to count down the remaining non-zero reward signals since the last execution of *"PrepareEvaluation"*. The Boolean variable SMP*changed* is initially false.

Basic cycle of operations. Starting at time 0, until time T (system death), the system repeats the following basic instruction cycle over and over again in real time:

1. Randomly generate an integer $j \in I$ according to matrix column SMP_{IP} (the probability distribution of the program cell pointed to by IP). Set program cell content $d_{IP} := j$. 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** $IP > m - N_j - 2$, **THEN** set IP to 0, go to step **1.** **ELSE** generate instruction arguments for the N_j cells immediately following IP, according to their probability distributions SMP_{IP+1}, ..., SMP_{IP+N_j}, and set $IP \leftarrow IP + N_j + 1$.

2. **IF** B_j is a PLA, and not currently disabled by a previous *"PrepareEvaluation"* instruction, **THEN** push copies of those SMP_i to be modified by B_j onto $\mathcal{S}$.

3. Execute instruction B_j. This may change (1) environment, (2) IP, (3) SMP itself. In case (3), set SMP$changed = TRUE$. **IF** B_j is *"PrepareEvaluation"* and SMP$changed = TRUE$, **THEN** set $NumR$ equal to B_j's actual parameter plus one.

4. **IF** $NumR > 0$ and non-zero reward occurred during the current cycle, **THEN** decrement $NumR$. **IF** $NumR = 0$, **THEN** call SSA to achieve the success-story criterion by backtracking as follows:

SSA.0. Set variable t equal to current time (t is a checkpoint).

SSA.1. **IF** there is no *"tag"* stored somewhere in $\mathcal{S}$ (tags are pairs of checkpoints in V (see section 2) and rewards pushed in earlier SSA calls), **THEN** push the tag $(t,\, R(t))$ onto $\mathcal{S}$, set SMP$changed = FALSE$, and go to **1** (this ends the SSA call).

SSA.2. Denote the topmost tag in $\mathcal{S}$ by $(t',\, R(t'))$. Denote the one below by $(t'',\, R(t''))$ (if there isn't any tag below, set variable $t'' = 0$ — recall $R(t'') = R(0) = 0$).

SSA.3. **IF** $\frac{R(t)-R(t')}{t-t'} > \frac{R(t)-R(t'')}{t-t''}$ **THEN** push $(t,\, R(t))$ as a new tag, set SMP$changed = FALSE$, and go to **1.** This ends the SSA call.

ELSE pop off all stack entries above the one for tag $(t',\, R(t'))$ (these entries will be former SMP-components saved during an earlier execution of step 2), and use them to restore SMP as it was before time t'. Then also pop off tag $(t',\, R(t'))$. Go to **SSA.1**.

Each step above will consume various amounts of system life-time. Note a few differences to "Genetic Programming": (1) no program populations, (2) self-modification ability, (3) stochastic policy, (4) instructions generated and executed online, (5) SSA calls evaluating the entire life so far.

Lifelong reward acceleration. After each call of SSA the history of surviving modifications is *guaranteed* to be a life-time success-story (in the worst case an empty one). We don't have to assume fully observable environments — compare SSC and generalization assumption in section 2.

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 E and $\mathcal{A}$. This lack of quantitative convergence results is shared by almost all other, less general RL schemes.

4 Experiment: Solving a Difficult POE Task

Environment. Figure 4 shows a partially observable environment (POE) with 600×800 pixels. This POE has many more states and obstacles than most reported in the POE literature — for instance, Littman et al.'s largest problem [23] involves less than 1000 states. There are two SMP/SSA-based agents A and B. Each has circular shape and a diameter of 30 pixels. At a given time, each is rotated in one of eight different directions. State space size exceeds 10^{13} by far, not even taking into account internal states (*IP* values) 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 4).

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. For each turn or move there is a small penalty (immediate reward -0.0001).

Positive reward is generated *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). Only then a new trial starts. There is no maximal trial length, and the agents have no *a priori* concept of a trial. Time is not reset when a new trial starts.

Instruction set. 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 10 "steps" in front of it. The step size is 5 pixel widths. 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 $m = 52$ program cells, and $n_{ops} = 13$ instructions (including JumpHome/PLA/evaluation instructions B_0, B_1, B_2 from section 3):

B_3: *Move(n)* — If $0 \leq n \leq 9$ then move $n - 3$ (if $n \geq 3$) steps forward or $-n$ (if $n < 3$) steps backward in the current direction.

B_4: *Turn(d)* — If $0 \leq d \leq 7$ then change current direction D to $(D + d + 4)\ mod\ 8$.

B_5: *TurnRelativeToKey(d)* — If $0 \leq d \leq 7$, turn to the direction that best matches the line connecting the centers of agent and its key, then *Turn(d)*.

B_6: *TurnRelativeToDoor(d)* — (analogous to B_5).

B_7: *TurnRelativeToGoal(d)* — (analogous to B_5).

B_8: *LookForKey(n)* — If $0 \leq n \leq 9$: if the agent's key is not within $n + 1$ 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(n)* — (analogous to B_8).

B_{10}: *LookForObstacle(n)* — (analogous to B_8).

B_{11}: *LookForGoal(n)* — (analogous to B_8).

B_{12}: *CondJumpBack(a1)* — If $a1 = 0$ and the agent does not hold its key, or if $a1 = 1$ and the corresponding door is closed, or if $a1 = 2$ and the agent does not touch the goal, then set agent's IP to the instruction following the last executed *CondJumpBack* (if there is no such instruction, set $IP = 0$).

Each instruction occupies 2 successive program cells except for *IncProb*, which occupies 4. Limited obstacle perception makes this problem a difficult POE — unlike key and door, obstacles are not visible from far away. Note that our agents may use memory (embodied by their *IP*s) to disambiguate inputs.

Results without learning. If we switch off SMP's self-modifying capabilities (*IncProb* has no effect), then the average trial length is 330,000 basic cycles (random behavior).

Results with Q-Learning. Q-learning assumes that the environment is fully observable; otherwise it is not guaranteed to work. Still, some authors occasionally apply Q-learning variants to partially observable environments, sometimes even successfully [38]. To test whether our problem is indeed too difficult for Q-learning, we tried to solve it using various $TD(\lambda)$ Q-variants [21]. We first used primitive actions and perceptions similar to SMP's instructions. 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 allows for turning without moving: "turn 45 degrees to the right". These actions are more powerful than SMP's instructions (most combine two actions similar to SMP's into one). There are $2 * 5 = 10$ different inputs telling the agent whether it has/hasn't got its key, and whether the closest object (obstacle, key, door, or goal) part of which is either 10, 20, 30, 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). For each executed action there is an immediate small penalty of -0.0001, to let Q-learning favor shortest paths.

This Q-learning variant, however, utterly failed to achieve significant performance improvement. We then tried to make the problem easier (more observable) 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, 20, 30, 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 $2 \times (4 \times 5 + 1)^3 = 18522$ different inputs corresponding to 18522 different rows in the Q-tables (with a total of 611226 entries). *This worked indeed a bit better than the simpler Q-variant.* Still, we were not able to make Q-learning achieve very significant performance improvement — see Figure 4.

Results with SMP/SSA. After 10^9 basic cycles (ca. 130,000 trials), average trial length was 5500 basic cycles (mean of 4 simulations). This is about 60 times faster than the initial random behavior, and roughly $\frac{1}{4}$ to $\frac{1}{3}$ of the optimal speed estimated by hand-crafting a solution (due to the POE setting and the random trial initializations it is very hard to calculate optimal average speed). Typical results are shown in Figures 4 and 4.

Q-learning as an instruction for SMP. Q-learning is not designed for POEs. This does not mean, however, that Q-learning cannot be plugged into SMP/SSA as a sometimes useful instruction. To examine this issue, we add $TD(\lambda)$ Q-learning [21]. to the instruction set of both agents' SMPs:

B_{13}: *Q-learning(a1)* — with probability $\frac{a1}{200 * n_{ops}}$, keep executing actions according to the Q-table until there is non-zero reward, and update the Q-table according to the $TD(\lambda)$ Q-learning rules [21]. Otherwise, execute only a single action according to the current Q-table.

Interestingly, this Q-plug-in leads to even better results near system death (see Figure 4). Essentially, the SMPs learn *when* to trust the Q-table.

Observations. Final stack sizes never exceeded 250, corresponding to about 250 surviving SMP-modifications. This is just a small fraction of the about 3.5×10^4 self-modifications executed during each agent's life. It is interesting to observe how the agents use self-modifications to adapt self-modification frequency itself. Towards death they learn that there is not as much to learn any more, and decrease this frequency (self-generated annealing schedule, see Figure 4). It should be mentioned that the adaptive distribution of self-modifications is highly non-uniform. It often temporarily focuses on currently "promising" individual SMP-components. In other words, the probabilistic self-modifying learning algorithm itself occasionally changes based on previous experience.

5 Discussion

The success-story algorithm collects more information than previous RL methods about long-term effects of policy changes and "bias shifts". In contrast to traditional RL approaches, time is not reset at trial boundaries. Instead SSA measures 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.

The more powerful an SMP's instruction set, the more powerful the stochastic learning algorithms it may construct, and the more general the bias shifts it can compute. The more general the bias shifts, the harder the credit assignment problem — but SSA provides a way of forcing even SMPs with very general (e.g., Turing machine-equivalent) instruction sets to focus on bias shifts causing histories of long-term performance improvements.

Previous work. Unlike SMP/SSA, Lenat's approach [39] requires human interaction defining the "interestingness" of concepts to be explored. Also, many essential aspects of system behavior in previous work on metalearning [39, 40] are not accessible to self-modifications.

The *meta-evolution* scheme [17] attempts to learn learning algorithms by a potentially infinite hierarchy of Genetic Programming levels. Higher-level pools contain program-modifying programs which are applied to lower-level programs, and rewarded recursively based on lower-level performance. Unfortunately there is no theoretical assurance that *meta-evolution* will spend overall computation time wisely.

Self-modifying recurrent neural nets [18, 41] are able to run their own weight change algorithms. Activations of special output units are used to address, read, and modify all of the network's own weights. Gradient descent in a standard temporal error function corresponds to gradient descent in the space of weight change algorithms, or learning algorithms. Interestingly, this can be done in "only" $O(n^4 \log n)$ operations per time step. One problem with this wild idea are the numerous local minima.

In contrast to SMP/SSA no previous metalearning approach evaluates learning processes by measuring their long-term effects on subsequent learning in terms of reward intake speed.

Limitations. (1) Like any machine learning algorithm, SMP/SSA suffers from the fundamental limitations mentioned in the first paragraph of this paper. (2) SSA does not make much sense in antagonistic 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 "fair", regular environments that do allow for long-term reward speed-ups. This does include certain zero-sum games though [42]. (3) We do not gain much by applying SMP/SSA to, say, simple "Markovian" mazes for which there already are efficient RL methods based on dynamic programming. In certain more realistic situations where standard RL approaches tend to fail, however, our method is of interest.

Application. A particular SMP/SSA implementation based on very simple, assembler-like instructions has been used successfully to solve a complex POE. To our knowledge, such complex

POEs have not been solved by previous POE algorithms, which appear to work only for tiny state spaces [23].

Outlook. There are many alternative ways of implementing SMP/SSA — for instance, instructions may be highly complex learning algorithms. Future work will focus on plugging a whole variety of well-known algorithms into SMP/SSA, and letting it pick and combine the best, problem-specific ones.

References

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

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

[3] J. Schmidhuber. Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 1997. In press.

[4] P. Utgoff. Shift of bias for inductive concept learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning*, volume 2, pages 163–190. Morgan Kaufmann, Los Altos, CA, 1986.

[5] R.J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

[6] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

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

[8] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer, 1993.

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

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

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

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

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

[14] O. Watanabe. *Kolmogorov complexity and computational complexity.* EATCS Monographs on Theoretical Computer Science, Springer, 1992.

[15] J. Schmidhuber. Discovering problem solutions with low Kolmogorov complexity and high generalization capability. Technical Report FKI-194-94, Fakultät für Informatik, Technische Universität München, 1994. Short version in A. Prieditis and S. Russell, eds., Machine Learning: Proceedings of the Twelfth International Conference, Morgan Kaufmann Publishers, pages 488–496, San Francisco, CA, 1995.

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

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

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

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

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

[21] L.J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

[22] M. B. Ring. *Continual Learning in Reinforcement Environments.* PhD thesis, University of Texas at Austin, Austin, Texas 78712, August 1994.

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

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

[25] L. Chrisman. 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, 1992.

[26] T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 345–352. MIT Press, Cambridge MA, 1995.

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

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

[29] M. Wiering and J. Schmidhuber. HQ-Learning: Discovering Markovian subgoals for non-Markovian reinforcement learning. Technical Report IDSIA-95-96, IDSIA, 1996.

[30] S. Russell and E. Wefald. Principles of Metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

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

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

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

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

[35] P. R. Kumar and P. Varaiya. *Stochastic Systems: Estimation, Identification, and Adaptive Control.* Prentice Hall, 1986.

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

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

[38] R.H. Crites and A.G. Barto. Improving elevator performance using reinforcement learning. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA, 1996. MIT Press.

[39] D. Lenat. Theory formation by heuristic search. *Machine Learning*, 21, 1983.

[40] P. S. Rosenbloom, J. E. Laird, and A. Newell. *The SOAR Papers.* MIT Press, 1993.

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

[42] J. Zhao and J. Schmidhuber. Incremental self-improvement for life-time multi-agent reinforcement learning. In Pattie Maes, Maja Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, 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, 1996.

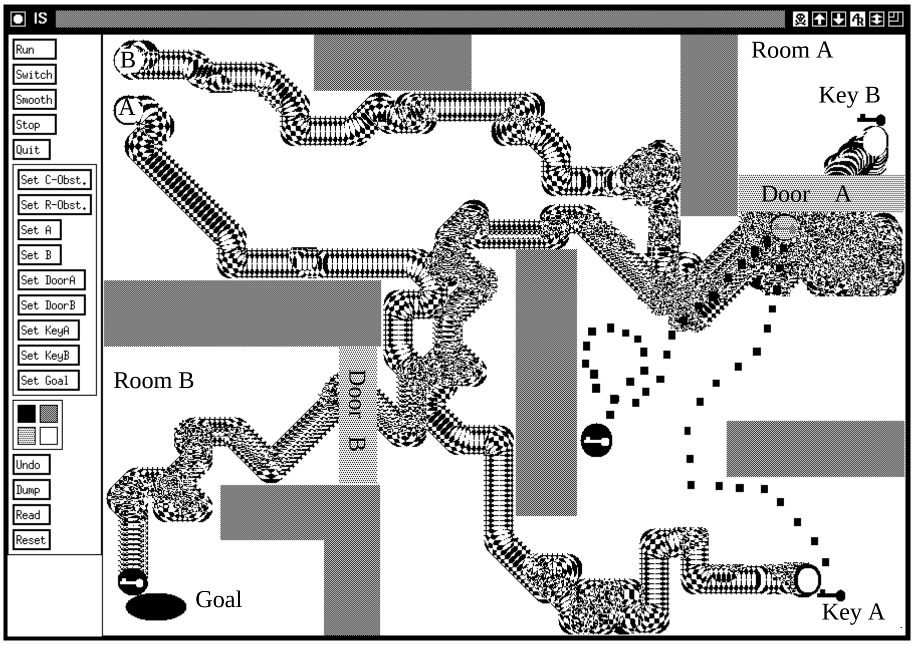

Figure 1: *Traces of interacting learning agents A and B in a partially observable environment with huge state space. Initially, both agents are randomly rotated and placed near the northwest corner. Agent A first moves towards the southeast corner to grab its key, then moves north (dotted trace) to 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-modifications (generated according to the policy itself via* IncProb *calls), although the strongly delayed reward is provided only if one of the agents touches the goal.*

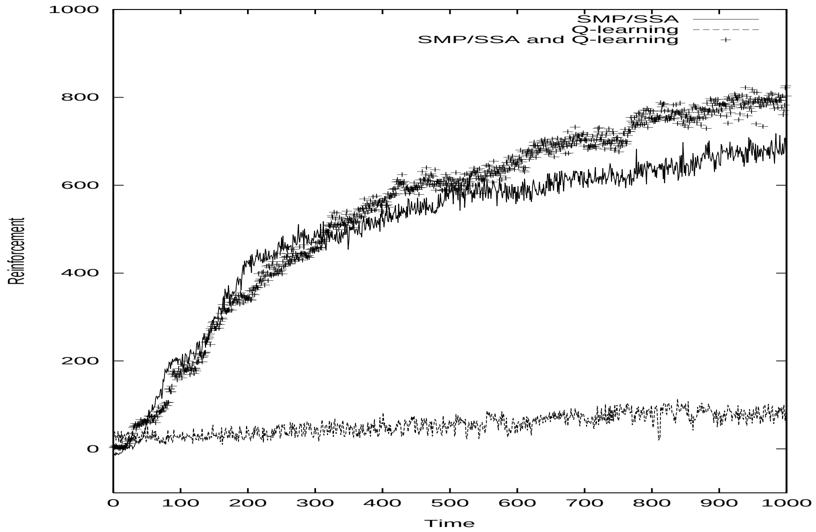

Figure 2: *Typical performance of SMP/SSA compared to one of the best Q-learning variants (reward intake sampled at intervals of* 10^6 *instruction cycles, mean of 4 simulations). Q-learning hardly improves, while SMP makes rather quick, substantial progress. Interestingly, adding Q-learning to SMP's instruction set tends to improve late-life performance a little bit (trace of crosses) — essentially, SMP learns when to trust/ignore the Q-table.*

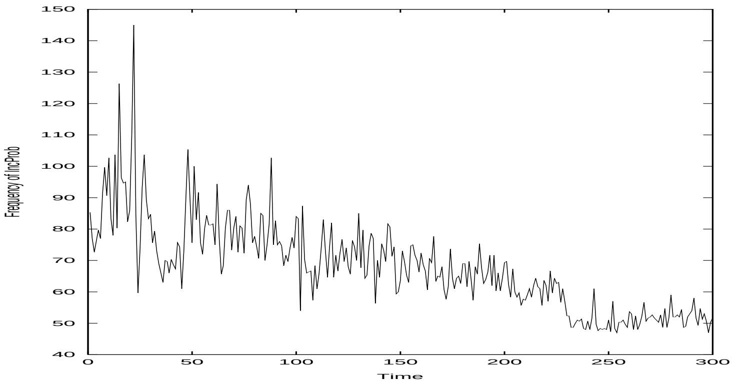

Figure 3: *Number of executed self-modifying instructions (IncProbs) plotted against time, sampled at intervals of* 2×10^6 *instruction cycles (mean of 3 simulations). In the beginning the agents use self-modifications to rapidly increase self-modification frequency. Towards death they learn that there is not as much to learn any more, and decrease this frequency, thus creating their own "annealing schedules".*