# A GENERAL METHOD FOR INCREMENTAL SELF-IMPROVEMENT AND MULTI-AGENT LEARNING*

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36
CH-6900-Lugano, Switzerland
juergen@idsia.ch
http://www.idsia.ch/~juergen

*In X. Yao, editor, Evolutionary Computation: Theory and Applications. Chapter 3, pp.81-123, Scientific Publ. Co., Singapore, 1999.

**Abstract**

I describe a novel paradigm for reinforcement learning (RL) with limited computational resources in realistic, non-resettable environments. The learner's policy is an arbitrary modifiable algorithm mapping environmental inputs and internal states to outputs and new internal states. Like in the real world, any event in system life and any learning process computing policy modifications may affect future performance and preconditions of future learning processes. There is no need for pre-defined "trials". At a given time in system life, there is only one single training example to evaluate the current long-term usefulness of any given previous policy modification, namely the average reinforcement per time since that modification occurred. At certain times in system life called checkpoints, such singular observations are used by a stack-based backtracking method which invalidates certain previous policy modifications, such that the history of still valid modifications corresponds to a history of long-term reinforcement accelerations (up until to the current checkpoint, each still valid modification has been followed by faster reinforcement intake than all the previous ones). Until the next checkpoint there is time to collect delayed reinforcement and to execute additional policy modifications; until then no previous policy modifications are invalidated; and until then the straight-forward, temporary generalization assumption is: each modification that until now appeared to contribute to an overall speed-up will remain useful. The paradigm provides a foundation for (1) "meta-learning", and (2) multi-agent learning. The principles are illustrated in (1) a single, self-referential, "evolutionary" system using an assembler-like programming language to modify its own policy, and to modify the way it modifies its policy, etc., and (2) another "evolutionary" system consisting of multiple agents, where each agent is in fact just a connection in a fully recurrent RL neural net.

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

MERRICK FURST

# 1 THEORETICAL CONSIDERATIONS

Previous work on reinforcement learning (e.g., [16, 2, 49, 51]) requires strong assumptions about the environment. In many realistic settings, however, these assumptions do not hold. In particular, any event/action/experiment occurring early in the life of a learning system may influence events/actions/experiments at any later time. To address these important but previously mostly ignored issues, this paper describes a novel, general framework for single-life reinforcement learning with limited computational resources in rather general environments. The focus is on theoretical considerations. However, experiments in sections 3 and 4 will serve to illustrate the basic concepts.

**Scenario.** Consider a learning system executing a lifelong action sequence in an unknown environment. For now, it won't even be necessary to introduce a detailed formal model of the environment. Different system actions may require different amounts of execution time (like in scenarios studied in [25, 4], and in references given therein). Occasionally the environment provides real-valued "reinforcement". The sum of *all* reinforcements obtained between "system birth" (at time 0) and time $t > 0$ is denoted by $R(t)$. Throughout its lifetime, the system's goal is to maximize $R(T)$, the cumulative reinforcement at (initially unknown) "death" $T$. There is only one life. Time flows in *one* direction (no resets to zero). Related, but less general scenarios were studied in papers on "bandit problems" (e.g., [3, 9] and references therein), which also require to wisely use limited resources to perform experiments. See also [10] and references therein for work on finding search space elements with optimal expected utility, subject to the resource constraint of spending only a feasible amount of time on finding such an element.

**Policy.** The system's current policy is embodied by modifiable parameters of an arbitrary algorithm mapping environmental inputs and internal states to output actions and new internal states. For example, parameters may be bits of machine code, or weights of a neural net. The number of parameters needs not be fixed. For now, we won't need a detailed formal model of the policy.

**Policy modification processes (PMPs).** Policy parameters are occasionally modified by finite "policy modification processes" (PMPs), sometimes also called "learning processes". Different PMPs may take different amounts of time. The $i$-th PMP in system life is denoted $PMP_i$, starts at time $t_i^1 > 0$, ends at $t_i^2 < T$, $t_i^2 > t_i^1$, and computes a policy modification denoted by $M_i$. Later we will require that a non-zero time-interval passes between $t_i^2$ and $t_{i+1}^1$, the beginning of $PMP_{i+1}$. While $PMP_i$ is running, the system's lifelong interaction

sequence with the environment may continue, and there may be reinforcement signals, too. In fact, $PMP_i$ may use environmental feedback to compute modification $M_i$ — for instance, by executing a known reinforcement learning algorithm. Later I will even take advantage of the possibility that PMPs may be generated and executed according to information embedded in the policy itself — this is of interest in the context of "self-modifying" policies that "learn how to learn how to learn ...". For the moment, however, I do not care for what exactly happens while $PMP_i$ is running: in what follows, I won't need a formal model of the PMP details. The following paragraphs are only concerned with the question: how do we measure modification $M_i$'s influence on the remaining system life? In particular, how do we measure whether $M_i$ successfully set the stage for later $M_k$, $k > i$?

**The problem.** In general environments, events/actions/experiments occurring early in system life may influence events/actions/experiments at any later time. In particular, $PMP_i$ may affect the environmental conditions for $PMP_k, k > i$. This is not addressed by existing algorithms for adaptive control and reinforcement learning (see, e.g., [16, 2, 49, 51]), and not even by naive, inefficient, but more general and supposedly infallible *exhaustive search* among all possible policies, as will be seen next.

**What's wrong with exhaustive search?** Apart from the fact that exhaustive search is not considered practical even for moderate search spaces, it also suffers from another, more fundamental problem. Let $n$ be the number of possible policies. For the sake of the argument, suppose that $n$ is small enough to allow for systematic, sequential generate-and-test of all policies within the system life time. Suppose that after all policies have been generated (and evaluated for some time), during the remainder of system life, we keep the policy whose test brought about maximal reinforcement during the time it was tested. In the general "on-line" situation considered in this paper, *this may be the wrong thing to do:* for instance, each policy test may change the environment in a way that changes the preconditions for policies considered earlier. A policy discarded earlier may suddenly be the "best" policy (but will never be considered again). Similar things can be said about almost every other search algorithm or learning algorithm — exhaustive search is just a convenient, very simple, but representative example.

**How to measure performance improvements?** Obviously, the performance criterion used by naive exhaustive search (and other, less general search and learning algorithms) is not appropriate for the general set-up considered in this paper. We first have to ask: what is a reasonable performance criterion for such general (but typical) situations? More precisely: if we do not make any assumptions about the environment, can we still establish a sensible criterion according to which, at a certain time, (1) the system's performance throughout its previous life always kept improving or at least never got worse, and which (2) can be *guaranteed* to be achieved by the system? Indeed, such a criterion can be defined, as will be seen below. First we will need additional concepts.

**Reinforcement/time ratios based on single observations.** At a given time $t$ in system life, we essentially have only one single "training example" to evaluate the current long-term usefulness of any given previous PMP, namely the *average reinforcement per time since that learning process occurred.* Suppose $PMP_i$ started execution at time $t_i^1$ and completed itself at time $t_i^2 > t_i^1$. For $t \geq t_i^2$ and $t \leq T$, the reinforcement/time ratio $Q(i,t)$ is defined as

$$Q(i,t) = \frac{R(t) - R(t_i^1)}{t - t_i^1}.$$

The computation of reinforcement/time ratios takes into account all computation time, including time required by PMPs.

Why measure time starting from the beginning of $PMP_i$, instead of starting from its end? Later we will see that the first evaluations of $PMP_i$'s performance will be delayed at least until after its end. While $PMP_i$ is still running, it is in a "grace period" which may be used to collect *delayed* reinforcement to justify $M_i$ — this is important when reinforcement signals occur long after policy modifications took place. Indeed, later I will make use of the fact that a "self-modifying" policy may determine the runtimes of its own PMPs, thus in principle being able to *learn* how long to wait for delayed reinforcement.

**Currently valid modifications.** After $t_i^2$, $PMP_i$'s effects on the policy will stay in existence until (a) being overwritten by later $PMP_k$, $k > i$, or until (b) being *invalidated* by the "success-story algorithm" (SSA), the method to be described below. To define *valid modifications*, only (b) is relevant: after $t_i^2$, the policy modification $M_i$ generated by $PMP_i$ will remain *valid* as long as SSA (see below) does not discard $M_i$ (by restoring the previous policy right before $PMP_i$ started).

**Success-story criterion (SSC).** At time $t$, SSC is satisfied if for each $PMP_i$ that computed a *still valid* modification $M_i$ of the system's policy,

(a) $Q(i,t) > \frac{R(t)}{t}$, and

(b) for all $k < i$, where $PMP_k$ computed a still valid $M_k$: $Q(i,t) > Q(k,t)$.

In words: SSC is satisfied if the beginning of each completed PMP that computed a still valid modification has been followed by long-term reinforcement acceleration — measured up until the current time. *Note that the success of some PMP depends on the success of all later PMPs, for which it is "setting the stage".* This represents an essential difference to previous performance criteria.

The method to achieve SSC is called "success-story algorithm" (SSA). SSA uses a stack to trace and occasionally invalidate certain policy modifications. Details are next.

**Success-story algorithm (SSA).** SSA uses an initially empty stack to store information about currently valid policy changes computed by PMPs.

Occasionally, at times called *"checkpoints"*, this information is used to restore previous policies, such that SSC holds (later we will see that a "self-modifying" policy can in principle *learn* to set checkpoints at appropriate times). SSA is based on two complementary kinds of processes:

**(1) Pushing.** Recall that $PMP_i$ starts at time $t_i^1$ and ends at $t_i^2$. Let $I_i$ denote a record consisting of the values $t_i^1$, $R(t_i^1)$ (which will be needed to compute $Q(i,t)$ values at later times $t$), plus the information necessary to restore the old policy (before modification by $PMP_i$). $I_i$ is pushed onto the stack (this will consume time, of course). By definition, $PMP_i$ is not finished until the entire pushing process is completed. From now on, the modification $M_i$ will remain *valid* as long as $I_i$ will remain on the stack.

**(2) Popping.** At certain times called *"checkpoints"* (see below), do:

> **WHILE** no time $t$ is reached such that one of the following conditions (1-3) holds, **DO:** pop the topmost $I_.$-value off the stack, and *invalidate* the corresponding modification $M_.$, thus restoring the corresponding previous policy.
>
> (1) $Q(i,t) > Q(i',t)$, where $i > i'$, and $M_i$ and $M_{i'}$ are the two most recent currently valid modifications (if existing),
>
> (2) $Q(i,t) > \frac{R(t)}{t}$, where $M_i$ is the only currently valid modification (if existing),
>
> (3) the stack is empty.

For all $PMP_i$, *checkpoints* occur right before time $t_i^1$, such that, by definition, $t_i^1$ coincides with the end of the corresponding popping process (note that checkpoints themselves may be set by a "self-modifying" policy that determines beginnings and runtimes of its own PMPs — care has to be taken, however, to avoid PMPs with infinite duration; see section 2.2). The time consumed by pushing processes, popping processes, and all other computations is taken into account (for instance, time goes on as popping takes place).

> **Theorem 1.** Whenever popping is finished, SSA will have achieved SSC.
>
> **Proof sketch.** Induction over the stack contents after popping.

Note that SSA does not care for the nature of the PMPs — for all completed $PMP_i$ (based, e.g., on conventional reinforcement learning algorithms), at each "checkpoint", SSA will get rid of $M_i$ if $t_i^1$ was not followed by long-term reinforcement speed-up (note that before countermanding $M_i$, SSA will already have countermanded all $M_k, k > i$). No modification $M_i$ is guaranteed to remain valid forever.

**Generalization assumption.** After popping, *until the next checkpoint*, SSA's straight-forward generalization assumption is: modifications that survived the most recent popping process (because they appeared to contribute

to speed-up of average reinforcement intake) will remain useful. In general environments, which other generalization assumption would make sense? Recall that since life is one-way (time is never reset), at each checkpoint the system has to generalize from a *single* experience concerning the usefulness of any given previous learning process: the average reinforcement per time since that learning process occurred. Learning from singular experiences contrasts other, less realistic kinds of reinforcement learning, which, in one way or another, require the assumption that it is possible to collect sufficient statistics from *repeatable* training sequences.

**To summarize:** SSA again and again (at each checkpoint) implicitly evaluates each still valid policy modification as to whether it has been followed by long-term performance improvement (perhaps because the modification set the stage for later useful modifications). If there is evidence to the contrary, SSA invalidates policy modifications until SSC is fulfilled again. SSA's stack-based backtracking is efficient in the sense that only the two most recent ("topmost") still valid modifications need to be considered at a given time (although a *single* popping process may invalidate *many* modifications). SSA is a general framework — you can use your favorite learning algorithm $A$ to generate the PMP. This makes sense especially in situations where the applicability of $A$ is questionable because the environment does not satisfy the preconditions that would make $A$ sound. SSA can at least guarantee that those of $A$'s policy modifications that appear to have negative long-term effects are countermanded.

Note that a trivial way of satisfying SSC is to never make a policy modification. But any system that cannot let modification probabilities entirely vanish occasionally will execute policy modifications, and keep those consistent with SSC. In this sense, it cannot help but getting better, if the environment does indeed provide a chance to improve performance, given the set of possible actions.

Due to the generality of the approach, no reasonable statements can be made about improvement *speed*, which indeed highly depends on the nature of the environment and the choice of the action set. This lack is shared by almost all other reinforcement learning approaches, though.

**Costs of environment-independence.** A price to pay for environment-independence is this: the beginning of the time interval considered to measure the current reinforcement/time ratio is marked by the beginning of the PMP that computed the most recent valid policy modification. The length of this time interval may vary unpredictably, because its beginning may wander back in time due to policy restaurations by SSA. In arbitrary environments, SSA cannot guarantee speed-up of reinforcement per *fixed* time interval (no algorithm can).

**Benefits of environment independence.** Next we will see that SSA provides a novel basis for two notoriously difficult issues in machine learning: (*) Multi-agent learning, (**) "Learning how to learn".

**(*) Multi-agent learning.** Consider the case where there are multiple, interacting, SSA-learning agents. For each agent, the others are part of the

changing (possibly complex) environment. This is the main reason why previous approaches to multi-agent learning are heuristic by nature. Not so with SSA, however: since SSA is environment-independent, each agent will still be able to satisfy its SSC after each checkpoint. In cases where all agents try to speed up the same reinforcement signals, and where no agent can speed up reinforcement intake by itself, this automatically enforces "learning to cooperate" [35, 36, 40]. Section 2.3 will illustrate this with an application of a system consisting of multiple agents, where each agent is in fact just a connection in a neural net.

**(**) Learning how to learn / Incremental self-improvement.** With SSA, the success of $PMP_i$ recursively depends on the success of later $PMP_k$, $k > i$: the cumulative reinforcement collected after $PMP_i$ includes the cumulative reinforcement collected after $PMP_k$, $k > i$. In particular, performance improvements include those improvements that make future additional improvements more likely: policy modification $M_i$ can prove its long term usefulness by setting the stage for additional, useful modifications $M_k, k > i$, etc. Now recall that SSA does not care for the nature of the PMPs — they may depend on the system policy itself. If we allow a system's policy to change itself by computing and executing its own PMPs (this is easy to implement, e.g. by using instructions of an assembler-like, "self-referential" programming language as system actions — see section 2.2), then SSA will keep only those self-modifications followed by reinforcement speed-ups, in particular those leading to "better" future self-modifications, etc., recursively: embedding the learning mechanism within the system policy immediately and naturally leads to "incremental self-improvement" and "learning how to learn how to learn ..." — there is no circularity involved, and the approach remains sound.

**Outline of remainder of paper.** The main contribution of this paper is given by what has been said above. The remainder of this paper mainly serves to illustrate the theory. In section 2, I will exemplify the ideas in paragraph (**) on incremental self-improvement and describe a particular, concrete, working, "evolutionary" system that implements them. Section 3 will then apply this system to various tasks, including a variant of Sutton's maze task [47]. One difference to Sutton's original task is that the policy environment continually changes because of actions generated by the system itself. Section 4 will then exemplify the ideas in paragraph (*) on multi-agent learning and also describe a particular, concrete, working, "evolutionary" system that implements them. With this system, each SSA-learning agent is in fact just a connection in a fully recurrent neural net. Each connection's policy is represented by its current weight. Connections do have a very limited policy space in comparison to typical, more complex agents used in standard AI — but this is irrelevant: SSA does not care for the complexity of the agents. A by-product of this research is a general reinforcement learning algorithm for recurrent nets. Again, an application to a variant of Sutton's maze task will serve to illustrate the operation of the system. In this application, the environment of each connection's policy continually changes, because the policies of all the other connections keep

changing.

**Note.** This paper is based on [32]. In the meantime we have published several additional papers on SSA, e.g., [50, 53, 40, 42].

## 2 SSA for Incremental Self-Improvement

**Outline.** The "evolutionary" system in this section (see also [32]) implements the ideas from section 1, in particular those in paragraph (**) on "incremental self-improvement". To improve/speed up its own (initially very dumb, highly random) learning strategy, the system policy makes use of an assembler-like programming language suitable to modify the policy itself. The policy actually is a set of conditional, modifiable probability distributions (initially maximum entropy distributions) on a set of assembler-like instructions. These distributions are required to compute the probability of the assembler-like instruction to be executed next (the system executes a lifelong sequence of such instructions). The $PMP_i$ from section 1 are self-delimiting "sequences of self-modifications" (SSMs). SSMs are special instruction subsequences (generated according to the current policy) that define their own beginning and their own end (by executing special instructions). With the help of special "self-referential" instructions, SSMs can compute almost arbitrary policy modifications (which actually are sequences of probability modifications). Policy modifications can be computed only by SSMs. SSMs affect the probabilities of future SSMs, which affect the probabilities of future SSMs, etc. These recursive effects are automatically taken care of by SSA: following the SSA principle, whenever an SSM computes a policy modification, information required to restore original probability distributions and to compute reinforcement/time ratios for currently valid modifications is pushed on a stack. After each instruction executed outside an SSM, the system runs a popping process (see section 1). It can be *guaranteed* that the system, after each action executed outside an SSM, will achieve SSC ($\rightarrow$ "learning how to learn"). In principle, the system can learn to deal with arbitrary reward delays by determining the lengths of its SSMs — SSMs define checkpoints (see section 1) by themselves.

The system is "evolutionary" in the sense that it lets survive only those (initially highly random) policy modifications that were followed by long-term reinforcement acceleration. The system has been implemented and tested on (non-Markovian) toy tasks in changing environments. The experiments illustrate the theoretical predictions: the system computes action subsequences leading to faster and faster reinforcement intake (section 3). Due to the generality of the approach, however, 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 instructions.

**Storage.** The system's *storage* is a single array of cells. Each cell has an integer address in the interval $[Min, Max]$. $Max$ is a positive integer. $Min$

is a negative integer. $a_i$ denotes the cell with address $i$. The variable contents of $a_i$ are denoted by $c_i \in [-Maxint, Maxint]$, and are of type integer as well ($Maxint \geq Max$; $Maxint \geq abs(Min)$). Special addresses, *InputStart*, *InputEnd*, *RegisterStart*, and *ProgramStart*, are used to further divide storage into segments: $Min < InputStart \leq InputEnd < 0 = RegisterStart < ProgramStart < Max$. The *input area* is the set of *input cells* $\{a_i : InputStart \leq i \leq InputEnd\}$. The *register area* is the set of *register cells* $\{a_i : 0 \leq i < ProgramStart\}$. "Registers" are convenient for indirect-addressing purposes. The *program area* is the set of *program cells* $\{a_i : ProgramStart \leq i < Max\}$. Integer sequences in the program area are interpreted as executable code. The *work area* is the set of *work cells* $\{a_i : Min \leq i < ProgramStart\}$. Instructions executed in the program area may read from and write to the work area. Both register area and input area are subsets of the work area.

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

**Primitives.** The assembler-like programming language is partly inspired by the one in [33, 38]. There are $n_{ops}$ primitive instructions ($n_{ops} << Maxint$). Each "primitive" is represented by a unique number in the set $\{0, 1, \ldots, n_{ops}-1\}$ (due to the code being written in C). The primitive with number $j$ is denoted by $p_j$. Primitives may have from zero to three arguments, each of which has a value in $\{0, 1, \ldots, n_{ops} - 1\}$. The semantics of the primitives and their corresponding arguments are given in Table 1. The non-self-referential ("normal") primitives include actions for comparisons, and conditional jumps, for copying storage contents, for initializing certain storage cells with small integers, and for adding, subtracting, multiplying, and dividing. They also include output actions for modifying the environment, and input actions for perceiving environmental states. The "self-referential" primitives will be described in detail below.

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

**Double indexed addressing.** Although program cells can take on only few values, the use of double-indexed addressing allows for addressing the entire

storage: a parameter in a program cell may point to a work cell with a small address. The content of this work cell, however, does not have essential limits, and may point to any address in storage. See [11, 53, 42, 40, 39], however, for alternative implementations without double indexed addressing.

**Current policy.** The set of all current $P$-values defines the system's *current policy*.

**Instruction cycle.** A single step of the *interpreter* works as follows: if *IP* points to program cell $a_i$, a primitive and the corresponding arguments are chosen randomly according to the current probability distributions, as already described. They are sequentially written onto the program area, starting from $a_i$. Syntax checks are performed. Rules for syntactic correctness are given in the caption of Table 1. If syntactically correct, the instruction gets executed. This may result in modifications of *IP* and/or environment and/or storage. If the instruction did not change the value of *IP* (e.g. by causing a jump), *IP* is set to the address of the cell following the last argument of the current instruction. If the instruction is syntactically incorrect, *IP* is reset to the first program cell. This *instruction cycle* represents the basic operation of the system.

**System life.** At time step 0, storage is initialized with zeros. The probability distributions of all program cells are initialized with maximum entropy distributions [44]. That is, all $P_{ij}$ values are initialized to the same value, so that there is no bias for a particular value in any cell. After initialization, the *instruction cycle* is repeated over and over again until system death at time $T$. Recall that $T$ does not have to be known in advance.

**Storage as part of the environment.** After initialization at time 0, the storage cells are never re-initialized: the storage has to be viewed as part of the total enviroment of the system policy. The storage is like an external notebook. Notes written into the notebook may influence future actions[1]. Notes may represent useful memories, or may have harmful long term effects.

**Self-referential primitives.** Two special primitives, *DecP* and *IncP*, may be used to address and modify the current probability distribution of any program cell (see Table 1). With the action *DecP*, the $P_{ij}$ value for a particular cell/value pair $(a_i, j)$ can be decreased by some factor in $\{0.01, 0.02, \ldots, 0.99\}$. The probabilities for that cell are then normalized. Likewise, with the action *IncP*, the *complement* $(1 - P_{ij})$ of the $P_{ij}$ value for a particular cell $a_i$ and value $j$ can be decreased by a factor in $\{0.01, 0.02, \ldots, 0.99\}$ (and the cell probabilities are again renormalized). *DecP* and *IncP* have no effect if the indirectly addressed cell contents $c_{c_{a3}}$ (see Table 1) are not an integer between 1 and 99, or if the corresponding probability modification would lead to at least one $P$-value below *MinP* (a small positive constant).

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

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

There is also another self-referential instruction, *EndSelfMod()*, that helps to group a sequence of probability-modifying instructions and other instructions into self-delimiting self-modification sequences (see below).

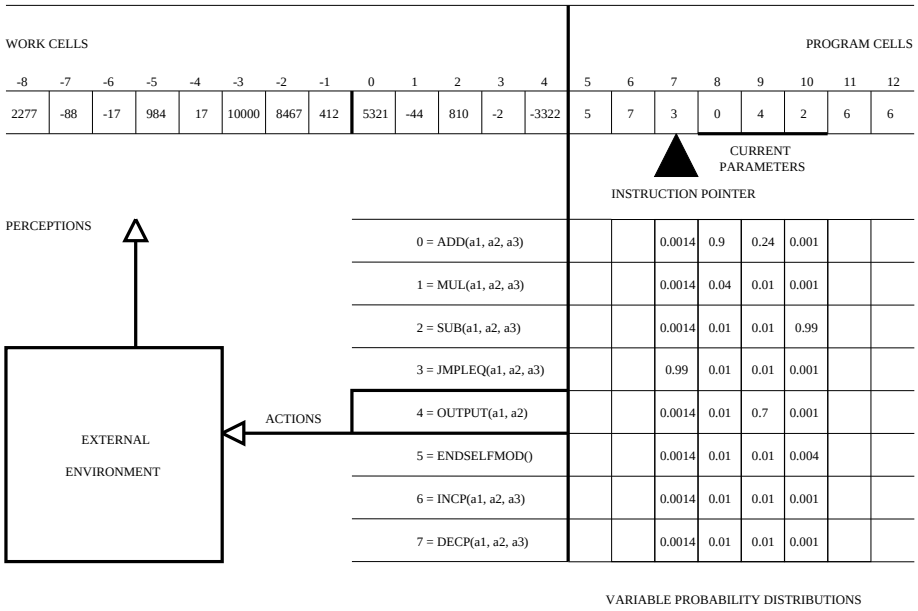


Figure 1: *Snapshot of parts of policy and storage (which is part of the policy environment). There are positive and negative addresses. In this hypothetical example, program cells start at address 5. Work cells end at address 4. For each program cell, there is a variable probability distibution on the* $n_{ops}$ *possible cell contents (shown only for cells 7, 8, 9, 10 — for simplicity, in this hypothetical example,* $n_{ops} = 8$*, and there is only one output instruction for manipulating the external environment). IP currently points to cell 7. With high probability (0.99), cell 7 is filled with a 3, which stands for instruction* Jmpleq. Jmpleq *requires 3 parameters: the contents of cells 8, 9, 10 (generated according to the corresponding probability distributions — in the example, the most likely values are chosen). Due to double indexed addressing, the semantics of the current instruction are: if the content of cell 5321 is smaller than the contents of cell -3322, then jump to cell 810.*

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

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

**Self-delimiting sequences of self-modifications (SSMs).** They correspond to the PMP from section 1. The beginning of the first $IncP$ or $DecP$ after an *EndSelfMod()* action or after system "birth" begins an SSM. The SSM ends itself by executing *EndSelfMod()*. Some of the (initially highly random) action subsequences executed during system life will indeed be SSMs. They can compute almost arbitrary sequences of modifications of $P_{ij}$ values, resulting in almost arbitrary modifications of context-dependent probabilities of future action subsequences, including future SSMs. However, due to $MinP$ being positive, the probability of selecting and executing a particular instruction at a particular time cannot entirely vanish. In agreement with section 1, the $i$-th SSM is called $PMP_i$.

**Keeping track with a stack.** Following the SSA principle, whenever the system uses $IncP$ or $DecP$ to modify one of its probability distributions, the following values are pushed onto a stack $S$: the current time, total reinforcement so far, the address of the modified program cell, its old probability distribution right before the current modification (represented by $n_{ops}$ real values), and a pointer to the stack entry corresponding to the first probability modification computed by the current SSM. This information is needed by the SSA popping process to be described later. More formally:

The $k$-th entry of $S$, $k \in \{0, 1, \ldots, StackSize\}$, is denoted $S(k)$. $S(k)$ consists of the following variables: $S(k).t$, $S(k).R$, $S(k).address$, $S(k).oldP$ (a vector of $n_{ops}$ variables), and $S(k).first$. The variable $sp$ points to the current topmost stack entry ($sp = 0$ at system startup). The zeroth stack entry, which cannot be popped, is initialized as follows: $S(0).t \leftarrow 0$; $S(0).R \leftarrow 0$; $S(0).first \leftarrow 0$. The remaining values are undefined. If some SSM modifies some probability distribution $P_i$ at time $t$ (by using $IncP$ or $DecP$), $sp$ is incremented, $S(sp).t \leftarrow t$; $S(sp).R \leftarrow R(t)$; $S(sp).address \leftarrow i$; $S(sp).oldP \leftarrow P_i$ *before* the modification (represented by $n_{ops}$ real values). If $t$ marks the beginning of an SSM then $S(sp).first \leftarrow sp$. Otherwise (in between beginning and end of an SSM), $S(sp).first \leftarrow S(sp-1).first$.

**SSA popping processes.** Popping occurs right before the first pushing process of each SSM, and also after each instruction except when an SSM is running (these are the possible additional checkpoints referred to in section 1): "old" probability distributions are successively popped and restored, until SSC (see section 1) is satisfied. Of course, the computation time required for pushing and popping probability distributions etc. is taken into account when reinforcement/time ratios are computed. More formally ($t$ denotes the current time):

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

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

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

Note: $t$ increases during execution of the while loop.

**Lemma 1.** According to the SSA principle (section 1), after each instruction (except during the execution of an SSM), an SSA popping process ensures that the beginning of each completed SSM that computed still valid probability modifications has been followed by *faster* reinforcement intake than the beginnings of all previous such SSMs.

**Proof sketch.** Induction over the stack contents after popping, in analogy to proof of Theorem 1, section 1.

**Arbitrary reward delays.** The *EndSelfMod()* instruction allows the system to delay SSA's evaluations of probability modifications arbitrarily (the expectation of the delay remains finite, however, due to $MinP$ being positive). These checkpoint setting capabilities are important, for two reasons: (1) In general, reinforcement events will be separated by long (unknown) time lags. Hence, novel probability modifications are not necessarily bad if they do not lead to immediate reinforcement. In principle, the system itself can learn how much time to spend on waiting for first reinforcement events. (2) Two successive modifications of two particular probability distributions may turn out to be beneficial, while each by itself may be harmful. For this reason, the system is allowed to compute arbitrary sequences of probability modifications, before facing SSA's evaluations.

Delaying evaluations *does* cost time, though, which is taken into account by SSA. In the long run, the system is encouraged to create useful SSMs of the appropriate size.

**Accelerating reinforcement intake.** Lemma 1 shows: the scheme keeps only modifications caused by SSMs followed by faster and faster reinforcement intake, thus satisfying SSC from section 1. Due to non-vanishing probabilities of arbitrary action sequences (recall that $MinP$ is positive), the system will eventually discover a way to improve its current performance, if there is any.

**Learning how to learn.** Performance improvements include improvements that make future improvements more likely: SSMs can prove their long term usefulness by setting the stage for additional, useful SSMs, which potentially include SSMs executing known (and not yet known) learning algorithms. The success of an SSM recursively depends on the success of all later SSMs. SSA automatically takes care of this, thus recursively encouraging "learning how to learn how to learn ...". This represents an essential difference to previous approaches to "continual" learning, see [24].

**Improvement speed?** Due to the generality of the approach, no reasonable statements can be made about improvement *speed*, which indeed highly depends on the nature of the environment and the choice of initial primitive instructions. This lack is shared by almost all other reinforcement learning approaches, though. Note, however, that unlike previous, less general systems, the

novel system in principle can exploit almost arbitrary environmental regularities [14, 5, 45, 21] (if there are any) to speed up performance improvement, simply because it can run almost arbitrary learning algorithms. For instance, in principle, unlike previous evolutionary and genetic algorithms [23, 43, 13, 12, 15], the system can learn to focus its modifications on interfaces between useful "subprograms" (*"divide and conquer"*), instead of mutating the subprograms themselves (if this proves to be beneficial in a given environment), thus creating a higher-level, more abstract search space ($\rightarrow$ directed mutations as opposed to random mutations). Just as evolution "discovered" that having the "genetic crossover operator" was a "good thing", the system is potentially able to discover that various more directed search strategies are "good things". There is just no feasible way of predicting the precise nature of such speed-ups.

**How many environments are regular?** The last paragraph mentioned that the novel system in principle can exploit almost arbitrary environmental regularities, if there are any. One might ask: how likely is it that a given environment contains regularities at all? How many environments do indeed allow for successful generalization from previous experience? A recent debate in the machine-learning community highlighted a fact that appears discouraging at first glance: in general, generalization cannot be expected, inductive inference is impossible, and nothing can be learned. See, e.g., [8, 26, 52, 31, 38, 37]. Paraphrasing from a previous argument [38, 37]: let the task be to learn some relation between finite bitstrings and finite bitstrings. Somehow, a training set is chosen. In almost all cases, the shortest algorithm computing a (non-overlapping) test set essentially has the same size as the whole test set. This is because most computable objects are irregular and incompressible [14, 5]. The shortest algorithm computing the test set, given the training set, isn't any shorter. In other words, the relative algorithmic complexity of the test set, given the training set, is maximal, and the mutual algorithmic information between test set and training set is zero (ignoring an additive constant independent of the problem — see [14, 5, 45, 21]). Therefore, in almost all cases, (1) knowledge of the training set does not provide any clues about the test set, (2) there is no hope for generalization, and (3) inductive inference does not make any sense. Similarly, almost all possible environments are "hostile" in the sense that they don't provide regularities exploitable by *any* learning algorithm. This may seem discouraging.

**Atypical real world.** Apparently, however, generalization and inductive inference *do* make sense in the real world! One reason for this may be that the real world is run by a short algorithm. See, e.g., [37]. Anyway, problems that humans consider to be *typical* are *atypical* when compared to the general set of all well-defined problems. Otherwise, things like "learning by analogy", "learning by chunking", "incremental learning", "continual learning", "learning from invariances", "learning by knowledge transfer" etc. would not be possible, and experience with previous problems could not help to sensibly adjust the prior distribution of solution candidates in the search space for a new problem

(shift of inductive bias, e.g. [48]).

To repeat: an interesting feature of incremental self-improvement is that its theoretical potential for exploiting environmental regularities, if there are any, exceeds the corresponding potential of previous learning systems.

# 3 Illustrative Experiments

The main purpose of this paper was to describe a theoretically sound principle for environment-independent, "one-way", single-life reinforcement learning. To illustrate basic aspects of the principle, the remainder of this paper will also present a few experiments. These, however, in no way represent a systematic experimental analysis. In fact, many much more complex experiments are described in other recent papers on this subject, e.g., [50, 42, 40].

The experiments in the current section demonstrate that the system from section 2 indeed can learn to compute SSMs leading to faster and faster reinforcement intake. The system uses low-level problem-specific instructions in addition to the 17 general, assembler-like instructions mentioned in section 2. The instructions reflect the system's initial (weak) bias. Of course, different problem-specific instructions lead to different initial bias and performance (even the primitive *numbers* may influence performance, because numbers of executed primitives appear in the corresponding program cells — later, these numbers may be abused, e.g., as addresses, or as arguments of arithmetic operations). In a given environment, reinforcement intake may be greatly accelerated within a few minutes using one set of instructions. However, the same degree of improvement may require a day of computation time using a different set of instructions. The purpose of this section, however, is *not* to perform a statistically significant experimental evaluation of the system's initial bias, or to study effects of introducing different kinds of initial bias, or to compare the system to other learning systems with different initial bias. Instead, this section's purpose is to illustrate typical aspects of the system's basic (bias independent) mode of operation. See, e.g., [42, 40] for more complex applications.

## 3.1 Writing Variable Sequences

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

**Details.** In addition to the 17 general primitives from Table 1 (not counting input/output primitives), there are two problem-specific primitives. Each

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

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

has two integer arguments: $Write(a1, a2)$ writes the contents of the storage cell indirectly addressed by the first argument into the variable indirectly addressed by the second argument. $Read(a1, a2)$ writes the contents of the variable indirectly addressed by the second argument into the work cell indirectly addressed by the second argument. See Table 2. Between two payoff events, each variable may be written only once (additional write operations have no effect). Write and read operations outside the valid ranges halt the current run.

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

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

**How difficult is this task?** For a number of reasons, the task is non-trivial — the system does not appear to have much built-in bias towards the task: (1) Only one of the 19 primitives ($Write$) may affect variable contents at all. But initially, the system does not even have such seemingly trivial knowledge — there is no built-in idea about which actions may lead to payoff. Therefore, it has to find out on its own. (2) The values referred to by the two arguments of $Write$ have to be identical and within the required ranges to lead to a useful result. (3) There are 30 *different* variables with 30 different values. Only one of them, namely $V_0$, is correctly re-initialized with its own address after each payoff event. (4) The most significant difficulty, however, is the continually changing policy environment.

**Changing policy environment.** The fact that the external variables are

occasionally reset does not imply that the policy environment does not change. In fact, many actions executed by the system change the contents of storage cells, which are *never* reset: as mentioned above, storage cells are like an external notebook and have to be viewed as part of the environment — typically, there are no exactly repeatable trials with identical initial conditions. And the storage cells are not just a negligible nuisance — they are essential for computing parameters for (e.g., self-modifying) instructions. To achieve significant performance improvement, the environmental conditions *have to* change due to actions executed by the system itself. This is an aspect that cannot be dealt with by *any* traditional reinforcement learning system, not even by naive exhaustive search, in a way that is sound.

**Comparison.** Performance was measured with and without self-modification capabilities. In the latter case, the primitives $IncP$ and $DecP$ had no effect. Both versions were run for $5 * 10^9$ time steps, corresponding to $5 * 10^6$ payoff events. Note that the *optimal* cumulative payoff is $1.5 * 10^8$. This value can be achieved only by a system with "optimal" prior bias — starting at birth, such a system keeps executing optimal actions without having to learn anything.

#### 3.1.1 Results Without Self-Modifications.

At system death, total payoff equaled about $7 * 10^6$. Average payoff per payoff event was about 1.4. Most of the total payoff (about $5 * 10^6$) could be attributed to the fact that $V_0$ was correctly re-initialized after each payoff event: the system received a little bit of payoff even in cases where it did not execute any *write* operations. As expected, average payoff intake did not significantly increase or decrease during the lifetime of the system. However, this was not *safely* predictable in advance, due to the changing environment.

#### 3.1.2 Results With Self-Modifications.

At system death, total payoff was about $1.16 * 10^8$ (recall that the theoretical optimum for a non-learning system with optimal initial bias would be $1.5 * 10^8$). To find out whether the incremental self-improvement paradigm did indeed lead to incremental self-improvement, let us have a look at the learning history (the results are slightly different from those reported in [32], where a slightly different implementation led to different calls of the random number generator).

**Self-generated reduction of numbers of probability modifications.** In the beginning, the system computed a lot of probability modifications but later preferred to decrease the number of probability modifications per time interval. After $10^9$ time steps, there were about 350,000 probability modifications per $10^8$ time steps. After $2 * 10^9$ time steps, there were about 40,000 probability modifications per $10^8$ time steps. Towards system death, there were about 20,000 probability modifications per $10^8$ time steps. Most of the useful SSMs computed either one or two probability modifications.

**Speed-up of payoff intake.** After $10^8$ time steps, the system already behaved much more deterministically than in the beginning. Average payoff per payoff event had increased from 1.4 to 15.8 (the optimal value being 30.0, of course), and the stack had 70 entries. These entries corresponded to 66 modifications of single cell probability distributions, computed by 45 SSMs — each being more "useful" than all the previous ones. *Storage already looked very messy.* For instance, almost all cells in the work area were filled with (partly big) integers quite different from the initial values. Recall that the storage is never re-initialized and has to be viewed as part of the policy environment.

**First maximal payoff.** After 1,436,383 payoff events, the system correctly had written *all* 30 variables for the first time, and received maximal payoff 30.0. Due to remaining non-determinism in the system, the current *average* payoff per payoff event (measured shortly afterwards, at time step 1,500,000,000) was about 21.7.

After 3,000,000 payoff events, current average payoff per payoff event was 25.6. But the stack had only 206 entries (corresponding to 174 "useful" SSMs). After 5,000,000 payoff events (at "system death"), the current average was about 26.0, with ongoing tendency to increase. By then, there were 224 stack entries. They corresponded to 192 SSMs, each being more "useful" than all the previous ones.

**Temporary speed-ups of performance improvement.** Performance did *not* increase smoothly during the lifetime of the system. Sometimes, no significant improvement took place for a time interval comparable to the entire learning time so far. Such "boring" time intervals were somtimes ended by unexpected sequences of rather quick improvements. Then progress slowed down again. Such temporary speed-ups of performance improvement indicate useful shifts of inductive bias, which may later be replaced by inductive bias created by the next "breakthrough".

**Evidence of "learning how to learn"?** A look at the stack entries revealed that many (but far from all) *useful* probability modifications focused on few program cells. Often, SSMs directly changing the probabilities of future SSMs were considered useful. For instance, 9 of the 224 stack entries at time step $5*10^9$ corresponded to "useful" probability modifications of the (self-referential) *IncP* action of the second program cell. Numerous entries corresponded to "useful" modifications of the *EndSelfMod* probability of various cells. Such stack entries may be interpreted as results of "adjusting the prior on the space of solution candidates" or "fine-tuning search space structure" or "learning to create directed mutations" or "learning how to learn".

## 3.2 A Navigation Task

**Non-Markovian variant of Sutton's Markovian maze task [47], with changing policy environment.** The external environment consists of a two-dimensional grid with 9 by 6 fields. $F_{i,j}$ denotes the field in the $i$-th row and

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

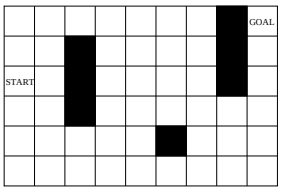


Figure 2: *Sutton's partially observable maze. It is a small part of the policy environment, which also includes storage cells with changing contents. See figure 1.*

**Why this task is non-trivial.** Unlike previous reinforcement learning systems, the system does *not* have a smart initial strategy for temporal credit assignment — it has to develop its own such strategies. Unlike with Sutton's original set-up, the system does not see a built-in unique representation of its current position on the grid. Its interface to the environment is non-Markovian [28]. This represents one reason why most traditional reinforcement learning systems do *not* have a sound strategy for solving this task. Another is the changing policy environment. See next paragraph.

**Changing policy environment.** The fact that the obstacles remain where they are and that the animat is occasionally reset to its start position does not imply that the policy environment does not change. In fact, many actions executed by the system change the contents of storage cells, which are *never* reset: as mentioned above, storage cells are like an external notebook and have

to be viewed as part of the policy environment — there are no exactly repeatable trials with identical initial conditions.

### 3.2.1 Results without Self-Modifications.

As with the previous task, the system was first tested for $10^{10}$ time steps with self-referential primitives *IncP* and *DecP* being switched off. Average trial length (number of time steps required to move from start to goal) was about 12600. This is consistent with results obtained by using Sutton's original set-up.

### 3.2.2 Results with Self-Modifications.

At system death (at time $10^{10}$), average trial length (including time required for SSA's pushing and popping processes, of course) was down to 56.6 time steps — more than 200 times faster than without self-improvement, and with ongoing tendency to decrease. Comparable results were obtained with additional runs.

As with the previous task, performance did not improve smoothly. The history of broken records reflects the history of performance improvements (the results from the run reported below are slightly different from those reported in [32], where a slightly different implementation led to different calls of the random number generator).

First, there was a rather quick sequence of improvements which lasted until time $4.6 * 10^7$. By then (after 155,741 payoff events), the shortest trial so far had taken 44 time steps. Then, the "current record" did not improve any more for a comparatively long time interval: $1.14 * 10^8$ time steps — the length of this "boring" time interval by far exceeded the entire previous learning time.

**Sudden improvement speed-up.** Then, quite unexpected to the observer, the system started to create a new sequence of additional improvements around time step $1.6 * 10^8$. At time $1.6 * 10^8$, the record was down to 42. At time $1.7 * 10^8$, the record was down to 40. At time $1.9 * 10^8$, the record was down to 38. At time $2.0 * 10^8$, the record was down to 36. Then, performance improvement slowed down again.

Similarly, much later, at time $2.4 * 10^9$, the record was 26. Then, not much happened during the next $10^9$ time steps. Suddenly, a new sequence of additional improvements started around time step $3.4 * 10^9$. At time $3.429 * 10^9$, the record was down to 24. At time $3.432 * 10^9$, the record was down to 22. At time $3.486 * 10^9$, the record was down to 20. Throughout this flurry of broken records, the number of stack entries steadily increased to 199. Apparently, the system made a little "revolutionary" discovery that permitted a sequence of smoother, additional directed self-mutations.

**Final system state.** At system death (time step $10^{10}$), the record was down to 19. The system's average payoff intake per time interval still had a tendency to increase. All in all, there were nearly $5 * 10^6$ SSMs during system life. The effects of only 200 of them, however, turned out to be worth keeping

(the others were countermanded by SSA): these 200 computed a total of 235 valid probability modifications (corresponding to only 235 stack entries). *Why so few?* Again, the reason is: "useful" SSMs are rare, because each has to lead to "better" results (faster reinforcement intake) than all previous ones.

**Messy storage environment.** In the end, the storage cells (which are part of the environment) looked very messy. Almost all work cells were filled with (partly big) integers quite different from their initial zero values. Still, without storage cells, the system could *not* achieve its dramatic performance improvement. It actually learned to make use of the changing policy environment.

**Evidence of "learning how to learn"?** As with the previous task, many useful SSMs directly computed valid modifications of probabilities of future SSMs ("learning to learn"). Compare the final paragraph of section 3.1.2.

**Stability of probability modifications.** With the experiments conducted so far, the top level hardly ever countermanded probability modifications other than the 10 most recent valid ones. For instance, once there were 120 stack entries, the 100 oldest stack entries appeared extremely safe and had a good chance to survive the entire system life.

**Revolutions.** In the tasks above, unexpected temporary speed-ups of performance improvements were observed. Even if the system appears to be stuck for a long time, the external observer never can be sure that it will not suddenly discover a new, "revolutionary" shift of bias that builds the basis for additional, smoother performance improvements. This is analoguous to the history of science itself. Informally, a "revolution" corresponds to a self-improvement with high *"conceptual jump size"* (an expression coined by Solomonoff [46]). One nice thing about open-ended incremental self-improvement is that there is no significant theoretical limit to the nature of the revolutions and to what the system may learn. This is, of course, due to the general nature of the underlying programming language.

**Inserting prior bias.** The few experiments above were designed to illustrate basic principles of the paradigm. They were based on low-level, assembler-like instructions (making even apparently simple tasks difficult — additional experiments using such low-level instructions can be found in [11, 53, 40, 42]). They are certainly not meant to convince the reader that from now on, he should combine the incremental self-improvement paradigm with the low-level programming language from section 2 and apply it to real world problems: of course, with large scale problems, it is desirable to *insert prior knowledge* into the system (if such knowledge is indeed available). With incremental self-improvement, *a priori* knowledge resides in the programmer's selection of primitives with problem-specific built-in bias (and in the payoff function he chooses). There is no reason why certain primitives should not be complex, time consuming programs by themselves, such as statistic classifiers, neural net learning algorithms, logic programs, etc. For instance, using different primitives for the navigation task from section 3.2 can greatly reduce the time required to achieve near-optimal trials. This paper, however, is not a study of the effects of different

kinds of initial bias.

# 4 SSA for Recurrent Neural Nets: a Multi-Agent Perspective

Now we attack the non-Markovian maze task from section 3.2 with multiple, very simple, non-self-modifying, learning agents. Each agent is in fact just a connection (whose current weight represents its current policy) in a fully recurrent neural net. A by-product of this research is a general reinforcement learning algorithm for such nets.

Viewing each connection as a separate agent may seem unconventional: standard AI research is used to more complex agents with full-sized knowledge bases etc. For the purposes of this paper, however, this is irrelevant: SSA does not care for the complexity of the agents.

Again, it should be emphasized that the following preliminary experiment in no way represents a systematic experimental analysis. The only purpose of the current section is to illustrate paragraph (*) (in the theoretical section 1).

**Basic set-up.** There is a fully recurrent neural net with 5 input, 4 output, and 5 hidden units. The variable activation of the $i$-th unit is denoted $o_i$ (initialized at time 0 with 0.0). $w_{ij} \in [-2.0, 2.0]$ denotes the real-valued, randomly initialized weight (a variable) on the connection $(i, j)$ from unit $j$ to $i$. There is a lifelong sequence of **"cycles"**. A cycle involves the following computations: the activation of the first (second, third, fourth) input unit is set to 0.0 if the field to the north (south, east, west) of the animat is blocked or does not exist, and set to 1.0 otherwise. Each noninput unit $i$ updates its variable activation $o_i$ (initialized at time 0 with 0.0) as follows: $o_i \leftarrow 1.0$ with probability $f(net_i)$, $o_i \leftarrow 0.0$ otherwise, where $f(x) = \frac{1}{1+e^{-x}}$, and the $net_i$ (initially 0) are sequentially updated as follows: $net_i \leftarrow net_i + \sum_j w_{ij} o_j$. If $i$ stands for an output unit, then $i$'s current *value* is defined as $v_i \leftarrow o_i + 0.05$. For each output unit $j$, there is an action $a_j$ (possible actions are *one-step-north*, *one-step-south*, *one-step-east*, *one-step-west*). $a_j$ is selected with a probability proportional to $v_j / \sum_i v_i$. The selected action gets executed. Whenever the animat hits the goal, there is constant reinforcement 1.0, the animat is transferred back to the start position (but the activations and *net*-values of the recurrent net are not reset), and the activation of the fifth input unit is set to 1.0 (0.0 otherwise).

**Weight stacks.** The current weight of a connection represents its current policy. For each connection $(i, j)$, there is a stack. Following the SSA principle (section 1), whenever a weight is modified (see below), the following values are pushed onto $(i, j)$'s stack: the current time, the total cumulative reinforcement so far, and the weight before the modification.

**Weight restauration.** After the goal is hit, or if the goal has not been hit for 10,000 cycles (in this case the animat is transferred back to the start

position), right before the next cycle, each connection sequentially pops entries from its stack and restores the corresponding previous weight values, until its SSC (see section 1) is satisfied.

**Weight modification.** So far, preliminary experiments were conducted only with very simple weight modification processes (corresponding to the $PMP_i$ from section 1). Weight modification is executed after each weight restauration (see above), and works as follows. For each weight $w_{ij}$ do: with small probability (0.05), replace $w_{ij}$ by a randomly chosen value in $[-2.0, 2.0]$. Of course, many alternative weight modification processes are possible, but this is irrelevant for the purposes of this paper.

**Measuring time.** By definition, each computation involving some weight — such as pushing, popping, computing unit activations etc.— costs one time step. Other computations do not cost anything. Still, measured time is of the order of total cpu-time.

**Where is the changing environment?** Although the animat is occasionally reset to its start position, the recurrent net activations and the *net*-values are not: the past may influence the future. Apart from this, each single connection's environment continually changes, simply because all the other connections in its environment keep changing. However, all connections receive the same global reinforcement signal. Hence, for each connection, the only way to speed up its *local* reinforcement intake is to contribute to speeding up *global* reinforcement intake. Since no connection can solve the task by itself, this enforces learning to cooperate.

**Results without SSA.** Again, for purposes of comparison, the system was first run with different weight initializations, and the weight changing mechanism being switched off. Often, the average number of cycles required to reach the goal exceeded 100,000 cycles.

Among the different weight initializations, one was picked that led to an average trial length of 2660 cycles (measured over 20,000,000 cycles). This initialization was used for SSA (see below), because the corresponding average trial length is of the same order of magnitude as the one obtained by using Sutton's original set-up.

**Results with SSA.** With multi-weight SSA being switched on, after about 10,000 "trials" (more precisely: 11,339,740 cycles — about $2,2 * 10^9$ time steps), the average trial length was down to 89. This corresponds to a speed-up factor of about 30. The shortest trial ever took 14 cycles, which is the theoretical optimum for Sutton's task. In the end, no weight stack had more than 7 entries (each followed by faster reinforcement intake than all the previous ones).

**Ongoing work.** It is intended to replace the random weight modification process by a process that may be strongly influenced by the current weights themselves. Following [30], the idea is to use activation patterns across special output units to address and modify the network's own weights. Then the recurrent net will be theoretically able to evolve its own weight modification strategies, to replace the random mutations by more directed self-mutations.

# 5 History of Ideas / Previous Work

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

**Meta-evolution.** My first attempts to come up with schemes for "true"[2] self-referential learning based on universal languages date back to 1987. They were partly inspired by a collaboration with Dickmanns and Winklhofer [7]. We used a genetic algorithm (GA) to evolve variable length programs for solving simple tasks (the system was implemented in PROLOG). Today, this approach would be classified as "Genetic Programming", e.g. [15]. [7] was in fact one of the first two papers on using GP-like algorithms to evolve assembler-like computer programs (but Cramer's work preceded ours [6]). We applied our system to simple tasks, including the "lawnmower problem" (later also studied by Koza, 1994). Since our programming language was general (for instance, we allowed for programs with loops etc.), our approach had at least the same potential as the one based on Koza's so-called "automatically defined functions". However, in subsequent work (1987 — 1995), we found GP unsatisfactory. One reason was: GP's way of constructing new code from old code does not improve itself — it always remains limited to the initial crossover and mutation mechanisms. This created a desire to improve the trivial mutation and crossover strategies used to construct new programs from old ones. In 1987, I developed an algorithmic scheme (called *"meta-evolution"*) for letting more sophisticated strategies be learned by a potentially infinite hierarchy of higher level GAs whose domains were to construct construction strategies. *Meta-evolution* recursively creates a growing hierarchy of pools of programs — higher-level pools containing program modifying programs being applied to lower-level programs and being rewarded based on lower-level performance. Details in [27].

**Collapsing meta-levels.** The explicit creation of "meta-levels" and "meta-meta-levels" seemed unnatural, however. For this reason, alternative systems based on "self-referential" languages were explored, the goal being to collapse all meta-levels into one [27]. At that time, however, no convincing global credit assignment strategy was provided.

**Self-referential neural nets.** Later work presented a neural network with the potential to run its own weight change algorithm [30, 29]. With this system,

[2] I am not talking about fixed learning algorithms for adjusting the parameters of others. For instance, GAs are sometimes used to adjust learning rates of gradient based neural nets, etc. Or a neural net is used to compute the weights of another neural net. In the literature, one can find quite a few approaches of this kind (too many to cite them all — I settle by citing none, not even my own). Although such approaches sometimes may have their merits, they do not deserve the attribute "self-referential" — the additional level typically just defers the credit assignment problem. However, there were a few apparently more general approaches. For instance, Lenat [17] reports that his EURISKO system was able to discover certain heuristics for discovering heuristics. His approach, however, as well as all other previous approaches I am aware of, were either quite limited (many essential aspects of system behavior being unmodifiable), and/or lacked a sound, convincing global credit assignment strategy (as embodied by the SSA pushing and popping processes).

top-level credit assignment is performed by gradient descent. This is unsatisfactory, however, due to problems with local minima, and because repeatable training sequences are required. In general, this makes it impossible to take the entire learning history into account.

**Algorithmic probability / Universal search.** Levin's universal search algorithm is theoretically optimal for certain "non-incremental" search tasks with exactly repeatable initial conditions. See Levin [19, 20]; see also Adleman [1]. There were a few attempts to extend universal search to incremental learning situations, where previous "trials" may provide information about how to speed up further learning, see e.g. [46, 22, 31]. For instance, to improve future performance, Solomonoff [45, 46] describes more traditional (as opposed to self-improving) methods for assigning probabilities to successful "subprograms". Alternatively, one of the actually implemented systems in [31] simply keeps successful code in its program area. This system was a conceptual starting point for the one in the current paper. With first attempts (in September 1994), the probability distributions underlying the Turing machine equivalent language required for universal search were modified heuristically. One strategy was to slightly increase the context-dependent probabilities of program cell contents used in successful programs, and then continue universal search based on the new probability distributions. With a number of experiments, this actually led to good results (at first glance, more impressive results than those in the current paper, at least if one does not take the lack of bias into account, as one should always do). The system, however, was unsatisfactory, precisely because there was no principled way of adjusting probability distributions. This criticism led to the ideas expressed in the current paper.

**Meta-version of universal search.** Without going into details, Solomonoff [46] mentions that self-improvement may be formulated as a time-limited optimization problem, thus being solvable by universal search. However, the straight-forward meta-version of universal search (generating and evaluating probability distributions in order of their Levin complexities [19]) just defers the credit assignment problem to the meta-level, and does *not* necessarily make optimal incremental use of computational resources and previous experience[3]. In fact, just like exhaustive search, but unlike SSA, universal search by itself cannot properly deal with changing environments. However, variants of universal search may be used as the parameter modification algorithms executed by PMPs (see section 1 and recent work with Marco Wiering [50, 42]).

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

# 6 Conclusion

It is easy to show that there can be no algorithm for general, unknown environments that is guaranteed to continually increase reinforcement intake per *fixed* time interval. For this reason, the reinforcement acceleration criterion (SSC) relaxes standard mesures of performance improvement, by allowing for consideration of *arbitrary* time intervals. The success-story algorithm (SSA) is guaranteed to achieve lifelong performance improvement according to this relaxed criterion. Since SSA takes into account all recursive long-term effects of policy modifications on all later policy modifications, and since it does not care for the nature of the policy modification processes, it provides a sound theoretical framework for "meta-learning", "meta-meta-learning", etc. Since SSA is environment-independent, it also provides a sound theoretical framework for "multi-agent learning", where each SSA-learning agent is part of the environment of the other agents.

There are many different ways of implementing SSA. Two of them are described in this paper. The first leads to a "self-referential" system using assembler-like primitive instructions to modify its own policy — the system's learning mechanism is embedded within the system, and accessible to self-manipulation. The second implementation leads to a general reinforcement learning algorithm for recurrent nets. Alternatively, however, the PMP from section 1 may be designed to execute arbitrary, conventional or non-conventional learning or search algorithms.

**Other SSA applications.** In recent work [50, 42], we combine SSA and Levin search (LS) [18, 20] to solve partially observable Markov decision problems (POMDPs). POMDPs received a lot of attention in the reinforcement learning community. LS is theoretically optimal for a wide variety of search problems including many POMDPs. We show that that LS can solve partially observable mazes (POMs) involving many more states and obstacles than those solved by various previous authors (here, LS also can easily outperform Q-learning). We then note, however, that LS is not necessarily optimal for "incremental" learning problems where experience with previous problems may help to reduce future search costs. For this reason, we introduce a heuristic, adaptive extension of LS (ALS) which uses experience to increase probabilities of instructions occurring in successful programs found by LS. To deal with cases where ALS does not lead to long term performance improvement, we use SSA as a safety belt. Experiments with additional POMs demonstrate: (a) ALS can dramatically reduce the search time consumed by successive calls of LS. (b) Additional significant speed-ups can be obtained by combining ALS and SSA.

In other recent work [53], we use SSA for a multi-agent system with agents much more complex than the ones in section 4. In fact, each agent uses incremental self-improvement as described in section 2. Experiments demonstrate the multi-agent system's effectiveness. For instance, a system consisting of three co-evolving agents chasing each other learns rather sophisticated, stochastic

predator and prey strategies. Additional applications of SSA to quite challenging, complex multiagent tasks are described in [41, 40].

## 7 Acknowledgements

I am grateful to Ray Solomonoff, Peter Dayan, Mike Mozer, Don Matthis, Clayton McMillan, and various NIPS*94 participants, for valuable comments/discussions on the first version of [32]. Many thanks to Sepp Hochreiter, Gerhard Weiß, Martin Eldracher, Margit Kinder, and Daniel Prelinger, for critical remarks on earlier drafts, and to Leslie Kaelbling, David Cohn, Tommi Jaakkola, and Andy Barto, for useful comments on later versions. Also, thanks to Marco Dorigo, Luca Gambardella, Rafal Salustowicz, Cristina Versino, and Marco Wiering, for hepful remarks on [34]. I am particularly indebted to Mark Ring for extensive and constructive criticism. The recent collaboration with Jieyu Zhao was supported by SNF grant 21-43'417.95 "Incremental Self-Improvement" and SNF grant 2100-49'144.96 "Long Short-Term Memory".

## References

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

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

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

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

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

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

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

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

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

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

[11] S. Heil. Universelle Suche und inkrementelles Lernen, diploma thesis, 1995. Fakultät für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

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

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

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

[15] J. R. Koza. *Genetic Programming II – Automatic Discovery of Reusable Programs.* MIT Press, 1994.

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

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

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

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

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

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

[22] W. Paul and R. J. Solomonoff. Autonomous theory building systems, 1991. Manuscript, revised 1994.

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

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

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

[26] C. Schaffer. Overfitting avoidance as bias. *Machine Learning*, 10:153–178, 1993.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

[34] J. Schmidhuber. Environment-independent reinforcement acceleration. Technical Report Note IDSIA-59-95, IDSIA, June 1995. Invited talk at Hongkong University of Science and Technology.

[35] J. Schmidhuber. A general method for multi-agent learning in unrestricted environments. In *Adaptation, Co-evolution and Learning in Multiagent Systems, Technical Report SS-96-01*, pages 84–87. American Association for Artificial Intelligence, Menlo Park, Calif., 1996.

[36] J. Schmidhuber. Realistic multi-agent reinforcement learning. In G. Weiss, editor, *Learning in Distributed Artificial Intelligence Systems. Working Notes of the 1996 ECAI Workshop*. 1996.

[37] J. Schmidhuber. A computer scientist's view of life, the universe, and everything. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Theory, Cognition, Applications*, volume 1337, pages 201–208. Lecture Notes in Computer Science, Springer, Berlin, 1997.

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

[39] J. Schmidhuber. What's interesting? Technical Report IDSIA-35-97, IDSIA, 1997.

[40] J. Schmidhuber, J. Zhao, and N. Schraudolph. Reinforcement learning with self-modifying policies. In S. Thrun and L. Pratt, editors, *Learning to learn*, pages 293–309. Kluwer, 1997.

[41] J. Schmidhuber, J. Zhao, and M. Wiering. Simple principles of metalearning. Technical Report IDSIA-69-96, IDSIA, 1996.

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

[43] H. P. Schwefel. Numerische Optimierung von Computer-Modellen. Dissertation, 1974. Published 1977 by Birkhäuser, Basel.

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

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

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

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

[48] 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.

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

[50] 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.

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

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

[53] 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.