# Solving POMDPs with Levin Search and EIRA

In *Machine Learning: Proceedings of the 13th international conference, 1996*

**Marco Wiering**
IDSIA
Corso Elvezia 36
CH-6900-Lugano (Switzerland)
marco@idsia.ch

**Jürgen Schmidhuber**
IDSIA
Corso Elvezia 36
CH-6900-Lugano (Switzerland)
juergen@idsia.ch

## Abstract

Partially observable Markov decision problems (POMDPs) recently received a lot of attention in the reinforcement learning community. No attention, however, has been paid to Levin's universal search through program space (LS), which is theoretically optimal for a wide variety of search problems including many POMDPs. Experiments in this paper first show 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 an 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 the recent technique of "environment-independent reinforcement acceleration" (EIRA) as a safety belt (EIRA currently is the only known method that guarantees a lifelong history of reward accelerations). 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 EIRA.

## 1 INTRODUCTION

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

**Search through program space is relevant for "POMDPs".** LS is a smart way of performing exhaustive search by "optimally" allocating time to programs computing solution candidates (details in section 2). Since programs written in a general language can use memory to disambiguate environmental inputs, LS is of potential interest for solving partially observable Markov decision problems (POMDPs), which received a lot of attention during recent years, e.g., (Jaakkola et al., 1995; Kaelbling et al., 1995; Ring, 1994; McCallum, 1993).

**Incremental extensions of LS.** LS by itself, however, is non-incremental: it does not use experience with previous tasks to speed up performance on new tasks. Therefore, it cannot immediately be used in typical, incremental reinforcement learning scenarios, where, in case of success, the system is given "reinforcement" (a real number) and tries to use that experience to maximize the sum of *future* reinforcements to be obtained during the remainder of system life. There have been proposals of "adaptive" variants of LS that modify LS' underlying probability distribution on program space (Solomonoff, 1986; Schmidhuber, 1995b).

None of these, however, can guarantee that the lifelong history of probability modifications will correspond to a lifelong history of reinforcement accelerations.

**EIRA**. The problem above has been addressed recently (Schmidhuber, 1996). At certain times in system life called checkpoints, a novel technique called "environment-independent reinforcement acceleration" (EIRA) invalidates certain modifications of the system's policy (the policy can be an arbitrary modifiable algorithm mapping environmental inputs and internal states to outputs and new internal states) such that all currently valid modifications are justified in the following sense: each still valid modification has been followed by long-term performance speed-up. To measure speed, at each checkpoint EIRA looks at the entire time interval that went by since the modification occurred. To do this efficiently, EIRA performs some backtracking (the time required for backtracking is taken into account for measuring performance speed-ups). EIRA is general in the sense that it can be combined with your favorite learning or search algorithm. Essentially, EIRA works as a safety belt where your favorite learning algorithm fails to improve things such that long term reinforcement intake speeds up (see details in section 4).

**Outline of paper.** Section 2 describes LS details. Section 3 presents the heuristic adaptation method (ALS — a simple, adaptive, incremental extension of LS related to the linear reward-inaction algorithm, e.g., (Kaelbling, 1993)). Section 4 briefly reviews EIRA and shows how to combine it with ALS. Section 5 presents results: in an illustrative application involving a maze that has many more states and obstacles than mazes solved by previous authors working on POMDPs, we show how LS can solve partially observable maze tasks with huge state spaces and non-trivial but low-complexity solutions (Q-learning fails to solve such tasks). Then we show that ALS can use previous experience to significantly reduce search time. Finally, we show that ALS augmented by EIRA can clearly outperform ALS by itself. Section 6 presents conclusions.

## 2 LEVIN SEARCH (LS)

**Basic concepts.** LS requires a set of $r$ primitive, prewired instructions $p_1, ..., p_r$ that can be composed to form arbitrary sequential programs. Essentially, LS generates and tests solution candidates $s$ (program outputs represented as strings over a finite alphabet) in order of their Levin complexities $Kt(s) = \min_q\{-logP_M(q) + log\ t(q,s)\}$, where $q$ stands for a program that computes $s$ in $t(q,s)$ time steps, and $P_M(q)$ is the probability of guessing $q$ according to a *fixed* Solomonoff-Levin distribution (Li and Vitányi, 1993) on the set of possible programs (in section 3, however, we will make the distribution variable).

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

**Practical implementation.** In our practical LS version, there is an upper bound $k$ on program length (due to obvious storage limitations). $a_i$ denotes the address of the $i$-th instruction. Each program is generated incrementally: first we select an instruction for $a_1$, then for $a_2$, etc. $P_M$ is given by a matrix $M$, where $M_{ij}$ ($i \in 1, ..., k$, $j \in 1, ..., r$) denotes the probability of selecting $p_j$ as the instruction at address $a_i$, given that the first $i-1$ instructions have already been selected. The probability of a program is the product of the probabilities of its constituents.

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

**Levin search(problem $N$, probability matrix $M$)**

```
    (1) Set $T$, the number of the current phase, equal to 1. In what follows, let $\phi(T)$ denote
        the set of *not yet executed* programs $q$ satisfying $P_M(q) \geq \frac{1}{T}$.
    (2) **Repeat**
        (2.1) **While** $\phi(T) \neq \{\}$ and no solution found **do**: Generate a program $q \in \phi(T)$,
              and run $q$ until it either halts or until it used up $\frac{P_M(q)T}{c}$ steps.
              If $q$ computed a solution for $N$, return $q$ and exit.
        (2.2) Set $T := 2T$
    **until** solution found or $T \geq T_{MAX}$.
    Return $\{\}$.
```

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

## 3 ADAPTIVE LS (ALS)

As mentioned above, LS is not necessarily optimal for "incremental" learning problems where experience with previous problems may help to reduce future search costs. To make an incremental search method out of non-incremental LS, we introduce a simple, heuristic, adaptive LS extension (ALS) that uses experience with previous problems to adaptively modify LS' underlying probability distribution. ALS essentially works as follows: whenever LS found a program $q$ that computed a solution for the current problem, the probabilities of $q$'s instructions $q_1, q_2, \dots, q_{l(q)}$ are increased (here $q_i \in \{p_1, \dots, p_r\}$ denotes $q$'s $i$-th instruction, and $l(q)$ denotes $q$'s length — if LS did not find a solution ($q$ is the empty program), then $l(q)$ is defined to be 0). The probability adjustment is controlled by a learning rate $\gamma$ $(0 < \gamma < 1)$. ALS is related to the linear reward-inaction algorithm (e.g., (Kaelbling, 1993)) — the main difference is: ALS uses LS to search through *program space* as opposed to single action space. As in section 2, the probability distribution $P_M$ is determined by $M$. Initially, all $M_{ij} = \frac{1}{r}$. However, given a sequence of problems $(N_1, N_2, ..., N_k)$, the $M_{ij}$ may undergo changes caused by ALS:

```
ALS (problems $(N_1, N_2, ..., N_k)$, variable matrix $M$)

    for $i := 1$ to $k$ do:
      $q :=$ Levin search($N_i$, $M$); Adapt($q$, $M$).
```

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

```
Adapt(program $q$, variable matrix $M$)

    for $i := 1$ to $l(q)$, $j := 1$ to $r$ do:
      if $(q_i = p_j)$ then $M_{ij} := M_{ij} + \gamma(1 - M_{ij})$
      else $M_{ij} := (1 - \gamma)M_{ij}$
```

**Critique of adaptive LS.** Although ALS seems a reasonable first step towards making LS adaptive (and actually leads to very nice experimental results — see section 5), there is no theoretical proof that it will generate only probability modifications that will speed up the process of finding solutions to new tasks – sometimes ALS may produce harmful instead of beneficial results. To address this issue, in the next section we augment ALS by a recent backtracking technique called "Environment-Independent Reinforcement Acceleration" (EIRA). EIRA ensures that the system will keep only probability modifications representing a lifelong history of performance improvements.

## 4 EIRA FOR ALS

**Basic set-up.** At a given time, the variable matrix $M$ above represents the system's current *policy*. Each call of the procedure **Adapt** (invoked by ALS) modifies the policy. Let us consider the complete sequence of such calls spanning the entire system life, which starts at time 0 and ends at some point in the future (time flows in one direction — there are no resets to 0). By definition, the $i$-th call occurs at time $t^i$, is denoted $\mathbf{Adapt}_i$, and generates a policy modification denoted by $M(i)$. In between two calls, a certain amount of time is consumed by **Levin search** (details about how time is measured will follow in the section on experiments).

**Goal.** Whenever ALS as above finds a solution, the system receives a reward of $+1.0$. The goal is to receive as much reward as quickly as possible, by generating policy changes that minimize the computation time required by *future* calls of **Levin search**. Let us denote the sum of all reinforcements between time 0 and time $t > 0$ by $R(t)$.

**Reinforcement/time ratios.** Right before each call of **Adapt**, EIRA (see details below) essentially *invalidates* those policy modifications that are not consistent with the so-called reinforcement acceleration criterion (RAC). To define RAC, we first introduce a measure indicating how useful $\mathbf{Adapt}_i$ has been until the current time $t$ — we simply compute the reinforcement/time ratio $Q(i, t)$:

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

At a particular time $t$, RAC is satisfied if for each $\mathbf{Adapt}_i$ that computed a still valid (not yet invalidated) policy modification M(i), we have

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

(b) $\forall k < i$ such that $M(k)$ is still valid: $Q(i,t) > Q(k,t)$.

Obviously, RAC only holds if the history of still valid policy modification represents a history of long-term reinforcement accelerations — each still valid modification has to be followed by more average reinforcement per time than all the previous ones. *Note that the success of some* **Adapt** *call depends on the success of all later* **Adapt** *calls, for which it is "setting the stage"!* This represents an essential difference to previous performance criteria.

**EIRA** uses a stack to store information about policy modifications computed by calls of **Adapt**. Right before **Adapt**$_i$ is executed, EIRA restores (if necessary) previous policies such that RAC holds. EIRA is based on two processes:

**(1) Pushing.** At time $t^i$, EIRA pushes the following information on the stack: $t^i$, $R(t^i)$, and the previous values of those columns of $M$ (representing probability distributions) changed by **Adapt**$_i$ (this information may be needed for later restoring the old policy, as it used to be before $M(i)$ was generated).

**(2) Popping.** Right before each call of **Adapt**, while none of the following conditions (1-3) holds, EIRA pops probability vectors off the stack and *invalidates* the corresponding policy modifications, by restoring the previous policies.

(1) $Q(k,t) > Q(l,t)$, where $M(k)$ and $M(l)$ are still valid, and $M(l)$ is the most recent valid policy modification generated earlier than $M(k)$.

(2) $Q(k,t) > \frac{R(t)}{t}$, where $M(k)$ is the only valid policy.

(3) the stack is empty.

**Theoretical soundness.** Using induction, it can be shown that this backtracking procedure ensures that RAC holds after each popping process (Schmidhuber, 1995a).

At any given time, EIRA's straight-forward *generalization assumption* is: modifications that survived the most recent popping process will remain useful. In general environments, what else could be assumed? Note that at any given time in system life, we have only one single "training example" to evaluate the current long-term usefulness of any given previous **Adapt** call, namely the average reinforcement per time since it occurred. During the next popping process, however, EIRA will reevaluate "usefulness so far" of still valid modifications.

**To conclude:** EIRA again and again 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, EIRA invalidates policy modifications until RAC is fulfilled again. EIRA's stack-based backtracking is efficient in the sense that only the two most recent still valid modifications have to be considered at a given time (although a *single* popping process may invalidate *many* modifications).

## 5 PARTIALLY OBSERVABLE MAZE PROBLEMS

This section will describe experiments validating the usefulness of LS, ALS, and EIRA. To begin with, in an illustrative application with a partially observable maze that has many more states and obstacles than those presented by various authors at ML95, we show how LS by itself can solve POMDPs with huge state spaces but low-complexity solutions (Q-learning variants fail to solve these tasks). Then we present experiments where the task requires to find a *stochastic* policy for finding multiple goals. We show that ALS can use previous experience to speed-up the process of finding solutions, and that EIRA combined with ALS (for short: ALS+EIRA) can outperform ALS by itself.

### 5.1 EXPERIMENT 1: A BIG PARTIALLY OBSERVABLE MAZE (POM)

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

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

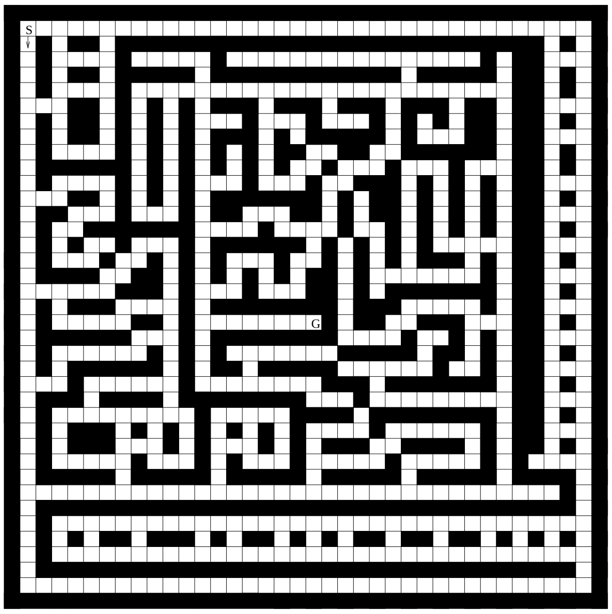

Figure 1: *An apparently complex, partially observable* $39 \times 38$*-maze with a low-complexity shortest path from start S to goal G involving 127 steps. Despite the relatively large state space, the agent can implicitly perceive only one of three highly ambiguous types of input, namely "front is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

two parameters: `nr-times` $\in 1, 2, \ldots, 6$, and `address` $\in 1, 2, \ldots, top$, where *top* is the highest address in the current program. `Jump` uses an additional hidden variable `nr-times-to-go` which is initially set to `nr-times`. The semantics are: If `nr-times-to-go` $> 0$, continue execution at address `address`. If $0 <$ `nr-times-to-go` $< 6$, decrement `nr-times-to-go`. If `nr-times-to-go` $= 0$, set `nr-times-to-go` to `nr-times`. Note that `nr-times` $= 6$ may cause an infinite loop. The `Jump` instruction is essential for exploiting the possibility that solutions may consist of *repeatable* action sequences and "subprograms" (thus having low algorithmic complexity). LS' incrementally growing time limit automatically deals with those programs that don't halt, by preventing them from consuming too much time.

As mentioned in section 2, the probability of a program is the product of the probabilities of its constituents. To deal with probabilities of the two `Jump` parameters, we introduce two additional variable matrices, $\bar{M}$ and $\hat{M}$. For a program with $l \leq k$ instructions, to specify the conditional probability $\bar{M}_{ij}$ of a jump to address $a_j$, given that the instruction at address $a_i$ is `Jump` ($i \in 1, ..., l$, $j \in 1, ..., l$), we first normalize the entries $\bar{M}_{i1}, \bar{M}_{i2}, ..., \bar{M}_{il}$ (this ensures that the relevant entries sum up to 1). Provided the instruction at address $a_i$ is `Jump`, for $i \in 1, ..., k$, $j \in 1, ..., 6$, $\hat{M}_{ij}$ specifies the probability of the `nr-times` parameter being set to $j$. Both $\bar{M}$ and $\hat{M}$ are initialized uniformly and are adapted by ALS just like $M$ itself.

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

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

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

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

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

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

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

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

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

**ALS: single tasks versus multiple tasks.** If we use the adaptive LS extension (ALS) for a single task as the one above (by repeatedly applying LS to the same problem and changing the underlying probability distribution in between successive calls according to section 3), then the probability matrix rapidly converges such that late LS calls find the solution almost immediately. This is not very interesting, however — once the solution to a single problem is found (and there are no additional problems), there is no point in investing additional efforts into probability updates. ALS is more interesting in cases where there are multiple tasks, and where the solution to one task conveys some but not all information helpful for solving additional tasks. This is what the next section is about.

## 5.2 EXPERIMENT 2: LEARNING TO FIND MULTIPLE GOALS

**Task.** The second experiment shows that ALS can use experience to significantly reduce average search time consumed by successive LS calls in cases where there are multiple tasks to solve, and that ALS can be further improved by combining it with EIRA. To be able to run a sufficient number of simulations to obtain statistically significant results, we replace the big maze from Figure 1 by the smaller maze from Figure 2, which indicates 10 different goal positions. At a given

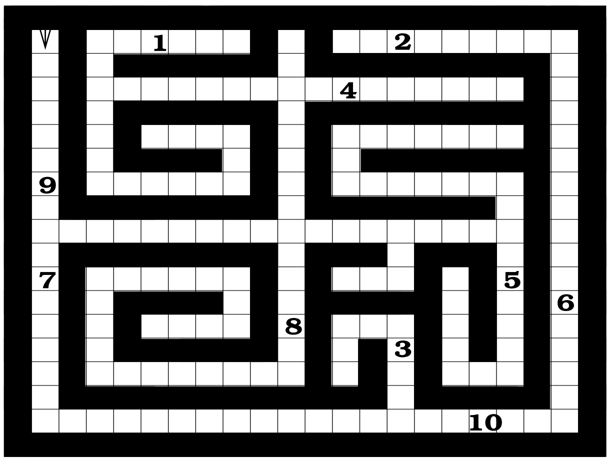

Figure 2: *An example of 10 goal positions to be found in a* $19 \times 22$ *maze. The arrow indicates the agent's initial position and direction.*

time, only one of the goal positions contains "food". But the agent does not know which! Whenever the agent finds food, it takes it home to eat it. Next time, food appears in another location. Therefore, there is no deterministic program that always can generate a shortest path. The best the agent can do is to learn a *stochastic* policy minimizing expected search time.

One experiment consists of 10 simulations. For each simulation, 10 goal positions are randomly generated. Each simulation consists of 100 "epochs", where each epoch consists of 10 "runs", where during the $i$-th run the $i$-th goal position has to be found (starting from the start state). $M$, $\bar{M}$ and $\hat{M}$ are adjusted whenever a solution is found.

**Comparison.** We compared (1) Random Search, (2) ALS and (3) the ALS+EIRA combination, where EIRA restores old policies if necessary, always right before ALS' matrices are adapted. For the LS calls triggered during ALS' runtime, we set $c$ to 0.02. ALS performed best with a learning rate $\gamma = 0.05$. ALS+EIRA performed best with a learning rate of 0.08.

**Results.** All methods always found all 10 goal positions before running into the time-limit ($10^8$ steps for each goal). The learning curves are given in figure 3. In the beginning, the LS calls triggered by ALS take a long time, but after a few epochs the search cost improves by a factor of about 100 (for scaling reasons, Figure 3 does not even show the initial search costs). Table 1 shows the average number of steps required to find all 10 goal positions in the $100^{th}$ epoch of the 10 simulations. The results show (1) that ALS finds

Table 1: *The number of steps required by ALS, ALS+EIRA and random search (RS) to find all 10 goal positions (always starting from the start position). The table shows the average number of steps (in thousands) consumed during the* $100^{th}$ *epoch. SD is the standard deviation, and MAX (MIN) stands for worst (best) performance in ten simulations with ten different goal positions (see Figure 3 to see that ALS dramatically reduces search costs for successive LS calls).*

| Method | Average | SD | MAX | MIN |
|---|---|---|---|---|
| ALS + EIRA | 7.5 | 3.7 | 12.5 | 3.3 |
| ALS | 19.2 | 17.3 | 65.5 | 4.6 |
| RS | 168 | 284 | 1005 | 10.2 |

the 10 goal positions on average much faster than random search. The table also shows (2) that the use of EIRA significantly further improves the results (the additional speed-up factor exceeds 2.0).

**The safety belt effect.** Figure 4 plots number of epochs against the average probability of programs computing solutions. The figure shows that ALS+EIRA tends to keep the probabilities lower than ALS by itself: high program probabilities are not always beneficial.

Effectively, EIRA is controlling the prior on the search space such that overall average search time is reduced. The total stack size (the number of instruction probability vectors on the stack) after the $100^{th}$ trial was 108 on average. Since the total amount of policy modifications is (number of goal positions) * (number of epochs) * (average solution length) = 3800, EIRA kept only about 3% of all modifications! The remaining 97% were deemed unworthy, because they were not observed to be followed by long-term reinforcement speed-ups. Clearly, EIRA prevents ALS from overdoing its policy modifications ("safety belt effect").

## 6 CONCLUSION

This paper makes three major points: **(1)** Levin search by itself can be useful for solving POMDPs. This has been demonstrated with a non-trivial, partially observable maze containing significantly more states and obstacles than those used to demonstrate the usefulness of previous POMDP algorithms, e.g., (McCallum, 1993; Ring, 1994; Littman, 1994; Cliff and Ross, 1994): for instance, McCallum's cheese maze has only 11 free fields, and Ring's largest maze is a 9×9-maze. This also illustrates that search in program

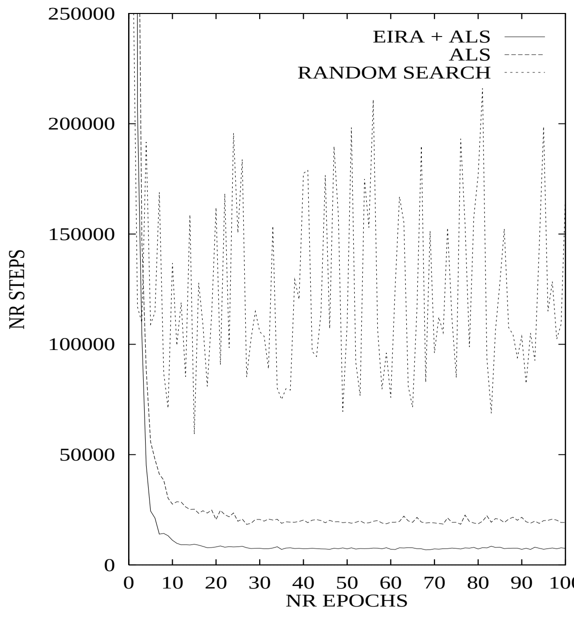

Figure 3: *Average number of steps required to find all 10 goal positions (each time starting anew from the start position), plotted against the number of epochs. The comparison involves random search, ALS, and ALS augmented by EIRA.*

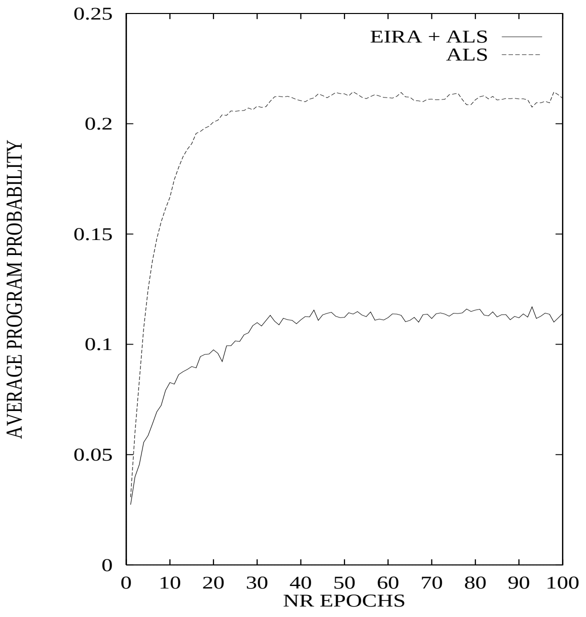

Figure 4: *The average probability of programs computing solutions. Without EIRA, the average probability of certain solution-computing programs is much higher. This does not improve search time, however (compare figure 3).*

space can have significant advantages over methods searching through simple action space, provided the algorithmic complexity of the solutions is low. **(2)** A straightforward, incremental, adaptive extension of non-incremental LS (ALS — introduced in this paper) can dramatically reduce the time consumed by successive calls of LS in cases where there are multiple tasks to solve. **(3)** ALS can further significantly benefit from "environment-independent reinforcement acceleration" (EIRA). EIRA helps to get rid of ALS-generated policy modifications for which there is no evidence that they contribute to long-term performance improvement. This actually provides the first example of how EIRA can improve heuristic learning methods in lifelong learning situations. Due to EIRA's generality (it is not limited to run in conjunction with ALS, but can be combined with all kinds of policy modifying learning algorithms), these results add to making EIRA appear a promising, general paradigm.

**Future Work.** ALS should be extended such that it not only adapts the probability distribution underlying LS, but also the initial time limit required by LS' first phase (the current ALS version keeps the latter constant, which represents a potential loss of efficiency). Again, EIRA should be combined with this ALS extension.

EIRA should also be combined with other (e.g., genetic) learning algorithms, especially in situations where the applicability of a given algorithm $A$ is questionable because the environment does not satisfy the preconditions that would make $A$ sound. EIRA can at least guarantee that those of $A$'s policy modifications that appear to have negative long-term effects on further learning processes are countermanded. Indeed, in separate POMDP experiments we were already able to show that EIRA can improve standard Q-learning's performance (recall that POMDP applications of Q-learning are not theoretically sound, although many authors *do* apply Q-variants to POMDPs). Another interesting application area may be the field of bucket-brigade based classifier systems: (Cliff and Ross, 1994) show that such systems tend to be unstable and forget good solutions. Here EIRA could unfold its safety belt effect.

## Acknowledgements

Thanks for valuable discussions to Rafał Sałustowicz, Sepp Hochreiter, and Jieyu Zhao (supported by SNF grant 21-43'417.95 "incremental self-improvement").

## References

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

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

Jaakkola, T., Singh, S. P., and Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable markov decision problems. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7, to appear.* MIT Press, Cambridge MA.

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

Kaelbling, L., Littman, M., and Cassandra, A. (1995). Planning and acting in partially observable stochastic domains. Technical report, Brown University, Providence RI.

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

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

Li, M. and Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications.* Springer.

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

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Machine Learning: Proceedings of the Tenth International Conference.* Morgan Kaufmann, Amherst, MA.

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

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

Schmidhuber, J. (1995a). Environment-independent reinforcement acceleration. Technical Report Note IDSIA-59-95, IDSIA. Invited talk at Hongkong University of Science and Technology.

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

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

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

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

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