Sequential Decision Making Based on Direct Search

Jürgen Schmidhuber

IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland

1 Introduction

The most challenging open issues in sequential decision making include partial observability of the decision maker's environment, hierarchical and other types of abstract credit assignment, the learning of credit assignment algorithms, and exploration without *a priori* world models. I will summarize why direct search (DS) in policy space provides a more natural framework for addressing these issues than reinforcement learning (RL) based on value functions and dynamic programming. Then I will point out fundamental drawbacks of traditional DS methods in case of stochastic environments, stochastic policies, and unknown temporal delays between actions and observable effects. I will discuss a remedy called the success-story algorithm, show how it can outperform traditional DS, and mention a relationship to market models combining certain aspects of DS and traditional RL.

Policy learning. A learner's modifiable parameters that determine its behavior are called its policy. An algorithm that modifies the policy is called a learning algorithm. In the context of sequential decision making based on reinforcement learning (RL) there are two broad classes of learning algorithms: (1) methods based on dynamic programming (DP) (Bellman, 1961), and (2) direct search (DS) in policy space. DP-based RL (DPRL) learns a value function mapping input/action pairs to expected discounted future reward and uses online variants of DP for constructing rewarding policies (Samuel, 1959; Barto, Sutton, & Anderson, 1983; Sutton, 1988; Watkins, 1989; Watkins & Dayan, 1992; Moore & Atkeson, 1993; Bertsekas & Tsitsiklis, 1996). DS runs and evaluates policies directly, possibly building new policy candidates from those with the highest evaluations observed so far. DS methods include variants of stochastic hill-climbing (SHC), evolutionary strategies (Rechenberg, 1971; Schwefel, 1974), genetic algorithms (GAs) (Holland, 1975), genetic programming (GP) (Cramer, 1985; Banzhaf, Nordin, Keller, & Francone, 1998), Levin Search (Levin, 1973, 1984), and adaptive extensions of Levin Search (Solomonoff, 1986; Schmidhuber, Zhao, & Wiering, 1997b).

Outline. DS offers several advantages over DPRL, but also has some drawbacks. I will list advantages first (section 2), then describe an illustrative task unsolvable by DPRL but trivially solvable by DS (section 3), then mention a few theoretical results concerning DS in general search spaces (section 4), then point out a major problem of DS (section 5), and offer a remedy (section 6 and section 7).

2 Advantages of Direct Search

2.1 DS Advantage 1: No States

Finite time convergence proofs for DPRL (Kearns & Singh, 1999) require (among other things) that the environment can be quantized into a finite number of discrete states, and that the topology describing possible transitions from one state to the next, given a particular action, is known in advance. Even if the real world was quantizable into a discrete state space, however, for all practical purposes this space will be inaccessible and remain unknown. Current proofs do not cover apparently minor deviations from the basic principle, such as the world-class RL backgammon player (Tesauro, 1994), which uses a nonlinear function approximator to deal with a large but finite number of discrete states and, for the moment at least, seems a bit like a miracle without full theoretical foundation. Prior knowledge about the topology of a network connecting discrete states is also required by algorithms for partially observable Markov decisicion processes (POMDPs), although they are more powerful than standard DPRL, e.g., (Kaelbling, Littman, & Cassandra, 1995; Littman, Cassandra, & Kaelbling, 1995). In general, however, we do not know *a priori* how to quantize a given environment into meaningful states.

DS, however, completely avoids the issues of value functions and state identification — it just cares for testing policies and keeping those that work best.

2.2 DS Advantage 2: No Markovian Restrictions

Convergence proofs for DPRL also require that the learner's current input conveys all the information about the current state (or at least about the optimal next action). In the real world, however, the current sensory input typically tells next to nothing about the "current state of the world," if there is such a thing at all. Typically, memory of previous events is required to disambiguate inputs. For instance, as your eyes are sequentially scanning the visual scene dominated by this text you continually decide which parts (or possibly compressed descriptions thereof) deserve to be represented in short-term memory. And you have presumably *learned* to do this, apparently by some unknown, sophisticated RL method fundamentally different from DPRL.

Some DPRL variants such as $Q(\lambda)$ are limited to a very special kind of exponentially decaying short-term memory. Others simply ignore memory issues by focusing on suboptimal, memory-free solutions to problems whose optimal solutions do require some form of short-term memory (Jaakkola, Singh, & Jordan, 1995). Again others can in principle find optimal solutions even in partially observable environments (POEs) (Kaelbling et al., 1995; Littman et al., 1995), but they (a) are practically limited to very small problems (Littman, 1996), and (b) do require knowledge of a discrete state space model of the environment. To various degrees, problem (b) also holds for certain hierarchical RL approaches to memory-based input disambiguation (Ring, 1991, 1993, 1994; McCallum, 1996; Wiering & Schmidhuber, 1998). Although no discrete models are necessary for

DPRL systems with function approximators based on recurrent neural networks (Schmidhuber, 1991c; Lin, 1993), the latter do suffer from a lack of theoretical foundation, perhaps even more so than the backgammon player.

DS, however, does not care at all for Markovian conditions and full observability of the environment. While DPRL is essentially limited to learning reactive policies mapping current inputs to output actions, DS in principle can be applied to search spaces whose elements are general algorithms or programs with time-varying variables that can be used for memory purposes (Williams, 1992; Teller, 1994; Schmidhuber, 1995; Wiering & Schmidhuber, 1996; Sałustowicz & Schmidhuber, 1997).

2.3 DS Advantage 3: Straight-forward Hierarchical Credit Assignment

There has been a lot of recent work on hierarchical DPRL. Some researchers address the case where an external teacher provides intermediate subgoals and/or prewired macro actions consisting of sequences of lower-level actions (Moore & Atkeson, 1993; Tham, 1995; Sutton, 1995; Singh, 1992; Humphrys, 1996; Digney, 1996; Sutton, Singh, Precup, & Ravindran, 1999). Others focus on the more ambitious goal of automatically learning useful subgoals and macros (Schmidhuber, 1991b; Eldracher & Baginski, 1993; Ring, 1991, 1994; Dayan & Hinton, 1993; Wiering & Schmidhuber, 1998; Sun & Sessions, 2000). Compare also work presented at the recent NIPS*98 workshop on hierarchical RL organized by Doina Precup and Ron Parr (McGovern, 1998; Andre, 1998; Moore, Baird, & Kaelbling, 1998; Bowling & Veloso, 1998; Harada & Russell, 1998; Wang & Mahadevan, 1998; Kirchner, 1998; Coelho & Grupen, 1998; Huber & Grupen, 1998).

Most current work in hierarchical DPRL aims at speeding up credit assignment in fully observable environments. Approaches like HQ-learning (Wiering & Schmidhuber, 1998), however, additionally achieve a qualitative (as opposed to just quantitative) decomposition by learning to decompose problems that cannot be solved at all by standard DPRL into several DPRL-solvable subproblems and the corresponding macro-actions.

Generally speaking, non-trivial forms of hierarchical RL almost automatically run into problems of partial observability, even those with origins in the MDP framework. Feudal RL (Dayan & Hinton, 1993), for instance, is subject to such problems (Ron Williams, personal communication). As Peter Dayan himself puts it (personal communication): *"Higher level experts are intended to be explicitly ignorant of the details of the state of the agent at any resolution more detailed than their action choice. Therefore, the problem is really a POMDP from their perspective. It's easy to design unfriendly state decompositions that make this disastrous. The key point is that it is highly desirable to deny them information – the chief executive of [a major bank] doesn't really want to know how many paper clips his most junior bank clerk has – but arranging for this to be benign in general is difficult."*

In the DS framework, however, hierarchical credit assignment via frequently used, automatically generated subprograms becomes trivial in principle. For in-

stance, suppose policies are programs built from a general programming language that permits parameterized conditional jumps to arbitrary code addresses (Dickmanns, Schmidhuber, & Winklhofer, 1987; Ray, 1992; Wiering & Schmidhuber, 1996; Schmidhuber et al., 1997b; Schmidhuber, Zhao, & Schraudolph, 1997a). DS will simply keep successful hierarchical policies that partially reuse code (subprograms) via appropriate jumps. Again, partial observability is not an issue.

2.4 DS Advantage 4: Non-Hierarchical Abstract Credit Assignment

Hierarchical learning of macros and reusable subprograms is of interest but limited. Often there are *non*-hierarchical (nevertheless exploitable) regularities in solution space. For instance, suppose we can obtain solution B by replacing every action "*turn(right)*" in solution A by "*turn(left).*" B will then be regular in the sense that it conveys little additional conditional algorithmic information, given A (Solomonoff, 1964; Kolmogorov, 1965; Chaitin, 1969; Li & Vitányi, 1993), that is, there is a short algorithm computing B from A. Hence B should not be hard to learn by a smart RL system that already found A. While DPRL cannot exploit such regularities in any obvious manner, DS in general algorithm spaces does not encounter any fundamental problems in this context. For instance, all that is necessary to find B may be a modification of the parameter *"right"* of a single instruction *"turn(right)"* in a repetitive loop computing A (Schmidhuber et al., 1997b).

2.5 DS Advantage 5: Metalearning Potential

In a given environment, which is the best way of collecting reward? Hierarchical RL? Some sort of POMDP-RL, or perhaps analogy-based RL? Combinations thereof? Or other nameless approaches to exploiting algorithmic regularities in solution space? A smart learner should find out by itself, using experience to improve its own credit assignment strategy (metalearning or "learning to learn") (Lenat, 1983; Schmidhuber, 1987). In principle, such a learner should be able to run *arbitrary* credit assignment strategies, and discover and use "good" ones, without wasting too much of its limited life-time (Schmidhuber et al., 1997a). It seems obvious that DPRL does not provide a useful basis for achieving this goal, while DS seems more promising as it does allow for searching spaces populated with arbitrary algorithms, including metalearning algorithms. I will come back to this issue later.

Disclaimer: of course, solutions to almost all possible problems are irregular and do not share mutual algorithmic information (Kolmogorov, 1965; Solomonoff, 1964; Chaitin, 1969; Li & Vitányi, 1993). In general, learning and generalization are therefore impossible for any algorithm. But it's the comparatively few, exceptional, low-complexity problems that receive almost all attention of computer scientists.

2.6 Advantage 6: Exploring Limited Spatio-Temporal Predictability

Knowledge of the world may boost performance. That's why exploration is a major RL research issue. How should a learner explore a spatio-temporal domain? By predicting and learning from success/failure what's predictable and what's not.

Most previous work on exploring unknown data sets has focused on selecting single training exemplars maximizing traditional information gain (Fedorov, 1972; Hwang, Choi, Oh, & II, 1991; MacKay, 1992; Plutowski, Cottrell, & White, 1994; Cohn, 1994). Here typically the concept of a surprise is defined in Shannon's sense (Shannon, 1948): some event's surprise value or information content is the negative logarithm of its probability. This inspired simple reinforcement learning approaches to pure exploration (Schmidhuber, 1991a; Storck, Hochreiter, & Schmidhuber, 1995; Thrun & Möller, 1992) that use adaptive predictors to predict the entire next input, given current input and action. The basic idea is that the action-generating module gets rewarded in case of predictor failures. Hence it is motivated to generate action sequences leading to yet unpredictable states that are "informative" in the classic sense. Some of these explorers actually like white noise simply because it is so unpredictable, thus conveying a lot of Shannon information. Compare alternative, hardwired exploration strategies (Sutton & Pinette, 1985; Kaelbling, 1993; Dayan & Sejnowski, 1996; Koenig & Simmons, 1996).

Most existing systems either always predict all details of the next input or are limited to picking out simple statistic regularities such as "performing action A in discrete, fully observable environmental state B will lead to state C with probability 0.8." They are not able to limit their predictions solely to certain computable aspects of inputs (Schmidhuber & Prelinger, 1993) or input sequences, while ignoring random and irregular aspects. For instance, they cannot even express (and therefore cannot find) complex, abstract, predictable regularities such as *"executing a particular sequence of eye movements, given a history of incomplete environmental inputs partially caused by a falling glass of red wine, will result in the view of a red stain on the carpet within the next 3 seconds, where details of the shape of the stain are expected to be unpredictable and left unspecified."*

General spatio-temporal abstractions and limited predictions of this kind apparently can be made only by systems that can run fairly general algorithms mapping input/action sequences to compact internal representations conveying only certain relevant information embedded in the original inputs. For instance, there are many different, realistic, plausible red stains — all may be mapped onto the same compact internal representation predictable from all sequences compatible with the abstraction "falling glass." If the final input sequence caused by eye movements scanning the carpet does not map onto the concept "red stain" (because the glass somehow decelerated in time and for some strange reason never touched the ground), there will be a surprise. There won't be a surprise, however, if the stain exhibits a particular, unexpected, irregular shape,

because there was no explicit confident expectation of a particular shape in the first place.

The central questions are: In a given environment, in absence of a state model, how to extract the predictable concepts corresponding to algorithmic regularities that are not already known? How to discover novel spatio-temporal regularities automatically among the many random or unpredictable things that should be ignored? Which novel input sequence-transforming algorithms do indeed compute reduced reduced internal representations permitting reliable predictions?

Usually we cannot rely on a teacher telling the system which concepts are interesting, such as in the EURISKO system (Lenat, 1983). The DPRL framework is out of the question due to issues of partial observability. Lookup-table approaches like those used in more limited scenarios are infeasible due to the huge number of potentially interesting sequence-processing, event-memorizing algorithms. On the other hand, it is possible to use DS-like methods for building a "curious" embedded agent that differs from previous explorers in the sense that it can limit its predictions to fairly arbitrary, computable aspects of event sequences and thus can explicitly ignore almost arbitrary unpredictable, random aspects (Schmidhuber, 1999). It constructs initially random algorithms mapping event sequences to abstract internal representations (IRs). It also constructs algorithms predicting IRs from IRs computed earlier. Its goal is to learn novel algorithms creating IRs useful for correct IR predictions, without wasting time on those learned before. This can be achieved by a co-evolutionary scheme involving two competing modules collectively designing single algorithms to be executed. The modules have actions for betting on the outcome of IR predictions computed by the algorithms they have agreed upon. If their opinions differ then the system checks who's right, punishes the loser (the surprised one), and rewards the winner. A DS-like RL algorithm forces each module to increase its reward. This motivates each to lure the other into agreeing upon algorithms involving predictions that surprise it. Since each module essentially can put in its veto against algorithms it does not consider profitable, the system is motivated to focus on those computable aspects of the environment where both modules still have confident but different opinions. Once both share the same opinion on a particular issue (via the loser's DS-based learning process, e.g., the winner is simply copied onto the loser), the winner loses a source of reward — an incentive to shift the focus of interest onto novel, yet unknown algorithms. There are simulations where surprise-generation of this kind can actually help to speed up external reward (Schmidhuber, 1999).

2.7 Summary

Given the potential DS advantages listed above (most of them related to partial observability), it may seem that the more ambitious the goals of some RL researcher, the more he/she will get drawn towards methods for DS in spaces of fairly general algorithms, as opposed to the more limited DPRL-based approaches.

Standard DS does suffer from major disadvantages, though, as I will point out later for the case of realistic, stochastic worlds.

3 Typical Applications

Both DPRL and DS boast impressive practical successes. For instance, there is the world-class DPRL-based backgammon player (Tesauro, 1994), although it's not quite clear yet (beyond mere intuition) why exactly it works so well. And DS has proven superior to alternative traditional methods in engineering domains such as wing design, combustion chamber design, turbulence control (the historical origins of "evolutionary computation"). There are overviews (Schwefel, 1995; Koumoutsakos P. & D., 1998) with numerous references to earlier work.

Will DS' successes in such domains eventually carry over to learning sequential behavior in domains traditionally approached by variants of DPRL? At the moment little work has been done on DS in search spaces whose elements are sequential behaviors (Schmidhuber et al., 1997b), but this may change soon. Of course, the most obvious temporal tasks to be attacked by DS are not board games but tasks that violate DPRL's Markovian assumptions. It does not make sense to apply low-bias methods like DS to domains that satisfy the preconditions of more appropriately biased DPRL approaches.

Parity. I will use the parity problem to illustrate this. The task requires to separate bitstrings of length $n > 0$ (n integer) with an odd number of zeros from others. n-bit parity in principle is solvable by a 3-layer feedforward neural net with n input units. But learning the task from training exemplars by, say, backpropagation, is hard for $n > 20$, due to such a net's numerous free parameters. On the other hand, a very simple finite state automaton with just one bit of internal state can correctly classify arbitrary bitstrings by sequentially processing them one bit at a time, and switching the internal state bit on or off depending on whether the current input is 1 or 0.

A policy implementing such a sequential solution, however, cannot efficiently be learned by DPRL. The problem is that the task violates DPRL's essential Markovian precondition: the current input bit in a training sequence does not provide the relevant information about the previous history necessary for correct classification.

Next we will see, however, that parity can be quickly learned by the most trivial DS method, namely, random search (RS). RS works as follows: *REPEAT randomly initialize the policy and test the resulting net on a training set UNTIL solution found.*

Experiment. Our policy is the weight matrix of a standard recurrent neural network. We use two architectures (A1, A2). A1(k) is a fully connected net with 1 input, 1 output, and k hidden units, each non-input unit receiving the traditional bias connection from a unit with constant activation 1.0. A2 is like A1(10), but less densely connected: each hidden unit receives connections from the input unit, the output unit, and itself; the output unit sees all other units. All activation functions are standard: $f(x) = (1 + e^{-x})^{-1} \in [0.0, 1.0]$. Binary inputs are -1.0

(for 0) and 1.0 (for 1). Sequence lengths n are randomly chosen between 500 and 600. All variable activations are set to 0 at each sequence begin. Target information is provided only at sequence ends (hence the relevant time delays comprise at least 500 steps; there are no intermediate rewards). Our training set consists of 100 sequences, 50 from class 1 (even; target 0.0) and 50 from class 2 (odd; target 1.0). Correct sequence classification is defined as "absolute error at sequence end below 0.1". We stop RS once a random weight matrix (weights randomly initialized in [-100.0,100.0]) correctly classifies all training sequences. Then we test on the test set (100 sequences).

In all simulations, RS eventually finds a solution that classifies all test set sequences correctly; average final absolute test set errors are always below 0.001 — in most cases below 0.0001. In particular, RS with A1 ($k = 1$) solves the problem within only 2906 trials (average of 10 trials). RS with A2 solves it within 2797 trials on average. RS for architecture A2, but without self-connections for hidden units, solves the problem within 250 trials on average. See a previous paper (Hochreiter & Schmidhuber, 1997) for additional results in this vein.

RS is a dumb DS algorithm, of course. It won't work within acceptable time except for the most trivial problems. But this is besides the point of this section, whose purpose is to demonstrate that even primitive DS may yield results beyond DPRL's abilities. Later, however, I will discuss smarter DS methods.

What about GP? Given the RS results above, how can it be that parity is considered a difficult problem by many authors publishing in the DS-based field of "Genetic Programming" (GP), as can be seen by browsing through the proceedings of recent GP conferences? The reason is: most existing GP systems are extremely limited because they search in spaces of programs that do not even allow for loops or recursion — to the best of my knowledge, the first exception was (Dickmanns et al., 1987). Hence most GP approaches ignore a major motivation for search in program space, namely, the repetitive reuse of code in solutions with low algorithmic complexity (Kolmogorov, 1965; Solomonoff, 1964; Chaitin, 1969; Li & Vitányi, 1993).

Of course, all we need to make parity easy is a search space of programs that process inputs sequentially and allow for internal memory and loops or conditional jumps. A few thousand trials will suffice to generalize perfectly to n-bit parity for *arbitrary* n, not just for special values like those used in the numerous GP papers on this topic (where typically $n << 30$).

4 DS Theory

The arguments above emphasize DS' ability to deal with general algorithm spaces as opposed to DPRL's limited spaces of reactive mappings. Which theoretical results apply to this case?

Non-incremental & Deterministic. We know that Levin Search (LS) (Levin, 1973, 1984; Li & Vitányi, 1993) is optimal in deterministic and *non*incremental settings, that is, in cases where during the search process there are no intermediate reinforcement signals indicating the quality of suboptimal solutions.

LS generates and tests computable solution candidates s in order of their Levin complexities

$$Kt(s) = \min_q \{-logD_P(q) + log\ t(q,s)\},$$

where program q computes s in $t(q,s)$ time steps, and $D_P(q)$ is q's Solomonoff-Levin probability (Levin, 1973, 1984; Li & Vitányi, 1993). Now suppose some algorithm A is designed to find the x solving $\phi(x) = y$, where ϕ is a computable function mapping bitstrings to bitstrings. For instance, x may represent a solution to a maze task implemented by $\phi(x) = 1$ iff x leads to the goal. Let n be a measure of problem size (such as the number of fields in the maze), and suppose A needs at most $O(f(n))$ steps (f computable) to solve problems of size n. Then LS also will need at most $O(f(n))$ steps (Levin, 1973, 1984; Li & Vitányi, 1993). Unlike GP etc., LS has a principled way of dealing with unknown program runtimes — time is allocated in an optimal fashion. There are systems that use a probabilistic LS variant to discover neural nets that perfectly generalize from extremely few training examples (Schmidhuber, 1995, 1997), and LS implementations that learn to use memory for solving maze tasks unsolvable by DPRL due to highly ambiguous inputs (Schmidhuber et al., 1997b).

Almost all work in traditional RL, however, focuses on *incremental* settings where continual policy improvement can bring more and more reward per time, and where experience with suboptimal solution candidates helps to find better ones. Adaptive Levin Search (ALS) (Schmidhuber et al., 1997b; Wiering & Schmidhuber, 1996) extends LS in this way: whenever a new candidate is more successful than the best previous one, the underlying probability distribution is modified to make the new candidate more likely, and a new LS cycle begins. This guarantees that the search time spent on each incremental step (given a particular probability distribution embodying the current bias) is optimal in Levin's sense.

It does not guarantee, though, that the total search time is spent optimally, because the probability adjustments themselves may have been suboptimal. ALS cannot exploit *arbitrary* regularities in solution space, because the probability modification algorithm itself is fixed. A machine learning researcher's dream would be an incremental RL algorithm that spends overall search time optimally in a way comparable to non-incremental LS's.

Incremental & Deterministic. Which theoretical results exist for incremental DS in general program spaces? Few, except for the following basic, almost trivial one. Suppose the environment allows for separating the search phase into repeatable, deterministic trials such that each trial with a given policy yields the same reward. Now consider stochastic hill-climbing (SHC), one of the simplest DS methods:

1. Initialize vector-valued policy p, set variables $BestPolicy := p$, $BestResult := -\infty$. **2.** Measure reward R obtained during a trial with actions executed according to p. **3.** If $BestResult > R$ then $p := BestPolicy$, else $BestPolicy := p$ and $BestResult := R$. **4.** Modify p by random mutation. Go to **2.**

If (1) both environment and p are deterministic, and if (2) the environment and all other variables modified by p (such as internal memory cells whose con-

tents may have changed due to actions executed according to p) are reset after each trial, then the procedure above will at least guarantee that performance cannot get worse over time. If step **4** allows for arbitrary random mutations and the number of possible policies is finite then we can even guarantee convergence on an optimal policy with probability 1, given infinite time. Of course, if the random mutations of step **4** are replaced by a systematic enumeration of all possible mutations, then we also will be able to guarantee finite (though exponential in problem size) time[1]. Similar statements can be made about alternative DS methods such as GP or evolutionary programming.

Of course, in real-world settings *no* general method can be expected to obtain optimal policies within reasonable time. Typical DS practitioners are usually content with suboptimal policies though. The next section, however, will address even more fundamental problems of DS.

5 DS: Problems with Unknown Delays and Stochasticity

Overview. As mentioned above, DS in policy space does not require certain assumptions about the environment necessary for traditional RL. For instance, while the latter is essentially limited to memory-free, reactive mappings from inputs to actions, DS can be used to search among fairly arbitrary, complex, event-memorizing programs (using memory to deal with partial observability), at least in simulated, deterministic worlds. In realistic settings, however, reliable policy evaluations are complicated by (a) unknown delays between action sequences and observable effects, and (b) stochasticity in policy and environment. Given a limited life-time, how much time should a direct policy searcher spend on policy evaluations to obtain reliable statistics? Despite the fundamental nature of this question it has not received much attention yet. Here I evaluate an efficient approach based on the *success-story algorithm* (SSA). It provides a radical answer prepared for the worst case: it *never* stops evaluating any previous policy modification except those it undoes for lack of empirical evidence that they have contributed to lifelong reward accelerations. I identify SSA's fundamental advantages over traditional DS on problems involving unknown delays and stochasticity.

The problem. In realistic situations DS exhibits several fundamental drawbacks: **(1)** Often there are unknown temporal delays between causes and effects. In general we cannot be sure that a given trial was long enough to observe all long-term rewards/punishments caused by actions executed during the trial. We do not know in advance how long a trial should take. **(2)** The policy may be stochastic, i.e., the learner's actions are selected nondeterministically according to probability distributions conditioned on the policy. Stochastic policies are widely used to prevent learners from getting stuck. Results of policy evaluations, however, will then *vary from trial to trial.* **(3)** Environment and reward may be

[1] Note that even for the more limited DPRL algorithms until very recently there have not been any theorems guaranteeing finite convergence time (Kearns & Singh, 1999).

stochastic, too. And even if the environment is deterministic it may appear stochastic from an individual learner's perspective, due to partial observability.

Time is a scarce resource. Hence all direct methods face a central question: to determine whether some policy is really useful in the long run or just appears to be (e.g., because the current trial was too short to encounter a punishing state, or because it was a lucky trial), how much time should the learner spend on its evaluation? In particular, how much time should a single trial with a given policy take, and how many trials are necessary to obtain statistically significant results without wasting too much time? Despite the fundamental nature of these questions not much work has been done to address them.

Basic idea. There is a radical answer to such questions: Evaluate a previous policy change at any stage of the search process by looking at the entire time interval that has gone by since the change occurred — at any given time aim to use *all* the available empirical data concerning long-term policy-dependent rewards! A change is considered "good" as long as the average reward per time since its creation exceeds the corresponding ratios for previous "good" changes. Changes that eventually turn out to be "bad" get undone by an efficient backtracking scheme called the *success-story algorithm* (SSA). SSA always takes into account the latest information available about long-term effects of changes that have appeared "good" so far ("bad" changes, however, are not considered again). Effectively SSA adjusts trial lengths retrospectively: at any given time, trial starts are determined by the occurrences of the remaining "good" changes representing a success story. The longer the time interval that went by since some "good" change, the more reliable the evaluation of its true long-term benefits. No trial of a "good" change ever ends unless it turns out to be "bad" at some point. Thus SSA is prepared for the worst case. For instance, no matter what's the maximal time lag between actions and consequences in a given domain, eventually SSA's effective trials will encompass it.

What's next. The next section will explain SSA in detail, and show how most traditional direct methods can be augmented by SSA. Then we will experimentally verify SSA's advantages over traditional direct methods in case of noisy performance evaluations and unknown delays.

6 Success-Story Algorithm (SSA)

Here we will briefly review basic principles (Schmidhuber et al., 1997a). An agent lives in environment E from time 0 to unknown time T. Life is one-way: even if it is decomposable into numerous consecutive "trials", time will never be reset. The agent has a policy POL (a set of modifiable parameters) and possibly an internal state S (e.g., for short-term memory that can be modified by actions). Both S and POL are variable dynamic data structures influencing probabilities of actions to be executed by the agent. Between time 0 and T, the agent repeats the following cycle over and over again ($\mathcal{A}$ denotes a set of possible actions):

REPEAT:
select and execute some $a \in \mathcal{A}$ **with conditional probability** $P(a \mid POL, S, E)$.[2]

Action a will consume time and may change E, S, and POL.

Learning algorithms (LAs). $LA \subset A$ is the set of actions that can modify POL. LA's members are called *learning algorithms* (LAs). Previous work on metalearning (Schmidhuber et al., 1997a, 1997b) left it to the system itself to compute execution probabilities of LAs by constructing and executing POL-modifying algorithms. In this paper we will focus on a simpler case where all LAs are externally triggered procedures. Formally this means that $P(a \in LA \mid POL, S, E) = 1$ if E is in a given "policy modification state" determined by the user, and $P(a \in LA \mid POL, S, E) = 0$ otherwise. For instance, in some of the illustrative experiments to be presented later, SHC's mutation process will be the only LA. Alternatively we may use LAs that are in fact GP-like crossover strategies, or a wide variety of other policy-modifying algorithms employed in traditional DS methods.

Checkpoints. The entire lifetime of the agent can be partitioned into time intervals separated by special times called *checkpoints*. In general, checkpoints are computed dynamically during the agent's life by certain actions in $\mathcal{A}$ executed according to POL itself (Schmidhuber et al., 1997a). In this paper, however, for simplicity all checkpoints will be set in advance. The agent's k-th checkpoint is denoted c_k. Checkpoints obey the following rules: (1) $\forall k \quad 0 < c_k < T$. (2) $\forall j < k \quad c_j < c_k$. (3) Except for the first, checkpoints may not occur before at least one POL-modification has been computed by some LA since the previous checkpoint.

Sequences of POL-modifications (SPMs). SPM_k denotes the sequence of POL-modifications computed by LAs in between c_k and c_{k+1}. Since LA execution probabilities may depend on POL, POL may in principle influence the way it modifies itself, which is interesting for things such as metalearning. In this paper's experiments, however, we will focus on the case where exactly one LA (a simple mutation process like the one used in stochastic hill-climbing) is invoked in between two subsequent checkpoints.

Reward and goal. Occasionally E provides real-valued reward. The cumulative reward obtained by the agent between time 0 and time $t > 0$ is denoted $R(t)$, where $R(0) = 0$. Typically, in large (partially observable) environments, maximizing cumulative expected reinforcement within a limited life-time would be too ambitious a goal for any method. Instead designers of direct policy search methods are content with methods that can be expected to find better and better policies. But what exactly does "better" mean in our context? Our agent's obvious goal at checkpoint t is to generate POL-modifications accelerating reward intake: it wants to let $\frac{R(T)-R(t)}{T-t}$ exceed the current average speed of reward intake. But to determine this speed it needs a previous point $t' < t$ to com-

[2] Instead of using the expression *policy* for the conditional probability distribution P itself we reserve it for the agent's modifiable data structure POL.

pute $\frac{R(t)-R(t')}{t-t'}$. How can t' be specified in a general yet reasonable way? Or, to rephrase the central question of this paper: if life involves unknown temporal delays between action sequences and their observable effects and/or consists of many successive "trials" with nondeterministic, uncertain outcomes, how long should a trial last, and how many trials should the agent look back into time to evaluate its current performance?

The *success-story algorithm* (to be introduced next) addresses this question in a way that is prepared for the worst case: once a trial of a new policy change has begun it will *never* end unless the policy change itself gets undone (by a simple backtracking mechanism which ensures that trial starts mark long-term reward accelerations).

Enforcing success stories. Let V denote the agent's time-varying set of past checkpoints that have been followed by long-term reward accelerations. Initially V is empty. v_k denotes the k-th element of V in ascending order. The *success-story criterion* (SSC) is satisfied at time t if either V is empty (trivial case) or if

$$\frac{R(t)-R(0)}{t-0} < \frac{R(t)-R(v_1)}{t-v_1} < \frac{R(t)-R(v_2)}{t-v_2} < \ldots < \frac{R(t)-R(v_{|V|})}{t-v_{|V|}}.$$

SSC demands that each checkpoint in V marks the beginning of a long-term reward acceleration measured up to the current time t. SSC is achieved by the *success-story algorithm* (SSA) which is invoked at every checkpoint:

1. WHILE SSC is not satisfied: Undo all POL modifications made since the most recent checkpoint in V, and remove that checkpoint from V.
2. Add the current checkpoint to V.

"Undoing" a modification means restoring the preceding POL — this requires storing past values of POL components on a stack prior to modification. Thus each POL modification that survived SSA is part of a bias shift generated after a checkpoint marking a lifelong reward speed-up: the remaining checkpoints in V and the remaining policy modifications represent a "success story."

Trials and "effective" trials. All checkpoints in V represent starts of yet unfinished "effective" trials (as opposed to externally defined trials with prewired starts and ends). No effective trial ever ends unless SSA restores the policy to its state before the trial started. The older some surviving SPM, the more time will have passed to collect statistics concerning its long-term consequences, and the more stable it will be if it is indeed useful and not just there by chance.

Since trials of still valid policy modifications never end, they embody a principled way of dealing with unknown reward delays. No matter what's the maximal causal delay in a given environment, eventually the system's effective trials will encompass it.

Metalearning? An interesting example of long delays between actions and effects is provided by metalearning (learning a learning algorithm or credit assignment algorithm) (Lenat, 1983; Schmidhuber, 1987; Schmidhuber et al.,

1997a). For instance, suppose some "metalearning action sequence" changes a learner's credit assignment strategy. To evaluate whether the change is good or bad, apparently we need something like a "meta-trial" subsuming several lower-level "normal" trials in which instances of additional policy changes produced by the modified learning strategy somehow get evaluated, to collect evidence concerning the quality of the modified learning strategy itself. Due to their very nature such meta-trials will typically consume a lot of time. SSA, however, addresses this issue in a natural and straight-forward way. There is no explicit difference between trials and meta-trials. But new effective trials do get opened within previously started, yet unfinished effective trials. What does this mean? It means that earlier SPMs automatically get evaluated as to whether they set the stage for later "good" SPMs. For instance, SSA will eventually discard an early SPM that changed the policy in a way that increased the probability of certain later SPMs causing a waste of time on evaluations of useless additional policy changes. That is, SSA automatically measures (in terms of reward/time ratios affected by learning and testing processes) the impact of early learning on later learning: SSA prefers SPMs making "good" future SPMs more likely. Given action sets that allow for composition of general credit assignment strategies from simple LAs, SSA will prefer probabilistic learning algorithms leading to better probabilistic learning algorithms. And it will end meta-trials as soon as they violate the constraints imposed by the success-story criterion, just like it does with "normal" trials.

Implementing SSA. SSA guarantees that SSC will be satisfied after each checkpoint, even in partially observable, stochastic environments with unknown delays. (Of course, later SSA invocations using additional, previously unavailable, delayed information may undo policy changes that survived the current checkpoint.) Although inequality (1) contains $|V|$ fractions, SSA can be implemented efficiently: only the two most recent still valid sequences of modifications (those "on top of the stack") need to be considered at a given time in an SSA invocation. A *single* SSA invocation, however, may invalidate *many* SPMs if necessary.

7 Illustration: How SSA Augments DS

Methods for direct search in policy space, such as stochastic hill-climbing (SHC) and genetic algorithms (GAs), test policy candidates during time-limited trials, then build new policy candidates from some of the policies with highest evaluations observed so far. As mentioned above, the advantage of this general approach over traditional RL algorithms is that few restrictions need to be imposed on the nature of the agent's interaction with the environment. In particular, if the policy allows for actions that manipulate the content of some sort of short-term memory then the environment does not need to be fully observable — in principle, direct methods such as Genetic Programming (Cramer, 1985; Banzhaf et al., 1998), adaptive Levin Search (Schmidhuber et al., 1997b), or Probabilistic Incremental Program Evolution (Sałustowicz & Schmidhuber,

1997), can be used for searching spaces of complex, event-memorizing programs or algorithms as opposed to simple, memory-free, reactive mappings from inputs to actions.

As pointed out above, a disadvantage of traditional direct methods is that they lack a principled way of dealing with unknown delays and stochastic policy evaluations. In contrast to typical trials executed by direct methods, however, an SSA trial of any previous policy modification *never* ends unless its reward/time ratio drops below that of the most recent previously started (still unfinished) effective trial. Here we will go beyond previous work (Schmidhuber et al., 1997a, 1997b) by clearly demonstrating how a direct method can benefit from augmentation by SSA in presence of unknown temporal delays and stochastic performance evaluations.

Task (Schmidhuber & Zhao, 1999). We tried to come up with the simplest task sufficient to illustrate the drawbacks of standard DS algorithms and the way SSA overcomes them. Hence, instead of studying tasks that require to learn complex programs setting and resetting memory contents (as mentioned above, such complex tasks provide a main motivation for using DS), we use a comparatively simple two-armed bandit problem.

There are two arms A and B. Pulling arm A will yield reward 1000 with probability 0.01 and reward -1 with probability 0.99. Pulling arm B will yield reward 1 with probability 0.99 and reward -1000 with probability 0.01. *All rewards are delayed by 5 pulls.* There is an agent that knows neither the reward distributions nor the delays. Since this section's goal is to study policy search under uncertainty we equip the agent with the simplest possible stochastic policy consisting of a single variable p ($0 \leq p \leq 1$): at a given time arm A is chosen with probability p, otherwise B is chosen. *Modifying the policy in a very limited, SHC-like way (see below) and observing the long-term effects is the only way the agent may collect information that might be useful for improving the policy.* Its goal is to maximize the entire reward obtained during its life-time which is limited to 30,000 pulls. The maximal cumulative reward is 270300 (always choose arm A), the minimum is -270300 (always choose arm B). Random arm selection yields expected reward 0. [3]

Obviously the task is non-trivial, because the long-term effects of a small change in p will be hard to detect, and will require significant statistical sampling.

It is besides the point of this paper that our prior knowledge of the problem suggests a more informed alternative approach such as "pull arm A for N trials, then arm B for N trials, then commit to the best." Even cleverer optimizers would try various assumptions about the delay lengths and pull arms in turn until

[3] The problem resembles another two-armed bandit problem for which there is an optimal method due to Gittins (Gittins, 1989). Our unknown reward delays, however, prevent this method from being applicable — it cannot discover that the current reward does not depend on the current input but on an event in the past. In addition, Gittins' method needs to discount future rewards relative to immediate rewards. Anyway, this footnote is besides the point of this paper whose focus is on direct policy search — Gittins' method is not a direct one.

one was statistically significantly better than the other, given a particular delay assumption. We do not allow this, however: we make the task hard by requiring the agent to learn solely from observations of outcomes of limited, SHC-like policy mutations (details below). After all, in partially observable environments that are much more complex and realistic (but less analyzable) than ours this often is the only reasonable thing to do.

Stochastic Hill-Climbing (SHC). SHC may be the simplest incremental algorithm using direct search in policy space. It should be mentioned, however, that despite its simplicity SHC often outperforms more complex direct methods such as GAs (Juels & Wattenberg, 1996). Anyway, SHC and more complex population-based direct algorithms such as GAs, GP, and evolution strategies are equally affected by the central questions of this paper: how many trials should be spent on the evaluation of a given policy? How long should a trial take?[4]

We implement SHC as follows: **1.** Initialize policy p to 0.5, and real-valued variables $BestPolicy$ and $BestResult$ to p and 0, respectively. **2.** If there have been more than $30000 - TrialLength$ pulls then exit ($TrialLength$ is an integer constant). Otherwise evaluate p by measuring the average reward R obtained during the next $TrialLength$ consecutive pulls. **3.** If $BestResult > R$ then $p := BestPolicy$, else $BestPolicy := p$ and $BestResult := R$. **4.** Randomly perturb p by adding either -0.1 or +0.1 except when this would lead outside the interval [0,1]. Go to **2.**

Problem. Like any direct search algorithm SHC faces the fundamental question raised in section 2: how long to evaluate the current policy to obtain statistically significant results without wasting too much time? To examine this issue we vary $TrialLength$. Our prior knowledge of the problem tells us that $TrialLength$ should exceed 5 to handle the 5-step reward delays. But due to the stochasticity of the rewards, much larger $TrialLength$s are required for reliable evaluations of some policy's "true" performance. Of course, the disadvantage of long trials is the resulting small number of possible training exemplars and policy changes (learning steps) to be executed during the limited life which lasts just 30,000 steps.

Comparison 1. We compare SHC to a combination of SSA and SHC, which we implement just like SHC except that we replace step **3** by a checkpoint (SSA-invocation — see section 6).

Comparison 2. To illustrate potential benefits of policies that influence the way they learn we also compare to SSA applied to a primitive "self-modifying policy" (SMP) with *two* modifiable parameters: p (with same meaning as above) and "learning rate" δ (initially 0). After each checkpoint, p is replaced by $p + \delta$, then δ is replaced by $2 * \delta$. In this sense SMP itself influences the way it changes, albeit in a way that is much more limited than the one in previous papers (Schmidhuber et al., 1997a; Schmidhuber, 1999). If $\delta = 0$ then it will be randomly

[4] In fact, population-based approaches will suffer even more than simple SHC from unknown delays and stochasticity, simply because they need to test many policies, not just one.

set to either 0.1 or -0.1. If $\delta > 0.5$ ($\delta < -0.5$) then it will be replaced by 0.5 (-0.5). If $p > 1$ ($p < 0$) then it will be set to 1 (0).

One apparent danger with this approach is that accelerating the learning rate may result in unstable behavior. We will see, however, that SSA precisely prevents this from happening by eventually undoing those learning rate modifications that are not followed by reliable long-term performance improvement.

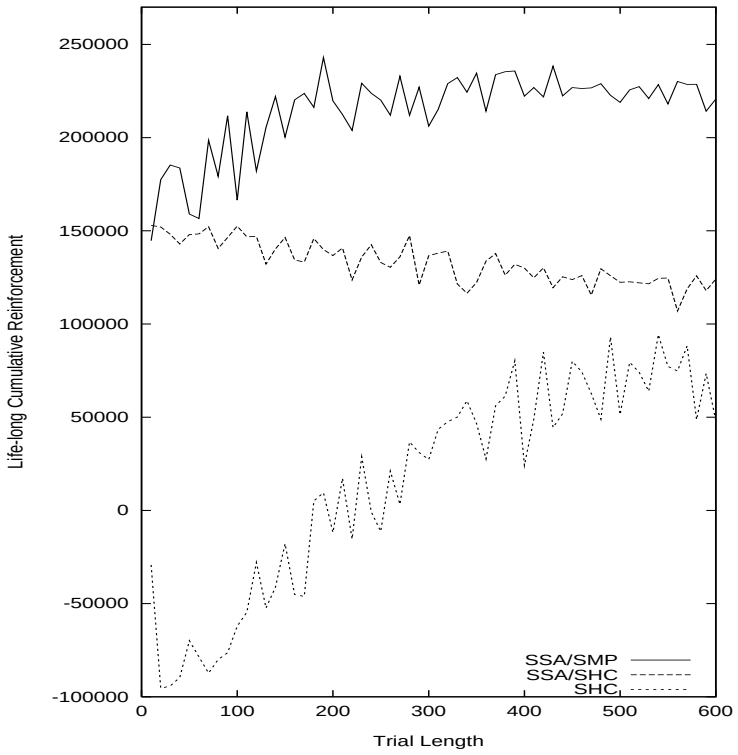

Fig. 1. *Total cumulative reinforcement (averaged over 100 trials) obtained by SHC (bottom), SSA/SHC (middle), SSA/SMP (top) for varying trial lengths. The picture does not change much for even longer trial lengths.*

Results. For all three methods Figure 1 plots lifelong cumulative reward (mean of 100 independent runs) against $TrialLength$ varying from 10 to 600 pulls with a step size of 10 pulls. For most values of $TrialLength$, SHC fails to realize the long-term benefits of choosing arm A. SSA/SHC, however, always yields satisfactory results because it does not care whether $TrialLength$ is much too short to obtain statistically significant policy evaluations. Instead it retrospectively readjusts the "effective" trial starts: at any given checkpoint, each previous checkpoint in V marks the begin of a new trial lasting up to the current checkpoint. Each such trial start corresponds to a lifelong reward-acceleration.

The corresponding policy modifications gain more and more empirical justification as they keep surviving successive SSA calls, thus becoming more and more stable.

Still, SSA/SHC's performance slowly declines with increasing $TrialLength$ since this implies less possible policy changes and less effective trials due to limited life-time. SSA/SMP (comparison **2**), however, does not much suffer from this problem since it boldly increases the learning rate as long as this is empirically observed to accelerate long-term reward intake. As soon as this is not the case any longer, however, SSA prevents further learning rate accelerations, thus avoiding unstable behavior. This primitive type of learning algorithm self-modification outperforms SSA/SHC. In fact, some of the surviving effective trials may be viewed as "metalearning trials": SSA essentially observes the long term effects of certain learning rates whose values are influenced by the policy itself, and undoes those that tend to cause "bad" additional policy modifications setting the stage for worse performance in subsequent trials.

Trials Shorter Than Delays. We also tested the particularly interesting case $TrialLength < 5$. Here SHC and other direct methods fail completely because the policy tested during the current trial has nothing to do with the test outcome (due to the delays). The SSA/SHC combination, however, still manages to collect cumulative performance of around 150,000. Unlike with SHC (and other direct methods) there is no need for *a priori* knowledge about "good" trial lengths, exactly because SSA retrospectively adjusts the *effective* trial sizes.

Comparable results were obtained with much longer delays. In particular, see a recent article (Schmidhuber, 1999) for experiments with much longer life-times and unknown delays of the order of $10^6 - 10^7$ time steps.

A Complex Partially Observable Environment. This section's focus is on clarifying SSA's advantages over traditional DS in the simplest possible setting. It should be mentioned, however, that there have been much more challenging SSA applications in partially observable environments, which represent a major motivation of direct methods because most traditional RL methods are not applicable here. For instance, a previous paper (Schmidhuber et al., 1997a) describes two agents A and B living in a partially observable 600×500 pixel environment with obstacles. They learn to solve a complex task that could not be solved by various TD(λ) Q-learning variants (Lin, 1993). The task requires (1) agent A to find and take a key "key A"; (2) agent A go to a door "door A" and open it for agent B; (3) agent B to enter through "door A", find and take another key "key B"; (4) agent B to go to another door "door B" to open it (to free the way to the goal); (5) one of the agents to reach the goal. Both agents share the same design. Each is equipped with limited "active" sight: by executing certain instructions, it can sense obstacles, its own key, the corresponding door, or the goal, within up to 50 pixels in front of it. The agent can also move forward, turn around, turn relative to its key or its door or the goal. It can use short-term memory to disambiguate inputs — unlike Jaakkola et al.'s method (1995), ours is not limited to finding suboptimal stochastic policies for POEs with an optimal solution. Each agent can explicitly modify its own policy via special

actions that can address and modify the probability distributions according to which action sequences (or "subprograms") are selected (this also contributes to making the set-up highly non-Markovian). Reward is provided only if one of the agents touches the goal. This agent's reward is 5.0; the other's is 3.0 (for its cooperation — note that asymmetric reward also introduces competition). Due to the complexity of the task, in the beginning the goal is found only every 300,000 actions on average (including actions that are primitive LAs and modify the policy). No prior information about good initial trial lengths is given to the system. Through self-modifications and SSA, however, within 130,000 goal hits (10^9 actions) the average trial length decreases by a factor of 60 (mean of 4 simulations). Both agents learn to cooperate to accelerate reward intake, by retrospectively adjusting their effective trial lengths using SSA.

While this previous experimental research has already demonstrated SSA's applicability to large-scale partially observable environments, a study of why it performs well has been lacking. In particular, unlike the present work, previous work (Schmidhuber et al., 1997a) did not clearly identify SSA's fundamental advantages over alternative DS methods.

8 Relation to Market Models

There is an interesting alternative class of RL systems that also combines aspects of DS and traditional RL. It exploits ideas from the field of economy that seem naturally applicable in the context of multiagent RL, and (unlike Q-learning etc.) are not necessarily limited to learning reactive behavior. In what follows I will ignore the extensive original financial literature and briefly review solely some work on RL economies instead. Then I will relate this work to SSA.

Classifier Systems and Bucket Brigade. The first RL economy was Holland's meanwhile well-known bucket brigade algorithm for classifier systems (Holland, 1985). Messages in form of bitstrings of size n can be placed on a global message list either by the environment or by entities called classifiers. Each classifier consists of a condition part and an action part defining a message it might send to the message list. Both parts are strings out of $\{0, 1, \_\}^n$ where the '_' serves as a 'don't care' if it appears in the condition part. A non-negative real number is associated with each classifier indicating its 'strength'. During one cycle all messages on the message list are compared with the condition parts of all classifiers of the system. Each matching classifier computes a 'bid' based on its strength. The highest bidding classifiers may place their message on the message list of the next cycle, but they have to pay with their bid which is distributed among the classifiers active during the last time step which set up the triggering conditions (this explains the name bucket brigade). Certain messages result in an action within the environment (like moving a robot one step). Because some of these actions may be regarded as 'useful' by an external critic who can give payoff by increasing the strengths of the currently active classifiers, learning may take place. The central idea is that classifiers which are not active when the environment gives payoff but which had an important role for setting the stage for

directly rewarded classifiers can earn credit by participating in 'bucket brigade chains'. The success of some active classifier recursively depends on the success of classifiers that are active at the following time ticks. Unsuccessful classifiers are replaced by new ones generated with the help of GAs.

Holland's original scheme is similar to DPRL algorithms such as Q-learning in the sense that the bids of the agents correspond to predictions of future reward. On the other hand, the scheme does not necessarily require full environmental observability. Instead it partly depends on DS-like evolutionary pressure absent in traditional RL. E.g., bankrupt agents who spent all their money are removed from the system.

Holland's approach, however, leaves several loopholes that allow agents to make money without contributing to the success of the entire system. This led to a lot of follow-up research on more stable RL classifier economies (Wilson, 1994, 1995; Weiss, 1994; Weiss & Sen, 1996) and other related types of RL ecomomies (see below). This work has closed some but possibly not all of the original loopholes.

Prototypical Self-referential Associating Learning Mechanisms. Pages 23-51 of earlier work (Schmidhuber, 1987) are devoted to systems called PSALM1 - PSALM3. Like in Holland's scheme, competing/cooperating RL agents bid for executing actions. Winners may receive external reward for achieving goals. Unlike in Holland's scheme, agents are supposed to learn the credit assignment process itself (metalearning). For this purpose they can execute actions for collectively constructing / connecting / modifying agents, for assigning credit (reward) to agents, and for transferring credit from one agent to another. To the best of my knowledge, PSALMs are the first machine learning systems that enforce the important constraint of total credit conservation (except for consumption and external reward): no agent can generate money from nothing. This constraint is not enforced in the original bucket brigade economy, where new agents enter with freshly created money (this may cause inflation and other problems). Reference (Schmidhuber, 1987) also inspired the *neural* bucket brigade (NBB), a slightly more recent but less general approach enforcing money conservation, where money is "weight substance" of a reinforcement learning neural net (Schmidhuber, 1989).

Hayek Machine (Baum & Durdanovic, 1998). PSALM3 does not strictly enforce individual property rights. For instance, agents may steal money from other agents and temporally use it in a way that does not contribute to the system's overall progress. Hayek machines are constitutional economies that apparently do not suffer from such "parasite problems" (although there is no proof yet at the moment). Hayek2 (Baum & Durdanovic, 1998) — the most recent Hayek variant — is somewhat reminiscent of PSALM3 (agents may construct new agents, and there is money conservation), but avoids several loopholes in the credit assignment process that may allow some agent to profit from actions that are not beneficial for the system as a whole. Property rights of agents are strictly enforced. Agents can create children agents and invest part of their money into them, and profit from their success. Hayek2 learned to solve rather complex

blocks world problems (Baum & Durdanovic, 1998). The authors admit, however, that possibly not all potential loopholes rewarding undesired behavior have been closed by Hayek2.

COINs/Wonderful Life Utility (Wolpert, Tumer, & Frank, 1999). Perhaps the only current RL economy with a sound theoretical foundation is the COllective Intelligence (COIN) by Wolpert, Tumer & Frank (1999). A COIN partitions its set of agents into "subworlds." Each agent of a given subworld shares the same local utility function. Global utility is optimized by provably making sure that no agent in some subworld can profit if the system as a whole does not profit. COINs were successfully used in an impressive routing application.

SSA and Market Models. One way of viewing SSA in an economy-inspired framework is this: the current "credit" of a policy change equals the reward since its creation divided by the time since its creation. A policy change gets undone as soon as its credit falls below the credit of the most recent change that has not been undone yet. After any given SSA invocation the yet undone changes reflect a success-story of long-term credit increases.

To use the parent/child analogy (Baum & Durdanovic, 1998): at a given time, any still valid policy change may be viewed as a (grand-)parent of later policy changes for which it set the stage. Children that are more profitable than their ancestors protect the latter from being undone. In this way the ancestors profit from making successful children. Ancestors who increase the probability of non-profitable offspring, however, will eventually risk oblivion.

9 Conclusion

Direct search (DS) in policy space offers several advantages over traditional reinforcement learning (RL). For instance, DS does not need *a priori* information about world states and the topology of their interactions. It does not care whether the environment is fully observable. It makes hierarchical credit assignment conceptually trivial, and also allows for many alternative, non-hierarchical types of abstract credit assignment.

Existing DS methods, however, do suffer from fundamental problems in presence of environmental stochasticity and/or unknown temporal delays between actions and observable effects. In particular, they do not have a principled way of deciding when to stop policy evaluations.

Stochastic policy evaluation by the success-story algorithm (SSA) differs from traditional DS. SSA never quits evaluating any previous policy change that has not yet been undone for lack of empirical evidence that it has contributed to a lifelong reward acceleration. Each invocation of SSA retrospectively establishes a success history of surviving self-modifications: only policy changes that have empirically proven their long-term usefulness so far get another chance to justify themselves. This stabilizes the "truly useful" policy changes in the long run.

Unlike many traditional value function-based RL methods, SSA is not limited to fully observable worlds, and does not require discounting of future rewards. It shares these advantages with traditional DS algorithms. Unlike stochastic

hill-climbing and other DS methods such as genetic algorithms, however, SSA does not heavily depend on *a priori* knowledge about reasonable trial lengths necessary to collect sufficient statistics for estimating long-term consequences and true values of tested policies.

On the other hand, many DS methods can be augmented by SSA in a straightforward way: just measure the time used up by all actions, policy modifications, and policy tests, and occasionally insert checkpoints that invoke SSA. In this sense SSA's basic concepts are not algorithm-specific — instead they reflect a novel, general way of thinking about how "true" performance should be measured in RL systems using DS in policy space.

Since SSA automatically collects statistics about long-term effects of earlier policy changes on later ones, it is of interest for improving the credit assignment method itself (Schmidhuber et al., 1997a).

Although the present paper's illustrative SSA application is much less complex than our previous ones, it is the first to provide insight into SSA's fundamental advantages over traditional DS methods in case of stochastic policy evaluations and unknown temporal delays between causes and effects.

Market models are similar to traditional DPRL in the sense that the bids of their agents correspond to predictions of future reward used in DPRL algorithms such as Q-learning. On the other hand, they are similar to DS in the sense that they incorporate evolutionary pressure and do not obviously require full environmental observability and Markovian conditions but can be applied to agents whose policies are drawn from general program spaces. For example, bankrupt agents who spent all their money are usually replaced by mutations of more successful agents. This introduces Darwinian selection absent in traditional DPRL.

SSA shares certain aspects of market models. In particular, at a given time any existing policy modification's reward/time ratio measured by SSA may be viewed as an investment into the future. The policy modification will fail to survive once it fails to generate enough return (including the reward obtained by its own "children") to exceed the corresponding investment of its "parent", namely, the most recent *previous*, still existing policy modification.

Future research will hopefully show when to prefer pure market models over SSA and vice versa. For instance, it will be interesting to study whether the former can efficiently deal with long, unknown causal delays and highly stochastic policies and environments.

10 Acknowledgments

Thanks to Sepp Hochreiter, Jieyu Zhao, Nic Schraudolph, Fred Cummins, Luca Gambardella for valuable comments. This work was supported in part by SNF grant 21-43'417.95 "Incremental Self-Improvement" and SNF grant 2100-49'144.96 "Long Short-Term Memory".

Bibliography

Andre, D. (1998). Learning hierarchical behaviors. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning.*

Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1998). *Genetic Programming – An Introduction.* Morgan Kaufmann Publishers, San Francisco, CA, USA.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-13*, 834–846.

Baum, E. B., & Durdanovic, I. (1998). Toward code evolution by artificial economies. Tech. rep., NEC Research Institute, Princeton, NJ. Extension of a paper in Proc. 13th ICML'1996, Morgan Kaufmann, CA.

Bellman, R. (1961). *Adaptive Control Processes.* Princeton University Press.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic Programming.* Athena Scientific, Belmont, MA.

Bowling, M., & Veloso, M. (1998). Bounding the suboptimality of reusing subproblems. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning.*

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

Coelho, J., & Grupen, R. A. (1998). Control abstractions as state representation. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning.*

Cohn, D. A. (1994). Neural network exploration using optimal experiment design. In Cowan, J., Tesauro, G., & Alspector, J. (Eds.), *Advances in Neural Information Processing Systems 6*, pp. 679–686. San Mateo, CA: Morgan Kaufmann.

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

Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. In Lippman, D. S., Moody, J. E., & Touretzky, D. S. (Eds.), *Advances in Neural Information Processing Systems 5*, pp. 271–278. San Mateo, CA: Morgan Kaufmann.

Dayan, P., & Sejnowski, T. J. (1996). Exloration bonuses and dual control. *Machine Learning, 25*, 5–22.

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

Digney, B. (1996). Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. In Maes,

P., Mataric, M., Meyer, J.-A., Pollack, J., & Wilson, S. W. (Eds.), *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pp. 363–372. MIT Press, Bradford Books.

Eldracher, M., & Baginski, B. (1993). Neural subgoal generation using backpropagation. In Lendaris, G. G., Grossberg, S., & Kosko, B. (Eds.), *World Congress on Neural Networks*, pp. III–145–III–148. Lawrence Erlbaum Associates, Inc., Publishers, Hillsdale.

Fedorov, V. V. (1972). *Theory of optimal experiments*. Academic Press.

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

Harada, D., & Russell, S. (1998). Meta-level reinforcement learning. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Hochreiter, S., & Schmidhuber, J. (1997). LSTM can solve hard long time lag problems. In Mozer, M. C., Jordan, M. I., & Petsche, T. (Eds.), *Advances in Neural Information Processing Systems 9*, pp. 473–479. MIT Press, Cambridge MA.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Holland, J. H. (1985). Properties of the bucket brigade. In *Proceedings of an International Conference on Genetic Algorithms*. Hillsdale, NJ.

Huber, M., & Grupen, R. A. (1998). Learning robot control using control policies as abstract actions. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Humphrys, M. (1996). Action selection methods using reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., & Wilson, S. W. (Eds.), *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pp. 135–144. MIT Press, Bradford Books.

Hwang, J., Choi, J., Oh, S., & II, R. J. M. (1991). Query-based learning applied to partially trained multilayer perceptrons. *IEEE Transactions on Neural Networks*, *2*(1), 131–136.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems 7*, pp. 345–352. MIT Press, Cambridge MA.

Juels, A., & Wattenberg, M. (1996). Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 430–436. The MIT Press, Cambridge, MA.

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

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

Kearns, M., & Singh, S. (1999). Finite-sample convergence rates for Q-learning and indirect algorithms. In Kearns, M., Solla, S. A., & Cohn, D. (Eds.),

Advances in Neural Information Processing Systems 12. MIT Press, Cambridge MA.

Kirchner, F. (1998). Q-learning of complex behaviors on a six-legged walking machine. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Koenig, S., & Simmons, R. G. (1996). The effect of representation and knowedge on goal-directed exploration with reinforcement learnign algorithm. *Machine Learning*, *22*, 228–250.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, *1*, 1–11.

Koumoutsakos P., F. J., & D., P. (1998). Evolution strategies for parameter optimization in jet flow control. *Center for Turbulence Research – Proceedings of the Summer program 1998*, *10*, 121–132.

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

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

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

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

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

Littman, M. (1996). *Algorithms for Sequential Decision Making*. Ph.D. thesis, Brown University.

Littman, M., Cassandra, A., & Kaelbling, L. (1995). Learning policies for partially observable environments: Scaling up. In Prieditis, A., & Russell, S. (Eds.), *Machine Learning: Proceedings of the Twelfth International Conference*, pp. 362–370. Morgan Kaufmann Publishers, San Francisco, CA.

MacKay, D. J. C. (1992). Information-based objective functions for active data selection. *Neural Computation*, *4*(2), 550–604.

McCallum, R. A. (1996). Learning to use selective attention and short-term memory in sequential tasks. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., & Wilson, S. W. (Eds.), *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cambridge, MA*, pp. 315–324. MIT Press, Bradford Books.

McGovern, A. (1998). acquire-macros: An algorithm for automatically learning macro-action. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Moore, A., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Moore, A. W., Baird, L., & Kaelbling, L. P. (1998). Multi-value-functions: Efficient automatic action hierarchies for multiple goal mdps. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

Plutowski, M., Cottrell, G., & White, H. (1994). Learning Mackey-Glass from 25 examples, plus or minus 2. In Cowan, J., Tesauro, G., & Alspector, J.

(Eds.), *Advances in Neural Information Processing Systems 6*, pp. 1135–1142. San Mateo, CA: Morgan Kaufmann.

Ray, T. S. (1992). An approach to the synthesis of life. In Langton, C., Taylor, C., Farmer, J. D., & Rasmussen, S. (Eds.), *Artificial Life II*, pp. 371–408. Addison Wesley Publishing Company.

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

Ring, M. B. (1991). Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In Birnbaum, L., & Collins, G. (Eds.), *Machine Learning: Proceedings of the Eighth International Workshop*, pp. 343–347. Morgan Kaufmann.

Ring, M. B. (1993). Learning sequential tasks by incrementally adding higher orders. In S. J. Hanson, J. D. C., & Giles, C. L. (Eds.), *Advances in Neural Information Processing Systems 5*, pp. 115–122. Morgan Kaufmann.

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

Sałustowicz, R. P., & Schmidhuber, J. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, *5*(2), 123–141.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, *3*, 210–229.

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

Schmidhuber, J. (1989). A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, *1*(4), 403–412.

Schmidhuber, J. (1991a). Curious model-building control systems. In *Proc. International Joint Conference on Neural Networks, Singapore*, Vol. 2, pp. 1458–1463. IEEE.

Schmidhuber, J. (1991b). Learning to generate sub-goals for action sequences. In Kohonen, T., Mäkisara, K., Simula, O., & Kangas, J. (Eds.), *Artificial Neural Networks*, pp. 967–972. Elsevier Science Publishers B.V., North-Holland.

Schmidhuber, J. (1991c). Reinforcement learning in Markovian and non-Markovian environments. In Lippman, D. S., Moody, J. E., & Touretzky, D. S. (Eds.), *Advances in Neural Information Processing Systems 3*, pp. 500–506. San Mateo, CA: Morgan Kaufmann.

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

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

Schmidhuber, J. (1999). Artificial curiosity based on discovering novel algorithmic predictability through coevolution. In Angeline, P., Michalewicz, Z., Schoenauer, M., Yao, X., & Zalzala, Z. (Eds.), *Congress on Evolutionary Computation*, pp. 1612–1618. IEEE Press, Piscataway, NJ.

Schmidhuber, J., & Prelinger, D. (1993). Discovering predictable classifications. *Neural Computation*, *5*(4), 625–635.

Schmidhuber, J., & Zhao, J. (1999). Direct policy search and uncertain policy evaluation. In *AAAI Spring Symposium on Search under Uncertain and Incomplete Information, Stanford Univ.*, pp. 119–124. American Association for Artificial Intelligence, Menlo Park, Calif.

Schmidhuber, J., Zhao, J., & Schraudolph, N. (1997a). Reinforcement learning with self-modifying policies. In Thrun, S., & Pratt, L. (Eds.), *Learning to learn*, pp. 293–309. Kluwer.

Schmidhuber, J., Zhao, J., & Wiering, M. (1997b). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, *28*, 105–130.

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

Schwefel, H. P. (1995). *Evolution and Optimum Seeking*. Wiley Interscience.

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

Singh, S. (1992). The efficient learning of multiple task sequences. In Moody, J., Hanson, S., & Lippman, R. (Eds.), *Advances in Neural Information Processing Systems 4*, pp. 251–258 San Mateo, CA. Morgan Kaufmann.

Solomonoff, R. (1964). A formal theory of inductive inference. Part I. *Information and Control*, *7*, 1–22.

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

Storck, J., Hochreiter, S., & Schmidhuber, J. (1995). Reinforcement driven information acquisition in non-deterministic environments. In *Proceedings of the International Conference on Artificial Neural Networks, Paris*, Vol. 2, pp. 159–164. EC2 & Cie, Paris.

Sun, R., & Sessions, C. (2000). Self-segmentation of sequences: automatic formation of hierarchies of sequential behaviors. *IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics*, *30*(3).

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

Sutton, R. S. (1995). TD models: Modeling the world at a mixture of time scales. In Prieditis, A., & Russell, S. (Eds.), *Machine Learning: Proceedings of the Twelfth International Conference*, pp. 531–539. Morgan Kaufmann Publishers, San Francisco, CA.

Sutton, R. S., & Pinette, B. (1985). The learning of world models by connectionist networks. *Proceedings of the 7th Annual Conference of the Cognitive Science Society*, 54–64.

Sutton, R. S., Singh, S., Precup, D., & Ravindran, B. (1999). Improved switching among temporally abstract actions. In *Advances in Neural Information Processing Systems 11*. MIT Press. To appear.

Teller, A. (1994). The evolution of mental models. In Kenneth E. Kinnear, J. (Ed.), *Advances in Genetic Programming*, pp. 199–219. MIT Press.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*(2), 215–219.

Tham, C. (1995). Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems*, *15*(4), 247–274.

Thrun, S., & Möller, K. (1992). Active exploration in dynamic environments. In Lippman, D. S., Moody, J. E., & Touretzky, D. S. (Eds.), *Advances in Neural Information Processing Systems 4*, pp. 531–538. San Mateo, CA: Morgan Kaufmann.

Wang, G., & Mahadevan, S. (1998). A greedy divide-and-conquer approach to optimizing large manufacturing systems using reinforcement learning. In *NIPS'98 Workshop on Abstraction and Hierarchy in Reinforcement Learning*.

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

Watkins, C. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Oxford.

Weiss, G. (1994). Hierarchical chunking in classifier systems. In *Proceedings of the 12th National Conference on Artificial Intelligence*, Vol. 2, pp. 1335–1340. AAAI Press/The MIT Press.

Weiss, G., & Sen, S. (Eds.). (1996). *Adaption and Learning in Multi-Agent Systems*. LNAI 1042, Springer.

Wiering, M., & Schmidhuber, J. (1998). HQ-learning. *Adaptive Behavior*, *6*(2), 219–246.

Wiering, M., & Schmidhuber, J. (1996). Solving POMDPs with Levin search and EIRA. In Saitta, L. (Ed.), *Machine Learning: Proceedings of the Thirteenth International Conference*, pp. 534–542. Morgan Kaufmann Publishers, San Francisco, CA.

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

Wilson, S. (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation*, *2*, 1–18.

Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, *3(2)*, 149–175.

Wolpert, D. H., Tumer, K., & Frank, J. (1999). Using collective intelligence to route internet traffic. In Kearns, M., Solla, S. A., & Cohn, D. (Eds.), *Advances in Neural Information Processing Systems 12*. MIT Press, Cambridge MA.