Completely Self-referential Optimal Reinforcement Learners

Jürgen Schmidhuber

IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany
juergen@idsia.ch
http://www.idsia.ch/~juergen

Abstract. We present the first class of mathematically rigorous, general, fully self-referential, self-improving, optimal reinforcement learning systems. Such a system rewrites any part of its own code as soon as it has found a proof that the rewrite is *useful,* where the problem-dependent *utility function* and the hardware and the entire initial code are described by axioms encoded in an initial proof searcher which is also part of the initial code. The searcher systematically and efficiently tests computable *proof techniques* (programs whose outputs are proofs) until it finds a provably useful, computable self-rewrite. We show that such a self-rewrite is globally optimal—no local maxima!—since the code first had to prove that it is not useful to continue the proof search for alternative self-rewrites. Unlike previous *non*-self-referential methods based on hardwired proof searchers, ours not only boasts an optimal *order* of complexity but can optimally reduce any slowdowns hidden by the $O()$-notation, provided the utility of such speed-ups is provable at all.

1 Introduction and Outline

Traditional reinforcement learning (RL) algorithms [6] are hardwired. They are designed to improve some limited type of policy through experience, but are not part of the modifiable policy, and cannot improve themselves. Humans are needed to create new / better RL algorithms and to prove their usefulness under appropriate assumptions.

Let us eliminate the restrictive need for human effort in the most general way possible, leaving all the work including the proof search to a system that can rewrite and improve itself in arbitrary computable ways and in a most efficient fashion. To attack this "Grand Problem of Artificial Intelligence," we introduce a novel class of optimal, fully self-referential [3] general problem solvers called *Gödel machines* [11,10]. They are universal RL systems that interact with some (partially observable) environment and can in principle modify themselves without essential limits besides the limits of computability. Their initial RL algorithm is not hardwired; it can completely rewrite itself, but only if a proof searcher embedded within the initial algorithm can first prove that the rewrite is useful, given a formalized utility function reflecting expected rewards and computation time. We will see that self-rewrites due to this approach are actually *globally optimal* (Theorem 1, Section 4), relative to Gödel's well-known fundamental restrictions of provability [3]. These restrictions should not worry us; if there is no proof of some self-rewrite's utility, then humans cannot do much either.

The initial proof searcher is $O()$-optimal (has an optimal order of complexity) in the sense of Theorem 2, Section 5. Unlike Hutter's hardwired systems [5] (Section 2), however, a Gödel machine can further speed up its proof searcher to meet *arbitrary* formalizable notions of optimality beyond those expressible in the $O()$-notation. Our approach yields the first theoretically sound, fully self-referential, optimal, general reinforcement learners.

Outline. Section 2 presents basic concepts, relation to previous work, and limitations, Section 3 the essential details of a self-referential axiomatic system, Section 4 the Global Optimality Theorem 1, and Section 5 the $O()$-optimal (Theorem 2) initial proof searcher.

2 Basic Overview and Relation to Previous Work and Limitations

Notation and Set-Up. Unless stated otherwise or obvious, throughout the paper newly introduced variables and functions are assumed to cover the range implicit in the context. B denotes the binary alphabet $\{0,1\}$, B^* the set of possible bitstrings over B, $l(q)$ denotes the number of bits in a bitstring q; q_n the n-th bit of q; λ the empty string (where $l(\lambda)=0$); $q_{m:n}=\lambda$ if $m>n$ and $q_m q_{m+1}\ldots q_n$ otherwise (where $q_0 := q_{0:0} := \lambda$).

Our hardware (e.g., a universal or space-bounded Turing machine or the abstract model of a personal computer) has a single life which consists of discrete cycles or time steps $t=1,2,\ldots$. Its total lifetime T may or may not be known in advance. In what follows, the value of any time-varying variable Q at time t will be denoted by $Q(t)$.

During each cycle our hardware executes an elementary operation which affects its variable state $s \in \mathcal{S} \subset \mathcal{B}^*$ and possibly also the variable environmental state $Env \in \mathcal{E}$. (Here we need not yet specify the problem-dependent set $\mathcal{E}$). There is a hardwired state transition function $F : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{S}$. For $t>1$, $s(t) = F(s(t-1), Env(t-1))$ is the state at a point where the hardware operation of cycle $t-1$ is finished, but the one of t has not started yet. $Env(t)$ may depend on past output actions encoded in $s(t-1)$ and is simultaneously updated or (probabilistically) computed by the possibly reactive environment.

At any given time t ($1 \leq t \leq T$) the goal is to maximize future success or *utility*. A typical *"value to go"* utility function (to be maximized) is of the form $u(s, Env) : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers:

$$u(s, Env) = E_\mu \left[\sum_{\tau=time}^{T} r(\tau) \;\middle|\; s, Env \right], \tag{1}$$

where $r(t)$ is a real-valued reward input (encoded within $s(t)$) at time t, $E_\mu(\cdot \mid \cdot)$ denotes the conditional expectation operator with respect to some possibly unknown distribution μ from a set M of possible distributions (M reflects whatever is known about the possibly probabilistic reactions of the environment), and the above-mentioned $time = time(s)$ is a function of state s which uniquely identifies the current cycle. Note that we take into account the possibility of extending the expected lifespan through appropriate actions.

Basic Idea. Our machine becomes a self-referential [3] *Gödel machine* by loading it with a machine-dependent, particular form of self-modifying code p. The initial code $p(1)$ at time step 1 includes a (typically sub-optimal) problem solving subroutine for interacting with the environment, such as Q-learning [6], and a general proof searcher subroutine (Section 5) that systematically makes pairs *(switchprog, proof)* until it finds a *proof* of a target theorem which essentially states: *'the immediate rewrite of* p *through current program* switchprog *on the given machine implies higher utility than leaving* p *as is'.* Then it executes *switchprog*, which may completely rewrite p, including the proof searcher. Section 3 will explain details of the necessary initial axiomatic system $\mathcal{A}$ encoded in $p(1)$.

The **Global Optimality Theorem** (Theorem 1, Section 4) shows this self-improvement strategy is not greedy: since the utility of *'leaving p as is'* implicitly evaluates all possible alternative *switchprog*s which an unmodified p might find later, we obtain a globally optimal self-change—the *current switchprog* represents the best of all possible relevant self-changes, relative to the given resource limitations and initial proof search strategy.

Proof Techniques and an $O()$-optimal Initial Proof Searcher. Section 5 will present an $O()$-optimal initialization of the proof searcher, that is, one with an optimal *order* of complexity (Theorem 2). Still, there will remain a lot of room for self-improvement hidden by the $O()$-notation. The searcher uses an online extension of *Universal Search* [7] to systematically test *online proof techniques*, which are proof-generating programs that may read parts of state s (similarly, mathematicians are often more interested in proof techniques than in theorems). To prove target theorems as above, proof techniques may invoke special instructions for generating axioms and applying inference rules to prolong the current *proof* by theorems. Here an axiomatic system $\mathcal{A}$ encoded in $p(1)$ includes axioms describing **(a)** how any instruction invoked by a program running on the given hardware will change the machine's state s (including instruction pointers etc.) from one step to the next (such that proof techniques can reason about the effects of any program including the proof searcher), **(b)** the initial program $p(1)$ itself (Section 3 will show that this is possible without introducing circularity), **(c)** stochastic environmental properties, **(d)** the formal utility function u, e.g., equation (1). The evaluation of utility automatically takes into account computational costs of all actions including proof search.

Hutter's Previous Work. Hutter's non-self-referential but still $O()$-optimal *'fastest' algorithm for all well-defined problems* HSEARCH [4] uses a *hardwired* brute force proof searcher. Assume discrete input/output domains X/Y, a formal problem specification $f : X \rightarrow Y$ (say, a functional description of how integers are decomposed into their prime factors), and a particular $x \in X$ (say, an integer to be factorized). HSEARCH orders all proofs of an appropriate axiomatic system by size to find programs q that for all $z \in X$ provably compute $f(z)$ within time bound $t_q(z)$. Simultaneously it spends most of its time on executing the q with the best currently proven time bound $t_q(x)$. It turns out that HSEARCH is as fast as the *fastest* algorithm that provably computes $f(z)$ for all $z \in X$, save for a constant factor smaller than $1 + \epsilon$ (arbitrary $\epsilon > 0$) and an f-specific but x-independent additive constant [4]. This constant may be enormous though.

Hutter's AIXI*(t,l)* [5] is related. In discrete cycle $k = 1, 2, 3, \ldots$ of AIXI*(t,l)*'s lifetime, action $y(k)$ results in perception $x(k)$ and reward $r(k)$, where all quantities may depend on the complete history. Using a universal computer such as a Turing machine, AIXI*(t,l)* needs an initial offline setup phase (prior to interaction with the environment) to examine all proofs of length at most L, filtering out those that identify programs (of maximal size l and maximal runtime t per cycle) which not only could interact with the environment but which for all possible interaction histories also correctly predict a lower bound of their own expected future reward. In cycle k, AIXI*(t,l)* then runs all programs identified in the setup phase (at most 2^l), finds the one with highest self-rating, and executes its corresponding action. The problem-independent setup time (where almost all of the work is done) is $O(L \cdot 2^L)$. The online time per cycle is $O(t \cdot 2^l)$. Both are constant but typically huge.

Advantages and Novelty of the Gödel Machine. There are major differences between the Gödel machine and Hutter's HSEARCH [4] and AIXI*(t,l)* [5], including:

1. The theorem provers of HSEARCH and AIXI*(t,l)* are hardwired, non-self-referential, unmodifiable meta-algorithms that cannot improve themselves. That is, they will always suffer from the same huge constant slowdowns (typically $\gg 10^{1000}$) buried in the $O()$-notation. But there is nothing in principle that prevents our truly self-referential code from proving and exploiting drastic reductions of such constants, in the best possible way that provably constitutes an improvement, if there is any.
2. The demonstration of the $O()$-optimality of HSEARCH and AIXI*(t,l)* depends on a clever allocation of computation time to some of their unmodifiable meta- algorithms. Our Global Optimality Theorem (Theorem 1, Section 4), however, is justified through a quite different type of reasoning which indeed exploits and crucially depends on the fact that there is no unmodifiable software at all, and that the proof searcher itself is readable and modifiable and can be improved. This is also the reason why its self-improvements can be more than merely $O()$-optimal.
3. HSEARCH uses a "trick" of proving more than is necessary which also disappears in the sometimes quite misleading $O()$-notation: it wastes time on finding programs that provably compute $f(z)$ for all $z \in X$ even when the current $f(x)(x \in X)$ is the only object of interest. A Gödel machine, however, needs to prove only what is relevant to its goal formalized by u. For example, the general u of eq. (1) completely ignores the limited concept

of $O()$-optimality, but instead formalizes a stronger type of optimality that does not ignore huge constants just because they are constant.
4. Both the Gödel machine and AIXI*(t,l)* can maximize expected reward (HSEARCH cannot). But the Gödel machine is more flexible as we may plug in *any* type of formalizable utility function (e.g., *worst case* reward), and unlike AIXI*(t,l)* it does not require an enumerable environmental distribution.

Limitations. The fundamental limitations are closely related to those first identified by Gödel's celebrated paper on self-referential formulae [3]. Any formal system that encompasses arithmetics (or ZFC etc) is either flawed or allows for unprovable but true statements. Hence even a Gödel machine with unlimited computational resources must ignore those self-improvements whose effectiveness it cannot prove, e.g., for lack of sufficiently powerful axioms in $\mathcal{A}$. In particular, one can construct pathological examples of environments and utility functions that make it impossible for the machine to ever prove a target theorem. Compare Blum's speed-up theorem [1] based on certain incomputable predicates. Similarly, a realistic Gödel machine with limited resources cannot profit from self-improvements whose usefulness it cannot prove within its time and space constraints. Nevertheless, unlike previous methods, it can in principle exploit at least the *provably* good speed-ups of *any* part of its initial software, including those parts responsible for huge (but problem class-independent) slowdowns ignored by the earlier approaches [5].

3 Essential Details of One Representative Gödel Machine

Theorem proving requires an axiom scheme yielding an enumerable set of axioms of a formal logic system $\mathcal{A}$ whose formulas and theorems are symbol strings over some finite alphabet that may include traditional symbols of logic (such as $\rightarrow, \wedge, =, (,), \forall, \exists, \ldots, c_1, c_2, \ldots, f_1, f_2, \ldots$), probability theory (such as $E(\cdot)$, the expectation operator), arithmetics ($+, -, /, =, \sum, <, \ldots$), string manipulation (in particular, symbols for representing any part of state s at any time, such as $s_{7:88}(5555)$). A proof is a sequence of theorems, each either an axiom or inferred from previous theorems by applying one of the inference rules such as *modus ponens* combined with *unification*, e.g., [2].

The remainder of this paper will omit standard knowledge to be found in any proof theory textbook. Instead of listing *all* axioms of a particular $\mathcal{A}$, we will focus on the novel and critical details: how to overcome problems with self-reference and how to deal with the potentially delicate online generation of proofs that talk about and affect the currently running proof generator itself.

Proof Techniques. Brute force proof searchers (used in Hutter's AIXI*(t,l)* and HSEARCH) systematically generate all proofs in order of their sizes. To produce a certain proof, this takes time exponential in proof size. Instead our $O()$-optimal $p(1)$ will produce many proofs with low algorithmic complexity [7] much more quickly. It systematically tests (see Section 5) *proof techniques* written in universal language $\mathcal{L}$ implemented within $p(1)$. A proof technique is composed of instructions that allow any part of s to be read, such as inputs x, or the code of $p(1)$. It may write on s^p, a part of s reserved for temporary results. It also may rewrite *switchprog*, and produce an incrementally growing proof placed in the string variable *proof* stored somewhere in s. *proof* and s^p are reset to the empty string at the beginning of each new proof technique test. Apart from standard arithmetic and function-defining instructions [9] that modify s^p, the programming language $\mathcal{L}$ includes special instructions for prolonging the current *proof* by correct theorems, for setting *switchprog*, and for checking whether a provably optimal p-modifying program was found and should be executed now. Certain long proofs can be produced by short proof techniques.

The nature of the five *proof*-modifying instructions below (there are no others) makes it impossible to insert an incorrect theorem into *proof*, thus trivializing proof verification:

1. **get-axiom(n)** takes as argument an integer n computed by a prefix of the currently tested proof technique with the help of arithmetic instructions such as those used in previous work [9]. Then it appends the n-th axiom (if it exists, according to the axiom scheme below) as a theorem to the current theorem sequence in *proof*. The initial axiom scheme encodes:
   (a) **Hardware axioms** describing the hardware, formally specifying how certain components of s (other than environmental inputs) may change from one cycle to the next. For example, the following axiom could describe how some 64-bit hardware's instruction pointer stored in $s_{1:64}$ is continually increased by 64 as long as there is no overflow and the value of s_{65} does not indicate that a jump to some other address should take place:

$$(\forall t \forall n : [(n < 2^{64} - 1) \wedge (n > 0) \wedge (t > 1) \wedge (t < T) \wedge (string2num(s_{1:64}(t)) = n)$$
$$\wedge (s_{65}(t) = \text{`0'})] \rightarrow (string2num(s_{1:64}(t+1)) = n + 1))$$

   Here the semantics of used symbols such as '(' and '>' and '→' (implies) are the traditional ones, while '*string2num*' symbolizes a function translating bitstrings into numbers. It is clear that any abstract hardware model can be fully axiomatized in a similar way.
   (b) **Reward axioms** defining the computational costs of any hardware instruction, and physical costs of output actions (e.g., control signals encoded in $s(t)$). Related axioms assign values to certain input events (encoded in s) representing reward or punishment (e.g., when a Gödel machine-controlled robot bumps into an obstacle). Additional axioms define the total value of the Gödel machine's life as a scalar-valued function of all rewards (e.g., their sum) and costs experienced between cycles 1 and T, etc.
   (c) **Environment axioms** restricting the way the environment will produce new inputs (encoded within certain substrings of s) in reaction to sequences of outputs encoded in s. For example, it may be known

in advance that the environment is sampled from an unknown probability distribution that is *computable,* given the previous history [12,5]. Or, more restrictively, the environment may be some unknown but deterministic computer program sampled from the Speed Prior [8] which assigns low probability to environments that are hard to compute by any method. Or the interface to the environment is Markovian, that is, the current input always uniquely identifies the environmental state [6]. Even more restrictively, the environment may evolve in completely predictable fashion known in advance. All such prior assumptions are perfectly formalizable in an appropriate $\mathcal{A}$ (otherwise we could not write scientific papers about them).

(d) **Uncertainty axioms; string manipulation axioms:** Standard axioms for arithmetics and calculus and probability theory and statistics and string manipulation that (in conjunction with the environment axioms) allow for constructing proofs concerning (possibly uncertain) properties of future values of $s(t)$ as well as bounds on expected remaining lifetime / costs / rewards, given some time τ and certain hypothetical values for components of $s(\tau)$ etc.

(e) **Initial state axioms:** Information about how to reconstruct the initial state $s(1)$ or parts thereof, such that the proof searcher can build proofs including axioms of the type

$$(s_{\mathbf{m:n}}(1) = \mathbf{z}),\ e.g.:\ (s_{7:9}(1) = \text{‘010’}).$$

Here and in the remainder of the paper we use bold font in formulas to indicate syntactic place holders (such as **m,n,z**) for symbol strings representing variables (such as *m,n,z*) whose semantics are explained in the text (in the present context z is the bitstring $s_{m:n}(1)$).

Note that it is **no fundamental problem** to fully encode both the hardware description *and* the initial hardware-describing p within p itself. To see this, observe that some software may include a program that can print the software.

(f) **Utility axioms** describing the overall goal in the form of utility function u; e.g., equation (1).

2. **apply-rule(k, m, n)** takes as arguments the index k (if it exists) of an inference rule such as *modus ponens* (stored in a list of possible inference rules encoded within $p(1)$) and the indices m, n of two previously proven theorems (numbered in order of their creation) in the current *proof.* If applicable, the corresponding inference rule is applied to the addressed theorems and the resulting theorem appended to *proof.* Otherwise the currently tested proof technique is interrupted. This ensures that *proof* is never fed with invalid proofs.
3. **set-switchprog(m,n)** replaces *switchprog* by $s^p_{m:n}$, provided that $s^p_{m:n}$ is indeed a non-empty substring of s^p, the storage writable by proof techniques.
4. ♠ **state2theorem(m, n)** takes two integer arguments m, n and tries to transform the current contents of $s_{m:n}$ into a theorem of the form

$$(s_{\mathbf{m:n}}(\mathbf{t_1}) = \mathbf{z}),\ e.g.:\ (s_{6:9}(7775555) = \text{‘1001’}),$$

where t_1 represents a time measured (by checking *time*) shortly after *state2theorem* was invoked, and z the bistring $s_{m:n}(t_1)$ (recall the special case $t_1 = 1$ of Item 1e). So we accept the time-labeled current observable contents of any part of s as a theorem that does not have to be proven in an alternative way from, say, the initial state $s(1)$, because the computation so far has already demonstrated that the theorem is true.

We must avoid inconsistent results through parts of s that change while being read. For example, the present value of a quickly changing instruction pointer IP *(continually updated by the hardware) may not be directly observable without changing the observable itself: the execution of the reading subroutine itself will already modify* IP *many times. So* state2theorem *first uses some fixed protocol to check whether the current $s_{m:n}$ is readable at all or whether it might change if it were read by the remaining code of* state2theorem. *If so, or if m, n, are not in the proper range, then the instruction has no further effect.*

5. ♠ **check()** verifies whether the goal of the proof search has been reached. First it tests whether the last theorem (if any) in *proof* has the form of a **target theorem**. A target theorem states that given the *current* axiomatized utility function u (Item 1f), the utility of a switch from p to the current *switchprog* would be higher than the utility of continuing the execution of p (which would keep searching for alternative *switchprog*s). Target theorems are symbol strings (encoded in s) of the (decoded) form

$$(u[s(\mathbf{t_1}) \oplus (\mathit{switchbit}(\mathbf{t_1}) = \text{`1'}), \mathit{Env}(\mathbf{t_1})] >$$
$$u[s(\mathbf{t_1}) \oplus (\mathit{switchbit}(\mathbf{t_1}) = \text{`0'}), \mathit{Env}(\mathbf{t_1})]) \qquad (2)$$

where the variable t_1 (represented by syntactic place holder $\mathbf{t_1}$) stands for a time step, while all other symbols belong to the alphabet of the theorem-proving calculus, that is, the set of possible target theorems is parameterized only by t_1. Here the calculus should permit the notation $s(\mathbf{t_1}) \oplus (\mathit{switchbit}(\mathbf{t_1}) = \text{`}b\text{'})$ as a shortcut for the state obtained when we replace *switchbit*(t_1), the true value of the variable bit *switchbit* (encoded in s) at time t_1, by $b \in \{0, 1\}$. This will facilitate the formulation of theorems that compare values conditioned on various alternative hypothetical properties of $s(t_1)$. (Note that $s(t_1)$ may be only partially known by the current proof technique even in environments where $s(t_1)$ and *switchbit*(t_1) are fully predetermined for all valid t_1.)

The purpose of introducing t_1 is to deal with hardware-specific temporal delays that may be involved in checking and switching—it may take a significant amount of time to match abstract symbol strings found during proof search to the Gödel machine's real current state. If a target theorem has been found, *check()* uses a simple prewired subroutine to check whether there is enough time left to set variable *switchbit* (originally 0) to 1 before the continually increasing *time* will equal t_1. If this subroutine returns a negative result, *check()* exits. Otherwise it sets *switchbit* $:= 1$ (there is no other way of changing *switchbit*). Then it repeatedly tests *time* until *time* $> t_1$, to make sure the condition of formula (2) was fulfilled at t_1. Then it transfers control to *switchprog* (there is no other way of calling *switchprog*). The *switchprog* may subsequently rewrite all parts of s, excluding hardware-reserved parts such as *time* and x, but including p.

Note that a proof technique does not necessarily have to compute the true expected utilities of switching and not switching—it just needs to determine which is higher. For example, it may be easy to prove that speeding up a subroutine of the proof searcher by a factor of 2 will certainly be worth the negligible (compared to lifetime T) time needed to execute the subroutine-changing algorithm, no matter what is the precise utility of the switch.

The axiomatic system $\mathcal{A}$ is a defining parameter of a given Gödel machine. Clearly, $\mathcal{A}$ must be strong enough to permit proofs of target theorems. In particular, the theory of uncertainty axioms (Item 1d) must be sufficiently rich. This is no fundamental problem: we simply insert all traditional axioms of probability theory.

4 Global Optimality Theorem

Intuitively, at any given time p should execute some self-modification algorithm only if it is the 'best' of all possible self-modifications, given the utility function, which typically depends on available resources, such as storage size and remaining lifetime. At first glance, however, target theorem (2) seems to implicitly talk about just one single modification algorithm, namely, *switchprog*(t_1) as set by the systematic proof searcher at time t_1. Isn't this type of local search greedy? Couldn't it lead to a local optimum instead of a global one? No, it cannot, according to the global optimality theorem:

Theorem 1 (Globally Optimal Self-Changes, given u and $\mathcal{A}$ encoded in p). *Given any formalizable utility function u (Item 1f), and assuming consistency of the underlying formal system $\mathcal{A}$, any self-change of p obtained through execution of some program* switchprog *identified through the proof of a target theorem (2) is globally optimal in the following sense: the utility of starting the execution of the present* switchprog *is higher than the utility of waiting for the proof searcher to produce an alternative* switchprog *later.*

Proof. Target theorem (2) implicitly talks about all the other *switchprog*s that the proof searcher could produce in the future. To see this, consider the two alternatives of the binary decision: (1) either execute the current *switchprog* (set *switchbit* = 1), or (2) keep searching for *proof*s and *switchprog*s (set *switchbit* = 0) until the systematic searcher comes up with an even better *switchprog.* Obviously the second alternative concerns all (possibly infinitely many) potential *switchprog*s to be considered later. That is, if the current *switchprog* were not the 'best', then the proof searcher would not be able to prove that setting *switchbit* and executing *switchprog* will cause higher expected reward than discarding *switchprog*, assuming consistency of $\mathcal{A}$. *Q.E.D.*

5 Bias-Optimal Proof Search (BIOPS)

Here we construct a $p(1)$ that is $O()$-optimal in a certain limited sense to be described below, but still might be improved as it is not necessarily optimal in the sense of the given u (for example, the u of equation (1) neither mentions nor cares for $O()$-optimality). Our Bias-Optimal Proof Search (BIOPS) is essentially an application of Universal Search [7] to proof search. Practical extensions [9] of Universal Search have been applied to *offline* program search tasks where the program inputs are fixed such that the same program always produces the same results. In our *online* setting, however, BIOPS has to take into account that the same proof technique started at different times may yield different proofs, as it may read parts of s (e.g., inputs) that change as the machine's life proceeds.

BIOPS starts with a probability distribution P (the initial bias) on the proof techniques w that one can write in $\mathcal{L}$, e.g., $P(w) = K^{-l(w)}$ for programs composed from K possible instructions [7]. BIOPS is *near-bias-optimal* [9] in the sense that it will not spend much more time on any proof technique than it deserves, according to its probabilistic bias, namely, not much more than its probability times the total search time:

Method 51 (BIOPS). *In phase* $(i = 1, 2, 3, \dots)$ DO*:*

```
FOR all self-delimiting [7] proof techniques $w \in \mathcal{L}$ satisfying $P(w) \geq 2^{-i}$
DO:
    1. Run $w$ until halt or error (such as division by zero) or $2^i P(w)$ steps
       consumed.
    2. Undo the effects of $w$ on $s^p$ (does not cost significantly more time
       than executing $w$).
```

A proof technique w can interrupt Method 51 only by invoking instruction *check()* (Item 5), which may transfer control to *switchprog* (which possibly even will delete or rewrite Method 51). Since the initial p runs on the formalized hardware, and since proof techniques tested by p can read p and other parts of s, they can produce proofs concerning the (expected) performance of p and BIOPS itself. Method 51 at least has the optimal *order* of computational complexity in the following sense.

Theorem 2. *If independently of variable* time(s) *some unknown fast proof technique w would require at most $f(k)$ steps to produce a proof of difficulty measure k (an integer depending on the nature of the task to be solved), then Method 51 will need at most $O(f(k))$ steps.*

Proof. It is easy to see that Method 51 will need at most $O(f(k)/P(w)) = O(f(k))$ steps—the constant factor $1/P(w)$ does not depend on k. *Q.E.D.*

Note again, however, that the proofs themselves may concern quite different, arbitrary formalizable notions of optimality (stronger than those expressible in the $O()$-notation) embodied by the given, problem-specific, formalized utility function u. This may provoke useful, constant-affecting rewrites of the initial proof searcher despite its limited (yet popular and widely used) notion of $O()$-optimality.

6 Conclusion

The initial software $p(1)$ of our machine runs an initial problem solver (e.g., one of Hutter's approaches [5] which have at least an optimal *order* of complexity). Simultaneously, it runs an $O()$-optimal initial proof searcher using an online variant of Universal Search to test *proof techniques*, which are programs able to compute proofs concerning the system's own future performance, based on an axiomatic system $\mathcal{A}$ encoded in $p(1)$, describing a formal *utility* function u, the hardware and $p(1)$ itself. If there is no provably good, globally optimal way of rewriting $p(1)$ at all, then humans will not find one either. But if there is one, then $p(1)$ itself can find and exploit it. This approach yields the first class of theoretically sound, fully self-referential, optimal, general RL machines.

After the theoretical analysis above, one practical question remains: to build a particular, especially practical Gödel machine with small initial constant overhead, which generally useful theorems should one add as axioms to $\mathcal{A}$ (as initial bias) such that the initial searcher does not have to prove them from scratch?

References

1. M. Blum. On effective procedures for speeding up algorithms. *Journal of the ACM*, 18(2):290–305, 1971.
2. M. C. Fitting. *First-Order Logic and Automated Theorem Proving.* Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1996.
3. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
4. M. Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002. (On J. Schmidhuber's SNF grant 20-61847).
5. M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2004. (On J. Schmidhuber's SNF grant 20-61847).
6. L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.
7. L. A. Levin. Randomness conservation inequalities: Information and independence in mathematical theories. *Information and Control*, 61:15–37, 1984.
8. J. Schmidhuber. The Speed Prior: a new simplicity measure yielding near-optimal computable predictions. In J. Kivinen and R. H. Sloan, editors, *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002)*, Lecture Notes in Artificial Intelligence, pages 216–228. Springer, Sydney, Australia, 2002.
9. J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.
10. J. Schmidhuber. Gödel machines: fully self-referential optimal universal problem solvers. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence.* Springer Verlag, in press, 2005.
11. J. Schmidhuber. Gödel machines: Towards a technical justification of consciousness. In D. Kudenko, D. Kazakov, and E. Alonso, editors, *Adaptive Agents and Multi-Agent Systems III (LNCS 3394)*, pages 1–23. Springer Verlag, 2005.
12. R. J. Solomonoff. Complexity-based induction systems. *IEEE Transactions on Information Theory*, IT-24(5):422–432, 1978.