Exploring the Predictable

In S. Ghosh, S. Tsutsui, eds., Advances in Evolutionary Computing, p. 579-612,

Springer, 2002.

Jürgen Schmidhuber IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland
juergen@idsia.ch - http://www.idsia.ch/~juergen

February 17, 2003

Abstract

Details of complex event sequences are often not predictable, but their reduced abstract representations are. I study an embedded active learner that can limit its predictions to almost arbitrary computable aspects of spatio-temporal events. It constructs probabilistic algorithms that (1) control interaction with the world, (2) map event sequences to abstract internal representations (IRs), (3) predict IRs from IRs computed earlier. Its goal is to create novel algorithms generating IRs useful for correct IR predictions, without wasting time on those learned before. This requires an adaptive novelty measure which is implemented by a co-evolutionary scheme involving two competing modules *collectively* designing (initially random) algorithms representing experiments. Using special instructions, the modules can bet on the outcome of IR predictions computed by 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. An evolutionary or reinforcement learning algorithm forces each module to maximize reward. This motivates both modules to lure each other into agreeing upon experiments involving predictions that surprise it. Since each module essentially can veto experiments 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 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 experiments. My simulations include an example where surprise-generation of this kind helps to speed up external reward.

We can learn only what we already almost know.

PATRICK WINSTON

Zwei Seelen wohnen, ach, in meiner Brust!
(Two souls, alas! are dwelling in my breast!)
JOHANN WOLFGANG GOETHE

1 Introduction

How does one explore the world? By creating novel, surprising situations and learning from them until they are understood and boring. But what exactly should constitute a surprise?

Intuitively, surprise or novelty implies unexpectedness. For instance, many would be surprised by the view of a flying elephant. Unexpectedness by itself, however, does not imply surprise. For instance, few would be surprised by the unpredictable and therefore unexpected details of the elephant's skin texture.

One difference between the flying elephant and its skin texture is that the former violates a confident prediction ("this elephant won't fly") while the latter does not. Obviously surprise has to be measured with respect to some given predictor. Only a predictor explicitly expressing an expectation that may be wrong can leave room for surprise and novelty: no surprise without the commitment of a prediction.

Now consider that in general it is not particular inputs that are surprising, but sequences of inputs. For instance, many would say an airborne elephant that jumped off the roof of a building is less surprising than one lifting off from the street and disappearing in the clouds. In fact, the predictable and the surprising things are usually spatio-temporal abstractions of the input sequences. For example, all possible visual input sequences caused by physically plausible trajectories of falling elephants (modulated by eye and head movements and varying lighting conditions) may map onto the same abstract internal representation "falling elephant," as opposed to the more surprising "flying elephant."

Let's have a closer look now at how the concept of a surprise is implemented in some existing, "curious," "inquisitive" machine learning systems designed to explore a given environment, and how this differs from what we intuitively might want.

Most previous work on exploring unknown data sets has focused on selecting single training exemplars maximizing traditional information gain [5, 10, 16, 18, 4]. Here typically the concept of a surprise is defined in Shannon's sense [38]: some event's surprise value or information content is the negative logarithm of its probability. This inspired simple reinforcement learning approaches to pure exploration [24, 23, 40] 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 the case of predictor failures. Since it tries to maximize reward, 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.

Most existing systems are limited to picking out simple statistical regularities such as "performing action A in discrete, fully observable environmental state B will lead to state C with probability 0.8." Essentially, they always predict all the details of single inputs (or the next state among a set of predefined states), and are not able to limit their predictions solely to certain computable aspects of inputs (as requested in [33]) 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 elephant, will result in the appearance of a big red stain on the street 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 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 big red stains — all may be mapped onto the same compact internal representation predictable from all sequences corresponding to the abstraction "falling elephant." Only if the final input sequence caused by eye movements scanning the street does not map onto the concept "big red stain" (because the elephant somehow decelerated in time and for some strange reason never touched the ground) will there be a surprise.

The central questions are: In a given environment, how does one extract the predictable concepts corresponding to algorithmic regularities that are not already known? Which novel input sequence-transforming algorithms do indeed compute 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 [13]. So how to discover novel spatio-temporal regularities automatically among the many random or unpredictable things that should be ignored?

To study these questions, I will use a rather general algorithmic setup. Consider an agent exposed to a lifelong sequence of complex (for example, visual) inputs from a real world-like environment that it can manipulate. The agent is able to compose algorithms written in a Turing-equivalent programming language (one that permits construction of a universal Turing machine). The instruction set includes instructions that can access the current input (for example, pixels of visual scenes), modify the environment via actions, and modify an internal memory consisting of many addressable memory cells. It also includes arithmetic instructions and conditional jumps (conditioned on current memory cell contents).

An important role is played by so-called *"Bet!"* instructions that can be used as parts of algorithms that make predictions about memory cell contents.

For instance, suppose there already exists a complex algorithm that writes value 7 into memory cell 22 whenever there has occurred one of the many visual input sequences caused by typical falling elephants. Suppose there exists another piece of code that writes value 7 into memory cell 44 shortly after a big red stain in the street is identified by an appropriate algorithm. Finally, suppose shortly after cell 44 is filled another algorithm looks at cell 22 and a history of recent eye movements represented in other memory cells and claims that cell 44's content is 7, *without looking at it.* The claim is simply implemented via a *Bet!* instruction that addresses the two memory cells and compares their contents. It corresponds to a prediction expressing abstract knowledge about the world. The prediction may be wrong. This will constitute a surprise — a violation of an explicit (confident) prediction limited to a very particular, abstract transformation of a complex input sequence.

Now, in the absence of an external programmer or teacher, how could such algorithms focusing on the essential, predictable aspects of the environment be learned? How can we motivate the system to focus on such *novel* algorithms without wasting time on previously learned algorithms whose effects are already known? How can we prevent it from focusing on trivial transformations mapping all input sequences on the same internal representation which is always predictable? Clearly, we somehow need to evaluate whether something is trivial or already known or novel or just plain irregular. The model of what's known needs to be adaptive, quickly accessible, and able to generalize from experience with, say, flying elephants, to yet unseen instances of flying elephants. A look-up table model, for instance, is out of the question, due to its lack of generalization capability, and the large number of possible algorithms to investigate.

Here I propose to implicitly represent both algorithmic knowledge and model of knowledge in an adaptive data structure evolving through co-evolution of two essentially identical, algorithm-generating modules that collectively design experiments and bet on their outcomes, competing and cooperating at the same time. I will motivate each module to show the other regularities it does not yet believe in.

Each module is a set of modifiable, real-valued parameters, and owns initially equal amounts of "money" represented by two real-valued variables — the money idea in the context of machine learning can be traced back at least to Holland's bucket brigade system [9]. There is a function that maps the current values of the parameters of *both* modules onto a *single*, possibly complex algorithm that may include *Bet!* instructions corresponding to statements such as "cells 22 and 44 are now equal." Thus the modules *collectively* design and run an algorithm they have *agreed* upon.

For each *Bet!* instruction the modules can bet in advance whether the prediction is wrong or true (bids are real values, either fixed or based on the current module parameters). If they bet on different outcomes, given a particular *Bet!* instruction, then the system will immediately check which module is right. The winner gets rewarded (it receives the other's bid which is added to its money), while the surprised loser is punished (loses its bid).

It is absolutely essential that both modules *agree* on each experiment they assemble and execute. This is to make sure that no module can cheat. In the elephant example above, for instance, cheating would be possible if one of the modules could insert into its current algorithm instructions (not authorized by the other module) that look at cell 44's contents and use this information before computing an appropriate bid. This would be like two differently privileged viewers watching a magic trick, one in the audience being surprised by something the other finds entirely predictable because the latter can observe hidden movements of the magician from backstage. By agreeing on the entire instruction protocol of the current experiment the modules implicitly agree on all the information that may be used for the current computation and the current predictions.

In trying to maximize reward, each module is motivated to surprise the other as often as possible. Each is also motivated to veto computations whose outcome it does not consider profitable. As a consequence, each is essentially motivated to lure the other into agreeing on experiments that will surprise it. But by agreeing, each module expresses its belief that it is actually the other module which is in for a surprise. Using an appropriate evolutionary or reinforcement learning (RL) algorithm, a surprised module will eventually adapt. In turn, the other will lose a source of reward — an incentive to shift the exploration focus and try new algorithms reveiling *novel*, yet unknown, predictable regularities.

One of the simplest such RL algorithms divides the learners' life times into trials, each lasting, say, 1000 successive instructions. After each trial, the current loser is replaced by a copy of the winner (the module with the most money) if there is one, thus "learning" what the other knew. Money is then equally redistributed among both modules, and the parameters of one of them are modified via a random mutation such that the new competitors in the next trial are not exactly alike and tend to bet on different outcomes.

In the experiments, however, I will use a more sophisticated RL scheme that (1) takes into account the possibility of additional external reward from the environment (to be maximized by both modules), (2) does not depend on pre-wired trial lengths, and (3) replaces the simple mutation process mentioned above by more powerful, self-referential "sequences of module parameter-modifying instructions" that allow the algorithm-generating modules to modify themselves in a highly directed manner [35, 34]. Section 2 will present formal details, Section 3 will describe details of the RL algorithm, Section 4 will show that algorithmic surprise-generation of this kind can actually help to speed up external reward, Section 5 will conclude, and the appendix will describe details of the concrete implementation used in the experiments.

2 More Formally

The explorer's life in environment $\mathcal{E}$ lasts from time 0 (birth) to unknown time T (death). Two $m \times n$ matrices (LEFT and RIGHT) of modifiable real values represent the learner's modules. LEFT's k-th variable, vector-valued column is denoted LEFT_k; its l-th real-valued component $\text{LEFT}_{k,l}$; similarly for RIGHT. A variable *InstructionPointer* (*IP*) with range $\{1, \ldots, m\}$ always points to one of the module pair's columns. $\mathcal{S}$ is the learner's variable internal state with i-th component $\mathcal{S}_i \in \{-M, \ldots, -2, -1, 0, 1, 2, \ldots, M\}$ for $i \in \{1, \ldots, m\}$.

Throughout its life the learner repeats the following basic cycle over and over: select and execute instruction $a_j \in \mathcal{A}$ with probability $Q(IP, j)$ (here $\mathcal{A}$ denotes a set of possible instructions), where

$$Q(i, j) = \frac{f(\text{RIGHT}_{i,j}, \text{LEFT}_{i,j})}{\sum_k f(\text{RIGHT}_{i,k}, \text{LEFT}_{i,k})}$$

for $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$. The *collective decision function* $f(x, y)$ maps real-valued x, y to real values. Given an appropriate f, each module may "veto" instructions suggested by the other module. Only instructions that are strongly supported by both "modules" are highly likely to be selected (in the experiments I use $f(x, y) = xy$). Some instructions require parameters — these are selected in an analogous fashion, using probability distributions $Q(IP + 1, .)$, $Q(IP + 2, .)$, $\ldots$, $Q(IP + v, .)$, where v is the number of parameters. Instructions consume time and may change (1) environment $\mathcal{E}$: there are instructions such as *MoveAgentInCurrentDirection(StepSize), TurnAgent(Angle)*, (2) state $\mathcal{S}$: new inputs from the environment may affect the internal state; and there are also arithmetic instructions such as *Add(x,y,z)* whose interpretation is $\mathcal{S}_z \leftarrow (\mathcal{S}_x + \mathcal{S}_y)\ mod\ M$; (3) instruction pointer *IP*, for example, through conditional jump instructions; and (4) the modules themselves: there are instructions called *learning instructions* (LIs) that can modify the parameters determining the conditional probability distributions. To ensure non-vanishing exploration potential LIs may not generate module modifications that will let an instruction probability go to zero. LIs and other instructions can be combined to form more complex (probabilistic) learning algorithms. See Figure 1 for an illustration of how instruction sequences can be drawn from two probabilistic policies. See the appendix for details of a concrete implementation.

Reward. Occasionally $\mathcal{E}$ may distribute real-valued *external reward* equally between both modules. $R(t)$ is each module's cumulative external reward obtained between time 0 and time $t > 0$, where $R(0) = 0$. In addition there may be occasional *surprise rewards* triggered by the special instruction *Bet!(x,y,c,d)* $\in \mathcal{A}$. Suppose the two modules have agreed on executing a *Bet!* instruction, given a certain *IP* value. The probabilities of its four arguments $x, y \in \{1, \ldots, m\}$ and $c, d \in \{-1, 1\}$ depend on both modules: the arguments are selected according to the four probability distributions $Q(IP + 1, .)$, $Q(IP + 2, .)$, $Q(IP + 3, .)$, $Q(IP + 4, .)$ (details in the appendix). If $c = d$ then no module will be rewarded.

$c = 1, d = -1$ means that LEFT bets on $\mathcal{S}_x = \mathcal{S}_y$, while RIGHT bets on $\mathcal{S}_x \neq \mathcal{S}_y$. $c = -1, d = 1$ means that LEFT bets on $\mathcal{S}_x \neq \mathcal{S}_y$, while RIGHT bets on $\mathcal{S}_x = \mathcal{S}_y$. Execution of the instruction will result in an immediate test: which module is wrong, which is right? The former will be punished (surprise reward minus 1), the other one will be rewarded (+1). Note that in this particular implementation the modules always bet a fixed amount of 1 — alternative implementations, however, may compute real-valued bids via appropriate instruction sequences.

Interpretation of surprise rewards. Instructions are likely to be executed only if both modules collectively assign them high conditional probabilities, given $\mathcal{E}$ and $\mathcal{S}$. In this sense both must "agree" on each executed instruction of the lifelong computation process. In particular, both collectively set arguments x, y, c, d in the case they decide to execute a *Bet!* instruction. By setting $c \neq d$ they express that their predictions of the *Bet!* outcome differ. Hence LEFT will be rewarded for luring RIGHT into agreeing upon instruction subsequences (algorithms) that include *Bet!* instructions demonstrating that certain calculations yield results different from what RIGHT expects, and vice versa. Thus each module is motivated to discover algorithms whose outcomes surprise the other; but each also may reduce the probability of algorithms it does *not* expect to surprise the other.

The sheer existence of the *Bet!* instruction motivates each module to act not only to receive external reward but also to obtain surprise reward, by discovering algorithmic regularities the other module still finds surprising — a type of curiosity.

Trivial and random results. Why not provide reward if $\mathcal{S}_x = \mathcal{S}_y$ and $c = d = 1$ (meaning both modules rightly believe in $\mathcal{S}_x = \mathcal{S}_y$)? Because then both would soon focus on this particular way of making a correct and rewarding prediction, and do nothing else. Why not provide punishment if $\mathcal{S}_x = \mathcal{S}_y$ and $c = d = -1$ (meaning that both modules are wrong)? Because then both modules would soon be discouraged from making any prediction at all. In case $c = d = 1$ the truth of $\mathcal{S}_x = \mathcal{S}_y$ is considered a well-known, "trivial" fact whose confirmation does not deserve reward. In case $c = d = -1$ the truth of $\mathcal{S}_x = \mathcal{S}_y$ is considered a subjectively "random," irregular result. Surprise rewards can occur only in the case both modules' opinions differ. They reflect one module's disappointed *confident* expectation, and the other's justified one, where, by definition, "confidence" translates into "agreement on the surprising instruction sequence" — no surprise without such confidence.

Examples of learnable regularities. A Turing-equivalent instruction set $\mathcal{A}$ (one that permits construction of a universal Turing machine) allows exploitation of arbitrary computable regularities [11, 2, 39, 14] to trigger or avoid surprises. For instance, in partially predictable environments the following types of regularities may help to reliably generate matches of computational outcomes. *(1) Observation/prediction:* selected inputs computed via the environment (observations obtained through "active perception") may match the outcomes of earlier internal computations (predictions). *(2) Observation/explanation:* memorized inputs computed via the environment may match the outcomes of later internal computations (explanations). *(3) Planning/acting:* outcomes of internal computations (planning processes) may match desirable inputs *later* computed via the environment (with the help of environment-changing instructions). *(4) "Internal" regularities:* the following computations yield matching results — subtracting 2 from 14, adding 3, 4, and 5, multiplying 2 by 6. Or: apparently, the computation of the truth value of "n is the sum of two primes" yields the same result for each even integer $n > 2$ (Goldbach's conjecture). *(5) Mixtures of (1–4).* For instance, raw inputs may be too noisy to be precisely predictable. Still, there may be internally computable, predictable, informative input transformations [33]. For example, hearing the first two words of the sentence "John eats chips" does not make the word "chips" predictable, but at least it is likely that the third word will represent something edible. Examples (1-5) mainly differ in the degree to which the environment is involved in the computation processes, and in the temporal order of the computations.

Curiosity's utility? The two-module system is supposed to solve self-generated tasks in an unsupervised manner. But can the knowledge collected in this way help to solve externally posed tasks? Intuition suggests that the more one knows about the world the easier it will be to maximize external reward. In fact, later I will present an example where a curious system indeed outperforms a non-curious one. This does not reflect a universal law though: in general there is no guarantee that curiosity will not turn out to be harmful (for example, by "killing the cat" [40]).

Relative reward weights? Let $RL(t)$ and $RR(t)$ denote LEFT's and RIGHT's respective total cumulative rewards obtained between time 0 and time $t > 0$. The sum of both modules' surprise rewards always remains zero: we have $RL(t) + RR(t) - 2R(t) = 0$ for all t.

If we adopt the traditional hope that exploration will contribute to accelerating environmental rewards, then zero-sum surprise rewards seem to afford less need to worry about the relative weights of surprise versus other rewards than the surprise rewards of previous approaches, which did not add up to zero.

Enforcing fairness. To avoid situations where one module consistently outperforms the other, the instruction set includes a special LI that copies the currently superior module onto the other (see the appendix for details). This LI (with never-vanishing probability) will occasionally bring both modules on a par with each other.

In principle, each module could learn to outsmart the other by executing subsequences of instructions that include LIs. But how can we ensure that each module indeed improves? Note that arithmetic actions affecting $\mathcal{S}$ and jump instructions affecting IP cause a highly non-Markovian setting and prevent traditional RL algorithms based on dynamic programming from being applicable. For such reasons I use *Incremental Self-improvement (IS)* [35, 34] to deal with both modules' complex spatio-temporal credit assignment problem.

3 Incremental Self-improvement (IS)

The currently surprising module wants to repeat similar surprises with higher probability in the future. The other wants to avoid further surprises by learning not to agree on similar computation sequences (implicitly learning what the other already knows). And it wants to be "creative" in the sense that it wants to generate new surprises for the other module instead. In principle, both can learn by executing subsequences of instructions that include LIs. How can we ensure that each module indeed improves by accelerating its reward intake?

In this chapter I will use the IS paradigm [35, 34] to deal with both modules' complex spatio-temporal credit assignment problem. This does not necessarily mean that IS is the best way of doing so. Other RL paradigms may be appropriate, too — this chapter's basic ideas are independent of the choice of RL method. IS seems attractive, however, because: (1) It does not make an explicit difference between learning algorithms and other instructions, or between learning, metalearning, metametalearning, etc. (2) It properly takes into account that the success of each module modification recursively depends on the success of all later modifications for which it is setting the stage. (3) Its objective takes into account the entire time consumed by lifelong learning itself. (4) It is designed for quite general non-Markovian credit assignment problems in lifelong learning situations — see [35, 34, 31] for recent IS applications. Following [34], the remainder of this section will describe LEFT's IS-based learning process. RIGHT's is analogous.

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

Sequences of module modifications. SLM$_k$ denotes the sequence of LEFT-modifications (SLM) computed by LIs between checkpoints l_k and l_{k+1}. Since LI execution probabilities depend on LEFT and RIGHT, the modules can in fact modify the way they modify themselves — they can devise their own probabilistic learning algorithms.

Goal. At some checkpoint t LEFT's goal is to generate LEFT-modifications that will accelerate long-term reward intake: it wants to let the value of $(RL(T) - RL(t))/(T-t)$ exceed the current average reward intake.

Success-story criterion (SSC). LEFT maintains a time-varying set V of past LEFT checkpoints that have led to long-term reward accelerations. Initially V is empty. Let v_k denote the k-th element of V in ascending order. SSC is satisfied at time t if either V is empty (trivial case) or if

$$\frac{RL(t) - RL(0)}{t - 0} < \frac{RL(t) - RL(v_1)}{t - v_1} < \ldots < \frac{RL(t) - RL(v_{|V|})}{t - v_{|V|}}. \tag{1}$$

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

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

```
1. WHILE  SSC is not satisfied

       Undo all LEFT-modifications made since the most recent
       checkpoint in $V$.
       Remove that checkpoint from $V$.

2. Add the current checkpoint to $V$.
```

"Undoing" a modification means restoring the preceding LEFT — this requires storing past values of LEFT-components on a stack prior to modification. (Components of LEFT and elements of V can be stored on the same stack — see the appendix.) Thus each LEFT-modification that survived SSA is part of a bias shift generated after a checkpoint marking a lifelong reward speed-up: since v_j there has been more reward per time than since v_i, for $v_j > v_i$ ($v_j, v_i \in V$).

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

SSA's generalization assumption. At the end of each SSA call, until the beginning of the next one, the only temporary generalization assumption for inductive inference is: LEFT-modifications that survived all previous SSA calls will remain useful. In the absence of empirical evidence to the contrary, each surviving SLM$_k$ is assumed to have set the stage for later successful SLM$_i$, $i > k$. Since life is one-way (time is never reset), during each SSA call the system has to generalize from a *single* experience concerning the utility of LEFT-modifications executed after any given previous point in time: the average reward per time since then.

Implementing SSA. Using stack-based backtracking methods such as those described in [35, 34] and the appendix, one can guarantee that SSC will be satisfied right after each new SLM-start, despite interference from S, $\mathcal{E}$, and RIGHT. Although inequality 1 contains $|V|$ fractions, SSA can be implemented efficiently: only the two SLMs on top of the stack need to be considered at any given time in an SSA call (see details in appendix). A *single* SSA call, however, may undo *many* SLMs if necessary.

What has been described for LEFT analogously holds for RIGHT. The appendix describes a particular implementation (the one used for the experiments) based on an assembler-like programming language similar to those used in [34, 26].

4 Experiments

It has already been shown that IS by itself can solve interesting tasks. For instance, [34] describes two agents A and B living in a partially observable 600×500 environment with obstacles. They learn to solve a complex task that could not be solved by various TD(λ) Q-learning variants [15]. The task requires (1) agent A to find and take "key A"; (2) agent A go to "door A" and open it for agent B; (3) agent B to enter through "door A," find, and take another "key B"; (4) agent B to go to another "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 unit lengths in front of it. The agent can also move forward (up to 30 unit lengths), change its direction, turn relative to its key or its door or the goal. It can use memory (embodied by its *IP*) to disambiguate inputs. 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. In the beginning, the goal is found only every 300,000 basic cycles. Through IS, however, within 130,000 trials the average trial length decreases by a factor of 60 — both agents learn to cooperate to accelerate reward intake [34].

This section's purpose is not to elaborate on how IS can solve difficult tasks. Instead IS is used as a particular vehicle to implement the two-module idea for preliminary attempts at studying "inquisitive" explorers. Subsection 4.1 will describe empirically observed system behavior in the absence of external rewards. In Subsection 4.2 there will be additional reward for solving externally posed tasks, to see whether curiosity can indeed be useful.

Experimental details. There are $n = 24$ instructions (see the appendix) and $m = BS(n^2 \; div \; BS) = 576$ columns per module. $M = 100,000$. Time is measured as follows: selecting an instruction head, selecting an argument, selecting one of the two values required to compute the next instruction addresses, pushing or popping a module column costs one time step. Other computations do not cost anything. This ensures that measured time is of the order of total CPU-time. For instance, selecting an instruction head plus six arguments plus the next *IP* address costs $1 + 6 + 2 = 9$ time steps.

Figure 2 shows a point-like agent's two-dimensional environment whose width is 1000 unit lengths. The large "room" in the south is a square. Its southwest corner has coordinates (0.0, 0.0), its southeast corner (1000.0, 0.0). There are infinitely many possible agent states: the agent's current position is given by a pair of real numbers. Its initial coordinates are $(50.0, 50.0)$, its initial direction is 0, its stepsize 12 unit lengths. Compare appendix A.3.2.

4.1 Experiment 1: No External Reward

This introductory experiment focuses on the totally unsupervised case — there is no external reinforcement for solving pre-wired tasks. Thus there also is no objective way of measuring the system's performance. The experiment's only purpose is to give an impression of what happens while the active learner is running freely. In what follows I will describe a single but rather typical run.

The entrance to the small room in Figure 2 is blocked. The system runs for 10^9 time steps corresponding to about 10^8 instructions. The system does make heavy use of its arithmetic instructions: soon the entire storage is filled with varying numbers generated as by-products of its computations.

Zero-sum reward. Consider Figure 3. The derivatives of the reward plots at a given time tell which module currently receives more reward than the other. There are long time intervals during which one of the modules dominates. The existence of zero crossings, however, shows that each module occasionally collects sufficient negative rewards to cancel all the previously collected positive rewards, and vice versa. This means that no module consistently outperforms the other.

Stack pointer evolution. The stack pointers reflect the number of currently valid module column modifications. Consider Figure 4. Initially the stacks grow very quickly. Then there is a consolidation phase. Later growth resumes, but at a much more modest level — it becomes harder and harder for each module to acquire additional "probabilistic knowledge" to outwit the other. Sometimes SSA pops off a module's entire stack (Figure 4's temporal resolution is too low to show this), but in such cases the stack pointer soon reaches its old value again. This is partly due to *SSAandCopy* instructions copying the currently superior module onto the other to enforce fair matches.

Reward acceleration. Even in later stages both modules are able to accelerate their long-term average reward intake again and again, despite the fact that the sum of all rewards remains zero. As each module collects a lot of negative reward during its life, it continually pops appropriate checkpoints/modifications off its stack such that the resulting histories of surviving module modifications correspond to histories of less and less negative reward/time ratios. Recall the occasional popping off of the entire stack.

Evolution of instruction frequencies. Consider Figure 5. Although the sum of all surprise rewards remains zero, *Bet!* instructions are soon among the most frequent types. Other instructions also experience temporary popularity, often in the form of sharp peaks beyond the plot's resolution. Usually, however, the module that does not profit from them learns to put its veto in.

Interestingness depends on current knowledge and computational abilities. These are very different for human observers and my particular implementation.

It seems hard to trace and understand the system's millions of self-surprises and self-modifications. Figure 5, for instance, provides only very limited insight into the nature of the complex computations carried out. It plots frequencies of selected instruction types but ignores the corresponding arguments and *InstructionPointer* positions. It does not reflect that for a while computations may focus on just a few module columns and storage cells, until this gets "boring" from the system's point of view. Much remains to be done to analyze details of the system's complex dynamics.

Creativity's utility? Can the probabilistic knowledge collected by a "creative" two-module system (solving self-generated tasks in an unsupervised manner) help to solve externally posed tasks? The following experiment provides a simple example where this type of curiosity indeed helps to achieve faster goal-directed learning in the case of occasional non-zero environmental reward.

4.2 Experiment 2: Additional External Reward

4.2.1 Experiment 2a.

Consider Figure 2. The entrance to the small room is blocked. Whenever the point-like agent moves into the 100×100 northeast corner (GOAL1) of the 1000×1000 field it receives external reward 100.0 and is immediately teleported back to start position $(50.0, 50.0)$; its direction is reset to 0.0.

Solution. GOAL1 can be reached by a simple stochastic policy making the agent head in the northeast direction. Once the system somehow has fixed the agent's direction in an optimal way, the shortest path requires about 100 calls of *MoveAgent* (due to the agent's limited stepsize). We will see that the major difficulties in *learning* such a policy are due to the system's extremely low initial bias towards solving this task, and the extremely rare occurrences of rewarding training instances.

Random behavior results. With its module-modifying capabilities being switched off (LIs such as IncProbLEFT(x_1, x_2, y_1) have no effect), the system exhibits random behavior according to its maximum entropy initialization. This behavior leads to about five visits to GOAL1 per 10 million time steps, which shows how little built-in knowledge there is about the nature of the goal. Following Geman et al. [6], the learning system's bias towards solving the task is extremely low, while variance is high. There is a confusing choice of $2 \times 576 = 1152$ module columns (many more than needed for solving the task), all being potential candidates for modifications. The two modules initially have no idea that two of the 23 instruction types (namely, *MoveAgent* and *SetDirection*) are particularly important. Furthermore, module modifications can be computed only by appropriate instruction subsequences (including LIs) generated according to the module columns themselves. Initially all instructions are extremely stochastic, however, partly due to the many arguments and addresses to jump to. Reducing this "free parameter overkill" by using smaller modules would be equivalent to inserting more a priori knowledge about the task.

Comparison. I compare the performance of two learning systems. One is curious/creative in the sense described in the previous sections, the other is not. The latter is like the former except that its *Bet!* action has no effect — external reward is the only type of reward. Ten simulations are conducted for each system. Each simulation takes 200 million time steps. During the 10th and the 20th 10^7 time step interval I count how often the agent visits GOAL1. Since external reward initially occurs only every 2 million time steps on average, goal-relevant training examples are very rare.

Results. Table 1 lists the plain and the curious systems' goal visits per 10^7 time steps after half their life times, and near death. All 10 results are given due to the high variance. The curious system's results tend to be more than an order of magnitude better than the plain's, which tend to be more than an order of magnitude better than the random's.

Figures 6, 7, 8 show details of the particularly successful simulation 1. We observe that total reward fluctuations due to surprise rewards appear negligible, although they seem important for overall performance improvement (compare Table 1). Between 20 and 30 million steps there are lots of *Bet* and *SSAandCopy* actions. Soon afterwards the instruction types *Move* and *SetDirection* start to dominate. Around 130 million steps their frequencies suddenly rise dramatically. The final breakthrough is achieved shortly thereafter. The modules' first 100 columns after simulation 1 are shown in Figure 9. Both modules are almost identical. The probability mass of many columns is concentrated in a single value. Additional inspection revealed, however, that some of the most frequently used instruction addresses point to maximum entropy distributions.

Possible explanation. How can curiosity/creativity help? In early system life external rewards occur very rarely. During this time, however, there are many surprise reward-oriented instruction subsequences. Possibly they provide the system with useful fragments of "probabilistic knowledge" about consequences of its innate instructions. For instance, to repeatedly surprise itself it may construct little "probabilistic subprograms" involving highly probable jumps to appropriate module column addresses. Once it has figured out how to increase the likelihood of certain jumps it may find it easier to solve external tasks that also require jumps.

Details of how the system actually came up with the final matrices remain quite unclear, however. More experimental analysis is necessary to better understand how solving external tasks can benefit from pure curiosity.

The outlier. Note the curious system's outlier in the second simulation. Inspection revealed that during the entire simulation the system never reached the goal at all — there was not a single training example concerning external reward. This outlier reflects curiosity's potential drawback: it may focus attention on computations that do not have anything to do with external reinforcement. On average, however, surprise rewards do help.

Runtimes. To some readers the runtimes of several hundred million time

Simulation	plain: half	curious: half	plain: final	curious: final
1	4	899	9	15774
2	4	0	4	0
3	3	204	15	381
4	124	558	1036	3150
5	399	539	1000	2073
6	9	1024	82	2937
7	9	1024	123	1181
8	6	891	20	1561
9	58	89	324	446
10	62	5603	476	9963
Average	68	1083	309	3746

Table 1: Experiment 2a: 10 simulations, each taking $2 * 10^8$ time steps. The table lists goal visits achieved by plain and curious systems during the 10th (half life time) and the final 10^7 time step interval. Random behavior leads to about 5 visits per 10^7 time steps. Curiosity typically seems to help a lot. But note the second simulation's "outlier."

steps may seem large. Note, however, that the number of training examples (trials) is much smaller because the goal is hit very rarely, especially in the beginning of the learning phase. Some initial trials take millions of time steps simply because the system does not know that there is a source of reward at all, and that some of its instructions (those for interaction with the environment) are much more important for achieving the goal than arithmetic instructions — not one biases its search towards the goal. The following simulations will require even longer runtimes.

4.2.2 Experiment 2b.

I keep the basic setup from Experiment 2a, but open the entrance to the small room in Figure 2 by removing the block. Whenever the point-like agent moves into the small room's 100×100 northeast corner (GOAL2) it receives external reward 1000 (10 times as much as for reaching GOAL1) and is teleported back to start position $(50.0, 50.0)$; its direction is reset to 0.0.

The idea is: knowledge collected by solving the simpler task (reaching GOAL1) may help to solve the more difficult, but also more rewarding task (reaching GOAL2). Note that *both* tasks are solvable by similar stochastic policies making the agent head northeast. To receive a lot of reward, however, the agent must avoid GOAL1 on its way to GOAL2 — otherwise it will be teleported back home.

This setup involves an obvious goal-directed exploration component: it is likely that the system will keep an exploration strategy that has helped to improve a policy for reaching GOAL1 (successful exploration strategies have an "evolutionary advantage"). This strategy may later also help to improve a policy for reaching GOAL2 — in principle, IS can use experience with exploration strategies to evaluate and refine them.

Random behavior results. With its module-modifying capabilities being switched off (LIs such as IncProbLEFT(x_1, x_2, y_1) have no effect), the system exhibits random behavior according to its maximum entropy initialization. On average this behavior leads to less than one visit of GOAL2 per 10 million time steps.

Comparison. Again I compare the performance of a "plain" and a "curious" system. Again the former is like the latter except that its *Bet!* instructions have no effect. Ten simulations are conducted for each system. Each simulation takes 4 billion time steps. I keep track of how often the agent reaches GOAL2 per 10^7 time steps.

Results. Figure 10 plots average goal visit frequencies during the first 500 million time steps. During this period, the curious system's results are clearly better than the plain's, and its performance improves much faster.

As more simulation time is spent, however, initial differences tend to level out. Figure 11 plots average goal visit frequencies during the entire 4 billion steps. As goal-oriented training examples become more and more frequent, we see that the performances of plain and curious systems eventually reach comparable levels — the former even becomes a bit better than the latter.

Advantage in the case of rare rewards? For this particular experiment, self-generated surprise rewards appear to boost initial external reward. Sufficient training time, however, cancels out the initial advantage. Could it be that curiosity is particularly useful as long as external reward is extremely rare and little has already been learned? This may seem intuitively plausible: in the beginning performance tends to be so bad that time spent on extensive exploration seems to be a good investment with little downside potential — things cannot get much worse. Once the system's strategy is rather efficient and yields frequent external rewards, however, additional progress depends on fine-tuning it. In this stage surprise reward-oriented curiosity may distract more than it helps — it tends to consume time without necessarily contributing much to optimizing goal-directed trajectories. Many additional experiments are necessary, however, to understand whether the above is a typical result or not.

5 Final Remarks

Previous work on IS (for example, [34, 35]) implicitly focused on goal-oriented exploration — in principle IS can learn to change its exploration strategy if this turns out to accelerate external reward in the long run. This chapter's IS implementation, however, involves an additional *pure* exploration component besides the goal-oriented one. Part of the learner receives internal reward for pointing out something another part did not know but thought it knew. The surprised part suffers to the extent the surprising part benefits — the sum of all internal rewards remains zero. The learner is "interested" in "creative" computations leading to unexpected results, while simultaneously trying to make formerly surprising things predictable and boring. It does not care much for irregular noise rich with Shannon information [38]. Instead it prefers easily learnable algorithmic regularities, taking into account the costs of gaining information in an RL framework.

From each module's perspective an instruction subsequence is "novel" as long as its outcome surprises the other. Since the surprised module will eventually figure out what is going on, there will be incessant pressure to create new novelties. What is the use of such an inquisitive system's computations? Curiosity's long-term justifiability depends on whether knowledge growth will eventually support goal-oriented behavior. When will this be the case? The question is reminiscent of G. H. Hardy's toast on "pure" mathematics — the kind that "would never be of any use to anyone" ([3], p. 185). History teaches us, however, that it is hard to decide which math will be useless forever. For instance, old results from "pure" number theory are used in today's encryption technology.

In general, however, it will always be possible to design environments where "curiosity kills the cat" [40], or at least has negative influence on external performance. For instance, as exemplified by simulation 2 of Experiment 2a, curiosity may occasionally prevent discovery of external reward sources. This is reminiscent of the situation in supervised learning. There often additional "regularizer" terms are added to the standard error function defining network performance on the training data. They can greatly help to remove redundant free parameters and improve generalization capability on unseen data (e.g., [8]), but in general this cannot be guaranteed.

This chapter's approach draws inspiration from several sources. For instance, the two-module system is based on two co-evolving modules. Co-evolution of competing strategies, however, is nothing new. See, for example, [7, 19] for interesting cases. Also, the idea of improving a learner by letting it play against itself is ancient. See, for example, [20, 41]. Even the idea of unsupervised learning through co-evolution of predictors and modules trying to escape the predictions is nothing new — it has been used extensively in our previous work on unsupervised sensory coding with neural networks [25, 36, 33, 32, 37]. Finally, co-evolutionary methods translating mismatches between reality and expectations into reward for "curious," exploring agents are not new either — see our previous work on "pure" RL-based exploration [24, 23, 40]. So, what is new?

Novel is the idea that both adaptive modules equally influence the probability of each executed instruction/computation. This (1) allows for a straightforward way of making both modules equally powerful (by copying the currently superior one onto the other), and (2) prevents each module from being able to enforce computations that will make the other lose no matter what it tries. For instance, details of white noise on a screen are inherently unpredictable, but none of the two opponents may exploit this to generate surprises if the other does not "agree" to the corresponding experiment. And it will agree only as long as it suspects that there is a regularity in the white noise that the other does not yet know. The precondition of a surprise is that the surprised module has expressed its confidence in a different outcome of the surprising computation sequence by participating in the collective decision process. Intuitively, my adaptive explorer continually wants to discover new, "creative" uses of its innate sensorium and computational potential. It wants to focus on those novel things that seem easy to learn, given current knowledge. It wants to ignore (1) previously learned, predictable things, (2) inherently unpredictable ones (such as details of white noise on a screen), and (3) things that are unexpected but not expected to be easily learned (such as the contents of an advanced math textbook beyond the explorer's current level).

Another novel aspect is the general setting. Instead of being limited to Markovian domains and simple reactive strategies such as approaches in [23, 40], this chapter's setup allows for quite arbitrary domains and computations. This is made possible by the recent IS paradigm [34, 35]. There is no essential limit (besides computability) to the nature of the regularities that may be exploited to generate surprises. Neither is there an essential limit to the nature of the learning processes that can make formerly surprising regularities predictable and boring. There may be RL schemes even more general than IS, but this is beyond the scope of this chapter.

Note that this chapter's notion of "simple regularities" differs from, e.g., Kolmogorov complexity theory's [11, 2, 39, 14, 26]. There an object is called simple relative to current knowledge x if the size of the shortest algorithm computing it from x is small. The algorithm's computation time is ignored, as are constant factors reflecting Kolmogorov complexity's machine independence. The current chapter, however, takes both into account.

As the explorer's knowledge about its environment and computational abilities expands, it keeps balancing on the thin, dynamically changing line between the subjectively random and the subjectively trivial. Unlike Nake and other authors he cites [17], I do not suggest a predefined optimal ratio between known and unknown information. Instead, the two cooperating/competing modules dynamically, implicitly determine this ratio as they keep trying to surprise each other.

Recent papers attempt to explain "beauty" with the help of complexity theory concepts [27, 29]. They argue that something "beautiful" need not be "interesting". They predict that the "most beautiful" object from a set of objects satisfying certain specifications is the one that can be most easily computed from the subjective observer's input coding scheme. Interestingness in the current chapter's sense, however, also takes into account whether the computational result is expected or not. Something that is both "beautiful" and already known may be quite boring — "beauty" needs to be unexpected to awaken interest.

Future work. The programming language used in the experiments is designed to allow for fairly arbitrary computations/explorations and learning processes. To make progress towards analyzing "inquisitive" explorers, however, one will probably have to study alternative systems with less computational power and less general RL paradigms but more accessible dynamics. On the other hand, it will also be interesting to study a curious learner's performance in the case of more difficult tasks and more powerful primitive instructions with more bias towards solving the task. Note that LIs can be almost anything: neural net algorithms, Bayesian analysis algorithms, etc.

Furthermore, although IS is a rather general RL paradigm, it may be possible to develop more general ones. In that case I would like to combine them with the two-module idea. Promising candidates may be RL schemes based on economy and market models, such as classifier systems and their variants [9, 44, 45, 42, 43], or the related "Prototypical Self-referential Associating Learning Mechanisms" (PSALMs) [21], the Neural Bucket Brigade [22], Hayek Machines [1, 12], Collective Intelligences (COINs) [46].

The basic ideas of the present chapter will probably remain unchanged, however: competing agents will agree on algorithmic experiments and bet on their outcomes, the winners profiting from outwitting others.

Acknowledgments

I would like to thank Jieyu Zhao (supported by SNF grant 21-43'417.95 "Incremental Self-improvement"), Nic Schraudolph, Rafal Salustowicz, Sepp Hochreiter, Marco Wiering, and Luca Gambardella for valuable discussions and comments on a draft of this chapter, which is based on the following earlier publications: [28, 30].

A Appendix

In what follows I will describe details of the system used in the experiments.

Architecture. The internal state $\mathcal{S}$ consists of m addressable cells with addresses ranging from 0 to $m-1$ (instead of ranging from 1 to m — this is due to a pecularity of C, the implementation language). $\mathcal{S}_k \in \{-M, -M+1, \ldots, 0, 1, \ldots, M\}$ are the current contents of the cell with address k. Instructions and arguments are encoded by a fixed set I of n integer values $\{0, \ldots, n-1\}$. For each value j in I, there is an instruction head a_j with $n_j < BS - 2$ integer-valued arguments, where BS is the instruction blocksize, and m is a multiple of BS. In the experiments I use $BS = 9$ (there are at most six arguments per instruction), $n = 24$, $m = BS(n^2 \; div \; BS) = 576$, and $M = 100,000$. See list below for instruction syntax and semantics.

$\textsc{Right}$ **and** $\textsc{Left}$ **modules.** All $\textsc{Right}_i$ and $\textsc{Left}_i$ ($i \in \{0, \ldots, m-1\}$) are vectors of n positive, real values that sum up to 1.0. The k-th component of $\textsc{Right}_i$ ($\textsc{Left}_i$) is denoted $\textsc{Right}_{i,k}$ ($\textsc{Left}_{i,k}$) for $k \in \{0, \ldots, n-1\}$. A variable *InstructionPointer* (*IP*) with range $\{0, \ldots, m-1\}$ always points to one of the module pair's columns. *IP* is viewed as a modifiable part of the environment.

Initialization. At system birth at time 0, all $\textsc{Right}_{i,k}$ and $\textsc{Left}_{i,k}$ are set equal to $1/n$. All $\mathcal{S}_k$ and *IP* are set to zero. They will never be re-initialized again. To be able to restore modified module columns if necessary, we introduce two initially empty stacks Stack$\textsc{Right}$ and Stack$\textsc{Left}$ that allow for variable-sized stack entries, and the conventional ***push*** and ***pop*** operations. Instructions may change the Boolean variables *BlockSSA*$\textsc{Left}$ and *BlockSSA*$\textsc{Right}$ (both are modifiable parts of $\mathcal{E}$ and initially FALSE at time 0).

A.1 Basic Cycle of Operations

Until unknown time T (system death), the system repeats the following basic instruction cycle over and over.

1. Select instruction head $a_j \in I$ with probability $Q(IP, j)$, where

$$Q(i,j) = \frac{f(\textsc{Right}_{i,j}, \textsc{Left}_{i,j})}{\sum_k f(\textsc{Right}_{i,k}, \textsc{Left}_{i,k})},$$

$$for\ i \in \{0, \ldots, m-1\}, j \in \{0, \ldots, n-1\}.$$

Here the *collective decision function* $f(x,y)$ maps real-valued x, y to real values. Given an appropriate f, each module may "veto" instructions suggested by the other module. Only instructions that are strongly supported by both modules are highly likely to be selected. One possibility is $f(x,y) = min(x,y)$. In the experiments I use $f(x,y) = xy$.

Comment: owing to pecularities of certain instructions to be introduced below, $Q(i,j)$ will later be refined for cases where i addresses an instruction head as opposed to an argument.

2. a_j's $n_j \leq 6$ arguments $\in \{0, \ldots, n-1\}$ are selected according to probability distributions $Q(IP{+}1, .), Q(IP{+}2, .), \ldots, Q(IP{+}n_j, .)$ (except when $a_j =$ *Bet!* — two of *Bet!*'s arguments will be treated differently — see Section A.3.4 below).

3. Execute the selected instruction. This will consume time and may change (1) environment $\mathcal{E}$, (2) *IP*, (3) internal state $\mathcal{S}$; (4a) $\textsc{Right}$, (4b) $\textsc{Left}$. If there is external reward R then set $\mathcal{S}_8 \leftarrow R$ (rewards become visible to the system in the form of inputs).

4. If an input has changed one of the cell contents $\mathcal{S}_0$, $\mathcal{S}_1, \ldots, \mathcal{S}_8$, then shift the contents of $\mathcal{S}_0$, $\mathcal{S}_1, \ldots, \mathcal{S}_{80}$ to components $\mathcal{S}_9$, $\mathcal{S}_{10}, \ldots, \mathcal{S}_{89}$, respectively.

This results in a built-in short-term memory (long-term memory can be implemented by the system itself by executing appropriate instruction sequences).

5. If a_j did not modify IP (no conditional jump — compare instruction list below), then compute the address of the next instruction head by setting $IP \leftarrow w - w \; mod \; BS$. Here $w = (w_1 n + w_2) \; mod \; m$, where $w_1 \in \{0, \ldots, n-1\}$ is selected according to probability distribution $Q(IP + 7, .)$, while $w_2 \in \{0, \ldots, n-1\}$ is selected according to $Q(IP + 8, .)$.

6. Goto **1.**

A.2 SSA Calls

At a given time LEFT's current stack will be either empty or of the form $Block_1$, $Block_2, \ldots, Block_u$, where $Block_i$ ($i \in \{1, \ldots, u\}$) is of the form

$$[v_i, RL(v_i), (c_i^1, \text{LEFT}_{c_i^1}), (c_i^2, \text{LEFT}_{c_i^2}), \ldots].$$

Here v_i denotes the i-th checkpoint still in LEFT's stack (compare section refis), $RL(v_i)$ denotes LEFT's reward until time v_i, c_i^k ($k \in \{1, 2, \ldots, \}$) is the address of the k-th LEFT-column modified in between v_i and the subsequent checkpoint, and $\text{LEFT}_{c_i^k}$ is the corresponding previous LEFT-column. The pairs $(c_i^k, \text{LEFT}_{c_i^k})$ were saved on the stack by the first LI that changed the corresponding LEFT-columns after v_i.

The procedures SSALEFT() and SSARIGHT() below will be invoked by module-modifying instructions to be discussed later.

SSALEFT():

1. If *BlockSSA*LEFT = TRUE then exit. Else:

2. Set *BlockSSA*LEFT = TRUE. Set variable t equal to current time (t is a new checkpoint). Use backtracking and the information in StackLEFT to undo as many of the most recent LEFT-modifications as necessary to achieve SSC — see inequality (1) in section 3. Pop off the corresponding blocks in StackLEFT. This procedure guarantees that SSC will eventually be satisfied — see, for example, [35].

3. Push t and $RL(t)$ onto StackLEFT. They are the first two elements of the next block to be pushed.

SSARIGHT() is analogous to SSALEFT().

A.3 Semantics of Instruction Heads and Their Arguments

In what follows, $x_1, x_2,\ y_1, y_2,\ z_1, z_2, \in \{0, \ldots, n-1\}$ stand for instruction arguments selected according to probability distributions $Q(IP+1, .), Q(IP+2, .), \ldots, Q(IP+6, .)$, respectively. They are used to address state components and module columns. For simplicity, instruction descriptions below use the following macros: $x := (x_1 n + x_2)\ mod\ m$; $y := (y_1 n + y_2)\ mod\ m$; $z := (z_1 n + z_2)\ mod\ m$.

A.3.1 Instructions Operating on Internal State Only

Jmpl$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** If $\mathcal{S}_x < \mathcal{S}_y$ then $IP \leftarrow z - z\ mod\ BS$.

Jmpeq$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** If $\mathcal{S}_x = \mathcal{S}_y$ then $IP \leftarrow z - z\ mod\ BS$.

Add$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** $\mathcal{S}_z \leftarrow (\mathcal{S}_x + \mathcal{S}_y)\ mod\ M$

Sub$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** $\mathcal{S}_z \leftarrow (\mathcal{S}_x - \mathcal{S}_y)\ mod\ M$

Mul$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** $\mathcal{S}_z \leftarrow (\mathcal{S}_x \mathcal{S}_y)\ mod\ M$

Div$(x_1, x_2, y_1, y_2, z_1, z_2)$ **:** if $\mathcal{S}_y \neq 0$ then $\mathcal{S}_z \leftarrow (\mathcal{S}_x\ div\ \mathcal{S}_y)\ mod\ M$

Mov(x_1, x_2, y_1, y_2)**:** $\mathcal{S}_y \leftarrow \mathcal{S}_x$

Init(x_1, x_2, y_1, y_2)**:** $\mathcal{S}_y \leftarrow x$.

A.3.2 Instructions for Interaction with the Environment

There is a point-like agent with "active perception." It can move about in a two-dimensional environment (details of the environment are described in Section 4). The agent's current position is given by a pair of real values. Its current direction is given by variable *Direction* with range $[0, \ldots, 2\pi]$.

MoveAgent(): Move the agent 12 unit lengths in direction *Direction*, unless an obstacle or the environment's boundary blocks the path (the value 12 was chosen arbitrarily — neither it nor any other arbitrarily chosen constants mentioned below have ever been changed during the experiments). Moving may lead to new inputs: $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2$ become equal to the agent's horizontal position, vertical position, and current direction, respectively. Here position components are rounded to integers between 1 and 1000, and directions are scaled to $[0, 100]$ and then rounded. If the point 24 unit lengths in front of the agent is within an obstacle or outside the enviroment, then $\mathcal{S}_3 \leftarrow 24$ (the value 24 is chosen arbitrarily). Otherwise $\mathcal{S}_3 \leftarrow 0$. Similarly for $\mathcal{S}_4$ (input 90 degrees to the right), $\mathcal{S}_5$ (behind), $\mathcal{S}_6$ (to the left).

SetDirection(x_1)**:** *Direction* $= \frac{x_1}{n} 2\pi$. Compute new inputs like instruction MoveAgent().

A.3.3 "Introspective" Instructions

GetLeft(x_1, x_2, y_1)**:** $\mathcal{S}_x \leftarrow round(M$ Left$_{x,y_1})$

GetRight(x_1, x_2, y_1)**:** $\mathcal{S}_x \leftarrow round(M$ Right$_{x,y_1})$.

A.3.4 Bet Instructions

Bet!$(x_1, x_2, y_1, y_2, c, d)$**:** Select a value $c' \in \{0, \ldots, n-1\}$ with probability $\frac{\text{Left}_{IP+5,c'}}{\sum_k \text{Left}_{IP+5,k}}$. If $c' > n/2$ then set variable $c \leftarrow 1$, else $c \leftarrow -1$. Select a value $d' \in \{0, \ldots, n-1\}$ with probability $\frac{\text{Right}_{IP+5,d'}}{\sum_k \text{Right}_{IP+5,k}}$. If $d' > n/2$ then set variable $d \leftarrow 1$, else $d \leftarrow -1$. If $c = d$ then exit. If $\mathcal{S}_y = \mathcal{S}_x$ then give reward c to Left and reward $-c$ to Right. Otherwise give reward $-c$ to Left and reward c to Right. Set $\mathcal{S}_7 \leftarrow c$ (surprise rewards become visible to the system in the form of inputs).

Comment: arguments x, y are indexed by $IP+1, IP+2, IP+3, IP+4$, while arguments c, d are chosen according to module columns indexed by $IP+5$. See Section 2 for Bet!*'s interpretation.*

A.3.5 SSA-enabling Instructions

As long as *BlockSSA*Left = TRUE, Left will be protected from evaluations and restorations by SSA. This is important to allow for (1) complex learning processes computing not just one but many correlated modifications of Left-columns, and (2) reward delays: using the EnableSSALeft() instruction below, Left can influence when to trigger evaluations and when to wait for delayed reward.

EnableSSALeft(x_1)**:** If $x_1 < 10$ (the value 10 is chosen arbitrarily) then set *BlockSSA*Left = FALSE.

EnableSSARight(x_1)**:** analogous to EnableSSALeft.

A.3.6 Primitive Learning Algorithms

All the module-modifying LIs below save the module columns they change on a stack. The first Left-modifying instruction executed after some EnableSSALeft() instruction will start a new block of Left-modifications, to be ended by the next EnableSSALeft(). Analogously for Right.

IncProbLeft(x_1, x_2, y_1)**:**

1. Call SSALeft().

2. If there is not already some $(x, .)$ entry above the most recent *(checkpoint, RL(checkpoint))* pair in StackLEFT, then push the pair (x, LEFT_x) onto StackLEFT.

3. Set $\text{LEFT}_{x,k} \leftarrow \lambda\text{LEFT}_{x,k}\ \forall k \neq y_1$. Set $\text{LEFT}_{x,y_1} \leftarrow 1 - \lambda(1 - \text{LEFT}_{x,y_1})$.

Here $0 < \lambda < 1$ is a real-valued constant. In the experiments I arbitrarily use $\lambda = 0.3$. Note that step **3.** includes renormalization.

IncProbLEFT has no effect though if the corresponding module modifications would lead to at least one LEFT value below *MinProb*, a small positive real value (I use $MinProb = 0.004$).

*Comment: IncProb*LEFT *is an LI that permits* LEFT *to modify itself (provided* RIGHT *agrees). IncProb*LEFT *instructions may be used in conjunction with other instructions to form complex probabilistic learning algorithms (running between subsequent checkpoints).*

DecProbLEFT(x_1, x_2, y_1)**:** like IncProbLEFT, but step **3.** is different:

3. $\text{LEFT}_{x,d} \leftarrow \lambda\text{LEFT}_{x,y_1}; \forall k \neq y_1 : \text{LEFT}_{x,k} \leftarrow \frac{1-\lambda\text{LEFT}_{x,k}}{1-\text{LEFT}_{x,k}}\text{LEFT}_{x,k}$.

MoveDistLEFT(x_1, x_2, y_1, y_2)**:** like IncProbLEFT, but step **3.** is different:

3. $\text{LEFT}_x \leftarrow \text{LEFT}_y$.

IncProbRIGHT(x_1, x_2, y_1)**:** analogous to IncProbLEFT.

DecProbRIGHT(x_1, x_2, y_1)**:** analogous to DecProbLEFT.

MoveDistRIGHT(x_1, x_2, y_1, y_2)**:** analogous to MoveDistLEFT.

IncProbBOTH(x_1, x_2, y_1)**:** call IncProbLEFT and IncProbRIGHT in random order.

DecProbBOTH(x_1, x_2, y_1)**:** call DecProbLEFT and DecProbRIGHT in random order.

SSAandCopy(x_1)**:** If $x_1 \geq 5$ then exit (the value 5 is chosen arbitrarily). Set *BlockSSA*LEFT and *BlockSSA*RIGHT = FALSE. Call SSALEFT() and SSARIGHT() in random order. Test if one of the modules has received more reward per time (since the most recent checkpoint still in its stack) than the other. If so:

Find those columns in the superior module that differ from the corresponding columns in the "loser." (In my implementation this is done efficiently by using a separate stack and a marker array tracing module differences as they occur.) Push the loser's different columns onto the loser's stack (just like with IncProbLEFT and all other LIs). Then copy the winner's different columns onto the loser's.

Comment: the LI SSAandCopy() allows for ending "unfair" matches in the case one module consistently outperforms the other. SSAandCopy will make both modules identical, although their stacks will in general be quite different and reflect quite different histories of successful module modifications.

A.3.7 Basic Cycle Modification

For didactic reasons I wait until the end of this appendix to introduce a slight change in the basic cycle's instruction selection procedure (compare Section A.1). In case i addresses an instruction head as opposed to an argument, redefine

$$Q(i,j) = \frac{f(\text{RIGHT}_{i,j}, \text{LEFT}_{i,g(j)})}{\sum_k f(\text{RIGHT}_{i,k}, \text{LEFT}_{i,g(k)})},$$

$$i \in \{0, BS, 2BS, \ldots\}, j \in \{0, \ldots, n-1\}.$$

If a_i is IncProbRIGHT then $g(i)$ returns the index of a_i's *antagonistic* instruction head IncProbLEFT. Similarly for the other pairs of antagonistic instruction heads: (DecProbRIGHT, DecProbLEFT), (MoveDistRIGHT, MoveDistLEFT), (GetRIGHT, GetLEFT), (EnableSSARIGHT, EnableSSALEFT). This is necessary because antagonistic instructions require special treatment to achieve module symmetry through *SSAandCopy*. For instance, suppose that LEFT's current advantage depends on supporting some IncProbRIGHT instruction. An equal RIGHT opponent should strongly support IncProbLEFT instead.

References

[1] E. B. Baum and I. Durdanovic. Toward a model of mind as an economy of agents. *Machine Learning*, 35(2):155–185, 1999.

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

[3] A. C. Clarke. *The ghost from the grand banks.* Orbit books, 1991.

[4] D. A. Cohn. Neural network exploration using optimal experiment design. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 679–686. Morgan Kaufmann, 1994.

[5] V. V. Fedorov. *Theory of optimal experiments.* Academic Press, 1972.

[6] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58, 1992.

[7] D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 313–324. Addison Wesley, 1992.

[8] S. Hochreiter and J. Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[9] J. H. Holland. Properties of the bucket brigade. In *Proceedings of an International Conference on Genetic Algorithms*. Lawrence Erlbaum, Hillsdale, NJ, 1985.

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

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

[12] I. Kwee, M. Hutter, and J. Schmidhuber. Market-based reinforcement learning in partially observable worlds. *Proceedings of the International Conference on Artificial Neural Networks (ICANN-2001), in press*, (IDSIA-10-01, cs.AI/0105025), 2001.

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

[14] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications (2nd edition)*. Springer, 1997.

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

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

[17] F. Nake. *Ästhetik als Informationsverarbeitung*. Springer, 1974.

[18] M. Plutowski, G. Cottrell, and H. White. Learning Mackey-Glass from 25 examples, plus or minus 2. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 1135–1142. Morgan Kaufmann, 1994.

[19] J. B. Pollack and A. D. Blair. Why did TD-Gammon work? In M. C. Mozer, M. I. Jordan, and S. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 10–16. MIT Press, 1997.

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

[21] J. Schmidhuber. Evolutionary principles in self-referential learning. Diploma thesis, Institut für Informatik, Technische Universität München, 1987.

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

[23] J. Schmidhuber. Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE press, 1991.

[24] J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In J. A. Meyer and S. W. Wilson, editors, *Proc. of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227. MIT Press/Bradford Books, 1991.

[25] J. Schmidhuber. Learning factorial codes by predictability minimization. *Neural Computation*, 4(6):863–879, 1992.

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

[27] J. Schmidhuber. Low-complexity art. *Leonardo, Journal of the International Society for the Arts, Sciences, and Technology*, 30(2):97–103, 1997.

[28] J. Schmidhuber. What's interesting? Technical Report IDSIA-35-97, IDSIA, 1997. ftp://ftp.idsia.ch/pub/juergen/interest.ps.gz; extended abstract in Proc. Snowbird'98, Utah, 1998.

[29] J. Schmidhuber. Facial beauty and fractal geometry, 1998. Published in the Cogprint Archive: http://cogprints.soton.ac.uk.

[30] J. Schmidhuber. Artificial curiosity based on discovering novel algorithmic predictability through coevolution. In P. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and Z. Zalzala, editors, *Congress on Evolutionary Computation*, pages 1612–1618. IEEE Press, 1999.

[31] J. Schmidhuber. A general method for incremental self-improvement and multi-agent learning. In X. Yao, editor, *Evolutionary Computation: Theory and Applications*, pages 81–123. World Scientific, 1999.

[32] J. Schmidhuber, M. Eldracher, and B. Foltin. Semilinear predictability minimization produces well-known feature detectors. *Neural Computation*, 8(4):773–786, 1996.

[33] J. Schmidhuber and D. Prelinger. Discovering predictable classifications. *Neural Computation*, 5(4):625–635, 1993.

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

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

[36] N. Schraudolph and T. J. Sejnowski. Unsupervised discrimination of clustered data via optimization of binary information gain. In Stephen José Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 499–506. Morgan Kaufmann, San Mateo, 1993.

[37] Nicol N. Schraudolph, Martin Eldracher, and Jürgen Schmidhuber. Processing images by semi-linear predictability minimization. *Network: Computation in Neural Systems*, 10(2):133–169, 1999.

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

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

[40] J. Storck, S. Hochreiter, and J. Schmidhuber. Reinforcement driven information acquisition in non-deterministic environments. In *Proceedings of the International Conference on Artificial Neural Networks, Paris*, volume 2, pages 159–164. EC2 & Cie, 1995.

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

[42] G. Weiss. Hierarchical chunking in classifier systems. In *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 2, pages 1335–1340. AAAI Press/The MIT Press, 1994.

[43] G. Weiss and S. Sen, editors. *Adaption and Learning in Multi-Agent Systems.* LNAI 1042, Springer, 1996.

[44] S.W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2:1–18, 1994.

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

[46] D. H. Wolpert, K. Tumer, and J. Frank. Using collective intelligence to route internet traffic. In M. Kearns, S. A. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems 12*. MIT Press, 1999.

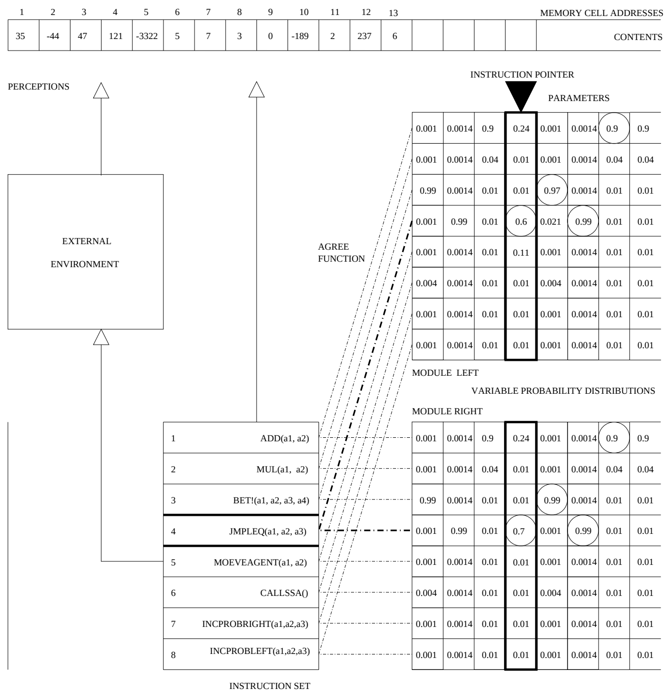

Figure 1: Snapshot of parts of two policies and some memory cells (which are viewed as part of the policy environment). Each policy is a set of variable probability distribution on n_{ops} possible instructions or parameters. For simplicity, in this hypothetical example, $n_{ops} = 8$, and there is only one output instruction for manipulating the external environment, which in turn may affect memory cells (active perception). Instruction pointer IP currently points to a particular pair of distributions which collectively determine the probability of selecting a particular instruction (here: JMPLEQ whose probability is high). JMPLEQ requires three parameters generated according to the subsequent probability distributions.

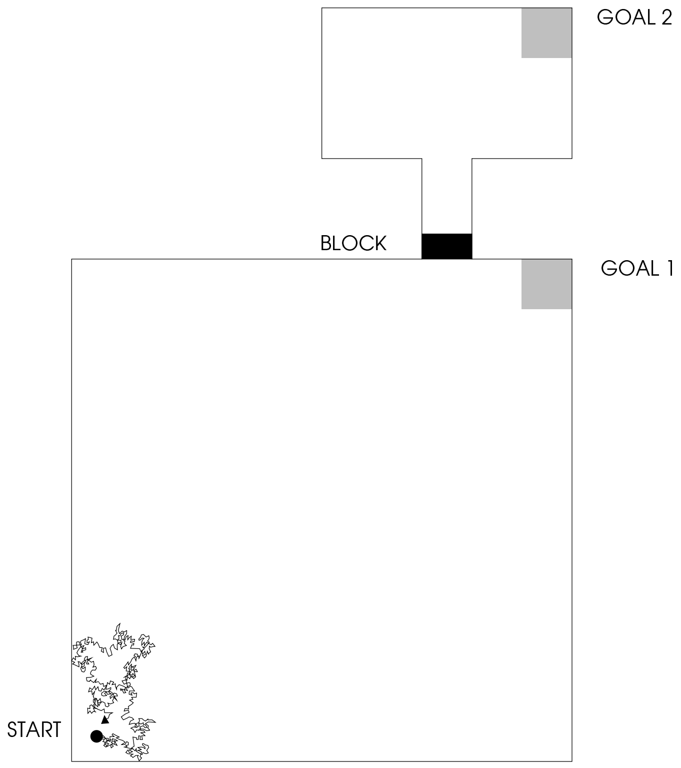

Figure 2: Hypothetical trace of the agent near START in the southwest corner. The width of the environment is 1000 unit lengths, the agent's stepsize 12 unit lengths. Whenever the agent hits GOAL1 or GOAL2 it gets teleported back to START (except in Experiment 1). GOAL2 provides 10 times as much external reward as GOAL1. The entrance to the small room is blocked in Experiments 1 and 2a, and open in 2b. Due to the tiny step size and numerous non-goal-specific instructions, random behavior requires millions of time steps to hit one of the goals by accident.

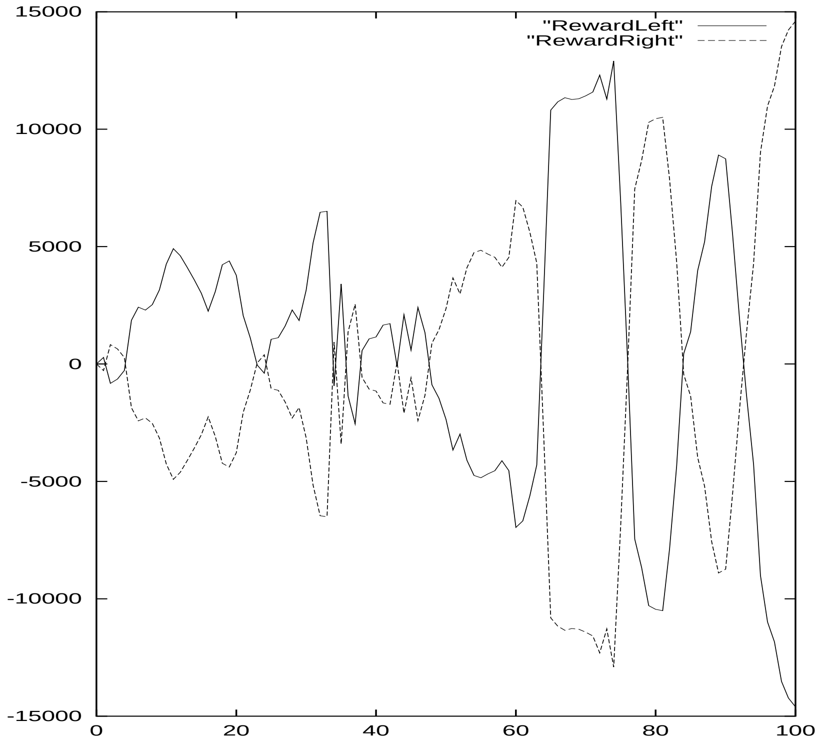

Figure 3: Experiment 1 (no external rewards): symmetric evolution of both modules' cumulative surprise rewards during the first billion time steps, sampled at intervals of 10 million steps.

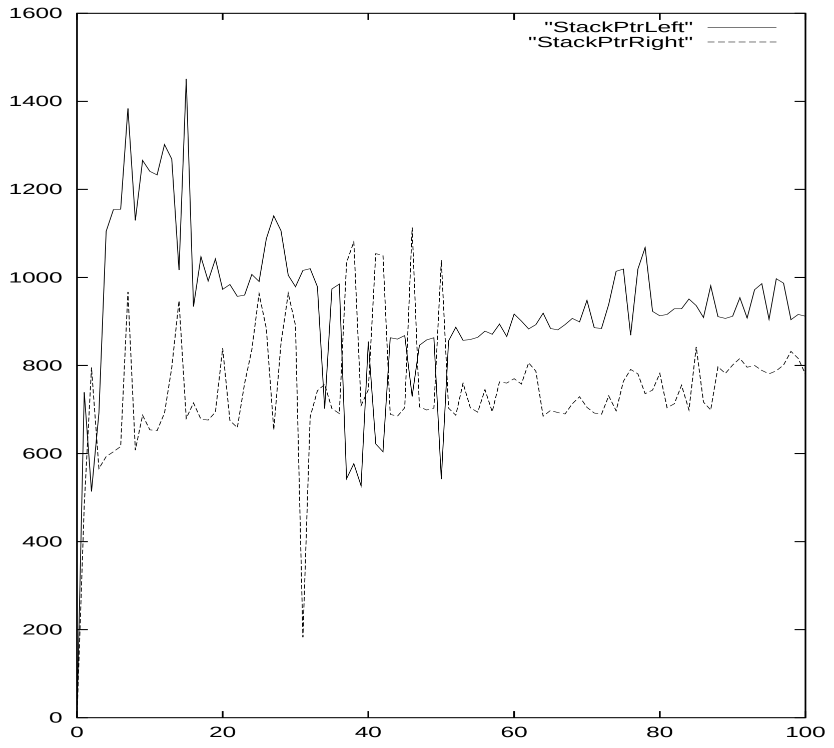

Figure 4: Experiment 1: evolution of both modules' stack pointers during the first billion time steps, measured at intervals of 10 million steps.

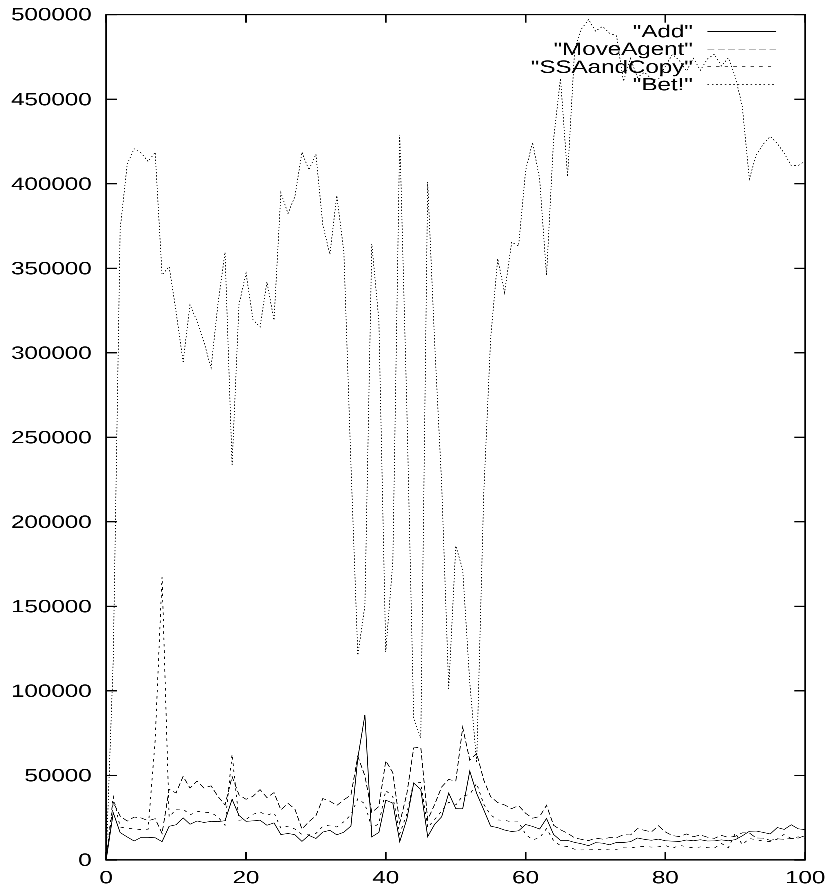

Figure 5: Experiment 1: execution frequencies of selected instruction types during the first billion time steps, sampled at intervals of 10 million steps.

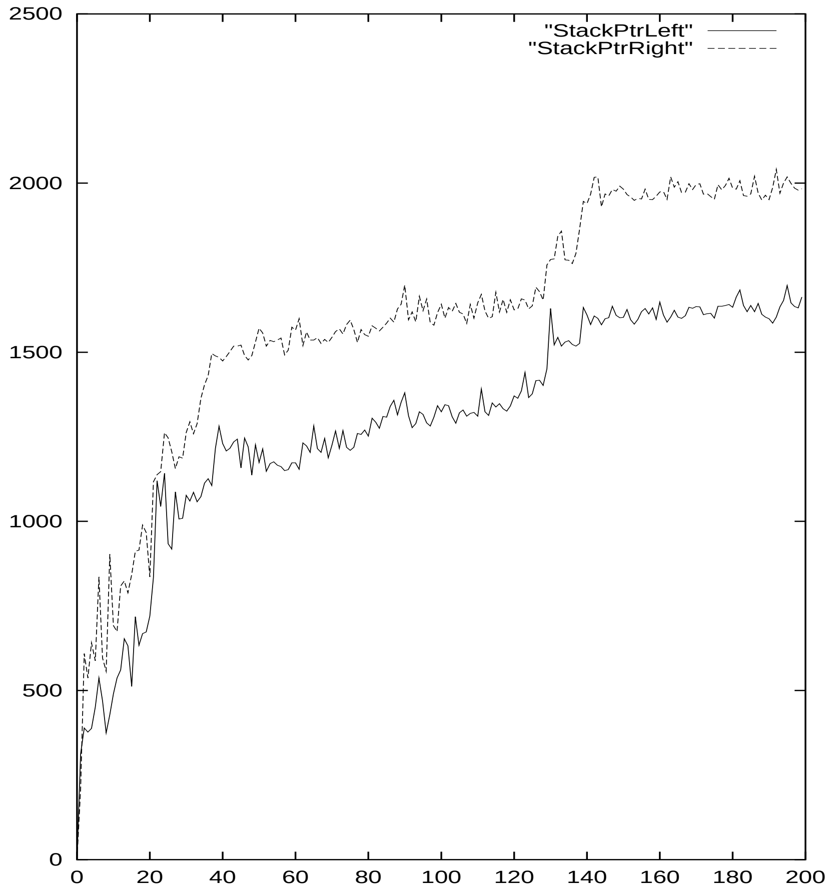

Figure 6: Experiment 2a (additional external reward): stack pointer evolutions during the first 200 million time steps of the particularly successful simulation 1, sampled at million step intervals. Compare Figures 7 and 8.

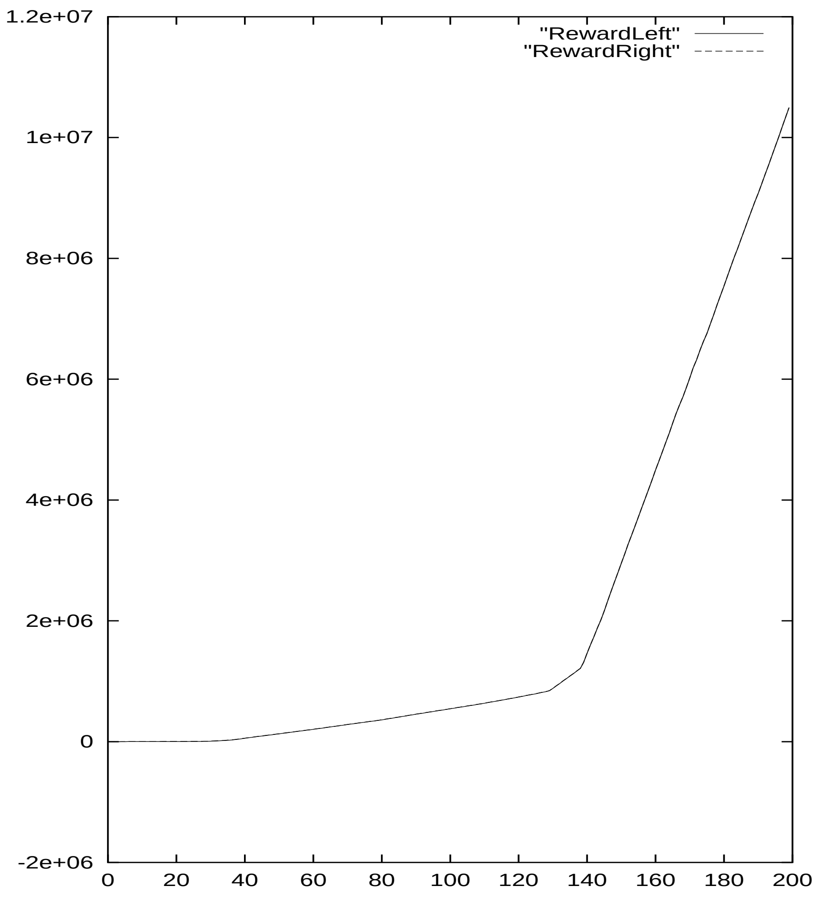

Figure 7: Experiment 2a: evolution of both modules' cumulative rewards during the first 200 million time steps of simulation 1, sampled at million step intervals. Plot resolution is too low to show that both curves are not identical — fluctuations due to surprise rewards seem negligible. Still, surprise rewards seem essential for significant performance improvement — compare Table 1. The final breakthrough occurs around 140 million steps. Compare Figures 6 and 8.

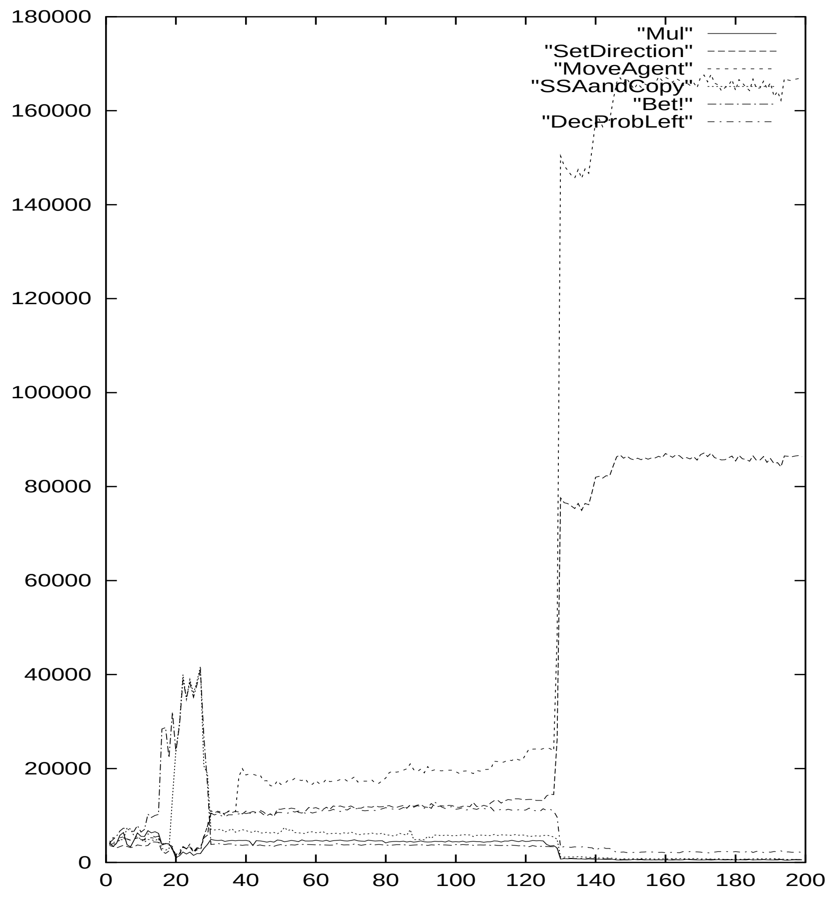

Figure 8: Experiment 2a: execution frequencies of selected instruction types during the first 200 million time steps of simulation 1, sampled at million time step intervals. Between 20 and 30 million steps there are many *Bet!* and *SSAandCopy* instructions. Soon afterwards *Move* and *SetDirection* start to dominate. The final breakthrough is achieved 10 million steps after their frequencies increase dramatically around 130 million steps. Compare Figures 6 and 7.

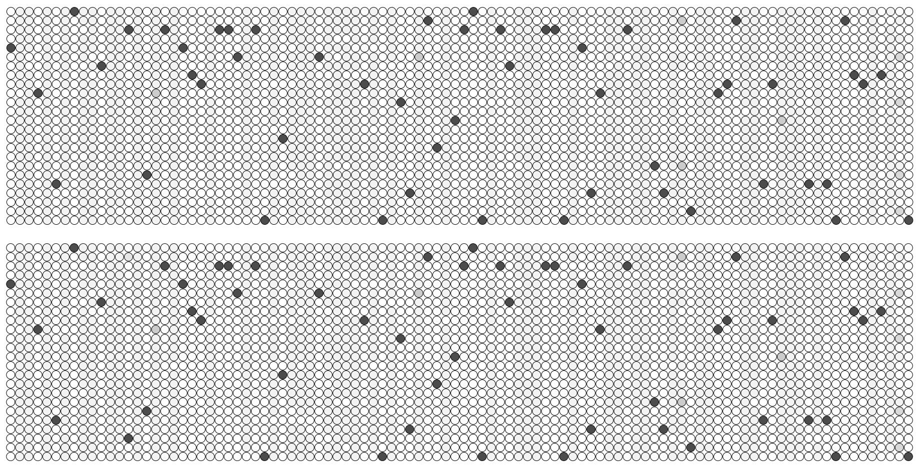

Figure 9: Experiment 2a: LEFT's (top) and RIGHT's first 100 (of 576) probability distributions after simulation 1. Grey scales indicate probability magnitudes (white = close to 0, black = close to 1). The probability mass of many (but not all) columns is concentrated in a single value. Both modules are almost identical due to *SSAandCopy* LIs. Their stacks are quite different though.

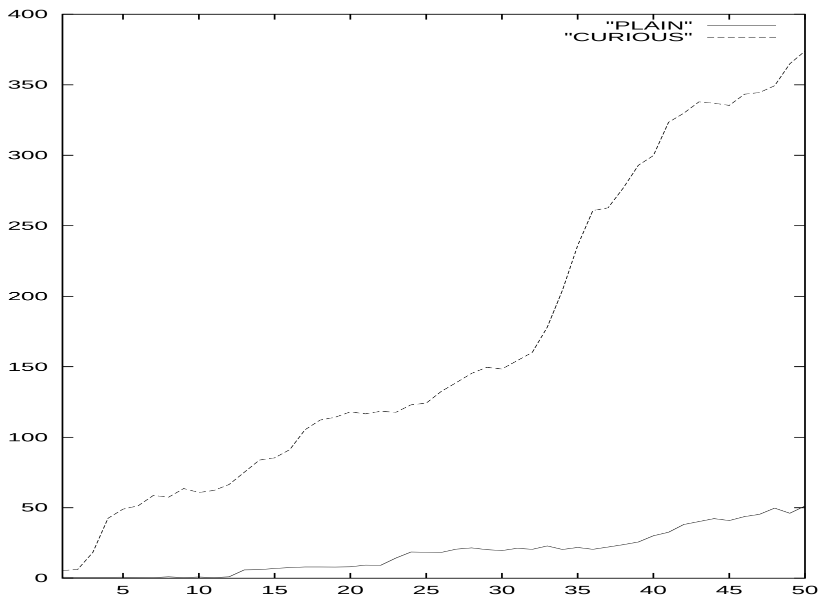

Figure 10: Experiment 2b: comparison of plain and curious systems. The plot shows visits to GOAL2 during the first 500 million time steps, sampled at intervals of 10^7 steps (averaged over 10 simulations; random behavior yields less than one visit per 10^7 steps). Compare Figure 11.

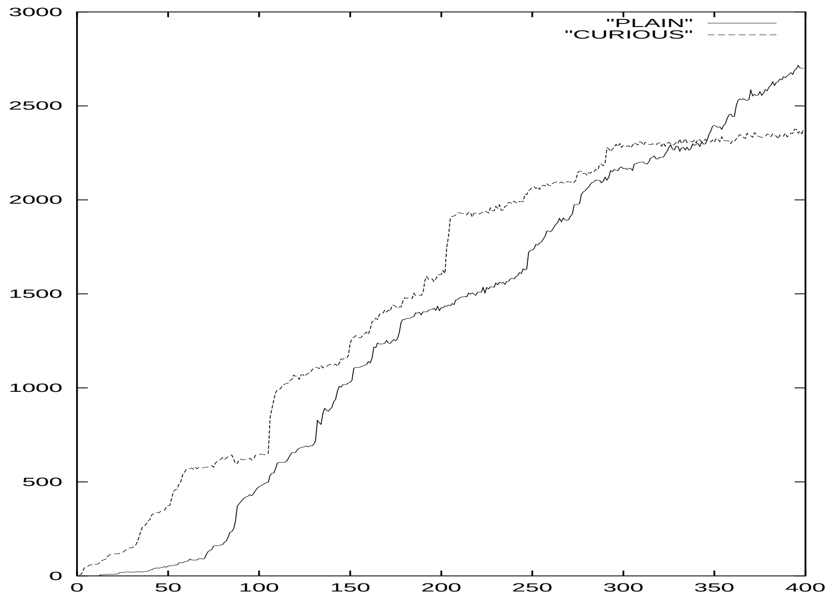

Figure 11: Experiment 2b: visits to GOAL2 during the entire 4 billion time steps, sampled at intervals of 10^7 steps (averaged over 10 simulations). Initial advantages of curiosity are lost as more and more goal-oriented training examples become available. Compare Figure 10.