Bias-Optimal Incremental Learning of Control Sequences for Virtual Robots

Jürgen Schmidhuber & Viktor Zhumatiy [*] & Matteo Gagliolo
IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland
juergen@idsia.ch - viktor@idsia.ch - matteo@idsia.ch

Abstract. Learning and planning control is hard. The search space of traditional planners consists of sequences of primitive actions. To exploit reusable subsequences and other algorithmic regularities, however, we should instead search the general space of *programs that compute* action sequences. Such programs may invoke very fast ”thinking actions” consuming only nanoseconds (such as conditional jumps to certain code addresses) as well as very slow control actions consuming seconds in the real world (such as *stretch-arm-until-obstacle-sensation*). What is an optimal way of allocating time to tests of such non-homogeneous programs? What is an optimal way of reusing experience with previous tasks to learn solutions to new tasks? One answer is given by the recent *Optimal Ordered Problem Solver* OOPS, a near-bias-optimal *incremental* extension of Levin's *non*incremental universal search, which we apply to virtual robotics for the first time: our snake robot uses OOPS to learn to walk and jump in a partially observable environment (POMDP) with a huge state/action space.

1 Introduction

Traditional AI planning procedures [11] do not learn but systematically explore all possible combinations of task-specific primitive actions. Due to the exploding search space they fail to solve problems such as Towers of Hanoi with n disks (solution size $2^n - 1$) for solution sizes $> 10,000$ moves (Jana Koehler, IBM Research, personal communication, 2002). In particular, they do not exploit algorithmic regularities in the solution space — they cannot find the simple double-recursive program which solves Towers of Hanoi for arbitrary n.

Traditional reinforcement learners (RLs) [6] often perform even worse than AI planners. For example, Anderson's pioneering RL[1] did not solve Hanoi problems requiring more than 7 moves . The problems of traditional RLs in realistic control domains are due not only to large state/action spaces and lack of exploitation of algorithmic regularities, but also to partial observability, which is almost inescapable in the real world.

So what is a good way of searching in program space as opposed to raw solution space? The control programs of Karl Sims [15] were recurrent neural networks which he *evolved* in parallel with his interesting artificial creatures. The related, evolution-based Genetic Programming (GP) [4, 2] is also sometimes viewed as a step in the right direction. Most existing GP implementations, however have very limited search spaces and do not even allow for programs with loops and recursion, thus ignoring a main motivation for searching in program

[*]This research is partly sponsored by grant of CSEM (Centre Suisse d'Electronique et de Microtechnique).

space. Neither the existing evolvers of recurrent networks nor the few GP implementations that do permit recursion etc. have a principled way of allocating time to program tests. This limits them to programs whose (typically small) runtimes have upper bounds. Finally, both GP and network evolvers have quite limited ways of making new programs from previous ones—they do not learn better program-making strategies [2].

The recent *Optimal Ordered Problem Solver* OOPS [12, 13] is an incremental program searcher that tries to overcome these limitations by extending Levin's **non**incremental universal search [7]. OOPS solves one task after another, efficiently searching the space of programs that compute solution candidates, including those programs that organize, manage, adapt and reuse earlier acquired knowledge, and even programs that rewrite the search procedure itself. This enables OOPS to incrementally learn to solve Hanoi instances for minimal solution sizes exceeding 10^9 moves [13].

Below we will first review essential background and basic concepts, then we will describe the first application of OOPS to virtual robotics in POMDPs [6] with huge state/action spaces. Robotics is of particular interest as it typically involves very non-homogeneous programs consisting of both fast "thinking actions" (such as those computing the name of the next action) and slow actions in the "real" world (such as executions of robot movements).

2 Optimal Ordered Problem Solver OOPS

Space limits force us to refer the reader to [13] and especially [12] for full formal details. In what follows we have to restrict ourselves to a basic OOPS overview.

A problem r is very broadly defined by a recursive procedure f_r that takes as an input any potential solution (a finite symbol string $q \in C$, where C represents a search space of solution candidates) and outputs 1 if q is a solution to r, and 0 otherwise. Typically the goal is to find as quickly as possible some q that solves r. (Later the search space will be a set of control programs for a robot; the robot's task may be to reach a given physical location.)

Define a prior belief or probability distribution P on a finite or infinite set of programs for a given computer. P represents the searcher's initial bias (e.g., P could be based on program length, or on a probabilistic syntax diagram). A *bias-optimal* searcher will not spend more time on any solution candidate than it deserves, namely, not more than the candidate's probability times the total search time:

Definition 2.1 (BIAS-OPTIMAL SEARCHERS**).** Let $\mathcal{R}$ be a problem class, $\mathcal{C}$ be a search space of solution candidates (where any problem $r \in \mathcal{R}$ should have a solution in $\mathcal{C}$), $P(q \mid r)$ be a task-dependent bias in the form of conditional probability distributions on the candidates $q \in \mathcal{C}$. Suppose that we also have a predefined procedure that creates and tests any given q on any $r \in \mathcal{R}$ within time $t(q, r)$ (typically unknown in advance). Then *a searcher is n-bias-optimal ($n \geq 1$) if for any maximal total search time $T_{total} > 0$ it is guaranteed to solve any problem $r \in \mathcal{R}$ if it has a solution $p \in \mathcal{C}$ satisfying $t(p, r) \leq P(p \mid r)\ T_{total}/n$. It is bias-optimal if $n = 1$.*

Therefore the most probable candidates should get the lion's share of the total search time, in a way that precisely reflects the initial bias.

How OOPS solves the first task. OOPS is fed a sequence of tasks. The first task is solved by invoking near-bias-optimal LSEARCH as follows (we notationally suppress the conditional dependency on the current task); compare [7, 16, 14, 9, 5]:

Method 2.1 (LSEARCH). *In the i-th phase $(i = 1, 2, 3, \ldots)$ test all programs q with runtime (including testing time) $\leq T \cdot P(q)$ (where $T := 2^i$) until the task is solved.*

Note that the method is the *asymptotically fastest method:* Given some problem class, if some unknown optimal program p requires $f(k)$ steps to solve a problem instance of size k, then LSEARCH will need at most $O(f(k)/P(p)) = O(f(k))$ steps — the constant factor $1/P(p)$ does not depend on k.

Time-optimal backtracking for resource-limited computers. While LSEARCH was originally described for universal Turing machines with unlimited storage, OOPS actually invokes a recursive procedure [12, 13] for time-optimal planning and backtracking in program space to perform efficient storage management on *realistic, limited computers.* This procedure essentially conducts a depth-first search in program space. The branches of the search tree are program prefixes modifying some internal storage holding the current state of network and environment. Backtracking (partial resets of state modifications) is triggered once the runtime of the current prefix on the current task exceeds the current time limit multiplied by the prefix probability. That is, during program runtime we need to save parts of the intermediate storage states on a stack such that backtracking can restore them when necessary. See [13, 12] for details.

OOPS also allows for the possibility that the current task actually is a *set of tasks* to be solved simultaneously by the same program, with separate task-specific storages or *tapes*.

OOPS vs plain Lsearch. Note that LSEARCH by itself neglects one potential source of speed-up: it is nonincremental in the sense that it does not attempt to minimize its constant slowdown factor $O(1/P(p))$ by exploiting experience collected in previous searches for solutions to earlier tasks. It simply ignores the constant — from an asymptotic point of view, incremental search does not buy anything. OOPS, however, solves subsequent tasks by using experience with previous tasks to greatly reduce these constants in a bias-optimal way, as will be seen next.

How OOPS solves the remaining tasks. Once the first task set is solved we try to solve the next. Unlike plain LSEARCH, OOPS will time-optimally reduce the ominous constant factor by reusing solutions to previous problems in computable ways, wherever this is profitable. This can greatly accelerate the search for new solutions [12, 13]. The basic ideas are as follows.

Freezing and incremental search. Whenever the current task or task set is solved, the corresponding program gets frozen (its storage space becomes non-writeable) such that it can be copy-edited and/or invoked as a subprogram by programs tested in later program searches.

Metalearning. We also allow program prefixes to temporarily rewrite the probability distribution on their suffixes, thus essentially rewriting the search procedure itself based on previous experience. That is, we metasearch for better search procedures with better constant factors. This can greatly accelerate the learning of new tasks [13, 12].

How OOPS spends its time. Given a new task, OOPS spends half the total search time on a variant of LSEARCH that searches only among self-delimiting [8, 3] programs starting with the most recently frozen code. It turns out that it is sufficient here to test only on the new task, *never* on previous tasks, even when we are looking for a universal solver for all tasks in the sequence. The rest of the time is spent on fresh programs starting right after the most recently frozen code. When we are looking for a universal solver, however, we must test such *fresh* programs on *all* previous tasks [12, 13].

Optimality of OOPS. OOPS is essentially *8-bias-optimal* (see Def. 2.1), given the initial bias and frozen solutions to previous tasks [12, 13]. Therefore OOPS can solve difficult tasks unsolvable by traditional reinforcement learners and AI planners [13].

Summary. OOPS essentially allocates part of the total search time for a new problem to programs that exploit previous solutions in computable ways. If the new problem can be solved faster by copy-editing / invoking previous code than by solving the new problem from scratch, then OOPS will discover this and profit thereof. If not, then at least it will not be significantly slowed down by the previous solutions—OOPS will remain 8-bias-optimal.

3 Applications to Snake Robots

We connected OOPS to the VORTEX[1] (tm) 3D physics simulation environment. VORTEX allows for realistic physics modelling, including gravity, friction, slipping and collisions.

Consider Figure 1. Initially a snake-like robot hovers 1*m* (in z direction) above an infinite (x,y) plane, with its tail (smallest x coordinate) right above the origin (0,0). It consists of three segments, each 4*m* long, separated by 1*m*, connected by two virtual hinge-like actuators, both working within the same plane (initially perpendicular to the ground). The snake's current position is defined as the position of the nose tip. By applying forces to its actuators the snake can modify its shape and try to move. The continous *state/action space is huge* when compared to those of many previous reinforcement learning test problems [6]. The space of parameterized control actions is very large as well (see Section 3.1). The snake robot's n-th task ($n = 1, 2, \ldots$) is:

a. After being released, land on the ground without tumbling aside (the centers of mass of the snake segments are not perfectly aligned, hence inaccurate movements may topple the snake and render it helpless).

b. Crawl / jump / move at least a distance of $2 \cdot (1m + d_{best})$ away from the origin, where d_{best} is the length of the longest walk so far.

We stop creating new tasks as soon as the snake has covered a distance of 100*m*.

3.1 Language / Instructions

For the experiments we wrote a miniature multi-tasking operating system and an interpreter for an exemplary, stack-based, universal programming language inspired by FORTH [10], whose disciples praise its beauty and the compactness of its programs. In fact, to demonstrate the system's versatility, we used exactly the same set-up that was used for quite different tasks described recently [12, 13]. Since we are dealing with an incremental optimization task we need only one task-specific storage or tape. The tape holds various modifiable stack-like data structures represented as sequences of integers, including a data stack *ds* (with stack pointer *dp*) for function arguments, an auxiliary data stack *Ds*, a function stack *fns* of entries describing (possibly recursive) functions defined by the system itself, a callstack *cs* (with stack pointer *cp* and top entry $cs[cp]$) for calling functions, where the local variable $cs[cp].ip$ is the current *instruction pointer,* and the *base pointer* $cs[cp].dp$ points into *ds* below the values considered as arguments of the most recent function call: Any instruction of the form *inst*

[1]VORTEX (tm) is a product of CMLabs Simulations.

($x_1, \ldots, x_n$) expects its n arguments on top of *ds*, and replaces them by its return values. Illegal use of any instruction will cause the currently tested program prefix to halt. In particular, it is illegal to set variables (such as stack pointers or instruction pointers) to values outside their prewired ranges, or to pop empty stacks, or to divide by 0, or to call nonexistent functions, or to change probabilities of nonexistent tokens. OOPS *will interrupt prefixes as soon as their runtime exceeds $T \cdot P$ (see Method 2.1).*

General Instructions. We defined more than 80 instructions of a general purpose language, essentially the same as those used for quite different tasks [13, 12]. Instructions include *oldq(n)* for calling the n-th previously found program q^n, or *getq(n)* for making a copy of q^n on stack *ds* (e.g., to edit it with additional instructions). Lack of space prohibits an explanation of all the instructions — see [12]. Here we have to limit ourselves to a few, especially those appearing in solutions found in the experiments, using readable names instead of the numbers that represent them [12, 13]: Instruction *c1()* returns constant 1. Similarly for *c2(), ..., c9(). m1(), ..., m9()* return negative constants *-1,...,-9. pip(n)* pops the next ip value n from stack; *jmp1(c, n)* pops two values c, n and sets instruction pointer *ip* equal to n if $c > 0$ (conditional jump); *add(x,y)* returns $x + y$; *sub(x,y)* returns $x - y$; *dec(x)* returns $x - 1$; *neg(x)* returns $-x$; *by2(x)* returns $2x$; *grt(x,y)* returns 1 if $x > y$, otherwise 0; *cpn(n)* copies the n topmost *ds* entries onto the top of *ds*, increasing *dp* by n; *cpnb(n)* copies n *ds* entries above the $cs[cp].dp$-th *ds* entry onto the top of *ds*; *exec(n)* interprets n as the number of an instruction and executes it; *bsf(n)* considers the entries on stack *ds* above its $cs[cp].dp + n$-th entry as code and uses callstack *cs* to *call* this code (OOPS executes code one instruction at a time; the instruction *ret()* causes a return to the address of the next instruction right after the calling instruction). Given n input arguments on *ds*, instruction *defnp()* pushes onto *ds* the beginning of a definition of a procedure with n inputs; this procedure returns if its topmost input is 0, and otherwise decrements it. *calltp()* pushes onto *ds* code for a call of the most recently self-defined function / procedure. Both *defnp* and *callp* also push code that makes a fresh copy of the inputs of the most recently defined code, expected on top of *ds*. *endnp()* pushes code for returning from the current call, then *calls* the code generated so far on stack *ds* above the n inputs, applying the code to a copy of the inputs on top of *ds*. All instructions between two *qot* are pushed on to the data stack, without being executed. *boostq(i)* sequentially goes through all tokens of the i-th self-discovered frozen program, *boosting* each token's probability by adding n_Q to its enumerator and also to the denominator shared by all instruction probabilities—*the denominator and all numerators are stored on the tape and can be modified by the currently tested prefix, which can thus rewrite the search procedure on its suffixes (*metasearching*).*

Snake-specific instructions. Now we add domain-specific instructions. The instruction *ApplyForces (m,n)* pops two numbers from the top of *ds* and interprets them as forces for the snake's two actuators. The forces are applied in the VORTEX 3D physics simulation; then we wait until the snake has stopped to move (checked via internal VORTEX flag) or until OOPS interrupts because the current prefix test has already consumed too much time.

Since the actuator space is 2-dimensional, we also add instructions for dealing with 2-dimensional vector data: *mul2* (x_1, x_2, y_1, y_2) pops 4 values x_1, x_2, y_1, y_2 off ds and pushes (returns) 2 values, namely, the products $x_1 x_2$ and $y_1 y_2$. Similarly, *add2* (x_1, x_2, y_1, y_2) returns $(x_1 + x_2, y_1 + y_2)$; *sub2* (x_1, x_2, y_1, y_2) returns $(x_1 - x_2, y_1 - y_2)$; *neg2* (x_1, x_2) returns $(-x_1, -x_2)$, *pushone2* $()$ returns the constant vector $(1, 1)$.

Probabilistic Bias. Of course, OOPS tracks the computation time consumed by all instructions, including the substantial time spent in the realistic physics simulation. For ex-

IP	OC	INSTRUCTION	DATA STACK
38	85	CALLTP	
15	82	QOT	
16	33	TOPF	
17	34	DOF	*33*
18	31	INTPF	*34* 33
19	49	CPN	*31* 34 33
20	82	QOT	*49* 31 34 33
21	37	RET	49 31 34 33
39	1	APPLYFORCES	**49 31** 34 33
40	74	C6	34 33
41	46	JMP1	*6* **34** 33
34	85	CALLTP	33
15	82	QOT	33
16	33	TOPF	33
17	34	DOF	*33* 33
18	31	INTPF	*34* 33 33
19	49	CPN	*31* 34 33 33
20	82	QOT	*49* 31 34 33 33
21	37	RET	49 31 34 33 33
35	1	APPLYFORCES	**49 31** 34 33 33
36	39	NEG	**34** 33 33
37	1	APPLYFORCES	*-34* **33** 33
38	85	CALLTP	33
15	82	QOT	33

TABLE 1: PROGRAM TRACE SHOWING INSTRUCTION OPCODES (OC) AND EVOLVING DATA STACK CONTENTS (RIGHT), AS THE SNAKE SUCCESSFULLY MOVES FORWARD (THE SEPARATE CALL STACK IS NOT SHOWN FOR LACK OF SPACE).

ample, as soon as the current $T \cdot P$ limit is exceeded (see Method 2.1) during some *ApplyForces (m,n)* instruction, OOPS interrupts and backtracks. We assign roughly equal probabilities to equal-sized programs containing at least one *ApplyForces* instruction, and near-zero probability to all other programs. We do this by starting with the maximum entropy distribution on the 84 instructions (all 84 numerators set to 1) and then acting as if the execution of an *ApplyForces* instruction in VORTEX was $V = 10000$ times faster than it really is. Let $pr_{n,k}$ denote a non-empty program with k instructions including n *ApplyForce*'s, then $P(pr_{n,k})/P(pr_{0,k}) = V \cdot n/k + (k-n)/k$. That is, for program size $k \leq 10$ instructions we ensure for $n > 0$ that $P(pr_{n,k}) > 1000 \cdot P(pr_{0,k})$, and for $0 < n_1, n_2$ that $1 \leq P(pr_{n_1,k})/P(pr_{n_2,k}) < 10$.

4 Experimental Results

After one hour on a PC (1.7 GHz Intel XEON) OOPS had solved 7 harder and harder tasks. The solution of the final task moved the snake beyond the 100*m* mark. In the experiment we used the same initial frozen code setup as in the previous work [12, 13].

The solution of the 1st task was found after only 890 program tests: the somewhat bizarre program *(defnp ApplyForces)* happened to move the snake for 2*m* $(T = 262,144)$.

The 2nd task (6*m* walk) was solved by the program prolongation *(defnp ApplyForces neg ApplyForces)*.

The 3rd task (14*m*) was solved much later by a fresh additon to the total code: *(calltp ApplyForces c6 jmp1)* (T=34,359,738,368). This program invokes and reuses parts of the previous code, and successfully solved the remaining tasks: 5th task (30*m*, T=2,048); 6th task (62*m*, T=4,096); 7th task (126*m*, T=8,192).

The complete code seems strange as it relies on the *calltp* instruction originally intended to facilitate the automated editing of code by pushing the code of a subroutine on the data stack. Consider Table 4, where we use **bold font** for instruction arguments and *italic font* for instruction outputs in the next line. Instruction $calltp_{38}$ (subscripts denote instruction addresses) pushes 4 integers onto the stack: it is a pre-wired subroutine (useful for certain

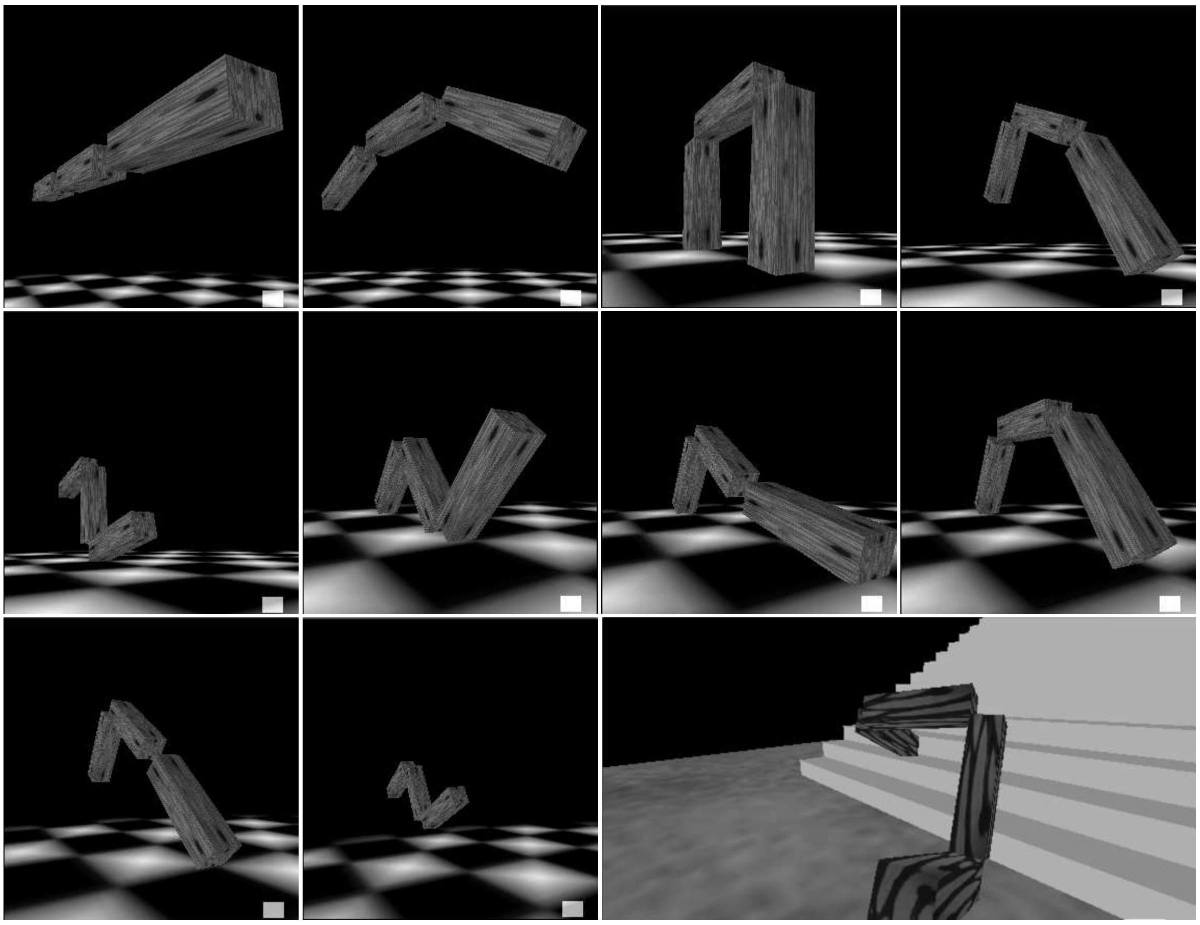

Figure 1: *10 snapshots (ordered like text) of walkinq / jumping snake after learning —compare instruction pointer* ip *in Table 4.* **1:** *Initial position.* **2, 3:** ApplyForces *executed at $ip = 39$ and 35.* **3, 4, 5:** *ip moves up to address 37.* **7, 8:** *until $ip = 39$.* **9, 10:** *correspond to $ip = 35, 37$. The last picture stems from ongoing experiments with a 4-segment snake.*

types of recursive function calls) whose definition is reflected in the trace between qot_{15} and ret_{21}. Then the two topmost stack elements become the input to $ApplyForces_{39}$. The remaining integer and the result of instruction *c6*$_{40}$ become the arguments of the conditional jump *jmp1*$_{41}$ which pops two arguments and interprets the second one as the next address to jump to, since 6 is a positive number. The following call to $calltp_{34}$ and $ApplyForces_{35}$ is like the one just described. It is followed by neg_{36} which forces the snake into a *Z*-like shape via the final $ApplyForces_{37}$, which closes an infinite loop. The first 10 pictures of Figure 1 show snake snapshots corresponding to this trace.

Conclusions

To find desirable trajectories of robot movements we used the Optimal Ordered Problem Solver to search in a very large space of general trajectory-computing programs written in a universal programming language with more than 80 instructions. The programs are highly heterogeneous, involving both very fast control decisions and very slow action execution in a virtual world with a realistic physics simulation and a huge state space. Still, OOPS bias-optimally allocates limited search time to programs computing solution candidates for a sequence of harder and harder tasks, and discovers problem-solving programs that are a bit opaque and difficult to understand but successful.

Of course, the initial bias is crucial, as always. But given *any* initial bias in the form of an initial programming language, OOPS represents a bias-optimal way of solving robotic control and planning tasks. This encourages us in our ongoing research to apply OOPS to more complex robots living in more complex environments. In particular, we are developing biases that are more robot-specific than the one embodied by our current system.

References

[1] C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci., 1986.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1998.

[3] G.J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

[4] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications, Carnegie-Mellon University, July 24-26, 1985*, Hillsdale NJ, 1985. Lawrence Erlbaum Associates.

[5] M. Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002.

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

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

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

[10] C. H. Moore and G. C. Leach. FORTH - a language for interactive computing, 1970. http://www.ultratechnology.com.

[11] S. Russell and P. Norvig. *Artifi cial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[12] J. Schmidhuber. Optimal ordered problem solver. Technical Report IDSIA-12-02, IDSIA, Manno-Lugano, Switzerland, 2002. Available at arXiv:cs.AI/0207097 or http://www.idsia.ch/˜juergen/oops.html. Accepted by *Machine Learning Journal*, Kluwer, 2003.

[13] J. Schmidhuber. Bias-optimal incremental problem solving. In *Advances in Neural Information Processing Systems NIPS 15*. MIT Press, Cambridge, MA, 2003. In press. Available at http://www.nips.cc/NIPS2002/NIPS2002preproceedings/ or http://www.idsia.ch/˜juergen/oops.html.

[14] 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.

[15] Karl Sims. Evolving virtual creatures. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 1994)*, Computer Graphics Proceedings, Annual Conference, pages 15–22. ACM SIGGRAPH, ACM Press, jul 1994. ISBN 0-89791-667-0.

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