Theoretically Optimal Program Induction and Universal Artificial Intelligence

Marcus Hutter

Istituto Dalle Molle di Studi sull'Intelligenza Artificiale
IDSIA, Galleria 2, CH-6928 Manno-Lugano, Switzerland
marcus@idsia.ch, http://www.idsia.ch/~marcus

ICML-2005, August 7

Table of Contents

Abstract

The idea of the 'typing monkeys', one of them eventually producing 'Shakespeare', or, more formally, exhaustive search over programs, is well known and appealing. In this talk I consider completely general parametric problem solving and the more particular universal AI (artificial intelligence) problem from this perspective. The difficulty here is that the right monkey (program) has to be selected algorithmically (rather than selecting it by hand which would add the intelligence of the selector), but the criterion for selection can itself be incomputable or hard to specify.

Keywords: Levin/program/proof search, fastest algorithm, universal artificial intelligence, AIXI(tl) models, optimal ordered problem solver, Gödel machine.

Introduction: Computability and Monkeys

- Let enough monkeys type on typewriters or computers, eventually one of them will write Shakespeare, or solve a hard problem, or write an AI program, etc.
- To pick the right monkey by hand is cheating, as then the intelligence of the selector is added.
- Problem: How to (algorithmically) select the right monkey?
- This talk: Generic solution for the inversion, computation, and AI problem.

The Fastest and Shortest Algorithm for All Well-Defined Problems

Introduction

- Searching for fast algorithms to solve certain problems is a central and difficult task in computer science.
- Positive results usually come from explicit constructions of efficient algorithms for specific problem classes.
- A wide class of problems can be phrased in the following way:
- Find a fast algorithm computing $f:X\to Y$, where f is a formal specification of the problem depending on some parameter x.
- The specification can be formal (logical, mathematical), it need not necessarily be algorithmic.
- Ideally, we would like to have the fastest algorithm, maybe apart from some small constant factor in computation time.

Blum's Speed-up Theorem (Negative Result)

There are problems for which an (incomputable) sequence of speed-improving algorithms (of increasing size) exists, but no fastest algorithm.

[Blum, 1967, 1971]

Levin's Theorem (Positive Result)

Within a (large) constant factor, Levin search is the fastest algorithm to invert a function $g:Y \to X$, if g can be evaluated quickly.

[Levin 1973]

SIMPLE is as fast as SEARCH

- SIMPLE: run all programs $p_1 p_2 p_3 \ldots$ on x one step at a time according to the following scheme: p_1 is run every second step, p_2 every second step in the remaining unused steps, ... if $g(p_k(x)) = x$, then output $p_k(x)$ and halt $\Rightarrow$ $time_{\text{SIMPLE}}(x) \leq 2^k time^+_{p_k}(x) + 2^{k-1}$.
- SEARCH: run all p of length less than i for $2^i 2^{-l(p)}$ steps in phase $i = 1, 2, 3, \ldots$. $time_{\text{SEARCH}}(x) \leq 2^{K(k)+O(1)} time^+_{p_k}(x)$, $\quad K(k) \ll k$.
- Refined analysis: SEARCH itself is an algorithm with some index $k_{\text{SEARCH}} = O(1)$
  $\Longrightarrow$ SIMPLE executes SEARCH every $2^{k_{\text{SEARCH}}}$-th step
  $\Longrightarrow$ $time_{\text{SIMPLE}}(x) \leq 2^{k_{\text{SEARCH}}} time^+_{\text{SEARCH}}(x)$
  $\Longrightarrow$ SIMPLE and SEARCH have the same asymptotics also in k.
- Practice: SEARCH should be favored because the constant $2^{k_{\text{SEARCH}}}$ is rather large.

Main New Result (The Fast Algorithm M_{p^*})

- Let $p^*: X \to Y$ be a given algorithm or specification.
- Let p be any algorithm, computing provably the same function as p^*
- with computation time provably bounded by the function $t_p(x)$.
- $time_{t_p}(x)$ is the time needed to compute the time bound $t_p(x)$.
- Then the algorithm M_{p^*} computes $p^*(x)$ in time

$$time_{M_{p^*}}(x) \;\leq\; 5 \cdot t_p(x) + d_p \cdot time_{t_p}(x) + c_p$$

- with constants c_p and d_p depending on p but not on x.
- Neither p, t_p, nor the proofs need to be known in advance for the construction of $M_{p^*}(x)$.

[H'00]

Applicability

- Prime factorization, graph coloring, truth assignments, ... are Problems suitable for Levin search, if we want to find a solution, since verification is quick.
- Levin search cannot decide the corresponding decision problems.
- Levin search cannot speedup matrix multiplication, since there is no faster method to verify a product than to calculate it.
- Strassen's algorithm p' for $n\times n$ matrix multiplication has time complexity $time_{p'}(x)\leq t_{p'}(x):=c\cdot n^{2.81}$.
- The time-bound function (cast to an integer) can, as in many cases, be computed very fast, $time_{t_{p'}}(x)=O(log^2 n)$.
- Hence, also M_{p^*} is fast, $time_{M_{p^*}}(x)\leq 5c\cdot n^{2.81}+O(log^2 n)$, even without known Strassen's algorithm.
- If there exists an algorithm p'' with $time_{p''}(x)\leq d\cdot n^2 log\, n$, for instance, then we would have $time_{M_{p^*}}(x)\leq 5d\cdot n^2 log\, n+O(1)$.
- Problems: Large constants c, c_p, d_p.

The Fast Algorithm M_{p^*}

$\boxed{M_{p^*}(x)}$

Initialize the shared variables

$L := \{\}, \quad t_{fast} := \infty, \quad p_{fast} := p^*.$

Start algorithms A, B, and C

in parallel with 10%, 10% and 80%

computational resources, respectively.

$\boxed{A}$

Run through all proofs.

`if` a proof proves for some (p,t) that

$p(\cdot)$ is equivalent to (computes) $p^*(\cdot)$

and has time-bound $t(\cdot)$

`then` add (p,t) to L.

$\boxed{B}$

Compute all $t(x)$ in parallel

for all $(p,t) \in L$ with

relative computation time $2^{-\ell(p)-\ell(t)}$.

`if` for some t, $t(x) < t_{fast}$,

`then` $t_{fast} := t(x)$ and $p_{fast} := p$.

`continue`

$\boxed{C}$

`for` k:=1,2,4,8,16,32,... `do`

run current p_{fast} for k steps

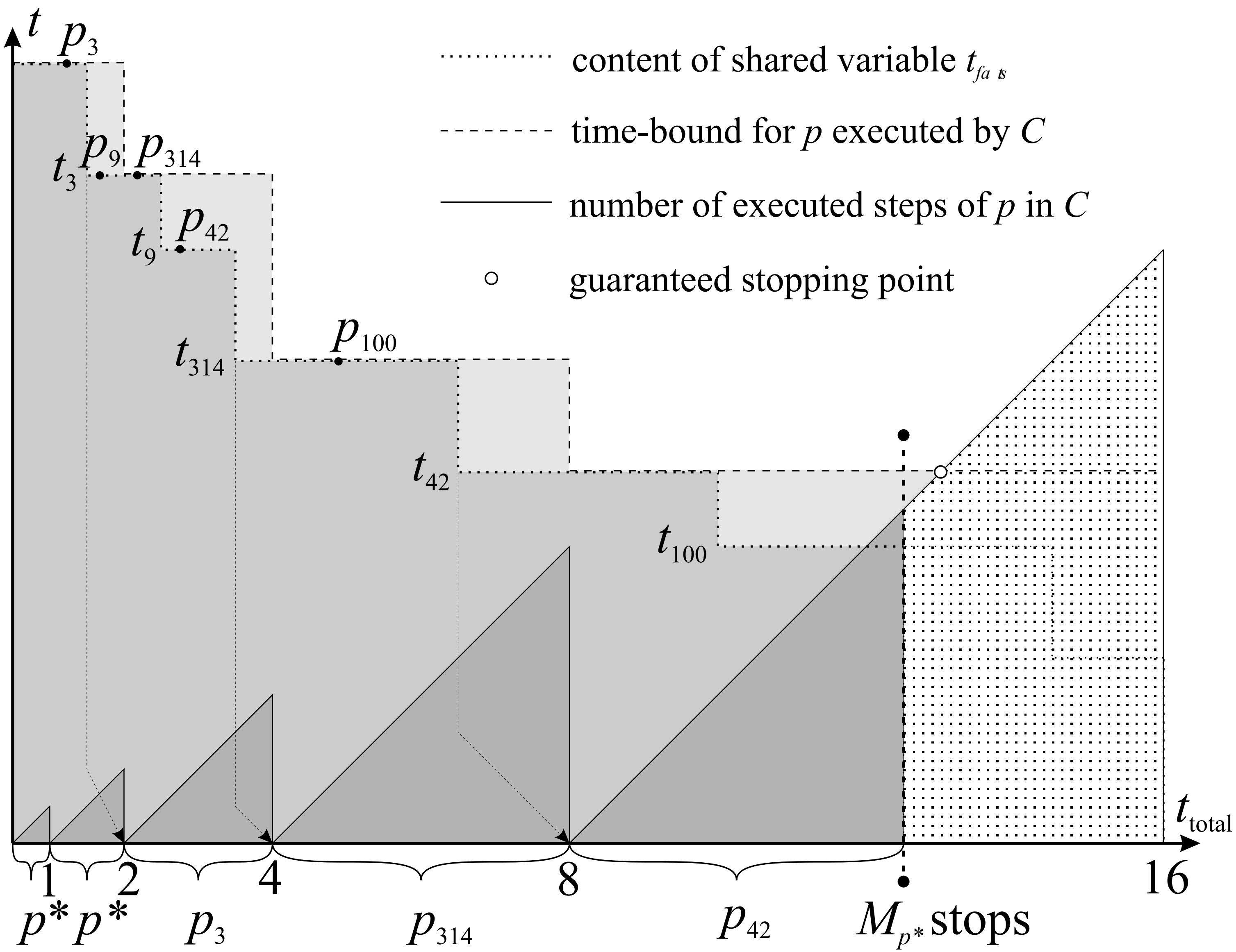
(without switching).

`if` p_{fast} halts in less than k steps,

`then` print result and abort A, B and C.

`else continue` with next k.

Fictitious Sample Execution of M_{p^*}



Time Analysis

$$T_A \leq \frac{1}{10\%} \cdot 2^{\ell(proof(p'))+1} \cdot O(\ell(proof(p'))^2)$$

$$T_B \leq T_A + \frac{1}{10\%} \cdot 2^{\ell(p')+\ell(t_{p'})} \cdot time_{t_{p'}}(x)$$

$$T_C \leq \begin{cases} 4T_B & \text{if } C \text{ stops not using } p' \text{ but on some earlier program} \\ \frac{1}{80\%}4t_{p'} & \text{if } C \text{ computes } p'. \end{cases}$$

$$time_{M_{p^*}}(x) = T_C \;\leq\; 5 \cdot t_p(x) + d_p \cdot time_{t_p}(x) + c_p$$

$$d_p = 40 \cdot 2^{\ell(p)+\ell(t_p)}, \quad c_p = 40 \cdot 2^{\ell(proof(p))+1} \cdot O(\ell(proof(p)^2))$$

Miscellaneous

- Using the shortest algorithm p' provably equivalent to p^* one can show that $M_{p'}$ is the fastest *and* shortest algorithm provably equivalent to p^*.

- The setting can be generalized to repeated p^* evaluation, i/o streams, and time measured in output rather than input length.

- More elaborate theorem-provers could lead to smaller constants.

- Transparent or holographic proofs allow under certain circumstances an exponential speed up for checking proofs [Babai et al. 1991].

- Will the ultimate search for asymptotically fastest programs typically lead to fast or slow programs for arguments of practical size?

Summary

- Under certain provability constraints, M_{p^*} is the asymptotically fastest algorithm for computing p^* apart from a factor 5 in computation time.

- The fastest program computing a certain function is also among the shortest programs provably computing this function.

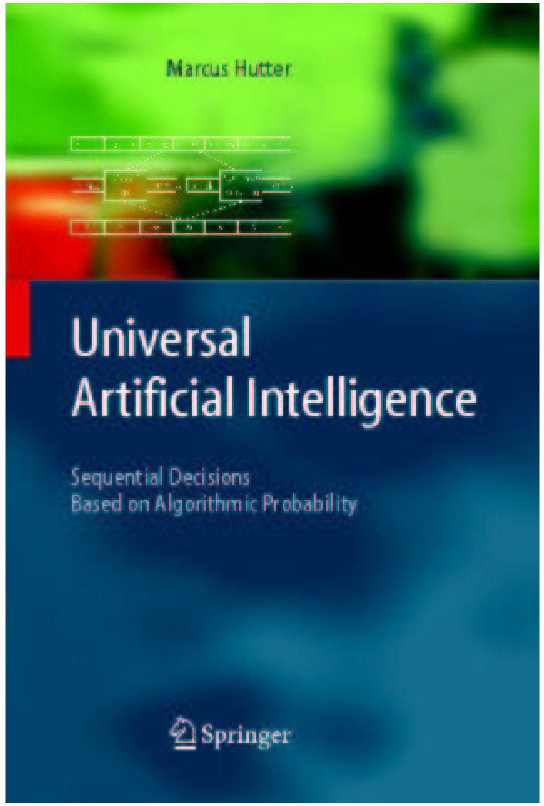
- The large constants c_p and d_p seem to spoil a direct implementation of M_{p^*}.

- On the other hand, Levin search has been successfully extended and applied even though it suffers from a large multiplicative factor [Schmidhuber 1996-2002].

Universal Artificial Intelligence

Overview

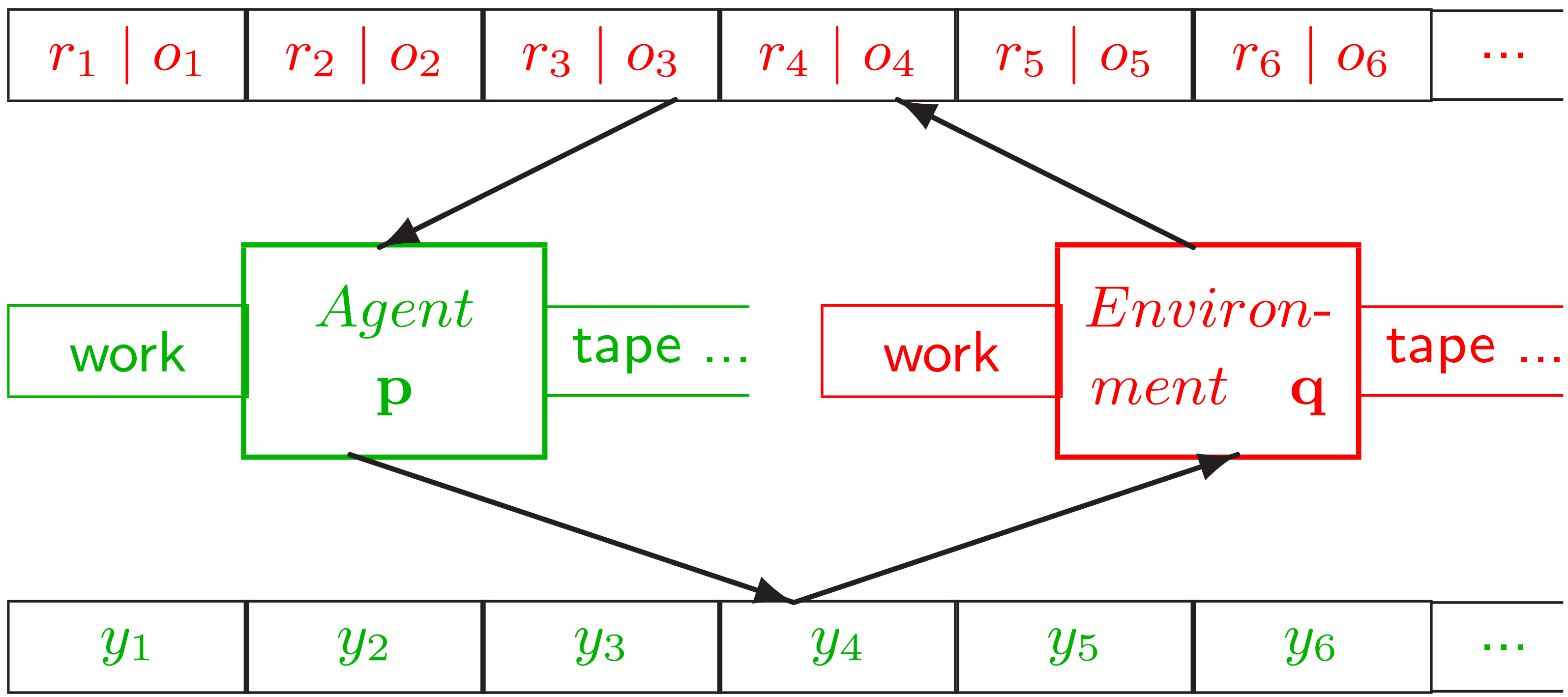
- Decision Theory solves the problem of rational agents in uncertain worlds if the environmental probability distribution is known.
- Solomonoff's theory of Universal Induction solves the problem of sequence prediction for unknown prior distribution.
- We combine both ideas and get



A Unified View of Artificial Intelligence

$$
\begin{array}{ccc}
 & = & \\
\text{Decision Theory} & = & \text{Probability} + \text{Utility Theory} \\
+ & & + \\
\text{Universal Induction} & = & \text{Ockham} + \text{Bayes} + \text{Turing}
\end{array}
$$

The Agent Model



Rational Agents in Deterministic Environments

- $p:\mathcal{X}^*\to\mathcal{Y}^*$ is deterministic policy of the agent,
  $p(x_{<k}) = y_{1:k}$ with $x_{<k} \equiv x_1...x_{k-1}$.
- $q:\mathcal{Y}^*\to\mathcal{X}^*$ is deterministic environment,
  $q(y_{1:k}) = x_{1:k}$ with $y_{1:k} \equiv y_1...y_k$.
- Input $x_k \equiv r_k o_k$ consists of a regular informative part o_k
  and reward $r_k \in [0..r_{max}]$.
- Value $V_{km}^{pq} := r_k + ... + r_m$,
  optimal policy $p^{best} := \arg\max_p V_{1m}^{pq}$,
  Lifespan or initial horizon m.

The Universal AIXI Agent

Problem: True env. q may be probabilistic and/or unknown.

Bayes: Take a mixture over environments.

Occam & Epicurus: Assign high/low prior weight to simple/complex q.

Solomonoff: Take all programs q and use prior$(q) = 2^{-\ell(q)}$.

Mixture over environments leads to mixture/expected

universal value: $V_{km}^{p\xi} \;:=\; \sum_q 2^{-\ell(q)} V_{km}^{pq}$

The program p^* that maximizes $V_{km}^{p\xi}$ should be selected.

Claim: AIXI policy $p^* := \arg\max_p V_{km}^{p\xi}$ is universally optimal agent.

Extended Policies

Problem: AIXI policy p^* is incomputable.

Supplement each policy p with a program that estimates $V_{km}^{p\xi}$ by w_k^p within time $\tilde{t}$.

Combine calculation of y_k^p and w_k^p and extend the notion of a policy to

$$p(\dot{y}\dot{x}_{<k}) \;=\; w_1^p y_1^p ... w_k^p y_k^p$$

with chronological order $w_1^p y_1^p \dot{y}_1 \dot{x}_1 w_2^p y_2^p \dot{y}_2 \dot{x}_2 ...$.

Notation:

- $\dot{y}\dot{x}_{<k}$= realized history.
- $w_i^p y_i^p$= estimates and actions by policy p.

Valid Approximations

- (Extended) policy p is not allowed to rate its output y_k^p with arbitrarily high w_k^p if we want w_k^p to be a reliable criterion for selecting the best p.
- Define (logical) predicate Valid Approximation:
  VA(p)=true $\Leftrightarrow$ p satisfies $w_k^p \leq V_{km}^{p\xi}$ $\forall k$, i.e. never overrates itself.
- Consider only p for which VA(p) can be proven in some formal axiomatic system.
- Enumerability of $V_{km}^{*\xi}$ ensures that for suff. large $\tilde{t}$ $\exists\tilde{p}$ for which VA($\tilde{p}$) can be proven and $w_k^{\tilde{p}}$ is arb. close to $V_{km}^{*\xi}$, i.e. $\tilde{p} \stackrel{\tilde{t}\to\infty}{\longrightarrow} p^*$.
- p is eff. more or equal intelligent than p' $:\Leftrightarrow$ $p \succeq^c p' \Leftrightarrow w_k^p \geq w_k^{p'} \forall k$.
- $\succeq^c$ is a co-enumerable partial order relation on extended policies.
- $\succeq^c$ orders valid policies w.r.t. the quality of their outputs and their ability to justify their outputs with high w_k.

The Universal Time-Bounded AIXItl Agent

Selection of the best algorithm p^{best} out of the time $\tilde{t}$ and length $\tilde{l}$ bounded p, for which there exists a proof of VA(p) with length $\leq l_P$:

1. Create all binary strings of length l_P and interpret each as a coding of a mathematical proof in the same formal logic system in which VA($\cdot$) was formulated. Take those strings that are proofs of VA(p) for some p and keep the corresponding programs p.

2. Eliminate all p of length $> \tilde{l}$.

3. Modify the behavior of all retained p in each cycle k as follows: Nothing is changed if p outputs some $w_k^p y_k^p$ within $\tilde{t}$ time steps. Otherwise stop p and write $w_k = 0$ and some arbitrary y_k to the output tape of p. Let P be the set of all those modified programs.

4. Start first cycle: $k := 1$.

AIXItl (continued)

5. Run every $p \in P$ on extended input $\dot{y}\dot{x}_{<k}$, where all outputs are redirected to some auxiliary tape: $p(\dot{y}\dot{x}_{<k}) = w_1^p y_1^p ... w_k^p y_k^p$. This step is performed incrementally by adding $\dot{y}\dot{x}_{k-1}$ for $k > 1$ to the input tape and continuing the computation of the previous cycle.

6. Select the program p with highest claimed value w_k^p:
$p_k^{best} := \arg\max_p w_k^p$.

7. Write $\dot{y}_k := y_k^{p_k^{best}}$ to the output tape.

8. Receive input $\dot{x}_k$ from the environment.

9. Begin next cycle: $k := k + 1$, goto step 5.

Roughly: If there exists a computable solution to some or all AI problems at all, the explicitly constructed algorithm p^{best} is such a solution.

Property of AIXItl

An algorithm p^{best} has been constructed for which the following holds:

- Let p be any (extended chronological) policy
- with length $\ell(p)\leq\tilde{l}$ and computation time per cycle $t(p)\leq\tilde{t}$
- for which there exists a proof of length $\leq l_P$ that p is a valid approximation.
- Then an algorithm p^{best} can be constructed, depending on $\tilde{l}$,$\tilde{t}$ and l_P but not on knowing p
- which is effectively more or equally intelligent according to $\succeq^c$ than any such p.
- The size of p^{best} is $\ell(p^{best})=O(\ln(\tilde{l}\cdot\tilde{t}\cdot l_P))$,
- the setup-time is $t_{setup}(p^{best})=O(l_P^2\cdot 2^{l_P})$,
- the computation time per cycle is $t_{cycle}(p^{best})=O(2^{\tilde{l}}\cdot\tilde{t})$.

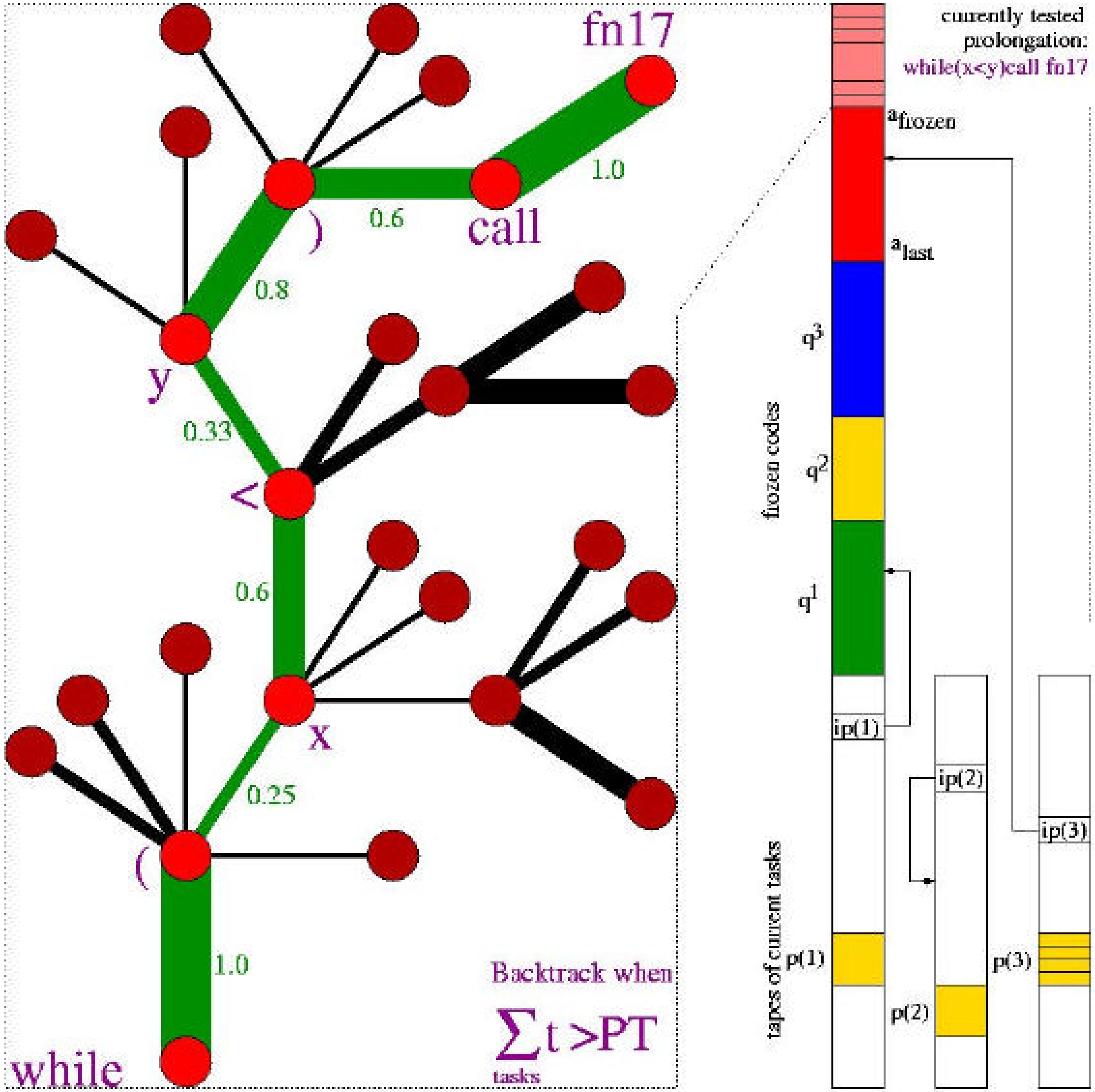
Optimal Ordered Problem Solver & Gödel Machine

$=$

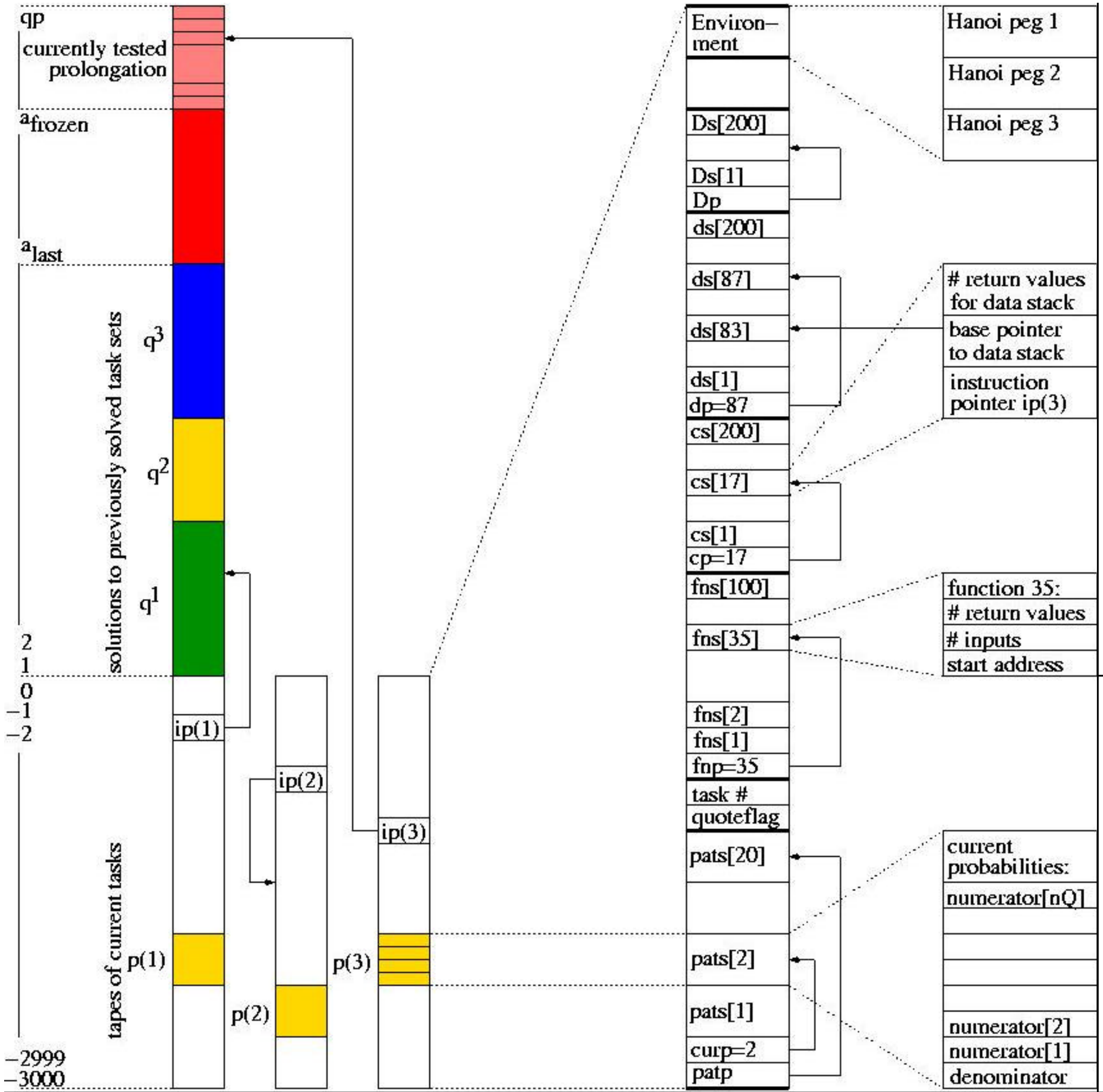
More practical adaptations/extensions

by Schmidhuber [2002-2004]

of FastPrg and AIXItl.



<u>OOPS:</u> bias-optimal reuse of previous solutions. In phase *n* solve tasks *1-n* (one tape per task) by new program (may call or copy-edit previous progs), or just task *n* by continuation of prog for phase *n-1*. *Search tree branches = program prefixes; widths = probabilities.* Backtrack to restore task sets / tapes *(including probability rewrites)* on error or when $\sum t$ > *probability * total time*. Just 8 times slower than bias-optimal method!



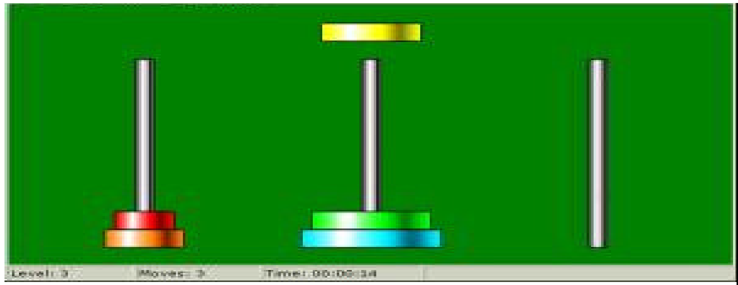
FORTH Pilot system

(other languages possible)

Miniature operating system for multitasking, interwoven with time-optimal backtracking. Programs = integer strings; data looks like code; functional programming easy; *~10⁶* steps/s on PC

"If it isn't 100 times shorter than C then it isn't FORTH." (C. Moore)

Schmidhuber may be 42 years old, but is still writing code by himself



Experiment:

Towers of Hanoi

3 pegs: S, A, D; n disks on S; move all to D, but never larger on smaller.

Anderson 1986: R-Learning, $n<4$.

Langley 1985: production systems, $n<6$.

Baum & Durdanovic 1999: simpler blocks problem scales linearly, $n<6$ (Kwee 2001)

Nonlearning AI planners: $n<15$; size $< 100{,}000$

(since search in raw solution space!)

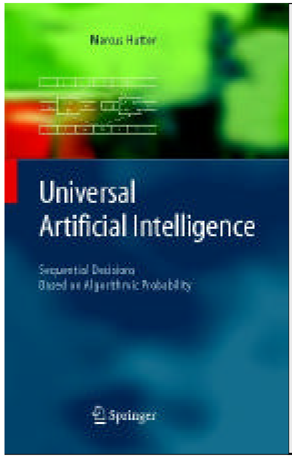
OOPS: $n \geq 30$; solution size $>10^9$

(because search in space of solution-computing programs!)

Optimal: 2^n-1 moves.

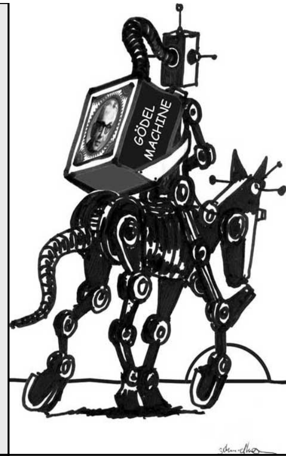
Speedup: first OOPS-learn seemingly unrelated language tasks $f(n)=1^n2^n$ for $n=1..30$, then Hanoi for $n=1…30$.

Demonstrates incremental learning - knowledge transfer from one task to the next - 1000 times faster than learning double-recursive Hanoi program from scratch.



Fastest algorithm for all well-defined problems

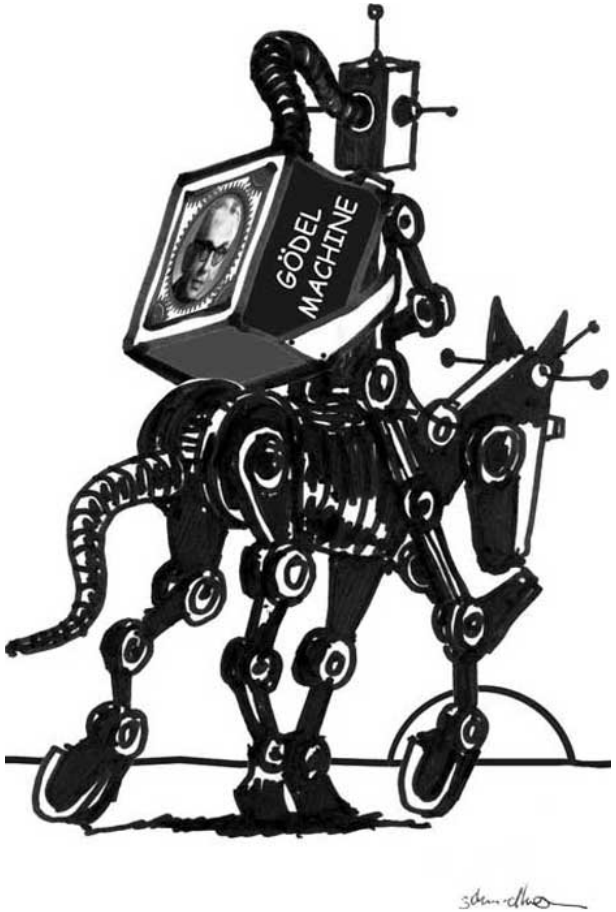
(Marcus Hutter, on Schmidhuber's SNF grant 20-61847):

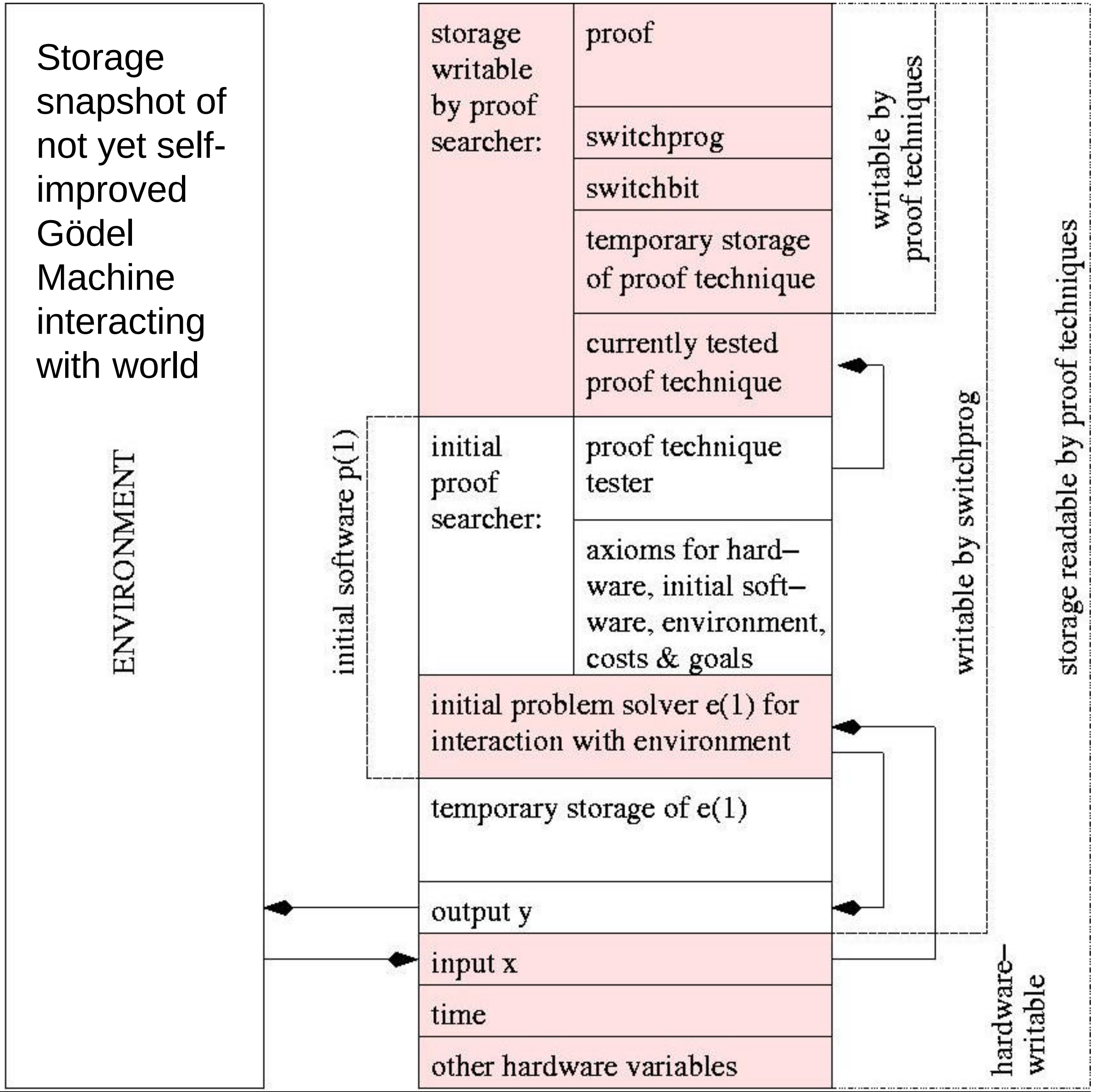


Given $f{:}X \rightarrow Y$ and $x \in X$, search proofs to find program q that provably computes $f(z)$ for all $z \in X$ within time bound $t_q(z)$; spend most time on $f(x)$-computing q with best current bound.
As fast as fastest f-computer, save for factor $1+\varepsilon$ and f-specific but x-independent constant!

But why all $z \in X$? Only f(x) needed!
How to reduce the huge constants?
Through the fully self-referential Gödel Machine (Schmidhuber, 2003):

Gödel Machine: Plug in *any* utility function as axiom stored in initial program *p*. *p* interacts with world and makes pairs *(q, proof)* until it finds proof of: *"rewrite of p through program q implies higher utility than leaving p as is."* Globally optimal self-change through executing *q*!




Storage snapshot of not yet self-improved Gödel Machine interacting with world
ENVIRONMENT
initial software p(1)
storage writable by proof searcher:
proof
switchprog
switchbit
temporary storage of proof technique
currently tested proof technique
initial proof searcher:
proof technique tester
axioms for hard–ware, initial soft–ware, environment, costs & goals
initial problem solver e(1) for interaction with environment
temporary storage of e(1)
output y
input x
time
other hardware variables
writable by proof techniques
writable by switchprog
storage readable by proof techniques
hardware–writable