Towards Practical Universal Search

Tom Schaul and Jürgen Schmidhuber
IDSIA, University of Lugano
Galleria 2, 6900 Manno
Switzerland

Abstract

Universal Search is an asymptotically optimal way of searching the space of programs computing solution candidates for quickly verifiable problems. Despite the algorithm's simplicity and remarkable theoretical properties, a potentially huge constant slowdown factor has kept it from being used much in practice. Here we greatly bias the search with domain-knowledge, essentially by assigning short codes to programs consisting of few but powerful domain-specific instructions. This greatly reduces the slowdown factor and makes the method practically useful. We also show that this approach, when encoding random seeds, can significantly reduce the expected search time of stochastic domain-specific algorithms. We further present a concrete study where Practical Universal Search (PUnS) is successfully used to combine algorithms for solving satisfiability problems.

Introduction

Universal Search is the asymptotically fastest way of finding a program that calculates a solution to a given problem, provided nothing is known about the problem except that there is a fast way of verifying solutions (Lev73). The algorithm has the property that the total time taken to find a solution is $O(t^*)$, where t^* is the time used by fastest program p^* to compute the solution. The search time of the whole process is at most a constant factor larger than t^*; typically this depends on the encoding length of p^*. The algorithm itself is very simple: It consists in running all possible programs in parallel, such that the fraction of time allocated to program p is $2^{-l(p)}$, where $l(p)$ is the size of the program (its number of bits).

More formally, assume a Turing-complete language L of binary strings that can encode all possible programs in a prefix-free code. Let p^* be the fastest program that solves a problem of problem complexity n. Then $t^* = f(n)$ is the number of time steps p^* needs to compute the solution. Let $l(p^*)$ be the size of p^* in L. Then the algorithmic complexity of Universal Search is $O(f(n))$. However, the multiplicative constant hidden by this notation turns out to be $2^{l(p^*)}$. (All the above assumes that there is a known way of verifying a given solution to the problem in time linear in the problem size n.)

Searching an infinite number of programs in parallel is impossible on a physical computer, thus an actual implementation of this algorithm has to proceed in phases, where in each phase more and more programs are run in parallel and the total search time per phase is continually increased. See algorithm 1 for the pseudocode.

Algorithm 1: Universal Search.

```
Input: Programming language, solution verifier
Output: Solution
phase := 1;
while true do
    for all programs p with l(p) ≤ phase do
        timelimit := 2^(phase−l(p));
        run p for maximally timelimit steps;
        if problem solved then
            return solution;
        end
    end
    phase := phase + 1;
end
```

For certain concrete problems and general-purpose languages it may seem improbable that the fastest program solving the problem can be encoded by fewer than, say, 50 bits, corresponding to a slowdown factor of $2^{50} \approx 10^{15}$, making Universal Search *impractical*.

Previous Extensions and Related Work

Several extensions of universal search have made it more useful in practice. The *Optimal Ordered Problem Solver* (OOPS, (Sch04)) incrementally searches a space of programs that may reuse programs solving previously encountered problems. OOPS was able to learn universal solvers for the Tower of Hanoi puzzle in a relatively short time, a problem other learning algorithms have repeatedly failed to solve. In (Sch95) a probabilistic variant of Universal Search called *Probabilistic Search* uses a language with a small but general instruction set to generate neural networks with exceptional generalization properties.

A non-universal variant (WS96) is restricted to strictly domain-specific instructions plus a jump statement. It is applied successfully to solving partially observable maze problems. The same paper also presents *ALS*, an adaptive version of Universal Search, which adjusts instruction probabilities based on experience.

Another recent development is Hutter's *HSearch* algorithm (Hut02). HSearch combines Universal Search in program space with simultaneous search for proofs about time bounds on their runtime. The algorithm is also asymptotically optimal, but replaces the multiplicative slowdown by an additive one. It may be significantly faster than Universal Search for problems where the time taken to verify solutions is nontrivial. The additive constant depends on the problem class, however, and may still be huge. A way to dramatically reduce such constants in some cases is a universal problem solver called the Gödel Machine (Sch09).

Other attempts have been made at developing practically useful non-exhaustive search algorithms inspired by Universal Search. This family of algorithms include time-allocation algorithms for portfolios of diverse algorithms (GS06).

Making Universal Search Practical

The more domain knowledge we have, the more we can shape or restrict the space of programs we need to search. Here we make Universal Search practically useful by devising a domain-specific language that encodes plausible (according to prior knowledge) programs by relatively few bits, thus reducing the slowdown factor to an acceptable size.

Dropping assumptions

Universal Search makes a number of assumptions about the language L. We will keep the assumption that L is a prefix-free binary code, and drop the following ones:

- L is Turing-complete,
- Every encoding corresponds to a valid program,
- L is infinite.

This does not mean that the opposites of those assumptions are true, only that they are not necessarily true (L is still allowed to be infinite or Turing-complete).

Another implicit assumption that is sometimes made on L is that its encodings represent a sequence of instructions in a standard programming language. Subsequently, we generalize this interpretation to include more restricted languages, such as encodings of parameter settings, random number generator seeds or top-level routines (e.g. 'localSearch()').

Thus, for *Practical Universal Search* (PUnS), L can encode an arbitrary set of programs, all of which can be domain-specific. While the language L thus may become more flexible, the search algorithm for it remains identical to Algorithm 1.

Optimality

PUnS inherits its optimality property directly from Universal Search. As long as the language remains Turing-complete, it has the same asymptotically optimal runtime complexity. In general it will be more restrictive, so this statement does not necessarily hold anymore. Still, the following, weaker one, holds:

Property 1 *For every problem instance, the order of runtime complexity of PUnS is the same as that of the best program which its language can encode.*

Integrating Domain Knowledge

There are two concrete approaches for integrating domain knowledge:

- We can restrict the language, to allow only programs that are appropriate for the problem domain. This can be done in a straightforward way if L is small and finite.
- We can bias the allocation of time towards programs that we suspect to perform better on the problem domain. Universal Search allocates time according to the descriptive complexity (i.e. the number of bits in its encoding) of the program. This is related to the concept of Occam's Razor, reflecting the hope that shorter programs will generalize better. Now, given domain knowledge about which programs will generally perform better, we can employ the same reasoning and encode those with fewer bits.

Fundamental Trade-off

Defining the language is the key element in PUnS – but this step has a strong inherent (and unresolvable) trade-off: the more general the language, the bigger the slowdown factor, and the more we reduce that one, the more biased the language has to be.

PUnS should therefore be seen as a broad *spectrum* of algorithms, which on one extreme may remain completely universal (like the original Universal Search) and cover all quickly verifiable problems. On the other extreme, if the problem domain is a single problem instance, it may degenerate into a zero-bit language that always runs the same fixed program (e.g. a hand-coded program that we know will efficiently solve the problem). In practice, neither of those extremes is what we want – we want an approach for solving a large number of problems within (more or less) restricted domains. This paper describes a general way of continually adjusting the universality/specificity of PUnS.

Practical Considerations

PUnS is a good candidate for multi-processor approaches, because it is easily *parallelizable*: the programs it runs are independent of each other, so the communication costs remain very low, and the overhead of PUnS is negligible.

Beyond the design of the language L, PUnS has no other internal parameters that would require tuning.

Furthermore, it is highly *tolerant* w.r.t. poorly designed languages and incorrect domain-knowledge: the result can never be catastrophic, as for every problem instance PUnS will still have the runtime complexity of the best solver that the language can express. Thus, an inappropriately designed language can be at most a constant factor worse than the optimal one, given the same expressiveness.

Languages for PUnS

The only condition we need to observe for L is that the encodings remain a prefix-free language. For complete generality, the language can always contain the original Turing-complete language of Universal Search as a fallback. Those encodings are then shifted to higher length, in order to free some of the encodings for the domain-specific programs.

The following sections will describe some variations of PUnS, discussing some specific points along the spectrum (as mentioned above) in more depth. Clearly, if appropriate in a domain, all those types of languages can be combined into a single *hybrid* language.

Domain-biased Programming Languages

Consider a domain where no efficient or general algorithms for solving problems exist, so that it is necessary to search very broadly, i.e. search the space of programs that might solve the problem. If we use a standard Turing-complete language that encodes sequences of instructions, we have more or less the original Universal Search - and thus a huge constant slowdown. However, we can integrate domain knowledge by adding (potentially high-level) domain-specific subroutines with short encodings to bias the search. Furthermore, we can make the language *sparser* by restricting how instructions can be combined (reminiscent of strong typing in standard programming languages). A language like this will remain Turing-complete, and the slowdown factor still risks to be high: the runtime will be acceptable only if the modified language is either very sparse, i.e. almost all bit-strings do not correspond to legal programs and thus only relatively few programs of each length are run[1], or it is compact enough to allow for solution-computing programs with no more than 50 bits. Successfully applied examples of this kind of PUnS can be found in (WS96; Sch95; Sch05).

A language that directly encodes solutions (with a domain-specific complexity measure) causes PUnS to perform a type of exhaustive search that iteratively checks more and more complex solutions. This was explored in a companion paper (KGS10) for searching the space of neural networks, ordered by their encoding length after compression.

[1]Note that the cost of finding legal programs dominates when the language is extremely sparse, that is, only solution-computing programs are legal.

Exploration of Parameter Space

If we know a good algorithm for arriving at a solution to a problem, but not the settings that allow the algorithm to solve the problem efficiently (or at all), PUnS can be used to search for good parameters for the algorithm. In this case, each program tested by PUnS is actually the same algorithm, run with different parameters. We can view the interpretation of that language as a non-Turing complete virtual machine that runs "programs" specified as parameter settings.

Any parametrized algorithm could be used as a virtual machine for this type of search. However, the algorithms that are best suited for this purpose are those where parameters are discrete, and can naturally be ordered according to the complexity of the search resulting from a particular parameter setting. There is a wide range of machine learning algorithms that exhibit this characteristic in various ways (e.g. the number of free variables in a function approximator used by an algorithm).

Stochastic Algorithms

Consider a domain where a good algorithm exists and the algorithm is either non-parametric, or good settings for its parameters are known. However, the algorithm is stochastic, and converges to a solution only in a small (unknown) fraction of the runs. In such a domain, universal search could be employed to search the space of random number generator seeds for the algorithm. These seeds are naturally ordered by length, encoded as prefix-free binary integers. While this is a very degenerate language, it fulfills all criteria for being used by Universal Search.

In this case PUnS will spawn more and more processes of the stochastic algorithm in every phase, each with a different seed, until one of them eventually finds a solution. As the encodings have incrementally longer encodings, we do not need to know anything about the probability of success: exponentially more time is allocated to processes with short encodings, so PUnS will only spawn many more processes if they are needed, i.e. if the first random seeds do not lead to convergence fast enough.

In the rest of this section, we will present one such example language, and analyze under which circumstances it is advantageous to apply PUnS to it. Consider the language that encodes an unlimited number of random seeds as '0^k1' for the kth seed, such that seed k is allocated 2^{-k} of the total time.

Let us assume a stochastic base-algorithm where the time T required to find the solution is a random variable, with a probability density function $\phi(t)$ and cumulative probability function $\Phi(t) = P(t_{req} \leq t)$.

Then the time required by PUnS to find the solution T' is the minimum of an infinite number of independent realizations of T, with exponentially increasing penalties:

$$T' = \min\left(2^1T, 2^2T, 2^3T, \ldots, 2^kT, \ldots\right)$$

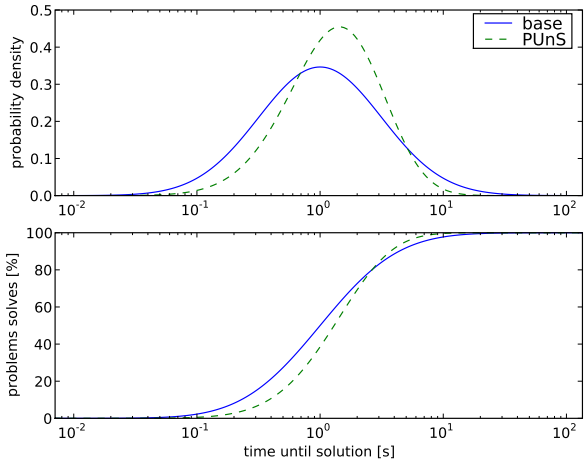

Figure 1: Above: Probability density functions ϕ and ϕ', of the base distribution ($\sigma = \frac{1}{2}\log(10)$) and PUnS, respectively. Below: percentage of problems solved faster than a certain time, for the base-algorithm and PUnS.

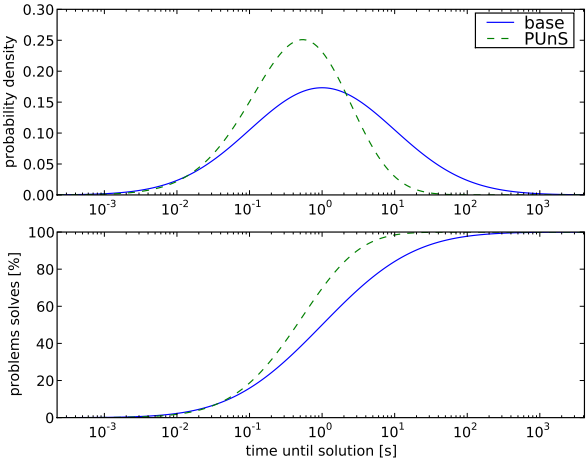

Figure 2: Above: Probability density functions ϕ and ϕ', of the wider base distribution ($\sigma = \log(10)$) and corresponding PUnS, respectively. Below: percentage of problems solved faster than a certain time, for the base-algorithm and PUnS.

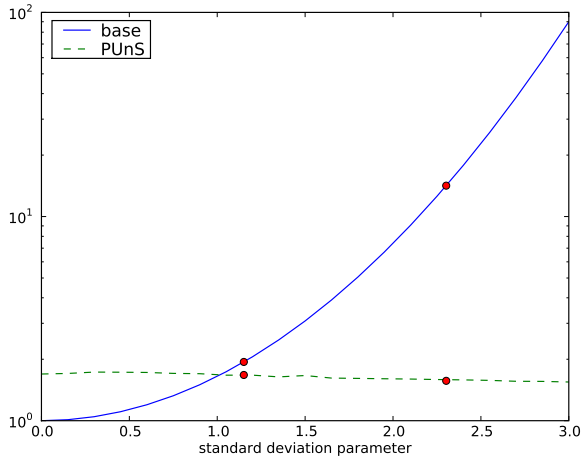

Figure 3: Mean times as a function of the σ parameter, for both the base-algorithm and PUnS. The circles correspond to the values for figures 1 and 2. Note the log-scale on the y-axis: the mean time for the base algorithm increases faster than exponential w.r.t. σ, while it decreases slightly for PUnS.

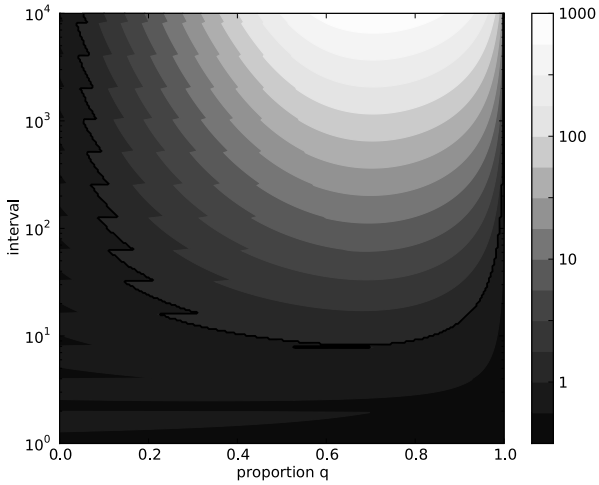

Figure 4: The shades of grey in this plot code for the proportion $\overline{t_b}/\overline{t_p}$, i.e. the factor by which the expected solution time is reduced when employing PUnS instead of the base-algorithm. The horizontal axis shows the dependency on proportion q, while the vertical axis corresponds to the interval size λ. The black line corresponds to limit cases, where both versions have the same expected time: in the whole upper middle part PUnS is better (large enough interval, and not too small q), sometimes by orders of magnitude. The discontinuities (dents) happen whenever λ traverses a power of 2, i.e. whenever k is incremented.

T' has density function

$$\phi'(t) = \sum_{k=1}^{\infty} \phi(t/2^k) \prod_{i=1, i\neq k}^{\infty} 1 - \Phi(t/2^i)$$

and cumulative density function

$$\Phi'(t) = 1 - \prod_{k=1}^{\infty} (1 - \Phi(t/2^k)).$$

Note that it is possible to truncate the computation of the infinite sums and products after a small number of terms, under the reasonable assumption that $\phi(t)$ decays fast as t approaches zero.

Figure 1 illustrates the simple case where the required time is normally distributed in log-time space (i.e. the log-normal distribution) with $\mu = 0$ and $\sigma = \frac{1}{2}\log(10)$. We observe that PUnS reduces the probability of long runtimes (over 5 seconds). In general it has the property of reducing the right (expensive) tail of the base distribution. When the base distribution has a larger standard deviation the effect is even more pronounced (see Figure 2, which shows the same plot as before, but for $\sigma = \log(10)$). In this case we observe an additional beneficial effect, namely that the mean time is reduced significantly. In figure 3 we plot the mean times as a function of σ to illustrate this effect in detail.

Another case of interest is a stochastic base-algorithm with two distinct outcomes: with probability q it finds the solution after t_1, otherwise it requires $t_2 = \lambda t_1$. This algorithm has an expected solution time of

$$\overline{t_b} = t_1 \left[1 + (\lambda - 1)(1 - q)\right].$$

Applying PUnS to the above language, it can be shown that the expected time changes too

$$\overline{t_p} = t_1 \left[1 + (\lambda - 2^k)(1 - q)^{k+1} + \sum_{i=0}^{k} 2^i (1 - q)^i\right],$$

where $k = \lfloor \log_2 \lambda \rfloor$ is the largest integer such that $2^k \leq \lambda$. Figure 4 shows for which values of q and λ PUnS outperforms the base-algorithm, and by how much (note that those results are independent of t_1).

To summarize, whenever we have access to a stochastic domain-specific algorithm with high variability in its solution times, using PUnS with a simple language to encode random seeds (e.g. the one introduced in this section) can reduce the expected solution time by orders of magnitude.

Case study: SAT-UNSAT

This section presents a small case-study of using PUnS on a mixed SAT-UNSAT benchmark with 250 boolean variables. We use as the underlying base-programs two standard algorithms:

- A local search algorithm (G2WSAT, (LH05)) which is fast on satisfiable instances, but does not halt on unsatisfiable ones.
- A complete solver (Satz-Rand (GSCK00)) that can handle both kinds of instances, but is significantly slower.

Both these algorithms are stochastic, but G2WSAT has a high variance on the time needed to find a solution for a given instance. We set all parameters to default values (G2WSAT: noise = 0.5, diversification = 0.05, time limit = 1h; Satz-Rand: noise = 0.4, first-branching = most-constrained) (GS06).

The language we define for PUnS combines both base-algorithms, employing the coding scheme introduced in the previous section for the high-variance G2WSAT: '11' encodes running of Satz-Rand, '01', '001', '0...01' encode running of G2WSAT with a random seed (a different seed for every number of '0' bits).

In this case, a third of the total computation time is allocated to each Satz-Rand, the first random seed for G2WSAT and all other random seeds combined. With this language, the optimal performance corresponds to that of *Oracle* which for every problem instance knows in advance the fastest solver and the best random seed.

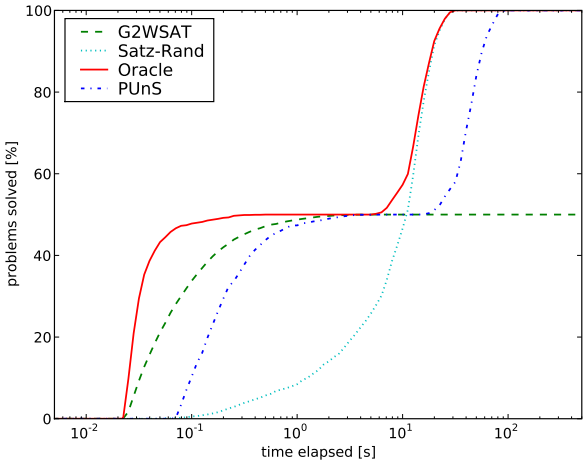

Figure 5: Percentage of instances solved, given a certain computation time for G2WSAT, Satz-Rand, Oracle and PUnS on the mixed SAT-UNSAT-250 benchmark (averaged over 20 runs with different random seeds).

Figure 5 shows the results of running all four algorithms (including Oracle) on a set of 100 satisfiability instances, half of which are unsatisfiable. We find that Practical Universal Search is indeed a robust way of combining the base-algorithms. By construction, it is never slower by more than a factor 3 w.r.t. the best base-algorithm. In addition, the reduced risk of a bad initialization (seed) for G2WSAT on the boundary cases (almost unsatisfiable) is clearly visible as well: Compare the much steeper increase of the PUnS plot, as compared to the G2WSAT one. Finally, as expected, the PUnS performance is approximately that of *Oracle* with a constant factor slowdown – the difference is

due to the fact that the encoding length of the optimal random seed is not bounded a priori.

Conclusions

Universal Search can be used in practice by biasing its language for encoding programs. We provided guidelines for integrating domain-knowledge, possibly (but not necessarily) at the cost of universality. We described a simplified language for non-universal problem domains, and emphasized the flexibility of the approach. In particular, we established that encoding random seeds for stochastic base-algorithms can be highly advantageous. Finally we conducted a proof-of-concept study in the domain of satisfiability problems.

Future work

One direction to pursue would be to develop a general adaptive version of PUnS, where program probabilities change over time based on experience, like in ALS (WS96). A related direction will be to extend PUnS along the lines of OOPS (Sch04), reducing sizes and thus increasing probabilities of encodings of programs whose subprograms have a history of quickly solving previous problems, thus increasing their chances of being used in the context of future problems. There also might be clever ways of adapting the language based on intermediate results of (unsuccessful) runs, in a domain-specific way.

Acknowledgments

We thank Matteo Gagliolo for permission to use his experimental data, as well as Julian Togelius for his valuable input. This work was funded in part by SNF grant number 200021-113364/1.

References

Matteo Gagliolo and Jürgen Schmidhuber. Learning Dynamic Algorithm Portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, August 2006.

Carla P Gomes, Bart Selman, Nuno Crato, and Henry A Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24:67–100, 2000.

Marcus Hutter. The Fastest and Shortest Algorithm for All Well-Defined Problems. *International Journal of Foundations of Computer Science*, 13:431–443, 2002.

Jan Koutnik, Faustino Gomez, and Jürgen Schmidhuber. Searching for Minimal Neural Networks in Fourier Space. In *Proceedings of the Conference on Artificial General Intelligence*, Lugano, Switzerland, 2010.

Leonid A Levin. Universal sequential search problems. *Problems of Information Transmission*, 9:265–266, 1973.

Chu M Li and Wenqi Huang. Diversification and Determinism in Local search for Satisfiability. In *Proceedings of the 8th International Conference on Satisfiability (SAT)*, pages 158–172, 2005.

Jürgen Schmidhuber. Discovering solutions with low Kolmogorov complexity and high generalization capability. In A Prieditis and S Russell, editors, *Proceedings of the International Conference on Machine Learning (ICML)*, pages 488–496, 1995.

Jürgen Schmidhuber. Optimal Ordered Problem Solver. *Machine Learning*, 54:211–254, 2004.

Tom Schaul. Evolving a compact, concept-based Sokoban solver, 2005.

Jürgen Schmidhuber. Ultimate Cognition à la Gödel. *Cognitive Computation*, 1:177–193, 2009.

Marco Wiering and Jürgen Schmidhuber. Solving POMDPs using Levin search and EIRA. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 534–542, 1996.