Algorithmic Probability, Heuristic Programming and AGI

Ray J. Solomonoff
Visiting Professor, Computer Learning Research Center
Royal Holloway, University of London

IDSIA, Galleria 2, CH–6928 Manno–Lugano, Switzerland
rjsolo@ieee.org http://world.std.com/~rjs/pubs.html

Introduction

This paper is about Algorithmic Probability (ALP) and Heuristic Programming and how they can be combined to achieve AGI. It is an update of a 2003 report describing a system of this kind (Sol03). We first describe ALP, giving the most common implementation of it, then the features of ALP relevant to its application to AGI.

They are: Completeness, Incomputability, Subjectivity and Diversity. We then show how these features enable us to create a very general, very intelligent problem solving machine. For this we will devise "Training Sequences" — sequences of problems designed to put problem-solving information into the machine. We describe a few kinds of training sequences.

The problems are solved by a "generate and test" algorithm, in which the candidate solutions are selected through a "Guiding Probability Distribution". The use of Levin's search procedure enables us to efficiently consider the full class of partial recursive functions as possible solutions to our problems. The guiding probability distribution is updated after each problem is solved, so that the next problem can profit from things learned in the previously solved problems.

We describe a few updating techniques. Improvements in updating based on heuristic programming is one of the principal directions of current research. Designing training sequences is another important direction.

For some of the simpler updating algorithms, it is easy to "merge" the guiding probabilities of machines that have been educated using different training sequences — resulting in a machine that is more intelligent than any of the component machines.

What is Algorithmic Probability?

ALP is a technique for the extrapolation of a sequence of binary symbols — all induction problems can be put into this form. We first assign a probability to any finite binary sequence. We can then use Bayes' theorem to compute the probability of any particular continuation sequence. The big problem is: how do we assign these probabilities to strings? In one of the commonest implementations of ALP, we have a Universal Turing Machine with three tapes: a unidirectional input tape, a unidirectional output tape, and a bidirectional work tape. If we feed it an input tape with 0's and 1's on it, the machine may print some 0's and 1's on the output — it could print nothing at all or print a finite string and stop or it could print an infinite output string, or it could go into an infinite computing loop with no printing at all.

Suppose we want to find the ALP of finite string x. We feed random bits into the machine. There is a certain probability that the output will be a string that starts out with the string x. *That* is the ALP of string x.

To compute the ALP of string x:

$$P_M(x) = \sum_{i=0}^{\infty} 2^{-|S_i(x)|}$$

Here $P_M(x)$ is the ALP (also called Universal Probability) of string x with respect to machine, M.

There are many finite string inputs to M that will give an output that begins with x. We call such strings "codes for x". Most of these codes are redundant in the sense that if one removes its most recent bit the resultant string will still be a "code for x". A "minimal code for x" is one that is not redundant. If one removes its last bit, the result will no longer be a"code for x". Say $|S_i(x)|$ is the length in bits of the i^{th} "Minimal code for x".

$2^{-|S_i(x)|}$ is the probability that the random input will begin with the" i^{th} minimal code for x".

$P_M(x)$ is then the sum of the probabilities of all the ways that a string beginning with x, could be generated.

This definition has some interesting properties:

First, it assigns high probabilities to strings with short descriptions — This is in the spirit of Ockham's razor. It is the converse of Huffman coding that assigns short codes to high probability symbols.

Second, its value is somewhat independent of what universal machine is used, because codes for one universal machine can always be obtained from another universal machine by the addition of a finite sequence of translation instructions.

A less apparent but clearly desirable property — $P_M(x)$ is *complete*. This means that if there is any

describable regularity in a batch of data, P_M will find it, using a relatively small amount of the data. At this time, it is the only induction method known to be complete (Sol78).

More exactly: Suppose $\mu(x)$ is a probability distribution on finite binary strings. For each $x = x_1, x_2 \cdots x_i$, μ gives a probability that the next bit, x_{i+1} will be 1: $\mu(x_{i+1} = 1|x_1, x_2 \cdots x_i)$

From P_M we can obtain a similar function $P(x_{i+1} = 1|x_1, x_2 \cdots x_i)$.

Suppose we use μ to generate a sequence, x, Monte Carlo-wise. μ will assign a probability to the $i + 1^{th}$ bit based on all previous bits. Similarly, P will assign a probability to the $i + 1^{th}$ bit of x. If P_M is a very good predictor, the probability values obtained from μ and from P_M will be very close, on the average, for long sequences. What I proved was:

$$\mathrm{E}_\mu \sum_{i=1}^{n} (\mu(x_{i+1} = 1|x_1, x_2 \cdots x_i) - P(x_{i+1} = 1|x_1, x_2 \cdots x_i))^2 \leq \frac{1}{2} k \ln 2$$

The expected value of the sum of the squares of the differences between the probabilities is bounded by about $.35k$. k is the minimum number of bits that M, the reference machine, needs to describe μ. If the function μ is describable by functions that are close to M's primitive instruction set, then k will be small and the error will be small. But whether large or small, the squared error in probability must converge faster than $\frac{1}{n}$ (because $\sum \frac{1}{n}$ diverges).

Later research has shown this result to be very robust — we can use a large, (non-binary) alphabet and/or use error functions that are different from total square difference (Hut02). The probability obtained can be normalized or unnormalized (semi-measure)(Gác97).

The function μ to be "discovered" can be any describable function — primitive recursive, total recursive, or partial recursive. When ALP uses an unnormalized semi-measure, it can discover *incomputable* functions as well.

The desirable aspects of ALP are quite clear. We know of no other model of induction that is nearly as good.

An apparent difficulty — $P_M(x)$ is *incomputable*: The equation defining $P_M(x)$ tells us to find all strings that are "minimal codes for x." Because of the Halting Problem, it is impossible to tell whether certain strings are codes for x or not. While it is easy to make approximations to $P_M(x)$, the fact that it is incomputable has given rise to the common misconception that ALP is little more than an interesting theoretical model with no direct practical application. We will show that *In Fact* incomputability is a *desirable* feature and imposes no serious restrictions on its application to the practical problems of AGI.

The usual question is — "What good is it if you can't compute it?" The answer is that for practical prediction we don't have to compute ALP *exactly*. Approximations to it are quite usable and *the closer an approximation is to ALP, the more likely it is to share ALP's desirable qualities.*

Perhaps the simplest kind of approximation to an incomputable number involves making rational approximations to $\sqrt{2}$. We know that there is no rational number whose square is 2, but we can get arbitrarily close approximations. We can also compute an upper bound on the error of our approximation and for most methods of successive approximation we are assured that the errors approach zero. In the case of ALP, though we are assured that the approximations will approach ALP arbitrarily closely, the incomputability implies that we *cannot ever* compute useful upper bounds on approximation error — but *for few if any practical applications do we need this information.*

The approximation problem for the universal distribution is very similar to that of approximating a solution to the Traveling Salesman Problem, when the number of cities is too large to enable an exact solution. When we make trial paths, we always know the total length of each path — so we know whether one trial is better than another. In approximations for the universal distribution, we also always know when one approximation is better than another — and we know how much better. In some cases, we can combine trials to obtain a trial that is better than either of the component trials. In both TSP and ALP approximation, we never know how far we are from the theoretically best, yet in both cases we do not hesitate to use approximate solutions to our problems.

The incomputability of ALP is closely associated with its completeness. Any complete induction system cannot be computable. Conversely, any computable induction system cannot be complete. For any computable induction system, it is possible to construct a space of data sequences for which that system gives *extremely* poor probability values. The sum of the squared errors diverges linearly in the sequence length.

Appendix B gives a simple construction of this kind.

We note that the incomputability of ALP makes such a construction impossible and its probability error always converges to zero for *any* finitely describable sequence.

To explain our earlier remark on incomputability as a very *desirable* feature: Incomputability is the *only* way we can achieve completeness. In ALP this incomputability imposes no penalty on its practical application. It is a true "Win, Win" situation!

Another item of importance: For most applications an estimate of future prediction error is needed. Cross Validation or one of its many variants is usually possible. In this aspect of the prediction problem ALP is certainly no worse than any other method. On the other hand, ALP gives a good theoretical framework that enables us to make better estimates.

Subjectivity

Occasionally in making extrapolations of a batch of data, there is enough known about the data so that it is clear that a certain prediction technique is optimal. However, this is often not the case and the investigator must make a (subjective) choice of inductive techniques. For me, the choice is clear: I choose ALP because it is the only complete induction method I know of. However ALP has another subjective aspect as well: we have to choose M, the reference machine or language. As long as M is universal, the system will be complete. This choice of M enables us to insert into the system any a priori information we may have about the data to be predicted and still retain completeness.

The choice of the M can be used very effectively for incremental learning: Suppose we have 100 induction problems: $X_1, X_2, \cdots X_{100}$.

The best solution would involve getting the machine to find a short code for the entire batch of 100 problems. For a large corpus this can be a lengthy task. A shorter, approximate way: using the machine M as reference, we find a prediction code for X_1. In view of this code, we modify M to become M' in such a way that if $P_{M'}$, makes a prediction for X_2, it will be the same as if we used P_M for both X_1 and X_2. M' becomes a *complete summary* of Ms increase in knowledge after solving X_1. We can consider the M to M' transformation as an *updating* of the M to M' It is possible to show that such an M can be found *exactly*, but the exact construction is very time consuming (reference). Approximations to M' can be readily found. After M' solves X_2, M' can be updated to M'' and have it solve X_3, and so on to X_{100}. We will discuss the update process later in more detail, for a somewhat different kind of problem.

To understand the role of subjectivity in the life of a human or an intelligent machine, let us consider the human infant. It is born with certain capabilities that assume certain a priori characteristics of its environment–to–be. It expects to breathe air, its immune system is designed for certain kinds of challenges, it is usually able to learn to walk and converse in whatever human language it finds in its early environment. As it matures, its a priori information is modified and augmented by its experience.

The inductive system we are working on is of this sort. Each time it solves a problem or is unsuccessful in solving a problem, it updates the part of its a priori information that is relevant to problem solving techniques. In a manner very similar to that of a maturing human being, its a priori information grows as the life experience of the system grows.

From the foregoing, it is clear that the subjectivity of algorithmic probability is a necessary feature that enables an intelligent system to incorporate experience of the past into techniques for solving problems of the future.

Diversity

In Section 1 we described ALP based on a universal Turing machine with random input. An equivalent model considers all prediction methods, and makes a prediction based on the weighted sum of all of these predictors. The weight of each predictor is the product of two factors: the first is the a priori probability of each predictor. It is the probability that this predictor would be described by a universal Turing machine with random input. If the predictor is described by a small number of bits, it will be given high a priori probability. The second factor is the probability assigned by the predictor to the data of the past that is being used for prediction. We may regard each prediction method as a kind of model or explanation of the data. Many people would use only the best model or explanation and throw away the rest. Minimum Description Length (Ris78), and Minimum Message Length (WB68) are two commonly used approximations to ALP that use only the best model of this sort. When one model is much better than any of the others, then Minimum Description Length and Minimum Message Length and ALP give about the same predictions. If many of the top models have about the same weight, then ALP gives better results — the other methods give too much confidence in the predictions of the very best model (PH06).

However, that's not the *main* advantage of ALP's use of a diversity of explanations. If we are making a single kind of prediction, then discarding the non-optimum models usually has a small penalty associated with it. However if we are working on a sequence of prediction problems, we will often find that the model that worked best in the past is inadequate for the new problems. When this occurs in science we have to revise our old theories. A good scientist will remember many theories that worked in the past but were discarded — either because they didn't agree with the new data, or because they were a priori "unlikely". New theories are characteristically devised by using failed models of the past, taking them apart, and using the parts to create new candidate theories. By having a large diverse set of (non-optimum) models on hand to create new trial models, ALP is in the best possible position to create new, effective models for prediction.

In the biological world, when a species loses its genetic diversity it can quickly succumb to small environmental changes — it soon becomes extinct. Lack of diversity in potato varieties in Ireland led to massive starvation.

When ALP is used in Genetic Programming, it's rich diversity of models can be expected to lead to very good, very fast solutions with little likelihood of "premature convergence".

Putting It All Together

I have described ALP and some of its properties, and to some extent, how it could be used in an a AGI system. This section gives more details on how it works. We

start out with problems that are input/output pairs (I/O pairs). Given a sequence of them and a new input we want the machine to get a probability density distribution on its possible outputs. We allow I and O to be strings or numbers or mixtures of strings and numbers. Very many practical problems fall into this class — e.g. classification problems, identification problems, symbolic and numerical regression, grammar discovery$\cdots$. To solve the problems, we create trial functions that map inputs into possible outputs. We try to find several successful functions of this kind for each of the problems. It is usually desirable, though not at all necessary, to find a common function for all of the problems.

The trial functions are generated by a universal function language such as Lisp or Forth. We want a language that can be defined by a context free grammar, which we use to generate trial functions. The functions in such languages are represented by trees, which display the choices made in generating the trial functions. Each node in the tree represents a choice of a terminal or non-terminal symbol. Initially if there are k possible choices at a node, each choice is given probability $1/k$. These probabilities which we call "The Guiding Probability Distribution"(GPD) will evolve considerably, as the training sequence progresses.

In a simplified version of the technique we use Levin's Search Procedure (Lsearch)[1] to find the *single* most likely solution to each of n problems. For each problem, I_i/O_i, we have a function, F_i represented in say, Reversed Polish Notation (RPN), such that $F_i(I_i) = O_i$. Using a suitable prediction or compression algorithm (such as Prediction by Partial Matching — PPM) we compress the set of function descriptions, $[F_i]$. This compressed version of the set of solutions can be predictive and enables us to get a probability distribution for the next sequence of symbols — giving a probability distribution over the next function, F_{n+1}. Levin Search gives us F_{n+1} candidates of highest probability first, so when we are given the next I_{n+1}/O_{n+1} pair, we can select the F_{n+1} of largest probability such that $F_{n+1}(I_{n+1}) = O_{n+1}$.

We then update the system by compressing the code for F_{n+1} into the previous sequence of solution functions and use this compressed code to find a solution to the next I/O in the training sequence. This continues until we have found solution functions for all of the problems in the sequence.

In a more realistic version of the system, using the diversity of ALP, we try to find several functions for each I/O pair in a corpus of say, 10 pairs. Suppose we have obtained 2 functions for each problem in the set. This amounts to $2^{10} = 1024$ different "codes" for the entire corpus. We then compress each of these codes and use the shortest say, 30 of them for prediction on the new input, I_{101}. The probability distribution on the output, O_{101} will be the weighted mean of the predictions of the 30 codes, weights being proportional to $2^{-code\ length}$. We also use these 30 codes to assign probabilities to grammar elements in constructing trial functions in future problems.

We mentioned earlier, the use of heuristic programming in designing this system. In both training sequence design and in the design and updating of the GPD, the techniques of heuristic programming are of much import.

Consider the problem of learning simple recursive functions. We are given a sequence of $n, F(n)$ pairs containing some consecutive values of n. We want to discover the function, $F()$. A heuristic programmer would try to discover how he himself would solve such a problem — then write a program to simulate himself.

For machine learning, we want to find a way for the machine to discover the trick used by the heuristic programmer — or, failing that, the machine should discover when to use the technique, or be able to break it down into subfunctions that are useful for other problems as well. Our continuing problem is to create training sequences and GPDs that enable the machine to do these things.

The Guiding Probability Distribution

The Guiding Probability Distribution (GPD) does two important things: first it discovers frequently used functions and assigns high probabilities to them. Second, it discovers for each function, contexts specifying the condition under which that function should be applied. Both of these operations are quantified by ALP.

In the previous section we described the GPD — how it made predictions of symbols in the next trial function — how the predictions were based on regularities in the sequence of symbols that represent in the solutions of earlier problems. Here we will examine the details of just how the GPD works.

Perhaps the simplest sequential prediction scheme is Laplace's rule: The probability of the next symbol being A, say, is proportional to (the number of times A has occurred in the past, plus one). There is no dependency at all on context. This method of guiding probability evaluation was used in OOPS (Sch02) a system similar in several ways to the presently described system.

Prediction by Partial Matching (PPM) is a very fast, relatively simple, probability evaluation scheme that uses context very effectively. It looks at the string of symbols preceding the symbol to be predicted and makes a probability estimate on this basis.

PPM and variations of it have been among the best compressors in Hutter's Entwiki challenge — a competition to discover the best compressor for 10^8 Bytes of the wikipedia.

Most of the improvements in PPM involve "context weighting" — they use several independent prediction schemes, each based on context in a different way. These systems are then merged by giving (localized) weights each of them.

[1] See Appendix A

Merging of this sort can be used on the GPDs of several learning systems trained with different training sequences. A weakness of this simple merging is that the system does *not* create new functions by composing functions discovered by the different prediction schemes.

— A relevant quote from von Neumann — "For difficult problems, it is good to have 10 experts in the same room — but it is far better to have 10 experts in the same head".

For really good induction, the GPD needs to recognize useful subfunctions and contexts to control the application of these subfunctions. PPM does this to some extent, but it needs much modification. While it is, in theory, easy to specify these modifications, it seems to be difficult to implement them with any speed. Much work needs to be done in this area.

Suppose Lsearch is generating candidate functions of I_j, the current input problem. If x is the part of the candidate that has been generated thus far, then in general, the probability distribution on the symbol to follow x, will be some function of I_j and x, i.e. $G(I_j, x)$. The form of G will slowly vary as we advance along the training sequence. For the best possible predictions, the form of G should be able to be any conceivable partial recursive function. Few prediction systems allow such generality.

How does the high compression obtained by PPM effect prediction systems? Compression ratio is the ratio of uncompressed string length to compressed string length. Compression ratio translates directly into increasing (geometric) mean probability of symbols and subsequences of symbols. A compression ratio of two increases probabilities to their square root. This translates directly into decreasing search times for solutions to problems. Using Lsearch, the time to find a particular solution will be about t_i/p_i, t_i being time needed to generate and test the solution, and p_i being the probability assigned to the solution. If $t_i = 10^{-6}$ seconds and $p_i = 10^{-16}$ for a particular uncompressed problem, then it will take about 10^{10} seconds — about 330 years to find a solution. A compression factor of two will increase p_i to the square root of 10^{-16} — i.e. 10^{-8}. So $t_i/p_i = 10^2$ seconds — about 1.7 minutes — a speed of up of 10^8.

What compression ratios can we expect from PPM? A recent version got a compression ratio of 6.33 for a 71 kbyte LISP file. Unfortunately, much of this ratio was obtained by compressing long words used for function names. This kind of compression does not help find functional solutions. From the compression ratios of other less efficient compressors, my guess is that the elimination of this "long word" regularity would still give a compression ratio of greater than two. — Perhaps as much as three — enabling the rapid solution of problems that without compression would take many years of search time.

It is notable that high compression ratios were obtained for long text files. For us, the moral is that we will not get full advantage of compression until our training sequences are long enough.

We have discussed PPM at some length as being a good initial GPD. A few other prediction methods that we have examined:

Genetic programming: Very effective. It can discover recursive prediction functions, but it is very slow. We have, however found many ways in which it could be sped up considerably (Sol06).

Echo State Machines (ESM). (JH04) A very deep neural net — very fast in implementation. Doesn't do recursive functions, but we have found a way to move in that direction.

Support Vector Regression (SVR). (SS09) This is the application of SVMs to regression. The predictions are very good, but the algorithms are very slow and do not support recursion.

In any of the compressors, speed and compression ratios are both important. In selecting a compressor it is necessary to consider this trade-off.

Training Sequences

It is clear that the sequence of problems presented to the system will be an important factor in determining whether the mature system will be very much more capable than the infant system. The task of the training sequence is to teach functions of increasing difficulty by providing problems solved by these functions in a learnable progression. The main criterion of excellence in a training sequence: it enables the system to solve many problems outside the training sequence itself (out of sample data). To do this the problems in the sequence should be solved using a relatively small number of powerful functions. Designing training sequences of this sort is a crucial and challenging problem in the development of strong intelligence.

The system must also be able to recognize the context in which each function should be applied. This, however is a task for the guiding probability distribution.

In most ways, designing a training sequence for an intelligent machine is very similar to designing one for a human student. In the early part of the sequence, however, there is a marked difference between the two. In the early training sequence for a machine, we know *exactly* how the machine will react to any input problem. We can calculate a precise upper bound on how long it will take it to solve early problems. It is just

$$T_i/P_i \tag{1}$$

P_i is the probability that the machine assigns to the solution known by the trainer. T_i is the time needed to test that solution. I call this upper bound the "Conceptual Jump Size" (CJS). It tells us how hard a problem is for a particular machine — a measure of how long we expect that machine will take to solve it. I say "upper bound" because the system may discover a better, faster, solution than that known by the trainer.

This CJS estimate makes it easy to determine if a problem is feasible for a system at a particular point in its education. The P_i for a particular problem will vary during the life of the system, and for a properly constructed training sequence it should increase as the system matures. This increase in P_i can be used as a rough measure of the machines "rate of learning".

Eventually in any training sequence for a very intelligent machine, the trainer will not be able to understand the system in enough detail to compute CJS values. The trainer then treats the machine as a human student. By noting which problems are easy and which are difficult for the machine, the trainer infers which relevant functions the machine has learned and which it has *not* learned and devises appropriate training problems.

Learning to train very intelligent machines should give useful insights on how to train human students as well.

There are at least two ways to write training sequences: "Bottom Up" and "Top Down". The Bottom Up approach starts with some simple problems that have easy solutions in terms of the primitives of the reference language. The next problems in the sequence have solution functions that are simple combinations of functions the machine has already learned. This increase in complexity of problem solutions continues to the end of the training sequence.

In Top Down training sequence design, we start with a difficult problem that we know how to solve. We express its solution as a function mapping input to output. This function is then decomposed into simpler functions, and we design problems that are solvable by such functions. These functions are in turn factored into simpler functions and again we devise problems that are solvable by such functions. This breakup of functions and designing of problems continues until we reached the primitive functions of the reference language. The desired training sequence is the set of problems designed, but we present them in an order reversed from that in which they were invented.

The functions themselves form a partially ordered set. Function F_1 *is greater than* function F_2 if F_2 is used to create F_1.

For more details on how to construct training sequences of this kind see (Sol89).

So far I've mainly worked on elementary algebra, starting with learning to evaluate algebraic expressions and solving simple equations — this can be continued with more complex equations, symbolic differentiation, symbolic integration etc. This list can go on to problems of arbitrary difficulty.

A promising source of training material: learning the definitions of the various operations in Maple and/or Mathematica.

Another possible source of training sequence ideas is "A Synopsis of Elementary Results in Pure and Applied Mathematics", a book by G. S. Carr. It was the principal source of information about mathematics for Ramanujan — one of the greatest creative geniuses of recent mathematics. His style was one of imaginative inductive leaps and numerical checking — much in the manner of how we would like the present system to operate.

After the machine has an understanding of algebra, we can train it to understand English sentences about algebra. This would not include "word problems" which typically require knowledge about the outside world.

It cannot be emphasized too strongly, that the goal of early training sequence design, is *not* to solve hard problems, but to get problem solving information into the machine. Since Lsearch is easily adapted to parallel search, there is a tendency to try to solve fairly difficult problems on inadequately trained machines. The success of such efforts is more a tribute to progress in hardware design then to our understanding and exploiting machine learning.

In Conclusion

We have a method for designing training sequences. We have a method for updating the guiding probability distribution. We have a method for detecting/measuring "learning" in the system.

These three techniques are adequate for designing a true AGI.

If the system does not learn adequately, the fault is in either the training sequence (the conceptual jumps needed are too large) — or that the update algorithm may not be able to recognize important kinds of functions that occur in the training sequence and know under what conditions they should be used — in which case we must modify the training sequence and/or the update algorithm. The update algorithm must be designed so that it can readily spot functions used in the past that are relevant to current problems.

What we have is a recipe for training an intelligence system, and a few ways to debug our attempts at training it. The system itself is built on ALP, which is certainly adequate to the task. Our understanding of much of our own human learning will probably be inadequate at first. There will be conceptual leaps in the training sequences which we don't know how to break down into smaller jumps. In such cases it may be necessary to practice a bit of "Brain Surgery" to teach the machine — direct programming of functions into the reference language. Usually we will try to avoid such drastic measures by simplification of problems or by giving auxiliary related problems — by giving "hints".

We have mentioned the possibility of merging the guiding probability distributions of different systems created by independent investigators. It would be well if there were several research groups working on systems of the type described, with enough similarity in the reference languages and update algorithms so that the guiding probability distributions could, indeed, be merged.

The system we have described will do fairly general kinds of prediction. It can be regarded as Phase 1 of

a larger project. Phase 2 (Sol03) — is built on Phase 1 and is designed to solve even more general kinds of problems. In particular, it is able to work time-limited optimization problems — for example, "Get as good a solution as you can to this traveling salesman problem in 10 minutes". Most practical problems in science and engineering are of this kind. This includes the problem of improving the GPD of Phase 1 — enabling the system to significantly improve itself.

Appendix A: Levin's Search Procedure

It would seem that evaluating a very large number of functions would take an enormous amount of time. However, by using a search technique similar to one used by Levin for a somewhat different kind of problem, it is possible to perform the search for acceptable functions in something approaching optimum speed. It may occasionally take a long time to find a very good solution — but it is likely that no other search technique with equivalent education and hardware could have found that solution any faster.

How the procedure works: Suppose we have an input string, I_1 and an output string, O_1. We also have a probability distribution p_j over all possible functions, F_j and we want to find high probability functions, F_j, such that $F_j(I_1) = O_1$.

We could simply apply many random functions to I_1 and watch for functions that meet our requirements. This would take a lot of time. There is,however, a much more efficient way:

We select a small time limit, T, and we test all functions, F_j such that

$$t_j < Tp_j \tag{2}$$

Here p_j is the probability of the function being tested, and t_j is the time required to test it. The test itself is to see if $F_j(I_1) = O_1$. If we find no function of this sort, we double T and go through the same test procedure. We repeat this routine until we find satisfactory functions. If F_j is one of the successful functions, then the entire search for it will take time $\leq 2t_j/p_j$.

There is a faster, time shared version of Lsearch that takes only time $\leq t_j/p_j$, but it takes much, much more memory.

An important feature of Lsearch is that it is able to deal with trials that do not converge — i.e. for $t_j = \infty$

Appendix B: Frustrating Computable Probability Distributions

Given a computable probability function μ, I will show how to generate a deterministic sequence (i.e. probabilities are only 0 and 1)

$Z = Z_1 Z_2 Z_3 \cdots$

to which μ gives probabilities that are *extremely bad*: they are always in error by $\geq .5$.

Let $\mu(Z_{n+1} = 1|Z_1 \cdots Z_n)$ be μ's estimate that Z_{n+1} will be 1, in view of $Z_1 \cdots Z_n$.

if $\mu(Z_1 = 1|\bigwedge) < .5$ then $Z_1 = 1$ else $Z_1 = 0$

if $\mu(Z_2 = 1|Z_1) < .5$ then $Z_2 = 1$ else $Z_2 = 0$

if $\mu(Z_k = 1|Z_1, Z_2, \cdots Z_{k-1}) < .5$ then $Z_k = 1$ else $Z_k = 0$.

In a similar way, we can construct probabilistic sequences in which μ is in error by $\geq \epsilon$, where, ϵ can have any value between 0 and .5.

References

P. Gács. Theorem 5.2.1. In *An Introduction to Kolmogorov Complexity and Its Applications*, pages 328–331. Springer-Verlag, N.Y., second edition, 1997. by Li, M. and Vitányi, P.

M. Hutter. Optimality of universal bayesian sequence prediction for general loss and alphabet. Technical report, IDSIA, Lugano, Switzerland, 2002. http://www.idsia.ch/~marcus/ai/.

H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, Vol. 304(5667):78–80, April 2004.

J. Poland and M Hutter. MDL convergence speed for Bernoulli sequences. *Statistics and Computing*, 16:161–175, 2006.

J. Rissanen. Modeling by the shortest data description. *Automatica*, 14:465–471, 1978.

J. Schmidhuber. Optimal ordered problem solver. TR Idsia-12-02, IDSIA, Lugano, Switzerland, July 2002. http://www.idsia.ch/~juergen/oops.html.

R.J. Solomonoff. Complexity–based induction systems: Comparisons and convergence theorems. *IEEE Trans. on Information Theory*, IT–24(4):422–432, 1978.

R.J. Solomonoff. A system for incremental learning based on algorithmic probability. In *Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*, pages 515–527, Tel Aviv, Israel, December 1989.

R.J. Solomonoff. Progress in incremental machine learning. TR Idsia-16-03, IDSIA, 2003. Given at NIPS Conference, Dec. 14, 2002, Whistler, B.C., Canada.

R.J. Solomonoff. Machine learning - past and future. Dartmouth, N.H., July 2006. Lecture given in 2006 at AI@50, The Dartmouth A. I. Conference: The Next Fifty Years.

N. Sapankevych and R. Sankar. Time series prediction using support vector machines: A survey. *IEEE Computational Intelligence*, Vol. 4(2):24–38, May 2009.

C.S Wallace and D.M. Boulton. An information measure for classification. *The Computer Journal*, 11:185–194, 1968.

All of Solomonoff's papers and reports listed here are available at http://world.std.com/~rjs/pubs.html