TR IDSIA–16–03, revision 2.0; 30 October 2003

Progress In Incremental Machine Learning

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

Abstract

We will describe recent developments in a system for machine learning that we've been working on for some time (Sol 86, Sol 89). It is meant to be a "Scientist's Assistant" of great power and versatility in many areas of science and mathematics. It differs from other ambitious work in this area in that we are not so much interested in knowledge itself, as we are in how it is acquired - how machines may learn. To start off, the system will learn to solve two very general kinds of problems. Most, but perhaps not all problems in science and engineering are of these two kinds.

The first kind is Function Inversion. These are the P and NP problems of computational complexity theory. They include theorem proving, solution of equations, symbolic integration, etc.

The second kind of problem is Time Limited Optimization. Inductive inference of all kinds, surface reconstruction, and image restoration are a few examples of this kind of problem. Designing an automobile in 6 months satisfying certain specifications and having minimal cost, is another.

In the following discussion, we will be using the term "Probability" in a special sense: i.e. the estimate given by the best probabilistic model for the available data that we can find in the available time.

Our system starts out with a small set of Problem Solving Techniques (PSTs) and a simple General Conditional Probability Distribution (GCPD). When the system is given a problem, the description of this problem is the "Condition" for the GCPD. Its output is a probability distribution on PSTs - the likelihood that each of them will solve the problem by time t. It uses these PSTs and their associated probability distributions to solve the first problem.

Next, it executes its Update Algorithm: The PSTs are modified, new ones may be added, some may be deleted. The GCPD is modified. These

changes incorporate into the system, information obtained in solving recent problems and prepare it to solve harder problems.

The next problem presentation and the system's solution is followed by the corresponding update and so on. After the n^{th} update, the system is usually able to work problems more difficult than the n^{th}. Giving the system a suitable training sequence of problems of progressive difficulty, makes it possible for the system to eventually solve very hard problems.

One critical component in the system is the initial set of PSTs. These include Levin's Universal Search Algorithm (Lsearch). Simple Lsearch is only practical for simple problems, since its solution time is exponential in the complexity of the solution. In the update phase, we modify the probability distribution that is used to guide Lsearch. This effectively reduces the complexity of the solutions of initially difficult problems, so that Lsearch becomes feasible for them.

The update algorithm is another critical component. It has been designed to facilitate "transfer learning", so that the system can utilize any similarities between problems in the same or disparate domains. It has been possible to express this algorithm as an inductive inference problem of a type the system normally solves - so we can simply ask the system to update itself as one of its regular problems. In the early phases of learning, we use a relatively simple update algorithm. Later, the accumulation of data enables us to use a more complex update algorithm that recognizes a broader range of regularities.

Perhaps the most critical component of all is the training sequence - To devise a sequence of problems so that each of them challenges the system to devise new concepts to solve it, yet keeps the challenges within the capacity of the machine. For PSTs that use simple Lsearch, this is not hard to do. It is always possible to get a good estimate of the time needed for the system to find a known solution to a problem. For Lsearch in which the guiding probability distribution is modified during the search, this is not so easy and designing appropriate training sequences can become as difficult as teaching people!

Consider two extreme approaches to designing intelligent machines:

In CYC, Lenat (Len 95) gives the machine much factual information in a very direct manner. The system's learning capability enables it to "interpolate", to "fill in" the knowledge it was not explicitly given.

In our system, the amount of information directly inserted into the machine is kept as small as we can afford. Almost all knowledge is obtained by inductive inference from the sequence of problems given to it by the trainer.

If successful, CYC will have a large, encyclopedic knowledge base which it will use to solve problems. On the other hand, if successful, our system will be very good at discovering new knowledge based on fewer facts and inductive integration of those facts. It has the Feynman-like approach of wanting to find its own way of understanding the data.

The system's generality invites comparison with Genetic Programming. We will discuss differences in the two approaches to optimization problems, and suggest a way in which Genetic Programming might be made a part of the present system.

Another way to look at the system is to think of it as a "Wrapper" system - a system that modifies its parameters in view of its experience with earlier problem solving. The present system is an "*Extreme* Wrapper System" in that - through the Magic of Levin's Universal Search - it is possible for the entire system to effectively replace itself by one that is more efficient.

Contents

Introduction: Description of overall operation of system

We will begin with the description of a simple kind of inductive inference system. We are given a sequence of Q, A pairs (questions and correct answers). Then, given a new Q, the system must give an appropriate answer. At first, the problems will be mathematical questions in which there is only one correct answer. The system tries to find an appropriate function F so that for all examples, $Q_i, A_i; F(Q_i) = A_i$. We look for F functions that have highest a priori probabilities — that have "short descriptions". In generating such functions, we use compositions of primitive functions and of sub-functions that we have previously defined. Sub-function definition is one of the main methods we use to get functions with short descriptions.

At first we only define functions that are solutions to problems. Later, we define sub-functions that have occurred several times in solution functions. We also define functions that specify contexts for functions.

Another important improvement in prediction is obtained by a set of functions that define classes of problems, so we can use a different kind of prediction function for each class.

From deterministic prediction we can graduate to probabilistic prediction. Among other advantages, it allows us to make errors in information given to the system. At first we will use simple Bernoulli prediction, then the most general kind of prediction based on Algorithmic Probability.

While each of these improvements/extensions of the system brings new challenges, perhaps the most difficult design task is the construction of suitable training sequences for the system. We will begin by teaching elementary algebra — first notation, then solutions of linear, then quadratic and then cubic equations. After the system has acquired a fair understanding of part of algebra, we begin to ask questions in a simple kind of English, about topics in algebra that it is familiar with. It is easier to learn a new language when you already know one that has many concepts in common with the one to be learned. In the present case, the common concepts will be the definitions of algebraic entities.

The system described uses a limited form of Lsearch (appendix C) that is particularly effective for induction problems. For the first Q_1, A_1 problem, we use Lsearch to find a function $F_1(\cdot)$ such that $F_1(A_1) = Q_1$, and a_0^1, the a priori probability of $F_1(\cdot)$ is as large as possible. In searching for solution to the second problem, we want a $F_2(\cdot)$ such that $F_2(A_1) = Q_1$ and $F_2(A_2) = Q_2$ and the a priori probability of $F_2(\cdot)$ is as large as possible. However, after we have solved the first problem, our a priori probability distribution changes. It includes the definition for $F_1(\cdot)$. As we solve more problems, the relevant probability distribution used for searching changes in accord with the past solutions.

To solve Inversion and Time Limited Optimization problems we start out using a technique similar to that for induction. In the case of Inversion problems, we are given a string s and a function $G(\cdot)$ that maps strings into strings. The problem is to find as rapidly as possible, a string x such that $G(x) = s$.

Conventional Lsearch searches over functions $F(\cdot,\cdot)$ that look at $G(\cdot)$ and s and generate trial x values as output. i.e. $x = F(G(\cdot), s)$ As in induction problems, Lsearch for Inversion problems uses an a priori probability to guide the search for a suitable $F(\cdot,\cdot)$ function. After a problem has been solved, the solution to that problem modifies the a priori probability for search for a solution to the next problem - just as in induction problems.

After several Inversion problems have been solved, it becomes possible to update the guiding probability distribution in a more effective manner. We have several problems that have been solved, we have the functions that have solved them and we know how long it has taken to solve each of them. From this data we can use our earlier Q, A induction system to get probability distributions, $h_i(t)$ for the time it will take for various $F_i(\cdot|\cdot)$ functions to solve a *new* Inversion problem. From each distribution, $h_i(t)$, we obtain a figure of merit, α_i, that gives an optimum ordering of a trial for its associated F_i function. We then use these $\alpha_i s$ to order our trials.

For Time Limited Optimization problems, we use a similar solution technique. Since induction problems are a kind of Time Limited Optimization problem, we can use this method to greatly improve our solutions to induction problems as well.

1 Inductive Inference: Detailed Operation of Q,A Induction

The problem we are addressing is that of very general probabilistic prediction: We are given an *unordered* set of (Q_i, A_i) pairs, $i = 1...n$. We are then given a new Q_{n+1}. The problem is to find a conditional probability distribution over all possible A_{n+1}. (Here the "i" indices of Q_i and A_i are used to indicate Q_i is associated with A_i. The information in the i ordering, however, is not to be used for prediction.)

The Q's and A's can be strings, and/or numbers. The Q's can be thought of as questions, the A's as answers. Q_i might be the description of a problem, and A_i its solution. A_i and Q_i might be in inputs and outputs of an unknown stochastic process. The Q,A pairs in the set need not all be of the same kind and may be drawn from very disparate domains of knowledge, such as linguistics, mathematics, physics, biology, or economics. Since we are considering stochastic models, the data can have "errors" in it — interpreted by the system as "noise".

We will first give a theoretical solution to the problem, involving infinite time and storage — then a discussion of various practical approximations.

For the data described above, the probability distribution of A_{n+1} is

$$\sum_j a_0^j \prod_{i=1}^{n+1} O^j(A_i|Q_i) \qquad (1)$$

Here $O^j(\cdot|\cdot)$ is the j^{th} possible conditional probability distribution relating its two arguments. $O^j(A_i|Q_i)$ is the probability of A_i, given Q_i, in view of the function O^j

a_0^j is the a priori probability of the function $O^j(\cdot|\cdot)$. It is approximately $2^{-l(O^j)}$ where $l(O^j)$ is the length in bits of the shortest description of O^j. Appendix A discusses the computation of the a_0^i.

We would like to sum over all *total* recursive functions, but since this set of functions is not effectively enumerable, we will instead sum over all *partial* recursive functions, which *are* effectively enumerable.

We can rewrite (1) in the equivalent form

$$\sum_j a_n^j O^j(A_{n+1}|Q_{n+1}) \qquad (2)$$

Here,

$$a_n^j = a_0^j \prod_{i=1}^{n} O^j(A_i|Q_i) \qquad (3)$$

In (2), the distribution of A_{n+1} is a weighted sum of all of the O^j distributions — the weight of each O^j being the product of its a priori probability and the probability of the observed data in view of O^j.

Appendix B shows that even with a relatively short sequence of Q,A pairs, these distributions tend to be very accurate. If we use the a_0^j to express all of our a priori information about the data, they are perhaps the most accurate possible.

Since we cannot compute this infinite sum using finite resources, we approximate it using a finite number of large terms — terms that in (2) have large a_n^j values. While it would seem ideal to include the terms of maximum weight, it has been shown to be impossible to *know* if a particular term is of maximum weight. The best we can do is to find a set of terms of largest total weight in whatever time we have available. *We can completely characterize the Problem of Inductive Inference* to be the finding, in whatever time is available, of a set of functions, $O^j(\cdot|\cdot)$ such that

$$\sum_j a_n^j \qquad (4)$$

is as large as possible.

Genetic Programming, using a Lisp–like functional language, (Cra 85, Ban 98) would seem to be a reasonable way to solve this problem. A large population of O^j functions could be mutated and/or recombined with suitable crossover algorithms using (4) as fitness function. A first glance suggests that while this may give very accurate probability distributions, it would be *much* slower than the system we will subsequently describe. Section 7.3 discusses techniques that we expect to significantly increase speed of Genetic Programming.

In 1973 L. Levin suggested (Lev 73, LiV 93, LiV 97) a universal method for solving sequential search problems that can be readily applied to the problem of finding functions of maximum weight. If there are a small number of simple problems, Levin's method (which we will call "Lsearch") can be a practical way to solve them. For more difficult problems and/or a large collection of easier problems, the method is too slow to find O^j functions of much weight *directly.* Lsearch can, however, be used as part of a more complex method with a much more acceptable search time.

One method of this sort expresses the set of O^j being used as a combination of two types of functions. The first type of function, $R^i(Q_l)$, recognizes what *kind* of problem Q_l is. Associated with each recognition function $R^i(\cdot)$, is a set of one or more of the second type of function, P_i^j , that solves problems of the i^{th} *kind.* For each value of n, (the number of Q, A pairs thus far), there will be w_n different kinds of problems. Usually $w_n << n$, since many problems will be of the same kind.

There are w_n recognition functions, R^i. Each of them recognizes one and only one problem type. If $1 \leq l \leq n$ then both Q_l and A_l are known so

$i = 1...w_n$ and $w_n \leq n$.

$R^i(Q_l) = 1$ if Q_l is of type i.

$R^i(Q_l) = 0$ if Q_l is not of type i.

If $l \leq n$ the R functions do not overlap i.e.:

$R^j(Q_l) = 1$ AND $R^k(Q_l) = 1$ implies $j = k$.

The second type of function $P_i^k(\cdot|\cdot)$ is a probability distribution associated with R^i. $P_i^k(A|Q_l)$ gives the conditional probability (as seen by P_i^k) of the answer A , to question Q_l.

If $R^i(Q_l) = 1$ then Q_l is a problem of the i^{th} kind, and

$$\sum_{k=1}^{k_i} a_i^k P_i^k(A|Q_l) / \sum_{k=1}^{k_i} a_i^k \tag{5}$$

is the conditional probability induced by the set of functions P_i^k on the set of all possible answers A, for question Q_l .

a_i^k is the a priori probability of the function, P_i^k

k_i is the number of prediction functions, P_i^k for the i^{th} kind of question.

$\sum_{k=1}^{k_i} a_i^k$ is a normalization factor.

More generally, if Q_{n+1} is a new problem/question never seen by the system before, its probability distribution on possible answers, A will be

$$\sum_{i=1}^{w_n} \sum_{k=1}^{k_i} R^i(Q_{n+1})\ a_i^k\ P_i^k(A|Q_l)/ \sum_{i=1}^{w_n} \sum_{k=1}^{k_i} R^i(Q_{n+1})\ a_i^k \tag{6}$$

If $R^i(Q_{n+1}) = 1$ for one and only one value of i , then expressions (5) and (6) are the same, except that $l = n + 1$.

If $R^i(Q_{n+1}) = 1$ for more than one value of i, then (4) gives a mean over the various *kinds* of problems Q_{n+1} seems to be.

If $R^i(Q_{n+1}) = 0$ for $i = 1...w_n$ then the system does not recognize Q_{n+1} and (6) is undefined — zero divided by zero.

1.1 Early Training: Operation of system at beginning of Training Sequence

In the early training of the system, we allow only one R — so there is only one *kind* of problem. It is possible to do a fair amount of useful learning within this constraint. Another simplifying constraint, is that $O^j(\cdot \mid \cdot)$ take on the values 0 and 1 only.

This corresponds to a kind of deterministic induction. It can be used for training sequences in parts of mathematics in which there is only one correct answer to a problem. Here $A_i = H(Q_i)$, but we do not know $H(\cdot)$. We want to find an F^j such that for all i,

$$A_i = F^j(Q_i) \tag{7}$$

and a_0^j, the a priori probability of $F^j(\cdot)$, is as large as possible.

To start off, we have only one Q_1, A_1 pair. We will show how Lsearch can be used to find a function F^1 such that $F^1(Q_1) = A_1$, and a_0^1, the a priori probability of F^1, is as large as possible.

Using the AZ language of Appendix A, we first add Q_1 to the argument list, so the output can be a function of Q_1. Then we do an exhaustive search of all possible codes for F^1, such that the time to generate and test F^1 is less than $a_0^1 \cdot \tau$, a_0^1 being the apriori probability of F^1, and τ being the time needed to execute a small number of computer operations. [The value of τ is not critical]. If no suitable F^1 is found, we double τ and repeat the search. This doubling and testing is repeated until we find a F^1 such that $F^1(Q_1) = A_1$. If time is available[1] , we continue doubling for a while longer (*oversearching*) in the hope that we will find F^1's with larger a_0^1's. If we do, we preserve in memory all of the F_j^1 functions that work, even though most of them may have a smaller a_0^1 than that of the best F^1 found. If we have only one satisfactory F^1 when Q_2 arrives, our guess for A_2 is $F^1(Q_2)$. If there is more than one F^1 in memory

[1]An upper bound on available time is given by the trainer as part of the description of the induction problem.

$(F_j^1 : j = 1...k)$, then we may have several predictions: The relative probability of the prediction $F_j^1(Q_2)$ is $a_{0,j}^1$, the a priori probability of F_j^1.

When the true A_2 is revealed, we don't have to do much updating if $F_j^1(Q_2) = A_2$ for any of j. We simply remove all F_j^1's that don't work. If no F_j^1 works, then we have to do more serious updating.

To do this, we remove Q_1 from the argument list and replace it with Q_2. We add to that list, the best F_j^1 that we've found. This enables us to build upon the functions we've found useful in the past. We then do an Lsearch to find a function with A_2 as output — if we do, we replace Q_2 by Q_1 in the argument list and see if the output is A_1. If it is, we have a successful F^2; if not, we continue the search, with Q_2 in the list rather than Q_1. We continue searching until we find a F^2 such that $F^2(Q_1) = A_1$ and $F^2(Q_2) = A_2$.

This routine continues as we update $Q_3, A_3; Q_4, A_4$.... Each time we find an F that works with all of the examples up to present, we put that F in the argument list.

In updating, we *backtrack* as little as possible. Suppose F^n works for $Q_i, A_i; i = 1...n$, but $F^n(Q_{n+1}) \neq A_{n+1}$. With $F^1, F^2...F^n$ in the argument list, we do a search for an F^{n+1} that will work with $Q_1, A_1...Q_{n+1}, A_{n+1}$. If we can't find one, we backtrack one level and try a search with $F^1, ...F^{n-1}$ in the argument list instead. If unsuccessful, we backtrack two levels, remove F^{n-1} from the argument list, and try again, etc.

Keeping in memory many alternative non-optimum solutions to problems obtained by "oversearching" has an effect similar to that of backtracking. They both give useful alternatives when the present trials fail. Of the two techniques backtracking, is much slower, but can be more exhaustive of possibilities.

Having many alternative non-optimum codes gives us a way to estimate the probability of a prediction.

1.2 Updating: How system is modified after problems are solved

The forgoing system must be augmented considerably if we have several R^i and associated P_i^k. A critical aspect of system operation is that of "updating". When we are given a new question, answer pair $(Q_{n+1}\ ,\ A_{n+1})$, how is the system modified? How do we update the sets of functions R^i and P_i^k so that the system will respond accurately to Q_{n+1}, continue to respond well for $Q_1...Q_n$ and be likely to respond well to (unknown) future questions?

There are five possible kinds of response of the system to the new Q_{n+1},A_{n+1}. Each has its own updating algorithm.

1. There is one and only one i such that $R^i(Q_{n+1}) = 1$ and $P = \sum_{k=1}^{k_i} a_i^k P_i^k(A_{n+1}|Q_{n+1}) / \sum_{k=1}^{k_i} a_i^k$ is acceptably large.[2] In this case

[2]The criterion for whether P is "large enough" will change during the course of training of the system. At first, the "largeness thresholds" will be given by the trainer as part of

the response of the system is satisfactory and no updating is needed.

2. Same conditions as 1, except that P is too small. In this case there are at least two ways to update:

2a. Modify R^i so the new $R'^i(Q_i) = R^i(Q_i)$ for $i \leq n$, but $R'^i(Q_{n+1}) = 0$

Devise a new R^{w_n+1} such that $R^{w_n+1}(Q_i) = 0$ for $i \leq n$ but $R^{w_n+1}(Q_{n+1}) = 1$

Find a set of $P^k_{w_n+1}$ functions such that $\sum_k a^k_{w_n+1} P^k_{w_n+1}(A_{n+1}|Q_{n+1})$ is as large as possible.

2b. Keep the same $R^i(\cdot)$, but modify the associated P^k_i functions, so that

$$\sum_{k=1}^{k_i} a^k_i \prod_l P^k_i(A_l|Q_l) / \sum_{k=1}^{k_i} a^k_i$$

is as large as possible. (Here l ranges over those values of l for which $R^i(Q_l) = 1$). This method of updating is usually better than method 2a in terms of future system performance, but takes more time.

3. $R^i(Q_{n+1}) = 0$ for $i = 1...\omega_n$.

Find a new R^{w_n+1} such that $R^{w_n+1}(Q_{n+1}) = 1$ but $R^{w_n+1}(Q_l) = 0$ for $l \neq n+1$.

Find a set of functions, $P^k_{w_n+1}$ such that

$$\sum_k a^k_{w_n+1} P^k_{w_n+1}(A_{n+1}|Q_{n+1})$$

is as large as possible.

4. $R^i(Q_{n+1}) = 1$ for more than one value of i , and at least one of the recognizers is associated with a set of P^j_i functions that give an adequate P for A_{n+1}.

Modify the other recognizers so they no longer recognize Q_{n+1} but otherwise behave the same.

5. $R^i(Q_{n+1}) = 1$ for more than one value of i, but none of the associated P^j_i functions gives an adequate P for A_{n+1}.

There are two ways to update in this case:

5a. Modify the R^i's that recognized Q_{n+1} so they no longer recognize Q_{n+1}, but otherwise behave the same. We are now in "Update situation 3" and can proceed accordingly.

5b. If there are two or more R^i such that $R^i(Q_{n+l}) = 1$, then the P^j_i's associated with those R^k_i's are likely to have important features in common. We can take advantage of this "commoness" if we can find a single set of P^i_m functions that is able to solve all of the problems formerly requiring many P^i_k

each problem definition. In later stages of training, the system itself will be able to induce reasonable thresholds — based on its own experience and its goal of maximizing the value of equation(4) for the entire data set.

sets. This would reduce the description length of O^j, and increase its a priori probability, thereby increasing the value of expression (1).

While this method of updating is usually very time-consuming, it is a very desirable kind of update since it "merges" P^i_k functions and it also merges the associated R functions. These operations can give a very large increase to equation (4). They can also speed up considerably both recognition and prediction.

This large increase in (4) resides mainly in the a priori probabilities of the O^j functions. Each such function consists of w_n sets of R^i and associated P^j_i functions. The description length of O^j is approximately the sum of the description lengths of the R^i and the P^j_i. By "merging" the functions, we reduce the number of functions that need to be described — which usually increases the a priori probability of O^j.

Discovery of new R functions corresponds to the *Analysis* phase of scientific inquiry - breaking a domain into smaller parts that can be dealt with using techniques peculiar to those parts. Merging of the R functions in update method 5b corresponds to *Synthesis* - the realization that several apparently disparate domains really have much in common and can be treated in a unified manner.

The history of science can be viewed as a continuing sequence of Analysis and Synthesis operations.

In the foregoing we have allowed $R(Q_l)$ to be 0 or 1 only, with no overlap of the R functions. It is also possible to have a system in which $R(Q_l)$ is a real between 0 or 1 — and overlap of R's are acceptable. We will not discuss such systems in this paper.

1.3 Scaling: Time needed to solve successive problems normally increases very rapidly. How to prevent this

As we increase the number and/or variety of our problems, the search times for the simple system just described, increase very rapidly. Suppose we have just k functions and constants when we begin our training sequence. The probability of each code element is k^{-1}. If we add one F^i to the list its probability becomes $(k+1)^{-1}$. If we add one function F^i to the list each time we solve a new problem, the probability of a single code element will be roughly $(k+n)^{-1}$ after we solve n problems ("roughly", because the probabilities of various code elements will usually change during problem solving). In Lsearch the total search time is proportional to the time needed to generate and test an acceptable solution, divided by the a priori probability assigned to that solution. For $k = 10$, $n = 20$ and as little as 5 symbols in the solution, we will have a search time of about $(10 + 20)^5 \approx 2.4 \times 10^7$ times the processing time of the solution.

This would be the factor needed if we only needed to find 5 symbols to add on to the common solution to the first 20 problems. If we have to backtrack, we retain the common solution to the first 19 problems, but we will usually need more than 5 symbols to find a solution including two new problems. — perhaps

10 symbols would be needed — giving a factor of about $(10+19)^{10} \approx 4.2 \times 10^{14}$, which begins to get *somewhat* unmanageable. With more backtracking or a larger number of problems, it becomes *very* unmanageable.

In the forgoing analysis, we have used "ball park" estimates. In an empirical study of Algebraic Notation Learning, using the system described, without backtracking, we did, indeed, find that the search time increased very rapidly with n.

There are several devices we can introduce to deal with this problem. First, the use of l different recognition functions, R^i, effectively reduces the value of n (the number of problems) by a factor of about l.

Another great reduction in search time can be obtained by "context discovery". In our definition of the AZ language, the probability of a particular symbol of any point in a function description is independent of the recent previous symbols, and is independent of the kind of problem or subproblem being solved. By introducing "context recognizers" we can take advantage of the fact that in certain contexts, certain symbols are much more likely than others. Suitably designed contexts can greatly increase the probabilities of certain symbols and greatly increase the probabilities of correct problem solutions — thus decreasing their associated search times.

How do we define contexts? The simplest kind of context is the preceding symbol. In prediction of English text, the "space" symbol is very important because the probability distribution of symbols following it are quite different from the allover average probability distribution.

More generally, any subset of postfixes of a string of symbols may be regarded as a kind of context for the following symbol. Examples of contexts of this kind: the last symbol is "sum" ; the last two symbols are the same; the entire sequence has an odd number of symbols; the number of symbols is a prime number, ...

It will be noted that the R (recognition) functions previously described are one way of implementing one kind of context.

Context can be expanded beyond the immediate code string. We can include the problem being solved (Q_{n+1}, A_{n+1}) and/or any number of previous QA pairs. We might want to include previous solutions to problems, if this information is not available in the earlier part of the function being coded. In human problem solving, a more global context is quite important. We want to know how and where the problem arose — from chemistry, physics, math... These kinds of contexts are important in machine learning as well, and we must find ways to give the system access to such information.

In the solving of mathematics problems by humans, a common heuristic is to *abstract* the problem – to remove apparently irrelevant features and retain as few features as are necessary to define the problem adequately.

For machines, this heuristic must be inverted. The problems given to the machine are often *already* abstracted. It is necessary for the trainer to *restore* any useful contextual information. This contextual information tells the machine what areas to explore to find problem solving techniques that have been successful in past problems similar to the current problem.

One way to give contextual information of this sort is to give the Q's "indices": each Q can have associated with it one or more symbols that convey contextual information. One symbol type might tell if the problem is a math or physics or biology problem. Another might tell if the problem involves differential equations – and if so, what kind, etc. The system will then learn to correlate these indices with problem solving techniques. As the machine matures, the trainer will be able to omit some of the indices – the machine will be able to induce them from the data it received in its earlier training.

One value of the indices is in "Transfer Learning". All of the data used in induction problems can be regarded as one large "corpus" (body of data). At first, it may be easier for the system to consider "sub-corpora" — data that have the same indices on their Q's. Each can be regarded as a separate training sequence, and they can be solved independently. It is, however, better to regard all of the data as part of the same large corpus, so that the system can use common functions in the inductions on sub-corpora to reduce the total number of bits to describe the data, thus materially increasing the value of equation 1. This "sharing of functions"; can be used to link any data in the system. What we have is an effective General Conditional Probability Distribution (GCPD) for all of the inductive data in the system. In sections 2 and 3, we will discuss Inversion problems and Optimization problems. Though they are not induction problems, they have probability distributions that guide the searches for solutions to their problems. These probability distributions will also be included in the GCPD.

2 Inversion Problems: Solution of P and NP Problems

In Inversion problems, we are given a string, s, and a function $G(\cdot)$ from strings to strings and we are asked to invert G with respect to s: to find an x such that $G(x) = s$. We want to do this as fast as possible.

2.1 Solution by Lsearch

One way to solve such problems is to use Lsearch[3]. In one simple kind of Lsearch, we look for a Problem Solving Technique (PST) in the form of a function, F_i that maps the description of $G(\cdot)$, and the target string, s, into a trial string x_i.

$$F_i(\widetilde{G}, s) = x_i$$

$\widetilde{G}$ is a string that describes $G(\cdot)$. It is usually a program. We can use the language AZ of appendix A, to obtain a probability distribution, $P_i = P_0(F_i)$, on all possible $F_i(\cdot, \cdot)$.

[3]Appendix C discusses different kinds of Lsearch.

From $P_0(\cdot)$ and $G(\cdot)$ and s, we can use Lsearch to obtain a solution, $x_i = F_i(\widetilde{G}, s)$, such that $G(x_i) = s$, in the following way:
Suppose τ is the time needed for a small number of machine instruction (τ is not a critical quantity). We try all possible F_i's, using a maximum time of τP_i to generate F_i and x_i and test x_i. This takes total time $\sum \tau P_i$ which is $\leq \tau$ since $\sum P_i \leq 1$. If we do not find a solution, we run all the trials again, using 2τ as our time limit. We keep searching with doubling and redoubling of our time limit until we find a solution. The total time for all of the trials will be $\leq 2t_l/P_l$.
P_l is the probability assigned to the F_l that solved the problem.
t_l is the time needed to generate and test the correct solution.

After we solve the first problem, we put the definition of F_i in the "argument list" of the language used to generate trials.

When we have solved several problems in this way, the language will have several useful functions in its argument list. It will usually be possible to find common subfunctions that can be defined, to increase the probabilities of past solutions to problems, just as was done in the solutions to the induction problems of section 1. In general, all of the techniques for increasing the probabilities of solutions of induction problems in section 1, can also be used for helping to solve Inversion problems.

In section 1 we wanted solutions of maximum probability. For Inversion problems, we also want solutions of high probability since search time is inversely proportional to probability of solution, and high probability solutions will tend to give us the minimal search times we are looking for.

2.2 How the system uses its optimization capabilities to improve its update and search methods

In the method of solving problems just described, the system does not "know" that it is supposed to try to find fast solutions. It simply tries to imitate PSTs or subfunctions of PSTs that have been successful in the past. Though this method works to some extent, it can be much improved.

From previous experience with various PSTs on various problems,we find F_l to be the most promising PST for the present problem, G_n. We apply F_l to G_n for a short time, then in view of its degree of success or lack of success,we revise our estimate as to which is the best PST for G_n. We then apply this "revised" PST (which may or may not be the same as the previous F_l) to G_n for a short time. This procedure is repeated until we solve the problem or until our time runs out.

The first question is – given a corpus of historical data on the results of various PSTs working on various problems, how can we find the PST that is most promising for the present problem?

Consider the set of quadruples: $\widetilde{G}_j$, s_j, $F_l(\cdot,\cdot)$, $t^{j,l}$.

$\widetilde{G}_j$ is a string that describes the j^{th} function to be inverted.

s_j is the argument for the inverse of G_j.

$F_l(\cdot,\cdot)$ is the function of $\widetilde{G}_j$ and s_j that solved the problem.

$t^{j,l}$ is the time it took for $F_jl(\widetilde{G}_j, s_j)$ to generate and test the correct solution to the j^{th} problem.

We have a quadruple like this for each problem solved.

For the new problem, $(\widetilde{G}_n\ ,\ s_n)$, given any $F_k(\cdot,\cdot)$ we can use the technique of the previous section on induction, to get a probability density distribution of the time it will take for F_k to solve that problem. Here, we regard $(\widetilde{G}_j,\ s_j,\ F_l(\cdot,\cdot))$ as $Q_{j,l}$ and $t^{j,l}$ as $A_{j,l}$.

Using Lsearch, we look for probability density functions O^i such that

$$a_0^i \prod_{j,l} O^i(t^{j,l}|(\tilde{G}_j, s_j, F_l(\cdot,\cdot))) \tag{8}$$

is as large as possible.

a_0^i is the a priori probability of O^i and the j,l product is over the known $Q_{j,l} = (\tilde{G}_j, s_j, F_l), A_{j,l} = t^{j.l}$ pairs.

Note that (8) uses only information from *successful* attempts to work problems. This expression must be modified to include information from $failures$ as well. Before explaining how to do this — some simplification of notation:

Let

$$h^{i\,\prime}_{j,l} = O^i(t^{j,l}|(\widetilde{G}_j, s_j, F_l))$$

This is the probability density (according to O^i) that PST,F_l will solve the problem $(\widetilde{G}_j, s_j)$ at time $t^{j,l}$.

Let

$$h^i_{j,l}(t) = \int_o^t h^{i\,\prime}_{j,l}(t^{j,l})dt^{j,l}$$

This is the probability (according to O^i) that by time t, F_l will have solved the problem $(\widetilde{G}_j, s_j)$.

$1 - h^i_{m,k}(t)$ is then the probability (according to O^i) that F_m has $failed$ to solve $(\widetilde{G}_k, s_k)$ by time t.

Consider the expression:

$$a_o^i \prod_{j,l} h^{i\,\prime}_{j,l}(t^{j,l}) \prod_{m,k}(1 - h^i_{m,k}(t^{m,k})) \tag{9}$$

The first product is over j,l pairs in which various F_l's have been *successful* at time $t^{j,l}$, as in equation(6).

The second product is over m,k pairs in which F^k has *failed* to solve problem m, by time $t^{m,k}$.

We want to find an O^i such that equation (9) is as large as possible — the O^i that makes most likely, the observed successes and failures.

When we find a suitable O^i, then for new problem $(\widetilde{G}_n,\ s_n)$ and arbitrary $F_l(\cdot,\cdot)$ we can obtain a probability density, $h^{i\prime}_{n,l}(t)$, that $F_l(\cdot,\cdot)$ will solve that problem at time t. This O^i becomes part of the updated GCPD (General Conditional Probability Distribution) and can be used to guide Lsearch. While it is, indeed, possible to run an Lsearch this way, we will describe a search technique that seems to be much faster than Lsearch.

Given a problem, $(\tilde{G}_n, s_n)$ and a good O^i function, there are an infinite number of $F(\cdot,\cdot)$ functions for which O^i obtains associated $h(t)$ distributions. We need a criterion for "utility" of a $h(t)$, as well as a means for finding a $h(t)$ that is optimum with respect to that criterion.

Consider $h(t)/t$. It gives us the probability of success per unit time expended. For each h, denote by α,the largest value of this ratio, and by t_α the time at which this occurs, so $\alpha = h(t_\alpha)/t_\alpha$.

In this particular situation, the first Gambling House Theorem[4] suggests that we will minimize expected total solution time if we schedule our F_l trials so that their associated $h(t)$'s are in α order: largest values first.

We say "suggests", because the Theorem assumes that the success probabilities of the trials are uncorrelated — that when one trial fails, the probabilities of success of all other trials do not change. In the present case the probabilities *are* correlated and we have modified our search procedure to take advantage of these correlations to speed up our search.

Once we have a good O^i, obtaining a set of PSTs of high α value is a time limited optimization problem that is solvable by the techniques of section 3.

To solve our original inversion problem, we first try the PST of largest α. If it has not been solved by time t_α, we reoptimize equation (9) using the additional information that up to time t_α the present PST has failed. If the reoptimization tells us that continuing to work on this PST still gives the best α, we continue using it — otherwise we switch to a more promising PST. This alternation between our use of the best PST and revision of our choice of best PST continues until the problem has been solved or until we have run out of time.

[4] "At a certain gambling house there is a set of possible bets available — all with the same big prize. The k^{th} possible bet has probability p_k of winning and it costs d_k dollars to make the k^{th} bet. All probabilities are independent and one can't make any particular bet more than once. The p_k need not be normalized.
If all the d_k are one dollar, the best bet is clearly the one of maximum p_k. If one doesn't win on that bet, try the one of next largest p_k, etc. This strategy gives the least number of expected bets before winning.
If the d_k are not all the same, the best bet is that for which p_k/d_k is maximum. This gives the greatest win probability per dollar.
Theorem I: If one continues to select subsequent bets on the basis of maximum p_k/d_k, the expected total money spent before winning will be minimal.
In another context, if the cost of each bet is not dollars, but time, t_k, then the bet ordering criterion p_k/t_k gives least expected time to win". (Sol 86 Section 3.2)

It will be noted that the forgoing technique is not at all, "Lsearch". In fact, it seems to overcome a serious deficiency of Lsearch. If there are many trials that are identical or nearly identical, but which have different descriptions of the same length, Lsearch, will test *all* of them — which is quite wasteful. The technique just described will usually test only one of them — when a candidate is abandoned because its α has been reduced and it is no longer maximum, then usually candidates that are identical or very similar to it, are also abandoned because of small α.

3 Time Limited Optimization Problems

In Time Limited Optimization, we have a function $G(\cdot)$, from strings to reals, and we have time limit t. We must find an x in time $\leq t$ such that $G(x)$ is as large as possible. More generally, the string, x, can represent a number.

3.1 Solution by Lsearch

These problems can be solved using Lsearch, much like the Inversion problems of section 2:

We have a set of PSTs, $[F_i]$, that look at the problem $(G(\cdot),\ t)$ and after a requested time t_i present a solution candidate, $x_i = F_i(\widetilde{G}(\cdot), t_i)$, which is evaluated as $G(x_i)$.

We have, as for Inversion problems, an a priori probability distribution $P_0(F_i)$ that initially, is the same for all Time Limited Optimization problems. We run the generation and testing of F_i and x_i for time $t \cdot P_0(\widetilde{F_i})$. Since $\sum_i P_0(\widetilde{F_i}) \leq 1$ the entire process takes $\leq$ time t. The x_i of maximum $G(x_i)$ is then selected as output. If there is remaining time, we can rerun the tests using times $2TP_0(\widetilde{F_i})$ — doubling and redoubling until all of the time is used up.

After we solve the first problem, we put the solution function, F_i, into the argument list of the language AZ used to generate the trials — just as we did with Inversion problems. All of the methods used to improve the guiding probability distribution for Inversion problems can be used with nominal modification in the present case.

3.2 How improvement techniques for Inversion Problems are applied to Optimization Problems

The improved methods of Section 2.2 can be applied to optimization problems as well.

Here we want to find O^js such that

$$a_0^i \prod_{j,\, l} O^i(G^{j,\, l}|(\widetilde{G}_j, t_j, F_l)) \tag{10}$$

is as large as possible.

$(\widetilde{G}_j, t_j)$ describes the j^{th} optimization problem: to find within time t_j, an x such that $G_j(x)$ is as large as possible.

The j, l product is over our history of $G^{j,\,l}, (\widetilde{G}_j, t_j, F_l)$ pairs that have occurred in the past.

Let us define $h^{i\,\prime}_{j,\,l}(G^{j,\,l}) = O^i(G^{j,\,l}|(\widetilde{G}_j, t_j, F_l))$. It is the probability density (in view of O^i) that PST, F_l will find an x within time t_j, such that $G_j(x) = G^{j,\,l}$

After we have found a good O^i function via equation (10), we can use it to obtain an h' function for an arbitrary problem and arbitrary PST.

Suppose we want to solve a new problem, G_m, t_m. Then for every F_l, O^i will give us a probability distribution h', over $G^{m,\,l}$. Since we want $G^{m,\,l}$ to be as large as possible, we will select for our first trial the f_l with a $h^{i\,\prime}_{m,\,l}$ such that its expected $G^{m,\,l}$ value i.e.

$$\gamma^i_{m,\,l} = \int_{-\infty}^{+\infty} G^{m,\,l} h^{i\,\prime}_{m,\,l}(G^{m,\,l}) dG^{m,\,l} \tag{11}$$

is as large as possible.[5]

We apply this most promising F_l to the m^{th} problem for time $t_m/10$ (the factor 10 is not critical). At the end of that time, we reevaluate equation (10) to see if F_l is still the most promising PST. If it is, we continue applying it to the m^{th} problem for additional time, $t_m/10$. If it is not, we apply a more promising PST to the problem. We continue this alternation of applying PSTs and reevaluating them, until all of our time, t_m has been used up.

We are proposing a technique for optimization that requires at least *two new* optimizations! Is anything being gained?

The first new optimization is equation (10). Obtaining a good O^i is useful for not only the present problem, but for all future problems — so it is a burden that is, to some extent, shared by all problems.

The second optimization involves finding the F_l with as large $\gamma^i_{m,\,l}$ as possible.

Since this is a common problem that is solved many times, we will try to find a way to solve it that is fast and effective. In optimizing equation (10) and again in optimizing the $\gamma^i_{m,\,l}$ of equation (11), we will usually be making small

---

[5]The sophisticated statistician will note that while the solution to any optimization problem is invariant if the utility function G is modified by a monotonic, possibly non-linear, transformation, the value of γ in equation (11) will not always be invariant under such a transformation.

Equation (11) is correct only if the utility function G is "linear", i.e. the utility of "$G(X)$ with probability one" is the same as the utility of "$G(X)/p$ with probability p" for all $0 < p \leq 1$.

If G is not linear and no information is available that can give us an equivalent linear utility, then while the solution to the optimization problem is "well-defined", the solution to the "strategy optimization problem" is *not* "well defined".

Fortunately, several important utility functions, such as time and money *are* linear or linearizable.

corrections to a previous optimization — so the process need not take much time.

The techniques of the present section and of Section 2.2 are meant to follow what seem to be common human methods for solving problems of these kinds.

Since the induction problems of section 1 are all Time Limited Optimization problems, we can apply the techniques of the present section to improve their solutions. These improved probabilities give us better $h()$ distributions, which enable us to recursively improve the $h()$ distributions and so on. This can eventually result in extremely good $h()$ distributions and extremely good solutions to both optimization and inversion problems.

4 Universal Distribution on Probability Distributions: Realization techniques

In the QA induction of Section 1, we look for probability distributions, $O^i(A|Q)$ such that equation 2.2 is maximized.

If the induction problem is deterministic (only one possible A for each Q), then we need a probability distribution on deterministic functions, $O^i(Q) = A$. The language AZ (Appendix A) describes a universal distribution of this sort, as does the FORTH-like language used by OOPS[6].

It is often possible to use languages of this kind for probabilistic induction as well. In Sections 2.2 and 3.2 we use the probability distributions $h'(t)$ and $h'(G)$ for updating Inversion and Optimization problems. In both kinds of problems it is often reasonable to assume that $h'(\,)$ is a monomodal distribution.

For Inversion problems, $0 \leq t \leq \infty$, so the Gamma distribution, $h'(t) = ax^r\epsilon^{-bt}$ is a reasonable approximation.

For Optimization problems, $-\infty \leq G \leq \infty$, so the Gaussian distribution, $h'(G) = ae^{-\frac{(G-\mu)^2}{2\sigma^2}}$ is reasonable.

In equation 10 we may set

$$O^i(G^{j,l}) = a\epsilon^{-\frac{(G^{j,l}-\mu)^2}{\sigma^2}}$$

in which a, μ and σ are all functions of $\widetilde{G}_j$ t_j and F_l.

The universal distributions of AZ or of OOPS would be adequate for finding suitable forms for these functions.

For more general induction problems, we need a universal distribution on probability distributions. One way to obtain such a distribution uses a three input universal machine. All three inputs are prefix sets.

The first input is a finite string S, that describes the function.

The second is the finite string, Q, the "question".

[6]OOPS (Sch 02) is a general problem solver that uses Lsearch. See section 7.2 for further discussion.

The third input is a random binary sequence.

For fixed S and Q, we have a machine with random input — inducing a probability distribution on the output, A, just as in the universal probability distribution. In the present case however, the S and Q inputs need not define a universal distribution on the output.

The forgoing formalism describes a universal distribution over all possible probabilistic relations between Q and A. For every describable probability distribution between Q and A, there exists at least one value of S that implements that distribution.

Though it is possible to realize a three input device of this sort using the AZ language, it is easier to implement using FORTH-like languages such as the one used in OOPS.

5 Training Sequences: How to design a sequence of problems to train the system

In an earlier paper (Sol 89) we discussed the design of Training Sequences at some length. The goal was to find a function that would solve a particular problem. In the early training of the system, the trainer would know a solution function for each of the problems presented. This function would be expressible as compositions of subfunctions. The subfunctions were composed of subsubfunctions etc, until we got to the primitive functions of the system. The Training Sequence would first give problems that were solvable by simple combinations of the primitives. These solutions were then usable for the next round of problems that had solutions moving toward the final goal problem. This succession of problem sequences continued until the final function was attained.

An important part of training sequence design involves "factoring" of problems into parts that are useful in themselves. These parts must be further "factored" and so on. This "factoring" may be understood as a kind of inverse of "chunking". We follow the "reusable parts" orientation of Object Oriented programming.

While Training Sequences of this sort *can* solve problems, they can't solve problems of much complexity in any reasonable time, for reasons discussed in section 1.3 on "scaling". Learning subfunctions that build up to the final solution to a problem is an essential part of a Training Sequence, but it is not in itself adequate - we must also deal with the scaling problem. The discovery of relevant contexts is a strong step in this direction.

In section 1, we mentioned use of "context" to speed up problem solving. These contexts *must be learned* and we must design Training Sequences to teach them. "Context" may be regarded as a kind of "catalyst" that makes certain combinations of functions more likely. There are probably many other kinds of "catalysts" used in human learning. We must find them and find ways to teach them to our system.

Finding "catalysts" is closely related to heuristic programming. We have to find observations ("contexts") that make it more likely that we will choose the desired components of a function that solves the problem.

There is a somewhat different direction that training can take. In section 2 on Inversion problems, the system had a number of PSTs that had been successful in past problems and it had to decide what probabilities to assign to them in working on a new problem.

The trainer can greatly augment this set of PSTs by including techniques that the *trainer* has found useful. The system would then learn what kinds of problems to which those new PSTs could be applied. It would correspond to having a student memorize a set of PSTs and learn how to use them.

We would like the student to *understand* the PSTs — how they were constructed and why they work, so the student could invent new PSTs that were more appropriate to new problems.

One way to do this would be to ask the system to regard the set of PSTs as the "acceptable sentences" of a language for which it must find a stochastic grammar — which it could use to extrapolate the set of PSTs. This process would be much facilitated if the trainer first "factored" the PSTs into common useful subfunctions. A very mature system might be able to do this factoring without external help, but in the early development of the system, external aid of this sort would be needed.

6 Efficiency of the System

6.1 Updating: How much time to spend on updating

After the system solves a problem, taking time t, It must update the associated probability distribution. This involves looking for repeated functions or subfunctions in the solution and/or evaluating equations 9 and 10.

Both of these tasks are open–ended in the sense one can spend an arbitrary amount of time on them. How much time should be spent on updating the guiding probability distribution, relative to the time spent using the distribution for Lsearch to solve problems?

A good approximate solution is to spend as much time on updating as one spends on searching. This is within "a factor of two of optimum" in the following sense: Suppose that we have a system that uses a different ratio of update to problem solving time and is better than our system. Then if our system has a clock that is twice as fast as that system and uses equal time for problems and update, it will have results that are at least as good as the other system, since at any time, it will have spent at least as many machine cycles as the other system, for both problem solving and for update.

6.2 Ultimate Limits of System: An argument that there may be no upper limit on performance of system

Normally, in the course of its operation, the system looks for better PSTs. Since it uses a universal language to describe candidate PSTs, there is really no limit on what PST it might consider. It might find a PST that is much better than Lsearch — in which case, it would nominally use Lsearch, but give that PST weight close to one, so that almost all of its time would be spent applying this new PST to problems. It will have effectively replaced Lsearch by the new PST.

7 Related Work

7.1 Lenat's CYC

In its goals, this work is similar to that of Lenat's CYC (Len 95). The main difference is in what we consider to be minimal "common sense" knowledge. Lenat is not sure just what knowledge a newborn child has, so he puts in whatever he thinks might be useful. When it reaches "critical mass", he feels the system will be able to acquire information with little guidance, by reading books and surfing the net.

"How to do induction" is just another set of facts that he inserts into the machine - like "How to do simple logic", and the "meanings" of various words.

We, too, want to make a minimal system that will be able to acquire knowledge without supervision. We feel, however, that "How to do induction" is the main thing that has to be given to the system to start - that the rest can be inserted with less care - we can make mistakes in teaching it, but the system doesn't expect perfect information - it's models of the world are always probabilistic.

When the system acquires new information, it always stores it probabilistically using its own internal language, in the best ways that it can at that time. This "internal language" changes as the system matures. This differs markedly from the way CYC stores information.

A good fraction of the human brain is devoted to processing visual and acoustic signals. Another very large part (the cerebellum) is devoted to physical equilibrium, thermal control, etc. We avoid the need for this kind of "common sense" by initially teaching abstract mathematics - giving the system problems that it must learn to solve. After it has leaned enough algebra to have a fair sized internal vocabulary of algebraic concepts, we begin to teach it to understand English statements and questions about Algebra. Next we teach it to understand English statements about topics slightly different from algebra, beginning a slow journey into a universe that is progressively different from algebra – eventually culminating in stories about events in the real world.

7.2 Schmidhuber's OOPS

A program for machine learning that is perhaps closest to the present system is Schmidhuber's OOPS (Sch 02).

For purposes of comparison we will call the present system "Alpha" and consider two phases of its behavior:

Phase 1 is the Q,A induction system of Section 1 using the AZ language of Appendix A. It can be considered as a "stand alone system" to do induction without the ability to work the Inversion or general Optimization problems described in Sections 2 and 3.

Alpha enters "Phase 2" when it has acquired enough skill in induction to implement the improved updating technique of Section 2.2.

The behavior of Alpha is analogous to the operation of a ramjet engine: A slower, less efficient propulsion system is used to get to critical velocity — at which point the ramjet begins to operate and the initial propulsion system becomes unnecessary. Phase 1 is needed only to get to Phase 2 — at which point the updating techniques of Phase 1 are used less and less.

A variety of induction systems would be adequate for Phase 1. Genetic Algorithms, certain types of Neural Nets, SVM's, and a modified form of OOPS are some possibilities.

OOPS and Phase 1 of Alpha are very similar. They both use universal distributions to solve problems using Lsearch [7]

The language used by OOPS is a kind of stack language – related to FORTH. The language used by Alpha is AZ, similar to LISP. Both of these languages can use definitions to compress code — an essential feature in incremental learning. AZ is a functional language: functions are represented by trees, and all subtrees are legal functions. In AZ, finding common subtrees in a large function enables compression. In the language used by OOPS we guess that it is less likely that an analog of this technique would often be useful.

Both systems use incremental learning to update the probability distribution that guides Lsearch. While the update systems are similar, Alpha uses contexts of various kinds to deal with scaling effects.

We are uncertain as to how OOPS approaches scaling — though apparently, it *is* a serious problem. OOPS was able to solve the general "Towers of Hanoi" problem about 1,000 times faster, by using a search pattern (a kind of probability distribution over problem trials) from the solution to an earlier problem. Since there weren't many "earlier problems", it was relatively easy to find the appropriate pattern. If the system had solved 1,000 problems before the "Towers of Hanoi", finding the correct pattern may have taken 1,000 times as long

[7] Schmidhuber's terminology is incorrect: what he terms "Osearch" is really Lsearch; what he terms "Lsearch" is really "SIMPLE", a minimally complex program devised by Li and Vitányi to illustrate an important characteristic of Lsearch — i.e. its ability to solve all solvable inversion problems within a constant factor of the speed of an optimum solution. The "constant factor" for SIMPLE is however, *much* larger than that for Lsearch.

— thus cancelling out the gain in search time.

Another difference is that Alpha is largely a theoretical system. The behavior of a variant of Phase 1, using a stack–based language (much like OOPS) has been analyzed for the learning of evaluation of algebraic expressions (Pau 94). Scaling effects were observed when we computed the CJS's of problems of increasing difficulty. While some analysis was made of more difficult problems, no CJS values were computed.

OOPS, on the other hand, *has* been realized as a computer program — demonstrating the importance of early learning in facilitating solutions of difficult problems.

A critical aspect of OOPS is its ability to "edit" old programs — break them up and reassemble the parts to make new promising trials. In the present system there are only a few such "editing" instructions, and only one, boostq, has been shown to be useful. Certainly more editing instructions will have to be added and the system will have to learn to use them. While OOPS does have facilities for "noticing" that certain instructions have been more useful than others, as well as facilities for defining macros, it is not yet clear as to how the system would be able to use those facilities to learn to do useful editing.

The most serious difference between OOPS and Alpha is in OOPS not having any concept of "optimization". In its present state of development it has nothing corresponding to Phase 2 of Alpha. Phase 2 has many useful features. Perhaps the most important are:

- It is able to use its improved updating scheme to improve its induction, which further improves the updating scheme, etc.
- It is able to use information about *failed* trials, as well as information about success.
- It is able to "invent" new PSTs as well as assign probabilities to them.
- It really tries to find the *optimum* solution to a problem, rather than an improvement over the best previous trial.

7.3 Genetic Programming

In section 1, on Inductive Inference, we mentioned the possible use of Genetic Programming (GP) for updating. It is a particularly good kind of problem for GP (Cra 85, Ban 98), since we are always working on roughly the same update problem. We can use the same population of PSTs but each new problem given to the system changes its "fitness function" to a small extent.

Current GP systems are much improved when the mutation and crossover algorithms are allowed to adapt to the problem being solved and to the nature of the extant population. The techniques of section 1 can be used to implement this adaptation.

In GP, new populations are obtained from old by mutation and/or crossover — which were originally attempts to simulate organic evolution. These operations can be usefully generalized. Mutation can be considered to be "induction from a sample of one", and crossover, "induction from a sample of two". Classical statistics works poorly if at all, with small sample sizes. Bayesian statistics can deal with small samples, but the prediction depends much on the a priori distribution. Appendix A tells us how to obtain a reasonable a priori distribution; sections 1 and 2 tell how to update it.

It is clear that we don't have to use samples of one or two — any sample size up to the size of the current population can be used.

The induction methods of sections 1 and 2 can be regarded as a kind of GA using the entire population of PSTs to create new PSTs.

8 State of Research: What has been done, what needs to be done

This report outlines the workings of the Alpha system and gives some mathematical details of its operation. At present, it appears that the theoretical foundation of Alpha is sound. The main immediate problem is the design of a suitable Training Sequence for it.

OOPS (Sch 02) is a program that has solved a set of problems using Lsearch. Could we not use these problems as the first steps of a Training Sequence?

The main values of OOPS has been in showing how Lsearch could be used to solve difficult problems, and that it could use information from earlier problems to significantly improve solutions of later problems.

It is not clear how the two sets of problems solved by OOPS could be continued to solve problems of increasing difficulty. Writing very short Training Sequences is relatively easy. Writing long Training Sequences that give the system great capabilities in many domains is more difficult.

We expect to start our Training Sequence with very easy problems: the number of trials need not exceed 10^6 — much smaller than the 10^{10} needed for OOPS to solve the Tower of Hanoi.

Not dealt with in OOPS is the problem of "scaling"(section 1.3), in which the time needed to solve a problem is significantly increased for each new problem. We expect that this difficulty can be dealt with by discovery of "contexts" (Section 1.3). Context discovery must be integrated with Training Sequence design.

Appendix A: The AZ System for apriori probability assignment

Though AZ is similar to LISP, modified versions of the system can be made using other languages such as Forth, APL, assembler language, etc.

We will first describe the language AZ, show how it generates functions such as O^j of section 1, and how we assign a probability $P(O^j)$, to its description.

We will show how O_n is used to evaluate $O^j(A_i|Q_i)$ in equation 1 of section 1.

The language AZ begins with a set of primitive unary, binary and possibly tertiary functions. Functions are defined sequentially as compositions of primitives and previously defined functions. An economical way to describe a large complex function is to first describe various simpler functions, that are important components in the description of the final function. In general, it is not *economic* to define a sub–function unless it is used at least twice.

By *economic* we mean *description shortening.* It involves finding regularities in strings and enables us to increase our estimates of the apriori probabilities of those strings.

An apriori probability is assigned to functions such as O^j, in the following manner:

We start with an "argument list" of primitive functions and primitive constants that are usable in the function definition. In the course of our calculations, we will occasionally add to and delete items from this argument list. Some examples of "Primitive functions" could be x, *sum*, *minus*, *mul*, *div*, *sin*, sin^{-1}, ... "Primitive constants" could be $0, 1, \pi, e$, ...

We will use Polish, parenthesis-free notation to define functions, so $3\times(1+7)$ is written: *mul* 3 *sum* 1 7.

To define a simple function, we will write an argument list of the primitive functions and constants it might use, followed by the function name and definition in terms of those primitives.

x *sum mul* 0 1 F_1 *sum mul* x_1 x_1 *mul* x_2 x_2 Δ defines the function

$$F_1(x_1, x_2) = {x_1}^2 + {x_2}^2$$

The "Δ" symbol indicates that the definition sequence is completed.

To define more complex functions compactly, we will define a sequence of functions and constants culminating in the complex function we want to define.

For example, to define $F_2(x_1, x_2, x_3) = x_1 F_1(x_2, x_3)$ we write the definition F_1 followed by the definition of F_2 in terms of F_1:

$$x\ sum\ mul\ 0\ 1\ F_1\ sum\ mul\ x_1\ x_1\ mul\ x_2\ x_2\ F_2\ mul\ x_1\ F_1\ x_2\ x_3\ \Delta \qquad (12)$$

The definition of each function or constant in a complex definition can only refer to symbols that have occurred earlier in the definition sequence.

At each point in a definition string, only certain symbols are legal. Using illegal symbols results in a meaningless (uninterpretable) string.

Except for special cases which we will discuss in the next paragraphs, the probability of some symbol, α, occurring at a particular point in the definition string is n/m. Here n is the number of times that α has occurred previously in the definition string. m is the total number of times that all symbol types that are legal at this point, have occurred previously in the definition string. This probability assignment corresponds to "Laplace's Rule".

There are three kinds of symbols whose probabilities are assigned in different ways: x, Δ, and F_i.

Whenever the symbol x occurs in a function definition, it will be followed by a subscript that is a positive integer. The first time an x occurs this subscript must be 1. Henceforth, the subscript of an x in the same function definition can be any integer from 1 to $n+1$; where n is the largest x subscript that has occurred thus far in that function definition. Each of these integers has probability $(1+n)^{-1}$.

The first time the function name F_1 occurs, it is the only possible symbol at that point. It is given probability 1. If $i > 1$, the first time the function name F_i occurs, its probability is $(i-1)/i$. The only other legal symbol at these points is the stop symbol, Δ, which has probability $1/i$. The stop symbol indicates that the sequence of definitions is completed. It always occurs at the end of the sequence of definitions.

x and the constants are considered to be illegal if they immediately follow the first occurrence of F_i. This is because the function defined would be redundant, e.g. F_i followed by 1 simply defines F_i to be the constant 1.

The symbols in (8), which defines F_2, are assigned probabilities as follows:

F_1 is given probability 1, since it is the only possible symbol at that point.

sum is given probability $1/(1+1)$, since $n_{sum} = 1$ and $n_{mul} = 1$.

mul is given probability $1/(1+1+2+1+1)$ since $n_{mul} = 1$, the legal symbols x, *sum*, 0, 1 have all occurred once before that point, and *mul* has occurred once.

x is given probability $1/(1+2+2+1+1)$

The subscript 1 is mandatory, so it has probability 1

x has probability $1/(2+2+2+1+1)$

The subscript 1 has probability $(2-1)/2$

mul has probability $2/(3+2+2+1+1)$ ...

Δ has probability $1/3$ since F_3 would be also legal at that point.

Table 1 summarizes the probabilities of symbols defining F_2. The product of these probabilities gives a first estimate of the apriori probability of F_2. Because F_2 can be described in many other equivalent ways, this first estimate has to be multiplied by a sizable factor to get closer to apriori probability.

The forgoing rules can only define functions that are compositions of other defined functions. To define recursive functions we must modify the rules that determine which symbols are legal at each point

The factorial function is recursively defined by the expression:

$$F(x) = xF(x-1); F(0) = 1$$

It is a member of a class of functions defined by

$$F(x) = h(x, F(g(x))); F(a) = b$$

A function of this sort can be defined by the list:

$$h(\cdot,\cdot), g(\cdot), a, b$$

Formalisms of this kind can be devised to define more general recursive functions.

TABLE 1

Symbol	Legal possible symbols at that point	Probabilities
x	x	1
sum	sum	1
mul	mul	1
0	0	1
1	1	1
F_1	F_1	1
sum	sum, mul	$1/(1+1)$
mul	$x, sum, mul, 0, 1$	$1/(1+2+1+1+1)$
x	$x, sum, mul, 0, 1$	$1/(1+2+2+1+1)$
$subscript1$	1	1
x	$x, sum, mul, 0, 1$	$2/(2+2+2+1+1)$
$subscript1$	$1, 2$	$1/2$
mul	$x, sum, mul, 0, 1$	$2/(3+2+2+1+1)$
x	$x, sum, mul, 0, 1$	$3/(3+2+3+1+1)$
$subscript2$	$1, 2$	$1/2$
x	$x, sum, mul, 0, 1$	$4/(4+2+3+1+1)$
$subscript2$	$1, 2, 3$	$1/3$
F_2	F_2, Δ	$(2-1)/2$
mul	sum, mul, F_1	$3/(2+3+1)$
x	$x, sum, mul, 0, 1, F_1$	$5/(5+2+4+1+1+1)$
$subscript1$	1	1
F_1	$x, sum, mul, 0, 1, F_1$	$1/(6+2+4+1+1+1)$
x	$x, sum, mul, 0, 1, F_1$	$6/6+2+4+1+1+2)$
$subscript2$	$1, 2$	$1/2$
x	$x, sum, mul, 0, 1, F_1$	$7/(7+2+4+1+1+2)$
$subscript3$	$1, 2, 3$	$1/3$
Δ	F_3, Δ	$1/2$

Appendix B: A Convergence Theorem for Q, A Induction

We will show that for an adequate sequence of (Q_i, A_i) pairs, the predictions obtained by the probability distribution of equation 1 can be expected to be extremely good.

To do this, we hypothesize that the sequence of A_i answers that have been observed, were created by a probabilistic algorithm, $\mu(A_i|Q_i)$ and that μ can be described with k bits, using the reference machine of Appendix A that assigns a priori probabilities to all partial recursive functions.

Any probability distribution that assigns probabilities to every possible A_i, must also assign probabilities to each bit of A_i. Suppose that a_r is a string of the first r bits of A_i. Then the probability given by μ that the $(r+1)^{th}$ bit of A_i is 0 is

$$\sum_j \mu(a_r 0 x^j|Q_i) \Big/ \sum_j \mu(a_r x^j|Q_i)$$

x^j ranges over all finite strings.

Similarly, the algorithm of equation 1, which we will call P, can be used to assign a probability to every bit of every A_i. We will represent the sequence of A_i's by a string, Z, of these A_i's separated by the symbols, s — denoting "space". Z, then, is a sequence of symbols from the ternary alphabet 0, 1, s. Using an argument similar to the foregoing, it is clear that both μ and P are able to assign probabilities to the space symbol, s as well as to 0, and 1.

We have, then, two probability distributions on the ternary strings, Z. The first distribution, μ is the creator of the observed sequence, and the second distribution, P, represents equation1, and tries to predict the symbols of Z.

For two such probability distributions on *binary* strings, the corollary of theorem 3 of [Sol 78] applies: The expected value, with respect to μ (the "generator"), of the sum of the squares of the differences in probabilities assigned by μ and P to the bits of the string are less than $\ln c$, c being the largest positive number such that $P > c\mu$ for all arguments.

Hutter [Hut 01] has generalized this corollary so it applies to systems with any finite alphabet. We are here concerned only with a ternary alphabets, but it implies a corresponding theorem if the symbols of the A_i are from any finite alphabet.

The result is

$$\sum_l \mu(Z_l) \sum_{i=1}^{n} \sum_{j=0}^{h_i^l+1} \sum_{t=0,1,s} (P_{i,j}^l(t) - \mu_{i,j}^l(t))^2 < k \ln 2 \tag{13}$$

l sums over all strings Z_l that consist of n finite binary strings separated by s symbols (spaces).

A_i^l is the i^{th} A of Z_l

$P_{i,j}^l(t)$ is the probability as given by P that the j^{th} symbol of A_i^l will be t, conditional on previous symbols of A_i^l's in the sequence, Z_l and the corresponding Q's.

t takes the values 0,1 and s.

$\mu_{i,j}^l(t)$ is defined similarly to $P_{i,j}^l(t)$, but it is independent of previous A_i^l's in the sequence.

h_i^l is the number of bits in A_i^l. The $(h_i^l+1)^{th}$ symbol of A_i^l is always s.

The total number of symbols in Z_l is $\sum_{r=1}^{n}(h_i^l+1)$.

$\mu(Z_l)$ is the probability that μ assigns to Z_l in view of the sequence of Q's.

k is the length of the shortest description of μ.

This implies that the expected value with respect to μ of the squared "error" between P and μ, summed over the individual symbols of all of the A_i, will be less than $k \ln 2$

Since the total number of symbols in all of the answers can be very large for even a small number of questions, it is clear that the error per symbol decreases rapidly as n, the number of Q, A pairs increases.

Equation (9) gives a very simple measure of the accuracy of equation (1). There are no "order of one" constant factors or additive terms. A necessary uncertainty is in the value of k. We cannot ever be certain that we know its value. If the generator of the data has a long and complex description, we are not surprised that we should need more data to make good predictions — which is just what equation 9 specifies.

The value of the constant, k, depends critically on just what universal reference machine is being used to assign a priori probability to the O_j and to μ. Any a priori information that a researcher may have can be expressed as a modification of the reference machine — by inserting low cost definitions of concepts that are believed to be useful in the needed induction — resulting in a shorter code for μ, (a smaller k), and less error.

We believe that if all of the needed a priori information is put into the reference machine, then equation 1 is likely to be the best probability estimate possible.

At first glance, this result may seem unreasonable: Suppose we ask the system many questions about Algebra, until it's mean errors are quite small — then we suddenly begin asking questions about Linguistics — certainly we would not expect the small errors to continue! However, what happens when we switch domains suddenly, is that k suddenly increases. A μ that can answer questions on both Algebra and Linguistics has a much longer description than one familiar with Algebra only. This sudden increase in k accommodates large expected errors in a new domain in which only a few questions have been asked.

Appendix C: Levin's Universal Search Algorithm (Lsearch)

Levin's original paper (Lev 73) gave some properties of Lsearch, but didn't tell how to do it. The Li, Vitányi book(s) (LiV 93, LiV 97) give more details — including some ways to implement it. We will discuss three methods of implementation: limit doubling, parallel search and random search.

Limit Doubling Lsearch

Section 1 on Inductive Inference described the application of the limit doubling method to Inductive Inference problems.
Section 2 on Inversion problems described its application to Inversion problems.
Section 3 described its application to Time Limited Optimization.

Parallel Search

For Inversion problems: we work on all PST trials in parallel, time sharing our CPU cycles. The fraction of time share spent on the generation of F_i, generation of x_i and testing x_i, is P_i, the a priori probability assigned to F_i.

This method is at least twice as fast as the limit doubling method. However, it requires much more memory and/or a clever disc swapping scheme.

For Time Limited Optimization, as with Inversion problems, we work on all PST trials in parallel, using a time share fraction of P_i, for PST, F_i. We quit when time has run out, and pick the x_i from the PST that had gotten best output at quitting time.

Random Search

Random Lsearch is a variety of parallel search. With probability P_i we randomly choose the PST, F_i, and we work on it for a fixed time τ. We then repeatedly randomly chose a PST and work on it for time τ. If a PST is ever again chosen, we continue where we left off, spending another τ on it. τ should be small, but "large" with respect to time needed to switch from one PST to another.

The properties of random Lsearch are about the same as those of parallel Lsearch. It is particularly useful when the value of P_i is available in Monte Carlo form — e.g. F_i can be generated by a Monte Carlo procedure, by randomly selecting operators or op codes with probabilities that correspond to their a priori probabilities.

Acknowledgement

I would like to thank IDSIA for the very productive month at their facility — particularly Marcus Hutter and Juergen Schmidhuber whose comments and

discussion inspired this report.

References

[1] (Ban 98) Banzhaf, Nordin, Keller, Francone, *Genetic Programming, an Introduction,* Morgan Kaufmann, 1998.

[2] (Cra 85) Cramer, N.L., "A Representation for the Adaptive Generation of Simple Sequential Programs." In *Proceedings of an International Conference on Genetic Algorithms and Their Applications, Carnegie-Mellon University, July 24–26, 1985,* J.J. Grefenstette, ed., Lawrence Erlbaum Associates, Hillsdale, N.J., 1985, pp. 183-187.
http://www.sover.net/˜nichael/nlc-publications/icga85/index.html

[3] (Hut 02) Hutter, M.,"Optimality of Universal Bayesian Sequence Prediction for General Loss and Alphabet,"
http://www.idsia.ch/˜marcus/ai/

[4] (Len 95) Lenat, D., "Cyc: A Large Scale Investment in Knowledge Infrastructure," *Communications of the ACM*, 38, no. 11, 1995.
http://cyc.com/

[5] (Lev 73) Levin, L.A., "Universal Search Problems," *Problemy Peredaci Informacii 9*, pp. 115–116, 1973. Translated in *Problems of Information Transmission 9*, 265–266.

[6] (LiV 93) Li, M. and Vitányi, P. *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, N.Y., 1993, pp. 410–413.

[7] (LiV 97) Li, M. and Vitányi, P. *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, N.Y., 1997, pp. 502–505.

[8] (Pau 94) Paul, W. and Solomonoff, R., "Autonomous Theory Building Systems,"*Annals of Operations Research*, 1994.

[9] (Sch 02) Schmidhuber, J., "Optimal Ordered Problem Solver," TR IDSIA-12-02, 31 July 2002. http://www.idsia.ch/˜juergen/oops.html

[10] (Sol 78) Solomonoff, R.J., "Complexity–Based Induction Systems: Comparisons and Convergence Theorems," *IEEE Trans. on Information Theory*, Vol IT–24, No. 4, pp. 422–432, July 1978.
http://world.std.com/˜rjs/pubs.html

[11] (Sol 86) Solomonoff, R.J. "The Application of Algorithmic Probability to Problems in Artificial Intelligence," in *Uncertainty in Artificial Intelligence*, Kanal, L.N. and Lemmer, J.F. (Eds), Elsevier Science Publishers B.V., 1986, pp. 473–491. http://world.std.com/˜rjs/pubs.html

[12] (Sol 89) Solomonoff, R.J. “A System for Incremental Learning Based on Algorithmic Probability,” Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, Dec. 1989, pp. 515–527. http://world.std.com/~rjs/pubs.html