OPTIMUM SEQUENTIAL SEARCH

R.J. Solomonoff
Oxbridge Research, Box 559
Cambridge, Mass. 02238

REV., 1985

There are two theorems in Levin's "Universal Sequential Search Problems" (1973). The first states the now well-known principle of NP completeness and is followed by an outline of a proof.

The second gives a solution to a very broad class of mathematical problems, but, partly because no proof was suggested in the paper, its great importance is not widely appreciated. It is our purpose to give an outline of Levin's proof and a simple extension of the theorem to another broad class of problems.

This will be followed by a discussion of the significance of these problem solving methods and a technique by which they may be used to obtain practical solutions to problems.

The second theorem gives a method for inverting functions defined by computing machines or any other well defined algorithm. Suppose we are given M, a machine that maps finite strings into finite strings. Given the finite string x, how can we find in minimal time, a string p such that $M(p) = x$?

Finding solutions of equations, symbolic integration, and proving mathematical theorems are but three types of problems in this broad class.

Appendix I proves this second theorem, i.e.: Suppose that A is an algorithm that can examine M and x and within time T produce a program, p, such that M(p) = x , then, not knowing A, we have an algorithm that can do the same thing that A does, but in time $\leq T \cdot 2^{l(\beta_M)} \cdot C_A$.

Here, $l(\beta_M)$ is the length of β_M , the description of algorithm A, using a suitable reference machine. C_A is a measure of how much slower our reference machine is than the machine that implements A directly. Both β_M and C_A are independent of M and x.

Another very important kind of problem that can be solved with Levin's search method is a common type of optimization problem. Here we are given a machine M that operates on finite strings, p and yields a number, G, for its output. The problem is to find within a limited search time, τ, an input that yields the highest possible value of G.

Making predictions from numerical and/or non-numerical data is a problem of this type, as is the problem of finding a good theory to fit empirical data. Many problems in engineering are also of this type.

Again, as in the previous proof, suppose there exists an algorithm A that is able to look at machine M and the time limit, τ, and produce the input $p = A(M, \tau)$ in time τ, yielding G value, M(p).

The theorem states that there exists a universal search procedure that will eventually find the same p and G as the algorithm A , and it will take less than $2^{l(\beta_M)} \cdot C_A$ times as long.

Appendix II gives the search procedure and outlines the proof, which is very similar to that of Appendix I.

The significance of these two search methods:

A very large fraction of the problems in mathematics and in engineering can be expressed as machine inversion problems, or optimization problems of the kinds treated here - or they can be adequately approximated by these formalisms.

If these methods can be practically implemented, they would be able to work an extremly large class of very difficult problems normally solvable by only highly intelligent humans.

The practicality of these searches hinges on the sizes of two constants: C_A, which measures the inefficiency of the KUSP machine in simulating A : and $2^{\ell(\beta_M)}$, which measures the unlikeliness of the machine A, and hence the difficulty of our finding it. Large $\ell(\beta_M)$ means that A has a long (and unlikely) description via the KUSP machine being used.

These two factors can be reduced to computationally acceptable levels through the use of training sequences of problems of progressively increasing difficulty.

One way to do this is to introduce an additional argument "D" (data) into the KUSP machine, $M^K(\alpha, D, M, x)$. D consists of the previous experience of the machine. Among the things included are all of the problems and their solutions that it has found thus far, along with the methods that it has used to find these solutions. For a particular problem M_1, x_1 having solution p_1, that was found using $A(M_1, x_1) = M^K(\alpha, D_1, M_1, x_1) = p_1$, D_2 (the new value of D) would contain at least M_1 , x_1 , p_1 , D_1 and α_1 or information from which they can be easily obtained.

Using the argument D, it is possible for the $\propto$ of a new problem to refer to solutions or parts of solutions to previously solved problems. This materially reduces the amount of information in $\propto$ and can often reduce $2^{\ell(\propto)}$ to a manageable magnitude. It corresponds to the human problem solving technique of making solution trials for new problems that are combinations of concepts successfully used for the solutions of previous problems.

Another trick can be borrowed from human problem solving: After a problem has been solved, the data, D, can be rewritten in more compact form, so that concepts that have often been used in successful problem solving are given short codes. The problem of making D "more compact" by finding regularities in it is a standard form of optimization problem and is directly solvable by Levin's search algorithm.

D can also be modified so that the hardware inefficiency factor, C_A , is minimized.

Using these and other devices to trim up the system, we obtain a problem solver that operates much as humans seem to.

We can build up a problem solving system of this sort by starting with no D at all and having the system work some problems with solutions that are simple combinations of the primitive instructions of the reference machine, M^K . The solutions to these problems become the initial D. The form of D is then improved by noting regularities in the problem solutions and expressing D more compactly in terms of these regularities.[+]

This compact coding of D uses short sequences of bits to define various useful concepts. The entire code for D consists of these definitions followed by the description of D in terms of the

definitions.

When D has been compressed in this way, the system is able to work more difficult problems with solutions that are simple combinations of the concepts used to solve the earlier problems.

Using a training sequence of problems of increasing difficuly, the system discovers more and more concepts of greater complexity that are needed to solve these problems.

+One of the fundamental ideas of algorithmic information theory is that any regularity in a body of data can be used to compress that data.

References

1. Levin, L.A. "Universal search problems." Problemy Peredaci Informacii 9 (1973), 115-116. Translated in Problems of Information Transmission 9, 265-266.

2. Kolmogorov, A.N. and Uspenski, V.A. "On the definition of an algorithm." 'Uspehi Mat. Nauk. 13 (1958), 3-28; AMS Transl. 2nd ser. 29 (1963), 217-245.

APPENDIX I

SECTION A: Introduction.

The demonstration that follows is based on discussions with Levin. We will describe his search algorithm and show that it satisfies the theorem.

Before giving the proof, we need some essential background. First, the concept of a Kolmogorov-Uspenski machine (1958).

An ordinary universal digital machine is one that can simulate any other digital machine if it is given a description of it. Suppose x is a finite binary string and M is some machine. Then M(x) will be the output of M for input x. M(x) may not always exist.

M^U is called a "universal machine" if when M^U is given D_i, a finite binary string that describes M_i, M^U can imitate the behavior of M_i. For all possible x, $M(D_i x) = M_i(x)$. $D_i x$ is a string formed by string D_i followed by string x.

Ordinarily no relationship is stipulated between the time needed to compute M (x) and that needed to compute $M^U(D_i x)$. In an ordinary universal machine, we are concerned mainly with the fact that M^U's program(s) to compute a certain string are at most only a constant number of bits longer (i.e. the length of D_i longer) than the length of M_i's program(s) to compute that string.

A universal Kolmogorov-Uspenski machine (KUSP machine), M^K is a kind of universal machine with an additional property: we can always find a description D_i of M_i such that for all x, the time needed to compute $M^U(D_i x)$ is not more than a constant factor, C_i larger than the time to compute $M_i(x)$. C_i is a

function of M_i and M^U , but is independent of x. We also have the simulation condition $M^U (D_i x) = M_i(x)$.

A universal Turing machine can perform this time simulation only of Turing machines having no more tapes than itself. However, a universal KUSP machine can do it for any Turing machine having a finite number of tapes.

Next we need what is called "Kraft's inequality". First we define a "prefix set" to be a set of finite binary strings with the property that no string of the set can be obtained by adjoining one or more bits onto the end of another string of the set.

Thus if 0110 is a member of a prefix set, then 011011 and 01100 cannot also be members of that set - nor can 011 or 01 or 0. 010 might possibly be a member. If $[x_i]$ are the members of a prefix set, then Kraft's inequality says

$$\sum_i 2^{-\ell(x_i)} \leq 1$$

Here $\ell(x_i)$ is the length of the sequence x_i - the number of bits in it. The set of programs, $[p_i]$ that causes a machine with sequential input to output a finite string, then stop, always constitutes a prefix set.

Now let's return to the problem of machine inversion. We have a machine M and a finite binary string x. We want to find p such that $M(p) = x$ and we want to find it as rapidly as possible.

Suppose there exists an algorithm, A, that can operate on x and the description of M and produce p from them . So $A(M,x) = p$ and $M(p) = x$.

We will use a simple search procedure to find A. First choose a small time limit, T. Then make trial values for A by simulation, using M^K . $M^K(\alpha_i, M, x)$ will be a trial to find $A(M,x)$. Using suitable values for α_i , generate the set of all solution trials, $p_i = M^K(\alpha_i, M, x)$ such that the generation and testing of p_i (via $M(p_i)$) takes time $\leq T2^{-\ell(\alpha_i)}$.

The total testing time for this set is

$$\sum_i T2^{-\ell(\alpha_i)} = T\sum_i 2^{-\ell(\alpha_i)}$$

We will show that the set, $[\alpha_i]$, generated in this way is a prefix set, so $\sum_i 2^{-\ell(\alpha_i)} \leq 1$ and thus the total testing time for this set of trials must be $\leq T$.

If a solution is found in this set, end the procedure. If not, then double T ($T \leftarrow 2T$) and repeat the procedure. This doubling and redoubling continues until a solution is found. If $T = T_o$ when we finally find a solution then the total search time will be $T_o + \frac{1}{2}T_o + \frac{1}{4}T_o \ldots < 2T_o$.

This is the essence of the search procedure.

Section B gives a more detailed program for the search, showing how the sets of trial strings $[\alpha_i]$ are generated.

Section C shows that each set, $[\alpha_i]$ generated in section B is a prefix set and includes all α_i that result in p_i's that can be generated and tested in time $\leq T$.

Section D compares the time needed to find the solution to $M(p) = x$ using the search procedure of Section B, with the time required by the algorithm A. It shows that the search of section B is slower than algorithm A by less than the constant factor $2^{\ell(\beta_M)+1} \cdot C_A$, which completes the proof.

SECTION B. A detailed search procedure for solving $M(p) = x$.

1. $T \leftarrow 1$

2. $T \leftarrow 2T$: Reset clock to zero: also reset test string, α, to 0 ($\alpha \leftarrow 0$) .

3. Start to compute $p = M^K(\alpha, M, x)$

If, before clock reads $T2^{-\ell(\alpha)}$, M has read all of α and requests another input bit, then keep feeding it 0's (i.e. $\alpha \leftarrow \alpha 0$) whenever it asks for more input until it either stops or until the clock reads $T2^{-\ell(\alpha)}$ - whichever is sooner. If the latter is true, go to 6.

4. Begin testing p. If we are able to verify that $M(p) = x$ before clock reads $T2^{-\ell(\alpha)}$, exit with solution to search problem!

5. If α is all 1's, we've exhausted all α's for this T value. We need a larger T. Go to 2.

6. Reset clock to zero. Generate a new α by changing the leftmost 0 in α to 1 and discarding all bits (if any) to the right of that 1. Go to 3.

7. End of program.

A possible sequence of α values that could be generated by this program for a particular value of T:

α_0 : 0000
α_1 : 00010
α_2 : 00011
α_3 : 001000
α_4 : 001001
α_5 : 00101
α_6 : 00110
α_7 : 00111
α_8 : 01
α_9 : 10
α_{10} : 110
α_{11} : 111

SECTION C. We will first show that for any particular T value, the set $[\alpha_i]$ is a prefix set.

From the procedure of Section B, it is clear that the strings, α_i are of only two kinds: those for which M^k reads all of α_i and stops in time less than $T2^{-\ell(\alpha_i)}$ and those for which it does not stop, but reads as many bits as possible up to that time. In either case, if we concatenate one or more bits onto α_i, M^k would not be able to read the extra bits of this string in time $T2^{-\ell(\alpha_i)}$ so those augmented strings cannot be members of the set. Since no member of the set can be an extension of any other member, it must be a prefix set.

Next we will show that the set $[\alpha_i]$ is a "complete" prefix set in the sense that $\sum_i 2^{-\ell(\alpha_i)} = 1$. This implies that there is no new string that could be added to the set and have it remain a prefix set, since the $\sum_i 2^{-\ell(\alpha_i)}$ would be greater than 1 - violating Kraft's inequality. Given a complete prefix set, every possible string must be either a prefix or continuation of a member of that set. The set 00,01,1 is a complete prefix set: we note $2^{-2} + 2^{-2} + 2^{-1} = 1$. The set of α_i's listed at the end of Section B is also a complete prefix set.

Lemma 1: For the set $[\alpha_i]$ generated in Section B for some value of T:

$$\sum_{j=0}^{i-1} 2^{-\ell(\alpha_j)} = D(\alpha_i)$$ Here $D(\alpha_i)$ is the value of α_i taken as a binary fraction. For example, $D(10110) = 2^{-1} + 2^{-3} + 2^{-4} = .10110$ in binary notation. Lemma 1 is readily proved by mathematical induction, starting with α_0, which is always a finite sequence of zeros.

If α_n is the last member of $[\alpha_i]$ we note that α_n must

consist of a finite number of 1's - say just k of them.

From Lemma 1,

$$\sum_{j=0}^{n-1} 2^{-\ell(\alpha_j)} = .111\ldots = 2^{-1} + 2^{-2} \ldots 2^{-k} = 1-2^{-k}$$

since $\ell(\alpha_n) = k$ we have

$$\sum_{j=0}^{n} 2^{-\ell(\alpha_j)} = 1-2^{-k} + 2^{-k} = 1$$ and so $[\alpha_i]$ must be a complete prefix set.

Next, we will show that if $M^k(\alpha, M, x) = p$ and $M(p) = x$ can be both together computed in time $\leq T2^{-\ell(\alpha)}$, then α must be in the complete prefix set $[\alpha_i]$ generated in Section B for time limit, T.

Since the prefix set $[\alpha_i]$ is complete, α must be either an extension or a prefix of some member of $[\alpha_i]$. Members of $[\alpha_i]$ are of two kinds: "stopping" members, α_i, are those for which $M^k(\alpha_i, M, x)$ stops within the time limit. Otherwise, α_i is a "nonstopping" member. If α is an extension of a stopping member of $[\alpha_i]$, it must be identical to that member. It can't be an extension by 1 or more bits of a non-stopping member because this would exceed the time limit.

Similarly, we can show that no member of $[\alpha_i]$ can be an extension of α by one or more bits. The only possible conclusion is that α must be a member of $[\alpha_i]$.

SECTION D: In Section A we showed that if $T = T_o$ when we found a solution using the program of Section B, then $2T_o$ would be an upper bound for the solution time of the entire program.

Suppose β is a code of algorithm A for KUSP machine M^K. i.e. for all M and x, $A(M,x) = M^K(\beta, M, x)$ and

(Time for $M^K(\beta, M, x)) < C_A \cdot$ (Time for $A(M,x)$).

Generally there will be many codes, β, having these properties.

Let T_β = Time to compute $M^K(\beta, M, x)$ + Time to test $M(p) = x$.

Let β_M be the code β for which $T_\beta \cdot 2^{\ell(\beta)}$ is minimum.

The search algrithm described will be successful not later than the time at which $T_o = T_{\beta_M} 2^{\ell(\beta_M)}$ and $\alpha_i = \beta_M$, since these conditions give $M(p_i) = x$ and therefor an exit from step 2.

The total search time is $< 2T_o$ which is $< 2T_{\beta_M} 2^{\ell(\beta_M)}$.

From the definition of T_{β_M}, this is

$< 2^{\ell(\beta_M)+1} \cdot$ (Time for $M^K(\beta_M, M, x)$ to compute and test p)

which is

$< 2^{\ell(\beta_M)+1} \cdot (C_A \cdot$ Time for $A(M,x)$ to compute and test p)

That the search algorithm described takes time that is only a constant factor longer than any other algorithm, A, is Levin's second theorem.

The factors $2^{\ell(\beta_M)+1}$ and C_A tell how much slower this search is than algorithm, A. These factors are both independent of M and of x.

APPENDIX II

A search procedure for finding p such that M(p) = maximum, within time limit, t .

1. $G_{Max} \leftarrow -\infty$: set clock to zero: also set test input string α to zero ($\alpha \leftarrow 0$).

2. Start to compute $p = M^x(\alpha, M, t)$.

If, before clock reads $t \cdot 2^{-l(\alpha)}$, M^x has read all of α and requests another input bit, then keep feeding it 0's ($\alpha \leftarrow \alpha 0$) whenever it asks for more input, until it either stops or until the clock reads $t \cdot 2^{-l(\alpha)}$ - whichever is sooner. If the latter occurs, go to 5.

3. Begin evaluating p by computing G= M(p) . If this is computed before clock reads $t \cdot 2^{-l(\alpha)}$ and $G > G_{Max}$, then update G_{Max} by $G_{Max} \leftarrow G$.

4. If α is all 1's,we've exhausted all α's for this t value. Exit this program. Otherwise, go to 5.

5. Reset clock to zero. Generate a new α by changing the rightmost 0 in α to 1 and discarding all bits (if any) to the right of that 1. Go to 2.

6. End of program.

We will now calculate the time needed to find p and G this way and compare it with the time needed by the algorithm A to find the same p and G .

Let β_M be the shortest string such that

$M^x(\beta_M, M, t) = A(M, t \cdot 2^{-l(\beta_M)}/C_A)$ and

Time for $M^x(\beta_M, M, t)$ < C_A · Time for $A(M, t \cdot 2^{-l(\beta_M)}/C_A)$.