STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-159

STAN-CS-253-72

TOTAL COMPLEXITY AND THE INFERENCE OF BEST PROGRAMS

BY

J. A. FELDMAN
P. C. SHIELDS

SUPPORTED BY
NATIONAL SCIENCE FOUNDATION
AND
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 459

APRIL 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-159

APRIL 1972

'COMPUTER SCIENCE DEPARTMENT
REPORT CS-253

TOTAL COMPLEXITY AND THE INFERENCE OF BEST PROGRAMS

by

J. A. Feldman and P. C. Shields

ABSTRACT: Axioms for a total complexity measure for abstract programs are presented. Essentially, they require that total complexity be an unbounded increasing function of the Blum time and size measures. Algorithms for finding the best program on a finite domain are presented, and their limiting behaviour for infinite domains described. For total complexity, there are important senses in which a machine can find the best program for a large class of functions.

This research was supported in part by the National Science Foundation and the Advanced Research Projects Agency.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the National Science Foundation.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151. Price: Full size copy $3.00; microfiche copy $0.95.

We are primarily concerned, in this paper, with the question of when a machine can learn a program from samples of its input-output pairs. This problem of program inference is closely related to the problem of grammatical inference, which has received a fair amount of consideration [2]. There are, in the grammatical inference literature, many results and discussions which can be carried over to program inference. This paper arose out of an attempt to carry out what we believed to be a trivial reworking of some of the results of [7] for programs. In fact, the results on programs turn out to be significantly different; we will discuss this issue further below.

We are interested in modelling the following situation. A machine M receives at each time t , an input-output pair (x,y) from an unknown program P in a known class $\mathcal{C}$ of programs. At each time, the machine is to guess some $P_j \in \mathcal{C}$ as the best program for the finite number of input-output pairs seen so far. We show that there are reasonable conditions under which M can guess the best program at each finite time and also have good behaviour in the limit. To do this, we need a formal notion of "best" program.

The key to our development is the combined complexity measure including both program size and running time. Many of the difficulties arising in other axiomatic treatments of complexity are elided in the combined complexity approach.

More formally, our results will be formulated for programs. A program can be taken to be any formal computational scheme for evaluating a recursive function , such as a Turing machine description. To simplify the discussion it is assumed that the input and output of a program are both positive integers. The graph $\mathcal{G}(P)$

of a program P is the set of all pairs (x,y) such that P is defined for x and the output of P given the input x is y . A sample S of a program P is a finite nonempty subset of $\mathcal{L}(P)$.

The class $\mathcal{C}$ denotes a class of programs which can be effectively enumerated by an admissible [17] enumeration, such as the class of all Turing machines, the class of FORTRAN programs, or the class of loop programs [19]. An inference machine $M = M_{\mathcal{C}}$ is any formal effective procedure for inferring programs from finite samples, that is, M is defined on the set of samples {S} of programs in $\mathcal{C}$ and M(S) is a program in $\mathcal{C}$. We will always require that S is a sample of M(S) , that is

(1) $\qquad \mathcal{L}(M(S)) \supseteq S$

Various complexity measures have been discussed, in particular program running time and program size (see [12] for a discussion of recent results). We wish to discuss measures of program complexity which take into account both the size and running time of programs. The simplest such measure is the product of size and running time. Other measures are also useful. In order to obtain general results we shall describe a complexity measure as any function satisfying a simple set of axioms. The axioms for size and running time'are the same as those discussed in [12], while the axioms for a combined complexity measure are equivalent to those in [7].

First we assume that the program size or length $L = L_{\mathcal{C}}$ satisfies the conditions

(2) There is an effective admissible enumeration $\{P_n\}$ such such that

(a) $r(n) = L(P_n)$ is a recursive positive integer valued total function

(b) For each n , the set $K_n = \{m \mid r(m) = n\}$ is finite

(c) The function $\bar{r}(n)$ = cardinality of K_n is a recursive function.

The running time $T(x,y,P)$ is a positive effectively computable rational function and is defined if and only if (x,y) is in the graph of P . There is a related recursive function

$$d(x,y,P,m) = \begin{cases} 0 \text{ if } T(x,y,P) \leq m \\ 1 \text{ otherwise} \end{cases}$$

We also assume that the combined running time $T(S,P)$ is of the form

$$T(S,P) = \varphi\left(\bigcup_{(x,y)\in S} \{T(x,y,P)\}\right) \tag{3}$$

where φ is a recursive function. The related function

$$D(S,P,m) = \begin{cases} 0 \text{ if } T(S,P) \leq m \\ 1 \text{ otherwise} \end{cases}$$

is then recursive.

Let c be a positive recursive rational valued function of two non-negative rationalvariables which is increasing and unbounded in each variable. The complexity measure $C = C_c$ is then given by

$$C(S,P) = c(L(P),T(S,P)) , S \subseteq \mathcal{S}(P)$$

Examples The size L(P) might be the number of symbols used to write the program in some alphabet or the number of symbols on the tape of a universal Turing machine needed to describe a simulation of the

program. Some plausible L(P) are excluded because of the requirement that there be only a finite number of programs of each size. For example, the number of statements in a FORTRAN program or the nesting depth of loop programs would not, as normally defined, satisfy (2b). Size measures which take structure into account are discussed in [2, 6] for grammars.

For a given pair (x,y) the running time T(x,y,P) could be the time the program P uses to derive output y from input x (possibly also including the time for reading x and printing y). Other possibilities are the number of moves or number of tape cells scanned by a Turing machine, the number of instructions executed by the program. One can also normalize by some function of x and y, for example, T(x,y,P) could be actual running time divided by xy.

The general function T(S,P) can be obtained from T(x,y,P) in many ways, for example we could take T(S,P) as

$$\max_{(x,y)\in S} T(x,y,P) \quad , \text{ or } \quad \sum_{(x,y)\in S} T(x,y,P)$$

or as an average of $T(x,y,P)$, $(x,y)\in S$.

The possibilities for the function c(L,T) are very large, for example each of the following satisfy the hypotheses for c :

$$(L+1)(T+1) \ , \ L+T \ , \ (L+1)^{(T+1)}$$

Notice that the simple product LT doesn't satisfy the hypotheses for it is not unbounded in L when T=0. We impose this requirement so as to simplify some later arguments. The very general nature of the function c precludes the possibility that all complexity measures are

recursively related, a result which is true both for the length $L(P)$ and time $T(x,y,P)$. (See [12])

Remark 1

Although the results below are quite general, some care must be used in applying them to actual inference situations. A major consideration is to choose measures which do not degenerate into strictly time or strictly size in the limit. For example, $\sum_{(x,y)\in S} T(x,y,P)$ may be unbounded is S gets large or the average of (time/length) may go to zero with large S . Depending on the choice of $c(L(P), T(S,P))$ either situation could lead to degeneracy. One must also choose complexity functions which reflect the intuitive meaning of the problem.

Our later proofs make use of the fact that the programs can be ordered in terms of increasing size. An Occam's enumeration of $\mathcal{C}$ relative to $L_{\mathcal{C}}$ is an admissible enumeration $\{P_i\}$ satisfying

(4) $\quad L(P_i) \leq L(P_j)$ if $i \leq j$.

It is obvious from (2)(b), (c) that a machine can find an Occam's enumeration relative to L . One consequence of this is the following simple result:

Lemma 1 Given a complexity measure $C = c(L,T)$ on the infinite class $\mathcal{C}$ and an Occam's enumeration of $\mathcal{C}$ relative to L then for any sample S of some $P \in \mathcal{C}$, there is an index k such that if $j > k$ then either

(5)(a) $C(S,P_j) > C(S,P)$

or

(b) S isnotasampleof P_j..

Proof This is a consequence of the assumption that c is increasing and unbounded in each variable. We merely choose k as the first index for which

$$c(L(P_k), 0) > C(S,P)$$

If $j \geq k$ and S is a sample of P_j then (4) guarantees that

$$L(P_j) \geq L(P_k)$$

and hence

$$C(S,P_j) = c(L(P_j), T(S,P_j)$$
$$\geq c(L(P_j), 0) > C(S,P)$$

This proves the lemma.

Now we prove the following general theorem.

Theorem 1 Given a complexity measure C(S,P) on a class $\mathcal{C}$ there is an inference machine $M = M_{\mathcal{C}}$ which infers programs of minimum complexity, that is, if S is a sample of some program in $\mathcal{C}$, then S is a sample of M(S) and for all $P \in \mathcal{C}$ for which S is a sample of (P)

(6) $C(S,M(S)) \leq C(S,P)$

Proof The intuitive idea for the proof is as follows: Run $P_1, P_2 \ldots P_t$ on S for time t , successively incrementing t until some P_i , $i \leq t$ runs successfully in time t . Then one need look at no programs whose total complexity exceeds $C(P_i,S)$, hence one need examine only a finite set of programs (cf. Lemma 1) and pick the best one.

To formally construct M we first assume an Occam's enumeration for $\mathcal{C}$ relative to length L . Then

Step 1 Calculate $D(S,P_i,t)$, $1 \leq i \leq t$. If $D(S,P_i,t)=1$ for $1 \leq i \leq t$, increment t by 1 and repeat Step 1. Otherwise let t_0 be the first t for which $D(S,P_i,t)=0$ for some $1 \leq i \leq t$ and let i_0 be the first i , $1 \leq i \leq t_0$ for which $D(S,P_i,t_0)=0$ and proceed to Step 2.

Step 2 Use Lemma 1 to calculate k so that if $j > k$ and S is a sample of P_j then

$$C(S,P_j) > C(S,P_{i_0})$$

Step 3 Compute the first integer $m \geq t_0$ such that

$$C(S,P_{i_0}) < c(0,m)$$

Step 4 Let G(S) denote the set of those j, $1 \leq j \leq k$ for which $D(S,P_j,m)=0$

Step 5 Compute $C(S,P_j)$, $j \in G(S)$

Step 6 Let i_1 be the first $i \in G(S)$ such that

$$C(S,P_i) = \min \{C(S,P_j) \mid j \in G(S)\}$$

and put $M(S) = P_{i_1}$.

Let us show that M(S) has the desired properties. M need choose no program with complexity greater than $C(S,P_{i_0})$. Step 2 rules out programs which are too long while Step 3 rules out programs which take too long to run on S, hence if $j \notin G(S)$ then either S is not a sample of P_j or $C(S,P_j) > C(S,P_{i_0})$ so (6) holds for M(S). This proves the Theorem.

The machine M constructed in the proof of Theorem 1 will in certain cases have reasonable convergence properties as the sample size increases. An information sequence $\mathcal{I}(P)$ is a sequence whose range is $\mathcal{L}(P)$. An initial segment S_n is the sample

$$S_n = \{\mathcal{I}(P)_i \mid 1 \leq i < n\}$$

Given an information sequence $\mathcal{I}(P)$, $P \in \mathcal{C}$, the machine M will eventually be correct on any input for which P is defined, that is

(7) If $(x,y) \in \mathcal{L}(P)$, then there is an N such that $(x,y) \in \mathcal{L}(M(S_n))$ for $n \geq N$.

This follows easily from the fact that $S \subset \mathcal{L}(M(S))$ and that $(x,y) \in S_n$ for large enough n.

It may not be possible to obtain $\mathcal{L}(P) \subseteq \mathcal{L}(M(S_n))$ for n large. If f is a recursive total function then it may happen that any program for f has such rapidly growing running time that $M(S_n)$ will be merely a table for S_n. In other words, if the running times

for programs for f are all unbounded then size becomes irrelevant in the complexity measure. If the running time is bounded then the machine of Theorem 1 will eventually pick only programs which agree with f wherever f is defined.

Theorem 2 Suppose g(P) is an information sequence for some program $P \in \mathcal{C}$ and that $C(S_n,P)$ is bounded as $n \to \infty$. Then for the machine M of Theorem 1, we will have

$$\mathcal{L}(M(S_n)) \supseteq \mathcal{L}(P) \text{ for } n \text{ large enough.}$$

Proof Let i_0 denote the first index i for which $\mathcal{L}(P_i) \supseteq \mathcal{L}(P)$ and $C(S_n, P_i)$ is bounded as $n \to \infty$. Put $b = \operatorname{lub}_n C(S_n, P_{i_0})$ and choose K so that

$$C(L(P_K),0) > b$$

The programs P_k for $k > K$ will never be $M(S_n)$ for their complexity must be larger than that of P_{i_0} on S_n . Furthermore if $k < K$ and $\mathcal{L}(P_k) \not\supseteq \mathcal{L}(P)$, we can choose n_k so that S_{n_k} will not be a sample of $\mathcal{L}(P_k)$. Thus if n is large enough, $M(S_n)$ must be one of the programs P_i for which $i < K$ and $\mathcal{L}(P_i) \supseteq \mathcal{L}(P)$. This proves the theorem.

Notice that if P is total, M(S) will eventually be only P_j such that $\mathcal{L}(P_j) = \mathcal{L}(P)$. This behaviour is called matching in the literature on grammatical inference [7].

Corollary Suppose that for all information sequences $\mathcal{I}(P)$ 'of a given $P \in C$, the limit

$$\gamma(\bar{P}) = \lim_n C(S_n,\bar{P})$$

exists for all $\bar{P}$ such that $\mathcal{L}(\bar{P}) \supseteq \mathcal{L}(P)$. Let γ be the minimum of these $\gamma(\bar{P})$. Then for n sufficiently large, $\gamma(M(S_n) = \gamma$.

Proof As the proof of Theorem 2 indicates, there is a K such that for all n , $M(S_n)$ is one of the programs P_i , $1 \leq i \leq k$, and for n large enough

$$\mathcal{L}(M(S_n)) \supseteq \mathcal{L}(P)$$

Suppose $i \leq K$ and $\mathcal{L}(P_i) \supseteq \mathcal{L}(P)$, $\gamma(P_i) = \gamma$. If $j \leq K$, $\mathcal{L}(P_j) \supseteq \mathcal{L}(P)$ and $\gamma(P_j) > \gamma$ then for n large enough

$$C(S_n,P_j) \geq (\gamma(P_j) + \gamma)/2$$

so $M(S_n)$ will not be P_j . This proves the Corollary.

Theorem 2 can be applied in any case where the program has bounded running times. A slight modification enables one to use this result in the case when a bounding function for the running time is known. To simplify our discussion we shall assume that

$$T(S,P) = \max_{(x,y)\in S} T(x,y,P) . \tag{8}$$

We could extend our results (Theorem 3) to more general φ , but the extensions do not seem to warrant the additional complexity of proof. We continue to search the right generalization of Theorem 3. A recursive total function b of two variables, which is increasing in both is called a bounding function. The running time of $P \in C$

is bounded by b if there is an $\alpha > 0$ such that

$$T(x,y,P) \leq \alpha \cdot b(x,y) \text{ , } (x,y) \in \mathcal{D}(P)$$

We will call the least such α the bounding constant of P . A bounding function gives rise to a new running time T_b defined for $(x,y) \in \mathcal{D}(P)$ by

$$T_b(x,y,P) = T(x,y,P)/b(x,y)$$

and for $S \subseteq \mathcal{D}(P)$ by

$$T_b(S,P) = \max_{(x,y)\in S} T_b(x,y,P) .$$

This in turn gives rise to the complexity measure C_b defined by

$$C_b(S,P) = c(L(P), T_b(S,P))$$

Thus if samples S are drawn from a program P which is known to have its times bounded by a bounding function b , then one can choose programs $M_b(S)$ ofminimal C_b complexity and know that

$$\mathcal{D}(M_b(S)) \supseteq \mathcal{D}(P)$$

if the sample S is large enough.

Remark 2 There are a number of classes of computations with known bounding functions in terms of various types of programs or machines [13]. The complexity measure C_b will be more sensitive if the bounding function chosen is a tight one. Thus, if we know a computation has polynomial bounds we should try to find the particular polynomial rather than just choose some b that grows faster than any polynomial. A bounding function that is too large may give rise to degenerate measures, of Remark 1.

It may even be possible to infer a good bounding function as part of the general procedure for program inference. Here we describe one method for doing this. Suppose $\{b_k\}$ is a sequence of bounding functions satisfying $b_1 \equiv 1$, $b_k(x,y) \leq b_{k+1}(x,y)$ and

$$(9) \qquad \lim_{x\to\infty} \frac{b_k(x,y)}{b_{k+1}(x,y)} = \lim_{y\to\infty} \frac{b_k(x,y)}{b_{k+1}(x,y)} = 0$$

We will now show how to infer both a bounding function and then good programs which run on the sample in that bound.

Theorem 3 Let $\mathcal{C}$ be a class with complexity measure C (where T(S,P) is given by (8)) and let $\{b_k\}$ be a sequence of bounding functions (satisfying (9)). There is an inference machine M which will, for any information sequence J(P), $P \in \mathcal{C}$, infer both a sequence of positive integers $\{\bar{k}_n\}$ and programs $M(S_n)$ such that

(a) $M(S_n)$ is a program in $\mathcal{C}$ of least $C_{b_{\bar{k}_n}}$ complexity whose graph contains S_n.

If, furthermore, there is some program $\bar{P}$ such that $\mathcal{L}(P) \subseteq \mathcal{L}(\bar{P})$ and $\bar{P}$ has its running times bounded by some b_k, then for n large enough

(b) $\bar{k}_n = \bar{k}$, a constant

(c) $\mathcal{L}(M(S_n)) \supseteq \mathcal{L}(P)$

Proof Let $\{P_i\}$ be an Occam's enumeration for $\mathcal{C}$ relative to L. M will use a sequence $\{\alpha_n\}$, α_b being the current guess as to a bounding constant. Initially $\alpha_1 = \uparrow 1$. The machine proceeds as follows to obtain $\bar{k}_n$ and $M(S_n)$.

Step 1 For each i , $1 \leq i \leq n$ and k , $1 \leq k \leq n$, M computes

(10) $$\delta_n(i,k) = \max_{(x,y)\in S_n} d(x,y,P_i,\alpha_n b_k(x,y))$$

If $\delta_n(i,k) = 1$ for $1 \leq i$, $k \leq n$ then M sets $k_n = 1$ and goes to Step 3. Otherwise M goes to Step 2 .

Step 2 M selects k_n as the first index k such that $\delta_n(i,k) = 0$ for some $i < k$. M then selects i_n as the first index such that $\delta_n(i,k_n) = 0$. M then selects $\bar{k}_n$ as the least integer such that $\delta_n(i_n,k_n) = 0$ and goes to Step 3 .

Step 3 If $\delta_n(i,k) = 0$ for some i and k $1 \leq i \leq n$, $1 < k < n$ and if $i_n = i_m$ for some $m < n$ then M sets $\alpha_{n+1} = \alpha_n$. Otherwise M sets $\alpha_{n+1} = 1 + \alpha_n$.

Step 4 M selects $M(S_n)$ as the best program using the algorithm of Theorem 1 with the measure $C_{b_{k_n}}$.

Let us now show that (a),(b),(c) hold. Condition (a) follows directly from that fact that Step 4 uses the algorithm of Theorem 1 .

Consider the set $\mathcal{R}$ of pairs (i,k) such that b_k bounds the running time of P_i and $\mathcal{L}(P_i) \supseteq \mathcal{L}(P)$. We will show that if $\mathcal{R}$ is not empty then (b), (c) hold. Towards this end let $(\hat{i},\hat{k})$ be the pair in $\mathcal{R}$ which minimize the maximum of i and k for $(i,k)\in\mathcal{R}$. (In case of ties we choose the one which comes first in the lexicographic ordering of pairs).

We let $\alpha_{\hat{i}}$ be the least constant such that

$$d(x,y,P_{\hat{i}},\alpha_{\hat{i}}\ b_{\hat{k}}(x,y)) = 0 \text{ , for all } (x,y) \in \mathcal{L}(P)$$

and consider two cases

Case 1 The machine M makes at least $\alpha_{\hat{i}}$ different guesses of i_n .

In this case for n large we always have $\delta_n(\hat{i},\hat{k}) = 0$. If $\hat{i}$ is not i_n it is because there is an (i,k) pair found first. In particular we have $i_n < k_n < \max\{\hat{i},\hat{k}\}$ so that α_n must be eventually constant. By our choice of $(\hat{i},\hat{k})$ any pair (i,k) of lower $\max\{i,k\}$ will eventually be rejected because $\mathcal{L}(P_i)$ doesn't contain $\mathcal{L}(P)$ or because $\delta_n(i,k) = 1$ for all k such that $\max\{i,k\} < \max\{\hat{i},\hat{k}\}$. Thus in Case 1, i_n will eventually be $\hat{i}$ and k_n will eventually be $\hat{k}$.

Case 2 The machine M makes $\tilde{\alpha}$ different guesses of i_n and $\tilde{\alpha} < \alpha_{\hat{i}}$.

In this case we consider the class $\tilde{\mathcal{R}}$ of pairs (i,k) with the following properties

i) $\mathcal{L}(P_i) \supseteq \mathcal{L}(P)$

ii) $d(x,y,P_i,\tilde{\alpha}\ b_k(x,y)) = 0$, for all $(x,y) \in \mathcal{L}(P)$

If $\tilde{\mathcal{R}}$ were empty M would make more than $\tilde{\alpha}$ guesses. It is easy that if n is large enough we will have $i_n = \tilde{i}$ and $k_n = \tilde{k}$ where

$$(\tilde{i},\tilde{k}) = \min_{(i,k)\in\tilde{\mathcal{R}}} \max\{i,k\}$$

where ties are again broken by taking the least number in the lexicographic order. This completes the proof of Theorem 3 .

One might think that Theorem 3 can be formulated so that M can actually infer the least integer k such that some program whose graph contains $\mathcal{G}(P)$ has its running time bounded by b_k. We suspect that this cannot in general be done.

Example Suppose f_1, f_2 are recursive functions such that there are programs $P(f_1), P(f_2)$ which compute each argument in time b_1 and b_2, respectively, and that no program does better for infinitely many arguments. Let

$$f(n) = \begin{cases} f_1(k) & , \quad n = 2k-1 \\ f_2(k) & , \quad n = 2k \end{cases}$$

and consider the sequence of programs $P^{(i)}$ such that

$P^{(i)}$ uses $P(f_1)$ to compute for n odd, is undefined for $n = 2k$, $k > i$, and computes $f(2k)$, $k \leq i$ by a table.

Thus the program length $L(P^{(i)})$ will be unbounded, yet the running time of $P^{(i)}$ will be bounded by b_1. If an inference scheme considers only a bounded number of programs one may be able to infer that f can be computed in b_k time for some $k > 2$. If, however, the scheme considers more and more programs, one eventually encounters the $P^{(i)}$ which would cause the erroneous guess of time bound b_1.

We have not been able to convert examples of this sort into a proof that no machine can always find the lowest k for which there is a P_j with $\mathcal{G}(P) \subseteq \mathcal{G}(P_j)$ and $T(S_n, P_j) \leq b_k(S_n)$. However, we do know that any machine that attemps to always find the lowest possible k will have to look at arbitrarily many P_i for some functions. This can be forced by taking some function of class k and replacing

it on a finite number of arguments with a function of class k+1 .

This suggests the following modification to Theorem 3.

We supply the machine M with an auxiliary function $A(\ell,k)$ which maps a size and a bounding index into a size. This tradeoff function $A(\ell,k)$ determines the size of program to be considered in searching for an improvement to an answer P_j of size ℓ and complexity index k . Intuitively, $A(\ell,k)$ says that the user of the inference machine M prefers a program of class k-1 and size $A(\ell,k)$ to a program of size ℓ and class k .

There is a "natural" A function derived from the complexity function, namely

$$A(\ell,k) = \text{first size m such that}$$
$$c(\ell,T_{b_k}(S,P)) > c(m,0) .$$

The construction for Theorem 3 can easily be modified to include $A(\ell,k)$. This still does not guarantee the minimum value for k , but seems to be a natural model of inference processes.

Remark 3 There has been a considerable amount of work [12] on complexity classes of functions. To remain consistent with this work, we would have to restrict the choice of $\varphi(\cup\{T(x,y,P)\})$ to ones which give the same complexity classes as $\varphi = \max_{(x,y\in S)}(T_b(x,y,P_j))$. A good choice would be

$$T_b(S,P_j) = \max_{(x,y)\in S} T_b(x,y,P_j) + \frac{1}{|S|}\sum_{(x,y)\in S} T_b(x,y,P_j) .$$

This (max + average) measure gives the same classes as max, and also distinguishes among programs with the same maximum time. Since the ratio of this measure with the max measure is bounded away from 0 and ∞ ,

Theorem 3 holds for it also. Furthermore, it is also bounded away from zero, avoiding the degeneracy problem for the usual choices of $C(L,T)$.

The results derived here for programs have a significantly different flavor from those developed [7] for grammars. A central issue in grammatical inference is the presence or absence of negative information, i.e., strings in a sample marked as not belonging to the language being learned. This problem does not arise in program inference for two reasons. With grammars, an answer which generates too many strings is normally considered wrong, but our constructions allowing answers whose graph includes that of the hidden function seem quite natural. This arises from the single-valuedness of functions - if (x,y) appears in a sample then no (x,y') with $y \neq y'$ can appear. When $\mathcal{L}(M(S)) \supseteq \mathcal{L}(P)$, M has simply chosen a program which may be defined for some arguments where P is not. If one attempted to extend our results to relations, the problems associated with negative information would reappear.

The results of this paper should be viewed in the context of a renewed interest in inductive and scientific (hypothetico-deductive) inference. In addition to the theoretical work on programs and grammars, there is work on predicate calculus [16] and real chemistry [5]. All of these efforts have applied as well as theoretical components. Some of our work on program inference is discussed in [8] and [1] and there is a fairly ambitious effort underway to infer loop programs from sample traces. Thus far, there has been surprisingly little carryover from one domain to the other and from theoretical results to programs, but a common understanding of the issues seems to be

emerging. There are also proposed applications of inference-techniques to pattern recognition [9] and natural language description [14] which provide constant reminders of the weakness of existing results.

References

[1] Biermann, A., "On the inference of Turing Machines from sample computations," CS241, Stanford Computer Science Department, October 1971.

[2] Biermann, A. and J. Feldman, "A Survey of Grammatical Inference," in S. Watanabe, (Ed.), Frontiers of Pattern Recognition.

[3] Blum, M., "A Machine-independent Theory of the Complexity of Recursive Functions," J. ACM 14, No. 2, April 1967, pp. 322-336.

141 Blum, M., "On the Size of Machines," Information and Control 11, (1967), pp. 257-265.

[5] Buchanan, B., E. Feigenbaum, and J. Lederberg, " A Heuristic Programming Study of Theory-Formation in Science," Proc. 2nd ICJAI, London, 1971.

[6] Feldman, J. A., J. Gips, J. J. Horning and S. Reder, "Grammatical Inference and Complexity," CS 125, Stanford University, June 1969.

[7] Feldman, J. A., "Some Decidability Results on Grammatical Inference and Complexity," Information and Control, 1972.

[8] Feldman, J. A., "Automatic Programming," CS255, Stanford University, February 1972.

[9] Fu, K. S., "On Syntactic Pattern Recognition and Stochastic languages," TR-EE71-21, Purdue University, 1971.

[10] Gold, M., "Limiting Recursion," J. Symb. Logic 30, (1965), pp. 28-48.

[11] Gold, M., "Language Identification in the Limit," Information and Control 10, (1967), pp. 447-474.

[12] Hartmanis, J. and J. Hopcroft, "An Overview of Computational Complexity," J. ACM 18, 3 (July 1971), pp. 444-475.

[13] Hopcroft, J. E., and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass., 1969.

[14] Klein, S., et al, "The Autoling System," TR 43, Computer Science, University of Wisconsin, September 1968.

[15] Pager, D., "On the Efficiency of Algorithms," J. ACM 17, 4 (October 1970), pp. 708-715.

[16] Plotkin, G. D., "Automatic Methods of Inductive Inference," Ph.D. Thesis, Machine Intelligence Dept., University of Edinburgh, 1971.

[17] Rogers, H., Jr., Recursive Functions and Effective Computability, McGraw-Hill, New York, 1967.

[18] Simon, H., "Experiments with a Heuristic Compiler," Journal ACM, October 1963, pp. 482-506.

[19] Meyer, A. R. and D. M. Ritchie, The Complexity of loop programs, Proc. ACM 22nd Nat. Conf., pp. 465-9.