Mount, Kenneth R.; Reiter, Stanley

Working Paper

On Modeling Computing with Human Agents

Discussion Paper, No. 1080

Provided in Cooperation with:
Kellogg School of Management - Center for Mathematical Studies in Economics and Management Science, Northwestern University

Suggested Citation: Mount, Kenneth R.; Reiter, Stanley (1994) : On Modeling Computing with Human Agents, Discussion Paper, No. 1080, Northwestern University, Kellogg School of Management, Center for Mathematical Studies in Economics and Management Science, Evanston, IL

This Version is available at:
https://hdl.handle.net/10419/221437

Discussion Paper No. 1080

ON MODELING COMPUTING WITH HUMAN AGENTS

by

Kenneth R. Mount
and
Stanley Reiter

January 1994

On Modeling Computing with Human Agents
by
K.R. Mount and S. Reiter

Mount and Reiter in a *Model of Computing with Human Agents* [17] (referred to hereafter as M-R) propose a model applicable to computing units consisting of human beings, or computers, or both. The motivation for that model stems in part from the fact that performance of economic units and economic systems is constrained by the limited ability of human beings to process information, even with the help of computers and telecommunications equipment. Theoretical analyses that ignore limitations on human rationality may lead to irrelevant or erroneous conclusions. It has long been recognized that many phenomena of economic behavior and economic institutions that appear to be beyond the reach of current theory might be understood and analyzed via theories that take into account limitations on the ability of humans to calculate and reason.

Limitations on information processing abilities may be expressed in terms of direct constraints. Cost of information processing and decision-making in turn are determined by the interaction of direct constraints and resource prices. The kind of reasoning or information processing that is typically found in economic models of human behavior or the functioning of institutions typically involve calculations susceptible to an algorithmic treatment. Radner [19] has presented data showing the importance of information processing tasks in the economy, as indicated by the resources devoted to them.

Simon [26] and [27] has long advocated that economic theory take into account the boundedness of human rationality. On the whole terms like bounded rationality and "satisficing" point to the consequences of information processing limitations, on behavior or performance, but do not refer to an analytical model in which basic limitations of computational capacity can be expressed, and performance consequences derived from them. The problem seems acute in game theory, where games, especially repeated games, with fully rational players have so many equilibria that some people call into question the usefulness of the analysis. (It should be noted that models with multiple equilibia are important in explaining diversity. Matsuyama [13] has emphasized this point.) In recent years there have been several attempts to explore the implications of bounded rationality of players. These have mainly used the finite state automaton as a model of boundedly rational players, although recently perceptrons have been used for this purpose. To model a player as a finite state automaton expresses the constraint on her computational ability by the number of states of the automaton that models her. [See Neyman [19] Dilip Abreu and Ariel Rubinstein [2], Ehud Kalai and William Stanford [10], Ariel Rubinstein [23] and [24].]

Since the finite state automaton is a special case of the Turing machine, when memory is assumed to be finite, this model also incorporates the restrictions on computation that flow from the requirement that everything involved be finite. But there is as yet no consensus among game theorists that this model is the way

to go. Indeed, it seems to be the consensus that the finite state automaton is not a satisfactory or even useful model of human behavior under computational limitations. The more recent use of perceptrons to model players in repeated games, (see Ariel Rubinstein [25], In-Koo-Cho [7], In-Koo-Cho [8]) rather than finite state machines, perhaps a development growing out of this consensus, comes closer to the point of view of complexity that is taken in M-R. This is made clear in Example 3 below, an example taken from In-Koo-Cho [8].

The aim of M-R is to provide a model of computation, applicable to systems involving human agents as component elements, in which limitations on the ability of agents to compute can be made explicit, and the complexity of computational tasks can be analyzed. The M-R model is a generalization of McCulloch-Pitts [14] networks in a direction that, we argue, makes it particularly suitable for application to calculations performed by human beings in the context of economic models. Since the M-R model was first presented in 1982 [16]), a number of attempts have been made to model computation in a similar spirit. We note particularly the work of Radner [20] and Radner and van Zandt [21], using a network model of computing similar in some respects to that in M-R [17], and the work of Rubinstein [25] and In-Koo-Cho [8] referred to above. Blum, Shub and Smale in [5] have constructed a model of computing based on Turing machines with elementary functions that are Real algebraic functions.

We present in this paper an informal account of the M-R network model of computing, and illustrate how that model applies to computations performed by human beings. We do this by examining how certain tasks are performed by machines and by humans. Human beings can easily do some things that are very difficult for machines to do. For example, humans easily recognize visual patterns. The same task can be formidably difficult for a machine. There is a range of possibilities for models of human pattern recognition, from models based on the neurophysiology of vision, to black-box models, (i.e., models whose inputs are patterns and whose outputs are names of patterns, with no attempt to analyze the process in more detail). The challenge here is not just to make a model of human computation, but to make one that:

(a) connects appropriately with models that express our basic understanding of computation;

(b) that applies to computations performed by machines;

(c) that can be analyzed and usefully applied to economic models in which human behavior matters.

Our aim in this paper is:

A. to present some of the salient ideas of the network model presented in detail in M-R, with a minimum of formal technicalities;

B. to present some examples that we hope will make clear how that model can apply to human agents;

C. to show how the model can be useful in the analysis of economic models.

In formulating the M-R network model we have made several strategic choices. These include:

1) the class of functions (operations) considered elementary should be a primitive of the model, i.e., not fixed once and for all, but to be chosen according to the application to be made;

2) the complexity of a function should depend on the class of functions considered to be elementary;

3) functions defined on Euclidean spaces should be among the possible candidates for elementary computational operations;

4) there should be a formal procedure for computing functions that are defined on topological spaces, and for determining their complexity, because such functions arise in classical economic models;

5) the model should be sufficiently tractable to yield results, at least in some economic models, and sufficiently sensitive to distinguish among different economic mechanisms in such models;

6) the model should have a clear relationship to the standard models of computing, such as the finite state automaton.

It is clear that 1), 2), 3) and 4) make the concept of complexity in this model one that is *relative*; it depends on the class of functions or operations that we choose to regard as elementary.

This feature seems to us essential for a model to apply to human beings. If the elementary functions were fixed in advance for all applications, they would most likely be Boolean functions. While these model the switching devices that make up electronic computers, it is not at all clear that they are useful in modeling computations performed by human beings. The level of reduction of a model in which every computation must be reduced to binary operations, or some finite equivalent of them, seems too fine for the analysis of computations represented in economic models as carried out by human beings or groups of them.

In any event, in the present state of knowledge we would not be able to carry out such a reduction, even in relatively simple cases.

In this paper we do not deal with the relationship our model has to standard models of computing. The relationship of the model in M-R to the standard models of computing is treated in M-R, where limit theorems relate our complexity measure to standard measures of complexity of certain approximations in the finite automaton model. We interpret these results to say that our model is an idealization of finite computing in the same sense that using the real numbers to model measurement is an idealization of physical measurement processes.

We deal here only briefly with applications of the model of computing to analysis of economic mechanisms. We discuss a special case in which the process of figuring out the complexity of a function, and the use of a theorem giving certain necessary and sufficient conditions is illustrated.

The rest of this paper is organized as follows. First we present a sketch of the M-R network model. This exposition includes the concepts of *computability*

and of *complexity* used in that model. We also show how complexity is measured in that model. We present the definition of *computing an encoded version of a function*. Following that we discuss how the complexity of a function is analyzed. Finally we present three examples of tasks performed by persons, or person-machine combinations.

As a first example of a person-machine combination processing data, we describe an idea introduced by Chernoff [6] in which human beings and computers combine to detect patterns in observations that involve a large number of variables.

The second of these is a task that is complex and difficult for computing machines, but which is performed routinely by human beings, namely reading of handwriting. We show how the M-R network model deals with that task when it is performed by a human agent and when it is performed by a machine.

A third example is taken from the work of In-Koo-Cho [8], in which he analyzes the repeated prisoner's dilemma using perceptrons to model players. This is a different model of the information processing capabilities of players than is the finite state automaton, which is the model used in earlier study of the complexity of strategies in repeated games. He considers two special perceptrons , one with a single layer, and the other, the simplest perceptron with two layers. He shows that perceptrons of this kind are sufficiently powerful to generate strategies in the repeated prisoners' dilemma that form Nash equilibria in the full repeated game, and to yield the "subgame perfect folk theorem" in the full repeated game. Cho does not introduce a formal concept of complexity. However, the concept of "simplest perceptron" implicit in his analysis, and discussed in his paper, can be related to the concept and measure of complexity in the M-R model. This is carried out in the analysis of Example 3 in this paper.

Each of the perceptrons used in Cho's paper is a network of the type used in the M-R network model, and the complexity of the perceptron is a special case of the measure of complexity given in M-R [17] for computations performed by an M-R network. Thus, Cho's analysis provides an example of the useful applicability of the Mount and Reiter network model to an economic (game theoretic) model with human agents.

The network model of computing introduced in M-R is a version of the (discrete) McCulloch and Pitts [14] (hereafter called M-P) model in which modules can be continuous functions. (A discussion of M-P model can be found in Arbib [3] in a form more easily related to our work than the original M-P paper.) The general idea is to replace the finite alphabet used in the M-P model by an open neighborhood of the origin in a Real vector space of finite dimension, and replace the neurons of M-P by (smooth) functions defined on that neighborhood. The move to networks using vector-valued functions is a natural one, as is the representation of networks by directed graphs. The use of directed graphs is common in automata theory. (see [3], or [9].)

We do not claim much originality in this. However, we have been unable to find a reference that carries out the construction we use. As we have pointed

out, the model is presented here somewhat informally. The formalities can be found in Appendix C of M-R [17].

In the M-P model, (as presented in Arbib [3]), there is an alphabet A, which is a finite set of cardinality d, (d a natural number); a module is a function $f : \prod_1^s A \rightarrow A$, from the s-fold product of A to A. The function f can be thought of as modeling a computing device, such as a finite state automaton. The input set of this device consists of sequences $(a_1, ..., a_s)$, $a_i \in A$, and the output set of the device is the set A. This automaton has A as the set of states. If the function is in state q at time t, and f accepts the input $(a_1, \ldots, a_s)$ at time t, then the output of f at time t+1 is $f(a_1, \ldots, a_s)$.

It is sometimes convenient to think of the module f as a pair ((1,...,s),f) where,

(1) (1,...,s) is the sequence of indices of the variables of f,

i.e., indices for the coordinates of the domain of f ;

(2) f names the function, and every function $f : \prod_1^s A \rightarrow A$, can be a module. I.e., the set of elementary functions is $F^* = \{f \mid f : \prod_1^s A \rightarrow A\}$.

At our convenience we will denote the module either ((1,...,s),f) or f.

An M-P *network* is a finite collection of modules from F^*, together with a rule of interconnection that describes how outputs from modules in the collection are distributed among the inputs of the collection. This rule is itself a function. The domain of the *interconnection function* is a subset of the collection of all pairs [j,f], where j is the index identifying a variable of the module f. The range of the interconnection function is the set of modules of the network. Because the interconnection rule is a function, if a pair [j,f] is in the domain of the interconnection rule, then the rule assigns exactly one module, say, $((1, \ldots, s'), g)$ to that pair. We visualize the j^{th} input variable of f as connected to the output set of $((1, \ldots, s'), g)$ by a delayless wire or line. When the module g computes a value, that value is instantly relayed to the j^{th} input variable of f. Some of the pairs [j,f] can be outside the domain of the interconnection rule. Such a pair is a *network input line*. It is visualized as a wire running from a point outside the network to [j,f]. Some modules are designated as *output modules*. A wire runs from an output module to a point outside the network. The output modules are where the results of a computation are read. Sometimes an output module is called an *output vertex*.

The interconnection rule is a function from the finite set of pairs [j,f], to the finite set of modules of the network, therefore the interconnection rule can be represented by a directed graph or *digraph*, that has as vertices the modules of the network. If the interconnection rule assigns a module $((1, \ldots, s'), g)$ to the j^{th} input variable of the module $((1, \ldots, s), f)$ then the digraph has an arc that starts at the vertex $g = ((1, \ldots, s'), g)$ and ends at the vertex $f = ((1, \ldots, s), f)$. The arc from g to f is indexed by j, to show that the arc ends at the j^{th} variable of f.

In the M-R model we are considering, the alphabet A is replaced by a subset (a neighborhood of the origin) of a finite dimensional Real vector space of

dimension d. The modules of the network are functions chosen from a specified class $\mathcal{F}$. We refer to the model as an *$\mathcal{F}$-network*. The class $\mathcal{F}$ is a primitive of the model. For example:

(i) the class $\mathcal{F}$ is the class of analytic functions of s- tuples of d-dimensional Real vectors, $1 \leq s$, that have d- dimensional vector values;

(ii) the class $\mathcal{F}$ consists of d-dimensional vector-valued continuous functions that have as variables d-dimensional Real vectors.

Because one of the limitations on computational powers of human beings (and also, perhaps less significantly, of machines) is a bound on the number of variables that can be attended to simultaneously, we often distinguish for special attention a parameter, r, that may be implicit in the specification of the class $\mathcal{F}$ of elementary functions. The parameter r is a positive integer, and represents the number of variables (d-dimensional real vectors) that can be arguments of a function in $\mathcal{F}$. When this assumption is made, the number of inputs s of a module is subject to the restriction $1 \leq s \leq r$. When this assumption is made, we speak of an *(r,d)-network with modules in $\mathcal{F}$*.

An example is useful. The diagram in Figure 1 represents a (2,1)- network $\mathcal{C}$. The class $\mathcal{F}_\mathcal{C}$ of functions used in the network consists of four functions of the two Real variables A and B;

$$\mathcal{F}_\mathcal{C} = \{A + B, AB, A/B, \text{Identity function}\}.$$

Each vertex of the digraph that represents $\mathcal{C}$ is denoted by a box with a label that represents the function assigned to that vertex. The vertices are labeled with upper case letters (rather than lower case) identifying the modules; L_1 and L_2 are the input vertices of the network, while the output vertex of the network is labeled F_3. Each arc of the digraph is labeled by a letter. We use the same labeling for a variable and for the arc that indicates the assignment made by the interconnection rule.

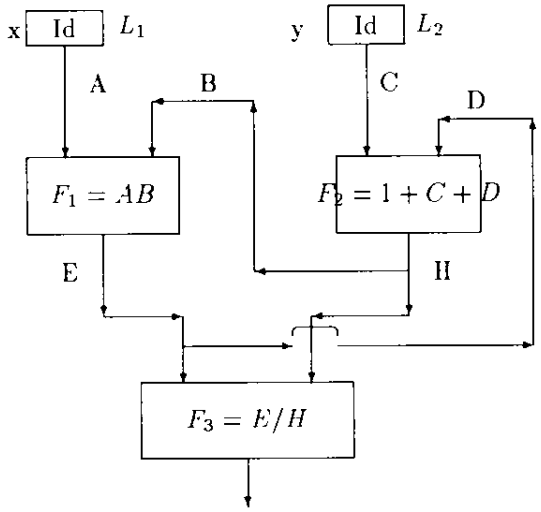

Figure 1

We consider next how an $\mathcal{F}$-network computes. We have assume that the output of a module f appears one unit of time after it receives its inputs. The *state of an $\mathcal{F}$-network* is an array whose entries are the state of each of the modules of the network in some prescribed order. We assume that the network is initially in some fixed state σ. A network with s input lines that is in state σ' acts on each s-tuple placed on the s input lines. If an s-tuple of values is placed on the network input lines, the network will undergo a sequence of changes of state over time. We assume that when the s-tuple of values on the input lines of the network is changed, the network returns to the fixed initial state σ for the start of the new computation. As long as the values of the s-tuple on the network input lines remains unchanged, the values produced by the network at the network output vertices at the end of any interval of time are functions of the s- tuple on the network input lines.

As an example, we return to the diagram in Figure 1. Assume that in the initial state σ, the vertices L_1, L_2, F_1, F_2, F_3 have the values 0,0,0,1,0, respectively. We represent the initial state σ by the row matrix

$$(00010).$$

	L_1	L_2	F_1	F_2	F_3
σ	0	0	1	0	0
t					
0	x	y	0	1	0
1	x	y	x	$(1+y)$	0
2	x	y	$x(1+y)$	$(1+x+y)$	$x/(1+y)$
3	x	y	$x(1+x+y)$	$(1+x)(1+y)$	$\frac{x(1+y)}{(1+x+y)}$
4	x	y	$x(1+x)(1+y)$	$(1+y+x(1+x+y))$	$\frac{x(1+x+y)}{(1+x)(1+y)}$
5	x	y	$x(1+x+y+x^2+xy)$	$(1+y)(1+x+x^2)$	$\frac{x(1+x)(1+y)}{(1+y+x(1+x+y))}$

Table 1

Table 1 shows the sequence of changes of state over time as the network $\mathcal{C}$ computes. Table 1 can be read as follows. The entry 0 in the column labeled F_1 and the row σ is the state of F_1 in the initial state σ of the network. The second row of the Table 1 indicates the new state of the network at time t = 0. At that time, the input lines of the network are changed to the state in which input vertex L_1 has state x and input vertex L_2 has state y, i.e., the values x and y are placed on the respective input lines at time 0. During the period from t =0 to t = 1, the state of the network changes to a new state in which the state of F_1 is the value of the module $F_1(A, B) = AB$. Since at t = 0 A has the value x and B has the value 1, which is the initial state of the vertex F_2, it follows that F_1 changes to the state x at t = 1.

After 4 units of time, when t = 4, starting from the time that x and y were first placed on the network input lines, the output line F_3, corresponding to the module F_3, carries the value

$$h(x, y) = \frac{x(1+x+y)}{(1+x)(1+y)}$$

and t = 4 is the earliest time at which this value appears on the output line of the network. The network $\mathcal{C}$ is said to compute the function h in 4 units of time.

Generally, as in the case of a finite M-P network (see Arbib [3]), an M-R network $\mathcal{C}$ is said to *compute a function F in time t from initial state σ* if t is the earliest time for which it is the case that for each sequence of values $(a_1, \ldots, a_p)$ assigned constantly to the network for t units of time, the value on the output lines of the network at time t is the function value $F(a_1, \ldots, a_p)$.

Complexity and computability

Let $\mathcal{F}$ be a class of modules, and let F be a given function. (It is understood that F maps a product of d-dimensional Real vector spaces into a d-dimensional Real vector space.) We may consider the class of all networks whose modules are in $\mathcal{F}$ and ask for the minimum time such that a network in that class computes F. That minimum time may or may not be finite.

If the time is not finite, we say the function F is *not computable* by a network with modules in F.

If a network in that class does compute F in finite time, then the complexity of F is the minimum over the class of networks of the time required to compute F. (If that time is not uniform over the domain of F, then the complexity of F is the minimum over the class of networks of the maximum time over the domain of F.) Thus, it is clear that the complexity of a computation is relative to the class of modules $\mathcal{F}$.

Computing an encoded version of a function.

In many situations it is necessary to evaluate a function F whose domain or range is not a subset of a Euclidean space. The complexity of such a function can be analyzed in the $\mathcal{F}$-network model by converting the computation of F into the computation of a function F^* derived from F, whose domain and range are acceptable to a network with modules in $\mathcal{F}$. The conversion is done using a real variable version of the concept of computing an encoded version of a function found in [3]. The idea is that the domain of the function is mapped into a product of Euclidean spaces by encoding functions, and the range of the function to be computed is embedded by one-to-one maps in a product of Euclidean spaces.

We also require that the encoding be done so as to preserve the product structure of the domain of F. The reason for this requirement is to accommodate a situation often encountered in economic theory in which the value of the function F describes some outcome for a group of agents each of whom is characterized by individual parameters.

For example, the function F might describe the allocation of resources among agents who are characterized by parameters. Then the domain of F is the product of the agents' parameter spaces, which of course need not be Euclidean. The definition of encoding we use requires that the product structure be preserved, i.e., the parameters of each agent must be kept together and associated with the relevant agent. The definition is the following.

Definition 1. Suppose that $F : X_1 \times \ldots \times X_n \rightarrow Y$ is a continuous function from a product of topological spaces X_i to a topological space Y. Suppose $V = R \times \ldots \times R$ is a d-fold direct product of the Real numbers. We say that an (r,d)-network $\mathcal{C}$ *computes an encoded version of* F in time t, if:

(a)there are Euclidean spaces E_i (E_i a w_i-fold direct product of copies of V) and continuous functions $g_i : X_i \rightarrow E_i, 1 \leq i \leq n$, and

(b) there are continuous functions $h_1, \ldots, h_b$ where $h_j : Y \rightarrow V$, such that the following conditions are satisfied:

(i) $h = (h_1, \ldots, h_b)$ is a bi-continuous one-to-one map to a topological subspace of $V \times \ldots \times V$ (h is an embedding),

(ii) there is a function $D = (D_1, \ldots, D_b) : \prod_1^n E_i \rightarrow V \times \ldots \times V$ from (Σw_i)-fold tuples of d-vectors to b-fold tuples of d- vectors that $\mathcal{C}$ computes in

time t,

(iii) the following diagram commutes,

$$\begin{array}{ccc} X_1 \times \ldots \times X_n & \xrightarrow{F} & Y \\ \prod_i g_i \downarrow & & \downarrow h=(h_1,\ldots,h_b) \\ E_1 \times \ldots \times E_n & \xrightarrow{D=(D_1,\ldots,D_b)} & V \times \ldots \times V \end{array}$$

The maps g_i encode the domain of F and the map h encodes the range of F.

Analyzing the complexity of a function

Consider next how to analyze the complexity of a function. Here we make explicit the restriction on the number of inputs that a module may have. (Without such a restriction, if the class $\mathcal{F}$ of elementary functions is sufficiently inclusive, an $\mathcal{F}$-network with one module would suffice to compute F. Thus, in this section, we confine attention to (r,d)-networks with modules in $\mathcal{F}$.

While, as we have just seen, the (2,1)-network in Figure 1 computes the value of h in time 4, it also computes

$$g(x,y) = \frac{x(1+y)}{(1+x+y)}.$$

in time 3 and

$$k(x,y) = x(1+x)(1+y)/(1+y+x(1+x+y))$$

in time 5. As this example illustrates, a given $\mathcal{F}$-network can compute many different functions, depending on the length of time the values on the input lines of the network remain unchanged. The number of functions computed by a given network can be arbitrarily large as time is allowed to increase.

For this reason, given an (r,d)-network and a function that it computes in time t, it is useful to construct another (r,d)-network that computes the given function in time t, using the same modules as the original network, and that computes the given function for all time after t.

Delooping

Suppose that $\mathcal{C}$ is an (r,d)-network with digraph G that computes a function F in time t. Suppose that the value of F is a d-dimensional vector. In that case, the network $\mathcal{C}$ has one output line. For simplicity suppose d = 1. Consider the special case in which the network G is a connected tree T with a single root to which each vertex can be connected by a sequence of arcs (directed edges) where for each arc in the sequence, except for the first, the beginning point is the endpoint of the previous arc. By a *tree* we mean a digraph without loops, even when the direction of edges is ignored. In the parlance of graph theory, each

vertex of the tree can be connected to the root by a *directed walk*. The *length of a directed walk*, i.e., of a sequence of arcs connecting a vertex to the root, is the number of arcs in the sequence. The maximum over the set of vertices in the tree of directed walks from a vertex to the root is the *length of the tree*.

It is a relatively easy task to show that if a function F can be computed by an (r,d)-network in time t using a finite subset $\mathcal{F}^*$ of $\mathcal{F}$, then one can construct an (r,d)-network whose directed graph is a tree, computes F in time t, and where the modules assigned to the vertices of the tree are either elements of $\mathcal{F}^*$, or are identity functions. The details of the delooping procedure can be found in M-R [17].

The (2,1)-network shown in Figure 2 constructed by this procedure computes in 4 units of time the function h also computed by the (2,1)-network shown in Figure 1 in 4 units of time.

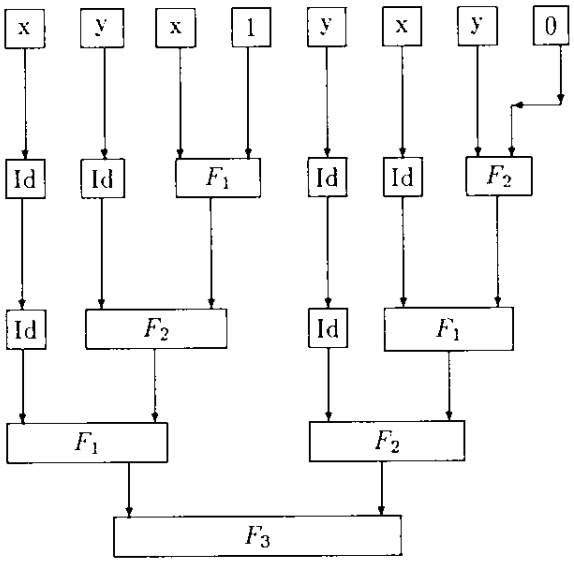

Figure 2

Superpositions

A network that is represented by a connected tree with a single root computes functions that are *superpositions* of the functions in $\mathcal{F}$, i.e. superpositions of functions that are the modules of the network. The *depth of the superposition*, i.e., the number of levels of functions used, is less than or equal to the length of the tree. Furthermore, if the network inputs are constantly on the input lines for a time that exceeds the length of the tree, the network output is constant for all times thereafter, and the depth of superposition is the length of the tree.

Thus, we see that *the complexity of a function F relative to (r,d)-networks with modules in the class $\mathcal{F}$, is equivalent to the minimum depth t such that F can be written as a superposition of functions from the class $\mathcal{F}$, with depth t.*

The complexity of F relative to (r,d)-networks with modules in $\mathcal{F}$, is related to a classical problem in mathematics, namely, Hilbert's Thirteenth Problem (see [12]). The essential substance of the problem is to decide whether a given function F of n variables can be written as a superposition of functions of fewer than n variables from a given class. It is implicit in the literature that the depth of the superposition expresses an intuitive notion of computational complexity.

When the functions in the superposition are restricted to be continuous and have fewer variables than F, Arnold and Kolmogorov have shown (c.f. [12] p.168, [27] Introduction]) that F can always be written as a superposition of continuous functions of fewer variables. Indeed they have shown that each function of n variables can be written as a superposition of continuous functions of *two* variables. Furthermore the depth of the superposition is bounded above by a function that depends only on the number of variables of F.

If, on the other hand, the functions used in the superposition are required to have the same degree of smoothness as F, then it is known that in general such a superposition representation cannot be guaranteed (c.f. [12], [27]).

Even if we reduce the class of functions and ask, as Hilbert did, whether an arbitrary analytic function of n variables, can be written as a superposition of analytic functions of at most two variables, then an argument of Hilbert says that the answer is No.

We may interpret these results as follows. The Kolmogorov and Arnold results suggests that there are *too many* continuous functions of two variables, and Hilbert's argument suggests that there are *too few* power series in two variables. Furthermore, in being too large, the class of continuous functions of two variables includes many functions that it would strain credulity to regard as elementary.

On the other hand real analytic functions of two variables can be considered as extensions or idealizations of arithmetic operations, especially if we restrict them to be truncated power series.

In any case, for the purpose of illustrating the analysis of complexity of functions, we take the class $\mathcal{F}$ of elementary functions to consist of power series in two variables up to degree M. Specifically, we assume:

A1) $\mathcal{F}$ consists of real analytic functions (power series) in two variables, truncated at degree M, with constant term identically 0, and linear term not zero;

A2) The functions to be computed, i.e., whose complexity is to be analyzed, consist of real analytic functions of n variables that vanish at the origin. They are to be computed only up to degree M.

Then, we say that a function F can be computed in time T to degree M if it can be written, to degree M, as a superposition of length T of functions from $\mathcal{F}$, but not as a superposition of length T-1.

Are there conditions on the function F that inform us whether it can be written as a superposition of length T of functions from $\mathcal{F}$, but not as a superposition of length T-1? And further, how can we construct a superposition of length T for F, or equivalently, an (r,d)-network with modules in F that computes F in time T?

For this informal presentation we restrict attention to representing F as a superposition of analytic functions of two variables. (See M-R [17] for a more general treatment.) Accordingly, throughout this discussion the function F and the class of elementary functions $\mathcal{F}$ satisfy assumptions A1) and A2) .

We use the following theorem, (Leontief [11] and Abelson [1].) We first state the theorem for a function

$$F : R^m \times R^n \rightarrow R,$$

and then illustrate in a special case how it can be used. We suppose that R^m has coordinates $\underline{x} = (x_1, \ldots, x_m)$ and R^n has coordinates $\underline{y} = (y_1, \ldots, y_m)$.In order to state the theorem we define two matrices associated with the function F. These are: 1)$BHF(\underline{x};\underline{y}) =$

$$\begin{pmatrix} \partial F/\partial x_1 & \partial^2 F/\partial x_1 \partial y_1 & \ldots & \partial^2 F/\partial x_1 \partial y_n \\ \vdots & \vdots & \ldots & \vdots \\ \partial F/\partial x_m & \partial^2 F/\partial x_m \partial y_1 & \ldots & \partial^2 F/\partial x_m \partial y_n \end{pmatrix}$$

where $\underline{x} = (x_1, \ldots, x_m)$, $\underline{y} = (y_1, \ldots y_n)$ and

2)BHF($\underline{y};\underline{x}$), is constructed by interchanging $\underline{x}$ and $\underline{y}$ in the construction of the matrix $BHF(\underline{x};\underline{y})$.

The Leontief result (as used by Abelson [1]) implies: A necessary and sufficient condition that a three times continuously differentiable function F with non-vanishing first partials can be written in the form $G(A(\underline{x}), \underline{y})$, where A is a function with continuous first derivatives in a neighborhood of a point $(\underline{a}, \underline{b}), \underline{a} = (a_1, \ldots, a_m), \underline{b} = (b_1, \ldots, b_n)$ is that the matrix $BHF(\underline{x};\underline{y})$ has rank at most 1. Abelson used this result to analyze the information transfer of a multistage distributed computation of the function F. The function $F = H(A(\underline{x}), B(\underline{y}))$ if and only if both matrices $BHF(\underline{x};\underline{y})$ and $BHF(\underline{y};\underline{x})$ have rank at most 1. (A complete proof can be found in Mount and Reiter [17] or [18].)

Suppose further that in addition to satisfying the hypotheses of the theorem F is a function of 2^N variables, that the linear part of F depends on all 2^N variables, and as assumed above F(0,...,0)=0. It is easy to see that the shortest time in which F can possibly be computed is N. To see this, note that to compute the linear combination of the 2^N variables x_j $j = 0, \ldots, 2^N - 1$ that is the linear part of F in minimal time, using (2,1)-modules in $\mathcal{F}$, one should compute as many of these as possible at the same time. This is done by the (2,1)-network shown in Figure 3, a *(2,1)- fan-in* of length N.

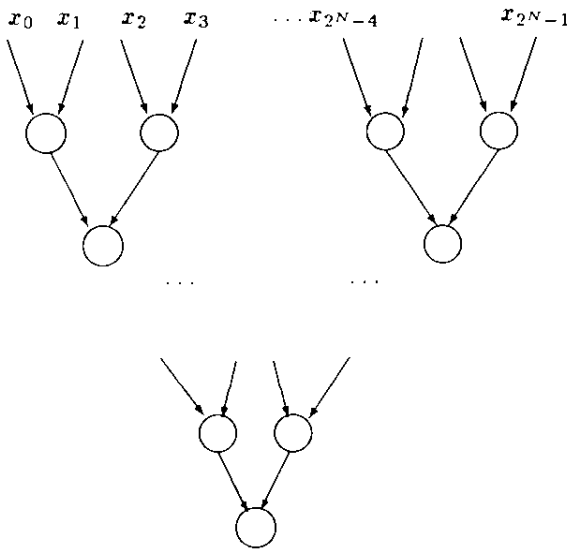

Figure 3

Now suppose that F can be computed up to degree M, (i.e., not just the linear part of F), in the minimal time N. Then the graph of the (2,1)-network that computes it must also be a fan-in of depth N. We need only specify the module assigned to each vertex of the fan-in in order to specify the network completely, i.e., to write F as a superposition of functions in F. We next show how to make that assignment. The Leontief-Abelson theorem can then be applied to verify that the assignment proposed can in fact be made.

Assigning modules to the vertices of the fan-in

We begin by labeling the vertices of the fan-in. This is done as follows. If the vertex is a leaf, (an input vertex) then it is labeled with the index of the variable that is input there. If the vertex is not an input vertex then it is labeled with an ordered pair. There are two cases:

1) The two inputs to the vertex v come from vertices labeled (i,j) and (k,l), respectively, where $i < k$. Then the label attached to v is (i,l);

2) The input lines of v are network input lines. Network input lines are labeled by the variables they carry. Thus, the inputs to v are the variables in an adjacent pair (x_{2j}, x_{2j+1}) for some j in $0, \ldots, 2^{N-1} - 1$. In this case the vertex v has the label (2j,2j+1).

This labeling is illustrated in Figure 4 for N = 3. The leaves are labeled in order from left to right,$0, 1, \ldots, 2^N - 1 = 0, 1, \ldots, 7$. The vertex whose inputs are x_0 and x_1 is labeled (0,1); the vertex whose inputs are the outputs of (0,1) and (2,3) is labeled (0,3), and so on. (There is no ambiguity in this labeling

scheme, because if a vertex has inputs from vertices (i,j) and (k,l), $i < k$ implies that $(i,l) \neq (i,j')$ for any vertex labelled (i,j').)

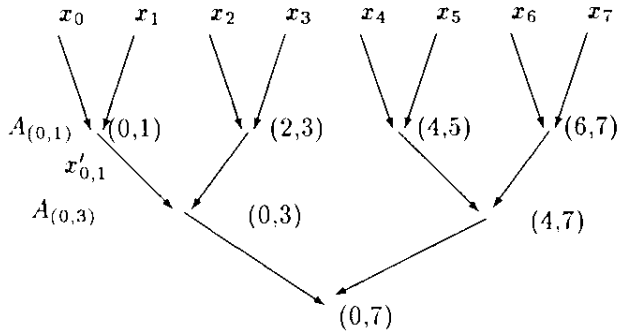

Figure 4

A function F computed by a (2,1)-network in minimal time determines the modules of the network essentially uniquely. The qualification 'essentially' refers to the fact that the modules are determined up to an equivalence relation, according to which two functions are equivalent if they are the same except for changes of variables in the domain and range. This is defined more explicitly below. In the interest of clarity and to minimize notational complexity, we illustrate the process of assigning modules for N = 3, and the tree in Figure 4. We suppose that

$$F : R^8 \rightarrow R,$$

where

$$F(x_0, \ldots, x_7)$$

is computed by a network whose tree is shown in Figure 4 with modules in $\mathcal{F}$.

The module assigned to vertex (i,j) is denoted $A_{(i,j)}$. Consider first the vertex (0,1), which has as inputs the variables x_0 and x_1, and therefore is assigned the function $A_{(0,1)}$. Consider how the network evaluates F at the point $(x_0, x_1, 0, \ldots, 0)$. Because the functions $A_{(i,j)}$ in F have the property that

$$A_{(i,j)}[0,0] = 0,$$

it follows that at any vertex (i,j) such that $i \neq 0$, the output of the module assigned to that vertex must be 0. Furthermore, at each vertex (0,j), where $j > 1$, only the left input line carries a value different from 0. Thus, each module $A_{(0,j)}$ where $j > 1$, acts like a (truncated) power series in one variable. Therefore, the composition of the modules from $A_{(0,3)}$ to the module assigned to the root, acts like a power series in one variable. Denote this composition h(z) (not to be confused with the function h of Table 1.) Thus

$$h(z) = A_{(0,7)}(A_{(0,3)}(z,0),0) \quad (E1).$$

Write $F(x_0, x_1, 0, \ldots, 0) = F_{(0,1)}(x_0, x_1)$. Then

$$h(A_{(0,1)}(x_0, x_1)) = F(x_0, x_1, 0, \ldots, 0) = F_{(0,1)}(x_0, x_1).$$

Similarly, for $F(0, 0, x_2, x_3, 0, \ldots, 0) = F_{(2,3)}(x_2, x_3)$,

$$g(A_{(2,3)}(x_2, x_3)) = F_{(2,3)}(x_2, x_3),$$

where,

$$g(z) = A_{(0,7)}[A_{(0,3)}(z), 0] = h(z).$$

The last equality is by equation (E 1). Therefore,

$$h(A_{(2,3)}(x_2, x_3)) = F_{(2,3)}(x_2, x_3).$$

I.e., the composition of the modules $A_{(0,j)}$ which is the function h works with both $A_{(0,1)}$ and $A_{(2,3)}$.

In the same way, the modules $A_{(4,5)}$ and $A_{(4,7)}$ can be assigned by using $F_{(4,5)}$ and $F_{(6,7)}$ defined analogously.

Next, define new variables, $y_{i,j}$, by

$$y_{i,j} = A_{(i,j)}(x_i, x_j) \text{ where} (i,j) = (0,1), (2,3), (4,5), \text{ or} (6,7),$$

and consider the network shown in Figure 4a.

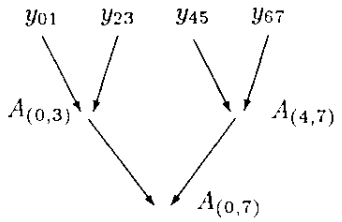

Figure 4a

The same process that was used to determine $A_{(0,1)}$ can be applied to the determination of $A_{(0,3)}$, using the function

$$G(y_{0,1}, y_{2,3}) = F(x_0, x_1, x_2, x_3, 0, \ldots, 0),$$

where $y_{01} = A_{(0,1)}(x_0, x_1)$ and $y_{23} = A_{(2,3)}(x_2, x_3)$. The other modules in Figure 4a can be assigned in the same way.

Let us define an equivalence relation on real analytic functions of two real variables as follows.

Two (real analytic) functions B(x,y) and C(x,y) are *equivalent* if there are nonsingular(have nonzero linear terms) analytic functions u, k, l of one variable such that

$$B(x, y) = u(C(k(x), l(y)).$$

The constructions illustrated in Figures 4 and 4a indicate that if the function F can be computed by a superposition as in Figure 4, then the modules $A_{(i,j)}$ are unique to within equivalence, i.e., they are equivalent to the functions $F_{(i,j)}$.

We digress here to present a determination of the modules $A_{(0,j)}$ in Figure 4, that makes explicit use of the power series expressions for all the functions involved. This material can be skipped by readers who find the foregoing sufficiently explicit. We assume that M = 2, i.e., that F is to be computed up to degree 2, and the modules are power series truncated at degree 2.

Then,

$$F(x_0, \ldots, x_7) = \Sigma_{i=0}^{7} a_i x_i + \Sigma_{i=0}^{7} b_i x_i^2 + 2\Sigma_{i=0,7\, j=0,7,\, i<j} c_{ij} x_i x_j.$$

Using the power series for F and the modules $A_{(0,j)}$ we have,

$$F_{(0,1)}(x_0, x_1) = a_0 x_0 + a_1 x_1 + b_0 x_0^2 + b_1 x_1^2 + c_{01} x_0 x_1$$

$$A_{(0,1)}(x_0, x_1) = \alpha_0 x_0 + \alpha_1 x_1 + \beta_0 x_0^2 + \beta_1 x_1^2 + \chi_{01} x_0 x_1,$$

$$A_{(0,3)}(y) = \alpha_0' y + \beta_0' y^2,$$

$$A_{(0,7)}(y', 0) = \alpha_0'' y' + \beta_0'' y'^2,$$

where,

$$y = \alpha_0 x_0 + \alpha_1 x_1 + \beta_0 x_0^2 + \beta_1^2 + \chi_{01} x_0 x_1,$$

$$y^2 = \alpha_0^2 x_0^2 + 2\alpha_0 \alpha_1 x_0 x_1 + \alpha_1^2 x_1^2,$$

and hence,

$$y' = A_{(0,3)}(y, 0) = \alpha_0'(\alpha_0 x_0 + \alpha_1 x_1 + \beta_0 x_0^2 + \beta_1 x_1^2 + \chi_{01} x_0 x_1 +$$

$$\alpha_1'(\alpha_0^2 x_0^2 + 2\alpha_0 \alpha_1 x_0 x_1 + \alpha_1^2 x_1^2).$$

Using the expression for y', and calculating y'^2, we evaluate $A_{(0,7)}(y', 0)$ to obtain,

$$\begin{aligned} z &= A_{(0,7)}(y', 0) \\ &= \alpha_0''(\alpha_0'\alpha_0 x_0 + \alpha_0'\alpha_1 x_1 + (\alpha_0'\beta_0 + \alpha_1'\alpha_0^2)x_0^2 + \\ &\quad (\alpha_0'\beta_1 + \alpha_1'\alpha_1^2)x_1^2 + (\alpha_0'\chi_{01} + 2\alpha_1'\alpha_0\alpha_1)x_0 x_1 + \\ &\quad \beta_0''((\alpha_0'\alpha_0)^2 x_0^2 + 2\alpha_0\alpha_1(\alpha_0')^2 x_0 x_1 + (\alpha_0'\alpha_1)^2 x_1^2). \end{aligned}$$

Because the superposition we have obtained is required to compute $F_{(0,1)}$, i.e.,because, for all (x_0, x_1),

$$F_{(0,1)}(x_0, x_1) = z,$$

we may equate coefficients of like terms, which yields the equations,

$$\begin{aligned} a_0 &= \alpha_0''\alpha_0'\alpha_0, \\ a_1 &= \alpha_0''\alpha_0'\alpha_1, \\ b_0 &= \alpha_0''(\alpha_0'\beta_0 + \alpha_1'\alpha_0^2) + \beta_0''(\alpha_0'\alpha_0)^2, \\ b_1 &= \alpha_0''(\alpha_0'\beta_1 + \alpha_1'\alpha_1^2) + \beta_0''(\alpha_0'\alpha_1)^2, \\ c_{01} &= \alpha_0''(\alpha_0'\chi + 2\alpha_1'\alpha_0\alpha_1) + \beta_0''(2\alpha_0\alpha_1(\alpha_0')^2). \end{aligned} \qquad (E2)$$

These are five equations in nine unknowns. We specify the values of four of the variables arbitrarily and solve for the remaining five. Thus, let

$$\alpha_0' = \alpha_0'' = \alpha_1' = \beta_0'' = 1 \qquad \text{(E3)}.$$

Then the equations (E2) reduce to

$$\begin{aligned} a_0 &= \alpha_0, \\ a_1 &= \alpha_1, \\ b_0 &= \beta_0 + 2\alpha_0^2, \\ b_1 &= \beta_1 + 2\alpha_1^2, \\ c_{01} &= \chi_{01} + 4\alpha_0\alpha_1. \end{aligned}$$

or equivalently,

$$\begin{aligned} \alpha_0 &= a_0 \\ \alpha_1 &= a_1 \\ \beta_0 &= b_0 - 2\alpha_0^2 \quad (E4) \\ \beta_1 &= b_1 - 2\alpha_1^2 \\ \chi_{01} &= c_{01} - 4\alpha_0\alpha_1. \end{aligned}$$

Equations (E3) and (E4) determine the modules $A_{(0,j)}$ for j=1,3,7. To verify that these modules do in fact compute $F(x_0, x_1, 0, \ldots, 0)$, we substitute from (E3) and (E4) into (E1). Then the expression for z becomes

$$z = x_0a_0 + x_1a_1 + x_0^2(b_0 - 2a_0^2 + 2a_0^2) +$$

$$x_1^2(b_1 - 2a_1^2 + 2a_1^2) + x_0x_1(c_{01} - 4a_0a_1 + 3a_0a_1) =$$

$$a_0x_0 + a_0x_0 + b_0x_0^2 + b_1x_1^2 + c_{01}x_0x_1 =$$

$$F(x_0, x_1, 0, \ldots, 0).$$

The process for determining the modules of the fan-in that computes F in the general case yields the following information about the expression of F as a superposition of functions in $\mathcal{F}$. Looking first at the root of the tree for F, we see that

$$F(x_0, \ldots, x_{2^N-1}) =$$

$$H(F(x_0, \ldots, x_{2^{N-1}-1}, 0, \ldots, 0), F(0, \ldots, 0, x_{2^{N-1}}, \ldots, x_{2^N-1})),$$

where H is shorthand for $A_{(0,2^N-1)}$.

The function $F(x_0, \ldots, x_{2^{N-1}-1}, 0, \ldots, 0)$, or a function equivalent to it, is computed in minimal time by the subtree whose root is the vertex $(0, 2^{N-1}-1)$. Similarly the function $F(0, \ldots, 0, x_{2^{N-1}}, \ldots, x_{2^N-1})$, or a function equivalent

to it, is computed in minimal time by the subtree whose root is the vertex $(2^{N-1}, 2^N - 1)$. In Figure 3 these are the subtrees with root (0,3) and (4,7) respectively.

Recall that the Leontief-Abelson theorem gives necessary and sufficient conditions for these computations. Therefore that theorem gives us two ways to verify whether the computation indicated in Figure 2 can actually be carried out, i.e., whether the functions involved exist.

We illustrate the application of the theorem in a simple example.

The function P to be computed gives the price as a function of the parameters characterizing the agents in a two-person, two-good exchange economy in which the utility functions of the agents are quasi-linear. Denoting these parameters by (x,z) for agent 1 and (x', z') for agent 2, the function P is given by,

$$P(x, z, x', z') = \frac{xz' - x'z}{x - x'}.$$

(See [16] p. 124 for the derivation of P.) The function P is to be computed by a (2,1)-network whose inputs are the variables x, z, x', z' in some order. In order to avoid singularity, we make a coordinate translation to coordinates R,S,T,U, where

$$R = x - 1,\ S = z,\ T = x' + 1,\ U = z'.$$

In the new coordinates

$$P(R, S, T, U) = \frac{S + U + RU - ST}{2 + R - T}.$$

Here the number of variables is $2^N = 4$, so that N = 2. Hence the minimal (2,1)-network is a tree whose depth is 2. In terms of superpositions, if P is computable in time 2, then there must exist real analytic functions A', B' and C', or A'', B'' and C'', defined on a neighborhood of the origin in R^2, such that P can be written,

$$P(R, S, T, U) = C[A'(S, T), B'(R, U)] \quad \text{(E5)},$$

or

$$P(R, S, T, U) = C''[A''(R, T), B''(S, U)] \quad \text{(E6)}.$$

Consider the case in which P is given by equation (E5). The Leontief-Abelson theorem states that a necessary condition for the existence of A', B', and C' satisfying (E5) is that the matrix

$$(E7) : BHP(S, T; R, U)_{(0,0;0,0)} = \begin{pmatrix} \frac{\partial P}{\partial S}(0,0) & \frac{\partial^2 P}{\partial R \partial S}(0,0) & \frac{\partial^2}{\partial S \partial U}(0,0) \\ \frac{\partial P}{\partial T}(0,0) & \frac{\partial^2 P}{\partial R \partial T}(0,0) & \frac{\partial^2 P}{\partial U \partial T}(0,0) \end{pmatrix}$$

have rank at most 1. But P, being real analytic, can be written in power series in the form

$$P = (S + U + RU - TS)(\Sigma_{j=0}^{\infty}(-1)^j(\frac{1}{2})^{j+1}(T-R)^j) =$$

$$(\frac{1}{2})[(S + U + RU - TS) + \frac{(S+U)(T-R)}{2}] + \theta,$$

where θ is a sum of monomials in R,S,T,U of degree at least 3. Evaluating the matrix in (E7) we see that

$$BHP(S,T;R,U)_{(0,0;0,0)} = \begin{pmatrix} \frac{1}{2} & \frac{-1}{4} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix}$$

which has rank 2.

We try next the possibility that

$$P(R,S,T,U) = C''[A''(R,T), B''(S,U)].$$

In this case the matrix BHP has the form

$$BHP(R,T;S,U)_{(0,0;0,0)} = \begin{pmatrix} \frac{\partial P}{\partial S}(0,0) & \frac{\partial^2 P}{\partial R \partial S}(0,0) & \frac{\partial^2 P}{\partial S \partial T}(0,0) \\ \frac{\partial P}{\partial U}(0,0) & \frac{\partial^2 P}{\partial R \partial U}(0,0) & \frac{\partial^2 P}{\partial T \partial U}(0,0) \end{pmatrix}.$$

When evaluated as above, this matrix is

$$\begin{pmatrix} \frac{1}{2} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

which has rank 2. Thus, the necessary condition of the Leontief-Abelson condition is not satisfied in either case. Therefore P cannot be computed in a neighborhood of the origin by a (2,1)-network with real analytic modules in less than 3 units of time from the inputs R,S,T,U. However, P clearly can be computed from R,S,T,U in 3 units of time.

Application of the $\mathcal{F}$-network model in examples

Example I. Chernoff's Faces

As a first example of the way in which humans and computers can interact to analyze complex information, we give an example in which the class of functions $\mathcal{F}$ used in the construction of an $\mathcal{F}$-network contain functions that are easily evaluated by humans, but so far have not been handled efficiently by computers. Human beings are good at seeing patterns. The ability to see patterns seems to depend on the structure and functioning of the human visual apparatus. Consequently, this ability applies to patterns in the space, or space-time, that the visual system evolved to function in, i.e. , in at most 3 or 4 dimensions.

On the other hand, situations arise in which we would like to detect patterns in high dimensional data, say observations represented by points in R^k, for k a positive integer much larger than 4. Computers are good at handling data of high dimensionality. While there are algorithmic processes, such as discriminant analysis or cluster analysis, for detecting patterns or regularities in some cases, we do not have algorithms for recognizing patterns that do as well as humans when restricted to low dimensions. Therefore, the idea of combining the power of computers to manipulate data with the ability of humans to see patterns is appealing. Indeed, the practice of making graphical representations of data as a way of bringing to bear the human visual system predates the electronic computer, and is widely used in physical, biological and social science and mathematics, as will as in business and everyday affairs.

Chernoff [6] introduced the idea of combining human beings and computers to detect patterns in a sample of observations of a relatively large number of variables. Specifically, he introduced the graphical representation of multidimensional data as *cartoon faces* drawn in two dimensions, and illustrated its use by two examples. These are: (i) a set of 8 measurements made on each of 87 fossils, and (ii) a set of 53 observations of 12 variables taken from mineral analysis of a 4,500 ft. core drilled from a Colorado mountain side.

The data are encoded as faces by a program that provides for up to 18 parameters that govern 18 features of a face. E.g., one variable determines the horizontal distance between the eyes; another determines the height of the eyes; another determines the curvature of the arc that forms the mouth, etc. If the number of variables observed is $k \leq 18$, then 18-k variables are fixed at some value and the remaining k variables determine the variable features of a face for each point observed. The computer prints out the set of faces, and a human being looks for a pattern in them. In the example with measurements made on fossils, the pattern sought was a classification of the fossils into groups of similar ones. In the second example, the observations were assumed to be generated by a multivariate stochastic process, and the problem was to detect a point in the time series of observations at which the process changed character.

Let

$$S \subset R^k, S = \{x^1, \ldots, x^n\},$$

be the sample of observations, each a k-dimensional point. Let

$$\eta : S \rightarrow R^2$$

be a correspondence, where

$$\eta(x^i) = y^i \subset R^2, i = 1, \ldots, n$$

is the subset of R^2 that constitutes the visual image encoding the observation x^i. (It is implicit in this notation that distinct points of S are assumed to be mapped to distinct subsets of R^2.) Thus, in Chernoff's first example, k=8, x^i is

the vector of 8 measurements made on the i^{th} fossil, and y^i is the cartoon face that encodes those measurements.

Since the problem is to classify the fossils, we seek a partition of the set S, or correspondingly a partition of the set

$$Y = \{y^1, \ldots, y^n\}.$$

Because S has n elements, the number of non-empty subsets in a partition of S, and *a fortiori* in a partition of Y, is a most n. Therefore a partition of S (or of Y) can be represented by characteristic functions as follows. Let

$$\xi :\rightarrow \{0,1\}^n$$

where

$$\xi = (\xi_1, \ldots, \xi_n),\ \xi_i : S \rightarrow \{0,1\},$$

and define

$$Q_i = \{x \in S \mid \xi_i(x) = 1\} = \xi_i^{-1}(1) \subset S.,$$

Then

$$Q = \{Q_1, \ldots, Q_r\}$$

is a partition of S, where, possibly after a renumbering of the characteristic functions, Q_i is the i^{th} nonempty subset defined by ξ.

Similarly a partition

$$P = \{P_1, \ldots, P_r\}$$

of Y is defined by characteristic functions

$$\chi : Y \rightarrow \{0,1\}^n,$$

where,

$$\chi = (\chi_1, \ldots, \chi_n),\ \chi_i : Y \rightarrow \{0,1\},$$

and

$$P_i = \{y \in Y \mid \chi_i(y) = 1\} = \chi_i^{-1}(1) \subset Y.$$

An algorithm, such as a cluster analysis of S, would compute a (vectorial) characteristic function ξ, using some class of elementary operations $\mathcal{F}$ in a computation representable by an $\mathcal{F}$-network. On the other hand, a human being looking at the set Y of faces encoding S would "compute χ directly." In that case, even if there were no algorithm for performing the required analysis on S, the function ξ could be defined as in Figure 5, and incorporated into the set $\mathcal{F}$. However, the interpretation which equates depth of superposition with time might be questionable if a long time were required for a person to make the classification of faces.

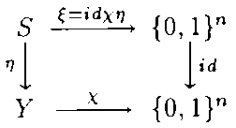

Figure 5

For our purposes here, Chernoff's second example differs from the first only in that the sets S and Y are ordered according to the time sequence of the observations and the functions ξ and χ are step functions, with the step at the point at which the stochastic process is deemed to change character.

Example II, Reading handwriting

Humans are very good at recognizing visual patterns, whereas that is often a difficult task for a machine. An important example of a computation that humans perform routinely and in many cases easily and that computers have great difficulty with is reading of handwriting.

The phrase "reading handwriting" can mean several different things. For example,

1) writing in non-cursive form,(i.e. print) the (English) expressions indicated by the given sample of cursive script;

2) uttering the expressions indicated by the given sample of cursive script;

3) extracting the "meaning" encoded in the cursive writing sample.

The first of these, namely, the translation of cursive script into printed form is still highly difficult, complex and even problematic for machines, as recent experience with Apple's Newton computer attests, while it is, of course routinely performed by literate persons,(though not without error). For example, Jenny reads aloud to Bob a postcard written (under conditions not favorable to good penmanship) by their friend Mike while hiking in Yellowstone Park.

A more impressive example is the reading of physicians' prescriptions by pharmacists.

We focus on the task of translating cursive statements into printed statements. Imagine a typesetter, a person or machine, who has before him (it) a manuscript (cursive statement) and a font of type, and whose task is to produce a sequence of type elements, (upper and lower case letters, punctuation marks, and spaces), that correctly translate the cursive manuscript into printed form.

A cursive writing sample is a plane 'curve', a curve that may have discontinuities, e.g., gaps between adjacent letters, due to the idiosyncrasies of handwriting of a particular person, or the normal spaces between words; it may have isolated points, e.g., the dot over the letter 'i' or the full stop that marks the end of a sentence; it may have crossing strokes, such as in the case of the letter 't'.

We may consider these curves to be concatenations of elements of a finite dimensional space consisting of conceivable finite samples of cursive script of no more than a given (unit) length. Thus we assume the writing to be constructed

of some collection of curves capable of being represented by a subset of a finite dimensional Euclidean space. Denote this space by C. (Other properties of curves that can be cursive writing samples, such as being of bounded variation, or having uniform upper and lower bounds on the height of letters could also be considered.) The space C can be made into a topological space, for example, by introducing a metric topology using a concept of the distance between curves of finite length.

The space, T, of printed text consists of a null element, and all finite strings over the alphabet made up of the elements of a font, such as the one referred to in the preceding paragraph. It is a topological space with the discrete topology.

The act of reading a cursive statement may be represented by a (continuous) function

$$\rho : C \rightarrow T.$$

Typically, this function will be many-to-one. Unreadable cursive samples are mapped to the null element of T. (If it is useful to do so, the function ρ may be assumed to depend on the person who writes, as well as on the person who reads, or both).

First, however, we consider how a machine might perform this act–read a given cursive writing sample. The curve that constitutes a cursive writing sample must be presented to the computer in a form the computer can accept. This may be done by a device like a scanner that converts the curve into a string of symbols from the alphabet recognized by the computer, or perhaps by a sensitive tablet on which the sample is written using some sort of stylus or light pen. The result of either of these input devices is an *encoded representation of the curve*. This may be as a graphic image, or an object specified by the equations that define the curve as a locus in two-space, perhaps relative to some given coordinate system. Another possibility, which however involves more information, is to give the curve parametrically, by equations

$$\begin{aligned} x(t) &= \phi_1(t) \\ y(t) &= \phi_2(t) \\ t_1 \leq \quad t \quad &\leq t_2 \end{aligned}$$

where (x,y) denotes a point in the plane, and t is in the interval between t_1 and t_2 in the Real line. This representation, or a discrete approximation to it, might describe someone writing cursively on a sensitive tablet.

Given this input, the computer would require programs to process the input into the ASCII code for the string of font symbols that constitute the output desired. Because the task is a complex one for a computer, the program is likely to be long and perhaps likely to produce incorrect results on many writing samples. (In the present discussion we may regard an incorrect result as equivalent to infinite computing time, i.e., we would have to wait forever for the correct result. A more satisfactory approach would be to measure the degree to which

the output approximates the correct result, but this seems too complicated for the present purpose.) The diagram in Figure 6 represents the situation.

In that diagram the function $e : C \rightarrow R^{l_1} \times \ldots \times R^{l_n}$ is an encoding of the elements of C, i.e., cursive statements, into elements of $R^{l_1} \times \ldots \times R^{l_n}$. (Note that if the encoding is done by a device such as a scanner, the encoding may depend on the position of the cursive statement on the screen of the device. In that case the coding would not be unique unless the positioning of the cursive sample is standardized. We may either assume that this is the case, or define a set of transformations in the plane that leave the cursive sample invariant except for position and define the encoding to be the same for any element of the equivalence class so generated. Evaluating this equivalence relation describes something humans do regularly in stabilizing the visual field; it is a complex task for a machine.)

Furthermore, the function, a, in the diagram is an encoding of T in $R^{l_1} \times \ldots \times R^{l_n}$. The ASCII code for alphanumeric characters is an example. (Here the null element of T is mapped to any element of $R^{l_1} \times \ldots \times R^{l_n}$ that is not the image of any character.) The inverse of the encoding a, performed by a device such as a printer, would produce the final result, the translation of the cursive writing sample into a printed writing sample.

$$\begin{array}{ccc} C & \xrightarrow{\rho} & T \\ {\scriptstyle e}\downarrow & & \downarrow{\scriptstyle a} \\ R^{l_1} \times \ldots \times R^{l_n} & \xrightarrow{f_\rho} & R^{l_1} \times \ldots \times R^{l_m} \end{array}$$

Figure 6

If the computer cannot read a cursive writing sample in one step, then the function f_ρ would not be elementary for that computer, i.e., not be in the set $\mathcal{F}$ consisting of the operations that are elementary for that computer. There would have to be a program written that computes f_ρ from the inputs using the operations that are elementary for that computer system.

The computation may be represented by an (r,d)-network with modules from the class $\mathcal{F}$, that computes f_ρ. The complexity of f_ρ is likely to be very high, if indeed that function is at all computable relative to the modules in $\mathcal{F}$.

If, for instance, the elementary operations consist of the arithmetic and logical operations, then a program that can read handwriting is likely to be long and involved, and the time required to compute F is likely to be long.

Consider next a person reading the cursive writing sample. The fact that a person can read cursive script may be expressed by saying that the evaluation of the function ρ is an elementary operation for that person. I.e., the person does it immediately or directly without any apparent intermediate steps, taking only a small (unit) interval of time per unit length of curve. Another way to describe this is that we do not analyze the process into steps internal to the reader.

While the function ρ is clearly a representation of the act of reading cursive writing, it is not in itself a useful model that meets the criteria stated above; it does not yet connect with any other model of computation, nor does there appear to be a way to use it in the analysis of economic models.

On the other hand, in modeling a person reading handwriting we may consider the function ρ to be equivalent to the composition

$$a^{-1} \circ f_\rho \circ e \equiv \rho \quad \text{(E8)}$$

in Figure 6. To say that a person can evaluate ρ in one unit of time can be interpreted as saying that the composition (E8) can be evaluated in one unit of time. This amounts to saying that the function f_ρ cannot take more than one unit of time. Hence it may be included as an elementary operation, i.e., a member of $\mathcal{F}$, in any application of the model in which a human being capable of reading cursive writing is among the computational resources.

For example, at the local drug store there is a pharmacist who reads the prescription form given to her by the customer, a form written by a physician in longhand. The pharmacist enters the prescription and other relevant information into a desk-top computer by typing it on the keyboard. The computer processes this input according to its internal program, a computation representable by an (r,d)-network. The act of translating a unit length of the handwritten prescription into type keys is elementary, and in the (r,d)-network model is formally seamless with the rest of the computation.

Example 3. Perceptrons Play the Repeated Prisoners' Dilemma

This example is taken from the work of In-Koo-Cho [8]. Game theorists have taken account of limitations on the capabilities of players to calculate their strategic behavior in repeated games, in particular, in the repeated prisoners' dilemma. Some have done this by modeling a player as a finite state automaton ([20,21,22,23]). The number of states limits what the player can figure out. In a closely related approach, the complexity of strategies has been studied by Kalai and Stanford [10]. Cho studies the repeated prisoners' dilemma using for players a model based on the perceptron. This model allows Cho to impose limitations on the capacity of a player to compute that are different from those imposed by the finite state automaton. Although Cho does not introduce explicitly a formal concept of complexity, his discussion makes it clear that he has a different idea of complexity than that implicit in the automaton model. The difference lies in the operations that are considered to be elementary. We describe his approach in terms that allow it to be compared with the concept of complexity introduced in the $\mathcal{F}$- network model. In order to make the exposition self contained we summarize Cho's model. He considers the 2×2 game G, a prisoners' dilemma, in which the set of actions of player i (i = 1,2) is $S_i = \{C, D\}$.

The set $S = S_1 \times S_2$ is the set of outcomes of the game G, and $u_i(s)$ is player i's payoff when the outcome is s in S. In the supergame G^∞ obtained by

repeating G infinitely many times, a history for the game played in period t +1 is

$$h^t = (s^1, \ldots, s^t),$$

where $s^j \in S$, for $j = 1, \ldots, t$. The set of all histories for time t +1 is H^t, and $H = \bigcup_{t=0}^{\infty} H^t$ is the set of all histories, where H^0 consists of the null history at the beginning of the game.

A strategy f_i for player i is a mapping from history to action in each period. I.e.,

$$f_i : H \rightarrow S_i, \ i = 1, 2.$$

is a strategy for player i in G^{∞}. The set of all repeated game strategies for player i is F_i, and $F = F_1 \times F_2$ is the set of all strategy pairs. Let $\sigma_i^t(f)$, be the action of player i at time t prescribed by the strategy pair f in F, and let $\sigma^t(f) = (\sigma_1^t(f), \sigma_2^t(f))$. The payoff to player i in G^{∞} is

$$\nu_i : F \rightarrow R,$$

where

$$\nu_i(f_1, f_2) = \liminf_{t \rightarrow \infty} (\frac{1}{T} \Sigma_{t=1}^{T} u_i(\sigma^t(f))).$$

As pointed out above, Cho models the players as perceptrons. [See Weisbuch [28] p.77 for a definition of perceptron. See also, Minsky, Marvin L. and Seymore A. Papert [15], or [22].]

A perceptron, as described by Weisbuch, is a network with three layers; an input layer, (called the *retina* by Weisbuch, and the *sensory layer* by Cho), a layer of *associational units*, and a *decision layer*. Because a perceptron has no feedback, i.e., is free of loops, the graph of the network is a tree. Because the output of the network is a single decision, the tree has a single root.

Cho's approach is to model each player by a simple perceptron, thereby limiting the set of strategies to those that are computable by such a perceptron; he studies properties of the equilibria of the game when only strategies in the restricted set are used.

In determining the restricted set of strategies he considers only two kinds of networks, those with one layer, and the simplest two- layered ones. Therefore, in this example we can confine our discussion to these perceptrons.

For a fixed t, for i = 1,2, the strategy f_i maps H^t into $\{C, D\}$. We may use the encoding function e to encode histories as strings of 0's and 1's. I.e.,for all $t \geq 0$,

$$e^t : H^t \rightarrow (\{0, 1\} \times \{0, 1\})^t.$$

Furthermore, for i = 1,2, the function

$$B_i : \{C, D\} \rightarrow \{0, 1\}$$

encodes S_i in 0's and 1's.

The following diagram applies the definition of a network computing an encoded version of f_i^t, where f_i^t denotes the restriction of f_i to H^t.

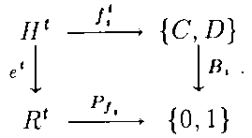

Figure 7

In Figure 7 the function P_{f_i} is the encoded version of f_i computed by a perceptron in the class admitted by Cho.

We describe the one and two-layer perceptrons considered by Cho in a way that facilitates comparison with the concept of complexity introduced for $\mathcal{F}$-networks.

In the case of a one-layer perceptron, the network is as shown in Figure 8.

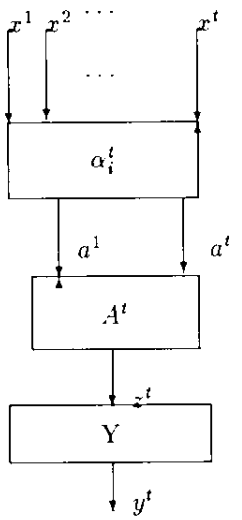

Figure 8

In Figure 8 the inputs are binary variables x_j, encoding a history h^t. The functions α_i^t associate a t-vector of real numbers to each encoded history of length t. (We use a slightly different notation than does Cho, who introduces functions $\alpha_i : S_i \to R$, and applies them histories $h^t = (s^1, \ldots, s^t)$ of length t, component by component, so that

$$\alpha_i(h^t) = (\alpha_i(s^1), \ldots, \alpha_i(s^t)).$$

This in effect makes α_i a function from the set of all histories to the space of all sequences of real numbers. The value $\alpha_i(h^t) = (\alpha_i(s^1), \ldots, \alpha_i(s^t))$, is

interpreted as being player i's perception of the history h^t. An example is

$$\alpha_i(s) = v_j^* - u_j(s) \quad i \neq j \ \forall s \in \{C, D\}^2,$$

where $v^* = (v_j^*, v_2^*)$ is any fixed individually rational payoff vector in the game.)

There may be several such perceptions. Because Cho considers only one-layered and the simplest two-layered perceptrons, it is sufficient to consider just two perception functions for each player, denoted α_i^{1t} and α_i^{2t}.

The next layer of the perceptron consists of associative units, which, according to Weisbuch, receive inputs from the units of the retina, in this case the values of the functions α_i. These inputs to the associative units are labeled $a^1, \ldots, a^t$ in Figure 8. Each associative unit (in Figure 8 there is just one such unit) computes a fixed function denoted A^t, of these inputs. In Cho's paper, the function A^t is

$$\Sigma_{t=1}^{T} \alpha_i(s^t) + A_i^0$$

where A_i^0 is a number, the initial value.

The last layer consists of a single decision unit, labeled Y in Figure 8, which accepts as input the outputs of the units of the preceding layer, labeled z^t in Figure 8, and has as its output the value y^t. The function Y is a *Heaviside function*, i.e., for ν in R,

$$Y_\nu : R \rightarrow \{0, 1\}$$

where,

$$Y_\nu(z) = \begin{cases} 1 & \text{if } z \geq \nu \\ 0 & \text{otherwise} \end{cases}$$

(We sometimes omit the index ν, especially when the Heaviside function is the one for $\nu = 0$.) Thus, the value of y^t, the output of the decision unit is either 0, or 1.

The perceptron in Figure 8 computes an encoded version of f_i^t if

$$f_i^t = B_i \circ Y \circ A^t \circ \alpha_i^t \circ e^t$$

or,

$$f_i^t(h^t) = B_i(Y(\Sigma_{j=0}^{t} \alpha_i^t(e^t(h^t)) + A_i^t)).$$

The perceptron itself is the superposition of the three functions, α_i^t, A^t, Y.

The composition

$$\theta_i^t = Y \circ A^t \circ a_i^t,$$

is called a *threshold automaton*. [Weisbuch [28], p. 47]. If we take threshold automata to be elementary functions, i.e., if we define the class $\mathcal{F}$ to consist of all threshold automata, where t can take any positive integer value, α_i^t can be any function from $0, 1^t$ to R^t, and Y any Heaviside function, then for every t, the network constituting the perceptron, shown in Figure 9 has one layer. I.e., for every t, the encoded version of f_i^t that it computes is expressed as a

superposition of functions in F of depth 1. We may define this to mean that the complexity of f_i is 1.

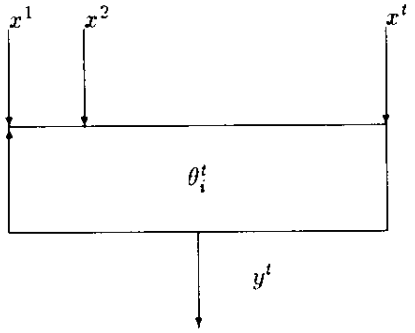

Figure 9

Cho also considers strategies generated by the simplest two-layer perceptrons. These are shown in Figure 10, in the notation used for the one-layer perceptron except where otherwise indicated.

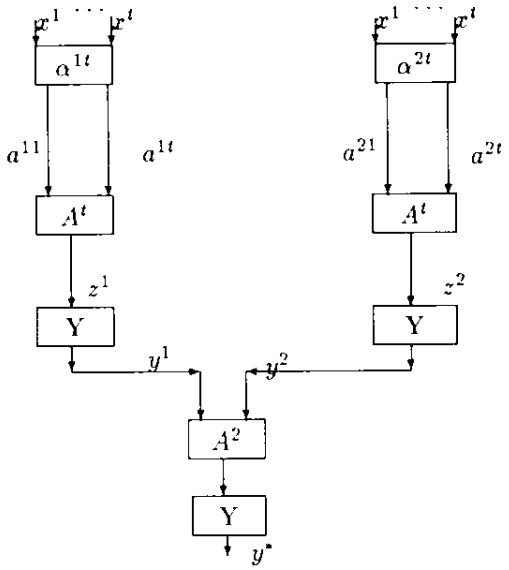

Figure 10

In Figure 10 the subscript i has been omitted, the functions α^{1t} and α^{2t} are two different functions from $\{0,1\}^t$ to R^t, and the function A^2 denotes a linear combination of its two arguments. I.e.,

$$A^2(y^1, y^2) = \beta^1 y^1 + \beta^2 y^2,$$

where β^1 and β^2 are real numbers.

The threshold automata in Figure 10 are

$$\begin{aligned} \theta_i^{1t} &= Y \circ A^t \circ \alpha_i^{1t}, \\ \theta_i^{2t} &= Y \circ A^t \circ \alpha_i^{2t}, \\ \theta_i^{3t} &= Y \circ A^2 \end{aligned}$$

Therefore, the network in Figure 10 can be written

$$\theta_i^{3t}(\theta_i^{1t}, \theta_i^{2t}),$$

or shown as in Figure 11.

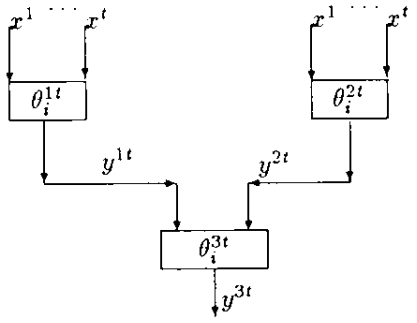

Figure 11

It is clear that if an encoded version of a strategy f_i^t is computed by a network as in Figure 11, then it has complexity 2, as measured by the depth of superposition of the elementary functions in the class of threshold automata considered by Cho.

In defining the class of elementary functions. Cho may be interpreted to have assumed that it is no more difficult for a player to add up, say, twenty numbers than to add up ten numbers. A finite state automaton would assign different complexity to these two addition problems.

Cho has also implicitly assumed that it is just as difficult for a player to convert any finite history in the game to an equally long string of real numbers as to convert any other history of different length, i.e., he assumes that the complexity of evaluating α_i^t is independent of t. This specification of the set $\mathcal{F}$ of elementary functions places the networks shown in Figures 10 and 11 within the $\mathcal{F}$-network model of computation in its general form. However,the perceptrons of Cho are not (r,d)-networks for any fixed r and d.

References:

[1] Abelson,H, *Lower Bounds on Information Transfer in Distributed Computations*, JACM, Vol 27, No.2, April 1980, pp. 384-392.

[2] Dilip Abreu and Ariel Rubeinstein [1988], *The Structure of Nash Equilibrium in Repeated Games with Finite Automata*, Econometrica, 56 pp. 1259-1288.

[3] Arbib, M.A., *Theories of Abstract Automata*, Prentice Hall, Inc. Englewood Cliff, New Jersey.

[4] Robert J Aumann [1981], *Survey of Repeated Games* Essays in Game Theory and Mathematical Economics in Honor of Oskar Morgenstern, Bibliographisches Institut, pp. 11-42.

[5] Lenore Blum, Mike Shub and Steve Smale, *On a Theory of Computation and Complexity Over the Real Numbers: NP - Completeness, Recursive Functions and Universal Machines*, Bull. AMS, vol. 21, No. 1, July 1989, pp. 1-46.

[6] Herman Chernoff, *The Use of Faces to Represent Points in k6-Dimensional Space Graphically*, Journal of the American Statistical Association, 68, 1973, pp. 361-368.

[7] In-Koo-Cho [1993], *Perceptrons Play Repeated Games with Imperfect Monitoring*, mimeo, University of Chicago.

[8] In-Koo-Cho [1993], *Perceptrons Play the Repeated Prisoner's Dilemma*, mimeo, University of Chicago.

[9] John J. Hopfield [1982], *Neural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proceedings of the National Academy of Science, 79, pp. 2554-2558.

[10] Ehud Kalai and William Stanford [1988], *Finite Rationality and Interpersonal Complexity in Repeated Games*, Econometrica, 56, pp. 397-410.

[11] Leontief, W. *A Note on the Interrelation of Subsets of Independent Variables of a Continuous Function with Continuous First Derivatives*, Bull. AMS, vol. 53 (1947), pp. 343-350.

[12] Lorentz, G.G., *Approximation of Functions*, Holt, Rinehart and Winston, New York 1966.

[13] Kiminori Matsuyama [1993], *Toward an Economic Theory of Pattern Formation*, CMSEMS Discussion Paper No. 1079, Northwestern University.

[14] McCulloch, W. and W. Pitts, *A Logical Calculus of the Ideas Imminent in Nervous Activity*, Bulletin of Mathematical Biophysics, V, (1943) pp. 115-133.

[15] Marvin L. Minsky and Seymour A. Papert [1988], *Perceptrons: An Introduction to Computational Geometry*, MIT Press.

[16] Kenneth R. Mount and Stanley Reiter [1982], "Computation, Communication, and Performance in Resource Allocation," paper presented at the CEME-NBER Decentralization Seminar, University of Minnesota (unpublished).

[17] Kenneth R. Mount and Stanley Reiter [1990], *A Model of Computing*

with Human Agents, CMSEMS Discussion Paper No. 890, Northwestern University.

[18] Kenneth R. Mount and Stanley Reiter [1993], *Essential Revelation Mechanisms and Computational Complexity*, CMSEMS Discussion Paper No. 1047, Northwestern University.

[19] Albert Neyman [1985], *Bounded Complexity Justifies Cooperation in the Finitely Repeated Prisoners Dilemma*, Economic Letters, 19, pp. 227-229.

[20] Roy Radner [1993], *The Organization of Decentralized Information Processing*, Econometrica, Vol. 61, No. 5 p. 1109.

[21] Roy Radner and T. VanZandt [1992], *Information Processing in Firms and Returns to Scale*, Annales d'Economie et la Statistique, No. 25-26, pp. 265-298.

[22] Frank Rosenblatt [1962], *Principles of Neurodynamics*, Spartan Books.

[23] Ariel Rubinstein [1979], *Equilibrium in Supergames with the Overtaking Criterion*, Journal of Economic Theory, 21, pp. 1-9.

[24] Ariel Rubinstein [1986], *Finite Automata Play the Repeated Prisoner's Dilemma*, Journal of Economic Theory, 39, pp. 83-96.

[25] Ariel Rubinstein [1993], *On Price Recognition and Computational Complexity in a Monopolistic Model*, Journal of Political Economy, 101, pp. 473-484.

[26] Simon, H.A. [1972] *Theories of Bounded Rationality*, Decision and Organization, ed. C.B. McGuire and Roy Radner, North- Holland pp. 161-176.

[27] Simon, H.A. [1987] *The Sciences of the Artificial* (2nd ed.), MIT Press.

[28] Ge'rard Weisbuch [1990], *Complex Systems Dynamics: An Introduction to Automata Networks*, Lecture Notes Volume II, Santa Fe Institute Studies in the Sciences of Complexity.

[29] Vituskin, A. G., *Complexity of Tabulation Problems*, Pergammon Press, Inc., New York 1961.