Discussion Paper No. 890

A Model of Computing With Human Agents

by

Kenneth R. Mount*) and Stanley Reiter**)
Northwestern University

June, 1990

---

*) Department of Mathematics, Northwestern University.
**) Department of Economics and Kellogg Graduate School of Management, Northwestern University. We have benefitted from comments from L. Hurwicz and R. Radner who cannot be held responsible for the use or lack of use of these remarks. This research was supported by National Science Foundation Grants No. SES-7715793, IST-8314504, and IST-8509678.

I Introduction

In this paper we present a model of computation intended to apply to human agents and computers. This model should permit us to express formally limitations on the information processing capacity of human agents. The broad aim of our research is to analyze the information processing task involved in operating an economic system. The focus is on the system of organization rather than on any particular set of activities, such as consumption or production. Economic theory has concentrated on the perfectly competitive market organization. The organization is thought of as a variable with several possible values. In that model, and more generally, economic theory has almost exclusively relied on optimization models (optimization is to be understood in a broad sense to include the kind of behavior dealt with in game theory.) The behavior of economic units, including individuals, households, and firms, is modeled as optimizing behavior. This has been the target of much criticism. We note specifically the criticism of Herbert Simon, who emphasized that optimizing models as they have usually appeared in economics pay no attention to the limited capacities of human beings to absorb, process and communicate information, and therefore often require agents to perform superhuman

feats of ratiocination. As a result, an economic system discussed without regard to information processing limitations can be utopian in the sense that the performance predicted by theory diverges greatly from actual performance. Simon pointed out that the behavior of economic agents in fact exhibits 'bounded rationality', and therefore differs in general from the behavior predicted by models that ignore those limitations. Simon's ideas about bounded rationality generated important work in several fields, including the behavioral theory of the firm in economics. But this approach has not had the degree of influence on economic theory that might have been expected from the introduction of an idea so evidently relevant and important. The reason for this, we think, is that the work built on it, for instance, in the behavioral theory of the firm, has for most economists an ad hoc character. It is often the case that there are several different heuristics that are equally plausible, that lead to different results, but the approach seems not to offer principles to justify choice among them. Furthermore, the approach does not generally yield behavior that is precisely specified. For instance, the notion that a manager of a firm tries to maximize profit is rejected, because to do so in a complex environment would exceed his powers of thought.

Instead the manager seeks to do acceptably well, (to 'satisfice'), but this does not in general lead to a quantitative prescription of the actions he will take.

What is needed is a model in which limitations on the capacities of humans, (individually, or in groups, operating with or without the aid of computers and communications devices), to acquire, process and transmit information can be expressed analytically, and in which the implications of such restrictions for individual behavior and the functioning of organizations can be studied. As a first step in this direction we aim 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 expressed, and the consequences of those limitations derived. In this our purpose is limited. We do not seek to model thought, or creative activity, or, at this stage, even learning. We focus on the kind of calculation that is typically found in economic models, mathematical calculations susceptible to an algorithmic treatment.

To motivate our model, consider the following thought experiment. Imagine that we are in a room containing a large, heavy table. We want this table moved into the next room through a doorway narrower than the width of the table. The protagonists in this

experiment are referred to as 'I' and 'he', and sometimes collectively as 'we', or 'us'. I see that the table is too big and heavy for me to move by myself. He will help me. Between us we should be able to move it. Before going further, notice that there is a problem of incentives here. We may have private objectives or concerns that diverge from or are even in conflict with the stated objective, to move the table into the other room. These issues, referred to as incentive problems, are the subject matter of a great deal of work in economics using games of incomplete information as the basic model. For the purposes of this experiment, we may abstract from incentive considerations. If we do so, what problem, if any, remains? It is clear that he and I must coordinate our actions in order to do the job.

Consider two cases.

First, suppose that he and I are the creatures envisaged in the optimization models, who are without information processing limitations. Each of us looks at the table and the doorway. We each apprehend all the aspects of the situation, including the size of the table in relation to the doorway and to our individual and collective strengths, and immediately figure out the optimal way to move the table, (assuming it is unique). Each of us would also know that the other has

the same data, and the same intellectual powers, and has therefore figured out the solution, including our proper individual roles. (If there were a sufficiently high degree of symmetry in the situation, such as each of us being equidistant from each of the lifting positions, we might have to agree about who would do what.) Each of us would therefore know his proper role, and we would do it.

Second, suppose he and I are mere mortals. We do not each immediately apprehend everything relevant to the problem, but rather see only some limited amount of the relevant information. Nor are we each able to deduce immediately the implications of what we observe. At some point, either after abortive attempts to move the table, or more likely on the basis of similar disillusioning experiences in the past, we realize that coordination will be needed and that it is not altogether obvious what to do. So, before doing anything more in the physical realm, we hold a conversation about how we are going to move the table. This problem may be a simple one relative to what one or both of us knows, or it may be complex. We might need to use information that is distributed between us, we might also have to divide the work of figuring out what to do. For instance I might take measurements of the door opening through which the table must pass, and

he might at the same time measure the table.

Even in the best of circumstances it may take us some time to figure out what to do. This can reduce the value of the answer we arrive at and of the actions we carry out.

What is clear is that the physical problem of moving the table has given rise to a symbolic problem of acquiring and communicating information and planning how to move the table. Limitations of physical capacities make it necessary to have two people involved; limitations of informational capacities can make it necessary to have both people involved in the information processing.

If, instead he and I were experienced moving men, it might well be obvious to us what to do. Our model should allow this possibility. This suggests that what we regard as elementary computational steps should not be absolute, but something we choose depending on the situation to be analyzed and the agents doing the analysis. Complexity of an information processing task--a computation-- would not be measured in absolute terms, but measured relative to the computational capabilities regarded as elementary for the situation at hand.

In our table-moving experiment the language that he and I use to analyze the problem might include for

example, real numbers, and smooth functions on them. We might want to apply calculus to this problem, which is, after all, one of mechanics. A model that formulates computing as a discrete process would not be directly applicable to such a formulation of the problem. In economic problems, including those of economic organization, the models usually are in terms of continua. In the simplest economic situations, such as trade between two people in two goods, the standard model, the Edgeworth Box, is stated in terms of real variables and continuous or smooth functions. A model of computing, or information processing, should be conveniently applicable to such situations. This might be done with a discrete model, i.e., a model with a discrete alphabet, but that would involve approximations. This would raise a question as to what extent any result depends on the approximation as opposed to the given data of the problem.

To play my part in moving the table I must internally coordinate the actions of various parts of myself. For some purposes it is attractive to think of a single model applicable to both the coordination internal to an agent and the coordination of several agents. Indeed, Marvin Minsky [20] might model the internal processes of a single person in terms of the interactions of several internal "agents". But it

seems likely that the level of resolution needed to model brain functioning, or intelligence in general, would require that problems of coordination and information processing that are relevant for economic organization be reduced to such terms as to make their analysis infeasible or useless for economics. This suggest that the coarseness of resolution of the model not be prescribed once and for all.

Finally, the process of computing, or reasoning proceeds by way of a limited class of elementary steps each of which takes a significant amount of time.

These considerations led us to a model in which, among other things, the elementary computational operations are primitive elements, and therefore are capable of being given different interpretations in different cases. In one important example evaluating a function of continuous variables is a primitive, or elementary, operation. While this makes the model applicable to economic examples without necessitating approximations, it raises questions about the relationship of this model, and particularly the measures of complexity coming from it, to analogous measures in the accepted models of computing, such as the finite state automaton, which typically involve only functions on discrete or finite alphabets. These relationships are clarified via limit theorems that

relate complexity measures and bounds on them for finite approximations to the analogous measures and bounds in the continuous model. The situation is analogous to that of measurement, say, of some linear dimension like length, where actual measurements can be at best rational numbers, but for analytical purposes the real numbers are used. This idealization is justified by a limit theorem.

The model of computation we present permits the formal expression of limitations on computational capacities. This is done by restricting the class of elementary operations.

Our model, which is presented in detail in Chapter III, is a continuous analog of the finite neural network model of McCulloch-Pitts [18], a model equivalent to the finite state automaton, or sequential machine. Such models are usually interpreted as models of electronic computers, which typically have fixed, though expandable, memories and which are made of devices that operate on finite alphabets. These models are not readily applicable to standard economic models, because economic models typically involve continua, while the finite state automaton is discrete. Motivated in part by the use to which we intend to put our model, we construct an idealized continuous model of computing, called the (r,d)-network. We do not

attempt to justify this model by pointing to physical devices[1)], but rather by showing that the continuous model is a limit, in several different senses, of finite state models, and that the indicator of complexity derived from our model is the limit of similar indicators in a sequence of approximating finite state models. This procedure has the advantage of making our model directly applicable to standard economic models without having to deal explicitly with approximations in each case. Another advantage of using a continuous model of computing is that it brings the subject into the domain of classical mathematics, and thereby opens the possibility of using the tools and methods of classical analysis.

Our continuous model is equivalent to the presentation of functions as superpositions(c.f. Lorentz [7]). This exposes a connection between our model and a literature in mathematics beginning with Hilbert's 13th problem, a problem that includes some aspects of the theory of approximation. Vituskin, Henkin, Lorentz and others [8,15] have used the concepts of entropy and capacity for compact subsets of normed spaces to investigate the possibility of

---

1) One might interpret an (r,d)-network as modelling a device similar to an analog computer. However, such a device is not formally an analog computer unless the elementary functions are from a specific class (c.f. [24]).

expressing a function of n variables as the superposition of functions of a smaller number of variables. In our model the question of computability is the question of whether a given function can be expressed as a superposition of functions from a specified class.

For a person equipped with a personal computer and appropriate software, to find the roots of a polynomial of given degree may well be regarded as an elementary operation, while the same person equipped only with pencil and paper, or even with the same computer but without the software, would find that task more difficult and complex. Furthermore, what constitutes a solution can vary with circumstances. A differential equation may be considered to be solved when we have an integral, but sometimes that same equation is not considered solved until we have a numerically specified solution path. To insist always on the latter concept of solution can render the model less useful especially in the context of theoretical analysis. Such considerations seem to us to point to the value of allowing the class of functions to be a primitive of the model, whose interpretation in a particular application expresses limitations on the computing power of the agents being modelled and through which the model can be connected to reality.

In our model of computation, as in the McCulloch-Pitts model, the primitive element is a module. Mathematically, a module is just a function. It can be visualized as a 'box' into which inputs (the values of the arguments of the function) flow and out of which outputs (the value of the function) flow some time later. Such boxes are wired together to form networks that perform computations. Limitations on the abilities of human beings to compute can be expressed in this model by the class of functions that are allowed to be modules. For example, psychologists tell us that the number of variables that a person can attend to simultaneously is small, about seven to eleven [19]. In our model we take the modules to be functions that have a limited number of arguments chosen from a given class, such as polynomials, vector valued functions of class C^n, or real analytic functions. The resulting modules are called (r,d)-modules, r being the number of variables permitted, each variable taking its values in a d-dimensional vector space; the value of the function is also a d-dimensional vector. We assume that a module takes one unit of time to compute a value. The measure of complexity of a computation is the least time it takes an (r,d)-network to carry it out. Thus, complexity of a computation is relative to the class of

functions taken to be elementary.

To illustrate the use of the model we apply it in this paper to analyze computational tasks involved in resource allocation, especially by means of a decentralized allocation mechanism. While this analysis may be more naturally carried out in a dynamic resource allocation process, the most extensively studied decentralized mechanisms are equilibrium models. We have taken equilibrium models as our example to illustrate the analysis. We study the trade-offs, if any, between communication and complexity in decentralized mechanisms. Is it possible to achieve a specified performance in different ways, some of which involve more communication and less complexity than others? In this paper we apply the model of computation to two familiar economic examples. This serves both as an illustration of the application of the model of computing, and as a first step toward a general analysis we are interested in.

We consider resource allocation mechanisms in their equilibrium form, as shown in the following diagram.

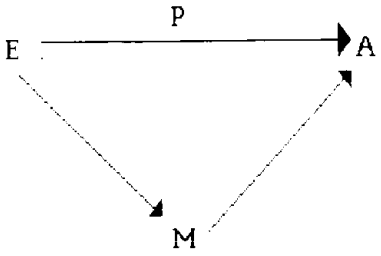

Figure 1.1

Figure 1.1

The set $E=E^1x \ldots xE^n$ is the set of economic environments, where E^i is the space of characteristics of agent i, A is the set of available actions and P is the mapping which associates to each environment a desired social action or outcome in A. Though P is often a correspondence, we assume here that it is a (single-valued) function. A mechanism $\pi=(\mu,M,h)$ consists of a message space M, an equilibrium message correspondence μ, which associates to each environment $e\epsilon E$ a subset $\mu(\ e\ )$ of messages, and an outcome function h, which translates those equilibrium messages into an outcome in A. The mechanism π is <u>decentralized</u> if the correspondence μ is privacy preserving. That is, each agent has a message correspondence, $\mu^i(\ e^i\ ):E_i\text{--->}M$ and $\mu(\ e^1x\ldots xe^n\ ) = \cap\ \mu^i\ (\ e^i\ )$. The mechanism π is said to <u>realize</u> the performance function P if $P = h\cdot\mu$, i.e., if for each environment the outcome specified by P is the same as the one resulting from π. In the examples that we study, the spaces E, M, and A are Euclidean and the function P is smooth.

An approach of computer science to the measure of the computational complexity of a function is to assign to a Turing machine computable function a time complexity that indicates the asymptotic character of the computation as the size of the input increases

(c.f.[9]). The measure derived using this theory is not sufficiently fine for the study of allocation mechanisms, but we recognize that using time as a measure of complexity is a standard approach. If we describe complexity by a cost function, then in our model the cost would depend on the number and capacity of modules, as well as time and other factors. Such a model could be used to analyze tradeoffs among these variables. In the interest of simplicity, we postpone such an analysis by restricting the measure of complexity to time. Note that because general (r,d)-networks can be reduced to equivalent loop free networks (Theorem C.1) a bound on the time required for a network to compute a function gives some information about the number of modules required to carry out a computation.[2)]

Given the performance function P there are generally several decentralized mechanisms that realize P. We focus here on two informational properties, the amount of communication required by the mechanism, and the amount of computing. The communication is measured by the dimension, m, of the message space M. Computation we suppose to be measured by a positive integer, t (time), defined in this paper, indicating

---

2) Roy Radner has studied some cases in which both the number of processors and time have entered the cost function [25].

computational complexity. Thus, each mechanism π would have associated with it a pair of integers $(m,t)(\pi)$ called the information-image of π, and representing the informational properties on which we focus, and the set of decentralized mechanism realizing P is mapped into a subset of the two-dimensional space whose coordinates are m and t (integer valued), called the information image of that set of mechanisms. This set represents the menu of mechanisms (insofar as information is concerned) from which a designer can choose. The lower boundary of the set is, of course, of particular interest because the efficient combinations of communication and computation are on that boundary. Figure 1.2 shows some possibilities.

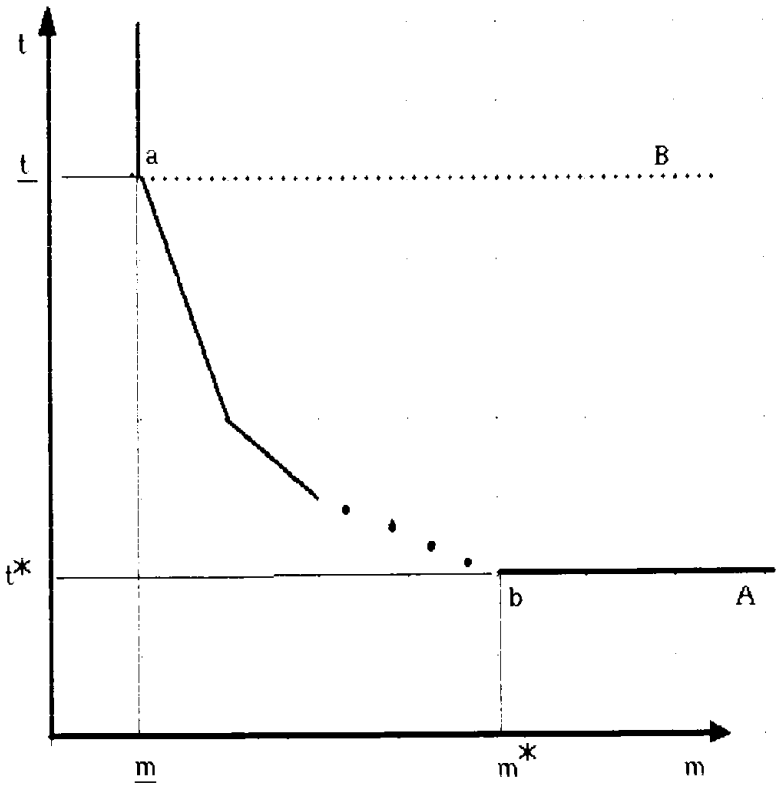

Figure 1.2

Figure 1.2

In Figure 1.2, the integer $\underline{m}$ is the lower bound on the dimension of the message space for the set of mechanisms realizing P. The point a corresponds to a mechanism whose message space is the one with minimum dimension $\underline{m}$, and whose measure of complexity $\underline{t}$ is the minimum over the set of mechanisms realizing P with a message space of dimension $\underline{m}$.

Two alternative types of lower boundary are indicated in Figure 1.2, one labelled A and the other B. If the lower boundary is like A, then there is a trade-off of computational complexity for communication, less complexity can be achieved by going to a bigger message space. This possibility exists over some range indicated by the point $b = (m^*, t^*)$ in Figure 1.2. From there on further increases in message space size do not result in a reduction in computation. If the lower boundary is like B then the points $b = (m^*, t^*)$ and $a=(\underline{m},\underline{t})$ coincide.

Two examples are analyzed using this model in Chapter VII. One of them, representing competitive allocations in classical environments, is an Edgeworth Box exchange economy, with two goods and two agents, each with a quadratic utility function involving two parameters. The performance function in this example is the Walrasian one. The other, representing nonclassical problems, has the same space of

environments, but the performance function is the inner product of the parameter vectors of the two agents.

The amount of computing required depends on the interpretation of the mechanism. In Chapter V, two interpretations are considered. First, the mechanism can be regarded as a one step iteration. Second, the mechanism is interpreted via the verification scenario. According to the verification scenario, the center 'posts' a candidate equilibrium message, and each agent determines whether or not that message satisfies his equilibrium condition, i.e., is an element of his message correspondence for the given environment. The efficient frontier of the information image is analyzed for each interpretation in Section II of Chapter VII. The model of computation is the (r,d)-network whose modules are real analytic functions of at most two real variables, i.e., r=2, and d=1. Such networks can accept as inputs at most two real numbers per input module. In the one step interpretation, each agent's message is a subset of the message space. Chapter V, contains an exposition of the computations required in the one step iteration process. It also contains a formal version of the computations required by the verification scenario, and the formal construction of the information image and its efficient frontier.

The efficient frontier in the first example, when

the equilibrium is calculated, is like A in Figure 1.2. For the verification scenario, it is like B, and for the inner product performance function it is like B. In Chapter VIII the effect of analytic coordinate changes in the parameter spaces of the agents and in the message space is discussed (Theorem 8.1 and Lemma 8.2). It is also shown that, at least locally, there is only one mechanism with a message space of minimum dimension that realizes the Walrasian function (Lemma 8.1).

It is possible to understand the model presented in this paper, and in particular, follow the discussion of the examples in Chapter VII, by reading Chapters II-VII. Chapter VIII studies the effect that analytic changes of coordinates in the parameter spaces of agents has on computing time. The remaining chapters address general questions on the computation of approximations of continuous functions by networks the use finite alphabets.

The paper is divided into two Parts and three appendices. Part I consists of Chapters II through VII.

In Chapter II the finite McCulloch-Pitts model is briefly summarized and a lower bound for computation time, due to Arbib and Spira [2], and involving the concept of (finite) separator set for an output line

(Theorem 2.2) is presented. Separator set is an important concept for expressing the essential information needed to compute a function.

Chapter III presents an informal discussion of our model of continuous computation based on McCulloch-Pitts networks. The model is formulated using graph theory. The necessary definitions and results about graphs and directed graphs and the formal model are presented in Appendix C. The (r,d)-network is defined formally (Definition C.12), as is what it means for such a network to compute a function in time t (Definition C.13). It is shown that an (r,d)-network that computes a function in time t can be replaced by an (r,d)-network, with the same class of modules, that is free of loops (Theorem C.1) and also computes the function in time t. An example of such a computation is presented in Chapter III.

Chapter IV deals with the extension of the concept of separator set (Definitions 4.1 and 4.5) to continuous functions. The definition of computing an encoded version of a function is given (Definition 4.3) and a lower bound on the time needed to compute a function by (r,d)-networks is established(Theorem 4.1- The Dimension Based Lower Bound). The formula giving the lower bound is analogous to that of Arbib and Spira, except that in the formula, cardinalities of

separator sets are replaced by dimensions of separator sets. It is shown that computation time depends on the various encodings involved. The problem of finding minimizing encodings for linear functions between linear vector spaces is analyzed (Lemma 4.3). An example is given.

Separator sets play an important role in Chapters IX and X in showing that certain lower bounds for the time needed to compute smooth functions with (r,d)-networks are limits of the lower bounds obtained for computing finite approximations to those functions using finite McCulloch-Pitts networks (Theorem 9.1 and Theorem 10.2). In Theorem 10.2 certain rank conditions on matrices of second derivatives of the function to be computed emerge. These are related to conditions in a theorem of Abelson [1] having to do with the communication necessary to evaluate a smooth function by a distributed computation. Abelson's theorem is a generalization of a theorem of Leontief [15]. Leontief used the result to study the independence of variables in a production function, or consumption function. The relations between separator sets and these various results and conditions are explored in detail in Chapter VI. In preparation for that certain equivalence relations are analyzed in Chapter IV (Definition 4.1 and Lemma 4.1).

In Chapter VI, concepts of adequate revelation and essential revelation mechanisms are introduced both for set functions and for topological functions, and certain universality properties established for functions satisfying a condition of differentiable separability (Definition 6.1, 6.3, and Theorem 6.3). A relation between essential revelation and the theorems of Abelson and Leontief is established in Theorem 6.5.

Chapter VI, as was noted above, deals with the relationships among the concepts of separator sets for smooth and finite functions, certain conditions, used in Chapters IX and X, on the derivatives of the function to be computed, and the theorems of Abelson and Leontief referred to above.

In Part II, Chapters IX to XI, limits of finite approximations are analyzed. In Chapter IX a fixed finite alphabet is used to compute finite approximations of a real valued function using approximating functions defined on lattices. In order to allow refinements of the approximation, the finite networks are allowed increasingly large numbers of output vertices. Under a condition of gradient separability (Definition 9.6) it is shown that locally, and in the limit, the Arbib and Spira lower bounds for the lattice approximations converge to the Dimension Based Lower Bound of Theorem 4.1. In Chapter X,

lattice approximations are again studied, only in this case the number of output vertices for the finite networks is fixed. To allow for refinements of approximation the alphabet is allowed to increase in cardinality. Again, convergence of Arbib and Spira lower bounds for finite local approximations to the Dimension Based Lower bound is established for a class of functions. The condition required on the functions to be approximated is differentiable separability (Definition 6.3 of Chapter VI). When a function is differentiably separated, separable sets can be studied using the concept of separator functions (Definition 10.1 and Definition 10.2). When a function $F:X_1x...xX_n--->R$ has separator functions, lattice approximations to F have readily constructed separator sets of cardinality related to the dimension of separable sub-manifolds in the X_i. From this one proves that the Dimension Based Lower Bound is a limit of Arbib and Spira lower bounds (Theorem 10.2).

In Chapter XI lattice approximations are computed by finite networks where the number of output vertices remains fixed, but the modules of the networks that compute the approximations are restricted to be of a specified class indexed by an integer (e.g., polynomials in a given number of variables indexed by degree), and the integer is allowed to grow. The

theorem of Chapter XI shows that the minimum computation time obtained from the (r,d)-network model is the limit of analogous computation times for the sequence of finite approximations. Thus, while the results in Chapters IX and X are for a lower bound, here the result is for the minimum computation time directly.

The three appendices present several technicalities. Appendix A discusses the concept of privacy preserving correspondence in the context of finite sets together with some examples. In Appendix A the definition of isomorphism of privacy preserving correspondence is introduced (Definition A2.3), and the relation isomorphism has to privacy preserving correspondences built from rectangular covers is analyzed (Theorem A3.1.) Appendix A ends with some lower bound results on the cardinality of message spaces for a function that is a characteristic function for a subset of a product X x Y. Appendix B is a discussion of the theorems of Abelson and Leontief. The results are used in Chapters VI, VII and VIII. Appendix C presents the formalities on graphs and networks, and is the formal presentation of the model of computing we use.

Chapter II

Finite Computation

We take as a point of departure a model called "modular network" introduced by McCulloch and Pitts [18]. That model was presented as a highly simplified representation of a network of neurons. It is also a model of the finite state automaton, or the finite state sequential machine. In this Chapter we present the McCulloch and Pitts model informally, following Arbib [3, pp 66 ff]. In Chapter III we present in a more formal way a more general version of the model that includes continuous computation. Before describing the McCulloch and Pitts model in more detail, it may be useful to clarify its relationship to automata and to the Turing Machine, perhaps the fundamental model of computing. A reader already acquainted with finite automata and Turing machines may wish to skip to the section on McCulloch and Pitts networks.

A sequential machine is characterized by three sets and two functions. There is a set X that is a finite set of inputs to the machine, a set Y that is a finite set of outputs from the machine, and a set Q that is the set of internal states of the machine. The

two functions are δ:Q x X ---> Q, which is the next state function, and the function Λ:Q x X ---> Y, which is the next output function. The machine processes finite sequences of elements from X, (the set of such finite sequences is denoted by X*), and produces elements of Y* (finite sequences of elements of Y). This processing is carried out in the following manner. Once an initial state q_0 for the machine is chosen, then a sequence $(x_1,\ldots,x_t)$ is fed to the machine, proceeding along the sequence from left to right. At the i^{th} stage of the process when the state of the machine is q_{i-1}, the machine changes to state $q_i=\delta(\ q_{i-1},x_i\ )$ and it prints the element $y_i=\Lambda(\ q_{i-1},x_i\ )$[3].

If the set Q is finite, then the sequential machine is called a finite state sequential machine, or a finite automaton. It is important to note that the i^{th} output of an automaton fed a string of t elements of X is determined by the first i elements of the input string and the initial state of the machine.

A Turing Machine is a computing device that consists of a tape (possibly infinitely long) that is divided into squares, a Head through which the tape is

---

3). This is the definition of a Mealy machine. There is another formulation that makes the output a function of the state alone. In that case the automaton is called a Moore machine. These two types of machine are equivalent. See [3, or 8].

fed, together with a Control Box that controls the action of the Head. (c.f. Figure 2.1).

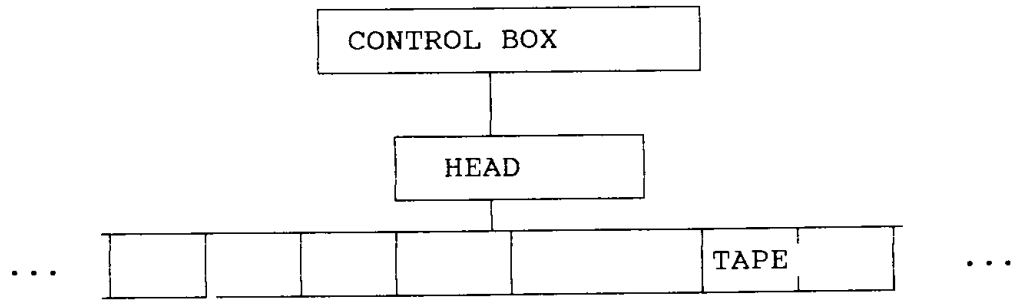

Figure 2.1

A finite alphabet A is given, that includes as an element "blank". Set X = A-{blank}. The Head can read elements of the alphabet from squares on the tape and communicate what it reads to the Control Box. The Head can also write on the tape; that is, it can r0xlace an alphabet entry on a square. Furthermore the Head can move one square at a time to the left, to the right, or it can hold its position. In one unit of time the Head reads a character on a square, communicates this to the Control Box, receives a command in response from the Control Box, writes something on the square it has just read, and then moves left or moves right or does not move. The Control Box, which determines the actions of 'read, 'write', and 'move' carried out by the Head, is a finite state sequential machine. That is, the Control Box consists of a finite set Q of states that includes a state "Halt", together with a function Prog

that maps the product Q x A to the product A x {Left,Stay,Right} x Q. At time t the Head reads an alphabet element from a square and communicates this to the Control Box. If the Control Box is in a state q, then the Box evaluates the function Prog(q,α) = (β,M,q'). The Head is then instructed to write β on the square that the Head has just read, and then carry out the move M. The Control Box then changes to state q'. If q' is not the state 'Halt' the cycle is repeated. If q' is 'Halt', then the process stops. If, as before, X* denotes the set of finite sequences of elements of X, then the way a Turing Machine with an initial state q_0 computes a function F from a subset S of X* to X* is as follows. If s is an element of S, then s is written in consecutive squares on an otherwise blank tape, the Head is moved to the left most non-blank entry and the Turing Machine is started in state q_0. The Turing Machine has computed F(s) if it halts after a finite number of operations with an element of X* on an otherwise blank tape, and that element is F(s).

The McCulloch and Pitts model is a model of the internal controls of a Turing Machine, that is, of the Control Box. It can be thought of, perhaps a bit fancifully, as a model of the brain of the Turing Machine.

II. McCulloch and Pitts Networks

The McCulloch and Pitts model has a finite alphabet A consisting of d elements, and a collection of functions, f(.), of t variables, $1 \leq t \leq r$, where $f(x_1,\ldots,x_t) \in A$ for each t-tuple $(x_1,\ldots,x_t)$ of elements of A[4]. Such functions are called (r,d)-modules. It is convenient to think of the module f as a device capable of computing the value $f(a_1,\ldots,a_t)$ once the values $a_1,\ldots,a_t$ have been assigned to the variables $x_1,\ldots,x_t$. If values $(a_1,\ldots,a_t)$ are assigned to the variables $(x_1,\ldots,x_t)$ in the module f, then f produces the value $f(a_1,\ldots,a_t)$ one unit of time later. The module is represented by the diagram in Figure 2.2.

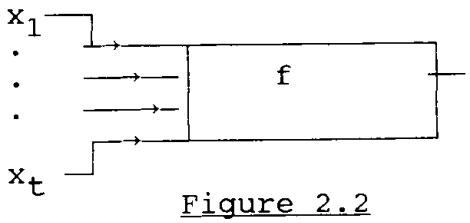

Figure 2.2

One thinks of the variables x_i as carried on input lines or wires into the "black box", f, and f has an output line, or wire, that carries the value of the function.

---

4) In the original McCulloch and Pitts model, the number of inputs r, though finite, was not restricted a priori.

In the terminology of automata theory, an (r,d)-module f is an automaton whose input set is the collection of t-tuples $(x_1,\ldots,x_t$, $x_i \epsilon A$ and $t \leq r$, whose output set is A, and whose set of internal states is A. The next state function is given by $\delta(q,x)=f(x)$ and the output function is given by $\Lambda(q,x)=f(x)$.

An (r,d)-network is a finite collection of(r,d)-modules together with a rule for interconnection. Each module has each input line connected to either a module output line or an input line from outside the network. The connection can be thought of as made by a delayless wire. Those input lines that are not output lines from other modules are called network input lines. The output lines of some modules lead outside the network and are called network output lines. A typical diagram for a (3,d)-network is shown in Figure 2.3. Note that the output line of a module can be split and connected to several input lines or lead outside the network.

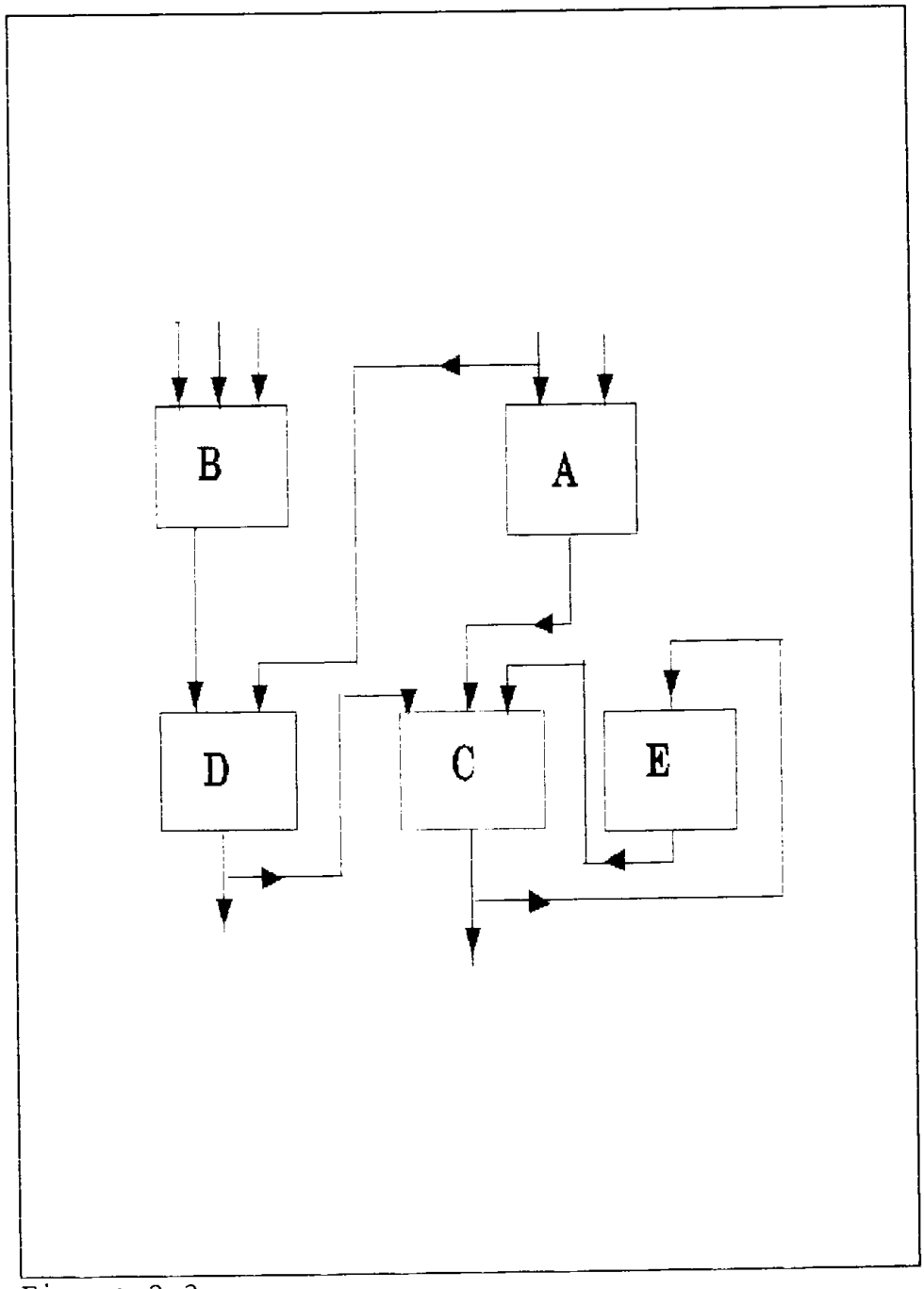

Figure 2.3

A connection such as that shown in Figure 2.4 is not allowed.

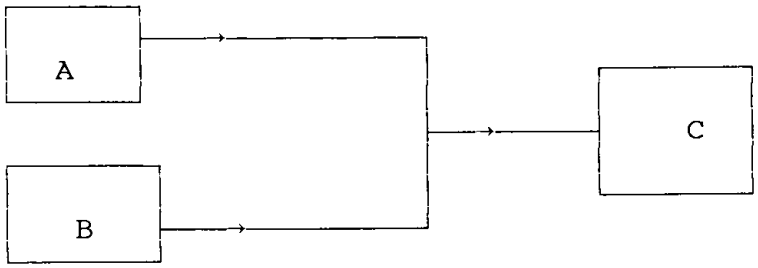

Figure 2.4.

One determines the state of a network by determining the state of each module in the network. A network with s input lines acts on each s-tuple of elements of A that is placed on the input lines of the network. The network starts with an initial state assigned to each of the modules of the network. One thinks of the actions of the network as occurring in discrete units of time. At a given time t-1 each module starts computing the value of its function using as inputs the values on its input lines at time t-1. The computation lasts one unit of time. At time t each module has completed its computation and that value computed becomes the state of the module at time t and the value is assigned to the module's output lines. The network is said to compute the function F in time τ from the initial state q_0 if for each sequence of values $(a_1,\ldots,a_s)$, $a_i \in A$, and assigned constantly to the input lines of the network for τ units of time,

then the value on the output lines of the network at time τ is the function value $F(a_1,\ldots,a_s)$. A given network can compute many different functions depending on the length of time the values on the input lines of the network remain unchanged. Consider, for example, the network represented by Figure 2.5.

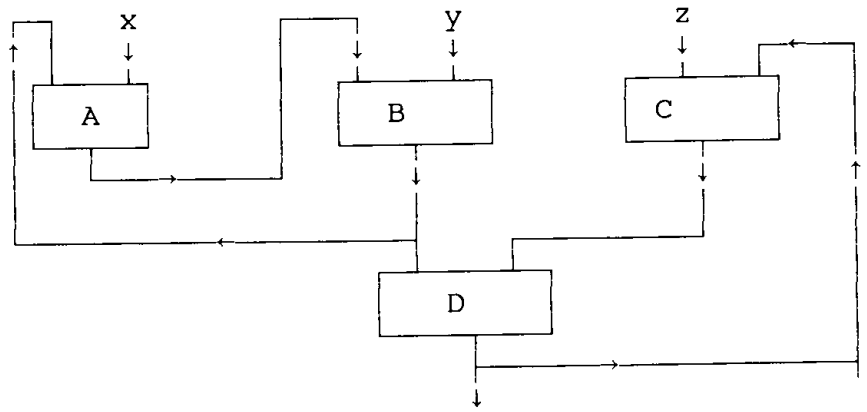

Figure 2.5

Assume that the alphabet used by the network is the set that consists of the symbols T(rue) and F(alse). Designate the input lines to each of the modules A, B, C, and D as Left or Right. We suppose that the module A computes (Left and Right), that B computes (Left or Right), C computes (Left implies Right), and D computes ((Left or Right)and not(Left and Right)); i.e. the module D computes "exclusive or". The output line of the module D is the single output line of the network. Table 2.1 catalogues the values on the input and output lines of the network and the states of the modules as a function of time τ during

which the network inputs x, y, and z on the network input lines remain constant. The last column lists the values on the network output line. We assume that the initial value assigned to each of the modules at time τ=0 is T. In the Table 2.1, denote (x or y) by xvy, (x and y) by x&y, (x implies y) by x->y, (exclusive or) by x$\oplus$y, and (not x) by ~x. In this example, each of the modules has two input lines and the network is a (2,2)-network. It is a matter of convenience that none of the modules have single input lines, in a (2,2)-network, each module must have at most 2 input lines, but a module can have less than two input lines. Table 2.1 shows the changes of state of each of the modules as a function of time τ. The first column indicates the time, τ. The columns x, y, z indicate that each of the input lines to the network, labelled x, y, and z, is to receive a constant value during the computation. The columns labelled A, B, C, and D indicate the states of the modules at the various times. For example, the entry in the row labelled τ=2 and in the column labelled B indicates that the state of the module B at time τ=2 is the value (xvy).

	x	y	z	A	B	C	D	Output
τ=0	x	y	z	T	T	T	T	T
τ=1	x	y	z	x	T	T	F	F
τ=2	x	y	z	x	xvy	~z	F	F
τ=3	x	y	z	x	xvy	~z	(xvy)⊕~z	(xvy)⊕~z
τ=4	x	y	z	x	xvy	z-> [(xvy)⊕0~z]	(xvy)⊕~z	(xvy)⊕~z

etc.

Table 2.1

If a network can compute a function F, then the time required for the computation is an indicator of the computational complexity of the function F in relation to the network. Fix positive integers r and d. We will treat as an indicator of the complexity of a function F with respect to (r,d)-networks, the minimum time required to compute F by where the minimum time is taken over all (r,d)-networks that can compute F. In the next chapter, we formalize this model in a way that allows us to replace the modules by functions on infinite alphabets. The motivation for this generalization is discussed in Chapter I.

We note two theorems, to be found in [3] that establish an equivalence between the modular network and the finite state sequential machine as models of computation.

Theorem 2.1 [3, p. 68] Any modular network is

describable as a state-output finite automaton(Moore machine).

The definitions of state-output finite automaton and a finite automaton(sequential machine) can be found in [3].

Theorem 2.2 [3, Theorem 7 p. 69] For every finite automaton M there is a modular network that simulates M.

Consider a function f: $\prod_1^n X_i$--->Y. Here, the sets X_i and Y are given finite sets, not to be considered as the input sets or the output sets of an automaton. In order to compute the function f using an (r,d)-network we must represent, or encode, the domain and range of the function f in terms of inputs and outputs of the network. An encoding of X_i is a map $g_i: X_i \rightarrow \prod_1^{m(i)} A$, where A is the alphabet accepted by the network. Similarly an encoding of Y, the range of f, is a one-to-one function $h=(h_1,\ldots,h_m): Y \rightarrow \prod_1^m A$. If there exist encodings $g_1,\ldots,g_n$, h and a network C that computes a function F such that

$$F(g_1(x_1),\ldots,g_n(x_n)) = h(f(x_1,\ldots,x_n))$$

we say that C computes an encoded version of the function f.

The time C required to compute the function F

depends on the modules available for use in the network and the on the number of input lines to the network. Arbib and Spira have obtained a lower bound on the number of input lines for encoding the sets X_i. For networks restrticted to consist of (r,d)-modules, for fixed r and d, this lower bound was used to obtain an lower bound on the time required to compute F.

The bound given in [3] is based on two observations. The first observation is exemplified in Figure 2.6. In Figure 2.6 we see that if in a (2,d)-network an output line depends on 8 input lines, then it must take at least 3 units of time for the network to produce its corresponding output.

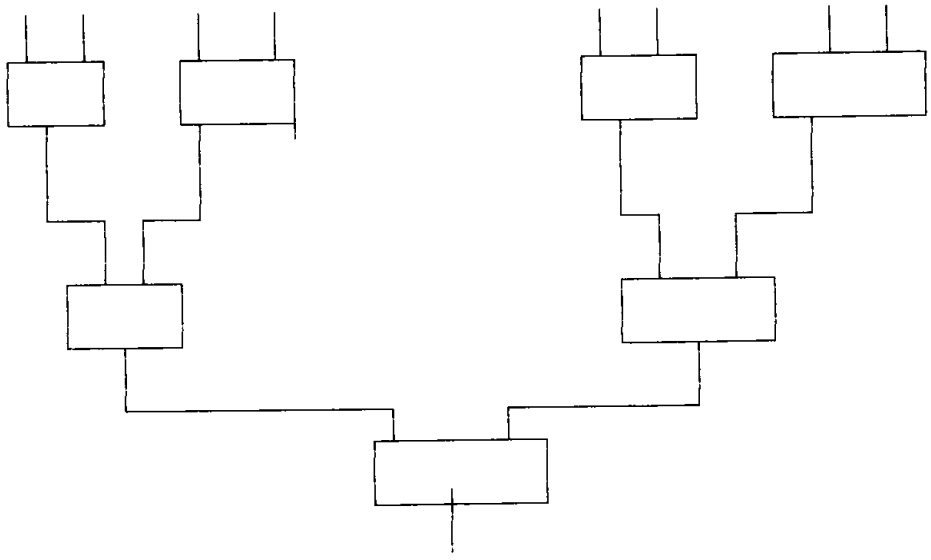

Figure 2.6

The second observation is that the time required

to compute an encoded version of a function $f:X_1 \times \ldots \times X_n \rightarrow Y$ depends on the number of copies of the alphabet A required to encode X_i. This in turn can depend on the encoding of Y.

The first observation is formalized by the following lemma.

Lemma 2.1. The output at time t on a given output line of an (r,d)-network can depend on the input values of at most r^t network input lines.

Let INT[x] be the smallest integer greater than or equal to x. Another way of stating Lemma 2.1 is to say that if the output of a network depends on n preceding values, then the time required to compute the output is at least $\text{INT}[\ \log_r(\ n\ )\ ]$.

For an (r,d)-network C that computes an encoded version of a function f, $h_j(\ y\ )$ is the value on the j^{th} output line when the output of the network is $h(\ y\ )=\prod_j h_j(\ y\ )$. A set $S \subseteq X_m$ is called an h_j-separator set for C in the m^{th} argument of f if for $s_1, s_2 \in S$ with $s_1 \neq s_2$ there exists $x_1,\ldots,x_{m-1},x_{m+1},\ldots,x_n$ with $x_i \in X_i$ such that

$$h_j[\ f(\ x_1,\ldots,x_{m-1},s_1,x_{m+1},\ldots,x_n\ )\ ] \neq$$
$$h_j[\ f(\ x_1,\ldots,x_{m-1},s_2,x_{m+1},\ldots,x_n\ )\ ].$$

Arbib and Spira noted that if S is an h_j-separator

set in X_m, then in order to compute an encoded version of f with Y encoded by h, the encoding g_m must be one-to-one on S. Let |S| denote the cardinality of S, and recall that |A|=d. Then at least Int[$\log_d S$] copies of A are required in the range of g_m. The formal statement is given in Theorem 2.2.

Theorem 2.2.(Arbib and Spira) Let $f:X_1 x...x X_n \rightarrow Y$. Let C be an (r,d)-network (r>1, d>1) that computes f in time t. Then

$t \geq$

$$\max_j[\log_r\{ \text{INT}[\log_d(|S_1(j)|)] +... + \text{INT}[\log_d(|S_n(j)|)] \}]$$

where $S_i(j)$ is an h_j-separator set for C in the i^{th} argument of f.

Chapter III

Graphs and Networks

In this chapter we present a version of the McCulloch and Pitts model for computing that allows continuous functions as modules. The general idea is to replace the finite alphabet used in the informal discussion of Chapter II by an open neighborhood of the origin in a Real vector space of finite dimension. The replacement is a fairly simple matter, but because we wish to relate this model to the representation of functions by superpositions, some of the details are notationally cumbersome (for a discussion of superpositions see [16]). We make no claim of originality. The move to (r,d)-networks using vector valued functions is certainly a natural one. We represent networks by directed graphs. The use of directed graphs is common in automata theory (c.f. [3], or [9]). However, we have been unable to find a reference that carries out the construction we use. We give an informal discussion of the model in this chapter and relegate the formalities to Appendix C.

In Chapter II there is an informal presentation of the development of the McCulloch and Pitts model ([3].) In that presentation, if A is a set of cardinality d,

then an (r,d)-module is a function $f:\prod_1^s A \rightarrow A$, $1 \leq s \leq r$, from the s-fold product of A to A. One can think of the function f as a computing device, or a finite state automaton. The alphabet for this computing device consists of the sequences $(a_1,\ldots,a_s)$, $a_i \in A$, and the output set of the device is the set A. This automaton has A as set of states. If the function is in state q at time t, and f accepts a sequence of elements $(a_1,\ldots,a_s)$ at time t it outputs $f(a_1,\ldots,a_s)$ one unit of time later. It is convenient to think of the module f as a pair $((1,\ldots,s),f)$ where,

(1) $(1,\ldots,s)$ is the sequence of indices for the variables of f, i.e. indices for the coordinates of the domain of f,

(2) f names the function,

(3) every function $f:\prod_1^n A \rightarrow A$, can be a module. Although a module is a pair $((1,\ldots,s),f)$ it is sometimes convenient to denote the pair by f.

An (r,d)-network is a finite collection of such (r,d)-modules 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 of interconnection is itself a function. The domain of the interconnection function is a subset of the collection of all pairs [j,f], where j is an index for 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 to that pair exactly one module, say ((1,...,s'),g). We think of the j^{th} variable of f as connected to the output set of the module ((1,...,s'),g) by a delayless wire, or line. When the module g computes a value, that value is instantly relayed to the j^{th} variable of the function 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 thought of 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 of the network. The output modules are where the results of a computation are read. Sometimes an output module is called an output vertex.

Because the interconnection rule is a function from the finite set of pairs [j,f], to the finite set of modules of the network, 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,...,s'),g) to the j^{th} variable of a module ((1,...,s),f), then the graph has an edge that starts at the vertex

g=((1,...,s'),g) and ends at vertex f=((1,..,s),f). That edge from g to f is then indexed by j, to show that the edge ends at the j^{th} variable of f.

In the model we consider, the alphabet A is replaced by a subset of a finite dimensional real vector space of dimension d. The modules of an (r,d)-network are function chosen from a restricted class $\mathfrak{F}$. The class $\mathfrak{F}$ is a primitive of the model. Examples in the case of (r,d)-networks would be the class of analytic functions of s-tuples of d-dimensional Real vectors,$1 \leq s \leq r$, and that have d-dimensional vector values,or, d-dimensional vector valued continuous functions that have as variables d-dimensional Real vectors.

The diagram in Figure 3.1 represents a (2,1)-network C. The class $\mathfrak{F}$ of functions used in the network consists of three functions of the two variables A and B. The class $\mathfrak{F}$={A+B, A*B, A/B}. Each vertex of the digraph that represents C is denoted by a box with a label that represents the function assigned to that vertex. Each edge of the digraph is labelled by a letter. We use the same labelling for a variable and for the edge that indicates the assignment made by the interconnection rule. The vertices labelled L_1 and L_2 are the input

vertices of the (2,1)-network, while the output vertex of the (2,1)-network is F_3.

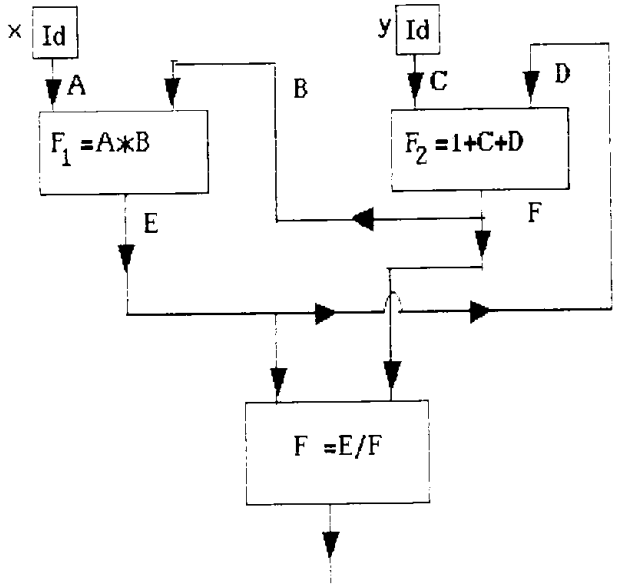

Figure 3.1

As in the discussion in Chapter II, the state of an (r,d)-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 and that is in a state σ', acts on each s-tuple that is placed on its 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. When the s-tuple of values placed on the input lines of the network is changed, we assume that the network returns to the fixed initial state σ from which state the new computation statrs. At the end of each interval of time, as long as the s-tuple placed on the input line remains unchanged, the values produced by the network at the network output modules are functions of the s-tuple of values placed on the network input lines.

As an example, we return to the diagram in Figure 3.1. Assume that in the initial state σ vertices L_1, L_2, F_1, F_2, and F_3 have values 0,0,0,1,0, respectively. We represent the initial state by the row matrix:

(σ)					
L_1	L_2	F_1	F_2	F_3
0	0	0	1	0

Table 3.1 shows the sequence of changes of state over time as the network C computes. Table 3.1 can be read as follows. The entry 0 in the column labelled F_1, and the row 0 represents the state of vertex F_1 at the initial state. The second row of the table indicates the new state of each of the modules of the network at time τ=0. At time τ=0 the input lines of the network are changed to the state in which the input vertex L_1 has state x and the input vertex L_2 has state y. During the period from τ=0 to τ=1 the state of C changes to a new state in which the state of F_1 is the value of F_1(A,B)=A*B, whe A has the value x and B has the value that is the initial state of the vertex F_2. Therefore, F_1 changes to the state x. At the time τ=4 which is the end of 4 units of time starting when the values x and y are placed on the network input lines, the line G carries the value $\frac{x(1+x+y)}{(1+x)(1+y)}$.

	L_1	L_2	F_1	F_2	F_3
σ	0	0	1	0	0
τ					
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))	$\frac{x(1+x+y)}{(1+x)(1+y)}$

Table 3.1

Just as in the discussion of Chapter II, the network is said to compute the function F in time τ from the initial state σ if for each sequence of values $(a_1,\ldots,a_s)$ chosen from A and assigned constantly to the input lines of the network for τ units of time then the value on the output lines of the network is the function value $F(a_1,\ldots,a_s)$ at time τ. The network given in Figure 3.1 computes the function

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

in time 4, but it computes the function $x(1+y)/(1+x+y)$ in time 3. A given 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.

Suppose that 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 C has one output line. For simplicity, we fix d to be 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 directed edges. By a tree we mean that even when the direction of edges is ignored, there are no loops. By a sequence of edges we suppose that the beginning point of an edge in the sequence is the endpoint of the preceding edge in the sequence. In the parlance of graph theory, each vertex of the tree can be connected to the root by a directed walk. Each sequence of edges required to connect a vertex to the root has as a length the number of edges in the sequence. The maximum of the lengths of sequences connecting vertices of the tree to the root is the length of the tree. A network that is represented by such a tree computes functions that are superpositions of the 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 same tree. Furthermore, when the time is larger than the length of the tree, the network continues to compute the same value of the function, and the depth of

superposition is the length of the tree. A classical problem in mathematics is to decide whether a given function F of n variables can be written as the superposition of functions of a fewer number of variables. This is the substance of Hilbert's Thirteenth Problem(c.f. [8,16,17]). It is implicit in the literature that the depth of the superposition expresses an intuitive notion of computing complexity. When the functions in the superposition are only restricted to be continuous and have fewer variables than F, Arnold and Kolomogorov have shown (c.f. [16,17]) that F can always be written as a superposition of continuous functions of fewer variables. However, if the functions used in the superposition are also required to have the same degree of smoothness as the function F, then it is known that in general such a superposition representation is impossible(c.f. [16,17,27]).

In the case that a function is computed in time τ by an (r,d)-network C that is not represented by a tree, one can replace the network C by a network C′ that is represented by a tree T of length τ. The network C′ can be constructed out of the same modules (but with an increase in the number of modules) as the network C with the possible addition of identity functions as modules. This remark is the substance of

Corollary C.1. The replacement procedure deloops C.

The (2,1)-network shown in Figure 3.1 has a digraph that is not a tree. As we have seen, this (2,1)-network computes the function

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

in four units of time. We use the procedure of Appendix C, Corollary C.1 to construct a (2,1)-network T that has a tree as digraph, that computes the function h(x,y) in time four, and has as the functions associated to the vertices of T the same functions associated to the vertices of C.

At time t=4 the output vertex F_3 has as its state the value x(1+x+y)/(1+x)(1+y).

At time t=3 the edges E and F correspond to variables that have values x(1+x+y) and (1+y)(1+x) respectively. We construct two (2,1)-networks C(1) and C(2), each with a digraph that is a tree, so that in time t=3 the network C(1) computes x(1+x+y) and C(2) computes (1+x)(1+y). Assign the output of C(1) to E and the output of C(2) to F with the output vertex of the tree assigned the function E/F. At time t=3 the variable E (that is the function F_1) has the value x(1+x+y) and the variable F (i.e., the function F_2) has the value (1+x)(1+y). The function F_1 has value x(1+x+y) at time t=3, because at time t=2 variable A carries the value x and variable B (assigned to the

function F_2) has value (1+x+y). Next we need two (2,1)-networks C(1,1) and C(1,2) that in time t=2 compute the values x and (1+x+y), respectively. Similarly we require two (2,1)-networks C(2,1) and C(2,2) that compute y and x(1+y), in time t=2. Connect C(1,1) and C(1,2) to A and B (respectively) of F_1 and connect C(2,1), C(2,2) to C and D, respectively. This produces the following diagram.

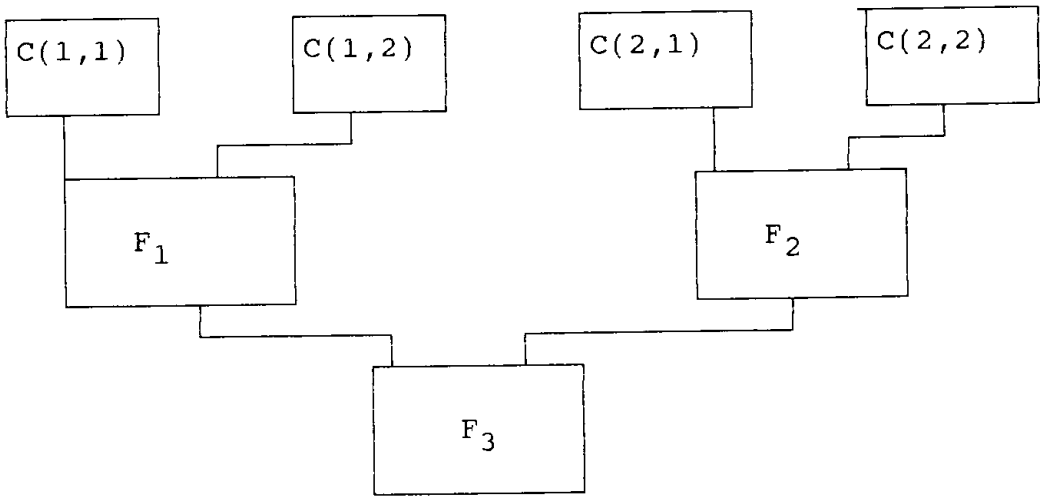

Figure 3.2

At time t=2, variable A carries x and variable B (function F_2) carries 1+x+y while variable D carries y and variable E (function F_1) carries x(1+y). Extend Figure 3.2 to Figure 3.3 where H and K are functions that are to produce x and y respectively in time t=2.

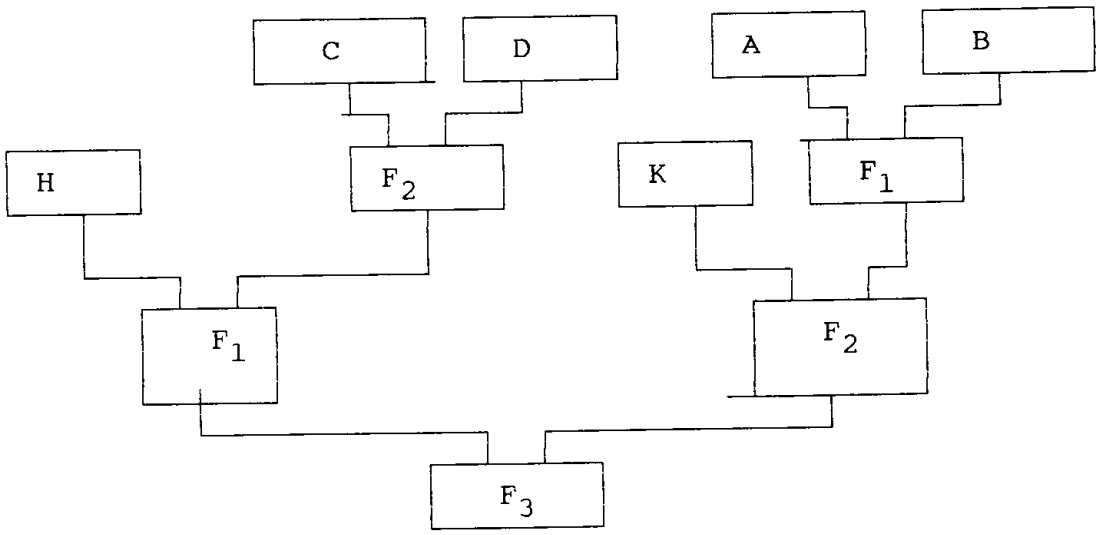

Figure 3.2

At time t=1, A and B are to have values x and 1+y, while C and D are to have the values y and x respectively. Extend Figure 3.2 to a Figure 3.3 as follows.

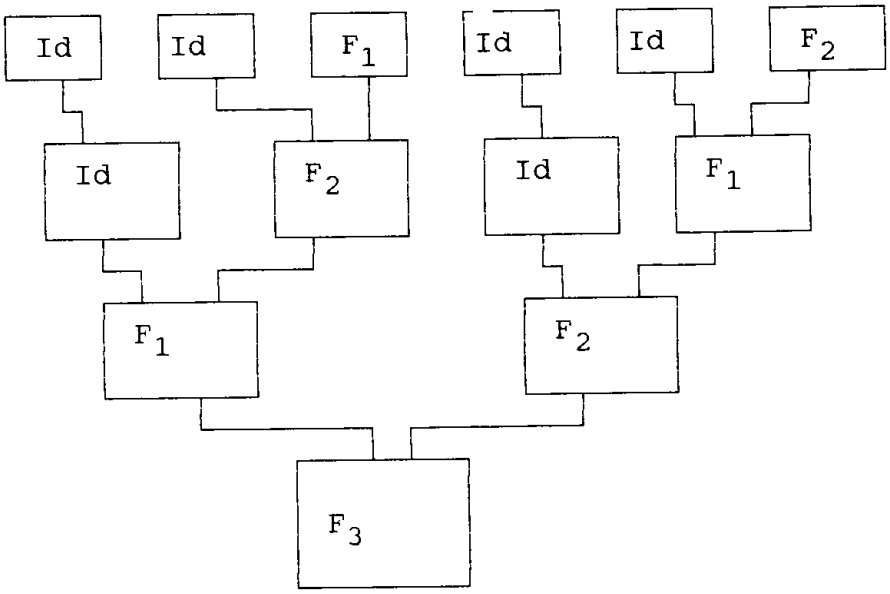

Figure 3.3

Finally, we may read off the required inputs from σ and the input vertices of C. The result is

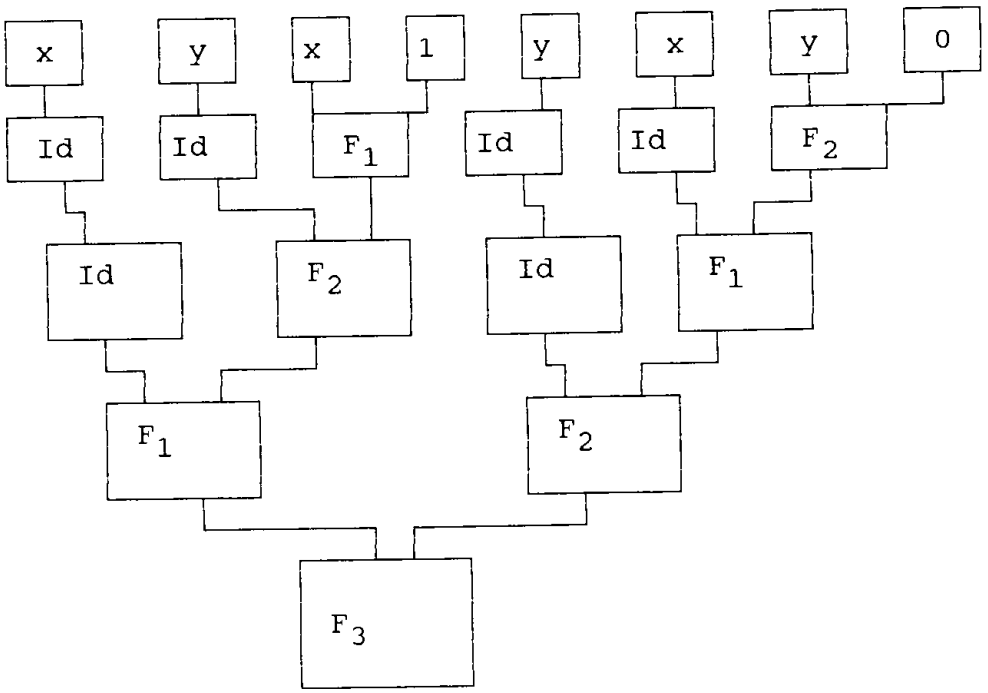

Figure 3.4

The function computed in time 4 by the (2,1)-network in Figure 3.4 has the following expression as a superposition,

$$F_3(F_1(x,F_2(y,F_1(x,1))),\ F_2(y,F_1(x,F_2(y,0)))) =$$

$$\frac{F_1(x,F_2(y,F_1(x,1)))}{F_2(y,F_1(x,F_2(y,0)))} = \frac{xF_2(y,F_1(x,1))}{1+y+F_1(x,F_2(y,0))} =$$

$$\frac{x(1+y+F_1(x,1))}{1+x+x*F_2(y,0)} = \frac{x(1+y+x)}{1+y+x(1+y)} = \frac{x(1+x+y)}{(1+x)(1+y)} .$$

Table 3.1 shows the changes of state of the(2,1)-network of Figure 3.1 that correspond to the successive intervals of time.

Chapter IV

Computing Encoded Functions and the Dimension Based Lower Bound on Computing Time

In the finite McCulloch and Pitts model of computing, a lower bound for the time needed to compute a given finite function by an (r,d)-network is given by the formula of Arbib and Spira [3]. This bound depends on the cardinality of certain separator sets associated with the function. There is an analogous concept of separator set for continuous functions. In this chapter we give a lower bound for the time needed to compute a given continuous function by continuous (r,d)-networks. This bound is given by an expression much like that of the Arbib and Spira lower bound, except that the size of separator sets is measured by (topological)dimension instead of cardinality. In Section I we discuss an equivalence relation, called F-equivalence, that a function defined on a product $X_1x...xX_n$ induces on each of the components X_i. The separator sets of Arbib and Spira are convenient representations of this equivalence relation. In Section II we introduce the concept of separator set for continuous functions and establish the relation between separator set and F-equivalence. The section

ends with the statement and proof of the Dimension Based Lower Bound on computing time. In Section III we use the results of section II to analyze the computation of linear functions.

Section I.

F-Equivalence

Suppose that F:X--->Y is a non-constant function, from a finite set X to a finite set Y. The function F defines an equivalence relation on X where points x and x′ in X are equivalent if F(x)=F(x′). The quotient set <u>(X/F)</u> is the collection of equivalence classes of that equivalence relation. Denote by q:X--->(X/F) the function that carries a point $x \in X$ into the equivalence class of x. The set (X/F) is in one-to-one correspondence with the image of F. One can factor F through the set (X/F); that is, there is a (unique) function F*:(X/F)--->Y such that F=F*·q. Furthermore, (X/F) is the smallest set in cardinality through which F can be factored.

Suppose that an (r,d)-network C is to compute an encoded version of F(c.f. Chapter II) and assume that C uses an alphabet A (of cardinality |A|=d) that is at least as large as the cardinality of the image of F. One can encode the image of F, or rather encode the quotient (X/F), by a map g:(X/F)---> A.

Encode the image of F by the same map g, and encode the domain of F by g•q. The network C computes an encoded version of F in one unit of time by setting x=x. The quotient map q and the inverse of g carry the burden of the computation. No network can decrease the computing time. Further, each network that computes F must have as input set a set of cardinality at least |A|, because the network C computes a function that has as image a set that contains a copy of the image of F. Each copy of A can be considered as the domain of a single variable. We conclude that in the case of a function with values in A, only one variable is required to compute the function.

Now consider the case of a function $G:X_1xX_2--->A$ defined on a product X_1xX_2 with values in A where each X_i is a nonempty finite set. Assume that a network C′ is to compute an encoded version of G. We attempt to follow the pattern established for the function F defined on the space X. The problem is complicated by the definition of computing an encoded version of G, because in the definition given in Chapter II, each of the spaces X_i is to be encoded separately by a function g_i. The set A is certainly the choice for the encoding of the range of G. We require two functions $g_i:X_i--->P_i$, each P_i a product of copies of A, and a function g: $P_1xP_2--->A$ that C′ is to compute, such that

for each $x_1 \in X_1$ and $x_2 \in X_2$,

$$G(x_1,x_2)=g(g_1(x_1),g_2(x_2)).$$

Each function g_i determines an equivalence relation $\equiv_i$ on X_i. For $x, y \in X_i$, $x \equiv_i y$ if $g_i(x)=g_i(y)$. Note also that if $x \equiv_1 y$ in X_1, then for each $z \in X_2$,

$$G(x,z)=g(g_1(x),g_2 (z))=$$

$$g(g_1(y), g_2(z))=G(y,z).$$

A similar remark applies to g_2 and X_2. The coarsest relation, that is a relation with largest equivalence classes, we can apply to X_i that yields an encoding $g_i:X_i \text{--->} P_i$ is given as follows. Define elements x and y in X_1 to be G equivalent if $G(x,z)=G(y,z)$ for all z in X_2. Similarly we say that for z and w in X_2, z and w are G equivalent if $G(a,z)=G(a,w)$ for all $a \in X_1$. To see that this is the coarsest relation that we can use to give encodings of the X_i, note that we have already seen that each pair of maps $g_i:X_i \text{--->} P_i$ used as encodings of the domain of G to compute an encoded version of G determine equivalence relations on the X_i in which points that are G equivalent are equivalent. Denote by (X_i/G) the quotient set of X_i determined by G equivalence on X_i. We can conclude, that the set of smallest cardinality we can use for the set P_i is a product of copies of A just large enough to contain a one-to-one copy of (X_i/G). The number of copies of A required for an embedding of (X_i/G) is

$\text{INT}[\ \log_d(\ |(X_i/G)|\ )\ ]$. If we think of each copy of A as the domain of a variable to be used by C′ to compute an encoded version of G, then we need at least $\text{INT}[\ \log_d(\ |(X_i/G)|\ )\ ]$ to encode each X_i.

Denote by q_i the quotient map from X_i to (X_i/G). For Arbib and Spira, a set S in X_i is a separator set for G if q_i is one-to-one on S. The use of the alphabet A to encode the sets X_i and Y in cases in which the cardinality of A is smaller than the cardinality of Y is accomplished by using sufficiently large products of copies of A.

We now carry out the construction of the sets (X_i/F) to be used in the case of a function defined on a product $\prod_1^n X_i$ and investigate some of the formal properties of the quotient. In particular we discuss in what sense the (X_i/F) are the smallest sets that must be embedded in a product of copies of A in order to compute an encoded version of F.

Notation. If X_j, $1\le j\le n$, are sets, then $\underline{X}_{<-j>}$ denotes the set

$$X_1 \times \ldots \times X_{j-1} \times X_{j+1} \times \ldots \times X_n.$$

If $x\in X_j$ and if $z=(z_1,\ldots,z_{j-1}, z_{j+1},\ldots,z_n)\in X_{<-j>}$, then $\underline{x}\int_j\underline{z}$ denotes the element $(z_1,\ldots,z_{j-1},x,z_{j+1},\ldots z_n)$ of $X_1\times\ldots\times X_n$.

Definition 4.1: Suppose that X_i, $1\le i\le n$, and Y are

sets, suppose that $F:\prod_1^n X_i \text{--->} Y$ is a function, and suppose that $1 \le j \le n$. Two points x and x′ in X_j are <u>F-equivalent</u> in X if for each $z \in X_{<-j>}$,

$$F(x \int_j z) = F(x' \int_j z).$$

It is elementary that F-equivalence in X_j is an equivalence relation on points of X_j.

Denote by $\underline{(X_j/F)}$ the collection of F-equivalence classes of X_j . Set q_j equal to the quotient map from X_j to (X_j/F).

The following lemma establishes the sense in which the set $(X_1/F)x...x(X_n/F)$ is the smallest product set through which F factors.

<u>Lemma 4.1</u>: Suppose that $X_1,...,X_n$, and Y are sets and suppose that $F:X_1x...xX_n \text{--->} Y$ is a function. There is a unique function $F^*:(X_1/F)x...x(X_n/F) \text{——>} Y$ that makes the Diagram 4.2 commute. Furthermore, if $Z_1,..,Z_n$ are sets, and if there are functions

$$g_i:X_i \text{——>} Z_i,\ 1 \le i \le n,$$

and a function

$$G:Z_1x...xZ_n \text{———>} Y$$

that makes Diagram 4.3 commute, then there are uniquely determined maps $g^*_1,...,g^*_n$, $g^*_i:Z_i \text{——>} (X_i/F)$, that make Diagram 4.4 commute.

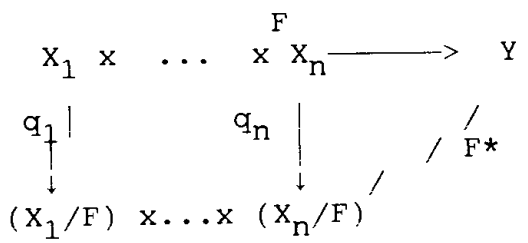

Diagram 4.2

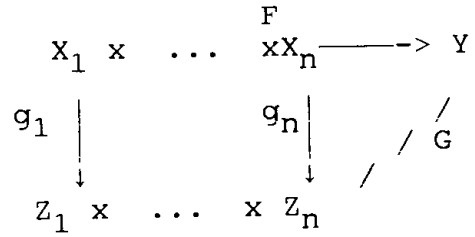

Diagram 4.3

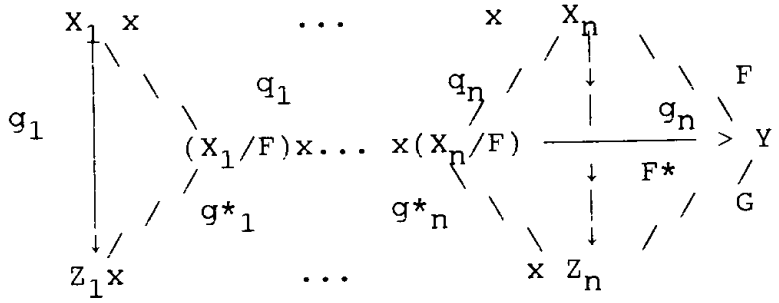

Diagram 4.4

Proof. If $z \in Z_i$, choose $x, x' \in X_i$ such that $g_i(x') = g_i(x) = z$. If $w \in X_{<-i>}$, set $g(w) = (g_1(w_1), \ldots, g_{i-1}(w_{i-1}), g_{i+1}(w_{i+1}), \ldots, g_n(w_n))$. Then

$$F(x \int_i w) = G(g_i(x) \int_i g(w)) =$$

$$G(g_i(x') \int_i g(w)) = F(x' \int_i w).$$

Therefore $q_i(x) = q_i(x')$. Set $g^*_i(z) = g_i(x)$. It is clear that Diagram 4.4 commutes.

As for uniqueness of the maps g^*_i, note that if $g^{**}_i : Z_i \longrightarrow (X_i/F)$, $1 \le i \le n$, are maps that make Diagram 4.4 commute when used in place of the maps g^*_i, then for each $z \in Z_i$ and each $x \in X_i$ so that $g_i(x) = z$, it follows that

$$g^*_i(z) = g^*_i(g_i(x)) = q_i(x) = g^{**}_i(g_i(x)) = g^{**}_i(z). \blacksquare$$

Section II

The Computation of an Encoded Version of a Function F and the Dimension Based Lower Bound on Computing Time

Arbib and Spira studied, in effect, the spaces (X_i/F). They did so by investigating subsets of X_i, called separator sets, on which the maps q_i are one-to-one. When the spaces X_i are topological spaces and the maps involved are continuous, working with such subsets has some advantages over working directly with the quotient space, because the quotient maps q_i can be badly behaved. This leads to the following definition.

Definition 4.1: Suppose that $X_1,\ldots,X_n$, Y are sets and suppose that $F:X_1x\ldots xX_n \longrightarrow Y$ is a function. A subset S of the set X_i is said to be an i-separator set for the function F if for each x, x′, $x \neq x'$, in S there is $z \in X_{<-i>}$ such that

$$F(x\int_i z) \neq F(x'\int_i z).$$

If x and x′ are elements of X_i that form an i-separator set for the function F, then x and x′ are said to be separated by F.

It is easy to see that q_i is one-to-one on each i-separator set S. In the situations represented by Diagram 4.3, a more general assertion about

separator sets is true.

Assume that S is an i-separator set for F and suppose that there is a function

$$G: Z_1 \times \ldots \times Z_n \rightarrow Y$$

and for each $1 \leq j \leq n$, there is a function

$$g_j: X_j \rightarrow Z_j$$

that makes Diagram 4.3 commute. If s and s' are elements of S that satisfy the equality $g_i(s)=g_i(s')$, then for each $w \in X_{<-i>}$,

$$F(s \int_i w) = G(g_i(s) \int_i (g_1(w_1), \ldots, g_n(w_n))) =$$

$$G(g_i(s') \int_i (g_1(w_1), \ldots, g_n(w_n))) =$$

$$F(s' \int_i w).$$

Therefore, $q_i(s)=q_i(s')$. Because the map q_i is one-to-one on S, s=s'. It follows that the maps g_i are one-to one on i-separator sets. Furthermore, the set $g_i(S)$ is an i-separator set for G, because if $F(s \int_i w) \neq F(s' \int_i w)$, then $g_i(s) \neq g_i(s')$. The property of being an i-separator set is inherited by the image under g_i of an i-separator set. Furthermore, the image $g_i(S)$ has the same cardinality as the original set S.

The result of Arbib and Spira relating the cardinality of separator sets and the minimum time required to compute an encoded version of a function was stated in Chapter II. There is an analogous lower bound result for computing an encoded version of a

continuous function. We begin the discussion of that analogous result with the following definition.

Definition 4.3: Suppose that $F:X_1x...xX_n--->Y$ is a continuous function from a product of topological spaces X_i to a topological space Y. Suppose $V=Rx...xR$ is a d-fold direct product of the Real numbers. We say that an (r,d)-network 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 $g_i:X_i--->E_i$, $1\leq i\leq n$,

and

(b) there are continuous functions $h_1,...,h_b$ where

$h_j:Y\text{———}>V$,

such that the following conditions are satisfied:

(i) $h=(h_1,...,h_b)$ is a bi-continuous one-to-one map to a topological subspace of Vx...xV (h is an embedding),

(ii) there is a function $D=(D_1,...,D_b)$: $\prod_1{}^n E_i$ $--->Vx...xV$ from (Σw_i)-fold tuples of d-vectors to b-fold tuples of d-vectors that

C computes in time t,

(iii) the following diagram

commutes,

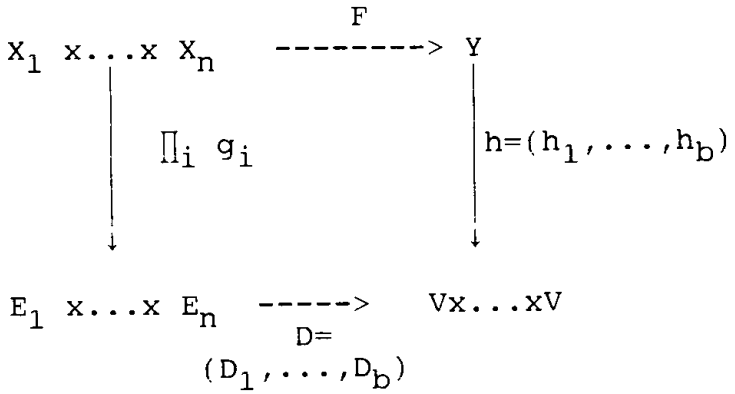

Diagram 4.5

The maps g_i encode the domain of F and the map h encodes the range of F. If v is an input vertex of C with domain vectors from E_i, we call v an input vertex for X_i.

Definition 4.4: Suppose that C is an (r,d)-network represented by a digraph G. A vertex v is connected to a vertex w if there is a walk from v to w in G. If C computes an encoded version of a function F and W is a walk from an input vertex v whose states are the inputs from a component of E_j, then W is a line from E_j.

Note: Each vector from E_j must be a possible

input to C through an input vertex. It follows that if $e_1,\ldots,e_p$ are the inputs from E_j, then dim $E_j \leq dp$.

Definition 4.5: Suppose that $F:X_1x\ldots xX_n \rightarrow Y$ is a continuous function, and as in Definition 4.4, suppose that C is an (r,d)-network that computes an encoded version of F with

$$h=(h_1,\ldots,h_w): Y\rightarrow\prod_i V$$

the encoding of the space Y. Let G be the digraph associated with C, and let the output vertices of G be v_j, $1\leq j\leq w$. For each fixed j, a subset $S \leq X_i$ is said to be an i-separator set for the j^{th} output vertex if S is an i-separator set for the function $h_j\cdot F$.
If S is locally Euclidean, then S will be called an LE-i-separator set for the j^{th} output vertex.

We can now state the lower bound assertion.

Theorem 4.1(Dimension Based Lower Bound): Let $F:X_1x\ldots xX_n\rightarrow Y$ be a continuous function. Assume that V is a d-fold direct product Rx...xR. Let C be an (r,d)-network that computes an encoded version D of F in time t where

$$g=\prod_i g_i:X_1x\ldots xX_n\rightarrow\prod_i E_i$$

is the encoding of the domain of F and

$$h=(h_1,\ldots,h_w):Y\rightarrow\prod_i{}^w V$$

is the encoding of the range of F. Suppose further that for each $1 \le i \le n$, a set $S(i;j) \le X_i$ is an LE-i-separator set for the j^{th} output vertex. Then $t \ge$

$\max_j \{ \text{INT}[\log_r \{ \text{INT}[\dim S(1;j)/d] + \ldots + \text{INT}[\dim S(n;j)/d] \}] \}$.

Proof: We first show that the j^{th} output vertex (which corresponds to h_j) must be connected to at least INT[dim S(i;j)/d] input vertices that have inputs that are vectors in the image of g_i. Suppose not, and suppose that the j^{th} line is connected to only q inputs from E_i where

$q <$ INT[dim S(i;j)/d].

Assume that E_i is the sum of q input spaces V. Then, as we have remarked in the note following Definition 4.4,

$\dim E_i \le q.d <$ INT[dim S(i;j)/d].d.

Set s = dim S(i;j) and write

$s = ad + m$, $0 \le m < d$. Thus

$$\text{INT}[\dim S(i;j)/d] = \begin{cases} a \text{ if } m = 0 \\ a+1 \text{ if } m > 0. \end{cases}$$

Since $q <$ INT[dim S(i;j)/d], there are two cases:

Case (i)

$m = 0$; then $q = a - P$ with $P \ge 1$.

In this case

$dq = da - dP < da = s$.

Case (ii)

$m > 0$; then $q = a - P + 1$, $P > 1$.

In this case

$dq = da + d - dP = (da + m) + (d-m) - dP = s + d(1-P) - m < s$.

In either case dim $E_i < s$.

The map $g_i: X_i \rightarrow E_i$ carries $S(i;j)$ into E_i. The space $S(i;j)$ is locally Euclidean and hence $S(i;j)$ is locally compact. Because $S(i;j)$ has dimension s, there exists a compact set K with a nonempty open interior U contained in $S(i;j)$ that has dimension s. The map g_i is continuous on X_i. If g_i is one-to-one on $S(i;j)$, then g_i is one-to-one on K, and, because K is compact, the restriction of g_i to K is a homeomorphism of the interior of K into E_i. Thus dim $E_i < s$ = dim U. This is impossible (See [10], p. 26). Therefore, the restriction of g_i to $S(i;j)$ cannot be one-to-one. Because the restriction of g_i to $S(i;j)$ is not one-to-one, there are two points s and s′ in $S(i;j)$ such that $s \neq s'$ and $g_i(s) = g_i(s')$. Because $S(i;j)$ was assumed to be an i-separator set for h_j there is an element x in $X_{<-i>}$ such that

$h_j[\, F(x_1,\ldots,x_{i-1},s,x_{i+1},\ldots,x_n)\,] \neq$

$h_j[\, F(x_1,\ldots,x_{i-1},s',x_{i+1},\ldots,x_n)\,]$.

By hypothesis C computes the encoded version D of F in time t. Therefore,at time t, $D \cdot \prod_i g_i = h \cdot F$. Then at

time t

$$D[\ \prod_i g_i(\ x_1,\ldots,x_{i-1},s,x_{i+1},\ldots,x_n\ )\ ] \neq$$

$$D[\ \prod_i g_i(\ x_1,\ldots,x_{i-1},s',x_{i+1},\ldots,x_n\ )\ ].$$

This is clearly impossible. Therefore the assumption $q < \text{INT}[\ \dim S(i;j)/d\ ]$ contradicts the assumption that $S(i;j)$ is a separator set for the j^{th} output vertex. Hence the j^{th} output vertex is connected to at least $\text{INT}[\ \dim S(i;j)/d\ ]$ outputs from E_i.

It follows that the j^{th} output vertex is connected to at least

$$\text{INT}[\ \dim S(1;j)/d\ ] +\ldots+ \text{INT}[\ \dim S(n;j)/d\ ]$$

vertices.

If we prove that when D is computed in time t the number of lines to the j^{th} output vertex is at most r^t, it will follow that

$$r^t \geq \text{INT}[\ \dim S(1;j)/d\ ] +\ldots+ \text{INT}[\ \dim S(n;j)/d\ ]$$

and therefore

$$t \geq \log_r(\ \text{INT}[\ \dim S(i;j)/d\ ] +\ldots+$$

$$\text{INT}[\ \dim S(n;j)/d\ ]\ ).$$

We proceed by induction on t.

First suppose that t=1. In this case a line from E_i to V can pass through at most one vertex that is not an output vertex, because if it passed through more than one vertex, the delay would be at least 2. Furthermore there must be at least one vertex in the network, since the time is t=1. Because each vertex in

an (r,d)-network can have at most r input lines the number of input vertices whose inputs consist of variables from E_i and that are connected to a single vertex is at most r. Thus

$r=r^t$ < (the number of inputs from E_i).

Assume that the delay is t, and assume that for each function that is computed in time t-1, the number of input vertices connected to the j^{th} output vertex is at most $r^{(t-1)}$. Consider the j^{th} output vertex of C. There are at most r edges with that vertex as endpoint. Consider an initial vertex v of an arbitrary such edge. For each i, the lines from E_i to v can have length at most t-1, otherwise the output at the j^{th} output vertex would take longer than t units of time. By the inductive assumption, the number of lines to v is at most r^{t-1}. Since there are at most r ways of choosing v, the number of lines to the j^{th} output vertex is at most $r.r^{(t-1)} = r^t$. ▒

The Dimension Based Lower Bound also can be interpreted in terms of the sets $(X_i/\ h_j \cdot F)$, when the quotient map q_i from X_i to $(X_i/\ h_j \cdot F)$ is sufficiently well behaved. Suppose that $F:X_1x...xX_n--->Y$ is a continuous function as in the statement of Theorem 4.1. Assume, as in the statement of the theorem, that S(i;j) is an LE-i-separator set

for the j^{th} output vertex of an (r,d)-network that computes an encoded version of F. If the set $(X_i/h_j \cdot F)$ is given the quotient topology, then the map $q_i: X_i \dashrightarrow (X_i/h_j \cdot F)$ is continuous. Suppose that the space $(X_i/ h_j \cdot F)$ is Hausdorff. Choose a point $s \in S(i;j)$. Because S(i;j) is a locally Euclidean space, there is a compact neighborhood U of s. The continuous map q_i carries U to a compact subset V of $(X_i/ h_j \cdot F)$. The function q_i is one-to-one on U because S(i;j) is a separator set for $h_j \cdot F$ (c.f. Definition 4.2). Because V is a subspace of a Hausdorff space, it is Hausdorff. Therefore, the restriction of q_i to U is a one-to-one and continuous map from a compact space to a Hausdorff space. It is, therefore, a homeomorphism. It follows that the map q_i has a continuous inverse $Q_i: V \dashrightarrow U$. The neighborhood U of the point s can be considered as the image of a local thread[5] Q_i of $h_j \cdot F$ defined on a subset of $(X_i/ h_j \cdot F)$. In particular, if the sets $(X_i/ h_j \cdot F)$ are themselves locally Euclidean, then the dimension of the separator set S(i;j) is bounded above by the dimension of $(X_i/h_j \cdot F)$.

---

[5] See [21] for the concept of a locally threaded function.

Section III

Computation of Linear Functions

Suppose $F= F^1x...x F^k$ is a product of maps $F^i:E^i---> Y^i$. Thus, $F:E^1 x...x E^n--->Y^1x...x Y^n=Y$. One can pose the question of computing an encoded version of F in at least two different ways. Each of these ways addresses a different encoding of the range space of F. In one case, we can consider F to be a product. In that case one can allow coordinate changes in the E^i and in the Y^i separately, but coordinate changes in Y that are not products of coordinate changes in the components Y^i are not allowed. In the second case, one can ignore the product structure on F and allow coordinate changes in Y that ignore the product structure on Y. Therefore, automorphisms of the range of F can be used to shorten the computation time. In order to clarify these distinctions we consider linear (homogeneous) functions between vector spaces that consist of n-tuples of Real numbers. We also restrict the modules to be linear homogeneous functions. The set R^n can be given many different vector space structures which depend on the definition of the addition given between n-tuples. Perhaps the

most familiar is the one in which n-tuples are added by adding the component Real numbers. In order to specify that the component addition is the one giving the vector space structure, we use the notation $R \oplus \ldots \oplus R$ (n-times) for R^n. The vector space $R \oplus \ldots \oplus R$ (n-times) has a basis that consists of the elements $(0,\ldots,0,1,0,\ldots,0)$ (where the 1 is in the i^{th} position), called the standard basis.
Consider the following example.

Example 1. F: $X_1 \times X_2$ ---> E where $X_1 = R \times R \times R$, $X_2 = R \times R \times R$, and E is a 4-dimensional Euclidean space. Assume that X_1 has the basis e_1, e_2, e_3, X_2 has the basis f_1, f_2, f_3, and E has the standard basis. Assume that F has, with respect to these bases, the matrix

1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1
1	2	3	1	5	6

In terms of coordinates x,y,z in X_1 and u,v,w in X_2 , $F=(F_1, F_2, F_3, F_4)^T$, where the superscript T, denotes transpose and

$F_1 = x + u,$

$F_2 = y + v,$

$F_3 = z + w,$

$F_4 = x + 2y + 3z + u + 5v + 6w.$

If $F_1,\ldots,F_4$ are all to be computed by a network that accepts 3 coordinates from X_1 and 3 coordinates from X_2, then with the given bases it is clear that F_4 can be computed in time 3 by the network represented by Diagram 4.6 that has input vertices x, y, z, u, v, and w .

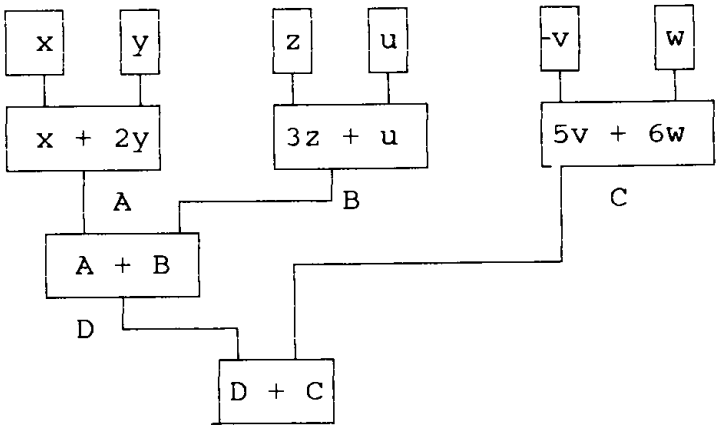

Diagram 4.6

It is easy to see that no network of delay two can carry out this computation. Because we can replace a network that computes a function in time t by a network without loops that computes the same function in time t, we need only consider networks of delay two that have the form of Diagram 4.7.

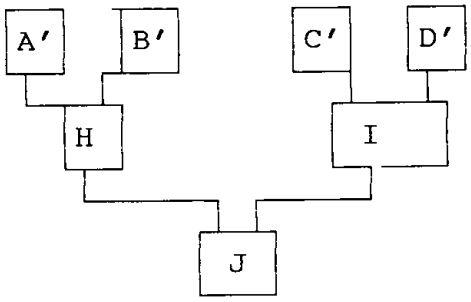

Diagram 4.7

Because J is linear, $J = aH + bI$ for a and b real numbers. Since H and J are each functions of two variables, the value of J can only depend on 4 variables. Each of $F_1,\ldots,F_4$ can be computed in time 1

since each is a function of two variables. If changes of basis are allowed separately in X_1 and X_2, then for change of basis matrices M and N (each a 3 x 3 invertible matrix), the matrix of F in the new bases is

$$\begin{matrix} M & & N \\ (1,2,3)M & & (1,5,6)N \end{matrix}$$

where (1,2,3,1,5,6) is the last row of the matrix of F. With a suitable choice of M and N it is possible to produce the matrix M(F) as the matrix of the function F.

$$M(F)= \begin{matrix} 1 & -2 & 0 & 1 & -5 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 3 & 1 & 0 & 6 \end{matrix}$$

Once the coordinate changes are made to produce the matrix M(F), then all of the functions F_i can be computed in time 2 by the network in Diagram 4.8. The network in Diagram 4.8 has input vertices along the top row labelled x', y', z', u', v', and w'.

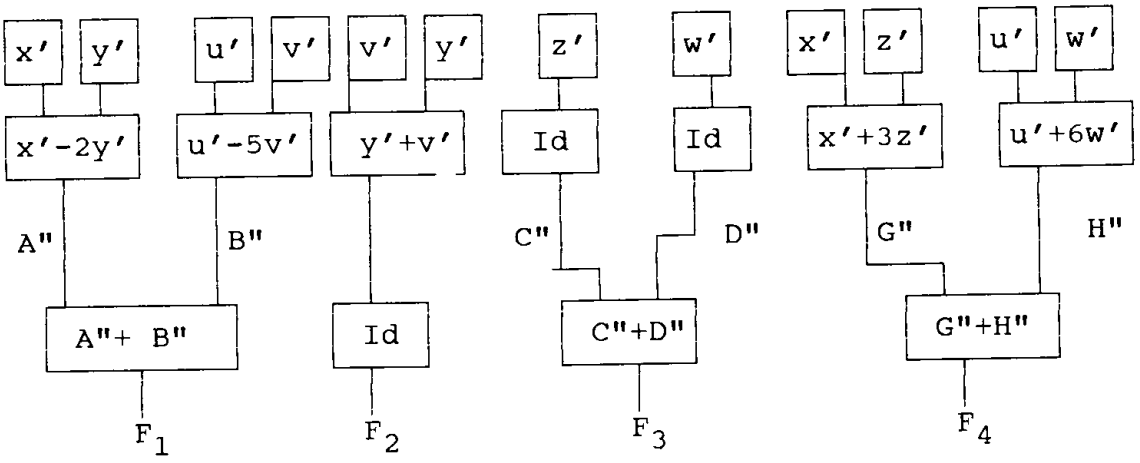

Diagram 4.8

Somewhat less obvious is the fact that no coordinate change in X_1 and X_2 (again with matrices P and Q, respectively) can decrease the computing time below 2. In order that a computation of $F_1,\ldots,F_4$ require computing time less than 2, each row of the matrix representation of F can have at most two nonzero entries (because the network consists of linear (2,1)-modules). Because each of the functions F_i involves variables from X_1 and X_2, it follows that, to within a permutation of columns, the matrix of P must be

$$\begin{matrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{matrix}$$

while the matrix for Q must also be diagonal with the diagonal entries e,f,g. But then (1,0,3)P = (a,0,3c) and (1,0,6)Q = (e,0,6g). Thus F_4 is computable in no less than 2 units of time.

If we now allow coordinate changes in the Euclidean 4-space E, row operations in the matrices for F are permitted. Starting again with the matrix

1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1
1	2	3	1	5	6

we may change this to

1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	0	3	3

(subtract row 1 from row 4, subtract twice row 2 from row 4, etc.). If we multiply the last row by 1/3 this becomes

1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	0	1	1

The resulting set of functions can be computed in one unit of time, because the computation of each function requires a single addition.

We have, of course, not computed the original functions $F_1,\ldots,F_4$. We can recover them, but only at the cost of more computing.

Suppose the encoding functions $g_i:X_i\text{--->}E_i$ can map X_i into a Euclidean space E_i of dimension greater than the dimension of the space X_i. Consider the case in which the encoding functions g_i carry X_i to Euclidean 4-space. Choose as the matrix for the function g_1

$$M_1 = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 2 & 3 \end{matrix}$$

and choose for the matrix of g_2

$$M_2 = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 5 & 6 \end{matrix} .$$

With this encoding, the matrix of the function

$$(F_1,\ F_2,\ F_3,\ F_4)^T$$

is the product of the matrix (M_1,M_2) with the matrix (I,I) where I is an identity matrix of order 3.

In this case a network need only compute the linear functions given by the rows of (I,I). This requires only one unit of time. Notice that no computation involving the g_i's is counted.

Theorem 4.1 and the remarks on its relation to the

sets (X_i/ h_j·F) still leave open the question of finding the highest dimensional separator sets for a given function. This is an issue because in the statement of the theorem the functions h_i are given. If these encoding functions are allowed to vary, then the size of the separator sets may also vary. We consider the problem of finding minimizing encodings for functions F: X_1 x...x X_n--->Y where X_i and Y are linear spaces and where F, all g_i and h_i are linear transformations.

The most elementary case is the one in which Y is one-dimensional.

Lemma 4.2: If $F(x_1,...,x_n) = x_1 +...+ x_n$, where the x_i are Real numbers, then there exists a (2,1)-network with n input spaces, each R, that computes F in time $INT[\log_2(n)]$.

Proof: We proceed by induction on n. If $n \leq 2$, then the assertion is trivial. Thus suppose that the assertion has been demonstrated for n-1 where $n \geq 3$.

If n = 2j, write

$F(x_1,...,x_n)=$

$[x_1 + x_3 +...+ x_{2j-1}] + [x_2 + x_4 +...+ x_{2j}]$.

By the inductive assumption, the function

$x_1 + x_3 +...+ x_{2j-1}$

can be computed in time $INT[\log_2(j)]$, and the same

time is sufficient for the computation of

$x_2 + x_4 + \ldots + x_{2j}$.

Thus $F(x_1,\ldots,x_n)$ can be computed in time

$INT[\log_2(j)] + 1$.

However

$INT[\log_2(j)] + 1 = INT[\log_2(n)]$.

Suppose that n is odd. If n is odd and

$2^{q-1} < n \leq 2^q$, then $INT[\log_2(n)] = q$.

Furthermore, $n \leq 2^q - 1$ and hence

$2^{q-1} < n+1 \leq 2^q$.

Thus $INT[\log_2(n+1)] = q$.

We can now reduce the case where n is odd to that in which n is even. To do this replace F by $F(x_1,\ldots,x_n) + x_{n+1}$ and compute this replacement only when $x_{n+1}=0$. ▒

Lemma 4.2 can be applied to the function F of Example 1. The lemma is applicable to the case in which no coordinate changes are allowed in either the domain or the range of F. We apply Lemma 4.2 separately to each of the functions F_i. For $1 \leq i \leq 3$ the function F_i has two arguments (the value of n in the lemma is 2). The function F_4 has 6 arguments. Each of the functions F_i must be computed by a (2,1)-network. It follows from Lemma 4.2 that each of the functions F_i, $1 \leq i \leq 3$, requires $1 = INT[\log_2(2)]$ units of

time, while F_4 requires INT[$\log_2$(6)] = 3 units of time. The next lemma, which is also a corollary of Lemma 4.2, is applicable to the case in which an encoded version of a linear (Real-valued) function is computed where the encoding functions may be chosen to make the computing time a minimum.

Lemma 4.3: Suppose that for $1 \leq i \leq n$, X_i is a Real vector space of dimension n_i, and suppose $F: X_1 \times \ldots \times X_n \rightarrow R$ is a linear function. Denote by $[F|X_i]$ the restriction of F to the subspace X_i. The minimum delay in computing F with a (2,1)-network is

INT[$\log_2$(ΣL_i)]

where

$$L_i = \begin{cases} 0 \text{ if } [F|X_i] \equiv 0 \\ 1 \text{ otherwise.} \end{cases}$$

Furthermore the lower bound on the computing time given by Theorem 4.2 is attained.

Proof: For each i with rank $[F|X_i] \neq 0$, choose a vector v_i so that $[F|X_i](v_i) \neq 0$. It is easy to see that the linear space spanned by v_i is a separator set in X_i for the output vertex of the network that computes F, because $[F|X_i]$ is 1-1 on the space spanned by v_i. Lemma 4.2 shows that the delay is at least INT[$\log_2$($\Sigma_i L_i$)]. For each v_i we choose a basis for X_i that contains v_i as one of the basis elements.

Further, without loss of generality, we can assume that $[F|X_i](v_i)=1$. For each $1\le i\le n$, let $\{e_{(i,j)}:1\le j\le n_i\}$ be a basis for X_i with $e_{(i,1)} = v_i$, if $[F|X_i]\neq 0$. Then we see that if $x\in X_1 x...x X_n$,

$F(x) =F(\Sigma_i x_i e_{(i,1)}+s_i)$,

where $F(s_i)=0$.

Thus

$F(x)=x_{i(1)}+...+x_{i(r)}$

where the indices $i(1),...,i(r)$ are exactly those i's for which $[F|X_i]\neq 0$. Then, $L_{i(j)} = 1$ for $1 \le j \le r$, and $r = \Sigma_i$ rank $[F|X_i]$. To complete the proof in the case of no encoding, apply the Lemma 4.1.

We turn to the encoded case. Assume that $g_i:X_i$ ---> E_i are a collection of linear maps that encode the domain of F and suppose that h:R---> R is an encoding of the range of F. Because h must be one-to-one, it is still true that the one-dimensional spaces spanned by the v_i are separator sets (in each X_i) for the output vertex of the network whose states are in the range of h. The lower bound given by Lemma 4.2 applies. Thus encoding cannot decrease the computing time in this case. ▒

With Lemma 4.3 in hand, let us return to Example 1. The lower bound on the minimum computing time for a (2,1)-network to compute F (with no coordinate changes

allowed in the range of F) is the maximum over i of the minimum computing times required for the F_i. In the case of Example 1, the minimum computing time for each of the functions F_1, F_2, and F_3 is 1, while the minimum computing time for the function F_4 is INT[$\log_2$(2)] = 1, since Σ L_i=2. However this lower bound of one unit of computing time can be achieved only by allowing separate choices of coordinates in X_1 and X_2 for each of the functions F_i. These coordinate choices are incompatible and it is for this reason that it is not possible to compute F in one unit of time even though each component F_i can be computed separately in one unit of time.

So far the concept of computing an encoded version of a function (Definition 4.3) allows the freedom to change coordinates in the range since we are free to change the h_i appropriately. It is easy to think of instances in which such freedom would not be appropriate.