Computing with Structured Connectionist Networks

Jerome A. Feldman, Mark A. Fanty,
Nigel Goddard and Kenton Lynne
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 213

April 1987

Abstract

Massively parallel (connectionist) computation is receiving widespread attention. Both the theory of connectionist computation and its applications to a broad range of problems are advancing rapidly. Somewhat less attention has been paid to the experimental methodologies of designing and testing massively parallel networks. This paper describes a coordinated set of specification, simulation and monitoring tools that have proven to be quite useful in our research. Also included are an overview of the structured connectionist paradigm and several sample applications.

This work was supported in part by NSF/CER grant No. DCR-8320136, in part by ONR/DARPA research contract No. N00014-82-K-0193 and in part by ONR research contract No. N00014-84-K-0655. The Xerox Corporation University Grants Program provided equipment used in the preparation of this paper.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-213
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Computing with Structured Connectionist Networks
5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Jerome A. Feldman, Mark A. Fanty, Nigel Goddard and Kenton Lynne
8. CONTRACT OR GRANT NUMBER(s): N00014-82-K-0193, N00014-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Rochester, Computer Science Department, Rochester, NY 14627
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209
12. REPORT DATE: April 1987
13. NUMBER OF PAGES: 35
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report): Distribution of this document is unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):
18. SUPPLEMENTARY NOTES: None
19. KEY WORDS (Continue on reverse side if necessary and identify by block number): massively parallel computation, connectionist models, massively parallel networks, simulation
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

Massively parallel (connectionist) computation is receiving wide spread attention. Both the theory of connectionist computation and its applications to a broad range of problems are advancing rapidly. Somewhat less attention has been paid to the experimental methodologies of designing and testing massively parallel networks. This paper describes a coordinated set of specification, simulation and monitoring tools that have proven to be quite useful in our research. Also included are an overview of the structured connectionist paradigm and several sample applications.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1 Computational Constraints

Rapid advances both in the neurosciences and in computer science are beginning to lead to a new interest in computational models linking animal brains and behavior. In computer science, there is a large and growing body of knowledge about parallel computation and another, largely separate, science of artificial intelligence. The idea of looking directly at massively parallel realizations of intelligent activity promises to be fruitful for the study of both natural and artificial computation. Much attention has been directed towards the biological implications of this interdisciplinary effort, but there are equally important relations with computational theory, hardware and software. In this article, we will focus on the design and use of massively parallel computational models, particularly in artificial intelligence. Much of the recent work on massively parallel computation has been carried out by physicists [Hopfield 1986] and examines the emergent behavior of large, unstructured collections of computing units. We are more concerned with how one can design, realize and analyze networks that embody the specific computational structures needed to solve hard problems. Adaptation and learning are treated as ways to improve structured networks, not as a replacement for analysis and design.

From a computational viewpoint, animal brains represent a remarkable machine with properties radically different from those of conventional computers. The line of research outlined in this article starts from the assumption that an abstract computer based on the computational properties of animal brains may prove to be particularly well suited for problems such as vision, locomotion and language understanding. By taking seriously the computational constraints faced by nature and the structure of the tasks known from artificial intelligence work, we hope to discover the algorithms that might be employed by animals and might be effective for artifacts.

Even a crude analysis of neural computation provides several major constraints on this enterprise. When asked to carry out any of a wide range of tasks, such as naming a picture or deciding if some sound is an English noun, people can respond correctly in about a half-second [Posner 1978]. This means that the brain, a device composed of neural elements having a basic computing speed of a few milliseconds, can solve difficult problems of vision and language in a few hundred milliseconds, i.e., in about a hundred time steps. The best AI programs for these tasks are not nearly as general and require millions of computational time steps. This hundred-step-rule is a major constraint on any computational model of behavior. It also turns out that the same timing considerations show that the amount of information sent from one neuron to another is very small, a few bits at most. The range of spike frequencies is limited and the system is too noisy for delicate phase encodings to be functional. This means that complex structures are not transmitted directly and, if present, must be encoded in some way. The fact that the critical information must be captured in the connections has given rise to the name "connectionist" for this kind of model.

The number of neurons in the human brain is estimated to be about 10^{11} and this presents a number of other serious constraints. For one thing, the number of cells in each retinal ganglion is greater than 10^6, which this means that any vision algorithm that required N^2 units can be automatically ruled out. For example, there could not be a separate unit for each possible line joining two points on the retina.

Many attractive algorithms for higher level tasks also run afoul of the size constraint. For example, one could not have separate units for visual percepts at each different location. Considerable work has gone into what amounts to coding tricks to circumvent these constraints [Feldman & Ballard 1982; McClelland & Rumelhart 1986].

One might think that biological constraints like those mentioned above need not be taken seriously in designing connectionist hardware and software. Certainly electronic switching times are a million times faster then neural ones. But the size constraint and particularly the connectivity constraints are much more serious for artificial systems. Animal brains derive much of their power from their very large fan-in and fan-out: each unit can be connected to thousands of others. For conventional chips, a fan-out of six is quite unusual. There is some research [Bailey & Hammerstrom 1986] on very high circuit connectivity, but nothing close to a solution. In software simulations like the ones described in this article, excessive connectivity leads to serious performance problems. It seems likely that the massively parallel computational solutions evolved by nature will prove useful in computer science and engineering for the foreseeable future. Connectionist computer research aims to understand the principles of massively parallel computation and to apply them to appropriate tasks.

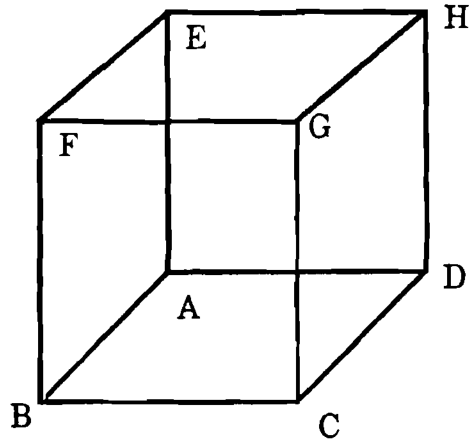
2 Introduction to Connectionist Modelling

We can capture some of the flavor of connectionist computation with a simple example. The cube shown in Fig. 1 is a famous optical illusion originally due to the Swiss naturalist, L.A. Necker (1832). Most people initially see the cube with the vertex B closer to them, but it also can be seen as a cube with vertex H closest to the observer. If you focus on vertex H and imagine it coming out of the paper toward you, the picture will flip to the H-closer cube. Notice that the flip also takes less than a second. The Necker cube is interesting to psychologists because it will flip spontaneously between the two views if you keep looking at it. It is interesting to us because of what it tells us about parallel computation.

You have observed how quickly the Necker cube flips state and know how slow the underlying human computing elements are. It seems unlikely that a sequential program on such a slow device could do the job. But the situation is much more complex. We know that both human and computer vision require several levels of processing. Typical levels include edge segments, lines, vertices, faces and object descriptions. The edges and lines are the same for both the H-closer and B-closer cubes, but many other visual features are seen differently in the two views. For example, vertex C is oriented into the paper in the B-closer reading, but out of the paper in the other reading. Similarly C is closer than G in the B-closer reading and all these perceptions are mutually consistent and reinforcing. The remarkable fact is that our visual system simultaneously flips all these perceptual decisions from one mutually consistent reading of the cube to the other. This illustrates the key cooperative property of massively parallel computation and why it is conceptually different from Von Neumann computation on standard machines.

Fig. 1 also illustrates some of the details of the connectionist paradigm. In our models each item of interest is represented as a computational unit, with connections to many other units. Each unit has a level of activity (here between -100 and 100) and automatically sends the value of this activity along all its outgoing connections. The connections can be drawn explicitly (cf. Fig. 2) or displayed implicitly as in Figs. 1 and 3. For example, the top double arrow in Fig. 1 depicts the fact that the unit representing the H-closer cube has positive 2-way links with the four relative distance detectors: A < E, etc. Each unit at each level in this network has a rival to which it is connected by a mutual inhibition link. The strengths of links into a particular unit can be displayed as shown in Fig. 3. The only other information needed for a complete model is the rule by which a unit computes its new activity from its inputs and its old activity; the simulator described in the next section allows a wide variety of update algorithms. We can assume for now that the units compute the sum of their positive and negative inputs. Networks like Fig. 1 are not very sensitive to the exact choice of unit computation rules; this is one of the reasons for their attractiveness. Units that are all mutually connected by negative links are said to comprise a winner-take-all network. Such networks are one of the main decision mechanisms in connectionist models and have known neurophysiological analogues.

Much of the effort in massively parallel AI is dedicated to using computational frameworks like that in Fig. 1 to build models of intelligent activity. Advantages of this approach include its link to natural intelligence, increased noise resistance and ease of implementation on parallel hardware. But the main advantage of the connectionist approach is that it provides a much better way of specifying some computations. There does not appear to be any alternative way to describe the Necker cube phenomenon that is nearly as clear and concise as Fig. 1.



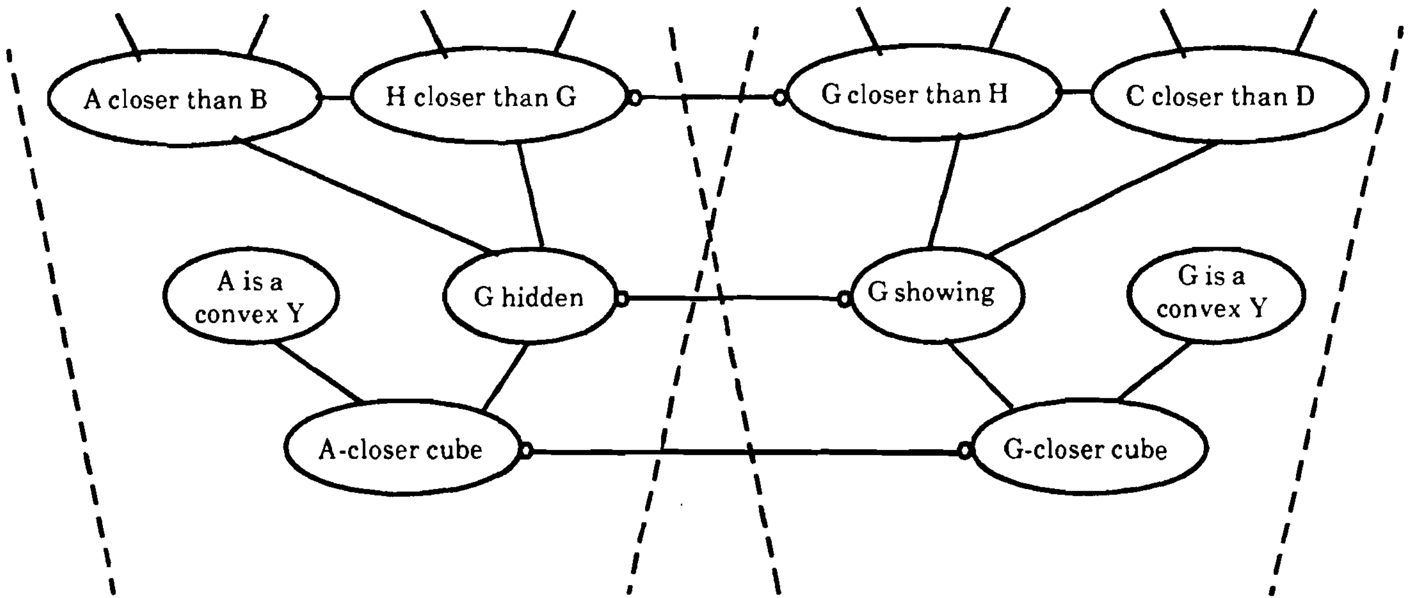


Figure 1: Network fragments for the two readings of the Necker cube.

Researchers in AI and related areas of Cognitive Science are using connectionist models to study a variety of tasks. Vision is an area where massive parallelism fits naturally. It is less obvious that the methodology will be effective in natural language research, but there have been some very nice results in that domain. One problem that has worked out particularly well is disambiguation. Consider what happens when you hear or read a sentence like:

Bob threw a ball.

You automatically assign a meaning to each word, despite the fact that most words in English are ambiguous. For example, the words "threw" and "ball" both have quite different meanings in a sentence like:

Bob threw a ball for charity.

The problem is to develop algorithms that exhibit this behavior and help explain its basis. Linguists and other cognitive scientists have worked at length on these issues and have developed sophisticated theories. The contribution of AI has been to encode these theories in programs so that they can be tested and, if correct, used in applications. As with the Necker cube example, it appears that massively parallel models constitute the best way to carry out the encoding.

In language, as in vision, theory calls for several distinct levels of representation and processing. Three of these are shown in Fig. 2. The lexical level comes after the individual sounds or letters have been formed into words and corresponds to what you would look up in a dictionary. The word-sense level corresponds to the various meanings listed under a word in the dictionary -- the problem, of course, is to pick the right one. The case level comes from linguistic theory and captures the idea of the different roles that words can play in a sentence. The ones used here are:

Agent: The person or thing carrying out an action.

Object: The thing acted upon.

Recipient: Beneficiary of the action.

The key linguistic insight is that there are constraints on the possible word-senses that can fill various case roles. For example, one can not propel a dance so these are incompatible. It also makes no sense to sponsor a sphere. The only other fact used in the model is that some word senses are more frequent than others.

We can now see how Fig. 2 presents a connectionist computational model of disambiguation. As with the Necker cube, compatible units have positive links and incompatible units have negative winner-take-all links. The model assumes that as each word reaches the lexical level it spreads activity to the various word senses to which it is connected. Since the more frequent sense has a higher weight, it will tend to dominate less frequent senses. As additional words come in, they will activate more word senses and case roles. The simple sentence "Bob threw a ball." will activate a mutually consistent set of units and the alternatives are never noticed. The additional words "for charity" will activate the "dance" meaning of ball. This will weaken the "sphere" meaning which will, in turn, reduce the activity of "propel"

because there is no longer a suitable object in the sentence. An alternative stable coalition will develop and suppress the original interpretation. The two alternative coalitions are quite similar to the two readings of the Necker cube. This is no accident: the idea of a cooperative-competitive network pervades parallel models. Work on language problems like disambiguation is quite advanced and offers simple explanations for many phenomena. For example, a context that biased us towards the "dance" sense of ball would be modeled as providing that meaning with a head start in activity for its competition with "sphere." Again, the massively parallel paradigm is the simplest way to express this idea. The example of Fig. 2 is part of the work of Gary Cottrell [1985]; related work has been done by Waltz and Pollock [1985] and there is currrently a great deal of interest in connectionist models of natural language. Before considering more applications, we turn our attention to computational issues and propose a framework for designing massively parallel computation networks.

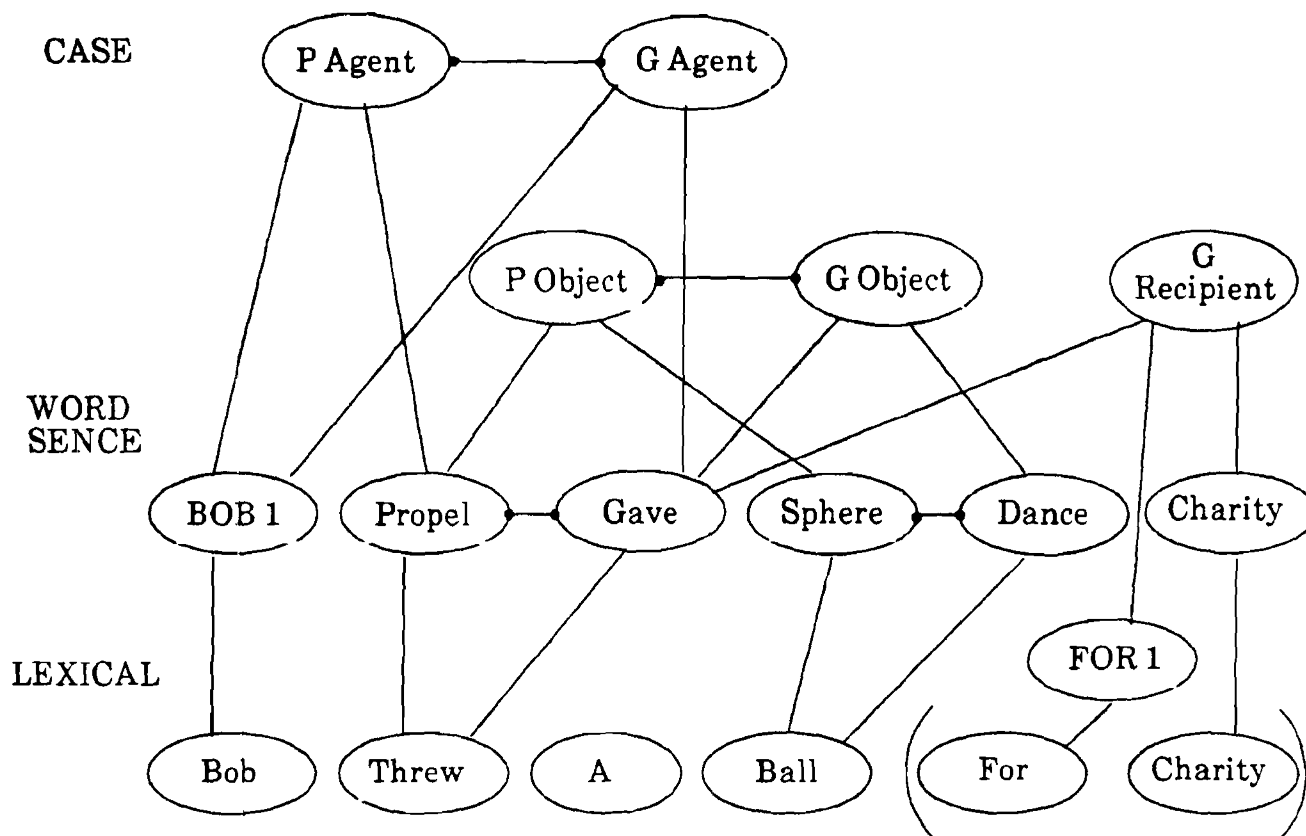


Figure 2: Disambiguation example.

3 Specifying and Simulating Networks

As with other computational models, it is important for the researcher to be able to implement and test his ideas. Different researchers working in diverse areas need the ability to build and simulate radically different networks. Aspects which may differ include connection pattern, activation functions, and amount of data associated with each network node.

Connectionist networks consist of simple computational elements (units) which communicate by sending their level of activation via links to other elements. The units have a small number of states and compute simple functions of their inputs. Associated with each link is a weight, indicating the "significance" of activation arriving over that link. The behaviour of the network is determined by the pattern of connections, the weights on the links, and the unit functions.

The Rochester Connectionist Simulator [Fanty and Goddard, 1987] has evolved from simple beginnings [Small et al., 1983] into a sophisticated research tool which supports construction and simulation of a wide variety of networks. The main design criterion has been flexibility. Each unit can compute a different function, any amount of data may be associated with each unit, and an arbitrary connection pattern may be specified. This flexibility exacts its cost in time and space as compared to special purpose simulators. It is time-expensive because each unit and link is simulated by a separate function call, and space expensive because each unit and link must have an explicit representation.

The particular network paradigm supported by the simulator is given in [Feldman and Ballard, 1983]. Each unit has a number of sites at which the incoming links arrive. The provision of sites allows differential treatment of inputs, since the links themselves do not indicate their origin at the destination site. Associated with each unit are various pieces of data, including potential, output and state. The potential corresponds to the unit's level of activation, the output is transmitted along all links emanating from the unit, and the state is used to make simple decisions about how to interpret the unit. Associated with each link, site and unit is a function representing its action.

Network construction

The construction environment provides a set of primitives for specifying a network. Typical primitives are those to make a unit or link, or name a unit or array of units. The primitives provide a conceptual structure through which networks may be specified. Rather than define an entire new language for specification, we use an existing programming language, augmented by the primitives.

A network is built in the simulator by a user program written in C. The primitives are implemented as library functions, called from the user program. The specification parameters for units, sites and links include initial data values and an activation function. These activation functions may be different in every case, either written by the user or supplied from a library. Within each unit, site and link, a general purpose data field exists, which can be used to point to an arbitrarily large structure. When displaying, saving or reloading the network, user-supplied functions are called by the simulator to handle these user-defined structures.

The large library of functions facilitates the construction task by supplying many of the commonly used unit, site and link functions. Library functions to create

particular network structures are also available. Researchers may add their own unit and structuring functions to the library, and so augment the utility of the simulator for themselves and others. An example of this is a library which greatly eases the construction of back-propagation networks [Rumelhart et al., 1986], either stand alone or as sub-modules within a larger network. This library allows arbitrary error propagation functions as well as any unit activation function.

Generic functions are provided to access the network data structure, enabling the user to write implementation-independent code. This has proved particularly useful when porting a network to the Butterfly version of the simulator (described in section 5). User code can also create and manipulate named sets of units, these sets then being accessible by name from the command interface during simulation. This enables sophisticated tracking of units whose behaviour is interesting. The simulator name table (containing names of units, sets and functions among other items) may be accessed and modified via functions called from user code.

An important consideration in specifying a network is the ability to give descriptions at different levels of abstraction. At the lowest level, single unit descriptions are given by the unit, site and link functions. At the next level, the pattern of connectivity is described by a set of functions which specify the links and groups of units. This may include modularity within the network, and for any large network necessarily reflects regularity. At a still higher level, network specifications given in user-defined languages may be read in and compiled into units and links by user-supplied functions (c.f. section 6 and Table 2 and also Blelloch [1986]).

The construction phase is a form of compilation, and as such involves error handling. Often mistakes are made in network specification, the most common being attempting to link two units, one of which does not exist. For large networks, where the construction phase can take a significant amount of time (minutes, or even hours on our large Butterfly machine), we would like to collect as many errors as possible before aborting the construction. The debugging facility will attempt to correct errors in a way which allows construction to continue, or can be set so the user is asked to correct errors from the command interface. The latter option allows examination of the current structure of the network, and construction of missing units and links on the fly.

A sample construction program

For an example of a network and its specification program, let us look at the Necker cube network in Figure 1, consisting of 26 units and 138 links. The units corresponding to the two views are displayed as black squares and dotted squares respectively. The figure shows the situation when the H-closer view has been recognized - all the units consistent with this view are at maximum activation, and those of the other view at minimum. Figure 4 in shows an intermediate stage when the two interpretations are competing.

The vertex units represent local 3D orientation information. The line units represent pairwise constraints between corresponding vertices, and the cube units represent the two possible interpretations. The links to or from a unit can be displayed with the graphics interface described in section 4, which was used to generate Figures 1, 3 and 4. Figure 3 shows the graphics panel when the links to one unit are displayed. The square black unit is the one being examined. The pentagons indicate which units send links to it, with the size of the pentagon indicating the

weight on the link. In the panel above the network diagram can be seen the state of four of these units, the cube unit, two depth units and one vertex unit, including link weights.

Table I shows a specification program for this network, written in pseudo-code. As can be seen, the structure in the network is reflected in the structure of the specification program. Units 0 to 8 represent the B-closer cube's vertex nodes, ABCDEFGH in order. Units 8 to 11 represent the depth level, A<E, B<F, C<G, and D<H in order. Unit 12 is the B-closer unit. Units 13 to 25 are encoded in a similar mannar for the H-closer network. The linking specifications implement this choice of encoding.

The build function calls *CreateUnits*, which makes the 26 units and names them. Each view's nodes are named as a 3 x 4 array of units, the first four for the "front" face, the second four for the "back" face, and the third four for the depth nodes. The two units representing the top level have special names, "B-CLOSER" and "H-CLOSER". *CreateInhibition* makes the inhibitory links between corresponding nodes in the two views' networks. *BiLink* is a library function which makes a bidirectional, symmetrically weighted link between two units, the range 0 to 1000 representing weights 0 to 1. *CreateExcite* makes the excitatory links betwen nodes within a single view network. The first line within the for loop makes links A-B, B-C, C-D and D-A. The next two lines make similar links for the EFGH and the four depth nodes. The fourth line links A-E, B-F, C-G and D-H. The fifth lines make the links between nodes ABCD and the depth nodes, the next line the links between nodes EFGH and the depth nodes. The two calls to the primitive MAKELINK make the links between the depth nodes and the cube view nodes; as can be seen, the weights in the two directions are not equal.

```
#define FRONT = 0 BACK = 1 DEPTH = 2 - literal constants

CreateUnits ()                                    - make units & sites
  {
    for i = 0 to 25
      j = MAKEUNIT("node",UFasymp)                - asymptotic function
      ADDSITE(j,"excite",SFweightedsum)           - weighted sum funct.
    NAMEUNIT("VIEW1",ARRAY,0,4,3)                 - view B units
    NAMEUNIT("B-CLOSER",SCALAR,12,0,0)            - view B top level
    NAMEUNIT("VIEW2",ARRAY,13,4,3)                - view H units
    NAMEUNIT("H-CLOSER",SCALAR,25,2,0)            - view H top level
  }

CreateInhibition ()                               - make inhibit links
  {
    for i = 0 to 7 BiLink(i,i+13,"excite",-575)   - link opposing vertex
    for i = 8 to 11 BiLink(i,i+13,"excite",-800)  - and depth units
    BiLink(INDEX("B-CLOSER"),INDEX("H-CLOSER"),"excite",-1000)
  }                                               - link top level units

CreateExcite (front,back,depth,cube)              - make excitatory links
  {
    for i = 0 to 3
      BiLink(front+i,front+((i+1)%4),200)         - around front face
      BiLink(back+i,back+((i+1)%4),200)           - around back face
      BiLink(depth+i,depth+((i+1)%4),300)         - between depth units
      BiLink(front+i,back+i,200)                  - between faces
      BiLink(front+i,depth+i,300)                 - between vertex and
      BiLink(back+i,depth+i,300)                  - depth units
      MAKELINK(depth+i,cube,"excite",500,NULL)    - from depth to cube
      MAKELINK(cube,depth+i,"excite",500,NULL)    - from cube to depth
  }

build() - build function
  {
    CreateUnits()                                 - make units
    CreateInhibition()                            - make inhibit links
    CreateExcite(INDEX("VIEW1",FRONT),INDEX("VIEW1",BACK),
    INDEX("VIEW1",DEPTH),INDEX("B-CLOSER"))       - VIEW1 links
    CreateExcite(INDEX("VIEW2",FRONT),INDEX("VIEW2",BACK),
    INDEX("VIEW2",DEPTH),INDEX("H-CLOSER"))       - VIEW2 links
  }
```

Code for setting up the Necker cube of Figure 1. Primitives in upper case.

TABLE 1

Simulation

Once a network has been constructed, simulation may begin. The simulation can be run synchronously or asynchronously. During synchronous simulation, all units use the output values computed during the previous step as their input. The order of simulation is unimportant, the network behaving as though all units update simultaneously. During asynchronous simulation, at each step a fraction of the units are updated, in pseudo-random order, with the new output value immediately transmitted to the other units. It is guaranteed that after a limiting number of steps every unit will have updated at least once. Both the fraction per step and the limiting number of steps are specified by the user. This asynchronous update protocol is useful for theoretical analysis of network operation. A special limiting case is when all units are asynchronously updated at each step.

Synchronous simulation is marginally faster than asynchronous, but in cases like the Necker cube example, synchronous updating with deterministic unit functions will always lead to the same outcome. Asynchronous simulation is a useful way to break symmetry conditions, and model random factors. By allowing the user to specify the seed for the random number generator, we enable repetition of a particular simulation run.

A command interface is used to control the simulation and modify the network during simulation. The interface allows any user function and a preloaded set of commands to be executed. In addition to this, user functions conforming to a special naming convention are considered by the simulator to be regular commands, thus allowing customization of the interface.

Several types of preloaded commands are available: simulation commands; modification commands; display commands; set commands; utility commands; and user commands. Simulation commands include those which adjust simulation parameters, such as update protocol. Modification commands include those to make units and links, and to modify network settings. Examination commands will display the units and links in varying detail. Set commands allow the user to manipulate named sets of units, providing operations such as union, intersection, inverse, etc. Utility commands include those to call UNIX shell commands, to read a command file, to save or restore a network from file, and to provide help. Command files greatly ease initialization of a network and repetition of simulations.

Performance

Two major issues in neural network simulation are size and performance. On a SUN3/260, a network of 2000 units (computing the weighted sum of their inputs), each with 100 links, took 25 seconds to construct, and performed 100 simulation steps in 83 seconds, giving an interconnect time of approximately 4μS. The number of links in a network is a crude but effective measure of its size; on the SUN3/260 with 8 megabytes of memory, the maximum number of links attainable before thrashing sets in is of the order of a quarter of a million. The parallel simulator discussed in section 5 can do much better.

With the flexibility now achieved, future work will be in the form of libraries of unit activation functions, unit structure customizations, and specialized commands. As specialized network structures are defined and analyzed, we will be able to produce libraries of functions to implement them.

The simulator command interface was designed for simple terminal operation, and is useful for controlling simulation and displaying limited amounts of information. The graphics interface described in the next section has proved to be an extermely powerful tool for displaying network information during simulation and aiding the network debugging process.

4 Graphics Interface

Designing a connectionist network to perform some computation is somewhatlike writing a program to execute on a sequential computer. However people find "programming" a connectionist network quite different from traditional computer programming. In sequential programs (and in most parallel ones) one expects specific sequences of events to occur at precise moments in the program's execution. Variables must have certain values, and groups of instructions must be executed exactly on cue with precise results in order for the computation to proceed correctly. To follow the progress of a sequential computation one usually needs to look at only a few critical variables and the instruction counter to verify that the program indeed is doing what one expects. This is in contrast to connectionist networks where the sequence of events and their results usually cannot be so easily specified or predicted. For example, the connections between units and the weights of connections are often randomly specified. In learning paradigms, the weights of connections between units will vary as a result of previous processing. Furthermore, in asynchronously executing networks, even whether a unit gets to compute during a network step is left to chance.

This freedom from rigidity allows connectionist networks to tolerate individual component failures as well as errors in input and intermediate computations. On the other hand, one must deal with the difficulties of trying to design and then debug a computation whose detailed progression cannot be predicted in advance. In a sequential computer program the end result of a computation is often contained in a single variable, while in a connectionist network, the "result" is often "spread out" over the network as a relationship among some subset of its units. Thus the questions of "where to look" and "what to look for" have very different answers for network computations as compared to traditional sequential programs. Consequently, traditional output and debugging tools are simply not adequate for dealing with connectionist networks. What is needed is a way to examine the entire network or some significant subset of it. Network designers soon learn this, and almost invariably develop some kind of graphical representation of their network's activity, but these are usually targeted to their specific problem or type of network. At Rochester we decided to develop a general purpose graphics interface that would be useful for a variety of network paradigms. Thus the Graphics Interface tool was developed as a companion to the Rochester Connectionist Simulator for the purpose of making the execution, debugging and documentation of connectionist networks easy, natural and even entertaining. A different approach to general graphics for connectionist simulation can be found in Zipser and Rabin [1986].

Overview

The Graphics Interface allows the user who has created a network with the simulator to graphically view and examine that network before, during and after it executes. Each significant aspect of each unit in the network can be displayed as a separate graphic object (or icon) whose size, shape or shading varies with the current value of that aspect. As the simulation runs, the icons are constantly updated to reflect changing values, giving an overall view of what the network is doing. In keeping with the philosophy of the simulator, the Graphics Interface was designed to give maximum flexibility to the user in how the network is to be displayed.

The Graphics Interface was specifically written to run on our Sun (Trademark) workstations using Sun Microsystems' graphics tool package. It was designed as a separate part of the simulator package: the user specifies that the Graphics

Interface be included as an option when the network is compiled into object code. If included, the Graphics Interface code will automatically create its own window which layers itself between the user and the simulator. The user now interacts with the simulator via the Graphics Interface window which either executes graphics commands given by the user or passes commands to the simulator as appropriate. It provides and maintains a graphics panel upon which the graphical representation of the user's network is displayed as the simulation proceeds. Fig. 3 shows the Graphics Interface tool being used in "Link" mode to examine the weights of connections for the Necker cube example. The simulator itself is unaware of the presence of the Graphics Interface and in fact will still generate text output to its own window, just as though the user were interfacing to it directly. In addition to displaying and running the network the Graphics Interface also has facilities that allow the user to

- display detailed textual information for specific units
- show network topology
- write the graphics display to a raster file
- put text and line drawings on the display for documentation purposes
- log the simulator and graphics commands for later replay
- map the mouse buttons to execute simulator or graphics commands

There is also a version of the Graphics Interface tool that communicates with the parallel connectionist simulator running on the department's BBN Butterfly multiprocessor described in section 5.

Operation

The user constructs the network as described in section 3. Once built, the user executes commands that display the network or parts of it. Fig. 4 shows part of the tool window used to set up the Necker cube example. The upper part of the right hand panel provides the commands for displaying the network; the lower part of that panel is used to run and show the network simulation itself. The display commands enable the user to specify that an icon be created to stand for some aspect -- potential, output, state or data -- of a particular unit in the network. If the user is interested in link weight modification, an icon could reflect the weight of a link from one unit to another. Multiple aspects of a unit can be displayed as separate icons thereby allowing as detailed a view of a particular unit's activity as desired. The icons, once created, will continue to "track" that aspect of their respective units for as long as the network runs. The manner in which the icons change in relation to the value of the aspect is controlled by the icon type selected. Each icon is realized by a family of icons, with each member representing a particular subrange of the range of values that the aspect can have. The number and size of icons in each family as well as how the icon "changes" as the aspect value changes are user defined. (Icons can be created in just a few minutes using Sun utilities). Typically, the size, shading or shape of the icon is varied, but in general the mapping of icon to value is arbitrary and up to the user. Default icons are available in several polygonal shapes which change size and shading but the user is encouraged to define specific icons for particular unit aspects when useful. For example, Fig. 5 shows part of a vision network used to detect edges and vertices. The

two square arrays on the right side are tracking the same array of units whose output is the gradient direction (or slope) of an edge in the "pixel array" on the left. The top array uses default icons which simply map size to output value. The bottom array uses icons specifically designed to reflect the slope representation of the units' output. The superiority of the bottom array over the top in conveying what these units are actually doing is apparent.

Through commands or mouse actions the user specifies exactly where the icons are to appear in display space which is effectively an unbounded Cartesian plane. The graphics panel (which can be stretched vertically and horizontally) always shows some finite rectangle of the display space and can be moved via commands or mouse actions to show different parts of display space. Thus the size of the network that can be displayed is limited only by the machine memory. (Each displayed aspect requires 48 additional bytes of memory). The user can also add text and line drawings to the display space to document the network as shown in Figs. 3, 4 and 5. All objects (icons, text, drawings) appearing on the graphics display can be freely moved around with commands or the mouse as dictated by taste or function.

Once the network has been laid out in display space, the user can run the network for a specified number of steps, watching the icons change according to what the network is doing. Only those icons currently within the display window are updated and even then only when necessary, so that the graphics interface adds a minimum amount of overhead to the simulator running time. At any time, the network can be examined more closely, or modified by typing simulator commands (or mapping them to mouse buttons). Selecting any icon on the display with the mouse will cause the detailed values for that unit to be displayed in the information panel and these values will then also be tracked as the network runs.

Different operation modes are provided in order to enable the user to perform specialized interactive functions. When in "Link" mode (Fig. 3), the user can click the mouse button over a unit's icon to see its connections and their weights. This is extremely useful in verifying that the network topology is what one wanted and in debugging learning networks. Through the "Draw" and "Text" modes, the user can create, respectively, line drawings or text objects that will appear on the display. In "Custom" mode the user can map mouse actions to simulator or graphics commands which will then be executed with a single click of the mouse button. For example, the user could map a mouse button to the simulator command for changing potentials and then click that button over a unit's icon to change its potential.

All the commands for displaying, changing, deleting and moving objects in display space can be read from a file and executed. In fact all the actions that the user executes interactively are internally converted to these commands which can be automatically written out to a specified log file, which can later be replayed. In addition, all commands can be executed by user code through a special programming command interface.

Using the Graphics Interface significantly reduces the time needed to get a connectionist network up and running. With the proper unit aspects displayed, one can quickly determine if a network is working properly and if not, where in the network the problem lies. Much of the power of the interface lies in its dynamic properties which do not show up in static pictures. For example, oscillations which can paralyze a network can be caught almost immediately. The graphics interface has made it much easier to specify and debug the large structured networks that we work with at Rochester. In essence, the Graphics Interface is analogous to a source

level debugger for sequential programs: it allows one to quickly and without interference watch the progress of a computation when that computation is being carried out by a connectionist network. It thus allows the network designer to concentrate on the scientific endeavors of understanding the properties of connectionist networks, with minimal effort directed toward figuring out how to display and debug them. We believe that developing useful tools is important in any research effort, but is particularly important in connectionist research where the concept of computation and what one does to observe it can be so different from traditional computer science.

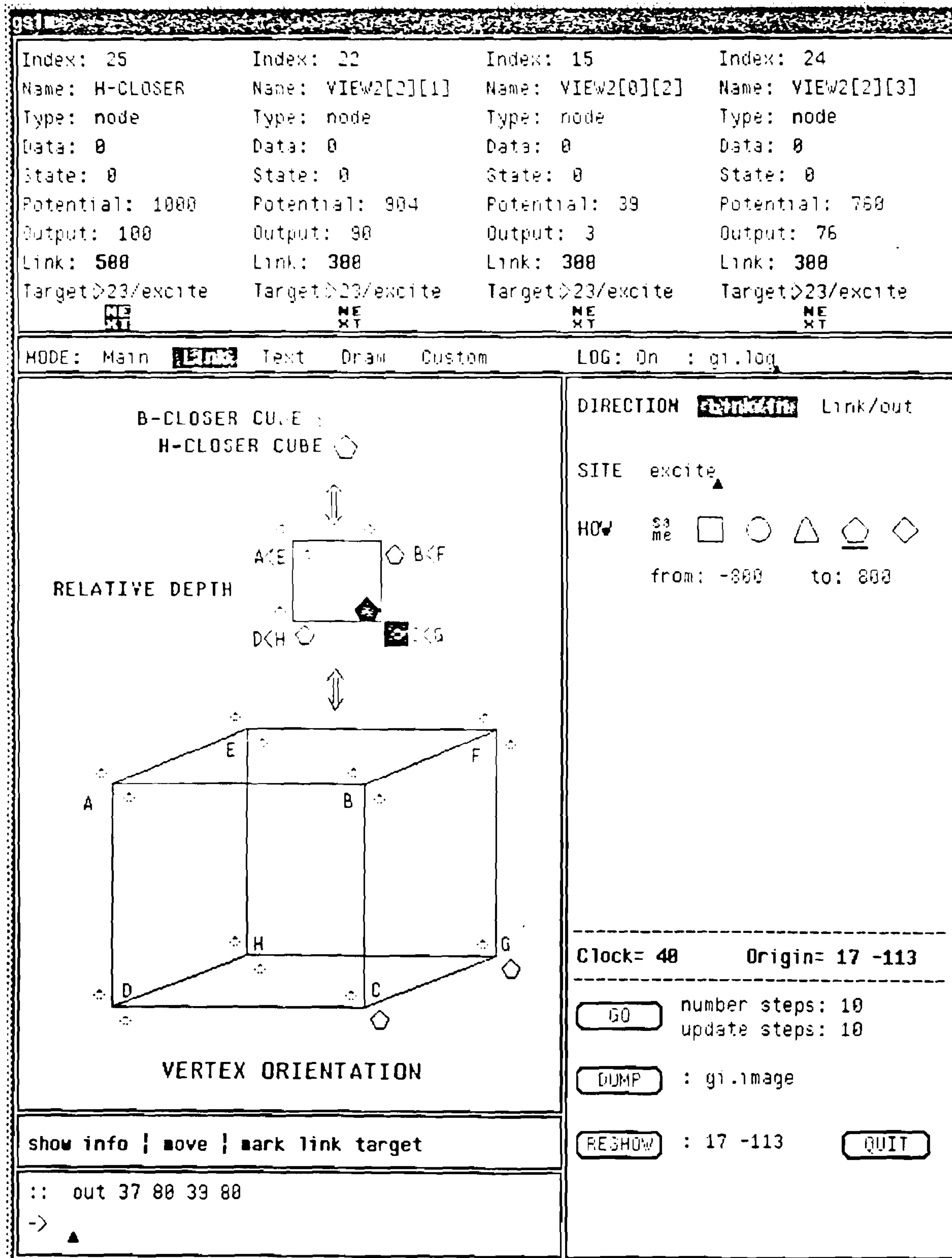


FIGURE 3.

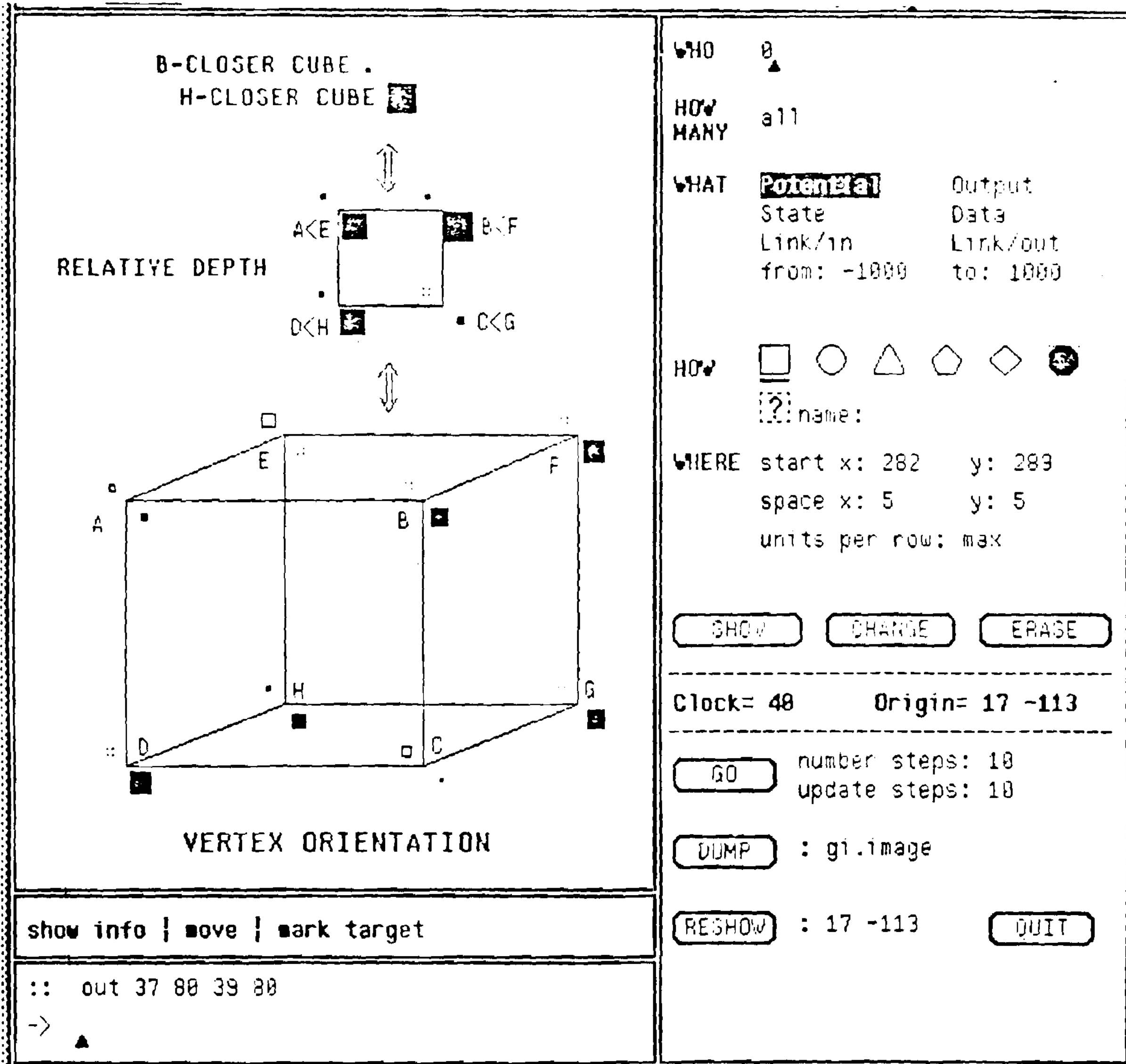


FIGURE 4.

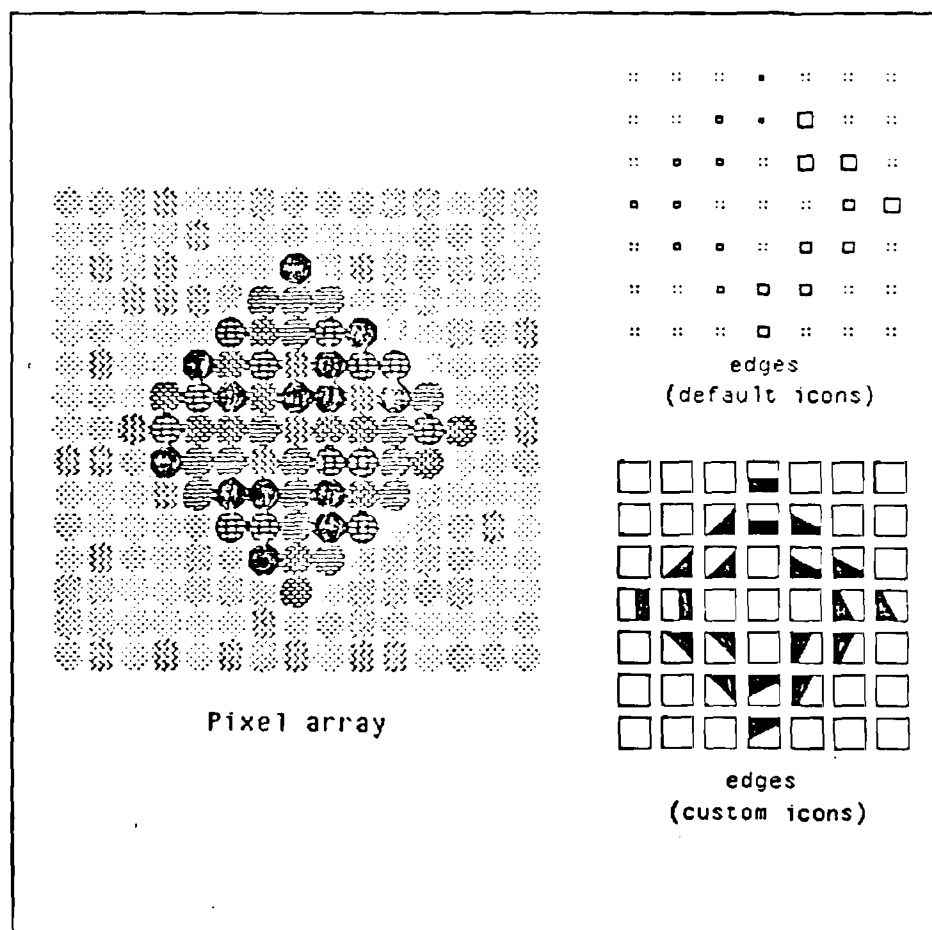


Figure 5: Two iconic representations of edge detectors.

5 Parallel Implementation

The connectionist simulator has been implemented on a BBN Butterfly Multiprocessor [BBN 1985]. The parallel simulator looks and functions much like the uniprocessor simulator. Code can be ported with little modification, as long as it does not directly access the network data structures. If the user is content with a naive network partition, he can ignore the fact that the simulator is actually running on several processors.

Using the Butterfly significantly increases the speed and capacity of network simulations. Our largest Butterfly configuration is 120 processors with a total of 120 megabytes of memory. This will easily run networks which will not begin to fit on any of our uniprocessor machines. Even with smaller networks, simulations that would run for hours take only minutes.

A Short Description of The Butterfly

The Butterfly Multiprocessor consists of up to 256 nodes, each of which has a Motorolla 680x0 processor, special switching hardware, and up to 4 megabytes of memory. The processors are connected by a log depth butterfly switching network. The Chrysalis operating system supports the C programming language (among others) and provides message passing and memory sharing primitives. There is no global memory, but portions of a processor's local memory can be mapped in by other processors. This is the major communication mechanism used by the simulator. Memory sharing is limited as each segment of memory mapped in (up to 64 kilobytes), as well as each message port, uses a segment address register (SAR), of which a process can have at most 256. This rules out strategies which require the entire network data structure to be shared.

There is currently no file system for the Butterfly. Programs and data are compiled and stored on a host machine. Executables are downloaded over an ethernet connection. Our machine does not have floating point hardware, though it is available as an upgrade. This supported our original decision to use integer arithmetic in the simulator.

Overview of the Implementation

The user sits at a UNIX workstation and communicates with a control process running on one of the Butterfly processor nodes. The network data structures and simulation functions reside in several simulator processes, each on a separate Butterfly processor node. The simulator processes are like scaled-down uniprocessor simulators. Each is responsible for some subset of the units being simulated. They receive commands from the control process, rather than directly from the user.

Unit, site and link structures are represented just as with the uniprocessor simulator. The only difficulty is the handling of links between units on different simulators. The solution to this problem is illustrated in Figure 6. Each simulator allocates and maintains an output array for the units stored locally. In order for links to cross processor boundaries, the output arrays must be shared. Every simulator maps in the output arrays of every other simulator. Once this is done, the

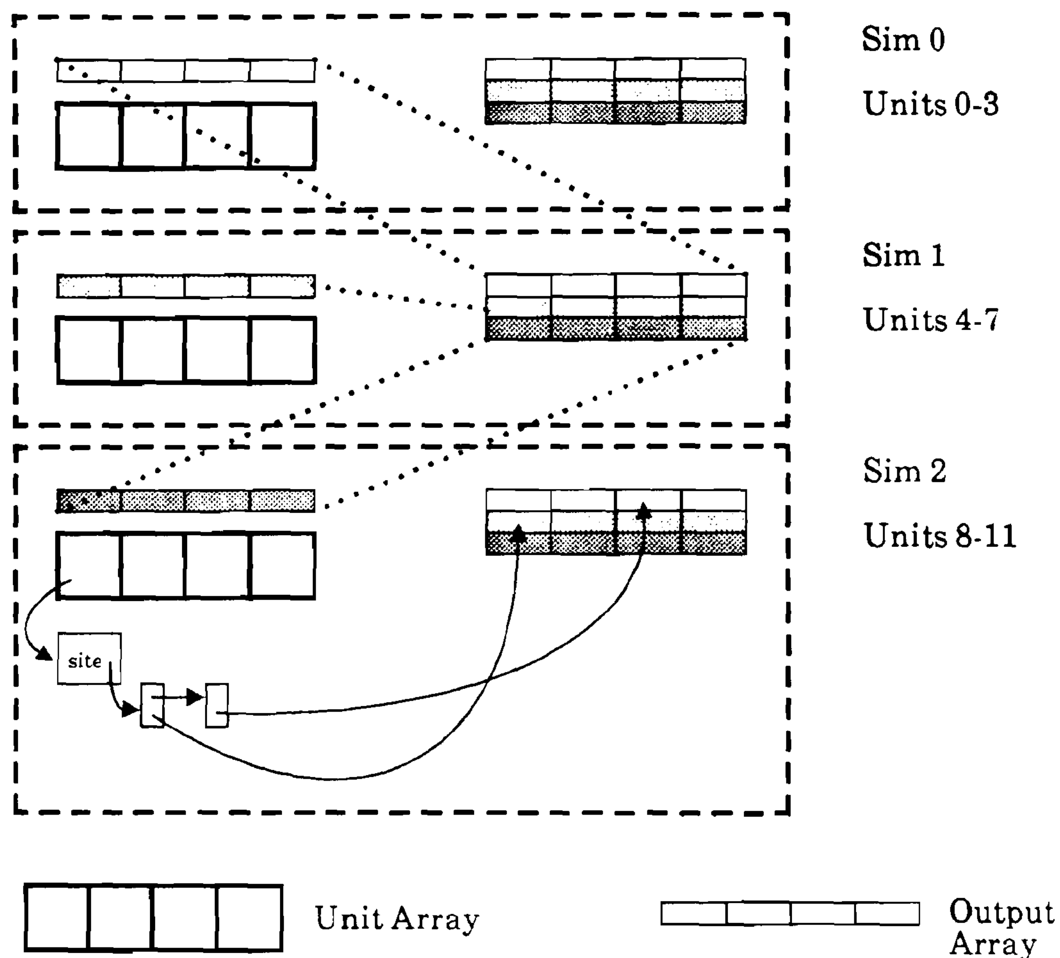


Figure 6.

Each simulator maps in the output arrays of the others.
The links from unit 2 and unit 4 to unit 8 are shown.

remote output arrays appear, to the programmer, to be local. Remote links use indirect references (C pointers) just as local links do. When a remote memory location is accessed, hardware translates this into an access over the switch.

The graphics interface described in section 4 can also be used with the Butterfly simulator. A display program runs on a workstation and communicates with the control process on the Butterfly over the ethernet. Every time the display is to be updated, the display process sends a data request for those units which are currently visible. The array of requests is put in a globally readable memory structure and the simulator processes simultaneously scan this structure looking for

requests they can satisfy. They write the requested data values in another globally writable array. When all the requests are fulfilled, the answers are sent back to the display process which updates the screen.

Sharing output arrays takes care of most of the communication requirements between simulators. The rest are handled by a global name table. Unit, set, type, site, function and state names are all entered in the global name table. The control or any simulator process can enter symbols and retrieve those entered by other processes. The name table consists of a number of hash tables. Two hash indices must be generated for each symbol: one for the table number and one for the offset. Memory for a table is allocated on each node running a simulator process. Due to a scarcity of SARs, it is not possible for each process to keep the entire name table mapped in all the time. For each name lookup, the correct table must first be mapped in using its global identifier. When entering symbols, the table is locked using an atomic-add on a lock variable.

Using the Butterfly Simulator

The user must provide a function to build the network as shown in Table 1. This is linked in with the control process code. Unless library functions are used, the user must also provide functions to describe the behavior of the units. These are linked in with the with the simulator process code (which is the same for every simulator). Except for minor adjustments, code written for the uniprocessor simulator can usually be used.

After establishing a connection to the Butterfly and grabbing a cluster of processor nodes, the user runs the control program. This program will prompt for the number of processors on which to start simulator processes and for the default number of units on each processor. The simulator processes are downloaded automatically and communication is established. When this finishes, the user is looking at a command interface (run by the control process) like that on the uniprocessor simulator.

A simple way to build a network is with a function executed by the control process. For each unit and link made, the information must be shipped across the switch to the appropriate simulator. This can be quite a slow process. It is also possible to build the network in parallel if the user is willing to write special-purpose code to do this. From the command interface, functions on any (or all) simulators can be called by name. Each simulator can build units which are to reside locally as well as make links to any local unit. Coordinating parallel constructing can be complicated, depending on the structure of the network. It is also possible to mix sequential and parallel building. For example, all the units could be made sequentially and the links could be made in parallel. Since there are typically many more links than units, this would result in a substantial speedup. Building networks in parallel presents a problem when more than one processor encounters an error. Our solution is to use an atomic lock to ensure that, at any time, only one simulator is trying to report errors and read fixes.

The default partitioning of the network fills the simulators with units sequentially, so that each simulator has the same number of units. Since the number of links per unit can vary, this might result in a very uneven partition of the computation. Currently, the only way to override this is to prematurely terminate allocation of units on a simulator. Custom layout and dynamic repartitioning of the network is a possible topic of future work. For each simulator, the number of nodes

and links allocated as well as the amount of time spent computing the last simulation step can be examined interactively by the user.

Because of the programming environment, some normally simple tasks are more difficult. On the Butterfly, UNIX-style pipes are not available. In particular, long listings cannot be piped through a scrolling program which waits for user feedback after each screenfull of text. A mechanism has been established for sending output to a waiting UNIX process which can then further process it. A more serious limitation is the lack of a file system. Though network descriptions could be sent to a UNIX process and saved on disk, we have so far not implemented this---partly because of the potentially huge size of the networks involved. When the Butterly acquires multiple disks which can be written in parallel, quick file saves will be possible. Using data files to build networks and run simulations is possible, either by down-loading the data files as memory objects or through the graphics interface communication link.

Scrolling through a list of units and their activations is all but useless for huge networks. The graphics interface can be used to display hundreds of units and links simultaneously, in a visually meaningful way. The magnitude of several thousand links can be examined in a few minutes by selecting units with the mouse. Except for a couple of extra steps at startup, the graphics interface works much the same way for the Butterfly and uniprocessor simulators, except the display program runs on a workstation as an independent process. Future generations of the graphics interface may run in parallel on the Butterfly as well.

Performance

The critical parameters are size, construction time and execution time. The size gain is substantial, and would be even bigger with a memory upgrade to the maximum of four megabytes per processor. We currently have one megabyte per processor. Each unit uses 40 bytes; each site and link uses 20 bytes. If there are 1,000 units per simulator, each can have about 40,000 links and sites. This means the total capacity is around 100,000 units and 4,000,000 links.

The sequential build time is fairly slow. A test network was built, which had 100,000 units and 3,000,000 links, 30 links to each unit from a random source. When it was built sequentially from the control process, the units and sites took 9 minutes to build and the links took 2.75 hours. When a network of the same size was built concurrently, the total time for units and links was 56 seconds. The super-linear speedup is due to concurrency and locality. This difference makes us anxious to develop concurrent build techniques for all our networks. The speedup would still be dramatic even if the units were built sequentially and the links in parallel.

When the above tests were run no names were used. We have conducted independent tests of the name table software. Ninety processes were used, one per processor. Each allocated a 64 Kbyte block of memory, which was enough for 2039 name records. So the global name table had over 180,000 name records. Each process entered 1000 distinct names (90,000 total names) simultaneously. This took about 6 seconds. Each process then looked up all 90,000 names (8,100,000 total look-ups) simultaneously. No two processes were looking up names in the same order. A look-up involved finding the record and retrieving three fields of data, one of which was generated from the name and used to check that the correct data was retrieved. The total look-up time was 409 seconds. This is fast enough so that even parallel

build programs which do a name look-up for each of several million links will still run reasonably fast.

These tests involved very intense switch traffic. They required that the default switch timeout be increased. Surprisingly, the intense switch traffic did not increase the running time significantly. This was determined by changing the test program so that only one process did the 90,000 name look-ups. This took 401 seconds, only 8 seconds less than when there was heavy switch traffic.

We ran a number of tests to measure the run time and speedup of the parallel simulator. The same network could not be used with different numbers of processors because of memory limitations (a small network does not show much improvement when run on a large number of processors because of overhead). We ran each test with the same number of units per processor (1000) and the same number of links per unit (30). With perfect speedup, the simulation time would be the same no matter how many processors were used.

The first user function tested was simple: the potential was set to the unweighted sum of the inputs. The link function simply recorded the value of the input. This represents about the fastest possible run time for a network this big. More complicated unit, site and link functions would slow it down appreciably. After all, the link function is called 30,000 times per step per simulator.

For the simple network running on one processor, the execution time was 2.0 seconds per step. Running on 90 processors, the execution time was 2.6 seconds per step. This works out to about 70 effective processors. A graph of the speed up for these and other configurations is shown in Fig. 7. As expected, when the build

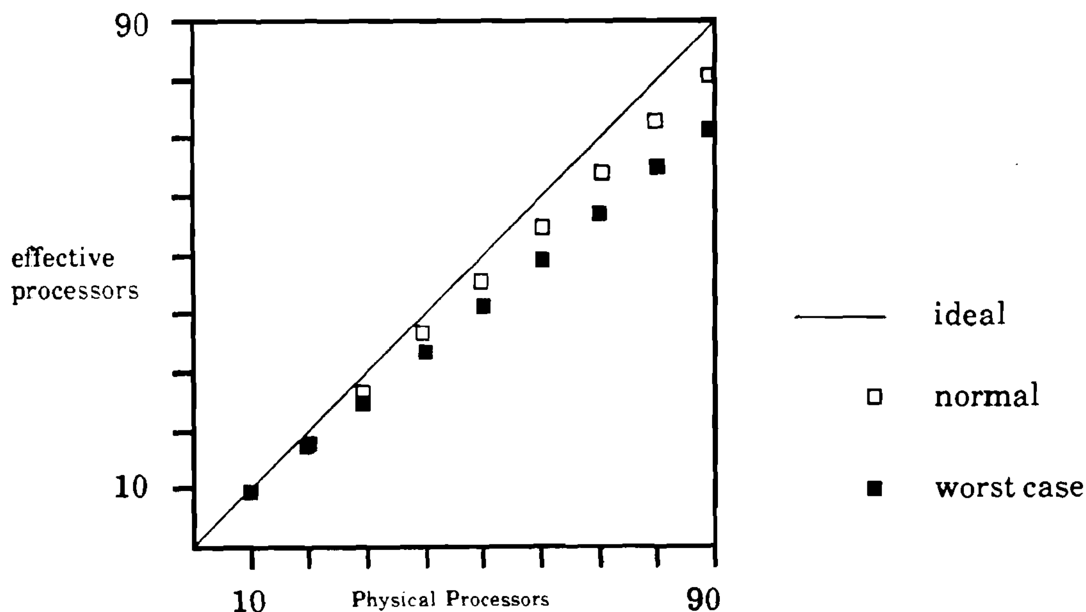


Figure 7.

Slow down due to parallelization overhead.

function was changed so that all links were from other units on the same processor, 90 simulators also took 2.0 seconds per step; there was no slowdown.

In the next series of tests, the inputs were multiplied by the weights and divided by 100. This represents a more typical computation. For one simulator, the time was 4.1 seconds per step; for 90 simulators the time was 4.6 seconds per step. This works out to about 81 effective processors. The speedup is better because a smaller percentage of time was spent retrieving outputs across the switch. The results of this test are also given in Fig. 7. We are very pleased with these figures. We originally expected to spend some effort repartitioning networks or caching remote outputs locally. The actual overhead proved low enough to make this a low priority enhancement.

Most of the simulations run on the Butterfly so far (other than tests) have involved moderately large networks which can fit on uniprocessors, but which run much faster on a partition of the Butterfly. When the turn-around time for a series of simulations is decreased to ten minutes from three hours, the effect on research is significant.

6 Applications

The driving force behind the system develoments described above has been applications of connectionist models, particularly to problems in artificial intelligence. Since all connectionist networks are currently simulated on inappropriate hardware, "applications" refer to concept demonstrations and scientific models, not to programs for practical use. There are many such applications described in this issue and in the literature [McClelland & Rumelhart 1986]. We will focus on some recent Rochester work which illustrates the structured connectionist style and the use of the simulator.

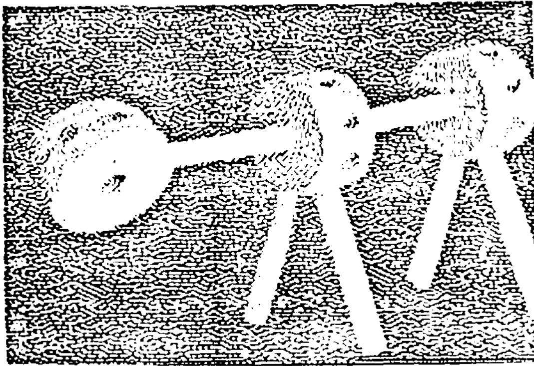
Three examples were presented in earlier sections: the Necker cube (Figs. 1, 3, 4), Cottrell's disambiguator (Fig. 2) and the 2-D shape learning network (Fig. 5). The major example of this section is a rather more ambitious vision project carried out by Paul Cooper and Susan Hollbach [Cooper & Hollbach 1987] with help from Steven Whitehead and others. The basic problem is to match real images of Tinker Toy objects to stored topological models. The low-level vision processing for this is depicted in Fig. 8. Fig. 8a shows a digitized picture of a Tinker Toy horse, the input to the system. Fig. 8b represents the magnitude of the gradient produced by the Kirsch operators (i.e. the edge picture.) Fig. 8c has both straight lines and circles found by the Hough technique from the edge information of 8b. And 8d demonstrates the connectivities found between rods and disks: the small circles represent joints between a rod and disk.

A typical model data base is give in Fig. 9. The idea is to match the analyzed input to all the models in parallel using a structured connectionist network. Fig. 10 shows a model horse, labelled by letters, and a possible input image, labelled by numbers. The matching network tries to match simultaneously compatible wheels (ones with the same number of rods) and to ensure that adjacent wheels in the image match adjacent model wheels. These mutual constraints are adequate to select the correct model for a wide range of tasks. Fig. 11 shows some of the consistency constraint, e.g. if the image pair 3-4 matches the model pair B-C, then 3 must match B or C. The details of the model are moderately complex, involving several winner-take-all networks and simultaneous comparison of all models.

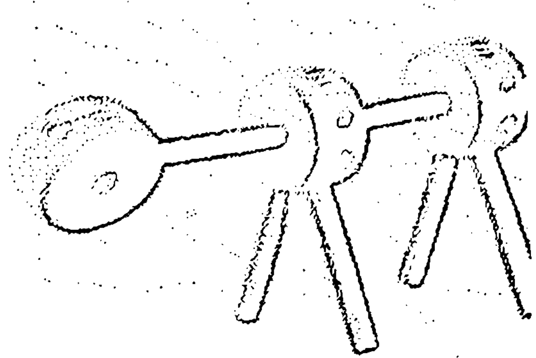
Fig. 9 also depicts the results of matching the results of Fig. 8 with the 21 models. The central goal of recognizing the target 'horse' figure is accomplished. It turns out that reasonable scores are obtained by partial matching of approximately similar figures. Although the network was designed primarily to accept or reject matches of objects with the same number of disks, partial matches and matches of objects with differing numbers of disks are computed also. For example, compare the extracted figure with models 3 and 7 (judged quite similar) and models 4, 6 and 15 (judged fairly similar).

The Necker cube, disambiguation, and Tinker Toy examples are all instances of what are called recognition problems in AI. Several other problems are like this, but many others are not. Can we apply structured connectionist models to other traditional AI issues such as knowledge representation and inference? There is much less research completed along these lines, but some promising starts have been made. The example in Fig. 12 should convey the flavor of this work.

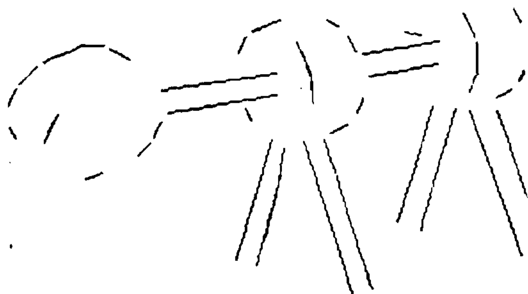
Low Level Processing: An Example



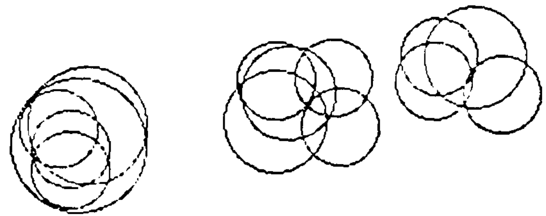
a) Original Figure



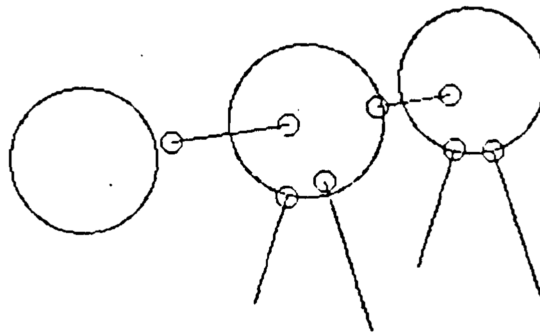
b) Response to Kirsch Operator



c) Extracted Lines



d) Extracted Circles



e) Final Symbolic Representation, including Connections Detected Between Parts

FIGURE 8.

Model Base

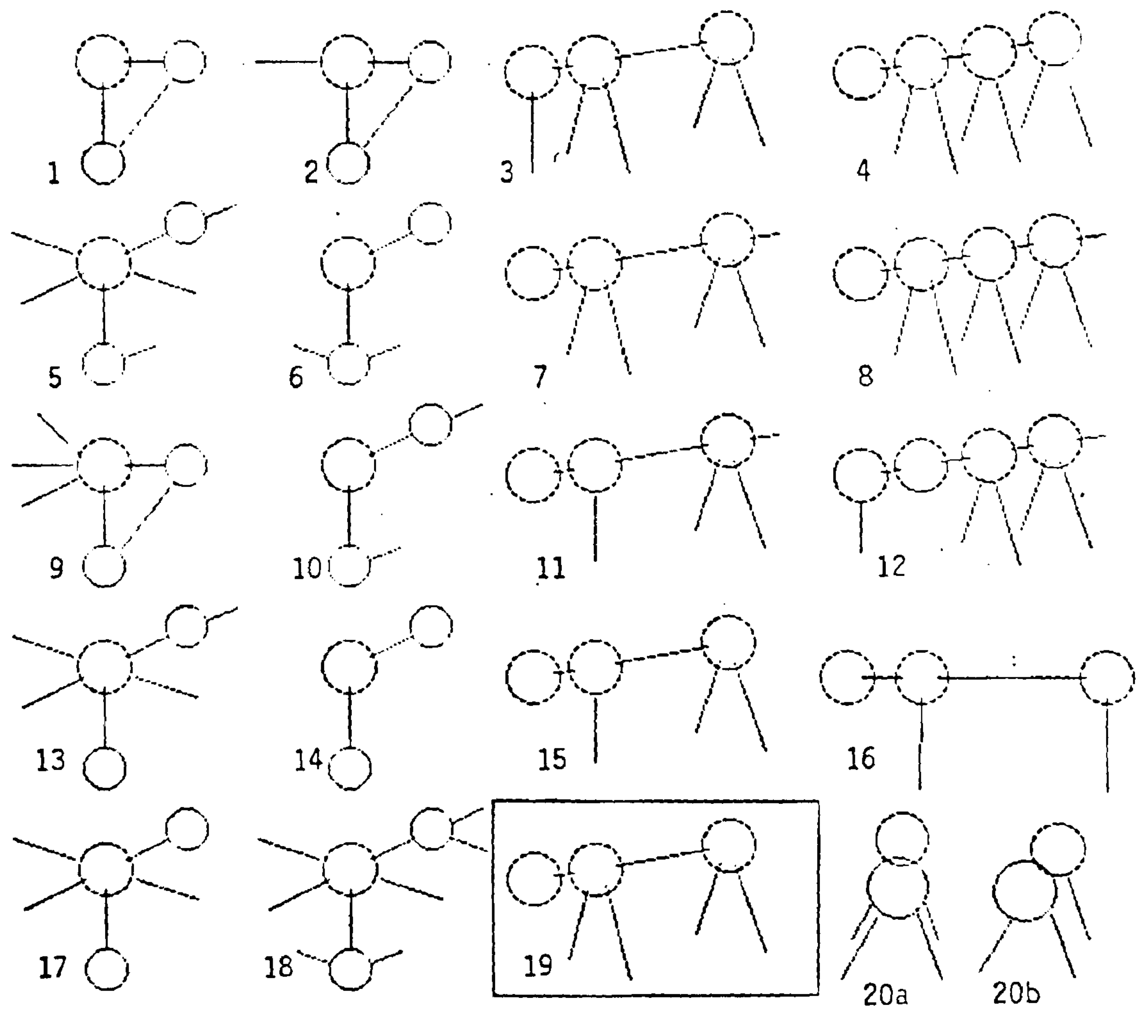


FIGURE 9.

Figure 10: example of a figure and matching model

The standard way to explore the issue of knowledge representation and inference is in terms of programs that can answer questions. There are many AI approaches to the development of question-answering systems but they all have the same basic requirements: One needs a way to store the knowledge, to pose questions, to compute and register the answers. In a connectionist model, all of these aspects must be expressed in terms of activity spreading among simple units like those in the previous examples. Neurons can compute complex functions, but their long-range interactions are simple and complex units do not simplify modelling.

It is easiest to start with the recording of answers. In Fig. 12, the possible tastes of foods form a winner-take-all network where each unit inhibits the others so only one answer will be active. The answer network is assumed to be part of a routine which also poses the questions and acts upon the answer. The units that make up the routine are assumed to be activated in sequence from left to right just like a standard program. A question is sent to the knowledge network by activating the appropriate units; this is shown in Fig. 12 as links from the hexagonal node to the nodes for [has-taste] and [ham]. The key to the operation of this network is the operation of the triangular-shaped nodes, like [b1] in Fig. 12. A unit shown as a triangle is defined to become active when two of its inputs are simultaneously active. In this case, [ham] and [has-taste] are both on, so [b1] becomes active and activates [salty]. Now the [salty] node in the knowledge network spreads activation to the response [r-salty] back in the routine and the question is answered. The same network can answer questions like "Name a salty meat" when activated appropriately. The answers returned by such a network will depend on context as people's answers do, contextual bias is again modeled by activation. This example is taken from the research of Lokendra Shastri [1985].

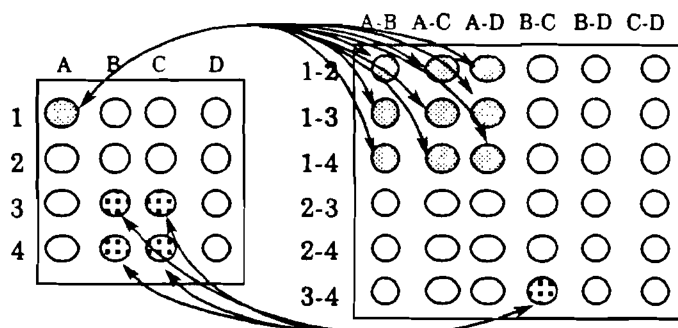


Figure 11: wheel matching array, constraint matching array, and example constraint propagation links

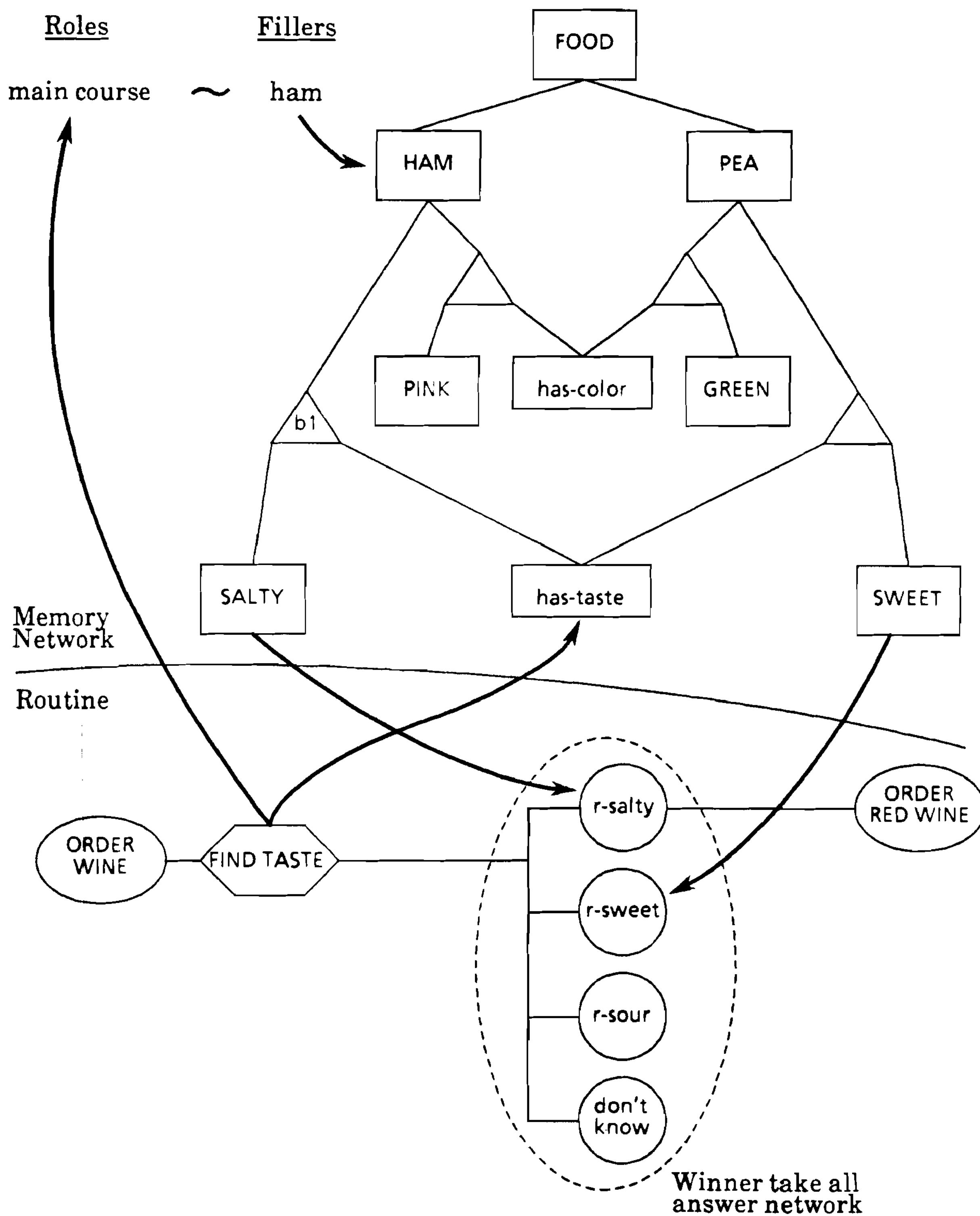


Figure 12: Interaction between a knowledge network and a routine.

In addition, Shastri showed that structured connectionist knowledge representations can handle problems that have proven difficult for logic-based approaches. Suppose we believe that Quakers tend to be pacifists and that Republicans are generally not pacifist. Given an individual who is both a Quaker and Republican, it is hard to decide how likely he is to be a pacifist. (the recent U.S. President who had these two beliefs was also a Marine officer). Shastri's system allows the relative strengths of conflicting beliefs and correlations to be combined according to maximum entropy rules of evidence and performs quite well. Again, the structured connectionist network provides a natural mechanism for representing the required knowledge and for capturing inferences based on partial, uncertain and conflicting knowledge.

Shastri's system was implemented on an earlier but similar version of the simulator (cf. section 3). An interesting aspect of the implementation was how the semantic networks were specified. The detailed construction of nodes and links that will yield correct inferences is quite elaborate and most of Shastri's thesis was concerned with designing and verifying these constructions. But once this was finished, he was able to build a translator that would take declarative input, like that in Table 2, and compile it into a network. The output of the translator was intermediate code in the style of Table 1 and the rest of the construction work proceeded as discussed in section 3.

An obvious question that arises in connection with work like Shastri's is how a network like Fig. 12 could be learned. The neural substrate of memory and learning is one of the great unsolved scientific questions for which we certainly have no definitive answers. But there are connectionist theories of learning that are compatible with current brain research and are computationally feasible [Rumelhart *et al.* 1986]. The key idea is that while new connections are rare, *weight change* in connections appears to be common. We also know that each unit can have thousands of incoming and outgoing connections. Our hypothesis is that most of these connections are only potentially important and that learning involves strengthening the appropriate connections. Suppose, for example, the network of Fig. 12 needed to learn that spinach was a salty vegetable. Our model suggests that there are uncommitted triangular nodes that are weakly connected to many combinations of objects, properties and values. In an ideal case, one of them will be linked to [spinach], [has-taste] and [salty] among other things. This unit will get highly activated by the simultaneous activation of three of its neighbors and, by strengthening its active connections, can become dedicated to the new association. This example omits many important issues; the whole learning theory is in a very primitive stage. A theoretical discussion of this learning model can be found in [Feldman 1982]. Experimental research on recruiting concept representations from a pool of uncommitted units is described by Fanty [1987].

While is it much too early to be certain, it appears that structured connectionist models could lead to major advances in our ability to automate complex tasks, such as those of Artificial Intelligence. The system tools described in the core of this paper have proved very useful in our work and are being made available to the research community. Another way to look at these developments is historically. Fig. 13 cartoons a merging of two different approaches to artificial intelligence that have been evolving separately for decades [Newell 1983]. If we are able to combine the best features of both paradigms, and develop appropriate hardware and software, the scientific and practical consequences could be considerable.

Two Approaches to Artificial Intelligence

Neural networks

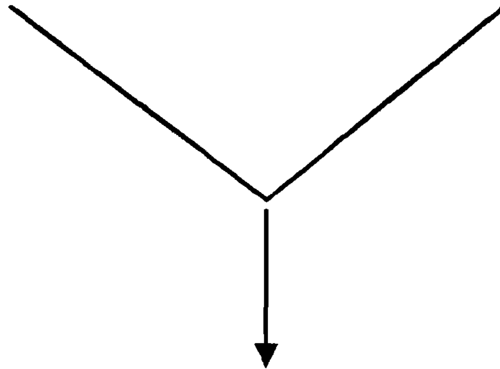
relaxation

adaptation

Algorithms + Data Structures

inference

representation



Structured Connectionist Models

Figure13: Two approaches to artificial intelligence.

Acknowledgements

Many thanks to Liudvikas Bukys for his assistance with the parallel implementation of the connectionist simulator. Susan Hollbach has helped extensively with this presentation. This work was supported in part by NSF/CER grant No. DCR-8320136, in part by ONR/DARPA research contract No. N00014-82-K-0193and in part by ONR research contract No. N00014-84-K-0655. The Xerox Corporation University Grants Program provided equipment used in the preparation of this paper.

References

Amari, S. and M.A. Arbib (eds.), *Competition and Cooperation in Neural Nets*, Vol 45, *Lecture Notes in Biomathematics*, S. Levin (ed.), Berlin: Springer-Verlag, 1982.

Bailey, J. and D. Hammerstrom, "How to make a billion connections," tech. rept. CS/E-86-007, Dept. of Computer Science/Engineering, Oregon Graduate Center, 1986.

Ballard, D.H., "Parameter networks," *Artificial Intelligence*, *22*, 235-267, 1984.

Ballard, D.H., G.E. Hinton and T.J. Sejnowski, "Parallel visual computation," *Nature*, *306*, no. 5938, 21-26, 3 Nov. 1983.

[BBN 1985] Butterfly Parallel Processor Overview. BBN Laboratories, Cambridge, Massachusetts, June 1985.

Blelloch, G.E., "AFL-1: A programming language for massively concurrent computers," MS thesis, MIT, 1986.

Cohen, M.A. and S. Grossberg, "Absolute stability of global pattern fformation and parallel meomory storage by competitive neural networks," *IEEE Transactions: Systems, Man and Cybernetics*, *13*, 815-825, 1983.

Cooper, P.R. and S.C. Hollbach, "Parallel recognition of objects comprised of pure structure," *Proceedings*, DARPA Image Understanding Workshop, Los Angeles, CA, February 1987.

Cottrell, G.W., "A connectionist approach to word sense disambiguation," Ph.D. and TR 154, Computer Science Dept., Univ. Rochester, May 1985.

Cottrell, G.W., "Connectionist parsing," *Proceedings*, 7th Annual Cognitive Science Society Con., Irvine, CA, 1985.

Fanty, M.A. "Learning in structured connectionist networks", forthcoming doctoral dissertation, Computer Science Dept., Univ. Rochester, 1987.

Fanty, M.A. and N. Goddard, "The Rochester connectionist simulator," forthcoming technical report, Computer Science Dept., Univ. Rochester, 1987.

Feldman, J.A., "Neural representation of conceptual knowledge," TR189, Computer Science Dept., Univ. Rochester, June 1986

Feldman, J.A., "Four frames suffice: a provisional model of vision and space," *Behavioral and Brain Sciences, 8*, 265-289, June 1985.

Feldman, J.A., "Dynamic connections in neural networks," *Biological Cybernetics, 46,* 27-39, 1982.

Feldman, J.A. and D.H. Ballard, "Connectionist models and their properties," *Cognitive Science, 6*, 205-254, 1982.

Hinton, G.E. and J.A. Anderson (eds.), *Parallel Models of Associative Memory.* L. Erlbaum Associates, 1981.

Hillis, W.D. and J. Barnes, "Programming a highly parallel computer," *Nature, 326*, 27-30, 5 March 1987.

Hopfield, J.J., "Neural networks and physical systems with emergent collective computational abilities," *Proceedings*, National Academy of Sciences of the Univetd States of America, *79*, 2554-2558, 1982.

McClelland, J.L. and D.E. Rumelhart (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 2: Applications.* MIT Press/Bradford Books, 1986.

Minsky, M. and S. Papert, *Perceptrons.* second edition, MIT PRess, 1972.

Newell, A., "Intellectual ussies in the history of artificial intelligence," in *The Study of Information: Interdisciplinary Messages.* F. Machlup and U. Mansfield (eds.), John Wiley & Sons, Inc., 1983.

Palm, G., "Associative networks and their information storage capacity," Cognitive Systems, *1*, 2, 107-118, June 1985.

Parker, D.B., "Learning-logic," TR-47, Center for Computational Research in Economics and Management science, MIT, April 1985.

Posner, M.I. *Chronometric Explorations of Mind.* L Erlbaum, Associates, 1978.

Rumelhart, D.E., G.E. Hinton & R.J. Williams, "Learning internal representations by error propagation," in: Rumelhart, D.E. & J.L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations.* MIT Press/Bradford Books, 1986.

Rumelhart, D.E. and J.L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations.* MIT Press/Bradford Books, 1986.

Sabbah, D., "Computing with connections in visual recognition of Origami objects," *Cognitive Science, 9*, 25-50, 1985.

Shastri, L., "Evidential resasoning in semantic networks: A formal theory and its parallel implementation," Ph.D. thesis and TR 166, Computer Science Dept., Univ. Rochester, 1985.

Shastri, L. & J.A. Feldman, "Evidential reasoning in semantic networks: A formal theory," *Proceedings*, 9th Int'l. Joint Conf. on Artificial Intelligence, 465-474, Aug. 1985.

Small, S.L., L. Shastri, M.L. Brucks, S. G. Kaufman, G.W. Cottrell and S. Addanki, "ISCON: A network construction aid and simulator for connectionist models," TR109, Computer Science Dept., Univ. Rochester, April 1983.

Sullins, J., "Value cell encoding strategies," TR165, Computer Science Dept., Univ. Rochester, Aug. 1985.

Thompson, R.F., "The neurobiology of learning and memory," *Science*, *233*, 941-947, 29 Aug. 1986.

von der Malsburg, C., "Nervous structures with dynamical links," *Ber. Bunsenges. Phys. Chem.*, *89*, 703-710, 1985.

Waltz, D.L. & J.B. Pollack, "Massively parallel parsing," *Cognitive Science*, *9*, 51-74, 1985.

Zipser, D. & D. Rabin, "P3: A parallel network simulating system," in: Rumelhart, D.E. & J.L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations*. MIT Press/Bradford Books, 1986.