CHAPTER 3

Constraint Satisfaction in PDP Systems

In the previous chapter we showed how PDP networks could be used for content-addressable memory retrieval, for prototype generation, for plausibly making default assignments for missing variables, and for spontaneously generalizing to novel inputs. In fact, these characteristics are reflections of a far more general process that many PDP models are capable of—namely, finding near-optimal solutions to problems with a large set of simultaneous constraints. This chapter introduces this constraint satisfaction process more generally and discusses three different specific models for solving such problems. The specific models are the *schema model,* described in *PDP:14*; the *Boltzmann machine,* described in *PDP:7*; and *harmony theory,* described in *PDP:6*. These models are embodied in the **cs** (constraint satisfaction) program. We begin with a general discussion of constraint satisfaction and some general results. We then turn to the schema model. We describe the general characteristics of the schema model, show how it can be accessed from **cs**, and offer a number of examples of it in operation. This is followed in turn by detailed discussions of the Boltzmann machine and harmony theory.

BACKGROUND

Consider a problem whose solution involves the simultaneous satisfaction of a very large number of constraints. To make the problem more difficult, suppose that there may be no perfect solution in which all of the constraints are completely satisfied. In such a case, the solution would involve the satisfaction of as many constraints as possible. Finally, imagine that

some constraints may be more important than others. In particular, suppose that each constraint has an importance value associated with it and that the solution to the problem involves the simultaneous satisfaction of as many of the most important of these constraints as possible. In general, this is a very difficult problem. It is what Minsky and Papert (1969) have called the *best match* problem. It is a problem that is central to much of cognitive science. It also happens to be one of the kinds of problems that PDP systems solve in a very natural way. Many of the chapters in the two PDP volumes pointed to the importance of this problem and to the kinds of solutions offered by PDP systems.

To our knowledge, Hinton was the first to sketch the basic idea for using parallel networks to solve constraint satisfaction problems (Hinton, 1977). Basically, such problems are translated into the language of PDP by assuming that each unit represents a hypothesis and each connection a constraint among hypotheses. Thus, for example, if whenever hypothesis A is true, hypothesis B is usually true, we would have a positive connection from unit A to unit B. If, on the other hand, hypothesis A provides evidence against hypothesis B, we would have a negative connection from unit A to unit B.

PDP constraint networks are designed to deal with *weak constraints* (Blake, 1983), that is, with situations in which constraints constitute a set of desiderata that *ought* to be satisfied rather than a set of *hard* constraints that *must* be satisfied. The goal is to find a solution in which as many of the most important constraints are satisfied as possible. The importance of the constraint is reflected by the strength of the connection representing that constraint. If the constraint is very important, the weights are large. Less important constraints involve smaller weights. In addition, units may receive external input. We can think of the external input as providing direct evidence for certain hypotheses. Sometimes we say the input "clamps" a unit. This means that, in the solution, this particular unit *must be on* if the input is positive or *must be off* if the input is negative. Other times the input is not clamped but is graded. In this case, the input behaves as simply another weak constraint. Finally, different hypotheses may have different a priori probabilities. An appropriate solution to a constraint satisfaction problem must be able to reflect such prior information as well. This is done in PDP systems by assuming that each unit has a *bias,* which acts to turn the unit on in the absence of other evidence. If a particular unit has a positive bias, then it is better to have the unit on; if it has a negative bias, there is a preference for it to be turned off.

We can now cast the constraint satisfaction problem described above in the following way. Let *goodness of fit* be the degree to which the desired constraints are satisfied. Thus, goodness of fit (or more simply *goodness*) depends on three things. First, it depends on the extent to which each unit satisfies the constraints imposed upon it by other units. Thus, if a connection between two units is positive, we say that the constraint is satisfied to the degree that both units are turned on. If the connection is negative, we can say that the constraint is violated to the degree that both units are

turned on. A simple way of expressing this is to let the product of the activation of two units times the weight connecting them be the degree to which the constraint is satisfied. That is, for units i and j we let the product $w_{ij}a_ia_j$ represent the degree to which the pairwise constraint between those two hypotheses is satisfied. Note that for positive weights the more the two units are on, the better the constraint is satisfied; whereas for negative weights the more the two units are on, the less the constraint is satisfied. Second, the a priori strength of the hypothesis is captured by adding the bias to the goodness measure. Finally, the goodness of fit for a hypothesis when direct evidence is available is given by the product of the input value times the activation value of the unit. The bigger this product, the better the system is satisfying this external constraint. Given this, we can now characterize mathematically the degree to which a particular unit is satisfying all of the constraints impinging on it. Thus the overall degree to which the state of a particular unit, say unit i, contributes to the overall goodness of fit can be obtained by adding up the degree to which the unit satisfies all of the constraints in which it is involved, from all three sources. Thus, we can define the goodness of fit of unit i to be

$$goodness_i = \sum_j w_{ij}a_ia_j + input_ia_i + bias_ia_i. \tag{1}$$

This, of course, is just the sum of all of the individual constraints in which the corresponding hypothesis participates. It is not the individual hypothesis, however, that is the problem in constraint satisfaction problems. In these cases, we are concerned with the degree to which the entire pattern of values assigned to all of the units are consistent with the entire body of constraints. This overall goodness of fit is the function we want to maximize. We can define our overall goodness of fit as the sum of the individual goodnesses. In this case we get

$$goodness = \sum_{i,j} w_{ij}a_ia_j + \sum_i input_ia_i + \sum_i bias_ia_i. \tag{2}$$

We have solved the problem when we have found a set of activation values that maximizes this function. It should be noted that since we want to have the activation values of the units represent the degree to which a particular hypothesis is satisfied, we want our activation values to range between a minimum and maximum value—in which the maximum value is understood to mean that the hypothesis should be accepted and the minimum value means that it should be rejected. Intermediate values correspond to intermediate states of certainty.

We have now reduced the constraint satisfaction problem to the problem of maximizing the goodness function given above. There are many methods of finding the maxima of functions. Importantly, there is one method that is naturally and simply implemented in a class of PDP networks. One restriction on this class of networks is the restriction that the

weights in the network be symmetric: that is, the condition that $w_{ij} = w_{ji}$. Under these conditions it is easy to see how a PDP network naturally sets activation values so as to maximize the goodness function stated above. To see this, first notice that the goodness of a particular unit, $goodness_i$, can be written as the product of its current net input times its activation value. That is,

$$goodness_i = net_i a_i \tag{3}$$

where, as usual for PDP networks, net_i is defined as

$$net_i = \sum_j w_{ij} a_j + input_i + bias_i. \tag{4}$$

Thus, the net input into a unit provides the unit with information as to its contribution to the goodness of the entire solution. Consider any particular unit in the network. That unit can always behave so as to increase its contribution to the overall goodness of fit if, whenever its net input is positive, the unit moves its activation toward its maximum activation value, and whenever its net input is negative, it moves its activation toward its minimum value. Moreover, since the global goodness of fit is simply the sum of the individual goodnesses, a whole network of units behaving in such a way will always increase the global goodness measure. These observations were made by Hopfield (1982). We will return to Hopfield's important contribution to this analysis again in our discussion of Boltzmann machines and harmony theory.

It might be noted that there is a slight problem here. Consider the case in which two units are *simultaneously* evaluating their net inputs. Suppose that both units are off and that there is a large negative weight between them; suppose further that each unit has a small positive net input. In this case, both units may turn on, but since they are connected by a negative connection, as soon as they are both on the overall goodness may decline. In this case, the next time these units get a chance to update they will both go off and this cycle can continue. There are basically two solutions to this. The standard solution is not to allow more than one unit to update at a time. In this case, one or the other of the units will come on and prevent the other from coming on. This is the case of so-called *asynchronous* update. The other solution is to use a *synchronous* update rule but to have units increase their activation values very slowly so they can "feel" each other coming on and achieve an appropriate balance.

In practice, goodness values generally do not increase indefinitely. Since units can reach maximal or minimal values of activation, they cannot continue to increase their activation values after some point so they cannot continue to increase the overall goodness of the state. Rather, they increase it until they reach their own maximum or minimum activation values. Thereafter, each unit behaves so as to never decrease the overall goodness. In this way, the global goodness measure continues to increase until all

units achieve their maximally extreme value or until their net input becomes exactly 0. When this is achieved, the system will stop changing and will have found a maximum in the goodness function and therefore a solution to our constraint satisfaction problem. When it reaches this peak in the goodness function, the goodness can no longer change and the network is said to have reached a *stable state;* we say it has *settled* or *relaxed* to a solution. Strictly speaking, this solution state can be guaranteed only to be a *local* rather than a *global* maximum in the goodness function. That is, this is a *hill-climbing* procedure that simply ensures that the system will find a peak in the goodness function, not that it will find the highest peak. The problem of local maxima is difficult. We address it at length in a later section. Suffice it to say, that different PDP systems differ in the difficulty they have with this problem.

The development thus far applies to all three of the models under discussion in this chapter. It can also be noted that if the weight matrix in an IAC network is symmetric, it too is an example of a constraint satisfaction system. Clearly, there is a close relation between constraint satisfaction systems and content-addressable memories.

We turn, at this point, to a discussion of the specific models and some examples with each. We begin with the schema model of *PDP:14.*

THE SCHEMA MODEL

The schema model is one of the simplest of the constraint satisfaction models, but, nevertheless, it offers useful insights into the operation of all of the constraint satisfaction models. In *PDP:2* we described a set of characteristics required to specify any model's particular features. The three models under discussion differ from one another primarily as to whether the units behave deterministically or stochastically (probabilistically), whether the units take on a continuum of values or only binary values, and by the allowable set of connections among the units. The schema model is deterministic; its units can take on any value between 0 and 1. The connection matrix is symmetric and the units may not connect to themselves (i.e., $w_{ij} = w_{ji}$ and $w_{ii} = 0$). Update in the schema model is asynchronous. That is, units are chosen to be updated sequentially in random order. When chosen, the net input to the unit is computed and the activation of the unit is modified. The logic of the hill-climbing method implies that whenever the net input (net_i) is positive we must increase the activation value of the unit, and when it is negative we must decrease the activation value. Thus we use the following simple update rule:

$$a_i(t+1) = a_i(t) + net_i(1 - a_i(t)) \tag{5}$$

when net_i is greater than 0, and

$$a_i(t+1) = a_i(t) + net_i a_i(t) \tag{6}$$

when net_i is less than 0. Note that in this second case, since net_i is negative and a_i is positive, we are decreasing the activation of the unit. This rule has two virtues: it conforms to the requirements of our goodness function and it naturally constrains the activations between 0 and 1.

As usual in these models, the net input comes from three sources: a unit's neighbors, its bias, and its external inputs. These sources are added. Thus, we have

$$net_i = istr\left(\sum_j w_{ij} a_j + bias_i\right) + estr(input_i). \tag{7}$$

Here the constants *istr* and *estr* are parameters that allow the relative contributions of the input from external sources and that from internal sources to be readily manipulated.

IMPLEMENTATION

The **cs** program implementing the schema model is much like **iac** in structure. It differs in that it does *asynchronous* updates using a slightly different activation rule. Like **iac**, **cs** consists of essentially two routines: (a) an update routine called *rupdate* (for random update), which selects units at random and computes their net inputs and then their new activation values, and (b) a control routine, *cycle,* which calls *rupdate* in a loop for the specified number of cycles while displaying the results on the screen. Thus, *cycle* is as follows:

```
cycle() {
  for(i = 0; i < ncycles; i++) {
    cycleno++;
    rupdate();
    update_display();
  }
}
```

Thus, each time *cycle* is called, the system calls *rupdate* and then displays the results of the computation. The *rupdate* routine itself does all of the work. It randomly selects a unit, computes its net input, and assigns the new activation value to the unit. It does this *nupdates* times. Typically, *nupdates* is set equal to *nunits*, so a single call to *rupdate*, on average, updates each unit once:

```
rupdate() {

  for (updateno = 0; updateno < nupdates; updateno++) {
    i = randint(0, nunits - 1);
    netinput = 0;
    for(j = 0; j < nunits; j++) {
      netinput += activation[j]*weight[i][j];
    }
    netinput += bias[i];
    netinput *= istr;
    netinput += estrength*input[i];

    if (netinput > 0)
      activation[i] += netinput*(1-activation[i]);
    else
      activation[i] += netinput*activation[i];
  }
}
```

RUNNING THE PROGRAM

The basic structure of **cs** and the mechanics of interacting with it are identical to those of **iac**. The **cs** program, like all of our programs, requires a template (*.tem*) file that specifies what is displayed on the screen and a start-up (*.str*) file that initializes the program with the parameters of the particular program under consideration. It also requires a *.net* file specifying the particular network under consideration, and may use a *.wts* file to specify particular values for the weights. It also allows for a *.pat* file for specifying a set of patterns that can be presented to the network. Once you are in the appropriate directory, the program is accessed by entering the command:

cs xxx.tem xxx.str

where *xxx* is the name of the particular example you are running.

The normal sequence for running the model involves applying external inputs to some subset of the units by use of the *input* command and using the *cycle* command to cause the network to cycle until it finds a goodness maximum. Typically, the value of the goodness is displayed after each cycle, and the system will cycle *ncycles* times and then stop. If the system has not yet reached a stable state, it can be continued from where it left off if the user simply enters *cycle* again. The system can be interrupted at any time by typing ^C (control-C).

Two commands are available for restarting the system. Both commands set *cycleno* back to 0, and both reinitialize the activations of all of the units.

However, one of these commands, *newstart*, causes the program to follow a new random updating sequence when next the *cycle* command is given, whereas the other command, *reset*, causes the program to repeat the same updating sequence used in the previous run. Alternatively, the user can specify a particular value for the random seed and enter it manually via the *set/ seed* command; when *reset* is next called, this value of the seed will be used, producing results identical to those produced on other runs begun with this same *seed* in force.

The cs program implements both the Boltzmann model and harmony theory in addition to the schema model. In this section we will introduce those aspects of cs relevant to all three models, even though some of these will not be explained until later in the chapter.

New or Altered Commands

newstart

Generates a new random seed for the random number generator and then issues a reset command. The effect is to cause the net to follow a new random sequence of updates when *cycle* is subsequently entered.

reset

Resets the network back to its initial state. In *clamp* mode, units with positive external input are clamped *on* and units with negative external inputs are clamped *off*. All others have their activations set to 0. The cycle number is reset to 0, and the random number generator is seeded with the value of the *seed* variable. Unless the *seed* has been changed, either by the *set/ seed* command or by calling *reset* via *newstart*, this means that the program will go on to repeat the same random sequence that was generated after the last *reset*.

get/ annealing

Allows the user to specify an annealing schedule for use in *boltzmann* or *harmony* mode (these modes are discussed later in the chapter). It begins by prompting for an initial temperature. The annealing schedule begins at this temperature. It then prompts for a sequence of time-temperature pairs. A carriage return or the string *end* given in response to the prompt will terminate the entry of the schedule. The system linearly interpolates from the initial temperature so that at the time (measured in number of cycles) specified in the first pair, the temperature will reach the value specified for that time. It then linearly interpolates from that temperature to the next temperature at the next time. This continues until the final time is reached. Thereafter the temperature remains constant at the final value.

Variables

The following variables are new or somewhat different in the **cs** program. They are accessed by way of the *set* and *exam* commands as in **iac**.

bias

A vector that contains the values of the biases for each of the units.

nupdates

Determines the number of updates per cycle. Generally, it is initialized in the *.net* file to be equal to *nunits*, so that each unit will be updated once per cycle, on the average.

seed

The current value of the seed used for reinitializing the random number generator. The *seed* is set to a random starting value when the program is first called, and this value is used to initialize the random number generator. Calls to *reset* cause the random number generator to be reinitialized to the current value of *seed*, and calls to *newstart* cause *seed* itself to be set to a new random value *before* resetting. The *seed* may be set to any desired integer value using the *set/ seed* command. Unless manually changed, the value of *seed* will be the value used last time the random number generator was reinitialized and it can be used again later to repeat the same sequence.

sigma

A vector that contains the values of the importance parameters associated with knowledge atoms in *harmony theory*.

stepsize

Determines exactly when information about the state of the program is displayed to the screen. If *stepsize* is set to *cycle* (the default value), the information will be displayed on the screen after every cycle. If the *stepsize* is set to *update*, information will be displayed on the screen after every time a unit is updated. If *single* is set to 1, the program will pause after every screen update.

mode/ boltzmann

If *boltzmann* is set to 1, the program will behave as a Boltzmann machine. If it is set to 0 it will act as the schema model. The default value is 0.

mode/ clamp

If *clamp* is set to 1, positive external inputs supplied via the *input* and *test* commands cause the units receiving them to come on and stay on, and negative inputs cause the units to go off and stay off. Zero inputs have no effect on the units. If *clamp* is set to 0, external inputs are graded and act as additional weak constraints on their units; they are simply added into the net input of the unit. The default is that *clamp* is set to 0.

mode/ harmony
: If *harmony* is set to 1, the program will behave as a *harmonium*, as defined in *PDP:6*. The default is that this is set to 0.

param/ istr
: Scales the magnitude of internal input to each unit.

param/ kappa
: This parameter is relevant in *harmony* mode. It is basically a global threshold that indicates the fraction of the inputs to a knowledge atom that must "agree with" that knowledge atom before that atom will tend to come on. See the discussion of harmony theory for details.

state/ cuname
: The name of the current unit (the one most recently updated).

state/ goodness
: Contains the current value of the goodness for the entire network.

state/ temperature
: A variable relevant to the Boltzmann machine and harmony theory, as will be explained in those sections. It is normally adjusted in accordance with an annealing schedule.

state/ unitno
: The number of the unit last updated.

state/ updateno
: Tells which update is currently being done, counting from the beginning of the current cycle.

OVERVIEW OF EXERCISES

We offer four exercises to try with the schema model. In Ex. 3.1, we give you the chance to explore the basic properties of this constraint satisfaction system, using the Necker cube example in *PDP:14* (originally from Feldman, 1981). Ex. 3.2 considers how the schema model deals with knowledge that has traditionally been taken as evidence for schemata, using the room example in *PDP:14*. Exs. 3.3 and 3.4 are more complex projects: Ex. 3.3 suggests you try the tic-tac-toe example in *PDP:14* and Ex. 3.4 is even more open ended. In Appendix E we provide answers for the questions in Exs. 3.1 and 3.2.

Ex. 3.1. The Necker Cube

Feldman (1981) has provided a clear example of a constraint satisfaction problem well-suited to a PDP implementation. That is, he has shown how a simple constraint satisfaction model can capture the fact that there are exactly two good interpretations of a Necker cube. In *PDP:14* (pp. 8-17), we describe a variant of the Feldman example relevant to this exercise. In

this example we assume that we have a 16-unit network (as illustrated in Figure 1). Each unit in the network represents a hypothesis about the correct interpretation of a vertex of a Necker cube. For example, the unit in the lower left-hand part of the network represents the hypothesis that the lower left-hand vertex of the drawing is a front-lower-left (*fll*) vertex. The upper right-hand unit of the network represents the hypothesis that the upper right-hand vertex of the Necker cube represents a front-upper-right (*fur*) vertex. Note that these two interpretations are inconsistent in that we do not normally see both of those vertices as being in the frontal plane. The Necker cube has eight vertices, each of which has two possible interpretations—one corresponding to each of the two interpretations of the cube. Thus, we have a total of 16 units. Three kinds of constraints are represented in the network. First, since each vertex can have only one interpretation, we have a negative connection between units representing alternative interpretations of the same vertex. Second, since the same interpretation cannot be given to more than one vertex, units representing the same interpretation are mutually inhibiting. Finally, units that represent locally consistent interpretations should be mutually exciting. Thus, there are positive connections between a unit and its three consistent neighbors. The system will achieve maximal goodness when all units representing one consistent interpretation of the Necker cube are turned on

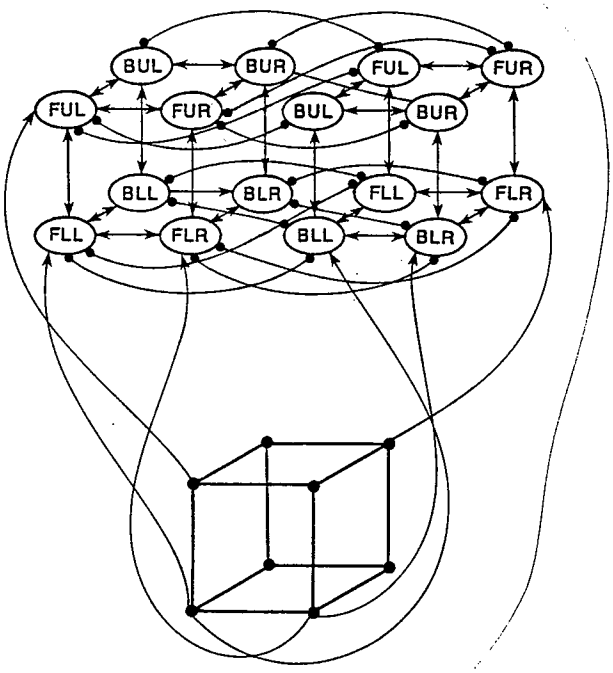

FIGURE 1. A simple network representing some of the constraints involved in perceiving the Necker cube. (From *PDP:14*, p. 10.)

and those representing the other interpretation are turned off. In the diagram, the two subsets of units are segregated so that we expect that either the eight units on the left will come on with the others turned off or the eight units on the right will come on.

Using the *cube.tem* and *cube.str* files, explore the Necker cube example described in *PDP:14*. Run the program several times and look at the obtained interpretations. Record the distribution of interpretations.

Start up the **cs** program with the *cube* template and start-up files:

cs cube.tem cube.str

At this point the screen should look like the one shown in Figure 2. The display depicts the two interpretations of the cube and shows the activation values of the units, the current cycle number, the current update number, the name of the most recently updated unit (there is none yet so this is blank), the current value of goodness, and the current temperature. (The temperature is irrelevant for this exercise, but will become important later.) The activation values of all 16 units are shown, initialized to 0, at the corners of the two cubes drawn on the screen. The units in the cube on the left, cube A, are the ones consistent with the interpretation that the cube is facing down and to the left. Those in the cube on the right, cube B, are the ones consistent with the interpretation of the cube as facing up and to the right. The dashed lines do not correspond to the connections among the units, but simply indicate the interpretations of the units. The connections are those shown in the Necker cube network in Figure 1. The vertices are labeled, and the labels on the screen correspond to those in Figure 1. All units have names. Their names are given by a capital letter

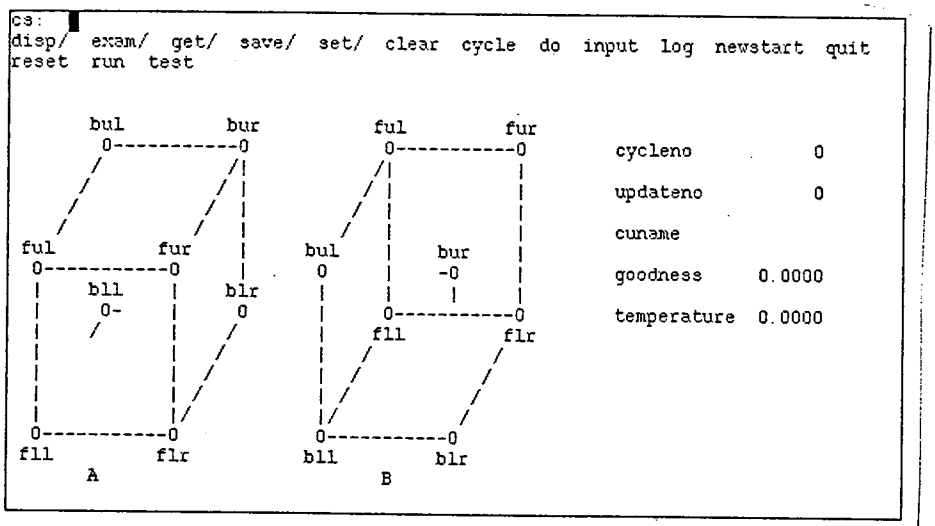

FIGURE 2. The initial image of the screen for the cube example.

indicating which interpretation is involved (A or B), followed by the label appropriate to the associated vertex. Thus, the unit displayed at the lower left vertex of cube A is named *Afll*, the one directly above it is named *Aful* (for the front-upper-left vertex of cube A), and so on. You can use these names to examine the activation values and connections among the units. Thus, for example, it is possible to examine the connection between the unit *Aflr* (the unit representing the hypothesis that the lower right-hand vertex of the Necker cube is in the frontal plane—interpretation A) and the unit *Bblr* (the unit representing the hypothesis that the lower right-hand vertex of the Necker cube is in the back plane—interpretation B) by giving the names *Aflr* and *Bblr* when examining weights. The weights between these two units should be −1.5. (This is reasonable since these two units represent alternative interpretations of the same vertex and so should be inhibitory.)

We are now ready to begin exploring the cube example. The biases and connections among the units have already been read into the program (they were specified in the *cube.net* file, read in by the *get/ network* command in the *cube.str* file). In this example, all units have positive biases, therefore there is no need to specify inputs. Simply type *cycle*. After the command is typed, the display will flash, and various numbers representing the activation values of the corresponding units will replace the 0s at the corners of the cubes. Only single digits are displayed. The numbers are the tenths digit of the activation levels, so that a 4 in the display indicates that the corresponding unit's activation is between 0.4 and 0.5. When the activation values reach 1.0 (their maximum value), an asterisk is plotted. After the display stops flashing you should see that the variables on the right have attained some new values, and you should have a display roughly like that in Figure 3. The variable *cycle* should be 20, indicating that the program has completed 20 cycles. The variable *update* should be at 16, indicating that we have completed the 16th update of the cycle. The *uname* will indicate the last unit updated. The *goodness* may have a value of 6.4. If it does, the network has reached a *global* maximum and has found one of the two "standard" interpretations of the cube. In this case you should find that the activation values of those units in one subnetwork have all reached their maximum value (indicated by asterisks) and those in the other subnetwork are all at 0. If the goodness value is less than 6.4, then the system has found a *local maximum* and there will be nonzero activation values in both subnetworks. You can run the cube example again by issuing the *newstart* command and then entering *cycle*. Do this, say, 20 times to get a feeling for the distribution of final states reached.

Q.3.1.1. How many times was each of the two valid interpretations found? How many times did the system settle into a local maximum? What were the local maxima the system found? Do they correspond to reasonable interpretations of the cube?

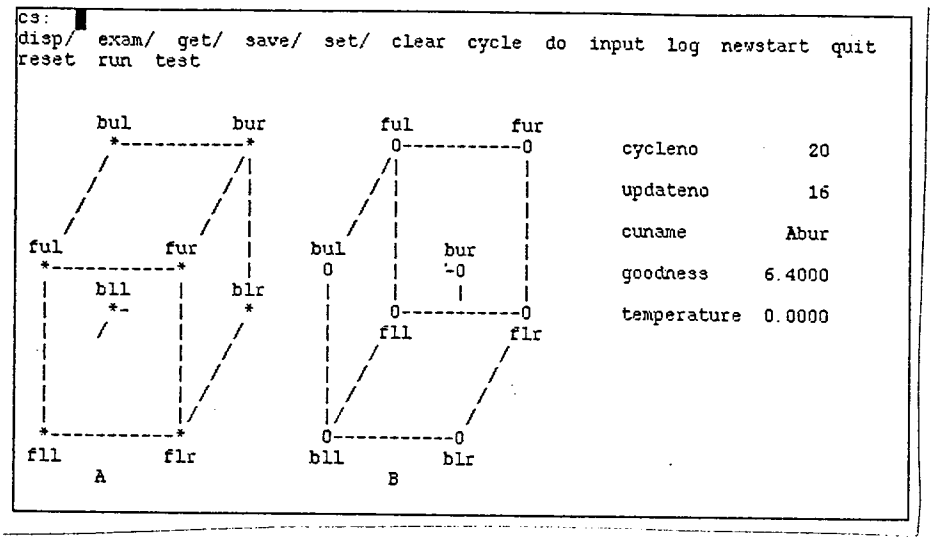

FIGURE 3. The state of the system after 20 cycles.

Now that you have a feeling for the range of final states that the system can reach, try to see if you can understand the course of processing leading up to the final state.

Q.3.1.2. What causes the system to reach one interpretation or the other? How early in the processing cycle does the eventual interpretation become clear? What happens when the system reaches a local maximum? Is there a characteristic of the early stages of processing that leads the system to move toward a local maximum?

Hints. The movement on the screen can be rapid, and it may be difficult to see exactly what is happening. It is sometimes useful to set *single* to 1 and to set *stepsize* to *update*. Under these conditions, the program refreshes the display and pauses after each update. Note also that if you wish to study the evolution of the system toward a particular end state, you can issue the *newstart* command repeatedly, followed by *cycle*, with *single* set to 0, until the system settles to the desired end state, and then use *reset* to repeat the identical run, perhaps first setting *single* to 1 and *stepsize* to *update*.

Q.3.1.3. There is a parameter in the schema model, *istr,* that multiplies the weights and biases and that, in effect, determines the rate of activation flow within the model. The probability of finding a local maximum depends on the value of this parameter. How does the relative frequency of local maxima vary as this parameter is varied? Try several values from 0.08 to 2.0. Explain the results you obtain.

Hints. You will probably find that at low values of *istr* you will want to increase *ncycles*. Alternatively, you can just issue the *cycle* command a second time if the network doesn't settle in the first 20 cycles. You may also want to set *single* back to 0 and *stepsize* back to *cycle*. Do not be disturbed by the fact that the values of *goodness* are different here than in the previous runs. Since *istr* multiplies the weights, it also multiplies the goodness so that *goodness* is proportional to *istr*.

Reset *istr* to its initial value of 0.4 before proceeding to the next question.

Q.3.1.4. It is possible to use external inputs to bias the network in favor of one of the two interpretations. Study the effects of adding an input of 1.0 to the units in one of the subnetworks, using the *input* command. Look at how the distribution of interpretations changes as a result of the number of units receiving external input in a particular subnetwork.

Ex. 3.2. Schemata for Rooms

Interestingly, a simple constraint satisfaction network of the type we have just been describing leads to an interesting interpretation of what a *schema* may be like and how schemata may be implemented in PDP networks. This idea was described in some detail in *PDP:14*. Here we offer a brief summary. The basic idea is that our knowledge is in the form of a constraint satisfaction network. Conventionally (cf. Rumelhart & Ortony, 1977), a schema is a higher-level conceptual structure for representing the complex relationships implicit in our knowledge base. Schemata are data structures for representing generic concepts stored in memory. They are like models of the outside world. Information is processed by first finding the schema that best fits the current situation and then using that model to *fill in* aspects of the situation not specified by the current input. In general, a consistent configuration of such models (or schemata) is discovered that constitutes an interpretation of the situation in question. Within the PDP framework there is no explicit unit or other representational entity corresponding to a schema. Rather, schemata are implicit in our knowledge and arise, while processing information, from the interactions of a large set of constraints. Thus, the units of such a constraint network correspond to hypotheses that certain semantic features are appropriate descriptions of the situation in question. Some of these features are available in the input and form the starting place of the interpretation process. Others are unspecified and must be *filled in* during the process of interpretation. The final state achieved by the network corresponds to the interpretation. Thus,