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

$$goodness = \sum w_{ij}a_ia_j + \sum input_ia_i + \sum_i bias_ia_i. \tag{2}$$

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

Note : In Equation 2, each pair of units contributes only one time to the first summation (w_{ij} is assumed equal to w_{ji}). The total goodness is not the sum of the goodnesses of the individual units. It is best to think of the goodness of unit i as the set of terms in the total goodness to which the activation of unit i contributes.

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

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

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

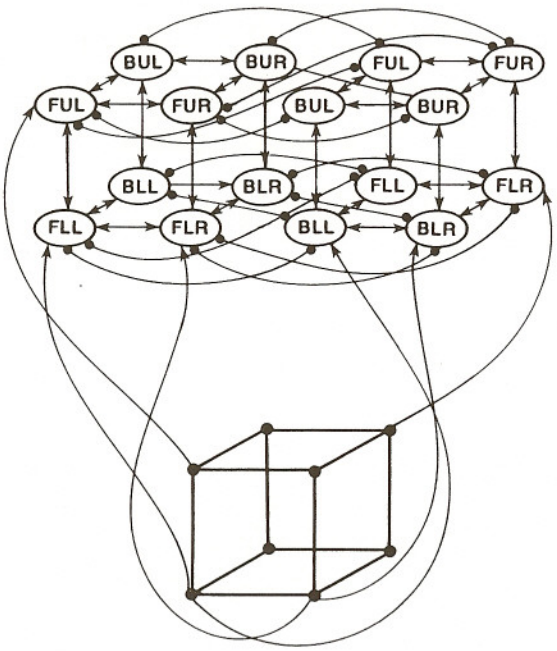

FIGURE 1. A simple network representing some of the constraints involved in perceiving the Necker cube. (From *PDP:14*, p. 10.)

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

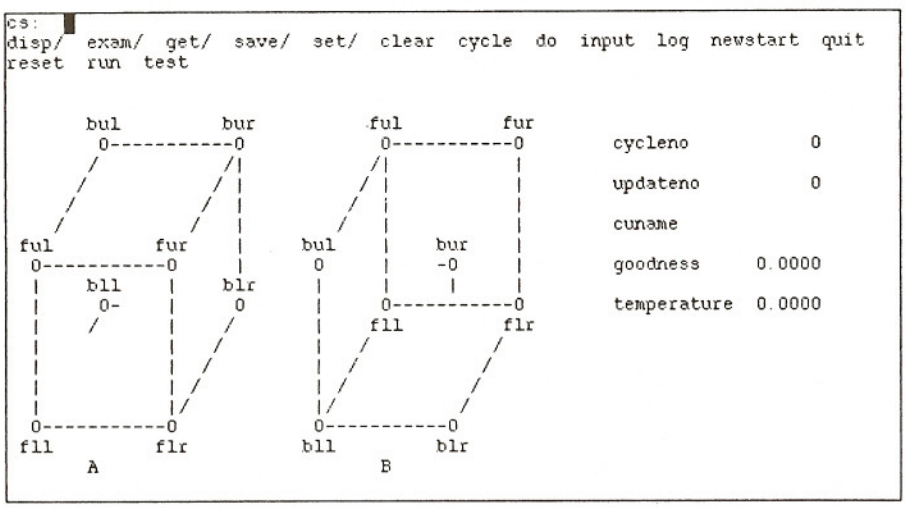

FIGURE 2. The initial image of the screen for the cube example.

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?

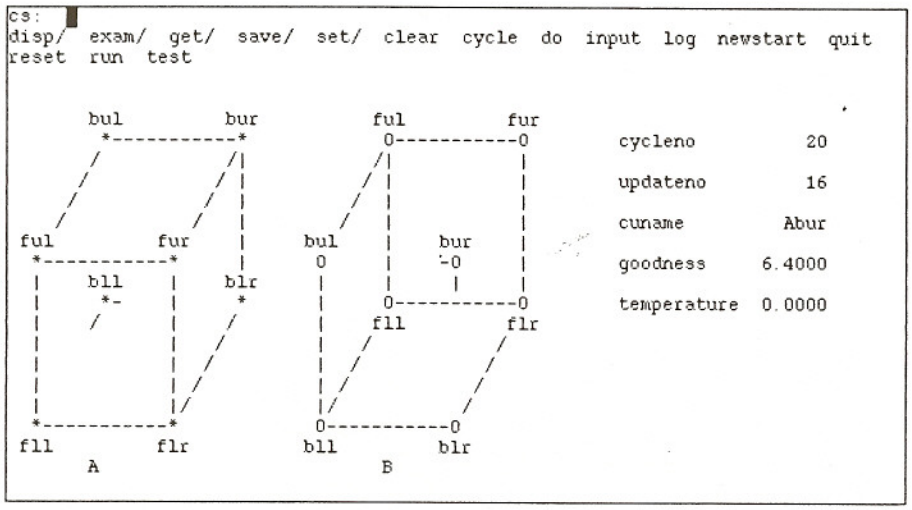

FIGURE 3. The state of the system after 20 cycles.

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,

interpretations correspond to local maxima defined over the "goodness" landscape. The constraint satisfaction network captures the two most basic characteristics of schemata: It has a goodness of fit measure and finds the interpretation that is maximally consistent with both the external constraints and the internal knowledge. It automatically completes the pattern and thereby fills in unspecified variables in the situation with their default values.

The primary example in *PDP:14* was an illustration of how a network constructed according to these simple principles could be a constraint network that behaved as if it contained schemata for five different kinds of rooms—for living rooms, kitchens, bedrooms, offices, and bathrooms. The units in this case stood for the hypotheses that a particular room in question contained a typewriter, coffee cup, sofa, bed, and so on. In all, 40 such features were considered. These features are shown in Table 1. The connection strengths themselves were derived from the co-occurrences of the particular characteristics generated from a simple experiment in which a subject imagined rooms of different types and then judged for all 40 features whether or not they were true of the rooms she was imagining. If two features generally occurred together, the connection between them was strong; if two features occurred separately, the connections between them were negative. The details are described in *PDP:14*.

To begin exploring the issues, use the *room.tem* and *room.str* files to replicate the basic five prototypes illustrated on pages 26 and 27 of *PDP:14*. As in the text, use *oven, desk, bathtub, sofa,* and *bed* as the seeds for the prototypes.

We can find the prototype kitchen, for example, by clamping the *oven* unit and letting the system settle to a solution, filling in the unspecified features.[1] You can use the *input* command to set the value of *oven* to 1 and thereby clamp it on. Enter *cycle*; this will cause 50 cycles to be run, since

TABLE 1

THE 40 ROOM DESCRIPTORS

ceiling	walls	door	windows	very-large
large	medium	small	very-small	desk
telephone	bed	typewriter	bookshelf	carpet
books	desk-chair	clock	picture	floor-lamp
sofa	easy-chair	coffee-cup	ashtray	fireplace
drapes	stove	coffeepot	refrigerator	toaster
cupboard	sink	dresser	television	bathtub
toilet	scale	oven	computer	clothes-hanger

(From *PDP:14*, p. 22.)

[1] In *PDP:14*, we clamped on the *ceiling* unit as well as one other unit. Here, we have set the biases on the *ceiling* and *wall* units that essentially force these units to come on, saving you the trouble of clamping one of them in the input command.

ncycles is set to 50. At this point, the system should almost always reach the state shown in Figure 4, with a goodness of about 21.2. The screen display gives the names of each of the units in the network. The external input to each unit is indicated to the left of the unit's name; its activation is indicated to the right. External inputs of +1.0 are indicated by two asterisks; inputs of −1.0 are indicated by two asterisks in reverse video. (Note that in this example, unlike the Necker cube example, the *clamp* variable is set to 1. Thus, units whose inputs are set to a positive number will be forced to stay on.) Activations and other values of external input are indicated in hundredths, so, for example, the activation of the *drapes* unit is 0.99 and the activation of the *oven* unit is 1.0.

Q.3.2.1. Does the system always settle to the same pattern for each prototype? How do the final values of goodness differ for the different prototypes? Why do they differ? Do other initial inputs lead to other patterns? Are there other prototypes in the network that can be accessed from clamping a single unit? Try a couple of runs replicating the results shown on page 34 of *PDP:14* when *bed* and *sofa* have been clamped on. What happens when you clamp other pairs of units, like *sofa* and *bathtub*?

Hints. It is better to use the *newstart* command rather than *reset*. This ensures that successive runs are independent of each other. Note that *newstart* should be called *after* changing the external input with the *input* command. To answer these questions, it will be useful to refer to the weights used in the room example. These are shown in Figure 5.

```
cs:
disp/  exam/  get/  save/  set/  clear  cycle  do  input  log  newstart  quit
reset  run  test

 0 ceiling 100    0 very-sm   0    0 desk-ch   0    0 fire-pl   0    0 dresser   0
 0 walls   100    0 desk      0    0 clock   100    0 drapes   99    0 televis   0
 0 door      0    0 telepho  98    0 picture   0    0 stove   100    0 bathtub   0
 0 window  100    0 bed       0    0 floor-l   0    0 sink    100    0 toilet    0
 0 very-la   0    0 typewri   0    0 sofa      0    0 refrige 100    0 scale     0
 0 large     0    0 book-sh   0    0 easy-ch   0    0 toaster 100    0 coat-ha   0
 0 medium    0    0 carpet    0    0 coffee-  99    0 cupboar 100    0 compute   0
 0 small    99    0 books     0    0 ash-tra   0    0 coffeep 100   ** oven    100

   cycleno  50    goodness 21.2014    temperature  0.0000
```

FIGURE 4. The kitchen prototype, as seen in the **cs** program's display.

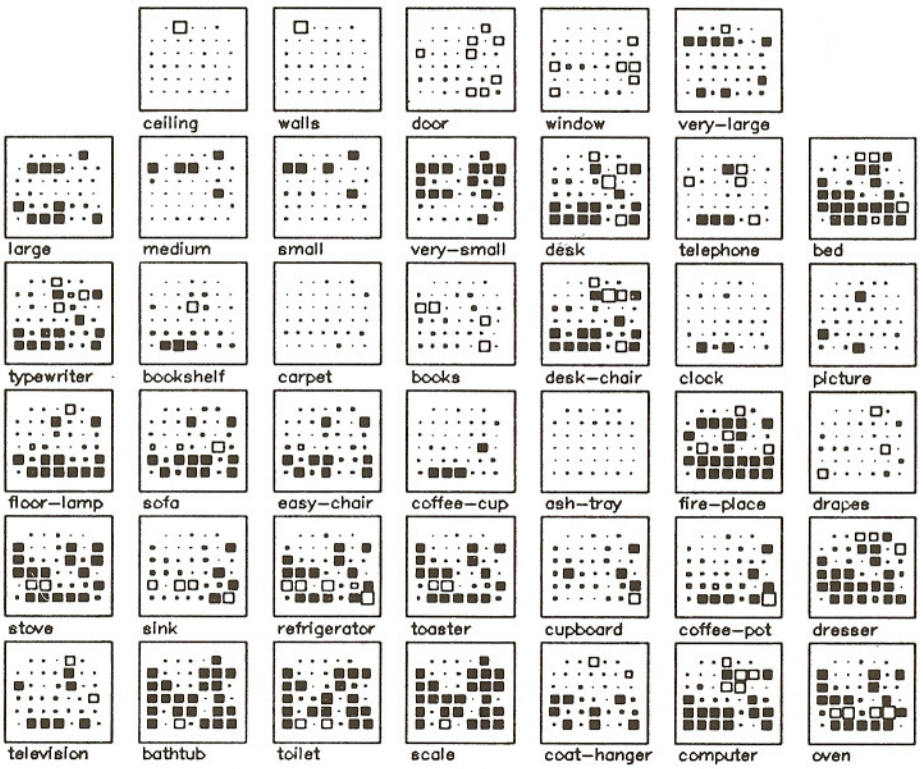

FIGURE 5. The figure uses the method of Hinton and Sejnowski (*PDP:7*) to display the weights. Each unit is represented by a square. The name below the square names the descriptor represented by each square. Within each unit, the small black and white squares represent the weights from that unit to each of the other units in the system. The relative position of the small squares within each unit indicates the unit with which that unit is connected. (From *PDP:14*, p. 24.)

On page 35 of *PDP:14* we illustrate a case in which the goodness is greater for *office* when the units *window* and *drapes* are both on or are both off than when only one of them is on. Using the program, find the numeric values of the goodnesses for the four combinations of *office* with or without windows and with or without drapes. The point of this example is that, in the case of *office*, window and drapes form a kind of a unit. An office is better with both or neither than with one alone. The window-drapes cluster interacts differently with different room types.

Finding the goodness values is a bit tricky. Essentially, you have to clamp all of the units to their prototype values and cycle a few times. Setting all 40 values using *input* can be tedious. In this case, the simplest way to proceed is to get a set of patterns from a file and then, using *test*, select one of the patterns and let the system cycle a few times to compute the goodness. For this example, there is a file called *room.pat* that contains the patterns for the prototype of each of the five rooms. This can be accessed

through the *get/ patterns* command. The following interchange will access the patterns.

cs: *get patterns*
filename for patterns: *room.pat*

The patterns for the prototypes are named *office*, *living*, *bedroom*, *bathroom*, and *kitchen*. You can examine these patterns with the command *display/ env*. It is a good idea to clear the screen first. The *display/ env* command produces a display that lists the names of the defined patterns down the left side of the screen and the names of the units (printed vertically) across the top. The patterns are rows of 1s and −1s. For example, in the office pattern, the units *ceiling*, *wall*, *door*, *large*, *desk*, *telephone*, *typewriter*, and several others are on.

You can test the system using the *office* pattern with the *test* command. It is a good idea to set *ncycles* to a small value since the final goodness value for the entire clamped network will be reached almost immediately. We are now in a position to try the window-drapes example. First test the *office* pattern. Note that both the *window* and the *drapes* units are off and that the goodness is 23.78. Now, turn *drapes* on without disturbing the rest of the inputs. This can be done with the *input* command, although you must be careful not to reset all of the input values. Now, reset the activation levels of the units and cycle.[2] You should find a goodness level of 23.11. Now clamp the *window* unit on, so that both the *window* and *drapes* units are clamped on, reset and cycle again. In this case with *both* units on, we have a goodness of 23.62. Finally, turn the *drapes* unit off (i.e., give it a negative input value), so that the *window* unit is on and *drapes* is off, and enter *reset* and *cycle* again. Here you should find a goodness of 23.35. You should now be able to carry out this procedure with the rest of the prototypes.

Q.3.2.2. First repeat the window experiment with the bathroom prototype. In what way are the results different? Try it with bedroom, kitchen, and living room prototypes as well. Note the different patterns of results in the different contexts. What sense can you make of these different patterns?

Ex. 3.3. Jets and Sharks

An additional example that can be studied within the context of the **cs** program is the Jets and Sharks example from Chapter 2. The appropriate

[2] Note that the *reset* and *newstart* commands are both fine to use in this case because all units are clamped and so the order of updating does not matter.

files are all set up; you can explore the network's performance on many of the same inputs we used in Chapter 2. To run, simply start up the cs program with the *jets.tem* and *jets.str* files, specify external inputs as appropriate, and then use the *cycle* command. See if you can understand and explain the differences between the way the two models behave on these two examples.

In experimenting with this example, it is interesting to separately explore the effects of varying the inhibition on the input units and the hidden units. This must be done by changing the values assigned to the constraint letters *h* (for hidden) and *v* (for visible) in the *jets.net* file. As supplied, *v* is set to −2.0 and *h* to −1.0. Thus, inhibition is relatively strong among the visible units, but not so strong among hidden (instance) units. You can change these values by editing this file if you wish to explore this matter further.

Ex. 3.4. Tic-Tac-Toe

As a final suggestion for an experiment to try with the schema model, we propose the following somewhat more challenging exercise. Using the basic design illustrated on page 49 of *PDP:14*, build a version of the schema model that is able to take a tic-tac-toe board as an input and settle into a state in which it produces the appropriate move. This will involve building an appropriate *tic.tem* file, an appropriate *tic.str* file, and appropriate *tic.net* and *tic.wts* files.

LOCAL MAXIMA AND THE PHYSICS ANALOGY

In this section we provide a brief description of Hinton and Sejnowski's Boltzmann machine and of Smolensky's harmony theory as they are described in *PDP:7* and *PDP:6*, respectively. These systems were developed from an analogy with statistical physics and it is useful to put them in this context. We thus begin with a description of the physical analogy and then show how this analogy solves some of the problems of the schema model described earlier. Then we turn to a description of the Boltzmann machine, show how it is implemented, show how the cs program can be used in *boltzmann* mode to solve constraint satisfaction problems, and finally discuss harmony theory and how it can be implemented using the cs program.

The primary advantage of these systems over the deterministic constraint satisfaction system used in the schema model is their ability to overcome the problem of local maxima in the goodness function. It will be useful to begin with an example of a local maximum and try to understand in some

detail why it occurs and what can be done about it. Figure 6 illustrates a typical example of a local maximum with the Necker cube. Here we see that the system has settled to a state in which the lower four vertices were organized according to interpretation A and the upper four vertices were organized according to interpretation B. Local maxima are always blends of parts of the two global maxima. We never see a final state in which the points are scattered randomly across the two interpretations. All of the local maxima are cases in which one small cluster of adjacent vertices are organized in one way and the rest are organized in another. This is because the constraints are local. That is, a given vertex supports and receives support from its neighbors. The units in the cluster mutually support one another. Moreover, the two clusters are always arranged so that none of the inhibitory connections are active. Note in this case, *Bfur* is on and the two units it inhibits, *Afur* and *Abur*, are both off. Similarly, *Bbur* is on and *Abur* and *Afur* are both off. Clearly the system has found little coalitions of units that hang together and conflict minimally with the other coalitions.

In Q.3.1.2 of Ex. 3.1, you had the opportunity to explore the process of settling into one of these local maxima. What happens is this. First a unit in one subnetwork comes on. Then a unit in the other subnetwork, which does not interact directly with the first, is updated, and, since it has a positive bias and at that time no conflicting inputs, it also comes on. Now the next unit to come on may be a unit that supports either of the two units already on or possibly another unit that doesn't interact directly with either of the other two units. As more units come on, they will fit into one or another of these two emerging coalitions. Units that are directly inconsistent with active units will not come on or will come on weakly and then probably be turned off again. In short, local maxima occur when units that

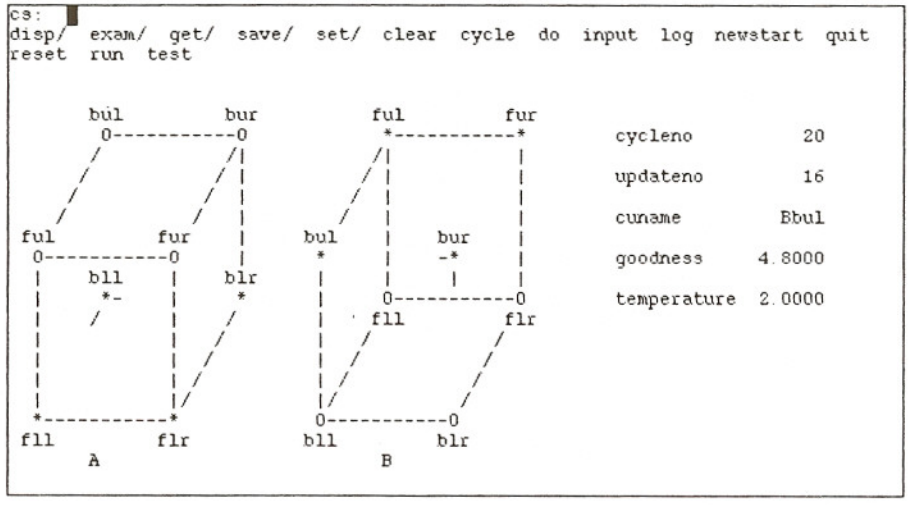

FIGURE 6. A local minimum with the Necker cube.

don't interact directly set up coalitions in both of the subnetworks; by the time interaction does occur, it is too late, and the coalitions are set.

Interestingly, the coalitions that get set up in the Necker cube are analogous to the bonding of atoms in a crystalline structure. In a crystal the atoms interact in much they same way as the vertices of our cube. If a particular atom is oriented in a particular way, it will tend to influence the orientation of nearby atoms so that they fit together optimally. This happens over the entire crystal so that some atoms in one part of the crystal can form a structure in one orientation while atoms in another part of the crystal can form a structure in another orientation. The points where these opposing orientations meet constitute flaws in the crystal.

It turns out that there is a strong mathematical similarity between our network models and these kinds of processes in physics. Indeed, the work of Hopfield (1982, 1984) on so-called Hopfield nets, of Hinton and Sejnowski (1983, *PDP:7*) on the Boltzmann machine, and of Smolensky (1983, *PDP:6*) on harmony theory were strongly inspired by just these kinds of processes. In physics, the analogs of the goodness maxima of the above discussion are *energy* minima. There is a tendency for all physical systems to evolve from highly energetic states to states of minimal energy.

In 1982, John Hopfield, a physicist, observed that symmetric networks using deterministic update rules behave in such a way as to minimize an overall measure he called *energy* defined over the whole network. Hopfield's energy measure was essentially the negation of our goodness measure. We use the term *goodness* because we think of our system as a system for maximizing the goodness of fit of the system to a set of constraints. Hopfield, however, thought in terms of energy, because his networks behaved very much as thermodynamical systems, which seek minimum energy states. In physics the stable minimum energy states are called *attractor states.* This analogy of networks falling into energy minima just as physical systems do has provided an important conceptual tool for analyzing parallel distributed processing mechanisms.

Hopfield's original networks had a problem with local "energy minima" that was much worse than in the schema model described earlier. His units were binary. (Hopfield, 1984, has since gone to a version in which units take on a continuum of values to help deal with the problem of local minima in his model. The schema model is similar to Hopfield's 1984 model.) For binary units, if the net input to a unit is positive, the unit takes on its maximum value; if it is negative, the unit takes on its minimum value (otherwise, it doesn't change value). Binary units are more prone to local minima because the units do not get an opportunity to communicate with one another before committing to one value or the other. In Q.3.1.3 of Ex. 3.1, we gave you the opportunity to run a version close to the Hopfield model by setting *instr* to 2.0 in the Necker cube example. In this case the units are always at either their maximum or minimum values. Under these conditions, the system reaches local goodness maxima (energy minima in Hopfield's terminology) much more frequently.

Once the problem has been cast as an energy minimization problem and the analogy with crystals has been noted, the solution to the problem of local goodness maxima can be solved in essentially the same way that flaws are dealt with in crystal formation. One standard method involves *annealing.* Annealing is a process whereby a material is heated and then cooled very slowly. The idea is that as the material is heated, the bonds among the atoms weaken and the atoms are free to reorient relatively freely. They are in a state of high energy. As the material is cooled, the bonds begin to strengthen, and as the cooling continues, the bonds eventually become sufficiently strong that the material freezes. If we want to minimize the occurrence of flaws in the material, we must cool slowly enough so that the effects of one particular coalition of atoms has time to propagate from neighbor to neighbor throughout the whole material before the material freezes. The cooling must be especially slow as the freezing temperature is approached. During this period the bonds are quite strong so that the clusters will hold together, but they are not so strong that atoms in one cluster might not change state so as to line up with those in an adjacent cluster—even if it means moving into a momentarily more energetic state. In this way annealing can move a material toward a global energy minimum.

The solution then is to add an annealing-like process to our network models and have them employ a kind of *simulated annealing.* The basic idea is to add a global parameter analogous to temperature in physical systems and therefore called *temperature.* This parameter should act in such a way as to decrease the strength of connections at the start and then change so as to strengthen them as the network is settling. Moreover, the system should exhibit some random behavior so that instead of always moving uphill in goodness space, when the temperature is high it will sometimes move downhill. This will allow the system to "step down from" goodness peaks that are not very high and explore other parts of the goodness space to find the global peak. This is just what Hinton and Sejnowski have proposed in the Boltzmann machine, what Geman and Geman (1984) have proposed in the *Gibbs sampler*, and what Smolensky has proposed in harmony theory.

The essential update rule employed in all of these models is probabilistic and is given by what we call the *logistic* function:

$$\textit{probability}\,(a_i\,(t) = 1) = \frac{1}{1 + e^{-net_i/T}} \tag{8}$$

where T is the temperature. This differs from the basic schema model in three important ways. First, like Hopfield's original model, the units are binary. They can take on only their maximum and minimum values. Second, they are *stochastic.* That is, the update rule specifies only a *probability* that the units will take on one or the other of their values. This means that the system need not necessarily go uphill in goodness—it can move downhill as well. Third, the behavior of the systems depends on a global parameter, *temperature*, which can start out high and be reduced

during the settling phase. These characteristics allow these systems to implement a simulated annealing process. One final point: It is not accidental that these three models all choose exactly the same update rule. This rule is drawn directly from physics and there are important mathematical results that, in effect, guarantee that the system will end up in a global maximum if the system is annealed slowly enough. Having made the analogy with physics, we can also make use of the results of physics to describe the behavior of our networks.

Figure 7 shows the probability values as a function of net input and the temperature. Several observations should be made. First, if the net input is 0, the unit takes on its maximum and minimum values with equal probability. Second, if the net input is large enough, the unit will always take on its maximum value no matter what value the temperature is; and if the net input in sufficiently negative, the unit will take on its minimum value no matter what the temperature. Third, as the temperature approaches 0, the function becomes deterministic and takes on its maximum value if the net input is positive and its minimum value if the net input is negative. This "zero temperature" case is identical to the Hopfield binary unit model. Thus, we see that there are two dimensions on which these models are varying—whether the units are binary or continuous values and whether the models are stochastic (probabilistic) or deterministic. The Hopfield (1982) model is deterministic and binary. The Boltzmann machine and harmony theory are stochastic and binary. The Hopfield (1984) model and the schema model are deterministic and continuous. All five models converge to binary, deterministic models as the temperature goes to 0 in the

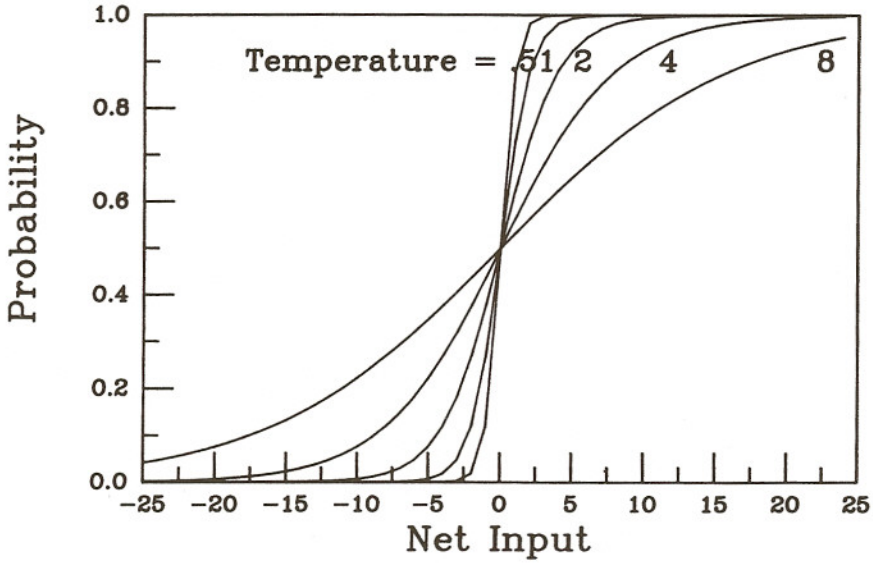

FIGURE 7. Activation probabilities according to the logistic function. (From *PDP:2*, p. 69.)

stochastic models and as the size of the step taken per update (controlled by *istr*) is increased in the schema model.

THE BOLTZMANN MACHINE

In *PDP:7* Hinton and Sejnowski described the Boltzmann machine as a constraint satisfaction system. Most of their chapter, however, focuses on the development of a very interesting learning procedure. Here, we focus on the use of the Boltzmann machine as a constraint satisfaction system. We will especially focus on its role in reducing the frequency of local goodness maxima. (Following Hopfield and the physics analogy, Hinton and Sejnowski spoke of energy and energy minimization. We will persist in looking at the negation of energy, which we call goodness. The principles are identical in either case, only the terminology varies.)

Implementation

The Boltzmann machine is conceptually very similar to the schema model. Indeed, the Boltzmann system is accessed as a mode of the cs program. As in the schema model, there are two essential subroutines for Boltzmann. These are a *cycle* and an *rupdate* routine. The *cycle* routine is absolutely identical for the two models. The *rupdate* routine differs by only a few lines of code—namely, the code for assigning an activation value to a unit. Since they are nearly the same, we have deleted the comments and reproduced the code for the Boltzmann version of *rupdate* below.

```
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 *= istrength;
      netinput += estrength * input[i];

      if(probability(logistic(netinput)) == 1)
        activation[i] = 1;
      else
        activation[i] = 0;
  }
}
```

where *logistic(x)* is a function given by

```
logistic(x) {

  return(1.0 / (1.0 + exp( -1 * x / temperature)));
}
```

and *probability(x)* is a function that takes on value 1 with a probability equal to the value of its argument:

```
probability(x) {

  if (rnd() < x)
    return(1);
  else
    return(0);
}
```

The *rnd()* function simply returns a uniformly distributed random number between 0 and 1.

Running the Program in Boltzmann Mode

Since the Boltzmann machine is so similar to our previous program, it is implemented as mode of program **cs**. It is accessed by setting *mode boltzmann* to 1. All of the commands and variables needed are described in the list that begins on page 56. Of these, the most important specifically for Boltzmann machines is the *get/ annealing* command, used to specify an annealing schedule.

Ex. 3.5. Simulated Annealing in the Cube Example

Compare simulated annealing using the Boltzmann machine to the results you obtained with the schema model for the cube example. To switch to Boltzmann mode, enter *set/ mode boltzmann 1* after starting up the program with *cube.tem* and *cube.str*. You may also want to increase *ncycles* to 60, since this is just beyond the point at which the annealing schedule levels off.

Q.3.5.1. Run 20 runs with the Boltzmann machine, using its default annealing schedule, and see how often the program gets stuck in a local minimum. Compare these results to the case where the continuous updating scheme is used.

Q.3.5.2. Examine the density of local maxima as a function of the annealing schedule. Try several schedules varying in gradualness and starting temperature. Which variable makes more of a difference? Can you explain why?

Hints. It is necessary to run 20 runs or so with each schedule to get reasonable estimates of the probabilities of getting stuck in local maxima. Use relatively high values of the starting temperature and very quick drops if you want to get a feel for what is happening. To study the time course of these simulations, it is useful to set *stepsize* to *update* and to set *single* to 1. Then you can step through and watch the network settle. You will note that things begin to settle down when the temperature begins to get rather low, say, in the range of 0.5 to 0.05.

Other Experiments With Boltzmann Machines

There are many more experiments that can be done with Boltzmann machines. For example, you may explore the effects of stochastic variability in the room example, the Jets and Sharks example, or the tic-tac-toe example. We provide one additional example, called *boltz*. To run it, execute the **cs** program with the *boltz.tem* and *boltz.str* files. The screen display indicates the excitatory connections; you will want to study the *boltz.wts* file to see what the inhibitory connections are. You can also construct examples of your own, perhaps setting up a network that you feel might be challenging for finding global minima reliably.

HARMONY THEORY

Harmony theory, which was developed by Paul Smolensky, is described in *PDP:6*. Although the basic mathematics of harmony theory is rather similar to the Boltzmann machine, the structure and motivation are different. Whereas we can think of the Boltzmann machine as an arbitrarily interconnected set of homogeneous units, harmony theory presupposes two distinct layers of units. As illustrated in Figure 8, a harmony network consists of a lower layer of *representational feature* units and an upper layer of *knowledge atoms*. The feature units take on activation values ± 1, whereas the knowledge atoms take on values 0 and 1. It is useful to think of the feature units as corresponding to the featural description of a situation. In a complete description, each feature is either present (+1) or absent (−1). The knowledge atoms, on the other hand, are best thought of as bits of knowledge about what configurations of features "go together." Knowledge

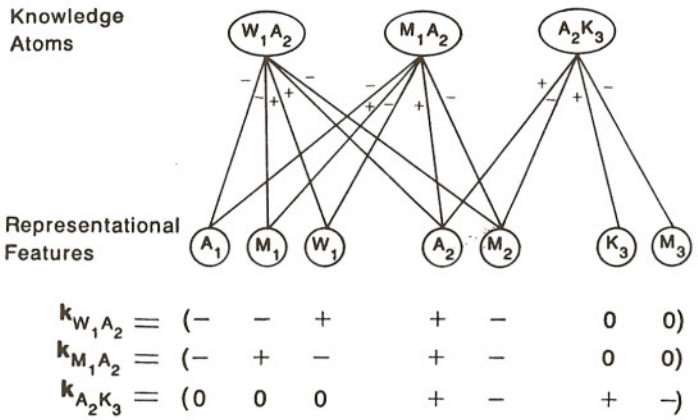

FIGURE 8. The graphical representation of a particular harmony model. In this model, the representational features represent letters in particular positions, and the knowledge atoms represent common letter combinations. (From *PDP:6*, p. 216.)

atoms may be either active or inactive. When a knowledge atom is active, it can be viewed as asserting that the configuration of input features it is looking for is present in the environment. When a knowledge atom is inactive it can be viewed as asserting that the evidence does not warrant such an assertion. All connections in a harmony model are symmetric, and all connections are between features and knowledge atoms. Thus, a given feature may either excite a knowledge atom that is consistent with it, inhibit a knowledge atom that is inconsistent with it, or have no effect on a knowledge atom to which the feature is irrelevant. Similarly, knowledge atoms specify certain configurations of features that are consistent with the knowledge represented by that atom. Thus a knowledge atom may activate those features that are consistent with the atom, inhibit those that are inconsistent, or not connect with those that are irrelevant to the contents of that atom. Neither features nor knowledge atoms are directly connected to one another. All connections in the system are ± 1. However, each knowledge atom has a *strength* designated σ. The strength corresponds to the degree that the knowledge atom in question insists that the features to which it is connected are present in the input.

Harmony theory is so named because, for any configuration of input features, the system finds the configuration of knowledge atoms that is maximally consistent, or harmonious, with the featural constraints. It is useful to see the configuration of active knowledge atoms as an *interpretation* of the input features.

In addition to creating an interpretation of a set of input features, the knowledge atoms themselves can *fill in* missing features in a way that is maximally consistent with those features that are fixed (clamped) and the set of knowledge atoms. This is the so-called *completion* problem. The

harmony function itself is very similar to the "goodness" function of the previous sections. It can be written in the following way:

$$harmony = \sum_i \sigma_i a_i h_i.$$

Here, i ranges over the knowledge atoms, and h_i is a measure of the degree to which the current set of feature values is consistent with knowledge atom i. The variable σ_i is a strength or importance value associated with unit i. The variable h_i is given by

$$h_i = \frac{\sum_j r_j k_{ij}}{n_i} - \kappa.$$

Here j ranges over features, r_j is the activation of representational feature j, and n_i is the number of nonzero connections to atom i. The variable k_{ij} is given by

$$k_{ij} = \begin{cases} 1 & \text{if positive connection} \\ -1 & \text{if negative connection} \\ 0 & \text{if no connection.} \end{cases}$$

In other words, the total harmony is given by the sum of contributions of each of the knowledge atoms. If a knowledge atom is not activated ($a_i = 0$), there is no contribution. If it is active ($a_i = 1$), then it contributes an amount that is proportional to the product of its importance, σ_i, and a term representing the consistency of that atom with the current pattern of activation among the representational features. This consistency term, h_i, is the proportion of relevant features that are consistent minus the proportion that are inconsistent, less a constant κ. Consider first the case in which κ is 0. In this case, turning on atom i will contribute a positive amount to the overall harmony of the system whenever the number of consistent features exceeds the number of inconsistent features. If κ is near 1, then it will contribute to the overall harmony only when all, or nearly all, of its features match the template for the atom. The standard goodness function discussed in the Boltzmann model and the schema model corresponds to the $\kappa = 0$ case.[3] Given the motivation of harmony theory, larger values of κ make more sense.

---

[3] Note that if $\kappa = 0$ we can represent the harmony function as

$$harmony = \sum_{i,j} a_i r_j w_{ij}$$

where $w_{ij} = \sigma_i k_{ij} / n_i$. This is simply the standard goodness function discussed above.

Implementation

The harmony model is implemented as a mode of the **cs** program. The main differences between *harmony* mode and *boltzmann* mode are

- Only the weights to each atom from each feature are specified. The weight to a feature from an atom is just the weight from the feature to the atom.
- The *rupdate* routine is modified so that there are two versions of the update function, depending on whether the unit being updated is a knowledge atom or a feature.
- The *goodness* measure (now called *harmony*) is computed as just described, taking the importance variables σ_i and κ into account.

Values for σ_i are set by default to 1.0. Other values may be specified in the *.net* file, as described in Appendix C.

Running the Program in Harmony Mode

Harmony mode is accessed simply by setting the *harmony* mode variable to 1. The *clamp* mode is also set to 1. The parameter *kappa* and the configuration variables *nunits* and *ninputs* must be defined. The variable *ninputs* indicates the number of features. The rest of the units are treated as knowledge atoms. The configuration variables are generally defined in the *.net* file; the weights and sigmas are also specified there. Once a set of templates has been specified and the network initialized, the harmony version of the **cs** program is run just like the Boltzmann version: Some features are clamped on (+1) or off (−1), an annealing schedule is defined, the network is reset to initialize the annealing schedule, and then the *cycle* command is entered to initiate processing.

Ex. 3.6. Electricity Problem Solving

Consider the electricity problem described in *PDP:6* (p. 240) and illustrated in Figure 9. This problem, first developed by Riley and Smolensky (1984), illustrates how harmony theory can be employed to solve certain "higher level" problems. In this case, the problem is to determine how different variables in an electrical circuit change when other variables are altered. For example, what happens to the total resistance in the circuit

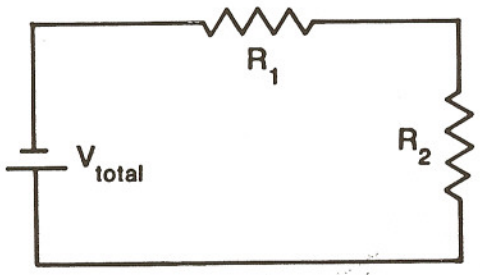

FIGURE 9. If the resistance of R_2 is increased (assuming that V_{total} and R_1 remain the same), what happens to the current and voltage drops across the two resistors? (From *PDP:6*, p. 240.)

shown when the resistance in one of the resistors is increased? Assuming total voltage stays constant, what happens to the voltage drop across each resistor, and what happens to the current? The first step is to develop a set of representational features. In this case, we must represent seven quantities: the total current, I; the resistances, R_1 and R_2; the total resistance, R_{total}; the voltage drops across the two resistors, V_1 and V_2; and the total voltage, V_{total}. For each of these quantities we must represent whether it goes up, goes down, or stays the same. This is done by assigning two units to each quantity: one to indicate whether or not a change occurs in that variable (+1 indicating change and −1 indicating no change) and one to indicate the direction of change. Use +1 to indicate an increase and −1 to indicate a decrease. (If no change occurred, the value of the second feature is irrelevant.) Figure 10 shows the screen layout for the electricity problem. There are columns for each of the seven variables. Below each column is a

```
cs:
disp/  exam/  get/  save/  set/  clear  cycle  do  input  log  newstart  quit
reset  run  test

                I  R1 R2 RT V1 V2 VT          cycleno          0
      Inputs    00 10 11 00 00 00 10          temp        1.0000
                cu cu cu cu cu cu cu          harmony     0.0000
      Features  00 00 00 00 00 00 00

              knowledge atom activations

              u u u u u s s s d d d d d
              u s d d d u s d u u u s d
              u u u s d u s d u s d d d

 V1 + V2 = VT  0 0 0 0 0 0 0 0 0 0 0 0 0
 R1 + R2 = RT  0 0 0 0 0 0 0 0 0 0 0 0 0
  I * R1 = V1  0 0 0 0 0 0 0 0 0 0 0 0 0
  I * R2 = V2  0 0 0 0 0 0 0 0 0 0 0 0 0
  I * RT = VT  0 0 0 0 0 0 0 0 0 0 0 0 0
```

FIGURE 10. Screen layout for the electricity problem.

set of pairs of features, one indicating whether or not that quantity changed (indicated by a *c*) and one indicating whether the quantity went up or down (indicated by a *u*). Note that the row labeled *Inputs* is a representation of the problem. The 0s indicate unclamped inputs—inputs to be filled in through processing. The ± 1 values indicate the clamped inputs, which constitute the problem specification. In this case, we have R_2 increasing (both the *change* feature and the *up* feature are clamped +1), and we have V_{total} and R_1 unchanged (the *change* feature is clamped to -1). All other features are left free.

The next problem in specifying a harmony network is to encode the knowledge constraints. In this case, the knowledge is of the facts of electrical circuits. We want to represent knowledge about electricity *qualitatively.* We can do this by taking the laws of electricity (Ohm's law and Kirchoff's law), determining the legitimate relationships among the variables involved, and building knowledge atoms for each such relationship. An example should clarify this. Consider first the law that the total voltage drop is the sum of the voltage drops over each resistor, $V_1 + V_2 = V_{total}$. This equation allows for 13 qualitative relationships among the variables. V_1 could increase and V_2 could increase, in which case V_{total} must increase; V_1 could increase and V_2 could stay the same, in which case V_{total} must increase; V_1 could increase and V_2 decrease, in which case V_{total} could increase, stay the same, or decrease; and so on. There are five such equations and 13 qualitative relationships for each equation. This leads to 65 knowledge atoms encoding these relationships. The relationships and knowledge atoms are shown in Figure 10. All of these relationships must be encoded in the network by specifying a positive, negative, or zero weight from each input feature to each knowledge atom. Here is the portion of the network specification for the 13 laws related to voltage:

```
........pppppp
........ppm.pp
........pppmpp
........pppmm.
........pppmpm
........m.pppp
........m.m.m.
........m.pmpm
........pmpppp
........pmppm.
........pmpppm
........pmm.pm
........pmpmpm
```

The *p* represents +1 and the *m* represents -1. Since the weights are symmetric in harmony theory, we only require that the connections from the input feature units to the knowledge atom units be specified. Since this problem involves the relationships between voltage, V, current, I, and

resistance, R, we have called this the VIR problem and have named the relevant files *vir.tem*, *vir.net*, *vir.str*, and so on.

You should now be ready to run the VIR problem. Note that the inputs specifying our example problem are set up, and an annealing schedule is defined in the *.str* file. All you need to do to run this exercise is start up the **cs** program with the *vir.tem* and *vir.str* files, then issue the *cycle* command.

Q.3.6.1. Watch the system solve the problem and note the final equations selected for the solution. Run the problem several times looking for local maxima.

Hints. This network settles very slowly; it may be necessary to run 300 or sometimes 400 cycles before it is safe to assume things have stopped changing; even then you will find that some of the knowledge atoms flicker on and off. It may be useful to set *stepsize* to *ncycles*, set *ncycles* to 100, and issue the *cycle* command three or four times in each run since much of the total elapsed time can be taken up in screen updates. At the end you may want to run a few more cycles with *single* set to 1 and *stepsize* set to *cycle* to see which units are flickering on and off.

The next two questions are somewhat time consuming and require you to follow the course of processing in the network carefully over time. You will probably have to do several runs in each case to get reliable results. It may be useful to make use of the *do* command facility and log files so that you can run several runs automatically and save the results; Appendix D describes utility programs that may be useful in analyzing the output.

Q.3.6.2. Smolensky reported that the system seemed to come to various conclusions sequentially. See how well you are able to replicate his findings.

Q.3.6.3. Select another problem in the VIR domain (i.e., input a different set of initial constraints) and look at the sequential character of the problem solving in this case. Does it seem reasonable that one might solve the problem in that order? Why does the system seem to settle on different aspects of the problem in different orders?