1-31-1991

An efficient annealing algorithm for global optimization in Boltzmann machines

Rajendra Sarasakrishna
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

Abstract

AN EFFICIENT ANNEALING ALGORITHM FOR GLOBAL OPTIMIZATION IN BOLTZMANN MACHINES

Rajendra Sarasakrishna, MSEE, New Jersey Institute of Technology

Thesis Advisor: Prof. Nirwan Ansari

This thesis proposes a new annealing algorithm for Boltzmann Machines. This algorithm uses an Exponential Formula for temperature scheduling that produces remarkably better solutions for global optimization. The superiority of the new algorithm is shown by computer simulations of several examples on Boltzmann Machine and it s variants. This is also shown to have better properties than algorithms like Generalized Simulated Annealing which possess somewhat similar dynamics.

An Efficient Annealing Algorithm for Global Optimization in Boltzmann Machines

by

Rajendra Sarasakrishna

Thesis submitted to the Faculty of the Graduate School
of New Jersey Institute of Technology in partial
fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering
1991

Approval Sheet

Title of Thesis: **An efficient annealing algorithm for Global Optimization in Boltzmann Machines**

Name of Candidate: Rajendra Sarasakrishna
Master of Science in Electrical Engineering, 1991

Thesis & Abstract Approved
by the Examining Committee:

Dr. Nirwan Ansari, Advisor — Date
Assistant Professor
Department of Electrical and Computer Engineerig

Dr. Edwin Hou — Date
Assistant Professor
Department of Electrical and Computer Engineering

Dr. Zoran Siveski — Date
Assistant Professor
Department of Electrical and Computer Engineering

New Jersey Institute of Technology, Newark, New Jersey.

VITA

Name: Rajendra Sarasakrishna

Permanent address:

Degree and date to be conferred: M.S.E.E., Dec, 1990.

Date of birth:

Place of birth:

Secondary education: V.V. Excelsior High School, Hyderabad, India

Collegiate institutions attended	Dates	Degree	Date of Degree
Osmania University, India	1979-83	B.S.E.E.	August, 1983
New Jersey Institute of Technology	1989-90	M.S.E.E.	Dec, 1990

Major: Electrical Engineering

Acknowledgements

I wish to express my sincere thanks to Dr. Nirwan Ansari for his valuable guidance and encouragement throughout the course of this thesis. I am specially indebted to him for his insightful and constructive criticism, without which this thesis would never have been completed. I also thank the members of my thesis committee, Dr. Edwin Hou and Dr. Zoran Siveski, for their suggestions and comments.

Last, but not least, I would like to thank all those who have helped me in one way or other during the course of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Boltzmann Machine and it s variants offer a generalized computational approach that can be applied to the basic research issues of search, representation and learning and have rigorous mathematical proofs [6][1]. They can also be considered a model for parallel implementation of simulated annealing. The problems solved by these systems can be considered as combinatorial optimization problems. These systems are made up of a network of units which try to reach a maximal consensus about their individual states, subject to the constraints set by the connection strengths, the connections having been learned for a particular problem. The units adjust their states to the states of the units to which they are directly connected. The final objective of the state transitions is to reach a global optimum of a cost-function for the given problem. This cost-function is called Energy in Boltzmann Machines, Harmony in Harmony Machines and can be considered as a measure of goodness-of-fit or the self-consistency of the system.

To achieve this optimization of the cost-function, the state transitions of the individual units are done iteratively using a probabilistic decision rule. The randomness of the decision depends on a parameter T, called temperature. Randomness is necessary to avoid local optima. A weakness of Boltzmann Machine is

[1]The numbers in the square brackets indicate the corresponding references in the bibliography.

that a proper method of varying the temperature has been difficult to determine. Many attempts have been forwarded both theoretical and empirical. However, no finite length path has been determined that can guarantee a global optimum. The path of the temperature, as it is varied, is called the Annealing Schedule.

The efficiency of an annealing schedule is determined by the probability of achieving the global optimum given equivalent computations. The most common annealing schedule is a linear one, where the temperature is started at a high value and decreased slowly as a function of iterations at a predetermined rate called coolrate. However, this schedule and it's many versions have the inherent drawback that after a certain number of iterations the temperature eventually reaches a very low value where the probabilistic decision rule becomes practically a deterministic decision rule. At this stage, it would be very difficult for the system to escape from a local optimum if it is stuck in one.

A proposal has been made in this thesis for an exponential schedule, partly empirical and partly theoretical, that does not encounter the above mentioned drawback of premature freezing. Hence the system never gets stuck in a local optimum. This also means that it would be difficult to determine when to stop iterating as the system never practically freezes. However, this problem can be overcome with a minor and general improvement of storing in memory the most optimum value obtained at any iteration for the cost-function and corresponding state of the system. This technique (of storing the optimum value in memory) can be termed a general improvement as it can be shown that this improves Simulated Annealing even with traditional cooling schedules[4].

A brief discussion has been given in chapter 2 about the the Boltzmann Machine and it's variants, the general technique of Simulated Annealing, and the linear cooling schedule. Chapter 3 deals with the new algorithm and it is shown

that this possesses better properties than algorithms like Generalized Simulated Annealing. In Chapter 4, a discussion of the results of applying the new algorithm on a few examples is given and a comparison is made with the results of applying the traditional linear temperature schedule. Conclusions are given in Chapter 5.

Chapter 2

Boltzmann Machine and Simulated Annealing

The Boltzmann machine, introduced by Hinton and Sejnowski [5], can be viewed as a generalization of Hopfield's content-addressable memory[9]. The units have binary-valued states and the connections are bidirectional. The Boltzmann machine uses a probabilistic state transition mechanism and can have hidden units to capture higher-order regularities during learning. There is no limitation on the number of units, number of layers, or the connection topology except that the connections are symmetric and units may not connect to themselves. For a complete reference on Boltzmann Machines, readers are referred to Reference[5].

The Harmony Machine developed by Paul Smolensky [7] can be considered a version of Boltzmann Machine and is explained at a later paragraph below.

Both the above machines can be used for combinatorial optimization and can be considered as constraint satisfaction systems. Constraint satisfaction involves many factors. For example, the solution to a problem may involve the simultaneous satisfaction of a very large number of constraints. To make the problem more difficult, there may be no perfect solution in which all the constraints are completely satisfied. In addition, some constraints may be more important than others. In general, this is a very difficult problem. It is what

Minsky and Papert have called the best match problem[8].

Connectionist systems like Boltzmann Machine and Harmony Machine solve this problem in a very natural way. Such problems can be translated into connectionist language 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 B. If the constraint is very important, the weights are large. Less important constraints involve smaller weights. Thus the strength of a connection in these systems can be considered as a quantitative measure of the desirability that the units joined by the connection are both 'on'.

The constraint satisfaction problem can now be cast in the following way. Let *goodness-of-fit* or goodness for short be the measure of the degree to which the desired constraints are satisfied. First, this measure 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, then the constraint is satisfied to the degree that both units are turned on. If the connection is negative, then the constraint is violated to the degree that both units are turned on. Thus, for units i and j, the product $w_{ij}a_ia_j$ represents the degree to which the pairwise constraint between the two hypotheses is satisfied. (w_{ij} represents the weight between the units i and j, a_i represents the activation of unit i, and a_j represents the activation of unit j). Secondly, 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 external direct evidence is available is given by the product of the external input value times the activation value of the unit. The bigger this product, the

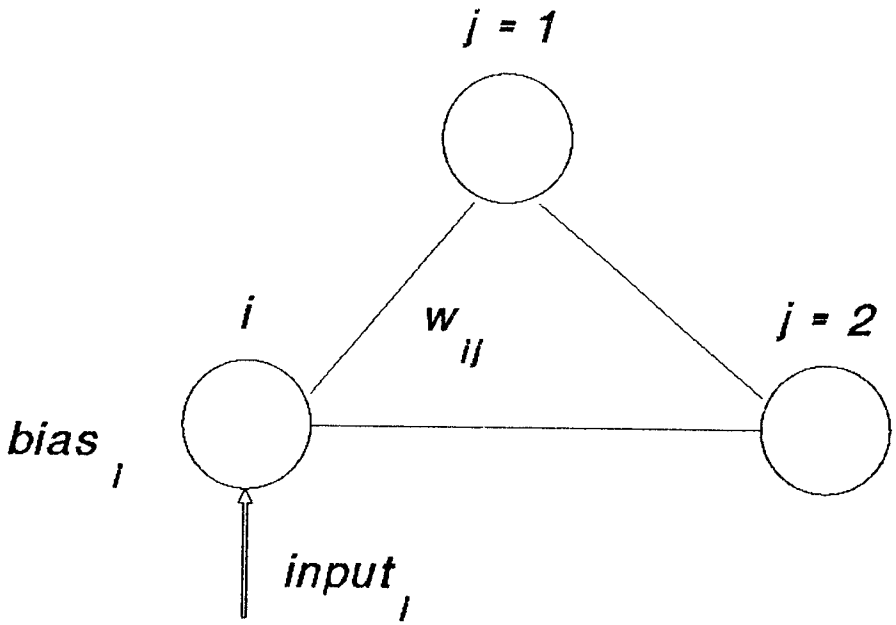


Figure 2.1: An example of the Boltzmann Machine network

better the system is satisfying this external constraint. Thus, the goodness of fit of unit i can be defined as[2]

$$goodness_i = \sum_j w_{ij} a_i a_j + input_i a_i + bias_i a_i. \tag{2.1}$$

This is the goodness of just one unit. In other words, this is just the sum of all of the individual constraints in which the corresponding hypothesis participates. We can define the overall goodness of fit as the sum of the individual goodnesses. In this case we get[2]

$$goodness = \sum_{ij} w_{ij} a_i a_j + \sum_i input_i a_i + \sum_i bias_i a_i. \tag{2.2}$$

Now, the constraint satisfaction problem can be solved if we can find a set of activation values that maximizes this function. It can be noted 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[2],

$$goodness_i = net_i a_i \tag{2.3}$$

where net_i is defined as[2],

$$net_i = \sum_j w_{ij} a_j + input_i + bias_i. \tag{2.4}$$

Thus, the net input into a unit provides the unit with information as to it's contribution to the goodness of the entire solution. Consider any particular unit in the network. That unit can always behave as to increase its contribution to the overall goodness of fit if, whenever its net input is positive, the unit moves its activation towards its maximum value; and whenever its net input is negative, it moves its activation towards its minimum value.

In summary, the constraint satisfaction problem can be stated as finding the global optimum of the cost function, i.e., goodness. (In this thesis, 'optimum' and 'maximum' mean the same, as the Boltzmann Machine reaches the optimum solution when the goodness is maximum.) However, this process cannot be completed in one step, as the activations of the units are mutually constraining. That is, if the units A and B are connected, the activation of unit A depends on the activation of unit B, and in turn the activation of unit B depends on the activation of unit A. Hence, the optimum values of activations of these units must be decided iteratively. An iteration can be defined as the process of updating the activations of all the units in the system once, based on some update rule. (In Boltzmann Machine, the units may be picked up randomly as all the units are of same type.) However, carrying on the updation through many iterations may still yield only a local goodness optimum and may not end in a global goodness optimum. It is easy to see why. The goodness space may contain many local optima and updating the units in such a way that the goodness is monotonously increased will

very likely lead to a local optimum depending on the problem and the starting point. State transitions that decrease the goodness must be allowed with a fixed probability to escape local optimum. The process of simulated annealing is used for this purpose.

The essential state transition rule employed in simulated annealing models is probabilistic and is given by the logistic function[2]:

$$probability(a_i = 1) = \frac{1}{1 + e^{-net_i/T}} \tag{2.5}$$

where T is the computational temperature. The updation of the activations of the units is carried out iteratively using the above probabilistic decision rule. The temperature is a scaling factor that is varied in such a way as to control the randomness of the decisions. Several observations can be made about the temperature from the above equation. 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 its maximum value no matter what value the temperature is; and if the net input is 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 minimum value if the net input is negative.

If none of the above is true, the unit will take on a maximum value depending on the probability given by the Equation 2.5. However, this is a tricky situation as much depends on the proper value of the temperature at the corresponding iteration. Hence, the ability of the system to escape from local optimum depends much on the path chosen for the temperature, called annealing schedule. In other words, it can be stated that the success of simulated annealing depends on the proper choice of the annealing schedule.

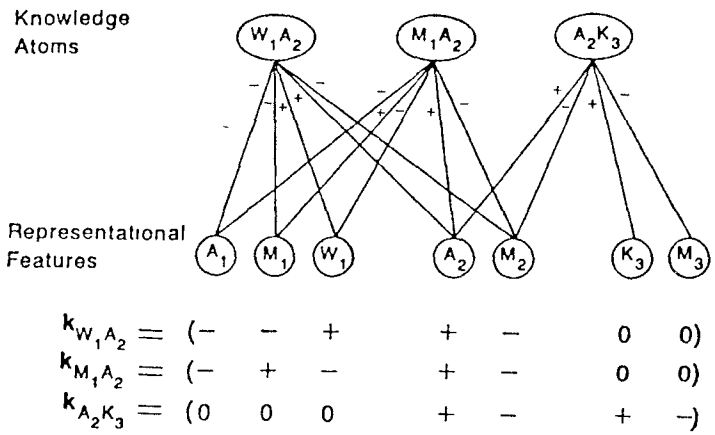


Figure 2.2: An example of the Harmony Machine Network

The traditional linear cooling schedule is explained at a later section below, after a brief discussion of the Harmony Machine, a version of Boltzmann Machine.

2.1 Harmony Machine

The basic mathematics and state transition mechanisms of Harmony Machine are rather similar to the Boltzmann Machine, though it's structure is a bit different. Whereas the Boltzmann Machine is an arbitrarily interconnected set of homogenous units, harmony machine presupposes two distinct layers of units. It consists of a lower layer of representational feature units and an upper layer of knowledge atoms. The feature units take on activation values -1 and $+1$, whereas the knowledge atoms take on values 0 and 1. For a complete reference on Harmony Machine see readers are referred to reference[7]. Due to the different structure of the Harmony Machine compared to the Boltzmann Machine, the equations

governing goodness and net input are slightly different and are given below[2].

$$overall\ goodness = \sum_i \sigma_i a_i h_i. \tag{2.6}$$

Here, a_i is the activation of the knowledge atom i, h_i is the 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

$$h_i = \frac{\sum_j r_j k_{ij}}{n_i} - \kappa. \tag{2.7}$$

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 Equation 2.7, κ is a constant which can be explained as follows: If κ is 0, turning on atom i will contribute a positive amount to the overall goodness of the system whenever the number of consistent features exceed the number of inconsistent features. If κ is near 1, then it will contribute to the overall goodness only when nearly all of its features match the template for the atom.

The net input to a knowledge atom is given by[7]

$$I_i = \sum_j \frac{\sigma_i k_{ij} r_j}{n_i} - \sigma_i \kappa. \tag{2.8}$$

The net input for a representational feature unit is given by[7]

$$I_j = 2 \sum_i \frac{\sigma_i k_{ij} a_i}{n_i}. \tag{2.9}$$

The formulae for I_i and I_j are both derived from the fact that the input to a node is precisely the harmony the system would have if the given node were to choose the value 1 minus the harmony resulting from not choosing 1. The factor

of 2 in the input to a feature node is in fact the difference (+1) - (-1) between its possible values. The term κ in the input to an atom comes from the κ in the harmony function; it is a threshold that must be exceeded if activating the atom is to increase harmony.

The net inputs for knowledge atoms and representational features as given above respectively are used in the probabilistic state transition equation. An iteration consists of updating each unit of one layer first and then updating each unit of the other layer. It may be noted that Goodness is termed as Harmony in Harmony Machines. It is also equal to the negative of Energy in Boltzmann Machines, and it follows that Harmony is equivalent to the negative of Energy[7].

2.2 The linear annealing schedule

In this schedule (Reference [2] discusses simulation software using linear annealing schedule), in general, the temperature T of the probabilistic state transition equation is decreased linearly as the number of iterations increases. First the *coolrate* is determined.

$$Coolrate = \frac{\text{Required change in temperature}}{\text{number of iterations over which the change is required}} \tag{2.10}$$

Then Temperature at an iteration n is given by

$$T(n) = \text{Initial temperature} - (n-1) * (Coolrate). \tag{2.11}$$

It is also possible to specify different coolrates between different iteration ranges, and calculate the temperature accordingly.

The major drawback of the above cooling schedule is, that irrespective of the system state, the temperature reaches a low enough value at a predetermined number of iterations. At a low temperature the probabilistic update rule becomes a deterministic decision rule and it would be very difficult for the system to escape

a local minimum if it is stuck in one. Many altered versions of this schedule also face similar drawbacks.

In the next chapter, a proposal is made for a new exponential cooling schedule that does not suffer from the above drawback. Simulation results of chapter 4 demonstrate that the exponential cooling schedule is much superior to the above linear schedule.

Chapter 3

The new Exponential Formula for Temperature Scheduling

We have discussed in the previous chapter how a linear cooling schedule may lead to an irreversible low value for the temperature factor where it would be difficult for the system to escape a local optimum. To avoid this, we need an annealing schedule where the temperature never gets stuck at a particular value, but varies dynamically in such a way as to increase the system optimality.

In general, the following properties are desirable:

1. The temperature must be very low to remove the randomness in the state transition decisions, and encourage the trend whenever the system is making state transitions as to increase the goodness.

2. However, when the increase in goodness approaches 0, or the system is stuck in a local goodness maximum, the temperature must increase and allow a series of random state transitions to make the system escape from the local goodness maximum.

3. After the system makes a series of random state transitions and escapes from local goodness maximum as explained in above, the temperature must again decrease as to begin a new search path for a goodness maximum that is greater than the previous goodness maximum.

4. It is also desirable to fix a maximum value for the temperature beyond which the temperature will not increase as long as the change in goodness is positive (goodness increases).

3.1 The Exponential Formula and the SA algorithm

We describe below how these properties can be achieved through an exponential formula for temperature scheduling. The basic idea of this formula is to keep the temperature small while the system is going uphill in the goodness space to encourage the trend, and keep the temperature high when the system is stuck in a local goodness maximum. Controlling the temperature this way binds the probabilities of accepting good and detrimental state transitions(The original idea of binding the probabilities is from Bohachevsky et. al. [1],[3]). It may be noted that the sufficient condition for finding global optimum in Boltzmann Machines is given by the formula of Geman et.al. [11] ($T_n = T_0/\log(1+n)$). Similarly, for Cauchy Machines, sufficient condition for finding global optimum is given by the formula of Szu[12], called Fast Simulated Annealing, ($T_n = T_0/(1+n)$). However, the sufficient conditions are too slow to be useful as the temperature attains a value of 0 only after an infinite number of iterations. Hence, to speed up the process, in practice, empirical schedules are used. In view of the above discussion, we may consider the Exponential formula for temperature given below as empirical.

The Exponential Formula for temperature scheduling:

$$T_n = \alpha e^{\frac{G_{n-1}-G_{n-2}}{G_{n-1}-T}} \tag{3.1}$$

where,

T_n	=	temperature for updations for iteration n, (**temperature**)
G_{n-1}	=	Goodness at the end of iteration $n-1$, (**current-goodness**)
G_{n-2}	=	Goodness at the end of iteration $n-2$, (**previous-goodness**)
Γ	=	Target goodness, (**target-goodness**)
	=	Maximum goodness achieved so far (**maximum-goodness-so-far**) in any iteration + a fixed constant C
α	=	a constant equal to the desired temperature when change in goodness is 0,(**alpha**)

It may be noted that $G_{n-1} - G_{n-2}$ = Change in goodness, (**change-in-goodness**). The letters in bold-face give the names of the variables as used in the Simulated Annealing Algorithm that uses the above formula for temperature schedule. This algorithm, given below, is in Pseudo-C and is self-explanatory. (Details of the actual C language code used for simulation studies are given in Appendix A and the code is listed at the end of the Thesis.)

```
main()
  initialize arrays best-state-so-far[] and current-state[];
  previous-goodness = 0;
  current-goodness = 0;
  change-in-goodness = 0;
  maximum-goodness-so-far = -∞;
  target-goodness = -∞;
  while (certain stop condition is not true)
    update-temperature();
    update-system-state();
    calculate-current-goodness();
    change-in-goodness = current-goodness - previous-goodness;
    previous-goodness = current-goodness;
    if (maximum-goodness-so-far < current-goodness)
```

```
        maximum-goodness-so-far = current-goodness;
        store contents of current-state[] in best-state-so-far[];
        target-goodness = maximum-goodness-so-far + C;
      end-if
    end-while
    output best-state-so-far[] as the final result state;
    output maximum-goodness-so-far as the final goodness;
end-main
```

The function **update-temperature()** computes the temperature essentially using the formula given above.

```
update-temperature()
temperature = α × exp((change-in-goodness)
/(current-goodness - target-goodness));
end-update-temperature
```

The function **update-system-state()** can be written as:

```
update-system-state()
select a unit i randomly;
calculate the netinput to unit i;
if(probability(logistic(netinput)) = 1)
current-state[i] = maximum value allowed for the unit;
else
current-state[i] = minimum value allowed for the unit;
end-update-system-state
```

It may be noted that some minor variations have to be applied in the function **update-system-state()** when this is used for Harmony Machine as discussed in chapter 2. First, each unit of one layer must be updated first before

taking up the units of the other layer. Second, the equations for calculating the **netinput** are different for representational feature units and knowledge atoms. Third, maximum and minimum values allowed for a Boltzmann Machine are 1 and 0 respectively; and, in a Harmony Machine they are 1 and 0 respectively for knowledge atoms, and +1 and −1 respectively for representational feature units.

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

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

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);
end-probability
```

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

The function **calculate-current-goodness()** can be computed using the equations given in Chapter 2 keeping in view the minor variations between Boltzmann Machine and Harmony Machine.

3.2 The dynamics of the Exponential Formula

We can now make the following observations about our new Exponential formula for Temperature Scheduling.

1. As goodness is equivalent to Harmony of Harmony Machines and negative of Energy of Boltzmann Machines, the objective of simulated annealing is to

maximize this goodness.

2. As the **target-goodness** is defined as **maximum-goodness-so-far** + a positive constant C, **target-goodness** is always greater than the **current-goodness** in the denominator of the power of the exponent in Equation 3.1. Hence, the denominator is always negative. The significance of the constant C will be explained later.

3. Whenever the numerator of the exponent, i.e. **change-in-goodness**, is 0, the temperature is equal to **alpha**.

Now, it is easy to see why the exponential formula satisfies all the desired properties discussed at the beginning of this chapter.

1. From Equation 3.1 it can be seen that whenever **change-in-goodness** is positive, the power of the exponent will be a large value with a negative sign if the **change-in-goodness** is large OR **current-goodness** is approaching **target-goodness**, (i.e. **current-goodness - target-goodness** is very near 0). Hence, the temperature will be very small. This is very desirable because by keeping the temperature small, we are reducing the randomness in state transition decisions and encouraging the system to continue the trend of going up-hill in the goodness space.

2. However, after certain number of iterations during which the system makes uphill moves in the goodness space, the system may get stuck in a local goodness maximum where the (positive) **change-in-goodness** is very small or zero. It can be seen from the equation that whenever **change-in-goodness** is small or approaching 0, the temperature tends to be large. Hence, the system will make a random state transition, and will escape from the local goodness maximum. It can also be seen that after the first state transition from the local goodness maximum, the **change-in-goodness** will be negative. The temperature

will continue to be high as long as the magnitude of **current-goodness - target-goodness** is small compared to the magnitude of **change-in-goodness.** This is desirable as this will generate a series of random state transitions making the system come out of the local goodness maximum.

3. When the **change-in-goodness** is negative, the power of the exponent will be large initially, as long as the magnitude of **current-goodness - target-goodness** is small compared to the magnitude of the **change-in-goodness.** However, after a certain number of state transitions downhill the goodness space, the magnitude of **current-goodness - target-goodness** will be much larger than magnitude of the **change-in-goodness** and at this stage the temperature will tend to become small. This will generate a series of state transitions that will tend to make the system go uphill again.

4. It can also be seen that whenever the **change-in-goodness** is positive, the temperature is below α, and whenever the **change-in-goodness** is negative, the temperature is above α. In general, the temperature oscillates about the value α as a function of iterations.

The significance of **target-goodness** is as follows. It is defined as **target-goodness** = **maximum-goodness-so-far** + **C**, where C is an arbitrary positive constant greater than 0. If the magnitude of the denominator in the power of the exponent (Equation 3.1) is small, the temperature will be small if the **change-in-goodness** is positive, and the temperature will be high if the **change-in-goodness** is negative. This is desirable as discussed above, and can be achieved by continuously evaluating the value of **target-goodness** from the **maximum-goodness-so-far** rather than setting it a rough estimate of the final achievable goodness. (Contrast this approach with many proposals[1],[3],[10] on annealing schedules where a rough estimate of optimum value of the cost-function is made.)

This also eliminates the need for making an estimate of the achievable goodness. The constant C is required to keep the difference **current-goodness - target-goodness** a negative non-zero number; otherwise, when **current-goodness** is also the **maximum-goodness-so-far**, the denominator in the power of the exponent will be 0, in which case the temperature will be unpredictable.

The Experimental results discussed in Chapter 4 amply demonstrate the validity of the above discussion. It is shown that the temperature oscillates as a function of iterations, and, as a result, the goodness also oscillates. This can be given the interpretation that the system always attempts to reach a new goodness peak from a new random starting point whenever it is stuck in a local goodness peak. The experimental results also show that this *chasing after* a new goodness peak makes the system visit the global maximum extremely quickly as compared to the linear cooling schedule. This also removes in one stroke, the possibility of the system getting stuck in a local maximum. But, by the same token, the system will not freeze when it reaches the global maximum but will continue to oscillate. However, this drawback can be easily overcome by storing in memory the maximum value of the goodness obtained during the iterative process and the corresponding system state. This can be trivially implemented as given in the algorithm above.

3.3 A discussion of other relevant schedules

The idea of dynamically varying the temperature as a function of the quality(good or detrimental) of state transitions is based on an earlier idea by Bohachevsky et. al. called Generalized Simulated Annealing (GSA) [1]. In this original paper on GSA [1], it has been described how GSA can be used for function optimization (not specific to connectionist systems). The state transition rule in GSA, when

suitably modified to be in correspondence for the Boltzmann Machine takes the following form:

$$probability(a_i = 1) = \frac{1}{1 + e^{\frac{\beta net_i}{G_u - E}}} \tag{3.2}$$

where, G_u equals the goodness at the end of the previous update (not the previous iteration), E equals estimated maximum goodness, β equals a positive constant to be determined empirically based on the problem, net_i is the net input to the unit i and, a_i is the activation of unit i. As, E is the estimated maximum goodness, the value of $G_u - E$ is always negative (expecting that the estimate is reasonable). It could be seen from this equation, that if net_i is positive, the value of the exponential is very small if net_i is large or as G_u approaches E. Hence, the value of logistic function approaches 1, which implies that activation of unit i will very likely be 1. On the other hand, if net_i is negative, the activation of the unit tends to be 0, as G_u approaches E. Hence, the system tends to be deterministic as the goodness approaches the estimated maximum. At other times, the system behaves probabilistically, and can escape from local maximum. The factor β acts as an accelerator as it amplifies the effect of net_i. Thus, in general, the dynamics of the system using exponential formula of this chapter are somewhat similar to that of GSA.

However, there are three main drawbacks of GSA:

1. It would be very impractical to apply the above rule to a Boltzmann Machine as it would require calculating the goodness after the update of every unit in a system (Contrast this with the algorithm using exponential formula for temperature where it is required to calculate goodness only at the end of every iteration). This would result in much greater computational time if the system has a large network of units.

2. The success of the GSA depends on a good estimate of the maximum

goodness. The system may prematurely freeze if the estimate is below the real maximum, and the system may never freeze if the estimate is above the real maximum. Moreover, the task of making an estimate may not always be easy.

3. If the system is stuck in a local maximum very near to the global maximum, it may take a very long time for the system to come out of the local maximum as G_u is very near to E.

Our algorithm using the exponential formula overcomes all the drawbacks above while retaining the desirable properties of GSA.

In connection with this discussion, it is also pertinent to make a mention of a temperature schedule somewhat similar to our schedule. In Reference[3], Enrique C.S et. al. briefly discuss the result of using a schedule of the type $\epsilon^{\frac{G_{n-1}-G_{n-2}}{G_{n-1}}}$ for a function optimization (not based on connectionist architecture) which they term as a failure. Here, $G_{n-1} - G_{n-2}$ gives the change in goodness, and G_{n-1} gives the current goodness. The failure and limitations of this schedule can be attributed to the following: First, this schedule works only if the global maximum is known *a priori* to be equal to 0. Second, there is no multiplicative factor (like α of our schedule) which binds the upper value of temperature for positive changes in goodness.

Chapter 4

Experimental Results

The new temperature scheduling algorithm has been applied to three different examples, and the results are compared with the results obtained with the traditional linear cooling schedule explained in Chapter 2. All the three examples were so chosen that they were sufficiently well studied with traditional methods and are representative of the problems in the domain. This is done in order to make the comparison of the results significant and meaningful.

The three examples are: 1. The Electricity Problem[2] of Harmony Machine, 2. The Graph Partitioning Problem[2] of the Boltzmann Machine, and 3. The Necker Cube Problem[2] of Boltzmann Machine. All the three problems are well studied with traditional cooling methods and well discussed in literature. Of all the three problems, the Electricity problem is the most versatile as it has many combinations of input-output states, and also its goodness space has many local maxima. This example is described first below.

4.1 The Electricity Problem

This problem was first developed by Riley and Smolensky[7] to illustrate how harmony Machine can be employed to imitate on a macro level the human intuitive problem solving. The problem is to determine how different variables in

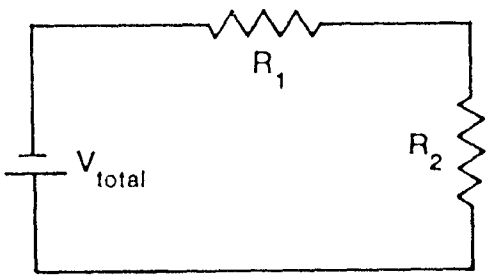


Figure 4.1: The Electricity Problem of Harmony Theory

an electrical circuit change when other variables are altered. For example, in a series resistive circuit consisting of two resistors R_1 and R_2 and a Voltage source V, what happens to the current and voltage drops across the two resistors if the resistance of R_2 is increased? The Harmony Machine network for this problem consists of 14 representational feature units, 65 knowledge atoms and connections between them encoding the qualitative knowledge about the electrical circuits. The strengths of all the knowledge atoms are 1. As each knowledge-atom is for a right consistent combination of representational features, the goodness will be maximum when all the knowledge atoms that are consistent with the given input constraints are on. (For more details on this problem readers may refer Appendix B)

For the Electricity Problem of Harmony theory and the two subsequent problems given below, simulated annealing was applied 100 times each with the exponential cooling schedule and with the linear cooling schedule. Each trial (or

each run of the simulated annealing) is started with a different random seed. The simulation results for each problem are given in a separate table containing all the relevant data. The results of the simulation of the Electricity problem are given in Table 4.1.

The results indicate that the Exponential Schedule is more than five times faster than the Linear Schedule in terms of the system visiting the global maximum (for this problem). Moreover, in 100 trials, the system never got stuck in local maximum with the Exponential Schedule. This greatly strengthens the argument forwarded in Chapter 3, that, with Exponential Schedule, the system can always escape from local maximum as the temperature varies dynamically as a function of goodness. This can be compared with the system's performance with the linear schedule where the system was stuck in local maximum 13 times. The other interesting observation was that with the Exponential Schedule, during certain trials, the system could visit global maximum in as early as 10 iterations; with the Linear schedule the fastest convergence in 100 trials was with 180 iterations.

4.2 The Necker Cube Problem

To understand this problem, let us consider a line diagram of a cube drawn on a paper. A viewer can perceive this cube in exactly two different ways. That means each vertex can be interpreted in exactly two different ways. For example, a vertex that can be perceived as being at the Front Lower Right corner of the cube can also be perceived as being at the Back Lower Right corner of the cube. Since there are 8 vertices and each vertex can be interpreted in two different ways, there are 16 units in the Boltzmann Machine Network. However, there must be negative connection between the two units that give different interpretations of

Simulation Results
The Electricity
Problem of Harmony Machine

Scheduling Method	Number of Trials	Number of times stuck in Local Maxima	Average Number of Iterations to reach Global Maximum (= 125)
Linear *coolrate=.00475* *Initial temp = 1*	100	13	251
Exponential Formula *alpha = .2* *C = .1*	100	0	49.33

Table 4.1: Simulation results of Electricity Problem

same vertex as a viewer cannot perceive in both ways at the same time. In other words, there are positive connections between a unit and it's consistent neighbours. (For more details on this problem readers may refer Appendix B) Hence, when simulated annealing is performed, the system will achieve maximum goodness when all units representing one consistent interpretation of the Necker Cube are turned on. The results are given in Table 4.2.

As in the case of The Electricity Problem, the Exponential Formula proved to be much faster than the Linear Schedule. In 100 trials, the system was never stuck in local maximum with the Exponential Schedule, whereas with the linear schedule, the system was stuck in local maximum 14 times. The fastest convergence with the Exponential Schedule was in 3 iterations, whereas with the Linear Schedule it was for 17 iterations.

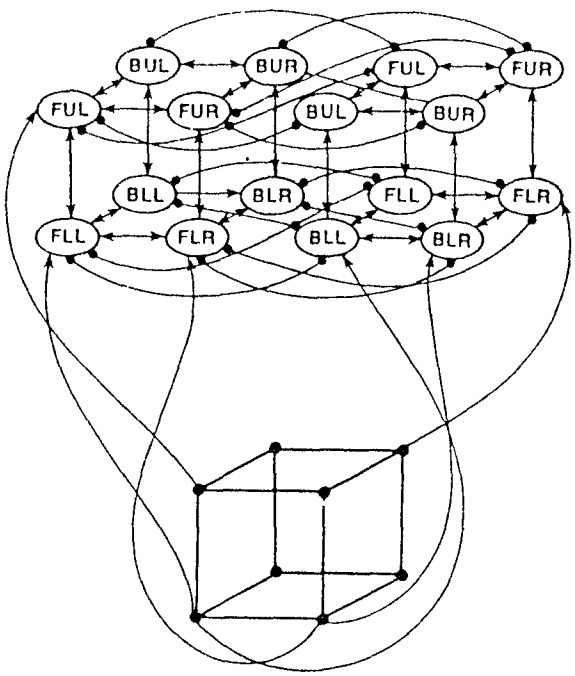


Figure 4.2: The Necker cube of Boltzmann Machine

Simulation Results

The Necker Cube
Problem of Boltzmann Machine

Scheduling Method	Number of Trials	Number of times stuck in Local Maxima	Average Number of Iterations to reach Global Maximum (= 6.40)
Linear *coolrate= 0975* *Initial Temp = 2*	100	14	18.8
Exponential Formula *alpha = 2* *C = 1*	100	0	11

Table 4.2: Simulation results of Necker Cube

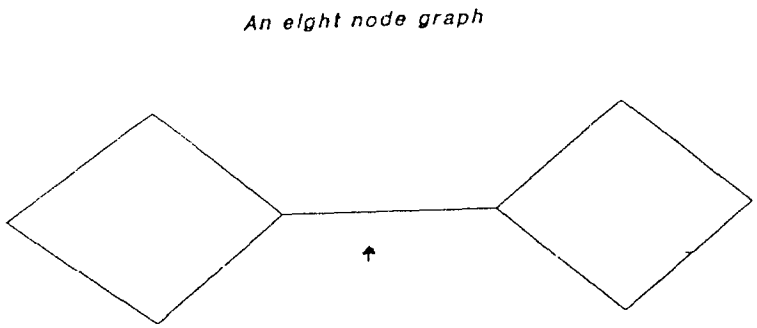


Figure 4.3: The Graph Partitioning Problem of Boltzmann Machine

4.3 The Graph Partitioning Problem

This problem can be stated as this: Given a connected graph of n nodes, each of which is connected to one or more other nodes in the network, divide the graph into two parts with half of the nodes in each while minimizing the number of links that connect nodes from the two different classes. Simulations are performed on an 8 node graph. (For more details on this problem readers may refer Appendix B) Simulation results are given in Table 4.3. The results demonstrate the superior performance of the Exponential Schedule compared to the Linear Schedule. The average number of iterations to reach the global maximum was only 10 with the exponential formula; it was 144 with the linear schedule. The system was never stuck in local maximum with the exponential formula. The fastest convergence was in 2 iterations with the exponential formula, whereas, it was in 112 iterations with the linear schedule.

Simulation Results
The Graph Partitioning
Problem of Boltzmann Machine

Scheduling Method	Number of Trials	Number of times stuck in Local Maxima	Average Number of Iterations to reach Global Maximum (= 3.20)
Linear *coolrate = .00975* *initial temp = 2*	100	7	144
Exponential Formula *alpha = .2* *C = .1*	100	0	10

Table 4.3: Simulation results of Graph Partitioning Problem

4.4 The oscillatory behaviour of temperature and goodness

We have discussed in Chapter 3, that temperature and goodness have an oscillatory variability as a function of iterations. This is demonstrated by means of graphs from the data of a sample run of the Electricity Problem of the Harmony Machine.

Figure 4.4 gives the graph of temperature versus iterations. Figure 4.5 gives the graph of **current-goodness** versus iterations. Figure 4.6 gives the temperature and goodness superposed versus iterations. Figure 4.6 shows that when temperature is low goodness tends to be high and vice versa.

These graphs coupled with simulation results clearly demonstrate that the exponential temperature schedule is greatly effective and superior compared to the linear schedule.

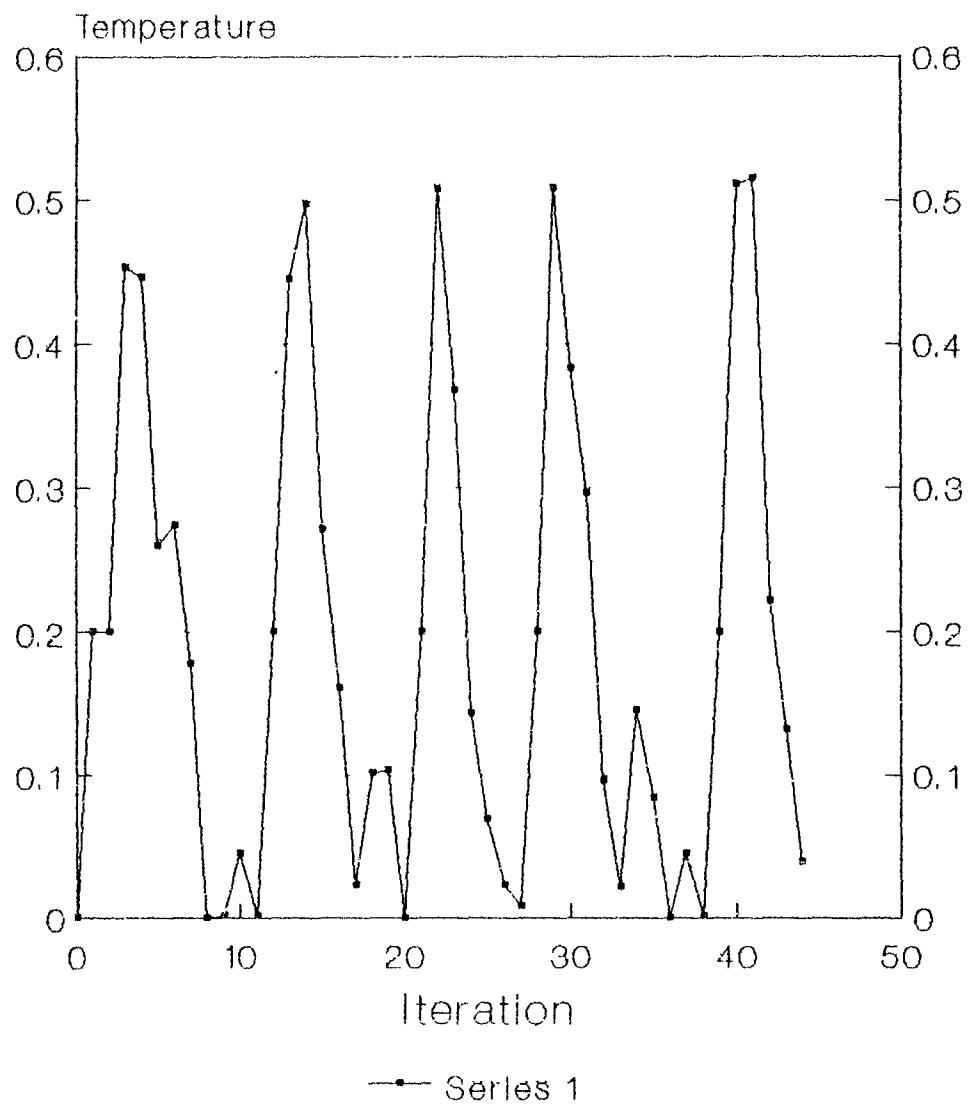


Figure 4.4: Iterations vs Temperature

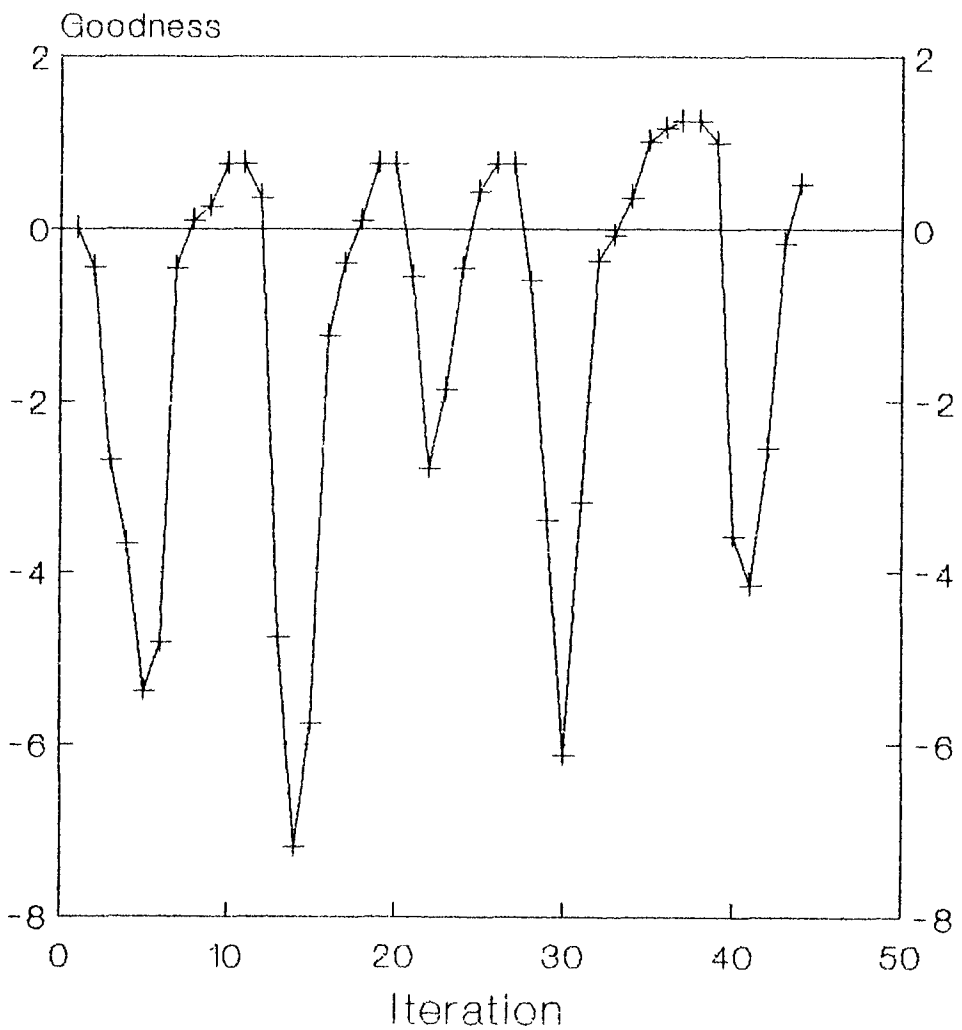


Figure 4.5: Iteration vs Goodness

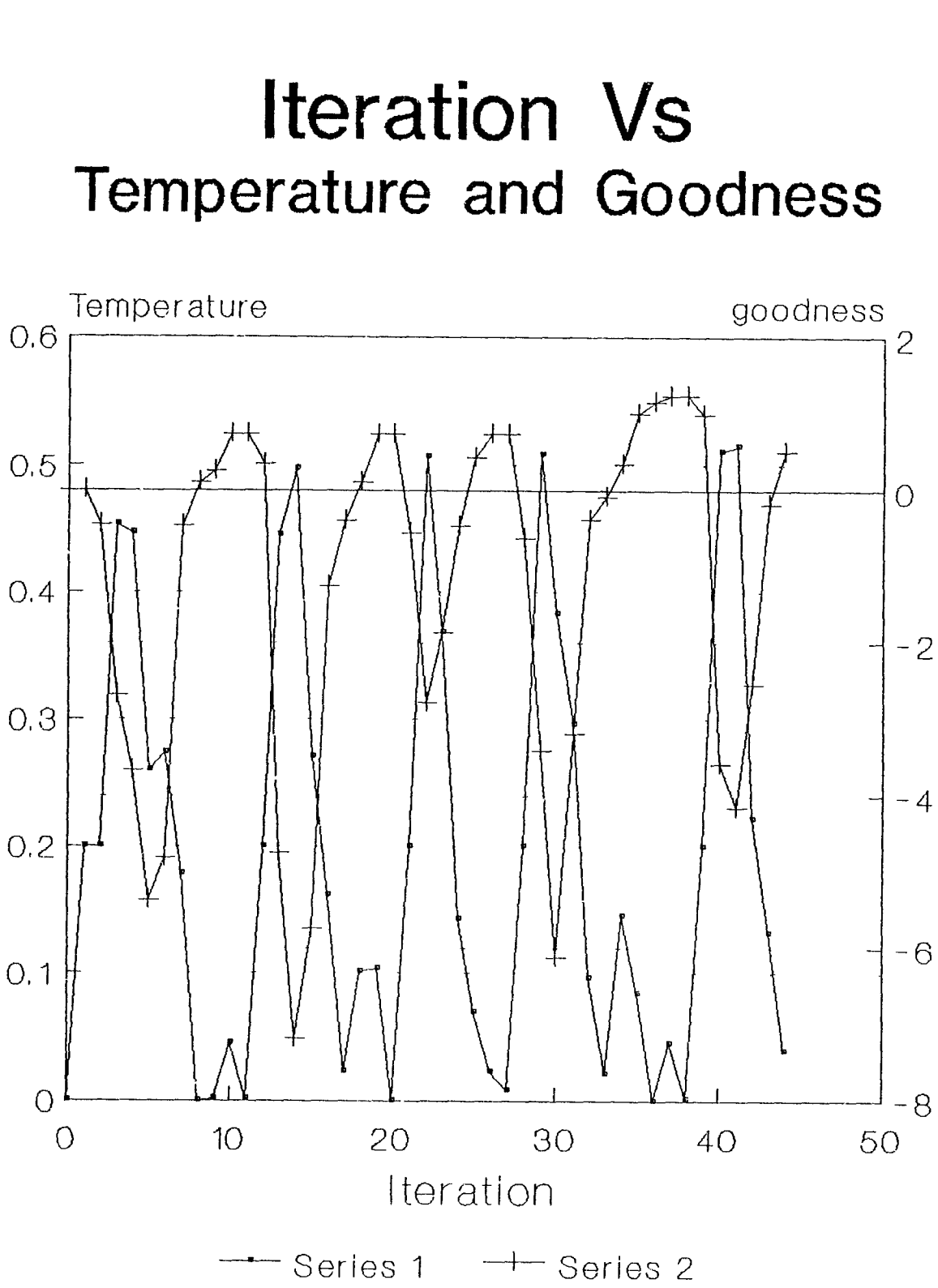


Figure 4.6: Iteration vs Temperature and Goodness

Chapter 5

Conclusions

The contributions of this thesis can be listed as follows:

1. A exponential temperature schedule that dynamically varies depending on the system state has been applied for the first time on Boltzmann Machines

2. The shortcoming of the Generalized Simulated Annealing(GSA)[1],and other annealing schedules (as given in [3], [6]) which require an estimate of the maximum goodness is not present in the new schedule.

3. The new temperature schedule can profitably be applied to any optimization problem (not necessarily Boltzmann Machines) as it overcomes the drawbacks of annealing methods like GSA which possess somewhat similar dynamics. (Refer to Chapter 3).

5.1 Further course of research

The following points are relevant to this topic and need further study.

1. As the system never freezes with the exponential temperature scheduling, it is required to keep in memory the maximum value obtained for goodness. Though the system never gets stuck in a local maximum, there is no way of judging if the maximum goodness obtained at any iteration is the global maximum. Hence, we can say that we do not have a stopping criterion that stops the itera-

tion process at global maximum (unless we know the global maximum *a priori*, as is the case in certain problems). This drawback can be partially overcome by empirically fixing a certain number of iterations during which time if there is no increase in the maximum value of goodness, we may stop iterating.

2. While the exponential temperature schedule is definitely superior to the linear schedule, a comparison with other cooling schedules is yet to be made.

3. The temperature equals the constant α when the **change-in-goodness** is zero. While a value of .2 was very satisfactory for the examples discussed, it may be possible to dynamically alter the value of α. For example, it may be desirable to increase the value of α by some means when previous goodness peak and the current goodness peak are the same. This deserves further study.

4. It has been discussed that the value of the positive constant C if chosen to be small gives better results. However, the precise effects of the value of C on the system performance may be more thoroughly studied.

Appendix A

Details of the Simulator Code

The code used for simulations in this thesis is in *C* language and can be divided into two parts. For simulating the Linear Cooling Schedule, we have used the software (the Constraint Satisfaction part) that is provided with Reference[2], called PDP software, without any modifications. For simulating the Exponential Temperature Scheduling, extensive modifications have been made to the basic algorithm of Constraint Satisfaction, i.e. *CS.C*, while retaining the basic structure and the display utilities of the PDP software. A listing of the modified version of *CS.C* along with comments is reproduced at the end of this Thesis, which may be referred for a general idea about the flow and structure of the main algorithm. The other utility and command interfacing programs are not modified and, hence, not reproduced here. For greater details about activating the program and command level interface, readers may refer to Reference[2]. As per the Licensing agreement of the PDP software, the Copyright notice of the PDP group has been retained in the modified version also.

Appendix B

Details of the Examples simulated

All the examples discussed in this thesis have been so chosen, that they have already been well studied with linear cooling schedule, and the results well documented. This is done in order to make the comparison of results meaningful and significant. As we are not dealing with learning, the network connectivity and the weights are fixed, and for all the three examples of this thesis, we have used the network configuration as discussed in the PDP software that comes with Reference[2]. For all the three examples, the network connectivity files, the weights files, the startup files and the display template files are included in the *Constraint Satisfaction* part of the PDP software. For example, for the Electricity Problem of Harmony Machine, all the files start with the name *vir* and have the extensions *.str* (for startup file), *.tem* (for template file), *.net* (for network file) and so on. Similarly, for the Necker Cube Problem, all the files start with the name *cube* and have corresponding extensions. For the Graph Partitioning Problem, all the files start with the name *boltz* and have the corresponding extensions. It may be noted that, while we have used the *.net* and *.wts* files unaltered, suitable modifications have been made in the *.tem* files to display and monitor the new parameters introduced by the Exponential formula.

Discussion of the problem statement and required connectivity for realising the network, and a thorough explanation of interpreting the conventions followed in each of the files above are given in Reference [2]. However, for the sake of reader's convenience, a brief discussion of the Electricity Problem of Harmony Machine is given here.

In the Electricity Problem, the objective is to determine how different variables in the electrical circuit change when other variables are altered. For example, what happens to the total resistance in the circuit (Figure 4.1) when the resistance of 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 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. +1 has been used to indicate an increase and -1 to indicate a decrease. (If no change occurred, the value of the second feature is irrelevant.) Figure B.1 shows the display screen layout for the electricity problem.

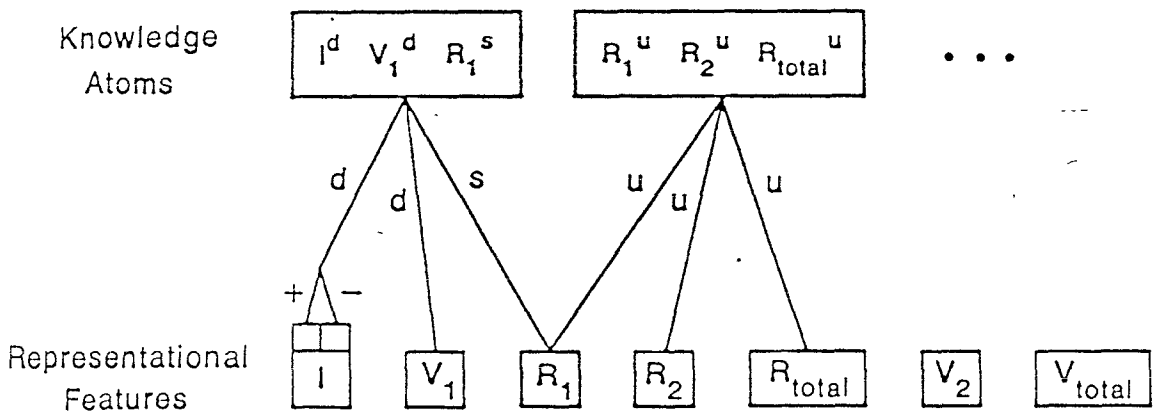
There are columns for each of the seven variables. Below each column is a set of pairs of features, one indicating whether or not that quantity changed (indicated by a c) and one indicating whether that quantity went up or down (indicated by a u). Note that the row labeled *Inputs* is a representation of the problem. The 0s indicate unclamped inputs - inputs to be filled in through pro-

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


                I  R1 R2 RT V1 V2 VT          cycleno       600
       Inputs   00 10 11 00 00 00 10          temp        0.0500
                cu cu cu cu cu cu cu          harmony     1.0000
       Features 11 11 11 11 11 11 11

                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 1 0 0 0 0 0 0
 R1 + R2 = RT   0 0 0 0 0 1 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 1 0 0 0
  I * RT = VT   0 0 0 0 0 0 0 0 0 1 0 0 0
```

Figure B.1: Screen layout for the electricity problem

cessing. The ±1 values indicate the clamped inputs, which constitute the problem specification. In the example 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 -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 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 B.1. 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. This is contained in the network specification file, i.e. *vir.net* which is reproduced at the end of this Thesis. In this file, *p* represents +1 and the *m* represents -1. Since the weights are symmetric, we only require that the connections from the feature units to the knowledge atoms be specified. The file *vir.str* and the modified version of the file *vir.tem* are also reproduced at the end of this Thesis. For a detailed discussion of the conventions and the terminology used, readers may refer [2].



A schematic diagram of the feature nodes and two knowledge atoms of the model of circuit analysis. *u*, *d*, and *s* denote *up*, *down*, and *same*. The box labeled *I* denotes the *pair* of binary feature nodes representing *I*, and similarly for the other six circuit variables. Each connection labeled *d* denotes a *pair* of connections labeled with the binary encoding (+,−) representing *down*, and similarly for connections labeled *u* and *s*.

Figure B.2: Harmony network for the Electricity Problem

Bibliography

[1] I. Bohachevsky, M. Johnson, and M. Stein, "Generalized Simulated Annealing for function optimization," *Technometrics*, Vol.28, No 3, 209 - 217, 1986.

[2] J.L. McClelland and D.E. Rumelhart, "Constraint Satisfaction in PDP Systems," *Explorations in Parallel Distributed Processing, A Handbook of Models, Programs, and Exercises*, MIT Press, Cambridge, MA, 1988.

[3] C.S. Enrique, C.F. Bruno, "Self Learning Simulated Annealing," *Proceedings of IJCNN90*, Vol. 1, I463-I467, January, 1990.

[4] L. Xu,"An improvement on simulated Annealing and Boltzmann Machine," *Proceedings of IJCNN90*, Vol. 1, I341 -I344, January, 1990.

[5] G.E. Hinton, T.J. Sejnowski, "Learning and Relearning in Boltzmann Machines," in *Parallel distributed Processing: Explorations in the Microstructure of cognition*, J.L. McClelland, D.E. Rumelhart, ed., vol. 1, MIT press, Cambridge, MA, 1986.

[6] E.H.L. Aarts, J.H.M. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley & Sons, New York, NY, 1986.

[7] P. Smolensky, "Information Processing in Dynamical Systems: Foundations of Harmony Theory," in *Parallel Distributed Processing: Explorations in the*

Microstructure of cognition, J.L. McClelland, D.E. Rumelhart, ed., Vol. 1, MIT press, Cambridge, MA, 1986.

[8] M. Minsky, S. Papert, *Perceptrons*, MIT press, Cambridge, MA, 1969.

[9] J.J. Hopfield,"Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences, USA*, 79,2554-2558, 1982.

[10] R. Richards, "An Efficient Algorithm for Annealing Schedules in Boltzmann Machines," *Proceedings of IJCNN90*, vol 1, I309-I312, January, 1990.

[11] S. Geman, D. Geman, "Stochastic Relaxation, Gibbs Distributions and the Baysian Restoration of Images," *IEEE Transactions of Pattern Analysis and Machine Intelligence*. 6, 721-741, 1984.

[12] H. Szu, "Fast Simulated Annealing," *Neural Networks Conference Proceedings*, AIP, Vol 151, 420-425, Denker ed., 1986.

```
/*
-------------------------------------------------------------------------
   This is the modified version of Constraint Satisfaction Program
   developed for Exponential Temperature Schedule. Copyright notice
   of the PDP software is retained as per the License notice.

                        Rajendra Sarasa K.
-------------------------------------------------------------------------


       This file is part of the PDP software package.

       Copyright 1987 by James L. McClelland and David E. Rumelhart.

       Please refer to licensing information in the file license.txt,
       which is in the same directory with this source file and is
       included here by reference.
 */


 /* file: cs.c

           Do the actual work for the cs program.

          First version implemented by Elliot Jaffe.

           Date of last revision:  8-12-87/JLM.
 */

 #include "general.h"
 #include "cs.h"
 #include "variable.h"
 #include "command.h"
 #include "patterns.h"
 #include "weights.h"

 #define     MAXTIMES        20
 #define  FMIN (1.0e-37)
 #define  fcheck(x) (fabs(x) > FMIN ? (float) x : (float) 0.0)

 char   *Prompt = "cs: ";
 char   *Default_step_string = "cycle";
 boolean System_Defined = FALSE;

 boolean     clamp = 0;
 boolean        boltzmann = 0;
 boolean        harmony = 0;

    float       temperature,coolrate,cur_temp;
    float       goodness,maxgood;
    float       fgoodness;
    float       dgoodness;
    float       pgoodness;
    float       deltagood;


    float  *activation;
    float  *netinput;
    float  *intinput;
    float  *extinput;
    float  *factivation;
    float  *dactivation;

    float   estr = 1.0;
    float   istr = 1.0;
```

```
    float   kappa;

    int     epochno = 0; /* not used in cs */
    int     patno = 0;
    int     ncycles = 10;
    int     nupdates = 100;
    int     cycleno = 0;
    int     updateno = 0;
    int     unitno = 0;
    char    cuname[40];
    int     ntimes = 0;

   struct anneal_schedule {
    int     time;
    float   temp;
   }  *anneal_schedule;

 int maxtimes = MAXTIMES;

 struct anneal_schedule *last_temp;
 struct anneal_schedule *next_temp;
 struct anneal_schedule *current_temp;

define_system() {
int     i,j;
float   *tmp;

activation = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("activation",Vfloat,(int *)activation,nunits,0,
                                                        SETSVMENU);
for (i = 0; i < nunits; i++)
    activation[i] = 0.0;

factivation = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("factivation",Vfloat,(int *)factivation,nunits,0,
                                                    SETSVMENU);

for (i = 0; i < nunits; i++)
    factivation[i] = 0.0;


dactivation = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("dactivation",Vfloat,(int *)dactivation,nunits,0,
                                                    SETSVMENU);

for (i = 0; i < nunits; i++)
    dactivation[i] = 0.0;

netinput = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("netinput",Vfloat,(int *)netinput,nunits,0,SETSVMENU);
for (i = 0; i < nunits; i++)
    netinput[i] = 0.0;

intinput = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("intinput",Vfloat,(int *)intinput,nunits,0,SETSVMENU);
for (i = 0; i < nunits; i++)
    intinput[i] = 0.0;

extinput = (float *) emalloc((unsigned)(sizeof(float) * nunits));
(void)install_var("extinput",Vfloat,(int *)extinput,nunits,0,SETSVMENU);
for (i = 0; i < nunits; i++)
    extinput[i] = 0.0;

anneal_schedule = ((struct anneal_schedule *)
          emalloc((unsigned)maxtimes*sizeof(struct anneal_schedule)));
```

```
next_temp = anneal_schedule;
last_temp = anneal_schedule;
last_temp->time = 0;
last_temp->temp = 0.0;

constrain_weights();

System_Defined = TRUE;
reset_system();
return(TRUE);
}


double  logistic (i) float   i; {
double  val;
double  ret_val;
double  exp ();

if( temperature <= 0.0)
    return(i > 0);
else
    val = i / temperature;

if (val > 11.5129)
    return(.99999);
else
    if (val < -11.5129)
        return(.00001);
else
        ret_val = 1.0 / (1.0 + exp(-1.0 * val));
if (ret_val > FMIN) return(ret_val);
return(0.0);
}

probability(val) double  val; {
return((rnd() < val) ? 1 : 0);
}

float
annealing1 ( iter) int     iter; {
/* COMPUTE THE TEMPERATURE AS PER THE NEW EXPONENTIAL FORMULA */

double tmp;

tmp = .2*exp(deltagood/(goodness - maxgood));

if (tmp >= 1.0) return(1.0);
return((tmp < FMIN ? 0.0 : tmp));
}

get_schedule() {   /* THIS FUNCTION IS NOT USED IN MODIFIED VERSION*/
int  cnt;
char  *str;
char   string[40];
struct anneal_schedule *asptr;

if(!System_Defined)
   if(!define_system())
      return(BREAK);

asptr = anneal_schedule;
next_temp = anneal_schedule;

restart:
str = get_command("Setting annealing schedule, initial temperature : ");
```

```
    if( str == NULL) return(CONTINUE);
    if(sscanf(str,"%f",&(asptr->temp)) == 0) {
       if (put_error("Invalid initial temperature specification.") == BREAK) {
        return(BREAK);
       }
       goto restart;
    }
    if(asptr->temp < 0) {
       if (put_error("Temperatures must be positive.") == BREAK) {
        return(BREAK);
       }
       goto restart;
    }
    cnt = 1;
    last_temp = asptr++;
    sprintf(string,"time for first milestone:  ");
    while((str = get_command(string)) != NULL) {
       if(strcmp(str,"end") == 0) return(CONTINUE);
       if (cnt >= maxtimes) {
          maxtimes += 10;
          anneal_schedule = ((struct anneal_schedule *)
                       erealloc((char *)anneal_schedule,
                     (unsigned)((maxtimes-10)*sizeof(struct anneal_schedule)),
                       (unsigned)maxtimes*sizeof(struct anneal_schedule)));
          next_temp = anneal_schedule;
          asptr = anneal_schedule + cnt;
          last_temp = asptr;
       }
       if (sscanf(str,"%d",&asptr->time) == 0) {
          if (put_error("Non_numeric time. ") == BREAK) {
           return(BREAK);
          }
          continue;
       }
       if(asptr->time <= last_temp->time) {
          if (put_error("Times must increase.") == BREAK) {
           return(BREAK);
          }
          continue;
       }
       sprintf(string,"at time %d the temp should be: ",asptr->time);
       if((str = get_command(string)) == NULL) {
          if (put_error("Nothing set at this milestone.") == BREAK) {
           return(BREAK);
          }
          goto retry;
       }
       if(sscanf(str,"%f",&(asptr->temp)) == 0) {
          if(put_error("Non_numberic temperature.") == BREAK) {
           return(BREAK);
          }
          goto retry;
       }
       if(asptr->temp < 0) {
          if(put_error("Temperatures must be positive.") == BREAK) {
           return(BREAK);
          }
          goto retry;
       }
       last_temp = asptr++;
       cnt++;
     retry:
       sprintf(string,"time for milestone %d: ",cnt);
    }
    return(CONTINUE);
    }
```

```
/*THE HEART OF THE CS PROGRAM. EXTENSIVELY MODIFIED*/
cycle() {
int     iter, i;
char        *str;

if (!System_Defined)
    if (!define_system())
        return(BREAK);



for (iter = 0; iter < ncycles; iter++) {
    cycleno++;
    if(boltzmann || harmony)
        temperature = annealing1(cycleno);
    if (rupdate() == BREAK) return(BREAK);
       get_goodness();
       deltagood = goodness - pgoodness;
       pgoodness = goodness;
       if (goodness >= maxgood)
             maxgood = goodness + .1;
       cur_temp = annealing1(cycleno);
      if(fgoodness < goodness) {
       fgoodness = goodness;
         for (i = 0; i < nunits; i++)
        factivation[i] = activation[i];
       }

    if(step_size == CYCLE) {
     dgoodness = goodness;
     for (i = 0; i < nunits; i++)
       dactivation[i] = activation[i];
       cs_update_display();
       if(single_flag) {
          if (contin_test() == BREAK){
          dgoodness = fgoodness;
          for (i = 0; i < nunits: i++)
           dactivation[i] = factivation[i];
          cs_update_display();
          return(BREAK);
          }


       }
    }
    if (Interrupt) {
        get_goodness();
        if(fgoodness < goodness) {
         fgoodness = goodness;
         for (i = 0; i < nunits; i++)
           factivation[i] = activation[i];
        }
        dgoodness = goodness;
        for (i = 0; i < nunits; i++)
         dactivation[i] = activation[i];
        cs_update_display();
        Interrupt_flag = 0;

        if (contin_test() == BREAK){
         dgoodness = fgoodness;
         for (i = 0; i < nunits; i++)
          dactivation[i] = factivation[i];
         cs_update_display();
         return(BREAK);
        }
```

```
        }
    }
    if (step_size == NCYCLES) {
            dgoodness = goodness;
            for (i = 0; i < nunits; i++)
             dactivation[i] = activation[i];
            cs_update_display();
     }
        dgoodness = fgoodness;
        for (i = 0; i < nunits; i++)
         dactivation[i] = factivation[i];
        cs_update_display();


        return(CONTINUE);
    }

    get_goodness() {

        int i,j;
        int num, sender, fs, ls; /* fs is first sender, ls is last */
        double dg;

        dg = 0.0;

        if(harmony) {
            for(i=ninputs; i < nunits; i++) {
               sender = first_weight_to[i];
               num = num_weights_to[i];
               for(j = 0; j < num && sender < ninputs; j++,sender++) {
                   dg +=
                       weight[i][j]*activation[i]*activation[sender];
               }
               if (activation[i]) dg -= kappa*sigma[i];
            }
            goto ret_goodness;
        }
        for(i=0; i < nunits; i++) {
           fs = first_weight_to[i];
           ls = num_weights_to[i] + fs -1;
           for(j = i+1; j < nunits; j++) {
                if ( j < fs ) continue;
                if ( j > ls ) break;
                dg += weight[i][j-fs]*activation[i]*activation[j];
           }
           dg += bias[i]*activation[i];
        }
     /* >> dont we want to let goodness be affected by istr whether or not
     clamp is 0? Boltz is always clamped, but not schema */
        if(clamp == 0) {
            dg *= istr;
            for(i=0; i < nunits; i++) {
                dg += activation[i]*extinput[i]*estr;
            }
        }
     ret_goodness:
        goodness = dg;
        return;
     }

    constrain_weights() {
    int    *nconnections;
    int     i,j,num;
    float   value;

    if(!harmony) return;
```

```
    nconnections = (int *) emalloc((unsigned)(sizeof(int) * nunits));
    for (i = 0; i < nunits; i++) {
        nconnections[i] = 0;
    }

    for (j = ninputs; j < nunits; j++) {
        num = num_weights_to[j];
        for (i = 0; i < num; i++) {
            if (weight[j][i])
                nconnections[j]++;
        }
    }

    for (j = ninputs; j < nunits; j++) {
        if (!nconnections[j])
            continue;
        value = sigma[j] / (float) nconnections[j];
        num = num_weights_to[j];
        for (i = 0; i < num; i++) {
            if (weight[j][i]) {
                weight[j][i] *= value;
            }
        }
    }
    free((char *)nconnections);
    }
    /* INITIALISE fgoodness, dgoodness, maxgood etc*/
    zarrays() {
    register int    i;

    if (!System_Defined)
        if (!define_system())
            return(BREAK);

    cycleno = 0;
    coolrate = 0;
    next_temp = anneal_schedule;
    temperature = next_temp->temp;
    cur_temp = temperature;
    if(last_temp != next_temp) {
        current_temp = next_temp++;
        coolrate =
                (current_temp->temp-next_temp->temp)/(float)next_temp->time;
    }
   maxgood = -100;
   goodness = 0;
    fgoodness = -100;
    dgoodness = 0;
    updateno = 0;
    pgoodness = 0;
    deltagood = 0;
    for (i = 0; i < nunits; i++) {
        intinput[i] = netinput[i] = activation[i] = factivation[i] = 0;
        dactivation[i] = 0;
    }
    if (clamp) {
        init_activations();
    }
    return(CONTINUE);
    }

    init_activations() {
    register int i;
    for (i = 0; i < nunits; i++) {
        if (extinput[i] == 1.0) {
```

```
            activation[i] = 1.0;
            continue;
        }
        if (extinput[i] == -1.0) {
            activation[i] == 0.0;
            continue;
        }
    }
    }

    rupdate() {
    register int     j,wi,sender,num,*fwp,*nwp,i,n,k;
    char *str;
    double dt, inti,neti,acti;

    for (updateno = 0,n = 0; n < nupdates; n++) {
        updateno++;
        unitno = i = randint(0, nunits - 1);
        inti = 0.0;
        if (harmony) {
             neti = 0.0;
             if (i < ninputs) {
                if (extinput[i] == 0.0) {
                   for (j = ninputs,fwp = &first_weight_to[ninputs],
                                    nwp = &num_weights_to[ninputs];
                                    j < nunits; j++) {
                        wi = i - *fwp++;
                        if ( (wi >= *nwp++) || (wi < 0) ) continue;
                        neti += activation[j]*weight[j][wi];
                   }
                   neti = 2 * neti;
                   if (probability(logistic(neti)))
                        activation[i] = 1;
                   else
                        activation[i] = -1;
                }
                else {
                   if(extinput[i] < 0.0) activation[i] = -1;
                   if(extinput[i] > 0.0) activation[i] = 1;
                }
             }
             else {
               sender = first_weight_to[i];
               num = num_weights_to[i];
               for (j = 0; j < num && sender < ninputs; j++,sender++) {
                      neti += activation[sender]*weight[i][j];
                }
                neti -=  sigma[i]*kappa;
                activation[i] = probability(logistic(neti));
                netinput[i] = neti;
             }
        }
        else {
          if (clamp) {
            if (extinput[i] > 0.0) {
                activation[i] = 1.0;
                goto end_of_rupdate;
            }
            if (extinput[i] < 0.0) {
                activation[i] = 0.0;
                goto end_of_rupdate;
            }
          }
          sender = first_weight_to[i];
          num = num_weights_to[i];
          for (j = 0; j < num; j++) {
```

```
        inti += activation[sender++] * weight[i][j];
      }
      inti  += bias[i];
      if (clamp == 0) {
        neti = istr * inti + estr * extinput[i];
      }
      else {
        neti = istr * inti;
      }
      netinput[i] = neti;
      intinput[i] = inti;
      if (boltzmann) {
        if (probability(logistic(neti)))
            activation[i] = 1.0;
        else
            activation[i] = 0.0;
      }
      else {
        if (neti > 0.0) {
          if (activation[i] < 1.0) {
            acti = activation[i];
            dt = acti + neti*(1.0 - acti);
            if (dt > 1.0) {
                activation[i] = (float) 1.0;
            }
            else activation[i] = (float) dt;
          }
        }
        else {
          if (activation[i] > (float) 0.0) {
            acti = activation[i];
            dt = acti + neti * acti;
            if (dt < FMIN) {
                activation[i] = (float) 0.0;
            }
            else activation[i] = (float) dt;
          }
        }
      }
    }
  end_of_rupdate:
    if(step_size == UPDATE)   {
       get_goodness();
       if (fgoodness < goodness) {
         fgoodness = goodness;
         for (k = 0; k < nunits; k++)
          factivation[k] = activation[k];
         }
       dgoodness = goodness;
       for (k = 0; k < nunits; k++)
        dactivation[k] = activation[k];
       cs_update_display();
       if (single_flag) {
          if (contin_test() == BREAK) {
            dgoodness = fgoodness;
            for (k = 0; k < nunits; k++)
              dactivation[k] = factivation[k];
            cs_update_display();
            return(BREAK);
          }
       }
    }
    if(Interrupt)   {
       Interrupt_flag = 0;
       get_goodness();
       if (fgoodness < goodness) {
```

```
        inti += activation[sender++] * weight[i][j];
      }
      inti  += bias[i];
      if (clamp == 0) {
        neti = istr * inti + estr * extinput[i];
      }
      else {
        neti = istr * inti;
      }
      netinput[i] = neti;
      intinput[i] = inti;
      if (boltzmann) {
        if (probability(logistic(neti)))
            activation[i] = 1.0;
        else
            activation[i] = 0.0;
      }
      else {
        if (neti > 0.0) {
          if (activation[i] < 1.0) {
            acti = activation[i];
            dt = acti + neti*(1.0 - acti);
            if (dt > 1.0) {
                activation[i] = (float) 1.0;
            }
            else activation[i] = (float) dt;
          }
        }
        else {
          if (activation[i] > (float) 0.0) {
            acti = activation[i];
            dt = acti + neti * acti;
            if (dt < FMIN) {
                activation[i] = (float) 0.0;
            }
            else activation[i] = (float) dt;
          }
        }
      }
    }
  end_of_rupdate:
    if(step_size == UPDATE)   {
       get_goodness();
       if (fgoodness < goodness) {
         fgoodness = goodness;
         for (k = 0; k < nunits; k++)
          factivation[k] = activation[k];
         }
       dgoodness = goodness;
       for (k = 0; k < nunits; k++)
        dactivation[k] = activation[k];
       cs_update_display();
       if (single_flag) {
          if (contin_test() == BREAK) {
            dgoodness = fgoodness;
            for (k = 0; k < nunits; k++)
              dactivation[k] = factivation[k];
            cs_update_display();
            return(BREAK);
          }
       }
    }
    if(Interrupt)   {
       Interrupt_flag = 0;
       get_goodness();
       if (fgoodness < goodness) {
```

```
            fgoodness = goodness;
            for (k = 0; k < nunits; k++)
             factivation[k] = activation[k];
           }
           dgoodness = goodness;
           for (k = 0; k < nunits; k++)
            dactivation[k] = activation[k];
           cs_update_display();
           if (contin_test() == BREAK) {
              dgoodness = fgoodness;
              for (k = 0; k < nunits; k++)
               dactivation[k] = factivation[k];
               cs_update_display();
                return(BREAK);
           }
       }
   }
   return(CONTINUE);
   }

   input() {
   int     i;
   char   *str,tstr[100];

   if (!System_Defined)
       if (!define_system())
           return(BREAK);
   if (!nunames) {
       return(put_error("Must provide unit names. "));
   }
   again:
   str = get_command("Do you want to reset all inputs?: (y or n)");
   if (str == NULL) goto again;
   if (str[0] == 'y') {
           for (i = 0; i < nunits; i++)
               extinput[i] = 0;
   }
   else if (str[0] != 'n') {
       put_error ("Must enter y or n!");
       goto again;
   }

   gcname:
   str = get_command("give unit name or number: ");
   if (str == NULL || strcmp(str,"end") == 0) {
       if (clamp) init_activations();
       cs_update_display();
       return(CONTINUE);
   }
   if (sscanf(str,"%d",&i) == 0) {
       for (i = 0; i < nunames; i++) {
           if (startsame(str, uname[i])) break;
       }
   }
   if (i >= nunames) {
       if (put_error("invalid name or number -- try again.") == BREAK) {
            return(BREAK);
       }
       goto gcname;
   }
   gcval:
   sprintf(tstr,"enter input strength of %s:  ",uname[i]);
   str = get_command(tstr);
   if (str == NULL) {
       sprintf(err_string,"No strength specified for %s",uname[i]);
       if (put_error(err_string) == BREAK) {
```

```
        return(BREAK);
        }
        goto gcname;
    }
    if (sscanf(str, "%f", &extinput[i]) != 1) {
        if (put_error("unrecognized value -- try again.") == BREAK) {
         return(BREAK);
        }
        goto gcval;
    }
    goto gcname;
    }

    setinput() {
    register int    i;
    register float  *pp;

    for (i = 0, pp = ipattern[patno]; i < nunits; i++, pp++) {
            extinput[i] = *pp;
    }
    strcpy(cpname,pname[patno]);
    }

    test_pattern() {
    char   *str;

    if (!System_Defined)
        if (!define_system())
            return(BREAK);

    if(ipattern[0] == NULL) {
       return(put_error("No file of test patterns has been read in."));
    }
    again:
    str = get_command("Test which pattern? (name or number): ");
    if(str == NULL) return(CONTINUE);
    if ((patno = get_pattern_number(str)) < 0) {
        if (put_error("Invalid pattern specification") == BREAK) {
         return(BREAK);
        }
        goto again;
    }
    setinput();
    zarrays();

    cycle();
    return(CONTINUE);
    }


     newstart() {
        random_seed = rand();
        reset_system();
     }

    reset_system() {
    srand(random_seed);
    clear_display();
    zarrays();
    cs_update_display();
    return(CONTINUE);
    }

    init_system() {
    int get_unames(),test_pattern(),read_weights(),write_weights();
```

```
epsilon_menu = NOMENU;

install_command("network", define_network, GETMENU,(int *) NULL);
install_command("weights", read_weights, GETMENU,(int *) NULL);
install_command("cycle", cycle, BASEMENU,(int *) NULL);
install_command("input", input, BASEMENU,(int *) NULL);
install_command("test", test_pattern, BASEMENU,(int *) NULL);
install_command("unames", get_unames, GETMENU,(int *) NULL);
install_command("patterns", get_patterns, GETMENU,(int *) NULL);
install_command("reset", reset_system, BASEMENU,(int *) NULL);
install_command("newstart", newstart, BASEMENU,(int *) NULL);
install_command("weights", write_weights, SAVEMENU,(int *) NULL);
install_command("annealing", get_schedule, GETMENU,(int *) NULL);

install_var("patno", Int,(int *) & patno, 0, 0, SETSVMENU);
init_patterns();
install_var("cycleno", Int,(int *) & cycleno, 0, 0, SETSVMENU);
install_var("updateno", Int,(int *) & updateno, 0, 0, SETSVMENU);
install_var("unitno", Int,(int *) & unitno, 0, 0, SETSVMENU);
install_var("cuname", String,(int *) cuname, 0, 0, SETSVMENU);
install_var("clamp", Int,(int *) & clamp, 0, 0, SETMODEMENU);
install_var("nunits", Int,(int *) & nunits, 0, 0, SETCONFMENU);
install_var("ninputs", Int,(int *) & ninputs, 0, 0, SETCONFMENU);
install_var("estr", Float,(int *) & estr, 0, 0, SETPARAMMENU);
install_var("istr", Float,(int *) & istr, 0, 0, SETPARAMMENU);
install_var("kappa", Float,(int *) & kappa, 0, 0, SETPARAMMENU);
install_var("boltzmann", Int, (int *) & boltzmann, 0, 0,
                                                    SETMODEMENU);
install_var("harmony", Int, (int *) & harmony, 0, 0, SETMODEMENU);
install_var("temperature",Float, (int *) & temperature, 0, 0,
                                                    SETSVMENU);
install_var("cur_temp",Float, (int *) & cur_temp, 0, 0, SETSVMENU);
install_var("goodness",Float, (int *) & goodness, 0, 0, SETSVMENU);
install_var("ncycles", Int,(int *) & ncycles, 0, 0, SETPCMENU);
install_var("nupdates", Int,(int *) & nupdates, 0, 0, SETPCMENU);
install_var("dgoodness",Float, (int *) & dgoodness, 0, 0, SETSVMENU);
install_var("fgoodness",Float, (int *) & fgoodness, 0, 0, SETSVMENU);
install_var("deltagood",Float, (int *) & deltagood, 0, 0, SETSVMENU);
install_var("maxgood",Float, (int *) & maxgood, 0, 0, SETSVMENU);
}


 cs_update_display() {
    if (unitno < nunames) strcpy(cuname,uname[unitno]);
    update_display();
 }
```

```
-------------------------------------------------------------------------
     Display template file  -  vir.tem
-------------------------------------------------------------------------
     define: layout

                I  R1 R2 RT V1 V2 VT            cycleno  $
       Inputs   $                               temp     $
                cu cu cu cu cu cu cu            harmony  $
       Features $

                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  $
 R1 + R2 = RT                                   goodnes  $
  I * R1 = V1                                   fgoodne  $
  I * R2 = V2                                   deltago  $
  I * RT = VT                                   cur_tem  $
                                                maxgood  $
end

input      look     2      $        1  extinput     1 1.0  1 vfeat.loo
features look       2      $        4  dactivation  1 1.0  1 vfeat.loo
atoms      look     2      $        5  activation   1 1.0  2 vatoms.loo
cycleno variable 1         $        0  cycleno      8 1
harmony floatvar 1         $        3  dgoodness    8 1.0
temperature floatvar 1     $        2  temperature  8 1.0
goodness floatvar 1        $        6  goodness     8 1.0
fgoodness floatvar 1       $        7  fgoodness    8 1.0
deltagood floatvar 1       $        8  deltagood    8 1.0
cur_temp  floatvar 1       $        9  cur_temp     8 1.0
maxgood   floatvar 1       $       10  maxgood      8 1.0
```

```
-----------------------------------------------------------------
The Start-up file  vir.str
-----------------------------------------------------------------
 set mode harmony 1
 get network vir.net
 get unames ic Iu R1c R1u R2c R2u Rc Ru V1c V1u V2c V2u Vc Vu end
 get anneal 1.0 200 .05 end
 set dlevel 2
 input n Vc -1 R1c -1 R2c 1 R2u 1 end
 set slevel 1
 reset
set ncycles 300
```

```
-------------------------------------------------------
 Network configuration file - vir.net
-------------------------------------------------------
definitions:
nunits 79
ninputs 14
kappa .75
nupdates 100
end
constraints:
p 1.0
m -1.0
end
network:
% 14 65 0 14
........pppppp
........ppm.pp
........pppmpp
........pppmm.
........pppmpm
........m.pppp
........m.m.m.
........m.pmpm
........pmpppp
........pmppm.
........pmpppm
........pmm.pm
........pmpmpm
..pppppp......
..ppm.pp......
..pppmpp......
..pppmm.......
..pppmpm......
..m.pppp......
..m.m.m.......
..m.pmpm......
..pmpppp......
..pmppm.......
..pmpppm......
..pmm.pm......
..pmpmpm......
pppp....pp....
ppm.....pp....
pppm....pp....
pppm....m.....
pppm....pm....
m.pp....pp....
m.m.....m.....
m.pm....pm....
pmpp....pp....
pmpp....m.....
pmpp....pm....
pmm.....pm....
pmpm....pm....
pp..pp....pp..
pp..m.....pp..
pp..pm....pp..
pp..pm....m...
pp..pm....pm..
m...pp....pp..
m...m.....m...
m...pm....pm..
pm..pp....pp..
pm..pp....m...
pm..pp....pm..
pm..m.....pm..
```

```
pm..pm....pm..
pp....pp....pp
pp....m.....pp
pp....pm....pp
pp....pm....m.
pp....pm....pm
m.....pp....pp
m.....m.....m.
m.....pm....pm
pm....pp....pp
pm....pp....m.
pm....pp....pm
pm....m.....pm
pm....pm....pm
end
sigmas:
..............ppppp
pppppppppppppppppppppppppppppppppppppppppppppppppppppppppppp
end
```